CSC 209 Assignment 4, Fall 2013: Multi-User Dungeon

Due by the end of Wednesday December 4, 2013; no late assignments without written explanation.

In this assignment you will complete the code for a simple "multi-user dungeon" program, and add a new feature to it. This software consists of a "server" program which keeps track of where all of the objects and players are, and mediates interactions between the players; and a "client" program which each player runs on their own computer, which can be any unix or linux machine anywhere on the internet.

Step 1: Get to know the operation of the program

Play `/u/ajr/209/a4/standalone_en` for a little while. Explore by typing compass directions (n, w, s, e). Pick up and drop some objects. Type "help" to get a list of the commands you can type. Try "inv", both while holding some objects and while not holding any objects. Note that in the "get" and "drop" commands, you have to type the numbers of the objects, not their names.

More exciting is when you run the non-standalone version with a server and clients. Open three terminal windows on your computer. In one, type `/u/ajr/209/a4/server` to run the server. If it says that the default port number of 1234 is in use, you can try another port number by executing (for example) `/u/ajr/209/a4/server -p 1235`.

In the second terminal window, type `/u/ajr/209/a4/client_en localhost` to run the English-language client. Or if you had to choose a different port number, type something like `/u/ajr/209/a4/client_en localhost 1235` (no "–p").

In the third terminal window, type `/u/ajr/209/a4/client_fr localhost` to run the French-language client (perhaps adding a port number as before).

Into the clients you first have to type the name you will be known by as a player in the multi-user dungeon. Use different names in the two windows to learn more about how the program functions. Now type some commands in each of the clients, as you did with the standalone version. Note the different kinds of interactions which are possible between the players. They can see when another enters or leaves the room; the locations of objects are affected by the other players; and you can "poke" each other too, only if you're in the same room.

The protocol between client and server is language-independent, so people playing in different languages can still play together.

Step 2: Make your own copy of the code and try a partial 'make'

Create a new directory for your working files, and copy in everything from the distribution directory /u/ajr/209/a4 (you don't need the subdirectory).

If you just type 'make' now, it will fail because server.c is incomplete and deliberately syntactically incorrect in some places where you need to complete the code. So don't do that. (Well, it won't hurt if you do, but it won't fully work, either.)

However, you can make *some* of the project. Try "`make standalone_en`", and run the resulting program from your own directory. It's also possible to make client_en or client_fr, or standalone_fr.

There are some student-contributed language files for other languages in /u/ajr/209/a4/lang, but I can't vouch for the accuracy or propriety of the translation.

Step 3: Start looking at the code

Look at `things.c` to see some data about the objects in the game (like the newspaper). Look at the beginning of `map.c` to see some data about the connections between the places. Note the idiom "`sizeof x / sizeof x[0]`" to find the number of items in an array—for example, if each element of the array `x` is ten bytes, then `sizeof x[0]` (a particular one of those elements) is 10; and if the array altogether is 80 bytes (so `sizeof x` is 80), then the number of items in the array is 80/10 = 8. This avoids hard-coding the number of elements which are in the array, so you can add places or things by just adding to the array without having to change anything else. (Although you won't do that in this

(continued)

assignment.)

Look at `lang_en.c`. All of the strings are in here, making most of the program language-independent. So the program would never printf("You have been poked by %s\n", who); but rather, printf("%s %s\n", lang_poked_by, who);. If instead you compile with `lang_fr.c` it will speak French (not always correctly I'm afraid). Try `standalone_fr`. Type "make clean" and then "make standalone_en" and "make standalone_fr", and notice the "`gcc -o standalone_en ...`" and "`gcc -o standalone_fr ...`" lines and how most of the files are the same (ignore the –D options for now).

To make this system multi-user, we will have a server, and everyone playing will be clients of that server. Skim `PROTOCOL` to see how the clients communicate with the server. (Don't get bogged down by the introductory first three paragraphs.)

Skim `client.c` and note how it uses select() in do_something() so that it can at any time receive either a command from the user or a command from the server. Note how commands from the server are handled in do_something_server(), and how commands from the user are handled in docmd() (explode() turns the string into a sequence of tokens).

Despite all I say above about language-independence, the error messages are all in English. (That is, error messages such as "can't connect to server" are in English; errors within the game such as "There is no way to go that direction" are still output in the appropriate language.)

