University of California, Santa Cruz
Board of Studies in Computer Engineering

CSE-13E: Computer Systems and "C" Programming

Lab 8 - Role-Playing Game
12 Points

Introduction

In this lab you are going to program a playable dungeon crawler role playing game (RPG). Files are provided in a directory (that you will copy) that contains information about a series of rooms making up a dungeon. Each file consists of room descriptions, any items contained inside, and how each room is connected to its neighbors. This lab will be programmed on the UNIX servers (as opposed to the embedded hardware), and you will be using the keyboard to read user input and display output on a terminal.

Reading

- **K&R** – Chapter 7.5

Concepts

- File I/O

Required Files:

- Lab08_main.c
- Game.c
- Player.c
- makefile
- README.md

Provided files

- DO NOT EDIT these files:
    - **Player.h** - Describes the functions necessary for the inventory management functionality necessary for the player in this RPG. You will implement the described functions within the corresponding Player.c file.
    - **Game.h** - This library controls the game state and handles inter-room navigation. You will implement the described functions within the corresponding Game.c file.

    - **UNIXBOARD.h** – Contains Standard #defines and system libraries used. Also includes the standard fixed-width datatypes and error return values. This is to replace the BOARD.c/h files that you have been using on the embedded processor.
- EDIT these files (included as *_template*, you must rename them):
    - **Lab08_main.c** - This file contains main(). Any functions you need to create that aren't related to either of the two libraries (Player.c/h and Game.c/h) should be placed in this file.
    - **makefile** – This file contains a basic command to build the rpg. You will need to modify it to build the intermediate .o files and a clean command. This file does not have an extension, do not add one.
- CREATE these files:
    - Player.c
    - Game.c

Assignment requirements

- **Player library:** Implement all of the functions defined in Player.h. This library is used exclusively by the Game.c/h library and will not be called directly from Lab08_main.c.
- **Game library:** Implement all of the functions defined in Game.h.
- **main():** Your `main()` within Lab08_main.c must implement the following functionality using the provided code along with the two libraries you will have implemented. You are also free to use any functions within the C standard library.
    - All Information for the user will be displayed in the terminal.
    - The program should display the room title and description in a reasonable fashion.
    - Below the room data there should be a prompt for the next direction. This prompt should indicate what directions are available for travel.
    - User input is captured using getchar(). This function blocks until it receives a newline. You should respond only to valid directions (n,e,s,w) or q and ignore all other inputs. Only one direction can be taken at a time and longer strings should be ignored as well.
        - Receiving q should exit the program using the `exit()` function with the SUCCESS code.
    - Upon entering a room the terminal should display the new room info.

    - The player has an inventory where they store things that are found in the rooms. These items can alter the room description and the available exits. This should be implemented as well.[1]
- **Code requirements:** When implementing this lab, your code should adhere to the following restrictions.
    - Use `fread()` and `fgetc()` to read the different parts of the file. `fseek()` can be used as well to skip over parts of the file.
    - All constants should be enums or macro constants.
    - A struct is used in Game.c to hold the title, description, and all 4 exits for the current room.
    - The `GameGo*()` functions all open the next room file, process the data and load the room struct with the correct information, and then close the file.
    - No more than one file should ever be open at a time and files should not be used outside of the `GameGo*()` functions.
    - All `GameGet*()` functions should extract the correct data from the room struct; they should NOT reopen/reparse the file.
    - A single event loop should be created within `main()` for processing all events (getchar).
    - If `GameInit()` fails when called within `main()`, the `FATAL_ERROR()` macro should be called.
    - Inventory management must be done through the Player library.
    - All code that utilizes macro constants should work if the macro constant value were to be changed.
    - All external variables must be limited to module-scope by using the static keyword. NO GLOBALS!
    - All calls to `fopen()` should be checked for failure, with `FATAL_ERROR()` called if they do.
    - No heap use in this lab (i.e.: no calls to `malloc()`).
    - A properly made makefile which supports both generating the executable and cleaning up both the object files and the executable. This makefile cannot have macros to illustrate that the proper targets have been made correctly.
- **Code style:** Follow the standard style formatting procedures for syntax, variable names, and comments.
    - Add the following to the top of every file you submit as comments:
        - Your name
        - The names of colleagues who you have collaborated with

[1] That is, you might get a different room description depending on what is in your inventory list. This will need to be checked so that you print the correct description.

