OOP project report

Hochan Lee hl2347

# Introduction

The application I created for this project is an SNS application. In my sns application, users can post sentences and let his or her friends see the posts. Users can also delete the posts they uploaded, but they cannot edit the posts. Users can see the feed which will contain all the posts posted by the user and friends, ordered in chronological order. Also, users can become friends by first sending a friend request and waiting for the other one to accept the friend request. Users can also unfriend a friend which will delete the friendship relationship from the records.

There is one admin who can view all the posts and all the users. After viewing the posts and users, the admin can delete posts and users.

# Approach

In my sns application, tasks are spread into different classes interacting with each other to perform what they should conceptually do.

## The main function

In the main function, when the program is initiated, the sns database stored as csv files is loaded into the class object database. The data are stored into three files, “profile.csv” which stores user/admin data, “friendship.csv” which stores friendship relationships, and “post.csv” which stores all the posts created by users. Since the program is not connected to a real time database system but it brings all the data whenever the program starts, it should make sure that the program stores the data back into the csv files in the right format.

When the database is loaded, the class instance called “Instance” is initiated. This object is the one and only instance in this program so it might have been designed as a singleton class. However the reason why I left it as a normal class is because normally an sns application in the server might have one database, but several instances serving different clients. In that case, there should be more than one instance so I chose not to design the class “Instance” as a singleton.

When the instance is created and attached to the database class, the application is started by calling its method, “startApp()”. Everything will happen inside this method, including login, logout, and different sns functionalities for a user and an admin. After the app is finished, which means the user chose to quit the program, the instance object calls the “save()” method which makes the database object store the data into the three csv files.

## The Instance class

The instance class can be considered a running application. It has one member, which is the address of the database object loaded in the system. Since the instance object knows the database address and it is a friend class of the database class, it can read and manipulate contents in the database.

The sns application basically runs in the “startApp()” method. Inside the startApp() function, the user has to choose from four options: sign up, login, admin login and quit.

If the user does not have an account, the user can sign up by entering an ID and a password. As it is usually the case in authentication for most apps, the user cannot create an account if the user ID already exists in the system. After the account is created, the user is sent back to the page where he or she can choose from the four options.

When a user logs in, the user sees a prompt that provides 9 options to choose from. The options are:

1. Show user feed
2. Show my posts
3. Show friend list
4. Post a new sentence
5. Request a friend
6. Accept a friend request
7. Unfriend a friend
8. Delete my post
9. Logout

When logged in as admin, the admin sees a prompt of 5 options that are:

1. Show all posts
2. Show all users
3. Delete a post
4. Delete a user
5. Logout

## The Database class

As mentioned above, when the sns program is loaded, the database stored in three csv files are loaded and saved in the database object. The database object plays a pivotal role in this application because this application is all about reading, manipulating and saving the database, with appropriate user/admin actions.

The database object has 5 member variables that are: userMap, loggedinUser, admin, postList, friendRelationships.

The userMap is a c++ STL map that stores pointers to User objects attached to the key of their ID. The database can access each user or sometimes loop through users through this map.

LoggedinUser is a pointer to the User currently logged in. This points to the User object only when a user has logged in. Otherwise it points to NULL.

Admin is a pointer to the only Admin object. This is used to verify when the admin is trying to log in.

PostList is a c++ STL vector that stores all the posts in the database in increasing order of post id. Since this list preserves the ordering of the posts and it is also preserved when storing and loading the csv file of posts, we do not need a sorting algorithm to sort the posts by the post ids.

FriendRelationships is a c++ STL set that stores pointers to Pair objects that stores the friend relationships of two users. This is created just to prevent writing double entries of one friend relationship. (i.e. A,B and B,A).

The role of the destructor for the Database class is important because it is the only class that stores objects like users, posts, friendships dynamically. So in the destructor, it deletes all the dynamically created objects before the program terminates.

## The Profile, User, Admin class

The classes User and Admin both inherit from the superclass Profile. I chose to apply inheritance because both users and admin have a unique ID and password, but their functionalities are different. So, ID and password are saved in the Profile class and inherited privately, while each functionality is defined differently in User and Admin classes.

