3460:210 Project Part A

# **Project Assignment Part A: 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**

You have signed a contract with several major universities to run their computer labs for their respective students. To get you started with your own computer labs, you will need to write some software to support your new business endeavor.

Introduction

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.

General Specifications for the program

You can run many computer labs. For now, we will have 8 universities under contract. These labs contain computer stations that hold the actual physical work stations that are 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 |

We are going to build each lab as an array. We also will be using a strategy to make these structures which allows them to be different sizes, and this is known as a jagged structure. We will be using a separate array for each university’s computer lab, and those arrays will be different sizes based upon the number of labs required by each university. For example, The University of Akron may have 44 lab stations while Case Western University may have 28, and so on.

Each student or user has a randomly generated unique five-digit ID number. Whenever a user logs on, the user’s ID (generated by your program), the lab number, the computer station number, name of the student as well as the user login time are inputs into the system. For example, if generated user 49193 logs onto station 2 in lab 3, then your system uses standard input operations and takes the lab number 3, computer station number 2, the name, and the time as the input data. Similarly, when a user logs off a station, then your system takes the lab number and computer station number as inputs.

Your program is used to track, by lab, which user is logged onto which computer, and additionally the program has the capability to maintain those labs’ computer stations via menu-driven requests. There is an array of lab sizes (one for each lab) that holds the values of the maximum number of possible stations for each given lab, and those sizes are used to build our jagged arrays. These jagged arrays will be dynamically allocated by having another array of pointers allocate and point to array. The jagged array holds the elements (data values) as objects of a lab class. Default values for stations should set to -1 (this indicates it is free or empty).

Support structures

Let’s stop for a moment and outline some support structures. We are going to create additional data structures and constants to help out with our programming efforts by reducing the amount of coding that we have to do. We will build an array which holds the values of the maximum number of possible stations for each of the given computer labs (LABSIZES). These represent the sizes of the jagged member arrays. So this array will simply be used to build the 2D jagged arrays. Let’s consider this static array to be a control *support structure*. Here is the code for your program. As you can see, this creates a constant fixed array of length 8 for the labs. Notice 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 code modification efforts and control.

**// 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 have more code for your program. This is another fixed array of information, this array has the name of the university for each lab. They correspond by position with LABSIZES, and so represent a parallel array.

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

**const std::string UNIVERSITYNAMES[NUMLABS] = {"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"};**

What happens logically is we can now create iterative code constructs that point to multiple sources of related lab information. So, for example, if we are looping through and stop at the third computer lab, it would represent Stanford University (UNIVERSITYNAMES) which holds 24 work stations (LABSIZES). We will be using this when we access and display arrays in our program.

More Specs…

We are going to define a class used 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.

So each user will have a randomly generated unique five-digit ID number, the name of the user, and the time of use too. Whenever a user logs on we input the data into our system. For example, if randomly generated user 49193 Joe Jones logs onto station 2 in lab 3 and uses the computer for 15 minutes, then your system has user ID 49193, Joe Jones, lab number 3, computer station number 2, and time of use 15 as the input data. When a user logs off a station your system uses the lab number and computer station number. The data stored in the array (the object) will be the user ID #, the name, and the amount of time. The lab and station are derived from the arrays structures, not stored.

*How it works*: we are creating a jagged array. Array #1 holds the pointers (addresses) to each *member* array. The array of objects, call it Array #2, holds the actual data, the lab user IDs, names, and user times. The array that holds the reference (pointers) for each of the computer stations, the pointers, is used in the dynamic allocation for each of the computer stations, the size of which is determined by the LABSIZES array. So we dynamically allocate the member arrays using the reference array. These are arrays that function as two dimensional. This is a key concept in our project’s implementation. The pointer array is static, and placed into local memory in your program. The member array is dynamic, and placed into the heap. If you find your code has references such as \*\*, you did the allocation incorrectly.

Your program should disallow a request for a previously filled slot. When a request to add a new user for an already filled spot occurs, have the program issue the message “sorry, that work station is already taken”, and return back to the menu for a new request.

Side note – so we want the array of objects to 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 will not use a vector, and your research will tell you why…

Class Specs…

Okay, so the class for 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. Default values for stations should be setting 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. Not that these should be the values you replace when someone requests to logoff. 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.

