



ALTE N



General Index

1. Setting Environment.....	4
2. C++ Basics.....	6
2.1 Local Variables.....	6
2.2 Global Variables.....	6
2.3 Relational Operators.....	7
2.4 C++ Loop Statements.....	10
2.5 Loop Control Statements.....	12
2.6 C++ Decision-making Statements.....	12
2.7 C++ Functions.....	13
2.7.1 Functions Declaration.....	13
2.7.2 Functions Definition.....	14
2.7.3 Calling a Function.....	15
2.7.4 Function Arguments.....	15
2.7.5 Default Values for Parameters.....	16
2.8 C++ Pointer.....	17
2.8.1 Using Pointer.....	17
2.9 C++Data Structures.....	18
2.9.1 Defining a Structure.....	18
2.9.2 Accessing Structure Members.....	19
3. C++ Object Oriented.....	19
3.1 Principle concepts of object-oriented programming.....	19
3.2 C++ Class Definitions.....	20
3.3 C++ Class Constructor.....	21
3.4 Class Member Function.....	22
4. Multithreading.....	23
4.1 Threads Usage in Linux (POSIX Standard).....	23
4.1.2 Threads Synchronization.....	25
5. C++ Design patterns.....	27
5.1 The Singleton patterns.....	27
Annex 1. Threads Usage in Windows.....	29

Owner

Author	Description
Raffaele Ciuffreda	<i>C++ Basics (main constructs, data structures, functions usage), Object-Oriented Programming examples</i>
Marco Punzi	<i>Multithreading programming, threads synchronization and Singleton design pattern</i>

1. Setting Environment.

To install Notepad++ environment on your PC visit the link below and follow the instructions:

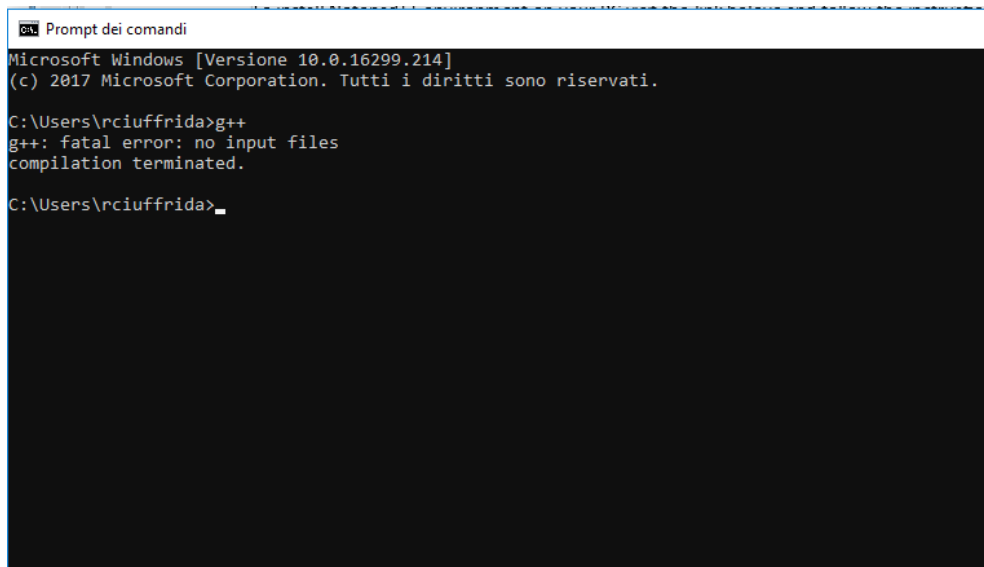
<https://notepad-plus-plus.org/download/v7.5.5.html>

MinGW (Minimalist GNU for Windows) is a free and open source software development environment for creating Microsoft Windows applications.

All the files necessary for the installation of MinGW will be provided by your trainer.

Finally add **C:\MinGW\bin;** to the PATH environment variable.

Test that everything is ok with **g++** command in the prompt as reported below.