## The Pair class

The pair class facilitates the process of storing a friend relationship between two IDs. This class stores two IDs as its member variables, and the object is stored in the c++ STL set. This makes the saving and loading this relationship into and from the csv file much easier and efficient.

## Separate Compilation

This program involves separate compilation and the way to do it is defined in the Makefile. The codes are mainly distributed into three fragments, sns.cpp, instance.cpp, and datastructures.cpp (and the corresponding header files). The sns.cpp is the file that has the main function, instance.cpp is where the class Instance is defined, and the datastructures.cpp contains the rest of the datastructures used in the application. When the make command is entered in the terminal, it compiles the three code fragments and outputs three object files. Then the three object files are linked together to output the final executable file, sns. This separate compilation step not only makes compiling the codes easy but also facilitates further project development. Since in many cases I only have to modify part of the code when I’m adding functionalities, by doing separate compilation, I do not have to re-compile every code that hasn’t changed. This will be very efficient as the application becomes bigger and more complicated.

# Solution

## How to implement

In the directory, there will be 9 files: datastructures.cpp, datastructures.h, instance.cpp, instance.h, sns.cpp, profile.csv, friendship.csv, post.csv and the Makefile. All you have to do is the enter “make” and run the sns executable file by the command “./sns”.

There are 3 users originally created and one admin. To test the functionality, you can create a new account or use the existing accounts. The account info of the three users are:

1. ID: a

PW: 1

1. ID: b

PW: 2

1. ID: c

PW: 3

And the admin:

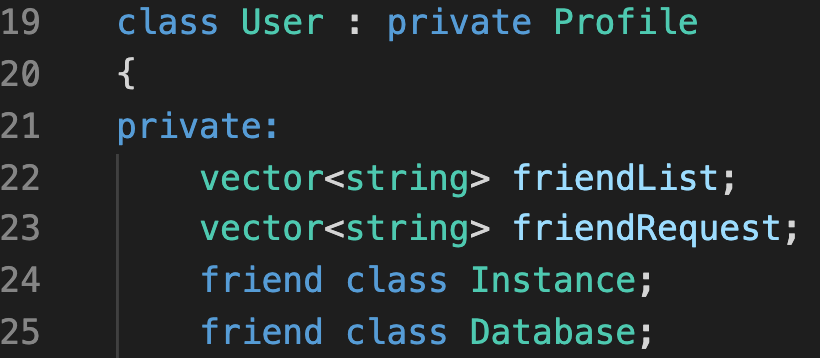
ID: z

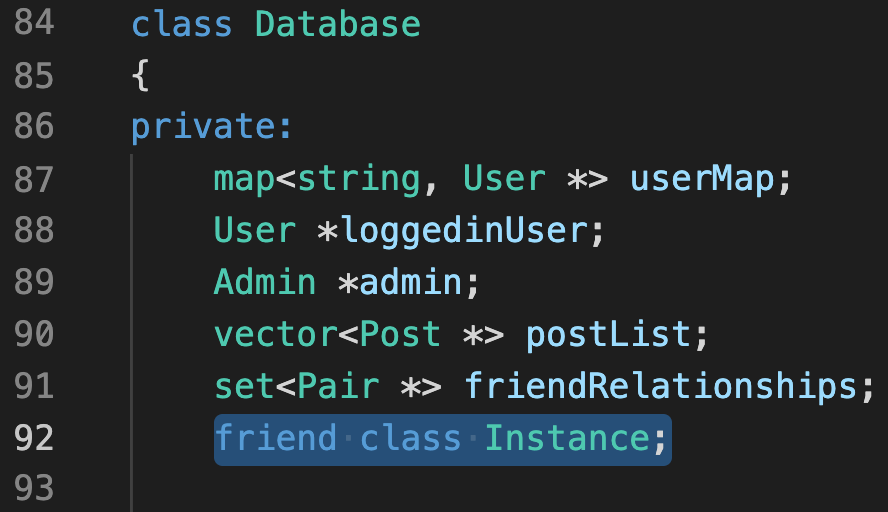
PW: 1

It is best to quit the program after you log out and enter the character ‘q’ rather than interrupting the program with “ctrl+c”, because only with this step the data will be saved.

