Your task for this assignment is to design, code (in C89 version of C) and test a program. In summary, your program will:

• Able to create dynamically-allocated 2D char array to make a simple ASCII-based game.

• Receive user input to control the game.

• Utilize pre-written random number generator and terminal behaviour configuration for the game.

• Write a proper makefile. Without the makefile, we will assume it cannot be compiled and it will negatively affect your mark.

• Extract information from a text file to initialize the game.

• Utilise linkedlist data structure to keep track the game progress, allowing the player to undo the steps taken.

**1 Code Design**

You must comment your code sufficiently in a way that we understand the overall design of your program. Remember that you cannot use “//” to comment since it is C99-specific syntax. You can only use “/\*………\*/” syntax. Your code should be structured in a way that each file has a clear scope and goal. For example, “main.c” should only contain a main function, “game.c” should contain functions that handle and control the game features, and so on. Basically, functions should be located on reasonably appropriate files and you will link them when you write the makefile. DO NOT put everything together into one single source code file. Make sure you use the header files and header guard correctly. NEVER #include .c files directly, instead #include .h files only. Make sure you free all the memories allocated by the malloc() function. Use valgrind to detect any memory leak and fix them. Memory leaks might not break your program, but penalty will still be applied if there is any. If you do not use malloc at all, you will lose marks.

**3.1 Quick Preview**

You will implement a simple game that you can play on Linux terminal. Further details on the specification will be explained on the oncoming sections.

**3.2 Command Line Arguments**

Please ensure that the executable is called “treasure”. Your executable should accept only one command-line parameter/argument:

./treasure <map\_file>

<map/\_file> is the text file name containing the information to initiate your game. It is your responsibility to check whether the file can be opened properly. You can assume the content of the file is valid to initiate the game. If the amount of the arguments are too many or too few, your program should print a message on the terminal and end the program accordingly (do NOT use exit() function).

**3.3 Main Interface**

Once you run the program, it should clear the terminal screen (watch the video about the method to clear the screen) and print the 2D map along with the Walls, Player, Treasure, Lantern, and Snake:

A screen shot of a black screen

Description automatically generated

You can type the command to move the Player. The Snake will move one grid randomly on 8 directions for every Player movement (See Section 3.6.3). Every time the user inputs the command, the program should update the map accordingly and refresh the screen (clear the screen and reprint the map). Please refer to Section 3.6.1 for more detail on the user input.

**3.4 Character on the Map**

You have to use the correct characters on the game interface:

• Use char ’\*’ for the map border.

• Use char ’O’ for the Wall.

• Use char ’@’ for the Lantern.

• Use char ’P’ for the Player.

• Use char ’~’ for the Snake.

• Use char ’$’ for the Treasure.

**3.5 File I/O**

The text file will look like this:

A screenshot of a black and white screen

Description automatically generated

The two integers on the first line are the playable row and column map size respectively. These map sizes do not include the border surrounding the map. The integer on second line indicates the amount of the Walls in the game. Feel free to use this information to allocate sufficient memory to store Wall location. Afterwards, the remaining lines are the content of the 2D map array. You can assume the text file contains valid information with valid datatype. Each integer represents:

• 0 represents ’ ’ (An empty space)

• 1 represents ’O’ (The Wall)

• 2 represents ’@’ (The Lantern)

• 3 represents ’P’ (The Player)

• 4 represents ’~’ (The Snake)

• 5 represents ’$’ (The Treasure)

The integers on first line will match the amount of rows and columns of the map. Similiarly, the integer on second line will match the correct amount of Walls on the map. Your program should be able to read it correctly to initialize the game interface.

**3.6 Gameplay**

The main gameplay is to move the Player to reach the Treasure while avoiding being eaten by the Snake.

**3.6.1 User Input**

The user only needs to type 1 character for each command (lower case). Here is the list of the possible commands: • ‘w’ moves the player one block above. • ‘s’ moves the player one block below. • ‘a’ moves the player one block left. • ‘d’ moves the player one block right. • ‘u’ undoes the movement of the Player and Snake. This includes returning the Lantern’s original position. • Any other character should be ignored, and nothing changes on the map. Feel free to print a warning message such as ”invalid key”. Your program should be able to receive each input without pressing ”enter” key everytime. Please refer to the sample code to disable the Echo and Canonical feature on Linux terminal temporarily. If your terminal is stuck on this mode, you only need to re-open a new terminal.

**3.6.2 Winning/Losing the Game**

The game will continue playing until either the Player manages to reach the Treasure (winning) OR the Snake manages to eat the Player (losing). You can print message such as ”You win!” or ”You lose!” when the game ended. Please do NOT use exit() or break to end the program. You must free all the malloc’ed memories and ensure the program reaches the end of the main function properly.

**3.6.3 Snake Movements**

Whenever the Player makes a valid movement, the Snake will also move for one grid on the map. However, the Snake’s movement is randomized. You have to use the random number generator function for the Snake to decide 1 out of 8 possible directions. If the Snake is at the edge of the map OR near the Walls, then some movements will be restricted. (Please be careful here. Moving the Snake outside the map might crash your program due to unauthorized out of bound memory access)

A black and white image of a snake

Description automatically generated

However, if the Player moves to a grid on which it is exactly ONE grid away from the Snake, then the Snake will just move to eat the Player without any random movement.

