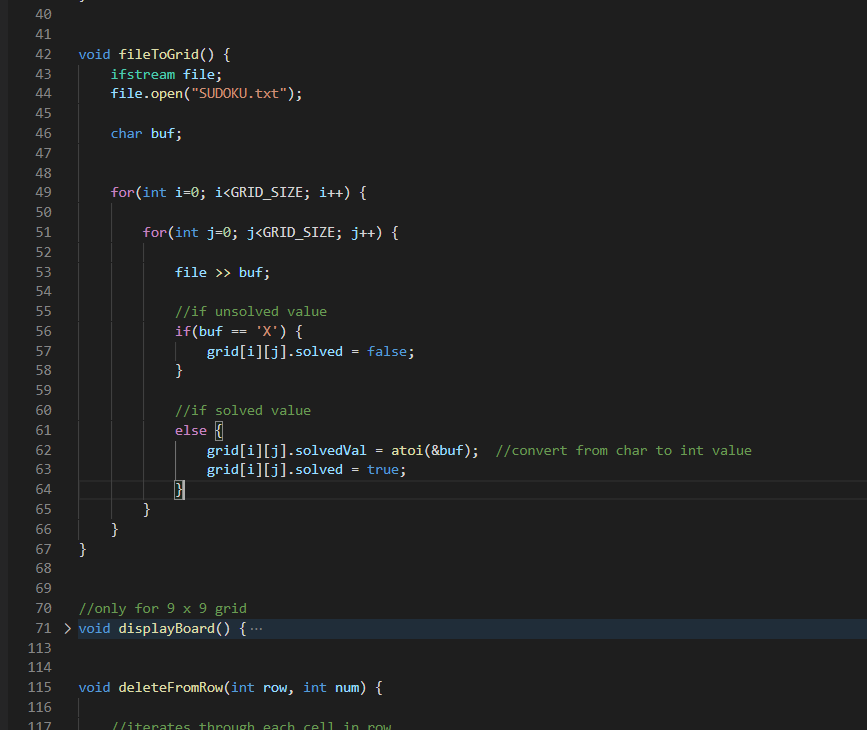
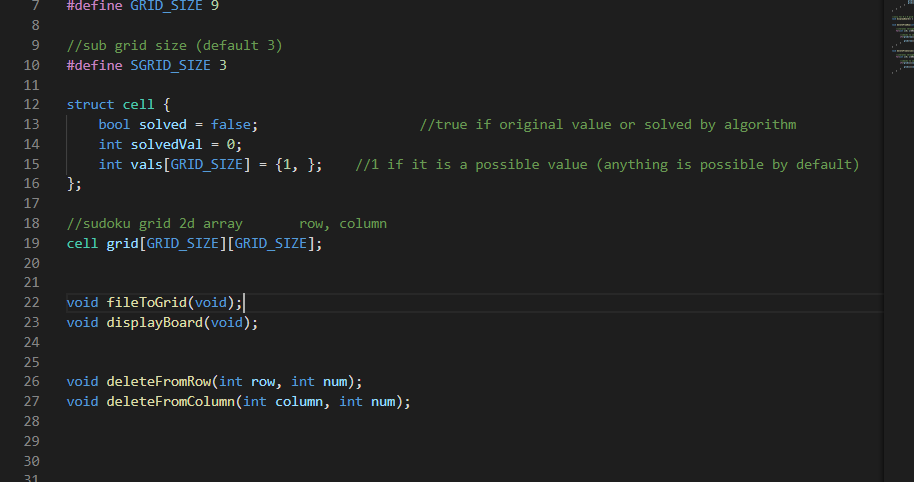
**P1 – Solving Sudoku**

3.1 – Final Reading

For the type of algorithm I am using to solve this sudoku a normal array would not be suitable. Therefore I created my own data type “cell” in order to do this.

