3460:210 Project Part C

# **Project Assignment Part C: My Computer Lab**

**Overview**

The purpose of this part of the project is to make sure that you know how to write a program that will leverage linked lists, creates separate compilation files, and also utilizes external linkage.

**PROGRAM SPECIFICATION**

The universities really liked the ability to bundle the information for the labs, but now there is some consideration given to increasing the performance of the system that have been accompanied with recommendations. Our task will be to implement these suggested changes.

Recap of current system

The computer lab is a system that hosts users of several universities’ computer labs, and allows users of the labs to log into and log off from the available machines. The computer lab system also keeps track of the machines that are available and those that are in use, allowing users to locate an existing computer that is free as well as free the computers (remove users) upon the user’s logout.

Specifications for the program

You have 8 universities under contract. These labs contain computer stations that are hold the work stations numbered as shown in the table below:

|  |  |
| --- | --- |
| Lab Number | Computer station numbers |
| 1 | 1-19 |
| 2 | 1-15 |
| 3 | 1-24 |
| 4 | 1-33 |
| 5 | 1-61 |
| 6 | 1-17 |
| 7 | 1-55 |
| 8 | 1-37 |

The universities would like to change the structure of the dynamic array of objects to that of a linked list. This linked list should continue to hold the same information as before, and also continue to store the data dynamically (on the heap). We will be creating linked lists to replace our member arrays, and the pointer array too! Each node will be dynamically allocate for this Linked List. We also will be leveraging one of the major benefits that come with the linked list by allocating only as many nodes as there are active users. What this means is that for each university’s lab, we can grow and shrink the linked list accordingly. We will continue to utilize the parallel arrays from prior, and we will use the maximum size for each linked list from the corresponding values found in the LABSIZES array. Whenever a linked list’s size reaches the threshold, issue a message that states the lab is full at this time and to try again later. So, for example, when I have allocated the 19th active node on linked list 1, the next request for a new user will exceed the limit and generate a message (assuming no users had been removed between requests).

Another requirement for the linked list pertains to requests for the removal of a user from a work station. You will have the choice of two mutually exclusive options here to perform this task, and you should *explicitly state in your comments* which of the two options that you are using. The first option is to remove the node, and rewire the pointers. The second option is to create what is called a tombstone. With this option you leave the node intact, but mark the node as being ‘not in use’ (use a sentinel?). These tombstones can then be reused for new users by searching through the list and locating one sequentially as you go. Consider how to manage this. Creating tombstones only when the threshold was reached? Allocate the entire linked list and initially fill it with tombstones? Make a choice based on solid reasoning, and weigh the tradeoffs. Overall, the first option of no tombstones allows us to save memory, by only allocating when someone uses a station, and by deallocating a node when the user logs off. The second option yields faster processing because we don’t do any individual Input/Output operations (allocations/deallocations) after we do the initial allocation. Notice, with this change you might consider the existing capacity checking procedure and modify that process as you need to … with efficiency in mind.

Next, the recommendation has been put forth to make use of iterators for the linked list. An iterator is a generalization of a pointer. Iterators are used to move through the elements in some range of a container. The operations ++ , -- , and re-referencing \* are usually defined for an iterator. The basic idea, and in fact the prototypical model for iterators, can easily be seen in the context of linked lists. A linked list is one of the prototypical data structures, and a pointer is a prototypical example of an iterator. Where applicable, use these iterators exclusively for all operations. Construct your iterator as constant iterator where it should be too.

Next, you will create a file for capturing login information. This file will log all of the information and activities in chronological order as new users login, and only for login information (not logout, etc…). The design of this file is up to you, but this file *must* contain the following:

* Timestamp, a c-string of 20 characters that looks like: Mon Jun 12 15:22:30
* Action, a char that is in/out, that is: I (for login), and O (for logout)
* UserID, this is the integer from the node’s element, can pad with leading zeroes
* User time, this is our integer from the node’s element.
* User name, this is our string (putting this last to make it easier to read back).

