CSE 326: Data Structures
Assignment #6
May 21, 1999
Due: Wednesday, June 2

Goals and Comments

The purpose of this assignment is to give you some experience putting together and designing appropriate data structures for a real application. It will also give you the opportunity to implement a simple graph algorithm.

You will be working in teams of two on this assignment, and submitting only one assignment per team. Your grade will be based equally on

- The design and analysis of your program, which we will evaluate from your writeup, and
- the quality and correctness of your implementation.

Unlike previous programming projects in this course, on this project you should feel free to use any pre-implemented data structures you like. For example, these could be your own implementations (e.g., lists and splay trees from the previous assignments in this course, or anything you implemented in a previous course) or anything you find in the Standard Template Library (though we will NOT be providing help with this, nor do we guarantee that everything in there works).

The `M326` Server

For this assignment you will be implementing the `M326` server. The `M326` server is similar to MUDs, MUSHes, MOOs, etc. which you may be familiar with. MUDs or Multi–User Dungeons are networked servers (much like chat servers) that allow users to connect to them (perhaps with `telnet`) and interact with other users and the server. Typical MUDs are role-playing adventure games where users are characters in a fictional world and they explore the virtual world fighting monsters.

`M326` is not quite so bloody. The virtual environment will consist of rooms connected together with doors. Users can connect to the `M326` server with `telnet`. Once connected, a user can walk from virtual room to virtual room, and talk with other people (users) that are in the same room. A user will also be able to find out the shortest path to the room a particular other person is in.

Part I: The `M326` Simulator

To simplify the implementation and debugging of the assignment, you are to implement it in two phases. For the first part of the assignment, you will implement a single–user version of the program that doesn't use the network. It will read in a map file, let a user (the person running your program) walk around from room to room, place fictitious people in rooms, and find the shortest distance between the user's current location and a person that has been placed in one of the rooms.

The Map

The virtual environment consists of rooms (vertices) connected to other rooms with doors (edges). This data will be read in from a file with the following format:

```
4
3 1 2 3
1 0
2 0 3
2 0 2
```

This corresponds to a 4 room world with:

room 0 Has 3 doors one to each of rooms 1, 2, and 3.

room 1 Has 1 door to room 0.

room 2 Has 2 doors one to each of rooms 0 and 3.

room 3 Has 2 doors one to each of rooms 0 and 2.

This corresponds to the following graph:

```
0-1
|\
2-3
```

More formally, the map can be described in terms of the parameters: n, the number of rooms, $d(r_i)$ the number of doors leaving room r_i, and $N(r_i)$ a list of the rooms adjacent to r_i (doors). The map format is then:

$$
\begin{array}{ll}
n & \\
d(r_0) & N(r_0) \\
d(r_1) & N(r_1) \\
 & \vdots \\
d(r_n) & N(r_n)
\end{array}
$$

An alternative map format (described at the end of the handout), in which rooms and doors are named, can be implemented for extra credit.

Commands for navigating the simulator

When the simulator starts up, the user is placed in room 0. The user will then execute commands that allow him/her to move from room to room, list the doors leaving the room, leave a person in a room, and find the shortest path from the user's current room to the room that a particular person is in. The syntax for these commands is the following:

`go <door>` This moves the user to the room on the other side of the the door given by `<door>`. When the user is in the new room, the rooms contents are displayed in the same fashion as if the user had executed the `look` command.

`look` This looks at the room that the user is currently in. It lists all the people in the room as well as the doors that leave the room.

`drop <person-name>` This drops a person named `<person-name>` in the room that the user is currently in. The name should be unique.

`find <person-name>` This finds the shortest path from the user's current location to the room that the person named `<person-name>` is in.

`who` This lists the names of all the people that have been dropped into the various rooms, and for each one, the name of the room they are in.

`quit` This quits the program.

The single-user program

Putting it all together, we can now describe the basic operation of the single-user program that simulates most of the functionality of the `M326` server. It will:

1. Read in the map file (the file name should be in the command line arguments).
2. Repeatedly:
   (a) Prompt the user for input (with `M326>`).
   (b) Read in a line of input.
   (c) Parse the line of input as a command.
   (d) Execute the command.