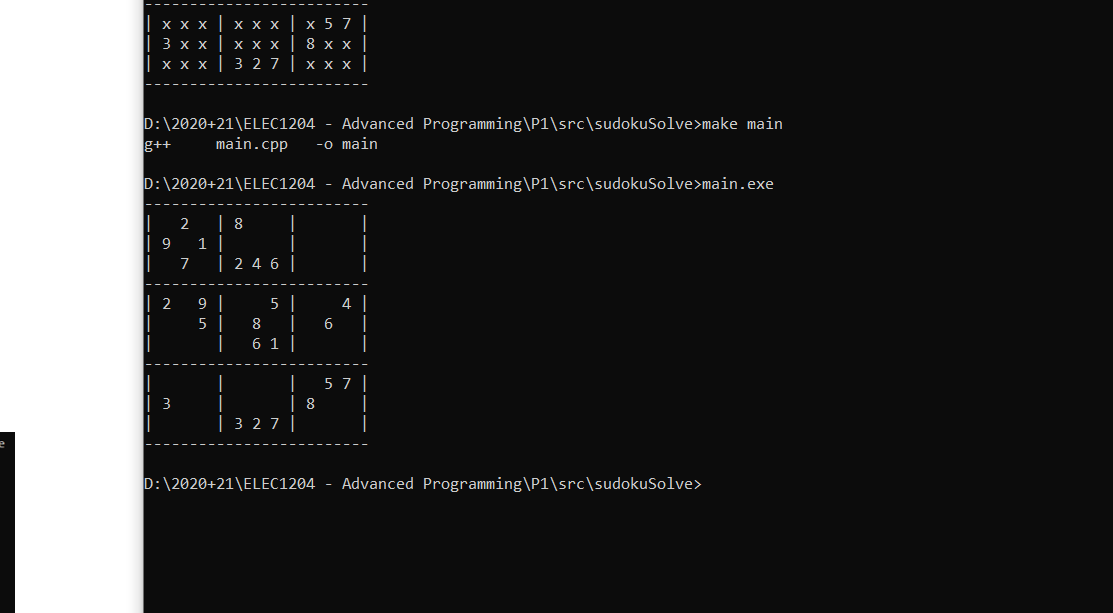




Whilst its not in a strict 9x9 integer array, it is in a format which will help me later.

