CS515 – Lab 3

The purpose of this lab is to reinforce your understanding of C++ classes and objects. You will work with GDB and also practice using assertions to test your program. You will debug a queue ADT and also complete the implementation of an integer stack ADT.

Note: the tasks in this lab can be used as the basis for assignment 4P—You'll be at a disadvantage if you skip it!

Download all files from `~cs515/public/3L`

Task 1: **Debug a queue.**

Starting code: `queue.h`, `queue.cpp` and `queuetest.cpp`

The `queue.h` file contains the declaration of the queue class and the `queue.cpp` file provides an implementation of the class using a fixed sized array. To test the implementation, we have a test driver called `queuetest.cpp`. In the test file, assertions are used to test the queue implementation.

In C/C++, **assertions** are implemented with the standard **`assert`** macro. The argument to assert must be true when the macro is executed; otherwise the program aborts and prints an error message. For example, the assertion

```
assert( size <= LIMIT );
```

will abort the program if `size` is greater than LIMIT, and print an error message like this:

```
Assertion failed:(size <= LIMIT),function main,file test.cpp,line 17.
```

To compile the queue class, use

```
% g++ -c queue.cpp
```

The above command with **`-c`** option only compiles the `queue.cpp` file and creates the object file named **`queue.o`**. We don't want to create an executable file (`a.out`) since there is no main().

To compile and run the queue class with the test driver `queuetest.cpp` (which has main()), use

```
% g++ queuetest.cpp queue.o -o queuetest
```

The above command compiles the test driver `queuetest.cpp` with `queue.o` and creates the executable `queuetest`. You can combine the above two steps and compile two source files together:

```
% g++ queuetest.cpp queue.cpp -o queuetest
```

Either way, you can run the test as:

```
% ./queuetest
```

Though the program compiles fine, it aborts due to assertion failures when you run the `queuetest` executable. ***Your task is to use the `gdb` debugger to debug the program and fix the issues.***

Since you are compiling multiple source files together, use the following syntax to set a breakpoint in a specific source file (this command will set a breakpoint at line 20 in file `queue.cpp`).

```
(gdb) break queue.cpp:20
```

In this task, you may only modify `queue.cpp` to fix the bug. This file contains the implementation of a queue using a circular array (a good explanation of circular arrays and how to use the modulus operator % can be found here: https://towardsdatascience.com/circular-queue-or-ring-buffer-92c7b0193326). You must **not** change `queue.h` or `queuetest.cpp`. You don't need to implement an expandable version of the queue for this lab. We assume the queue size is fixed and will not exceed the preset limit.

Once you fix the problem(s), the `queuetest` program should run without any errors. You also need to make sure both of your `dequeue` and `enqueue` methods run in O(1) time (constant time).

Task 2: Implement a stack

Starting code: `stack.h`, `stacktest.cpp`

You are given the header file **`stack.h`**. Write the implementation of the stack in source file **`stack.cpp`**. You must not modify the header file. A test driver `stacktest.cpp` is provided to test your implementation.

The stack to be implemented uses an underlying dynamic array to store all values. The stack's capacity is provided as a constructor parameter or set to a default size (if no size is provided). When a stack is created, there will be a dynamic array created based on this given size. If the number of elements on a stack reaches its capacity, an exception is thrown if additional push operation is requested. **You don't need to implement an expandable version for this lab.**

The public interface of the stack class is specified in **`stack.h`**. Note that since the stack uses dynamic memory, a constructor and the Big-5 methods must also be implemented.

- a **default constructor** that creates a stack of a specified size (with a default size of 8);
- a **copy constructor** to ensure deep copy of objects;
- a **destructor** to avoid memory leaks;
- an **assignment operator** to ensure deep copy and avoid memory leak during assignment;
- a **move constructor** and a **move assignment operator** to speed up the copying of objects that are about to be destroyed anyway by "stealing" their dynamic memory.

You are not allowed to use any STL containers in your implementation.

To compile the stack class use

```
% g++ -c stack.cpp
```

To compile the stack class with the test driver and name the executable `stacktest`, use

```
% g++ stacktest.cpp stack.cpp -o stacktest
```

To run the test:

```
% ./stacktest
```

The exceptions are already defined in the `stack.h` header file. Here's a usage example:

```
if(_tos == 0)
    throw EmptyStackException();
```

If your program has passed all the test cases, the driver will output:

```
exception caught
exception caught
exception caught
```

If any of the test cases fail, you will get a message from the corresponding assert statement and the program will abort. You must also run the program with Valgrind and make sure that there are no memory errors.

NOTE: Your submissions will be tested on the files named `queuetestX.cpp` and `stacktestY.cpp`, where `X` is a number from 1 to 4, and `Y` is from 1 to 5. We will also be running Valgrind on two of these tests. Once you get your programs working with the basic `queuetest.cpp` and `stacktest.cpp` files, you should try compiling against these other files and run your own tests with them.

Submission:

You may change the provided `makefile` locally, but we will be testing with our own—so be sure your `makefile` has something like the following lines to ensure proper compilation with our grading test cases.

```
All: stack.o queue.o
stack.o: stack.cpp stack.h
	$(CXX) -c stack.cpp -o stack.o
queue.o: queue.cpp queue.h
	$(CXX) -c queue.cpp -o queue.o
```

Submit two source files **`stack.cpp`**, **`queue.cpp`**, two header files **`stack.h`, `queue.h`**, and your **`README`**. To submit the files in Mimir, follow the link for this lab in MyCourses. Here is a list of files you are expected to submit:

`stack.cpp queue.cpp stack.h queue.h README`

As always, do not turn in executables or object code, and make sure your submission compiles successfully on Agate. Programs that fail to compile will receive a zero mark (even if it might work perfectly on your home computer.) To check this, use the make command with the provided makefile, and check that your submission passes tests on Mimir.

You must include the following in the comment block of each of your source files.

```
/**    CS515 Lab X
       File: XXX.cpp
       Name: Your Name
       Section: X
       Date: XXX
       Collaboration Declaration:
       Lab Partner: <partner's name from lab session number> or <none>
       Collaboration: <anyone who may have helped that was not the primary partner>
*/
```

You should also include detailed collaboration declaration information in your comments. If you worked in pairs in this lab, each of you must include the partner's name in your program comment. Both you and your partner must complete an individual submission in order to earn a grade. If you work by yourself, you must indicate in the program comments that you have worked on your own independently.

You can resubmit as needed—your last submission will be graded. Here is a tentative grading scheme:

queue test run 1	10
queue test run 2	10
queue test run 3	10
queue test run 4	10
stack test run 1	10
stack test run 2	10
stack test run 3	10
stack test run 4	10
stack test run 5	10
stack Valgrind checks	10