- **Readme:** Create a file named README.md containing the following items. Note that spelling and grammar count as part of your grade so you'll want to proof-read this before submitting. This will follow the same rough outline as a lab report for a regular science class. It should be on the order of three to four paragraphs with several sentences in each paragraph.
    - First you should list your name and the names of anyone else who you have collaborated with.[2]
    - In the next section you should provide a summary of the lab in your own words. Highlight what you thought were the important aspects of the lab. If these differ from how the lab manual presents things, make a note of that.
    - The following section should describe your approach to the lab. What was your general approach to the lab? Did you read the manual first or what were your first steps? What went wrong as you worked through it? What worked well? How would you approach this lab differently if you were to do it again? Did you work with anyone else in the class? How did you work with them and what did you find helpful/unhelpful?
    - The final section should describe the results of you implementing the lab. How did it end up finally? How many hours did you end up spending on it? What'd you like about it? What did you dislike? Was this a worthwhile lab? Do you have any suggestions for altering it to make it better? What were the hardest parts of it? Did the point distribution for the grading seem appropriate? Did the lab manual cover the material in enough detail to start you off? Did examples or discussions during class help your understanding of this lab or would more teaching on the concepts in this lab help?
- **Submission:**
    - Submit your files (Player.c, Game.c, Lab08_main.c, README.md, and makefile).

Grading

This assignment consists of 12 points:

- 4 points - Game library
- 2 points - Player library
- 2 points - RPG functionality
- 2 points – Program requirements
- 1 point – README (you lose this point if you fail to submit README.md)
- 1 point – Style

You will lose points for the following:

[2] NOTE: collaboration != copying. If you worked with someone else, be DETAILED in your description of what you did together. If you get flagged by the code checker, this is what will save you.

- No credit for any sections where libraries/code doesn't compile.
- -2 points: Any warnings when compiling all code together
- -4 points: For file access outside the `GameGo*()` functions.

Room description files

This lab relies extensively on reading files from the file system. These files contain all of the information for the dungeon of the RPG. These files are all organized in a subdirectory of the system (specified in Linux as "RoomFiles/"). Their file names are room1.txt through room65.txt. When accessing these files with the file functions, their full filename will be required (e.g.: "RoomFiles/room1.txt"). You will need to copy the Roomfiles directory given to your local directory where you will run your code. DO NOT use a switch statement to individually select the proper room name; use a function like `sprintf()` to generate the necessary filename.

The files should be opened in binary mode and each file starts with the asci characters 'RPG'. After the header, the file is laid out to have a title first, then a repeating list of room properties; these properties are dependent on the player's inventories for whether or not they should be displayed. Note that these values are always repeated the same way in order.

RPG	Title	Item requirements	Description	Items contained	Exits
		(repeated)			

The first section of the file is the title. This is stored as a byte holding the length of the string and then that many ASCII characters (each is one byte). There is no NULL-character terminating this string![3]

Title	Item requirements	Description	Items contained	Exits
	(repeated)			

Following the title is a repeating group of values representing the portion of the room that can vary: item requirements, description, items contained, and the exits. The first of these, the item requirements, is a sequence of bytes containing the number of the item that is required for this version of the room to be displayed. This list of bytes is prepended by the length of this list. So if the room has a single item, item #3, that is required, the file would have a byte of value 1 followed by a byte of value 3 for this section. If multiple

[3] Be careful with this. If you want to treat the title as a string (e.g. for use in printf) you will need to store the characters into a character array and then append a '\0' to the end.

items are listed here, all of the items must be present in the player's inventory for this room version to be displayed. If not items are required, then the prepended list length is 0.

Title	Item requirements	Description	Items contained	Exits
	(repeated)			

The description follows a similar layout to the Title in that it is a sequence of ASCII characters that are not NULL-terminated but are prepended with the length of the string. Again, similar care needs to be taken when using these as strings.

