This week’s section handout has practice with classes and objects.

**Ring Buffer Queue**

Think back to week 2 when we studied collections. We learned about Queues, a "first-in, first-out" data structure. Today in section, we're going to implement a special type of queue called a Ring Buffer Queue.

A Ring Buffer Queue, or RBQ, is implemented by using an underlying array. In our implementation, the capacity is capped; once the array is full, additional elements cannot be added until something is dequeued.

Another "interesting" thing about RBQs is that we don't want to shift elements when an element is enqueued or dequeued. Instead, we want to keep track of the front and tail of the Queue. For example, say our queue can hold 5 elements and we enqueue 3 elements: 1, 2, 3. Our queue would look like this:

```
1 2 3
```

If we enqueued two more elements, our queue would then be full:

```
1 2 3 4 5
```

At this point, we cannot add any additional elements until we dequeue at least one element. Dequeueing will remove the element at head, and head will move onto the next element. If we dequeue 2 elements, our queue will look like this:

```
3 4 5
```

Now there's room to add more elements! Since we still don't want to shift any elements, adding an additional element will wrap around. So, if we enqueue an element, our queue will look like this:

```
6 3 4 5
```

Handout created by Marty Stepp, Julie Zelenski, and Kate Rydberg.
Week 5 Section Handout

Your job is to implement a RingBufferQueue class. Your class should have the following public methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void enqueue(int elem)</td>
<td>Enqueues elem if the queue has room; throws an error if queue is full</td>
</tr>
<tr>
<td>int dequeue()</td>
<td>Returns and removes the element at the front of the queue; throws a string exception if queue is empty</td>
</tr>
<tr>
<td>int peek()</td>
<td>Returns element at the front of the queue; throws a string exception if queue is empty</td>
</tr>
<tr>
<td>bool isEmpty()</td>
<td>Returns true if queue is empty and false otherwise</td>
</tr>
<tr>
<td>bool isFull()</td>
<td>Returns true if queue is full and false otherwise</td>
</tr>
<tr>
<td>int size()</td>
<td>Returns number of elements in the queue.</td>
</tr>
</tbody>
</table>

You are welcome to add any private methods or fields that are necessary. It can be hard to know where to start when writing an entire class, so we've given you this breakdown:

1. Start by identifying the private fields you will need, then write the constructor and destructor to initialize the fields and do any cleanup, if necessary. Questions to think about:
   - Is it easier to keep track of head and tail (as pictured in the diagrams above)? Or would it be better to track head and size?

2. Write isEmpty(), isFull(), size(), and peek(). Questions to think about:
   - Which of these methods can be const? In general, how do you know when a method can be const?

3. Write enqueue() and dequeue(). Remember to handle error conditions! Questions to think about:
   - Can you call the methods from part 2 to reduce redundancy?
   - Would using modular math help with wrapping around?
   - Should either of these methods be const?

4. Finally, deal with ostream insertion!

If you want more practice with writing classes, think about how you could modify this class to implement a double ended queue. (A double ended queue, or deque, is one where you can enqueue and dequeue from either the front or the back)

Handout created by Marty Stepp, Julie Zelenski, and Kate Rydberg.