

Project 06 : Class Registration system

(doc v1.1)

Assignment: C++ program for a simple class reservation system

Evaluation: Gradescope followed by manual review

Policy: Individual work only

Complete By:

Part 01 (70pts): Friday April 30th @ 11:59pm CDT, late day Sat 5/1 @ 11:59pm

Part 02 (30pts): Friday April 30th @ 11:59pm CDT, late day Sat 5/1 @ 11:59pm

Pre-requisites: HW 14 (part 01), HW 15 (part 02). Project [overview](#) (4/19 class).

Overview: an object-oriented class registration system

In project 05 you built a set of libraries to implement linked-lists and priority queues. Here in project 06 you are going to build upon those concepts to implement a simple class registration system for CS classes. Here's a look at the user-interface, which inputs commands from the user to enroll students into CS classes. This represents part 01:

```
**CS Enrollment System**
Enter enrollments filename>
enrollments.txt

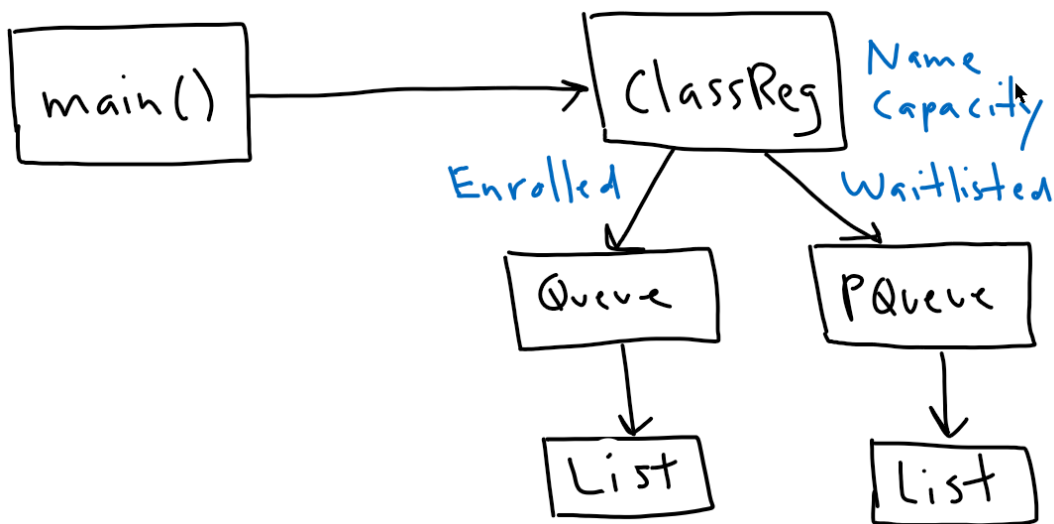
Enter a command (help for more info, quit to stop)>
help
stats
list class
increase class capacity
enroll class netid
waitlist class netid priority
process filename
output filename
quit

Enter a command (help for more info, quit to stop)>
list cs141
cs141
Capacity: 280
Enrolled (5): amalla5 hbrahm3 jkarim5 jmanus2 xlin48
Waitlisted (10): aalawi5 (10) zlnu3 (10) cmei25 (12) hali31 (20) aevdok3 (20) amacat4 (99) dpate320 (100)
yandry2 (110) ldmoy2 (112) mtened2 (112)

Enter a command (help for more info, quit to stop)>
```

The registration system inputs registration data from a file, and supports 9 commands: **help**, **stats**, **list**, **increase**, **enroll**, **waitlist**, **process**, **output**, and **quit**. What makes this program different from earlier projects is that we're going to use classes and object-oriented programming to implement the solution. In particular, the

program is going to be written using a set of 4 classes: **ClassReg**, **Queue**, **PQueue**, and **List**. Here's a visual diagram of the relationship:



In **part 01** you'll focus on writing the `main()` program in "main.cpp"; the classes will be provided to you as compiled .o files. In **part 02** you'll write the `List` class, which implements a one-way linked-list much like you did in project 05. Note that parts 01 and 02 are due on the same day --- Friday April 30th. The parts are independent and can be solved in either order; part 01 is worth more, but part 02 is more familiar since it's based on HW 15 and project 05.

