Programming Paradigms Reflective Report

Elias Percy

January 21, 2021

1 Introduction

In this document I shall outline my approach to the **Twist-and-Turn Connect 4** part of the coursework. I will elaborate on the general philosophy behind my programming in terms of stylistic and structural choices as well as the use of specific variables, functions, etc. Additionally, I will highlight the methods I used to ensure *robustness* and *memory-efficiency* within my program.

My code was almost entirely written in Vim, and to establish readable and consistent code I wrote according to Stroustrup's *PPP Style Guide*[1] as, after some investigation, this is the standard that appeared to me to best blend legibility and intuition. Although this style guide is intended for C++, it is applicable to C as well. In the following, I will elaborate on various key areas of the code in order to discuss the aforementioned points.

2 Data Structure for the "Grid"

My initial idea was to use a dynamic 2-D array to represent the grid; my final decision was to use a 1-D array of **char**s (akin to a string) instead, which I indexed as if it were a 2-D array; this approach consumes less memory than the 2-D approach. It at first seemed as if there'd be no real benefit for doing this, as a readily allocated 2-D array would have the same size as a 1-D array containing the same number of elements of the same type (i.e., a $\mathbf{3{\times}3}$ 2-D array of chars would have a size of **9** bytes, if the size of **char** is **1** byte, which would be the same for a $\mathbf{1{\times}9}$ 1-D array of chars.) However, since the inputted board structure could vary in shape and size, *dynamically-allocating* the grid is necessary: for the 2-D approach, this would have involved initialising a double **char** pointer, and the 2-D array created will contain extra memory: the memory taken by the **char**s themselves in addition to the memory taken by the pointers. A $\mathbf{3{\times}3}$ dynamically allocated array of chars would contain **3** pointers (one to each "row") as well as the pointer to the entire array. So, if a pointer takes **8** bytes, it would require $\mathbf{4*8=32}$ bytes more of memory, totalling **41** bytes. For very large boards, this makes a fairly moderate difference. By using a dynamic 1-D array, you don't need to make *pointers-to-pointers* that take up more memory, instead you only need a single pointer (so the overall size would be the sum of the size of the type stored in the array plus the size of only one pointer.) Therefore, it is more *memory-efficient.* Another advantage of using a 1-D array is the increased simplicity that dynamically allocating the grid now has, as well as **free**-ing being slightly simpler (no loop required). The slightly increased complexity that indexing the array now has is easily overcome by some simple mathematics. In general, dealing with a single pointer is much simpler than a pointer-to-pointers, which is what a dynamic 2-D array is.

Also, by appending '\0' to the end of this array, it now becomes (synonymous with) a string: importantly, this allowed me to utilise certain library functions, such as **strcpy()** and **strchr()**, so that I was able to implement certain functionality in a neater and more efficient manner.

3 Various Methods for Robustness

The definition of *robust* that underlies my programming philosophy within this project follows from the definition outlined by The Linux Information Project[2]. In short, it entails writing code that anticipates the unexpected, and handles any possible situation thrown at the program. My main approach in terms of abiding by this was to incorporate error handling substantially throughout my code, focusing on any areas that are particularly susceptible to errors, such as memory management and IO parsing. If any error is to occur, my program calls **exit(1)** to halt the program (in doing so, all allocated memory is **free**'d).

- At any point where a **malloc()** function was called, I first initialised the pointer to **NULL** before calling the function, and then I checked to ensure the pointer was not still **NULL** afterwards (as, upon failure, these functions return **NULL**, and if this occurred then there was likely a fatal memory issue). If failure was to occur, the error would be appropriately handled by reporting the problem to stderr and haling the program with **exit(1)**.
- Additionally, I minimised the quantity of **-alloc()** usage to reduce the probability of any of the associated errors to occur (in total, my program uses **malloc()** twice, excluding the optional **is_valid_move()** function which uses it an additional two times to duplicate the board).
- Whenever the program deals with a file, before configuring anything it first ensures that the file exists and is valid (by checking whether it is equal to **NULL**). With regards to the **initial_board.txt** itself, my program thoroughly checks to confirm that it is valid, including checking the dimensions of the board are consistent and usable, checking for any invalid or floating tokens, and assuring that there is a valid ratio of 'x' and 'o' tokens.