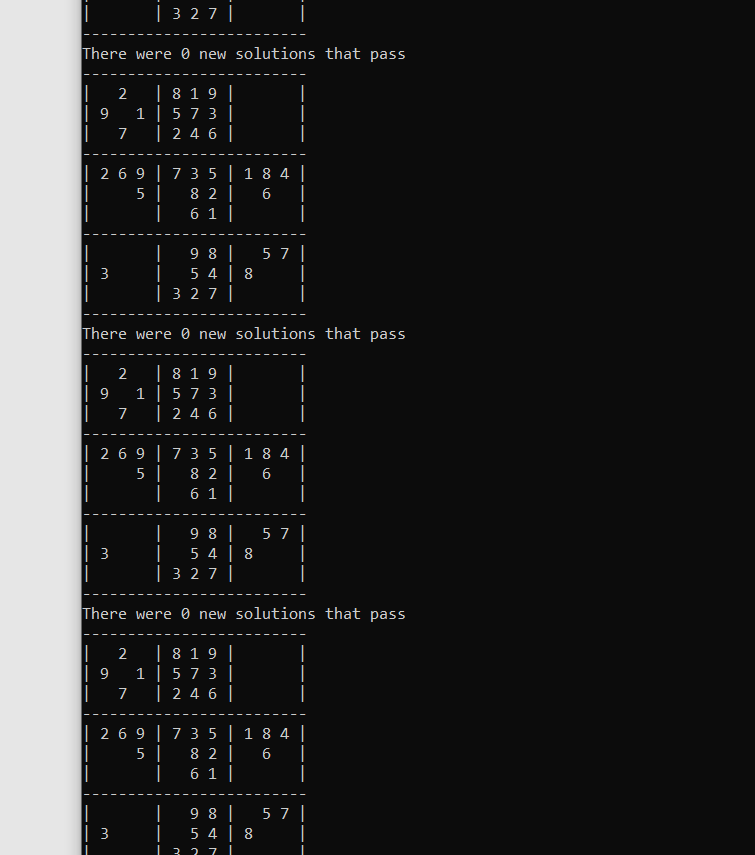
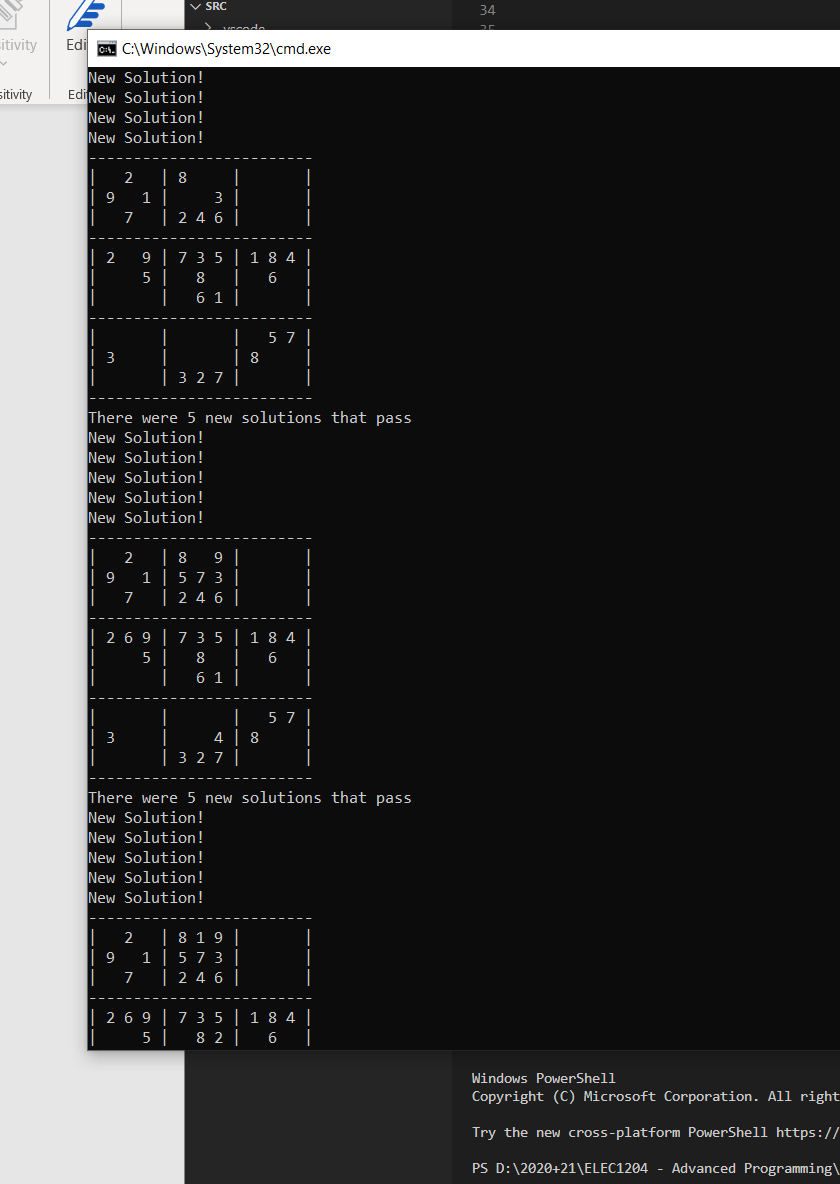
3.2 Sudoku Display





