Programming Project: Recommender

CPSC 1020 – Spring 2024 – version 4/6/24

School of COMPUTING Clemson University

Introduction

For this project, you will implement a recommendation system, like what Amazon does to recommend books, products, movies or music to users based on other user recommendations.

Background:

If you've ever bought a book online, the bookseller's website has probably told you what other books you might like. This is handy for customers, but also very important for business. In 2010, online movie-rental company Netflix awarded one million dollars to the winners of the Netflix Prize (https://en.wikipedia.org/wiki/Netflix_Prize). The competition simply asked for an algorithm that would perform 10% better than their own algorithm. Making good predictions about people's preferences was that important to this company. It is also an important current area of research in machine learning, which is part of the area of computer science called artificial intelligence.

What you will do:

So how might we write a program to make recommendations for books? Consider a recommender named Carlos. How is it that the program should predict books Carlos might like? The simplest approach would be to make almost the same prediction for every customer. In this case the program would simply calculate the average rating for all the books in the dataset and display the highest rated book. With this simple approach no information about Carlos is used.

We could make a better prediction about what Carlos might like by considering his actual ratings in the past and how these ratings compare to the ratings given by other recommenders. Consider how you decide on movie recommendations from friends. If a friend tells you about several movies that s(he) enjoyed and you also enjoyed them, then when your friend recommends another movie that you have never seen, you probably are willing to go see it. On the other hand, if you and a different friend always tend to disagree about movies, you are not likely to go to see a movie this friend recommends.

Your task for this project is to write a program that reads the name of a ratings file from the command line and takes people's book ratings and makes book recommendations to them using both techniques described above. We will refer to the people who use the program as "users".

Terminology

Program_user: the person running your program who is looking for book recommendations

recommender: the user who made a recommendation; these come from the recommendation file and are compared against the requested_user

requested_user: recommender that the program user is asking for their recommendations to be based upon

books: list of the names of all the books known to your program, read in from a file, the name of the file comes from the command line

recommender_ratings_map: a dictionary containing each recommender and their list of ratings for each book

bookAverages: list of books along with their rating average

similarList: used to compute the top 3 ratings based on similarity among recommenders

recommenderSimAvg: takes top 3 most similar recommenders, averages the books that are non 0

commands: a list of commands the program must recognize and process; these are: averages, debug, dotprod, ratings, recommend, similar, quit

ratings_large.dat: a large text file of book ratings used in testing

ratings_small.dat: a small text file of books ratings used in testing

Logic

Your program will read recommendation data from a file (name of this data file comes from command line) that contains sets of three lines. The first line will contain a recommender name, the second line a book title and the third line the rating that recommender gave the book. Rating is an integer value from 1 to 5 and -1 to -5. Here are the names of 2 ratings files you can test with: ratings_small.dat and ratings_large.dat.

BEFORE THINKING ABOUT THE MAIN PROGRAM - focus on implementing the class specified in Recommend.h. Then work on implementing main.

When your program starts it should read through the file and create a list with one occurrence of each book from the file. For example, the file ratings_small.dat would produce the following list:

['1984', 'Animal Farm', 'Cats', 'Harry Potter', 'Lord of the Rings', 'Watership Down']

We suggest that you create this list by putting all books in the file into a ***set*** and then convert that ***set*** to a ***vector***. https://www.geeksforgeeks.org/convert-set-to-vector-in-cpp/

Once you have this ***vector***, create a ***dictionary (map)*** to store the rating data and loop through the file again. This time add each person in the file as a key to the dictionary. The value that is associated with them should be a list the same length as the list of books you created in your first pass through the file. You should store the rating at the same index that book's name appears at in in the list of books. For example, when the first three lines of the file ratings_small.dat are read, we would add a mapping from the key Bob to a value of [0, 0, 5, 0, 0, 0]. Books that the recommender has not rated should be represented by 0s.

The dictionary created with the file ratings_small.dat would be as follows:

{ 'Bob': [0, 0, 5, -3, 5, 0], 'Carlos': [-5, 1, 0, 5, 0, 0], Kalid: [0, -3, 0, 3, 0, 5], Suelyn: [5, -3, 0, 1, 0, 0] }

Menu Commands for Main Program

Now your program is ready to make recommendations. It should then wait for the program-user to enter a command.

