Project Two: Simple Twitter

Out: May 29, 2020; Due: June 16, 2020

Motivation

This project is designed to give students project experience in using different types of streams and I/O, compound types and exceptions.

Introduction

1. Overview

When the master server of a large website goes down, how does the server side recover itself? In fact, one of the (simplified) mechanisms is as follows: When the master server is down, a shadow master server will record the requests in a logfile. When the master server restarts, it first initializes itself based on existing data, and then processes the logfile and handles the requests.

In this project, you will be simulating this process. You don't have to worry about how the master server restarts and how the master server updates the database, instead, you will be in charge of the **initialization** and **logfile processing**.

2. Filesystem

The filesystem consists of two parts: the **logfile** and the existing **user data**. This section introduce the general format and filesystem. For details, refer to the walkthrough section.

The following is an example of a filesystem

```
.
├── logfile
├── username
└── users
    ├── fyq990508
    │   ├── posts
    │   └── user_info
    ├── leepace666666
    │   ├── posts
    │   └── user_info
    ├── marstin
    │   ├── posts
    │   │   ├── 1
    │   │   ├── 2
    │   │   └── 3
```

```
│   └── user_info
├── paul.weng
│   ├── posts
│   │   ├── 1
│   │   └── 2
│   └── user_info
├── xie_jinglei
│   ├── posts
│   └── user_info
├── yinguoxin2017
│   ├── posts
│   └── user_info
└── zhuzhuoer
    ├── posts
    └── user_info
```

(1) User Data

(a) `username` file

The user data are stored in a directory, whose information is summarized in a `username` file. The `username` file is formatted as follows:

```
<name of user data directory> (Relative path)
<username 1>
<username 2>
...
<username n>
```

A sample `username` file looks like:

```
users
paul.weng
leepace666666
fyq990508
marstin
xie_jinglei
```

```
yinguoxin2017

zhuzhuoer
```

The file indicates that the user data are stored in a directory called `users`, and there are 7 users directories within the directory, each named `paul.weng`, `leepace666666`, `fyq990508`, `marstin`, `xie_jinglei`, `yinguoxin2017` and `zhuzhuoer`.

(b) User directories

In each user's personal directory, there is a text file named `user_info` and a directory named `posts`. The `user_info` file contains the account information of a user and the `posts` directory contains all the posts that the user has made.

The `user_info` file is formatted as follows.

```
<num_posts>

<num_following> (n for short)

<following 1>

<following 2>

...

<following n>

<num_follower> (m for short)

<follower 1>

<follower 2>

...

<follower m>
```

A sample `user_info` file looks like:

```
3

4

paul.weng
```

```
fyq990508
xie_jinglei
yinguoxin2017
3
fyq990508
xie_jinglei
yinguoxin2017
```

(c) Posts directory

In the `post` directory, there are some integer-named files. The integer in the filename is the `post_id` of a post and post IDs start from 1. A file contains all information of one post and is formatted as follows.

```
<title>
<tag 1> (Begin and end with #)
...
<tag n>
<text>
<num_likes> (n for short)
<liked user 1>
...
<liked user n>
<num_comments> (m for short)
<username 1>
<comment 1>
...
<username m>
```

```
<comment m>
```

An example looks like the following:

```
Course Description of VE280

#VE280#

#SU2020#

This course is an introduction to programming and provides a
foundation for data structures. The emphasis of this course will
be techniques and principles to quickly write correct programs.
This course is not simply a course about programming. Rather, it
focuses on concepts and methods underpinning the C++ language.

6

leepace666666

fyq990508

marstin

xie_jinglei

yinguoxin2017

zhuzhuoer

3

leepace666666

Glad to be the TA of VE280 this semester, let's work!

fyq990508

Welcome to VE280!

zhuzhuoer

VE280 is a foundation for future courses.
```

(2) Logfile

The log information is stored in **one single text file**, referred to as `logfile`. In the `logfile`, users' requests are stored and represented by one or a few lines.

