COMP 2404 B/C – Assignment #3

Due: Thursday, March 7 at 11:59 pm

1. Goal

For this assignment, you will write a program in C++, in the Ubuntu Linux environment of the course VM, to manage the registrations of students in university courses. The program will support two menus: one for administrators, and one for students. The administrator menu allows the user to view all students, or view all courses, or view all registrations, or show the student menu. The student menu allows the user to view courses for an academic term, or view registrations for a student, or add a new registration.

NOTE: This assignment is built along the same theme as Assignment #2, and it consists of implementing different features of a university registration system. Assignment #3 *can be implemented as a stand-alone*, so you **DO NOT** need to have completed Assignment #2 in order to do Assignment #3. However, there are bonus marks available for the full integration of the code for both assignments.

2. Learning Outcomes

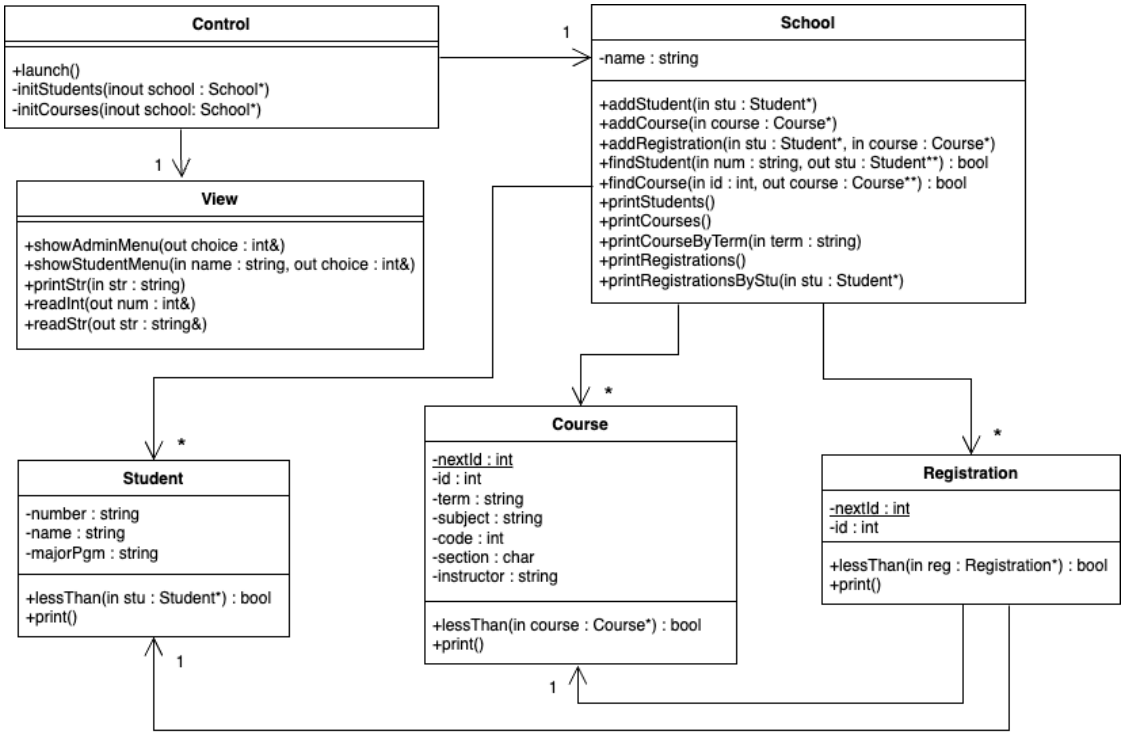
With this assignment, you will:

- practice implementing a design that is given as a UML class diagram
- implement a program separated into control, view, entity, and collection objects
- implement multiple collection classes, including arrays and a linked list
- optionally, fully integrate this program with your code for Assignment #2

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 class 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. We want our class design to show that students register for courses, and courses have registered 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 in which the student is registered, and each course object has a collection of the students that are registered for 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 `Registration` class as our association class. Each `Registration` object will represent the fact that a specific student is registered in a specific course.