Part 01: a program for class registration

The program inputs enrollment data from an input file specified when the program starts (step 1). If the file does not exist, an error message is output and the program ends.

```
**CS Enrollment System**
Enter enrollments filename>
pizza.txt
**Error: unable to open enrollments file 'pizza.txt'
```

The program offers registration for exactly 5 CS classes: cs111, cs141, cs151, cs211, and cs251. The program supports 9 commands as shown to the right: help, stats, list, increase, enroll, waitlist, process, output, and quit. These commands will be discussed in more detail in the following sections. Since the goal of this project is an object-oriented solution, you are required to write your program using the `ClassReg` class defined in Appendix A. This class provides the functionality needed to

```
**CS Enrollment System**
Enter enrollments filename>
enrollments.txt
Enter a command (help for more info, quit to stop)>
help
stats
list class
increase class capacity
enroll class netid
waitlist class netid priority
process filename
output filename
quit
Enter a command (help for more info, quit to stop)>
list cs141
cs141
Capacity: 280
Enrolled (5): amalla5 hbrahm3 jkarim5 jmanus2 xlin48
Waitlisted (10): aalawd5 (10) zlnu3 (10) cmei25 (12) hall31 (20) aevdok3 (20) amacat4 (99) dpat320 (100)
yandry2 (110) ldmoy2 (112) mtened2 (112)
Enter a command (help for more info, quit to stop)>
```

enroll students, waitlist students etc. Since the program support 5 CS classes, this implies your main program will declare 5 instances of ClassReg, one per CS class supported by the program:

```
int main()
{
    ClassReg  cs111, cs141, cs151, cs211, cs251;
```

If you prefer, you could instead work with an array of 5 ClassReg instances:

```
int main()
{
    ClassReg  classes[5];  // cs111, cs141, cs151, cs211, cs251
```

The array is easier to extend if you wanted to support a larger registration system; you may also find the array easier to program. However, the first approach using the 5 individual instances might be easier to understand.

Enrollment Data:

Enrollment data is input from a text file formatted as follows. The file contains information about 5 classes, in this order: **cs111**, **cs141**, **cs151**, **cs211**, and **cs251**. Each class consists of a class name, the enrollment capacity, a list of students enrolled in the class (by netid), and a list of students waitlisted in the class (netid and priority pairs). Each list of students ends with a #. Here's an example:

```
cs111
300
#
#
cs141
10
amalla5 xlin48 jkarim5 hbrahm3 jmanus2 #
aalawi5 10 hali31 20 aevdok3 20 amacat4 99 dpate320 100 mtened2 100 #
cs151
5
#
zlnu3 10 cmei25 12 yandry2 110 ldmoy2 112 #
cs211
0
#
#
cs251
3
psarkar3 #
waiting 10 #
```

For example, cs141 has an enrollment capacity (i.e. maximum enrollment) of 10 students. Currently, cs141 has 5 students enrolled; the students are listed in the order they were enrolled. As you input the data, you'll use the ClassReg class function member **enrollStudent()** to enroll each student into "cs141". Also, cs141 has 6 students on the waitlist; the students are listed in their priority order, and you'll use the ClassReg member function **waitlistStudent()** to place these students on the "cs141" waitlist. You may assume the file is

formatted correctly, and that the number of students enrolled is \leq the class's enrollment capacity. Inputting from the file is designed to be straightforward; use the `>>` operator.

help command:

The "help" command is triggered by "h" or "help". In response, the program outputs 8 of the 9 commands supported by the program (the help command is not output). See earlier screenshots.

quit:

The "quit" command is triggered by "q" or "quit". In response, the program exits the command loop, outputs "***Done***", and ends.

stats:

The "stats" command is triggered by "s" or "stats". In response, the program outputs the number of students currently enrolled and waitlisted in each class.

```
Enter a command (help for more info, quit to stop)>
stats
CS111: enrolled=0, waitlisted=0
CS141: enrolled=5, waitlisted=6
CS151: enrolled=0, waitlisted=4
CS211: enrolled=0, waitlisted=0
CS251: enrolled=1, waitlisted=1

Enter a command (help for more info, quit to stop)>
```

list class:

