ST. CHARLES COMMUNITY COLLEGE

CPT-281 - Introduction to Data Structures with C++

Team Project 1 (30 Points)

Teamwork Guideline

- Each team can choose either **Project 1A** or **Project 1B**.
- Allowed programming language: **C++**
- Source code should be stored and well maintained in a GitHub project repository. Every team member should contribute to the repository. The repository should be able to be downloaded as a Microsoft Visual Studio project.
- Project report should be a `.doc`, `.docx` or `.pdf` file that is stored in the GitHub project repository. The report should include the following sections:
  1) There should be a **cover page** that shows the project name and all the team members' names. The cover page should be designed professionally.
  2) Show how the system is designed by your team. You need to explain your design, list all data structures you choose to implement the system and explain the role of each data structure.
  3) Draw a **UML class diagram** for your system (please study UML by yourself). Clearly show the logic relationship among all classes in the diagram. For example, class `Movie_List` is an aggregation of class `Movie`, which is a derived class of `Media`.
  4) Show at least **two test cases**. Each test case should contain a sequence of input data and operations. You need to give expected output and compare with the actual output.
  5) In a separate page, clearly list each team member's contribution to the project.
  6) Finally, you need to discuss what improvements to the system could be done in future.
- You **cannot** change team or create new team without the instructor's permission. If a team member refuses to cooperate, please first have a conversation with him/her. If it still does **not** work, then please let the instructor know as early as possible. If you tell the instructor just one day before the project's deadline, saying that one team member **never** replied to emails and **never** did the assigned jobs, the instructor can hardly help you and the project grades of all team members will be suffered.
- "Do Not Cooperate" includes (but not limited to):
  1) Refusing to communicate with other team members, e.g., **never** replying to emails (or reply very late), **never** attending team project meetings (or always show up late for 30 minutes).
  2) Refusing to complete his/her assigned jobs on time so that other team members **cannot** work in the next step, since some jobs can start only after some other jobs are done.

3) Sending "completed" code to others that does **not** compile or does **not** make any sense. Anyone **must** test the completed code before sending to other team members to make sure that the code works properly.
4) Being rude (or in a very unprofessional manner) to other team members, e.g., always having lots of excuses for **not** finishing the assigned jobs on time.

- Source code's value is 20 points; project report's value is 10 points. In principle, all team members will receive the same project grade. However, if the grader believes that a team member did much less job than others, the team member will receive a lower grade than others. If the grader believes that a team member did **nothing** (or almost nothing) in the project, the team member will receive **zero** credits for the project.
- Every team member **must** do some coding job. You **cannot** let a team member writing documents only without doing any coding work.
- Some grading policies:

1) The grader will download and run your source code. If your code does **not** compile, you will lose <u>at least 20 points</u> and your code will **not** be further graded.
2) At least 20% of your code should be comments. All variable names, function names, and class names should make good sense. You need to let the grader put the least effort to understand your code. The grader will take off points, no matter whether your code passes all the test cases, if he/she has to put extra unnecessary effort to understand your code.
3) All the files and folders need to be well organized in the repository. There should be **no** useless files or useless pieces of code.
4) All the diagrams in your project report **must** be nicely cropped, clean and clear, with **no** unnecessary margins, watermarks, logos, or backgrounds.

- All your work (e.g., source code, project report) should be stored and well maintained in the GitHub project repository. On Canvas, please only submit the URL of your project repository. One submission per team.

Develop a system to maintain two lists of movies: movies "showing" in the theater and movies "coming" to the theater.

Technical Requirements

- (Weight: 10%) Keep track of the movies using doubly-linked lists.
- (Weight: 10%) Use iterators to iterate through the lists.
- (Weight: 5%) Display the movies ("showing" and "coming").
- (Weight: 10%) Add a new movie (to the "coming" list).

  However, do **not** add the movie in these cases:

  1) The dates (receive or release date) are invalid.
  2) The release date is earlier than or equal to the receive date.
  3) The movie already exists in the list.
- (Weight: 5%) Start showing movies in the theater with a specified release date. You first need to find all the movies with the specified release date. Then, remove it from the "coming" list and add it to the "showing" list.

