COMP 2404 – Assignment #2

Due: Thursday, October 15 at 11:59 pm

1. Goal

For this assignment, you will write a C++ program to manage the data for a school with students and courses. You will implement your program using objects from the different object design categories, based on a UML class diagram provided for you.

2. Learning Outcomes

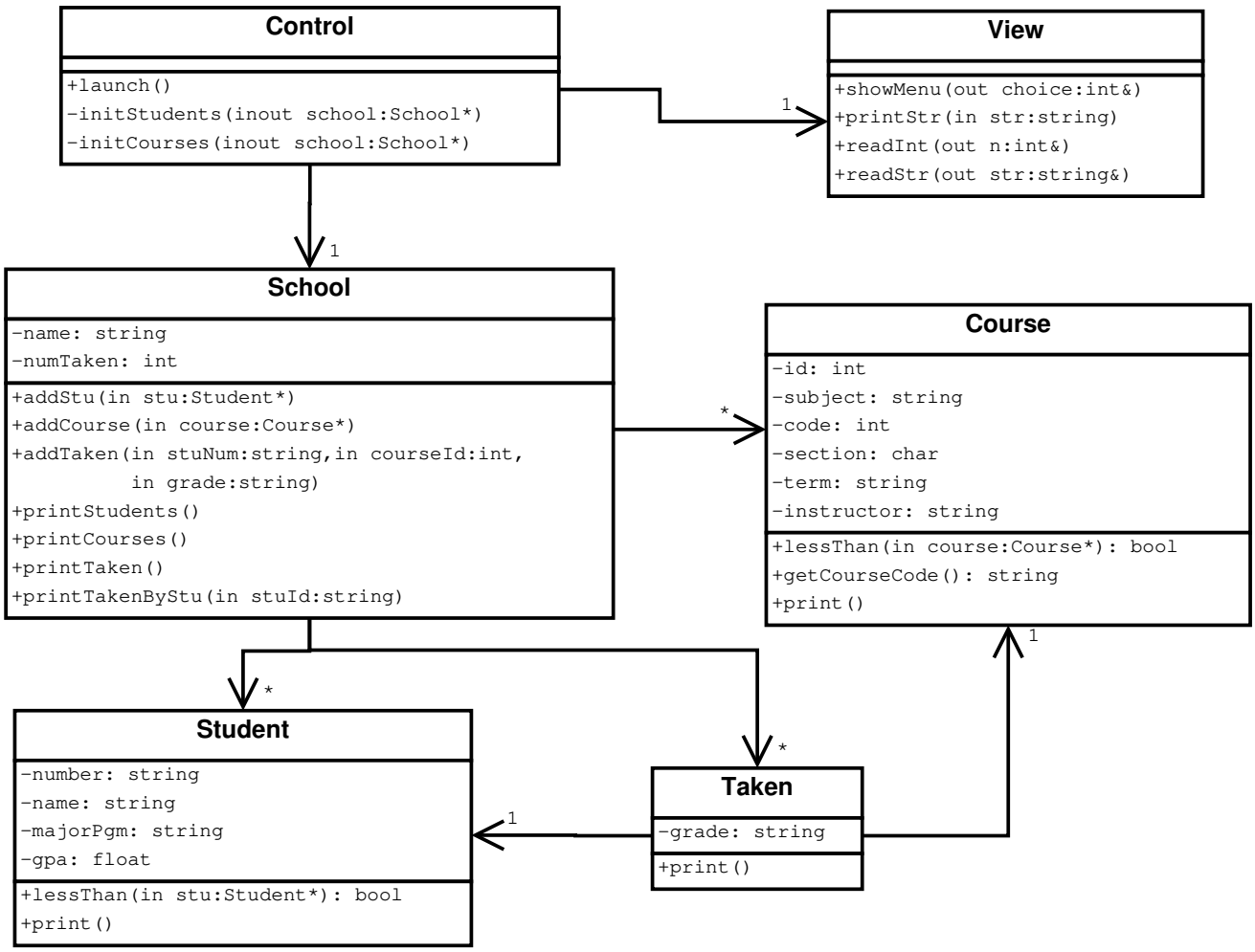
With this assignment, you will:

- practice implementing a design that is given as a UML class diagram
- implement a program separated into control, view, entity, and collection objects
- work with statically allocated and dynamically allocated arrays

3. Instructions

3.1. Understand the UML class diagram

You will begin by understanding the UML class diagram below. Your program will implement the objects and associations represented in the diagram, as they are shown. UML class diagrams are explained in the course material, in section 2.3.