- **averages - display average ratings for each book**
- **books – display list of books**
- **debug - toggles debugging outputs**
- **dotprod - display current dot products**
- **file – display ratings in fancy format**
- **menu – display menu of commands**
- **names – display list of recommender names**
- **ratings - display ratings**
- **recommend <name> - display book recommendations based upon recommender <name>**
- **similar - display current similarities**
- **simavg – display averages for current similarities**
- **quit - exits the program**

DO NOT OUTPUT ANY PROMPTS UNLESS DEBUGGING IS TURNED ON. This will mess up the autograding.

The program should repeat processing commands until quit is entered.
Here is what each command should do:

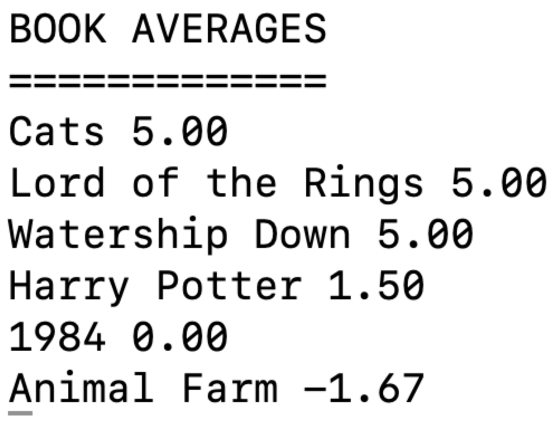
Command: averages

Display all of the books sorted by average rating from highest to lowest. If there is a tie then list books in alphabetic order.

We suggest that you figure this out by building up a list of pairs containing the average rating for a book first and the title of that book second. You can build up this list by going through the list of books one at a time and for each person in the dictionary adding up their rating of that book and counting how many people in the dictionary rated it something other than 0. The average score for that book is the sum scores divided by the count of non-zero ratings. Once you have created this list you can sort it.

Since the averages will stay the same throughout the run of the program you should calculate them once in the Recommend constructor.

Example output for this commend:

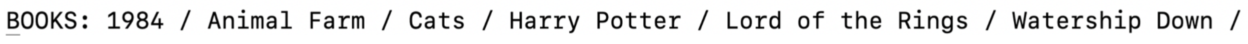


```
BOOK AVERAGES
=============
Cats 5.00
Lord of the Rings 5.00
Watership Down 5.00
Harry Potter 1.50
1984 0.00
Animal Farm -1.67
```

Command: books

Display an alphabetical list of all books.

Example output for this command:



```
BOOKS: 1984 / Animal Farm / Cats / Harry Potter / Lord of the Rings / Watership Down /
```

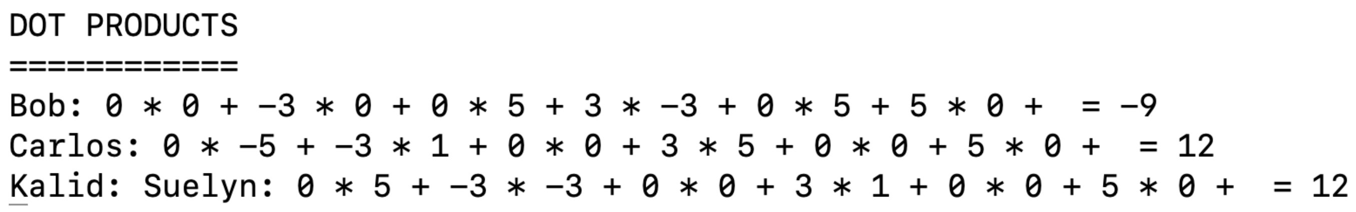
Command: debug

Turns the DEBUG option on and off. Works as a toggle switch. If it is on then turn it off. It off then turn it on. Easiest way to do this is: DEBUG = !DEBUG; where DEBUG is your toggle switch variable which is a bool. No output is produced from this command.

Command: dotprod

Display the current dot product calculation. This will be a HUGE HELP in debugging. Current means the recommendation that was most recently made. If NO recommendation has been requested then this should display “Request a recommendation”.

