CS515 Assignment 4

The purpose of this assignment is to reinforce your understanding of C++ classes and objects. You will implement the stack ADT and the queue ADT. Download all files from `~cs515/public/4P`. *Note that the header files you download may be different from the files you had from 3L.*

Part 1: Implement an expandable stack

Modify the stack implementation from Lab 3 and make it expandable. An expandable stack can grow in various ways. In this assignment, the stack grows by doubling its current size. For example, given a stack with current capacity of 8, if a new element is to be pushed on the stack and the stack already contains 8 elements, the capacity of the stack will automatically increase to 16 so it can accommodate the new element. The capacity of a stack only grows—it doesn't shrink when elements are popped.
To implement such a stack, you will need to use a dynamic array to hold all stack elements. Then, when the capacity is doubled, a new, larger dynamic array is created and is used to replace the current array. Make sure you implement the first three of the Big-5 methods for the stack class to avoid memory errors (you don't need to make move operations). You should throw exceptions for any unwanted user behavior as in Lab 3. However, we assume all dynamic memory allocations are successful so the expandable stack will not throw stack-full exceptions. Write your expandable-stack implementation in stack.cpp, **without modifying the header file stack.h.**

You are not allowed to use any STL containers in your implementation.

Some sample test code is provided. You may use separate compilations to compile the test cases
First, to compile the stack class use the **–c** option.

```
% g++ -c stack.cpp
```

Then, compile the stack class with the test driver and name the executable `stackSampleTest1`, use

```
% g++ stackSampleTest1.cpp stack.cpp -o stackSampleTest1
```

To run the test:

```
% ./stackSampleTest1
```

Part 2: Implement an expandable queue

Similar to the expandable stack, you are asked to build an expandable queue based on the queue you worked on in Lab 3. The expandable queue is still based on a circular array implementation, but now it can grow when the number of queue elements exceeds the capacity of the queue.

To implement such a queue, you will need to use a dynamic array to hold all queue elements. Any time the capacity is doubled, a new, larger dynamic array is created and is used to replace the current array. Make sure you implement the three non-move methods of the Big-5 for the queue class to avoid memory errors. You should throw exceptions for any invalid access of your queue. Similar to the expandable stack, we assume all dynamic memory allocations are successful for the expandable queue, so you don't need to worry about queue-full exceptions.

You need to make sure the enqueue() and dequeue() operation is O(1) on average—the queue only expands when its capacity reaches the limit.

Write your expandable-queue implementation in `queue.cpp`, **without modifying the header file `queue.h`.**

You are not allowed to use any STL containers in your implementation.

Some sample test code is provided. The compilation rules are also included in the provided makefile, so that you can invoke

```
% make tests
```

to compile all the sample tests for both stack and queue implementations.

Submission:

- Submit two source files **`stack.cpp`** and **`queue.cpp`,** but not your **`makefile`**.
- You must also fill out the README file and submit it as well. Submit the following files to the appropriate assignment on Mimir:

  `stack.cpp queue.cpp README`

- Do not turn in executables or object code. Programs that produce compile time errors or warnings will receive a zero mark (even if it might work perfectly on your home computer).
- Be sure to provide comments in your program. You must include the information as the section of comments below:

```
/**    CS515 Assignment X
File: XXX.cpp
Name: XXX
Section: X
Date: XXX
Collaboration Declaration: assistance received from TA, PAC etc.
*/
```

Some notes on grading:

- Programs are graded for correctness (output results and code details), following directions and using specified features, documentation and style.
- Here is a tentative grading scheme.

stacktest1	5
stacktest2	5
stacktest3	10
stacktest4	10
stacktest5	10
queuetest1	5
queuetest2	5
queuetest3	10
queuetest4	10
queuetest5	10
Valgrind check	20
Using STL container	-100
O(n) en-q/de-q, push/pop	-20