CSCI-1200 Data Structures — Fall 2010
Homework 8 — City Chase

This assignment explores 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 head pointer (and perhaps a 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 cities linked by train service. Once this is implemented, we will play a "Pursuit-Evasion" game on the graph. One or more TAs (the pursuers) will chase one or more students (the evaders) across the country trying to collect homework. If a TA and a student (who has not yet submitted his/her homework) ever end up in the same city, the student must submit his or her homework to the TA. *Please read the entire handout before you begin.*

Graph Structure Input and Output

Your program must read from an input file and output to `cout`. The program expects 3 command line arguments: the input file, and 2 strings that indicate the selected evader and pursuer strategies. If the user-controlled evader strategy is specified, the program will also request interactive input from `cin` to play the game. 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, **cname** and **pname** refer to strings that give names of cities and people. You may assume the names are strings that contain no whitespace, and you may assume that case matters. There will be nothing tricky about the input formatting for this assignment.

add-city cname : Add the city with the given **cname** to the graph.

remove-city cname : Completely remove city **cname** from the graph, including all links.

add-link cname1 cname2 : Add a link from city **cname1** to city **cname2** and a link from city **cname2** to city **cname1**. All links between cities are bi-directional.

remove-link cname1 cname2 : Remove the link from city **cname1** to city **cname2** and from city **cname2** to city **cname1**.

place-evader pname cname : Add an evader (student) named **pname** to the city **cname**. Each student starts the game with a single (unsubmitted) homework assignment.

place-pursuer pname cname : Add a pursuer (TA) named **pname** to the city **cname**. The TA starts the game with no homework assignments and will attempt to collect homework assignments during game play.

print : For each city in the graph, output all of the cities that city is linked to. These are its immediate neighbors. This output should be alphabetical (both the list of cities and the list of each city's neighbors). Output the location of the evader and pursuers.

tick num-ticks : Move the evader and pursuers through the graph for the specified number of "ticks" (or until all homework is submitted) using the strategies specified on the command line. If a student who has not yet submitted his/her homework and a TA are in the same city at the end of a "clock tick", the student must submit his homework to the TA.

Once all commands in the input file are processed, the final "score" of the game is printed. The score is simply the number of homeworks held by the evaders (the students) vs. the pursuers (the TAs).

Make sure you do simple error checking to ensure that the operations are allowed. For example, don't add a city or a link between cities if it is already in the graph, and don't remove a link between cities that are not already linked or when one or both cities is non-existent. If an error such as these occurs, the output error message does not need to describe it precisely; but it should just indicate that the operation failed (see example output). The program should continue after one of these failures.

The Graph Class

The main problem in this assignment is to implement a `Graph` class. A `Graph` object (your program will build only one each time it is run) stores a vector of *pointers* to `City` objects, a vector of *pointers* to the evaders and a vector of *pointers* to the pursuers (who are all of type `Person`). Each `City` object will store its name, a vector of *pointers* to the other `City` objects it is linked to. Each `Person` object will store its name and a *pointer* to the `City` where that person is currently located.

Note that you may not use a `std::map` or `std::set` anywhere in your code. At first this might seem to make the program less efficient. However, since the `Graph` object stores pointers to `City` objects and `City` objects store pointers to other `City` objects, your code has direct access to the cities any one city is linked 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 & sets are not allowed**.

We have provided a number of files from our solution to get you started (`graph.h`, `city.h`, `person.h`, `main.cpp`, `tick.cpp`, `evader.cpp`, and `pursuer.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` and `City` classes should each store a `std::vector` of pointers to `City` objects. This may cause some confusion in the syntax when iterating through the vector and accessing pointers. As an example, the `City` class has a `name` member function that returns a `string&`. Here is code to print the names of all the cities:

  ```
  for (vector<City*>::iterator p = m_cities.begin(); p != m_cities.end(); ++p)
     cout << (*p)->name() << endl;
  ```

  The iterator `p` refers to a pointer to a `City` object and `(*p)` uses the iterator to access the `City*` (the pointer). The `->` follows the pointer to the `City` object and calls its `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 `City` 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 city itself is being removed, either through the `remove-city` command or in the destructor when the `Graph` is being destroyed. It is not appropriate to delete the `City` object when links (pointers) to it are being eliminated.