  However, do **not** show the movie in these cases:

  1) The movie does **not** exist in the "coming" list.
  2) The movie already exists in the "showing" list.
  3) The specified release date is invalid.
- (Weight: 10%) Edit a movie in the "coming" list (e.g., update the release date or description). You first need to find the movie with the specified name.

  However, do **not** edit the movie if it does **not** exist in the "coming" list.
- (Weight: 10%) Keep the "coming" list of movies ordered by release date (in non-decreasing order).
- (Weight: 10%) On demand, count the number of "coming" movies with release date earlier than a specified date.

  However, do **not** count if the specified date is invalid.
- (Weight: 15%) Your program should be **menu-based**. Therefore, the user can choose which command to run (display movies, add movies, edit release dates, edit movie description, start showing movies in the theater, number of movies before a date, save, exit). Just show a **text menu** in the console; a GUI is **not** required.
- (Weight: 10%) Initially, your program should read movies from a file, and populate your movie lists. It is your job to design the format of your input file.
- (Weight: 5%) Write a method that overwrites the file of movies to reflect the changes (e.g., newly added movies, edited movies, movies started to show in the theater). The method can be called on demand.

Facts and Assumptions

- A movie has the following attributes:
  1) release_date (Date type). You probably need to create your own Date class.
  2) name (string type).
  3) description (string type).
  4) receive_date (Date type).
  5) status (enum type). This data field tells if a movie is received or released.
- Movies in the "showing" list have status of released; movies in the "coming" list have status of received.
- The file that keeps track of the movies is a plain text file. Based on the example below, you need to design your own input file format.

  ```
  Glass, 01/18/2019, Drama/Fantasy, 01/12/2019, released
  Miss Bala, 02/01/2019, Mystery/Thriller, 01/17/2019, received
  ```

  In the example above, each line stores a movie. The information of the movie is structured like this: name, release_date, description, receive_date, status.
- You can use either the list class in C++ STL or the Linked_List class implemented in the lectures.

Project 1B - Addition of Two Polynomials

A **polynomial** can be represented as an ordered list of **terms**, where the **terms** are ordered by their **exponents**. The addition of two polynomials is performed by adding **terms** with the same exponent together, generating a new polynomial as the addition result (see the example below).

$$(3x^4 + 2x^2 + 3x + 7) + (2x^3 + 4x + 5) = (3x^4 + 2x^3 + 2x^2 + 7x + 12)$$

Technical Requirements

- (Weight: 10%) Use doubly-linked lists to keep track of the polynomials.
- (Weight: 10%) Use iterators to iterate through the lists.
- (Weight: 30%) Your program needs to ask the user to enter polynomials in a user-friendly fashion. For instance, $3x^3 - x^2 + 1$ should be entered as 3x^3–x^2+1. You may assume that there is only one variable (which is x) and there are **no** spaces.

  Here are examples of some polynomials:

User-Entered Text	Polynomial in Mathematical Format
-x+5+x^2-10	$x^2 - x - 5$
5x-5x^-2+10-5x+x^2	$x^2 + 10 - 5x^{-2}$

  When the program reads a polynomial, sort the terms by exponent in decreasing order. In addition, if the polynomial has terms with equal exponents, make sure you add the coefficients of that term. Therefore, the polynomial should be sorted by term exponent, and there should be **no** multiple terms with the same exponent. Also, do **not** keep track of terms with zero coefficients.
- (Weight: 10%) You need to define a class **`Term`** that contains the exponent and coefficient. This class should overload the comparison operators; the term with greater exponent is greater.
- (Weight: 20%) Add two polynomials.
- (Weight: 10%) Store the result in a new list and show it to the user in a user-friendly fashion.
- (Weight: 10%) Your program should be **menu-based**. Therefore, the user can choose which command (e.g., enter first polynomial, enter second polynomial, add polynomials, show result) to run. Just show a **text menu** in the console; a GUI is **not** required.

Facts and Assumptions

- A **`Polynomial`** is a list (doubly-linked list) of **`Terms`**.
- The coefficients and exponents are all integers but may be more than one digit.
- You may assume that user enters only one polynomial at a time.
- You may assume that user will **not** enter invalid polynomials.
- You can use either the `list` class in C++ STL or the `Linked_List` class implemented in the lectures.