CS246—Assignment 3 (Fall 2015)

Due Date 1: October 26, 4:55pm
Due Date 2: November 4, 4:55pm

Questions 1a, 2a, and 3a are due on Due Date 1; the remainder of the assignment (including any bonus submissions) is due on Due Date 2.

Note: You must use the C++ I/O streaming and memory management facilities on this assignment. Moreover, the only standard headers you may `#include` are `<iostream>`, `<fstream>`, `<sstream>`, `<iomanip>`, `<string>`, `<cassert>` and `<cstdlib>`. Marmoset will be programmed to **reject** submissions that violate these restrictions.

Note: Each question on this assignment asks you to write a C++ program, and the programs you write on this assignment each span multiple files. For this reason, we **strongly** recommend that you develop your solution for each question in a separate directory. Just remember that, for each question, you should be *in* that directory when you create your zip file, so that your zip file does not contain any extra directory structure.

Note: Questions on this assignment will be hand-marked for style, and to ensure that your solutions employ the programming techniques mandated by each question.

Note: Sample test cases have been provided for each question. These are available under the `a3/sample-tests` directory.

Note: Starter code, when applicable, is provided under the `a3/starter-code` directory.

1. Consider the following object definition for an "improved"[1] string type:

```
struct iString {
  char * chars;
  unsigned int length;
  unsigned int capacity;
  iString();
  iString(const char *a);
  iString(const iString &a);
  ~iString();

  iString &operator=(const iString &other);
};
```

   You are to implement the undefined constructors and destructors for the `iString` type. Further, you are to overload the input, output, assignment, addition, and the ~operator (which we call the prefixes operator) according to the following examples:

[1] For some definition of improved. Namely, overloading unary ~operator.

```
iString s1; // Create an empty string (length is zero, chars is NULL)
iString s2("foobar"); // Create a string initialized with the word "foobar"
iString s3(s2); // Call the copy constructor,
                //initialize s3 contents to be a copy of the contents of s2
iString s4;

cin >> s1 >> s4; // Read in whitespace-delimited strings from stdin
cout << s1 << " " << s2 << " " << s3 << " " << s4 << endl; // Print iStrings to stdout

s1 = s1 + s4; // Concatenate s1 and s4
s2 = s2 + "baz"; // Concatenate s2 and the word "baz"
s3 = ~s2 ; // unary operator that concatenates the prefixes in s2
           //Since s2 is foobar, s3 will be ffofoofoobfoobafoobar
```

Implementation notes

- The declaration of the `iString` type can be found in `istring.h`. For your submission you should add all requisite declarations to `istring.h` and all routine and member definitions to `istring.cc`.
- **You are not allowed use the C++ `string` type to solve this question.** However, you may include the header `<cstring>` and use the functions declared therein.
- Becoming familar with `cin.peek()` and the `isspace` function located in the `<locale>` library may aid you in solving this question. In particular, note that `cin.peek()` does not by default skip leading whitespace. Also note that `cin.peek()` returns an `int`.
- The provided driver (`a3q1.cc`) can be compiled with your solution to test (and then debug) your code. Please keep in mind that the purpose of the test harness is to provide a convenient means of verifying that code you are asked to write is working correctly. Therefore, although some effort has been expended to make the harness reasonably robust, we do not guarantee that it is perfect, as that is not the point. The test harness should function correctly if you use it as intended; it may fail horribly if you abuse it. But the point of your testing is to verify *your* code, rather than the harness, and so test cases that attempt to find flaws in the harness are not required in your test suites. Note also that if your test case causes a new `iString` object to be created in the heap, then your test case must cause the same `iString` object to be deleted as well.

Deliverables

(a) **Due on Due Date 1**: Design a test suite for this program (call the suite file `suiteq1.txt` and zip the suite into `a3q1a.zip`)

(b) **Due on Due Date 2**: Implement this `iString` type in C++. Submit the istring.h and istring.cc files as a zip file, `a3q1b.zip` (**do not submit a3q1.cc, we will use our own copy**).

2. Graphs are commonly used to represent relationships between individual entities. In this problem, you are to use a graph to store `follower` information between people in a social network. Moreover, you will implement operations to update the graphs to reflect changes in the social network.

