HW: Hello, CPPeers!

Overview

Objectives

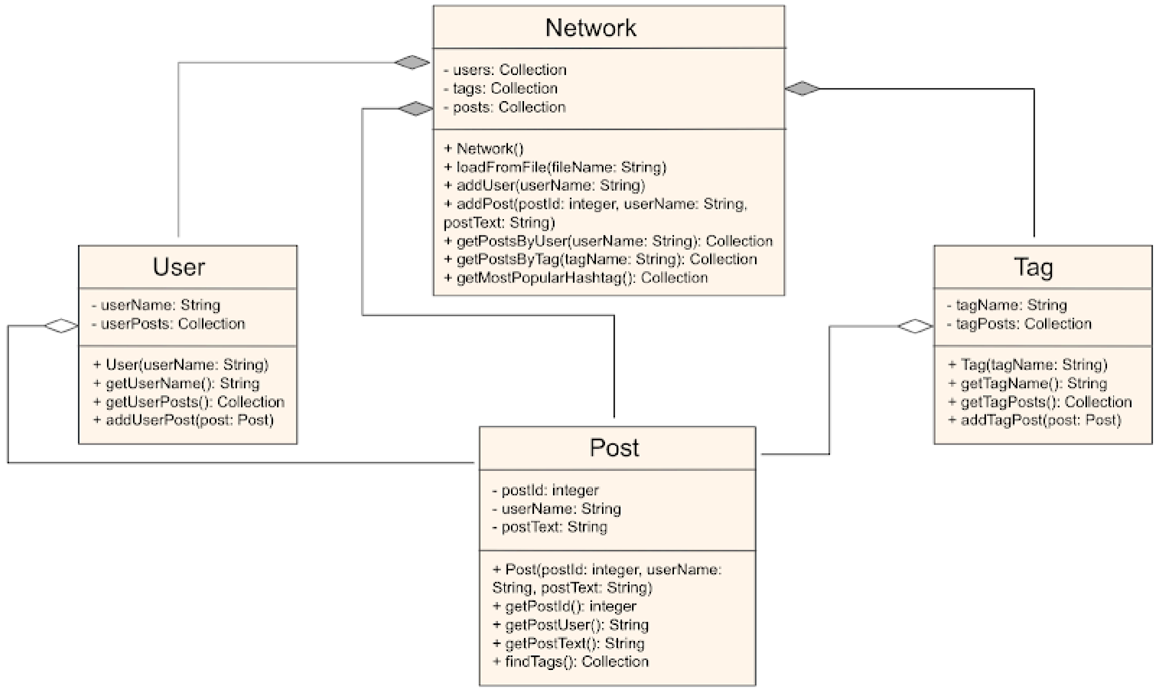
1. Practice with classes (constructors, getters, setters, and encapsulation).
2. Working with dynamic memory and pointers.
3. Using vectors.
4. Additional practice with file I/O.
5. Additional practice with exceptions.

Introduction

Many of you love Piazza given the immediate help you can get by using the platform. Alas, Piazza will no longer be available next semester and you are tasked with creating a replacement. You decide to use the skills you have gained through this course to come up with the platform CPPeers! As you are a big fan of social media, you decide to design this platform as a network of users and posts. You choose an object-oriented approach.

Getting Started

- Get the starter code
  - `Network.h/.cpp`: the class definition for the fabulous cppeers network; an interface to all users, posts, and hashtags.
  - `User.h/.cpp`: the class definition for the users of the cppeers network.
  - `Post.h/.cpp`: the class definition for the posts on the network.
  - `Tag.h/.cpp`: the class definition for the hashtags included within posts.
  - `main.cpp`: this file is useful for testing your code locally; also, contains the choice menu.
- Get the data files
- Review the code.
  - Read header and source files.
  - Compile and run the initial state of the starter code and submit to Gradescope to get an idea of the test cases.
  - Review the organization of the classes. The following UML class diagram shows relationships among `Network`, `User`, `Tag`, and `Post` classes used for this program.