Step 4: Get to know the protocol between the client and the server

There is a compiled version of the completed server available in /u/ajr/209/a4/server for you to experiment with. Try running the server and also running the network tool *nc*, into which you can type commands to your server as described in the "lab stuff" part of tutorial 8 at

`http://www.cdf.toronto.edu/~ajr/209/tut/08/`

Note that the first thing the client has to send is their "handle"; type your name on the first line, then type commands according to `PROTOCOL`.

Also try the client by "playing server" by typing something like "`/bin/nc -l 1234`", and then running the supplied client (either my compiled version or your own). Note how the client_en and client_fr programs still use the same protocol.

(Note: if you complete the tutorial 8 instructions before the tutorial time, you can submit your work early, but you still need to go to the tutorial to get the point for the tutorial. Pick a lab partner who needs help, log in to their account rather than yours, and help them generate appropriate transcripts and submit them! (But still remember to run /u/ajr/209/present for both of you.))

Step 5: Complete the code for the server

The supplied server.c is missing most of main(). This function should contain (or call functions which contain) the main select() loop, which will accept new connections and will process requests from existing connections.

The supplied "struct client" (which you do not need to modify or add to) contains a "buf" element representing a command partially-received. For example, suppose the client sends the string "inv\r\n". You might receive this in two reads. If the first read() gives you just the bytes "in", you store this in buf, and just proceed around your main loop. If on a subsequent loop iteration select() tells you there's more data from that client, you have to do a read() into buf after the existing data (as tracked by "bytes_in_buf"). If that read() gives you the "v\r\n", then you have a complete line and can pass this to do_something(), which is already implemented and performs the command and removes it from buf, adjusting bytes_in_buf appropriately. The utility function memnewline() will check whether there is an entire line in the buffer, and if so, return a value suitable for passing as the second argument to do_something(). If there isn't an entire line in the buffer, memnewline() returns a null pointer.

(continued)

Make sure that you always follow the network newline convention in transmissions to the client, or (later when you are working on the client) in transmissions from the client to the server. This will allow the programs to continue to work if the client and the server are on platforms with different newline conventions. See

`http://www.cdf.toronto.edu/~ajr/209/notes/sockets/newline.html`

However, both the client and the server will *accept* either \r or \n or \r\n as a newline, in keeping with the general internet engineering principle: "An implementation should be conservative in its sending behaviour, and liberal in its receiving behaviour." (This is already implemented for you in memnewline().)

Step 6: Add a 'say' command

Modify both the client and the server so as to add a *say* command. The user can type a line like (in English):

```
say hi there
```

using lang_say as the command. The rest of the line after the lang_say value and one or more whitespace characters is the text to be said.

The command in the protocol from client to server will be "say *text*" (no matter what the language of the client), and the server will then send "said *n text*" to everyone in the same room. Their clients will then say (if in English) something like:

```
Wilma said: hi there
```

using lang_says_format for the format of the bit up to and including the colon (this string will contain %s at the appropriate point to insert the name, which in some languages is after the verb).

Adding the 'say' command involves five steps, the first of which is already done:

1. Add `lang_say` and `lang_says_format` to all language files (already done)
2. Add a do_say() function in the server which sends the "said" message to everyone in the same room, and add a check for `"say "` in do_something() and call do_say() when applicable (I suggest using strncmp(), not match_arg())
3. Add a check for the *say* command in do_something() in the client (using `lang_say`, but before the call to `explode(buf)`), and send the appropriate command to the server
4. Add a check for the *said* command from the server, in do_something_server() in the client, and output the appropriate feedback to the user
5. Add the `lang_say` command as one of the commands displayed by help() in the client.

I suggest that you test your code in some other language to make sure that you've got the language-independence right.

Other notes

Your programs must be in standard C. They must compile on the CDF machines with "gcc –Wall" with no errors or warning messages, and may not use linux-specific or GNU-specific features.

Submission commands are as in previous assignments and the labs. You need to submit client.c and server.c, and they will be compiled with the original files—do not make modifications to other .c or .h files.

Please see the assignment Q&A web page at

`http://www.cdf.toronto.edu/~ajr/209/a4/qna.html`

for other reminders, and answers to common questions.

And as always, your submission must be your own.

If you want to create other maps or add languages, or modify anything else outside server.c and client.c, be sure to test your assignment code with the original files before submitting. As for other languages, if you stick to UTF-8 then other students will be able to display them without any special terminal program settings, and UTF-8 is designed so that no bytes in the multibyte character sequences

(continued)

are zeroes, so the traditional C str* functions still work. The alternate language files in /u/ajr/209/a4/lang have all been converted to UTF-8.