FITPED

C++ specific

Jiří Rybička
Viera Michaličková
Juan Carlos Rodríguez-del-Pino
José Daniel González-Domínguez
Zenón José Hernández-Figueroa
Małgorzata Przybyła-Kasperek

www.fitped.eu

2021

Co-funded by the
Erasmus+ Programme
of the European Union

Work-Based Learning in Future
IT Professionals Education
(Grant. no. 2018-1-SK01-KA203-046382)

C++ Specific

Published on

November 2021

Authors

Jiří Rybička | Mendel University in Brno, Czech Republic

Viera Michaličková | Constantine the Philosopher University in Nitra, Slovakia

Juan Carlos Rodríguez-del-Pino | University of Las Palmas de Gran Canaria, Spain

José Daniel González-Domínguez | University of Las Palmas de Gran Canaria, Spain

Zenón José Hernández-Figueroa | University of Las Palmas de Gran Canaria, Spain

Małgorzata Przybyła-Kasperek | University of Silesia in Katowice, Poland

Reviewers

Anna Stolińska | Pedagogical University of Cracow, Poland

Peter Švec | Teacher.sk, Slovakia

Eugenia Smyrnova-Trybulska | University of Silesia in Katowice, Poland

Piet Kommers | Helix5, Netherland

Graphics

Ľubomír Benko | Constantine the Philosopher University in Nitra, Slovakia

David Sabol | Constantine the Philosopher University in Nitra, Slovakia

Erasmus+ FITPED

Work-Based Learning in Future IT Professionals Education

Project 2018-1-SK01-KA203-046382

Co-funded by the
Erasmus+ Programme
of the European Union

ISBN 978-80-558-1781-1

Table of Contents

Introduction to C++

Chapter 1

1.1 OOP introduction

📖 1.1.1

Object-oriented programming tries to make the transition from the problem domain to the programming language as simple as possible, also preventing the existing relationship between the elements of the real world and its representation on the computer from being lost.

Object-Oriented Programming (OOP) is based on objects, basic elements that are used to represent the real world and computer artefacts used in solving the problem. The type of object is called class. Each object belongs to a class that determines its structure and behavior. The classes are organized through inheritance. Inheritance allows you to establish a hierarchy of classes. The polymorphism enables the use of inheritance to handle objects of different kinds using the same interface. Sometimes, the classes are organized into modules.

1.1.2

Object-oriented programming tries to make the transition from the problem domain to the programming language as simple as possible.

- True
- False

1.1.3

Please, select the correct description:

The _____ are the fundamental elements that are used to represent the real world and computer artefacts used in solving the problem.

The _____ are types of objects.

The _____ allows establishing a hierarchy of classes.

The _____ enables the use of inheritance to handle objects of different kinds using the same interface.

- objects
- classes
- polymorphism

- inheritance

📖 1.1.4

The objects

Objects are the axis on which Object-Oriented Programming (OOP) is centred. They are characterized by saving their state and performing operations or functions. Its state may change due to the action of any of your operations.

📝 1.1.5

When can an object change of state?

- Depends on the change of an external object
- Based on an internal trigger
- Due to the action of any of its operations or functions

📖 1.1.6

The description of the characteristics and behavior of a type of object is called a class. The relationship of an object to its class is the same as a variable to its type. Notice that the concept of class is eminently static while the object is dynamic.

The state of an object is determined by its attributes that are nothing but objects (variables) of other classes belonging to it.

The operations or functions that are defined in a class are those that objects can perform. These operations are equivalent to functions or procedures with their name, their parameters, and, if applicable, their value returned, but with the addition that they act on a specific object.

📝 1.1.7

The relationship of an object to its class is the same as ...

- a variable to its type
- a variable to its value
- a value to its type
- a type to its value

📖 1.1.8

Inheritance is the ability to define new classes from existing ones so that they present the same characteristics as those, plus new ones. Therefore, inheritance allows you to create classes that are specializations of others. Notice that inheritance is not intended to reuse existing classes to save us some lines of code. Inheritance has a sense of hierarchy of classes at the level of abstraction and specialization.

📝 1.1.9

Inheritance _____ to create classes that are specializations of others. Notice that inheritance _____ to reuse existing classes by taking advantage of what a class provides, _____ some lines of code. Inheritance has a sense of hierarchy of classes at the level of abstraction and specialization.

- saving us
- allows you
- ignoring
- is not intended

📖 1.1.10

Polymorphism allows us to use the same interface to handling objects of different kinds. This mechanism improves the abstraction in the use of objects, allowing a homogeneous use of different types of objects.

📝 1.1.11

Polymorphism allows you to create classes that are specializations of others

- False
- True

📝 1.1.12

Polymorphism improves the abstraction in the use of objects

- True
- False

🕮 1.1.13

C++ is a general-purpose language with a certain tendency to system programming (construction of operating systems, controllers, etc.) that:

- It is compatible with C, improving it in many aspects.
- It supports object-oriented programming.
- It supports genericity or template programming.
- It handles exceptions.
- It has no garbage collector.
- It provides a standard class library, with containers, character strings, I/O, etc.

1.1.14

C++ features:

It is _____ with C, improving it in many aspects.

It supports _____ programming.

It supports _____ functions and classes.

It handles _____.

It has no _____.

It provides a _____, with containers, character strings, I/O, etc.

- object-oriented
- compatible
- standard class library
- garbage collector
- exceptions
- generic

1.2 C++ non OOP features (1/2)

📖 1.2.1

As is clear from its name, C++ is a language that derives from C, one of the most influential in the computer world. The creator of C++, Bjarne Stroustrup, was clear from the beginning that to achieve rapid acceptance of the new language, it must be compatible with its predecessor so that C language programmers could gradually introduce the new concepts.

C++ not only introduces changes in C to support object-oriented programming. The creators of C++ take the opportunity to correct defects and improve the undesirable characteristics of its predecessor.

📝 1.2.2

C++ derives from the C programming language.

- True
- False

📝 1.2.3

Who was the initial creator of C++?

- Dennis Ritchie
- Steve Jobs
- Bjarne Stroustrup
- Richard Stallman

📝 1.2.4

The design of C++ tries to be compatible with C.

- True
- False

1.2.5

The C++ programming language can be used to develop non-Object-Oriented Programming code.

- True
- False

1.2.6

In the C programming language the declaration of a local variable can hide a global variable with the same name. In the next piece of the program, the global variable "global_var" is not accessible from the *fn()* function. C++ allows the use of an "::" scope operator to explicitly refer to the scope of a variable. In this case, "global_var" refers to the local variable and "*::global_var*" refers to the global variable.

Example:

```
int global_var;
void fn() {
   int global_var;
   global_var = 1; // Refers to the local variable
   ::global_var = 2 // Refers to the global variable
}
```

1.2.7

Write the correct code to resolve the function "plus"

```
int value;
int plus(int value) { //return global "value" plus parameter
"value"
   return _____ + value;
}
```

1.2.8

The **bool** type represents the logical type, and the corresponding *false* and *true* literals are added.

Example:

```
bool flag = true;
int v [TV];
...
for (int i = 0; i < TV && flag; i ++) {
   if (v[i] == element) flag = false;
   else flag = true;
   // You could have written "flag = !(V [i] == element);"
}
```

Although this type of data has been added, it is still possible to use the integers as logical types.

1.2.9

Select the correct keyword for the new boolean type:

- bool
- Bool
- boolean
- Boolean
- logical
- Logical
- BOOLEAN

1.2.10 Using bool type

Define a function named "oneTrue" that accepts as the first parameter an array of bool and second its size. The function returns true if only one boolean in the array is true and false if not.

1.2.11

References allow defining alternative names to a variable, object, array element, and etc. The reference became an alias of another variable that needs to be initialized to the element they reference. References once initialized cannot be separated from the variable they refer to and are indistinguishable from it. They make programming work much more comfortable without the inconvenience of pointer notation. Passing a variable to a function by reference generates a type of input and output parameters, which is much clearer than passing it by address.

The declaration or definition of a reference is made preceding its name of the character "&". Notice the difference of references with the operator "&" to return the address of a variable or other "&" uses in C. It should be noted that references must be initialized. Here are some examples:

```
int i;
int * pi = & i; // Declare a pointer to i
int & ri = i; // Declare a reference to i
...
// The following three instructions have the same effect
i = 10;
* pi = 10;
ri = 10;
...
// The reference ri cannot be distinguished from i, so the
next instruction is equivalent to "pi = & i"
pi = & ri;
```

The variable ri is a reference to i, this means that any use of ri has the same effect as using i. Since ri is an alias of i, ri has exactly the same type as i.

Reference variables are used essentially as past arguments or values returned from a function.

1.2.12

Pair the uses of & in C++ with its description:

type & name = expr; // expr. must be a lvalue _____

expr1 & expr2 _____

expr1 && expr2 _____

& expr // expr. must be an lvalue _____

- the address-of operator
- bitwise "and" operator
- reference definition
- logical "and" operator

1.2.13

If we have an array of ints of size 10, select the code to define a reference named "mid" to the element at index 5.

```
int vec[10];
```

- int & mid = v[5];
- int mid = & v[5];
- int mid = v[5];
- int *mid = & v[5];
- int & mid = * v[5];

1.2.14 Parameters by reference

Define a void function named "sort2" that accepts two references to float. The function reorders the two values, leaving the lowest value in the first parameter and the highest in the second.

1.2.15

Review content (optional)

The C/C++ language supports passing parameters by value. It is simple: when a function call is done, parameters defined in the function receives a copy of the passed values. As the formal parameters act as local variables within the function, the modifications made to them do not affect the external variables.

```
int f (int param); // Prototype statement
void g () {
   int arg;
   arg = -3;
   f (arg); //Call
   f (5); //Call
}

int f (int param) { // Definition
   printf ("% d \ n", param);
   param= 10; // Assign the value 10 to param, does not affect
arg
}
```

When the variable arg is passed to f, the function receives a copy of the value of arg that is stored in the param parameter. Any change of param within the body of the function f does not affect the value of arg at all. The function cannot, therefore, directly alter the parameter passed by value. Thus, during the execution of function f, there is the variable arg and the variable param. When the function f ends, the param variable ceases to exist, and the memory it occupies is released.

🕮 1.2.16

Review content (optional)

When an address is passed to a function, it is called a parameter passed by address, although in reality, it is only a parameter by value with the value passed being an address. Through this address, you can access and modify the value of external variables.

```
int f (int * param); // Prototype statement
void g () {
   int arg, * pi = & arg;
   arg = -3;
   f (& arg); // Call passing the arg address
   f (pi); // Call passing the value of pi
}

int f (int * param) {// Definition
   printf ("% d \ n", * param);
   // ...
   * param = 10;
   // Assign the value 10 to the address indicated by the
param
   // therefore, 10 is assigned to arg
}
```

The address of arg (& arg) is passed by value and the function receives a copy of it in param. The function accesses arg through the address it receives and thus can access or modify the value of arg.

Notice that a parameter defined as an array is an address, the address of the first element of the array.

Important! any change in the array parameter affects the passed array. e.g.

```
int summation(int[] v, int n) { // v is the address of the
first element of the array
    int s = 0;
```

```
    for ( int i = 0; i < n ; i++  ) {
        s = v[i];
    }
    return s;
}

int sumation2(int[] v, int n) { // v is the address of the
first element of the array
    for ( int i = 0; i < n - 1 ; i++  ) {
        v[i + 1] += v[i]; // Change the external array
    }
    return v[n-1];
}

int f() {
    int a[10];
    // ...
    int r = summation( a, 10); // a do not change
    modification( a, 10); // a changes
    // ...
}
```

📖 1.2.17

The passing of a variable by reference does that; once the call is made, the formal parameter becomes an alias of the passed variable, It is, any action taken on the parameter is being performed on the variable. The main advantages of the pass by reference with respect to the pass by address are security and simplicity since it is not necessary to pass or use explicit addresses.

```
int f (int & parm); // Prototype statement
void g () {
   int arg;
   ...
   f (arg); // Call passing the variable arg
}
int f (int & parm) { // Definition
   printf ("% d \ n", parm);
   // ...
   parm = 10;
   // Assign the value 10 to parm, but parm is the alias of
arg;
   // therefore, modifying parm modifies arg
}
```

1.2.18

Select the type of parameter pass for the functions shown

int plus(int a, int b); _____

int swap(int *a, int *b); _____

int swap(int &a, int &b); _____

- Parameters by address
- Parameters by reference
- Parameters by value

1.2.19

We want to develop a swap function that passing two integers, swaps its values.

```
void swap ([Select]){
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int v1 = 3;
    int v2 = 4;
    swap (v1, v2);
    if ( v1 == 4 && v2 == 3 ) {
       return 0;
    } else {
       return 1; // Error
    }
}
```

Select one:

- int & a, int & b
- int a, int b
- int * a, int * b
- int a[], int b[]

1.3 C++ non OOP features (2/2)

1.3.1

The type returned by functions may be declared as a reference. This has the advantage, as in the passing of parameters, that avoid the copy of an object. But it has the disadvantage that the object itself is being returned. Of course, it is wrong to return references to local variables of the function, if done, the receiver will get a variable already deleted.

Example of correct use:

```
int & min (int & a, int & b) {
   return (a <b)? a: b; // The reference of the minor variable
is returned
}
void f () {
   int i = 3, j = 8;
   min (i, j) = 4; // The value 4 is assigned to the minor
variable
}
```

1.3.2

Select the functions that return valid references.

Select one or more:

- int & f(int &a) { return a; }
- int & f(int a) { return a; }
- int & f(int &a) { int b = a + 1; return b; }
- int & f(int v[], int size) { return v[size/2]; }

1.3.3 Return references

Define a function named "maxArrayElement" that passing an array of integer and its size, returns a reference to the element of maximum value.

1.3.4

In the C++ language, it is possible to have two or more functions with the same name. This feature allows using a more appropriate name for functions with the same purpose. When we have several functions with the same name, and there is a call to one of them, the compiler chooses which function to call during compilation.

The compiler distinguishes between one and another function using the number of parameters and the type of each one. It is not possible to define two functions with the same name, with the same amount of parameters and the same types in the same order. The type of the returned value by the function is not used to differentiate them.

Example:

```
#include <stdio.h>
void print (int i, int nd) {
// The * in the format indicates that it is taken as the
displayed size of i nd
   printf ("%*d", nd, i);
}
void print (float f) {
   printf ("%f", f);
}
void print (const char * s) {
   printf ("%s", s);
}
void print () {
   printf ("\n");
}
int main () {
   print ("This is a character string test");
   print();
   print ("A real");
   print (10.0);
   print();
   print ("A two-digit integer");
   print (2.38);
   print();
   return 0;
}
```

1.3.5

Select the correct functions overloading.