## The OOP paradigm

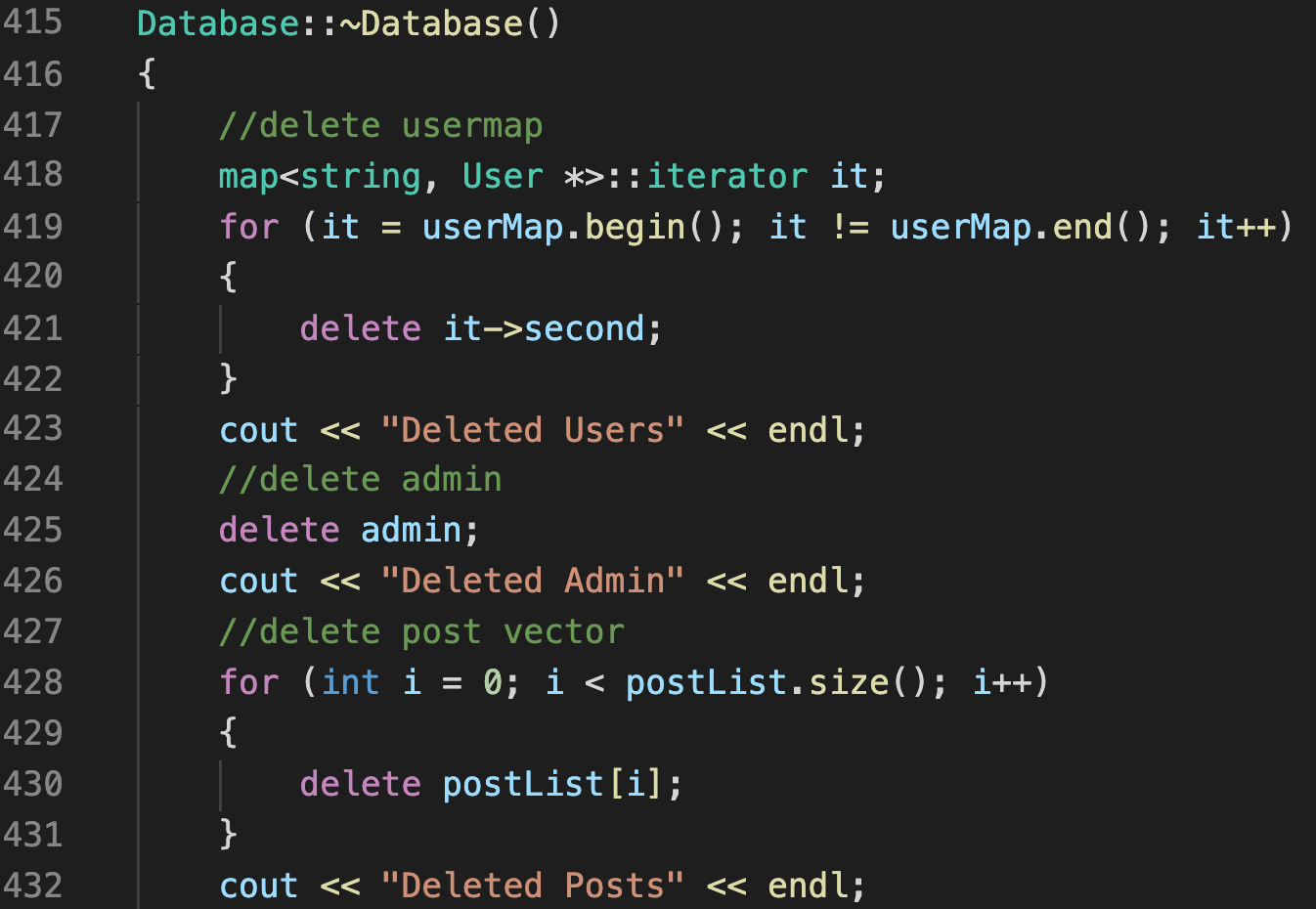
Generally, the biggest OOP paradigm used in this project is the use of different classes interacting with each other with their given access authorities. If some classes have to access private members of other classes, they were declared as friends. For example, in the file datastructures.h, classes Instance and Database are declared friends of the User class, and the Instance class is declared as the friend class of Database class.





