p2: Magic Square

1. Logistics

1. This project must be done individually.

   It is academic misconduct to share your work with others in any form including posting it on publicly accessible web sites, such as GitHub.

   It is academic misconduct for you to copy or use some or all of a program that has been written by someone else.

2. All work for this project is to be done on one of the CS department's instructional Linux machines. You're welcome to remotely login using ssh or MobaXterm as was shown in lecture (or any other way that you figure out on your own).
3. All projects in this course are graded on CS department's instructional Linux machines. It is your responsibility to make sure that your code runs as you expect on these machines.

2. Learning Goals

The purpose of this assignment is to get comfortable writing C programs, gaining the experience of working in a non-object oriented language. By the end of this assignment you should be comfortable with arrays, command-line arguments, file I/O, pointers, and structures in C.

3. Specifications

A magic square is a square matrix of size *n* x *n* with positive numbers from 1… n^2 arranged such that the sum of the numbers in any horizontal, vertical, or diagonal (primary and secondary) line is always the same number. You can read more about it in this reference (Links to an external site.)Links to an external site..

For this assignment you will be writing two programs, **verify_magic.c** and **generate_magic.c** to verify and generate a magic square respectively. Both the programs will require you to work with dynamically allocated 2D arrays.

The caveat in this assignment is that while accessing the 2D array of the magic square you are **not allowed** to use square brackets, like array**[i][j]** that you would normally use to index an array. Instead you are required to use pointer arithmetic and dereferences to traverse and index the 2D array. This is so that you can get comfortable with using pointers.

Using square brackets to index the 2D square will result in a **deduction of points**.

3.1. Verify a magic square

You have been provided skeleton code for the program **verify_magic.c** at the location

```
/p/course/cs354-skrentny/public/code/p2/verify_magic.c
```

The program **verify_magic.c** will be run as follows:

```
./verify_magic <filename>
```

Where <filename> is the name of the file that contains the square that needs to be verified. The format of the file will be as follows:

- The first line will be a positive integer *n* for the number of rows/columns of the square. So the dimensions of the square are *n* x *n*.
- Every line after that represents a row in the magic square, starting with the first row. There will be *n* such lines where each line has *n* numbers(columns) delimited by a comma.

So for instance a file containing a magic square of size 3 will look as follows:

```
3
4,3,8
9,5,1
2,7,6
```

A sample input file is provided at:

```
/p/course/cs354-skrentny/public/code/p2/magic-3.txt
```

The program should read and parse the input file to construct a magic square using the struct provided in the skeleton code. You will need to dynamically allocate memory for the 2D array. Check section 5 to see how to parse the input.

To verify the square is indeed a magic square all you need to do is sum up all the rows, columns, and the two main diagonals (top-left to bottom-right, and top-right to bottom-left) and check that they should sum to the same number. **The program should print true if the input is a magic square, and false otherwise** (keywords like right/wrong, valid/not valid, etc. should not be used).

The sample run below shows the expected behavior of the program:

```
[haseeb@espresso] (76)$ ./verify_magic
Usage: ./verify_magic <filename>
[haseeb@espresso] (77)$ cat magic-4.txt
4
1,15,14,4
12,6,7,9
8,10,11,5
13,3,2,16
[haseeb@espresso] (78)$ ./verify_magic magic-4.txt
true
```

```
[haseeb@espresso] (79)$ cat not-magic-3.txt
3
4,3,8
9,1,5
2,7,6
[haseeb@espresso] (80)$ ./verify_magic not-magic-3.txt
false
[haseeb@espresso] (81)$
```

3.2. Generate a magic square

The second program **generate_magic.c** should generate a magic square of a specified size and write the output to a file. Copy the skeleton code from the location:

```
/p/course/cs354-skrentny/public/code/p2/generate_magic.c
```

The program will be run as follows

```
./generate_magic <output-filename>
```

Where output-filename is the name of the file to write the output to.