Select one or more:

- int f(int a); int f(char *a);
- int g(int a); void g(int*a);
- int h(int a); int h(int &a);
- int k(int a); void k(int b);

⌨ 1.3.6 Function overloading

Define two void functions named "sort". The first function accepts two references to float, and the second function accepts two references to int. These functions reorder the two values, leaving the lowest value in the first parameter and the highest in the second.

🕮 1.3.7

C++ allows that parameters passed to a function have a default value. That is, if a parameter is not specified, it takes a preset value. Default arguments can only be provided for the last parameters of the function. Example:

```
float distance (Coordinates a, Coordinates b = {0,0}) {
   Dif coordinates;
   dif.x = a.x-b.x;
   dif.y = a.y-b.y;
   return module (dif);
}
void g () {
   Coordinates r, t;
   ...
   Print (distance (r, t));
   Print (distance (r));
   ...
}
void f (int q, float r, int u = 4, int v = 5, int w = 6);
void h () {
   f (1,2.0,3); // f (1,2.0,3,5,6)
   f (1,2); // f (1,2,4,5,6)
}
void f2 (float r = 0, int u, int v, int w = 6); //Error
```

The use of default parameters in conjunction with the function overload may generate ambiguity for the compiler. Example:

```
void fn (int a);
void fn (int a, int b = 0);
void function () {
   fn (1,2); // Clearly refers to the second
   fn (1); // Which of the two do you mean?
}
```

You can define a function that accepts a variable number of parameters, of any type parameter. This is achieved using the ellipsis (fn (...)). A typical case is the function printf that accepts any number and type of parameters.

Functions that contain ellipsis may cause ambiguity with the function overload. Consider the following code:

```
void fn (int a, ...); // Function # 1
void fn (char * pa, ...); // Function # 2
void fn (int a, int b); // Function # 3
void function () {
   fn (1); // Refers to function # 1
   fn ("my name"); // Refers to function # 2
   fn (1, "my name"); // Refers to function # 1
   fn (1,2); // Does it refer to function # 1 or # 3?
}
```

1.3.8

Select the correct affirmations about default parameters.

Select one or more:

- are not valid with const parameters.
- are not valid if some parameter at its right has no default value.
- one parameter can have two or more default values.
- may conflict with function overloading.

1.3.9

There are three different zones of memory where you can create objects or variables :

Global memory: The variables defined outside functions and classes, and variables and attributes with the modifier **static** are located in this zone. The life of the objects or variables in this zone is identical to that of the program.

Stack memory: This memory zone stores the parameters and local objects of functions. These objects and variables exist while the function is in execution.

Dynamic memory (the heap): This memory zone store objects of any type. The creation and destruction of the objects are under the control of the programmer.

In C, getting and releasing dynamic memory is done manually using library functions such as **malloc** and **free**. C++ adds two operators, **new** and **delete**, for performing these tasks smartly.

1.3.10

Considering this code, select the correct answers.

```
int base = 5;
int f(int parm) {
   static int ncalls = 0;
   ncalls++;
   int size = parm + base + ncalls;
   int *v = (int *) malloc( size * sizeof int);
   v[0] = 1;
   for (int i=1; i < size; i++) { v[i] = i + v[i-1]; }
   int result = v[size -1];
   free (v);
   return result;
}
```

Variables in global memory: _____

Variables in the stack: _____

Variables in dynamic memory (the heap): _____

- *v
- *v and i
- base
- parm, ncalls, size, i and result
- parm, size, i, v and result
- parm, size and result
- v
- *v, base and ncalls
- base and ncalls
- v[]

1.3.11

The **new** operator accepts as a parameter the type of the object to be created, and it returns a pointer to a variable of this type created in dynamic memory; if **new** does not find enough free memory to create the object, it returns the NULL pointer.

The **delete** operator accepts a pointer as the parameter and frees the memory that it points to. The pointer must have been taken with the operator **new**.

Example:

```
int * p;
const int n = 30;
p = new int; // Creation of an integer
delete p; // Destruction of the integer
p = new int [10]; // Creating a vector of 10 integers
delete [] p; // Destruction of the previous vector
p = new int [n]; // Creating a vector of n integers
delete [] p;
```

1.3.12

Select the correct use of *malloc*, *free*, *new* and *delete*.

Select one or more:

- int *p = new int; delete p;
- int size = 5; int *p = new int[size]; delete []p;
- int size = 5; int *p = new int[size]; free(p);
- int *p = new int[]; delete p;
- int *p = (int *) malloc(sizeof int); delete p;

1.3.13 Using new and delete

Define a void function named "*usingNewDelete*" that accepts the size of an array of ints. The function must create an array of ints using **new** and fill it with the value of its size, and then the function must destroy the array using **delete**.

Classes and Objects

Chapter 2

2.1 Classes, objects and member functions (1/2)

📖 2.1.1

From an external point of view, the characteristics of an object are the functions that accept and the state it keeps. On the other hand, internally, an object has attributes, which determine its state, and member functions that act on it. The attributes of an object are variables inside the object.

There is a correspondence between the external and the internal vision of the object. An external state of the object is a specific set of values of its attributes. On the other hand, the execution of a member function of the object can change the values of its attributes, taking the object to a new state. The member functions of an object have several alias names: member functions, services, methods, or messages.

📝 2.1.2

Select the correct affirmation.

- From an external point of view, the characteristics of an object are only the state it keeps
- From an internal point of view, the characteristics of an object are only the functions that accept.
- The external vision of the object is independent of the internal one.
- The execution of a member function of the object can change the values of its attributes, taking the object to a new state

📖 2.1.3

When an object-based program is running, its objects are entities that behave according to a predetermined pattern. The classes are the definition of the characteristics of the objects. The objects are the elements that exist during the execution of the program. The relationship of an object to its class is similar to the variable to its type. Notice that the concept of class is mainly static while the object is dynamic.

📝 2.1.4

The relationship of an object to its class is similar to the variable to its type.

- True
- False

📖 2.1.5

We can define the Fraction class, and within it, we can declare two integer attributes, one that represents the numerator and another the denominator. Also, as member functions, we can define the addition, the multiplication and the division of two fractions, the conversion from a fraction to a float number, etc. A fraction instance will be an object of the Fraccion class. The object will be located in memory and will have two integers as its attributes, the numerator, and the denominator. For this object, we can call the functions defined in the class, for example, the inverse fraction, or the conversion to a float number.

📝 2.1.6

Taken into account the class *Fraction*. Select the correct solution to create some fractions.

- Create a new class for each needed fraction.
- Alter the class Fraction to keep inside more numerators and denominators.
- Create objects of the class Fraction.
- Add new member functions to the class Fraction.

📖 2.1.7

The member functions are the actions that objects can perform. Each member function has a name that must be a valid identifier according to the syntax of the language used. A member function can take parameters as a standard function; also, the member function can return a result.

The member function can be public or private. Private member functions are those that can only be invoked from another member function of the same class, and cannot be invoked from functions external to the class.

On the other hand, the functions commonly act on an object, but it also is possible to define member functions that do not need an object. These functions are static member functions. The static member functions are responsible for carrying out independent actions that do not act on a current object.

2.1.8

Select the correct affirmation.

- The member functions are enumerated to distinguish one of other.
- A member function does not accept parameters.
- The public member functions do not act on the current object.
- The static member functions do not act on the current object.

2.1.9

Classes in C++ are declared and defined using the keyword **class**, **struct** or **union**, then the class name, and followed, enclosed in curly brackets, the definition or declaration of member functions, and the attributes. The format is as follows:

```
[class | struct | union] Class_Name {
   Attribute1;
   Attribute2;
   ...
   Member function 1;
   Member function 2;
   ...
};
```

The notation of the classes is an adaptation of the syntaxis of the **struct** definition of the C language. A difference with C is that when defining variables of a struct type, the keyword struct is no longer necessary with the name of the type. Example:

```
struct DataType {
   int a;
   int b;
};
struct DataType data; // Correct in C, but incorrect in C++
DataType data; // Correct in C++, but incorrect in C
```

2.1.10

You can define a class using the keyword(s) _____, then the name, and followed, enclosed in _____, the definition or declaration of member functions, and the attributes.

- class, struct or union
- only struct

- angle brackets
- square brackets
- def
- Object
- curly brackets
- brackets
- only class
- parentheses

🕮 2.1.11

Each member function and attribute has an accessibility kind for the rest of the classes and functions. There are three kinds:

public: Everyone can access and modify the attribute or call the function; It is the default option for the **struct** and **union** keywords.

private: Only members of the same class can read and modify the attributes, and execute the functions; It is the default option for the **class** keyword.

protected: Only members of the same class and derived classes (see inheritance) can read and modify the attributes, and execute the functions.

A difference between **struct** and **class** is that the members of a struct are public by default, while the members of a class are private by default. A common practice is to use the "struct" keyword for C-like structures that do not implement member functions or inheritance and to use "class" to hide data in other cases.

Member functions can directly access the attributes of the object for which they are called. The same is true among member functions: they can call each other without specifying which object they apply to, this being the object for which the first call was made. In the example shown below, a MyClass class has a private integer attribute, called data, and public functions "int getValue()" and "void setValue(int)".

```
class MyClass {
   int data;
public:
   int getValue() { return data; }
   void setValue(int n) { data = n; }
};
```

To establish one of the three types of accessibility, you must write the associated keyword followed by a colon ":". The attributes and functions declared from that point will have that type of access established.

2.1.12

_____ : Only members of the same class can read and modify the attributes, and execute the functions. It is the default option for classes.

_____ : Only members of the same class and derived classes (see inheritance) can read and modify the attributes, and execute the functions.

_____ : Everyone can access and modify the attribute or call the function. It is the default option for **struct** and **union**.

- public
- protected
- private

2.1.13

All member functions of a class must be declared within it. The definition of a member function can be done in two ways: inside the class, and it implies its declaration; or outside the class, but still, the inside declaration is required.

```
class MyClass {
   int data;
public:
   int getValue() { return data; } // Declare and define
   void setValue(int n); // Only declare
};
void MyClass::setValue(int n) { // Only define
   data = n;
}
```

A defined member function within the class is assumed as an inline function. In the previous example, the int getValue() function is inline. An inline function is a function that the compiler will try to use as a macro. The call to an inline function will generate the expansión of its code instead of calling the function. The expansion is done in such a way that the semantics of the action remains.

When defining a member function outside the class, you must use its full name. The full name of a member function consists of the class name followed by "::" and the name of the function.

2.1.14

Complete the following sentences taken into account the code.

The member function "setData" is _____ the class. The member function "show" is _____ the class. The member function "changeName" is _____ the class.

```
class Student {
private:
   int idNumber; // Student's id number
   char* name; //Student's name
public:
   void setData(char* nam, int idNum);
   void show();
   void changeName(char * nam) {
      delete name;
      name = new char [strlen (nam) +1];
      strcpy (name, nam);
   }
};
void Student::setData(char* nam, int idNum) {
   name = new char [strlen (nam) +1];
   strcpy (name, nam);
   idNumber = idNum;
}
```

- declared inside and defined outside
- only declared inside
- declared and defined inside

2.1.15 Adding getPerimeter function to Rectangle class

Please, add a member function "getPerimeter" to the Rectangle class. The "getPerimeter" function returns the perimeter of the rectangle. Please, do not modify the class except by adding the member function requested.

```
class Rectangle {
public:
    float width;
    float height;
};
```

⌨ 2.1.16 Encapsulating Rectangle

Please, add the public member functions "*setWidth*", "*setHeight*", "*getWidth*" and "*getHeight*" to the *Rectangle* class and change to "*protected*" the attributes of the class.

- *setWidth* and *setHeight*: are void functions that accept a float number and change the corresponding attribute.
- *getWidth* and *getHeight*: are float functions that do not take parameters and return the corresponding attribute.

2.2 Classes, objects and member functions (2/2)

🕮 2.2.1

The objects are defined as a common variable but using the class name as its type. The access to the member functions or attributes of an object uses the same syntax that accessing fields of a struct in C. Objects use dot notation "." to access its attributes and member functions "object.member". Pointers to objects can use arrows "->" notation as a simplification of dot notation with asterisk "pointertoobject->member <=> (*pointertoobject).member".

Example:

```
int main () {
   MyClass object; // Object of the class MyClass
   MyClass *pointer; // Pointer to object of the class MyClass
   MyClass array[10]; // Array of objects of the class MyClass
   MyClass & referenceObject = array[5]; // Reference to
object of the class MyClass
   object.setValue(5);
   array[0].setValue(3);
   pointer = & array[1];
   pointer->setValue(object.getValue());
   referenceObject = * pointer;
   object.data = 3; // Compilation error. the data field is
not accessible
   return 0;
};
```

2.2.2

Select the **correct** executable code after this definition of elements.

```
class MyClass {
   int data;
public:
   int getValue() { return data; } // Declare and define
   void setValue(int n) { data = n; }; // Only declare
};
int main () {
   MyClass object; // Object of the class MyClass
   MyClass *pointer; // Pointer to object of the class MyClass
   MyClass array[10]; // Array of objects of the class MyClass
   MyClass & referenceObject = array[5]; // Reference to
object of the class MyClass
```

Select one:

- object->setValue(3);
- int v = pointer.getValue();
- pointer->data = 5;
- pointer = & referenceObject;

2.2.3

The compilation of one C++ source file can contain multiple declarations of a function, but only one definition of each function or class. You can not define a class twice in one file, but the class may be repeatedly defined in different source files (no header file).

The linkage of all compiled C++ source files of a program can include only one definition of each function or member function. That is, there must be only one definition of each function (not inline) in the source code of the whole program.

2.2.4

Select the correct answers.

Select one or more:

- The compilation of one C++ source file can not contain multiple declarations of a function.
- The compilation of one C++ source file can contain multiple definitions of each function or class.
- A class may be repeatedly defined in different source files.

- The linkage of all compiled C++ source files of a program can include only one definition of each function or member function.
- There must be only one definition of each function (not inline) in the source code of the whole program.

