

Program 2 - adventure

- Due Oct 27, 2016 by 8:59pm
- Points 160
- Submitting a file upload
- Available Oct 11, 2016 at 9:01pm - Oct 29, 2016 at 8:59pm 18 days

This assignment was locked Oct 29, 2016 at 8:59pm.

Program 2 – CS 344

This assignment asks you to write a simple game akin to old text adventure games like Adventure:

http://en.wikipedia.org/wiki/Colossal_Cave_Adventure (Links to an external site.)[Links to an external site.](#)

This program will introduce you to programming in C on UNIX based systems, and will get you familiar with reading and writing files.

Overview

Your program will first create a series of files that hold descriptions of the rooms and how rooms are connected, and then it will offer to the player an interface for playing the game using those generated rooms.

The player will begin in the “starting room” and will win the game automatically upon entering the “ending room”.

Finally, the program will exit and display the path taken by the player.

Specifications

The first thing your program must do is generate 7 different room files, one room per file, in a directory called `<username>.rooms.<process id>`. You get to pick the names for those files, which should be hard-coded into your program. For example, the directory, if I was writing the program, should be hard-coded (except for the process id number) as:

`brewsteb.rooms.19903`

Each room has a Room Name, at least 3 outgoing connections (and at most 6 outgoing connections, where the number of outgoing connections is random) from this room to other rooms, and a room type. The connections from one room to the others should be randomly assigned – i.e. which rooms connect to each other one is random - but note that if room A connects to room B, then room B must have a connection back to room A. Because of these specs, there will always be at least one path through. Note that a room cannot connect to itself.

Each file that stores a room must have exactly this form, where the ... is additional room connections, as randomly generated:

```
ROOM NAME: <room name>
CONNECTION 1: <room name>
...
ROOM TYPE: <room type>
```

Choose a list of ten different Room Names, hard coded into your program, and have your program randomly assign a room name to each room generated. For a given run of your program, 7 of the 10 room names will be used. Note that a room name cannot be used to in more than one room,

The possible room type entries are: START_ROOM, END_ROOM, and MID_ROOM. The assignment of which room gets which type should be random. Naturally, only one room should be assigned as the start room, and only one room should be assigned as the end room.

Here are the contents of files representing three *sample* rooms from a full set of room files. My list of room names includes the following, among others: XYZZY, PLUGH, PLOVER, twisty, Zork, Crowther, and Dungeon.

```
ROOM NAME: XYZZY
CONNECTION 1: PLOVER
CONNECTION 2: Dungeon
CONNECTION 3: twisty
ROOM TYPE: START_ROOM
```

```
ROOM NAME: twisty
CONNECTION 1: PLOVER
CONNECTION 2: XYZZY
CONNECTION 3: Dungeon
CONNECTION 4: PLUGH
ROOM TYPE: MID_ROOM
```

... (Other rooms) ...

```
ROOM NAME: Dungeon
CONNECTION 1: twisty
CONNECTION 2: PLOVER
CONNECTION 3: XYZZY
CONNECTION 4: PLUGH
CONNECTION 5: Crowther
CONNECTION 6: Zork
ROOM TYPE: END_ROOM
```

The ordering of the connections from a room to the other rooms, in the file, does not matter. Note that the randomization you do here to define the layout is not all that important: just make sure the connections between rooms, the room names themselves and which room is which type, is somewhat different each time, however you want to do that. We're not evaluating your randomization procedure!

Now let's describe what should be presented to the player. Upon being executed, after the rooms are generated, the game should present an interface to the player. Note that the room data must be read back into the program from the files, for use by the game. You can either do all of this reading immediately after writing them, or read each file in as needed in the course of the game.

This interface should list where the player current is, and list the possible connections that can be followed. It should also then have a prompt. Here is the form that must be used:

```
CURRENT LOCATION: XYZZY
POSSIBLE CONNECTIONS: PLOVER, Dungeon, twisty.
WHERE TO? >
```

The cursor should be placed just after the > sign. Note the punctuation used: colons on the first two lines, commas on the second line, and the period on the second line. All are required.

