CSCE 2004 - Homework 7 (hints)
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.

Hints and Suggestions

1.1 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 could also store a single array of Course objects as well. You are free to select a design that makes sense to you.

In the requirement file, each blank line separates one requirement from another requirement. Once a Requirement class is developed, we may have an array of Requirement objects just as we have an array of Course objects. Such an array is built from information in the requirement file.

The attributes or data members of the Requirement class includes all the bold font elements in the project description (see first paragraph of this section) : **group label, subgroup label, hours required, list of courses and their credit hours that can be used to meet the requirement, hours earned, courses matched.**

Note that the first five items are initialized from one section of the requirement file. Since the list of courses varies from one requirement to another, we have to either update the requirement as we read in a pair (course name and credit hour) or store all the pairs (in arrays), then update.

Suppose we use two parallel arrays (or one array of another class) for the **list of courses and their credit hours that can be used to meet the requirement**, we need one attribute for the capacity of the array (using a static int) and another attribute for the number of elements in the array.

If we decide to update the requirement as we read in a pair (course name and credit hour), we may need to have an "add" or "push_back" method that inserts the information at the end of the array (the logic is similar to adding a new course in our previous project).

To store all the courses used for meeting a requirement, we may need another array of Courses as an attribute for the Requirement class. Let us refer to it as "**course used**".

To implement the match concept as part of the Requirement class, we may need to have a method that takes in one course and uses it to search the **list of courses and their credit hours that can be used to meet the requirement.** If a match is found, the course is added to the **course used** array and **hours earned** is increased accordingly. Let us call this "**MATCH**" method.

To decide is a requirement is met or not, we may have another method that compare **hours required** and **hours earned**.

Last but not the least, we may develop a print method that displays the audit information for that requirement: met or not met, partially met or completely met, and the courses used to satisfy that requirement. Note that the Requirement object has all the needed information.

Another alternative is to write a validate() method for the requirement class that takes as arguments an array of unique courses and the number of unique courses and returns a Boolean

value (true if the requirement was satisfied and false if it was not). (This means you need fewer data members inside the Requirement class).

1.2 An algorithm for processing the requirement file and setting up Requirement array:

Assume the file format is correct, so error checking may be omitted for now. Use the blank line between requirements to delineate one requirement from another. Every time a blank line is encountered, the number of requirements in the array is increased by one. This is some pseudocode that might be helpful:

```
// get the name of the requirement file either from the user or
// as an argument to the function

// get things ready so we can write code to read from the file

// initialize requirement array index to zero

Use getline to read the first line (should be a group name);
Loop while not eof
{
        Use getline to read sub group;
        Use >> to get hour needed;
        Use a set method to initialize the group, sub group, and
hour;
        // note the Requirement object of the above is
        // determined by the array index

        Use getline to get rid of the stuff in the previous line
        Use getline to get course number;
        Loop while not eof && course number is not a blank line
        {
            Use >> to get hour;
            Use the add method to insert the course number and
hour pair into the parallel arrays (or objects) in a Requirement
object

            Use getline to get rid of stuff after >> in get hour
            Use getline to get course number;
        }

         Increase array index by one;

        Use getline to read the group of the next requirement;
}
```

1.3 Having an array of Requirement for only University Core:

In the previous step, we have an array of all the requirements for a particular degree. From that array, we may copy those requirements whose group is “University Core” into another array of Requirements, which will be used in our degree audit.

1.4 About the suggested new menu option:

Even though it is definitely a good idea to have a new menu option to repeat the degree audit action, you are not required to do that. Instead, you may simply perform the degree audit action right before the program termination using the following algorithm. (The menu option is definitely the better approach, though.)

1.5 An algorithm for processing degree audit:

We assume that we have two arrays, one for the distinct list of courses taken and one for University Code requirements. Note that he algorithm may be broken down into functions so we do not need to have two nested loops.

```
For each course in the distinct list of courses do
{
    For each requirement in the University Core do
    {
        if ( the requirement is not met (use a method) )
            use MATCH method of the requirement for the course
    }
}
```

Alternatively, you could do something like this (Note that the same course may not be used twice for the same requirement group (e.g. University Core) and some book keeping is needed to make sure of that. For example: HIST 2003 should not be used for both “Social Sciences” and “US History”.)

```
For each requirement in the University Core do
{
     Call validate(), passing the list of distinct courses and
the number of unique courses
}
```

Use the following algorithm to show the degree audit results:

```
For each requirement in the University Core do
{
    use print method of the requirement for the course
}
```

(Of course, this assumes that all of the important information is stored inside the “Requirement” class. If it’s not, you’ll have to do something more complicated to display the results.)