Project 01

For this project, you will complete the provided partial C++ program that implements a modified version of John Conway's Game of Life cellular automata simulation of a biological system. In our version, there are two different types of living cells, Type1 and Type2, that may evolve based upon the rules summarized below. The simulation consists of a square grid of cells (**hint:** 2-D array) where each cell is either dead or alive. The current arrangement of living and dead cells is used to compute the next generation of cells using the following rules that determine the **Birth**, **Survival**, and **Death** of each cell.

Survival Rule
A living cell survives if it has some specified number of living neighbors of the
SAME type. **Hint:** If a living cell does not survive, it dies!
Death Rule
A living cell will die if it has too few neighbors of the SAME type (loneliness) or too many
neighbors (overcrowding).
Birth Rule
A dead cell becomes a living cell in the next generation if it has some specified number of living neighbors.

- If the number of living Type1 neighbors equals this number AND is greater than the number of living Type2 neighbors, then a Type1 cell is born.
- If the number of living Type2 neighbors equals this number AND is greater than the number of living Type1 neighbors, then a Type2 cell is born.
- Otherwise, no cell is born.

To make this simulation **reconfigurable**, you will be inputting the number of iterations, the birth and survival constraints, and the initial arrangement of living cells (**2**'s or **1**'s) and dead cells (**0**'s) from a file.

The simulation algorithm implemented by the **main()** function from the provided file **main.cpp** is summarized below:

(1) Load the number of iterations, birth rules and survival rules from the input
 file **Hint:** First line of file is comment describing file contents. Skip this
 comment.
(2) Load the initial arrangement of cells from the input file into the current grid
(3) Output initial grid in desired format (**Hint:** Iteration 0 is the initial grid from the file)
(4) Use birth/survival/death rules to compute next grid of cells from current grid of cells
(5) Copy (overwrite) current grid with next grid contents

(6) Display current grid (which now contains the new arrangement of cells) in desired format. **Hint: PrintGrid** function provided prints out grid in an easy to read format

(7) Repeat (4) – (7) until desired number of generations have been simulated

To implement the above algorithm, the **main()** function utilizes a number of support functions that you must implement and test.

Support Functions

```
void OpenInputFile(string filename, ifstream& inFile);
// OpenInputFile -- opens file whose name is stored in filename

void LoadConstraints(ifstream& inFile, int& num, string& bstring, string& sstring);
//   LoadConstraints -- loads only simulation constraints from inFile after skipping header comment.
     Constraints input in order num, birth string, survival string

void LoadGrid(ifstream& inFile, int grid[][CMAX]);
//   LoadGrid -- loads the cell grid from inFile

void ComputeNextGrid(int current[][CMAX], int next[][CMAX], int birth[], int survival[]);
//   ComputeNextGrid -- uses current generation to compute next generation using constraints
     specified in birth and survival arrays

void CopyGrid(const int source[][CMAX], int destination[][CMAX]);
// CopyGrid -- copies contents of source array into destination array

int    CountType1Neighbors(int grid[][CMAX], int row, int col);
//   CountType1Neighbors -- counts the total number of LIVING Type1 neighbors for the cell at the
     grid position specified by row and col.

int    CountType2Neighbors(int grid[][CMAX], int row, int col);
//   CountType2Neighbors -- counts the total number of LIVING Type2 neighbors for the cell at the
     grid position specified by row and col.

void ParseRequirementsString(string requirements, int reqs[]);
//   ParseRequirementsString -- takes a birth or survival string and converts it to integer array
     assumes that the B or S appears as the first character.   Order of constraints does not matter.
//   For example, B13 should produce same result as B31
//
//   (1's in both reqs[1] and reqs[3], zeros elsewhere)
//
//   Hint: use string functions to help you scan through each character in the requirements string one
     at a time and use nested IF or SWITCH to implement logic.
```