The “list” command is triggered by “l” or “list”. In response, the program outputs information about the specific class; if the class is not one of cs111, cs141, cs151, cs211 or cs251 then an error message is output. Note that the list of enrolled students is output in sorted order; dynamically-allocate an array, fill with netids, and then call a sort function of your own writing (any sort is fine). The waitlisted students are listed in the same order as they are stored in the priority queue.

```
Enter a command (help for more info, quit to stop)>
list cs342
**Invalid class name, try again...

Enter a command (help for more info, quit to stop)>
l cs141
cs141
Capacity: 10
Enrolled (5): amalla5 hbrahm3 jkarim5 jmanus2 xlin48
Waitlisted (6): aalawi5 (10) hali31 (20) aevdok3 (20) amacat4 (99) dpate320 (100) mtened2 (100)

Enter a command (help for more info, quit to stop)>
```

increase class capacity:

The “increase” command is triggered by “i” or “increase”. In response, the program changes the enrollment capacity of the specified class; if the class is not one of cs111, cs141, cs151, cs211 or cs251 then an error message is output. Note that a class’s enrollment capacity cannot be decreased; attempting to decrease the enrollment capacity also yields an error message.

```
Enter a command (help for more info, quit to stop)>
increase cs261 1000
**Invalid class name, try again...

Enter a command (help for more info, quit to stop)>
i cs141 5
**Error, cannot decrease class capacity, ignored...

Enter a command (help for more info, quit to stop)>
i cs141 20
cs141
Capacity: 20

Enter a command (help for more info, quit to stop)>
l cs141
cs141
Capacity: 20
Enrolled (5): amalla5 hbrahm3 jkarim5 jmanus2 xlin48
Waitlisted (6): aalawi5 (10) hali31 (20) aevdok3 (20) amacat4 (99) dpate320 (100) mtened2 (100)

Enter a command (help for more info, quit to stop)>
```

Changing a class’s enrollment capacity does not impact the students on the waitlist; as shown above, the students in cs141 are still waiting to be enrolled in the class.

enroll class netid:

The “enroll” command is triggered by “e” or “enroll”. In response, the program attempts to enroll the student in the specified class; if the class is not one of cs111, cs141, cs151, cs211 or cs251 then an error message is output. If the student is already enrolled, the command has no impact (nothing changes). If the student is on the waitlist, then a check is made to see if there’s room in the class; if there’s room the student is removed from the waitlist and enrolled in the class, otherwise the command has no impact (the student remains on the waitlist with their original priority and position).

If the student is not enrolled and not on the waitlist, then an attempt is made to enroll them in the class. If the class is not full, then the student is enrolled. If the class is full, then the student is placed on the waitlist; what priority should be used? If the priority queue is empty, use a priority of 1, otherwise use the same priority as the last student in the priority queue (thereby forcing the new student to the end). Study the following sequence carefully:

```
Enter a command (help for more info, quit to stop)>
l cs251 1
cs251
Capacity: 3
Enrolled (1): psarkar3
Waitlisted (1): waiting (10)

Enter a command (help for more info, quit to stop)>
e cs251 psarkar3 2
Student 'psarkar3' enrolled in cs251

Enter a command (help for more info, quit to stop)>
e cs251 test1 3
Student 'test1' enrolled in cs251

Enter a command (help for more info, quit to stop)>
e cs251 waiting 4
Student 'waiting' enrolled in cs251

Enter a command (help for more info, quit to stop)>
e cs251 test2 5
Class full, 'test2' waitlisted for cs251

Enter a command (help for more info, quit to stop)>
l cs251 6
cs251
Capacity: 3
Enrolled (3): psarkar3 test1 waiting
Waitlisted (1): test2 (1)

Enter a command (help for more info, quit to stop)>
```

In #2, nothing happens because psarkar3 is already enrolled. In #3, test1 is enrolled because there’s room. In #4, waiting is enrolled from the waitlist. At this point the class is full; this causes test2 to be waitlisted in #5. We see from #6 the class is full with 3 students enrolled and 1 on the waitlist.

waitlist class netid priority:

The “waitlist” command is triggered by “w” or “waitlist”. In response, the program attempts to waitlist the student in the specified class; if the class is not one of cs111, cs141, cs151, cs211 or cs251 then an error message is output. If the student is already enrolled, the command has no impact (nothing changes). If the student is already waitlisted, then the student’s position in the waitlist is readjusted based on the priority. If the student is neither enrolled or waitlisted, they are inserted into the waitlist based on the priority.

```
Enter a command (help for more info, quit to stop)>
l cs141 1
cs141
Capacity: 10
Enrolled (5): amalla5 hbrahm3 jkarim5 jmanus2 xlin48
Waitlisted (6): aalawi5 (10) hali31 (20) aevdok3 (20) amacat4 (99) dpate320 (100) mtened2 (100)

Enter a command (help for more info, quit to stop)>
waitlist cs141 xlin48 123 2
Student 'xlin48' enrolled in cs141

Enter a command (help for more info, quit to stop)>
w cs141 dpate320 0 3
Student 'dpate320' waitlisted for cs141

Enter a command (help for more info, quit to stop)>
w cs141 test1 10 4
Student 'test1' waitlisted for cs141

Enter a command (help for more info, quit to stop)>
l cs141 5
cs141
Capacity: 10
Enrolled (5): amalla5 hbrahm3 jkarim5 jmanus2 xlin48
Waitlisted (7): dpate320 (0) aalawi5 (10) test1 (10) hali31 (20) aevdok3 (20) amacat4 (99) mtened2 (100)

Enter a command (help for more info, quit to stop)>
```

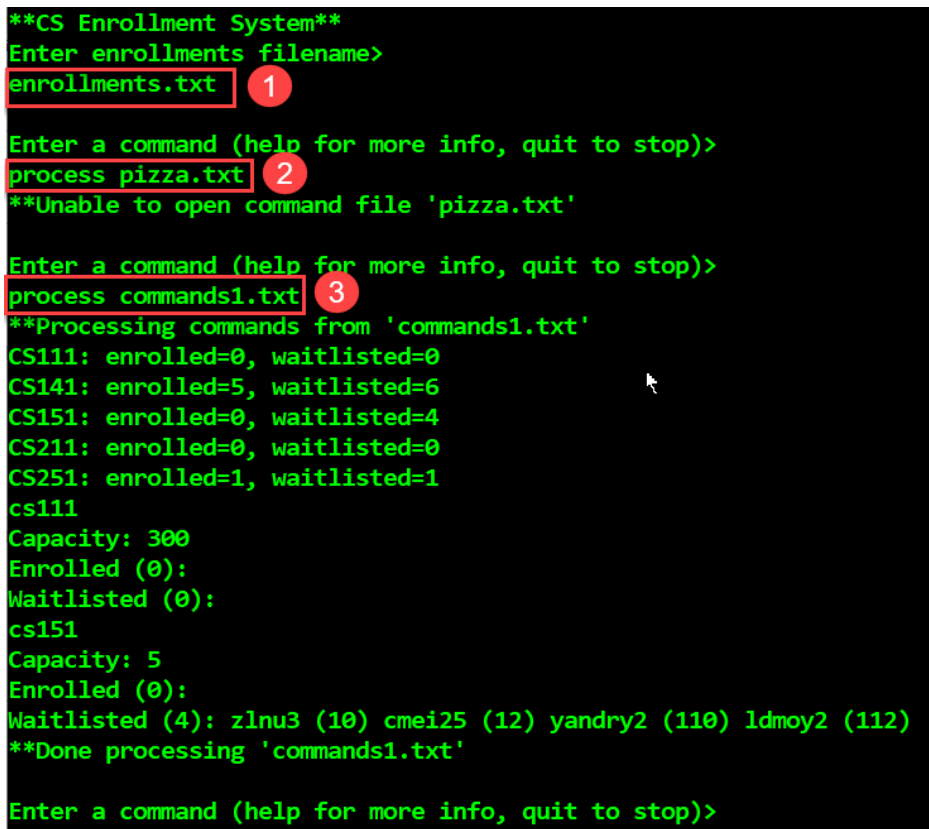
In #2, nothing happens because xlin48 is already enrolled. In #3, dpate320 is repositioned in the waitlist (to the front). In #4, test1 is waitlisted with a priority of 10. The end results can be seen in #5.