The following is a valid logfile:

```
marstin follow zhuzhuoer
marstin refresh
zhuzhuoer post
Writing Project 2
#VE280#
I am writing the sample code for ve280 project 2.
marstin like zhuzhuoer 1
marstin comment zhuzhuoer 1
Ummm, good luck dude.
marstin unlike zhuzhuoer 1
zhuzhuoer delete 1
marstin refresh
marstin unfollow zhuzhuoer
```

There are 11 types of possible requests in total.

(a) Follow request

- Description:

  When `user 1` follows `user 2`, a follow request is made to the server.

- Format:

  `<user 1> follow <user 2>`

- Example:

  `zhuzhuoer follow marstin`

(b) Unfollow request

- Description:

  When `user 1` unfollows `user 2`, a follow request is made to the server.

- Format:

  `<user 1> unfollow <user 2>`

- Example:

  `zhuzhuoer unfollow marstin`

(c) Like request

- Description:

  When `user 1` likes the n_{th} post (n = `post_id`) of `user 2`, a like request is made.

- Format:

  `<user 1> like <user 2> <post_id>`

- Example:

  `zhuzhuoer like marstin 1`

(d) Unlike request

- Description:

  When `user 1` unlikes the n_{th} post (n = `post_id`) of `user 2`, an unlike request is made.

- Format:

  `<user 1> unlike <user 2> <post_id>`

- Example:

  `zhuzhuoer unlike marstin 1`

(e) Comment request

- Description:

  When `user 1` comments the n_{th} post (n = `post_id`) of `user 2`, a comment request is made to the server.

- Format:

  `<user 1> comment <user 2> <post_id>`

  `<text>`

- Example:

  `zhuzhuoer comment marstin 1`

  `Ummm, good luck dude.`

(f) Uncomment request

- Description:

  When `user 1` uncomments the k_{th} comment (k = `comment_id`) of a post of `user 2`, an uncomment request is made to the server.

- Format:

  `<user 1> uncomment <user 2> <post_id> <comment_id>`

- Example:

  `zhuzhuoer uncomment marstin 1 1`

(g) Post request

- Description:

  When `user` uploads a new post, a post request is made to the server.

- Format:

  `<user> post`

  `<title>`

  `<tag 1> (Again, tags begin and end with #)`

```
...
<tag n>
<text>
```

Since this is a new post, the initial number of likes and comments should both be 0. Yet, please note that `post_id` and `comment_id` start from 1.

- Example:

```
zhuzhuoer post
Writing Project 2
#VE280#
I am writing the sample code for ve280 project 2 with Martin.
```

(h) Delete request

- Description:

  When `user` deletes a post, a delete request is made to the server.

- Format:

  `<user 1> delete <post_id>`

- Example:

  `marstin delete 1`

(i) Refresh request

- Description:

  When `user 1` refreshes, your server should print all the posts of the users that `user 1` is following, and of `user 1` himself/herself.

- Format:

  `<user 1> refresh`

- Example:

  `marstin refresh`

(j) Visit request

- Description:

  When `user 1` visits `user 2`, your server should print information of `user 2` and his/her social relationship with `user 1`.

- Format:

  `<user 1> visit <user 2>`

- Example:

  `marstin visit zhuzhuoer`

(k) Trending request

- Description:

  When a user requests to view the top-`n`-popular tags, your server should print the first `n` tags that are most discussed. The measurement will be discussed in later section.

- Format:

  `trending <n>`

- Example:

  `trending 3`

3. Input/Output

(1). Program Argument

Your program will obtain the names of the `username` file and the `logfile` via program arguments. The expected order of arguments is: `<username> <logfile>`, both arguments are mandatory.

Suppose that you program is called p2. It may be invoked by typing in a terminal:

```
./p2 namelist1 log1
```

This means that your program should read the username file from a file called `namelist1` and read the log information from a logfile called `log1`.

You should ignore the excessive arguments if more than 2 arguments are provided. If the number of arguments passed to your program is less than 2, your program should raise an exception (will be discussed in exception section).

(2) Input

All the log information and user information are stored in files, and these files will be read by your program to set up the simulation environment. **When you read files, you must use input file stream `ifstream`.** Otherwise, since the files are read-only on our online judge, you may fail to read the files.

