COMP 220	**Data Structures**
LAB/HWK 9/8	
Assigned: November 08	*Due: November 15*

Abstract

In this lab you'll improve upon the limits of basic C++ arrays by implementing your own `IntArray` class. This comes from problems 4 and 5 from chapter 12. In doing so you'll cut your teeth on the basic use of Valgrind's memcheck feature for detecting memory leaks. In the homework you'll extend this class into a basic `IntVector` class using the array-doubling idea discussed in class.

Lab 9

The arrays built in to C++ have two flaws: they don't keep track of the size and they don't check that index values are inside the array bounds. We now know enough to implement our own special array class that fixes these. Implement your own class IntArray contains integers and implement the following methods

1. Default constructor
2. A constructor `IntArray(n)` that creates an IntArray with n integers, each initialized to 0.
3. A destructor that frees the heap storage used by IntArray.
4. A `size()` method
5. A `get(k)` method that returns the integer at position `k`. If `k` is outside the bounds of the allocated array, throw an appropriate exception.
6. `put(k, value)` method that assigns value to the element at position `k`. Again, throw an appropriate exception if `k` is out of bounds.

As usual, make sure you have appropriate tests and documentation. However, now that we're using heap memory we're going to supplement our tests with a tool that checks for *memory leaks*.

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, since memory leaks are a runtime problem, 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 [1], 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 destructor doesn't delete the array when the object 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.

Submit what you have done at the end of lab as assignment *lab9*.

[1] Actually, Valgrind can't find *all* memory errors, but in general it does a very good job.

Homework 8

For the homework, extend this `intArray` into `intVector` by adding the following methods:

1. operator<<
   Print the vector contents on a single line as comma-separated values surrounded by parentheses, i.e. $(1, 3, 5)$. No newline at the end. No comma after the final value.

2. operator==
   This should be true if the two vectors have the same values. Note that the current capacity does not matter for this.

3. push_back
   If adding the item fills the array, then double the capacity and copy the elements over, just like we have seen for a stack.

4. copy constructor

5. operator=

Again, add appropriate tests and documentation, and make sure *memcheck* doesn't detect any leaks.

Submit your completed files as assignment *hwk8* by the start of next lab.

Bonus

Implement any (or all) of the following methods for bonus points:

1. insert(k, value)
   Insert value at position `k`, moving the later elements (`k` through `size`) one position to the right. Resize the array as necessary, and throw an exception for invalid `k`.

2. pop_back
   Remove and return the latest integer (like a stack). 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.

3. int& operator[](int k)
   This is overriding brackets so you can use your class just like an actual C++ vector or array. Unlike the C++ classes, however, you should check for out-of-bounds indices and throw an appropriate error.