Another OOP paradigm used in the project is inheritance. As you can see in the code snippet above, User class inherits from the Profile class privately. It is the same for the Admin class. This was designed this way because both User and Admin should have the same members such as ID and password, but should have different functionalities and methods.

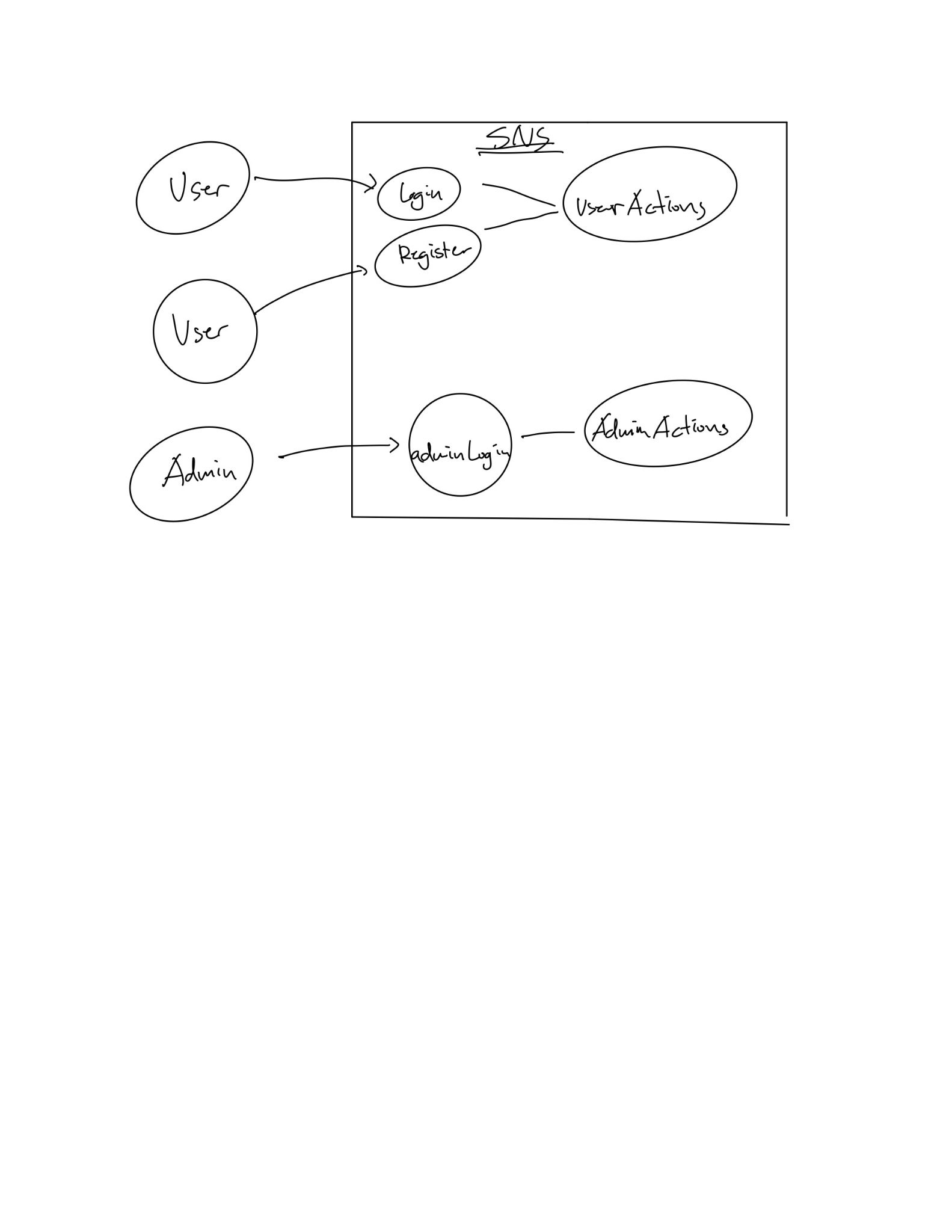
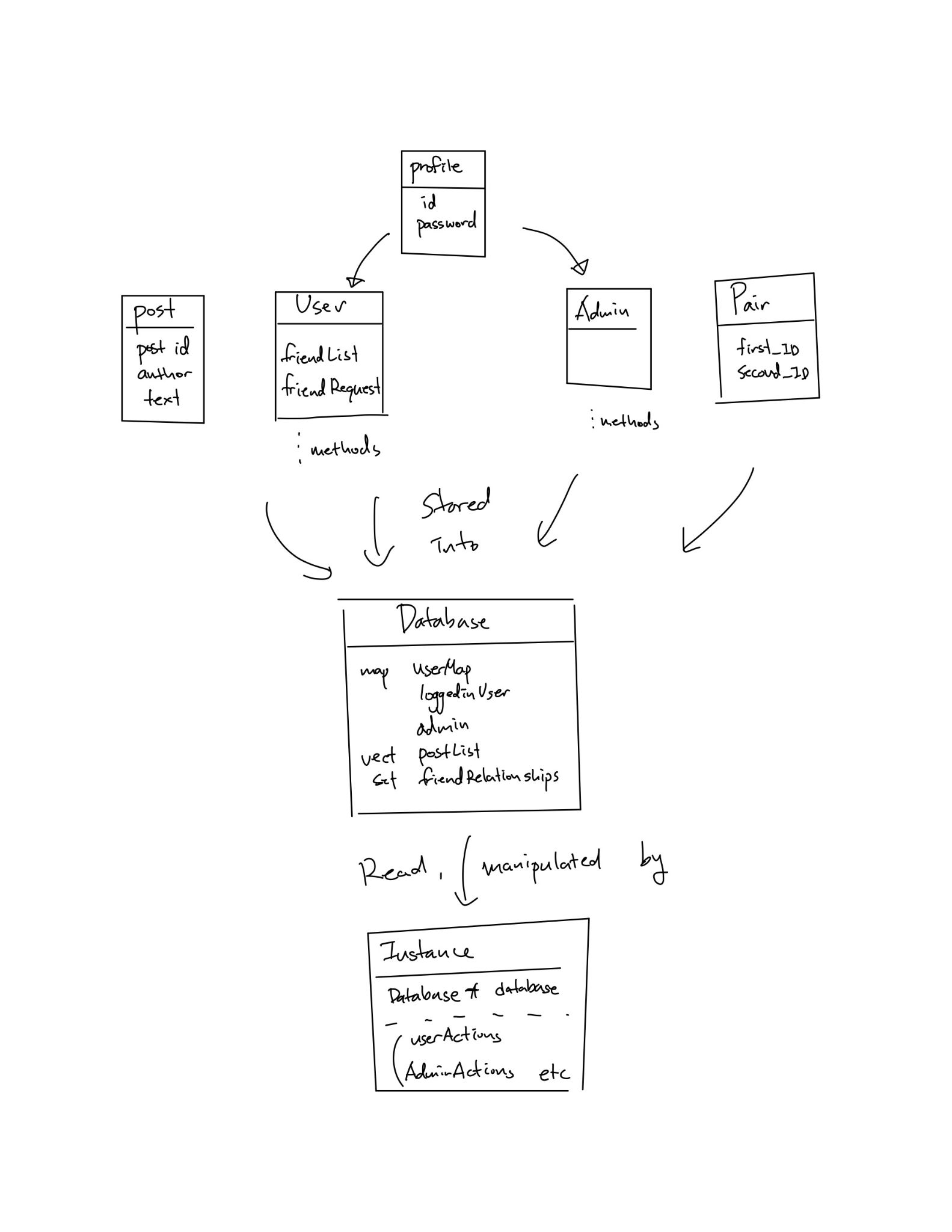
## Optimization

This sns application dynamically creates new objects like users, and posts. All the objects newly created can be accessed through the database object since their pointers are saved. I defined a custom destructor for the database class so that when the program terminates, the database object deletes all the dynamically created objects like Users, Posts, and Pairs. This will prevent memory leaks.

