Assignment Queues: Circular Array Buffers and Priority Queues

COSC 2336: Data Structures and Algorithms

Summer 2021

Objectives

- Implement Queue API functions.
- Introduction to the concept of managing a fixed sized buffer as a circular buffer for an array based queue.
- Learn about priority queues and how to use them.
- More examples and practice of using inheritance and class abstractions / ADT in C++.

Description

In this assignment, we will be looking at array and linked list implementations of the Queue API/abstraction you should have been learning about in this unit. We will switch back and concentrate more on the array based version of the Queue `AQueue` this week, though you will be implementing one of the methods for the linked list version as well.

As with the previous assignment, both our textbook sources give examples of a `Queue` ADT Malik 6th edition pg. 1201, and Shaffer 3.2 C++ edition, pg. 129. The `Queue` ADT we give and complete in this assignment is similar in structure to the `List` and `Stack` ADT from the previous assignments. An array based (`AQueue`) and linked list based (`LQueue`) implementations of the `Queue` abstraction are again given and used for this assignment.

A Queue is a is a first-in-first-out (FIFO) data structure. The first value put into a queue will be at the front, and values enqueued after this one line up behind it.

The array based implementation of a Queue has some interesting considerations in terms of performance. As you should be familiar with by now, we have an issue if we try and use an array for our Queue data type. If we enqueue new items onto the end of the array, this is typically a constant $\mathcal{O}(1)$ time operation (unless the array has to be grown and copied). But taking an item off of the front (index 0) of the array means we have to shift all items down by 1 index, turning the `dequeue()` operation into a $\mathcal{O}(n)$ operation.

And we can't get around this by trying to reverse which ends we use for enqueuing and dequeueing. If we use the back (last) index of the array as the front of the queue for dequeueing, then that operation becomes constant time. But to insert new values into the array at index 0 means we will have to shift values every time we `enqueue()` onto the queue. So at least one of the operations will have to be $\mathcal{O}(n)$ if we use the array based implementation of the queue in the normal way.

However, we can use an array in such a way to allow for constant time $\mathcal{O}(1)$ operations for `enqueue()` `dequeue()` and `front()` operations of our `Queue`. However, to do this we have to treat the array of values as a circular buffer instead of as a regular array. A circular buffer is a common data structure used in programs to allow a fixed sized array to work as an efficient queue. We still have the problem that if we need to add more values than the array will hold, we will have to grow the array. So `enqueue()` operations can always be $\mathcal{O}(n)$ in the worst case if we have to allocate more memory and copy the existing values to the new memory. To implement a circular buffer, we have to keep track of the current front and back index. Initially when we enqueue the first item onto the queue, the front and back index will both be index 0, the location where we place the first item. When we add more items, we add them into index 1, 2, 3, etc. So the front would still be index 0, but back will point to an item at a higher index, and we increase back index for each new item we enqueue onto the queue. But if we dequeue an item to remove it from the queue, instead of shifting values, we just increment the front index by one. So for example, lets say we enqueue items `5, 7, 3, 2` onto the queue in that order (so 5 is at the front of the queue) and then dequeue the front item `5`. We would end up with an array that looks like the following:

```
index       0      1          2          3         4   ...
values =           7          3          2
              frontIndex             backIndex
```

The array is a circular buffer, because we will wrap around the end of the array as needed. For example, lets assume that the allocation size of the array is `allocationSize = 5`, so that the valid indexes are 0 to 4 as shown in the previous example. If we enqueue two more items, `8, 1`, the array of `values` would now look like the following:

```
index        0             1          2          3         4   ...
values =     1             7          3          2         8
         backIndex     frontIndex
```

When we enqueued 8 it just ended up at index 4, the last valid index for this array. But when we enqueue 1 the array is not yet full. So we wrap back around to index 0 and this value is placed there. At this point the array is full, the `size` of the queue is 5, because it is holding 5 values. If we want to enqueue another value on this queue, it will have to be dynamically grown so we can fit more values. But if we end up dequeueing 1 or more values before we need to enqueue some more, this would make some room, and we might be able to keep going for a bit before we have to dynamically grow the queue.

