PIC 10B Section 1 - Homework # 6 (due Wednesday, May 15, by 6 pm)

You should upload each .cpp and .h file separately and submit them to CCLE before the due date/time! Your work will otherwise not be considered for grading. Do not submit a zipped up folder or any other type of file besides .h and .cpp.
Be sure you upload files with the precise name requested of you and that it matches the names you used in your editor environment otherwise there may be linker and other errors when your homeworks are compiled on a different machine. Also be sure your code compiles and runs on Visual Studio 2017.

Organize your code well and at the end of this work, submit **.h** and **.cpp** files that, when compiled and linked, will produce a program to solve a magic square.

At the end of this work you should submit a header file **MagicSquare.h** that defines all classes and declares all functions, a **MagicSquare.cpp** that implements those functions, and a **Solve.cpp** file with your main routine.

MAGIC SQUARE SOLVER

A magic square is an $n \times n$ arrangement of the numbers 1 through n^2 such that *each number only appears once*, and the sum of *every row*, *every column*, and *both main diagonals* is the same. You might remember solving these back in elementary school: teachers would do this to keep the students busy... and quiet.

Here's an example of one below:

$$\begin{bmatrix} 4 & 9 & 2 \\ 3 & 5 & 7 \\ 8 & 1 & 6 \end{bmatrix}$$

Note that the sums for *each row*: $4 + 9 + 2$, $3 + 5 + 7$, $8 + 1 + 6$ are all 15... as are the sums for *each column*: $4 + 3 + 8$, $9 + 5 + 1$, $2 + 7 + 6$... as are the sums for *both* diagonals: $4 + 5 + 6$, $2 + 5 + 8$.

In this homework, you will write the implementations to solve an n by n magic square where the user may, if they wish, enforce the placement of some numbers. **You must use recursion in this assignment. This is the whole point of this exercise. Credit will not be given if you do not use recursion.**

By having the numbers $1, 2, ..., n^2$ in the board, it can be shown mathematically that the sum of all those entries is $n^2(n^2+1)/2$. Since there are n rows (and columns) and the sum of each row (and column) is the same, the "magical" target sum is $1/n$ of the total sum. In other words to be a magic square solution, every row, column, and diagonal must sum to

$$S = n(n^2+1)/2.$$

The program should operate as follows

Enter a square size: [USER ENTERS VALID SQUARE SIZE]
Enter square format:
[USER ENTERS DATA ROW BY ROW WITH ENTRIES SEPARATED BY SPACES, ROWS SEPARATED BY [ENTER] AND WHERE * SIGNIFIES THE VALUE DOES NOT MATTER]
A list of all the solutions of the magic squares satisfying the user's constraints is displayed.
Solving complete!
There were [CORRECT NUMBER] of solutions!

See sample outputs:

```
Enter a square size: 2
Enter square format:
* *
* *
Solving complete!
There were 0 solutions!
```

```
Enter a square size: 3
Enter square format:
* 9 *
* * *
* * *

2 9 4
7 5 3
6 1 8

4 9 2
3 5 7
8 1 6

Solving complete!
There were 2 solutions!
```

```
Enter a square size: 3
Enter square format:
* * *
* * *
* * *
2 7 6
9 5 1
4 3 8

2 9 4
7 5 3
6 1 8

4 3 8
9 5 1
2 7 6

4 9 2
3 5 7
8 1 6

6 1 8
7 5 3
2 9 4

6 7 2
1 5 9
8 3 4

8 1 6
3 5 7
4 9 2

8 3 4
1 5 9
6 7 2

Solving complete!
There were 8 solutions!
```

You will write a **MagicSquare** class. It will store the square as a **std::vector** < **std::vector<int>>** where 0 indicates no value has been placed in the slot at a given row and column yet (recall the valid numbers are from 1 to n^2).

