COMP1000 Assignment 2 Report

Mohammed-Ali Hussein 20184133

2021 Semester 2

Introduction

This report is written to explain the rational, functioning and possible problems arising in relation to the functioning of assignment two.

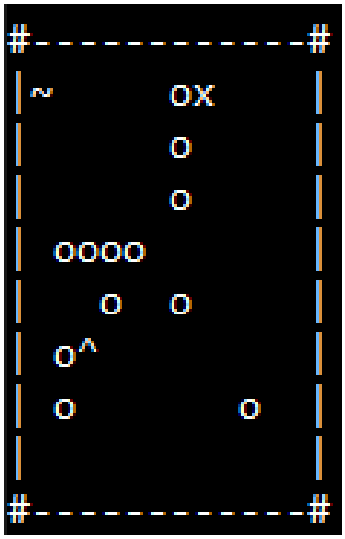
(Mentioned at the end are possible points of failure or unsuccessful operation of the program if the marker of this has any.)

Functioning

At the current moment, the program functions correctly with no memory leaks or suppressed errors with the provided sample input file. The following images will show the sequence of the program starting from the compilation of the program through '**make**' to execution of the program via the valgrind command with the provided sample map to display the memory and error data after the program finishes running. Start command: **'valgrind ./maze map.txt'**

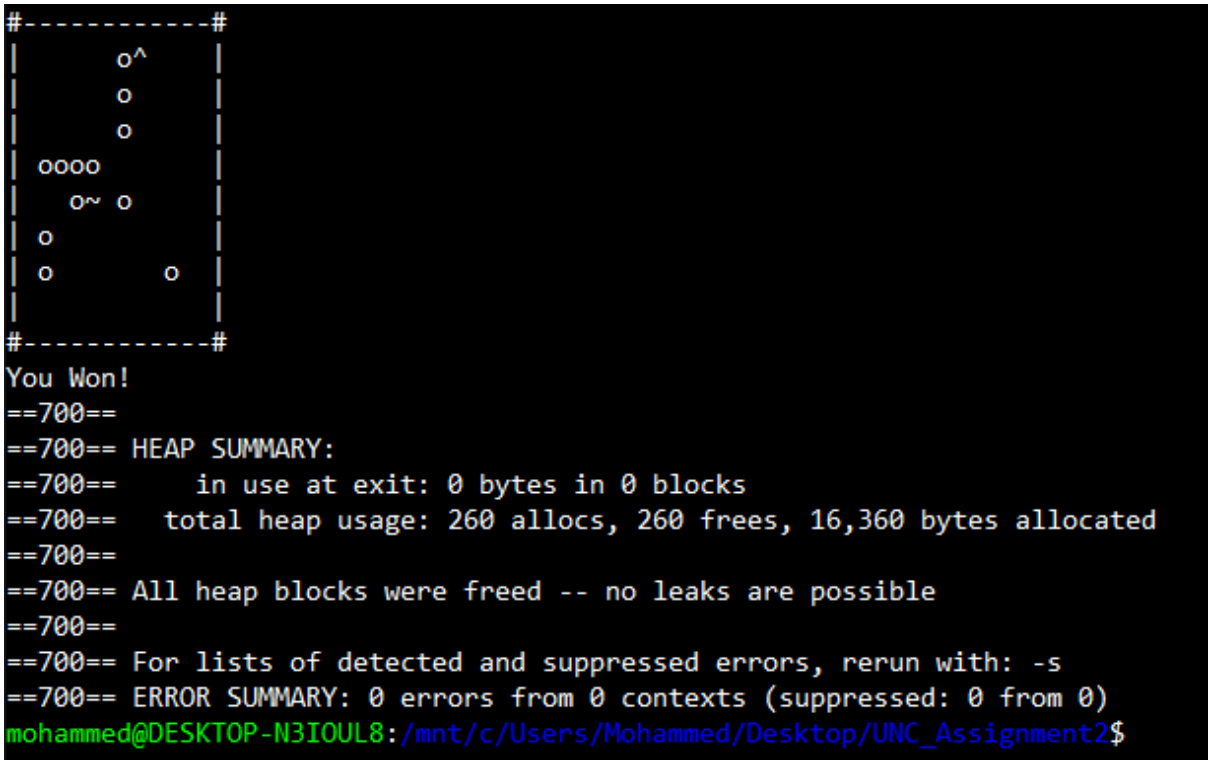
```
mohammed@DESKTOP-N3IOUL8:/mnt/c/Users/Mohammed/Desktop/UNC_Assignment2$ make
gcc -c main.c -Wall -pedantic -ansi
gcc -c terminal.c -Wall -pedantic -ansi
gcc -c mazeSetup.c -Wall -pedantic -ansi
gcc -c updateMaze.c -Wall -pedantic -ansi
gcc -c freeMemory.c -Wall -pedantic -ansi
gcc -c renderMaze.c -Wall -pedantic -ansi
gcc -c fileIO.c -Wall -pedantic -ansi
gcc -c snakeMovement.c -Wall -pedantic -ansi
gcc -c LinkedList.c -Wall -pedantic -ansi
gcc main.o terminal.o mazeSetup.o updateMaze.o freeMemory.o renderMaze.o fileIO.o snakeMovement.o LinkedList.o -o maze
mohammed@DESKTOP-N3IOUL8:/mnt/c/Users/Mohammed/Desktop/UNC_Assignment2$
```

