Homework – Queues

Brad Peterson – Weber State University

Assignment goals:

- Help review C++.
- To better understand classes, arrays, dynamic allocation and deallocation, error handling, templates, and basic logic.

Description:

For my unit tests, I made a base class with methods that simply have enough logic to compile. You should not modify the base class. Instead, you should modify the derived class and override the constructor, destructor, and all methods there (I gave you an example of overriding the constructor's declaration).

The `QueueForCS2420` class needs to have the following members:

Data members:

- Four unsigned integer data members. Their purposes are to store 1) the total capacity of the array, 2) the number of slots currently used in the queue (the size), 3) the head of the queue, and 4) the tail of the queue. All can be initialized to zero. *Note that #3 and #4 should not be named `front` and `back`, as upcoming methods need to be named `front()` and `back()`.*
- Note that `arr` is already defined in the base class, so you don't need to define it. For most compilers, you must access it using `this->arr` instead of just `arr`.

Constructors and methods:

- A constructor that accepts a capacity parameter. The parameter should be copied into the appropriate data member. The array must be allocated using the new keyword and the address stored in `arr`.
- A destructor is needed as the constructor uses a new keyword.
- `size()` -- The return type is unsigned int. Returns the number of used slots in the queue.
- `push_back()` -- This method should have a single parameter, the data type of that parameter should be `const T&`. The `push_back()` method should have a void return value. This method should see if the queue is full. If so, simply state an error message and return. Otherwise, place the data in the array at the last index. Increment the last. If last is too large (past the bound of the array and the same as capacity), set last back to zero. Increment the number of elements.
- `pop_front()` -- This method should not have any parameter. The return type should be void. The method first checks if the queue is empty. If so, simply state an error message and return. Otherwise, increment the first. If first is too large (past the bounds of the array and the same as capacity), set first back to zero. Decrement the number of elements.
- `front()` and `back()` -- These two methods should check if the queue is empty. If so, state an error message and throw an `std::out_of_range` error. Otherwise, return the data at the head or the tail of the queue, respectively. (Remember, the last index is always one past the actual item).
- `resize()` -- This method has a const unsigned int `newCapacity` integer parameter. This method must create a new Queue, copy items from the old queue to the new queue, and then ensure the old queue's array is reclaimed. For most of this process, use the existing constructor, destructor, and methods. A harder approach which you should avoid is working with both arrays directly and tracking two different set of indexes. For this `resize()` method, start by creating another `Queue` object, sized to the new capacity. Create a while loop that iterates as long as the this object's size is greater than zero. Inside the loop, copy an item from the old queue to the new queue. You can do that by calling the new Queue object's `push_back()` and pass in this's `front()` as the argument. Also inside the while loop, call this's `pop_front()`. After the while loop, call `delete[]` on this's arr to reclaim that space in heap. Now we need to clone over the new Queue object into the old/this Queue object. Copy over all five data members from the new Queue object into the this object. Finally, we need to ensure we don't have

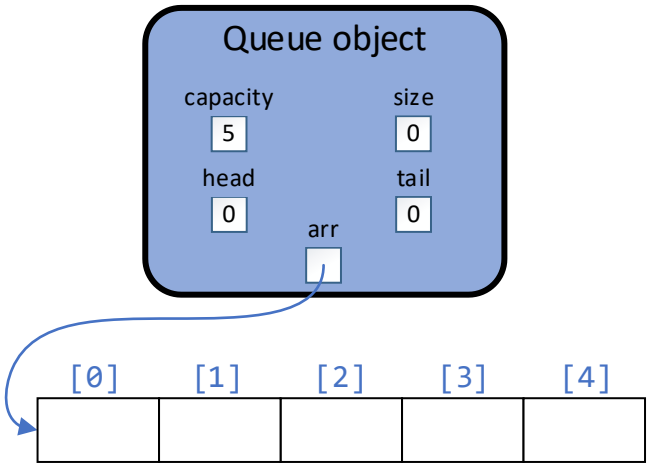
problems with the destructor. When the `resize()` method ends, the new Queue object will trigger its destructor, which can call `delete[]` on the new Queue's `arr`, however, the this's Queue object also has the same address for its `arr`, and we don't want to reclaim an array in use. An easy way to fix this problem is with two additional modifications. First, at the end the `resize()` method, set the new Queue's `arr` to `nullptr`. Second, inside the destructor, add an if statement to check if this's `arr` isn't `nullptr`, and if so, invoke `delete[]`. (Note that much of this process we just did in resize can be managed in a smoother way if we knew about C++'s move semantics, but we haven't been taught these in class yet.)

Use the .cpp file given. Your assignment should pass all tests.

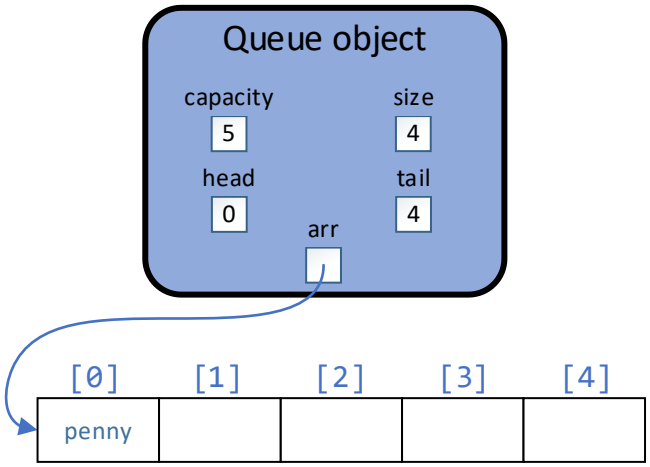
If you haven't yet learned how to use your debugger, please do so!

