Basic C++ Programming- Exercises

Exercise 1

Write a class *Vector* with :

- private data members : three *double* components,
- public member functions :

* *initialize*	to assign values to the components,
* *multiply*	to multiply the components by an integer value passed by argument,
* *display*	to display the components' values.

Exercise 2

Write a class *Vector* similar to the previous one, in which the initialize function will be replaced by a constructor.

Exercise 3

Add to the previous class two constructors:

* the first one without any argument, initializing each component to 0,

* the second one with one argument, initializing each component to the same value.

Exercise 4

Add a function to multiply two vectors. The result will be a double, sum of the products of the corresponding components.

Exercise 5

Add a function to sum two vectors. The result will be a vector, which components are the sum of the corresponding components.

Exercise 6

Modify the constructors in order to count the number of vectors. Add a function that indicates the number of existing vectors. Decrement the number of created vectors when destroying one of them. Check first in the main.

Modify the *sum* function to pass the argument reference. Check again in the main.

Exercise 7

Write a class *int_stack* that will manage integers, stored in a dynamically allocated array.

This class will propose the following member functions:

int_stack (int n)	constructor that will dynamically allocate n integers,
int_stack ( )	constructor allocating 20 integers,
~ int_stack ( )	destructor,
void push (int p)	pushes the p value on the stack,
int pop ( )	returns (and remove) the value on the top of the stack,
int full ( )	the return value is 1 if the stack is full, 0 otherwise,
int empty ( )	the return value is 1 if the stack is empty, 0 otherwise.

Write the main function that uses the previous class. Declare automatic (in a local block) or dynamic objects and initialize them with existing static objects, or call a function (empty body) that expects a stack as argument. (example : *f1(int_stack){ }* and *f2(){ int_stack s; }*)

Add to the previous class a copy constructor that corrects the problems.

Exercise 8

Write a class *pair_vect* that contains two three-dimensional vectors (exercise 1):

- write 2 constructors : one with 6 *double* arguments, a second with 2 vectors arguments.
- add functions to get the first or second vector, and carry on modifying these vectors with the first *multiply* function (expecting an integer as argument).

Exercise 9

In the usual class *point*:

```
class point
{    int x, y ;
     public :
             point (int = 0, int = 0);
} ;
```

add the == operator, that compares 2 points; the return value will be 1 if the points have the same coordinates, 0 otherwise.

Exercise 10

In previous int_stack class (exercise 7), add the > and < operators :

if p is an *int_stack* and n an integer :

p < n   add the n value to the stack (no value returned)

p > n   suppress the stack's top value and returns it. The result is stored in variable n.

Exercise 11

Define a class *point* which components are integers, and define on this class the way to obtain a point sum of two others. Add the way to display a *point*.

Define a class *complex* which real and imaginary parts are integers.

Define the way to convert a *complex* into a *point*.

Use this operator to sum two *complex*es, and to display a *complex*.

Exercise 12

Write a square integer class *matrix* (unknown dimension). Add a function to multiply a *vector* and a *matrix*.

Exercise 13

In previous int_stack class (exercise 10), define and handle *FullStackException* and *EmptyStackException*.

Exercise 14

In C language, there is no real string type, but a conventional representation of strings (array of characters ending by the null ASCII code character). The different functions uses this conventional notation to manipulate strings (copy, concatenate, …) .

In this exercise, you will write a *mystring* class that could be compared to the classical type (in sense of what you can find in Basic or Pascal language)

Thus, you will propose the following members:

- the actual length of the string,
- the address of a dynamically allocated zone, that will contain the characters (without the null character at the end, because the length is stored).

The content of the string will dynamically change.

Propose the following constructors:

- *mystring()*	initialize with an empty string,
- *mystring (char *)*	initialize a new string with a string as it is known in C and which address is transmitted as an argument,
- *mystring (mystring &)*	copy constructor.

Define the following operators:

=	to assign strings (think about cascaded assignment),
==	to compare two strings,
+	to concatenate two strings. If a and b are of type *mystring*, a + b will be a new *mystring* (a and b will not change)
[ ]	to access a character in a *mystring* (think about assignment)

Add a *display* function.

Exercise 15

Add to the previous class *mystring* the way to convert a string to a pointer to a character, and a conversion from a pointer to a character to a string.

Exercise 16

On previous class *mystring*, define an exception when out of bounds values to [ ] operator.

Exercise 17

Write a class *geometric_figure* that proposes a function that computes its perimeter.

Write the class *circle*.

Write the class *polygon*, where will be stored the lengths of the sides. When declaring a polygon, the only information passed to the constructor is the number of sides. Add the function that calculates the perimeter.

Write the classes *quadrilateral*, *rectangle*, *square* and *triangle*.

Declare objects of the different classes using the following constructors :

```
        quadrilateral q ( 1, 2 , 3, 4 );        rectangle r (3 ,4 );   square s (4 );
```

where the parameters are the different sides' lengths.

Override function *perimeter* in class *square*.

Exercise 18

Write a *list* class that will handle linked objects which specific details are unknown.

On the class, the following operations are proposed :

- add an information in the list (at the front),
- initialize the navigation through the list,
- go to the next information,
- get the current information,
- detect if the end of the list has been reached.

Exercise 19

Use the previous class to create a *list_of_points*. Add a function that prints the information of each point stored in the list.

Exercise 20

Write a class *mixed_list* obtained from the previous list that will store information of different types, but that can be displayed using a function *display*. Use it to store points, complexes and vectors and display the information of all objects stored in that list.

Exercise 21

On the previous class *vector*, add the way to store dynamically allocated vectors in a linked list.