*C++ requirements*

* The program must make use of a two dimensional array that is 21 rows by 21 columns. The values in the array must be of type int.
* To help facilitate this you need to create a global const int value for the maximum number of rows and columns. *No other global variables are allowed*.
* Your program must properly open and close all files.
* You must use function prototypes for all functions (except main).
* You are required to have at least four functions, including main. You need to have a readSquare function, a validateSquare function, and a display magic square function. You can have additional functions, but you must have these four functions.
* The signatures for the readSquare, and validateSquare functions must match the requirements below. There will be unit tests that call these functions.
* The function signatures of the display function and any other functions you want to create are at your discretion.
* All functions except main must have function prototypes.
* The only global variable you are allowed to have is the const int value for the number of rows and columns in the magic square.

Failure to follow the C++ requirements could reduce the points received from passing the tests.

*General overview*

You will determine if a magic square is valid or not. The size of the magic square will be some number between 3 and 21 inclusive.

This program will make use of a two dimension array.

A Lo Shu Magic Square is a grid with 3 rows and 3 columns as shown here:

+---+---+---+

| 4 | 9 | 2 |

+---+---+---+

| 3 | 5 | 7 |

+---+---+---+

| 8 | 1 | 6 |

+---+---+---+

Note that the actual input file will only contain the numbers and not the other characters shown above.

The 3x3 Lo Shu Magic Square has the following properties:

* The grid contains the numbers 1 through 9. The range of values will be different for puzzles of sizes other than 3.
* The sum of each row, each column, and each diagonal all add up to the same number - 15. This sum will be different for puzzles of different size.

Your program will simulate a magic square using a two-dimensional 21 x 21 array of int values. Your program needs to be able to support input files of 3x3 up to 21x21.

For a 3x3 magic square the code for testing if a row or column adds up to the requirement sum is fairly straight forward. The sum is 15.

The code for testing the diagonal values and for testing that the square contains all of the numbers 1 through n is trickier (n is size \* size). You need to sit down with a pad of paper and some pencils and work out the strategy you will need to figure these out. For a 3x3 puzzle the valid values in the puzzle are 1 to 9.

For the diagonal going from the top left to the bottom right of the magic square you need to add up square[n][n] where n goes from 0 to 2. The diagonal going from the lower left to the upper right will require that you add up square[2][0], square[1][1], and square[[0][2]. Note that the first array subscript is going down (2, then 1, then 0) while the second array subscript is going up (0, 1 and then 2). For both of these you need to use a loop to add these values instead of just hard coding the tests. We would like to be able to change the program in the future to work with magic squares of larger sizes.

For the testing of all values you need to check to see if any element has a value of 1, If it does you then check to see if any array element has a 2, and so on to check for all of the values 1 through n. You will probably need nested loops to do this. If any of the elements 1 through n are not found you can stop testing and declare that it isn't a valid magic square. There are other ways to solve this. If you have a different solution you can feel free to use that.

In the program you will be writing two required functions. You will have more than two functions, but these two are required with specific signatures.

For other sizes other than 3 the value that the rows, columns and diagonals add up to is not going to be 15, but is calculated as:

size \* (size \* size + 1) / 2

The numbers you need to check will not be 1 through 9 but rather 1 through (size \* size). A 3x3 magic square would contain exactly 1 of the values 1 through 8 (size\*size for size 3). A 5x5 magic square would contain the values 1 through 25 (size \* size where size is 5).

*Validate function*

The validate function has the following signature:

const int MAX\_SIZE = 21;

bool validateSquare(const int square[MAX\_SIZE][MAX\_SIZE], int size);

Note that MAX\_SIZE is a global const int value. This is the *only* global variable you are allowed to have in the program.

The validateSquare function will return true if the square array contains a valid magic square and false if any of the tests fail.

*Read function*

The read function will read the contents of a file into the passed in 21x21 array. If the file exists you can assume it contains valid date. You do, however, have to check to see if the file is opened properly or not.