So notice that, again ignoring the case where `enqueue()` needs to first grow the array allocation, both `dequeue()` and `enqueue()` operations are constant time $\mathcal{O}(1)$ operations. In both cases we simply have to increment the front or back index. Though we do have to be careful that we wrap around the end of the buffer when we increment either of these indexes. A common method to do this when incrementing the front or back index is to use modulo arithmetic. For example, in your `enqueue()` method which adds a value onto the end of the array, you might first increment the `backIndex` to the next location of the array where the new value will be inserted. You can do the following:

```
backIndex = (backIndex + 1) % size;
```

This has the same effect as explicitly testing if we have gone past the end of the array

```
backIndex++;
if (backIndex >= size)
{
  backIndex = 0;
}
```

The second example may be clearer, but they both perform the same task. In the first case, if the new value of `(backIndex + 1)` is equal to the array `size`, the remainder of dividing by `size` will be 0, which will wrap the `backIndex` back around to 0 as desired. Using modulo (remainder) arithmetic for circular buffer indexing is common and you will see it done this way often if you look at other code using an array as a circular buffer.

In this assignment you will be required to implement the `enqueue()` `dequeue()` and `front()` methods of the array based `AQueue` class. For both `enqueue()` and `dequeue()` you will need to correctly increment the back and/or front index, and wrap it around the circular buffer correctly as just discussed.

The linked list implementation of the `Queue` abstraction does not have a similar performance issue as the array based implementation. For the linked list `LQueue` implementation, it is natural to use the front of the linked list as the `Queue` front, and the back of the linked list as the back of the `Queue`. Both `enqueue()` and `dequeue()` can be done in constant time $\mathcal{O}(1)$ time when we use the linked list in this way. For example, if we enqueue on the back of the queue, as long as we keep a pointer to the last node of the linked list, it is easy to create a new node and make it the new last node. Likewise, removing the front node of the linked list is constant time if we have a pointer to the front of the linked list, as we have seen before.

You will not be implementing any of the member methods of the `LQueue` class. However, there are two additional tasks to be done once you get the missing `AQueue` member functions added. We will be adding new `PriorityQueue` implementations to the hierarchy of defined queue types. A priority queue is a queue that works similar to the queues that we discussed, but the front item of a priority queue will always be the item in the queue with the highest priority, instead of the item that has been waiting the longest. Priority can be defined as needed for the application. We will assume for our priority queue classes that items of type `T` are ordered by their priority using the normal boolean operations like `operator<()`, `operator>()`, etc. This means that the item with a higher priority will give a boolean result of true if we compare it to a lower priority item using the greater than operator.

```
T value1;
T value2;

if (value1 > value2)
{
  cout << "value1 is higher priority than value2" << endl;
}
```

You will be adding in new `enqueue()` methods for both an array based priority queue `APriorityQueue` and the linked list based priority queue `LPriorityQueue`. In both cases, you will modify the enqueue method to maintain the queue of values in a sorted order, so that the highest priority item is always at the front of the queue, and the lowest is always at the back of the queue. You will do this in both cases by inserting any new values into the queue into the correct position so that the queue is ordered by priority. In both cases this will require you to search the queue, so this means that `enqueue()` will become a $\mathcal{O}(n)$ operation for both the array and linked list priority queues. There are ways to improve on this, some of which we discussed in this units materials, e.g. keeping the items organized as a heap.

Overview and Setup

For this assignment you will be given the following files that you will be using and adding code to for this assignment.

