p2B: My Magic Square

Due Oct 5 by 11:59pm **Points** 60 **Submitting** a file upload
Available Sep 23 at 12am - Oct 6 at 12:33pm 14 days

This assignment was locked Oct 6 at 12:33pm.

Learning GOALS

The purpose of this assignment is to continue practice writing C programs and gain more experience working in C, a low-level, non-object oriented language. After completing both parts A and B of this project, you should be comfortable with pointers, arrays, address arithmetic, structures, command-line arguments, and file I/O in C. For part B, you will also be working with structures and file I/O.

OVERVIEW

For this assignment you will be writing the program **myMagicSquare.c,** which generates a magic square of a specified size and writes it to an output file. A magic square is a matrix of size *n* x *n* with positive numbers from 1… n^2 arranged such that the numbers in any line (horizontal, vertical, and both main diagonals) sum to the *same* values. You can read more about magic squares in https://en.wikipedia.org/wiki/Magic_square (https://en.wikipedia.org/wiki/Magic_square)

Like p2A, this project requires you to work with a dynamically allocated 2D array (heap allocation), and you are **not allowed** to use indexing to access the array that is used to represent the matrix. Instead, you are required to use address arithmetic and dereferencing to access it. Submitting a solution using only indexing to access the matrix will result in a **50% reduction of your score**. You may use indexing to access any other arrays that might be used by your program.

You're welcome to develop the solution in two phases. First, code a solution that uses indexing. Once you have that solution working, you can copy and edit to replace the indexing syntax with pointer arithmetic before final testing and submission. You are strongly encouraged to use incremental development to code your solution rather than coding the entire solution followed by debugging that entire code. Incremental development adds code in small increments. After each increment is added, you test the new code to ensure that it works as desired before adding the next increment of code. Bugs are then easier to find since they're more likely to be in the new code increment rather than the code you've already tested.

SPECIFICATIONS

You are to develop your solution using the skeleton code in the file **myMagicSquare.c** found on the CS Linux computers at:

```
/p/course/cs354-deppeler/public/code/p2B/myMagicSquare.c
```

The program **myMagicSquare.c** is run as follows:

```
./myMagicSquare <output_filename>
```

Where <output_filename> is the name of the file where the generated magic square is to be written. The format of the output file will be as follows (also see example run below):

- The first line will contain a positive integer, **n**, that is the size of the matrix.
- Every line after that represents a row in the matrix, starting with the first row. There will be **n** such lines where each line has **n** numbers (columns) separated by commas.
- Every number in the matrix will be unique.

The program prompts and reads input from **stdin** to get the size of the magic square to generate. The program only generates magic squares of odd dimensions like 3x3 or 5x5. You may assume that the user will input an integer. However, the program must check that the input is an odd number greater than or equal to 3, and if not, print the appropriate message and exit as shown in the sample run below.

Siamese Method

Read about the **Siamese method** (https://en.wikipedia.org/wiki/Siamese_method) (← **read the link**) to generate the square matrix of an odd size with the numbers 1… n^2.

Start by placing 1 at the center column of the topmost row in the square, and then for every number till n^2

- Move diagonally up-right, by one row and column, and place the next number in that position. Wrap around to the first column/last row if the move takes you out of the square.
- If the next position is already filled with a number then place it one row **below** the current position.

Alternate Siamese Method

Alternatively, and to make the modular arithmetic easier, you can consider the reflection variant of the above Siamese algorithm by **starting from the central row in the last column** and then moving in a down-right diagonal fashion. Then, follow the pattern described above.

Finally, the program must write the alternate magic square to the output filename specified in the command line.

The sample runs below show the program's expected behavior:

```
[deppeler@liederkranz] (88)$ ./myMagicSquare
Usage: ./myMagicSquare <output_filename>
[deppeler@liederkranz] (89)$ ./myMagicSquare magicSquare1.txt
Enter magic square's size (odd integer >=3)
1
Size must be >= 3.
[deppeler@liederkranz] (90)$ ./myMagicSquare magicSquare2.txt
Enter magic square's size (odd integer >=3)
2
Size must be odd.
[deppeler@liederkranz] (91)$ ./myMagicSquare magicSquare3.txt
Enter magic square's size (odd integer >=3)
3
[deppeler@liederkranz] (92)$ cat magicSquare3.txt
3
4,3,8
9,5,1
2,7,6
```

HINTS

Using library functions is something you will do a lot when writing programs in C. Each library function is fully specified in a manual page. The man command is very useful for learning the parameters a library function takes, its return value, detailed description, etc. For example, to view the manual page for **fopen**, you would issue the command "**man fopen**". If you are having trouble using **man**, the same manual pages are also available online. You will need these library functions to write this program. You do not need to use all of these functions since a couple of them are just different ways to do the same thing. We encourage you to refer to our code in the p2A code skeleton for an example of file I/O.