The user starts out in room 0.

If the map given above is in the file called `simple.map`, then a run of the simulator might look like:

```
orcas% M326sim simple.map
Welcome to the M326 Simulator

Room 0
```

```
People:
Doors: 1 2 3

M326> go 1

Room 1
People:
Doors: 0

M326> drop jason

Dropped jason in room 1

M326> look

Room 1
People: jason
Doors: 0

M326> go 0

Room 0
People:
Doors: 1 2 3

M326> go 2

Room 2
People:
Doors: 0 3

M326> drop anna

Dropped anna in room 2

M326> go 3

Room 3
People:
Doors: 0 2

M326> find jason

Take path: 0 1
```

```
M326> find anna

Take path: 2

M326> go 1

This room does not have a door to room 1.

M326> go 2

Room 2
People: anna
Doors: 0 3

M326> quit

Bye!
```

Design

The design decisions you will need to make include how to represent the graph so that you can efficiently navigate the room graph and find shortest paths between pairs of rooms, how to keep track of where people are and how to keep track of what people are in each room.

You should try to make your design efficient, so that all the commands run fast if the room graph is very large and if extremely large numbers of people are distributed (dropped) throughout the rooms.

Part II: The `M326` Server

Once you have written the simulator, you can (and should) use the same basic data structures and algorithms to implement your `M326` server. The more general `M326` server will work as follows. When it starts running, the virtual world (the graph describing the rooms and their interconnections) is set up. Other people (clients) can then connect to the server, walk around in the virtual world, and talk to other people that are also connected to the server (when they are in the same room in the virtual world). Thus, people sitting at different computers can all be connected to your server and can use it to move around this virtual world and chat with each other.

The structure of your server code will be as follows:

1. Read in the graph and set the server up to listen to a specified I/O port.

2. Repeatedly:

(a) Get a new server event. (There is a *server event* whenever a client connects to the server, sends a message to the server or disconnects from the server.)

(b) If the server event is a *client connection*, add the new client to the list of connected clients. (You'll want to prompt the user for his or her name when he/she first connects. The new client always starts out in room 0.)

(c) If the server event is a *client message*:

   i. Parse the client message as a command. (Commands will be things like "move to an adjacent room", or "talk to the people in the same room". The precise syntax is described below, and is very similar to that used in Part I.)

   ii. Execute the command.

(d) If the server event is a *client disconnection*, remove the client from the list of connected clients.

You should not make any assumptions about the number of clients that can connect.

All of the tricky networking code for the assignment has already been implemented for you in the `Server` and `Event` classes, described below. Your job will be to parse and implement client commands and keep track of the state of the virtual world. You will have already have done most of the work by implementing the first part of the assignment.

The commands

A client can execute (and your server must process) the following commands:

`name <person-name>` This sets the name of the client to `<person-name>`. Your server should encourage the client to run this command when he/she first connects. (Until the client has run this command, you can associate a number with each client, as a substitute for a real name.)

`go <door>` This moves the client to the room on the other side of the door given by `<door>`.

`look` This looks at the room that the client is currently in. It lists all the people (clients) in the room as well as the doors that leave the room.

`say <message>` This sends a message to every client in the same room as this client. If the name of the client issuing the command is `<person-name>`, then the message that will be sent to these clients is:

```
<person-name> says, "<message>"
```

`find <person-name>` This finds the shortest path from the client's current location to the room that the client named `<person-name>` is in.

`who` This lists the names of all clients that are connected to the server, and for each client, the name of the room that client is in.

`quit` This disconnects the client.

Note that these commands are essentially the same as those of the simulator from part I except that `drop` has been removed (since now new people, i.e. clients, enter the virtual world by connecting to the server) and `name` and `say` have been added. Clearly, now, instead of keeping track of the people that have been "dropped" in various rooms, your server will keep track of the clients that have connected, and where in the virtual world they are. For the client command `say`, you will be using the network, and sending data to the clients connected to your server that are in the same room as the client issuing the `say` command. You will be able to do this using the `Server` class.