Unless explicitly required in the exception section in the later part of the document, you may assume that the input is of correct format. For example, you may assume that if user A is in the following list of user B, user B will be in the follower list of user A naturally, since this is not required in the exception handling list.

Also, we will **not** be testing you on non-printable character handling, including `spaces`, `tabs` or `\0` in usernames etc.

(3) Output

For simplicity, **your program only needs to print to the standard output, and you don't have to (and must not) update the filesystem.** In order not to waste your time in trying to fit your output format with the solution, we have provided the `print` function for `users`, `posts` and `tags`. You shouldn't modify the output format, including adding spaces, blank lines, etc. Yet, you may adapt the argument passed in the function based on your implementation.

For the format, ordering and tie breaking of standard output, refer the logfile processing section. See also the appendix for a sample output.

For the output of error handling, refer to the exception section.

Breakdown and Walkthrough

1. Starter Files

It may be a jump from friendly project 1 to project 2. Therefore, in this project, we will provide 3 starter files for you with hints to get started.

(1). `server_type.h`

In completing this project, you will have some constants and compound types available to you. They are defined in the file `server_type.h`.

For the constants and `Exception_t` type, please leave them as they are.

For the compound types defined in the header, you are recommended to stick with them. Yet, you can always delete them all and redesign the types on your own, since clearly what we provide is not the unique (not even the optimal) solution. However, you should notice that the print functions we provided are developed upon them, and you will have to carefully adapt the print function.

(2). `simulation.h` and `simulation.cpp`

In `simulation.h` you should declare all the functions you write. In `simulation.cpp`, all the implementations of those functions declared in the `simulation.h` are defined. We implemented the `print` functions for `users`, `posts` and `tags`. Again, you shouldn't modify the output format, but may adapt the argument passed in the function based on your implementation.

You should write one other source code file called `p2.cpp`, which contains only the main function.

2. Server Initialization

During the server initialization, your server should be able to read the user data from files. This could break into the following steps:

(1) Open and read in the `username` file.

- You may assume that the `user` directory specified in the first line and the personal directories (directories named by the usernames) within the `user` directory always exist.

- You may assume that the `username` file is of correct format, if one exists.
- You may assume that the usernames in the file contain no non-printable characters and fit exactly in one line.

(2) Open the personal directories of each users and read in the user information and post information.

- You may assume that the `posts` directory within each personal directory always exists.
- Even if the numerical value `<num_*>` is 0, the text files will have 0 in it explicitly.
- You may assume that the `user_info` file is of correct format, if one exists. For example, you may assume that if `<num_following>` is 3, the next 3 lines will be 3 usernames, and the fourth line will be `<num_follower>`.
- The order of the list of following/follower matter. Please keep the order as they were in the `user_info` file, and when following or followed by a new user, append his/her name to the end of the list to ensure order.
- You may assume that the `post` file is named out of an integer and of correct format, if one exists.
- You may assume that all posts have a main text that does not start and end with #.

3. Logfile Processing

(1) Open the `logfile`.

(2) Process the requests in the `logfile`. See also the appendix for a sample output.

- You may assume that the requests are of correct format. For example, consider a visit request: `marstin visit zhuzhuoer`. You may assume that there will be exactly 3 terms provided in 1 line and the second term is exactly "visit" of lower case.
- When a request is read in, your program should print out `>> <request name>` before any output. For example, if `marstin follow zhuzhuoer` is read in, your program should print out:

```
>> follow
```

- For `refresh` request:

  When `user 1` refreshes, your server should print all the posts (with the given `printPost` function) of the users that `user 1` is following, and all the posts of `user 1` himself/herself. The order of posts is determined by the following rules:

  (a). Owner's username: You should print `user 1`'s own posts first, and then print the posts made by the users that `user 1` follows, according to their order in the following list.

  (b). Post ID: If a user has more than 1 post, print his/her posts by post ID in ascending order.