Here is the signature:

int readSquare(int square[MAX\_SIZE][MAX\_SIZE], string inputFileName);

The array you are reading into is passed as the first argument to the function. The second argument is the input file name.

If the input file does not exist (the open fails) your function needs to return a value of 0.

If the file is opened properly you need to read the contents of the file into the array and return the actual size of the magic square.

As always, close the file once you have finished reading from it.

The input file format for a 3x3 puzzle would be:

3

4 9 2

3 5 7

8 1 6

Note that the first value read in (3 above) is the size of the puzzle. This is a 3x3 puzzle.

For a 5x5 it would be:

5

9 2 25 18 11

3 21 19 12 10

22 20 13 6 4

16 14 7 5 23

15 8 1 24 17

Note that the first value read in (5 above) is the size of the puzzle. This is a 5x5 puzzle.

*The main function*

You will need your main function and you will need a function that displays the contents of the magic square, one row per line of output.

The pseudocode for main is:

Read in the input file name from cin

Call the readSquare function to read the magic square from the specified input file.

If the readSquare function worked (returned a \*true\* value) you need to:

1. Call the display function to display the contents of the magic square

2. Call the validateSquare funciton.

3. If the magic square is valid

-. Display "Valid magic square"

4. If the magic square is not valid

-. Display "Invalid magic square"

else

1. Display message indicating the file could not be opened (see sample output for syntax of message)

*display magic square function*

The display function should output the magic square one row at a time with a heading of "Magic square".

Here is the output from the display function (assuming the magic square shown at the top of these directions):

Magic square

4 9 2

3 5 7

8 1 6

*Other (optional) functions*

You can have additional functions if you want. You may want to have a function that tests the contents of a row, another that validates a column, one that validates the diagonals, and so on. You could then call those functions from the *validateSquare* function.

One reason to do this is that it helps you break the problem into smaller pieces. That way you can focus on one part of the problem before having to deal with the next part.

*Here are some sample runs*

*Sample run 1 (valid data)*

Contents of cin:

square1.txt

Contents of *square1.txt*:

3

4 9 2

3 5 7

8 1 6

Here is the output to cout:

Magic square

4 9 2

3 5 7

8 1 6

Valid magic square

*Sample run 2 (missing file)*

Contents of cin:

invalidfile.txt

Here is the output to cout:

File "invalidfile.txt" could not be opened

*Sample run 3 (invalid magic square)*

Contents of cin:

square2.txt

Contents of *square2.txt*:

3

2 9 4

3 5 7

8 1 6

Here is the output to cout:

Magic square

2 9 4

3 5 7

8 1 6

Invalid magic square

*Expected output*

There are 12 tests. Some of the tests will check the output to make sure it is correct. There are also unit tests that call the readSquare and validateSquare functions.

You will get yellow highlighted text when you run the tests if your output is not what is expected. This can be because you are not getting the correct result. It could also be because your formatting does not match what is required. The checking that zyBooks does is very exacting and you must match it exactly. More information about what the yellow highlighting means can be found in course "How to use zyBooks" - especially section "1.4 zyLab basics".

Finally, do not include a system("pause"); statement in your program. This will cause your verification steps to fail.

*Note:* that the system("pause"); command runs the pause command on the computer where the program is running. The pause command is a Windows command. Your program will be run on a server in the cloud. The cloud server may be running a different operating system (such as Linux).

*Error message "Could not find main function"*

Now that we are using functions some of the tests are unit tests. In the unit tests the zyBooks environment will call one or more of your functions directly.

To do this it has to find your main function.

Right now zyBooks has a problem with this when your int main() statement has a comment on it.

For example:

If your main looks as follows:

int main() // main function

You will get an error message:

Could not find main function

You need to change your code to:

// main function

int main()

If you do not make this change you will continue to fail the unit tests.