# Application

**List - Stack - Queue**

**Stacks** are based on the LIFO (last in- first out) principle, i.e., the element inserted at the last, is the first element to come out of the **list**.

**Queues** are based on the FIFO (first in- first out) principle, i.e., the element inserted at the first, is the first element to come out of the **list**.

Using inheritance, templates and exceptions, the team will develop base a **class template** MyList that stores an array of generic-typed elements **elems** with a capacity **itsSize.**

## The class MyList is an abstract class which has the following:

* Parameterized constructor that takes the array size.
* A destructor
* getSize which returns the capacity of the array
* The pure virtual functions:
  + addElem which adds an element
  + getElem which returns an element, then remove it from the array.
  + isEmpty which returns true if the array is empty
  + isFull()which returns true if the array is full
  + clearItems()=0;

The classes MyStack and MyQueue will inherit from MyList, and implement the pure virtual functions according to the definition of each above.

In MyStack the last element added will be the element returned if we call the getElem method.

In MyQueue the first element added will be the element returned if we call the getElem method.

In main the user will be able to store **integers**, **strings** or **Rectangle objects** in our stack/queue based on the user’s choice. The class Rectangle has the following specification:

**class** Rectangle

{

int length; int width;

## public:

Rectangle(); Rectangle(int,int); int getLength(); int getWidth(); int getArea();

friend ostream& operator<<(ostream&,const Rectangle&);

//the overloaded function prints length, width & area of the rectangle

};

Before adding an element check that the stack/queue is **not full**, and before getting an element check that the stack/queue is **not empty**.

In main: **First,** ask the user whether to use stacks or queues (**S/Q**),

**Second,** ask about the number of elements to be added,

**Then,** ask whether the user wants to store integers, strings or rectangles (**1/2/3**),

**Finally,** loop for number of elements to take in the data to store, then display all the data. In case rectangle object are stored, length, width and area are displayed. The stack should show the elements in reverse order, while a queue shows the elements the same order they were entered.

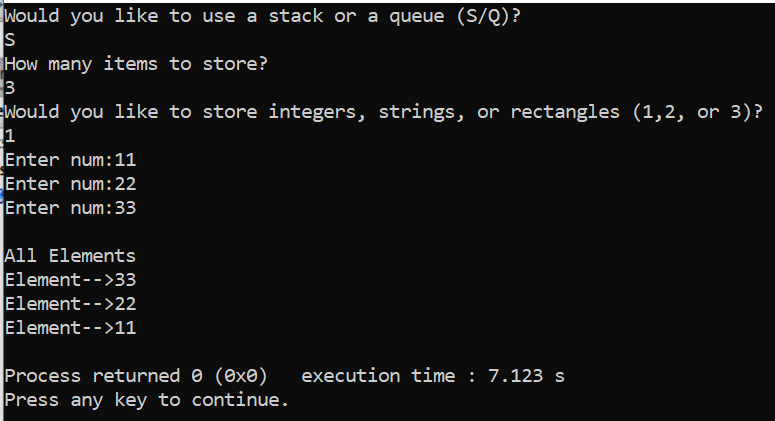
**Exceptions:**

Handle exceptions that might occur because of a wrong input from the user, **such as:**

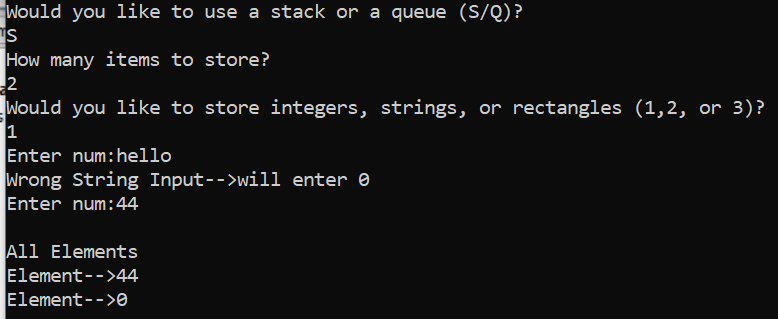
User entered negative or zero number of elements, User chose to store integers but entered a string value,

User chose to store Rectangles objects and entered a negative length or width (use exception classes defined in Rectangle class for this kind of exception)

