Overview

The aim of this practical was to produce a basic sudoku solver.

Design and Implementation

Checking Validity of Sudokus

A struct would be needed to store the data of the given sudoku. Within this struct would be stored n value passed, the grid size and the values of the grid stored within an array.

On instantiation of a sudoku object, memory is allocated for the object, as well as the necessary memory needed to store the sudoku grid. Once the object is done being used, the memory pointers are freed.

The sudoku is read through the console, with each value within the grid stored within its respective position in the data structure, allowing an easy visualisation of the structure, providing ease when it comes to manipulating the sudoku grid.

To check the validity of a given sudoku, each row, column and block have to checked to ensure no duplicates occur. Validity of a sudoku is represented by an enum with the possible values INVALID, INCOMPLETE and COMPLETE.

If a duplicate value is found, the validity returned is INVALID. If no duplicates are found, but a zero (which represents an empty slot within the sudoku grid) is found, the validity returned is INCOMPLETE, as the sudoku is yet unsolved. However, if no duplicates or zeros are discovered, the sudoku is solved, and so COMPLETE is returned.

When checking the rows, columns and blocks, discovering duplicates is prioritised over empty values. If a duplicate value is found, the system instantly stops checking all over values, as the entire sudoku is now invalid. However, when a zero is found, the rest of the grid must be checked *to ensure* no invalidities are found. This means when a zero is found, a boolean flag is triggered, but the system must continue in analysing the rest of the grid. If, at the completion of the analysis, no invalidities are discovered, it is still possible to solve the current problem.

Solving

To solve the sudoku a recursive function is used. The sudoku grid currently being inspected is passed into the method. The validity of this sudoku is then assessed. If the sudoku is invalid, INVALID is returned and the recursion stops. Likewise, if the sudoku is solved, COMPLETE is returned. Since there is a possibility for a problem to have multiple solutions, there is a global counter that counts how many COMPLETES are returned. If this is greater than one at the end of the recursion, the system prints “MULTIPLE”. If the output is INVALID, the system prints “UNSOLVABLE”. However, if only one solution has been found, the system shall print the solved sudoku grid.

When the validity of a sudoku returns INCOMPLETE, a recursive call is made. Before this, a new sudoku object is made, a copy of the original, and then altered. The new sudoku is passed into a method which finds the first zero and replaces it with another value. This new value is determined by a for loop ticking through all possible values and inserting each value into a unique copy of the sudoku object. Each of these are then passed into a recursive call of the solving method, until every zero value is occupied and either the object is COMPLETE of INVALID.

Since the solving method returns the validity of the overall problem, the actual solution (if there is one) never gets returned. To get around this, I implemented a global sudoku object to store the first solution found. If multiple solutions are discovered, these never need to be printed, so it doesn't matter which one is stored. However, if only one solution is found, this is the one that is printed to the console.

Testing

The stacscheck output can be seen within the image contained in the hand-in folder. My solution passes 33 out of 34 tests, the 34th being the build test for the advanced solver that doesn't exist.

Evaluation and Conclusion

I fell I have met the requirements of the specification to a high standard. This practical has greatly helped in expanding my understanding of pointers and their functionality.

The efficiency of my solution could be improved by having the system determine the possible values for unfilled slots of the grid, rather than solver checking every single possible value.