3460:210 Project Part B

# **Project Assignment Part B: 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 declare and use classes and objects, and extends the practice into the use of arrays of objects.

**PROGRAM SPECIFICATION**

The universities are happy with the current system supporting their computer labs for students. However, like any successful venture there are changes. The universities would definitely like to have more data for their labs.

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 add the name of the student as well as the user login time. To make this change, we are going to build each lab as an array of objects. The objects will allow us to “bundle” information, and the array will provide a medium for holding, maintaining, viewing, and arranging this data. We will continue to be using a separate array for each university’s computer lab, and those arrays will continue to be different sizes based upon the number of labs required by each university. The real difference is that the arrays hold objects rather than just a single data type.

We are also going to continue using the static array as a control support structure.

Each user continues to have a unique five-digit ID number, and now we add the name of the user, and the time of use too. Whenever a user logs on, the user’s ID, the user’s name, the lab number, the time of use, and the computer station number are input into our system. For example, if user 49193 Joe Jones logs onto station 2 in lab 3 and uses the computer for 15 minutes, then your system receives user ID 49193, Joe Jones, lab number 3, computer station number 2, and time of use 15 as the input data. Similarly, when a user logs off a station, then your system receives the lab number and computer station number.

The array of lab sizes (one for each lab) holds the values of the maximum number of possible stations for each given lab, and then those sizes are used to build our arrays of objects. We want the array of objects be dynamically allocated in memory, i.e., placed on the heap? What if we had used a vector? Is a vector allocated on the stack or the heap? How about the use of the string class within our user written class; will it continue to have the same functionality as a member of this class? Take some time to research these and other questions you might have considered.

We are still creating a jagged array. Array #1 holds the pointers to each member array, and our array of objects, call it Array #2, holds the lab user IDs, names, and user times.

Keep the array to hold the reference (pointers) for each of the computer stations. This array, an array of pointers, is used in the dynamic allocation for each of the computer stations whose size is determined by the LABSIZES array. Next step will be to dynamically allocate member arrays pointed to by our reference array. Again, each of these holds the information for the respective computer stations. And the logic is the same: these are arrays that function as two dimensional. This was a key concept in our prior project’s implementation. Now it is all about the member array holding the objects. We continue to use the pointers in the same manner as before.

The class in the member array contains three elements: the userID, an int, a string containing the user’s full name, and the time of use which is also an int. This is a simple class without much space or time overhead when processing. More details are provided under the design of structures sections. Default values for stations will continue to set the userID to -1 (this indicates it is free or empty), and the name and time of use to be a blank space and zero respectively.

We haven’t addressed the fact that this array of objects may fill up, and run out of places to put our data. The changes for accommodating this will be to check on the capacity of the array during login requests, and issue a message as specified below. So your program will now have to check the array capacity during each login request. Since all unused slots have a value of -1, we can add a function during insertion requests to do this. If the array has no capacity then the program takes a different path after issuing the message and returns back to the menu without adding the user. So this will be a case where the behavior of an add operation changes.

So, to recap, if user 49193 (randomly generated) requests lab 3, our program will check the capacity for lab 3 before attempting any further add logic. If the lab is not full then the request for the new user continues with the computer station, and so on. This was the expected behavior from before. Your program should continue to disallow a request for a previously filled slot as before too. If the lab is full, however, then the message is displayed, the screen is paused, and then the program returns back to the main menu for the next request.

A sample view of the lab array of objects is given here, and you can see where user 49193 is logged into station 2 in lab 3, and user 99577 is logged into station 1 of lab 4:

|  |  |
| --- | --- |
| Lab Number | Computer station numbers |
| 1 | 1: empty 2: empty 3: empty 4: empty …… 19: empty |
| 2 | 1: empty 2: empty 3: empty 4: empty … 15: empty |
| 3 | 1: empty 2: 49193, Joe Jones, 60 3: empty 4: empty ……… 24: empty |
| 4 | 1: 99577, Mary Pearl, 15 2: empty 3: empty 4: empty …………… 33: empty |
| 5 | 1: empty 2: empty 3: empty 4: empty ………………………….……………………………….… 61: empty |
| 6 | 1: empty 2: empty 3: empty 4: empty .… 17: empty |
| 7 | 1: 89098, Al Gore, 30 2: 67890, George Bush, 60 3: empty ……..……………….…….…. 55: empty |
| 8 | 1: 12345, Bjarne Stroustrup, 45 2: 67899, Prof Will, 15 3: empty …..…37: empty |

*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 arrays (the sizes).

