CSCE 2004 - Homework 7
Due Date - 12/8/2016 at 11:59 PM

0. Overview:

In this assignment, you will develop a program to handle university degree auditing and course management. That is, after taking a set of courses, a student would like to know if she/he has met the graduation requirements for a particular degree (or what the student is missing to meet those requirements). The requirements include required courses, total credit hours, minimum GPA, etc. As a college student, can you see the value of such a tool in the real world?

To build any software system, one must be equipped with the following: a firm grasp of the problem to be solved, the ability to divide the problem into smaller problems, devising related algorithms to solve the smaller problems and combining their solutions, the ability to translate the high level design into a computer program in a particular programming language (in our case C++), and the ability to debug and test the program.

All of our projects in this course are related to this core problem. So before you begin, you may want to review your own degree requirements to make sure you really understand them. If we are not able to understand a problem, then no matter how good we are as programmers or software engineers, we will not be able to solve that problem.

We currently have an interactive program to "manage" the coursework of a student. The program is also able to retrieve and store a student's course work from and to a file, respectively. We also practiced using C++ classes, a user defined type. Instead of or in addition to using five parallel arrays, we used a single array of **Course** objects to hold the same information. We also developed a way to represent the degree requirements in a file. Project seven has all the functionalities of project six, plus the additional features of **reading the requirement file into a data structure** and **performing a simple degree audit for the University Core requirements**. Project seven is an extension of project six, and some data structures (classes or arrays) could be introduced and adjusted as you see fit. You also have another chance to fix any bugs from the previous project.

1.1 A basic algorithm for degree audit:

To perform a degree audit, we will first need to create a new course list that contains **only the distinct** courses with grades between **A to D** from the current list of courses taken (our old five parallel arrays or our new array of **Course** objects). Note that if a student repeats the same course twice (for example a social sciences elective), **only one of these two can be used**. Hence, we want the list to be distinct. Also courses with grades **F, W, or I** cannot be used to meet any course requirement, so we will not add them to the list of distinct classes. When we create this distinct list and the same class has been taken multiple times, we want to save the one with the **higher grade**. To save time, we'll call the distinct list "**L**" from now on.

Once we have L, we can perform the actual degree audit. Conceptually, we will have two lists. One list contains the various requirements (e.g. 8 hours of math courses) for a single requirement group, and the other is the list of distinct classes (L). Furthermore, each requirement will have its own list of classes that can be used to satisfy the requirement. (E.g MATH 2004 and MATH 2014 might both satisfy the math requirement above.) For each requirement, we will need to search the entire list of distinct classes (L) to determine whether or not that requirement has been met. To do so, we maintain an accumulator. When a class is found that meets the requirements, we add that class's hours to the accumulator. If, after we have gone through all of the classes, the value of the accumulator is ≥ the hours specified by the requirement, the requirement has been satisfied. If all of the requirements in the group have been satisfied, we claim the degree audit is successful.

Note that **within a group**, each course can be used only once. For example, PLSC 2003 can either be used for social sciences electives or U.S. history exclusively. However, a course might be used to meet multiple requirements in **different** requirement groups. For example, PHIL 3103 meets humanities of the University Core as well as BACS, BSCS, BSCmpE majors Ethics Core. So in general, once we finish with one requirement group, we will repeat the search of each course in L again for each different requirement group. **However, you are not required to audit other requirement groups beyond the University Core for this project.**

You will use the above algorithm to perform the degree audit for the University Core. Note that it is possible for the algorithm to give an incorrect audit result if all social sciences electives are from the same department, or if no lab courses are taken for the natural sciences. Do you see why? It's okay for now. To make the program more robust, we would have to refine our requirement modeling and our requirement file so that rules such as "social sciences electives must be taken from at least two departments" can be represented and captured in the requirement file. We will ignore that problem for now. It will be left to the "food for thought" section.

1.2 Each requirement as a class:

In the requirement file, each blank line separates one requirement from another requirement. Each requirement has a **group label** and a **subgroup label**, the **hours required**, and a **list of courses and their credit hours that can be used to meet the requirement** (please review the HW6 document). To better support the above algorithm, we will introduce a requirement class (e.g. **Requirement**) that contains all the above information and some additional members to keep track of the **courses matched**, **hours earned**, and **whether the requirement is met or not**. We may also introduce useful member functions to support the algorithm described above. One way to design such a class is to have *parallel arrays* as members of the class for the list of courses. Another way is to introduce an *additional class*, and then to store an array of that class. You are free to select a design that makes sense to you.

1. Problem Statement:

One goal of this assignment is to give students a chance to synthesize what they have learned in the entire semester by finishing the semester long "degree audit" project. In so doing, students will gain experience with designing and using their own classes, text file processing, writing and using functions, using and introducing array data structures and various variables of built-in types and the string type, and using if statements and loops. A secondary goal is to give students a chance to modify an existing program to make it more useful. To meet these goals, students will be required to improve their previous course management and degree requirement program to make the program more modular, flexible and robust. Students will be **required** to introduce a suitable class or user defined type to represent **a single requirement** and suitable arrays or classes to hold all the requirements in a requirement group (such as the University Core). Students will also be **required** to implement a degree audit for the University Core Requirements using the algorithm provided (section 0.1). Students have the freedom in designing new data structures or functions and modify existing data structures and functions as they see fix. Students should document their design and modification in their report.