When the user types in the exact name of a connection to another room (Dungeon, for example), and then hits return, your program should write a new line, and then continue running as before. For example, if I typed twisty above, here is what the output should look like:

```
CURRENT LOCATION: XYZZY
POSSIBLE CONNECTIONS: PLOVER, Dungeon, twisty.
WHERE TO? >twisty

CURRENT LOCATION: twisty
POSSIBLE CONNECTIONS: PLOVER, XYZZY, Dungeon, PLUGH.
WHERE TO? >
```

If the user types anything but a valid room name from this location (case matters!), the program should return an error line that says "HUH? I DON'T UNDERSTAND THAT ROOM. TRY AGAIN.", and repeat the current location and prompt, as follows:

```
CURRENT LOCATION: XYZZY
POSSIBLE CONNECTIONS: PLOVER, Dungeon, twisty.
WHERE TO? >Twisty

HUH? I DON'T UNDERSTAND THAT ROOM. TRY AGAIN.

CURRENT LOCATION: XYZZY
POSSIBLE CONNECTIONS: PLOVER, Dungeon, twisty.
WHERE TO? >
```

Trying to go to an incorrect location does not increment the path history or the step count. Once the user has reached the End Room, the program should indicate that it has been reached. It should also print out the path the user has taken to get there, the number of steps, and a

congratulatory message. Here is a complete game example, showing the winning messages and formatting, and the return to the prompt:

```
CURRENT LOCATION: XYZZY
POSSIBLE CONNECTIONS: PLOVER, Dungeon, twisty.
WHERE TO? >Twisty

HUH? I DON'T UNDERSTAND THAT ROOM. TRY AGAIN.

CURRENT LOCATION: XYZZY
POSSIBLE CONNECTIONS: PLOVER, Dungeon, twisty.
WHERE TO? >twisty

CURRENT LOCATION: twisty
POSSIBLE CONNECTIONS: PLOVER, XYZZY, Dungeon, PLUGH.
WHERE TO? >Dungeon

YOU HAVE FOUND THE END ROOM. CONGRATULATIONS!
YOU TOOK 2 STEPS. YOUR PATH TO VICTORY WAS:
twisty
Dungeon
$
```

Note the punctuation used: I expect the same punctuation in your program.

When your program exits, set the error code to 0, and leave the rooms directory in place, so that it can also be examined.

If you need to use temporary files, place them in the directory you create, above. Do not leave any behind once your program is finished. We will not test for early termination of your program, so you don't need to watch for those signals.

Do not use the -C99 standard or flag when compiling - this should be done using raw C.

What to Hand In

What you'll submit is your program, named `<username>.adventure.c`. It will be compiled using this line, with my username on here only as an example (replace it with yours):

```
$ gcc -o brewsteb.adventure brewsteb.adventure.c
```

Hints

You'll need to figure out how to get C to read input from the keyboard, and pause until input is received. I recommend you use the `fgets()` function. You'll also get the chance to become proficient reading and writing files. You may use either the older `open`, `close`, `lseek` method of manipulating files, or the `STDIO` standard input library methods that use `fopen`, `fclose`, and `fseek`.

I HIGHLY recommend that you develop this program directly on the eos-class server. Doing so will prevent you from having problems transferring the program back and forth, which can cause compatibility issues.


If you do see ^M characters all over your files, try this command:

```
$ dos2unix bustedFile
```

Grading

You should be warned that if your program doesn't compile or doesn't generate room files on execution, you may receive a zero for the grade.

144 points are available for meeting all of the listed specifications, while the final 16 points will be based on your style, readability, and commenting. Comment well, often, and verbosely, approximately every four or five lines or so, as appropriate: we want to see that you are telling us WHY you are doing things, in addition to telling us WHAT you are doing.

The TAs will use this exact set of instructions: [Program2 Grading.pdf](#)  to grade your submission.