

AY 2021 Assignment 6

Date / Time	3 December 2021 – 17 December 2021 23:59
Course	[M1522.600] Computer Programming
Instructor	Youngki Lee

- You can refer to the Internet or other materials to solve the assignment, but you ***SHOULD NOT*** discuss the problems with anyone else and need to code **ALONE**.
- We will use the automated copy detector to check for potential plagiarism in the codes between the students. The copy checker is made to be very reliable so that it is highly likely to mark a pair of code as a copy even though two students quickly discuss the idea without looking at each other's code. Of course, we will evaluate the similarity of a pair compared to the overall similarity for the entire class.
- We will also inspect the codes manually. In case we doubt that the code may be written by someone else (outside of the class), we reserve the right to request an explanation about the code. We will ask detailed questions that can only be answered when the codes were written by yourself.
- If one of the above cases happens, you will get 0 marks for the assignment and may get a further penalty. Please understand that we will apply these methods for the fairness of the assignment.
- Download and unzip "HW6.zip" file from the autolab. "HW6.zip" file contains skeleton codes for Problem 1 and Problem 2 (in the "problem1", "problem2" directory).
- When you submit, compress the "HW6" directory which contains "problem1" and "problem2" directories in a single zip file named "20XX-XXXXX.zip" (your student ID) and upload it to the autolab as you submit the solution for HW1~HW5. Contact the TA if you are not sure how to submit. Double-check if your final zip file is properly submitted. You will get 0 marks for the wrong submission format.
- C / C++ Standard Library (including Standard Template Library) is allowed.
- CLion sometimes successfully completes the compilation even without including some header files. Nevertheless, you **MUST include all the necessary header files in the code** to make it successfully compile in the autolab server. Otherwise, you may not be able to get the points.
- Do not use external libraries.
- Trailing whitespaces (space, line break) at the end of the lines are allowed.
- It is allowed to use c++17 features with <filesystem> library.

Contents

Submission Guidelines

Problem 1. Shopping Application [4 Marks]

- 1-1. Interface for Admins [0.5]
- 1-2. Interface for Users [1.5]
- 1-3. Product Recommendation [2]

Problem 2: Porting a Java Program (HW 3-2) to C++ [5 Marks]

- 2-1. Authentication [1]
- 2-2. Post User's Post [1]
- 2-3. Recommend [1.5]
- 2-4. Search Posts [1.5]

References: Standard Template Library (STL) for Assignment 6

- 1. vector (for assignment 6-1 and 6-2)
- 2. unordered_set (for assignment 6-1)
- 3. priority_queue (queue) (for assignment 6-2)
- 4. find (algorithm) (for assignment 6-2)

Submission Guidelines

1. You should submit your code on autolab.
2. After you extract the zip file, you must have an “HW6” directory. The submission directory structure should be as shown in the table below.
3. You can create additional directories or files in each problem directory. Make sure to update CMakeLists.txt to reflect your code structure changes.
4. You can create additional methods or classes, but do not remove or change signatures of existing methods.
5. Compress the “HW6” directory and name the file “20XX-XXXXX.zip” (your student ID).

Submission Directory Structure (Directories or Files can be added)

- Inside the “HW6” directory, there should be “problem1” and “problem2” directory.

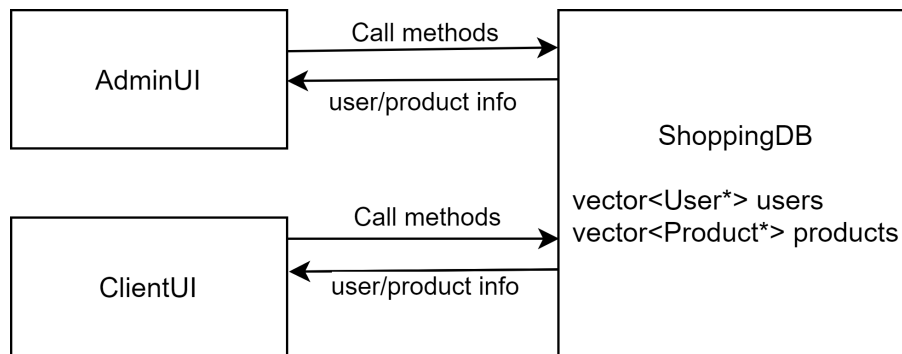
Directory Structure of Problem 1	Directory Structure of Problem 2
problem1/ └─ test/ └─ CMakeLists.txt └─ admin_ui.cpp └─ admin_ui.h └─ client_ui.cpp └─ client_ui.h └─ shopping_db.cpp └─ shopping_db.h └─ ui.h └─ ui.cpp └─ user.h └─ user.cpp └─ product.h └─ product.cpp └─ test.cpp └─ (You can add more files.)	problem2/ └─ data/ └─ test/ └─ CMakeLists.txt └─ app.cpp └─ app.h └─ config.h └─ main.cpp └─ test.cpp └─ (You can add more files.)