Visual explanation:

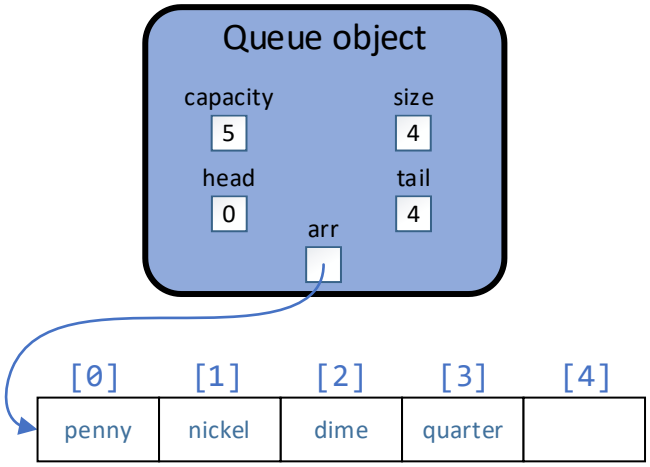
Your queue should start with this state:



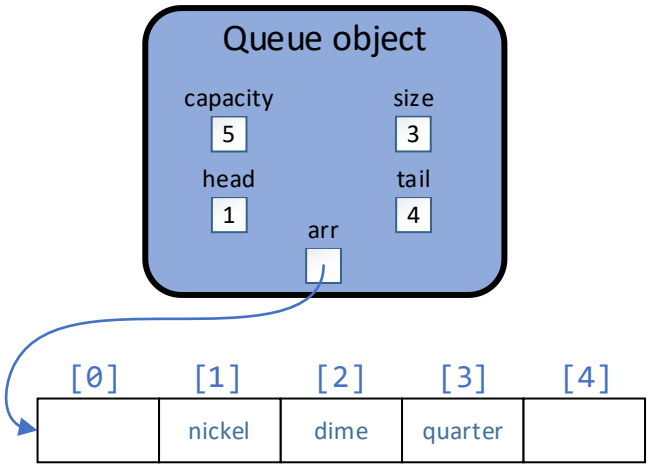
Upon calling `push_back("penny")`, the queue's state is shown below. The size increases by 1. The tail increases by 1 as the item was added to the tail of the queue:



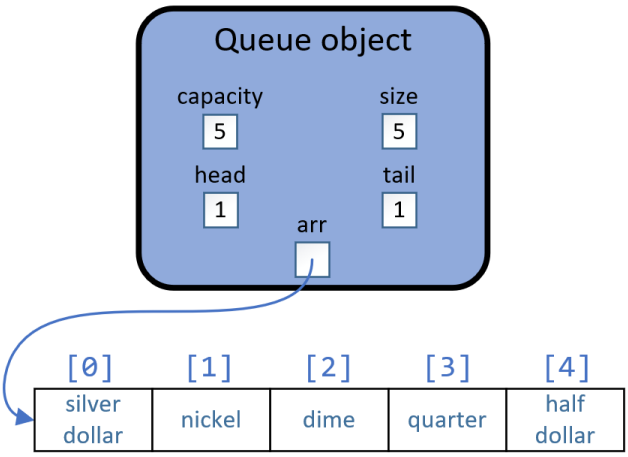
Upon calling `push_back()` for "`nickel`", "`dime`", and "`quarter`", the queue's state is shown below. The size increases by 3 and the tail increases by 3.



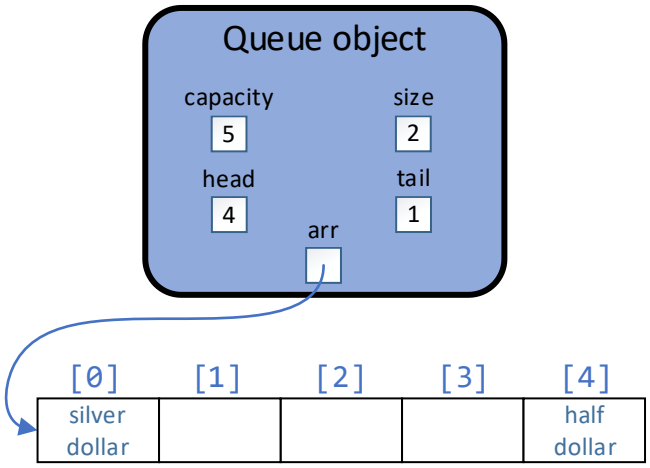
Upon calling `pop_front()`, the queue's state is shown below. The head increases by 1 and the size decrements by 1.



Upon calling `push_back()` for "`half dollar`", and again for "`silver dollar`" the queue's state is shown below. Note that wrapping around occurred. The tail wrapped around to *0* for the "`half dollar`" and then became 1 for "`silver dollar`". The size incremented by 2. The queue is now full and can no longer accept new items.



Upon calling `pop_front()` three times, the queue's state is shown below. The head increased by 3 and the size decreased by 3. (Remember that "`half dollar`" is now at the head of the queue and is "`silver dollar`" is at the tail.)



Whenever `front()` is called, obtain the data at the head of the queue. First check if size is larger than 0 and if so, return the data found at the index indicated by the head data member.

Whenever `back()` is called, obtain the data at the tail of the queue. First check if size is larger than 0 and if so, return the data found at the index indicated by index prior to the tail's current index. This may involve wrapping around as well.

Suppose the prior queue state is modified with `resize(8)`. The new queue state is shown below. The resizing process must allocate a new array, copy items from the old array into the new array, delete the old array, and assign the arr data member to hold the address for the new array. It must also update head and tail values appropriately.