   Your program will accept the following commands on standard input (commands and components of commands are separated by arbitrary, non-zero, whitespace):

   - `+ UserID` – add a user with the `UserID` to the graph. If a user with the id `UserID` already exists, the graph remains unchanged and the program outputs the message "User

`UserID` already has an account[newline]". If the add is successful, the program outputs the message "User `UserID` has joined the social network[newline]".

- `- UserID` – remove a user with the `UserID` from the graph. If a user with the id `UserID` does not exist, the graph remains unchanged and the program outputs the message "User `UserID` does not have an account[newline]". If the remove is successful, the program outputs the message "User `UserID` has deleted their account[newline]".

- `follow UserID1 UserID2` – let `UserID1` follow `UserID2`. If `UserID1` or `UserID2` does not exist, the command has no effect on the graph and outputs a message similar to the - (remove) command above. Note that for consistency of output, given two user-ids, the program only checks the second id if the first id exists in the graph. If `userID1` is the same as `userID2`, the graph remains unchanged and the program outputs the message "Cannot follow yourself[newline]". If `userID1` is already following `userID2`, the graph remains unchanged and the program outputs the message "User `UserID1` is already following `UserID2`[newline]". If the command is successful, the program outputs the message "User `UserID1` is now following `UserID2`[newline]".

- `unfollow UserID1 UserID2` – let `UserID1` unfollow `UserID2`. See above for what to output if `UserID1` or `UserID2` are not in the graph. If `userID1` is the same as `userID2`, the graph remains unchanged and the program outputs the message "Cannot follow yourself therefore cannot unfollow yourself[newline]". If `UserID1` is not following `UserID2`, the graph remains unchanged and the program outputs the message "User `UserID1` was not following `UserID2`[newline]". If the command is successful, the program outputs the message "User `UserID1` is no longer following `UserID2`[newline]".

- `print UserID` – print the users `UserID` is following. If the `UserID` is not in the graph, a message similar to the one in the - (remove) command is output. If `UserID` is not following anyone, then the program outputs "User `UserID` is following no one[newline]". Otherwise, the program outputs the message "User `UserID` is following `UserList`[newline]", where `UserList` is a single user id if `UserID` is only following one user and a comma separated list of ids otherwise. For consistency of output, the follow list is output in the order `UserID` began to follow users.

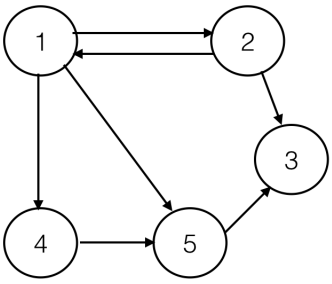
- `printall` – convenience command which calls `print UserID` for all users in the social network in the order in which they joined the network.

- `list UserID n` – prints the number of unique users reachable from `UserID` through up to n levels of follow relationships. For example, if Jim (UserID 1) follows Tom and Tom follows David, then David is reachable from Jim via two levels of relationships. Note that if Jim also follows Alice who follows David, then `list 1 2` would only count David once (even though there are two paths to David). If `UserID` is not in the graph, a message similar to the one in the - (remove) command is output. You may assume that n is greater than 0.

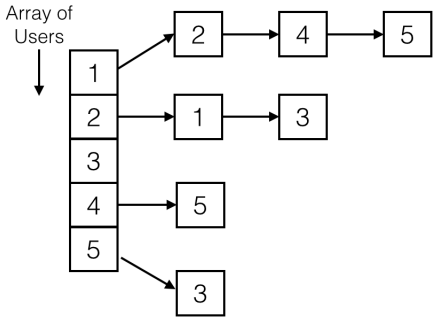
- `include filename` – reads the file `filename` and executes the commands contained therein. There is no restriction on what command might occur in the specified file.

- `quit` – quits the program.

Implementation guidelines:
You must adhere to the following implementation guidelines. We will use the `Adjacency List` representation of directed graphs. For a directed graph G with n nodes, the adjacency list consists of an array containing n elements, each of which represents a node in G. Each element of this array contains a list of nodes associated with that element. More precisely, the list associated with array element i contains node j, if there is an edge from node i to node j in G.