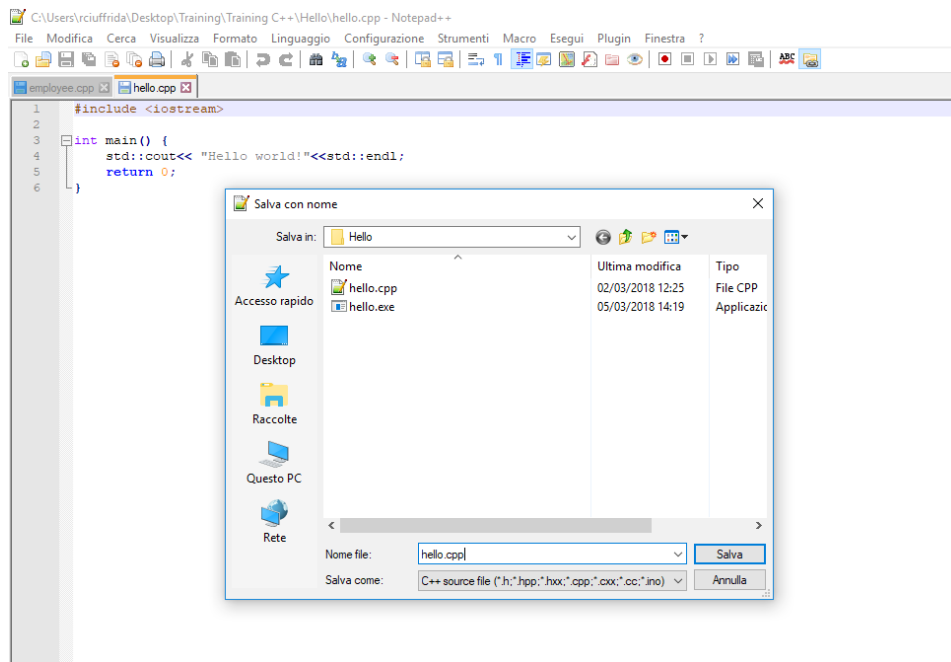
```
Prompt dei comandi
Microsoft Windows [Versione 10.0.16299.214]
(c) 2017 Microsoft Corporation. Tutti i diritti sono riservati.

C:\Users\rchiuffrida>g++
g++: fatal error: no input files
compilation terminated.

C:\Users\rchiuffrida>_
```

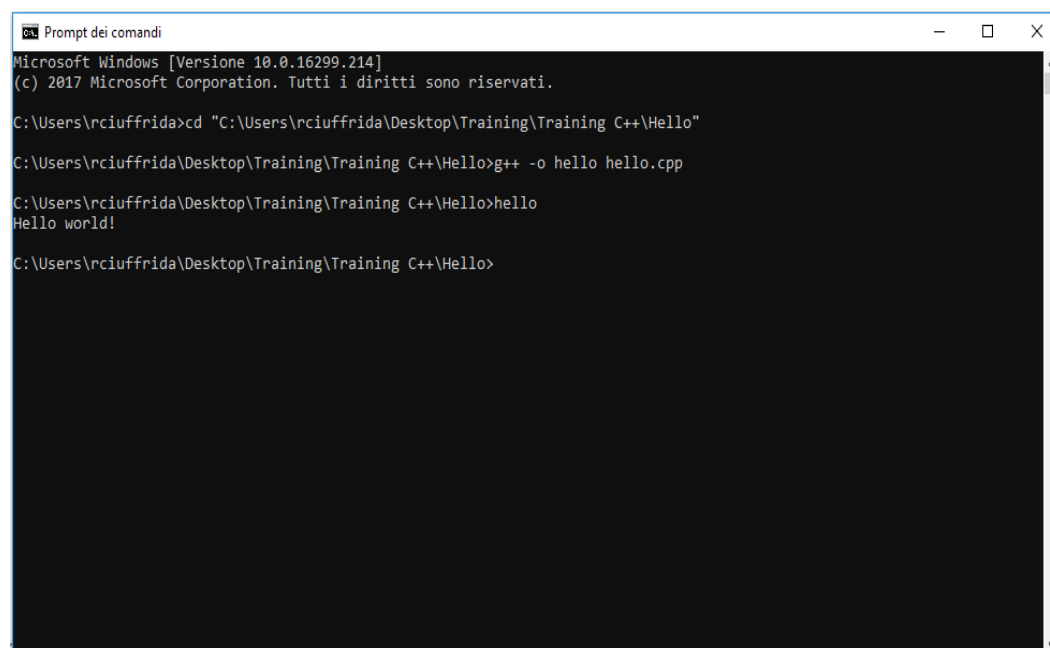
Example of a C++ project creation on Notepad++:

Now you are ready to implement your test. Finally save project with **.cpp** extension.



