behold the pilgrims, serpentine
like ants that merge in endless line

Queues

Well, not just any old queues. But queues on top of arrays.

In CS2A, you got to implement a stack on top of an array. Here, you get to build a queue like that.

But to keep it challenging, you may not explicitly access the heap or invoke the `new/delete` operators. You can, however, use the built-in `std::vector` class.

Even though `vector` gives you an array that is re-sizeable at run-time, you should not resize your vector except when your queue itself is resized by its user. When your queue is instantiated (or resized), you get one chance to make a (or the) vector just big enough to accommodate exactly as many elements (plus any small fixed overhead you may need).

Ideally, if you haven't yet encountered this problem, stop reading the spec now and try to implement it on your own. If you get stuck, then you can always refer to the section below for tips and insights.[1]

Implementation details

If you got here, then you've probably at least thought about implementing queues the same way as you implemented stacks (with arrays). Maybe you found that it's not as easy. if you found yourself having to shift elements around in your vector every now and then to *make space*, then it's likely your implementation was off. All queueing and dequeuing (sometimes called push and pop) must operate in constant time that is *independent* of the size of the queue. How?

Again, a simple perspective shift will help. Just look at this dependency from the other side. By making the enqueue and dequeue operations dependent on the size of the queue, you can avoid making the queue size dependent on them (which is what you'd have had to do before). How do you code this insight in practice?

Take a break now to think up a solution to this problem.

The answer, if you haven't managed to zero in on it yet, is to treat the array as *circular*. Every element in a circular array has a unique successor and predecessor, including the first and the

[1] Remember: Nobody likes data structures with unpredictable performance. For example, if your enqueue (or dequeue) operations were, like, blindingly fast most of the time, but every once in a while something kicks in to slow it down to a grind, I'll stop using it. And, oh: faster is better than slower, for the same price.

last. The successor of the last element is the first element and the predecessor of the first element is the last element.

You can get this effect by indexing into location `j % array.size()` whenever you want to index into location `j`.

See Figure 1 for a pictorial representation of this idea.

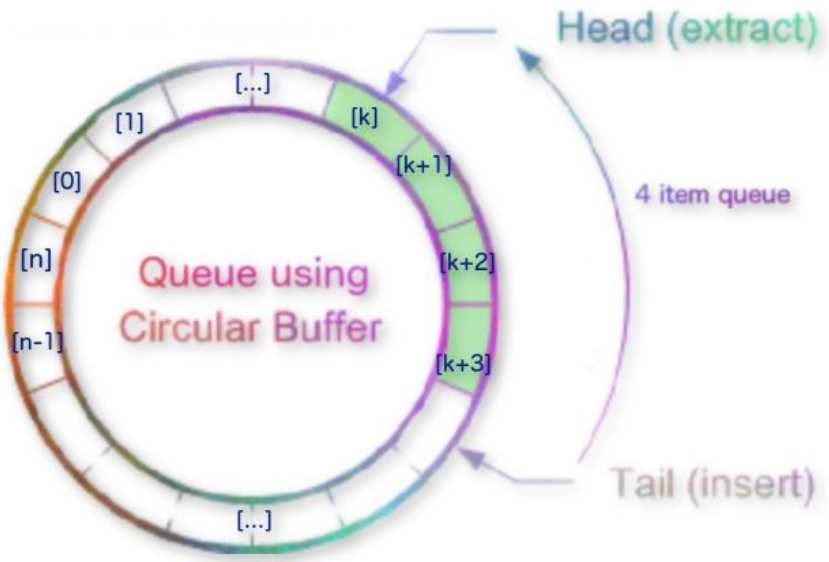
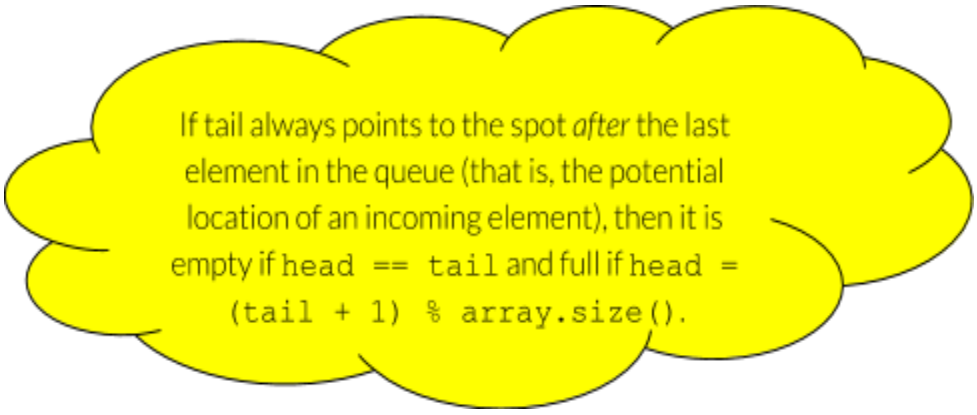