2.2.5 The header file of the Point class

We have developed a Point class, but we need to separate the declaration from the definition in a header file ".h" and a body file ".cpp".

```
class Point {
    int x;
    int y;
public:
    int getX() {return this->x;}
    int getY() {return this->y;}
    void setXY(int x, int y) {
        x = x < 0 ? 0  : x;
        y = y < 0 ? 0  : y;
        this->x = x;
        this->y = y;
    }
};
```

Please, write the declaration of the Point class in the "point.h" file assuming that all the member functions will be defined in "point.cpp".

point.h

```
/* class Point {
    int x;
    int y;
public:
    int getX() {return this->x;}
    int getY() {return this->y;}
    void setXY(int x, int y) {
        x = x <| 0 ? 0  : x;
        y = y <| 0 ? 0  : y;
        this->x = x;
        this->y = y;
    }
};
*/
```

⌨ 2.2.6 The body file of the Point class

We have developed a Point class, but we need to separate the declaration from the definition in a header file ".h" and a body file ".cpp".

```
class Point {
    int x;
    int y;
public:
    int getX() {return this->x;}
    int getY() {return this->y;}
    void setXY(int x, int y) {
        x = x < 0 ? 0  : x;
        y = y < 0 ? 0  : y;
        this->x = x;
        this->y = y;
    }
};
```

Please, write the definition of the Point class in the "point.cpp" file assuming that no member functions are defined in the header file "point.h".

point.cpp

```
/* class Point {
    int x;
    int y;
public:
    int getX() {return this->x;}
    int getY() {return this->y;}
    void setXY(int x, int y) {
        x = x <| 0 ? 0  : x;
        y = y <| 0 ? 0  : y;
        this->x = x;
        this->y = y;
    }
};
*/
```

🕮 2.2.7

The **static** modifier can be applied to both attributes and member functions of a class. In the case of an attribute, it means that you have a single attribute shared by all objects of the class and exists while the program is running. When applied to a

function, it determines that it can only access static attributes or static functions of the class. To access static members you can use the usual way or without specifying an object, but using its full name "classname::static_identifier". The static member functions do not have the "this" pointer. As shown in the example, the definition and initialization of static attributes are always done outside the class.

Below is an example of how to keep a counter of the number of existing objects in a class using a static attribute. The constructor increases the counter, and the destructor decreases the counter.

```
class Node {
   static int counter; // Class member attribute
public:
   Node() { // Constructor
      ...
      conter ++;
   }
   ~ Node() { // Destructor
      ...
      counter --;
   }
   static int getNumberOfObjects() {return counter;};
   ...
}
// The next line must go in a .cpp file as part of the
definition of the class
int Node::counter = 0;
```

2.2.8

Select the correct answers.

Select one or more:

- The static modifier applied to attributes indicates that the attribute can not be modified after its initialization.
- When the static modifier is applied to a member function indicates that it can no modify the attributes.
- To access static members you can use its full name "classname::static_name".
- The static member functions do not have the "this" pointer.
- The definition and initialization of static attributes are always done outside the class.

2.2.9 Limiting the values of the points

You may have noticed that the setXY member function limits the minimum values of "x" or "y" to zero. We now want to limit the maximum "x" and "y" values of points when modified by setXY. Please, add two static int attributes, e.g. maxX and maxY and modify setXY to add this feature. The default value for these limits is 10000. To set the maximum values, you must develop a static member function "setMaxXY".

Notice that the static attributes must be declared in the class and defined outside.

```
class Point {
    int x;
    int y;
public:
    int getX() {return this->x;}
    int getY() {return this->y;}
    void setXY(int x, int y) {
        x = x < 0 ? 0  : x;
        y = y < 0 ? 0  : y;
        this->x = x;
        this->y = y;
    }
};
```

exercise.cpp

```
class Point {
    int x;
    int y;
    // Declare here the static atributes
public:
    int getX() {return this->x;}
    int getY() {return this->y;}
    void setXY(int x, int y) {
        x = x <| 0 ? 0  : x;
        y = y <| 0 ? 0  : y;
        // Cut here to maxX and maxY the values of x and y
        this->x = x;
        this->y = y;
    }
    // Declare or define here the static function
};

// Define here the static atributes. Set the initial value to
10000
```

2.2.10

In case it is necessary, although it is not very convenient, you can grant access to other functions or classes to the non-public elements of your class. C++ provides this possibility by using **friend** functions and **friend** classes.

A friend function of a class is one that, without being a member of it, is allowed to access the non-public elements of the class. Any function, member, or not, of a class, can be declared a friend of one or several classes. When a class does a friend function/class declaration, it grants access to all its internal elements.

The following example illustrates the difference between member and friend functions. The declaration of friend functions breaks the criteria of modularity and protection of information. The only justification for declaring this type of function is when it is necessary to access internal elements of two classes at the same time, or for reasons of comfort in the operators' overload that will be seen later. If the friend statement applies to a class, all member functions of the class become friends. The way to declare a function or class friend is by making a declaration of it, within the class that is given the access permission, preceded by the reserved word **friend**.

Example:

```
class X {
    int a; // Private member
    // The friend statement can be anywhere in the class
    friend void friendFunction(X *, int);
public:
    void memberFunction(int);
};
void X::memberFunction(int i) {a = i;}
void friendFunction(X* o, int i) {
    o->a = i;
}
void f (){
    X obj;
    friendFunction(& obj, 10);
    obj.memberFunction(10);
}
```

2.2.11

Select the correct answers.

Select one or more:

- A friend function of a class is allowed to access the non-public elements of that class.
- Any function or member function can be declared friend of one or several classes.
- A class does a friend function/class declaration to grants access to all its non-public elements.
- If the friend statement applies to a class, all member functions of the class become friends.
- The way to declare a function or class friend is by making, within the class that is given the access permission, a declaration of it preceded by the reserved word friend.

🕮 2.2.12

The **non-static** member function has a pointer named "this" that points to the current object. The "this" pointer can be used to access the object as a whole (*this) or to disambiguate the collision of attribute names with variable names.

Example:

We have a class called Point, used to represent the coordinates of a point on the screen. We have defined, as members of the class Point, two integer attributes, x, and y and functions that allow initializing, displaying, hiding and moving the point. In this way, the Point class not only serves to save data but also to use correctly that data.

```
const int BLACK = 0;
const int WHITE = 0;
void putpixel(int x, int y, int color) {}

class Point {
   int x, y; // Default private
protected:
   void clear(); // Can only be called by member functions and
derived classes
   void write();
public: // From here the declared functions or attributes are
public
   void initialize(int x, int y);
   void moveTo(int x1, int y1);
};

void Point::clear() {putpixel (x, y, BLACK);}
void Point::write() {putpixel (x, y, WHITE);}
```

```
void Point::initialize(int x, int y){ // The parameters hide
the attributes
   this-> x = x; // We access the attributes using the "this"
pointer
   this-> y = y;
   write();
}

void Point::moveTo(int x1, int y1) {
   clear();
   x = x1; // We can access the attributes without "this"
because there are no variables that hide them
   y = y1;
   write();
}

int main () {
   Point v[10]; // Declaration an array of objects
   for (int i = 0; i <10; i ++) v [i].initialize(i, 2 * i);
   for (int i = 0; i <10; i ++) v [i].moveTo(i * 2, i * 3);
   return 0;
}
```

2.2.13

Select the correct answer.

- The "this" pointer can be used to access the object as a whole using "*this".
- The "this" pointer may generate a name collision problem.
- In the static member functions, the "this" pointer is NULL .

2.2.14 Fill an array using the this pointer

We want to add a new feature to the class Point that allows filling an array of points with the values of the current one. The function "fillArray" accepts an array of points and their size. Please, copy the current point to each element of the array using "this" (notice that this is a pointer).

```
class Point {
    int x;
    int y;
public:
    int getX() {return this->x;}
```

```
    int getY() {return this->y;}
    void setXY(int x, int y) {
        x = x < 0 ? 0  : x;
        y = y < 0 ? 0  : y;
        this->x = x;
        this->y = y;
    }
};
```

exercise.cpp

```
class Point {
    int x;
    int y;
public:
    int getX() {return this->x;}
    int getY() {return this->y;}
    void setXY(int x, int y) {
        x = x <| 0 ? 0  : x;
        y = y <| 0 ? 0  : y;
        this->x = x;
        this->y = y;
    }
    void fillArray(Point a[], int size) {
        // Please, write here the code to fill the array with
the current object
    }
};
```

2.3 Constructors and destructors

🕮 2.3.1

The attributes of an object must be initialized before being used. Non-initialization of objects is very dangerous since initial nonsense values can become fatal for some member functions. C++ provides mechanisms for defining special member functions called **constructors** that initialize objects when they are created.

The constructors are procedures with the same name as the class and do not return any type, even void. A constructor can be overloaded following the same overload rules of any other function. When an object is created, either global, local, or in the heap, one of the constructors is called automatically. These procedures transform a part of memory into a valid object. There may be a constructor without parameters or that all its parameters have default values. This constructor is

named the **default constructor**, and the compiler will call it automatically every time an object of this class is created, and no parameters are specified.

The Student class is shown as an example:

```
#include <string.h>
class Student {
private:
   int idNumber; // Student's id number
   char* name; //Student's name
public:
   Student(char* nam, int idNum) { // Constructor
      name = new char [strlen (nam) +1];
      strcpy (name, nam);
      idNumber = idNum;
   }
   // ...
};

int main () {
   Student student("Student test", 31); // Constructor called
   // ...
   return 0;
}
```

2.3.2

Select the correct answers.

Select one or more:

- The constructors are a replacement for the new operator for objects.
- The constructors are procedures do not return any type, even void.
- The constructors can have any name but the class name.
- A constructor can be overloaded following the same overload rules of any other function.
- After creating an object, the programmer must call a constructor.
- There may be a constructor without parameters or that all its parameters have default values. This constructor is named the default constructor.

⌨ 2.3.3 Adding constructors to the Point class

Please, add a default constructor that initializes the point coordinates x and y to zero. Also, add a constructor that initializes the point coordinates *x* and *y*. Notice that one constructor may do the work of the two.

```
class Point {
    int x;
    int y;
public:
    int getX() {return this->x;}
    int getY() {return this->y;}
    void setXY(int x, int y) {
        x = x < 0 ? 0  : x;
        y = y < 0 ? 0  : y;
        this->x = x;
        this->y = y;
    }
};
```

exercise.cpp

```
class Point {
    int x;
    int y;
public:
    // Add here the constructor(s)
    int getX() {return this->x;}
    int getY() {return this->y;}
    void setXY(int x, int y) {
        x = x <| 0 ? 0  : x;
        y = y <| 0 ? 0  : y;
        this->x = x;
        this->y = y;
    }
};
```

🕮 2.3.4

Some classes may need to "deinitialize" the objects once their life is over. A special procedure named **destructor** does this task. The **destructor** has the same name as its class but is preceded by a tilde "~". The destructor is called automatically when:

- An object (in the heap) is removed with the **delete** operator.

- The scope where an object was created is abandoned. e.g., when a function ends, the destructor is called for parameters and local variables.
- The main function ends the destructor of global and static objects are called.
- The destructor has no parameters, does not return any type, and cannot be overloaded.

The *student* class is shown as an example:

```
#include <string.h>
class Student {
private:
   int idNumber; // Student's id number
   char* name; //Student's name
public:
   Student(char* nam, int idNum); // Constructor
   ~Student(); // Destructor
   void show();
   void changeName(char * name);
};
Student::Student(char* nam, int idNum) {
   name = new char [strlen (nam) +1];
   strcpy (name, nam);
   idNumber = idNum;
}
Student::~Student() {
   delete name;
}
int main() {
   Student student("Student test", 31);
   student.show();
   student.changeName("New name");
   student.show();
   return 0;
   // The destructor is automatically called at this point
}
```

2.3.5

Select the correct answers.

Select one or more:

- The destructor has the same name as its class but preceded by a tilde "~".

- The destructor is called automatically when an object is removed from the heap with the function "free()".
- The destructor needs to be called manually for parameters and local variables.
- When the "main" function ends, the destructor of global and static objects is automatically called.
- The destructor cannot be overloaded.

⌨ 2.3.6 Adding a destructor to the Point class

We want to know the number of objects of type Point that exists during the program execution. We have added a static counter and a static function to get this value (the number of current objects). Please, add a default constructor that initializes the point coordinates x and y to zero and increments the counter of objects, also add a destructor that decrements the counter of objects.

```
class Point {
    int x;
    int y;
    static int counter;
public:
    // Add here the constructor and destructor
    int getX() {return this->x;}
    int getY() {return this->y;}
    void setXY(int x, int y) {
        x = x < 0 ? 0  : x;
        y = y < 0 ? 0  : y;
        this->x = x;
        this->y = y;
    }
    static int getCounter() {
        return counter;
    }
};

int Point::counter = 0;
```

exercise.cpp

```
class Point {
    int x;
    int y;
    static int counter;
public:
```

```
    // Add here the constructor and destructor
    int getX() {return this->x;}
    int getY() {return this->y;}
    void setXY(int x, int y) {
        x = x <| 0 ? 0  : x;
        y = y <| 0 ? 0  : y;
        this->x = x;
        this->y = y;
    }
    static int getCounter() {
        return counter;
    }
};

int Point::counter = 0;
```

📖 2.3.7

It is highly recommended to use only the new and delete operators to manage objects in the heap. The use of C language allocation and deallocation functions (malloc, free, etc.) for objects leads to the loss of the automatic and required use of constructors and destructors.

📝 2.3.8

You can safely mix the use of the **new** and **delete** operators, and the "**malloc**" and "**free**" C language functions, for managing objects in the heap.

- False
- True

📖 2.3.9

The following example shows another definition of the Stack class using a constructor with a default parameter. Also, the destructor is defined because the **new** operator is used to create one of the attributes.

```
class Stack {
   protected:
   int nelements;
   int maxElements;
```

```
   int *vector;
public:
   Stack(int t = 100); // Constructor with one parameter and
default constructor
   ~Stack() {delete vector;} // Destructor to free dynamic
memory
   bool isEmpty() {return nelements == 0;}
   bool isFull() {return nelements == maxElements;}
   void push(int);
   void pop();
   int top();
};
Stack::Stack (int t) {
   vector = new int[t]; // The vector is created by taking
dynamic memory
   nelements = 0;
   maxElements = t;
}
void Stack::push(int i) {vector [nelements ++] = i;}
void Stack::pop() {--nelements;}
int Stack::top() {return vector [nelements-1];}
void f () {
   Stack p; // The constructor is automatically called
   // default value of the parameter
   p.push(3); // The interface is identical, it is not
necessary
   // modify the rest of the code
   p.push(5);
   Stack* pp;
   pp = new Stack(20); // The constructor is called being 20
the value of the parameter
   pp-> push(p.top());
   //...
   delete pp; // The destructor is automatically called for *
pp
} // The destructor is automatically called for p
```

2.3.10

Select the correct answer about the Stack class.

```
class Stack {
   protected:
   int nelements;
```

```
   int maxElements;
   int *vector;
public:
   Stack(int t = 100); // Constructor with one parameter and
default constructor
   ~Stack() {delete vector;} // Destructor to free dynamic
memory
   bool isEmpty() {return nelements == 0;}
   bool isFull() {return nelements == maxElements;}
   void push(int);
   void pop();
   int top();
};
Stack::Stack (int t) {
   vector = new int[t]; // The vector is created by taking
dynamic memory
   nelements = 0;
   maxElements = t;
}
void Stack::push(int i) {vector [nelements ++] = i;}
void Stack::pop() {--nelements;}
int Stack::top() {return vector [nelements-1];}
void f () {
   Stack p; // The constructor is automatically called
   // default value of the parameter
   p.push(3); // The interface is identical, it is not
necessary
   // modify the rest of the code
   p.push(5);
   Stack* pp;
   pp = new Stack(20); // The constructor is called being 20
the value of the parameter
   pp-> push(p.top());
   //...
   delete pp; // The destructor is automatically called for *
pp
} // The destructor is automatically called for p
```

- The class Stack has two constructors.
- The member function "push" fails if the stack is full.
- The member function "top" returns 0 if the stack is empty.