A screenshot of a video game

Description automatically generated

**3.7 DARK mode with Makefile Conditional Compilation**

You need to implement an additional feature that allows for higher game difficulty. This DARK mode is activated via Conditional Compilation with Makefile (e.g when you execute ”make DARK=1”). When the keyword DARK is defined when compiling, the Player will have a limited visibility range.

By default, the Player can only see as far as 3 blocks distance away from the Player’s position. Any object that is not within the visibility range of the Player will be invisible (print out empty space ’ ’ instead). If there is an empty space (no object) within the Player’s visibility range, print out char ’.’ instead. (This will make it easier to see the Player’s visibility boundary)

You will use Manhattan Distance for the Player’s visibility range (also known as Taxicab Distance OR City Block Distance). Manhattan Distance refers to a method to measure the distance between two points (within 2D grid-based array in this context). Instead of drawing a straight line between both points and measure it, Manhattan Distance sums the absolute distance of both points in vertical direction and horizontal direction. For example, if we have the Player located at row 1 and column 5, and the Treasure is located at row 7 and column 2, then:

Manhattan Distance = |Player row − Treasure row| + |Player col − Treasure col|

= |1 − 7| + |5 − 2|

= 6 + 3

= 9(blocks distance)

When the Player reaches the Lantern ’@’, the Lantern will disappear and the vision distance will be increased to 6 blocks. This effect will last until the game ends.

A screenshot of a computer game

Description automatically generated

**3.8 Struct and Generic Linked List to implement UNDO feature**

In this section, you will need to write a generic linked list with appropriate struct to implement UNDO feature. This feature is triggered when the user enters the key ’u’. Everytime the key is pressed, the player and the snake will move back to their previous position. The player can use the undo feature at any point of the game as long as the game is not over. It should be possible to keep undo-ing up to the initial state of the game.

You can use the linkedlist node to store any information that you think are useful. For example, you can store the previous player ’P’ location and the snake ’~’ location (This is only one example, you can store more information if you need it). You will need to create a new linkedlist node every time the player moves (with malloc()) and store it in the linkedlist. When the player presses ’u’, your program should be able to retrieve the most recent node and update the map to the previous state. If game state is back to the initial configuration (when you just started the game), pressing ’u’ should NOT do anything. One possible way to confirm this is by checking if the linkedlist is empty. (Once again, this depends on your implementation. Implement what you need to achieve this task.)

Note: Remember, if you UNDO the game back until the point the Player reaches the Lantern, then the Lantern will be put back on the ground, and the vision distance is reduced back to 3 again.

**3.9 Makefile**

You should manually write the makefile. It should include the Make variables, appropriate flags, appropriate targets, correct prerequisites, Conditional Compilation, and clean rule. We will compile your program through the makefile.

**3.10 Assumptions**

For simplification purpose, these are assumptions you can use for this project:

• The content of the text files are valid. (If there is NO error when opening it)

• The size of the map in the text file is at least 10 rows AND at least 10 columns.

• All game objects exist. There will be ONLY one Player, ONLY one Snake, ONLY one Treasure, and ONLY one Lantern. The Wall is AT LEAST one.

• There will be a winning path from the Player to the Treasure. Similiarly, there is also a path from the Snake to the Player. It means each map will always have a possibility of winning and losing.

• You can still win the game even without grabbing the Lantern. Lantern just makes it easier when DARK mode is activated.

• Snake cannot step on the same grid as the Treasure and Lantern.

• Both Player and Snake cannot step inside the Wall.

• The size of the map will be reasonable to be displayed on terminal. For example do not test the map with gigantic size such as 300 x 500.

• You only need to handle lowercase inputs (’w’, ’s’, ’a’, ’d’, ’u’). The other keys should be ignored (feel free to add warning message).

Note: When you create your own custom maps for testing, please make sure to follow these assumptions

**EXTRA NOTES:**

• Properly structured makefile

• Program can be compiled with the makefile and executed successfully without immediate crashing and showing reasonable output

• Usage of header guards and reasonable in-code commenting

• The whole program is readable and has reasonable framework. Multiple files are utilized with reasonable category

• Avoiding bad coding standard. Some of the most common mistakes are:

– Using global variables

– Calling exit() function to end the program abruptly

– Using “break’‘ NOT on the switch case statement

– Using “continue’‘

– Using goto – Having multiple returns on a single function

– #include the .c files directly instead of the .h files

• No memory: Please use valgrind command to check for any memory leak.

**FUNCTIONALITIES:**

– Correct command line arguments verification with proper response. This includes detecting the success/failure of opening the text file

– Proper memory allocation for the 2D map array.

– Able to clear the screen and re-print the map on every action.

– Able to move the Player with the keyboard input from the user without pressing ’Enter’ key. (and cannot go through Wall)

– The Snake moves randomly 1 out of 8 directions after every Player movement correctly. (and not hitting Border and Walls)

– When the Player moves too close to the Snake, the Snake immediately eats the player

– Winning when the Player reaches the Treasure AND Losing when the Snake eats the Player.

– File IO is done successfully to retrieve all game information from the map text file.

– Able to UNDO the game utilizing linkedlist

– DARK mode is implemented correctly in makefile and the game. (including lantern usage to increase visibility range)