You must write:

- an overloaded **operator>>** to read from a stream into a **std::vector<std::vector<int>>** that will properly process the user's input (the vector being written to need not start empty – hint!);
- an overloaded **operator<<** to write a **std::vector<std::vector<int>>** with an output stream in the format demonstrated;
- a function **empty** to check if a given position in the magic square is empty;
- a function **taken** to check if a given number is already used in the magic square (it would make sense to store the taken positions in an **std::set** for easy lookup);
- a function **checkRow** that, after each row is filled (except for the last), checks if the square could potentially be valid - it will check that the sum of each row is the target sum S and that the sum of each partially filled column does not exceed S, returning **true** if the square is valid so far and **false** if it fails;
- a function **checkValid** to check if a complete magic square satisfies the proper conditions to be a solution; and
- a recursive function **solveSquare** that accepts an index tracking the number of slots already considered in the recursion, places values, and does the appropriate calls to **checkValid** and **checkRow**.

To solve this, there are a couple of strategies you could implement.

Strategy 1: emulate the permutation example done in lecture. You effectively need to generate all permutations of numbers 1 through n^2. You'll just need to make some modifications to not swap numbers that have a fixed place in the board and to do appropriate checks after each row is filled and at the end of the filling process.

Strategy 2: suppose that we look at the slots, possibly placing values within them, left-to-right, top-to-bottom, and that we have been through *num* slots already. Let's call *square* the **std::vector<std::vector<int>>>** and *used_up* the data structure storing all the numbers that have been used (not required but makes for easy lookup). Then:

- if *num* is equal to 0 then we should take the opportunity to store all of the "taken" positions already stored in the *square* in *used_up*;
- if *num* is equal to the total number of slots, we should check if the magic square is a valid solution and if so, print it, and increase our count of valid solutions;

- if not, then
  - if a row was just filled, we should **checkRow** and only proceed if it is **true**;
  - we should determine if the slot is empty (recall the user could specify values, too!) by looking at the corresponding row and column of *squares* and if empty,
    - we should try every possible value in the current slot, which is not already part of *used_up*, and try solving the magic square for each value we insert by repeating the process with *num* now $num+1$ and with the newly chosen number added to *used_up*;
    - and afterwards, for each unused value that we placed in the *square*, we should remove that value from *square* and *used_up* (in a similar spirit to the recursive process of generating permutations)
  - otherwise, we should skip over the current slot and try solving the magic square by looking at the next slot and replacing *num* by $num + 1$.

Your magic squares will only be tested up to sizes of 4×4, which should run in a reasonable time with some values filled in. Solving a 4×4 square from scratch is very slow (on Visual Studio) and there are many cases to check and more than 7000 solutions to find!

To test a 4×4, give yourself a main diagonal read left-to-right, top-to-bottom of 1, 13, 11, 9 - there should be 8 solutions (and even on Visual Studio, it should only take around 10-15 seconds). Or give yourself that same diagonal but impose a 16 in the top right and 8 in the bottom left - there should be 4.

Some initial guidance to get you on your way...

1. It may be helpful to allow the value 0 to indicate that a slot is open. Recall that for a **std::vector<int>** of a given size, this is the default value assigned to each element.
2. To write
   std::istream& operator>>(std::istream&,
   std::vector<std::vector<int>>&), your logic could depend upon the size of the vector.
3. Write your operator overloads first. Be sure you can read from and write to the vector.
4. Your **MagicSquare** class could store a **std::set** or **std::unordered_set** to track values that have been used up.
5. Your **empty** function may have signature **bool empty(size_t row, size_t col)**

6. Your **taken** function may have signature
   bool taken(int i)

7. Write **empty**, **taken**, **checkRow**, and **checkValid** next. You should test known magic squares as to whether they are solutions. Don't worry about solving them yourself yet, just use the examples in the display or come up with your own.

8. Finally, write **solveSquare**. Note that if you have found a solution, you should print it with **std::cout** and increase your count.