Due Date: Mar 30, 2022 @ 11:59:59

Object Oriented Programming – University Department

This assignment will focus on the concepts of Object-oriented programming in C++. In this assignment, you will create 5 classes with member variables and functions, using inheritance. You will also demonstrate multiple inheritance and write a program that creates a vector of type base pointer, which can hold the object to any derived class instance. Using polymorphism, the display functions will produce output based on the instantiated class.

Your will submit your work to Gradescope and it will be both autograded and human graded. The autograder will compile your code with g++-7.5.0.

Be sure to watch Piazza for any clarifications or updates to the assignment.

Don't forget the Assignment 4 Canvas Quiz! Your assignment submission is not complete without the quiz.

University Department Overview

Your submission must consist of 5 classes each consisting of a header file `.h` and a source file `.cpp` : `Person`, `Employee`, `Student`, `Instructor`, and `TeachingAssistant`. The classes should follow the following UML schema displayed in Figure 1.

Recall that in UML:

- + prefix denotes public.
- - prefix denotes private.
- {query} denotes a member function that does not alter the state of the class.
- = *val* denotes the default value.
- = 0 denotes a pure virtual when used for a member function.

For header files, in general:

- Be sure to have include guards.
- Don't import anything that is unnecessary.
- Include file header comments that have your name, student id, a brief description of the functionality, and be sure to cite any resource (site or person) that you used in your implementation.
- Each function should have appropriate function header comments.
  - You can use javadoc style (see doxygen) or another style but be consistent.
  - Be complete: good description of the function, parameters, and return values.
  - In your class declarations (and header files in general), only inline appropriate functions.

