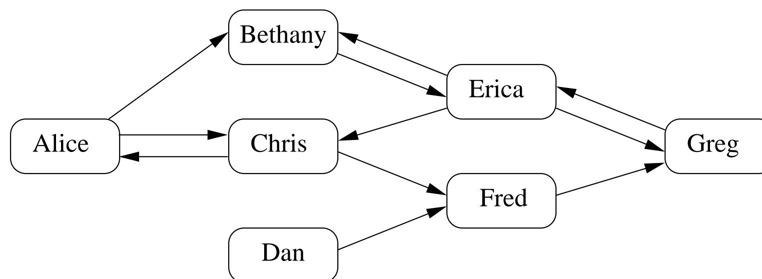
CSCI-1200 Data Structures — Spring 2014
Homework 8 — Friendly Recursion

Online friend networks such as Facebook, Google+, and Twitter are a popular way to communicate with friends and meet new people. Each person can make a personal webpage or profile and then list and link to the pages of their closest friends, as illustrated in the diagram below. Note: With Facebook, friendships are mutual (both people must agree to establish the friendship), but for this homework we'll model our friendships more like the one-way friendships of Google+ or Twitter ("following" or "sharing" with another person can be done without that person's explicit agreement). There are lots and lots of people out there, and we'll explore this very complex and interwoven database of information. We'll use a generalization of linked lists and trees called graphs. Thus far we have seen nodes in linked lists storing 1 or 2 pointers to other nodes. In graphs, nodes may have any number of pointers to other nodes. Moreover, instead of having a root or head (and/or tail) pointer through which the rest of the data are accessed, we can access the contents of the graph starting at any node. We will build a graph data structure to represent people linked by friend-of relationships. *Please read the entire handout before you begin.*

We provide the *class declaration* for three classes: a `Person` class that stores the person's name and a list of pointers to their friends; a `Message` class to represent messages that can be passed along links in the network; and a `Graph` class that stores a vector of all the people and messages currently in the system. Note: In both the diagram below and the `Person` data structure, friendship relationships are directed: i.e., Chris listed Fred as a friend, but Fred did not list Chris as a friend. We also provide a `main()` function that parses an input file to build the graph. Your main task is to finish the implementation of these classes and ensure the program works correctly with no memory errors or memory leaks (as verified by the memory debuggers Valgrind and/or Dr. Memory).



Making a Party Invitation List

Though decreasing in frequency, sometimes people actually get together in non-virtual forums to socialize. We'd like to automatically generate an invitation list for such a party from the online friend database. Implement this functionality by writing a *recursive* function to collect a list of all the people within n forward links from the host. Given the host's name and this integer the function returns a list of the people to invite, in alphabetical order. In the above example, the invitation list for a party hosted by Chris with friend chain length $n = 2$ will contain 4 names: `"Alice"`, `"Bethany"`, `"Fred"`, and `"Greg"`, in that order.

Network Complexity If we assume that each person is allowed to put at most k people on their list of friends, what is the maximum number of people who will be at a party with chain length n (including the host), as a function of n? For example, if $n = 0$, there is only 1 person in the network. If $n = 1$ and $k = 5$, there are at most 6 people in the network. Include this analysis in your README.txt file.

Graph Structure Input and Output

Your program will read from an input file and output to `std::cout`. The program expects 1 command line argument, the input file. You will start with an empty graph and modify it as directed by the input requests,

one per line, in the input file. In the following description of these requests, **name** and **msg** refer to strings for the people and message. A helper function in the `main.cpp` file parses the message phrases, which are given in double quotes. There will be nothing tricky about the input formatting for this assignment.

add_person name : Add the person with the given **name** to the graph.

remove_person name : Completely remove person **name** from the graph, including both the friendship links *from* that person and any links *to* that person.

add_friendship name1 name2 : Add a forward link from person **name1** to person **name2**.

remove_friendship name1 name2 : Remove the forward link from person **name1** to person **name2**.

add_message name msg : Add a message **msg** to the person **name**.

print : For each person in the graph, output all of the people they consider to be their friend (the forward links). This output should be alphabetical (both the list of people and the list of friends). Similarly, output all the messages held by each person.

print_invite_list name n : Create the alphabetic invitation list for a party hosted by **name** that contains all people who can be reached with $\leq n$ links from the host through forward friend links.

pass_messages : Move the messages between people (along forward links only).

