Introduction  
A ring queue is a finite-sized queue that never gets full and never grows past a certain size. Instead, once the queue operates at full capacity, the newest element replace the oldest element.

A simple application of a ring queue consists of keeping track of the most recent values in some stream of numbers, the older values are automatically thrown away when newer ones are added.

For example, one might use a ring queue to get the average of the N most recently processed numbers.

The assignment

For purposes of this assignment, a ring queue will be created with a statement like this:

RingQueue<int,7> rq;

Just like a regular queue, elements can be pushed [on the back of the queue] and popped [off the front]. However, unlike a regular queue:

* A ring queue keeps only the N most recently pushed elements.
* push\_back()and pop\_front() are used instead of push() and pop() to support using generic algorithms and back\_inserter.
* push\_front() on a full ring queue adds the new element and removes the oldest (i.e., the one at the front).
* Iterator access to the queue is supported with begin() and end().

A common way to implement ring queues is as a simple array with two numbers:

* buffer: the array with capacity for N elements (ring capacity),
* begin\_index: an integer that indicates where in the buffer the first element of the ring queue is located, and
* ring\_size: an integer that indicates the number of meaningful elements in the ring queue.

When the ring queue is created, begin\_index is set to 0. The end of the ring queue is calculated as needed via the formula:

**end\_index=(begin\_index+ring\_size)%ring\_capacity**.

Assuming ring\_size is not zero, we can then define the accessor member functions front(), back(), that return buffer[begin\_index], and buffer[end\_index], respectively.

The rules for changing begin\_index and ring\_size (and hence end\_index) are given by:

* pop\_front() increments begin\_index and decrements ring\_size.
* push\_back() stores the value in buffer[end\_index] and increments ring\_size up to the capacity of the ring queue. After capacity is reached, the value is stored in the same place, but  begin\_index is increased.
  + If ring\_size < MAX\_SIZE 🡪 store value in buffer[end\_index] and ring\_size is incremented
  + If ring\_size == MAX\_SIZE 🡪 store value in buffer[end\_index] and begin\_index is increased
* Whenever begin\_index reaches the capacity of the ring, it is reset back to 0. On the other hand, ring\_size is never incremented past the capacity of the ring, nor decremented below 0.
  + Ring\_size is ONLY incremented when the ring\_size < MAX\_SIZE
  + If begin\_index == max\_size, it needs to go back to 0

The table below shows what happens when we add and remove some elements to a 4 element ring queue.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Action | buffer | begin\_index | ring\_size | end\_index |
|  | [ ][ ][ ][ ] | 0 | 0 | 0 |
| push 1 | [1][ ][ ][ ] | 0 | 1 | 1 |
| push 2 | [1][2][ ][ ] | 0 | 2 | 2 |
| pop => 1 | [1][2][ ][ ] | 1 | 1 | 2 |
| push 3 | [1][2][3][ ] | 1 | 2 | 3 |
| push 4 | [1][2][3][4] | 1 | 3 | 0 |
| push 5 | [5][2][3][4] | 1 | 4 | 1 |
| push 6 | [5][6][3][4] | 2 | 4 | 2 |
| pop => 3 | [5][6][3][4] | 3 | 3 | 2 |

Note that no data is ever shifted in the array. A push\_back() when the queue is full simply replaces the oldest element with the newest.

~~The ring queue iterator~~

~~Given the definitions of front() and back() above, you might think that you can define~~

* ~~begin() as the address of buffer[begin\_index]~~
* ~~end() as the address of buffer[end\_index]~~

~~However, notice that in the event that the ring queue is at full capacity, these two functions will return the same address. Clearly then the loop~~

~~// rq defined as above and populated to capacity  
for ( auto it = rq.begin() ; it != rq.end(), ++it ){  
    std::cout << \*it << '\n';  
}~~

~~will produce no output, as the condition rq.begin() != rq.end() is false.~~

~~Therefore, another way is needed to indicate where an element is in the queue. A clear solution is to simply keep track of an offset variable that indicates how far along we are from begin\_index:~~

* ~~A 0 offset indicates the beginning of the queue.~~
* ~~A ring\_size offset indicates the end of the queue.~~
* ~~An offset between 0 and ring\_size indicates that the element is somewhere in the middle of the queue.~~

The ring queue iterator will therefore need two fields:

* a pointer to its parent ring queue, and
* an offset variable.

Each of the operators we need to define becomes simple, at least conceptually:

* operator!=() should return true if two iterators have either different ring queue parents or different offsets.
* operator++() should increment the offset.
* ~~operator\*() should return the array location indexed by (begin\_index + offset) % ring\_capacity.~~

What is the objective of this assignment?

The objective is simple: to guide you through the process of writing a home-made iterator that does not involve delegation. Please be aware that you DO NOT have to start from scratch. Instead you are to provide the missing code in, based on the description provided in this document.

If the provided code is correct, the output of your program should be similar to the one provided at the end of the file.

What to submit?

Once you are done modifying the provided file, upload your version to CCLE. Notice that unlike other assignments, your actual program must be submitted in order for you to receive credit.