  Considering the following example in the testcase in the starter files. The request `marstin refresh` will result in the following output:

```
>> refresh

marstin

TA Appointment

Hey guys, I will be the TA for VE280 and VE492 this semester.
Let's have fun.

Tags: VE492 VE280 SU2020

Likes: 2

- - - - - - - - - - - - - - -

marstin

VE280 Lab1

As introduced in the first lecture, we have labs this semester.
In each lab, you will be completing around 3 short programming
exercises, designed by one of the staffs. When you finish the
lab exercises, simply submit your answer to the online-judge.

Tags: VE280

Likes: 2

Comments:

fyq990508: Hey dude, I will be in charge of lab2.

- - - - - - - - - - - - - - -

marstin
```

Designing VE280 Project

Why are projects so hard to design?????????

Tags: VE280

Likes: 0

Comments:

zhuzhuoer: True.

- - - - - - - - - - - - - - -

paul.weng

Course Description of VE492

Introduction to the core concepts of AI, organized around building computational agents. Emphasizes the application of AI techniques. Topics include search, logic, knowledge representation, reasoning, planning, decision making under uncertainty, and machine learning.

Tags: VE492 SU2020

Likes: 4

Comments:

xie_jinglei: This course is very interesting and useful for CS students!

marstin: I am glad to be the TA of this course this semester.

- - - - - - - - - - - - - - -

paul.weng

Course Description of VE280

This course is an introduction to programming and provides a foundation for data structures. The emphasis of this course will be techniques and principles to quickly write correct programs. This course is not simply a course about programming. Rather, it focuses on concepts and methods underpinning the C++ language.

Tags: VE280 SU2020

Likes: 6

Comments:

```
leepace666666: Glad to be the TA of VE280 this semester, let's
work!

fyq990508: Welcome to VE280!

zhuzhuoer: VE280 is a foundation for future courses.

- - - - - - - - - - - - - - -
```

- For `visit` request:

  When `user 1` visits `user 2`, your server should print information of `user 2` and his/her social relationship with `user 1` with the `printUser` function given. The social relationship is defined as follows:

  (a). If `user 1` is following `user 2` and `user 2` is following `user 1` as well, then the relationship is `friend`.

  (b). If `user 1` is following `user 2` but `user 2` is not following `user 1` as well, then the relationship is `following`.

  (c). If `user 1` is not following `user 2`, then the relationship is `stranger`.

  (d). If `user 1` is visiting himself/herself, then the relationship is an empty string.

  Considering the following example in the testcase in starter files.

  The request `marstin visit xie_jinglei` will result in the following output:

```
>> visit

xie_jinglei

friend

Followers: 1

Following: 2
```

- For `trending` request

  When a user requests to view the top-`n`-popular tags, your server should print the first `n` tags that are most discussed, measured by `tag score`. The tag score is calculated by the following formula:

  p = number of posts containing the tag

l = total number of likes of all posts containing the tag

c = total number of comments of all posts containing the tag

Tag score = $5 \times p + 3 \times c + l$

Note that a tag exists if at least one post contains it. For the `trending` request, You should print out the tags in descending order of their tag scores. If the number of tags requested is larger than the total number of tags, just print out all tags.

If the scores of two tags are the same, consider the ASCII order and let the one with smaller ASCII code have the higher rank. For example, if VE280 and SU2020 both have the same score, SU2020 should go first because character 'S' has smaller ASCII code than 'V'. If the ASCII codes of the first characters are the same, then consider the second. And if all the characters are the same, the shorter tag should go first.

Considering the following example in the testcase in the starter files.

The request `trending 5` will result in the following output:

```
>> trending
1 VE280: 45
2 SU2020: 42
3 VE492: 22
```

4. Exceptions

(1) `Exception_t` structure

Refer to `server_type.h` for the `Exception_t` type used in this project. It actually inherits the `std::exception` class. Since we haven't talked much about C++ classes during the lecture, we will provide an example of how to throw an exception.

```
/* Exception */
enum Error_t {
    INVALID_ARGUMENT,
    FILE_MISSING,
    CAPACITY_OVERFLOW,
```

```
    INVALID_LOG,
};
struct Exception_t: public exception{
    Error_t error;
    string error_info;
    Exception_t(Error_t err, const string& info){
        this->error = err;
        this->error_info = info;
    }
};
```