Figure 1: A queue (size n) using a circular array (size n+1). This queue has 4 elements in it, with the first at array index k and the last at array-index k+3.

Nuance: Here is a tip that may save you a great deal of book-keeping pain. To implement a queue of N elements, use an array of N+1 elements. This 1-element overhead will be one of the easiest ways for you to distinguish between an empty queue and a full queue. How?



With these checks, you can disallow insertions if your last element is occupying an index, which, if incremented (circularly) would end up making it point at the head (because then it would be indistinguishable from an empty queue).

Every enqueue and dequeue operation will incur this limit check cost. But can you think of a less expensive way to maintain the queue? Even if you had to use moderately more space?[2]

One final detail before you can get started on the coding: The Queue class you implement must be a template class. If you're not familiar with templating in c++, this would be a good time to hit your reference material (or simply clarify all your doubts in our subreddit). Here's the TL;DR version:

If you created two different Stacks (an int stack and a string stack) in the elephant quest (2A) you might have wondered at the amount of code you duplicated. You were probably already thinking "Gee! I have to copy my `Int_Stack` code into a new file and replace all occurrences of '`int`' with '`string`' ... I wonder if this could be automated..."

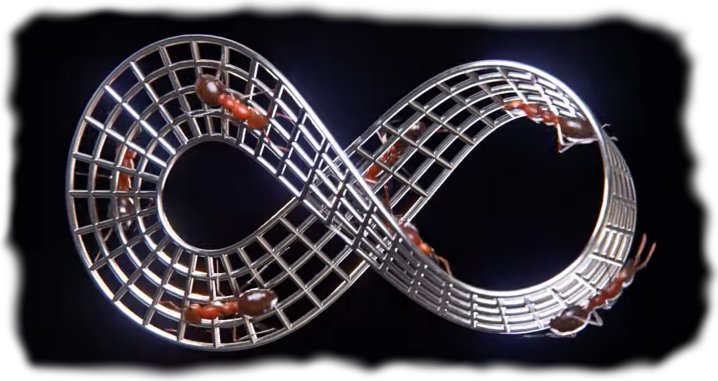
That automation is exactly what templating gives you. Now you could create *template* code for the compiler and tell it - "Here's the template. If someone wants a stack of a particular type, just plug in that type anywhere you see `T` and you'll get their source code".

In our example,

```
class Stack<T> { … }
```

will make the compiler auto-generate a brand new Stack class depending on the value of `T`, the template parameter.

The queue you implement in this quest will be a template class.



[2] The version with the one-element overhead is the one you will have to implement in this quest. This means that the size of the queue (as visible to the outside world through your `.size()` method) is always one less than the size of your backing array, which should contain at the very least your *special* element. Please discuss this in the forums if it isn't clear because I'll know how to revise this paragraph better in the next iteration.

Starter code

Because of the way in which most compilers process template code, you may have to include the entire class implementation within the header file.

