Nathan Coburn – snhx33

# C: Reflective Report

Throughout the production of connect4.c, I have aimed to be memory efficient by dynamically allocating data structures to contain the contents of the board, as well as freeing memory spaces when necessary such as when an error has occurred. I have aimed to be robust, by adding appropriate error checking in all areas of the program and outputting error messages to inform the user of what has gone wrong.

1. **board_structure:**
   The way in which I held the grid in memory was to create a 2D char array that would be filled when the input file is read in. This made it very easy to access every element of the grid so that I could manipulate it according to the users' inputs. In addition to a board array, I set variables to keep track of the number of rows, columns, as well as who's turn it is to make a move next. This made it very easy to perform other things later on in the program such as setting ranges for loops and validating the users' inputs.
2. **setup_board:**
   In this section of the program, a define a new instance of a board structure that can be used throughout the program by allocating a block of memory the size of the board_structure. This is a memory efficient method as it doesn't use more memory than is necessary to store the instance of the board as it is dynamically allocated. This also theoretically allows for multiple boards to be stored in memory at once. For the allocation of the memory for the board, I have error checks to ensure the malloc is valid and if not, a relevant error message is displayed.
3. **cleanup_board:**
   This function makes use of the free() function to clear all strings stored in the board structure as well as the board structure itself. This becomes useful throughout the rest of the program to free up memory in the case of an error that causes the program to halt.
4. **read_in_file:**
   In this function, the grid is read in from the input file and stored in a dynamically allocated 2d char array. To do this, I read through each character on the board and added it to a current line variable until it reached the \n character which indicated that it was the end of the line. Once the new line character had been found, I added the contents of the current line to the board array by allocating a block of memory the size of the current line. I iterated through each line of the file reallocating memory as I go along to accommodate the data that was being inserted. This is a memory efficient method as the memory allocated for the board is the exact amount needed to store the contents of the input file, so no memory is wasted. In addition to this, the current line variable is freed after every iteration to minimise unnecessary data stored in memory. For robustness, I added error checking to ensure that the file entered was valid, such checking to see if there are invalid characters and if the file has contents.
5. **write_out_file:**
   In this function, the final grid is outputted to the out file provided. I have also checked to ensure that the file has been opened correctly from main.
6. **next_player:**
   In this function, I return the player who's turn it is next. I do this by reversing the current player whos value was initially set when the file was read in by counting up the number of x's and o' on the grid. If the current player is 'x', then the function will return 'o' and vice

versa. This was a memory efficient method as it meant that the file didn't have to be iterated through more times than necessary as the values were found while it was initially being read in.

7. **current_winner:**
   In this function, the current board is iterated through with the aim of finding winning tokens. The possible cases of winning tokens are 4 in a row horizontally, vertically, ascending diagonally and descending diagonally. If winning tokens are found, it capitalises them through the use of the *capitaliseWinner* function and outputs the state of the board being, either a draw, xWin or oWin. If no winning tokens are found, then no win is outputted, and the game continues. To improve the efficiency of the function. I checked for wins from each player in the same loop so that it doesn't have to iterate through the board twice. I also check to see if the board is full which would result in a draw being outputted.
8. **read_in_move:**
   This function takes in the users input for the column that they want to place their token in, the row they wish to rotate (if any) and stores it in the appropriate variables so they can be easily accessed in proceeding functions.
9. **is_valid_move:**
   This function checks whether the move that has just been made is within the range of both the number of rows, and the number of columns in the grid. It also checks to ensure that the column that the user wishes to enter their token into is not full. For robustness, if any of these are invalid, then an appropriate error message is displayed, and the user is given the opportunity to re-enter their input so that they don't lose all the progress they have made so far in the event that they make a mistake and enter an invalid input.
10. **is_winning_move:**
    In this function, a copy of the current board is created through the user of the *copyBoard* function. It then applies the current move to the board copy and outputs the resulting board state, whether it be win, draw or lose. The board along with all of its members are then freed from memory. This is a memory efficient method as firstly; the size of the board copy is dynamically allocated to be the same size as the main board, so no memory is wasted. In addition to that, the contents of the board copy are freed from memory at the end of the function so no unnecessary data is stored in memory.
11. **play_move:**
    In this function, a token is inserted into the chosen column and then a row is rotated if the user chooses to do so. After the row rotation, gravity is applied to any tokens that may have been affected by it. It does this through the user of my *gravity* function which finds the lowest free position for each token in the row that was rotated and placing the tokens into them spaces. It then checks for the spaces above the row that was rotated and applies gravity to any tokens that may be there.

**Improvements to main.c and connect.h:**

The main issue with the main.c is that an invalid move could result in a lot of inefficiency. This is due to the fact that when an invalid move is inputted, the loop returns back to the start and checks for the current winner on the board so even when no move has actually been made it checks for a change in the board. To improve the efficiency of this, the validity check should be made as soon as the move has been read in so that only valid moves cause the current winner to be checked.