process filename:

The “process” command is triggered by “p” or “process”. In response, the program attempts to open the given filename and input commands from this file until “q” or “quit” is encountered; you may assume the file contains valid commands, and ends with either “q” or “quit”. For example, suppose the file “commands1.txt” contains the following:

```
stats
list cs111
list cs151
quit
```

Here’s what happens when we process this file:



```
**CS Enrollment System**
Enter enrollments filename>
enrollments.txt 1

Enter a command (help for more info, quit to stop)>
process pizza.txt 2
**Unable to open command file 'pizza.txt'

Enter a command (help for more info, quit to stop)>
process commands1.txt 3
**Processing commands from 'commands1.txt'
CS111: enrolled=0, waitlisted=0
CS141: enrolled=5, waitlisted=6
CS151: enrolled=0, waitlisted=4
CS211: enrolled=0, waitlisted=0
CS251: enrolled=1, waitlisted=1
cs111
Capacity: 300
Enrolled (0):
Waitlisted (0):
cs151
Capacity: 5
Enrolled (0):
Waitlisted (4): zlnu3 (10) cmei25 (12) yandry2 (110) ldmoy2 (112)
**Done processing 'commands1.txt'

Enter a command (help for more info, quit to stop)>
```

In #2 the process command is unable to open the file; an error message is output and the next command is input. In #3, the process command is able to open and execute the commands from the file. The main rationale for this command is to make testing easier; you can put commands into a file and execute them using the process command.

But implementing this command is also an interesting design problem. Commands to the program now come from two sources: the keyboard (cin), and an input file (e.g. infile). The obvious solution is to write a function to process commands from the keyboard, and then copy-paste this function and modify it to input

commands from a file. But the better way is to design one function that executes commands from an input stream:

```
void processCommands(istream& input, bool promptUser, ...)
{
    string cmd;

    if (promptUser) // prompt user at the keyboard:
    {
        cout << endl;
        cout << "Enter a command (help for more info, quit to stop)>" << endl;
    }

    input >> cmd;

    .
    .
    .
}
```

When you want to input commands from the keyboard, call the function like this:

```
processCommands(cin, true, ...);
```

When you input from the keyboard, you also want to prompt the user. However, when you want to input commands from a file, you open the file and then call the function like this:

```
ifstream infile;
.
. // open the file, make sure it was opened, and if successful process the commands:
.
processCommands(infile, false, ...);
```

Not only is this approach more elegant (using one function for two purposes), it naturally handles nested command files recursively! For example, “commands2.txt” can process commands1.txt:

```
process commands1.txt
enroll cs111 jhummel2
process commands1.txt
quit
```

This is perfectly legal.

output filename:

The “output” command is triggered by “o” or “ouput”. In response, the program opens the given file for output and writes the current enrollment data to this file. The data is output in the same format as the enrollments input file. You may assume the file will open successfully.

```
**CS Enrollment System**  
Enter enrollments filename>  
enrollments.txt  
  
Enter a command (help for more info, quit to stop)>  
e cs111 jhummel2  
Student 'jhummel2' enrolled in cs111  
  
Enter a command (help for more info, quit to stop)>  
o enrollments2.txt  
Enrollment data output to 'enrollments2.txt'  
  
Enter a command (help for more info, quit to stop)>
```

Here’s the contents of the output file “enrollments2.txt”:

```
cs111  
300  
jhummel2 #  
#  
cs141  
10  
amalla5 xlin48 jkarim5 hbrahm3 jmanus2 #  
aalawi5 10 hali31 20 aevdok3 20 amacat4 99 dpate320 100 mtened2 100 #  
cs151  
5  
#  
zlnu3 10 cmei25 12 yandry2 110 ld moy2 112 #  
cs211  
0  
#  
#  
cs251  
3  
psarkar3 #  
waiting 10 #
```

Part 02: a List class

In part 02 you're going to implement the functions defined in a provided "list.h" header file; see Appendix B of this document. These functions implement a one-way linked-list, and are similar to the functions you developed in project 05. The most important difference is the addition of a **Tail** pointer that points to the last node in the linked-list; this allows efficient access to the last element of the list, which is needed by the **Queue** class. You will need to update your logic to keep the Tail pointer updated in any functions that create or modify the list.