To compile a .cpp file open the prompt and move to the folder that contains the file, run the command **g++ -o nameOfTheExecutableFile nameOfTheSourceFile.cpp**

Finally launches the **.exe**.



2. C++ Basics

A scope is the region of the program where variables can be declared. Particularly there are three kinds of variable:

- **local variables** if they are defined inside a function or block;
- **formal parameters** which are called – by reference or by value – by a function;
- **global variables** if they are defined outside of all functions.

Here let us explain what are local and global variables.

2.1 Local Variables

Variables that are declared inside a function or block are **local variables**. They can be used only by statements that are inside that function or block of code. Local variables are not visible outside the functions where they are defined.

Exercise 1: Give an example using local variables.

Compare your solution with the official one available here:

https://technogit.altenitalia.it/Trainig_project/Training_Solutions/Training_Cpp_Solutions/tree/Branch_marcoLello/4_src/Exercise1

2.2 Global Variables

Global variables are defined outside all the functions. The global variables will hold their value throughout the life-time of your program. They can be accessed by any function and can be available for use throughout your entire program execution.

Exercise 2: Give an example using global and local variables.

Compare your solution with the official one available here:

https://technogit.altenitalia.it/Trainig_project/Training_Solutions/Training_Cpp_Solutions/tree/Branch_marcoLello/4_src/Exercise2

2.3 Relational Operators

Operator	Description	Example
<code>==</code>	Checks if the values of two operands are equal or not, if yes then condition becomes true.	<code>(A == B)</code> is not true.
<code>!=</code>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	<code>(A != B)</code> is true.
<code>></code>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	<code>(A > B)</code> is not true.
<code><</code>	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	<code>(A < B)</code> is true.
<code>>=</code>	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	<code>(A >= B)</code> is not true.
<code><=</code>	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	<code>(A <= B)</code> is true.

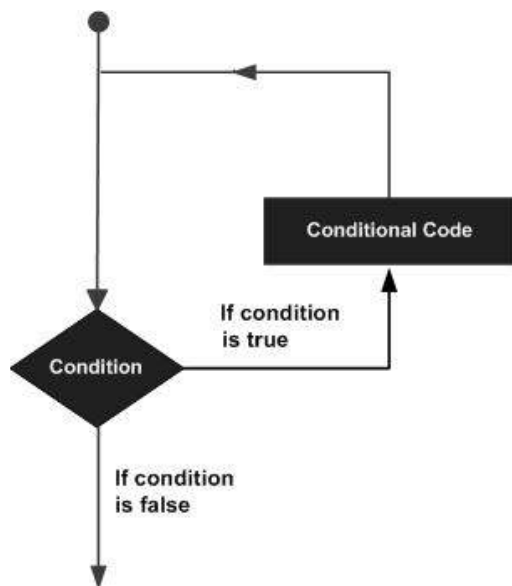
There are following logical operators supported by C++ language.

Assume variable A holds 1 and variable B holds 0, then:

Operator	Description	Example
<code>&&</code>	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	<code>(A && B)</code> is false.
<code> </code>	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	<code>(A B)</code> is true.
<code>!</code>	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	<code>!(A && B)</code> is true.

2.4 C++ Loop Statements

There may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially: the first statement in a function is executed first, followed by the second, and so on. A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages.



C++ programming language provides the following type of loops to handle looping requirements.

Loop Type	Description
While loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
For loop	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
do...while loop	Like a 'while' statement, except that it tests the condition at the end of the loop body.
Nested loops	You can use one or more loop inside any another 'while', 'for' or 'do..while' loop.

2.5 Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

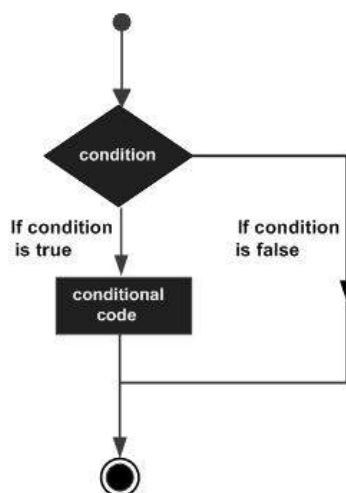
C++ supports the following control statements.

Control statement	Description
Break statement	Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.
Continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
Goto statement	Transfers control to the labeled statement. Though it is not advised to use goto statement in your program.