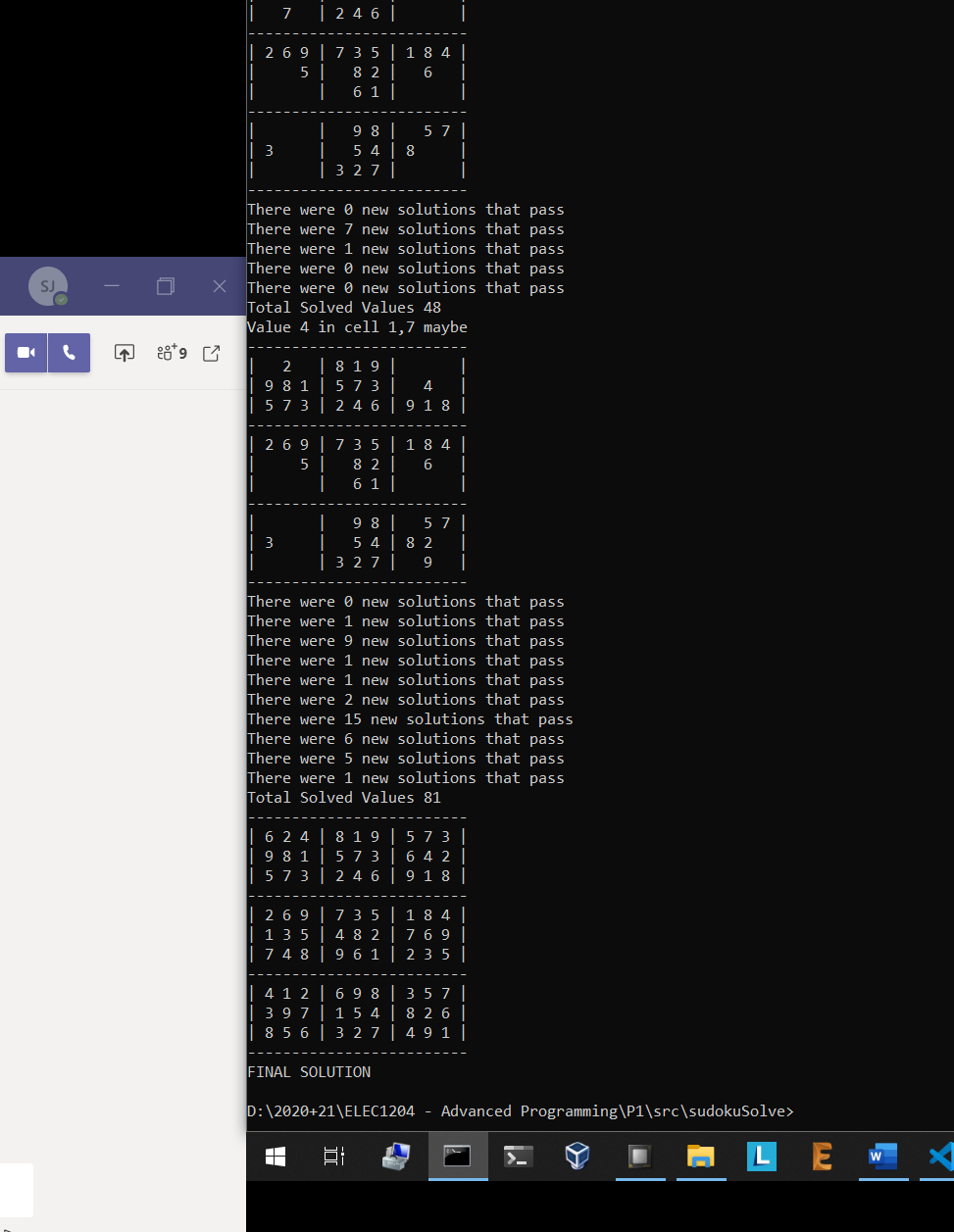
3.3 Sudoku Solving

On implementing my algorithm into my program it initially shows promising signs, but on testing the sudoku it gets stuck. It runs through deleting all the potential possible values in each the rows, columns and sub cells and then sees if there are any cells where only one possible solution is possible. This can be seen working below.



The only problem is that it gets to a state where it cannot continue since all the unsolved cells contain more than one possible value (as can be seen above). This means that a brute force method of trial an error must be used but this isn’t all bad news. Due to the way my algorithm works, each cell has an array of the possible values that could go there. This helps brute force solving quicker since in some cells there may only be 2 or 3 possible values that can go there so we don’t need to check all 9 possible numbers which can go there. This helps to reduce iterations and therefore quickens the process. To implement this easier I added two more variables. In the cell data type I added numSolutions which holds the number of possible solutions left for that cell. Also, I added a zeroError Boolean which will flag true if there is a cell with 0 solutions, this means a clash has occurred since every cell must have a solution.