Problem 1: Shopping Service [4 Marks]

Objective: Implement a simple shopping service with client/admin interfaces and a database to store the product and user information.

Description: You are a senior developer of an online shopping mall. Your mission is to implement an online shopping service to manage the product and user information. In addition, you are requested to implement a customized product recommendation functionality.

Before you jump into the implementations, you may want to understand the overall structure of this service. The below figure shows the overall structure of the shopping system.

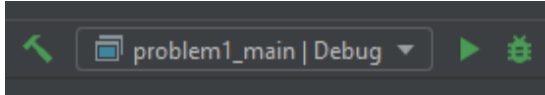
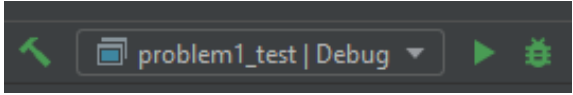
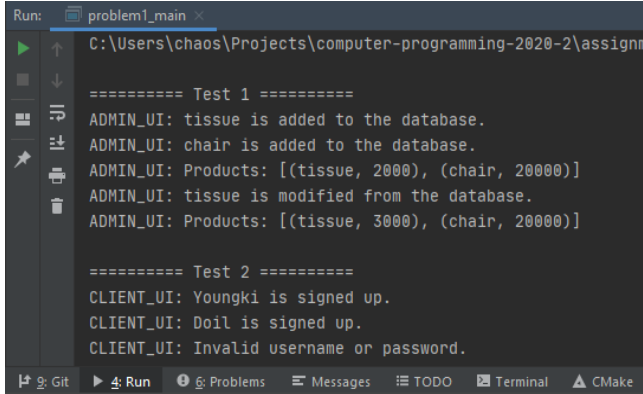
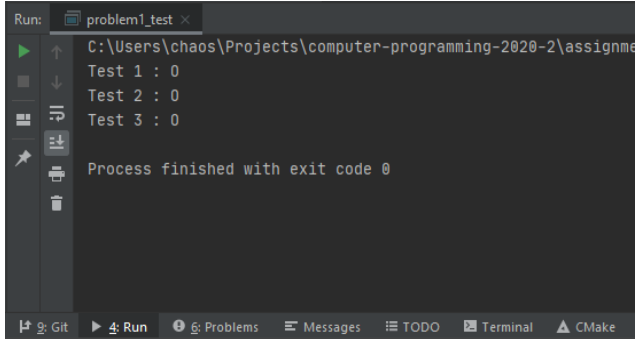


The system mainly consists of 3 classes: `AdminUI`, `ClientUI`, and `ShoppingDB` (along with a few additional classes supporting these key classes.)

- The `AdminUI` class provides interfaces for admins.
- The `ClientUI` class provides interfaces for clients.
- The `ShoppingDB` object receives the queries from the `AdminUI` object and `ClientUI` object, and returns the corresponding data.

How to develop and test assignment 6-1:

- For convenience, we provide two different modes to run your code: "MAIN" and "TEST". In the MAIN mode, "test.cpp" prints the output to the console. In the TEST mode, it compares your results and the expected results in "test/{1,2,3}.out". You can choose the desired mode in the CLion toolbar at the top.
- "test.cpp" includes the example test cases specified in this document. We also provide additional test cases for Autolab. However, we will evaluate your code with a richer set of test cases. We strongly encourage you to add more test cases to make sure your application works as expected.
- If you add more test cases to "test/{1,2,3}.out", press **"File → Reload CMake Project"** in the CLion before you run "test.cpp". Otherwise, the additional test cases are not copied to the build directory (cmake-build-debug).

MAIN mode	TEST mode
	
 <pre> Run: problem1_main x C:\Users\chaos\Projects\computer-programming-2020-2\assignm ===== Test 1 ===== ADMIN_UI: tissue is added to the database. ADMIN_UI: chair is added to the database. ADMIN_UI: Products: [(tissue, 2000), (chair, 20000)] ADMIN_UI: tissue is modified from the database. ADMIN_UI: Products: [(tissue, 3000), (chair, 20000)] ===== Test 2 ===== CLIENT_UI: Youngki is signed up. CLIENT_UI: Doil is signed up. CLIENT_UI: Invalid username or password. </pre>	 <pre> Run: problem1_test x C:\Users\chaos\Projects\computer-programming-2020-2\assignm Test 1 : 0 Test 2 : 0 Test 3 : 0 Process finished with exit code 0 </pre>

A few important notes:

- Do ***NOT*** modify the method signatures of `AdminUI` and `ClientUI` declared in "admin_ui.h" and "client_ui.h", respectively. For evaluation, we will use the functions as declared in these header files. Also, do ***NOT*** modify "ui.h" and "ui.cpp".
- Do ***NOT*** remove `add_executable(problem1_test ...)`, and `target_compile_definitions(problem1_test PRIVATE TEST)` from "CMakeLists.txt". We will use them for evaluation.
- Feel free to add new functions to `AdminUI` and `ClientUI`. You can also add new functions or modify existing functions in other classes.
- You can add new ".h" and ".cpp" files, but you ***MUST*** add them to "CMakeLists.txt". We will use your "CMakeLists.txt" to evaluate your code.
- There are dependencies between subquestions. **This means that the wrong implementation of a method may affect other methods**, and you may lose scores for multiple problems.
- You do not have to store the product and user data in files. It is sufficient to store the data in memory with proper data structures.
- You ***MUST*** use `std::ostream` os of the `AdminUI` and `ClientUI` class to print outputs to the console or for the final evaluation.

Great! You are now ready for the implementation of the cool shopping system.

Problem 1-1: Interface for Admins [0.5 Mark]

Objectives: Implement three methods of `AdminUI` to manage product data.

Description: For the first step, implement an interface for admins to manage product data. In particular, implement the following methods of `AdminUI`. Feel free to change the `ShoppingDB` class; for instance, you can add functions or modify given functions and parameters. However, do ***NOT*** change the function signatures in the `AdminUI` class.

Function 1. *Add Product*

- Implement `void ShoppingDB::add_product(std::string name, int price)`
 - This function adds a product entry with the given name and price in the database.
- Implement `void AdminUI::add_product(std::string name, int price)`
 - This function adds a product entry in the database using the `ShoppingDB::add_product` function.
 - If the given price is positive, print "ADMIN_UI: <PRODUCT_NAME> is added to the database.". ("<PRODUCT_NAME>" should be replaced with the name of the product; do **not** print "<" and ">".)
 - If the given price is equal to or less than zero, print "ADMIN_UI: Invalid price.".
 - Assume that no duplicated names of products or users will be given.

Function 2. *Edit Product*

- Implement `bool ShoppingDB::edit_product(std::string name, int price)`
 - This function modifies the prices of the product of a given name in the database.
 - Modify the price only when the price is positive.
 - Return true when the modification is successful.
- Implement `void AdminUI::edit_product(std::string name, int price)`
 - This function modifies the price of the product of a given name using the `ShoppingDB::edit_product` function.
 - If there is a product with the given name in the database, print "ADMIN_UI: <PRODUCT_NAME> is modified from the database.". ("<PRODUCT_NAME>" should be replaced with the name of the product; do **not** print "<" and ">".)
 - If the given price is equal to or less than zero, print "ADMIN_UI: Invalid price." without modifying the info of the product.
 - If there is no product with the given name in the database, print "ADMIN_UI: Invalid product name.". If this is the case, do not check if the given price is valid.

Function 3. *List Products*

- Implement `get` method of `products` in the `ShoppingDB` class.
- Implement `void AdminUI::list_products()`.
 - This function prints the list of entire products with the following format:
 "ADMIN_UI: Products: [(`<PRODUCT_NAME1>`, `<PRODUCT_PRICE1>`), (`<PRODUCT_NAME2>`, `<PRODUCT_PRICE2>`), ..., (`<PRODUCT_NAMEN>`, `<PRODUCT_PRICE_N>`)]". (`<PRODUCT_NAME#>` should be replaced with the name of the product, `<PRODUCT_PRICE#>` should be replaced with the price of the product; do **not** print "`<`" and "`>`". Note that there is a space behind the comma.)
 - Print the products in the order of the sequence the products are added to the database with the `add_product` method.
 - If there is **no** product in the database, print "ADMIN_UI: Products: []".

Expected result for the skeleton test case:

```

===== Test 1 =====
ADMIN_UI: tissue is added to the database.
ADMIN_UI: chair is added to the database.
ADMIN_UI: desk is added to the database.
ADMIN_UI: Products:[(tissue, 2000), (chair, 20000), (desk, 50000)]
ADMIN_UI: tissue is modified from the database.
ADMIN_UI: Products:[(tissue, 3000), (chair, 20000), (desk, 50000)]

```

Problem 1-2: Interface for Users [1.5 Mark]

Objective: Implement methods of `ClientUI` to support the users.

Description: `ClientUI` provides an interface for clients to login, logout, search for products. Also, it allows a user to add products to his/her shopping cart and purchase them. There are two types of users, "normal user" and "premium user". Premium users get discounts off the normal price while normal users do not have any discount. (HINT: Refer to `user.h` to set the discount rate for each user type.) Feel free to change the `ShoppingDB` class; for instance, you can add functions or modify given functions and parameters. However, do ***NOT*** change the parameters of the functions in the `ClientUI` class.

