Assignment 5: Treebook

Due Friday, May 23rd at 11:59PM

Overview

The newest Stanford social media startup is Treebook, and you're a founding member of the team! To get the product off the ground and compete with an unnamed, completely legally unaffiliated app from Harvard, you've been assigned the task of implementing user profiles.

For this assignment, you will be implementing parts of a class to allow for operator overloads, as well as to modify some aspects of the special member functions.

There are two files you'll work with for this assignment:

- `user.h` - Contains the declaration for the `User` class that you will extend with special member functions and operators.
- `user.cpp` - Contains the definition of the `User` class.

To download the starter code for this assignment, please see the instructions for **Getting Started** on the course assignments repository.

Running your code

To run your code, first you'll need to compile it. Open up a terminal (if you are using VSCode, hit `Ctrl+`` or go to **Terminal > New Terminal** at the top). Then make sure that you are in the `assign5/` directory and run:

```
g++ -std=c++20 main.cpp user.cpp -o main
```

Assuming that your code compiles without any compiler errors, you can now do:

```
./main
```

which will actually run the `main` function in `main.cpp`.

As you are following the instructions below, we recommend intermittently compiling/testing with the autograder as a way to make sure you're on the right track!

> [!NOTE]
>
> Note for Windows
>
> On Windows, you may need to compile your code using
>
> ```
> g++ -static-libstdc++ -std=c++20 main.cpp user.cpp -o main
> ```
>
> in order to see output. Also, the output executable may be called `main.exe`, in which case you'll run your code with:
>
> ```
> ./main.exe
> ```

Part 1: Viewing Profiles

Take a look at the `user.h` header file. Your coworkers have begun to write a `User` class that will store the name and friends list of each user who joins your social media platform! In order to keep this class super efficient, they have chosen to represent the list of friends as a raw pointer array of `std::string` (kind of like how a `std::vector` stores its elements behind the scenes). Thankfully, they have already written logic for creating a new `User` and for adding friends to an existing `User`'s friend list (`add_friend`), but they've begun to experience some strange issues when working with `User` objects.

To begin with, there's no easy way to print information about each `User` object to the console, which has made debugging at Treebook difficult. To help your coworkers out, write an `operator<<` method that prints a `User` to a `std::ostream`. **This operator should be declared as a friend function in `user.h` and implemented in `user.cpp`.** For example, a user named `"Alice"` with friends `"Bob"` and `"Charlie"` should give the following output when printed to the console:

```
User(name=Alice, friends=[Bob, Charlie])
```

Note: `operator<<` should not print any newline characters.

> [!IMPORTANT]
> In your implementation of `operator<<`, you will need to access and loop through the `_friends` private field of the `User` class in order to print out a user's friends. Normally, you cannot access private fields inside of a class in a non-member function—in this case, we can get around this

> restriction by marking `operator<<` as a **friend function inside of the `User` class.** See the slides for Tuesday's lecture for more information!

Part 2: Unfriendly Behaviour

With the help of your `operator<<`, your coworkers have been able to make good progress on the social media app. However, they can't quite wrap their head around some seemingly bizzare issues that occur when they try to make copies of `User` objects in memory. Having recently taken CS106L, you suspect that it might have something to do with the special member functions (or the lack thereof) on the `User` class. To fix this issue, we'll implement our own versions of the special member functions (SMFs) for the `User` class, and remove some of the others for which the compiler generated versions are insufficient.

To be specific, you will need to:

1. Implement a destructor for the `User` class. To do so, implement the `~User()` SMF.
2. Make the `User` class copy constructible. To do so, implement the `User(const User& user)` SMF.
3. Make the `User` class copy assignable. To do so, implement the `User& operator=(const User& user)` SMF.
4. Prevent the `User` class from being move constructed. To do so, delete the `User(User&& user)` SMF.
5. Prevent the `User` class from being move assigned. To do so, delete the `User& operator=(User&& user)` SMF.

In performing these tasks, you are expected to make changes to **both** the `user.h` and `user.cpp` files.

> [!IMPORTANT]
> In your implementations of points 2 and 3 above, you will need to copy the contents of the `_friends` array. Recall from Thursday's lecture on special member functions that you can copy a pointer array by first allocating memory for a new one (possibly within a member initializer list), and then copying over the elements with a for loop. Make sure that you also set the `_size`, `_capacity`, and `_name` of the instance you are changing as well!

Part 3: Always Be Friending

After making changes to the special member functions, you've been able to scale out Treebook across Stanford and word has started to spread at other universities! However, your coworkers and you have found that some common use cases for the `User` class are either inconvenient or impossible given how the class is currently written, and you think you might be able to fix this by implementing some custom operators.

You will overload two operators for the `User` class. **Please implement both of these operators as member functions** (i.e. declare them inside of the `User` class in `user.h` and provide implementations in `user.cpp`).

`operator+=`

The `+=` operator will representing adding a user to another user's friend list. This should be symmetric, meaning that adding, for example, Charlie to Alice's friend list should cause Alice to also be in Charlie's list. For example, consider this code:

```cpp
User alice("Alice");
User charlie("Charlie");

alice += charlie;
std::cout << alice << std::endl;
std::cout << charlie << std::endl;

// Expected output:
// User(name=Alice, friends=[Charlie])
// User(name=Charlie, friends=[Alice])
```

The function signature for this operator should be `User& operator+=(User& rhs)`. Note that like the copy-assignment operator, it returns a reference to itself.

`operator<`

Recall that the `<` operator is required to store users in a `std::set`, as `std::set` is implemented in terms of the comparison operator. Implement `operator<` to compare users alphabetically by name. For example:

```cpp
User alice("Alice");
User charlie("Charlie");

if (alice < charlie)
  std::cout << "Alice is less than Charlie";
else
  std::cout << "Charlie is less than Alice";

// Expected output:
// Alice is less than Charlie
```

The function signature for this operator should be `bool operator<(const User& rhs) const`.

🚀 Submission Instructions

Before you submit the assignment, please fill out this short feedback form. **Completion of the form is required to receive credit for the assignment.** After filling out the form, please upload the files to Paperless under the correct assignment heading.

Your deliverable should be:

- `user.h`
- `user.cpp`

You may resubmit as many times as you'd like before the deadline.