2.6 C++ Decision-making Statements

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages.



C++ programming language provides following types of decision making statements.

Statement	Description
If statement	An 'if' statement consists of a boolean expression followed by one or more statements.
If...else statement	An 'if' statement can be followed by an optional 'else' statement, which executes when the boolean expression is false.
Switch statement	A 'switch' statement allows a variable to be tested for equality against a list of values.
Nested if statements	You can use one 'if' or 'else if' statement inside another 'if' or 'else if' statement(s).
Nested switch statements	You can use one 'switch' statement inside another 'switch' statement(s).

2.7 C++ Functions

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions. You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task. A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

A function is known with various names like a method or a sub-routine or a procedure etc.

2.7.1 Functions Declaration

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts:

```
return_type function_name(parameter list);
```

For the above defined function `max()`, following is the function declaration:

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so following is also valid declaration:

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

2.7.2 Functions Definition

The general structure of a C++ function definition is as follows:

```
return_type function_name(parameter list) {  
    body of the function  
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function:

- **Return Type** – A function may return a value. The `return_type` is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return type is the keyword **void**.
- **Function Name** – This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** – When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; a function may contain no parameters.
- **Function Body** – The function body contains a collection of statements that define what the function does.

Exercise 3: Build a function called `max()` that takes two parameters `num1` and `num2` and return the biggest of both.

Compare your solution with the official one available here:

https://technogit.altenitalia.it/Trainig_project/Training_Solutions/Training_Cpp_Solutions/tree/Branch_marcoLello/4_src/Exercise3

2.7.3 Calling a Function

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function. When a program calls a function, program control is transferred to the called function. A called function performs defined task and when it's return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value.

Exercise 4: Build a program that calls the function of the previous exercise. While running final executable, it would produce the following result:

```
Max value is : 200
```

Compare your solution with the official one available here:

https://technogit.altenitalia.it/Trainig_project/Training_Solutions/Training_Cpp_Solutions/tree/Branch_marcoLello/4_src/Exercise4

2.7.4 Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function. The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function:

Call Type	Description
Call by Value	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Call by Pointer	This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
Call by Reference	This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function.

2.7.5 Default Values for Parameters

When you define a function, you 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. This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead.

Exercise 5: Build a program with a function called `sum()` that takes two parameters `num1` and `num2` and returns the sum of both. Finally call the same function using default parameters and passing new parameters. When code is compiled and executed, it produces the following result:

```
Total value of default parameters is: 120
Total value of new parameters is: 300
```

Compare your solution with the official one available here:

https://technogit.altenitalia.it/Trainig_project/Training_Solutions/Training_Cpp_Solutions/tree/Branch_marcoLello/4_src/Exercise5

2.8 C++ Pointer

A **pointer** is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is:

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C++ type and **var-name** is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

```
int    *ip;    // pointer to an integer
double *dp;    // pointer to a double
float  *fp;    // pointer to a float
char   *ch     // pointer to character
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

2.8.1 Using Pointer

There are few important operations, which we will do with the pointers very frequently:

- We define a pointer variable
- Assign the address of a variable to a pointer
- Finally access the value at the address available in the pointer variable.

This is done by using unary operator `*` that returns the value of the variable located at the address specified by its operand.

Exercise 6: Build a program that prints the value of a variable, the pointer variable, and the address stored in pointer variable. When code is compiled and executed, it produces result something as follows:

```
Value of var variable: 20
Address stored in ip variable: 0xbfc601ac
Value of *ip variable: 20
```

Compare your solution with the official one available here:

https://technogit.altenitalia.it/Trainig_project/Training_Solutions/Training_Cpp_Solutions/tree/Branch_marcoLello/4_src/Exercise6

2.9 C++Data Structures

C/C++ arrays allow you to define variables that combine several data items of the same kind, but **structure** is another user-defined data type which allows you to combine data items of different kinds. Structures are used to represent a record, suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:

- Title
- Author
- Subject
- Book ID

2.9.1 Defining a Structure

To define a structure, you must use the **struct** statement. The **struct** statement defines a new data type, with more than one member, for your program. The format of the **struct** statement is the following:

```
struct [structure tag] {  
    member definition;  
    member definition;  
    ...  
    member definition;  
} [one or more structure variables];
```

The **structure tag** is optional and each member definition is a normal variable definition, such as `int i;` or `float f;` or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure:

```
struct Books {  
    char title[50];  
    char author[50];  
    char subject[100];  
    int  book_id;  
} book;
```

2.9.2 Accessing Structure Members

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use **struct** keyword to define variables of structure type.

