Programming Assignment: Object-Oriented Programming

Topics and references

- Inheritance and run-time polymorphism
- C++ standard library: file I/O, `std::string` and `std::vector`

Learning Outcomes

- Inheritance
- Run-time polymorphism
- Practice using C++ standard library
- Practice file I/O
- Practice C++ standard library `std::string` and `std::vector` classes
- Practice C++ standard library `std::vector` class
- Gain practice in reading and understanding code.

Introduction

This assignment aims to give you an introduction at the key components of object-oriented programming [OOP] - inheritance and dynamic binding [or run time polymorphism]. We consider the problem of grading a course that enrolls both graduate and undergraduate students. Course assessments consist of a single project and a number of homework assignments. To obtain credit, graduate students must satisfy stricter grading requirements compared to undergraduate students. As you'll see, this problem specification is common to most real-world programming problems and lends itself to an object-oriented solution. Hopefully, implementing this solution will expose you to the language features that C++ offers to support OOP.

Inheritance hierarchy

In our grading problem, we know that a graduate student's record has some attributes that are common with an undergraduate student's record - all students have an id, a name, year of birth, grade for project work, and a set of homework assignment grades. Graduate students have additional properties - a research area and the name of a faculty research advisor. Undergraduate students also have additional properties - the name of their dormitory hall and the year [*Freshman*, *Sophomore*, *Junior*, *Senior*]. Graduate and undergraduate students use different algorithms for computing the total grade and for computing the course grade. Such contexts are natural places for inheritance. The basic idea is that we can often think of one class as being just like another, except for some extensions. In this problem, common attributes of both graduate and undergraduate students are encapsulated by *abstract base class* `Student` [defined in `student.hpp`].

> ***An abstract base class cannot be directly used to create objects. Instead it is used to define an interface to derived classes. A class is made abstract by having a pure virtual function or a*** `protected` ***constructor.***

Class `Graduate` [defined in `grad.hpp`] will publicly derive from base class `Student` to capture the extra requirements for graduate students in the course. Likewise, class `Undergraduate` [defined in `ug.hpp`] will also publicly derive from base class `Student` to capture the extra requirements for undergraduate students in the course. Because classes `Graduate` and `Undergraduate` publicly inherit from class `Student`, every member of `Student` is also a member of `Graduate` and also a member of `Undergraduate` - except for the constructors and destructor. Both derived classes can add members of their own, as we do here with data members `gresearch` and `gadvisor` for class `Graduate` and data members `udorm` and `uyear` for class `Undergraduate`.

Task

In this assignment, you'll be doing the following:

1. Read a comma-separated file `students.txt` containing details about both graduate and undergraduate students.
2. Use operator `new` to dynamically allocate memory for each student object [of type `Undergraduate` or `Graduate`] specified in `students.txt`; initialize the allocated object using appropriate constructor function; and store the free store pointer to the object in a container `std::vector<Student*>`. The application requires tasks to be implemented on all students, or only on undergraduate students, or only on graduate students. To avoid down casting a `Student*` to a `Graduate*` or a `Undergraduate*` during run time, the application uses three separate containers of type `vector<Student*>`: one to store pointers to all students, the second to store pointers to objects of type `Graduate`, and the third to store pointers to objects of type `Undergraduate`. Remember only a single object is dynamically allocated for each student. However, the pointer to this student object is stored in two containers - the container of pointers to all student objects irrespective of their type and a second container of pointers to only `Undergraduate` students or only `Graduate` students.
3. Print counts of students followed by details of each student including the student's total score and letter grade. Then, process containers of type `std::vector<Student*>` to print some statistics related to undergraduate, graduate, and all students.

How to complete assignment

The code for this assignment is split across 9 source and header files. You are provided 5 files that should not be modified: `driver-oop.cpp`, `student.hpp`, `ug.hpp`, `grad.hpp`, and `process.hpp`. A partially implemented file `process.cpp` is also provided. You'll to author source files `student.cpp`, `ug.cpp`, and `grad.cpp` that will contain the definitions of static data members and member functions of classes defined in corresponding header files.