2.3.11

When code contents complex expressions, it is necessary to store intermediate data that is not accessible. Example:

```
int f () {..}
void g ()
{
   int x = 0, y = 1, z = 2;
   x = (x-y) / (z + y * x) + y-f ();
   ...
```

The intermediate values of subexpressions as (x-y), (y * x), f (), etc. must be stored somewhere. In many cases, the evaluation of arithmetic expressions uses a stack of evaluation; also, the CPU registers may store temporary results. When these expressions involve objects from user-defined classes, the same thing happens: the compiler may construct and destroy hidden objects if necessary. This happens, for example, in objects returned by a function.

Notice that the "return by value" of a function requires copying the value returned in a hidden object. The hidden object must be available in the context of the function call.

There is also the possibility of using unnamed objects that must be used on the fly, for example, passing it to a function. The format of creating an unnamed object is equivalent in form to a call to the class constructor. Example:

```
Point g(Point p) {
   ...
   return p; // return by value
}
void fn() {
   Point p(10.10), r(3,3);
   r = g(p); // You call g, passing the object p, the returned
object (hidden) is asigned
   g(Point(10.20)); // You call g, passing an unnamed object,
the returned object (hidden) is ignored
} // The destructor of p, r and the hidden unnamed object is
called
```

2.3.12

Hidden objects do not need to be destroyed due they do not affect the program execution.

- False
- True

🕮 2.3.13

A constructor that accepts a single parameter, specifies a conversion from the type of the parameter to the type of the class. This automatic conversion is applied whenever it is needed an object of the class and appears a value of the type of the parameter of the constructor. The conversion is indeed the creation of an object of the class using the constructor that accepts that type of parameter.

```
class Fraction {
    int numerator;
    int denominator;
public:
    Fraction(int n, int d = 1) {
        numerator = n;
        denominator = d;
    }
    Fraction plus( Fraction o ) {
        //...
    }
    // ..
};

void f() {
    Fraction f1 = 3; // Equivalent to: Fraction f1(3, 1);
    Fraction f2(3,4);
    f1 = f2.plus(5); // Equivalent to: f1 =
f2.plus(Fraction(5, 1));

}
```

⌨ 2.3.14 Conversor from Point to Point3D

We have two classes *Point* and *Point3D*, and we want to do automatic conversions from *Point* to *Point3D*. The class *Point3D* has three values *x*, *y*, *z*. When converting from *Point* to *Point3D*, *x* and *y* take the same values, and z takes the value zero. Please, add a constructor to the class Point3D to allow code as following:

```
Point p;
p.setXY(2,3);
Point3D p3d(p);
```

```
Point3D p3d1 = p;
```

The classes *Point* and *Point3D* are as follows:

```
class Point {
    int x;
    int y;
public:
    int getX() {return this->x;}
    int getY() {return this->y;}
    void setXY(int x, int y) {
        this->x = x < 0 ? 0  : x;
        this->y = y < 0 ? 0  : y;
    }
};
class Point3D {
    int x;
    int y;
    int z;
public:
    Point3D() {
        this->x = 0;
        this->y = 0;
        this->z = 0;
    }
    int getX() {return this->x;}
    int getY() {return this->y;}
    int getZ() {return this->z;}
    void setXYZ(int x, int y, int z) {
        this->x = x < 0 ? 0  : x;
        this->y = y < 0 ? 0  : y;
        this->z = z < 0 ? 0  : z;
    }
};
```

exercise.cpp

```
class Point {
    int x;
    int y;
public:
    int getX() {return this->x;}
    int getY() {return this->y;}
    void setXY(int x, int y) {
        this->x = x <| 0 ? 0  : x;
        this->y = y <| 0 ? 0  : y;
```

```
    }
};

class Point3D {
    int x;
    int y;
    int z;
public:
    Point3D() {
        this->x = 0;
        this->y = 0;
        this->z = 0;
    }

    // Add here the conversion constructor

    int getX() {return this->x;}
    int getY() {return this->y;}
    int getZ() {return this->z;}
    void setXYZ(int x, int y, int z) {
        this->x = x <| 0 ? 0  : x;
        this->y = y <| 0 ? 0  : y;
        this->z = z <| 0 ? 0  : z;
    }
};
```

2.3.15

If we add a constructor whose only parameter is "**const char ***", will the function *f()* be valid?

```
class Fraction {
    int numerator;
    int denominator;
public:
    Fraction(int n, int d = 1) {
        numerator = n;
        denominator = d;
    }
    Fraction(const char *s) {
        // ...
    }
    Fraction plus( Fraction o ) {
        //...
```

```
    }
    // ...
};

void f() {
    Fraction f1 = 3;
    Fraction f2(3,4);
    f1 = f2.plus("5");
}
```

- True
- False

2.4 Source files organization

📖 2.4.1

Medium and large programs need to organize the code and use modularity. In the case of C++, thanks to the possibility of separate compilation, the different classes, functions, type definitions, etc. can be distributed in various source files that function as modules. You can use the following rules to organize the source files in the form of modules:

The definition of one or more related classes goes in a ".cpp" file. The definition means the development of member functions outside of classes and the definition of static attributes.

Each class may be declared in a header file with its name with the extension ".h"; or maybe located, with other classes, in a header file with the name of the module with the extension ".h". The class declaration is the "class" keyword with the class body (class .. {..};).

Classes internal to a module may go declared and defined in a ".cpp" file.

2.4.2

Select the correct affirmation.

- The definition of a class is the development of member functions outside of classes and the definition of static attributes.
- A class definition goes in a header file.
- A class definition is the "class" keyword with the class body (class .. {..};).

2.4.3

There are other source file extensions commonly used. For body source files: C, cc, cxx, and c++. For header files: H, hh, hpp, and h++.

Notice that there is no official extension, and the compiler accepts any extension.

2.4.4

Select the correct affirmations.

- The official extension for C++ source files are "cpp" and "h"
- Common source file extensions for C++ are cpp, C, cc, cxx, and c++
- Common header file extensions for C++ h, H, hh, hpp, and h++

2.4.5

The purpose of the ".cpp" source files is to be compiled separately. Each ".cpp" file may be compiled without the compilation or existence of other ".cpp" files. The program is generated by linking all the compiled ".cpp" files, commonly files with ".o" extension, that compound the program.

The purpose of the ".h" header files is to be included in one or more ".cpp" files to inform the compiler about the declaration of elements that may be defined elsewhere. The header files may also contain definitions that can be compiled multiple times as the definition of inline functions or templates classes.

2.4.6

Select the correct affirmations.

- All ".cpp" files of a program need to be compiled at the same time.
- The program is generated by linking all the compiled header files.
- The purpose of "X.h" header file is to allow the use of elements defined in "X.cpp" in others ".cpp" or ".h" files.
- The header files may contain declarations of elements that may be defined elsewhere.

📖 2.4.7

A definition or declaration of a class or a **typedef** cannot appear two or more times in a ".cpp" file. But the inclusion of different header files in a ".cpp" may lead to this problem. This type of error is avoided by establishing a different macro in each header file. This macro will act as a flag to know if the header file has already be included in the current ".cpp" file. Example:

File: db.h

```
#ifndef INC_DB
#define INC_DB
// Here comes the content of the header file that will be
included only one time
#endif
```

📝 2.4.8

Preprocessor instructions are used to avoid the multi-inclusion of one header file. A specific macro is defined in each header file to check if the file has already been included.

- True
- False

⌨ 2.4.9 Avoiding multidefinition errors in point.h

We have developed a *Point* class, and we separate the declaration from the definition of the class. The declaration of the class is at the "point.h" file, and we found that in some source file that includes twice "point.h" the compilation fails. Please, add preprocessor instructions to avoid multi-inclusion errors.

```
class Point {
    int x;
    int y;
public:
    int getX() {return this->x;}
    int getY() {return this->y;}
    void setXY(int, int);
};
```

📖 2.4.10

Each source file contain logically related elements. These relationships can be expressed in C++ using namespaces. A namespace is a mechanism to establish a related group of language elements as functions, classes, constants, etc. The namespace format is:

```
namespace [Space_Name] {
   ... // Declarations and definitions
}
```

Within the namespace, you can make definitions and declarations of all kinds. When using an element of a namespace, outside the namespace, we use the qualifier Space_Name::. Example:

```
namespace A {
   struct S {...};
   void f(S x) {...}
}
void g() {
   A::S y;
   A::f(y);
}
```

You can make declarations within the namespace and make the definitions outside, using the namespace qualifier.

```
namespace B {
   void f(S x); // Declaration
}
void B::f(S x) { //Definition
}
```

The "using" statement eliminates the need for a fully qualified name. From the "using" statement, it is possible to use the element inside the namespace directly without the qualifier.

Format:

```
using Space_Name::name;
```

Example:

```
void g() {
   using A::f;
   A::S y;
   f(y); // It is A::f
```

```
}
```

With the following format, it is also possible to indicate that all names in a namespace can be used:

```
using namespace Space_Name;
```

Example:

```
void g() {
   using namespace A;
   S y; // A::S
   f(y); // A::f
}
```

2.4.11

Select the correct affirmations.

- A namespace is a C++ mechanism to establish a group of logically related elements.
- To use an element of a namespace, outside the namespace, we can use the qualifier Space_Name::
- The namespace must contain the full declaration and definition of its elements.
- The "import" statement eliminates the need for a fully qualified name. From the "import" statement, it is possible to use the element inside the namespace directly without the qualifier.
- With the following format, it is also possible to indicate that all names in a namespace can be used: "using namespace Space_Name;".

2.4.12 Using class Rectangle in a namespace

We have developed a *Rectangle* class inside the namespace "Polygon". Please, write a function "perimeters" that returns the sum of the perimeter of each Rectangle in an array. The function accepts an array of pointers to Rectangles and the size of the array. The class *Rectangle* has the following declaration:

```
class Rectangle {
    float width;
    float height;
public:
    Rectangle(float w, float h);
    float getWidth();
```

```
    float getHeight();
    float getPerimeter();
};
```

2.4.13 Adding the Point class to a namespace

We have developed the *Point* class. Please, modify the code to add the class *Point* to the namespace "Coordinate".

```
class Point {
    int x;
    int y;
public:
    Point(int x = 0, int y = 0) {
        setXY(x, y);
    }
    int getX() {return this->x;}
    int getY() {return this->y;}
    void setXY(int x, int y) {
        this->x = x < 0 ? 0  : x;
        this->y = y < 0 ? 0  : y;
    }
};
```

exercise.cpp

```
// Modify the code to define the class Point inside namespace
Coordinate
class Point {
    int x;
    int y;
public:
    Point(int x = 0, int y = 0) {
        setXY(x, y);
    }
    int getX() {return this->x;}
    int getY() {return this->y;}
    void setXY(int x, int y) {
        this->x = x <| 0 ? 0  : x;
        this->y = y <| 0 ? 0  : y;
    }
};
```

2.4.14

It is possible to define an anonymous namespace in each file. The purpose of these namespaces is to prevent the elements declared within it from being used outside its source file. The elements defined in the anonymous namespace can be used without qualifier within its source file; however, it is not possible to use them from another file since the unnamed namespace of each source file is considered to be different.

It is possible to set aliases for namespaces. Format:

```
namespace aliasname = Space_Name
```

From this point, it has the same effect to use *aliasname* or *Space_Name*.

Namespaces are open in the sense that they can be made up of several separate declarations of the same namespace, all of which form a single namespace.

Example:

```
namespace A {
   void f ();
}
...
namespace A {
   void g ();
}
```

Here, f and g belong to the same namespace. The following recommendations may be used to construct a namespace correctly:

The definition of a namespace must be in a ".cpp" file with the same name. The ".cpp" file must contain the definition of member functions outside of classes and the definition of static attributes. All this is enclosed in a namespace.

The interface of each namespace must be in another file with the extension ".h". The interface would be the class declarations within the namespace.

In the header file ".h" a macro is established to avoid errors in case of multiple inclusions.

Example:

```
#ifndef INC__MY_SPACE
#define INC__MY_SPACE
namespace my_space {
   ...
```

```
}
#endif
```

2.4.15

Select the correct affirmations.

- We can use an anonymous namespace to prevent the elements declared within it from being used from other source files.
- It is possible to set aliases for namespaces. Format: "namespace aliasname = Space_Name;". From this point, it has the same effect to use one name or another.
- Namespaces are open in the sense that they can be made up of several separate declarations of the same namespace, all of which form a single namespace.

Operators

Chapter 3

3.1 Operator overloading

📖 3.1.1

The C++ programming language allows you to **overload most of the existing** operators for the user-defined classes. The operators allow the new classes to resemble the predefined types in their syntax. It also solves some severe problems that can occur with the assignment operator if we could not redefine it.

The only operators that cannot be overloaded are: ".", ". *" and "::". The programmer can not invent new operators or changes the precedence of the existing ones.

The overload of an operator is done by defining a function with the name "**operator operator_symbol**". E.g. "**operator +**". Notice that between "**operator**" and the **operator_symbol** you can write spaces.

📝 3.1.2

The C++ programming language allows the programmer to invent new operators or changes the precedence of the existing ones.

- False
- True

📖 3.1.3

The operators can be overloaded in the same way as functions, and, as such, they can be either as a member function of a class or as a non-member function. If the operator is defined as a member function of a class, the left operand is the object to which the function is applied; the other operand will be the parameter passed to the function. In the case that it is defined as a non-member function, the parameters will be the operands. In this second case, these functions may be established as a friend of the class of the operands.