Exercise 7: Give an example to explain usage of structure. When code is compiled and executed, it produces the following result:

```
Book 1 title : Learn C++ Programming
Book 1 author : Chand Miyan
Book 1 subject : C++ Programming
Book 1 id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Yakrit Singha
Book 2 subject : Telecom
Book 2 id : 6495700
```

Compare your solution with the official one available here:

https://technogit.altenitalia.it/Trainig_project/Training_Solutions/Training_Cpp_Solutions/tree/Branch_marcoLello/4_src/Exercise7

3. C++ Object Oriented

The prime purpose of C++ programming was to add **object orientation** to the C programming language, which is in itself one of the most powerful programming languages.

The core of the pure object-oriented programming is to create an object, in code, that has certain properties and methods. While designing C++ modules, we try to see whole world in the form of objects. For example a car is an object which has certain properties such as color, number of doors, and the like. It also has certain methods such as accelerate, brake, and so on.

3.1 Principle concepts of object-oriented programming

- **Object** - This is the basic unit of object oriented programming. That is both data and function that operate on data are bundled as a unit called as object.
- **Class** - When you define a class, you define a blueprint for an object. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

- **Abstraction** - Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details. For example, a database system hides certain details of how data is stored and created and maintained. Similar way, C++ classes provides different methods to the outside world without giving internal detail about those methods and data.
- **Encapsulation** - Encapsulation is placing the data and the functions that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object.
- **Inheritance** - One of the most useful aspects of object-oriented programming is code reusability. As the name suggests Inheritance is the process of forming a new class from an existing class that is from the existing class called as base class, new class is formed called as derived class. This is a very important concept of object-oriented programming since this feature helps to reduce the code size.
- **Polymorphism** - The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.
- **Overloading** - The concept of overloading is also a branch of polymorphism. When the exiting operator or function is made to operate on new data type, it is said to be overloaded.

3.2 C++ Class Definitions

A class definition starts with the keyword `class` followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Employee data type using the keyword `class` as follows:

```
class Employee {
    public:
        static int empCount;
        int salary;
        string name;
};
```

The keyword **public** determines the access attributes of the members of the class that follow it. A public member can be accessed from outside the class anywhere within the scope of the class object. The keyword **static** declares members that are not bound to class instances. You can also specify the members of a class as **private** or **protected**.

3.3 C++ Class Constructor

A class constructor is a special member function of a class that is executed whenever we create new objects of that class. A default constructor does not have any parameter, but if you need, a constructor can have parameters. This helps you to assign initial value to an object at the time of its creation.

Exercise 8: Define a class and implement its constructor.

Compare your solution with the official one available here:

https://technogit.altenitalia.it/Trainig_project/Training_Solutions/Training_Cpp_Solutions/tree/Branch_marcoLello/4_src/Exercise8

3.4 Class Member Function

A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object. You can define the same function outside the class using the scope resolution **operator (::)**.

Exercise 9: Define the displayCount() function that returns empCount and the displayEmployee() function that returns other information about the employee object.

Here, only important point is that you would have to use class name just before :: operator. A member function will be called using a dot operator (.) on a object where it will manipulate data related to that object only as follows:

```
Employee emp1 ("Zara",2000);    //Create an object
Employee emp2 ("Manni",5000);   //Create an object

emp1.displayEmployee();         // Call member function for the object
emp2.displayEmployee();         // Call member function for the object
```

Compare your solution with the official one available here:

https://technogit.altenitalia.it/Trainig_project/Training_Solutions/Training_Cpp_Solutions/tree/Branch_marcoLello/4_src/Exercise9

Exercise 10: Now, putting all the concepts together and build a program that creates an instance of Employee class and uses its methods. When code is executed, it produces the following result:

```
Name: Zara ,Salary: 2000
Name: Manni ,Salary: 5000
Total Employee 2
```

Compare your solution with the official one available here:

https://technogit.altenitalia.it/Trainig_project/Training_Solutions/Training_Cpp_Solutions/tree/Branch_marcoLello/4_src/Exercise10

4. Multithreading.

In this section we'll talk about how to manage concurrent programming, which requires many flows of the same program to share common resources or memory. At the beginning, when a process is created, it is associated only to one execution flow, which is the main thread. This process could request to the O.S. (Operative System) the creation of other secondary threads and the scheduler will assign - in a non-deterministic way – the available resources to each one of them.