Part 1. Sign-up, Login, and Logout: Implement a sign-up, login, and logout functions of `ClientUI`. You also need to modify/add functions in the `ShoppingDB` class to store user data in the `ShoppingDB` object.

Function 1. *Sign-up*

- Implement `void ShoppingDB::add_user(std::string username, std::string password, bool premium)`.
 - This function creates an account with the given username and password. Then, it adds the account in the database.
 - Assume that **no** duplicated username is given.
 - The premium parameter specifies whether the type of the user is a "premium user" or a "normal user".
- Implement `void ClientUI::signup(std::string username, std::string password, bool premium)`.
 - This function creates an account with the given username and password using the `ShoppingDB::add_user` function.
 - Print "`CLIENT_UI: <USERNAME> is signed up.`" when a user signed up. ("`<USERNAME>`" should be replaced with the name of the user; do **not** print "`<`" and "`>`".)

Function 2. *Login*

- Implement `void ClientUI::login(std::string username, std::string password)`.
 - This allows a user to log in to the service. This means that you need to keep the log-in state of a user in an appropriate place.
 - HINT: The private variable `current_user` can be used to indicate and check whether the user is currently logged in.
 - If the login is successful, print "`CLIENT_UI: <USERNAME> is logged in.`". ("`<USERNAME>`" should be replaced with the name of the user; do **not** print "`<`" and "`>`".)
 - If there is no user with the given name or the password does not match, print "`CLIENT_UI: Invalid username or password.`".

- If there is a user currently logged-in, print "CLIENT_UI: Please logout first.". If this is the case, do not check if the given username and password are valid.

Function 3. Logout

- Implement `void ClientUI::logout()`.
 - This allows a user to log out from the service.
 - If there is a currently logged-in user, print "CLIENT_UI: <USERNAME> is logged out.".
 - Otherwise, print "CLIENT_UI: There is no logged-in user.".

Part 2. Add to cart and Purchase: Implement “add-to-cart” and “purchase” functions. In particular, implement the following methods of the `ClientUI`. You also need to modify/add functions in the `ShoppingDB` class to store user data in the `ShoppingDB` object.

- For all the functions (i.e., `buy`, `add_to_cart`, `list_cart_products`, and `buy_all_in_cart` functions), check if a user is logged in.
 - If there is no logged-in user, print “CLIENT_UI: Please login first.”.
 - If a user is logged in, perform the requested task.

Function 1. Buy

- Implement `void User::add_purchase_history(Product* product)`.
 - Before buying the product, store the purchase information in the database. The `ClientUI::buy_all_in_cart` function and Problem 1-3 will use this purchase information of users.
- Implement `void ClientUI::buy(std::string product_name)`
 - This function prints the price of a product with the given name (format: "CLIENT_UI: Purchase completed. Price: <PRICE>.").
 - A "premium user" gets a 10% discount (round off to the nearest integer, round up if the first digit of the decimal point is 5). Print the discounted price for a premium user, and the original price for a normal user.
 - If the given product name is invalid, print "CLIENT_UI: Invalid product name.".
 - Note that this method has nothing to do with the products in the cart.

Function 2. Add to the cart

- Implement `void ClientUI::add_to_cart(std::string product_name)`.
 - This function adds a product to the cart of the logged-in user.
 - The user can add the same product to the cart multiple times.

- Print "CLIENT_UI: <PRODUCT_NAME> is added to the cart." after adding the product to the cart.
- If there is no product with the given name, print "CLIENT_UI: Invalid product name."

Function 3. *List all products in the cart*

- Implement `void ClientUI::list_cart_products()`.
 - This function prints all products in the cart of the logged-in user in ascending order of time (oldest first) that the products are added to the cart.
 - The output format is "CLIENT_UI: Cart: [(<PRODUCT_NAME1>, <PRODUCT_PRICE1>), (<PRODUCT_NAME2>, <PRODUCT_PRICE2>), ...]". For example, if the user put an apple, a banana, and an apple to the cart in a sequence, then print "CLIENT_UI: Cart: [(Apple, 1000), (Banana, 1500), (Apple, 1000)]".
 - If there is no product in the cart, print "CLIENT_UI: Cart: []".
 - You can assume that products in the database are not deleted when listing the product in the cart.
 - Again, the discounted prices (i.e., 10% discount, round off to the nearest integer, round up if the first digit of the decimal point is 5) should be displayed for a premium user and the original prices for a normal user.