After the program starts initially, this is all that will be displayed:



(See next page for final display)

Once the player moves to the goal or is ‘killed’ by the snake, the following display will be shown via valgrind:



As seen in the above image, “All heap blocks were freed – no leaks are possible” and “ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)”. As mentioned earlier the program successfully compiles, runs and does not result in memory leaks or error messages. This case was produced using the sample file map.txt on blackboard.

Newly Added Assignment 2 Functionality

For assignment 2, the newly added functionalities of the program include file reading of metadata, implementation of a snake entity moving towards the player and the undo feature through the linked list. Each of these features will be discussed in the coming sections.

File Reading (fileIO.c)

Newly added to the program is the **fileIO.c** file. This handles reading in the metadata and providing it to the main function in sorted order. The order is determined by the object codes going in ascending order. In other words, the player’s metadata first, the snake’s metadata second, the goal’s metadata third and the obstacles from fourth to the end of the metadata. The file is read using the file pointer but more importantly, read using **fscanf** with an expected data format of (“%d %d %d”). The data is initially read into an **int**** pointer.

```
/*now the metadata needs to be sorted by the object code i.e. 0, then 1, 2 ...*/
metadata = sortObjectsByCode(metadata, metaDataAmount, mapRows, mapCols);
```

This data is then sent to the function **sortObjectsByCode** to get a consistent file order regardless of the file so that the program always knows where the player, snake and goal metadata is in the table. The sorting functions

will create dynamically allocate a new array and look for the player, snake and goal entities in the original data. Each entity will go to a predefined spot in the new array.

```
/*find the player, snake and goal*/
for(i = 0; i < metadataAmount; i++)
{
    /*the player*/
    if(metadata[i][2] == 0)
    {
        addToSorted(sortedMetadata, metadata, 0, i);
    }

    /*the snake*/
    if(metadata[i][2] == 1)
    {
        addToSorted(sortedMetadata, metadata, 1, i);
    }

    /*the goal*/
    if(metadata[i][2] == 2)
    {
        addToSorted(sortedMetadata, metadata, 2, i);
    }
}

/*now fill in the metadataAmount - 3 spots with the obstacles*/
j = 3;
for(i = 0; i < metadataAmount; i++)
{
    if(metadata[i][2] == 3)
    {
        addToSorted(sortedMetadata, metadata, j, i);
        j++;
    }
}
```

The original metadata is freed after this is complete and the new data is assigned to pointers whose origins lie in the main function.

```
/*assigning the main function's data to the newly read data*/
*metadataTable = metadata;
*metadataAmnt = metaDataAmount;
*mapRow = mapRows;
*mapCol = mapCols;
```

This data is called for via the **getMetadata** function shown here:

```
int getMetadata(int *** metadataTable, int * metadataAmount, int * mapRow, int * mapCol, char* filename)
{
    return readFile(metadataTable, metadataAmount, mapRow, mapCol, filename);
}
```

The return value is an integer indicating if there was a problem reading the file so that the main function does not call the starting functions and provide the game with no metadata.