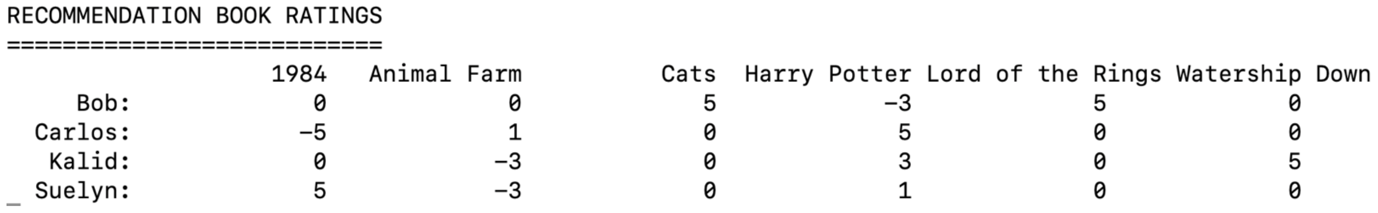
Example output for this command:



```
DOT PRODUCTS
============
Bob: 0 * 0 + -3 * 0 + 0 * 5 + 3 * -3 + 0 * 5 + 5 * 0 +  = -9
Carlos: 0 * -5 + -3 * 1 + 0 * 0 + 3 * 5 + 0 * 0 + 5 * 0 +  = 12
Kalid: Suelyn: 0 * 5 + -3 * -3 + 0 * 0 + 3 * 1 + 0 * 0 + 5 * 0 +  = 12
```

Command: file

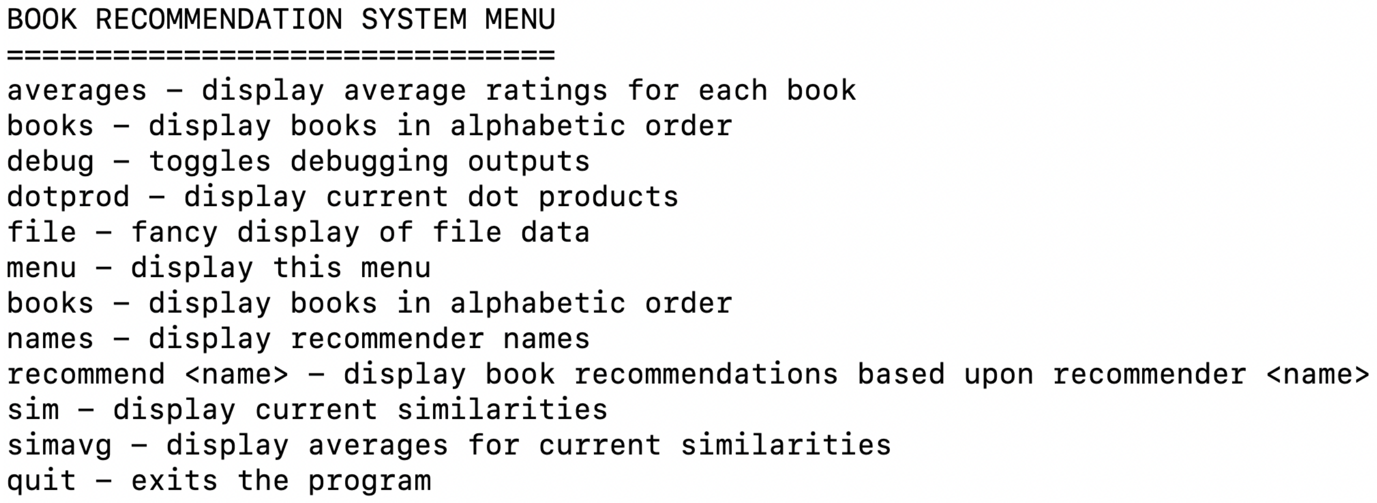
Display the data that the recommendation system is using, aka, what was read in. Here’s an example of the output. NOTE: use tabs between columns.



```
RECOMMENDATION BOOK RATINGS
==========================
                1984   Animal Farm          Cats  Harry Potter Lord of the Rings Watership Down
     Bob:          0             0             5            -3             5             0
  Carlos:         -5             1             0             5             0             0
   Kalid:          0            -3             0             3             0             5
  Suelyn:          5            -3             0             1             0             0
```

Command: menu

Display the following menu.

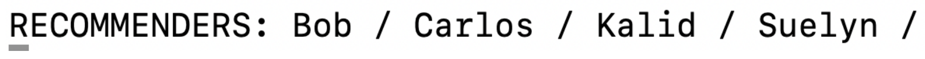


```
BOOK RECOMMENDATION SYSTEM MENU
===============================
averages - display average ratings for each book
books - display books in alphabetic order
debug - toggles debugging outputs
dotprod - display current dot products
file - fancy display of file data
menu - display this menu
books - display books in alphabetic order
names - display recommender names
recommend <name> - display book recommendations based upon recommender <name>
sim - display current similarities
simavg - display averages for current similarities
quit - exits the program
```

Command: names

Display an alphabetical list of all recommender names.

