CMSC 202 Fall 2019

Project 5 – UMBC Scheduler

**Assignment:** Project 5 – UMBC Schedule

**Due Date:** Wednesday, November 27, 2019 at 8:59pm

**Value:** 80 points

# Overview

In this project, you will:

* Practice basic C++ syntax including branching structures
* Write classes and instantiate those classes using a constructor
* Create a templated data structure (queue or stack)
* Use simple file input
* Use overloaded operators to access templated data structure

# Background

Many of you have had the opportunity to register for courses for next semester already. As such, you were probably able to navigate the sometimes-frustrating process of finding a course only to have it cruelly ripped out from under you when the course was full. What if you had the ability to be added to the waitlist and then when the waitlist was full, a new section of the class was offered automatically?! What a magical place that would be.

For this project, we will be building a scheduler that takes in a list of student requests and will generate the required courses to meet the demand for those students. In order to do this, we will be building a templated data structure that works like a queue where the juniors and seniors would be inserted into the front of the structure and the other students would be added behind them. We will be building this structure using a dynamically allocated array.

# Assignment Description

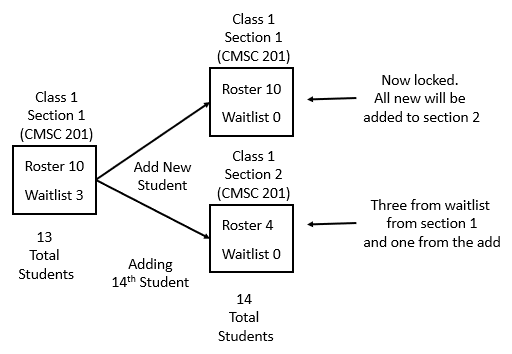
**Class 1 – Tqueue** – This is a very important class. It is used to manage each Classes roster and waitlist. It is templated so that it could store the information about whatever it is designed to hold (in this case Student pointers). As a dynamically allocated array, the Tqueue has a maximum capacity (which is passed to it) You will need to implement normal queue functions such as Enqueue (which inserts in the end), Dequeue (which removes from front), Sort (which sorts the content in the Tqueue – NOTE: For the sort function, you may need to write it specially to use pointers), capacity functions (including IsEmpty, IsFull, and Size), an overloaded[] operator (to get specific location in the Tqueue), and ClearData (which removes all data from the Tqueue). Finally, you must implement the copy constructor and assignment operator in this class. There should be absolutely **NO** references to anything about classes or students in Tqueue!

**You should implement this class by itself and then test it completely before using it. There is sample test code at the bottom of Tqueue.cpp. You can uncomment it as you code functions to test it incrementally.** Do not forget how we must implement templated classes!

**Class 2: Class –** For this project, a Class represents a specific offering of a course. A Class object is made up of a name (like CMSC 201), a section (integer starting at 1), and two Tqueues (one named m\_roster and one named m\_waitlist). The m\_roster represents the number of students in the class (starts capped at 10 per class). The m\_waitlist represents the students waiting for the class (starts capped at 3 per class). When a class’s roster and waitlist are both full, a new section of that class is opened and everyone in the waitlist is moved to the new class. Otherwise, read the function descriptions for more details in Class.h.

Once a class’s waitlist is full, a new section of that class is created, and the students from the waitlist are moved to the new class’s roster, the original class is closed. The waitlist from the first section will never be repopulated.

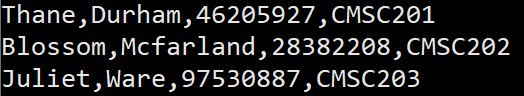
For example, if class 1 – section 1 has 10 students on the roster and 3 on the waitlist when we add another student to class 1, a new class (section 2) would be created and the three from the waitlist and the new student would all four be moved to the roster of section 2. The figure below describes how it should work:



**Class 3: Scheduler –** This is the class that loads the file of students and loads students into specific classes. It is called directly from driver.cpp. The main menu manages the scheduling tool and allows the user to display all classes (and sections), rosters for specific sections, search for a specific string (searches first and last names) and sorts all rosters. Scheduler does most of the heavy lifting of the waitlist mechanics listed above as it loads a file. For more details, check out Scheduler.h.

**Class 4: Student** – This is the class that contains all the information about each student. Each student has a first name, last name, and student ID. There is an overloaded << operator and a < operator (for sorting).

The input files are formatted with first name, last name, student id, and the course.



There are four test files (proj5\_test1.txt, proj5\_test2.txt, proj5\_test3.txt, and proj5\_test4.txt) each increasingly complicated (and large). The first test file is named proj5\_test1.txt and has the student’s names be numbers such as One One or Six Six. This is to help with debugging the **m\_roster** to **m\_waitlist** process described above.

# Requirements:

This is a list of the requirements of this application. For you to earn all the points you will need to meet all the defined requirements.

