Reflective Report: Code Analysis

Programming Paradigms: Systems Coursework Part C
James Judd; Banner ID: 000688453
Submitted: 12/01/2021

In this report I shall discuss and justify the methods used in **connect4.c**, giving particular focus to robustness and memory efficiency. I shall also comment on possible improvements that could be made to **main.c** and **connect4.h**.

I chose a struct to be my main data structure, which included a set of seven member variables. One of these variables was a pointer to an array of pointers, which was used to store my connect four board. These member variables eliminated any need for global variables, making my code more generalised and versatile. The tiny memory expenditure for my four integer and two Boolean variables was greatly justified by the extent to which they streamlined my code, saving me a huge number of recursions of the same code. An array of pointers is a very memory-efficient way of storing a set of co-ordinates, with each co-ordinate containing a character. Using pointers made manipulating and duplicating the grid very simple and allowed me to use dynamic memory allocation to delay allocation until run-time. This was exceptionally advantageous as the board was only read-in at run-time and can have a width of anywhere between 4 and 512 characters, with up-to near-unlimited height. Fortunately, the board dimensions remain constant throughout the game so there is no need to reallocate memory after initialisation. Static memory allocation occurs during linking and is therefore more time efficient during execution, however with memory efficiency being the focus of the coursework, I used dynamic allocation whenever it was reasonable to do, such as with the board array. As the board can be infinitely tall, dynamic allocation also avoided any stack overflow errors.

I implemented multiple safety nets against both file errors and user input errors to improve robustness. File errors include invalid input and output files and empty or incorrectly formatted input files (for example non-uniform rows, blank lines or too few or many columns). I also checked that the board had abided to gravity, and that the game was not already either won or drawn. User input errors include empty inputs, non-integer inputs or inputs of more than twenty digits (the maximum length of an unsigned long integer). These scenarios send an error message to standard error and then exit with a non-zero error code, as per the specification. For other invalid user inputs (for example if they are outside of the board's boundaries), the specification asks for the *read_in_move* function to return 1, which I put in place of the standard error proceedings.

My program was made far more robust and concise by capitalising the winning tokens in my *current_winner* function instead of my *write_out_file* function. If *write_out_file* was to capitalise the tokens itself, it would have large amounts of identical code to my *current_winner* function as they both would have checks for winning positions. One way to avoid this would be to have *current_winner* store the location of the winning tokens in memory, however as memory efficiency is a focus of this coursework, I decided against this. Condensing the token capitalisation into *current_winner* requires *write_out_file* to be able to call it, otherwise a script that simply plays moves and writes to file (as in **test1.c**) would lead to my code not realising that there was a winner. This creates the potential issue of a script calling *current_winner* twice in one turn (directly and via *write_out_file*), thus capitalising more than one set of identical characters, should a player get more than multiple "connect fours" simultaneously. I therefore gave my board structure a member variable that keeps track of whether there has been a winner. This also allowed me to improve the time-efficiency of my program with *current_winner* having fewer internal iterative loops and *write_out_file* having the option to skip *current_winner* entirely. This function format is subsequently robust and both memory and time efficient.

Further efficiency was gained through the logic that each rotated character only has 2 outcomes. If a twist-row character is blank: after twisting, iterate upwards until you hit another blank, dropping intermediate tokens by 1. If, instead, the character is a token, you can ignore all positions above it as an overhead token guarantees a token underneath as well. This saves many iterative loops thus improving speed.

The ***while** (current_winner)* loop in **main.c** followed by a *write_out_file* call initially allowed scope for my code to double-capitalise boards, before I adjusted for this in **connect4.c**. This problem is specific to my function structure however to improve robustness against this problem, a *is_winning_move* branch could be implemented within the ***if*** statement. This branch would call a dedicated **connect4.c** function that plays the move and capitalises the tokens. This would require a new function to be added to **connect4.c** but would be by far the most robust method.

There are no safety nets in **main.c** for invalid inputs, however as it is simply a sequence of function calls, the nets can be set up in the individual functions. The robustness of **main.c** is therefore very reliant on the code it is compiled with, which is not a reliable strategy.

As a **.h** file, **connect4.h** is simply a list of declarations and thus difficult to make more robust. Despite this, there is scope for optimisation. The variable *column* is type ***int***, despite only ever taking valid values of zero or above. Using type ***unsigned int*** would therefore be more appropriate and allow *column* to store larger values for the same amount of memory, should **connect4.c** ever be expanded well beyond 512 columns.