The main difference between the two forms of operators overloading is the automatic conversion of operands. If the function is a member, it is only possible to apply automatic converters to the second operand. If the function is not a member function, automatic conversions can happen for both operands.

In the case of unary operators, member functions without parameters and non-member functions with a single parameter must be used.

3.1.4

Select the correct affirmations.

- The operator overloading must be done with member functions.
- If the operator defined as a member function of a class, the left operand is the object to which the function is applied.
- Automatic conversions are applied only to parameters of the operator functions.
- The unary operators cannot be overloaded.

3.1.5

An example of the use of operator overloading is the *Complex* class. A complex number consists of two parts: a real and an imaginary part. The operators overloading allow adding to the class the common operations over complex numbers.

The *Complex* class is as follows:

File "complex.h":

```
#ifndef INC_COMPLEX
#define INC_COMPLEX
class Complex {
   float imaginary, real;
public:
   Complex (float r = 0, float i = 0) {imaginary = i; real =
r;};
   Complex operator = (Complex c) { // It is not necessary to
overload it in this case
      // The operator = cannot be friend
      imaginary = c.imaginary;
      real = c.real;
      return * this;
   };
   friend Complex operator + (Complex c1, Complex c2)  {
      return Complex(c1.real + c2.real, c1.imaginary +
c2.imaginary);
   };
   Complex operator * (Complex c)   {
      Complex res;
      res.real = real * c.real - imaginary * c.imaginary;
      res.imaginary = real * c.imaginary + imaginary * c.real;
```

```
      return res;
   }
};
#endif
void f () {
   Complex c1, c2 (1.1), c3(2), c4(2.1, 3.2);
   c1 = c2 * c3 + (c3 + 8) * c4;
   c2 = 10 + c1; // Equivalent to c2.operator = (operator+
(Complex (10), c1))
   //c4 = 10 * c2; // Error, operator * cannot be applied
since
   // the expression is equivalent to c4.operator =
(10.operator * (c2))
}
```

3.1.6

Select the valid instructions that can follow the next code.

```
class Complex {
   float imaginary, real;
public:
   Complex (float r = 0, float i = 0) {imaginary = i; real =
r;};
   friend Complex operator + (Complex c1, Complex c2) {
/*...*/ };
   Complex operator - (Complex c) { /*...*/ };
   Complex operator * (Complex c) { /*...*/ };
};
void f() {
   Complex c1, c2 (1.1), c3(2), c4(2.1, 3.2);
   // Here come the instruction
```

Select one or more:

- c1 = c2 - c3
- c1 = c2 - 4
- c1 = c3 * c2 - 4
- c1 = 4 - c3 * c2
- c1 = 4 * c2
- c1 = 4 + c2

⌨ 3.1.7 Adding operator+ to the Point class as a member function

We have developed a Point class, but we think that it will be a good idea to have a plus ("+") operation for points. Adding two points will result return a new one with the x and y coordinates added. Please, add a **member function** that accepts a point and returns a new point with x as the two x of the current object and parameter object added and the same for y.

```
class Point {
    int x;
    int y;
public:
    Point(int x=0, int y=0) {
        setXY(x, y);
    }
    int getX() {return this->x;}
    int getY() {return this->y;}
    void setXY(int x, int y) {
        this->x = x < 0 ? 0  : x;
        this->y = y < 0 ? 0  : y;
    }
};
```

exercise.cpp

```
class Point {
    int x;
    int y;
public:
    Point(int x=0, int y=0) {
        setXY(x, y);
    }
    int getX() {return this->x;}
    int getY() {return this->y;}
    void setXY(int x, int y) {
        this->x = x <| 0 ? 0  : x;
        this->y = y <| 0 ? 0  : y;
    }
    // Add here operator+ member function
};
```

⌨ 3.1.8 Adding operator+ to the Point class as a non-member function

We have developed a Point class, but we think that it will be a good idea to have a plus operation for points. Adding two points will result return a new one with the x

and y coordinates added. Please, add a **non-member function** that accepts two points and returns a new point with x as the two x of the parameters object added and the same for y.

```
class Point {
    int x;
    int y;
public:
    Point(int x=0, int y=0) {
        setXY(x, y);
    }
    int getX() {return this->x;}
    int getY() {return this->y;}
    void setXY(int x, int y) {
        this->x = x < 0 ? 0  : x;
        this->y = y < 0 ? 0  : y;
    }
};
```

exercise.cpp

```
class Point {
    int x;
    int y;
public:
    Point(int x=0, int y=0) {
        setXY(x, y);
    }
    int getX() {return this->x;}
    int getY() {return this->y;}
    void setXY(int x, int y) {
        this->x = x <| 0 ? 0  : x;
        this->y = y <| 0 ? 0  : y;
    }
};
// Add here operato+ non-member function
```

🕮 3.1.9

It is possible to establish conversion operators from the class to which it belongs to another class or another predefined type. The format is a function that does not specify its returns type, and it has no parameters and whose name is "operator type".

Example:

```
class Student {
private:
   int idNumber; // Student's id number
   char * name; //Student's name
public:
   operator int() {return idNumber;} // Student to int
converter
   operator char *() {return name;}; // Student to char*
converter
   operator Y(); // Student to the class Y converter
};
void f (Student peter) {
   int i = peter; // i = peter.operator int ()
   char c [100];
   strcpy(c, peter); // strcpy(c, peter.operator char * ())
}
```

3.1.10

Select the correct functions name.

```
using namespace std;
class Student {
private:
   int idNumber; // Student's id number
   char * name; //Student's name
public:
   _____ () {return name;};
   _____ () {return idNumber;}
   _____ () { return string("Exp: ") + to_string(idNumber) + "
" + name;};
void f (Student peter) {
   int i = peter; // i = peter.operator int ()
   string peters = peter;
   cout << peters << endl;
}
```

- operator Y
- operator int
- operator string
- operator char *

3.1.11 Adding a conversion operator from Point to string

We have developed a *Point* class, but we think that it will be a good idea to add a conversion from points to strings. Please, add a conversion operator member function that returns a string. The string that represents a point is an open parenthesis, the value of *x*, a comma, the value of *y*, and an end parenthesis. For example, *Point(3,5)* returns the string "(3,5)". Remember using function *to_string* to convert int to string and the operator *+* to concatenate strings.

```
class Point {
    int x;
    int y;
public:
    Point(int x=0, int y=0) {
        setXY(x, y);
    }
    int getX() {return this->x;}
    int getY() {return this->y;}
    void setXY(int x, int y) {
        this->x = x < 0 ? 0  : x;
        this->y = y < 0 ? 0  : y;
    }
};
```

exercise.cpp

```
class Point {
    int x;
    int y;
public:
    Point(int x=0, int y=0) {
        setXY(x, y);
    }
    int getX() {return this->x;}
    int getY() {return this->y;}
    void setXY(int x, int y) {
        this->x = x <| 0 ? 0  : x;
        this->y = y <| 0 ? 0  : y;
    }
    // Add here operator string member function
};
```

3.2 Assignment operator and copy constructor

📖 3.2.1

The assignment operator

There is a default assignment operator for each class when no other one is defined. The default assignment operator does the assignment of each of the attributes of the right operand on the left operand and returns a reference to the left operand. There are some situations in which it is essential to overload the assignment operator for the proper running of the class. Commonly the assignment operator needs to be overloaded when the class has a destructor. The overloaded assignment operator may need to replicate the de-initialization process of the destructor. The assignment operator must be a member of the class; that is, it cannot be a non-member function, even if it is a friend function.

Example of what the default assignment operator does:

```
class Complex {
   float imaginary, real;
public:
   Complex (float r = 0, float i = 0) {imaginary = i; real =
r;};
   Complex& operator = (Complex& c) { // Equivalent to the
default assignment operator
      imaginary = c.imaginary;
      real = c.real;
      return * this;
   };
   // ...
};
```

📝 3.2.2

Select the correct affirmations.

- There is a no default assignment operator for the user-defined classes.
- The default assignment operator does the assignment of each of the attributes of the right operand on the left operand and returns a reference to the left operand.
- Commonly the assignment operator needs to be overloaded when the class has a destructor.
- The assignment operator must be a friend non-member function.

📖 3.2.3

The need for overloading the assignment operator.

```
class Stack {
protected:
   int nelements;
   int maxElements;
   int *vector;
public:
   Stack (int t = 100) {
       vector = new int [t]; // The vector is created by
taking dynamic memory
       nelements = 0;
       maxElements = t;
    }
    ~Stack () {delete vector;} // Destructor to free dynamic
memory
   void push(int i) { /* ... */ };
   // ...
};
void f(){
   Stack p1, p2(33);
   p1.push(3);
   p1.push(5);
   p2 = p1;  // Assign the attributes of one object to
another. Wrong instruction
} // The destructor is automatically called for p1 and p2
```

Why is the instruction p2 = p1 an erroneous instruction?

It is wrong because the default assignment is not explicitly overloaded, all it does is assign the attributes of one object in the other. This results in three bad situations: The assignment changes the value of the attribute "vector", losing the previous pointer, and doing impossible to free it. The p1 and p2 objects share the same array of elements and are dependant, everything done in p1 may affect p2 and vice-versa. The destructor of p2 releases the array it uses. The destructor of p1 automatically attempts to free the same memory area.

📝 3.2.4

In the following code, why is the instruction p2 = p1 an erroneous instruction?

```
class Stack {
protected:
   int nelements;
   int maxElements;
   int *vector;
public:
   Stack (int t = 100) {
       vector = new int [t];
       nelements = 0;
       maxElements = t;
    }
    ~Stack () {delete vector;}
   void push(int i) { /* ... */ };
   // ...
};
void f(){
   Stack p1, p2(33);
   p1.push(3);
   p1.push(5);
   p2 = p1;
}
```

Select one or more:

- The assignment changes the value of the attribute p2.vector, losing the previous pointer.
- The two objects share the same array of elements.
- The destructor will try to releases twice the same array.
- All other answers are erroneous.

🕮 3.2.5

To **overload the assignment operator** we add a prototype declaration in the *Stack* class:

```
Stack & operator = (const Stack &);
```

and we define the function outside the class:

```
Stack & Stack :: operator = (const Stack & s) {
    if (maxElements != s.maxElements) {
        maxElements = s.maxElements;
        delete [] vector;
        vector = new int [maxElements];
```

```
    }
    nelements = s.nelements;
    for (int i = 0; i < nelements; i ++) vector [i] = s.vector
[i];
    return * this;
}
```

3.2.6

What line of the assignment operator takes into account the action the destructor does?

```
  Stack & Stack :: operator = (const Stack & s) {
1    if (maxElements != s.maxElements) {
2        maxElements = s.maxElements;
3        delete []vector;
4        vector = new int [maxElements];
     }
5    nelements = s.nelements;
6    for (int i = 0; i < nelements; i ++) vector[i] =
s.vector[i];
7    return *this;
  }
```

3.2.7

The problem of self-assignment

Often the parameter in the assignment operator is a reference. In this case, when developing the assignment operator, it is necessary to take into account that the current object and the parameter may be the same.

```
Stack p1, p2 ;
...
p1 = maxStack (p1, p2); // Return a reference to the stack
with the largest element
```

In the previous example, p1 may be assigned to itself, in which case the assignment operator might not work well. In the implementation shown in the *Stack* class, there would be no problem, since we delete the array of the current object only if it is of a different size than the parameter. If we eliminate this condition, the assignment will work in all cases except in the self-assignment.

3.2.8

What line of the assignment operator avoids the self-assignment problem?

```
  Stack & Stack :: operator = (const Stack & s) {
1     if (maxElements != s.maxElements) {
2         maxElements = s.maxElements;
3         delete []vector;
4         vector = new int [maxElements];
      }
5     nelements = s.nelements;
6     for (int i = 0; i < nelements; i ++) vector[i] =
s.vector[i];
7     return *this;
  }
```

3.2.9 Adding an assigment operator to the Student class

We have developed a *Student* class that saves the name and the id of students. We keep the name as an array of chars. All was correct until we assign *Student* objects. Please, resolve the problem by adding an assignment operator. Necessary: you must use references to pass or return *Student* to the "operator=" member function.

```
class Student {
    int id;
    char* name;
public:
    Student(int id, const char* name) {
        this->id = id;
        this->name = new char[strlen(name)+1];
        strcpy(this->name, name);
    }
    ~Student() {delete this->name;}
    const char* getName() {return this->name;}
    int getId() {return this->id;}
};
```

exercise.cpp

```
#include <|string.h>
class Student {
    int id;
    char* name;
public:
    Student(int id, const char* name) {
```

```
        this->id = id;
        this->name = new char[strlen(name)+1];
        strcpy(this->name, name);
    }
    ~Student() {delete this->name;}
    const char* getName() {return this->name;}
    int getId() {return this->id;}
    // Write here the assigment operator
};
```

🕮 3.2.10

The need for a copy constructor

Parameters of pointers to objects or references work in the usual way. But when passing objects by value, a copy of the external object is made in the internal one. Passing each object requires copying each of the attributes of the external object into the attributes of the internal one. Below is an example of passing an object to a function by value.

```
int stackSum(Stack stp) {
    int sum = 0;
    while (! stp.isEmpty() ) {
        sum += stp.top();
        stp.pop();
    }
    return sum;
}

void g() {
    Stack stack1;
    stack1.push(3);
    stack1.push(5);
    stack1.push(13);
    int sum;
    sum = stackSum(stack1);
}
```

Why the instruction "**sum = stackSum(stack1);**" is wrong?

It is incorrect because the default copy of stack1 in the stp parameter is carried out by copying the attributes of stack1 in stp, which results in two disastrous situations:

stack1 and stp share the same array, and any possible modification on the array made by stp may affect stack1. When the execution of the function "sumStack" ends, the destructor for stp is automatically called, and it releases the array it uses. After the end of sumStack, stack1 can use the array already freed by stp. Once the execution of g ends, stack2 is automatically destroyed, trying to free the same array again.

🕮 3.2.11

The return of objects by value

When a function calls another that returns an object, a temporary and non-visible object is created. The temporary object exists in the scope of the calling function and is a copy of the object returned by the called function. The copy constructor will initialize the new object. The copy constructor is a constructor with only one parameter of type reference to an object of the same class. The classes that do not create a copy constructor, have a default one that copies (using the proper copy constructor) each attribute from the original object to the new one.

```
Stack generateStack(int n) {
    Stack generated;
    for (int i = 0; i < n; i++) generated.push(i);
    return generated;
}

void g() {
    Stack st;
    st = generateStack(10);
    st.push(3);
    // ...
}
```

In the previous example, the process is carried out in the following way: when the execution of "generateStack" ends, the object "generated" is first returned to g, making a copy over the temporary object that exists in the function "g", and then the destructor is called for the object "generated". This copy of the object returned gives two harmful situations:

- The two objects share the same array, and they are not independent.
- The destructor of the object "gen" releases the array used. When the temporary object is automatically destroyed, it tries to free the same array again.