It is highly recommend that you follow these steps to complete the assignment:

1. Begin by reading source code in function `main` in `driver-oop.cpp` - this step is very important to get started on this assignment as it provides you not only the overall program flow but also an idea of the required inputs and the corresponding program output.
2. Read `student.hpp` to understand definition of abstract base class `Student`. Create file `student.cpp` and add definitions of static data member `Student::scount` and member functions of class `Student`. Note that member function `Student::age()` is defined inline in the class definition. For now, provide a skeleton definition of constructor `Student(std::string&)`. Notice that this class contains a `protected` member function that must be defined - other derived classes will rely on this function. Ensure `student.cpp`

compiles [use `-c` option and for now disable `-Werror` option of `g++`] before proceeding to the next step.

3. Read `ug.hpp` to understand the definition of derived class `Undergraduate`. Create file `ug.cpp` and add definitions of static data member `Undergraduate::ucount` and member functions of class `Undergraduate`. For now, provide a skeleton definition of constructor `Undergraduate(std::string&)`. Ensure `ug.cpp` compiles before proceeding to the next step.
4. Similarly, read `grad.hpp` to understand the definition of derived class `Graduate`. Create file `grad.cpp` and add definitions of static data member `Graduate::gcount` and member functions of class `Graduate`. For now, provide a skeleton definition of constructor `Graduate(std::string&)`. Ensure `grad.cpp` compiles before proceeding to the next step.
5. Dummy functions are already defined in `process.cpp`. Later, you'll replace the dummy definitions with concrete definitions to process containers of type `std::vector<Student*>`.
6. The next step is the hardest part of the programming task and involves the replacement of dummy definitions of constructors `Student::Student(std::string&)`, `Undergraduate::Undergraduate(std::string&)`, and `Graduate::Graduate(std::string&)`. Begin by examining file `students.txt` which contains one record per line with a record describing details of either a graduate or undergraduate student. The first few lines of `students.txt` look like this:

```
U, Ryan Borucki, 1994, 89 65 75 55 71 71 63 60 78 83, Iowa, Senior
G, Matthew Festa, 1992, 92 80 91 75 91 93 92 95 76 78 88, Ohio, Scott
Servais
```

   Lines starting with `U` describe an undergraduate student's record while lines starting with `G` describe a graduate student's record.

   The attributes common to all students includes *students' name*, *year of birth*, *project grade*, and an *unknown* number of *homework assignment grades*. *Unknown* means that different students may have been assigned different number of homework assignments. For example, the first record consists of 9 homework grades while the second record consists of 10 homework grades. Parsing of common attributes in a line [represented by a `std::string`] will be implemented by `Student` constructor: `Student::Student(std::string&);`.

   The extra requirements for undergraduate students consist of their dormitory hall name and the year of study [*Freshman*, *Sophomore*, *Junior*, and *Senior*]. For example, the second line specifies *Iowa* as the dormitory and *Senior* as the year of study. Constructor `Undergraduate::UnderGraduate(std::string&)` will use base class constructor `Student::Student(std::string&)` to initialize the `Student` part of a `Undergraduate` object.

   The extra requirements for graduate students are their research department and faculty advisor. For example, the first line specifies *Ohio* as the research area with *Scott Servais* as the name of the faculty advisor. The `Graduate` constructor `Graduate::Graduate(std::string&);` will use base class constructor `Student::Student(std::string&)` constructor to initialize the `Student` part of a `Graduate` object.

7. The process of parsing `students.txt` begins in function `parse_file` in file `process.cpp`. The function uses the file stream parameter [the file has been opened in function `main`!!!] to read each line from the file. After identifying whether a line specifies a graduate or undergraduate student record, memory for the corresponding [`Graduate` or `Undergraduate`] object is allocated and initialized on the free store. The pointer returned by operator `new` is inserted to a `std::vector<Student*>`. The application requires tasks to be implemented separately on only undergraduate and only on graduate students. To avoid down casting a `Student*` to a `Graduate*` or a `Undergraduate*` during run time, the application uses three separate containers of type `vector<Student*>`: one to store pointers to all students, the second to store pointers to objects of type `Graduate`, and the third to store pointers to objects of type `Undergraduate`. The pseudocode of this function looks like this:

```
void parse_file(std::ifstream &ifs, std::vector<Student*>& vs,
                std::vector<Student*>& vus,
                std::vector<Student*>& vgs) {
  std::string str;
  while (getline(ifs, str)) {
    char ch = str[0];
    if (ch == 'U') {
      Student *ps = new Undergraduate(str);
      vs.push_back(ps); // vs will be used when all students
                        // are to be processed
      vus.push_back(ps); // vus will be used when only undergrads
                         // are to be processed
    } else if (ch == 'G') {
      Student *ps = new Graduate(str);
      vs.push_back(ps);  // vs will be used when all student are
                         // to be processed
      vgs.push_back(ps); // vgs will be used when only grads are
                         // to be processed
    }
  }
}
```