Your program will implement several new classes representing a school, and its students and courses. Naturally, we want our class design to show that students take courses, and courses contain students. However, there is more than one way to design the relationship between student and course objects.

The direct way is to have a *bidirectional association* between students and courses, where each student object has a collection of the courses that the student has taken, and each course object has a collection of the students who have taken that course. However, this is actually not an efficient or scalable approach.

A better design, which we will use in this program, is to implement an **association class**. The purpose of an association class is to store information about the relationship between two *other* classes. In our program, we will implement the `Taken` class as our association class. Each `Taken` object will represent the fact that a specific student has taken a specific course, and it will store the grade that the student earned in that course.

3.2. Modify the `Student` class

You will begin with the `Student` class that we worked on during the lectures. You can find this class in the coding example posted in *cuLearn*, in section 3.1, program #4.

You will make the following modifications:

3.2.1. implement a getter member function for the student number

3.2.2. implement the `lessThan()` member function, as indicated in the UML diagram. This function compares the student on the left-hand side (the `this` student) with the student passed in as the parameter; a student is considered "less than" another if their name comes first in alphabetical order; you can use the `string` class less-than operator for this

3.3. Implement the `Course` class

You will implement a new `Course` class that contains the following data members, as indicated in the UML diagram:

3.3.1. the unique identifier of the course

3.3.2. the subject of the course; for example, the subject of this course is "COMP"

3.3.3. the course code; for example, the code for this course is 2404

3.3.4. the course section

3.3.5. the term when the course was offered; you can represent this using the first letter of the semester, followed by the last two digits of the year; for example, "F20" and "W21" would be the valid terms for this academic year

3.3.6. the course instructor

The `Course` class will contain the following member functions:

3.3.7. a constructor that takes an identifier, a course subject, code, section, term, and instructor as parameters, and initializes all the data members

3.3.8. a getter function for the course id

3.3.9. a `getCourseCode()` function that concatenates the course subject and code, and returns this value; for example, the value returned for this course would be "COMP2404"

3.3.10. a `lessThan()` member function that compares the course on the left-hand side (the `this` course) with the course passed in as the parameter; a course is considered "less than" another according to the following rules, applied sequentially:

(a) order by subject

(b) if the two subjects are the same, order by course code

(c) if the two course codes are the same, order by term

(d) if the two terms are the same, order by section

3.3.11. a `print()` function that prints to the screen all the course information

3.4. **Implement the `Taken` class**

You will implement a new `Taken` class that contains the following data members:

3.4.1. a pointer to the `Student` object representing the student that has taken the course

3.4.2. a pointer to the `Course` object representing the course taken by the student

3.4.3. the grade earned by the student in the course, represented as a letter grade

The `Taken` class will contain the following member functions:

3.4.4. a constructor that takes a student pointer, a course pointer, and a grade as parameters, and initializes all the data members

3.4.5. a getter function for the `Student` pointer

3.4.6. a getter function for the `Course` pointer

3.4.7. a `print()` function that prints to the screen the corresponding student's name, the concatenated course code (subject and code), and the grade earned

3.5. **Implement the `DynArray` class**

You will modify the `Array` class that we implemented in the coding example of section 2.2, program #1. You will rename this class to `DynArray`, and you will modify it as follows:

3.5.1. modify the array to be a *dynamically allocated* array of `Student` pointers

(a) you can refer to the coding example in section 1.6, program #5, for examples of the four different kinds of arrays

3.5.2. modify the constructor and destructor accordingly; you can assume a fixed size for the capacity

3.5.3. modify the `print()` member function as required

3.5.4. modify the `add()` function as follows:

(a) take a `Student` pointer as parameter

(b) add the given student to the array in its correct place, in *ascending* (increasing) order

(i) you must **shift** the elements in the array towards the back of the array to make room for the new element in its correct place; **do not** add to the end of the array and sort; **do not** use any sorting function or sorting algorithm

(ii) you must use the `Student` class `lessThan()` function to perform the comparison

3.5.5. implement a `bool find(string num, Student** stu)` function that searches the array to find the student with the number indicated in the `num` parameter, and returns the corresponding student in the `stu` parameter; this function returns true if no errors occurred, and false otherwise

3.6. **Implement the `StatArray` class**

You will modify the `Array` class that we implemented in the coding example of section 2.2, program #1. You will rename this class to `StatArray`, and you will modify it as follows:

3.6.1. modify the array to be a *statically allocated* array of `Course` pointers

(a) you can refer to the coding example in section 1.6, program #5, for examples of the four different kinds of arrays

3.6.2. modify the `print()` member function as required

3.6.3. modify the `add()` function as follows:

(a) take a `Course` pointer as parameter

(b) add the given course to the array in its correct place, in *ascending* (increasing) order

(i) you must **shift** the elements in the array towards the back of the array to make room for the new element in its correct place; **do not** add to the end of the array and sort; **do not** use any sorting function or sorting algorithm

(ii) you must use the `Course` class `lessThan()` function to perform the comparison

3.6.4. implement a `bool find(int id, Course** c)` function that searches the array to find the course with the identifier indicated in the `id` parameter, and returns the corresponding course in the `c` parameter; this function returns true if no errors occurred, and false otherwise

3.7. Implement the `School` class

You will implement a new `School` class that contains the following data members:

3.7.1. the school name

3.7.2. the collection of all students in the school, stored as a `DynArray` object

3.7.3. the collection of all courses in the school, stored as a `StatArray` object

3.7.4. the collection of student-course pairs, representing which students have taken which courses; this collection will be stored as a statically allocated array of `Taken` object pointers

3.7.5. the number of elements in the `Taken` collection

The `School` class will contain the following member functions:

3.7.6. a default constructor that takes a school name as parameter, and initializes all the data members

3.7.7. a destructor that deallocates the required dynamically allocated objects

3.7.8. an `addStu()` member function, as indicated in the UML diagram, that adds the given student to the student collection

3.7.9. an `addCourse()` function that adds the given course to the course collection

3.7.10. an `addTaken()` function that does the following:

(a) find the student object with the given student number

(b) find the course object with the given course id

(c) create a new `Taken` object with the found student and course objects, and with the given grade

(d) add the new object to the back of the `Taken` collection

NOTE #1: all error checking must be performed, and all errors must be handled

NOTE #2: existing functions must be reused everywhere possible

3.7.11. a `printStudents()` function that prints all the students in the student collection

3.7.12. a `printCourses()` function that prints all the courses in the course collection

3.7.13. a `printTaken()` function that prints all the `Taken` objects in the collection

3.7.14. a `printTakenByStu()` function that prints the full course information for every course taken by the given student, along with the grade earned by the student in that course

3.8. Implement the `Control` class

A skeleton `Control` class has been provided for you, and it is posted in *cuLearn* in the `a2-posted.tar` file. You will implement the `Control` class so that it contains the following data members:

3.8.1. the `School` object to be managed

3.8.2. the `View` object that will be responsible for most user I/O; the `View` class is provided for you

The `Control` class will contain the following member functions:

3.8.3. a default constructor that initializes the data members

3.8.4. an `initStudents()` member function that initializes the students contained in the school

3.8.5. an `initCourses()` member function that initializes the courses contained in the school

3.8.6. a `launch()` function that implements the program control flow and does the following:

(a) call the initialization functions

(b) use the `View` object to display the main menu and read the user's selection, until the user exits

(c) if required by the user:

(i) print out all the students in the school

(ii) print out all the courses in the school

(iii) print out all the `Taken` objects in the school's collection

(iv) print out all the courses taken by a specific student, and the grades earned; this will require using the `View` object to read from the user the student number

(v) add a new `Taken` object for a specific student and course; this will require using the `View` object to read from the user the student number, the course id, and the grade earned

NOTE: Both initialization functions have been provided for you. You must use these functions, *without modification*, to initialize the data in your program.

