## 1 Instructions

When you are finished submit all your work through the MyClasses page for this class. Create a directory called Lab08, put each programming exercise into its own subdirectory of this directory, zip the entire Lab08 directory up into the file Lab08.zip, and then submit this zip file to Lab #8.

Make sure that you:

- Follow the coding and documentation standards for the course as published in the MyClasses page for the class.
- Check the contents of the zip file before uploading it. Make sure all the files are included.
- Make sure that the file was submitted correctly to MyCLasses.

All non-templated class structures are to have their own guarded specification file (.h) and implementation file (.cpp) that has the same name as the class. No inline coding in the (.h) files.

All templated class structures are to have their own guarded specification/implementation file (.h) that has the same name as the class. As with non-templated class structures, no inline coding in the specification.

In addition, you must create a make file that compiles and links the project on a Linux computer with a Debian or Debian branch flavor.

## 2 Exercise

This exercise is to construct a templated priority queue using the STL vector class as the underlying storage structure. The declaration of the class is below. Complete the implementation of this class. The priority of a node will be an integer type and we will use the convention that the higher number will represent the higher priority, so priority 3 will be higher than priority 1.

```
template <class T> class PQNode {
public:
  T data;
  int priority;

  PQNode(T Data) {
    data = Data;
    priority = 1;
  }

  PQNode(T Data, int Priority) {
    data = Data;
    if (Priority < 1)
```

```
      Priority = 1;

    priority = Priority;
  }
};

template <class T> class PriorityQueue {
private:
  vector<PQNode<T>> queue;

public:
  // Constructors and Destructor
  PriorityQueue();
  PriorityQueue(const PriorityQueue &obj);
  ~PriorityQueue();

  // Acessors and Mutators
  void enqueue(T Data, int Priority = 1);
  T dequeue();

  // Other Functions
  void print();
  bool isEmpty();
  void clear();
  int size();
};
```

The way a priority queue works is just like a queue, first in first out, but the items with a higher priority are dequeued first, in order. So if there are items in the queue with priorities 1, 2, and 3, then those of priority 3 are dequeued first, in the order they are in the queue. Then we dequeue those of priority 2, in order, and finally those with priority 1, in order.

- enqueue(T Data, int Priority = 1) — Creates a queue node with the default priority of 1 and loads in the Data parameter to the data of the node. Finally, pushes the node onto the back of the queue.
- dequeue() — removes the first element of the highest priority from the queue and returns the data value. If the queue is empty the templated element's default value is returned. Not the greatest way to handle the situation of an empty queue, in practice we would implement an exception structure for this class to use.
- print() — Prints the contents of the queue to the screen, each item is given its own line and both the data contents and priority are printed.
- isEmpty() — Returns true if the queue is empty and false otherwise.
- clear() — Removes all contents of the queue.
- size() — Returns the current number of elements in the queue.

As a test, if you create and run the following program,

```
#include <iostream>
#include <vector>

#include "PriorityQueue.h"

using namespace std;

void println(string s = "") { cout << s << endl; }

int main() {
  PriorityQueue<int> queue;

  queue.enqueue(7);
  queue.enqueue(5);
  queue.enqueue(15, 3);

  queue.print();

  int t = queue.dequeue();
  cout << t << endl;
  queue.print();

  t = queue.dequeue();
  cout << t << endl;
  queue.print();

  t = queue.dequeue();
  cout << t << endl;
  queue.print();

  t = queue.dequeue(); //  Too Far
  cout << t << endl;
  queue.print();

  for (int i = 0; i < 10; i++)
    queue.enqueue(i, i);

  queue.print();

  cout << queue.size() << endl;

  queue.clear();

  cout << queue.size() << endl;

  for (int i = 0; i < 10; i++)
    queue.enqueue(i, i % 3 + 1);

  queue.print();

  while (!queue.isEmpty()) {
    t = queue.dequeue();
    cout << t << " ";
  }
  println();

  PriorityQueue<string> Squeue;

  Squeue.enqueue("Sam", 2);
  Squeue.enqueue("John");
  Squeue.enqueue("Jack");

  string strarr[] = {"considerable", "substantial",   "pronounced",
                     "significant",  "appreciable",   "serious",
                     "exceptional",  "extraordinary", "tremendous",
                     "stupendous",   "unlimited",     "boundless",
                     "cosmic"};

  for (int i = 0; i < 13; i++) {
    Squeue.enqueue(strarr[i], i % 3 + 1);
  }

  Squeue.print();
  println();

  while (!Squeue.isEmpty()) {
    string c = Squeue.dequeue();
    cout << c << " // ";
  }
  println();

  return 0;
}
```

You should get the following output.

```
7  ---  1
5  ---  1
15  ---  3
15
7  ---  1
5  ---  1
7
5  ---  1
5
22003
0  ---  1
1  ---  1
2  ---  2
3  ---  3
4  ---  4
5  ---  5
6  ---  6
7  ---  7
8  ---  8
9  ---  9
10
0
0  ---  1
1  ---  2
2  ---  3
3  ---  1
4  ---  2
5  ---  3
6  ---  1
7  ---  2
8  ---  3
9  ---  1
```

```
2 5 8 1 4 7 0 3 6 9
Sam  ---  2
John  ---  1
Jack  ---  1
considerable  ---  1
substantial  ---  2
pronounced  ---  3
significant  ---  1
appreciable  ---  2
serious  ---  3
exceptional  ---  1
extraordinary  ---  2
tremendous  ---  3
stupendous  ---  1
unlimited  ---  2
boundless  ---  3
cosmic  ---  1

pronounced // serious // tremendous // boundless // Sam // substantial //
    appreciable // extraordinary // unlimited // John // Jack // considerable
    // significant // exceptional // stupendous // cosmic //
```

---

Just as a note, the STL does have a priority queue structure but we will not be using it for this assignment. Our implementation using a vector will not be as efficient as the STL version since the STL stores the underlying data in a structure called a *heap*. This is a data structure not heap memory. A heap is a special binary tree that is stored as an array (or list) structure that makes the enqueue and dequeue operations more efficient. On the other hand, the heap structure will not guarantee that the extraction of data with the same priority will be in the order of input, our implementation will.