CS 103 – The Social Network

1 Introduction

This assignment will be part 1 of 2 of the culmination of your C/C++ programming experience in this course. You will use C++ classes to model a social network, allowing users to be added, friend connections made, etc.

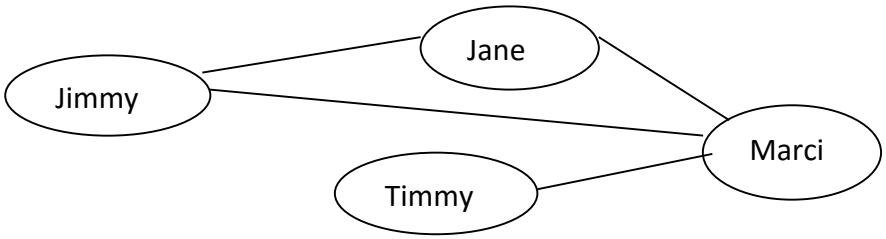
2 What you will learn

In this assignment you will:

1. Create your own C++ classes and learn the syntax of defining data members and methods.
2. Use File I/O and stringstreams to read/write text files containing structured data (records)
3. Understand the abstract representation of graphs (vertices=Users, edges=Friend connections)

3 Background Information and Notes

Graph Representation: A graph is an abstract representation of the connections between objects. Each object (known as a vertex) is connected to others via 'edges'. Visually, graphs are quite intuitive:



Here we see four users (vertices) and the friend connections (edges) between them. Many interesting problems in computer science can be represented as graphs and algorithms defined to extract information from the connections within the graph. For our purposes we will create a class to represent a single User which contains information about each user (name, etc.) as well as the edge connections (friend relationships) for the user. Note: In our system edges are '**undirected**' meaning friend connections are reflexive (i.e. if I'm a friend with you, you are a friend with me). Other problems using graphs may require '**directed**' edges (i.e. non-reflexive relationships). Thus, when you add a friend connection from user1 to user2, be sure to add the connection from user2 to user1.

Defining a File Format: This program requires you to use File I/O techniques to read and write the user database from and to a file. Whenever we write a file containing structured data we need to define a file format (syntax) so we

understand how to parse the file. We will represent all information as ASCII text and use the format specified below.

`1:`	*single number representing how many users are in the database*
`2:`	*id_0*
`3:`	`<TAB>`*user_name*
`4:`	`<TAB>`*birth year*
`5:`	`<TAB>`*zip code*
`6:`	`<TAB>`*ids of friends (separated by spaces)*
…	…
`n-4:`	*id_k*
`n-3:`	`<TAB>`*user_name*
`n-2:`	`<TAB>`*birth year*
`n-1:`	`<TAB>`*zip code*
`n:`	`<TAB>`*ids of friends (separated by spaces)*

An example file is shown below with 3 users where Professor Redekopp is friends with both TA Tommy and Student Jimmy, whereas they are only friends with Professor Redekopp and not each other.

```
3
0
  Professor Redekopp
  1978
  90018
  1 2
1
  TA Tommy
  1983
  90007
  0
2
  Student Jimmy
  1991
  90089
  0
```

Note: a <TAB> character is represented as '\t' in C/C++.

Note: if you are using gedit with "insert spaces instead of tabs" enabled, and you want to make more input files by hand, you can use copy-paste from the sample file to create more tab characters.

4 Prelab

None.

5 Procedure

1. Create a new folder under your "`cs103`" directory and download the skeleton.

```
$ cd cs103
$ mkdir network1
$ cd network1
$ wget ftp://bits.usc.edu/cs103/network1.tar
$ tar xvf network1.tar
      Downloads:
        user.h      - Models a single user and their information
        network.h   - Network class contains all the Users and implements
                      the menu options
        social_network.cpp  - main() function that instantiates the
                            - network and implements the menu I/O
        users_small.txt     - Text file containing a small social
                              network database
        users_new.txt       - Expected output from sample run
        grade_input         - Canned sequence of menu options for
                              testing purpose
        Makefile            - Compile script
        Readme.txt          - Feedback from you to us
```

   You will need to create your own `user.cpp` and `network.cpp` files

'Network' classes member functions may not be modified. You may however add other member functions. User class functions may be added, modified, removed as you see fit provided it implements the menu options correctly and adheres to the requirements, unless otherwise noted.