Consider the following example. You should ignore any extra arguments if more than 2 are provided. If the number of arguments passed to your program is less than 2, then one of the mandatory arguments is missing and your program should raise an exception of error type `INVALID_ARGUMENT`. Also, you should print the following error message:

```
Error: Wrong number of arguments!
Usage: ./p2 <username> <logfile>
```

To throw this exception, we can do this in the code:

```
try{
    if(argc < 3){
        ostringstream oStream;
        oStream << "Error: Wrong number of arguments!" << endl;
        oStream << "Usage: ./p2 <username> <logfile>" << endl;
        throw Exception_t(INVALID_ARGUMENT, oStream.str());
    }
}
catch (Exception_t &exception){
    cout << exception.error_info;
```

```
    return 0;
}
```

Note that the first parameter provided to `Exception_t` is one of the values inside the `enum Error_t` type and the second parameter is a string. You can also use your own method to generate this string. After you throw an `Exception_t` you can reach the attributes using the dot operator inside the `catch` block, like normal `structs`.

Due to the limitation of the auto judge, please always `return 0` even an exception is thrown.

(2) Error Types

Error types are defined in the `enum Error_t`, including `INVALID_ARGUMENT`, `FILE_MISSING`, `CAPACITY_OVERFLOW` and `INVALID_LOG`. Specifically, **initialization error** occurs during the server initialization, including `INVALID_ARGUMENT`, `FILE_MISSING` and `CAPACITY_OVERFLOW`. The other type is called **processing error**, which occurs during the logfile processing and includes `INVALID_LOG` and `CAPACITY_OVERFLOW`.

To make things clear, the initialization errors will only be triggered during the server initialization, that is, when you are reading the user data (`<username>` etc). The processing error will only be triggered during the logfile processing that is when you are reading the `<logfile>`.

(3) The Initialization Error

Your program should check for initialization errors **before** it starts to process the logfile. If any error happens, your program should issue an error message. If there are no errors happening, then the initial state of the server is legal and your program can start processing the logfile.

We require you to do the following error checking and print the error message in **exactly** the same way as described below. Note that some of the output error message has two lines and each error message should be ended with a newline character. All error messages should be sent to the standard output stream `cout`, none to the standard error stream `cerr`.

- Check whether the number of arguments is less than 2. If true, then one of the mandatory arguments is missing. You should raise an `INVALID_ARGUMENT` error and print the following error message:

  ```
  Error: Wrong number of arguments!
  ```

```
Usage: ./p2 <username> <logfile>
```

- Check whether files are successfully opened. If you fail to open a username file, logfile, user info file or any post file (like the file to be opened does not exist), you should raise a `FILE_MISSING` error and print the following error message:

  ```
  Error: Cannot open file <filename>!
  ```

  where `<filename>` should be replaced with the name of the file that fails to be opened.

  Note that if that file is not in the same directory as your program, you need to include its **relative path** in the `<filename>`, i.e. `<dir>/<filename>`. Mind your platform, please note that JOJ may reject answers like `<dir>//filename`, etc. Similarly, once you find a file that cannot be opened, issue the above error message and terminate your program.

- Check whether the number of users listed in the username file exceeds the maximal number of users `MAX_USERS`. If so, you should raise a `CAPACITY_OVERFLOW` error and print the following error message:

  ```
  Error: Too many users!

  Maximal number of users is <MAX_USERS>.
  ```

  where `<MAX_USERS>` should be replaced with the maximal number of users set by your program.

- Check whether the number of posts listed in the userinfo file exceeds the maximal number of posts `MAX_POSTS`. If so, you should raise a `CAPACITY_OVERFLOW` error and print the following error message:

  ```
  Error: Too many posts for user <USER_NAME>!

  Maximal number of posts is <MAX_POSTS>.
  ```

  where `<USER_NAME>` should be replaced with the name of the users who has posts more than the maximal number allowed and `<MAX_POSTS>` should be replaced with the maximal number of posts a user can have set by your program.

- Check whether the number of followings listed in the `user_info` file exceeds the maximal number of following `MAX_FOLLOWING`. If so, you should raise a `CAPACITY_OVERFLOW` error and print the following error message:

```
Error: Too many followings for user <USER_NAME>!

Maximal number of followings is <MAX_FOLLOWING>.
```

where `<USER_NAME>` should be replaced with the name of the users whose number of followings is more than the maximal number allowed and `<MAX_FOLLOWING>` should be replaced with the maximal number of followings a user can have set by your program.