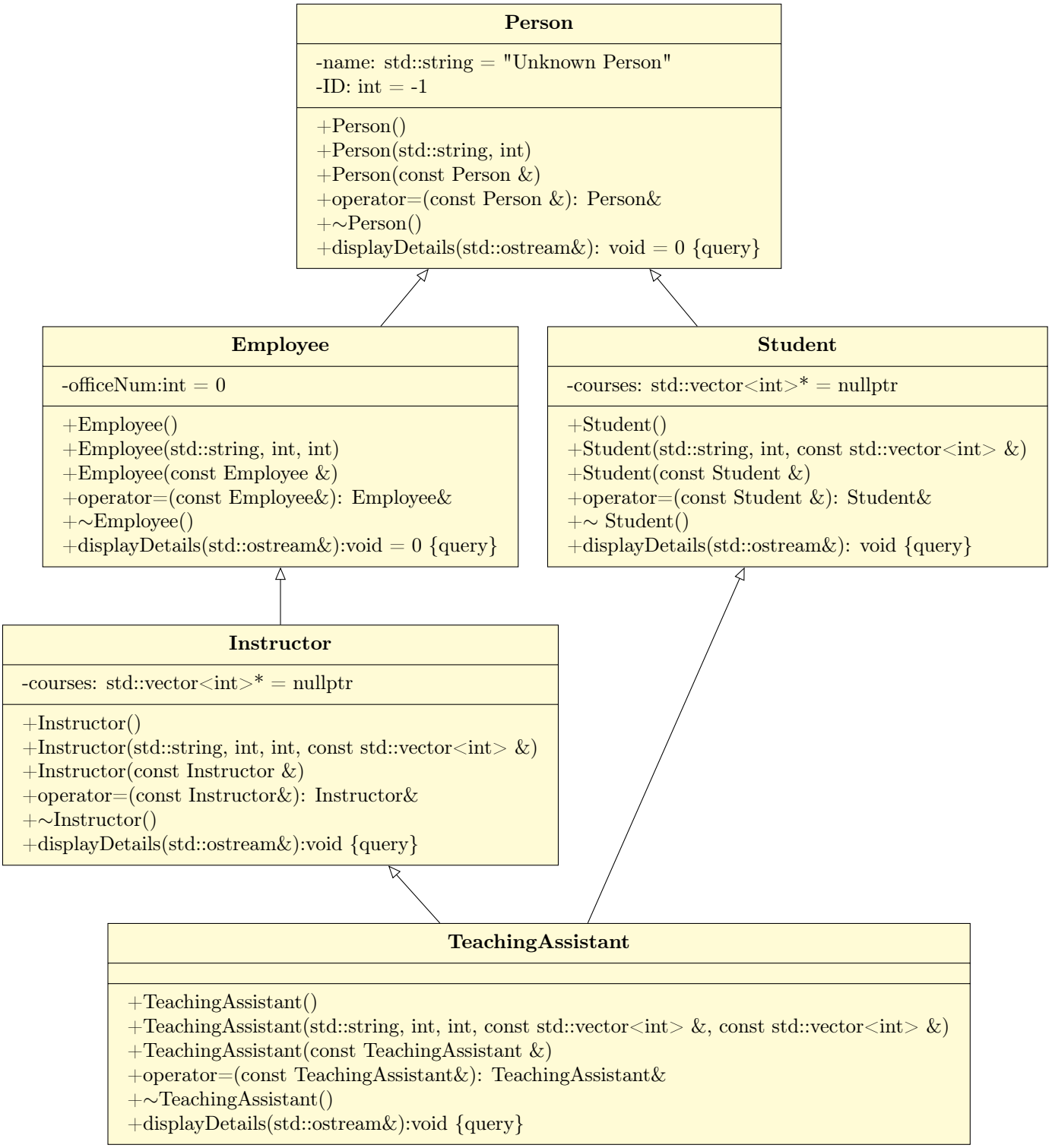

Figure 1: UML Diagram of University Department Classes

For source files, in general:

- All the function bodies not defined in the header file should be here.
- Don't clutter your code with useless inline comments (it is a code smell [Inline Comments in the Code is a Smell, but Document the Why], unless it is the why).
- Follow normal programming practices for spacing, naming, etc.: Be reasonable and consistent.
- Be sure to avoid redundant code.

For classes, in general:

- Remember the rule of 3: Explicitly define the destructor, copy operator=, and copy constructor if needed (i.e. for RAII).
- Use `= default` to explicitly use the default version of default constructor, copy constructor, destructor, and copy operator=.
- Reuse the code better by calling base class functions and constructors from derived classes.
- Do not make a member variable public or protected, unless it makes sense to do so.
- Use the keyword `override` for all functions that override a parent's method.
- Use *initializer list* for constructor delegation, constructor chaining and setting class member values.

Specific OO and Inheritance notes:

- Person and Employee are *abstract classes*.
- TeachingAssistant should only have a single copy of Person.name and Person.ID.
- displayDetails should exhibit polymorphic behaviour.
- The UML diagram (Figure 1) denotes the accessibility (private/public) of the members. You are required to respect the diagram. That is, you are not allowed to change the accessibility to make your code "work".
- All classes will have a `default` empty constructor.

Constructor Notes for the Classes:

Person(std::string, int) std::string sets the name, int sets the ID.

Employee(std::string, int, int)] std::string sets the name, 1st int sets the ID, 2nd int sets the officeNum.

Student(std::string, int, const std::vector<int> &) std::string sets the name, int sets the ID, const std::vector<int> & populates courses.

Instructor(std::string, int, int, const std::vector<int> &) std::string sets the name, 1st int sets the ID, 2nd int sets the officeNum, const std::vector<int> & populates courses.

TeachingAssistant(std::string, int, int, const std::vector<int> &, const std::vector<int> &) std::string sets the name, 1st int sets the ID, 2nd int sets the officeNum, 1st const std::vector<int> & populates Instructor::courses, 2nd const std::vector<int> & populates Student::courses.

displayDetails Implementation Notes:

Person::displayDetails A pure virtual function. While pure virtual, it is still possible to implement a default implementation that can be used by subclasses. The Person::displayDetails should output two lines both terminated by a newline (`'\n'`):

- The first line should be `"Name: "` followed by the name member.
- The second line should be `"ID: "` followed by the ID member.

Employee::displayDetails A pure virtual function. While pure virtual, it is still possible to implement a default implementation that can be used by subclasses. The Employee::displayDetails should output everything from Person::displayDetails followed by two lines both terminated by a newline (`'\n'`):

- The first line should be `"Employee details: "`.
- The second line should begin with a tab, then `"Office: "` followed by the officeNum member.

Instructor::displayDetails The Instructor::displayDetails should output everything from Employee::displayDetails followed by two lines both terminated by a newline (`'\n'`):

- The first line should be `"Instructor details: "`.
- The second line should begin with a tab, then `"Course: "` followed by the each element in the courses member separated by `", "`.