On implementing this it works. This uses the same algorithm to do the actual solving except if it cannot solve it in the initial passes then it goes onto this combinational method. This tests a value to see if it works in that position if there is a zero-solution flag then the value cannot legally go there so it passes onto the next possible value. Once it goes through each of the values it should find one which helps the algorithm which I have written to solve the sudoku. This can be seen working below.



**Finished code**

#include <iostream>

#include <fstream>

#include <string.h>

using namespace std;

//one side of the grid (default 9)

#define GRID\_SIZE 9

//sub grid size (default 3)

#define SGRID\_SIZE 3

struct cell {

    bool solved = false;                    //true if original value or solved by algorithm

    int solvedVal = 0;

    int numSolutions = 0;

    int vals[GRID\_SIZE] = {1, 1, 1, 1, 1, 1, 1, 1, 1};    //1 if it is a possible value (anything is possible by default)

};

//sudoku grid 2d array       row, column

cell grid[GRID\_SIZE][GRID\_SIZE];

bool zeroError = false;

void fileToGrid(const char \* fileName);

void displayBoard(void);

bool isSolved(int row, int column);

void handleSolution(int row, int column);

bool solveGrid(void);

void deleteFromRow(int row, int num);

void deleteFromColumn(int column, int num);

void deleteFromSubcell(int row, int column, int num);

int main(int argc, char \*argv[]) {

    if(argc != 2) {

        fileToGrid("SUDOKU.txt");

    }

    else {

        fileToGrid(argv[1]);

    }

    displayBoard();

    int totalSolutions = 0;

    int newSolutions = 5;

    if(solveGrid()) {

        displayBoard();

        cout << "FINAL SOLUTION\n";

    }

    else {

        displayBoard();

        cout << "Initial Pass Failes\n";

    }

    cell savedGrid[GRID\_SIZE][GRID\_SIZE];

    memcpy(savedGrid, grid, sizeof(grid));      //save the last status of the solved grid

    //perform combination solve

    for(int row=0; row<GRID\_SIZE; row++) {

        for(int column=0; column<GRID\_SIZE; column++) {

            //finds the not solved cells

            if(!grid[row][column].solved) {

                for(int x=0; x<GRID\_SIZE; x++) {

                    //if its a possible value

                    if(savedGrid[row][column].vals[x] == 1) {

                        //reset grid to last know good point

                        memcpy(grid, savedGrid, sizeof(savedGrid));

                        grid[row][column].solvedVal = (x + 1);

                        grid[row][column].solved = true;

                        //test solution

                        if(solveGrid()) {

                            cout << "Value " << (x + 1) << " in cell " << row << "," << column << " WORKS\n";

                            displayBoard();

                            cout << "FINAL SOLUTION\n";

                            return 1;

                        }

                        else {

                            if(zeroError) {

                                zeroError = false;

                                cout << "Value " << (x + 1) << " in cell " << row << "," << column << " failed\n";

                                //memcpy(grid, savedGrid, sizeof(savedGrid));

                            }

                            else {

                                cout << "Value " << (x + 1) << " in cell " << row << "," << column << " maybe\n";

                            }

                        }

                        displayBoard();

                    }

                }

            }

        }

    }

}

bool solveGrid() {

    int totalSolutions = 0;

    int newSolutions = 0;

    int passNumber = 0;

    while((totalSolutions != (GRID\_SIZE \* GRID\_SIZE)) && (newSolutions != 0 || passNumber < 5)) {

        newSolutions = 0;

        totalSolutions = 0;

        for(int r=0; r<GRID\_SIZE; r++) {

            //iterate through columns

            for(int c=0; c<GRID\_SIZE; c++) {

                //test to see if solved (previously solved or newly solved)

                if(isSolved(r, c)) {

                    if(zeroError) {

                        //zeroError = false;

                        return false;

                    }

                    totalSolutions++;

                    newSolutions = (grid[r][c].solved) ? newSolutions : newSolutions + 1;

                    handleSolution(r, c);

                }

            }

        }

        passNumber++;

        //displayBoard();

        cout << "There were " << newSolutions << " new solutions that pass\n";

    }

    //true if there is a full grid of solutions

    cout << "Total Solved Values " << totalSolutions << "\n";

    return (bool) (totalSolutions == (GRID\_SIZE \* GRID\_SIZE));

