Assignment: Queue Class Using Circular Buffer

Learning Outcomes

This assignment will provide you with the knowledge and practice required to develop and implement software involving:

1. Demonstrate ability to design and implement class based abstract data types.
2. Demonstrate ability to understand and implement circular buffer data structure.

What is a Circular Buffer?

A circular buffer is a fixed-size array that behaves as though the first and last element were connected to form a virtual circular memory layout. Such a circular buffer can be treated as a finite-sized queue that never gets full and never grows past a certain size. Instead, new elements replace the oldest elements. This first-in-first-out [FIFO] mechanism lets you keep track of the most recent values in some stream of numbers, automatically throwing away older ones as you add new ones. Its main benefit is that you don't need a large amount of memory to retain data, as older entries are overwritten by newer ones. Circular buffers are used in I/O buffering, bounded logging [when you only want to retain the most recent messages], buffers for asynchronous processing, and many other applications

Like a regular queue, you can apply operation push to add a value on the back and operation pop to remove the value off the front but unlike a regular queue, a circular buffer queue with N elements:

- keeps only the N most recently pushed elements.
- push on a full queue adds the new element and removes the oldest, i.e., the one at the front.

A circular buffer queue with a buffer of N elements can be implemented using two additional values:

- head: an unsigned integer indicating the index in buffer where the front of the queue is.
- size: an unsigned integer indicating how many elements are in the queue.

Initially, head is indexing the first element in buffer and therefore has value 0. The index where a new element will be pushed at the end of the queue - we'll call it tail but it is something that is calculated as needed - is given by the formula $\text{tail} = (\text{head} + \text{size})\%N$.

The rules for changing head and size [and hence tail] are simple but slightly surprising:

- Operation pop increments head and decrements size [assuming a non-empty queue].
- Operation push writes a value in element $\text{buffer}[\text{tail}]$ and increments size; when size reaches N, push continues to store values in the same place $\text{buffer}[\text{tail}]$, but it increments head!!!
- Whenever head reaches N, it resets to 0.
- size is never incremented past N nor decremented below 0.

The following table illustrates what happens when operations push and pop are applied to a circular buffer queue with a buffer of 4 elements. Note that symbol X in the table indicates an unknown or invalid value in the corresponding buffer element.

Operation	buffer	head	size	$\mathrm{tail} = (\mathrm{head}+\mathrm{size})\%4$
	XXXX	0	0	0
$\mathrm{push}(1)$	$\mathrm{1XXX}$	0	1	1
$\mathrm{push}(2)$	$\mathrm{12XX}$	0	2	2
$\mathrm{pop}()$	$\mathrm{X2XX}$	1	1	2
$\mathrm{push}(3)$	$\mathrm{X23X}$	1	2	3
$\mathrm{push}(4)$	$\mathrm{X234}$	1	3	0
$\mathrm{push}(5)$	5234	1	4	1
$\mathrm{push}(6)$	5634	2	4	2
$\mathrm{pop}()$	$\mathrm{56X4}$	3	3	2

Task

Your task is to define a class data structure `Queue` that represents a circular buffer of a fixed size that behaves exactly as described in the previous section. The class you must define should:

- Represent a circular buffer queue that will behave exactly as described in the previous section.
- The queue's buffer must be dynamically allocated and must handle values of type `char`.
- Each object of type `Queue` must encapsulate exactly four values: 8 bytes for the pointer to dynamically allocated memory that will store the circular buffer of `char` values, and 8 bytes each for the three data members representing the dynamically allocated buffer's maximum capacity, the current count of values stored in the queue, and the index of the value at the front of the queue. All of this means that `sizeof(Queue)` must evaluate to exactly 32 bytes.
- Support the following member types for class `Queue`:

```
using value_type      = char;
using reference       = value_type&;
using const_reference = const value_type&;
using pointer         = value_type*;
using const_pointer   = const value_type*;
using size_type       = unsigned long;
```

- Prohibit default construction.
- Support the construction of objects with a specified capacity that will be used to dynamically allocate memory for the circular buffer.
- Support copy construction.
- Support a destructor that returns memory allocated during construction back to the free store.