- Within the **read_in_move()** function, I implemented some methods to ensure robust IO parsing. This included using a buffer to read in the input as a string first to validate the input as being a usable integer: if this is not the case (or if the input exceeds the buffer size), the column or row is simply set to a value that exceeds the dimensions of the board, so the move will go on to be detected as invalid when **is_valid_move()** is called.
- Moreover, the code includes consistent checks within almost every function to validate that the procedure was successful and nothing unexpected arose. This includes asserting that what is being returned is consistent with the range of valid possibilities, and that internal function variables aren't set to anything they shouldn't be. Additionally, whenever a function (other than **is_valid_move()**) uses a **move** as a parameter, **is_valid_move()** is called to assure that this move is valid and won't result in any error (or segmentation fault).

Additionally, my program compiles with zero errors or warnings when using -Wall, -Wextra, and -pedantic. Also, valgrind does not detect any errors at all when running the program.

4 Various Methods for Memory Efficiency

To ensure and enforce *memory efficiency* within my program, I undertook several strategies. This includes:

- Minimising the size of the **board** struct; this was achieved by only storing *essential* information, that is to say the dimensions of the grid (number of rows and number of columns) as well as the grid itself. This was viable as any other important elements pertaining to the board (such as the next player, current winner, etc) could be obtained by scanning and observing the grid (which is incredibly quick in C, and particularly easy with a 1-D array), rather than keeping these stored in memory.
- Minimising the size of the **grid** itself; as outlined in the section about the grid specifically, I made a conscious effort to minimise the size of the grid, particularly by storing it as what is essentially a string - this lead to saving memory with regards to the number of *pointers* used. Furthermore, my program begins by scanning the input file to obtain the exact dimensions: this meant that I could allocate only the *precise amount* of memory when using **malloc()** for the grid, and no more.
- Making **is_valid_move()** optional; calling this function requires the program to *duplicate* the current board in memory so that this duplicate can be used for the duration of the function. Due to the obvious burden this has on memory, I constructed the program in such a way that this function is entirely optional: as my implementation of the program doesn't rely on this function to store the winner, the function can be omitted completely - it merely serves the purpose of informing whether or not a move will be a winning one *before* it has been played (so it can be implemented into **main.c** for a few uses). Therefore, the program will run smoothly without duplicating the board after each iteration (unless, of course, one decides to incorporate this function to some extent).

Moreover, I consistently checked my program using valgrind to thoroughly verify that there was no possibility of memory leakage. Additionally, I consciously minimised the program's use of **-alloc()** functions.

5 Possible Improvements for main.c and connect4.h

- When interacting with a file in **main.c**, I believe it would be most sensible to check for any errors immediately after the file is opened, rather than relying on another function to do so. So one improvement would be to include more *error checks* in **main.c**.
- Furthermore, I think it would be better for **main.c** to accept dynamic board input files. This is to say, rather than relying on the input board being called **initial_board.txt**, the user should be able to input a specific filename with their board structure in; or, users could simply input the dimensions of the board, and the board would then be generated from that information alone, rather than relying on an external file. This would mean that the program wouldn't need to open and interact with a file in order to display the board, rather all it would require would be two integer inputs from the user. This would lead to a more memory efficient program.
- The size of **connect4.h** could be reduced if **main.c** calls fewer functions from **connect4.c**. This could be achieved if the functions in **connect4.c** were to call each other in a specific sequential way.

---

1. `https://www.stroustrup.com/Programming/PPP-style.pdf`
2. `http://www.linfo.org/robust.html`