Directed graph

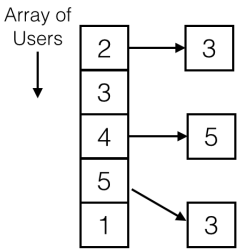


Adjacency list
for directed graph

In the example above, the adjacency list on the right represents the directed graph on the left. The head of each linked list, must be a node in the array. For our program, the heads of the linked lists are the registered users. In a typical implementation of adjacency lists, the order of other nodes (apart from the head) in the list does not matter. For example, it would be valid to arrange the first linked list from the diagram above, as 1→4→2→5. However, for consistency of output, we force the following ordering: each linked list contains the followers that the user represented by the head of the linked list (the array element) is following, in the order in which the user followed them. Therefore, the linked list 1→2→4→5, represents the users that userid 1 is following. Moreover, userid 1 must have followed users in the order: 2 then 4 then 5.

Also, consider the linked list: 1→2→4→5. Now suppose that user 1 chooses to unfollow user 2. This will cause the linked list to be modified to 1→4→5. If user 1 now decides to follow user 2 again, the linked list will be: 1→4→5→2. In other words, the linked list always stores followed users in the order they were `last` followed.

Consider the adjacency list shown above again. If user 1 was removed and then added again, this would result in the adjacency list shown below. Notice how user 1 now appears at the end of the array of users (since they were added last). Also note that user 2 is no longer following user 1 since the follow lists of all users must be updated when a user deletes their account.



Below is a sample program interaction that produces the adjacency list shown in the original example (input in **bold**). This example is available in the assignment directory.

+ 1
User 1 has joined the social network
+ 2
User 2 has joined the social network
+ 3
User 3 has joined the social network
+ 4
User 4 has joined the social network
+ 5
User 5 has joined the social network
+ 1
User 1 already has an account
+ 6
User 6 has joined the social network
follow 1 2
User 1 is now following 2
follow 1 4
User 1 is now following 4
follow 1 5
User 1 is now following 5
follow 2 1
User 2 is now following 1
follow 6 1
User 6 is now following 1
- 6
User 6 has deleted their account
- 7
User 7 does not have an account
print 1
User 1 is following 2,4,5
follow 2 3
User 2 is now following 3
follow 2 3
User 2 is already following 3
follow 2 4
User 2 is now following 4
unfollow 2 4
User 2 is no longer following 4
follow 2 6
User 6 does not have an account
follow 4 5
User 4 is now following 5
follow 5 3
User 5 is now following 3
printall
User 1 is following 2,4,5
User 2 is following 1,3
User 3 is following no one
User 4 is following 5
User 5 is following 3

quit

- Use the provided files `user.h` and `adjacency.h` as a starting point for building the adjacency list data structure. You must follow the recommended fields in the `User` and `AdjacencyList` classes. You may add additional fields/methods as needed.
- You must put the member functions that operate on the adjacency list in separate `user.cc` and `adjacency.cc` files. Your main function should reside in a separate file a3q2.cc. Your submission will be handmarked to ensure that you follow proper procedures for building separately-compiled modules.
- Note that the file `adjacency.h` restricts the number of users in the social network to 100. This simplification was made so that the `users` array in the `AdjacencyList` class does not have to be dynamically allocated. Your program (and testing) may assume that the number of registered users will not exceed this limit.
- The `User` and `AdjacencyList` classes must define appropriate constructors for initialization and destructors for proper deallocation of nodes.
- You must deallocate all dynamically allocated memory by the end of the program using appropriately implemented destructors; Marmoset and handmarkers will be checking for this.

(a) **Due on Due Date 1:** Design a test suite for this program (call the suite file `suiteq2.txt` and zip the suite into `a3q2a.zip`).

(b) **Due on Due Date 2:** Implement this program in C++ (put your mainline program in the file `a3q2.cc`, and include all `.h` and `.cc` files that make up your program in your zip file, `a3q2b.zip`).

