Ryan Winterhalter, M12727389

Thomas Rafeld, M12808058

Lab 3 Report

**Objectives:**

For lab 3 we focused on two learning outcomes: understanding stacks and queues and using advanced array structures. The first outcome was the main focus of this lab, as for each task we had to utilize a stack, a queue, or both. Working through the tasks, we created a stack and queue to be able to test a set of inputs to be added and removed.  Then, using those stacks and queues we already made, we completed an actual objective of sending and receiving messages in big endian to the queue then through little endian to a stack allowed us to gain a better understanding of how the functions of stacks and queues work. This outcome is important to the course because it gives options to store data in different ways and gives us more knowledge of how we can use classes along with being an introduction to some easier data structures. These are important to a career as queues or stacks could be utilized in code by someone else and us having a base understanding beforehand would allow us to be able to read and understand what is happening and how so any errors could be fixed if needed. The second outcome using advanced array structures was used in task three to move between stacks and queues and utilized by the stack and queue classes. This is important to class as arrays are a type of data structure and will probably be used many times in the future of this course as they are an important part of coding. The same is true of a career in engineering that arrays are used all the time for important projects so being able to use arrays and advanced array structures will allow us to be successful in said career.

**Task 1:**

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer screen

Description automatically generated

**Task 2:**

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer screen

Description automatically generated

**Task 3:**

The way we designed the stack and the queue was heavily based on the structures that we learned about in the lecture videos for the week.  For the stack, we made sure to include all of the proper member functions of a stack, including IsEmpty and IsFull (which we included as our first two member functions, as we called them in multiple other member functions), Push and Pop to add and remove/return items to and from the stack, MakeEmpty to clear the stack, Top to peek at the top item of the stack, and Length to return the number of items in the stack.  We also included a default constructor of the Stack class to initialize our stack to be empty.  To implement all of these member functions, we had to use two attributes: one array of characters of size 25 that would hold all of the items in the stack and one integer called “next” that would store the index of the array where the next item to be added to the stack would go.  In our default constructor, we just initialized the next variable to be index 0.  When we would Push items to the stack, we would add them to the array at the index held by our next variable and increment our next variable by 1, since the next item in the stack would then have to go in one index greater than the one our next variable is currently holding.  Similarly, when we would Pop items from the stack, we would return the item from the array and decrement our next variable by one.  We also added a PrintStack member function that would display all of the values in our stack in the order in which they would be returned.  For our Pop and Top member functions, we had to throw exceptions when the stack was empty and nothing could be returned.  We had to include try and catch statements in our main code every time we called one of these member functions in case an exception was thrown.

When we designed our queue, we made sure to include all of the proper member functions, much like what we did for our stack.  We created IsFull and Isempty functions, which we also called in our other member functions (much like with our stack), Enqueue and Dequeue member functions to add and remove/return items to and from the queue, MakeEmpty to clear the queue, Peek to preview the next item that would be removed and returned from the queue, and Length to return the number of items in the queue.  We also included a default constructor of the Queue class to initialize our queue to be empty.  To implement all of these member functions, we had to use four attributes: one array of characters of size 25 that would hold all of the items in the queue, one integer called “start” that would store the index of the array where the next item to be returned (first item to be added) is located, one integer called “next” that would store the index of the array where the next item to be added to the queue would be located, and one integer called “count” that stores the number of items in the queue.  In our default constructor, we would initialize the start, next, and count variables to all be 0.  When we would Enqueue (add) items to the queue, we would add them to the array at the index held by our next variable and increment our next variable by 1, since the next item in the queue would then have to go in one index greater than the one our next variable is currently holding.  Similarly, when we would Dequeue (remove) items from the queue, we would return the item from the array and increment our start variable by one, since queues operate on a first in, first out basis.  Therefore, incrementing start would take us to the location of where the next item to be removed would be located.  To help make a “wrap-around” queue, we followed the instructions of the lecture videos and, when we would increment our start and next variables, we included the modulus operator and our array size of 25 to get the proper values for start and next, so that they would wrap around the queue to the beginning of the array when the end of the array is reached.  The count variable is also incredibly important for the design of the queue, as there are two possibilities when the start and next variables are the same: the queue could either be empty or full.  To distinguish between these two cases, we created a count variable and added another condition to both of our cases for IsFull and IsEmpty: if start and next were the same and the count was equal to zero, then we knew the queue was empty, and if start and next were the same and the count was equal to our max size, then we knew the queue was full.  We also added a PrintQueue member function that would print all of the items in our queue in the order in which they would be returned.  For our Dequeue and Peek member functions, we had to throw exceptions when the queue was empty and nothing could be returned.  We had to include try and catch statements in our main code every time we called one of these member functions in case an exception was thrown.

For the main code, we placed everything into a while loop so that we would allow the user to keep testing different messages without having to restart the program every time.  We also give them an option to exit the program by asking them after they test a message if they would like to test another one.  If they select the option to not test another message, then the program sets the Boolean value of “exit” to be false so that it exits the while loop and therefore exits the program.  Within our while loop, we create an instance of a stack to act as the sender, a queue to act as the receiver buffer, and a receiver stack to transform our message to be what it should be.  We do this each time that the while loop executes so that we start with a new, empty stack and queue for every instance that we execute the code.  We would then ask the user to input their message that they would like to send and use the getline function to grab the entire line, even if it included spaces, up until it reached a newline.  Then we would add the entire message, character by character, into the sender stack, so that when we would remove everything from the stack and “send it” to the receiver, it would be in big endian form with the last letter of the message being sent first.  Then, our queue receiver would act as a buffer and receive all of these inputs from the sender stack and would store them in the order in which they were returned from the stack.  Then, the queue would add all of its items to the receiver stack, so that the last letter of the message (the first item of the queue) would be the first item added to the receiver stack.  This way, when the message from the receiver stack is returned and added to a final message string (in a character by character format), the first letter of the message is the last item added to the receiver stack, so it will be the first item returned to the final message, therefore displaying your final message in the format that you desired after going through the sending and receiving process.

We had one consideration for the tasks 1 and 2 main code.  We considered modifying our code so that when the user would try to add an item to a full stack or queue, the code would not prompt them for an input before telling them that the stack or queue was full and that nothing could be added.  However, we decided to not include it as we had to move on to task 3 and we saw it unnecessary to include this change to our code considering that the item was not added to the stack or queue anyway, regardless of the user’s character input.  Therefore, we believed that our code was good as it was and moved on to complete task 3.

**Task 3 Screenshots:**

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer screen

Description automatically generated

**Group Member Contributions:**

The lab was worked on together by both Ryan and Thomas while on a call together in Microsoft Teams. Both worked on coding the stack and queue implementation in the header and then the main for task 1 and task 2 and then chose the header and the main that worked the best for final submission. Task 3 was worked on by both but since it was more complicated than task 1 and 2 we had one person code it and share their screen so we could both see it and comment on anything that should be added or changed. The main for task 1 and task 2 was written by Thomas while the header for those tasks was written by Ryan. For task 3 Ryan typed the code out but it was worked on together. For the final grade each member of the group should receive 100 percent of the grade as we feel that we both evenly contributed to the lab and worked on it together the whole time it was being worked through.