2. Global variables must not be used for this project.

3. Define a class to model a `User` (in `user.h`). It should contain:

User
+ User(*your choice*) + add_friend(int id) : void + delete_friend(int id) : void + *accessors* + *mutators, if needed*
- _id : int - _name : string - _year : int - _zip : int - _friends : ???

   a. An integer id (should be set to the entry/index in the users array/vector of the Network class described below where this user is located; thus it should start at 0)

   b. A user name that consists of a first and last name separated by spaces. You should store the full name in a single string. It is recommended to use a C++ 'string' object rather than a simple C character array to represent the name. [Hint: use the getline(ifstream &if, string &s) function]

   c. An integer indicating the birth year of the user.

   d. An integer indicating the user's zip code.

   e. A list of integer entries for friend connections (will not exceed 100 entries). Each entry will be the corresponding integer ID of their friend. Depending on your implementation you may need to keep another variable to track how many friend connections are present in the list.

4. The `User` class should support the following operations:

    a. Constructor (depending on your implementation you may provide the user info as arguments to the constructor or initialize them later via mutator methods).

    b. A destructor (it may be empty if no dynamic memory is allocated for a user).

    c. An `add_friend` method accepting the ID of a user to add as a friend. If the indicated user is already a friend of this user, do nothing (i.e. don't add it a second time).

    d. A `delete_friend` method accepting the ID of a user to delete as a friend. If the friend list is implemented as a vector/array, then that ID should be 'removed' and all following friend ID's moved up one slot. If the friend ID provided is NOT in the list of friends, do nothing.

    e. Individual accessor methods to return the user's name, their user ID, birth year, and zip code, and a **pointer** or **reference** to the friend list (array or vector)

    f. Any other methods you feel necessary or helpful

5. Define a `Network` (in `network.h`) class. It should contain:

    a. A list of at most 100 Users (you can implement this with an array or vector and you can choose whether to store the actual User objects or pointers to dynamically allocated User objects). Depending on implementation, you may need to have a variable to store how many users are currently being stored.

6. The `Network` class should support the following operations:

Network
+ Network() + add_user(*your choice*) : void + add_connection(string s1, string s2) : int + remove_connection(string s1, string s2) : int + get_id(string name) : int + read_friends(char* fname) : int + write_friends(char *fname) : int
- // array or vector of User objects/ptrs

    a. You **must define a default (no-argument constructor)** and and possibly destructor (if needed)

    b. A `read_friends` method that initializes all of the network's information from a file. This method should accept a char * (string) indicating the name of the file to read the users from and return 0 on success, -1 on failure.

    c. A `write_friends` method that writes all of the network's information to a file. This method should accept a char * (string) indicating the name of the file to write the users to and return 0 on success, -1 on failure.

d. An `add_user` method to add a User to the Network database. You can decide whether this method should accept a User object (or pointer to a User object) or the component information pieces and then have `add_user` create a User object with the given info.

e. An `add_connection` method accepting two strings (format: first name + last name separated by a space in between) corresponding to the names of the Users to make friends. Return 0 on success, -1 if either of the users are invalid. We will not attempt to create a self-connection (from user1 to user1).

f. A `remove_connection` method accepting two strings (format: first name + last name separated by a space in between) of the names of Users to delete friend connections. Return 0 on success, -1 if either of the users are invalid.

g. A `get_id` method accepting a user name and returning the corresponding ID for that user, or -1 otherwise.

7. Write a main program (`social_network.cpp`) that will create a Network object and then read Users data from a text file whose name is specified as a command line argument. E.g.,

```
./social_network users_small.txt
```

The program should then loop through the progression of displaying a menu of options, accepting the user input, and processing the option. The menu options should be as follows. **Your menu must accept the Option numbers indicated below**.

