Assignment 3 & 4

Documents Due:	2:45 pm, Tuesday, February 12, 2002	(Assignment 3)
	2:45 pm, Tuesday, February 19, 2002	(Assignment 4)
Code Due:	9:30 am, Thursday, February 14, 2002	(Assignment 3)
	9:30 am, Thursday, February 21, 2002	(Assignment 4)

This assignment will give you more practice writing basic C++ code, and introduce you to pointer manipulation, explicit dynamic memory allocation and primitive arrays.

Preliminaries

As with all assignments in this course, part of your grade will be based on your coding style and on the way your written work is presented. The *Homework Policies* handout discusses these issues in depth and also explains how to submit your work. If you do not have a copy of this handout, you can obtain a copy by visiting the course website at http://www.cs.hmc.edu/courses/2002/spring/cs70/.

Scenario

It is a little-known fact that Harvey Mudd has a sister school, Deep Glen Polytechnic, in northern Scotland. Because the depressed state of the Scottish economy has limited their funding, their registrar currently maintains course rosters by hand, using 3″ × 5″ cards. Your job is to provide them with a rudimentary course-database system.

As well as being the finest scientific college in Europe, Deep Glen Polytechnic is famous for its playful students. A long-standing tradition at Deep Glen is that students attempt to confuse professors by changing their appearance frequently—notably by dying their hair—so that the professors will have trouble remembering the names of individual students even though there are only 128 of them in the entire school. The professors have countered this game with a nasty trick of their own: they keep changing the meeting times of their courses, so that almost nobody will be able to show up at the correct time and therefore there will be fewer names to learn.

The registrar, fondly known to all as Jeanne "Lumpy" MacBettykin, has been caught in the middle of this battle. The professors have required her to keep track of the current hair color of all students, while the students have successfully bribed her (using precious chocolate-chip cookies sent to them by their doting mothers) to also track time changes and periodically provide them with printouts showing the latest times. Your new database program must therefore support these features.

This is actually the second attempt to automate DGP's course registration system. An earlier, buggy, version was written by a physics student at DGP, who graduated long ago. Because the registrar is now accustomed to the earlier user interface, you must rebuild the deeper levels of the code but keep the user interface intact.

Before You Begin

Download the source code for this assignment from the Homework area of the course website. The file archive to download is `assign3.tgz`. On turing, you can download and unpack the file archive by executing

```
curl -O http://www.cs.hmc.edu/courses/2002/spring/cs70/homework/assign3.tgz
tar xzvf assign3.tgz
```

This archive contains the following files and directories:

```
cs70ass3/Makefile              cs70ass3/student.cpp
cs70ass3/regsystem.cpp         cs70ass3/course.hpp
cs70ass3/dbmanager.hpp         cs70ass3/course.cpp
cs70ass3/dbmanager.cpp         cs70ass3/allout.txt
cs70ass3/regdb.hpp             cs70ass3/input.txt
cs70ass3/regdb.cpp             cs70ass3/interactivesession.txt
cs70ass3/student.hpp           cs70ass3/noerrs.txt
```

The assignment directory contains the beginings of a C++ implementation of a database system for the registrar's office of Deep Glen Polytechnic. At present, the code is only partially complete, and does not even compile. However, the user-interface component of the code (`regsystem.cpp`, `dbmanager.hpp`, and `dbmanager.cpp`) is essentially complete. The remaining code is only skeletal. It requires a small number of changes to compile—but substantially more work to finish and debug.

System Design

The registrar's database system is implemented using several classes, outlined below.

Registrar_DB

This class holds the following data:

- An array containing all students (*Student* objects)
- An array containing all courses (*Course* objects)

The lists of courses and students are kept in the order in which they were inserted. Do *not* attempt to sort or alphabetize them.

RegDB_Manager

This class holds the following data:

- A reference to a *Registrar_DB* object containing a database

The *RegDB_Manager* class provides the entire user interface (which is discussed in the next section). You should not need to modify this class.

Student

This class holds the following data:

- Full name of the student (represented as a string containing the name in the format "*lastname*, *firstname*"—for example, "`Bly, Nellie`")
- Hair color (another string)
- A fixed-size array of pointers to the *Course*s (at most five) currently being taken
- Any other information you may feel is required

The array of pointers to courses is not sorted; instead, it is kept in the order in which courses were added. When a student drops a course, the associated pointer should be set to `NULL` (zero), and the array should *not* be compacted or otherwise reorganized. When a course is added, the first `NULL` *Course* pointer should be used to add a pointer to that *Course*.

Course

This class holds the following information:

- Course number (an integer)
- Department (a string)
- Meeting time (a string; e.g., "MWF 11")
- Maximum enrollment (an integer)
- Roster (a dynamically allocated array of pointers to the *Student*s in the course)
- Any other information you may feel is required

The roster for each course is to be kept alphabetized by name. When alphabetizing, use the entire full-name string—don't attempt to separate the first and last name.