Each and every time that your program runs, this file will be appended to. The name of the log file is totally up to you, but should be a .txt extension. Design a separate function(s) to handle all of this. Be sure to write onlywhen the user has *successfully* completed a transaction. Don’t forget to think through what happens when a request goes onto the waiting list too.

Consider the following code example for the time (must #include ctime, which you may already have from randomizing):

// current date/time based on current system

time\_t now = time(0);

// convert now to c-string form

char\* timeOf = ctime(&now);

// we want a way to limit the size to be just 20 in length

timeIn[20] = '\0'; // this effectively truncates the c-string

This code works and can be used directly in your program. Placing spaces between each item/field in your file output will probably be a good idea, in case we need to read this log later (and we will). Make use of manipulators for the output.

Finally, we will now use separate compilation files, and we will try to externalize our CONSTANTS (see below).

*Design of the structures*

We will continue to use the following code that creates a constant fixed array of length 8 for the labs. This is our array that holds the number of possible work stations. NUMLABS is also created to enable us to add or subtract a lab. Using constants here allows our program to be much more dynamic in terms of modification efforts and control. This array is also used for the allocation of the linked list (the sizes).

Separate files by placing your code into implementations files (.cpp), and also all of your headers (.hpp). A header contains class and functions definitions. The implementation file holds the implementation of the class. By doing this, if our class implementation doesn’t change then it won’t need to be recompiled. Don’t forget your include guards.

Next, modify this part of your existing code to declare these constants using either an unnamed Namespace or a Namespace named myConstants.

**// Global Constants**

**// Number of computer labs**

**const int NUMLABS = 8;**

**// Number of computers in each lab**

**const int LABSIZES[NUMLABS] = {19, 15, 24, 33, 61, 17, 55, 37};**

**// Names of university of each lab**

**const std::string UNIVERSITYNAMES[NUMUNIVERSITIES] = {"The University of Michigan", "The University of Pittsburgh", "Stanford University", "Arizona State University", "North Texas State University", "The University of Alabama, Huntsville", "Princeton University", "Duquesne University"};**

You are going to figure out how to share the constants with the other functions. Consider **extern** to explicitly make the constants NUMLABS, LABSIZES, and UNIVERSITYNAMES external. We want these constant identifiers to be defined in one translation unit (hpp) and shared with other translation units. Don’t make extra copies (see the lecture notes).

Consider your login logic and insert the function(s) to add your log file. Make sure you successfully logged in first before adding the record to your file.

*Design of the application*

Everything else in your system should remain intact, and the functionality should be the same. This means all off the displays (menu, etc…) look and behave exactly as they did before. Carefully consider your redesign effort strategy to maintain your functionality, and to change code as minimally as possible. Test thoroughly, and test incrementally.

That’s a **lot** to change, and at a high level here’s what we are doing: 1) convert arrays of objects to linked lists (holding the same objects); 2) distribute code into separate compilation files, 3) convert to iterators for searching, reading, and writing to the linked lists; 4) create a log file for each and every login, and; 5) employ external linkage for the constants listed above.

You may continue to maintain your data as classes in the linked list or convert them to structs, your choice. Your list class may encapsulate a separate node class too (see the demo), but whatever you decide you will want to have all of the supporting methods in your design. Do not forget to manage (and free ~destructor) the LL memory since we are on the heap. Do not use #pragma directives in your files in place of include guards. #Pragma is not approved by the C++ Standards Committee. Be careful not to use the linked lists from the STL either or you will receive no credit ☹. Yikes!

Submission Instructions – for programming solutions

On Brightspace, go to the matching Assignments for the **PROJECT PART-#**, where # is the number or character of the project part assigned (eg., 5 for project part 5), and submit your cpp, and any hpp files (if they exist). Unless otherwise specified, they may be submitted under any name that you prefer (such as main.cpp).

*Last updated 8.22.2017 by Will Crissey.*

*Be aware that programming falls under all of the rules of plagiarism. Be careful when using any coding found in the outside world that is not your own. Any evidence of plagiarism is subject to sanctions like forfeits, suspension, and even ejection, as determined by the Department of Student Conduct and Community Standards.*