Assignment #5

Question #1 - Zig-Zag lists

Assignment

Write a function that receives two list heads (`h1`, `h2`) on success the function returns a pointer to a zig-zag list where the zig-zag list is constructed by taking an element from each list by turn.

If one list is bigger append the remaining elements to the end, if one of the input pointers is invalid return NULL.

Notice you are allowed to destroy the input lists `h1` and `h2` in the process.

> When submitting remove the main function if exists.

prototype

```
node_t *zig_zag_lists(node_t *h1, node_t *h2);
```

where `node_t` is defines as,

```
typedef struct node_rec {
    int data;
    struct node_rec *next;
} node_t;
```

Validations

1. All pointers are valid.

Test cases

Uploaded with the assignment.

Question #2 - Replace

Assignment

Write a function that receives an input file path, an output path and two strings (`s1`, `s2`) and replaces all occurrences of `s1` in the input file with `s2`. On success the function returns the number of replaces made o.w -1.

You are required to also replace a substring inside a word, i.e. for `s1 = "foo"` you need to find and replace only occurrences of `foo` and `barfoobar`.

Assumptions

1. The maximum length of a line will be 256 characters.
2. `s1` and `s2` will be bounded to one line and will not stretch multiple lines.

> When submitting remove the main function if exists.

prototype

```
int replace(const char *input_file, const char *output_file, const char *s1, const char *s2);
```

Validations

1. All pointers are valid.
2. input file exists.
3. output file is writable.

Test cases

Uploaded with the assignment.

Requirements

1. The input file may be very big, hence you are not allowed to store the entire input file inside a preallocated buffer.

Question #3 - Maze Runner

In this question you will implement your own game! You are given a basic framework and you are required to improve the game. The final game will include a maze contained in a file, the maze will contain special tiles, walls and portals.

At any point of failure print a relevant message, exit the game and free allocated resources.

Part of the task is to understand the existing code, so no explanations will be given besides the comments that are in the files.

> There is no need to change the files `terminal.c` and `Makefile`, only `maze.h` and `maze.c`.

Guide

Work according to the following guide, make sure you save a copy of your code after completing each of the following steps in order to revert to a working game in case of any mistake.

> Do not attempt more than one step at a time -- after each step you should have a working program.

1. Compile the game by running `make` in the terminal. If it is not installed, install it using `sudo apt install make`.
2. Run the game in the terminal (`./maze`). Play a bit to understand what the player can and cannot do. Find an existing bug relating to walls. Try pressing wrong keys. Watch the frame number (top right) to see which keys generate multiple chars.
3. Read the given files and try to understand the basic flow. Questions to ask yourself: What functions are there? What do they do? What constants are there? What data is stored? What is the main loop? How does the player interact with the game?
4. Try to make small modifications to familiarize yourself with the code. Change the color of the walls. Change the player symbol from `@` to `*`. Change the maze size. Change the controls from `A/S/D/W` to `Z/X/C/S` (don't try arrow keys, they are multi-char). Now go back to the original version.
5. Make the game display `"Goodbye!"` when quitting, using the messaging system.
6. Refactor the code of `print_maze` by locating a pair of code lines repeated twice and moving them to a separate function.
7. Fix the wall bug and display the message `"Ouch!"` when hitting a wall.
8. Refactor the code by defining `struct maze` which contains all game state (currently the state consists only of player location and the message to display). Do not use global variables, use a local variable defined in `main`. This requires changes to multiple functions, receiving `struct maze *` as an argument.

> Do not add any new functionality at this point!

9. Refactor the code by making the maze size part of `struct maze` instead of the defines that are currently defined. Set the size when initializing the `struct maze` local variable in `main`.

10. Refactor the code by splitting handling of user input to a separate function called `handle_input` (like the way `print_maze` handles display). The return value should be assigned to the local variable `game_over`. Now `main` should be short and self-explanatory.

11. Add a target (a location the player will try to reach) location to `struct maze` (two more `unsigned` fields). Update `print_maze` accordingly. Use `$` as the target symbol when displaying (add a `#define` like `S_WALL` and `S_PLAYER`). Choose your favorite color for it. Default target location is bottom right corner.

12. Update `handle_input` to check whether player reached target after moving. If so, display the message `"You win!"` using the messaging system. Add a `#define` like `MSG_START`. Also, the game should end.

13. Now that we have two locations (player and target). It makes sense to have `struct location` instead of using two separate `unsigned` variables for each location. Define the `struct` in `maze.h` and refactor the code to use it.

14. Instead of having exactly one message for each event, add support for multiple messages (see example below). When displaying a message for an event, choose randomly one of the variants. For this, define global const string arrays in `maze.c`, e.g.,
    `const char *msg_quit[] = {“Bye!”, “Farewell!”, “Goodbye!”};` and `#define MSG_QUIT` in `maze.h` as `msg_quit[rand() % 3]`. Of course, don’t put the constant 3 there, but do something that doesn’t need to be changed when more messages are added. Do this for all messages.