Make sure you do simple error checking to ensure that the operations are allowed. For example, don't add a person if they are already in the graph, and don't remove a link between people that are not already linked or when at least one of the people is non-existent. If an error such as these occurs, print a simple but descriptive message to `std::cerr`.

The Graph Class

Your task is to implement the classes, including a `Graph` class. A `Graph` object (your program will build only one each time it is run) stores a vector of *pointers* to `Person` objects, and a vector of *pointers* to the messages currently in the system. Each `Person` object will store its name and a list of *pointers* to the other `Person` and `Message` objects it is linked to. Each `Message` object will store its name and a *pointer* to the `Person` that currently holds that message.

Note that you may not use a `std::map` anywhere in your code. At first this might seem to make the program less efficient. However, since the `Graph` object stores pointers to `Person` objects and `Person` objects store pointers to other `Person` objects, your code has direct access to the people a person has forward links to by simply following the pointers. There may be places where you think a map is useful and you may be right, but to ensure you get practice with pointers and graphs, **maps are not allowed on this assignment**.

We have provided a number of files from our solution to get you started (`graph.h`, `person.h`, `message.h`, and `main.cpp`). Study this code carefully as you work.

Pointers to Objects and Vectors of Pointers

There are several syntactic challenges in this assignment:

- The `Graph` class stores a `std::vector` of pointers to `Person` objects (and each `Person` class stores a `std::list` of `Person` objects). This may cause some confusion in the syntax when iterating through these data structures and accessing pointers. As an example, the `Person` class has a `get_name` member function that returns a `string&`. Here is code to print the names of all the people from the graph:

```
for (vector<Person*>::iterator p = m_people.begin(); p != m_people.end(); ++p)
   cout << (*p)->get_name() << endl;
```

The iterator `p` refers to a pointer to a `Person` object and `(*p)` uses the iterator to access the `Person*` (the pointer). The `->` follows the pointer to the `Person` object and calls its `get_name` member function. The parentheses are required here to ensure that the operators are applied in the correct order. You may need to use the `(*p)->` idiom at several places throughout your code.

- In this course you must always deallocate (with `delete`) all memory that was dynamically allocated (with `new`), even if you are exiting the program and know that the operating system will clean it up for you. The submission server will run Valgrind on your program to help you check for problems.
- When an object (e.g., a `Person` object) has multiple pointers to it, a question arises about when it is appropriate to delete the object. In this case it should occur when the person itself is being removed, either through the `remove-person` command or in the destructor when the `Graph` is being destroyed. It is not appropriate to delete the `Person` object when friendship links (pointers between people) are being eliminated.

Passing Messages

Once your basic friend graph data structure has been implemented and debugged, you will pass messages between the people in the graph. Each time the `pass_messages` command appears in the input file, `Message` objects will each move up to one (forward) link through the graph. The program will choose *uniformly at random* from the possible moves. If a message is currently held by a person that has listed 3 friends, then there is a 1/4 chance of the message not being passed and a 1/4 chance of the message moving to each of the neighboring people. Like HW7 we recommend you use the provided MersenneTwister random number generator to implement the message passing feature.

Since people may be removed from the graph at any time, you will also need to consider what happens if there are any `Message` objects in the person at that time. The simplest thing to do in these cases is to simply remove these messages from the system (don't forget to `delete` memory as appropriate.)

Mutating Messages

For extra credit you can implement the game of *Telephone* or *Chinese Whispers* where the message is slightly garbled each time it is passed through the system. Read about the game here:

`http://en.wikipedia.org/wiki/Chinese_whispers`

After many steps the message might not resemble the initial message at all! When humans play the game the intermediate phrases will usually contain only real words, but the meaning might become nonsensical. Describe your implementation in your README.txt file.

Additional Instructions & Submission

Do all of your work in a new folder named `hw8` inside of your homeworks directory. Be sure to make up new test cases and don't forget to comment your code! Submit at least one new test case named `my_input.txt` that fully exercises the corner cases in the implementation. Describe what is interesting/challenging about this test case in your README.txt file.

Use the provided template `README.txt` file for any notes you want the grader to read. **You must do this assignment on your own, as described in the "Academic Integrity for Homework" handout. If you did discuss the problem or error messages, etc. with anyone, please list their names in your `README.txt` file.** When you are finished please zip up your folder (including any new test cases) exactly as instructed for the previous assignments and submit it through the course webpage.