Example output for this command:

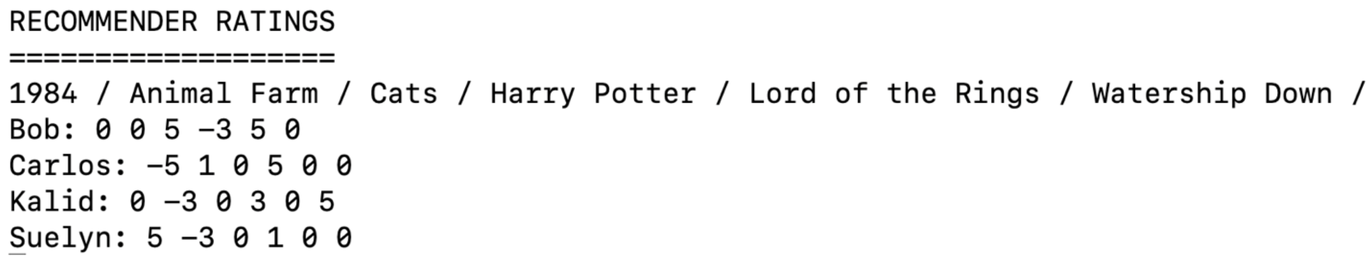


```
RECOMMENDERS: Bob / Carlos / Kalid / Suelyn /
```

Command: ratings

It should show the following:
- book titles in order from the vector
- recommenders and book ratings vector

here's an example from ratings_small.dat:



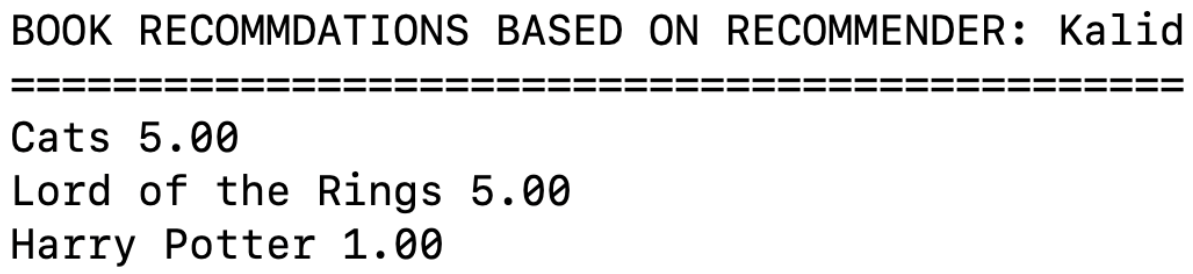
```
RECOMMENDER RATINGS
===================
1984 / Animal Farm / Cats / Harry Potter / Lord of the Rings / Watership Down /
Bob: 0 0 5 -3 5 0
Carlos: -5 1 0 5 0 0
Kalid: 0 -3 0 3 0 5
Suelyn: 5 -3 0 1 0 0
```

Command: recommend <name>

When the program-user selects the recommend option the program should read the <name> that the program-user wants recommendations for. If the <name> of the recommender isn't in the dictionary of ratings, the program should output the same thing as when the program-user selects the averages option.

If the <name> the program-user inputs is in the dictionary of ratings the program should use the data in the dictionary to find the other recommenders that are most similar to the recommender (requested) you are looking for recommendations for. Refer to "How to calculate recommendations" below.

Here's an example output for "recommend Kalid":

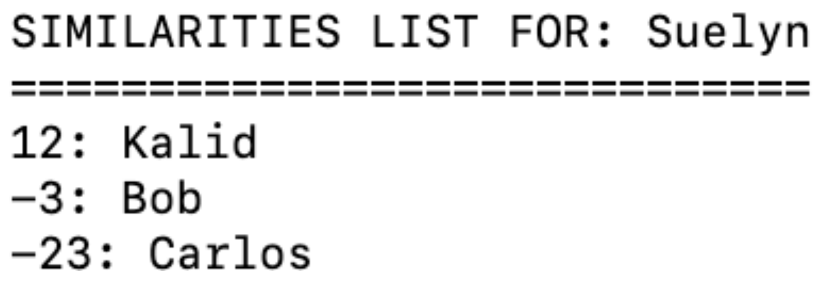


```
BOOK RECOMMDATIONS BASED ON RECOMMENDER: Kalid
==============================================
Cats 5.00
Lord of the Rings 5.00
Harry Potter 1.00
```

Command: similar

Display the similarList map – This will be very useful for debugging. If NO recommendation has been requested then this should display “Request a recommendation”.