**// 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};**

We also introduce these constants for each of the labs that we have a contract with. We will use these in our program too. They correspond by position with LABSIZES, and so represent a parallel array.

**// 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"};**

This array is used for the link to the appropriate array. So, for example, if we are looking at the third computer lab, it would be represented as Stanford University. We will be using this when we access and display arrays in our program.

The configuration will continue to be a jagged array, with an array of pointers that reference the member structure (which holds the data). Our member arrays will be an array of objects and is represented below:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | etc… |
| User id  Name of student  Time used | User id  Name of student  Time used | User id  Name of student  Time used | User id  Name of student  Time used | User id  Name of student  Time used | User id  Name of student  Time used | User id  Name of student  Time used | …  … |

Accessing the objects, we would be able to use the index to locate any particular element.

*Design of the application*

The repeating menu will basically remain intact. One thing we will add, however, is a computer generated user ID rather than the ID being entered. We will modify part of the existing logic to *generate* an ID for the user. This user ID will continue to be the data type of integer, and will be up to 5 digits in length. The values are 1 – 99999, and should appear with all 5 digits (leading zeroes) on the display upon generating them. Certainly consider random and seed appropriately. Also, check out the setfill manipulator as a possible helper for this leading zeroes thing. Setfill allows you to set a fill character to a value using an input parameter, such as ‘0’ for leading zeroes. This will allow you to print user ID 34, for example, as 00034. That is what is required. Researching it, you’ll see that you will need to reset setfill after use too, or you will be printing a lot of zeroes on the display! Your program also should guarantee an absolutely unique user ID has been generated for each new activity that is added. Think about how this works…

Now, when the login request is invoked, you will no longer take input for the user ID, but instead use a randomly generated user ID, and then take input for the name of the user (std::string type) and the time used (integer). Can we verify these? Yes, and we absolutely must do that. Consider what is a valid name? Let’s establish the maximum length to be 35 characters long. And let’s say valid names should not be blank, either. Write the logic to support all of that. Input of the time used requires it to be either 15, 30, 45, or 60. These are the only valid entries that the system accepts for the time used. Validation may be done using the method you write to take the data, or it can be made into separate helper functions.

So whenever someone logs in, the array should be updated now with the user ID, the student name, and with the time used.

Along with the options to log in and log out, include a display option that offers the user the opportunity to print any one of the labs on the screen. Also, include a search option so that the administrator can type in a user ID and the system will output what lab and station number that user is logged into, or “that user is not logged on” if the user ID is not logged into any computer station. Finally, add a check of the capacity of the array for login requests. When a lab gets full, and issue the accompanying message “Lab 2, The University of Pittsburgh, is at full capacity. Please try your request again later…”.

Again, make use of manipulators such as setw for the output and match the output samples.

## *The user interface design*

The user interface must be built as described.

Firstly, we want to build the class that holds the user information. This will be our user id and our name. This class will be used as the type of our array. You will want to include a constructor, and may create other constructors that you need to support the creation of objects. Our class has private data, so a user cannot completely initialize an object without the use of a constructor. Hence, you will provide a constructor and should specify its meaning.

Next, you want to define the methods that will support this class. You will definitely want all of the interfaces to go through setters and getters. Consider whether or not you need any operator overloading, such as for outputting (<<), inputting (>>), assignment (=), and for comparisons (== or > or <), and subscripting as necessary. You may or may not find the need to overload. You will expect to find the need to create helper functions. Remember, any and all entries should be validated. Also, any login actions that find the user seeking an already filled computer should be rejected, and an appropriate message should be displayed such as “this computer station is already in use”.

You will need to modify parts of the program to include the displaying of the user name and time used, and in particular searches should now show the user id and additionally the user name and time of use. The display of the labs will remain the same as before, that is, you will only be displaying the user ID.

You also should display the name of the university when inquiries are made for displaying a lab such as Lab # 2, The University of Pittsburgh…

Again, make use of manipulators such as setw for the output and match the output.