3.2.12

Select the correct affirmation.

- When a function returns an object, a temporary and non-visible object is created in the scope of the calling function.
- The copy constructor is a constructor with only one parameter of type pointer to an object of the same class.
- The classes that do not create a copy constructor have a default one that assigns each attribute from the original object to the new one.
- The copy constructor is used to speed up the pass of parameters and returns of objects of a function.

3.2.13

The copy constructor

In C++, objects are copied when a parameter is passed by value and when an object is returned from a function. Copies can be handled using the copy constructor. The copy constructor initializes an object from an existing one by making a copy of it. The copy-constructor can also be used as a common constructor to initialize an object from another. The problems raised in the previous example using parameters by value and the return by value in the Stack class can be solved by creating a copy constructor as follow:

```
class Stack {
protected:
   int elements;
   int maxElements;
   int *vector;
public:
   Stack(int t = 10) {
      vector = new int[t];
      elements = 0;
      maxElements = t;
   }
   Stack(const Stack& n); //Copy constructor declaration
   ~Stack(){ delete [] vector;} // Destructor
   Stack & operator=(const Stack &); // Assignment operator
   // ...
};
Stack::Stack(const Stack& n) {
   elements = n.elements;
   maxElements = n.maxElements;
```

```
   vector = new int [maxElements];
   for (int i = 0; i < elements; i ++) vector[i] =
n.vector[i];
}
```

3.2.14

Select the correct affirmations.

- The copy constructor must contain the same code that the assignment operator.
- The copy constructor must always be called explicitly.
- The copy constructor is used automatically when an object is passed by value, or an object is returned from a function.
- The copy constructor needs to be created when using pointers.

3.2.15 Adding a copy constructor to the Student class

We have developed a *Student* class that saves the name and the id of students. We keep the name as an array of chars. All was correct until we start *passing/returning Students* to functions. Please, resolve the problem by adding a copy constructor.

```
class Student {
    int id;
    char* name;
public:
    Student(int id, const char* name) {
        this->id = id;
        this->name = new char[strlen(name)+1];
        strcpy(this->name, name);
    }
    ~Student() {delete this->name;}
    const char* getName() {return this->name;}
    int getId() {return this->id;}
};
```

exercise.cpp

```
#include <|string.h>
class Student {
    int id;
    char* name;
public:
    Student(int id, const char* name) {
```

```
        this->id = id;
        this->name = new char[strlen(name)+1];
        strcpy(this->name, name);
    }
    // Write here the copy constructor
    ~Student() {delete this->name;}
    const char* getName() {return this->name;}
    int getId() {return this->id;}
};
```

Inheritance and Polymorphism

Chapter 4

4.1 Inheritance

🕮 4.1.1

The inheritance

The inheritance is the ability to define new classes from existing ones so that they present the same characteristics and new others. The inheritance allows you to create classes that are **specializations of another**. The new class is called a **subclass** or **derived class**, and the existing class is called the **base class**. Subclasses inherit the attributes and member functions of their base class. Also, they can add new attributes and member functions or redefine inherited functions.

📝 4.1.2

The inheritance is the ability to define a _____ from existing ones. The new class is called a _____, and the existing class is the _____ of the new one. The new class inherit the attributes and member functions of their _____, and also, they can add new attributes or member functions and redefine inherited member functions.

- old class
- father class
- new class
- base class
- base class
- subclass or derived class

🕮 4.1.3

Multiple inheritances

In certain situations, it can be interesting to build a class that heiress of several classes, giving the multiple inheritances.

Multiple inheritances make sense to build new classes that acquire the characteristics of several. However, it has the problem of collision of identifiers, which occurs when different attributes or functions have the same name in more than one of the base classes. When this happens, there must be a mechanism that solves that ambiguity, either by prohibiting such inheritances, forcing a name change on one of them or by establishing a selection rule between them.

4.1.4

Multiple inheritances have the problem of collision of identifiers, which occurs when different attributes or functions have the same name in more than one of the base classes.

- True
- False

4.1.5

The inheritance use

The inheritance is a mechanism that allows programmers to reuse and extend existing code. The inheritance improves the reliability of the programs and decreases their development time. Programmers use inheritance:

- When different classes belong to more general abstraction. For example, in the game of chess, pawns, bishops, kings, queens, knights, and rooks are pieces of the game. Therefore, we can create a base class that has the common aspects of the pieces of the game, and each type of piece class will derive from that base class.
- When we need to extend the functionality of a program, and we want to make minimal modifications to the existing code without breaking the sense of each class.

The concept of inheritance can be applied in two different ways:

- As a way of representing the natural hierarchy of classes according to an appropriate taxonomy.
- To reuse existing code, adapting it to new needs by creating a new specialized class derived from one that already exists.

4.1.6

Select the correct affirmations.

- The inheritance is a mechanism that allows programmers to reuse and extend existing code.
- The inheritance weakness the reliability of the programs and increase their development time.
- Programmers create base classes when they realize that different classes belong to more general abstraction.

- Programmers create derived classes when they realize that they need to extend the functionality of a program, and we want to make minimal modifications to the existing code without breaking the sense of each class.

📖 4.1.7

The inheritance in C++

In C++, inheritance allows creating a new class from others. C++ allows multiple inheritances. If one class inherits from another, the first acquires all the characteristics of the second. That is, the derived class has all the attributes and almost all member functions of the base class. Constructors and destructors are not inherited, although they are called automatically or manually.

The derived class can add new attributes, but not modify or remove existing ones, and it can also add member functions or replace inherited member functions.

📝 4.1.8

Select the correct affirmations.

- The C++ programming language does not allow multiple inheritance.
- Constructors and destructors are inherited and cannot be overridden.
- The derived class can add new attributes, but not modify or remove existing ones.
- The derived class can add new member functions or replace the inherited ones.

📖 4.1.9

Inheritance and access control (encapsulation)

In C++, it is possible to establish how inheritance affects the type of accessibility of inherited elements. It can be inherited in a public, protected, or private ways.

- By inheriting "**private**", the public and protected elements become private for the derived class, and this is the default option.
- If the inheriting is "**protected**", then public and protected elements become protected for the inheriting class.
- When using "**public**" inheritance, the public and protected elements become public and protected for the derived class.

The private part of the base class is inherited, but it is not possible to access it.

Format:

```
class Class_name: [public, proteced, private] Base_class1,
                  [public, proteced, private] Base_class2, ... {
   ...
};
```

Example:

```
class StackUnlimited: public Stack {
   // ...
};
```

4.1.10

Select the correct affirmation about C++ inheritance.

- The inheritance only can be public or private.
- The default option for Inheritance is public
- The private part of the base class is inherited, and derived class can access it.
- When using public inheritance, the public and protected elements become public and protected for the derived class.
- A definition of a derived class in C++ can be "public class StackUnlimited <|- private Stack { // ... };"

4.1.11

Inheritance, constructors and destructors

Constructors and destructors are not inherited but are called automatically. The constructors of the base classes are executed before the constructor body of the derived class. The destructors of the base classes are executed after the destructor of the derived class.

If you want to control the pass of parameters to the constructors of the base class, you must call them before the body of the constructor of the derived class. If a constructor of a base class is not invoked manually, its default constructor is called automatically. The format to invoke the constructor of the base class also allows passing parameters to the constructor of the attributes of the derived class. The format to invoke the other constructors is as follows:

```
class Class_name: [public, protected, private] Base_class1,
                  [public, protected, private] Base_class2,
... {
   Type1 attribute1;
   Type2 attribute2;
   Class_name(...): Base_class1 (...), Base_class2 (...),
attribute1 (...), attribute2 (...), ... {
       // ...
   }
   // ...
};
```

This format allows us to pass parameters to the base class constructors and attributes constructors, taking them from the received parameters.

```
class StackUnlimited: public Stack {
   string name;
public:
   StackUnlimited(string n): Stack(50), name(n) {}
   // ...
};
```

4.1.12

Select the correct affirmation.

- The constructors and destructors must be called manually.
- The constructors of the base classes are executed after the constructor body of the derived class.
- The destructors of the base classes are not executed.
- If a constructor of a base class is not invoked manually, its default constructor will not be called automatically.
- If you want to control the pass of parameters to the constructors of the base class, you must call them before the body of the constructor of the derived class.

4.1.13 Calling attribute constructors in class constructors

We have developed a class *Person*, and we need a class *Couple* with two persons. We have developed the class *Couple* partially. Please, add a constructor to the class *Couple* that accepts as parameters two strings representing the names of the persons. The class *Person* does not have a default constructor, then the attributes

"one" and "two" of the class *Couple* need to be initialized using the constructor that accepts a string.

Alternative: Taking into account that all classes have a default copy constructor and that we have an automatic conversion from string to *Person*, a constructor of the class *Couple* for two persons can also be used as a constructor for two strings.

```
class Person{
    string name;
public:
    Person(string name) {this->name = name;}
    void setName(string name) {this->name = name;}
    string getName() {return this->name;}
};
class Couple{ // Needs a constructor
    Person one;
    Person two;
public:
    Person getPersonOne() {return this->one;}
    Person getPersonTwo() {return this->two;}
};
```

exercise.cpp

```
#include "person.h"

class Couple{ // Needs a constructor
    Person one;
    Person two;
public:
    // Write here the constructor
    Person getPersonOne() {return this->one;}
    Person getPersonTwo() {return this->two;}
};

void exampleOfUse() {
    // Couple c("Jane Doe","John Doe");
}
```

📖 4.1.14

The redefinition of member functions

In the derived class, you can redefine the member functions inherited from the base class, which allows you to adapt the existing code to new needs.

Example:

```
class StackUnlimited: public Stack { // Inherited from the
Pila class
public:
   StackUnlimited(int t = 100): Stack(t) {};
   StackUnlimited (const StackUnlimited &);
   StackUnlimited & operator = (const StackUnlimited &);
   bool isFull () {return false;} // The isFull function is
redefined
   void push(int); // The push function is redefined
};
void StackUnlimited::push (int i) {
   if (elements == maxElements) {
      int * nv;
      nv = new int [maxElements * 2];
      for (int j = 0; j < maxElements; j ++) nv[j] =
vector[j];
      delete [] vector;
      vector = nv;
      maxElements *= 2;
   }
   vector [elements ++] = i;
}
```

4.1.15

You can redefine the member functions inherited from the base class, but you cannot add new member functions.

- False
- True

4.1.16 Creating a new class Worker from Person

We have developed a *Person* class, a Job class, and we need a *Worker* class, a worker is also a person. Please, **create a new class *Worker* that inherited from *Person*** and have a function *setJob* and a function *getJob* to manage the job that the worker does.

```
class Person{
    string name;
public:
```

```
    void setName(string name) {this->name = name;}
    string getName() {return this->name;}
};
class Job{
    string position;
    float salary;
public:
    void setPosition(string position) {this->position =
position;}
    string getPosition() {return this->position;}
    void setSalary(float salary) {this->salary = salary;}
    float getSalary() {return this->salary;}
};
```

exercise.cpp

```
class Person{
    string name;
public:
    void setName(string name) {this->name = name;}
    string getName() {return this->name;}
};

class Job{
    string position;
    float salary;
public:
    void setPosition(string position) {this->position =
position;}
    string getPosition() {return this->position;}
    void setSalary(float salary) {this->salary = salary;}
    float getSalary() {return this->salary;}
};

void exampleOfUse() {
    // Worker me;
    // me.setName("JohnDoe");
    Job pjob;
    pjob.setPosition("President");
    pjob.setSalary(100000);
    // me.setJob(pjob);
    Job otherJob;
    // string position = me.getJob().getPosition()
    // float salary = me.getJob().getSalary();
}
```

4.1.17

The hide of member functions

When a function is redefined or created in a derived class, all functions with the **same name** in the base class **are hidden** in the derived class, even if they do not match the type or number of parameters. The function of the derived class hides those of the base class, but they can be called using its full name.

For example, if the Stack class has two functions "push", one "push(int)" that add one value to the stack and the other "push(int, int)" that adds two values to the stack when the class StackUnlimited redefine the "push(int)" function hide all the "push" functions of the base class. The hidden functions still can be called using its **full name**: the name of the base class followed by "::" and its name, in this case, "Stack::push(2, 3)".

4.1.18

When a function is created in a derived class, all functions with the same name in the base class are hidden in the derived class.

- True
- False

4.1.19

The hidden functions inherited from the base class can not be used in the derived class.