Bonus: Add to your program the command `dot filename` which generates a DOT file named `filename` containing the graphical representation of the current social network as a directed graph. To gain the bonus points (a 5% bump to your A3 marks), you may not ask anyone for help in doing this bonus. Information on DOT can be found at the following URL: `https://en.wikipedia.org/wiki/DOT_(graph_description_language)`

- **Bonus Submission:** Submit the complete program as a zip file, `a3q2bonus.zip`, to the special a3q3Bonus Marmoset project. Only submissions made to this directory will be considered for the bonus.

3. For this problem, the classes you write must be implemented using the `class` keyword. In this problem you will write a C++ program to administer the game of Tic-Tac-Toe (`http://en.wikipedia.org/wiki/Tic-Tac-Toe`), which involves two players, X and O (uppercase alphabet not the value zero). Players take turns claiming squares on a 3x3 grid, until one player has claimed a straight line of three squares (that player wins the game) or until the grid is full (and no one wins). Your program will play several rounds of this game and report the winner. A sample interaction follows (your input is in **bold**):

game stdin stdin
`X's move`
NW
`O's move`
C
`X's move`
SW
`O's move`
N

```
X's move
W
X wins
Score is
X 1
O 0
quit
```

In between games, two commands are recognized:

- `game sX sO` Starts a game. `sX` denotes the name of the file from which X's moves will be taken. Specifying the string `stdin` instead of a filename indicates that the moves will come from `cin`, i.e., X's moves will be interactive. Similarly for O.
- `quit` Ends the program

Within a game, players take turns claiming squares. The nine squares are arranged as follows:

```
NW N NE
W  C E
SW S SE
```

When a player's moves come from stdin, it is considered invalid input to claim a square that has already been taken. On the other hand, when the moves come from a file, it is hard to know in advance what squares will have been taken. Thus, when the moves come from a file, a player's move is defined to be the next square in the file that is not already claimed. Note that it would be redundant for a square to occur more than once in the file, and therefore we consider any file that contains a square more than once to constitute invalid input.

A sample interaction where both players' moves come from files is presented below. Suppose that `movesX.txt` contains the following:

```
NW SE NE C E N S W SW
```

Suppose that `movesO.txt` contains the following:

```
C SE SW E N NW NE S W
```

Then the interaction would be as follows:

```
game  movesX.txt movesO.txt
X's move
(plays NW)
O's move
(plays C)
X's move
(plays SE)
O's move
(plays SW)
X's move
(plays NE)
O's move
(plays E)
X's move
(plays N)
```

```
X wins
Score is
X 1
O 0
quit
```

Note, in particular, that when the moves come from a file, your program prints to stdout the move that was made (e.g., `(plays NW)` above).

You may assume that if a player's moves are taken from a file, then the file will contain enough moves to complete the game. You may also assume that if a player's moves are taken from a file, then that file exists and is readable.

Within a game, play alternates between X and O. X plays first in odd-numbered games (starting from 1), and O plays first in even-numbered games.

A win is worth one point. A loss is worth no points. If a game is drawn (no one wins), then no points are awarded for that game. In the case of a draw, print `Draw` to stdout, instead of `X wins` or `O wins`. The score is printed after every game.

The quit command is considered valid input only when no game is in progress.

To structure this game, you must include at least the following classes:

- `ScoreBoard` which tracks the number of games won by each player and the current state of the board. This class must be responsible for all output to the screen, except `X's move` and `O's move` (these will be printed by code in the `Player` class, described below).
- `Player` which encapsulates a game player, and keeps track of the source from which a player is receiving input.

Each `Player` object must possess a pointer to the scoreboard. Each player object must be responsible for registering its move with the scoreboard by calling a `ScoreBoard::makeMove` method. This method should take parameters indicating which player is calling the method, and the move being made. If necessary, a player object may query the board by calling a method `ScoreBoard::isOccupied` to find out whether a given position is taken. This method would only be called if the player object's input comes from a file, for the purpose of determining the next unoccupied position in the file.

Your main program will be responsible for keeping track of which player's turn it is. The main program will call a method in `ScoreBoard` to start a game, whenever the user enters a `game` command. In addition, the main program will alternately call a method on the two player objects that will cause the player to get the next move from its input source and then pass that move on to the scoreboard.