Student::displayDetails The Student::displayDetails should output everything from Person::displayDetails followed by two lines both terminated by a newline (`'\n'`):

- The first line should be `"Student details: "`.
- The second line should begin with a tab, then `"Course: "` followed by the each element in the courses member separated by `", "`.

TeachingAssistant::displayDetails The TeachingAssistant::displayDetails should output everything from Instructor::displayDetails followed by Student::displayDetails without repeating the Person::displayDetails. Hint: `stringstream` will likely be helpful in implementing this method.

demo.cpp

You have been provided a `demo.cpp` on Canvas. When your classes are properly defined, the output from the `main` function should match the `ExpectedOutput.txt` on Canvas.

Makefile

Your submission must include a `Makefile`. The minimal requirements are:

- Compile using the demo executable with the `make` command.
- Use appropriate variables for the compiler and compiler options.
- Have a rule to build the object file, with appropriate dependencies, for each module (header and source pairs).

- It must contain a clean rule.
- The demo executable must be named something meaningful (not a.out).

README.md

Your submission must include a `README.md` file (see https://guides.github.com/features/mastering-markdown/). The file should give a brief description of the program, the organization of the code, how to compile the code, and how to run the program.

Submission

You will upload all the required files to Gradescope. There are no limits on the number of submissions. See the syllabus for the details about late submissions.

Grading

For this assignment, half of the marks will come from the autograder. For this assignment, none of the test details have been hidden, but there may be hidden tests in future assignments.

The other half of the marks will come from human-grading of the submission. Your files will be checked for style and to ensure that they meet the requirements described above. In addition, all submitted files should contain a header (comments or otherwise) that, at a minimum, give a brief description of the file, your name, and wisc id.

Rubric

- Autograder Tests - 40
- README File - 5
  - Name / student id
  - Typeset/organized with markdown
  - Program description
  - Compilation instructions
  - Run instructions
  - Code organization description
- Makefile - 5
  - File header comments including name/id
  - Demo build rule with appropriate named executable
  - SummarizeGrades.o build rule
  - Clean build rule
  - Correct file name
- {Person, Employee, Student, Instructor, TeachingAssistant}.h - 3 each
  - File header comments including name/id
  - Include guards

  - All includes are required for header file
  - Function header comments with function description, return value and parameters
  - Comment detailing other elements like structs
  - Function prototypes include default values as per the specification
  - Only inline function bodies are defined in header
  - Design follows UML diagram (inheritance, accessibility, names, etc).
  - Correct use of virtual, and pure virtual.
  - Follows rule of big 3, uses default where appropriate.
  - Uses override when appropriate.

- {Person, Employee, Student, Instructor, TeachingAssistant}.cpp - 3 each
  - File header comments including name/id
  - No unused/unneeded includes
  - Consistent style (braces, spacing, etc)
  - Avoid inline comments
  - Avoid duplicate code, call parent methods and use helper functions as needed
  - Avoid c-style casting
  - Avoid gotos
  - Well-designed functions

Tackling Assignment 4

1. Create a directory for your assignment 4 project, and create the required files: {`Person, Employee, Student, Instructor, TeachingAssistant`}`.h/.cpp`, `README.md`, and `Makefile`. Download `demo.cpp` and the sample output files from Canvas.

2. Get things compiling: Set the basic classes, the function stub details, and set up your makefile. (Pro tip: When adding you function prototypes, also do the function header comment).

3. Start from the top: Implement the Person class. This is an abstract class so you can't instantiate to test it, but you could test it before making the displayDetails pure virtual. Note that the Person::displayDetails will have an implementation. Next, work on the Student class. When the Student class is working, you can check if it passes the Student autograder tests.

4. Now work on the Employee class. Just like the Person class, you can do some tests on Employee before setting Employee::displayDetails as pure virtual. Next, get the Instructor class working. When the Instructor class is working, you can check if it passes the Instructor autograder tests.

5. Finally, implement the TeachingAssistant class. This is the most complicated because it has two parent classes.

6. Don't forget to do the `README.md` file.

7. Final submission: make sure you have file comments, and name on all files. Double check the rubric, and do your final submission to Gradescope.

HAPPY CODING!