15. Add a 2D dynamic array to `struct maze`, which will hold the maze tiles. Write a function that allocates and initializes a `struct maze` and a function that deletes it (and frees memory). You can use one of the methods shown in the lecture.
    Note: outer walls are not part of `struct maze` nor calculated for the size!

16. Update `print_maze`. To test, initialize the maze "by-hand" to something small and simple (don’t forget to set the size accordingly).
    Example: (4 rows, 11 columns)

```
# # # #  #
#   # # #
 #   # ##
 ####   #
```

17. Is the wall bug back? If so, fix it. Both outer and inner walls cannot be entered.

18. Add support for the player standing on top of a non-floor tile. The symbol of the tile should be displayed, but the color should be the color of the player. This requires changes to `print_maze`; it makes sense to write a `print_tile` function that gets a symbol and a color separately. Fix the `#defines` of the tile accordingly.

19. Add one new tile type to the maze. This requires changing `handle_input` and `print_maze`. Don’t forget colors. Test your code by hard-coding maze from step 16 and add the new tile type.
    Easy tiles (**choose one**):

1. Vertical (`|`) and horizontal (`-`) walls. A vertical wall tile cannot be entered from left/right, only from top/bottom. The other way around for horizontal walls.

2. One sided floor. The player can only exit the tile from one side. You need 4 types, one for each direction, and so you need 4 symbols: `v/^/</>`.

3. Trap and Secret wall. A trap tile looks like a floor tile, but when the player enters it, the game ends. Display the message `"Boom! You lose..."` using the messaging system. A secret wall tile looks like a regular wall tile, but the player can enter it. Display the message `"This wall sounds hollow..."` using the messaging system.

4. Different height floors. There are 3 types of floor tiles: `' '` (space), `.` (dot), and `:` (colon). It is possible to move from low (space) to middle (dot) and from middle to high (colon) but not possible to move directly between low and high floors. Display the message `"Too high!"` using the messaging system.

5. Your own idea, cool ideas may get you bonus points. 😃

20. Write a function `read_maze` that reads a maze from a text file. The argument of `read_maze` is the filename, and the return value is `struct maze *` (dynamically allocated). If the file is not readable, or the format is not correct (including tile types your code doesn't support), print a message and `exit` the program. The file will contain the name of the maze, its dimensions and the maze you can view the attached example `maze.txt`. The "Free text area" inside the file should be printed using the messaging system.

21. Instead of a hard-coded maze, use the `read_maze` function in your main to read from the file `maze.txt`. Don't forget basic checks (exactly one `@`, exactly one `$`).

22. Implement more tiles from 19 and this step. Have at least two easy and at least two challenging tile types. Don't try to add all at once, do them one by one. When your functions grow, consider refactoring (e.g., the switch in `handle_input` can be kept to a reasonable length by using subroutines). Remember to check as in step 21 the new tile's restrictions.
    Challenging tile (note some tile types also require changes to `struct maze`):

    1. Multi-targets. The maze can contain more than one target tile, and to win the player needs to visit all of them (order is not important). To track which targets were already visited, use two separate tile types – visited target and unvisited target (same symbol, different color).

    2. Keys (`&`) and doors (`+`). When the player enters a key tile, she picks it up. Player without keys cannot enter a door tile (display a message). When a player with a key enters a door tile, the door disappears (replaced by floor) and one key is used (display a message). You need to track the number of available keys in the `struct maze` and display it all the time (change `print_maze` and put number of keys next to the frame number).

    3. Portals (`0`-`9`). There should be two tiles with the digit `d` in the maze. Whenever the player enters one of them, her location "magically" changes to the other. This requires modifications `struct maze` (e.g., an array of portals, where each portal consists of two locations). Display a message when using the portal like `"Wheee!"`.

    4. Multi-Portals. It is possible to have `k≥2` tiles with the digit `d` in the maze. Entering one sends the player to the next tile (and from the last one back to the first). Portal locations should be saved as a linked list.

23. Design at least 3 interesting mazes (size of at least 10 rows and 50 columns) and put each in a separate file according to the format of `maze.txt` given. Demonstrate the tile types you chose to implement. Use them to test your code from step 22. They should be submitted with your code.

24. Change main to accept command line arguments. There should be only one argument, which is a filename. Use it to load the maze instead of a fixed filename.

Note

The purpose of this question is to expose you to working on a big project with a given code base, to summarize the skills you acquired during the course and to for you to enjoy the game you create.

Bonus points will be given for creative tiles, cool mazes, funny messages or any other unique game feature.

Grading

Note that the grading for this question will be more focused on white-box evaluation and my general impression of your game.