}

void fileToGrid(const char \* fileName) {

    char buf;

    ifstream file;

    file.open(fileName);

    for(int i=0; i<GRID\_SIZE; i++) {

        for(int j=0; j<GRID\_SIZE; j++) {

            file >> buf;

            //if unsolved value

            if(buf == 'X') {

                grid[i][j].solved = false;

            }

            //if solved value

            else {

                grid[i][j].solvedVal = atoi(&buf);  //convert from char to int value

                grid[i][j].solved = true;

            }

        }

    }

}

//only for 9 x 9 grid

void displayBoard() {

    char val[1];

    //itterate row

    for(int i=0; i<GRID\_SIZE; i++) {

        //draw horizontal line

        if((i % SGRID\_SIZE) == 0) {

            for(int x=0; x<(GRID\_SIZE \* 2) + (GRID\_SIZE \* 2 / SGRID\_SIZE) + 1; x++) {

                cout << "-";

            }

            cout << "\n";

        }

        //itterate collumn

        for(int j=0; j<GRID\_SIZE; j++) {

            //draw vertical lines

            if((j % SGRID\_SIZE) == 0) {

                cout << "| ";

            }

            //write specific value

            if(grid[i][j].solved == true) {

                /\*val[0] = ' ';

                itoa(grid[i][j].solvedVal, val, 10);

                cout << val[0] << " ";\*/

                cout << grid[i][j].solvedVal << " ";

            }

            else {

                cout << "  ";

            }

        }

        cout << "|\n";

    }

    //horizontal line finishes grid

    for(int x=0; x<(GRID\_SIZE \* 2) + (GRID\_SIZE \* 2 / SGRID\_SIZE) + 1; x++) {

        cout << "-";

    }

    cout << "\n";

}

void deleteFromRow(int row, int num) {

    //iterates through each cell in row

    for(int i=0; i<GRID\_SIZE; i++) {

        //checks to see if not already solved

        if(!grid[row][i].solved) {

            grid[row][i].vals[num - 1] = 0;

        }

    }

}

void deleteFromColumn(int column, int num) {

    //iterates through each cell in column

    for(int i=0; i<GRID\_SIZE; i++) {

        //checks to see if not already solved

        if(!grid[i][column].solved) {

            grid[i][column].vals[num - 1] = 0;

        }

    }

}

void deleteFromSubcell(int row, int column, int num) {

    int subCellRow = (int) row / SGRID\_SIZE;

    int subCellColumn = (int) column / SGRID\_SIZE;

    for(int ir=0; ir<SGRID\_SIZE; ir++) {

        for(int ic=0; ic<SGRID\_SIZE; ic++) {

            grid[(subCellRow \* SGRID\_SIZE) + ir][(subCellColumn \* SGRID\_SIZE) + ic].vals[num - 1] = 0;

        }

    }

}

bool isSolved(int row, int column) {

    int total = 0;

    //if already existing solution

    if(grid[row][column].solved) {

        return true;

    }

    //if total is one then theres only one solution

    for(int i=0; i<GRID\_SIZE; i++) {

        total = (grid[row][column].vals[i] == 1) ? total + 1: total;

    }

    //cout << "Possible Solutions for " << row << "," << column << ": " << total << endl;

    grid[row][column].numSolutions = total;

    if(total > 1) {

        return false;

    }

    else if(total == 1) {

        return true;

    }

    else {

        cout << "ERROR 0 possible values\n";

        zeroError = true;

        return false;

    }

}

void handleSolution(int row, int column) {

    //if there is a new solution but the cell has not been set to true

    if(!grid[row][column].solved) {

        grid[row][column].solved = true;

        //cout << "New Solution!\n";

        for(int i=0; i<GRID\_SIZE; i++) {

            //finds the last remaining possible solution

            if(grid[row][column].vals[i] == 1) {

                grid[row][column].solvedVal = i + 1;

            }

        }

    }

    deleteFromColumn(column, grid[row][column].solvedVal);

    deleteFromRow(row, grid[row][column].solvedVal);

    deleteFromSubcell(row, column, grid[row][column].solvedVal);

}