Technical Requirements

There are a number of technical requirements you must meet when completing the implementation of the above classes:

1. None of the classes use the *vector* class. (Part of the learning value of this assignment comes from dealing with primitive arrays.)

2. Each *Course* object contains a pointer to a dynamically allocated array of pointers to students in the course. The data in this array should be kept packed—if you remove a student who isn't the last element of the array, you must move the subsequent items in the array to close up the gap.

3. Each *Student* object contains a fixed-size array of pointers to courses being taken. The data in this array should *not* be kept packed—if you remove a course that isn't the last one in the array, you should simply set the pointer to NULL. A later "add" can then reuse this storage location. *Note that the requirements of the Course class and Student class are different.*

4. The *Registrar_DB* class contains a pointer to a dynamically allocated array with room to hold all students. The *Course* class contains an array of pointers to students. Notice that these two arrays have *different* types. This design is deliberate, giving you the opportunity to practice manipulating both types of arrays. The same applies to the arrays of courses in the *Registrar_DB* and *Student* classes. (When in doubt or confused, draw diagrams to understand the type structure.)

5. The top-level interactive interface assumes that there are no more than 128 students and no more than 20 courses. These limits are fixed by the trustees of DGP. Therefore the program should generate an error if these limits are exceeded rather than dynamically expanding the arrays.

6. Students and courses can be added to the arrays in *Registrar_DB* but they are never deleted. Nevertheless, you must provide proper destructors.

User Interface

The interactive interface supports the following commands:

quit terminate program

student add new student

course add new course

enroll enroll student in course

drop drop student from course

hair change student's hair color

time change course's meeting time

show students display all students

show courses display all courses

The two display commands should show the full information associated with each course or student, and should be identical to the sample outputs (see the *Testing* section on page 7).

Written Component — Assignment 3

Unless otherwise stated, answers to the questions in this section should be one sentence (or, in some cases, one line of code) only. You can make modifications to the code for the assignment to answer these questions, but should always return the code to its previous (working) state before moving on to the next question.

W1. Consider the scenario in which the registrar begins with an empty database and then

- Adds a new student `Anne Sterling`, who has `Brown` hair
- Adds a new student `Michael Souris`, who has `Black` hair
- Adds a new course `Life 101`, meeting at `MWF 9:00 am`, with a maximum enrollment of `3`
- Adds a new course `Anim 365`, meeting at `TT 5:30 pm`, with a maximum enrollment of `5`
- Enrolls `Anne Sterling` in `Life 101`
- Enrolls `Michael Souris` in `Anim 365`
- Enrolls `Anne Sterling` in `Anim 365`
- Enrolls `Michael Souris` in `Life 101`

Based on this information, the detailed technical description herein, and the provided code, draw a diagram showing the *Registrar_DB* data structure and the objects it points to after this interaction. You must draw the same kind of diagram you have seen in class, including types and field labels. (You can find the field labels by examining the source, although you may have to invent additional field labels to adequately express necessary parts of the data structure. To make your drawing task easier, you should also assume that the program has been changed so that `MAX_STUDENTS = 3` and `MAX_COURSES = 3`.)

W2. Each of the following declarations declare a variable that can be used as "an array" containing ten pointers to *Student*.

```
{
  (a) vector<Student*> students(10);
  (b) Student* students[10];
  (c) Student** students = new Student*[10];
      // ...
}
```

For each declaration above, state

i. What would happen at the end of the block when the variable goes out of scope. If there would be a memory leak, indicate what additional statement(s) would be required to prevent the leak.

ii. The type of `*students[3]`;

iii. Whether `students` is an object.

iv. How a similar declaration would be written for a class. Include both the code that would be placed in the class declaration and the code that would be written inside the constructor.

v. Whether the number of students (the size of the array) could be a parameter passed into the constructor in your code for iv.

Coding Component — Assignment 3

These coding questions are designed so that you can submit your code after completing each question. You will, however, only be graded on your final submission.

In undertaking this assignment, you *may not* modify any of the following files: `Makefile`, `regsystem.cpp`, `dbmanager.hpp`, `dbmanager.cpp`, `regdb.hpp`.

C1. Add any additional fields to the class declarations in the header files that you feel are necessary, based on the technical description in this document.

C2. Add the bare minimum code necessary to make the program compile (exception: colon initializers can be written in their final form, if desired).

C3. Read the *Testing* section on page 7 to understand the output format required.

C4. Examine each of the C++ files. In each file, replace comments that read `ADD PSEUDOCODE` with comments describing *in pseudocode form* how that routine should be implemented. (There will be a few places where "necessary" equals "nothing", but in most places you'll need to write at least a few lines.)

Do not submit perfect C++ code. If you submit C++ code, or something that is even close to finished working code, you will receive a *much* lower grade, even if the code works correctly. The point of this assignment is to teach you to use pseudocode as a design tool, and if you submit your final working program we will have no way of knowing whether you used pseudocode. Good pseudocode skips over the messy but obvious details, and concentrates on developing the tricky parts of the algorithms. (However, if a detail is particularly tricky, it can be good to mention it in the pseudocode so you won't forget it later.)

