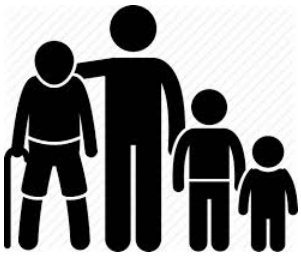
Project 1: OPP and Inheritance



Project1 will set the baseline for this course. Its purpose is to get you up-to-speed with working with multiple classes and multiple source files, while applying the new concept of Inheritance.

You will write 4 very simple classes. The first class is **`Person.`** Every Person has a first name, last name and ID by which they can be identified. Everyone in this course is a Person, however there are different *roles* a Person may take. We have **`Students`**, **`TeachingAssistants`** and **`Instructors`**. These are all `Person` (with a first name, last name and ID), but they also have other attributes specific to their role.

You will write 4 classes:
- `Person`
- `Student`
- `TeachingAssistant`
- `Instructor`

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

Implementation:

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). Remember that accessor functions (e.g. getID()) are used to access the private data members (e.g. all getID() will do is return id_). As you implement these classes think about what is inherited and what is not (e.g. constructors are not inherited!!!). Also think about the order in which constructors are called, and how/where must you explicitly call the base class constructor.
Remember, **you must thoroughly document your code!!!**

class Person public members:

```
Person(); //will initialize data members with some initial values
Person(int id, std::string first, std::string last);
int getID() const;
std::string getFirstName() const;
std::string getLastName() const;
```

class Person protected members:

```
int id_;
std::string first_name_;
std::string last_name_;
```

---

class Student public members:

```
Student();
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, FULL_ASSISTANT};
```

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

```
class TeachingAssistant public members:
     TeachingAssistant();
     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();
    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. First implement and test the Person class. Start from the constructor(s), then move on to the other functions. Instantiate an object of type Person and as you implement each method, call it in main and test that it is working correctly. 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!!! If you put the word STUB in a comment, some editors will make it more visible so you will remember to implement it later)
For example:

```
//******** STUB ************//
double Student::getGpa() const
{
    return 0;
}
```

Note: this will make much more sense as your programs become more complex, but it is very important to understand the fundamental concepts and develop good implementation/testing/debugging habits from the very beginning.

Once you are done with the Person class, you can move on to implementing Student, then TeachingAssistant, then Instructor.
In your main function you also want to test for inheritance. Think about:

- **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!**

Grading Rubric:

- **Correctness 80%** (distributed across unit testing of your submission)
- **Documentation 15%**
- **Style and Design 5%** (proper naming, modularity and organization) - there isn't much deigns for you to do in this project (thus the lower %) still you need to comply with the style and design guidelines

Submission:

You will submit all files for the 4 classes (**8 files**):

- The Person class (`Person.hpp` and `Person.cpp`)
- The Student class (`Student.hpp` and `Student.cpp`)
- The Instructor class (`Instructor.hpp` and `Instructor.cpp`)

- The TeachingAssistant class (`TeachingAssistant.hpp` and `TeachingAssistant.cpp`)

Your project must be submitted on Gradescope.

Although Gradescope allows multiple submissions, it is not a platform for testing and/or debugging and it should not be used for that. You MUST test and debug your program locally.
Before submitting to Gradescope you MUST ensure that your program compiles (with g++) and runs correctly on the Linux machines in the labs at Hunter (see detailed instructions on how to upload, compile and run your files in the "Programming Rules" document). That is your baseline, if it runs correctly there it will run correctly on Gradescope, and if it does not, you will have the necessary feedback (compiler error messages, debugger or program output) to guide you in debugging, which you don't have through Gradescope.
"But it ran on my machine!" is not a valid argument for a submission that does not compile.
Once you have done all the above you submit it to Gradescope.

The due date is Thursday June 6 by 6pm. No late submissions will be accepted.

Have Fun!!!!!