3.2. Understand the provided base code

The `a3-posted.tar` file contains the `View` class that your code must use for most communications with the end user. It also contains a skeleton `Control` class, with the data initialization member functions that your code is required to call.

3.3. Implement the `Student` class

You will create a new `Student` class that represents a student in the registration system. The `Student` class will contain the data members indicated in the UML class diagram. In addition:

3.3.1. The default constructor must take three parameters and initialize the corresponding data members.

3.3.2. The `lessThan()` member function must compare student names.

3.3.3. The `print()` member function must print every data member, in the **same format** shown in the workshop video.

3.3.4. The program requires getter member functions for the student number and name.

3.4. Implement the `Course` class

You will create a new `Course` class that represents a course in the registration system. The `Course` class will contain the data members indicated in the UML class diagram. In addition:

3.4.1. The default constructor must take five parameters: the term, subject, code, section, and instructor. It must initialize all the data members accordingly.

3.4.2. The `nextId` data member must be declared as a `static` member, and it must be initialized as we saw in the in-class coding example of section 3.1, program #6, using a provided constant. This data member must be used to initialize the unique identifier of every new course.

3.4.3. The `lessThan()` member function compares whether the `Course` object should be ordered before the given parameter, when compared first by term, then by subject, then by code, then by section.

3.4.4. The `print()` member function must print every data member, in the **same format** shown in the workshop video.

3.4.5. You must provide getter member functions for the course id and term, as well as a getter member function that formats the entire course code into a single string using the following format: `"subject code-section"`, for example "COMP 2404-C".

3.5. Implement the `Registration` class

You will create a new `Registration` class that represents the registration of one student in one course. The `Registration` class will contain the data members and member functions indicated in the given UML class diagram. In addition:

3.5.1. The `nextId` data member must be declared as a `static` member, and it must be initialized using a provided constant. This data member must be used to initialize the unique identifier of every new registration.

3.5.2. The student data member must be stored as a pointer to a `Student` object already stored in the school's students collection.

3.5.3. The course data member must be stored as a pointer to a `Course` object already stored in the school's courses collection.

3.5.4. The default constructor must take two parameters, a `Student` pointer and a `Course` pointer, and it must initialize all the data members accordingly.

3.5.5. You may need a getter member function for the student pointer.

3.5.6. The `lessThan()` member function compares whether the `Registration` object should be ordered before the given parameter, when compared by their courses.

3.5.7. The `print()` member function prints out the registration id, the student's name, and the course's term and course code, as computed in instruction 3.4.5. Make sure that each part of the registration data is printed so that it aligns with the other registrations. You **MUST** follow the format shown in the assignment workshop video for this.

3.6. Implement the `CourseArray` class

You will copy the `Array` class that we implemented in the coding example of section 2.2, program #1, into a new collection class called `CourseArray`. Then, you must make the following changes:

3.6.1. Modify the collection class so that it stores `Course` object pointers as data.

3.6.2. Modify the `add()` member function so that it adds the given course to the array, in its correct position, so that the courses remain in ascending order. You must shift the elements towards the back of the array to make room for the new element in its correct place, and you must use the `Course` class's `lessThan()` member function to perform the comparisons.

3.6.3. Implement a new `bool find(int id, Course** c)` member function that searches the array for the course with the given id. If the course is found, a pointer to that `Course` object is "returned" in the `c` parameter, and the function returns success. Otherwise, the pointer returned in `c` is set to null, and the function returns failure.

3.6.4. In addition to the `print()` member function that prints all the courses in the array, you must implement a `void print(string term)` member function that prints only the courses in the array that are offered in the given academic term.

3.7. Implement the `StuArray` class

You will copy the `Array` class that we implemented in the coding example of section 2.2, program #1, into a new collection class called `StuArray`. Then, you must make the following changes:

3.7.1. Modify the collection class so that it stores `Student` object pointers as data.

3.7.2. Modify the `add()` member function so that it adds the given student to the array, in its correct position, so that the students remain in ascending order. You must use the `Student` class's `lessThan()` member function to perform the comparisons.