Your implementation must reside in the file "list.cpp". You'll also need to follow the rules of C++ classes, which implies the need for a **copy constructor** and the overloading of **operator =**. To help you implement these functions the proper way, a skeleton "list.cpp" file is provided with initial implementations.

Requirements

How you solve the problem is just as important as developing a correct solution. In this assignment we are going to impose the following restrictions, which you must adhere to. Breaking a restriction typically leads to a score of 0.

1. No global variables. A global variable is a variable declared in "main.cpp" but outside of any functions. Use local variables and parameters instead of global variables. Global struct definitions are fine.
2. In part 01 you must use the provided **ClassReg** class to represent the 5 classes and maintain the underlying enrollment data. Do not abandon the class and solve the problem using your own data structures. In part 02 you must implement the **List** class as given in the provided "list.h" file. You can add data and function members, but you cannot change or remove what is originally given. In particular, a **Tail** pointer has been added that your logic will need to use and maintain to meet the required time complexities on the List functions.
3. No use of the built-in algorithms for searching, sorting, min, max, etc. Write your own functions.
4. All functions must be less than 100 lines of code (not counting comments); this includes main().
5. Your program must represent a general solution, the ability to input & process any properly-formatted input files.

Getting Started

As it stands right now, for part 01 you'll need to work on Codio ("cs141-project06") since we are providing only the compiled versions of the 4 classes: ClassReg, Queue, PQueue, and List. When working on part 01, you'll need to compile the program as follows:

```
g++ -g -Wall main.cpp classreg.o queue.o pqueue.o list.o -o main
```

If time permits, we'll compile the classes for other platforms such as Ubuntu 20.04 and Mac OS X, allowing you to work on those platforms as well. Watch for announcements on Piazza and BB. For part 02, you'll be able to

work on any platform; be sure to review the skeleton code on Codio in the provided “list.cpp” file. Do not change anything in the “list.h” file; you can add data and function members, but you cannot change or remove what is originally given. You’ll also need to meet the time complexities noted in the comments; e.g. `push_back()` must execute in $O(1)$ time (Hint: use the Tail pointer).

Gradescope submissions

Part 01: submit your “main.cpp” file to Gradescope under “Project 06, part 01: main.cpp”. The # of submissions may be limited.

Part 02: submit your “list.cpp” file to Gradescope under “Project 06, part 02: list.cpp”. The # of submissions may be limited.

Both parts will be graded based on two factors: (1) correctness as determined by Gradescope, and (2) manual review of for commenting, readability (consistent indentation and whitespace), appropriate variable names, use of functions, and adherence to project restrictions. Deductions are made based on lack of commenting / readability, bad variable names, and breaking project restrictions.

In terms of grading, note that we expect all submissions to compile, run, and pass at least some of the test cases; do not expect partial credit with regards to correctness unless your submission compiles and runs. By default, we grade your **last** submission. Gradescope keeps a complete submission history, so you can **activate** an earlier submission if you want us to grade a different one; this must be done before the due date. We assume **every** submission on your Gradescope account is your own work; do not submit someone else’s work for any reason, otherwise it is considered academic misconduct (how else did you end up with someone else’s program?).

Academic Honesty

In this assignment, all work submitted for grading **must** be done individually. While we encourage you to talk to your peers and learn from them, this interaction must be superficial with regards to all work submitted for grading. This means you **cannot** work in teams, you cannot work side-by-side, you cannot submit someone else’s work (partial or complete) as your own. The University’s policy is available here:

<https://dos.uic.edu/conductforstudents.shtml>

In particular, note that you are guilty of academic dishonesty if you **extend or receive any kind of unauthorized assistance**. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, screen sharing, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at <https://dos.uic.edu/conductforstudents.shtml>.

