COMP220 — Lab 9 & Homework 7

Fall 2016

Abstract

In this lab you'll work on parts of the array based Queue implementation. In doing so you'll cut your teeth on the basic use of Valgrind's memcheck feature for detecting memory leaks. Submit what you have done at the end of lab as assignment *lab9* then complete the remainder and submit it as assignment *hwk7*.

We'll work on the remaining Queue methods in the following order. Be very mindful of this order when writing tests. Because we're putting assignment and copy construction last, you shouldn't use those in writing tests for methods earlier on the list. Properly testing the output and equality operators will require push_back, but you should at least get them working on the empty queue then come back and verify they continue to work on non-empty queues. We're doing them first to play nice with gTests. In implementing these methods, don't hesitate to write private helper methods and procedures for work with arrays that occurs in multiple Queue methods.

1. operator<<
   Print the queue contents on a single line as comma separated values surrounded by parenthesis, i.e. $(1, 3, 5)$. No newline at the end. No comma after the final value. There are a fair amount of hoops to jump to get this to work with templates. You just need to fill in the implementation and write some tests. The hoops will be explained Friday.
2. operator==
   This should be true if the two Queues will behave the same. This allows for equality between queues with different capacity and different locality of data within the array.
3. push_back
   If adding the item fills the array, then double the capacity.
4. pop_front If the capacity is larger than the initial capacity of a default queue and removing the item causes the array to be at 30% capacity then reduce capacity by half.
5. copy constructor
6. operator=

Work through the list one method at a time. For each method you should first write documentation, then tests, then work out the implementation. Play around with using different contained types in your tests so that you don't get too attached to one particular type.

1 Using Valgrind to find Leaks

Valgrind provides a suite of Linux tools that let you profile various parts of your program. The *memcheck* tool runs your code and checks for leaks. It requires a running executable as a memory leak is a runtime problem and not a static, compile time problem. We'll typically run it along side our gTests.

Code::Blocks has a Valgrind plugin that makes running memcheck dead simple. In the menu bar you'll find a Valgrind option and in that menu you'll see an option for *Run Memcheck*. Simply select this and the current build will be run through Valgrind. If your program leaks memory, then a message window will show up in Code::Blocks with some information about the nature of the leak. To test this, simply comment out the delete line in the destructor and run the core tests again. The program will leak because the Queue destructor doesn't delete the array when the Queue is cleared from the runtime stack.

Now that we're using heap memory, you need to not only design and run tests for behavioral correctness but run tests that when run through memcheck will verify the absence of memory leaks in your code.