Title	Item requirements	Description	Items contained	Exits
	(repeated)			

Following are the items that are contained within this room. This follows the same format as the item requirements: 1 byte indicating length as a uint8 followed by that many bytes representing item numbers of the items found within the room.

Title	Item requirements	Description	Items contained	Exits
	(repeated)			

Next four bytes represent the room numbers of the available exits for this room version. They are ordered as north, east, south, west exits (clockwise from north). A 0 indicates that there is no exit or any connected room in that direction.

At this point it is either the end of the file (EOF) or another set of item requirements meaning that there is another room version available (if you have the right items). The versions are always specified in order of decreasing item requirements. There should also always be a final room version with 0 item requirements which is the version that will be displayed if no other versions passed their item requirements check.

As a simple example, “/room1.txt” looks like the following:

1. **Title**, 16 characters, “Council chambers”
2. **Version 1**
    a. Item requirement: None (0x00)
    b. Description: 214 characters long
    c. Items contained: None (0x00)

d. Exits: East is room 2 (0x00, 0x02, 0x00, 0x00)

A more complicated example is “/room32.txt”, which looks like the following:

1. **Title**, 15 characters, “The throne room”
2. **Version 1**
    a. Item requirement: 3 (0x01, 0x03)
    b. Description: 169 characters long
    c. Items contained: None (0x00)
    d. Exits: South is room 30 (0x00, 0x00, 0x1E, 0x00)
3. **Version 2**
    a. Item requirements: None (0x00)
    b. Description: 238 characters long
    c. Items contained: 3 (0x01, 0x03)
    d. Exits: South is room 30 (0x00, 0x00, 0x1E, 0x00)

File API

Read the standard library documentation for all of the file functions that you’d like to use (look up stdio.h). You should know and handle any and all failure cases. Also read all of the documentation for the functions, as some have additional requirements that need to be met before they are useable.

Makefiles

Throughout this class you have been using MPLABX IDE to both write and compile your code. While IDE’s exist for generic C code, for this lab you will compile the code directly. The standard call to gcc (our UNIX compiler) is shown below:

```
gcc Simple.c –o Simple
```

This will compile the source given into an executable with name “Simple”. If you do not give the –o option it will generate an executable of name “a”. A more complex program that involves multiple source files can call gcc with multiple files like the example below.

```
gcc Game.c Player.c Lab08_main.c
```

While this is possible to do this for any project it is unwieldly and forces all of your files to be compiled for every change. For large projects a full compile can take hours to perform. Instead of compiling every file, gcc supports the generation of object files. These object files contain compiled C code for the file but not the library code. For example, the object file for Lab08_main.c will contain references to GameGetCurrentRoomExits() but not its definition. At the same time changes to Lab08_main.c will not require recompilation of Game.c. Object files are generated with the command below

```
gcc -c Player.c
```

A set of object files can then be combined to form an executable

```
gcc Player.o Game.o Lab08_main.o -o rpg
```

While these commands are all runnable within the terminal directly, combining them within a makefile allows them to be consolidated and easily reused. When you hit make inside MPLABX it is calling its own makefile to generate the hex file.

Makefiles are text files containing commands that are parsed by the make command. To use the makefile you simply call "make" on the terminal. A Sample makefile is shown below.

```
All: Simple
Simple.o: Simple.c
            gcc –c Simple.c
Simple: Simple.o
            gcc Simple.o -o Simple
```

The first line, "`all: Simple`" consists of two parts: target, a desired item to create; and a dependency, the requirements to make the desired target. This is also a special case as calling "`make`" is the same as calling "`make all`". The other targets follow the same format but also have commands associated with them to generate that specific target.

When make is called, it will follow the dependency chain as needed to build the requested target. Additionally it will only perform the action if the dependencies are newer than the target. For example, this means that it will only compile new .o files as needed, not every time.

As make can also call targets directly, auxiliary commands can be made such as the "`make clean`" command below.

```
clean:
            rm *.o Simple
```