File Name	Description
`src/test-AQueue.cpp`	Unit tests of the array based `AQueue` implementation of the `Queue` API
`src/test-LQueue.cpp`	Unit tests of the link list based `LQueue` implementation of the `Queue` API
`src/test-APriorityQueue.cpp`	Unit tests of the array based `APriorityQueue` implementation of the `Queue` API
`src/test-LPriorityQueue.cpp`	Unit tests of the link list based `LPriorityQueue` implementation of the `Queue` API
`include/Queue.hpp`	Header file of the ADT base `Queue` class that defines the `Queue` interface / abstraction
`include/AQueue.hpp`	Header file of the concrete array based implementation of the `Queue` abstract data type
`include/APriorityQueue.hpp`	Header file of the concrete array based implementation of a priority queue for the `Queue` abstract data type
`include/LQueue.hpp`	Header file of the concrete linked list based implementation of the `Queue` abstract data type
`include/LPriorityQueue.hpp`	Header file of the concrete linked list based implementation of a priority queue for the `Queue` abstract data type
`include/Node.hpp`	Header file of the Node structure used by the linked list implementation
`src/Queue.cpp`	Implementation file of common methods of the `Queue` base class
`src/APriorityQueue.cpp`	Implementation file for the `APriorityQueue` member functions that implement the concrete priority queue array based `Queue`
`src/LQueue.cpp`	Implementation file for the `LQueue` member functions that implement the concrete linked list based `Queue`
`src/LPriorityQueue.cpp`	Implementation file for the `LPriorityQueue` member functions that implement the concrete priority queueu linked list based `Queue`

This week you will mainly be working in the array based `AQueue` and `APriorityQueue` header and implementation files. You will add in the missing `front()`, `enqueue()`, and `dequeue()` methods for the array based `AQueue` class. Then you will implement the modified `enqueue()` methods to insert items into the queue by priority order for both the `APriorityQueue` class, and the linked list `LPriorityQueue` class.

As usual, before starting on the assignment tasks proper, you should make sure you have completed the following setup steps.

1. Copy the assignment repository on GitHub using the provided assignment invitation link for 'Assignment

Recursion' for our current class semester and section.
2. Clone the repository using the SSH URL to your local class DevBox development environment.
3. Configure the project by running the `configure` script from a terminal.
4. Confirm that the project builds and runs, though no tests will be defined or run initially. If the project does not build on the first checkout, please inform the instructor.
5. You should create the issue for Task 1 and/or for all tasks for the assignment now before beginning the first task. On your GitHub account, go to issues, and create it/them from the issue templates for the assignment. Also make sure you link the issues to the `Feedback` pull request after creating them.

Assignment Tasks

Task 1: Implement the `AQueue` `front()` Accessor Method

As usual make sure that you have created Task 1 on your GitHub repository for this assignment and have linked the Task 1 issue to the open `Feedback` pull request in your GitHub assignment repository.

There is a test case in `test-AQueue.cpp` for task 1. It only does a few simple tests of the `front()` member method, but as usual you should uncomment this test case first, and add in the function prototype and a stub method to check that you can compile and run the tests before proceeding with the implementation. In addition, the declaration of the `front()` method is commented out of the base class `Queue.hpp` header file. You should uncomment that as well before proceeding with implementing this method.

The `front()` accessor method is basically the same as the `top()` method of our `Stack` class from a previous assignment. Though in this case, the `AQueue` class has an explicit member variable named `frontIndex` that is the index of the front item of the queue in the `values` array.

You should implement this function by simply returning the front item of template type `T` from the queue. But as usual for these accessor methods, you should first check if the queue is empty (reusing the `isEmpty()` method), and throw an exception of type `QueueEmptyException` if an attempt is made to peek at the front item of an empty queue.

Once you are satisfied with your work and can pass the tests for the task 1, commit your work and push it to the `Feedback` pull request of your GitHub repository for this assignment.

Task 2: Implement the `AQueue` `dequeue()` Mutator Method

Start task 2 by uncommenting the next unit test in the `test-AQueue.cpp` and creating the function declaration and a stub function to make sure the tests run. Also don't forget to uncomment the `dequeue()` declaration in the `Queue.hpp` base class header before beginning implementation.