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 can expect to find the need to create helper functions. Remember, any and all entries should be validated. Any login actions that find the user seeking an already filled computer should be rejected, and the message “this computer station is already in use” should be displayed.

Looking at the member arrays, an array of objects, it can be represented as 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 | …  … |

For accessing the objects we would use the index to locate any particular element.

Another sample view of the lab array of objects with data 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 |

Lab is Full Specs…

We haven’t addressed the fact that this array of objects may fill up, and run out of places to put our data. The logic for accommodating this will be to check on the capacity of the array during login requests, and issue the message “sorry, this lab is full. Please try again later”. So your program will have to check the array capacity during each login request. Since all unused slots have a value of -1, we can have a helper function do this during insertion requests. If the array has no capacity then the program does not do any insert logic and takes a different path, issuing the message and returning back to the menu without adding the user.

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. 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.

Menu Specs…

As your program starts, before it displays the menu and starts processing requests, display the universities and their appropriate labs as seem below. Do this only once.

Create a repeating menu that allows the user of the system to login or logoff users and collect the appropriate data. Whenever someone logs in or logs out, the array should be updated. 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 four 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. Formatting of your menu should match exactly to the example below.

Important – do not ask the user if they wish to continue… that is the purpose of the menu, and specifically option 5. You should pause the program so the user can view the result (consider cin.get()) and press any key to continue. If your program prompts the user for something like “enter Y to repeat” then you’ll lose points ☹

Input Specs…

Create logic to randomly *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! These are integers – **do not use strings**. 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…

When the login request is invoked you will 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.

Output Specs…

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. For searches you must show the user id and additionally the user name and time of use. Notice the display of the labs will be displaying the user ID and also the name of the university such as Lab # 2, The University of Pittsburgh…

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…”.

See the examples below for more regarding output specs.

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

## *Test, test, and test again*

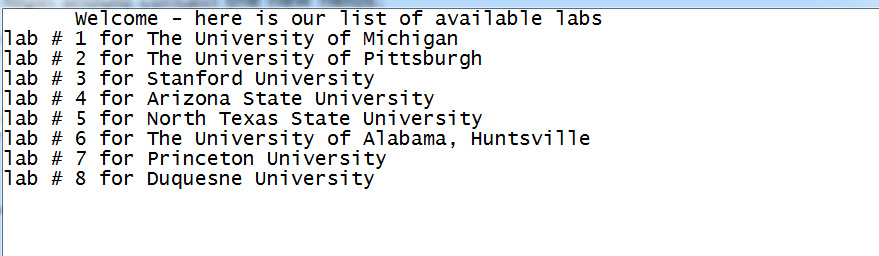
Code as minimally as possible. Test thoroughly, and test incrementally.

**What’s going on for this release?:** 1) create a repeating menu with the 5 options described; 2) you’ll be making your member arrays as arrays of objects (dynamically allocated arrays of objects); 3) randomly generate user IDs, and include two validated fields from input into the member array; 4) include a parallel array that holds the names of the university of each lab that is displayed at the initial execution of the program; 5) Check lab capacity for login requests, and; 6) match your system’s input or output of the data as described.

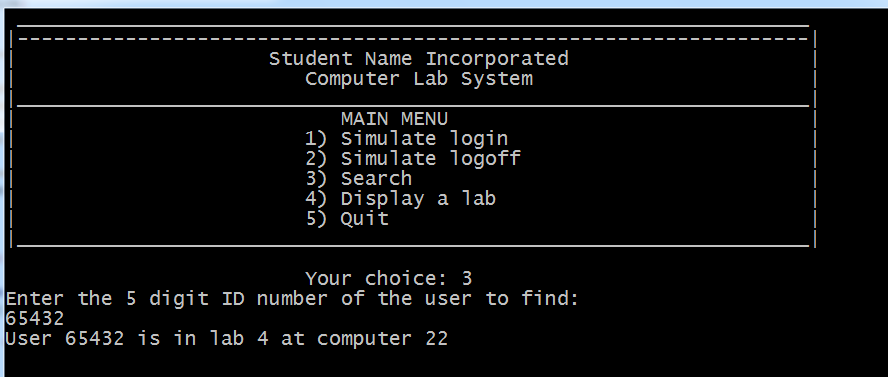
Make sure that your programs follow good documentation standards and follow the requirements for assignments. Reference the rubric standards on Brightspace. Do not use namespace std, do validate data. DO NOT USE VECTORS, use arrays. DO NOT USE SEPARATE COMPILATION, all in one file. Projects using vectors or separate compilation will not get credit.