Function 4. *Buy all products in the cart*

- Implement `void ClientUI::buy_all_in_cart()`.
 - Before buying all products in the cart, store the purchase information in the database, as you did in `ClientUI::buy` function.
 - Print the following message: "CLIENT_UI: Cart purchase completed. Total price: <TOTAL_PRICE>." After that, clear the cart.
 - If the cart is empty, print a message in the same format with zero total price.
 - After purchase, make the cart empty

Expected result for the skeleton test case:

```
===== Test 2 =====  
CLIENT_UI: Youngki is signed up.  
CLIENT_UI: Doil is signed up.  
CLIENT_UI: Invalid username or password.  
CLIENT_UI: Youngki is logged in.  
CLIENT_UI: tissue is added to the cart.  
CLIENT_UI: chair is added to the cart.  
CLIENT_UI: desk is added to the cart.  
CLIENT_UI: Cart: [(tissue, 2700), (chair, 18000), (desk, 45000)]  
CLIENT_UI: Cart purchase completed. Total price: 65700.  
CLIENT_UI: Please logout first.  
CLIENT_UI: Youngki is logged out.  
CLIENT_UI: Please login first.  
CLIENT_UI: Doil is logged in.  
CLIENT_UI: Purchase completed. Price: 20000.  
CLIENT_UI: Doil is logged out.  
CLIENT_UI: There is no logged-in user.
```

Problem 1-3: Product Recommendation [2 Marks]

Objectives: Implement `void ClientUI::recommend_products()`.

Description: Now, you are asked to implement a product recommendation method based on the user's purchase pattern. The recommendation for premium users will be different from that of normal users. In particular, implement `void ClientUI::recommend_products()` which prints the recommended products for the logged-in user. The following are requirements and guidelines with details.

- The printing format of `void ClientUI::recommend_products()` is "CLIENT_UI: Recommended products: [(<PRODUCT_NAME1>, <PRODUCT_PRICE1>), (<PRODUCT_NAME2>, <PRODUCT_PRICE2>), ...]".
- If there is no product to recommend, print "CLIENT_UI: Recommended products: []".
- The discounted prices (i.e., 10% discount, round off to the nearest integer, round up if the first digit of the decimal point is 5) should be displayed for a premium user and the original prices for a normal user.
- The print messages should reflect the current prices. Remember that the prices of the products can be modified with the `void AdminUI::edit_product(std::string name, int price)` method.
- For the method `recommend_products` check if a user is logged in.
 - If there is no logged-in user, print "CLIENT_UI: Please login first.", instead of printing the recommended products.
 - If a user is logged in, perform the requested task.

The following explains more details about the recommendation policies:

- Recommendation for a "normal user":
 - Recommend three most recently purchased items of the logged-in user. Sort them in descending order of purchase time (the latest one first).
 - The three recommended items should be unique. If there are duplicate purchases of the same product, consider it only once. For instance, if a user purchased products "a-b-c-b-d-c" in sequence, the recommended items should be "c-d-b".
 - If the user purchased multiple products at the same time with a `void ClientUI::buy_all_in_cart()` call, assume that the product added to the cart later is purchased later. For example, if 1) a user adds a "tissue", and then a "chair" to the cart, 2) calls `void ClientUI::buy_all_in_cart()`, and 3) buys a "wallet" with `void ClientUI::buy()`, the recommendation list should be "wallet", "chair", and "tissue" in order.
 - The number of items can be less than three if the purchase history is short.
- Recommendation for a "premium user":

- Recommend recently purchased items of other users with the “similar” purchase history. The detailed logic is described below.
- First, sort all users (except for the currently logged-in user) based on purchase history similarities. The similarity between two users is defined as the number of products purchased by both users. In case two users have the same similarities with the logged-in user, the user registered earlier goes first.
- Then, recommend the most recently purchased products of the three users with the highest similarities.
- The three recommended items should be unique. If there are duplicate items, consider it only once and include the most recently purchased product of the next similar user. Assume that there always is the next user.
- If the user purchased multiple products at the same time with a `void ClientUI::buy_all_in_cart()` call, assume that the product added to the cart later is purchased later.
- For example, let's see how the recommended list can be decided for a premium user, Alexa. The table below shows the purchase history of 5 different users (including Alexa) and the purchase history similarities with Alexa.

The most recently purchased product of a user is marked as **blue**.

Name	Purchase History	Commonly purchased items	Purchase history similarity
Alexa	A-B-C-D- C		-
Bob	A-C- A	Two As, One C	3
Chloe	B-B-C-D-A- E	One A, Two Bs, One C, One D	5
David	A-E- B	One A, One B, One E	3
Emily	C-C- A	One A, Two Cs	3
Hyuna	A-A-A-A-A- A	Six As	6

The top three users with the highest similarities are Hyuna, Chloe, and Bob; here the similarities of Bob, David and Emily are the same, but Bob was considered since he was registered before David and Emily. The most recently purchased products of the three users with the highest similarities is A-E-A in order. However, A is duplicated. Thus, instead of recommending two As, recommend B as the third item, which is the most recently purchased product of the next similar user, David. So the final recommendation is A-E-B.