This approach allows us to exploit the bottlenecks associated to I/O operations, going on with the execution of our algorithms while they're in progress. Furthermore, because of the sharing of memory, the workload associated with the interprocess communication decreases and the parallelization of many operations is possible and less-consumption. On the other side, the overall program complexity increases, the execution is not deterministic anymore and new errors must be solved. The memory must be considered as a shared world in which many threads access in a synchronized way to preserve a correct behaviour; therefore, new synchronization constructs will be introduced.

Operative Systems' APIs allow to manage the entire lifecycle of each thread:

2. Creation and termination of a thread
3. Synchronization with other threads
4. Management of private memory areas

Each O.S. provides different statements and libraries to do that, increasing the complexity of applications' portability.

4.1 Threads Usage in Linux (POSIX Standard).

POSIX Threads, or **Pthreads**, is a POSIX standard for threads. It defines an API for creating and manipulating threads especially on Unix systems.

Implementations of the API are available on many Unix-like POSIX systems such as FreeBSD, NetBSD, GNU/Linux, Mac-OS and Solaris, but Microsoft Windows implementations also exist. For example, the **pthreads-w32** is available in MinGW and supports a subset of the Pthread API for the Windows 32-bit platform.

Pthreads are defined as a set of C++ language programming types and procedure calls, implemented with a **<pthread.h>** header file. In GNU/Linux, the **pthread** functions are not included in the standard C++ library. They are in **libpthread** and so we should add the option **-lpthread** to link our program.

4.1.1 The Pthread API

Suppose now that our **main()** program is a single, default thread. All other threads must be explicitly created by the programmer.

Initially, the function **pthread_create()** creates a new thread and makes it executable. This routine can be called any number of times from anywhere within our code. In details, it requires the following arguments:

pthread_create

(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)

Where:

2. **thread** is an identifier for the new thread returned by the subroutine. This is a pointer to **pthread_t** structure. When a thread is created, an identifier is written to the memory location to which this variable points. This identifier enables us to refer to the thread.
3. **attr** is an attribute object that may be used to set thread attributes. We can specify a thread attributes object, or NULL for the default values.
4. **start_routine** is the routine that the thread will execute once it is created. We should pass the address of a function taking a pointer to void as a parameter and the function will return a pointer to void. So, we can pass any type of single argument and return a pointer to any type. While using **fork()** causes execution to continue in the same location with a different return code, using a new thread explicitly provides a pointer to a function where the new thread should start executing.
5. **arg** is a single argument that may be passed to **start_routine**. It must be passed as a void pointer. NULL may be used if no argument is to be passed.

While the function **pthread_create()** is used to create and make executable a thread, the function **pthread_exit()** is one of the several ways in which a certain thread could be terminated. Typically, the **pthread_exit()** routine is called after a thread has completed its work and is no longer required to exist. If the **main()** finishes before the threads it has created - and the function **pthread_exit()** has been called - the other threads will continue to execute. Otherwise, they will be automatically terminated when **main()** finishes.

Exercise 12: Now, build an algorithm in which the main() function creates a defined number of worker threads (using a FOR loop) that call a service routine Function(). This function must print a message, for each thread, with the thread ID associated to the index of the loop passed as parameter. Remember that the return value from the pthread_create() call will be zero if it's successful, otherwise, it returns an error.

When the code contained in the file example.cpp is executed with the following command:

```
g++ example.cpp -lpthread -o executable_file
```

it must produce the following output:

```
main(): Creating thread, 0
main(): Creating thread, 1
main(): Creating thread, 2
Hello! Thread ID0Hello! Thread ID0H
ello! Thread ID12
```

Compare your solution with the official one available here:

https://technogit.altenitalia.it/Trainig_project/Training_Solutions/Training_Cpp_Solutions/tree/Branch_marcoLello/4_src/Exercise12

Each thread is called in a non-deterministic way by the scheduler and consequently the associated messages are out-of-order because of the absence of synchronization between threads. This is the reason why we need a synchronization variable to manage race conditions, as explained in the following section.