3.7.3. Implement a new `bool find(string num, Student** stu)` member function that searches the array for the student with the given student number. If the student is found, a pointer to that `Student` object is "returned" in the `stu` parameter, and the function returns success. Otherwise, the pointer returned in `stu` is set to null, and the function returns failure.

NOTE: Do not use class templates to merge the two array classes together, as we have not yet covered this material. You must have two separate array classes.

3.8. Implement the `RegList` class

You will implement a `RegList` class that manages a collection of registrations as a singly linked list, with a tail. The list must be implemented as we saw in class, with no dummy nodes.

3.8.1. The `RegList` class must contain a corresponding `Node` class, as we saw in the in-class coding example of section 3.1, program #8. The data stored in the list's nodes consists of `Registration` object pointers, and the list must store two data members: one pointer to the head node of the list, and another pointer to the tail node.

3.8.2. Implement a default constructor that initializes the data members to a default value.

3.8.3. Implement a second constructor with the prototype `RegList(RegList& otherList, Student* stu)`. This constructor reuses an existing member function to initialize the new list with only the registrations currently in the `otherList` parameter that belong to the given student `stu`. **Do not** make any copies of registrations! The new list must contain new nodes whose data consists of pointers to registration objects already in the `otherList`.

3.8.4. Implement destructor that deallocates *the nodes only*.

3.8.5. Implement a `void add(Registration* r)` member function that inserts the given registration into the linked list, directly in its correct position, so that the list remains in ascending order by the registration courses.

3.8.6. Implement a `void cleanData()` member function that deallocates all the data in the list.

3.8.7. Implement a `void print()` member function that prints out the details of each registration in the list. After the list is printed, the data corresponding to both the head and the tail of the list must be printed a second time, as shown in the workshop video.

3.9. Implement the `School` class

You will create a new `School` class that represents a university. This class will contain all the data members and member functions indicated in the given UML class diagram. In addition:

3.9.1. The school's collection of students must be stored as a statically allocated `StuArray` object.

3.9.2. The school's collection of courses must be stored as a statically allocated `CourseArray` object.

3.9.3. The school's collection of registrations must be stored as a statically allocated `RegList` object.

3.9.4. The default constructor must take one string parameter for the school's name, and initialize the corresponding data member.

3.9.5. The destructor must call an existing function to deallocate all the data in the registration list.

3.9.6. The `addStu()` member function adds the given student to the school's students collection.

3.9.7. The `addCourse()` member function adds the given course to the school's courses collection.

3.9.8. The `addRegistration()` member function dynamically allocates a new `Registration` object with the given parameters, and adds the new registration to the school's registrations collection.

3.9.9. The `findStu()` member function finds the student with the given student number in the school's students collection, and it returns it using the output parameter.

3.9.10. The `findCourse()` member function finds the course with the given id in the school's courses collection, and it returns it using the output parameter.

3.9.11. The `printStudents()` member function prints out the school name, and the details of each student in the students collection.

3.9.12. The `printCourses()` member function prints out the school name, and the details of each course in the courses collection.

3.9.13. The `printCoursesByTerm()` member function prints out the school name and given academic term, and the details of each course in the courses collection that is offered in the given term.

3.9.14. The `printRegistrations()` member function prints out the school name, and the details of each registration in the registrations collection.

3.9.15. The `printRegistrationsByStu()` member function creates a locally declared, statically allocated copy of the school's registrations list, containing only the registrations for the given student. Then the function prints out the school name and the student's name, and the details of each registration for that student in the local registrations list.

NOTE #1: Every member function in this class must reuse functions implemented in a previous step.

NOTE #2: If any error is encountered, a detailed error message must be printed for the end user, and the program must continue executing.