- **fopen()** to open the file. Make sure you specify the correct mode (read or write) for which you are opening the file.
- **malloc()** to allocate memory on the heap
- **free()** to free up any dynamically allocated memory
- **fgets()** to read each input from a file. fgets can be used to read input from the console as well, in which case the file is stdin, which does not need to be opened or closed. An issue you need to consider is the size of the buffer. Choose a buffer that is reasonably large enough for the input.
- **fscanf()/scanf():** Instead of fgets() you can also use the fscanf()/scanf() to read input from a file or stdin. Since this allows you to read formatted input you might not need to use strtok() to parse the input.
- **fclose()** to close the file when done.
- **printf()** to display results to the screen.
- **fprintf()** to write to a file.
- **atoi()** to convert the input which is read in as a C string into an integer
- **strtok()** to tokenize a string on some delimiter. In this program the input file for a square has every row represented as values delimited by a comma. See here

(http://www.tutorialspoint.com/c_standard_library/c_function_strtok.htm) for an example on how to use strtok to tokenize a string.

REQUIREMENTS

- Your program must dynamically allocate (i.e., on the heap) the matrix structure and matrix array.
- Your program must use address arithmetic and dereferencing to access the array representing the matrix.
- Your program must use the **->** operator when accessing data members of a structure via a pointer variable.
- Your programs should operate exactly as the sample run above.
- Your program must print error messages exactly as shown in the sample run above, and then call exit(1).
- Your program must check the return values for errors of the library functions, malloc(), fopen(), and fclose(). Handle errors by displaying an appropriate error message and then calling exit(1).
- Your program must properly free up all dynamically allocated memory at the end of the program.
- Your program must close any opened files when it's done using them.
- Your program must follow style guidelines as given in the **Style Guide**.
- Your program must follow commenting guidelines as given in the **Commenting Guide**.
- We will compile your programs with **gcc -Wall -m32 -std=gnu99** on the Linux lab machines. So your program must compile there, and without warnings or errors.

SUBMITTING & VERIFYING

SUBMISSION FOR p2B HAS BEEN ENABLED.

Leave plenty of time before the deadline to complete the two steps for submission found below. There is a 33 minute grace period after the deadline for last minute emergencies. Submitting during this grace period results in your submission being marked late but it will be accepted for grading without penalty. No submissions or updates to submissions are accepted after this grace period.

1.) Submit only the file listed below under Project p2B in Assignments on Canvas. Do not zip, compress, or submit your file in a folder.

- myMagicSquare**.c**

Repeated Submission: You may resubmit your work repeatedly so we strongly encourage you to use Canvas to store a backup of your current work. If you resubmit, Canvas will modify your file names by appending a hyphen and a number (e.g., **myMagicSquare**-1.c).

2.) Verify your submission to ensure it is complete and correct. If not, resubmit all of your work rather than updating just some of the files.

- **Make sure you have submitted all the files listed above.** Forgetting to submit or not submitting one or more of the listed files will result in you losing credit for the assignment.

- **Make sure the files that you have submitted have the correct contents.** Submitting the wrong version of your files, empty files, skeleton files, executable files, corrupted files, or other wrong files will result in you losing credit for the assignment.
- **Make sure your file names exactly match those listed above.** If you resubmit your work, Canvas will modify your file names as mentioned in **Repeated Submission** above. These Canvas modified names are accepted for grading.

Project p2B

Criteria	Ratings				Pts
1. Compiles without warnings or errors	**6.0 pts** **No warnings or errors**	**3.0 pts** **Incorrect output format** Please refer to the project page on Canvas	**0.0 pts** **One or more errors or at least 5 warnings**		6.0 pts
2. Follows commenting and style guidelines	**6.0 to >0.0 pts** **Followed**		**0.0 pts** **Not followed**		6.0 pts
3. Implements CLA checking	**3.0 pts** **Meets specifications**	**2.0 pts** **Minor problem** Error message does not match specification	**1.0 pts** **Major problem** Incorrect argc check, no error message displayed, or does not exit	**0.0 pts** **Does not check CLAs**	3.0 pts
4. Checks return values of malloc() and fopen()	**6.0 pts** **Checks all**	**3.0 pts** **Checks some**	**0.0 pts** **Checks none**		6.0 pts
5. Closes all opened files - fclose()	**3.0 pts** **Closed**	**2.0 pts** **Some closed**	**0.0 pts** **None closed**		3.0 pts
Execution test: frees heap memory	**6.0 pts** **Freed**	**0.0 pts** **Not all freed**			6.0 pts
Execution test: size = 3	**5.0 pts** **Correct result**	**0.0 pts** **Incorrect result**			5.0 pts
Execution test: size = 7	**5.0 pts** **Correct result**	**0.0 pts** **Incorrect result**			5.0 pts
Execution test: size = 11	**5.0 pts** **Correct result**	**0.0 pts** **Incorrect result**			5.0 pts
Execution test: size = 13	**5.0 pts** **Correct result**	**0.0 pts** **Incorrect result**			5.0 pts

Criteria	Ratings		Pts
Execution test: size = 17	**5.0 pts** **Correct result**	**0.0 pts** **Incorrect result**	5.0 pts
Execution test: size = 19	**5.0 pts** **Correct result**	**0.0 pts** **Incorrect result**	5.0 pts
			Total Points: 60.0