Running the `M326` Server

When the `M326` server is started up, a particular I/O port is specified – this will be the port the server *listens* to. You should be able to run your `M326` server on `orcas` on a I/O port of your choice. When a client program, like `telnet`, is used to connect with the `M326` server, a port is also specified. This lets the `telnet` program know which port the `M326` server is listening on. So, for example, a `M326` server running on `orcas port 4444` can be connected to with `telnet orcas 4444`. You should be able to run your server on port 4445 like this:

```
orcas% M326 simple.map 4445
```

where `simple.map` is the name of your map file.

To explore the interface your `M326` server should have, you can connect to a solution `M326` server at: `orcas 4444`. You can do this with `telnet` as:

```
orcas% telnet orcas 4444
```

You may also use clients other than `telnet`. A Java based client program that is a little bit more user friendly than `telnet` will be posted to the course web pages.

Note that only one server can be running on a port at once. If your server cannot connect to the port, then it is likely that someone else is using that port. You should just try another one. Only use ports between 1024 and 32,000[1].

Implementation details

As previously mentioned, you are free to use any pre-implemented data structures you like (e.g. your splay trees from the previous assignment, anything available in the Standard Template Library, etc.) However, you must implement the shortest path computation for the command `find` yourself.

You will be implementing your `M326` server by using some C++ classes (that we will give you) that provide a convenient interface to the POSIX networking system calls that are

[1]Ports below 1024 are reserved for special operating system programs like the FTP, SMTP (email), NNTP (news), and HTTP (web) servers. The port number is a 16 bit unsigned number which is why the port must be less that 32,000. You actually might be able to use up to 64,000 but to be safe it is good not to use the most significant bit of the port number as some buggy implementations of TCP/IP might not implement the sign bit correctly.

available on most UNIX platforms. The two C++ classes you will be interfacing with are the `Server` class and the `Event` class. You can download these from the course web page. Basically, these classes allow you to connect your server to a port, and obtain server events corresponding to client connections, disconnections, and messages.

You will be using the following methods that have been implemented for you:

`Server::Server(int portnum)` This constructor initializes the server to listen to port `portnum`. `portnum` should be between 1024 and 32,000.

`Event *Server::read()` This reads the next event from the server. This is a blocking operation. If there is no event available, then it will wait until there is an event available. It returns a pointer to an `Event` object that encapsulates the three types of client events that can happen.

`ostream &Server::client(int c)` This returns an `ostream` that can be used to send data to the client given by client id `c`.

`void Server::disconnect(int c)` This disconnects client `c`. No `CLIENT_DISCONNECTED` event will be generated.

`int Event::getClient()` This returns the client id for the client that is the source for this `Event`. You can assume that at any time the active client ids are unique integers. (Keep in mind that if a particular client disconnects, his/her client id may be reused for a client that connects later on.)

`int Event::getType()` This returns the type of the `Event`. This is either `CLIENT_CONNECTED`, `CLIENT_DISCONNECTED`, or `CLIENT_MESSAGE`. If it is `CLIENT_MESSAGE` then you should use `Event::getMessage()` to get the string data for the message that the client sent.

`int Event::getMessage()` This returns the string message that the client sent. It is a `'\0'` terminated array of characters corresponding to doing a `getline()` on an input stream coming from the client. The newline symbol has already been removed. When you get an `Event` that is `CLIENT_MESSAGE` type, you must use `getMessage()` to get the actual data. You are responsible for deallocating this data.