- Support copy assignment to assign an object of type `Queue` to another object of the same type.
- Provide a member function `c_buff` that will return a pointer to the queue's buffer [such that clients can only examine but not change the buffer's contents].
- Provide member functions for checking the buffer status and capacity: `empty`, `full`, `size`, and `capacity`.
- Provide member function `push` to add a new element [an operation that could potentially overwrite the oldest element in the buffer].
- Provide member function `pop` to remove the oldest element from the buffer.
- Provide member function `front` that is overloaded to provide both accessor and mutator capabilities in relation to the oldest element in the buffer.
- Provide member function `back` that is overloaded to provide both accessor and mutator capabilities in relation to the newest element in the buffer.
- Provide member functions `front_index` and `back_index` that return the indices to the oldest and newest elements in the buffer.
- Support member function `swap` that will swap the current object of type `Queue` with another object of type `Queue`.
- Support a non-member function `swap` that will swap two objects of type `Queue`.

You can assume that clients will not use `Queue` objects incorrectly, that is, call `pop` on an empty queue; or try to create a `Queue` object with capacity 0.

The class must be defined in namespace `HLP3` in header file `queue.hpp` and the member functions must be defined in namespace `HLP3` in source file `queue.cpp`.

> ***If you have questions about the member functions of class*** `Queue` ***or the non-member function*** `swap`***, read the unit tests in the driver source file.***

> ***There should be exactly one*** `#include` ***driver in*** `queue.cpp` ***[to include*** `queue.hpp`***]. There should be no*** `#include` ***directives in*** `queue.hpp` ***- they're not necessary.***

> ***There should be no variables of type*** `short` ***or*** `int` ***or their*** `unsigned` ***equivalents or any equivalent text in either file.***

> ***Member functions of class*** `Queue` ***should not be defined inline in the class definition. Instead, every member function should be defined outside class*** `Queue` ***in source*** `queue.cpp`***.***

Submission Details

Header and source files

You will submit `queue.hpp` and `queue.cpp`.

Compiling, executing, and testing

Download driver source `driver-queue.cpp` with a variety of unit tests and correct output files. Refactor a `Makefile` from previous assignments for use with this assignment.

Valgrind is required

Since your code is interacting directly with physical memory, things can go wrong at the slightest provocation. The range of problems that can arise when writing code dealing with low-level details has been covered in lectures [and a handout that goes into all the ugly details of memory errors and leaks]. You should use Valgrind to detect any potential issues that may lurk under the surface. The online server will execute your submission through Valgrind. If Valgrind detects memory-related problems with your submission, the diagnostic output generated by Valgrind will be added to your program's output causing that output to be different than the correct output.

Documentation

This module will use Doxygen to tag source and header files for generating html-based documentation. Every source and header file *must* begin with *file-level* documentation block. Every function that you declare and define and submit for assessment must contain *function-level documentation*. This documentation should consist of a description of the function, the inputs, and return value. If you're not sure of what to do, contact the instructor.

Submission and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit the source file.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
   - F grade if your submission doesn't compile with the full suite of `g++` options.
   - F grade if your submission doesn't link to create an executable.
   - F grade if Valgrind reports memory errors and/or leaks.
   - A deduction of one letter grade for each missing documentation block in submission. Your submission must have **one** file-level documentation block for each submitted file and a function-level documentation block for each member and non-member function defined in `queue.hpp`. The definitions of these functions in `queue.cpp` need not have function-level documentation blocks [but it is recommended]. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing or incomplete or copy-pasted [with irrelevant information from some previous assessment] block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an A grade and one documentation block is missing, your grade will be later reduced from A to B. Another example: if the automatic grade gave your submission a C grade and the two documentation blocks are missing, your grade will be later reduced from C to E.