3.10. Implement the `Control` class

You will implement the `Control` class with all the data members and member functions indicated in the given UML class diagram. In addition:

3.10.1. The school data member must be stored as a `School` object pointer.

3.10.2. The constructor must dynamically allocate and initialize the school to be managed.

3.10.3. The destructor must clean up the necessary memory.

3.10.4. The `launch()` member function does the following:

(a) call the initialization functions provided in the `a3-posted.tar` file; you must use these functions, *without modification*, to initialize the data in your program

(b) use the `View` object to repeatedly print out the administrator menu and read the user's selection, until the user chooses to exit

(c) from the administrator menu, if required by the user:

(i) print out all the students in the school's students collection

(ii) print out all the courses in the school's courses collection

(iii) print out all the registrations in the school's registrations collection

(iv) use the `View` object to show the student menu

(d) if the user chooses to enter the student menu, use the `View` object to prompt the user to enter a student number, and find the corresponding student in the school's students collection; the user must be prompted until a valid student number is entered; all the operations in the student menu must be performed for that selected student

(e) use the `View` object to repeatedly print out the student menu and read the user's selection, until the user chooses to exit; exiting the student menu **must** return the user to the administrator menu, and not exit the program

(f) from the student menu, if required by the user:

(i) use the `View` object to prompt the user to enter an academic term, and print the courses from the school's courses collection for that term only

(ii) print the registrations from the school's registrations collection for the selected student only

(iii) use the `View` object to prompt the user to enter a course id, find the corresponding course in the school's courses collection, and add a new registration for the selected student and the found course to the school's registrations collection

NOTE: The above functionality must reuse existing functions, everywhere possible.

3.11. 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.

3.12. Packaging and documentation

3.12.1. Your code must be correctly separated into header and source files, as we saw in class.

3.12.2. You must provide a `Makefile` that separately compiles each source file into a corresponding object file, then links all the object files to produce the program executable. Your `Makefile` must also include the `clean` target that removes all the object files and the program executable. **Do not** use an auto-generated Makefile; it must be specific to this program.

3.12.3. You must provide a plain text `README` file that includes:

(a) a preamble (program author, purpose, list of source, header, and data files, if applicable)

(b) compilation and launching instructions, including any command line arguments

3.12.4. Use either the `tar` or the `zip` utility in Linux to package into one `.tar` or `.zip` file **all the files** required to build and execute your program.

3.12.5. Do not include any additional files or directories in your submission.

3.12.6. All class definitions must be documented, as described in the course material, section 1.3. Please **DO NOT** place inline comments in your function implementations.

3.13. Integration with Assignment #2 code

To earn the full 5 bonus marks for integration, your program must fully meet all of the criteria found in all the instructions for both assignments, and it must execute and print all data perfectly.

This means that the main menu and all features from Assignment #2 must be integrated into the student menu as additional options. Where there are conflicting instructions between the two assignments, the more advanced option must be implemented.

4. Constraints

Your design and implementation must comply with all the rules of correct software development and programming conventions that we have learned during the course lectures, including but not restricted to:

4.1. The code must be written using the C++11 standard that is supported by the course VM, and it must compile and execute in that environment. It must not require the installation of libraries or packages or any software not already provided in the course VM.

4.2. Your program must comply with the principle of least privilege, and it must follow correct encapsulation principles, including the separation of control, view, entity, and collection object functionality.

4.3. Your program must not use any library classes or programming techniques that are not used in the in-class coding examples. Do not use any classes, containers, or algorithms from the C++ standard template library (STL), unless explicitly permitted in the instructions.

4.4. If base code is provided, do not make any unauthorized changes to it.

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

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

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

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

  4.5.4. Functions must return data using parameters, not using return values, except for getter functions and where otherwise permitted in the instructions.

  4.5.5. Existing functions and predefined constants must be reused everywhere possible.

  4.5.6. All basic error checking must be performed.

  4.5.7. All dynamically allocated memory must be explicitly deallocated.

4.6. You may implement helper classes and helper member functions, but only if the design is consistent with the provided instructions. Design and implementation choices that undermine the learning outcomes will not earn any marks.

5. Submission Requirements

5.1. You will submit in *Brightspace*, before the due date and time, one `tar` or `zip` file that includes all your program files, as described in the **Packaging and documentation** instructions.

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 *Brightspace* will be graded. Submissions that contain incorrect, corrupt, or missing files will be graded as is. Corrections to submissions will not be accepted after the due date and time, for any reason.

6. Grading Criteria

- 35 marks: code quality and design
- 35 marks: coding approach and implementation
- 25 marks: program execution
- 5 marks: program packaging
- *5 marks*: *bonus marks for full integration with Assignment #2 code*