* You must follow the coding standard as defined in the CMSC 202 coding standards (found on Blackboard under course materials). This includes comments as required.
* The project must be turned in on time by the deadline listed above.
* The project must be completed in C++. You may not use any libraries or data structures that we have not learned in class. Libraries we have learned include **<iostream>, <fstream>, <iomanip>, <vector>, <cmath>, <ctime>, <cstdlib>, <sstream>,** and **<string>**. You should only use **namespace std**.
* Using the provided files, **Tqueue.cpp, Student.h, Class.h, Scheduler.h, makefile, proj5\_test1-5.txt and the driver.cpp file** write the program as described. (Finish **Tqueue** first though!) You must use a **Tqueue** to build a **Class** which is done in **Scheduler.h.**
* As a reminder, **Tqueue.h** is templated and all functions must exist in ONE file (**Tqueue.cpp**). **Class** uses **Tqueues** to manage the roster and waitlist but could easily be modified to use practically any type of data. **Tqueue** must include a functioning **copy** constructor and **assignment** operator.
* You can copy the files from my directory in **/afs/umbc.edu/users/j/d/jdixon/pub/cs202/proj5**.
* Class member variables must be **private** for project 5.
* All user input must be validated. For example, if a menu allows for 1, 2, or 3 to be entered and the user enters a 4, it will re-prompt the user. However, the user is expected to always enter the correct data type. i.e. If the user is asked to enter an integer, they will. If they are asked to enter a character, they will. You do not need to worry about checking for correct data types.
* The code must not have memory leaks.

# Recommendations

* We will test your **Tqueue** file separately from your application (as well as part of it). It must be fully implemented with a destructor, copy constructor, and an assignment operator. Test it first using something simple like ints.
* After the **Tqueue** works (including all tests), write **Student** (it is relatively easy). **Tqueue** is tricky though.
* After **Tqueue** and **Student** are written, start **Class**. **Class** isn’t too bad because it calls a lot of the functions from **Tqueue**.
* The **Scheduler** will load the information about each **student** from a file into sections of the **Classes**. The **AddStudent**, **FindClass**, and **LoadFile** functions are challenging but if you implement them without the waitlist functionality, it may be easier to test it and then come back and add it later. The **SearchStudent** and **SortRoster** functions can be implemented last as nothing else relies on these functions.

# Sample Input and Output

For this project, there are no input files. Command line arguments have been included (as well as the **makefile**) so you can use **make run1** or **make val1** (all 4 have been included: **make run1**, **make run2**, **make run3**, and **make run4** including their val counterparts) to test your code.

|  |
| --- |
| ./proj5 proj5\_test1.txt  Loading file: proj5\_test1.txt  Welcome to the UMBC Scheduler  1. Display All Courses  2. Display a Specific Course  3. Search for Specific Student  4. Sort Roster  5. Quit  1  Course: CMSC201 Section: 1  Course: CMSC201 Section: 2  1. Display All Courses  2. Display a Specific Course  3. Search for Specific Student  4. Sort Roster  5. Quit |

Below is a sample run where there are 21 students loaded. In this case, section 1 should have 10 students, section 2 should have 10 students with one of the waitlist.

|  |
| --- |
| 1. Display All Courses  2. Display a Specific Course  3. Search for Specific Student  4. Sort Roster  5. Quit  2  Which course would you like to display?  1. Course: CMSC201 Section: 1  2. Course: CMSC201 Section: 2  1  Displaying Student  Course: CMSC201 Section: 1  Enrolled (10 enrolled)  One, One  Two, Two  Three, Three  Four, Four  Five, Five  Six, Six  Seven, Seven  Eight, Eight  Nine, Nine  Ten, Ten  1. Display All Courses  2. Display a Specific Course  3. Search for Specific Student  4. Sort Roster  5. Quit  2  Which course would you like to display?  1. Course: CMSC201 Section: 1  2. Course: CMSC201 Section: 2  2  Displaying Student  Course: CMSC201 Section: 2  Enrolled (10 enrolled)  Eleven, Eleven  Twelve, Twelve  Thirteen, Thirteen  Fourteen, Fourteen  Fifteen, Fifteen  Sixteen, Sixteen  Seventeen, Seventeen  Eighteen, Eighteen  Nineteen, Nineteen  Twenty, Twenty  Waitlisted (1 waitlisted)  Twenty-one, Twenty-one  1. Display All Courses  2. Display a Specific Course  3. Search for Specific Student  4. Sort Roster  5. Quit |

If we sorted the roster then our students would now be out of order but they would be alphabetized by last name.

|  |
| --- |
| 1. Display All Courses  2. Display a Specific Course  3. Search for Specific Student  4. Sort Roster  5. Quit  4  1. Display All Courses  2. Display a Specific Course  3. Search for Specific Student  4. Sort Roster  5. Quit  2  Which course would you like to display?  1. Course: CMSC201 Section: 1  2. Course: CMSC201 Section: 2  1  Displaying Student  Course: CMSC201 Section: 1  Enrolled (10 enrolled)  Eight, Eight  Five, Five  Four, Four  Nine, Nine  One, One  Seven, Seven  Six, Six  Ten, Ten  Three, Three  Two, Two |