4.1.2 Threads Synchronization.

The mutual exclusion lock (mutex) is the simplest and most primitive synchronization variable. It provides a single, absolute owner for the section of code (critical section) that it brackets between the calls to `pthread_mutex_lock()` and `pthread_mutex_unlock()`. The first thread that locks the mutex gets ownership, and any subsequent attempts to lock it will fail, causing the calling thread to go to sleep. When the owner unlocks it, one of the sleepers will be awakened, made runnable, and given the chance to obtain ownership.

A mutex lock is a mechanism that can be acquired by only one thread at a time. For other threads to get the same mutex, they must wait until it is released by the current owner of the mutex. To create a mutex that can be shared between processes, we need to set up the attributes for **pthread_mutex_t initialization**. In particular, initialize the mutex defined globally in the following mode:

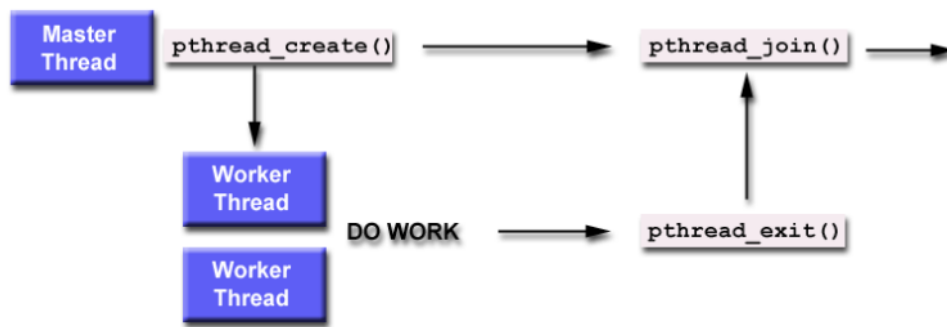
```
pthread_mutex_t mutex_name = PTHREAD_MUTEX_INITIALIZER;
```

or using the **pthread_mutex_init** function in the following mode:

```
pthread_mutex_init(&mutex_name, NULL);
```

After that, simply acquiring and releasing the lock allow programmers to synchronize different threads' operations.

Finally, with the subroutine **pthread_join()**, the calling thread – the `main()` thread - blocks until the specified/called worker thread terminates. The figure below describes the global flow of a concurrent algorithm.



Exercise 13: Now, using the code of the Exercise 12, try to synchronize the worker threads which are launched and which share a global resource (for example an int variable), defining a global mutex, initializing it and calling the lock/unlock functions in the service routine associated to the threads' execution. Remember to join the threads in the main() function.

Compiling and running the code, the following output must be obtained:

```
Hello, my threadID is': 1  
value of the shared resource = 5  
  
Hello, my threadID is': 3  
value of the shared resource = 10  
  
Hello, my threadID is': 2  
value of the shared resource = 15  
  
Hello, my threadID is': 0  
value of the shared resource = 20  
  
Hello, my threadID is': 4  
value of the shared resource = 25  
  
Hello, my threadID is': 5  
value of the shared resource = 30
```

Compare your solution with the official one available here:

https://technogit.altenitalia.it/Trainig_project/Training_Solutions/Training_Cpp_Solutions/tree/Branch_marcoLello/4_src/Exercise13

5. C++ Design patterns

Software design patterns are abstractions that help structure systems design. A **pattern** is a way to describe and address by name (mostly a simplistic description of its goal), a repeatable solution or approach to a common design problem, that is, a common way to solve a generic problem.

5.1 The Singleton patterns

The **Singleton** design pattern ensures that only one instance of the C++ class is instantiated. It assures that only one object is created and no more. It is often used for a logging class so only one object has access to log files, or when there is a single resource, where there should only be a single object in charge of accessing the single resource. The singleton pattern discussed here gives the class itself, the responsibility of enforcement of the guarantee that only one instance of the class will be allowed to be generated.

A **Singleton** is an elegant way of maintaining global state and it offers several advantages over global variables because it does the following:

- Enforces that only one instance of the class can be instantiated
- Allows control over the allocation and destruction of the object
- Provides thread-safe access to the object's global state
- Prevents the global namespace from being polluting.

