Systems Programming Reflective Report

Finlay Boyle

December 2020

1 Board Data Structure

To represent the grid in `struct board_structure` I have chosen to use an array of arrays by using `char**`. I have chosen to use this because the board consists of rows and columns. It was a suitable analogue for the board since each cell is represented by 2 indices representing the row and column so it was an appropriate choice to represent the grid. It was also suitable because it did not require a restriction on the number of rows. Instead I was able to use `realloc` to dynamically allocate space.

For the rows I have used `char*`. Each row can be stored as a `char*` with each character representing the values in each column on the row. Using `char` was also a memory-efficient choice because the C standard requires that `char` uses 1 byte. Compared to `int`, which could have been an alternative choice, it saves a significant amount of memory as `int` requires 4 bytes (although this is machine/compiler dependant unlike `char`).

Overall, this data structure was a suitable choice to represent the grid for the board and was simpler and more memory-efficient than some of the alternatives I considered (another alternative I considered was using bit fields in a `struct` as there are only 4 choices for the value in a position so only 2 bits would be needed although this would be complex to implement and unnecessary since we are using a small amount of memory anyway). It also was sensible because it matched the file format for importing and exporting the board so it was simple to visualise and keep track of what was happening when debugging.

2 Robustness

When loading the input files I needed to make sure that I only accepted valid input files. These had to fulfil the following requirements:

- Every row has the same number of characters
- Every row only contained 'x', 'o', '.' and '\n'
- There were at least 4 columns and at most 512 columns
- 'x' and 'o' had made an equal number of moves or 'x' had made one additional move
- Gravity had already taken affect on the input file

The board was only deemed valid if these conditions were met.

Another important aspect for robustness was accepting input from `stdin`. To ensure that the input for the next move was valid I checked that:

- `scanf` matched at least one integer
- The column was between at least 1 and at most the number of columns on the board
- The row was at least -(number of rows) and at most +(number of rows)
- The selected column is not already full

There were other places were I made sure the program was robust some of these are:

- Check that `*infile` and `*outfile` were not `NULL`
- Freeing memory and setting the corresponding pointers to `NULL` afterwards to prevent accidentally attempting to access free'd memory.
- Checking that `malloc` (and similar functions) did not return `NULL` (if they did then I printed to `stderr` and exited with a non-zero code)

3 Memory Efficiency

To make the program memory efficient I made use of `malloc` to dynamically allocate memory so that I was only using what I needed. Whenever I was done with a pointer I would free it immediately to reduce the memory taken up by the program. For example, when processing diagonals and columns I would only hold one column or one diagonal in memory at a time. Then I would do whatever operations necessary and free it before I allocated for another column or diagonal.

I also stored as little information about the board in order to save memory. The `struct board_structure` has little overhead. Excluding the grid (`char** positions`) there are 3 integers and 2 `struct move*` (each of which point to a maximum of 4 `struct move`s consisting of 2 integers).

The `get_diagonal`, `get_col` and `rotate_array` each use additional memory because they copy data from the grid into arrays. As discussed previously these duplicates are temporary and are free'd immediately after their use instead of at the end of the program to reduce the memory usage.

4 Improving `main.c` and `connect4.h`

The robustness of `main.c` could be improved by checking that `infile*` and `outfile*` are not `NULL`. If they are then a non-zero error code could be returned or alternatively it could ask the user for a different input file.

For `connect4.h` it could be improved by using a header guard (using `#ifndef` and `#define`) to prevent the inclusion of the header multiple times. This would prevent something unexpected happening if the header is included via another file later on if the project was expanded.