- False
- True

4.1.20 Creating a new class Worker from Person with constructors

We have developed a *Person* class, a Job class, and we need a *Worker* class, a worker is also a person. Please, **create a new class *Worker* that inherited from *Person*** and have a function *setJob* and a function *getJob* to manage the job done by the worker. The class *Person* and the class *Job* has a constructor. Notice that you must call the constructor of the base class and attributes at the constructor of the derived class. You can initialize each attribute using "*atribute_name(init_value)*".

```
class Person{
```

```
    string name;
public:
    Person(string name) {this->name = name;}
    void setName(string name) {this->name = name;}
    string getName() {return this->name;}
};

class Job{
    string position;
    float salary;
public:
    Job(string position, float salary) {
        this->position = position;
        this->salary = salary;
    }
    void setPosition(string position) {this->position =
position;}
    string getPosition() {return this->position;}
    void setSalary(float salary) {this->salary = salary;}
    float getSalary() {return this->salary;}
};
```

exercise.cpp

```
class Person{
    string name;
public:
    Person(string name) {this->name = name;}
    void setName(string name) {this->name = name;}
    string getName() {return this->name;}
};
class Job{
    string position;
    float salary;
public:
    Job(string position, float salary) {
        this->position = position;
        this->salary = salary;
    }
    void setPosition(string position) {this->position =
position;}
    string getPosition() {return this->position;}
    void setSalary(float salary) {this->salary = salary;}
    float getSalary() {return this->salary;}
};
```

```
void exampleOfUse() {
    Job pjob("President",1000);
    // Worker me("JohnDoe", pjob);
    // string name = me.getName()
    // string position = me.getJob().getPosition()
    // float salary = me.getJob().getSalary();
}
```

⌨ 4.1.21 Creating a new class Worker from Person redefining a member function

We have developed a *Person* class, a Job class, and we need a *Worker* class, a worker is also a person. Please, **create a new class *Worker* that inherited from *Person*** and have a function *setJob* and a function *getJob* to manage the job done by the worker. The class *Person* and the class *Job* has an automatic conversion function "*operator string*" that represents the object as a string. Redefine this function at the *Worker* class to return the value returned by *Person* string concatenate with space and followed by the string representing the job. Remember that you can use a hidden function using the function preceded by its class name and "::".

```
class Person{
    string name;
public:
    void setName(string name) {this->name = name;}
    string getName() {return this->name;}
    operator string() {return "Name: " + this->name;}
};

class Job{
    string position;
    float salary;
public:
    void setPosition(string position) {this->position =
position;}
    string getPosition() {return this->position;}
    void setSalary(float salary) {this->salary = salary;}
    float getSalary() {return this->salary;}
    operator string() {
        return "Position: " + this->position + " Salary: " +
to_string(this->salary);
    }
};
```

exercise.cpp

```
class Person{
    string name;
public:
    void setName(string name) {this->name = name;}
    string getName() {return this->name;}
    operator string() {return "Name: " + this->name;}
};
class Job{
    string position;
    float salary;
public:
    void setPosition(string position) {this->position =
position;}
    string getPosition() {return this->position;}
    void setSalary(float salary) {this->salary = salary;}
    float getSalary() {return this->salary;}
    operator string() {
        return "Position: " + this->position + " Salary: " +
to_string(this->salary);
    }
};

void exampleOfUse() {
    // Worker me;
    // me.setName("John Doe");
    Job pjob;
    pjob.setPosition("President");
    pjob.setSalary(100000);
    // me.setJob(pjob);
    Job otherJob;
    // string position = me.getJob().getPosition()
    // float salary = me.getJob().getSalary();
}
```

4.2 Polymorphism

🕮 4.2.1

The polymorphism in C++

The inheritance facilitates the construction of new classes from others already defined. However, its introduction is more transcendent since it involves

polymorphism. This element is essential in object-oriented languages, enhancing the possibilities of reusability of the software developed.

The polymorphism represents the ability to use objects of different classes through the same interface. That is, by using a polymorphic variable is possible to use objects of different classes transparently. The call of a member function with dynamic binding, using a polymorphic variable, results in calling the corresponding function of the class to which the object belongs.

The dynamic binding is the mechanism responsible for determining, at runtime, what function to execute, according to the object involved.

4.2.2

Invoking a member function **without dynamic binding**, using a polymorphic variable, results in calling the corresponding function of the class to which the object belongs.

- False
- True

4.2.3

Polymorphic variables

In the case of C++, there are two types of polymorphic variables: pointers and references.

Pointers and references to a base class can point to and reference objects of derived classes.

Pointers and references can call functions of the objects they point to or reference. The derived classes can redefine member functions so that depending on the class of the object, one function or another will be executed. Notice that the object to the pointer points to, may vary during execution.

4.2.4

What are the polymorphic variables in C++?

Select one:

- Pointers and references
- Pointers
- References
- Parameters
- Local variables
- Hidden objects

4.2.5

The polymorphic variables allow accessing objects of derived classes transparently.

- True
- False

4.2.6 Using polymorphism

We have developed classes that represent 3D geometric shapes. We have the class Cube, Cylinder, and an abstract class *Shape3D* gathering the common of 3d shapes.

We have an array of pointers to *Shape3D*. Notice that this means that we have an array of pointers to objects of any class derived from *Shape3D*, here *Cube*, or *Cylinder*. We can not create objects of the class *Shape3D* due it is an abstract class. Please, write a function called "areaOfBases" that accepts this array and its size. The function returns the sum of the area of the base of all objects in the array.

```
#include <cmath>

class Shape3D {
protected:
    float heigth;
public:
    Shape3D(float heigth) {this->heigth = heigth;}
    float getHeigth() {return this->heigth;}
    virtual float getAreaOfBase() = 0; //Pure virtual function
    virtual float getPerimeterOfBase() {return heigth *
getAreaOfBase();}
    virtual float getVolume() {return heigth *
getAreaOfBase();}
};

class Cube: public Shape3D{
public:
```

```
    Cube(float side): Shape3D(side) {}
    float getAreaOfBase() { return getHeigth() *
getHeigth();}; //Define a function
    float getPerimeterOfBase() {return 4 * getHeigth();}
};

class Cylinder: public Shape3D{
protected:
    float radio;
public:
    Cylinder(float radio, float heigth): Shape3D(heigth) {
this->radio = radio;}
    float getRadio() {return this->radio;}
    float getAreaOfBase() { return M_PI *  radio * radio;};
//Define a function
    float getPerimeterOfBase() {return 2 * M_PI *  radio;}
};
```

exercise.cpp

```
#include "shapes3d.h"

// Write here the function areaOfBases

float exampleOfUse() {
    Shape3D* v[5];
    Cube cu1(3.3), cu2(7.7), cu3(1);
    Cylinder cy1(1, 5), cy2(5.7, 1);
    v[0] = & cu1;
    v[1] = & cy1;
    v[2] = & cu2;
    v[3] = & cy2;
    v[4] = & cu3;
    return areaOfBases(v,5);
}
```

🕮 4.2.7

Virtual member functions

In C++, for reasons of efficiency, there is a **fixed bind and dynamic bind**. The fixed binding resolves the function to call at compile-time. The dynamic binding gets the function to call at runtime. By default, the calls to functions or procedures are of fixed bind type, and the member function call is resolved at compile time. Notice

that a significant number of modern programming languages are interpreted and always use dynamic binding.

Establishing the dynamic binding to a member function is done by writing the modifier "**virtual**" at the start of its declaration. This statement will indicate that the dynamic binding will occur for the corresponding function of this class and the functions redefined in derived classes. When this function is invoked using a polymorphic variable, the function executed depends on the class of the current object.

4.2.8

Establishing the dynamic binding to a member function is done by writing the modifier "virtual" at the start of its declaration.

- True
- False

4.2.9

Calling a function overloaded or a virtual function

Calling a member function overloaded should not be confused with calling a member function with polymorphism. In a function overload, the decision of which function should be called is resolved at compile time according to the number and the type of the parameters. When using dynamic binding, knowing which function should be called, is a decision at runtime based on the class to which the current object belongs.

4.2.10

Select the correct affirmations about calling a member function.

- In function overload in C++, the decision of which function should be called is resolved at runtime according to the number and the type of the parameters.
- Using dynamic binding, knowing which function should be called is a decision at runtime based on the class to which the current object belongs.

📖 4.2.11

When defining a hierarchy of classes, normally, the most basic class has an interface as generic as possible and defines its functions as virtual, allowing greater flexibility of use.

The polymorphism can occur in a semi-hidden way when another function of the same class calls a polymorphic function.

```
class public Stack { // Base class
   // ...
public:
   virtual bool isFull(); // Called using dynamic binding
   virtual void push(int); // Called using dynamic binding
   // ...
};
class StackUnlimited: public Stack { // Inherited from the
Pila class
public:
   bool isFull () {return false;} // Redefine a virtual
function
   virtual void push(int); // Redefine a virtual function
   // virtual is optional if already used in the base class
};
void usingStack(Stack & st) { // st reference polymorphic
   st.push(1); // Call the push function of the class of st
(Stack or StackUnlimited)
   st.push(2);
   st.push(2);
}
void f() {
   Stack st1(20);
   StackUnlimited st2();
   usingStack(st1);
   usingStack(st2);
}
```

When a virtual function is redefined in a derived class it is optional to put the virtual modifier, the new function will be virtual anyway.

📝 4.2.12

Please, indicate the line number that uses a dynamic binding call in the following code.

```
class public Stack { // Base class
   // ...
public:
   virtual bool isFull(); // Called used dynamic binding
   virtual void push(int); // Called used dynamic binding
   // ...
};
class StackUnlimited: public Stack { // Inherited from the
Pila class
public:
   bool isFull () {return false;} // Redefine a virtual
function
   virtual void push(int); // Redefine a virtual function
   // virtual is optional if alredy used in the base class
};
void usingStack(Stack *st) { // st referece polymorphic
   StackUnlimited localStack();
1   localStack.push(1);
2   st->push(1);
}
void f() {
   Stack st1(20);
   StackUnlimited st2();
3   st2.push(4);
4   usingStack(&st1);
5   usingStack(&st2);
}
```

📖 4.2.13

Pure virtual functions. Abstract classes

When a base class represents an abstraction, sometimes, there is not enough information to implement some member functions. C++ supports this, allowing member functions declared but not defined. To indicate that a function will not be defined in this class, its declaration ends with "= 0". These member functions must be "virtual". These functions are known as "pure virtual functions". A class that has some "pure virtual function" not defined is an "abstract class". It is not possible to create objects of an "abstract class" but you can create polymorphic variables that allow managing objects of derived classes.

For example, we can declare a Figure3D class with some pure virtual functions as follows:

```
class Figure3D {
```

```
    double height;
public:
    static const double PI;
    Figure3D(double height) { setHeight(height);  }
    double getHeight() { return height; }
    void setHeight(double height ) { this->height = height; }
    virtual double getBaseArea() = 0; // Pure virtual function
    virtual double getBasePerimeter() = 0; // Pure virtual
function
    virtual double getVolume() { return getBaseArea() *
getHeight(); }
};
const double Figure3D::PI = 3.14159265358979323846;
```

We have two pure virtual functions. Notice that we can call pure virtual functions as in getVolume()

```
class Cube: public Figure3D {
public:
   Cube (double height): Figure3D(height){}
   double getBaseArea() { return getHeight() * getHeight();}
   double getBasePerimeter() { return 4 * getHeight();}
};
class Cylinder: public Figure3D {
    double radius;
public:
    Cylinder (double height, double radius): Figure3D(height)
{ this->radius = radius; }
    double getBaseArea() { return PI * radius * radius; }
    double getBasePerimeter() { return 2 * PI * radius; }
};
class Cone: public Figure3D {
    double radius;
public:
    Cone (double height, double radius): Figure3D(height) {
this->radius = radius; }
    double getBaseArea() { return PI * radius * radius; }
    double getBasePerimeter() { return 2 * PI * radius; }
    double getVolumen() { return Figure3D::getVolume() / 3; }
};
```

4.2.14

Select the correct affirmations.

- Pure virtual functions are member functions with the modifier "virtual" that are declared but not defined.
- To indicate that a function is a "pure virtual function" its declaration ends with "= 0".
- The class that has some "pure virtual function" is a "virtual class".
- It is possible to create objects of an "abstract class".
- You can create polymorphic variables of an abstract class. This allows managing objects of derived classes.

⌨ 4.2.15 Creating a class derived from an abstract class

Please, add a class *Cone* derived from *Shape3D*. The area of the base of a cone is *2*PI*r*. The volume of a cone is (*area * height / 3*). Notice that a cone is not a cylinder. The constructor of the *Cone* must accept the radius and the height (in this order).

```
#include <cmath>

class Shape3D {
protected:
    float heigth;
public:
    Shape3D(float heigth) {this->heigth = heigth;}
    float getHeigth() {return this->heigth;}
    virtual float getAreaOfBase() = 0; //Pure virtual function
    virtual float getPerimeterOfBase() {return heigth *
getAreaOfBase();}
    virtual float getVolume() {return heigth *
getAreaOfBase();}
};
```

⌨ 4.2.16 Creating an abstract class

We want to develop classes that represent pieces of chess. We have started developing a class *Knight* and a class *Rook*. We have found common features. Please, create a new class called *Piece* that gathers the common features of all chess pieces. The class *Piece* must have all its functions virtual and virtual pure when needed.

```
class Knight {
    int posx, posy;
    int color;
public: Knight(
```

```
    int color) {this->color = color; this->x = 0; this->y =
0;} string getName() {return "Knight";}
    void setPosition(int x, int y) {this->x = x; this->y = y;}
    bool isOnboard() { return x > 0 && x <9 && y > 0 && y <
9;}
    bool canDoMove(int x, int y) {...}
};

class Rook {
    int posx, posy;
    int color;
public: Rook(
    int color) {this->color = color; this->x = 0; this->y =
0;} string getName() {return "Knight";}
    void setPosition(int x, int y) {this->x = x; this->y = y;}
    bool isOnboard() { return x > 0 && x <9 && y > 0 && y <
9;}
    bool canDoMove(int x, int y) {...}
};
```

exercise.cpp

```
// Write here the abstract class Piece

/* Do not uncomment
class Knight {
    int posx, posy;
    int color;
public:
    Knight(int color) {this->color = color;this->x = 0;this->y
= 0;}
    string getName() {return "Knight";}
    void setPosition(int x, int y) {this->x = x;this->y = y;}
    bool isOnboard() { return x > 0 && x <|9 && y > 0 && y <|
9;}
    bool canDoMove(int x, int y) {...}
};

class Rook {
    int posx, posy;
    int color;
public:
    Rook(int color) {this->color = color;this->x = 0;this->y =
0;}
    string getName() {return "Knight";}
    void setPosition(int x, int y) {this->x = x;this->y = y;}
```

```
    bool isOnboard() { return x > 0 && x <|9 && y > 0 && y <|
9;}
    bool canDoMove(int x, int y) {...}
};
*/
```

Templates

Chapter 5

5.1 Templates

📖 5.1.1

Genericity and parametrization

When developing new functions and classes, you must specify the types of data involved. In many cases, only by changing the types of data, we would obtain a fully operational new class or function. Therefore, there are higher-level abstractions that are independent of the type of data involved.

For example, the process of ordering an array is the same, regardless of the type of data. It is sufficient that the type of data stored in the array, supports the operations required during the sorting process, such as comparison, assignment, etc.

Parameterization is the mechanism that languages have for building generic classes and functions; allowing to write the same code to describe the different implementations of a class or function in which only some of the types of data that intervene vary.

📝 5.1.2

Parameterization is the mechanism for building generic classes and functions that are ready to be used varying the types of data used.

- True
- False

📖 5.1.3

Parametric functions

The parameterization of functions allows defining an unlimited set of variant functions for the type of data they process. For example, a single parameterized function *sort()* can define a function that orders arrays of any data type.

The format to parameterize a function is:

```
template <class Type1, class Type2, ...>
Function Definition
```

Where *Type1*, *Type2*, etc. are the parametric names of unspecified data types. All these types must appear in the function parameters.