(only part of the desctuctor is shown in the picture).

Another optimization involved is the use of Makefile. This not only makes compilation steps easy, but it also optimizes development steps in the future. Usually, application development is an incremental process, and having to recompile every code each time a new functionality is added will be very inefficient. The separate compilation instructions defined in the Makefile will prevent that.

# Diagrams



**Evaluation Scheme**

|  | 3 points | 2 points | 1 point | 0 point | Student self-evaluation | Instructor’s Evaluation |
| --- | --- | --- | --- | --- | --- | --- |
| Code (15 points) | | | | | | |
| *Readability* | The code is exceptionally well organized and very easy to follow, with extensive use of comments, functions, and files to organize code. | The code is fairly easy to read, with moderate use of comments, functions, and files to organize code. | The code is readable only by someone who knows what it is supposed to be doing. | The code is poorly organized and very difficult to read. | 3 |  |
| *Optimization*            *Indicate file(s) and line number(s).* | Code implements a number of coding paradigms / topics covered in the class that aid in code optimizations for speed and memory. | Code implements a few coding paradigms / topics covered in the class that aid in code optimizations for speed and memory. | Code implements one coding paradigms / topics covered in the class that aid in code optimizations for speed and memory. | Code implements no coding paradigms / topics covered in the class that aid in code optimizations for speed and memory. | 3 |  |
| 1. Destructor(datastructures.cpp line 418 ) 2. Makefile 3. Saving using sets(datastructures.cpp line 371 ) | | | |
| *Object-oriented Paradigms*      *Indicate file(s) and line number(s).* | Code implements a number of object-oriented coding paradigms / topics covered in the class. | Code implements a few of object-oriented coding paradigms / topics covered in the class. | Code implements a few of object-oriented coding paradigms / topics covered in the class. | Code implements no object-oriented coding paradigms / topics covered in the class. | 3 |  |
| 1. Inheritance (datastructrues.h line 12) | | | |
| *Output* | Code functionality prints very clearly to the console with good use of headers and newlines. | Code functionality prints to the console with fair use of headers and newlines. | Code functionality rarely prints to the console. | Code does not print to console to demonstrate functionality. | 3 |  |
| *Correctness* | Code runs perfectly with no errors. | Code runs with a few errors. | Code runs with several errors. | Code does not run at all. | 3 |  |
| Report (18 points) | | | | | | |
| *Engaging* | Writing was deeply engaging, with extensive use of real-word examples, suggesting deep thought in content delivery. | Writing was moderately engaging, with some reference to real-world examples. | Writing made no use of real-world examples. | Writing was difficult to follow and composed in a rush. | 3 |  |
| *Grammar* | Grammar is perfect. | Some number of grammatical errors. | Significant number of grammatical errors, indicating work was reviewed but not carefully. | Full of grammatical errors, indicating work was only a first draft. | 3 |  |
| *Structure* | Extensive use of spacing between paragraphs, headings, sub-headings. | Moderate use of spacing between paragraphs, headings, sub-headings. | Limited use of spacing between paragraphs, headings, sub-headings. | No use of spacing between paragraphs, headings, sub-headings. | 3 |  |
| *Visual* | Excellent use of flowchart / diagram to present logic of code system. | Fair use of flowchart / diagram to present logic of code system. | Poor use of flowchart / diagram to present logic of code system. No clear structure, use of color scheme, with unreadable text. | No use of flowchart / diagram to present logic of code system. | 3 |  |
| *Content* | Report highlights *all* key elements of code, and presents code snippets within document to help describe system. | Report highlights *most* key elements of code, and presents code snippets within document to help describe system. | Report highlights *some* key elements of code, and *poorly* presents code snippets within document in an attempt to describe system. | Report does not explicitly highlight key elements of code, with either no code snippets, or inserts a dump of the whole code base with little effort. | 3 |  |
| *References* | Significant number of references cited in bibliography and writing which may include articles, code blogs/repos, news stories, and website. | A number of references cited. | Almost no references. | No references cited, and writing makes claims that should be supported by references. | 1 |  |