Here’s an example output:

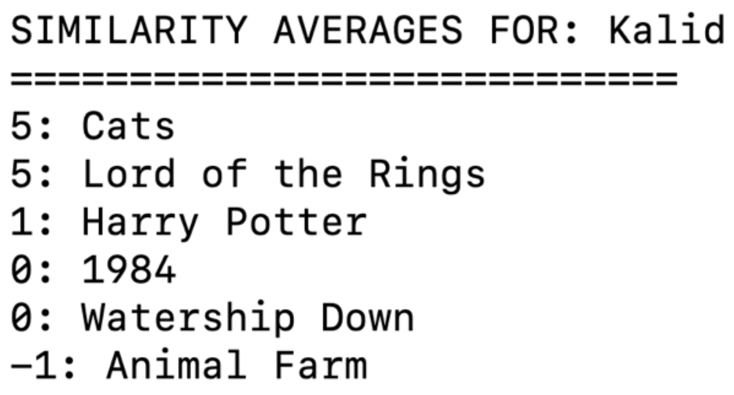


```
SIMILARITIES LIST FOR: Suelyn
=============================
12: Kalid
-3: Bob
-23: Carlos
```

Command: simavg

Display the simAvg map – This will be very useful for debugging. If NO recommendation has been requested then this should display “Request a recommendation”.

Here’s an example output:



```
SIMILARITY AVERAGES FOR: Kalid
============================
5: Cats
5: Lord of the Rings
1: Harry Potter
0: 1984
0: Watership Down
-1: Animal Farm
```

How to calculate recommendations

The first step to do this is to calculate the similarities between the requested recommender and the other recommenders. We will use the dot product between the recommeders' lists of ratings to calculate their similarity. This means that we will multiply each element in your user's list with the element at the same index in the other recommender's list and sum the result. For example, if we were looking for a recommendation for Kalid we would do the following to calculate his similarity to Carlos: [refer to the "how to solve" PDF provided in canvas, and/or watch the "how to solve" video provided by your professor]

Compute this similarity for each recommender in the dictionary. Store pairs containing first the similarity number and second the name of the other recommenders in a list. You can use the sort function to sort this list.

Note that the requested recommender that you are looking for will always be in the dictionary. We are not interested in how similar the requested recommender is to themselves. You may find it helpful not to add the requested recommender to the list or to remove them after sorting. Note that the requested recommender will always be most similar to themselves.

Now that we have a list of the most similar recommenders, we can use this to figure out which books to recommend. To generate recommendations take an average of the ratings of the three recommenders with the highest similarity to the requested recommender you are looking for.

To average the ratings create a new list the same length as the list of books and filled with 0s. Loop through this list. For every index of this list loop through the first three recommenders, add up their ratings and then divide by the number of non-zero ratings. If there are no non-zero ratings for a book you should not divide as you will get a divide by 0 error.

Once you have calculated these averages, create a list of pairs that contain first the average rating and then the book title for all books that have non-zero ratings in the averages list. Then, sort this list. Now you have a list of books to recommend.

Note that it must be possible for the program-user to choose options multiple times in any order and get the correct results each time.

OUTPUT FORMAT for ratings display:

When you display a rating, use fixed precision with 2 decimal places. For example: cout << fixed << setprecision(2) << rating

Development Strategy and Hints:

Completing Lab 10 and Lab 11 will help you with this project.
Use generous print statements to view the state of your containers.
Use the small input file (ratings_small.dat) to help you debug.
Once you are getting correct results, move up the larger data file, ratings_large.dat.

Style Guidelines and Grading:

Part of your grade will come from appropriately using data structures and following the algorithms described in this document.

Redundancy is always a major grading focus; avoid redundancy and repeated logic as much as possible in your code. Design your code following the UML design described below.

Follow good general style guidelines such as: appropriately using control structures like loops and if/else statements; avoiding redundancy using techniques such as functions, loops, and if/else factoring; good variable names, and naming conventions. You may have no global variables (except constants).

Create a class per the UML design with appropriate member variables, constructors and member functions.

Comment descriptively at the top of your program, each class, and on complex sections of your code. Comments should explain each function's behavior, parameters and returns.