The `dequeue()` method should remove the item that is currently at the front of the queue, similar to the `pop()` and `removeFront()` methods you have seen in previous data types. As usual with removing items from data structures, fist check if the queue is empty before doing the dequeue, and if it is throw a `QueueEmptyException` to let the caller know something is amiss.

There is one wrinkle with the `dequeue()` method. You are again using the `frontIndex` member variable to determine the index of the current value at the front of the queue. To dequeue the front value, you simply need to increment the `frontIndex` by one. But as discussed in the introduction above, we are treating the `values` array as a circular buffer. So there is the possibility that, after incrementing the front index, you need to wrap back around to index 0 of the circular buffer. You can do this with an explicit test using an `if` statement, or using modulo arithmetic as described above. Make sure that you are testing the `allocationSize` of the array, not the queue `size`. These are different. The `size` of the queue is the actual number of items currently on the queue, while the `allocationSize` is the amount of memory currently allocated for the `values` array.

Once you have correctly incremented the `frontIndex`, don't forget that your queue has shrunk by 1 item, so you also need to update the `size` member variable appropriately.

Once you are satisfied with your work and can pass the tests for the second test case, commit your work and push your commit to your GitHub assignment repository `Feedback` pull request.

Task 3: Implement the AQueue enqueue() Mutator Method

Do the same as previous 2 tasks to start task 3, uncomment the next test case in test-AQueue.cpp, add in the declaration and stub function for the enqueue() method, and don't forget to also uncomment the enqueue() virtual declaration in the Queue.hpp base class header file.

The enqueue() method will be similar to the push() and insertBack() methods from previous assignment, though again one difference is that we now have an explicit backIndex member variable that indicates the current index of the last or back item on the queue.

First, before trying to insert the new value on the back of the queue, you need to ensure there is enough room in the values array to hold another value. As with previous methods and classes, a growQueueIfNeeded() private member method has already been provided for your AQueue class. You should start by calling this method. After this function returns, you will be sure there is room to put the new value into the array at the back of the queue.

The backIndex should always point to the index of the actual current back item. So to insert a new value on the back of the queue, you must first increment the backIndex. As with the dequeue() method, after incrementing this value, you have to check that you have not wrapped around the circular buffer first before proceeding, and make backIndex point to index 0 if you have indeed wrapped around the buffer.

Once you have incremented the backIndex, you can gen safely copy the value given to the enqueue() method into the values array. Also don't forget that you array has now increased in size by 1.

Once you are satisfied with your work and can pass the tests for the third test case, there are some additional tests in the test-AQueue.cpp testing file. You should uncomment the additional 2 tests cases as well and ensure that your new methods can pass all of the more extensive testing of the AQueue class. Once you are passing all of the tests, commit your work and push it to the Feedback pull request of your repository.

Task 4: Implement the APriorityQueue enqueue() Overridden Method

The first 3 tasks, to add in the missing member functions of the AQueue class, were meant to be warm-up exercises, and hopefully were not too tough to figure out. The next 2 tasks will be a bit more challenging.

For this task you will be adding a new method to the APriorityQueue class. The purpose of the priority queue is discussed a bit above in the introduction for the assignment. To implement an array based priority queue, you need to inherit from the AQueue class, and just override the enqueue() method so that new values are enqueued onto the queue in sorted order, instead of simply being put to the back of the queue.

The header file and implementation file have already been given to you for the APriorityQueue.[hpp|cpp] class. You need to add in a declaration for the enqueue() method in the header file first. This member function will have the same signature as for the AQueue class you are inheriting from, so you can simply copy the declaration from that header into the header for the priority queue version.

Likewise it is suggested that you start by simply copying your full enqueue() implementation that you just finished from the AQueue.cpp implementation file into the APriorityQueue.cpp implementation. Your implementation of enqueue to create a priority based queue will start off the same way, we simply need to modify its behavior to search for and insert any new values into their correct sorted position on the queue. Though don't forget now that your enqueue() is a member of the APriorityQueue<T>:: class, so you do have to make a small change to the function signature in the implementation file here.

