Programming Paradigms: Systems Programming Reflective Report by

Luke Smith (mkkq27)

Introduction

In this document I will be reflecting on the code I have written for part B of the Systems Programming assignment. The main focus of this report will be on the robustness and memory-efficiency of the code.

Board Data Structure

For the board data structure I chose to use three different struct members to represent it.
First, in order to store the board's contents I decided to use an array of character pointers called *rows*. The reason I chose to do this was because this offers the 2-dimensional functionality that I need in order to access each token within the board, as well as the flexibility of only needing to store pointers to strings within this array, enabling me to assign the memory for the actual strings when I am iterating over the input file itself. This means that the actual string memory allocation is abstracted away from the board data structure itself at first until I absolutely need to allocate it. Given the high chance for unexpected and/or incorrect input within the board input file, waiting to allocate the majority of the memory until I am iterating over the file itself is preferable.

As for the other two members of the board data structure, I realised that having the dimensions of the board defined would be useful for iterating over the board which would be necessary for operations such as checking for wins and applying gravity. As such I gave the *board_structure* two integer members: *height* and *width.* My decision to use the integer data type comes from the requirements that the column bounds were between 4 and 512, and the row bounds would fit into an integer. This decision means that all of the iteration bounds are defined at once rather than having to compute these for each iteration.

Robustness

Firstly, a number of robustness considerations are implemented within *read_in_file.* With this function handling the inputted board file, there are a lot of potential issues with unexpected and/or incorrect input which could leave the program's behaviour unexpected if unaddressed.

The first thing that was necessary to check is that the supplied *infile* file pointer is not null. Without this, the program could halt with a segmentation fault if the supplied input file does not exist. Within *read_in_file* I also check that the board obeys the valid row and column dimension limits and exit with an error if not. Similarly, I also check that each line in the input file has the same amount of characters, otherwise there are positions in the board that are not defined. Again, if the board does not pass this test the program terminates with an error. I also check the input file for invalid tokens in the board during the memory allocation iteration. I simply check that each token in the board is '.', 'x' or 'o' and exit if this is not the case as anything else is classed as invalid per the specification. Another robustness implementation I made within *read_in_file* was to check that the input file has correctly applied gravity to it or not. In the assignment specification it mentions that such a file would be deemed invalid but if somehow such a file were to be inputted, the program would not be defined for it and thus the results would be unpredictable. To solve this I simply iterate over the board and check that it obeys gravity and terminate with an error if not. Finally, I also make sure to check that each memory allocation is possible both in the initial *malloc* calls for the structures, and in the *realloc* call I make when dynamically allocating memory. This is important as the program could terminate with a segmentation fault without this.

Reading in player moves is another place within the program that has a high potential for incorrect and/or unexpected input. Within *read_in_move* I implemented a check that the player input was indeed an integer. This check is important as the specification explicitly mentions that input should be two integer values. Originally I did this by re-prompting the user for another input in the case that the input was not

an integer. I did this by checking *scanf*'s return value and if it was not 1 (i.e it did not find an integer value in the input) it would display an error message and reprompt until it did. However upon hearing guidance in the lecture to simply throw an error rather than re-prompting, I decided to simply throw an error in order to not interfere with any automated marking procedures.

Memory-efficiency

Firstly, one memory-efficiency decision I made was to validate input files during the memory allocation process rather than after. For example, checking tokens are valid, board dimensions are within the valid bounds and checking that the board's rows are all the same length are all validation tests that I perform during the file iteration so that I don't end up allocating a lot of memory to an invalid board that I then have to free.

Another memory-efficiency measure that I use in connect4.c is to make use of the modulo (%) operator when performing win checks. Doing this enables the easy wraparound functionality within the board without having to copy the board to check for winning combinations by lining the boards up together and checking for winning combinations.

Improving Robustness of main.c & connect4.h

main.c

Firstly, one area that could be improved in main.c is checking whether in fact the input file used for the *infile* file pointer is valid or not. I implement a workaround for this by checking the validity inside the *read_in_file* function, but it would be better to check validity before allocating memory to the board through the calling of *setup_board* as well as calling *read_in_file.*

Another area of improvement could be to allow for the generation of a standard m x n board where m and are valid, user-supplied integers representing the generated board's rows and columns, respectively. This would make the program more robust in the case that the user hasn't already generated a board input file prior to running the program or the board that they have supplied is invalid.

While more of an efficiency improvement rather than robustness, the call to *write_out_file* on line 11 could be removed completely by moving the *write_out_file* call on line 17 to the first expression inside the while loop beginning on line 13. This would produce very similar results with one less call to *write_out_file* which is a fairly expensive function due to its board iteration when capitalising winning combinations. Also, having a counter for turns within the while loop to allow for quick computation of which player's turn it is rather than having a function in *next_player* would be much more efficient as *next_player* requires a full iteration of the board to count the tokens and compute the next player.

connect4.h

Firstly, one robustness improvement to connect4.h could be adding documentation to describe definitions. In the case of online publication or distribution, definitions within this file for struct member purposes would be useful for using the header file or extending it.

Another improvement could be to make use of the #ifndef and #ifdef directive. For this assignment's purposes this isn't as much of an issue but if it were to be extended and used in a larger project with other header files, I could end up overwriting definitions. Checking if I have already defined objects before I define them prevents various compile issues.

Conclusion

In conclusion, this has been a reflective report on part B of the Systems Programming assignment.