Enhanced features: Using the information from the degree requirement file and the list of courses, the program performs degree audit for the University Core Requirements and outputs the audit outcome.

Extra credit will be given if the program performs a degree audit beyond the University Core Requirements. Your code will have to support multiple requirement sets in order to accomplish this goal.

Students are allowed to use any "built-in" functions related to files and any combination of user defined classes, functions, arrays, data structures, if statements, and loops (while loops or for loops). You are **NOT** required or allowed to look ahead to more advanced C++ features like **pointers or vectors** to perform this task. Students are free to develop all the classes or functions as they see fit (you have more freedom in your program design than before).

2. Design:

Your first design task is to create a class for one requirement (e.g. **Requirement**). This class must have at least the following information stored in its members: **requirement group, requirement subgroup, hours needed for this requirement, and a list of courses** (each course has an associated course name and credit hour). You may want to introduce additional variables or members to indicate the number of courses in the list and to "remember" those courses used to meet the requirement. You may also wish to maintain useful information such as the number of credit hours earned, and so on. In addition to members, you must decide which member functions (or **methods**) are necessary and their purposes should be, what their arguments and types are, and what their return values should be in order to support the degree audit algorithm.

Since each requirement group (such as the University Core) has several subgroups or requirements, we must maintain all subgroups in data structures to perform degree audit. Your next task is to decide where and how to store all the requirements (subgroups) of a given group such as the University Core.

Once the data structures are designed, you should decide on the logic needed to process the requirements file and use that information to initialize the data structures. You will need to use blank lines to determine where a requirement and the requirement group line (first line of each requirement) begins. Note that in the requirement file, different groups of requirements may be intermixed.

Finally, you have to decide when to perform degree audit and how to present the degree audit results to the user.

You may, if you want, adjust the behavior of the hw6 program to make it “better” according to your taste. If you do, please make sure that you document your rationale and the adjustment in your report.

3. Implementation:

First correct any errors from HW6 to make sure the program maintains the correct information in the main course list. Then develop and test the data structures or classes for one requirement. After that, develop the program to initialize the requirement data structures from the requirement file.

After making sure the list of courses are correct and the requirement data structures have the right information, you will implement the degree audit algorithm and output the audit results (as a way of debugging as well).

These are some implementation suggestions (you may choose to follow this advice or you may try something else completely. It’s up to you.)

- You could add an option to your menu for performing the degree audit. It would ask which requirements file should be read, open the file, process the file contents, do the audit, and then finally inform the user of the results.
- The problem is significantly easier if you break it up into related functions. For example, you might write a function to create the list of distinct classes (L from above), a different function to perform the audit for a single Requirement, and a third function to process all of the requirements in the group. If you choose to pursue the extra credit opportunity, you could have another function to process each of the requirement groups.

Since you are starting with your previous program, you should already have something that compiles and runs. Since you must add new features and maintain the current behavior of the program, it is important to make these changes **incrementally** one action or function at a time, writing comments, adding code, compiling, and debugging. This way, you always have a program that "does something" even if it is not complete.

4. Testing:

Test your program to check that it operates correctly for all of the requirements listed above. Also check for the error handling capabilities of the code. Try your program with several input values **[2-3 executions of your program]**, and save your testing output in text files for inclusion in your project report. You can include the testing at the bottom of your report.

5. Documentation:

When you have completed your C++ program, write a report (no page limit) describing what the objectives were, what you did, and the status of the program. Does it work properly for all test cases? Are there any known problems? Save this report in a separate text file to be submitted electronically.

Save this report in a separate text file to be submitted electronically. Please use the **Project Documentation Template** provided on Moodle as the basis for your documentation.

6. Project Submission:

In this class, we will be using electronic project submission to make sure that all students hand their programming projects and labs on time, and to perform automatic analysis of all programs that are submitted. When you have completed the tasks above go to Blackboard to "upload" your documentation (a single **.pdf**, **.doc**, or **.docx** file), and your C++ program (a single .**cpp** file). Do **NOT** upload an executable version of your program or files in other extensions or format.

The dates on your electronic submission will be used to verify that you met the due date above. **No late projects will be accepted for credit.** Please refer to class syllabus about no late assignment policy as well as suggestions given regarding to submitting a partially working program.

You will receive partial credit for all programs that compile even if they do not meet all program requirements, so please hand projects in on time.

7. Food for thought:

How should we refine the information in the requirements file and the degree audit algorithm so that the University Core requirements such as social sciences and natural sciences could be audited accurately?

How do we devise and refine the current algorithm for a **complete** degree audit (with multiple requirement groups)?