COP 4530: Data Structures
Project 1

You are not allowed to use the Internet. You may only consult approved references*.
This is an individual project.
This policy is strictly enforced.

You must submit a **hard copy** of all of the items requested below. You must also submit your *code*† to Canvas.

For full credit, the code that is submitted must:

- Use the specified signature, if applicable.
- Be implemented in a file using the specified file name, if applicable.
- Be correct (i.e., it must always return the correct result).
- Be efficient (i.e., it must use the minimum amount of time and the minimum amount of space necessary to be a correct implementation).
- Be readable and easy to understand. You should include comments to explain when needed, but you should not include excessive comments that makes the code difficult to read.
  - Every class definition should have an accompanying comment that describes what is for and how it should be used.
  - Every function should have declarative comments which describe the purpose, preconditions, and post conditions for the function.
  - In your implementation, you should have comments in tricky, non-obvious, interesting, or important parts of your code.
  - Pay attention to punctuation, spelling, and grammar.
- Follows ALL coding guidelines from section 1.3 of the textbook. Additional coding guidelines:
  - No magic numbers. Use constants in place of hard-coded numbers.
  - No line of the text of your source code file may have more than 80 characters (including whitespace).
  - All header files should have #define guards to prevent multiple file inclusion. The form of the symbol name should be <FILENAME>_H_
  - Do not copy and paste code. If you need to reuse a section of code, then write a function that performs that code.
  - Define functions inline only when they are small, say, 10 lines or less
  - Function names, variable names, and filenames must be description. Avoid abbreviation.
  - Use only spaces (no tabs), and indent 3 spaces at a time.
- Compile and run on the C4 Linux Lab machines (g++ compiler, version 4.8.2). *The shell script and makefile that I will use to compile and run your code will be posted on Canvas. Please note that I may use my own* `main.cpp` *file to test the code you submit.*
- Have no memory leaks.

*The list of approved references is posted on Canvas. You must cite all references used.
†Your code must compile and run on the C4 Linux Lab machines

Project Description

A *magic square* is an $n \times n$ matrix in which each of the integers $1, 2, 3, \ldots, n^2$ appears exactly once an all column sums, row sums, and diagonal sums are equal. For example, the following is a 5×5 magic square in which all rows, columns, and diagonals add up to 65:

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

The following is a procedure for constructing an $n \times n$ magic square for any odd integer n. Place 1 in the middle of the top row. Then, after integer k has been placed, move up one row and move one column right to place the next integer, $k + 1$, unless one of the following occurs:

- If a move takes you above the top row in the j^{th} column, move to the bottom of the j^{th} column and place $k + 1$ there.
- If a move takes you outside to the right of the square in the i^{th} row, place $k + 1$ in the i^{th} row at the left side.
- If a move takes you to an already filled square or if you move out of the square at the upper right-hand corner, place $k + 1$ immediately below k

Project Tasks

1. Define a class to hold a `MagicSquare` object in file `magicSquare.h`. The implementation of each of the member functions will be in file `magicSquare.cpp`.

   (a) [5 points] The default public constructor should generate a 5×5 magic square.
   `MagicSquare()`

   (b) [5 points] Implement a public constructor that takes an odd integer, n, as a parameter, which should generate an $n \times n$ magic square. *You should assume that n will always be an odd integer.*
   `MagicSquare(const int& n)`

   (c) [10 points] Implement a private generator function to create a magic square using the procedure described in the project description. *You should assume that you are only using this function to create magic squares for an odd n. Hint: your constructors should call this function.*
   `void generateMagicSquare()`

   (d) [4 points] Overload the << operator to output the magic square to an `ostream`.
   Magic Squares are output on $n + 1$ lines. The first list should indicate n, the size of the magic square. The next n lines should each have n integers, separated by one space, which is the magic square. For example, the output for a 5×5 magic square would be:

```
5
17 24 1 8 15
23 5 7 14 16
4 6 13 20 22
10 12 19 21 3
11 18 25 2 9
```

   (e) [4 points] Overload the >> operator to input a magic square from `istream`. *The magic square will be input in the same format as it was output.*

   (f) [10 points] Implement a public function to test whether the object is a magic square. This function must test all of the criteria that we used to define a magic square. The function should return `true` if the object meets the criteria for a magic square and `false` otherwise.
   This function would be useful to verify that a magic square that was input using the overloaded >> is valid.
   `bool isMagicSquare()`

   (g) [10 points] If some $n \times n$ matrix is a magic square, then rotating the matrix by 90^0 would also be a magic square. Implement a public function to create a new magic square object that rotates the calling magic square object
   `MagicSquare rotate()`
   For example, below is the 90^0 rotation of the 5×5 magic square provided in the project description:

11	10	4	23	17
18	12	6	5	24
25	19	13	7	1
2	21	20	14	8
9	3	22	16	15

You should implement additional helper functions as needed. Helper functions must also be located in the `magicSquare.h` and `magicSquare.cpp` files.

2. Write a main function to test each of the member functions of your MagicSquare class in file `main.cpp`.
   (a) [2 points] Create a magic square object using the default constructor.
      i. Verify that the magic square object is in fact a magic square.
      ii. If the object is a valid magic square, then output the magic square to file `output.txt`
   (b) [5 points] Create magic squares of sizes 7,9,11,13,15,17,19,21,23,25. For each magic square:
      i. Verify that the object created is in fact a magic square.
      ii. If the object is a valid magic square, then output the magic square to file `output.txt` (*There should be a blank line between each magic square output*)
      iii. Rotate the magic square. Verify that the result is in fact a magic square. If the result is a valid magic square, then output the magic square to file `output.txt` otherwise, output the magic square to console.
   (c) [5 points] Test all instances of magic squares that are in input file named `input.txt`.
      i. The format of the input file is as follows. The first line of the input file indicates the number of instances in the file followed by a blank line. Each instances consists of $n + 1$ lines, where n denotes the size of the instance. The following n lines are a row from the magic square instance, where each line contains n integers which are separated by spaces. There is a blank line between consecutive instances. For example, an `input.txt` file with two 5×5 instances could be:

```
2

5
17 24 1 8 15
23 5 7 14 16
4 6 13 20 22
10 12 19 21 3
11 18 25 2 9

5
11 10 4 23 17
18 12 6 5 24
25 19 13 7 1
2 21 20 14 8
9 3 22 16 15
```

      ii. For each instance, test that is a valid magic square.
      If instance i is a magic square you should output (to console): `i: is valid` If instance i is not a magic square you should output (to console): `i: is not valid` For example, one sequence of outputs might be:

```
0: is valid
1: is valid
2: is not valid
3: is not valid
4: is valid
```

      iii. You should create your own `input.txt` file to test various potential matrices to determine if they are a magic square - you must submit your `input.txt` file. *Note that I will be using my own* ***`input.txt`*** *to test your code.*