Expected result for the skeleton test case:

```
===== Test 3 =====  
CLIENT_UI: HyunA is signed up.  
CLIENT_UI: Kichang is signed up.  
CLIENT_UI: Hyunwoo is signed up.  
CLIENT_UI: HyunA is logged in.  
CLIENT_UI: Purchase completed. Price: 20000.  
CLIENT_UI: Purchase completed. Price: 20000.  
CLIENT_UI: Purchase completed. Price: 20000.  
CLIENT_UI: Purchase completed. Price: 50000.  
CLIENT_UI: HyunA is logged out.  
CLIENT_UI: Hyunwoo is logged in.  
CLIENT_UI: Purchase completed. Price: 20000.  
CLIENT_UI: Purchase completed. Price: 3000.  
CLIENT_UI: Recommended products: [(tissue, 3000), (chair, 20000)]  
CLIENT_UI: Hyunwoo is logged out.  
CLIENT_UI: Youngki is logged in.  
CLIENT_UI: Recommended products: [(desk, 45000), (tissue, 2700), (chair, 18000)]  
CLIENT_UI: Youngki is logged out.
```

Problem 2: Porting a Java program to C++ [5 Marks]

Objective: Rewrite HW 3-2, a simple console-based SNS system, in C++. A user can write posts, search for them, and view the recommended friends' posts.

Description: You will rewrite the Java program you have written for the HW 3-2 (SNS) in C++. This problem is a good opportunity to compare the similarity and differences between Java and C++. We provide minimal skeleton codes so that you can decide on the overall design of the program. As in HW 3-2, each user can write the posts, search for them, and friends' posts can be recommended. Most of the specifications are similar to that of HW 3-2, but **there are small differences (e.g., output format and data directory). Make sure to check all the details in this document.**

- The program should do nothing in case of any exception (e.g., non-existence of the user id). It means that if an exception occurs before login, the program must be terminated, and if an exception occurs after login, it must be processed to receive command again.
- Default test cases specified in this document are provided in the "test" directory. You can test your code with "test.cpp" by comparing your result with the expected outputs. We will provide additional test cases for Autolab, so use it wisely. For grading, we will apply a richer set of test cases. We strongly encourage you to add more diverse test cases to make sure your application works as expected.
- We provide two target executable code: "test.cpp" and "main.cpp". "test.cpp" tests your code with given test inputs, and "main.cpp" provides you with an interactive interface for convenience.
- You ***MUST*** use `std::istream is` of the `App` class to get console or test input. Also, use `std::ostream os` of the `App` class to print output to the console or for the testing.
- For grading, we will execute `void App::run()` (declared in "app.h"), and compare its output with the expected output.
- We provide "config.h" where the initial data path of the SNS application is defined.
- Do ***NOT*** modify "test.cpp", "app.h", and "config.h", and do ***NOT*** remove `add_executable(problem1_test ...)` from "CMakeLists.txt" since we will use them for grading.
- You can add new ".h" and ".cpp" files, but do ***NOT*** forget to add them to "CMakeLists.txt". We will use your "CMakeLists.txt" to test your codes.
- If you want to add or modify the test cases in the "test" directory, press **"File → Reload CMake Project"** in the CLion before you run "test.cpp" or "main.cpp". Otherwise, the added/modified test cases are not copied to the build directory (cmake-build-debug).
- There are 4 subproblems that are meant to be solved in order. If you proceed without solving the earlier problems, it may influence the later problems.

Great! You are now ready for the last problem of this course! You are almost there!

Problem 2-1: Authenticate [1 Mark]

Objective: Implement an authentication feature. In particular, it compares the input password with the password stored in the server to check its validity.

Description: Upon the program start, the console will ask for the user id and the password. We want to make login possible with a valid password.

- The password of a user is stored at the path $\$(DATA_DIRECTORY)/(User\ ID)/password.txt$; here, the (User ID) is the id of the user.
- Assume that all the names of the direct child directories of $\$(DATA_DIRECTORY)$ are valid user ids.
- Assume that every $\$(DATA_DIRECTORY)/(User\ ID)$ has a password.txt. The format of the password.txt is given in the following example (plain text without a newline). Suppose the password of the user 'root' is 'pivot@eee'.

[File Format] $\$(DATA_DIRECTORY)/root/password.txt$
pivot@eee

- For successful authentication, the input password and the stored password should be **identical** including white spaces, but case-insensitive.
- If the login fails, the program terminates.