Appendix A: "classreg.h"

```
/*classreg.h*/

//
// Author: Prof. Hummel, U. of I. Chicago, Spring 2021
//
// Implements class registration for one class, e.g. "CS141".
// Supports the enrollment of students via netid, and a waitlist
// of students based on a priority; the lower the priority the
// higher the chance of getting enrolled.
//
#pragma once

#include <iostream>
#include <string>
#include <exception>
#include <stdexcept>

#include "queue.h"
#include "pqueue.h"

using namespace std;

class ClassReg
{
private:
    string Name;           // class name
    int Capacity;          // enrollment capacity for class
    Queue Enrolled;        // ordinary queue of enrolled students, in enrollment order
    PQueue Waitlist;       // priority queue of waiting students, in priority order (low to
high)

public:
    //
    // default constructor
    //
    // Sets name to "", capacity to 0, and queues are empty.
    //
    ClassReg();

    //
    // parameterized constructor
    //
    // Sets name and capacity to the given values; queues are empty.
    //
    ClassReg(string name, int capacity);

    //
    // destructor
    //
    // Frees the memory related to the queues.
```

```

//
~ClassReg();

// NOTE: copy constructor not needed because underlying data provide copy constructor.
// NOTE: operator= not needed because underlying data provide operator=.

//
// getName
//
// Returns the class name.
//
string getName();

//
// setName
//
// Sets the class name to the given name.
//
void setName(string newName);

//
// getCapacity
//
// Returns the class enrollment capacity.
//
int getCapacity();

//
// setCapacity
//
// Sets the class enrollment capacity to the given value.
// Note that new capacity cannot be less than the current
// capacity, otherwise an exception is throw, i.e.
// throw invalid_argument("ClassReg::setCapacity: smaller");
//
void setCapacity(int newCapacity);

//
// numEnrolled
//
// Returns the # of students currently enrolled.
//
int numEnrolled();

//
// enrollStudent
//
// Enrolls the given student in the class by adding them
// to the end of the Enrolled queue; if the student is already
// enrolled, then nothing happens --- the Enrolled queue is
// left as is.
//
// If the class is full, an exception is throw, i.e.
// throw runtime_error("ClassReg::enrollStudent: full");

```

```

//
void enrollStudent(string netid);

//
// searchEnrolled
//
// Searches the Enrolled queue for the given student. If
// found the student's position is returned; positions are
// 0-based, meaning the first student has position 0. If
// the student is not found, -1 is returned.
//
int searchEnrolled(string netid);

//
// retrieveEnrolledStudent
//
// Retrieves the student netid from the Enrolled queue at the
// given position. Positions are 0-based, meaning the first
// student has position 0. An exception is thrown if the position
// is invalid, i.e.
// throw throw invalid_argument("ClassReg::retrieveEnrolledStudent: invalid
position");
//
string retrieveEnrolledStudent(int pos);

//
// removeEnrolledStudent
//
// Removes the student from the Enrolled queue at the given
// position. Positions are 0-based, meaning the first student
// has position 0. An exception is thrown if the position is
// invalid, i.e.
// throw invalid_argument("ClassReg::removeEnrolledStudent: invalid position");
//
void removeEnrolledStudent(int pos);

//
// numWaitlisted
//
// Returns the # of students currently waitlisted.
//
int numWaitlisted();

//
// waitlistStudent
//
// Waitlists the given student in the class by adding them
// to the Waitlisted queue; their position in the queue is
// determined by the given priority. If the student is already
// waitlisted, they are reomved and then re-inserted; this
// updates the student's position based on the new priority.
//
void waitlistStudent(string netid, int priority);

```



```

//
// searchWaitlisted
//
// Searches the Waitlisted queue for the given student. If
// found the student's position is returned; positions are
// 0-based, meaning the first student has position 0. If
// the student is not found, -1 is returned.
//
int searchWaitlisted(string netid);

//
// retrieveWaitlistedStudent
//
// Retrieves the student netid and priority from the Waitlisted
// queue at the given position. Positions are 0-based, meaning the
// first student has position 0. An exception is thrown if the
// position is invalid, i.e.
// throw invalid_argument("ClassReg::retrieveWaitlistedStudent: invalid position");
//
void retrieveWaitlistedStudent(int pos, string& netid, int& priority);

//
// removeWaitlistedStudent
//
// Removes the student from the Waitlisted queue at the given
// position. Positions are 0-based, meaning the first student
// has position 0. An exception is thrown if the position is
// invalid, i.e.
// throw invalid_argument("ClassReg::removeWaitlistedStudent: invalid position");
//
void removeWaitlistedStudent(int pos);
};

