COMP 2404 -- Assignment #1

Due: Tuesday, February 12, 2019 at 12:00 pm (noon)

Goal

You will be working with an existing student auditing program throughout the term. The base code that you must start with is posted in *cuLearn*. For this assignment, you will modify the base code to improve the design and add a new entity class.

Learning Outcomes

With this assignment, you will:

- understand an existing object-oriented program
- add code to an existing code base with simple C++ classes
- work with dynamically allocated memory and pointers

Instructions

1. Understand the base code:

- Read and make sure that you thoroughly understand the existing student auditing system (SAS) program provided in the base code.
- Build the program and run it several times with different input. Use pipelining for standard input redirection from the given `in.txt` file to test the program, as you did in the tutorials. Add your own data to this file to test the program more thoroughly.

2. Modify the Course class

You will be adding new data members to the `Course` class, and you will be changing its member functions accordingly. You will also be removing some functions no longer needed because a later part of this assignment will require that you allocated `Course` objects dynamically instead of statically. Modify the `Course` class as follows:

- add two data members: the term when the course was taken, and the name of the course instructor
  - the term will be represented as an integer, using the Carleton University standard format YYYYTT, where YYYY is the four-digit year, and TT represents the term, which is 10 for winter, 20 for summer, and 30 for fall; for example, winter 2019 would be represented as 201910
  - the course instructor can be represented as a string
- modify the constructor so that it initializes the new data members from parameters
- modify the `print()` function to print out the new data members
- remove the `setGrade()` and `setCode()` functions, as your code will no longer use them once `Course` objects are dynamically allocated

3. Modify the Student class

You will modify the `Student` class to work with dynamically allocated `Course` objects, instead of statically allocated ones, and you will remove some member functions that are no longer needed. Modify the `Student` class as follows:

- modify the `courses` data member so that it holds a primitive array of `Course` **pointers** instead of `Course` objects
- modify the constructor so that it no longer initializes the course array; it still needs to initialize the other data members
- write a destructor to clean up the dynamically allocated `Course` objects
- write the `addCourse()` function that adds a course to the back (the end) of the array; the function will have the prototype: `void addCourse(Course*)`
- modify the `print()` function to work with `Course` pointers instead of objects
- remove the `setId()`, `setCourse()`, and `setNumCourses()` functions, as your code will no longer use them

4. Implement the Storage class

You will create a new class called `Storage`. This class will contain all the student information stored in the program. The `Storage` class will contain the following:

- a collection of students, represented as an array of `Student` pointers
- the number of students currently in storage
- a constructor
- a destructor to clean up the dynamically allocated `Student` objects
- an `addStu(Student*)` member function that adds a new student to the back of the array
- a `print()` member function that prints out all the student information to the screen; the printed information must include all student data, including the student’s course information; you must reuse existing functions to do this

5. Modify the main() function

You will modify the program so that the `main()` function:

- doesn’t declare an array of students anymore; instead, it will declare a `Storage` object
- prompts the user to enter the student id, and dynamically allocates a `Student` object
- prompts the user to enter the data for the student’s courses, including the new data members that you added for instruction #2; for each course, the `main()` function:
  - ○ dynamically allocates a new `Course` object with the user entered data
  - ○ adds the new course to the `Student` object using existing functions
- adds the student to storage using functions implemented in previous steps
- prints the content of the storage using a member function of that object

NOTES:

- The `main()` function will no longer be manipulating the students or the array directly; it will use the `Storage` object and its member functions instead.
- The `printStorage()` global function will no longer be needed and must be removed.

6. Test the program

- You will modify the `in.txt` file so that it provides sufficient datafill for a minimum of 15 students, each with at least 5 courses.
- Check that the student and course information is correct when the storage is printed out at the end of the program.
- Make sure that all dynamically allocated memory is explicitly deallocated when it is no longer used. Use `valgrind` to check for memory leaks.

Constraints

- your program must follow the existing design and organization of the base code
- do not use any classes, containers, or algorithms from the C++ standard template library (STL)
- do not use any global variables
- do not use structs; use classes instead
- objects must always be passed by reference, not by value
- your classes must be thoroughly documented in every class definition
- all basic error checking must be performed
- existing functions must be reused everywhere possible

Submission

You will submit in *cuLearn*, before the due date and time, the following:

- one **`tar`** or **`zip`** file that includes:
  - all source, header, and data files, including the code provided
  - a Makefile
  - a readme file that includes:
    - a preamble (program and revision authors, purpose, list of source/header/data files)
    - compilation. launching, and operating instructions

NOTE: Do **not** include object files, executables, swap files, or duplicate files in your submission.

Grading (out of 100)

Marking components:

- 10 marks: correct modifications to `Course` class
  - 6 marks: correct definition and initialization of new data members
  - 4 marks: correct changes to `print()` function
- 22 marks: correct modifications to `Student` class
  - 5 marks: correct definition of `Course` array
  - 2 marks: correct modifications to constructor
  - 7 marks: correct implementation of destructor
  - 6 marks: correct implementation of add function
  - 2 marks: correct modifications to print function

- 28 marks: correct implementation of `Storage` class
  - 5 marks: correct definition of student array
  - 2 marks: correct definition of number of students
  - 3 marks: correct implementation of constructor
  - 7 marks: correct implementation of destructor
  - 6 marks: correct implementation of `addStu()` function
  - 5 marks: correct implementation of `print()` function
- 40 marks: correct modifications to `main()` function
  - 3 marks: correct prompting for student information
  - 7 marks: correct creation and initialization of `Student` object
  - 6 marks: correct prompting for course information
  - 12 marks: correct creation and initialization of `Course` objects
  - 4 marks: correct addition of courses to student
  - 4 marks: correct addition of student to storage
  - 4 marks: correct printing of storage

Execution requirements:

- all marking components must be called, and they must execute successfully to receive marks
- all data handled must be printed to the screen for marking components to receive marks

Deductions:

- Packaging errors:
  - 10 marks for missing Makefile
  - 5 marks for a missing readme
  - 10 marks for consistent failure to correctly separate code into source and header files
  - 10 marks for bad style or missing documentation
- Major programming and design errors:
  - 50% of a marking component that uses global variables, or structs
  - 50% of a marking component that consistently fails to use correct design principles
  - 50% of a marking component that uses prohibited library classes or functions
  - 50% of a marking component where unauthorized changes have been made to the base code
- Execution errors:
  - 100% of a marking component that cannot be tested because it doesn’t compile or execute in VM
  - 100% of a marking component that cannot be tested because the feature is not used in the code
  - 100% of a marking component that cannot be tested because data cannot be printed to the screen
  - 100% of a marking component that cannot be tested because insufficient datafill is provided