At the same time, you must also avoid submitting pseudocode that is so high level that it does not clearly state the approach. Ideally, good pseudocode can be translated into working code by a reasonably good programmer. If you aren't sure about the level of your code, talk to me or to a tutor *before* you get very far into the project.

Some functions in the skeleton program are so simple that it is silly to pseudocode them. Those functions contain the comment `ADD STUFF (PSEUDOCODE NOT NEEDED)`. You will have to complete those functions for Assignment 4, but it is not necessary to write pseudocode for them for Assignment 3.

You should submit your code using the electronic submission system described in the *Homework Policies* handout. You are not required to include a `README` file for this part of the assignment.

Written Component — Assignment 4

The written component for Assignment 4 is to create a `README` file for your program. The `README` file must be submitted at the deadline for written work, not the deadline for code.

Coding Component — Assignment 4

In undertaking this assignment, you *may not* modify any of the following files: `regsystem.cpp`, `dbmanager.hpp`, `dbmanager.cpp`, `regdb.hpp`. You may only modify the `Makefile` in the unlikely event that you add additional source files to the project.

C1. Convert your pseudocode to C++ code.

C2. Test and debug your code, ensuring that it meets the output specification.

You should submit your code using the electronic submission system described in the *Homework Policies* handout. You will have already submitted a `README` file for this part of the assignment, so you need not resubmit the `README` (if you do, the old one will be used regardless). You may submit an additional file `NEWS` describing any changes you have had to make to your design since writing the `README`.

Testing

When you run the program, you will find that there is a simple interactive interface that will prompt you for commands and let you test various operations. To help you see how your program should behave, a log of a sample interactive session is included with the source (`interactivesession.txt`).

When you begin to be confident that your code works, you can run two tests against the sample output files, both of which were generated from the same input file, `input.txt`:

`allout.txt` is the contents of both `cout` (where listings are printed) and `cerr` (where prompts and errors are printed). This file contains everything that the program would write to the terminal if you ran it interactively.

`noerrs.txt` is the contents of only `cout`. This file contains the listings of courses and students that are produced by the `show courses` and `show students` commands.

You can compare your output to the samples with the following commands. Note that they are different for `csh/tcsh` and `bash/sh/ksh` users. `csh/tcsh` users should type

```
unix% ./regsystem < input.txt |& diff -u - allout.txt
unix% (./regsystem < input.txt | diff -u - noerrs.txt > temp) >& /dev/null
unix% cat temp
unix% rm temp
```

whereas Bash/Bourne/Korn shell users should type

```
unix$ ./regsystem 2>&1 | diff -u - allout.txt
unix$ ./regsystem 2> /dev/null | diff -u - noerrs.txt
```

If everything is working correctly, none of these commands should produce any output. If `diff` reports differences, but you can't spot them, the differences may be in the whitespace. Try passing the output of `diff` through `cat -vet`:

```
unix% ./regsystem < input.txt |& diff -u - allout.txt | cat -vet
unix% (./regsystem < input.txt | diff -u - noerrs.txt > temp) >& /dev/null
unix% cat -vet temp
unix% rm temp
```

Or, for Bourne-like shells,

```
unix$ ./regsystem 2>&1 | diff -u - allout.txt | cat -vet
unix$ ./regsystem 2> /dev/null | diff -u - noerrs.txt | cat -vet
```

Tricky Stuff

As usual, there are some tricky parts to this assignment. Some of them are

- Start by thoroughly understanding the code provided in both `regsystem.cpp` and `dbmanager.cpp`. Even though you do not have to change it, you should understand how these files use the code you *do* have to write. In essence, the code in `dbmanager.cpp` is a specification for your own code.
- Despite the above, your code should not assume that it has been called with error-free arguments. For example, even though `dbmanager.cpp` will not try to add a student to a full course, your own code should double-check to make sure there is room in the course. That way, if somebody later breaks `dbmanager.cpp`, the program won't crash horribly (this technique is called "defensive programming").
- Remember to keep the student roster within a course alphabetized by name. However, do *not* keep the registrar's main database alphabetized. (The latter requirement is somewhat silly but will certainly simplify your life.)
- When a student is enrolled in a course, both the course and the student must be modified. Be careful how you go about this, or you will get yourself into trouble. The same goes for removing students from courses.
- Since it doesn't make sense for a student to be enrolled in a course twice, you should check for that possibility. If a student is already enrolled, you can simply ignore the enrollment request and return an error indication.
- Test your boundary cases. Does your program handle things correctly when a course is full? When a student is taking a maximum load and tries to add a course? When the registrar exceeds the limits on the number of students or the number of courses? You may find it convenient to temporarily reduce the built-in limits to smaller numbers for testing, but don't forget to put them back before you submit!

- Don't forget to write destructors for all three classes.
- The print routines return a reference to an `ostream`. The value returned should simply be the stream that was passed as an argument.