We could then search for a specific student (either first name or last name). It is a string comparison for either.

|  |
| --- |
| 1. Display All Courses  2. Display a Specific Course  3. Search for Specific Student  4. Sort Roster  5. Quit  3  What name do you want to search for?  Five  Items with Five in them:  Course: CMSC201 Section: 1  1. Display All Courses  2. Display a Specific Course  3. Search for Specific Student  4. Sort Roster  5. Quit |

If you were to execute **make run4** then you would be testing a file with more than 2000 entries. Each of the test input files is focused on something different. Make sure to test each of them and when in doubt, create your own test input.

If you wanted to make sure that duplicates work (which they should) then you could copy and paste some additional lines at the end of **proj5\_test4.txt**. In this case, there are many sections of each class. Here is a quick screenshot of the list:

|  |
| --- |
| Course: CMSC304 Section: 20  Course: CMSC331 Section: 19  Course: CMSC203 Section: 23  Course: CMSC341 Section: 22  Course: CMSC421 Section: 21  Course: CMSC313 Section: 22  Course: CMSC201 Section: 19 |

# Compiling and Running

You will need to implement the **makefile** for this project.

Once you have compiled using the **makefile**, enter the command **make run** or **./proj5**  to run your program. If your executable is not proj5, you will lose points. It should look like the sample output provided above.

Because we are using dynamic memory for this project, you are required to manage any memory leaks that might be created. Anything that you use “new” for needs to be deleted. Remember, in general, for each item that is dynamically created, it should be deleted using a destructor.

One way to test to make sure that you have successfully removed any of the memory leaks is to use the **valgrind** command.

Since this project makes extensive use of dynamic memory, it is important that you test your program for memory leaks using **valgrind**:

**valgrind ./proj5 proj5\_test1.txt**

Note: If you accidently use valgrind make run, you may end up with some memory that is still reachable. Do not test this – test using the command above where you include the input file.

If you have no memory leaks, you should see output like the following:

|  |
| --- |
| ==5606==  ==5606== HEAP SUMMARY:  ==5606== in use at exit: 0 bytes in 0 blocks  ==5606== total heap usage: 87 allocs, 87 frees, 10,684 bytes allocated  ==5606==  ==5606== All heap blocks were freed -- no leaks are possible  ==5606==  ==5606== For counts of detected and suppressed errors, rerun with: -v  ==5606== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6) |

The important part is “in use at exit: 0 bytes 0 blocks,” which tells me all the dynamic memory was deleted before the program exited. If you see anything other than "0 bytes 0 blocks" there is probably an error in one of your destructors. We will evaluate this as part of the grading for this project.

Additional information on **valgrind** can be found here: <http://valgrind.org/docs/manual/quick-start.html>

# Completing your Project

When you have completed your project, you can copy it into the submission folder. You can copy your files into the submission folder as many times as you like (before the due date). We will only grade what is in your submission folder.

For this project, you should submit all files to the **proj5** subdirectory:

**Student.h, Student.cpp**

**Class.h, Class.cpp**

**Scheduler.h, Scheduler.cpp**

**Tqueue.cpp**

**proj5.cpp**

For this project, you are allowed to edit **Tqueue.cpp** but no other header (.h) files.

As you should have already set up your symbolic link for this class, you can just copy your files listed above to the submission folder.

* 1. cd to your project 5 folder. An example might be cd **~/202/projects/proj5**
  2. **cp Student.h Student.cpp Class.h Class.cpp Scheduler.h Scheduler.cpp Tqueue.cpp proj5.cpp ~/cs202proj/proj5**

You can check to make sure that your files were successfully copied over to the submission directory by entering the command. Please note: You are responsible for turning in all required files.

ls ~/cs202proj/proj5

You can check that your program compiles and runs in the **proj5** directory, but please clean up any **.o** and executable files. Again, do not develop your code in this directory and you should not have the only copy of your program here. Uploading of any **.gch** files will result in a severe penalty.

For additional information about project submissions, there is a more complete document available in Blackboard under “Course Materials” and “Project Submission.”

**IMPORTANT:** If you want to submit the project late (after the due date), you will need to copy your files to the appropriate late folder. If you can no longer copy the files into the proj5 folder, it is because the due date has passed. You should be able to see your proj5 files but you can no longer edit or copy the files in to your proj5 folder. (They will be read only)

* If it is 0-24 hours late, copy your files to **~/cs202proj/proj5-late1**
* If it is 24-48 hours late, copy your files to **~/cs202proj/proj5-late2**
* If it is after 48 hours late, it is too late to be submitted.