- Check whether the number of followers listed in the userinfo file exceeds the maximal number of followings `MAX_FOLLOWERS`. If so, you should raise a `CAPACITY_OVERFLOW` error and print the following error message:

```
Error: Too many followers for user <USER_NAME>!

Maximal number of followers is <MAX_FOLLOWERS>.
```

where `<USER_NAME>` should be replaced with the name of the users whose number of followers is more than the maximal number allowed and `<MAX_FOLLOWERS>` should be replaced with the maximal number of followers a user can have set by your program.

- Check whether the number of tags listed in the post file exceeds the maximal number of tags `MAX_TAGS`. If so, you should raise a `CAPACITY_OVERFLOW` error and print the following error message:

```
Error: Too many tags for post <POST_TITLE>!

Maximal number of tags is <MAX_TAGS>.
```

where `<POST_TITLE>` should be replaced with the title of the post whose number of tags is more than the maximal number allowed and `<MAX_TAGS>` should be replaced with the maximal number of tags a post can have set by your program.

- Check whether the number of likes listed in the post file exceeds the maximal number of likes `MAX_LIKES`. If so, you should raise a `CAPACITY_OVERFLOW` error and print the following error message:

```
Error: Too many likes for post <POST_TITLE>!

Maximal number of likes is <MAX_LIKES>.
```

where <POST_TITLE> should be replaced with the title of the post whose number of likes is more than the maximal number allowed and <MAX_LIKES> should be replaced with the maximal number of likes a post can have set by your program.

- Check whether the number of comments listed in the post file exceeds the maximal number of comments MAX_COMMENTS. If so, you should raise a CAPACITY_OVERFLOW error and print the following error message:

```
Error: Too many comments for post <POST_TITLE>!
Maximal number of comments is <MAX_COMMENTS>.
```

where <POST_TITLE> should be replaced with the title of the post whose number of comments is more than the maximal number allowed and <MAX_COMMENTS> should be replaced with the maximal number of comments a post can have set by your program.

(4) The Processing Error

Your program should check for processing errors **after** it starts to process the logfile. If any error other than CAPACITY_OVERFLOW happens, your program should issue an INVALID_LOG error and then ignore current problematic request and **execute the next request** in the logfile.

We require you to do the following error checking and print the error message in **exactly** the same way as described below. Note that some of the output error message has two lines and each error message should be ended with a newline character. All error messages should be sent to the standard output stream cout; none to the standard error stream cerr.

For simplicity, you may ignore the CAPACITY_OVERFLOW errors that happens due to follow, like, comment and post requests, etc. Yet, you are encouraged to include them for completeness, even though we won't be testing you on this.

The following exceptions are all of INVALID_LOG error type.

- For like request: <user 1> like <user 2> <post_id>
  - Check whether <user 2> has posted post <post_id>. If not so, print the following error message:

```
Error: <user 1> cannot like post #<post_id> of <user 2>!
<user 2> does not have post #<post_id>.
```

- Check whether `<user 1>` has already liked the post `<post_id>` of `<user 2>`. If so, print the following error message:

  ```
  Error: <user 1> cannot like post #<post_id> of <user 2>!
  <user 1> has already liked post #<post_id> of <user 2>.
  ```

  where `<user 1>`, `<user 2>` and `<post_id>` should be replaced with the corresponding items in the command.

- For `unlike` request: `<user 1> unlike <user 2> <post_id>`

  - Check whether `<user 2>` has posted post `<post_id>`. If not so, print the following error message:

    ```
    Error: <user 1> cannot unlike post #<post_id> of <user 2>!
    <user 2> does not have post #<post_id>.
    ```

  - Check whether `<user 1>` has not liked the post `<post_id>` of `<user 2>`. If so, print the following error message:

    ```
    Error: <user 1> cannot unlike post #<post_id> of <user 2>!
    <user 1> has not liked post #<post_id> of <user 2>.
    ```

  where `<user 1>`, `<user 2>` and `<post_id>` should be replaced with the corresponding items in the command.