Here is the incomplete listing of your Queue class and its implementation (Queue.h).

```
// TODO: Fill in as necessary

template <typename T>
class Queue {
private:
    std::vector<T> _data;
    size_t _head, _tail;
    static T _sentinel;

public:
    static const int MAX_DISP_ELEMS = 100;

    Queue(size_t size);

    static void set_sentinel(const T& elem) { _sentinel = elem; }
    static T get_sentinel() { return _sentinel; }

    bool is_empty() const;
    size_t size() const;
    void resize(size_t size);

    const T& peek() const;
    bool dequeue();
    bool enqueue(const T& elem);

    std::string to_string(size_t limit = MAX_DISP_ELEMS) const;

    friend class Tests; // Don't remove this line
};
template <typename T> T Queue<T>::_sentinel = T();

template <typename T>
Queue<T>::Queue(size_t size) {
    // TODO
}

// TODO - Fill in the missing implementations. Experiment with the aesthetics by moving
// some of these implementations inline (into your class def above). See which gives
// you more readable code.

#endif /* Queue_h */
```

This queue has a sentinel. When `peek()` is invoked on an empty queue, you must return the sentinel instead of throwing an exception.

What about the value of the sentinel? Well, there's no way you can tell what it is, because its type is only known at compile time from your actual template parameter. So you simply provide

a facility to allow the user of your class to set the sentinel to whatever value they deem is illegal in their set of elements.

If I use your Queue to create my own queue of integers, I might instantiate it thus:

```
...

Queue<int> my_queue(100);

Queue<int>::set_sentinel(0);

...

if (my_queue.peek() == 0) {
```

Your first miniquest - Constructor

Implement:

```
template <typename T> Queue<T>::Queue(size_t size);
```

The constructor needs to size the `_data` element correctly, and set initial values for the `_head` and `_tail` members. It doesn't matter that you set them to any specific values. Only that the values are consistent across multiple queue operations.

The constructor is not the place to set the sentinel, which is a static member. In fact, only the user of your Queue class is responsible for setting the sentinel. You don't have to worry about it at all.

Your second miniquest - Enqueue

Implement:

```
template <typename T>
    bool Queue<T>::enqueue(const T& elem);
```

If the queue is not already full, insert a copy of the given element into the end of the queue and return true. Otherwise (if full) return false.

Your third miniquest - Dequeue

Implement:

```
template <typename T> bool Queue<T>::dequeue();
```

If the queue is not empty, remove the (front) element and return true. Otherwise (if empty) return false.

Where is this dequeued (popped) element? It's not returned to you.

That's correct. Please discuss possible reasons why a programmer might choose to make methods like `Stack::pop()` or `Queue::dequeue()` *not* return the removed element.

Your fourth miniquest - Peek

Implement:

```
template <typename T> const T& Queue<T>::peek() const;
```

Return a copy of the front element in the queue (without changing it). Why do we need it? Or... do we need it?

Your fifth miniquest - Is Empty

Implement:

```
template <typename T> bool Queue<T>::is_empty() const;
```

Please check in our subreddit or ask if you don't know what to do here.

Your sixth miniquest - Resize

Implement:

```
template <typename T> void Queue<T>::resize(size_t size);
```

Whoo! Biggie.

I don't know about you, but the best way I've found to resize[3] a queue like this one is to create a brand new queue and dequeue everyone off the old queue and enqueue them in the new queue, in order.

[3] In this spec, size means capacity, not the number of items actually in the queue. A queue of size N can contain a maximum of N items. But it may contain fewer.

See if you can find a cheaper way to do the same thing. Remember that if the old queue has more elements than can fit in the new one, some of the latecomers may have to be booted. Yeah. That's too bad.

Your seventh miniquest -Popalot

Implement the global (non-instance) method:

```
template <typename T>
    void popalot(Queue<T>& q);
```

Thought I'd give you guys another easy freebie. Just implement a global scope template method (not instance or class method - what's the difference?) with the above signature. All it has to do is to clear the queue by emptying it.

Do it however you want. But the queue I give you should be empty when I get it back.

Your eighth miniquest - To string

As usual, `to_string()` is all about attention to detail.