The chain of method calls is therefore roughly the following:

- main program, in response to a `game` command from the user, calls a method `ScoreBoard::startGame` to initiate a game.
- main program calls a method for each of the two player objects, to pass to them their respective input streams
- main program alternately calls `Player::makeMove` for player objects A and B. This method should take no parameters.
- The `Player::makeMove` method will be responsible for fetching the next move and calling the `ScoreBoard::makeMove` method to register the move with the scoreboard.

Your solution must use `const` declarations for variables, members, and parameters whenever possible.

Your solution must not leak memory.

Some sample test cases are available in the assignment directory. No starter code is provided for this question.

Note: You may assume that all input is valid.

Due on Due Date 1: Design a test suite for this program (call the suite file `suiteq3.txt` and zip the suite into `a3q3a.zip`).

Due on Due Date 2: Full implementation in C++. Your mainline code should be in file `a3q3b.cc`, and your entire submission should be zipped into `a3q3b.zip` and submitted. Your zip file should contain, at minimum, the files `a3q3b.cc`, `scoreboard.h`, `scoreboard.cc`, `player.h`, and `player.cc`. If you choose to write additional classes, they must each reside in their own `.h` and `.cc` files as well.

Bonus: Add a graphical component to your game using XWindows graphics. To gain the bonus points (a 5% bump to your A3 marks), you may not ask anyone for help.

To attempt this question you must first make sure you are able to use graphical applications from your Unix session. If you are using Linux you should be fine (if making an ssh connection to a campus machine, be sure to pass the `-Y` option). If you are using Windows and putty, you should download and run an X server such as XMing, and be sure that putty is configured to forward X connections. Instructions on installing an XServer were provided to you in the Getting Started PDF document made available through Piazza at the beginning of the term. It can also be found in the repository. You can confirm that you have configured the XServer correctly if you can run a program such as `eyes`.

If working on your own machine, make sure you have the necessary libraries to compile graphics. A graphicsdemo directory is available within the assignment directory. From within the graphicsdemo directory, try executing the following:

```
g++ window.cc graphicsdemo.cc -o graphicsdemo -lX11
./graphicsdemo
```

Note: (thats lower case L followed by X and one one)

Note for Mac OS users: On machines running newer Mac OS you will need to install XQuartz. `http://xquartz.macosforge.org/`. Once installed, you might have to explicitly tell g++ where `X11` is located. If the above does not work, browse through your Mac's file system looking for a directory `X11` that contains directories `lib` and `include`. You must then specify the lib directory using the `-L` option and the include directory using the `-I` (uppercase i) option. For example, on my MacBook I used:

```
g++ window.cc graphicsdemo.cc -o graphicsdemo -lX11 -L/usr/X11/lib -I/usr/X11/include
```

You know that the above test is successful if the following happens:

- Two windows open
- The big window prints the strings Hello!, ABCD, Hello! followed by rectangles containing a rainbow of colours
- The small window prints ABCD

Bonus implementation hint: In the graphicsdemo directory you are provided with a class `Xwindow` (files `window.h` and `window.cc`), to handle the mechanics of getting graphics to display. Declaring an `Xwindow` object (e.g., `Xwindow xw;`) causes a window to appear. When the object goes out of scope, the window will disappear (or you could allocate it dynamically

in which case the window will disappear when you delete the object). The class supports methods for drawing rectangles and printing text in different colours. To implement the bonus all you should need to use is Xwindow's `fillRectangle` method and different colours (e.g. Xwindow::Red, Xwindow::Green, Xwindow::White etc). See graphicsdemo.cc for usage examples.

Bonus Requirement: You are **not** expected to take input graphically i.e. via mouse clicks. In other words, the game will continue to work with the user typing in game commands on standard input. The only requirements are that when the game begins a graphical window is displayed with a 3x3 grid. You should choose two different colours to represent the X and O players and these colours should be different from the background colour. Whenever a player makes a move, the appropriate square in the window should change to that user's colour. When a game finishes, and a new game is started, the graphical display should be reset as well.

Bonus Submission: Submit the complete program as a zip file, `a3q3bonus.zip`, to the special a3q3Bonus Marmoset project. Only submissions made to this directory will be considered for the bonus.