Once you have added these to the APriorityQueue header and implementation file, uncomment the test cases in the test-APriorityQueue.cpp file. There are three test cases that test an APriorityQueue with integers, with strings and on a Job user defined class. You can uncomment both of these, and make sure your project compiles and runs the tests. All of the tests you just uncommented should run, but of course, since you have not yet modified the enqueue() function to insert items by their priority, some of them will be failing.

You should leave in all of the code from your AQueue enqueue() method, as we want to start by inserting the new value to the back of the queue. There are two basic approaches you can then take to position the new value from the end to its correct location:

1. Use a reverse bubble sort pass, e.g. iterating from the back of the queue towards the front, compare each value to the previous value, and if they are out of order swap them. Though it doesn't hurt to do this all of the way till the front, you can stop bubble/swapping as soon as you find two items that are not out of order.

or

2. Like an insertion sort. Again iterating from the back of the queue towards the front, shift up items by 1 location. You perform the shifting up until you detect an index with a greater or equal priority to the new value being placed in the queue. Once you find this location, you have just shifted the item with a smaller priority up by 1 location, leaving a hole in the array, where you should insert the new value.

For approach 2, you do need to increase the size of the queue by incrementing the `backIndex`, but you don't really need to start by placing the new value into this back location, since the first shift will simply overwrite the value, and you will be inserting the value into the whole/position you determine as the location the value should end up at.

Both of these approaches can be made to work. Performing a shift and insertion can be a bit more efficient, since it takes more work to swap values than to simply shift them (3 copies vs. 1 copy). But you can attempt either that seems easiest to you to implement.

Here are a couple of general hints and ideas:

1. Iterating backwards through the circular buffer is tricky. You can start by defining something like:

```
int currentIndex = backIndex;
int previousIndex = currentIndex - 1;
```

   Now you can check and swap the current with the previous, or simply shift up the previous to the current, if they are out of order. However, be aware, that whenever you decrement any index you have into the circular buffer, you have to take into account that the index could have been 0, so when you decremented, your previous index here could be -1. So every time after you decrement an index into your circular buffer, you have to take care of potentially wrapping it, e.g.

```
if (previousIndex < 0)
{
    previousIndex = allocationSize - 1;
}
```

   Notice that the correct index to wrap back around the circular buffer is `allocationSize - 1`. The array of `values` always has `allocationSize` elements in the array. So the last index of this array is at `allocationSize - 1`. Also note that the modulo operator/trick does not work in C/C++. Unfortunately, the `%` operator actually performs the remainder function in C/C++, so if you do `-1 % 5` you will get a result of `-1`. The true modulo operator should return a value from 0 to 4 for a modulus of 5. Many other languages (like Python, Scheme, Haskell) actually will do this correctly, or have separate operators/functions for remainder and true modulo. You can explicitly test using an `if` condition, as shown above. Or you should find that there is a member method named `modulo()` in the `APriorityClass` which you could use to calculate the modulo.

2. You need to be careful when testing that you stop when you find a value that is of greater priority or of **EQUAL** priority. We test for this a bit in the test cases. You can't really tell when managing `int` or `string` values if this is being done correctly. But the idea is as follows. If two items are of equal priority, then you want them to end up being enqueued by their order of arrival (like in a regular queue). Thus you have to make sure that you do not end up inserting in front of any equal priority items, new items should end up in back of any items of equal priority.

NOTE: We ran into this issue before. `APriorityQueue` inherits from `PriorityQueue`. All of the member variables you will be using in your `enqueue()` method, like `frontIndex`, `backIndex`, `allocationSize`, etc. are all defined in either the `AQueue` class, or even in the original `Queue` abstract base class. Usually this is not a problem, but because of some technical cruftiness of C++ template implementations, this means if you want to refer to a member variable defined in a class you inherit from, you need to reference it either using `this`:

```
this->frontIndex--;
```

or you can prepend a namespace qualifier like the following:

```
AQueue<T>::frontIndex--;
```

You don't normally have to do this for regular classes, but because of some cruftiness with templates in C++, the C++ compiler can't figure out a reference to a member variable name is in a parent class without one or the other of these qualifiers.