Playing the Game

Once your graph data structure has been implemented and debugged, you can test the pursuit-evasion game component of the assignment. At each timestep, or "`tick`", of the game the evaders and pursuers may each move up to one link along the graph. The moves of the different `Person` objects within one timestep are defined to happen *simultaneously*. Two functions `evader_choice` and `pursuer_choice` (in the files `evader.cpp` and `pursuer.cpp`, respectively) parse the 2nd and 3rd command line arguments and call the appropriate helper function to control the motion of the `Person` objects. For example, the command line:

```
graph.exe game_test.txt evader_QUERY_USER pursuer_MOVES_RANDOMLY
```

will set up a game between user-controlled evaders and randomly-moving pursuers. On each tick, the program uses `cout` to list the possible moves for each evader, starting with the option to stay at the current city followed by the cities that are directly reachable from the current city in alphabetical order.

Each option is numbered starting with 0 (see the sample output). The program pauses for the user to enter an integer via `cin`. Once the evader has moved, each pursuer will select a move *uniformly at random* from the possible moves. If a pursuer is in a city that is linked to 3 other cities, then there is a 1/4 chance of the pursuer staying at the current city and a 1/4 chance of the pursuer moving to each of the neighboring cities. This random choice is implemented with the `rand()` function, which returns an integer between 0 and `RAND_MAX` (a constant defined by your system). To use this function, we `#include <stdlib.h>`. Modulo arithmetic converts this to a random number in the appropriate range. We need to *seed* the random number generator in `main` using `srand()`. If you're interested, you can read up on pseudo-random number generators and why some methods of generating random numbers are better than others.

At the end of the timestep we check to see if any of the TAs (pursuers) is in the same city as a student (evader) who has not yet submitted his/her homework. Note: if a pursuer and the evader "pass each other" riding in trains going in opposite directions between two cities, the evader has *not* been captured and does not have to submit his/her homework.

Since city removals may occur at any time within the program, you will also need to consider what happens if there are any `Person` objects in the city at that time. The simplest thing to do in these cases is to also remove these people from the game (don't forget to `delete` memory as appropriate.) Another strategy would be to force the people to get on a train out of town.

Alternate Strategies for the Evader and Pursuers & HW8 Contest

If you look carefully at the prototype for the pursuer and evader functions, you will see that they have read access to the full state of the system. How can the evaders take advantage of knowing the current location of all pursuers? Similarly, can the pursuers leverage knowledge of the location of all evaders who have not yet submitted their homework? What if the evaders know (or can guess) the strategy of the pursuers? How can multiple pursuers work together to corner an evader? What if the structure of the graph is only partially known to the game participants? (In our case the graph structure is fully accessible.) *Warning: Versions of these questions are open problems in theoretical computer science research!*

For the assignment, you must implement and submit an alternate strategy for both the evader and pursuer (you must write the functions `evader_MY_STRATEGY` and `pursuer_MY_STRATEGY`). We will have a contest and pit each submitted evader strategy against each submitted pursuer strategy. We will use both simple and complex graphs with 1 or more evaders and 1 or more pursuers. Prizes will be awarded to the best overall evader strategy and the best over pursuer strategies. Furthermore, the top performers in each category will receive extra credit on the assignment. For additional extra credit, you may create and submit interesting new test case graph networks and evader & pursuer starting locations on which to play the game.

To work correctly in the contest format, your strategies must be implemented in files named `evader.cpp` and `pursuer.cpp` and should *use only the provided* `Graph`, `City`, *and* `Person` *interfaces.* Any helper functions you use must be defined in the `evader.cpp` and `pursuer.cpp` files. To run the contest we will compile student A's `evader.cpp` file, student B's `pursuer.cpp` file, and all other files from the solution code. If your code does not compile properly for the contest, you will not receive full credit for this portion of the assignment.

Submission

Do all of your work in a new folder named `hw8` inside of your Data Structures homeworks directory. Please 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 exactly as instructed for the previous assignments and submit it through the course webpage.