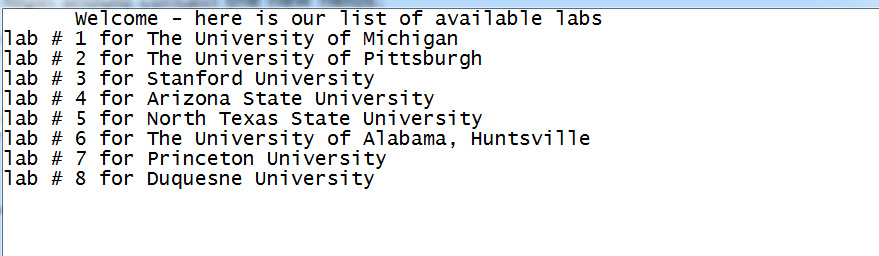
Everything else in your system should remain intact, and the functionality should be the same for this new release. Carefully consider your redesign effort strategy to maintain your functionality, and to change code as minimally as possible. Test thoroughly, and test incrementally.

**What’s changing for this release?:** 1) you’ll be converting your member arrays to member arrays of objects (dynamically allocated arrays of objects); 2) randomly generate user IDs, and add two new fields into the array; 3) add a parallel array that holds the names of the university of each lab that is displayed at the initial execution of the program; 4) Check lab capacity for login requests, and; 5) add or modify your system’s input or output of the new data as described above.

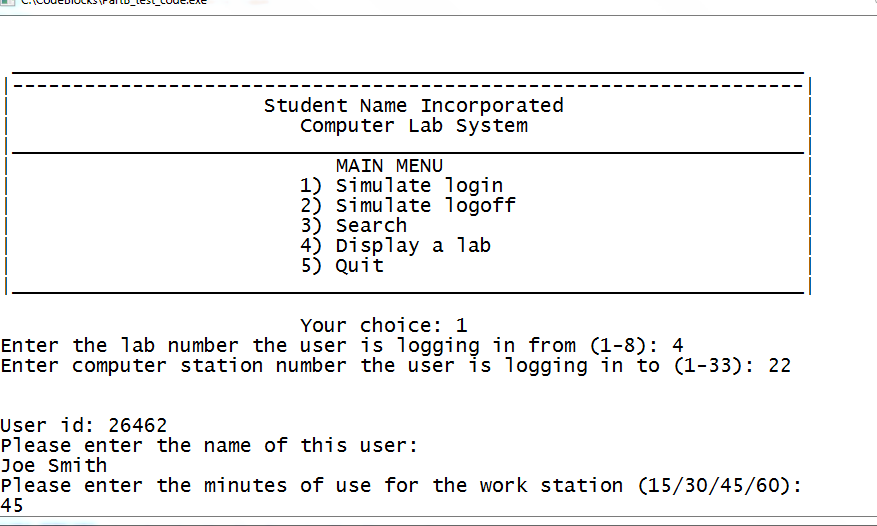
The most difficult part of this project is the logic to sync the arrays of objects with the reference array effectively and efficiently.

Some examples of how your program appears follow:

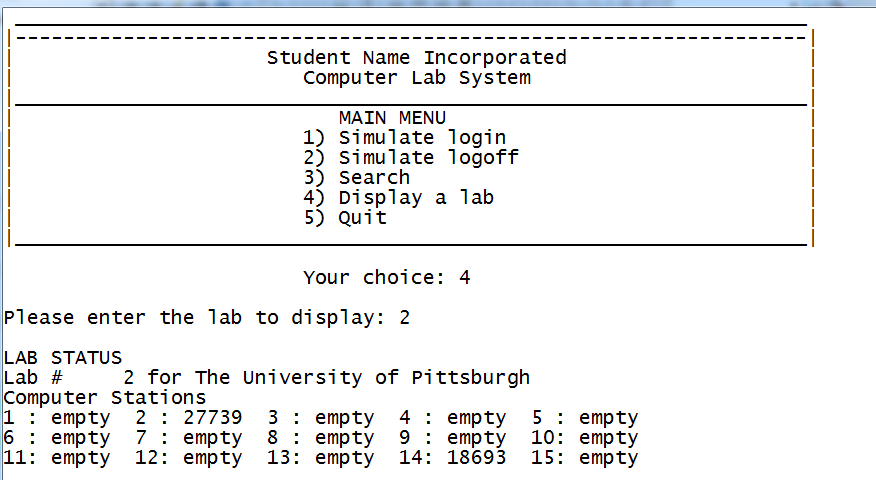
***At initialization (once, only at start up), display the universities and their appropriate labs:***



***Your login should generate user IDs, and contain the new fields:***



***The display of labs is as it was, only with the appropriate university added:***



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 8.20.2017 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.*