However, if you look in the `APriorityQueue.cpp` implementation file, you will see the following preprocessor define macros:

```
#define size AQueue<T>::size
#define allocationSize AQueue<T>::allocationSize
#define frontIndex AQueue<T>::frontIndex
#define backIndex AQueue<T>::backIndex
#define values AQueue<T>::values
#define growQueueIfNeeded() AQueue<T>::growQueueIfNeeded()
```

This is a little bit of a trick here to make the code look a bit less crufty. But for example the `#define` for `frontIndex` means that whenever you simply use `frontIndex` the C preprocessor will replace that string with the string with a namespace qualifier. So simply put, this means that if you rely on these defines, you can write your references to the member variables from the `AQueue` parent class without qualifying them with `this` or the parent class namespace, and the C preprocessor will fix it for you so that the code still compiles. This should make your code a bit less crufty to write.

Task 5: Implement the `LPriorityQueue` `enqueue()` Overridden Method

For your final task, we will be switching gears back to the linked list implementation of our `Queue` abstraction. If you haven't looked at the linked list based implementation, you might want to look over it now. For the linked list queue, we end up with a very similar implementation to the `List` class we had in a previous assignment. It is most natural to treat the end of the list as the back of the queue, which is kept track of by the `backNode` member variable. And likewise the beginning of the list works as the front of the queue, and the `frontNode` member variable points to this node. By enqueueing on the front of the linked list and dequeueing from the back, both operations are $\mathcal{O}(n)$ as we discussed a bit in the introduction.

To begin task 5, do similar steps as in task 4. Copy the declaration of the `enqueue()` method from the `LQueue.hpp` header file, and the implementation of `enqueue()` from `LQueue.cpp`. Also go ahead and uncomment all of the tests in `test-LPriorityQueue.cpp`. The project should compile and run like in task 4, but since the `enqueue()` method is not keeping items sorted by priority, some of the tests will be failing.

The approach for keeping the linked list is similar as you just did for the array based implementation. We cannot iterate backwards through the linked list, so we have to work from the front of the list. You can do either of the following strategies to implement the priority enqueuing algorithm:

1) A bubbling approach again. Create a new node and insert it on the front of the linked list. Then compare node values with the next node value, and swap the actual values if they are out of order. Notice for this approach you actually swap the values in two nodes, not the nodes themselves. You can stop once you detect you no longer need to swap values because they are in the correct order.

or alternatively you can do an insertion approach

2) Create a new node with the new value. But instead of inserting on the front, start a search of the list. You want to find the position where the current node is greater or equal in priority to the new value, but the next node it points to is lower priority. When you find that position, insert the new node, by pointing the current position to the new node, and the new node next link points to the next node with the lower priority.

For both approaches you need to be careful about inserting a node to become the new `frontNode`. In fact you probably want to have 1 special case that checks if the list is empty, and just makes the new node the front and back and is then done in that case.

After that, for approach 1 you can just start bubbling. For approach 2 you might also want a special case that checks if the new node should become the new front node because it has a higher priority than any existing node, and if so just insert this new node to become the new front of the linked list.

For the priority queue, you actually don't really need the `backNode` member variable, since you will not be enqueing on the back of the list, but need to be inserting into the list at the correct priority position. So for either approach 1 or 2 you don't really need to worry if the `backNode` member variable is updated correctly or not, though you might want to still try and consider that in case in future we needed to add some functionality that did need to keep track of the back node.

Once you are satisfied with your implementation and it can pass the tests in the `test-LPriorityQueue.cpp` test file, commit your work and push it to the `Feedback` pull request of your assignment repository.