Let's think about a class which has private constructor, first requirement of our **Singleton** design pattern. It becomes a class that can't be instantiated because it has private constructor.

```
Class Singleton{  
  
private:  
    Singleton() {}  
};
```

We need to have an instance of the class to call it, but we can't have an instance because no other class can't instantiate it. We can use the constructor from an object of type **Singleton** but we can never instantiate that object because no other object can use:

```
new Singleton();
```

So, we should find a way of using a static method, limiting the **instantiation** of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system.

Furthermore, we should consider the following constraints:

- We do not want the singleton by copied so that there is only one instance. This can be achieved by declaring a private copy constructor and a private assignment operator
- A **getInstance()** method should return the global object reference. This blocks a client from deleting the object. Also, by making destructor private, we can achieve the same effect

Exercise14: Write a Singleton class in which:

- **private constructor and destructor are defined**
- **private copy constructor and a private assignment operator are declared**
- **a public static method getInstance() is defined to return the object instance reference**

Then, in the main() function, call two instances of the global object.

Compare your solution with the official one available here:

https://technogit.altenitalia.it/Trainig_project/Training_Solutions/Training_Cpp_Solutions/tree/Branch_marcoLello/4_src/Exercise14

Exercise15: Use the Singleton pattern in a multithreading program.

In particular, referring to the Exercise13, try to substitute a Singleton object to the shared resource and modifying the service routine function so that the constructor of the shared object is called only ONE time while the getInstance() method is invoked every time a thread begins its execution.

Compare your solution with the official one available here:

https://technogit.altenitalia.it/Trainig_project/Training_Solutions/Training_Cpp_Solutions/tree/Branch_marcoLello/4_src/Exercise15

Annex 1. Threads Usage in Windows

Microsoft Windows operating systems support for multithreaded programming is almost similar to the support provided by POSIX threads. The differences are not in the actual functionality but the names in the API functions.

Every Win32 process has at least one thread, which we call as the **main thread**. We will assume that the O.S. will give a time slice to each program thread, in round-robin fashion. In fact, the threads in a Win32 program will be competing for the CPU with threads in other programs and with system threads, and these other threads may have higher priorities.

To use Windows multithreading functions, we must include `<windows.h>` in our program. To create a thread, the Windows API supplies the **CreateThread()** function which creates the thread and returns a handle to that thread but it does not set up the thread to work with the thread-local data structures provided by the developer environment. So, instead of calling **CreateThread()**, we should use the calls by the runtime libraries. The two recommended ways of creating a thread are the calls **_beginthread()** and **_beginthreadex()** included in `<process.h>`. They differ in the number of parameters they take but also because a thread created by **_beginthread()** will close the handle to the thread when the thread exits, while the handle returned by **_beginthreadex()** will have to be explicitly closed by calling the **CloseHandle()** function.

About terminating threads, there are several ways to do that. It is possible to make threads to terminate using the **ExitThread()** or **TerminateThread()**. But these function calls are not recommended since they may leave the application in an unspecified state. They also don't give the run-time libraries the opportunity to clean up any resources that they have allocated for the thread. These are the reasons why a thread should terminate with a call to **_endthread()** or **_endthreadex()** matching the call that was used to create the thread (**_beginthread()** or **_beginthreadex()**). If the thread exits with a call to **_endthreadex()**, the handle to the thread still needs to be closed by another thread calling **CloseHandle()**.

For a fork-join type model, there will be a master thread that creates multiple worker threads and then waits for the worker threads to exit. There are two routines that the master thread can use to wait for the worker to complete: **WaitForSingleObject()** or **WaitForMultipleObjects()**. These two functions will wait either for the completion of a single thread or for the completion of an array of threads. The routines take the handle of the thread as a parameter together with a timeout value that indicates how long the master thread should wait for the worker thread to complete. Usually, the value INFINITE will be appropriate.

The following example demonstrates the code necessary to wait for a single thread, created with one of the approach described above:

Exercise 11: Now, putting all the concepts together and build a program that creates a new worker thread with the function `_beginthreadex` described above and waits for his execution.

When the code is executed, it must produce the following output:

```
Thread: 4360
Thread: 1368
```

Run again your code substituting **WaitForSingleObject()** with the **WaitForMultipleObjects()** function as reported below:

```
WaitForMultipleObjects(2, myhandle, true, INFINITE);
```

Check if there are some differences or not.

Compare your solution with the official one available here:

https://technogit.altenitalia.it/Trainig_project/Training_Solutions/Training_Cpp_Solutions/tree/Branch_marcoLello/4_src/Exercise11