```

Appendix B: “list.h”

```
/*list.h*/

//
// Author: Prof. Hummel, U. of I. Chicago, Spring 2021
//
// Implements a one-way linked-list with optimized insertion at
// head and tail of list. The nodes contain 2 data values, a string
// and an integer.
//

#pragma once

#include <iostream>
#include <string>
#include <exception>
#include <stdexcept>

using namespace std;

class List
{
private:
    class NodeData
    {
    public:
        string  Value1;
        int     Value2;
    };

    class Node
    {
    public:
        NodeData Data;
        Node*    Next;
    };

    //
    // data members:
    //
    Node* Head;
    Node* Tail;
    int   Count;

    //
    // private member functions:
    //
    void initAndDeepCopy(const List& other); // inits this list then makes deep copy of
other
    void freeAndReset(); // free the nodes in this list and then resets to empty
}
```

```

public:
    //
    // default constructor
    //
    // Initializes the list to empty.
    //
    // Time complexity: O(1)
    //
    List();

    //
    // copy constructor
    //
    // Makes a deep copy of the other list into this list.
    //
    // Time complexity: O(N)
    //
    List(const List& other);

    //
    // destructor
    //
    // Frees all memory related to this list, and then resets to empty.
    //
    // Time complexity: O(N)
    //
    ~List();

    //
    // assignment operator (e.g. L2 = L;)
    //
    // Makes a deep copy of the other list into this list; since this
    // list already exists, the existing elements of this list are freed
    // before the copy is made.
    //
    // Time complexity: O(N)
    //
    List& operator=(const List& other);

    //
    // size
    //
    // Returns the # of elements in the list.
    //
    // Time complexity: O(1)
    //
    int size();

    //
    // empty
    //
    // Returns true if empty, false if not.
    //
    // Time complexity: O(1)

```

```

//
bool empty();

//
// search
//
// Search the list for the first occurrence of the string value.
// If found, its position in the list is returned. Positions are
// 0-based, meaning the first node is position 0. If the value is
// not found, -1 is returned.
//
// Time complexity: on average O(N)
//
int search(string value);

//
// retrieve
//
// Retrieve's the data from node at the given position; the list
// remains unchanged. Positions are 0-based, meaning the first node
// is position 0. Throws an "invalid_argument" exception if the
// position is invalid, i.e.
// throw invalid_argument("List::retrieve: invalid position");
//
// Time complexity: on average O(N)
//
void retrieve(int pos, string& value1, int& value2);

//
// insert
//
// Inserts the given data in the list such that after
// the insertion, the value is now at the given
// position.
//
// Positions are 0-based, which means a position of 0
// denotes the data will end up at the head of the list,
// and a position of N (where N = the size of the list)
// denotes the data will end up at the tail of the list.
// If the position is invalid, throws an exception, i.e.
// throw invalid_argument("List::insert: invalid position");
//
// Time complexity: on average O(N)
//
void insert(int pos, string value1, int value2);

//
// remove
//
// Removes and deletes the node at the given position; no data is
// returned. Positions are 0-based, meaning the first node
// is position 0. Throws an "invalid_argument" exception if the
// position is invalid, i.e.
// throw invalid_argument("List::remove: invalid position");

```

```

//
// Time complexity: on average  $O(N)$ 
//
void remove(int pos);

//
// front
//
// Returns the data from the first node in the list. Throws an
// exception if the list is empty, i.e.
// throw runtime_error("List::front: empty list");
//
// Time complexity:  $O(1)$ 
//
void front(string& value1, int& value2);

//
// back
//
// Returns the data from the last node in the list. Throws an
// exception if the list is empty, i.e.
// throw runtime_error("List::back: empty list");
//
// Time complexity:  $O(1)$ 
//
void back(string& value1, int& value2);

//
// push_front
//
// Inserts the given data at the front of the list.
//
// Time complexity:  $O(1)$ 
//
void push_front(string value1, int value2);

//
// push_back
//
// Inserts the given data at the back of the list.
//
// Time complexity:  $O(1)$ 
//
void push_back(string value1, int value2);
};

```