Reflective Report

Skbh77

Board Data Structure

For the board data structure, I created a new "Position" struct. A single position struct represents one position in the grid. This struct holds the contents of the position as well as pointers to the position above, below, to the right and to the left. This allows the position to be read and updated for checking/adding counters to the board as well as to aid in traversing through the board when checking the current state of the board. This approach was used as it implements a slightly tweaked linked list, where the positions are the nodes, and the board is the list itself. The board structure then holds a pointer to the top-left position of the board, similarly to how a normal linked list would have a head, in addition to the number of rows and columns in the board which is updated upon reading in the file.

This approach makes traversing the list easy as going you can go from position to position in either of the 4 directions by following the pointers. Additionally, to change the contents of a position, either because of adding a counter or moving a row, you only need to change the character the pointer holds. Checking adjacent positions is also made simple using the pointers and is useful for checking for wins or checking if the position below is empty when correcting for gravity.

This data structure also makes reading in the board easier. As I do not know the size of the board when starting I need to dynamically create the board such that enough memory is allocated. This data structure makes this much easier, as I create a new position and assign it some memory upon reading in a new character and only if the character is a valid character, not including new line characters. This new position is then assigned the read in character as its contents and has its neighbour position pointers set to the correct positions. A key reason this makes it easier is because I didn't need to reallocate any memory. The data structure also allows for ease of freeing memory upon completion of the game.

Robustness

Checking for Malloc Failure

Firstly, every time malloc is called to allocate something to memory there is a check for whether null has been returned. In the event it is an error message is printed to stderr describing the error and the program is exited. This is particularly useful when allocated a new board or new position for an existing board.

Reading in board

A file is read in character by character. For each character read I compare it to a list of valid characters, consisting of the char '.', the two player counters 'x' and 'o' and finally the new line char. If the character is not one of those then the board is invalid, and I print an appropriate error message and exit the program.

Additionally, I keep a track of the current width of each row as well as storing the previous row's width. I then check that the board has a consistent width. To do this I check that the width has not exceeded the previous lines width as well as checking it is not less than the

previous width upon reading in the whole line. A suitable error message is then printed if this occurs, and the program is exited.

Files

When dealing with files I check if the file provided by the main programs are not null, so that they can be accessed. In the case they are null, an appropriate error message is returned, and the program is exited.

Reading in moves

As well as checking if a move is valid, in the is_valid_move function, I also check the user input is of the correct type. This is done by continuing to ask for an input until the user inputs a valid response. The input is read in as a string and then the string is checked to make sure all of the characters are digits, as any other characters would render the input invalid. Only when the entire string has been checked is it converted to an integer and the input as counted as valid. This procedure is done for both the column and row number inputs.

Memory Efficiency

Creating the board

As mentioned earlier, as the board is read in a new position struct is created for each character. This position is then allocated memory and adding to the correct place in the board. The reason this is memory efficient is that the memory is dynamically allocated position by position, meaning there is never too much or too little memory being allocated. Additionally, I only allocate memory for positions which represent a counter, discarding any newline characters, reducing the amount of memory needing to be allocated.

Cleaning the board

Upon calling the cleanup_board function the entire board is traversed through position by position, freeing each as it goes along, before the board structure is freed last. This stops any memory from being leaked when the program has finished running. Additionally, if the program has to exit because of an error found then the currently allocated memory for the board and positions is freed before exiting. This way, regardless of how the program finishes it should have freed all allocated memory before exiting to minimise leaks.

Improving main.c and connect4.h

Files

To improve upon the robustness of the main.c file, upon opening both the infile and outfile a check could be made to make sure these files are not null. As it currently stands, if there is an error opening either of the files no check is made and this could cause issues with reading in the board or writing the finishing board to file.