Assignment Submission

For this class, the submission process is to correctly create pull request(s) with changes committed and pushed to your copied repository for grading and evaluation. For the assignments, you may not be able to complete all tasks and have all of the tests successfully finishing. This is ok. However, you should endeavor to have as many of the tasks completed before the deadline for the assignment as possible. Also, try and make sure that you only push commits that are building and able to run the tests. You may loose points for pushing a broken build, especially if the last build you submit is not properly compiling and running the tests.

In this problem, up to 25 points will be given for having at least 1 commit that compiles and runs the tests (and at least some attempt was made to work on the first task). Thereafter 15 points are awarded for completing each of the 5 tasks. However you should note that the autograder awards either all points for passing all tests, or no points if any test is failing for one of the tasks. Also note that even if you pass all tests, when the instructor evaluates your assignment, they may remove points if you don't follow the requirements for implementing the code (e.g. must reuse functions here as described, need to correctly declare parameters or member functions as `const` where needed, must have function documentation correct). You may also loose points for style issues. The instructor may give back comments in pull requests and/or create new issues for you if you have issues such as these, so it is good to have work committed early before the due date, so that the instructor may give feedback requesting you to fix issues with your current submission.

Program Style

At some point you will be required to follow class style and formatting guidelines. The VSCode environment has been set up to try and format your code for some of these guidelines automatically to conform to class style requirements. But not all style issues can be enforced by the IDE/Editor. The instructor may give you feedback in your pull comments and/or create issues for you for the assignment that you need to address and fix. You should address those if asked, and push a new commit that fixes the issue (or ask for clarification if you don't understand the request). In general the following style/formatting issues will be required for programs for this class:

1. All programs must be properly indented. All indentation must be consistent and lined up correctly. Class style requires 2 spaces with no embedded tabs for all code indentation levels. The editor style checker should properly indent your code when you save it, but if not you may need to check or correct this if code is misaligned or not properly indented.
2. Variable and function names must use `camelCaseNameingNotation`. All variable and function names must begin with a lowercase letter. Do not use underscores between words in the variable or function name. Often function names will be given to you, but you will need to create variables, and maybe some functions, that conform to the naming conventions.
    - Global constants should be used instead of magic numbers. Global constants are identified using `ALL_CAPS_UNDERLINE_NAMING`.
    - User defined types, such as classes, structures and enumerated types should use camel case notation, but should begin with an initial upper case letter, thus `MyUserDefinedClass`.
3. You are required to use meaningful variable and function names. Choosing good names for code items is an important skill. The code examples and starting code tries to give examples of good and meaningful names. In general, do not use abbreviations. Single variable names should be avoided, except maybe for generic loop index variables `i`, `j`, etc. Make your code readable, think of it as writing a document to communicate with other developers (and with your instructor who will be evaluating your code).
4. There are certain white space requirements. In general there should usually never be more than 1 blank line in a row in your code. Likewise there should usually not be more than 1 blank space on a line. There should be 1 blank space before and after all binary operators like `+`, `*`, `=`, `or`.
5. Function documentation is required for all regular functions and all class member functions. You need to follow the correctly formatted Doxygen function documentation format. We will use function documentation generation, and you should be sure your documentation can be built without emitting warnings or errors. Likewise all files should have a file header documentation at the top. You should edit the file header of files where you add in new code (not simply uncommenting existing code). Make sure the information has your correct name, dates, and other information.

6. Practice using proper Git commit messages. You should refer to issues and tasks correctly in commit messages.

Additional Information

The following are suggested online materials you may use to help you understand the tools and topics we have introduced in this assignment.

- Lecture U10-1 Implementing Queues
- Data Structures - Full Course Using C and C++ The videos 4:32:17 Introduction to Queues through 4:56:33 Linked List implementation of Queue are all excellent introductions to the fundamentals of implementing Queues as arrays and linked lists.
- C++ Classes and Objects
- C++ Inheritance
- C++ Templates