- Look at the dependencies in classes. Best to implement in approximately the following order:
    1) Tag
    2) Post
    3) User
    4) Network
    5) main()
- Allowed includes:
    - `<iostream>`
    - `<fstream>`
    - `<sstream>`
    - `<vector>`
    - `<string>`
    - `<cctype>`
    - `<stdexcept>`
    - `"Post.h"`
    - `"User.h"`
    - `"Tag.h"`
    - `"Network.h"`
- Refer to the appendix as needed.

Recommendations

- As you develop your code, you only need to implement the methods/functions you need at the moment. You don't have to get everything done at once. It is not a good strategy to do so.
- You can utilize `main.cpp` to test the functions that you have implemented so far. Alternatively, you can create a separate tester file with its own main function.
- Plan and think before you code (write test cases first).
- **Test locally and start early**.

Sample Execution

```
Welcome to CPPeers
The options are:
1. load data file and add information
2. show posts by user
3. show posts with hashtags
4. show most popular hashtag
9. quit
--------> Enter your option: 1
Enter filename: TAMU-csce121-small.txt
Added User ajitjain
Added User michaelm
Added Post 42412212 by ajitjain
Added Post 58687888 by michaelm
Added Post 42353253 by michaelm
Welcome to CPPeers
The options are:
1. load data file and add information
2. show posts by user
3. show posts with hashtag
4. show most popular hashtag
9. quit
--------> Enter your option: 2
Enter username: michaelm
When an #array is passed to a #function, it decays to a pointer.
You can use pass-by-reference, if you need multiple updated
values in the calling #function. It also helps save memory and
CPU cycles.
Welcome to CPPeers
The options are:
1. load data file and add information
2. show posts by user
3. show posts with hashtag
4. show most popular hashtag
9. quit
--------> Enter your option: 3
Enter tagname: #dynamic
When deallocating #dynamic arrays, use delete[].
Welcome to CPPeers
```

```
The options are:
1. load data file and add information
2. show posts by user
3. show posts with hashtags
4. show most popular hashtag
9. quit
--------> Enter your option: 4
#function
```

Requirements

- You will be implementing the four classes as well as the functions in `main.cpp`.
- Constructors should **use** member initialization lists.
- Exceptions should have meaningful descriptions.
- The program must compile without warnings or errors.

Post Class

We provide

- Post class definition (`Post.h`)
- Constructor (`Post.cpp`)
    - `Post::Post(unsigned int postId, string userName, string postText)`
        - Throw `std::invalid_argument` if any of the following are true:
            - `postId` is zero
            - `userName` is empty
            - `postText` is empty
- Getters (`Post.cpp`)
    - `unsigned int Post::getPostId()`
    - `string Post::getPostUser()`
    - `string Post::getPostText()`

You implement in Post.cpp

- Methods / Member functions
    - `vector<string> Post::findTags()`
        - Return a vector containing unique candidate tags extracted from postText
            - Tags are words within `postText` that begin with a `'#'`.
            - Tags are not case-sensitive, i.e. #happy and #HAPPY are the same tag
            - Remove punctuations (`'!'`, `','`, `'.'`, `'?'`) at the end of each extracted tag, if any.
            - As an example, the following need to be treated as the same, and need to be added as the lowercase version (`#happy`) to the vector:

            - `#happy`
            - `#Happy`
            - `#HaPpY`
            - `#HAPPY`
            - `#happy!`
            - `#happy!!`
            - and so on...
        - When finding tags, this function returns all candidate strings beginning with `‘#’`.
            - Note: The function does NOT filter out invalid `Tags`, such as `#`, `#1a23`, or `#!happy`. Such checks are performed in `Tag`’s constructor (see `Tag`). Relatedly, the reason for removing punctuation above is to identify unique candidate strings, not validation.