Some examples of how your program appears follow:

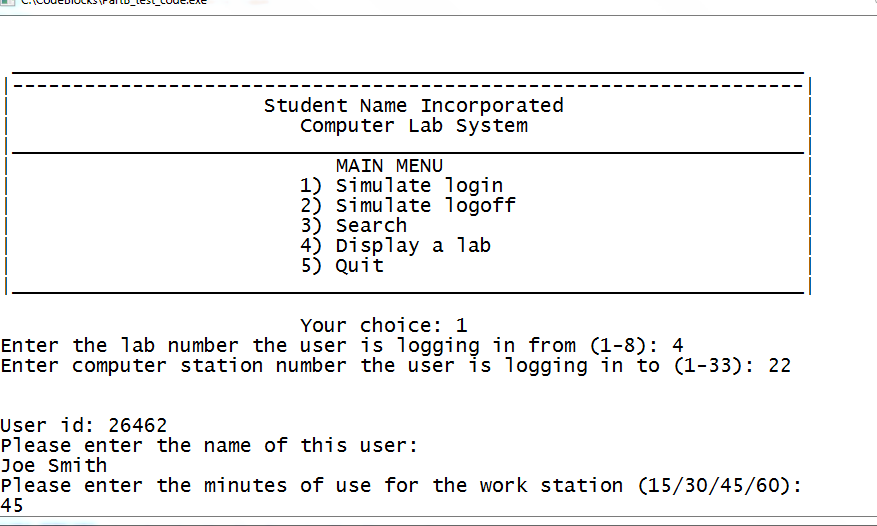
***At initialization (once, only at start up), display the universities and their appropriate labs:***



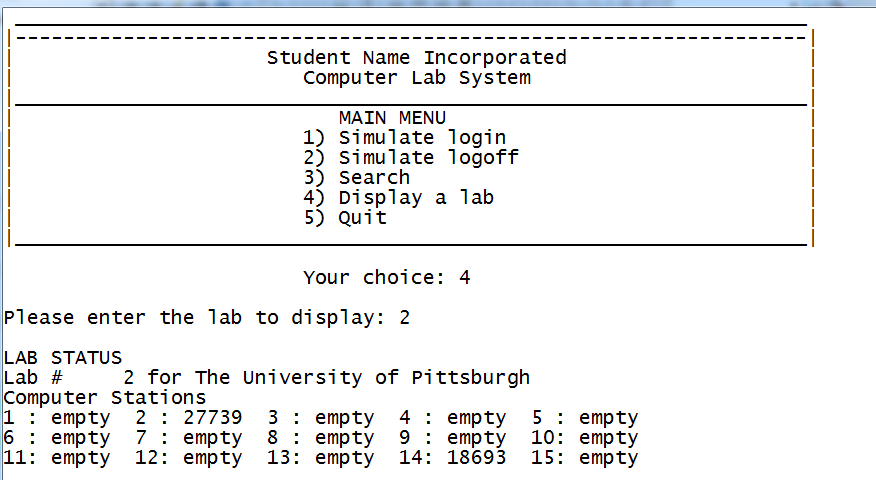
***Here is an example of the menu, and a search request:***



***Your login should randomly generate user IDs, and input the following fields as seen:***



***The display of labs with the appropriate university displaying***



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. 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.*

**COPYRIGHT © 2017 Will Crissey Renwald, Assistant Professor, All rights reserved.**

This document is intended exclusively for use by The University of Akron's Department of Computer Science. Do not distribute document copies to the public in any way.

***THIS PROGRAM'S CONTENT IS PROTECTED***

The document and associated content are either the property of, or used with permission by, The University of Akron, Department of Computer Science and may be protected by copyright and other restrictions. Copyrights in the site content may also be owned by individuals and entities other than, and in addition to, The University of Akron, Department of Computer Science. The use of this document by you, or anyone else authorized by you, is prohibited, except for purposes of completing the homework/project. Any unauthorized use of the document may violate copyright laws, trademark laws, the laws of privacy and publicity, and communications regulations and statutes.