8. In file `process.cpp`, complete the definition of function `parse_file` [described above].
9. In file `process.cpp`, implement function `print_records` to print details of all students that the container [passed as a reference] points to. Note that this function uses dynamic binding. That is, the function doesn't care about the type [`Graduate` or `Undergraduate`] of the object that each container element [of type `Student*`] points to.
10. In file `process.cpp`, implement function `print_stats` that computes and prints the following:
    1. Number of students specified in parameter of type `std::vector<Student*>` [this means each container element could point to either a `Undergraduate` or a `Graduate` object].
    2. Mean [or average] of the total score of all the students specified in the parameter

3. Sorted list of students based on their total score in *descending order*. Since the parameter is a reference to a read-only container, you'll have to make a copy of the input container and use `std::sort` to sort the new copy along with a function object or function that specifies descending order sorting criterion.

4. Sample output of this function for undergraduate students is shown below:

```
Number of students = 7
The mean of the total score = 77.7229
The sorted list of students (id, name, total, grade) in descending order of total:
3, Gabriel Martinelli, 85.3625, CR
10, Pablo Mari, 79.26, CR
8, Nicolas Pepe, 79.25, CR
5, E.S. Rowe, 79.2, CR
12, Rob Holding, 75.1, CR
1, Hector Bellerin, 75, CR
9, Bukayo Saka, 70.8875, CR
```

5. Note that this function uses dynamic binding. That is, the function doesn't care about the type [`Graduate` or `Undergraduate`] of the object that each container element [of type `Student*`] points to.

Submission Details

Please read the following details carefully and adhere to all requirements to avoid unnecessary deductions.

Source and header files

You will be submitting the following files: `student.cpp`, `ug.cpp`, `grad.cpp`, and `process.cpp`.

Compiling, executing, and testing

Download `driver-oop.cpp`, `process.hpp`, `process.cpp` [partially complete], `student.hpp`, `ug.hpp`, `grad.hpp`, input file `students.txt`, and following output files: `out1` [when input to program is 1 and output is only concerned with undergraduate students], `out2` [when input to program is 2 and the output is only concerned with graduate students], and `out3` [when input to program is 3 and the output is about all students in input file]. Follow steps from previous assignment to refactor a `makefile` and test your program.

Documentation

This module will use Doxygen to tag source and header files for generating html-based documentation. Every source and header file *must* begin with *file-level* documentation block. Every function that you declare and define and submit for assessment must contain *function-level documentation*. This documentation should consist of a description of the function, the inputs, and return value.

Submission and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit `student.cpp`, `ug.cpp`, `grad.cpp`, and `process.cpp`.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
    - F grade if your submission doesn't compile with the full suite of `g++` options.
    - F grade if your submission doesn't link to create an executable.
    - F grade if Valgrind reports even a single memory error and/or leak.
    - A deduction of one letter grade for each missing documentation block in a source file. Your submission must have **one** file-level documentation block and function-level documentation blocks for ***every function you're defining***. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing or incomplete or copy-pasted (with irrelevant information from some previous assessment) block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and one documentation block is missing, your grade will be later reduced from $A+$ to $B+$. Another example: if the automatic grade gave your submission a C grade and the two documentation blocks are missing, your grade will be later reduced from C to E.
    - The auto grader will provide a proportional grade based on how many incorrect results were generated by your submission. $A+$ grade if your output matches correct output of auto grader.