D version - constructor works as specified and then responds to the "ratings" command.

C version - all of D plus able to process input as specified and get averages working

A/B version - works according to specifications, as determined by GradeScope

NOTE: for D and C level it will not pass any specific tests, but I will take a look at your output to determine your grade level

HELPER LINKS

vector - already covered in this course, great way to create lists

pair - https://www.geeksforgeeks.org/pair-in-cpp-stl/

set - https://www.geeksforgeeks.org/set-in-cpp-stl/

map - https://www.geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl/

tuple - https://www.geeksforgeeks.org/tuples-in-c/

multiple field sorting - https://www.geeksforgeeks.org/structure-sorting-in-c/

file redirection (for testing) - https://www.geeksforgeeks.org/input-output-redirection-in-linux/, https://www.loom.com/share/fd61f40f783d46e7bca2b3cceaad557d

debugging tip (this is REQUIRED if you need help from teacher or TA) - https://www.loom.com/share/f091096c53184f8b8f3390f17e102bfe

removing extra newline character in text files - https://www.loom.com/share/7bccb80d2a3e4131ad45c6427e3f1456

Class Specification / UML

Implement the following class to build Recommend.

```
Named data structures to simplify code
#define BOOK_TITLE string
#define BOOK_LIST vector<BOOK_TITLE>
#define RECOMMENDER_NAME string
#define RECOMMENDER_LIST vector<RECOMMENDER_NAME>
#define RATINGS double
#define RATINGS_LIST vector<RATINGS>
#define RECOMMENDER_RATINGS_MAP map<RECOMMENDER_NAME, RATINGS_LIST>
#define BOOK_AVG_MAP map<BOOK_TITLE, double>
#define BOOK_AVG_LIST vector<pair<BOOK_TITLE, double>>
```

Recommend
- books: BOOK_LIST - recommenders: RECOMMENDER_LIST - ratings: RECOMMENDER_RATINGS_MAP - bookAverages: BOOK_AVG_LIST - similarList: BOOK_AVG_LIST - recommenderSimAvg: BOOK_AVG_LIST

+Recommend(string)
+computeRecommendation(RECOMMENDER)
+computeBookAverages()
+computeSimilarities(RECOMMENDER)
+computeAverageSim(BOOK_AVG_LIST)
+checkRecommender(RECOMMENDER) : bool
+getBookCount() : int
+getRecommenderCount() : int
+getRecommenderBookRating(RECOMMENDER, BOOK_TITLE) : double
+getBookIndex(BOOK_TITLE) : int
+getBookAverage(BOOK_TITLE) : double
+printDotProducts(RECOMMENDER)
+printRecommendation(RECOMMENDER)
+strRecommendation() : string
+printAverages()
+strAverages() : string

```
+printRecommenderRatings()
+strRecommenderRatings() : string
+printSimilarities(RECOMMENDER)
+strSimilarities(): string
+printFancyRatings()
+strFancyRatings(): string
-removeCR(string) : string
-strDivider(char, int) : string
```

Submission Instructions

Submit your complete and correct program to Gradescope. Your submission must include the following files:

1. main.cpp
2. Recommend.h
3. Recommend.cpp
4. Makefile

- Make sure your program compiles and runs on the SoC Linux machines; the autograder automatically assigns 0 points to programs that do not compile.
- Code style (proper indentation, consistency, readability, use of comments, etc.) will be considered when grading your submission.

Collaboration Policy: This project is an <u>individual</u> programming assignment. All work you submit must be your own individual work. You may identify errors or ask about ambiguous specifications in a programming assignment on MS Teams, but your are not allowed to complete an assignment with the help from peers or outside tutors, and you are not allowed to share code with others or copy someone else's code or code from the internet. Please make sure to carefully read the Academic Integrity Statement in the syllabus, which explains what is considered cheating in this class.

Late Policy: Work submitted late will be subject to a 10 percentage point deduction on the 0-100 scale (i.e., one letter grade) per day.

Gradescope Submission: Code must be submitted via Gradescope. Please note that the Gradescope "autograder" is designed to give you feedback on the correctness on your submission. This includes checking if all necessary files are submitted, if the unit tests used for autograding compile, and if the core functionality of your submission is correct (e.g., default constructor, argument constructor, some of the member functions). The points displayed by the autograder when you submit your program do NOT correspond to your final grade for this assignment. Additional tests might be added to the autograder after the submission deadline, and parts of the program will be manually graded.