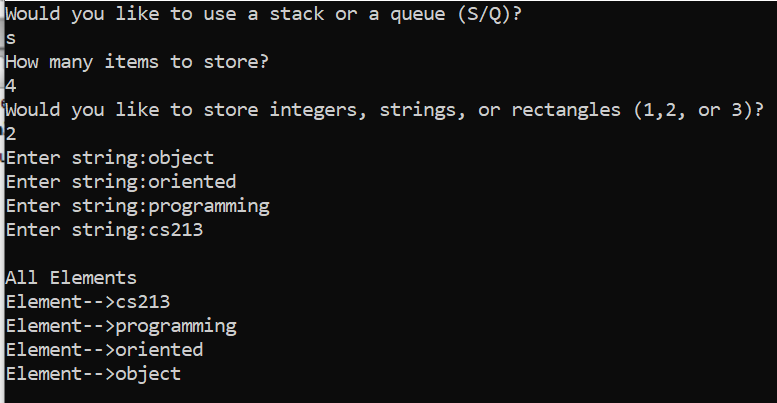
# User Scenario 1:



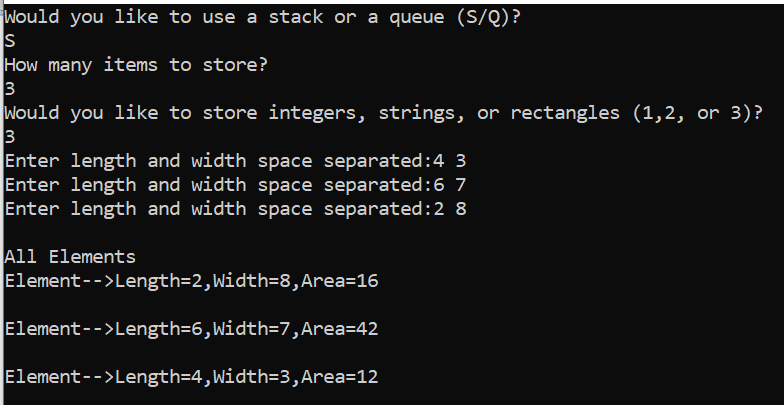
**User Scenario 2:**



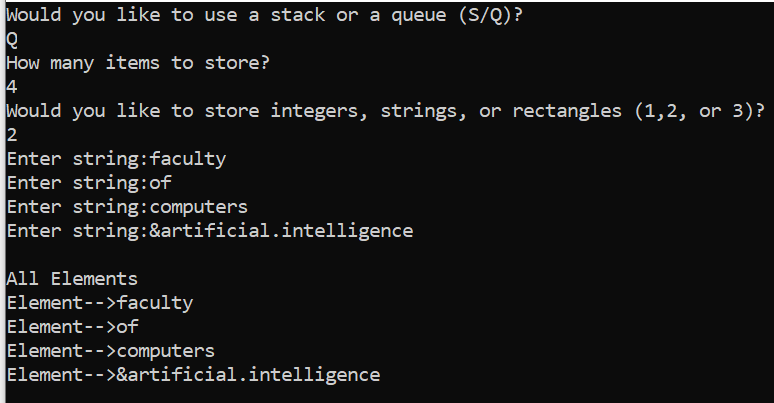
# User Scenario 3:



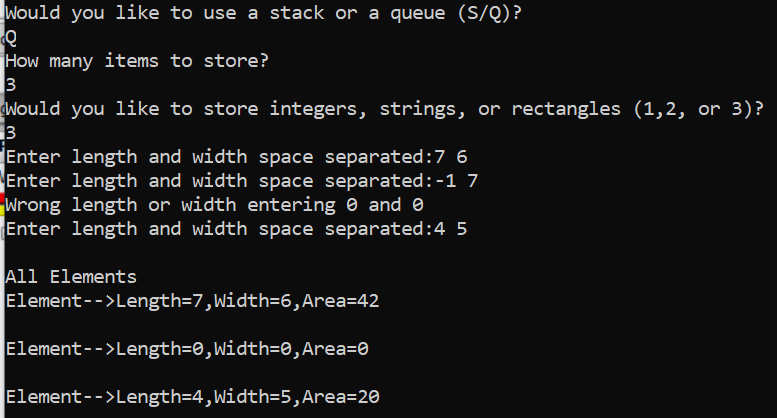
**User Scenario 4:**



# User Scenario 5:



**User Scenario 6:**



# Bonus(4 marks to be scaled to 1 mark):

Make your stack & queue resizeable such that no capacity is taken as input from the user and the user is asked to add 1 element at a time after being asked if (s)he wishes to enter another element. Modify **addElem** of the stack & queue such that it actually extends the memory allocated to the array by 1 element every time it is called.

Note that you will not need to call isFull before adding an element to the stack because the stack/queue is never full and you can always add more.

# Sample Scenario for the Bonus:

