Project 1B: Inheritance

Part B of Project1 will let you practice a bit more with writing simple classes and become accustomed to working with multiple classes and multiple source files, while applying the new concept of Inheritance.

In part A you wrote the **`CourseMember`** `class`. Every CourseMember has a first name, last name and ID by which they can be identified. Everyone in this course is a CurseMember, however there are different *roles* a CourseMember may take. We have Students, TeachingAssistants and Instructors. These are all CourseMembers (with a first name, last name and ID), but they also have other attributes specific to their role.

For part B of Project1 you will write **3 classes**:
- `Student`
- `TeachingAssistant`
- `Instructor`

Both Students and Instructors are CourseMembers. A TeachingAssistant is not only a CourseMember but also a Student. Thus the **inheritance structure** will be as follows:
- `Student` will be a derived class of `CourseMember`
- `TeachingAssistant` will be a derived class of `Student`
- `Instructor` will be a derived class of `CourseMember`

Implementation:

For this part of Project1 I will not provide the header files on Blackboard, you must write the 3 classes (both .hpp and .cpp files **for each class**) based on the following specification (FUNCTION PROTOTYPES AND MEMBER VARIABLE NAMES MUST MATCH EXACTLY). As you implement these classes think about what is inherited and what is not (e.g. constructors are not inherited!!!)
Remember, **you must thoroughly document your code!!!**

class Student public members:

```
Student(int id, std::string first, std::string last);
std::string getMajor() const;
double getGpa() const;
void setMajor(const std::string major);
void setGpa(const double gpa);
```

class Student protected members:

```
std::string major_;
double gpa_;
```

---

class TeachingAssistant auxiliary types:

The TeachingAssistant class uses an enum (a user-defined data type) to keep track of the specific role the TA has:

```
enum ta_role {LAB_ASSISTANT, LECTURE_ASSISTANT, BOTH};
```

You may assume for initialization purposes that the default role is LAB_ASSISTANT.

class TeachingAssistant public members:

```
TeachingAssistant(int id, std::string first, std::string last);
int getHours() const;
ta_role getRole() const;
void setHours(const int hours);
void setRole(const ta_role role);
```

class TeachingAssistant private members:

```
int hours_per_week_;
ta_role role_;
```

```
class Instructor public members:

    Instructor(int id, std::string first, std::string last);
    std::string getOffice() const;
    std::string getContact() const;
    void setOffice(const std::string office);
    void setContact(const std::string contact);

class Instructor private members:

     std::string office_;
     std::string contact_;
```

Testing:

You must always test your implementation **INCREMENTALLY**!!!

What does this mean?

- Implement and test one class at a time!!!
- For each class:
    - Implement one function/method and test it thoroughly (multiple test cases + edge cases if applicable)
    - Implement the next function/method and test it …
    - …

How do you do this?

Write your own **`main()`** function to test your classes. Start from the constructor(s), then move on to the other functions. Choose the order in which you implement your methods so that you can test incrementally (i.e. implement mutator functions before accessor functions). Sometimes functions depend on one another. If you need to use a function you have not yet implemented, you can use **stubs:** a dummy implementation that always returns a single value for testing (don't forget to go back and implement the stub!!!)

In your main function you also want to test for inheritance. Think about:

- What changes must you make to the CourseMember class to support inheritance?

- **Can you access members of the base class from the derived class? Test it!!!**
- **Test calling a member function of the base class via an object to type derived. Make sure it works!**

Submission:

I split the Gradescope submission for this part of the project into 3, so you will have a Gradescope submission for each class. For each class you will **submit both interface and implementation files**. I hope you realize that to submit TeachingAssistant you must resubmit Student as well. You may assume that CourseMember is as per the interface of Project1A, and you **do not** need to **resubmit CourseMember**.

- Project1B_Student submit Student.hpp and Student.cpp
- Project1B_Instructor submit Instructor.hpp and Instructor.cpp
- Project1B_TeachingAssistant submit Student.hpp, Student.cpp, TeachingAssistant.hpp and TeachingAssistant.cpp

Your project must be submitted on Gradescope. The due date is Friday February 15 by 6pm. No late submissions will be accepted.

Have Fun!!!!!