- For `comment` request: `<user 1> comment <user 2> <post_id>`

  Check whether `<user 2>` has posted post `<post_id>`. If not so, print the following error message:

  ```
  Error: <user 1> cannot comment post #<post_id> of <user 2>!
  <user 2> does not have post #<post_id>.
  ```

  where `<user 1>`, `<user 2>` and `<post_id>` should be replaced with the corresponding items in the command.

- For uncomment request: `<user 1> uncomment <user 2> <post_id> <comment_id>`
  - Check whether `<user 2>` has posted post `<post_id>`. If not so, print the following error message:

    ```
    Error: <user 1> cannot uncomment comment #<comment_id> of
    post #<post_id> posted by <user 2>!

    <user 2> does not have post #<post_id>.
    ```

  - Check whether post `<post_id>` of `<user 2>` has comment `<comment_id>`. If not, print the following error message:

    ```
    Error: <user 1> cannot uncomment comment #<comment_id> of
    post #<post_id> posted by <user 2>!

    Post <post_id> does not have comment #<comment_id>.
    ```

  - Check whether `<user 1>` is the owner of comment `<comment_id>` of the post `<post_id>` by `<user 2>`. If not so, print the following error message:

    ```
    Error: <user 1> cannot uncomment comment #<comment_id> of
    post #<post_id> posted by <user 2>!

    <user 1> is not the owner of comment #<comment_id>.
    ```

  where `<user 1>`, `<user 2>`, `<post_id>` and `<comment_id>` should be replaced with the corresponding items in the command.

- For `delete` request: `<user 1> delete <post_id>`

  Check whether `<user 1>` has posted post `<post_id>`. If not so, print the following error message:

  ```
  Error: <user 1> cannot delete post #<post_id>!

  <user 1> does not have post #<post_id>.
  ```

  where `<user 1>`, `<user 2>` and `<post_id>` should be replaced with the corresponding items in the command.

Implementation Requirements

1. In writing your code, you may use the following standard header files: `<iostream>`, `<fstream>`, `<sstream>`, `<iomanip>`, `<string>`, `<cstdlib>`, `<cctype>`, `<cassert>` and `<algorithm>`. **No other header files can be included.**

2. You may not define any global variables yourself. You can only use the global constant ints and string arrays defined in `server_type.h`.

3. Pass large structs by reference rather than value. Where appropriate, pass const references / pointers-to-const. Do not pass lots of little arguments when you can pass an appropriate, larger structure instead.

4. All required output should be sent to the standard output stream; none to the standard error stream.

5. You should strive not to duplicate identical or nearly-identical code, and instead collect such code into a single function that can be called from various places. Each function should do a single job, though the definition of "job" is obviously open to interpretation. Most students write too few functions that are too large.

Compiling and Testing

To compile, type the following linux command:

```
g++ -Wall -o p2 simulation.cpp p2.cpp
```

You should test your program extensively. Yet, we've provided you a sample test set. Try to run your program on our test set and redirect the output to a text file and compare it with the correct answer we provide in `test.ans`.

```
./p2 username logfile > test.out

diff test.out test.ans
```

Submitting and Due Date

You should submit your source code files `server_type.h`, `simulation.h`, `simulation.cpp`, and `p2.cpp`. These files should be submitted as a tar file via the online judgment system. See announcement from the TAs for details about submission. The due date is 11:59 pm on June 16th, 2020.

Grading

Your program will be graded along three criteria:

1. Functional Correctness

Functional Correctness is determined by running a variety of test cases against your program, checking against our reference solution.

The composition of testcases is **approximately** as follows:

Type	Description	Proportion	Status
Simple	Small testcases that specifically test your program on 1-2 requirements	40%	50% Public
Comprehensive	Large testcases that comprehensively test your program on most of the requirements	40%	10% Public
Excetption	Testcases that test your program on exceptions.	20%	50% Public

2. Implementation Constraints

We will grade Implementation Constraints to see if you have met all of the implementation requirements and restrictions.

3. General Style

General Style refers to the ease with which TAs can read and understand your program, and the cleanliness and elegance of your code.