Note that text in **blue** in the following example indicates the user input, and the text in **green** indicates the currently authenticated user ID:

Console prompt (Login in this case)
----- Authentication ----- id= root passwd= pivot@eee ----- root@sns.com post : Post contents recommend <number> : recommend <number> interesting posts search <keyword> : List post entries whose contents contain <keyword> exit : Terminate this program ----- Command=

Console prompt (Fails Login in this case)
----- Authentication ----- id= root passwd= admin2 Failed Authentication.

Problem 2-2: Post a User Article [1 Mark]

Objective: Implement a posting feature to store the written post on the server.

Description: When a user inputs the “post” command to the console, he/she can start writing a post with the title and content. The content of the post ends when the user inputs “enter key” twice.

- Store the user’s post at the path $\$(DATA_DIRECTORY)/(User\ ID)/post/(Post\ ID).txt$. (User ID) is the user’s id used for the login, and the (Post ID) is the nonnegative integer assigned uniquely to each and every post in the $\$(DATA_DIRECTORY)$. The newly assigned ID should be **1 + the largest post id in the entire posts in $\$(DATA_DIRECTORY)$** or 0 in case when $\$(DATA_DIRECTORY)/(User\ ID)/post/$ is empty.
- For the given directories in data/file, you can assume all the $\$(DATA_DIRECTORY)/(User\ ID)$ has a directory named post, and each post directory has at least one post.
- The format of the post file is given in the examples below.
- The content of the post should not include the trailing empty line.

For example, let’s say that the user name is ‘root’, and the largest post id in the entire $\$(DATA_DIRECTORY)$ is 302. Then, the new post id should be $302 + 1 = 303$. Also, let the post date be 2021/12/03 21:01:02, the post creation date/time for the local timezone. The post date format is YYYY/MM/DD HH:mm:ss.

Console Prompt

Command=**post**

New Post

* Title=**my name is**

* Content

>**root and**

>**Nice to meet you!**

>

root@sns.com

post : Post contents

recommend <number> : recommend <number> interesting posts

search <keyword> : List post entries whose contents contain <keyword>

exit : Terminate this program

Command=

Then the post is saved to "\$(DATA_DIRECTORY)/root/post/303.txt", as shown below.

Specifically, the post is saved to

"HW6/problem2/cmake-build-debug/\$(DATA_DIRECTORY)/root/post/303.txt" if you are editing your code with CLion, not "HW6/problem2/\$(DATA_DIRECTORY)/root/post/303.txt". The date/time (post creation date/time for the local timezone) and title are written in each new line, respectively. There is an empty line after the title, and finally, the content is written.

[File Format] \$(DATA_DIRECTORY)/root/post/303.txt
2021/12/03 21:01:02 my name is root and Nice to meet you!

Problem 2-3: Recommend Friends' Posts [1.5 Marks]

Objective: Implement the recommendation feature to print the latest posts of the user's friends.

Description: Our SNS service recommends a user the latest posts of her friends. When the user inputs the "recommend <number>" command to the console, up to <number> latest posts of the friends should be displayed.

- The list of the user's friends is stored at the path "\$(DATA_DIRECTORY)/(User ID)/friend.txt". The format of the friend.txt is given in the following example. Suppose the user "root" has 3 friends, "admin", "redis", and "remp".
- You need to look at all the posts of the friends and print up to <number> posts with the latest created dates.
- How do we decide which <number> posts to recommend?
 - Sort posts by the created date specified in a post file in descending order (from latest to oldest).
 - Select the first <number> posts from the sorted list.
 - If there are less than <number> posts in the sorted list, recommend all posts in the list.
- Assume the created date and time of each post is unique. No two posts have the same created date and time.
- Assume all the friend IDs on the friend.txt are valid, and the corresponding folders exist in the \$(DATA_DIRECTORY).
- The post should be printed in the format below.

[File Format] \$(DATA_DIRECTORY)/root/friend.txt

admin
redis
remp

In the example below, the command 'recommend 1' displays 1 latest post of "admin", "redis", and "remp" users.

Console Prompt (Authenticated with 'root')

Command=**recommend 1**

id: 330
created at: 2019/08/13 00:00:00
title: her corespondent conscienceless cobitidae aneurysmal
content:
my where agonized
why he are a balistes why detort
her then aeronaut antithetically brachyurous
a birdseye my where delegacy her carbonic the beslubber am
why
are
egoism why where then is then awl claudius her decretive a am
are

root@sns.com
post : Post contents
recommend <number> : recommend <number> interesting posts
search <keyword> : List post entries whose contents contain <keyword>
exit : Terminate this program

Command=

Addition to the above example, the command 'recommend 3' displays 3 latest posts of "admin", "redis", and "remp" users, and the output posts' ids should be 330, 340 and 314 in a sequence. As the output is too long, let's skip the screenshot.

Problem 2-4: Search Posts [1.5 Marks]

Objective: Implement the searching feature to display up to 10 posts that contain at least one keyword.