Example 1:

```
#include <iostream>
#include <cstdlib>
using namespace std;
template <class T> void swap(T & e1, T & e2) {
   T e;
   e = e2;
   e2 = e1;
   e1 = e;
}
template <class T> void sort (T v[], int n) {
   for (int i = 0; i < n; i ++)
      for (int j = 0; j <(n-i-1); j ++)
        if (v[j] > v[j + 1])
           swap(v[j], v[j + 1]);
}
int main() {
   int v[20];
   float g[20];
   for (int i = 0; i <20; i ++) {
      v[i] = rand();
      g[i] = v[i] / 3.0 * rand();
   }
   sort(v, 20);
   sort(g, 20);
   for (int i = 0; i < 20; i++)
      cout << i << ": " << v[i] << " " << g[i] << endl;
   return 0;
}
```

5.1.4

Select the valid codes using the *swap* function.

```
#include <iostream>
#include <cstdlib>
using namespace std;
template <class T> void swap(T & e1, T & e2) {
   T e;
   e = e2;
   e2 = e1;
   e1 = e;
}
```

Select one or more:

Code A

```
// ...
string a = "Hello!";
int b = 3;
swap(a, b);
```

Code B

```
// ...
string a = "Hello!";
string b = "bye";
swap(a, b);
```

Code C

```
// ...
char *a = "Hello!";
char *b = "bye";
swap(a, b);
```

Code D

```
struct S {
   int d1;
   int d2;
};
// ...
S a, b;
// ...
swap(a, b);
```

Code E

```
struct S {
   int d1;
   int d2;
   S(int p) {d1 = p; d2 = p}
};
// ...
S a(2), b(5);
// ...
swap(a, b);
```

- Code A
- Code B
- Code C
- Code D
- Code E

⌨ 5.1.5 Creating a template function

Please, define a function named "arraySum" that accepts an array of any type and its size. The function returns the sum of the elements in the array. The type of this value is the same that the elements in the array.

🕮 5.1.6

The definition of a template function must appear in the ".h" header files, not in the ".cpp" files.

Example 2:

```
template <class T> T max (T a, T b) {
  return a> b? a: b;
}
```

Parameterized functions do not generate code until they are explicitly or implicitly defined with their use.

Example of explicit definition:

```
void fn () {
   int max(int, int);
   int a = 1, b = 2;
   a = max(a, b);
}
```

Implicit definition example:

```
void fn () {
   int a = 1, b = 2;
   char c = 'Z', d = 'A';
   a = max(a, b); // The function max(int, int) is generated
   c = max(c, d); // The function max(char, char) is generated
}
```

5.1.7 Creating other template function

Please, define a void template function named "sort3" that accepts three references to variables of a parametric type. The function reorders the three values, from left to right.

5.1.8

To allow the use of template functions in other source files, the definition must go in a ".cpp" file and the declaration in a header file.

- False
- True

5.1.9

Generic classes

A **template** or generic class describes a family of classes that varies in some types of data not specified at the time of its definition.

The format to parameterize a class is:

```
template <class Type1, class Type2, Type3 param, ...> class
ClassName
Class definition
```

where Type1, Type2, etc. are the parametric names of unspecified data types. It can also set values as **param of Type3,** which will be used as a constant value in the class.

The declaration and implementation of a parametric class must be placed in a ".h" header file.

Parametric classes are always explicitly defined, and their name is the class name followed by the parametric types separated by "," between "<" and ">". Format of use of a parametric class:

```
ClassName <Type1, Type2, value of Type3, ...>
```

When used to reference the class, within the class itself, the names of the formal parameters are used; when used to define a specific class, the formal parameters are replaced by real parameters, that is, by real types.

The following example defines a **Vector template** class.

```
template <class T> class Vector {
private:
   T * v;
public:
   Vector(int length) {v = new T[length];}
   ~Vector() { delete []v;}
   T & operator[] (int index);
};
template <class T>
T & Vector<T>::operator[] (int index) {
    return v[index];
}
```

Notice that the definition of functions outside the class need "template<Parameter>" before definition and the name of the class must contain "<Parameter>".

This class can be used as follows:

```
int main () {
   Vector<int> ints(20);
   Vector<Complex> complexes(10);
   for (int i = 0; i < 20; i++) ints[i] = i;
   for (int i = 0; i < 10; i++) complexes[i] = Complex(i, i *
2);
   return 0;
}
```

The first statement creates a **Vector<int> class,** and then an object of this class is generated. The second statement defines the **Vector<Complex> class** and an object of this class. For the Vector class to be fully functional, the assignment operator and the copy constructor would have to be added.

5.1.10

Select the correct affirmations.

Affirmation A

The declaration and implementation of a parametric class must be placed in a ".h" header file.

Affirmation B

```
template <class T> class Vector {
private:
   T * v;
public:
   Vector(int length) {v = new T[length];}
   ~Vector() { delete []v;}
   T & operator[] (int index) { return v[index]; }
};
```

To use T in Vector, the type T must allow the creation of arrays of Ts, then if T is a class must have a default constructor.

Affirmation C

```
template <class T> class Vector {
private:
   T * v;
public:
   Vector(int length) {v = new T[length];}
   ~Vector() { delete []v;}
   T & operator[] (int index) {
       return v[index];
   }
};
```

The assignment of two objects of type Vector will work correctly.

Affirmation D

```
template <class T1, class T2> class Pair {
public:
   T1 first;
   T2 second;
};
Pair<int, int> a;
```

The object "a" of type "Pair<int, int>" can not be created because T1 and T2 must be different types.

Affirmation E

```
template <class T, int length> class Vector {
private:
   T v[length];
```

```
public:
   T & operator[] (int index) {
       if (index < 0 || index >= length) throw "Index out of
bound";
       return v[index];
   }
};
// ...
int main() {
    Vector<int, 10> v;
    v[3] = 4;
    return 0;
}
```

It Will not compile because the class Vector requires a constructor and a destructor.

- Affirmation A
- Affirmation B
- Affirmation C
- Affirmation D
- Affirmation E

⌨ 5.1.11 Creating a template class

We have developed a Stack class of integers of limited size. Please, change the class to be a template class that can keep data of any type. **Remember** that the definition of functions outside the class need "template<Parameter>" before the definition and the name of the class must contain "<Parameter>".

```
class Stack {
protected:
   int nelements;
   int maxElements;
   int vector;
public:
   Stack(int t = 100); // Constructor with one parameter and
default constructor
   ~Stack() {delete vector;} // Destructor to free dynamic
memory
   bool isEmpty() {return nelements == 0;}
   bool isFull() {return nelements == maxElements;}
   void push(int);
   void pop();
```

```
   int top();
};

Stack::Stack (int t) {
   vector = new int[t]; // The vector is created by taking
dynamic memory
   nelements = 0;
   maxElements = t;
}
void Stack::push(int i) {vector [nelements ++] = i;}
void Stack::pop() {--nelements;}
int Stack::top() {return vector [nelements-1];}
```

exercise.cpp

```
class Stack {
protected:
   int size;
   int maxsize;
   int *vector;
public:
   Stack(int maxsize = 100);
   ~Stack() {delete []vector;}
   bool isEmpty() {return size == 0;}
   bool isFull() {return size == maxsize;}
   void push(int);
   void pop();
   int top();
};
Stack::Stack (int maxsize) {
   this->vector = new int[maxsize];
   this->size = 0;
   this->maxsize = maxsize;
}
void Stack::push(int data) {vector [size ++] = data;}
void Stack::pop() {--size;}
int Stack::top() {return vector [size-1];}
```

Exceptions

Chapter 6

6.1 Exceptions

📖 6.1.1

Exceptions

An exception is an unusual event that affects the execution of a program in an uncontrolled manner. There are two types, exceptions: hardware and software:

- Hardware exceptions occur when an operation produces an abnormal condition detected by the operating system. For example, division by zero, access to a prohibited memory area, etc.
- The software exceptions occur under the decision of the programmer, for example:
- There is an unexpected result detected by the software. For example, unexpected values when reading a file.
- You use the wrong parameters calling a function. For example, you try to calculate the square root of a negative number.
- An internal malfunction of some function is detected.

Notice that if the program resolves an unusual situation executing code where it's found, then it is not an exception.

Historically, the detection of an exception generated an abnormal program termination. Instead, the application should go to an execution point where the program can resolve the problem or the user informed. This process of returning to a safe location in the code is complex to implement if it is not supplied directly by the programming language. Today, most programming languages provide mechanisms to perform this task.

📝 6.1.2

Please, select the situations where you must use exceptions.

Select one or more:

- A function that takes an array to return the minimum value and receives an array of size zero.
- A function that takes an array to return the sum of its values and receives an array of size zero.
- A program prints a billing ticket, and the printer fails.
- A function that takes an array to return the sum of its values, but receives a NULL pointer.
- A function that searches for the number of words in a text file found that is going to return zero.

📖 6.1.3

Exceptions in C++

The C++ exception handling system provides a secure way to return from where the exception occurs to a safe point in the code. It also establishes exception managers, that is, code that runs after an exception occurs. After an exception, objects created in the stack, from the place where the exception was thrown to the safe return point, are automatically destroyed. The return point is an exception handler that is executed.

The exception mechanism needs to create a **try-block** from which execution **can throw an exception**. After the try-block, the exception handlers are defined. Each exception is of a type of data, commonly a class, and each handler manages a type of exception.

```
try {  // try-block
   // Exceptions can occur inside any function called
}
```

You must use the **throw** instruction to launch an exception. The **throw** instruction has the format:

```
throw expression;
```

The data/object resulting from evaluating the expression will be the data representing the exception. This data is received as a parameter by the exception handler.

Exception handlers **catch** are defined after the **try** block with the following format:

```
try { /* ... */ }
catch (type1 parameter) { // type1 is the type of the
exception
   exception management code
}
catch (type2 parameter) { // type2 is the type of the
exception
   exception management code
}
// Execution flow continue here after the try-block or after
executing a handler (catch)
```

The handler is like a function that receives information from the exception. The specified type of that parameter determines the type of exception that can accept the handler.

6.1.4

Please, select the correct code where exceptions have sense.

Select one or more:

Code A

```
// ...
try {
    int a = b;
}
catch (Exception e) {
   // ...
}
```

Code B

```
int f(int b) {
    if ( b < 0 ) throw Exception();
    // ...
}
// ...
try {
    int a = f(b);
} catch (Exception e) {
   // ...
}
```

Code C

```
int f(int b) {
    if ( b < 0 ) throw 3;
    // ...
}
// ...
try {
    int a = f(b);
} catch (int e) {
   // ...
}
```

Code D

```
// ...
try {
    if ( b < 0 ) throw 3;
    b = 2 * b;
} catch (int e) {
   b = b + 2;
}
```

Code E

```
int f(int b) {
    if ( b < 0 ) throw Exception();
    // ...
}
int g(int b) {
    return f(b * 2);
}
// ...
try {
    int a = g(b);
}
catch (int e) {
   // ...
} catch (Exception e) {
   // ...
}
```

- Code A
- Code B
- Code C
- Code D
- Code E

6.1.5 Adding exceptions to a class

We have developed a *Stack* class of integers of limited size. Please, change the code to detect the wrong use of the class. If you found bad use, you must throw an exception. You can use the type of exception that you want. **Hint**: four locations need to check the parameters or the state of the stack to work correctly.

```
class Stack {
protected:
   int nelements;
   int maxElements;
```

```
   int vector;
public:
   Stack(int t = 100); // Constructor with one parameter and
default constructor
   ~Stack() {delete [] vector;} // Destructor to free dynamic
memory
   bool isEmpty() {return nelements == 0;}
   bool isFull() {return nelements == maxElements;}
   void push(int);
   void pop();
   int top();
};

Stack::Stack (int t) {
   vector = new int[t]; // The vector is created by taking
dynamic memory
   nelements = 0;
   maxElements = t;
}
void Stack::push(int i) {vector [nelements ++] = i;}
void Stack::pop() {--nelements;}
int Stack::top() {return vector [nelements-1];}
```

exercise.cpp

```
class Stack {
protected:
   int size;
   int maxsize;
   int *vector;
public:
   Stack(int maxsize = 100);
   ~Stack() {delete []vector;}
   bool isEmpty() {return size == 0;}
   bool isFull() {return size == maxsize;}
   void push(int);
   void pop();
   int top();
};
Stack::Stack (int maxsize) {
   this->vector = new int[maxsize];
   this->size = 0;
   this->maxsize = maxsize;
}
void Stack::push(int data) {vector [size ++] = data;}
void Stack::pop() {--size;}
```

```
int Stack::top() {return vector [size-1];}
```

📖 6.1.6

The management of exceptions

For each try-block, you can have as many handlers as you wish. After an exception, one handler is executed, if any match the exception.

It is possible that during the execution of a try-block, you enter another try-blocks. When an exception is thrown, the last try-block is explored. If the type of exception does not match any handler, the exception goes on with the previous try-block. If there are no more try-blocks, the program aborts with a message.

After executing the handler, the execution flow goes on after the last handle of the current try-block. The execution flow does not go on at the point where the exception was thrown.

📝 6.1.7

Please, select the correct affirmations.

- Each try-block can have only one exception handler.
- When an exception is thrown, and a matching exception handler is not found, the execution flow returns to the exception point.
- It is possible that during the execution of a try-block, you enter another try-blocks.
- When an exception is thrown, the last try-block is explored. If the type of the exception does not match any handler, the exception goes on with the previous try-block.
- After executing the exception handler, the execution flow goes on after the last handle of the current try-block.

⌨ 6.1.8 Using try-catch

We have developed a template *Array* class that allows creating arrays of any type and size. We control the bad use of the class, throwing two types of exceptions *IndexOutOfBound* and *ArrayOfNegativeSize*.

We have written a function that sorts an object of the *Array* class, but we are not sure of its correct functionality. Please, modify the function *testSort* to print a message if the sort function throws *IndexOutOfBound* or *ArrayOfNegative*. Please, print the name of the exception.

```
class ArrayOfNegativeSize {};
class IndexOutOfBound {};

template <class T> class Array {
   T * v;
   int length;
public:
   Array (int length) {
       if(length < 0 ) throw ArrayOfNegativeSize();
       this->length = length;
       v = new T[length];
   }
   ~Array () { delete []v;}
   int getLength() { return length; }
   T & operator[] (int index) {
       if (index < 0 || index >= length) throw
IndexOutOfBound();
       return v[index];
   }
};
```

exercise.cpp

```
#include "array.h"
#include "sort.h"
template<|class T>
void testSort(Array<|T>& a) {
    // Control here with try-catch the exceptions that
Array<|> can throw.
    // Use cout <|<| name <|<| endl; to print the exception
class name.
    sort(a);
}
```

PRISCILLA

priscilla.fitped.eu