Here are the 7 actions your program should support, with examples of their use.

a. Option 1. Add a user

- When chosen the user should provide their name (both first and last), birth year, and zip code input on the same line.

```
> 1 Steph Curry 1988 94027
```

b. Option 2. Add friend connection

- When chosen the user should provide the two usernames to make friends. If a user does not exist, print an error message and continue.

```
> 2 Mark Redekopp Steph Curry
```

c. Option 3. Remove friend connection

- When chosen the user should provide the two usernames to make friends. If a user does not exist or the Users are not friends, print an error message and continue.

```
> 3 Mark Redekopp Juju Smith
```

d. Print users

- Print a list of users and their associated info in a table format (ID, Name, Birth Year, Zip Code). Format this table so it displays nicely to the screen with fixed column widths (use the `iomanip` library and the `setw` manipulator.

```
> 4
```

In the output, there should be a header row with column titles and then one user per row. An example is shown below.

```
ID      Name            Year    Zip
============================================================
0.      Mark Redekopp   1978    90018
1.      Juju Smith      1995    90271
2.      Tommy Trojan    1885    90089
3.      Max Nikias      1945    91103
4.      Jane Doe        1994    94027
```

e. List friends

- When chosen, the user should provide a username and then your program should print all the friends of that user (along with their information) in the same table form used for Print users. If the user does not exist, print an error message and continue.

```
> 5 Mark Redekopp
```

Example output:

```
ID      Name            Year    Zip
============================================================
2.      Tommy Trojan    1885    90089
3.      Max Nikias      1945    91103
5.      Steph Curry     1988    94027
```

f. Write to file

- When chosen, the user should provide a filename to write the user database to. Your program should write the current

User database and friend connections to the specified file given by the user in the same format as the input file. E.g.,

```
> 6 users_new.txt
```

(See **'users_new.txt'** downloaded with your other files for the expected output.)

g. Exit the program

- Your program should exit the program on ANY invalid command number (i.e. -1, 7, 8, etc.)

8. If any of your functions returns an error code, please print an appropriate error message to the screen and then return to the menu and accept more commands.

9. Approach: To break this into pieces consider the following approach to build up your program:

   a. Write the user.cpp file with appropriate User methods

   b. Make sure it compiles and runs. Make a small program to test all of its methods.

   c. Write the network.cpp file implementing the methods to add users.

   d. Write a simple main program that implements the Menu system but with just the options to add users and print all users, and quit.

   e. Continue to add in methods of the Network class (add friend connections, remove friend connections, print all friends of a user) one at a time and implement that option in the menu system, testing each one individually before moving on to the next.

   f. Lastly, implement the File I/O capability to read in the database from a file or write it back out to a file.

- It might help you to use stringstreams and the getline(istream, string) function to do your parsing of the database file:

```
string myline;
ifstream myfile(filename);
...
getline(myfile, myline);
stringstream ss(myline);
// you may now add the following in a loop or other code
ss >> val;
```

10. A `Makefile` has been provided so you can compile your code using '`make`'. For your own benefit, view the Makefile in a text editor and try to understand its format and rules. It would be very beneficial if you did some of your own investigation on how Makefiles and the make utility works.

11. Run your program: `$ ./social_network users_small.txt`
    You can enter menu options manually to test certain features of your program.

12. We also provide you a text file containing commands in a format that we will use to grade your submission (we'll provide other commands but you should ensure this set of commands functions correctly). You should open the file `grade_input` in a text editor to understand what command it is performing and ensure your program outputs expected values.

    ```
    $ ./social_network users_small.txt < grade_input
    ```

    In Unix/Linux, the contents of a text file can be ***redirected*** to your program as if someone had typed it from the keyboard without using File I/O methods (i.e. still using cin). This permits and makes automated scripting easier. To use redirection, simply enter the normal program command line but then use a '<' symbol followed by the text file with the command.

13. Ensure all commands work and that you can write the updated database to a new file. Inspect that file to ensure its format and data are correct. Try loading it up in a new run of the program.

6 Review

None