Description: Our SNS service enables users to search for posts with multiple keywords. When the user inputs the “search” command along with a set of keywords, the console should display up to 10 posts containing the most number of keywords.

- The range of the search is the entire posts of all users (NOT friends only) in the \$(DATA_DIRECTORY).
- The command string starts with “search” followed by keywords.
- Two keywords are separated with space(' '). The newline should not be considered as a keyword.
- Duplicate keywords should be ignored. For example, the output of "search hi hi" should be identical to the output of "search hi".
- You should count the number of occurrences of the **exact** keyword from the title and the content of the post.
- More specific details for the keyword matching:
 - It should be a case-sensitive comparison.
 - You should only count the word that is identical to the provided keyword. You don't need to consider the word that has the given keyword as a substring.
- For example, the input command is "search to to be", and the content of a post "To **be** trusted is a greater compliment than **to be** loved.". Then the keywords are "to" and "be", and the number of occurrences of the keyword from the content is **3**.
- How do we decide which posts to show?
 - Sort the candidate posts based on two criteria. First, sort by the number of occurrences of the keywords in descending order. When multiple posts have the same number of occurrences, sort them by the number of words in the contents of the post in descending order.
 - Assume that no two posts have the same number of words and occurrences of the keywords.
 - Select up to 10 posts from the beginning of the sorted list.
- Print a summary (id, creation date/time, title) of each post in one line in the following format:

Console Prompt

Command=**search centavo**

id: 282, created at: 1988/03/28 00:00:00, title: astounding cavity cerussite her then

id: 397, created at: 2016/03/31 00:00:00, title: corundom cudgels asphyxiating he dubitousness

id: 371, created at: 1981/05/14 00:00:00, title: am baby-wise despumate anatomist

breeched

id: 368, created at: 1977/07/21 00:00:00, title: balanidae why everyone decretive corral

id: 356, created at: 1988/08/05 00:00:00, title: are my blastopore a is

id: 347, created at: 2010/06/15 00:00:00, title: balderdash begird why the arguementum

id: 309, created at: 2008/03/26 00:00:00, title: boltonia my are derris canard

id: 234, created at: 1995/03/22 00:00:00, title: deo ester he admired antiinflammatory

id: 181, created at: 1972/12/18 00:00:00, title: delphinidae enets camisade why
determinedly

id: 169, created at: 2001/10/08 00:00:00, title: centavo you you carpellate chives

root@sns.com

post : Post contents

recommend <number> : Recommend <number> interesting posts

search <keyword> : List post entries whose contents contain <keyword>

exit : Terminate this program

Command=

References: Standard Template Library (STL) for Assignment 6

Consider using the following data structures provided by STL. In particular, refer to the following.

1. class template <class T> std::vector in <vector> (for assignment 6-1 and 6-2)
 - a. overview: <http://www.cplusplus.com/reference/vector/vector/>
 - b. void push_back(const T& val) method
 - i. Add items to the end of the sequence.
 - ii. https://www.cplusplus.com/reference/vector/vector/push_back/
 - c. iterator begin() and iterator end() methods
 - i. You can iterate the vector with iterator begin() and iterator end() methods. See an example here:
 - ii. <http://www.cplusplus.com/reference/vector/vector/begin/>
2. class template <class T> std::unordered_set in <unordered_set> (for assignment 6-1)
 - a. overview: http://www.cplusplus.com/reference/unordered_set/unordered_set/
 - b. pair<iterator, bool> insert(const_value_type& val) method
 - i. Inserts a new element to the unordered_set.
 - ii. https://www.cplusplus.com/reference/unordered_set/unordered_set/insert/
3. class template <class T> std::priority_queue in <queue> (for assignment 6-2)
 - a. overview: http://www.cplusplus.com/reference/queue/priority_queue/
 - b. void push(const_value_type& val) method
 - i. Inserts a new element at the end of the priority_queue.
 - ii. http://www.cplusplus.com/reference/queue/priority_queue/push/
 - c. void pop() method
 - i. Removes the first element.
 - ii. http://www.cplusplus.com/reference/queue/priority_queue/pop/
 - d. const_reference top() method
 - i. Returns a reference to the first element.
 - ii. http://www.cplusplus.com/reference/queue/priority_queue/top/
 - e. bool empty() method
 - i. Returns whether the priority_queue is empty.
 - ii. http://www.cplusplus.com/reference/queue/priority_queue/empty/
4. function
template <class InputIterator, class T>
InputIterator std::find(InputIterator first, InputIterator last, const T& val)
in <algorithm> (for assignment 6-2)
 - a. Find the given value in the vector, unordered_set, or priority_queue.
 - b. <http://www.cplusplus.com/reference/algorithm/find/>