/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* readme.txt template

\* Guitar Hero: RingBuffer implementation with unit tests and exceptions

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Name: Dheera Vuppala

Partner: Danny Lisamay

Hours to complete assignment: Between 6-8 hours

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Briefly discuss the assignment itself and what you accomplished.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

In this assignment we implemented a circular array where we could push integers onto the back of the array and remove them from the front of the array. The back of the array connected to the front of the array, making it circular. The circular array represents the ring buffer feedback mechanism that determines the frequency of the sound.

To test our implementation, we wrote unit test cases that individually tested each method. This means we checked that isFull(), isEmpty() were working right after initialization, after enqueuing multiple elements and again after dequeuing. We also checked edge cases (for example: what happens when you enqueue after you reach capacity, before you reach capacity and the same for dequeue. We also checked that the peek function didn’t work on an empty ring buffer).

In the case, when a function shouldn’t work, rather than the program crashing, we added exceptions so the user would know what is wrong with their program. The ring buffer’s size has to be greater than 0, so if initialized with a size less than 0, we threw an invalid argument exception and printed an error message telling the user what was wrong.

For the enqueue, dequeue, and peek function if the user tried to push on another element when the ring buffer was full, or dequeue/peek even if it was empty then we threw a runtime error exception and printed the message explaining what was wrong (either the ring was empty or it was full).

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Discuss one or more key algorithms, data structures, or

\* OO designs that were central to the assignment.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

The circular array was crucial to this assignment because it allowed for us to push and pop elements (to the back and from the front of the array). Everytime we added an element we had to make sure that the front connected to the new back and that when we deleted an element the new front connected to the back of the array. We also had to check for scenarios when there was only one element and the front and rear were the same item. If this didn’t work properly, our ringbuffer wouldn’t have worked correctly.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Briefly explain the workings of the features you implemented.

\* Include code excerpts.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

In the constructor, we created the circular array which would act as our ringbuffer by initializing the array to the capacity passed into the constructor. Code snippet is shown below:

if (cap < 1) {

std::cout << ("RB constructor: capacity must be greater than zero.")

<<std::endl;

throw std::invalid\_argument

("RB constructor:capacity must be greater than zero.");

}

this->cap = cap;

front = rear = -1;

arrCirc = new int[cap];

sizeOfArr = 0;

We also checked that if the capacity was less than 1, and if so we threw an invalid argument exception.

To add an element to the array, we implemented the enqueue function. Here we checked three scenarios (if the array was empty, had only one element, or had multiple). Code snippet shown below:

else if (front == -1) {

front = rear = 0;

arrCirc[rear] = x;

} else if (rear == cap - 1 && front != 0) {

rear = 0;

arrCirc[rear] = x;

} else {

rear++;

arrCirc[rear] = x;

}

sizeOfArr++;

}

Once we added the element to the back of the array, we increment the current number of elements in the array.

We also checked that the array wasn’t already full. If it was we threw an exception.

if ((front == 0 && rear == cap -1) || (rear ==(front-1)%(cap-1))) {

std::cout << "enqueue: can't enqueue to a full ring" << std::endl;

throw std::runtime\_error("enqueue: can't enqueue to a full ring");

}

We followed the same idea for the dequeue function shown below:

int16\_t RingBuffer::dequeue() {

if (front == -1) {

std::cout << "Ring is empty" << std::endl;

throw std::runtime\_error("Ring is empty");

}

int data = arrCirc[front];

if (front == rear) {

front = rear = -1;

} else if (front == cap -1) {

front = 0;

} else {

front++;

}

sizeOfArr--;

return data;

}

However in this case we decremented the number of elements in the array, once we popped the front out and returned the element we popped out.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Did you complete the whole assignment?

\* Successfully or not?

\* Indicate which parts you think are working, and describe

\* how you know that they're working.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

We completed the whole assignment successfully! Everything works, including the cpplint style checks! We know that they’re working because the unit tests successfully passed, there are 0 errors in cpp lint for both the RingBuffer and test cpp files.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Does your RingBuffer implementation pass the unit tests?

\* Indicate yes or no, and explain how you know that it does or does not.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Yes! In the unit tests we used the BOOST Library functions including BOOST\_REQUIRE, BOOST\_REQUIRE\_THROW and BOOST\_REQUIRE\_NO\_THROW. These functions should have displayed a fatal error if our unit tests didn’t pass, but instead it says No errors detected, and prints the exceptions when they are supposed to happen, meaning that RingBuffer passed all our unit tests.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Explain the time and space performance of your RB implementation

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

For time, our RB implementation is O(1) because it’s all in constant time. There are no loops, and since we add to the back of a fixed size array, we never have to traverse or search for any element. Enqueue, dequeue, and peek functions are all O(1).

The space complexity is O(n) because the circular array is a fixed size and is not dynamically allocated.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* List whatever help (if any) you received from lab TAs,

\* classmates, or anyone else.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

I worked with Danny as partners + also looked at BOOST library online to use for the test.cpp

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Describe any serious problems you encountered.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

The biggest problem we encountered was getting the BOOST\_REQUIRE\_THROW checks to work correctly. Originally, even though we were throwing exceptions, the REQUIRE\_THROW would result in a fatal error. This was because we were catching the exceptions within the method, so the BOOST checks weren’t receiving the exceptions that we threw.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* List any other comments here.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

We completed the extra credit point which was to use a lambda function. Implemented the lambda function in the isFull() method. Before using lambda function, we had another method called size() and in the isFull() method we said return true if size()==cap, else return false.   
  
To use the lambda function, we made the size() method an inline function within the isFull method(). Code snippet shown below:

auto lambaSize = [](int arrSize, int cap) {

if (arrSize == cap) return true;

return false;

};

return lambaSize(sizeOfArr, cap);