The two examples below hopefully shed enough light on how the queue is to be serialized.[4]

Implement:

```
template <typename T> string Queue<T>::to_string(size_t lim) const;
```

This method should return a string that looks like in Figure 2:

```
# Queue - size = {SIZE reported by your size() method}↲
data : {_data[_head]} {_data[...]} {etc. up to limit}↲
```

Figure 2 - output of to_string(size_t limit)

The enter key character (↲) stands for a single newline. Portions between curly braces (also in red) must be replaced by you. There are spaces where they're obviously there in the above pic. There are none where you wouldn't put them.

[4] Thanks to Paul Hayter, (STEM Center) for help in defining the output format.

If this needs to be reworded, just let me know.

A more concrete example (Figure 3):

```
# Queue - size = 2
data : four five
```

Figure 3: A Queue of size 2, with the items "`four`" and "`five`"

The line that starts with *data* should list the items in the queue. The rest of the line after the colon may be empty. If there are more than `limit` items, you must print the first `limit` items followed by a space and a single token of 3 periods (`...`) in place of everything else.

Finally, note that data, above, denotes the data in the queue from the user's perspective. Not the queue's underlying `_data` element (See Figure 4).

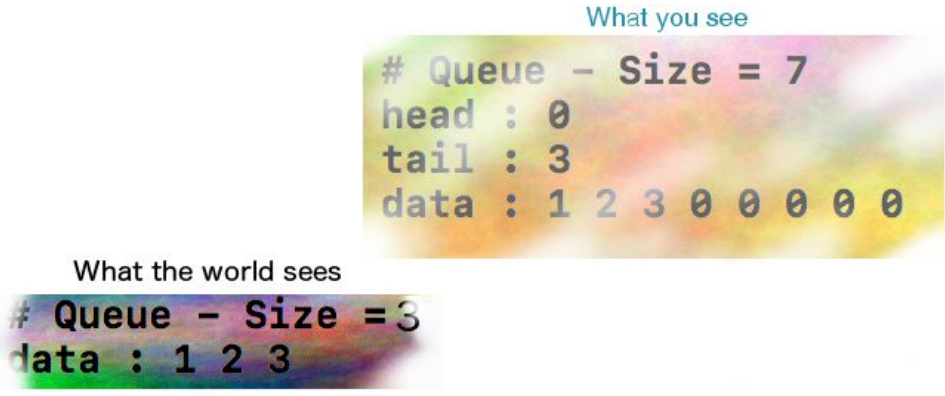


Figure 4 - What the world sees (output of `to_string`), and what you see

Your ninth miniquest - Queue of Objects

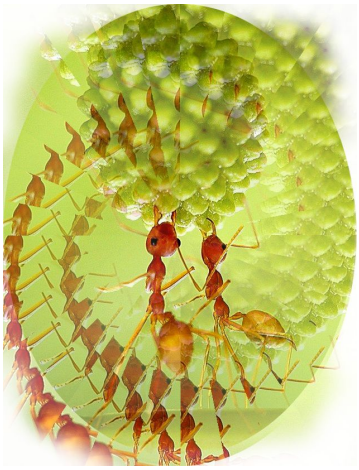
You don't have to do anything special here. All your previous miniquests were with integers. If your templating works as expected, it should succeed on non-integer types as well. I'll test that bit here and you get a freebie reward just for doing things a particular way. Cool!

How long is a well written solution?

Well-written is subjective, of course.

But I think you can code up this entire quest in about 150 lines of clean self-documenting code (about 25-50 chars per line on average), including additional comments where necessary. You can use the provided starter code as one example of acceptable code density.

Submission



When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to https://quests.nonlinearmedia.org
2. Enter the secret code for this quest in the box.
3. Drag and drop your `source` files into the button and press it.
4. Wait for me to complete my tests and report back (usually a minute or less).

Points and Extra Credit Opportunities

Extra credit points await well thought out and helpful discussions on our subreddit.

Happy hacking,

&


photo by Sasikumar
of Vellore