Recitation 7: Separate Compilation

Topics

- Separate Compilation
- Namespaces
- Operator overloading

Recitation Project

Implement a transaction processing program for the Registrar.
It will have to be able to add/drop courses and students.
It should also allow for some student information to change over time.

Details

Your program should allow someone using your classes to do these activities:

- add a course to the set of offered courses
- add a student to the student body
- enroll a student in a course
- cancel a course
- print a university report listing all students in each course
- start a new semester by purging all students and courses
- [if time] change a student's name
- [if time] drop a student from course
- [if time] remove a student from the student body

Some of these activities imply that other activities must be or have been done.
We are assuming that *names are unique* (if they weren't we would have to turn everyone into a number).
Also, we are assuming that, miraculously, students' names are all single tokens (if they weren't, things would just be more annoying).

First Part

The registrar maintains all the data for the University.
[You won't write a University class - just tester code]
First we need to make sure that we have implemented the *functionality* of the Registrar for keeping track of students and courses by writing a

method in the Registrar class to do the above activities and, of course, create the other classes needed by the Registrar class to do its jobs. Your first task is to get the following code working.

You should write one method at a time and test it..

```
#include "Registrar.h"
#include <iostream>
//using namespace BrooklynPoly; // Uncomment when namespace 
added
using namespace std;

int main() {

    Registrar registrar;

    cout << "No courses or students added yet\n";
    cout << registrar << endl;  // or 
registrar.printReport()

    cout << "AddCourse CS101.001\n";
    registrar.addCourse("CS101.001");
    cout << registrar << endl;  // or 
registrar.printReport()

    cout << "AddStudent FritzTheCat\n";
    registrar.addStudent("FritzTheCat");
    cout << registrar << endl;  // or 
registrar.printReport()

    cout << "AddCourse CS102.001\n";
    registrar.addCourse("CS102.001");
    cout << registrar << endl;  // or 
registrar.printReport()

    cout << "EnrollStudentInCourse FritzTheCat 
CS102.001\n";
    registrar.enrollStudentInCourse("FritzTheCat", 
"CS102.001");
    cout << "EnrollStudentInCourse FritzTheCat 
CS101.001\n";
    registrar.enrollStudentInCourse("FritzTheCat", 
"CS101.001");
```

```
    cout << registrar << endl;  // or
registrar.printReport()

    cout << "EnrollStudentInCourse Bullwinkle CS101.001\n";
    cout <<  "Should fail, i.e. do nothing, since
Bullwinkle is not a student.\n";
    registrar.enrollStudentInCourse("Bullwinkle",
"CS101.001");
    cout << registrar << endl;  // or
registrar.printReport()

    cout << "CancelCourse CS102.001\n";
    registrar.cancelCourse("CS102.001");
    cout << registrar << endl;  // or
registrar.printReport()

    /*
    // [OPTIONAL - do later if time]
    cout << "ChangeStudentName FritzTheCat MightyMouse\n";
    registrar.changeStudentName("FritzTheCat",
"MightyMouse");
    cout << registrar << endl;  // or
registrar.printReport()

    cout << "DropStudentFromCourse MightyMouse
CS101.001\n";
    registrar.dropStudentFromCourse("MightyMouse",
"CS101.001");
    cout << registrar << endl;  // or
registrar.printReport()

    cout << "RemoveStudent FritzTheCat\n";
    registrar.removeStudent("FritzTheCat");
    cout << registrar << endl;  // or
registrar.printReport()
    */

    cout << "Purge for start of next semester\n";
    registrar.purge();
    cout << registrar << endl;  // or
registrar.printReport()
}
```

Technical Requirements

- Separate compilation. Each class definition should (ok, must) each go in its own header file and *all* method definitions for the class will (must) go in the associated implementation file.
  Don't forget all the issues involved with separate compilation.
- Namespace. Place all of your classes in the BrooklynPoly namespace.
  Don't forget all the issues involved with namespaces.
- Student
  Each student can be enrolled in more than one course - how about a vector of *pointers to* Course objects.
- Course
  Each course can have more than one student taking it - same idea, how about a vector of *pointers to* Student objects.
- Registrar
  In order to keep up with all the courses and students, it should have two vectors, one for courses and one for students.

Development

Incremental development is a great help. Write a little bit, then test. Try to test after you write each method. Here's how I would start.

As you go through these steps, each time you introduce a new class, you must put all it's code in a separate file and make sure the namespace issue is correctly handled.

1. Write just the Registrar constructor and test it. Of course there's not much to test and not much to see yet. [In fact, what does the constructor have to do?]
2. Write the printReport method for Registrar.
   How much more will you have to do for the printReport method? Certainly the basic class definitions for the Course and Student classes - probably constructors for them and their own print/display methods that the Registrar's class would call to accomplish it's own printReport.
3. Write the Registrar's addCourse. And test with printReport.
   How do you know that the course isn't already there?
   Where do you think the Course and Student objects should live? Suppose you just kept a vector of them, why might that prove to be a disaster? You are going to have pointers from a Course to the

Students. If the Students are in a vector, i.e. a vector<Student>, what might happen to their addresses with you push_back another Student? If the vector was full, then the underlying array would get reallocated and all the pointers to them would become invlid.
Probably writing the code that looks for the course would make a useful method.
What should it return?

4. Write the Registrar's addStudent. And test with printReport.
Yeah, the same comments about having a find method for "is this student in the U" probably hold here.
5. Write the Registrar's EnrollStudentInCourse. Yes, and you know what comes next.
6. Keep going. Bit by bit (well, method my method - methodically?) and test each bit (method) before doing the next part in the code above.
Well, actually there's only cancelCourse left to do (and purge if there is enough time).

In general, does every method in a class need to be public?
What about methods like the ones suggested to find a student or course in the registrar by it's name.

Second Part

Only after you have made sure that all of the Registrar's *required* functions work, then *if you have time in lab*, you can write a program to read and process a transaction file.

The file of transactions

Your program will read a file that contains "transactions". Each line will tell the Registrar to do something. A file might look like:

```
 PrintReport
AddCourse CS101.001
PrintReport
AddStudent FritzTheCat
PrintReport
AddCourse CS102.001
PrintReport
EnrollStudentInCourse FritzTheCat CS102.001
```

PrintReport
EnrollStudentInCourse Bullwinkle CS101.001
PrintReport
CancelCourse CS102.001
[OPTIONAL - do later if time] ChangeStudentName
FritzTheCat MightyMouse
[OPTIONAL - do later if time] PrintReport
[OPTIONAL - do later if time] DropStudentFromCourse
MightyMouse CS101.001
[OPTIONAL - do later if time] PrintReport
[OPTIONAL - do later if time] RemoveStudent FritzTheCat
[OPTIONAL - do later if time] PrintReport
PrintReport
Purge

Each transaction occurs on a separate line, ending with a newline character.
Some transactions have "parameters" like MighyMouse (a student's name) and CS101.001 (a course) in the "transaction":
DropStudentFromCourse MightyMouse CS101.001

The above file contents show all the possible transactions/commands that the file might contain.
It matches the same testing that was done in the tester code above - it's better to have a transaction file than having to constantly rewrite your tester program, eh?
Code for this part of the problem will read a line from the transaction file and call the appropriate methods in the appropriate objects to get this transaction done.
Note that it would be pretty useless to attempt this part until all the required functions are working!

Error Checking in the Transaction File

Transactions may have errors. For example, a transaction may attempt to enroll a student in a course, who is not in the school.
In the file above, the line:

`EnrollStudentInCourse Bullwinkle CS101.001`
should result in an error.

Other errors in a command, such as the line below (which appeared in the file above) which fails to mention what course the student is enrolling would also produce an error:

`EnrollStudentInCourse FritzTheCat`
The program should print an appropriate error message and then continue, processing the next line.

Third Part

If you have even more time (ok, not too likely), write the code for the [if time] items.