User Class

We provide

- User class definition (`User.h`)

You implement in User.cpp

- Constructor
    - `User::User(string userName)`
        - Throw `std::invalid_argument` if any of the following are true:
            - `userName` is empty
            - `userName` does not start with a letter `‘a’ - ‘z’`
            - `userName` contains uppercase letters `‘A’ - ‘Z’`
                - In other words it should be all lowercase letters
- Getters
    - `string User::getUserName()`
    - `vector<Post*>& User::getUserPosts()`
- Methods / Member Functions
    - `User::addUserPost(Post* post)`
        - If post is `nullptr`, throw `std::invalid_argument`

Tag Class

We provide

- class definition (`Tag.h`)

You implement in Tag.cpp

- Constructor

        - `Tag::Tag(string tagName)`
            - Throw std::invalid_argument if any of the following are true:
                - `tagName` length is less than 2
                - the first character of `tagName` is not `‘#’`
                - the second character in `tagName` is not from `‘a’ - ‘z’`
                - `tagName` contains uppercase letters `‘A’ - ‘Z’`
                    - In other words it should be all lowercase letters
                - there is one or more consecutive punctuations at the end of the `tagName` (e.g., `#happy!`, `#happy!?`, etc.), where punctuation is one of the following: `('!', ',', '.', '?')`
- Getters
    - `string Tag::getTagName()`
    - `vector<Post*>& Tag::getTagPosts()`
- Methods / Member Functionsusername
    - `Tag::addTagPost(Post* post)`
        - If post is `nullptr`, throw `std::invalid_argument`

Network Class

We provide

- `Network` class definition (`Network.h`)

You implement in Network.cpp

- Methods / Member Functions
    - `Network::addUser(string userName)`
        - If a user with this name already exists, throw `std::invalid_argument`
            - User names are not case sensitive. For example, the following are treated as the same and added as the lowercase version (`ajitjain`)
                - `ajitjain`
                - `AjitJain`
                - `AJITJAIN`
                - `AjItJaIn`
                - and so on…
            - Let any exceptions from the `User` constructor go through, i.e., do not catch exceptions from the `User` constructor
            - When creating a new `User` object, remember that we are working with dynamic memory (Use the `‘new’` keyword to allocate memory on the heap)
            - If no exception, add the user to the `Network` data member users, and at the end of the function, output "`Added User `" followed by the username to standard output
    - `Network::addPost(unsigned int postId, string userName, string postText)`

        - ■ If a post with this id already exists or if no user with this name exists, throw `std::invalid_argument`
        - ■ Adding a post to the network is a multi-step process and includes message passing among various components to accomplish the following:
            - ● Creation of the post (see `Post`; remember we are working with dynamic memory)
            - ● Addition of the post to the Network data member posts
            - ● Addition of the post information to the corresponding user (see `User`)
            - ● Extraction of candidate hashtags contained within the post (see `Post`)
            - ● For each candidate hashtag: search (within the collection of tags stored in the `Network`) OR create (see `Tag`; only if the tag does not exist and remember we are working with dynamic memory)
                - ○ Note: When attempting to create a `Tag`, if an exception is thrown (by `Tag` constructor), catch the exception and resume processing for the next candidate hashtag
                - ○ Remember to add a newly created tag to the Network data member tags
            - ● addition of the post information to each tag (see `Tag`)
        - ■ Let any exceptions from the `Post` constructor go through, i.e., do not catch exceptions from the `Post` constructor
        - ■ If no exception, at the end of the function, output `"Added Post "` followed by the post ID, “`by `”, and username to standard output
    - ○ `Network::loadFromFile(string fileName)`
        - ■ If file could not be opened, throw `std::invalid_argument`
        - ■ Refer to the Data Files section, which specifies the format of `User` and `Post` information included in the file. Add users and posts as you read the lines from the file by calling `addUser()` and `addPost()`.
        - ■ If the file content does not match the specified format, throw `std::runtime_error`. This includes:
            - ● `User` or `Post` entry not following the specified format
            - ● Unknown entry (neither `User` nor `Post`)
        - ■ Note: You throw `std::invalid_argument` only when the file could not be opened. If there are `std::invalid_argument` thrown by add user and post operations, you catch them here and `throw std::runtime_error` instead.
    - ○ `vector<Post*> Network::getPostsByUser(string userName)s`
        - ■ If `userName` is empty or not found, throw `std::invalid_argument`
        - ■ Note: Assuming you followed the multi-step process for `addPost()` above, you now do not need to traverse all the posts in the `Network`
    - ○ `vector<Post*> Network::getPostsWithTag(string tagName)`
        - ■ If `tagName` is empty or not found, throw `std::invalid_argument`
        - ■ Note: Assuming you followed the multi-step process for `addPost()` above, you now do not need to traverse all the posts in the `Network`
    - ○ `vector<string> Network::getMostPopularHashtag()`
        - ■ Return the tag which occurs in the maximum number of posts. In case of a tie, return all such hashtags. If the network has no hashtag, return an empty vector.