Example

Here is a small code segment to help you understand how a program might interact with the `Server` and `Event` classes. This code implements a very simple server that just repeats to each client that connects to it whatever that client types. This code illustrates the process of getting the server set up, dealing with client connection and disconnection events and sending a message to a client. This code is also posted on the course web page.

```
#include "Server.h"
#include <stdio.h>

int main(int argc, char **argv)
```

```
{
   int port;                        // port to listen on.
   Server *server;                  // pointer to the server object.

   /*
    * read in arguments for port to run server on.
    */
   if (argc != 2 || sscanf(argv[1],"%d",&port) != 1)
   {
      cerr << "USAGE: " << argv[0] << " portnum\n";
      exit(-1);
   }

   /*
    * create server on port.
    */
   server = new Server(port);

   /*
    * if the server did not correctly connect,
    * then something went wrong.  quit!
    */
   if (!server->isConnected())
   {
      cerr << "ERROR: could not connect to port " << port << ".\n";
      exit(-1);
   }

   /*
    * run forever!
    */
   while(1)
   {
      /*
       * read in a new Message
       */
      Event *event;
      event = server->read();
      char *msg;

      /*
       * get the client number
       * for this event.
       */
      int client = event->getClient();

      /*
```

```
     * process the event.
     */
    switch (event->getType())
      {
      case CLIENT_MESSAGE:

         /*
          * if the event is a client message,
          * then print out that message to the client.
          */
         msg = event->getMessage();

   if (strcmp(msg,"quit")==0)
     server->disconnect(client);
   else
     (server->client(client)) << "You say \"" << msg << "\"\n";

         delete msg;
         break;

      case CLIENT_CONNECTED:

         /*
          * if the event is a client connection, then welcome
          * that client.
          */
         server->client(client) << "Welcome to the default "
            << "echo server number "
            << client << ".\n";
         break;

      case CLIENT_DISCONNECTED:
         /*
          * if the event is a client disconnection,
          * do nothing.
          */
        break;
      }

    delete event;
  }
}
```

Electronic turnin

Turn in an electronic copy of your source code, along with your `Makefile`. You program must be able to be run by executing `make` and then `search`. You can do this by running the `turnin` program on `orcas` as:

```
turnin -v -c cse326 Makefile *.cpp *.h
```

or a whole directory with:

```
turnin -v -c cse326 hw6/
```

Feel free to turn in a `README` file as well. It is very important that you turn in all of your files at once. **If you forget some files the first time then turn everything in again. The `turnin` program only keeps your last turn in.** Do not turn in object files `.o` or your compiled main programs.

Paper turnin

The paper turnin should contain the following:

1. Hardcopy of all source code used *that you have written yourself*. Please acknowledge the source of any code that you have not written yourself.

2. Give a high-level description and explanation of your design and data structure choices. Include a runtime analysis for each of the commands.

3. Provide output that demonstrates that the `M326` Simulator (Part I) works.

Optional Modifications

For a small amount of extra credit, feel free to make any of the following modifications.

Named rooms and doors

Have your simulator and your `M326` server use an alternative map format that has named rooms and named doors. An example is:

```
4

Red Room
3
1 east
2 south
3 southeast
```

```
Green Room
1
0 west

Pink Room
2
0 north
3 east

Lavender Room
2
0 northwest
2 west
```

The `go` command should take as its argument the name of the door, so, starting in the `Red Room`, you could go `south` then `east` to get to the `Lavender Room`.

If `anna` is in the `Pink Room`, `jason` is in the `Green Room`, and you are in the `Lavender Room` then `find anna` should result in:

```
Take path: west
anna is in the Pink Room.
```

And likewise, `find jason` would result in:

```
Take path: northwest east
jason is in the Green Room
```

As you can see, this map format is a bit more complicated. This map file should be line-oriented. To read in all n rooms repeatedly read in the first non-blank line as the name of the room, then the next line as the number of doors leaving the room. Subsequent lines will contain the room number that the door leads to followed by door name.

The go command

It might be convenient to make the `go` command implicit. That is if there are doors to room 1, then it might be nice to type `1` instead of `go 1` to get there. You could implement this by trying to parse the command that the user types, and if it does not match any of the existing commands, you should check and see if it matches the name of one of the doors in the room. If it does, then move the user through that door.

The say command

It is also nice for clients not to have to type a whole lot in order to talk to people in the same room. A common approach is to look for user input that begins with the `"` character. If it does, take the entire rest of the line as the message to be sent. For example, if the user, `Jason` types `"hi there!` then everyone in room sees the text:

```
Jason says, "hi there!"
```

Extra Credit

Use your imagination!!!! Come up with your own ideas for cool things to add to this program, and then, just do it!!!

(If you do this, be sure and describe what you've done in your writeup.)