3.9. **Write the `main()` function**

Your `main()` function must declare a `Control` object and call its `launch()` function. The entire program control flow must be implemented in the `Control` object as described in the previous instruction, and the `main()` function must do nothing else.

3.10. **Test the program**

You must provide code that tests your program thoroughly. For this program, the use of the provided initialization functions will be sufficient. Specifically:

3.10.1. Make sure that the data you provide exercises all your functions thoroughly. Failure to do this will result in major deductions, *even if the program appears to be working correctly*.

3.10.2. Check that all the data is correct when it is printed.

3.10.3. Make sure that all dynamically allocated memory is explicitly deallocated when it is no longer used. Use `valgrind` to check for memory leaks.

4. Constraints

Your program must comply with all the rules of correct software engineering that we have learned during the lectures, including but not restricted to:

4.1. The code must be written in C++98, and it must compile and execute in the default course VM. It must not require the installation of libraries or packages or any software not already provided in the default VM.

4.2. Your program must follow correct encapsulation principles, including the separation of control, view, entity, and collection object functionality.

4.3. Your program must not use any classes, containers, or algorithms from the C++ standard template library (STL), unless explicitly permitted in the instructions.

4.4. Your program must follow basic OO programming conventions, including the following:

4.4.1. Do not use any global variables or any global functions other than `main()`.

4.4.2. Do not use `struct`s. You must use classes instead.

4.4.3. Objects must always be passed by reference, never by value.

4.4.4. Except for simple getter functions, data must be returned using output parameters, and not using the return value.

4.4.5. Existing functions must be reused everywhere possible.

4.4.6. All basic error checking must be performed.

4.4.7. All dynamically allocated memory must be explicitly deallocated.

4.5. All classes must be thoroughly documented in every class definition, as indicated in the course material, section 1.3.

5. Submission

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

5.1.1. One `tar` or `zip` file that includes:

(a) all source and header files

(b) a Makefile

(c) a README file that includes:

(i) a preamble (program author, purpose, list of source and header files)

(ii) compilation and launching instructions

NOTE: Do not include object files, executables, hidden files, or duplicate files or directories in your submission.

5.2. Late submissions will be subject to the late penalty described in the course outline. No exceptions will be made, including for technical and/or connectivity issues. Do not wait until the last day to submit.

5.3. Only files uploaded into *cuLearn* will be graded. Submissions that contain incorrect, corrupt, or missing files will receive a grade of zero. Corrections to submissions will not be accepted after the due date and time, for any reason.

6. Grading

6.1. **Marking components:**

- 2 marks: correct modifications to `Student` class
- 12 marks: correct implementation of `Course` class
- 6 marks: correct implementation of `Taken` class
- 20 marks: correct implementation of `DynArray` class
- 10 marks: correct implementation of `StatArray` class
- 40 marks: correct implementation of `School` class
- 10 marks: correct implementation of `Control` class

6.2. **Execution requirements:**

6.2.1. all marking components must be called and execute successfully in order to earn marks

6.2.2. all data handled must be printed to the screen for marking components to earn marks

6.3. **Deductions:**

6.3.1. Packaging errors:

(a) 10 marks for missing Makefile

(b) 5 marks for missing README

(c) up to 10 marks for failure to correctly separate code into header and source files

(d) up to 10 marks for bad style or missing documentation

6.3.2. Major design and programming errors:

(a) 50% of a marking component that uses global variables or `struct`s

(b) 50% of a marking component that consistently fails to use correct design principles

(c) 100% of a marking component where unauthorized changes are made to provided code

(d) 100% of a marking component that uses prohibited library classes or functions

(e) 100% of a marking component where Constraints listed are not followed

(f) up to 10 marks for bad style

(g) up to 10 marks for memory leaks

6.3.3. Execution errors: 100% of any marking component that cannot be tested because it doesn't compile or execute in the course VM, or the feature is not used in the code, or data cannot be printed to the screen, or insufficient datafill is provided.