■ Note: A hashtag should be counted only once per post. Assuming you followed the multi-step process for `addPost()` above, you now do not need to traverse all the posts in the `Network` and extract tags again from each post.

`main.cpp`

Use the menu example in the Overview above to fill in the gaps when implementing the following helper functions. **You will need to uncomment the loop in main when you have finished all other tasks**.

- `void processPostsByUser(Network& cppeers)`
- `void processPostsWithHashtags(Network& cppeers)`
- `void processMostPopularHashtag(Network& cppeers)`

Appendix

Data File Format

The information format is given below. Each line starts with the string **User** or **Post**, which identifies the type of information. Based on the type of information, one or more values follow. In the case of `User`, the value that follows is a username. In the case of `Post`, values that follow are *`PostId`*, *`UserName`*, and *`PostText`*. Implementing the functions listed in the previous section will allow you to check whether the read information is valid or not.

User Information Format

```
User UserName
```

Example

```
User ajitjain
```

Post Information Format

```
Post PostId UserName PostText
```

Example

```
Post 42412212 ajitjain When deallocating dynamic arrays, use delete[].
```

Example valid file

```
User ajitjain
User michaelm
Post 42412212 ajitjain When deallocating #dynamic arrays, use delete[].
Post 58687888 michaelm When an #array is passed to a #function, it
decays to a pointer.
Post 42353253 michaelm You can use pass-by-reference, if you need
multiple updated values in the calling #function. It also helps save
memory and CPU cycles.
```

Useful functions and classes

You may use any of the functionality provided by C++ strings and C++ vectors in your solution. Most students may find the following operations useful:

- `getline`
  - An example of how to use this is provided in `Cppeers-main.cpp`
- `vector`
  - `push_back(elem)`: Adds a new element elem at the end of the vector
  - `size()`: Returns the number of elements in the vector
  - More information about vectors is available in the Vector class Overview section
- `istringstream`
  - You already used this in a labwork. If you need to refresh your memory, an example of how to use this is provided in Cppeers-main.cpp

`Vector` class Overview

`std::vector` is a C++ standard class that provides the functionality of dynamically allocated arrays. You can learn more about C++ vectors in the documentation (link to std::vector documentation) or in the Zybook readings.

A simple example of using `std::vector` to store a list of consecutive int values is below:

```
#include <iostream>
#include <vector>
int main() {
    // n is not fixed (providing the same capability as dynamic arrays)
    size_t n = 0;
    std::cout << "Enter value of n: ";
    std::cin >> n;

   std::vector<int> v;
    for (size_t i = 0; i < n; i++) {
        v.push_back(i); // add i to the last position in v
    }

   // let's print the elements in a vector
    // we can use a for loop as before
    for (size_t i = 0; i < n; i++) {
        std::cout << v.at(i) << " " ;
    }
    std::cout << std::endl;
    return 0;
}
```