Snake Movement Feature (snakeMovement.c)

The snake's movement is defined by a simple algorithm which follows the following rules after each player input:

- If the snake is **above** the player, the **snake will move down**
- If the snake is **below** the player, the **snake will move up**
- If the snake is **in line with the player horizontally** the following are the rules:
  - If the snake is in the same row but to the **left** of the player, the **snake will move right**
  - If the snake is in the same row but to the **right** of the player, the **snake will move left**

Each time these movements finish and the snakes co-ordinates are updated, the **checkEndGame** function checks to see if the player and snake co-ordinates are equal meaning the player has 'died' and the game will exit the **gameLoop** function.

```
void checEndGame(int** mazeData, char** maze)
{
    /*game ends/ exits loop by player character becoming the snake
    i.e. position of snake and player are the same*/
    int playerRow = mazeData[0][0];
    int playerCol = mazeData[0][1];
    int snakeRow = mazeData[1][0];
    int snakeCol = mazeData[1][1];

    if((playerRow == snakeRow) && (playerCol == snakeCol))
    {
        /*now change the player to ~*/
        mazeData[0][2] = SNAKE;
    }
}
```

Please see the end of the file if there are problems with the game, this will be mentioned as a possible point of error.

Undo Feature with Linked List (updateMaze.c)

The undo feature works with the idea that only the snake and player co-ordinates need to be stored per player input to complete the undo feature. This data is store in a **mazeEntitiesState** structure:

```
#ifndef MAZEENTITYSTATES_H
#define MAZEENTITYSTATES_H

typedef struct mazeEntitiesState {
    int* playerEntityState;
    int* snakeEntityState;
} mazeEntitiesState;

#endif
```

Each **playerEntityState** and **snakeEntityState** are dynamically allocated integer arrays which hold data in the form:

playerEntityState: [playerRow, playerColumn, playerArrowType]

snakeEntityState: [snakeRow, snakeColumn]

Each time the user makes a valid move in the game, even into a wall, this state of both the snake and the player will be stored in a new dynamically allocated structure and added to the **end** of the **pastStates** linked list.

```
if(input != 'u')
{
    addToPastStates(table, pastStates);
    updatePlayerPosition(&table, input, &rows);
    updateSnakePosition(&table, &rows);
}
```

Once the player presses **'u'**, the program will call the **removeLast** function of the linked list and return a **void*** pointer to the data stored in the last node. This is the **mazeEntitiesState** structure from the past move.

```
entityStates = (mazeEntitiesState*)removeLast(pastStates);

/*if entityStates is null, that means the linked list is empty*/
if(entityStates != NULL)
{
    revertEntityStates(entityStates, table);
    free(entityStates->playerEntityState);
    free(entityStates->snakeEntityState);
}
```

If the returned pointer is NULL, this means that the end of the linked list was reached and the **removeLast** function (in my case at least) was written to return NULL if the linked list was empty. If the data retrieved is valid, the **revertEntityStates** function will replace the current snake and player metadata with the past metadata stored in the entityStates variable.

```
void revertEntityStates(mazeEntitiesState* entityStates, int** table)
{
    /*player past state*/
    table[0][0] = entityStates->playerEntityState[0];
    table[0][1] = entityStates->playerEntityState[1];
    table[0][2] = entityStates->playerEntityState[2];

    /*snake past state*/
    table[1][0] = entityStates->snakeEntityState[0];
    table[1][1] = entityStates->snakeEntityState[1];
}
```

Since the metadata was sorted upon being read in, this can be done.

The previously stored snake and player data within the structure is freed.

Possible Points of Failure of the Program

At the moment, the program has only been used with the sample metadata provided on blackboard. However, upon changing some things there did occur an error.

When changing the order of the data in the file, an issue occurred where after reaching the goal, to the [snakeRow][snakeColumn + 1] co-ordinate would appear the player's arrow. The errors stopped happening after removing a condition in the snakeMovement.c file but I am not certain as to why this happened in the first place and since I have only used one input file it is possible I did not fix it and the issue is still there. So if there are issues with:

- **Game saying 'You Lost.' Even though the player made it to the goal, snakeMovement.c is most likely the issue.**
- **The co-ordinate [snakeRow][snakeColumn+1] showing the player arrow after the game has been won. If this happens, snakeMovement.c is the most likely reason.**

If snakeMovement.c is not the issue, then the only reasonable reason is in the sorting of the file. So fileIO.c would be the next most likely cause. I cannot seem to replicate the error so it may be fixed but if not, the information is above.