The program should prompt and read user input from stdin which specifies the dimensions of the magic square to generate. The program will only generate magic squares of odd dimensions like 3x3 or 5x5. You can assume that the user input will be an integer however the program should check if the input is an odd number greater than or equal to 3, and if not, print an appropriate message and exit.

You will be using the Siamese method (Links to an external site.)Links to an external site.(**read this 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. 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.

Remember you are not allowed to index the 2D array directly and should use pointers to access the elements in the array.

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

Finally the program should create the output file and write the magic square to the file in the same format as shown in section 3.1.

The sample run below shows the expected behaviour of the program.

```
[haseeb@espresso] (89)$ ./generate_magic
Usage: ./generate_magic <filename>
[haseeb@espresso] (90)$ ./generate_magic out-magic-3.txt
Enter size of magic square, must be odd
1
Size must be an odd number >= 3.
[haseeb@espresso] (91)$ ./generate_magic out-magic-3.txt
Enter size of magic square, must be odd
3
[haseeb@espresso] (92)$ cat out-magic-3.txt
3
4,3,8
9,5,1
2,7,6
[haseeb@espresso] (93)$
```

4. Error Handling

- If the user invokes the either of the two programs incorrectly (for example, without an argument, or with two or more arguments), the program should print an error message and call exit(1) as shown in the sample runs above.

- **Be sure to always check the return value of library functions**. For example, if a file cannot be opened, then the program should not read input. Instead it should print an error message as shown below. Similarly for other functions like malloc() make sure you check the return values for errors and handle them in a similar manner. **Points will be deducted for forgetting to check return values from library functions**.

```
[haseeb@espresso] (45)$ ./verify_magic non-existent-file.txt
Cannot open file for reading.
[haseeb@espresso] (46)$
```

- Make sure to close up any opened files when you are done with them.
- Make sure to free up all dynamically allocated memory at the end of the program. For the 2D array you would need to free up all allocated memory by freeing it in the reverse order of how you allocated it, i.e don’t just free the pointer to the array of arrays. Failure to free up dynamically allocated memory will result in a **deduction of points**.

5. Notes and 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. Note, you may not need to use all of these functions since a couple of them are just different ways to do the same thing.

- **fopen()** to open 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 the integers 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 columns delimited by a comma. See here (Links to an external site.)Links to an external site. for an example on how to use strtok to tokenize a string.

When opening a file for reading using fopen, make sure you specify the correct mode (read or write) for which you are opening the file.

6. Requirements

- Your program must follow style guidelines as given in Style Guidelines. The guide is from CS302 for Java, but most of it is applicable for C as well. The relevant sections are braces, whitespace and indentation.
- Include a comment at the top of each source code file with your **name and section**. You must comment every function with a header comment. See the Commenting Guide, where applicable for C.
- Your programs should operate exactly as the sample outputs shown above.
- We will compile each of your programs with

  ```
  gcc -Wall -m32 -std=gnu99
  ```

  on the Linux lab machines. So, your programs must compile there, and **without warnings or errors**. It is your responsibility to ensure that your programs compile on the department Linux machines, and **points will be deducted** for any warnings or errors.

- Remember to do error handling in all your programs. See the instructions on error handling for more details.

7. Submitting Your Work

Submit the following **source files** under Project 2 in Assignments on Canvas on or before the deadline:

1. verify_magic.c
2. generate_magic.c

It is your responsibility to ensure your submission is complete with the correct file names having the correct contents. The following points will seem obvious to most, but we've found we must explicitly state them otherwise some students will request special treatment for their carelessness:

- **You will only receive credit for the files that you submit.** You will not receive credit for files that you do not submit. Forgetting to submit, not submitting all the listed files, or submitting executable files or other wrong files will result in you losing credit for the assignment.
- **Do not zip, compress, submit your files in a folder, or submit each file individually.** Submit only the text files as listed as a single submission.
- **Make sure your file names exactly match those listed.** If you resubmit your work, Canvas will modify the file names by appending a hyphen and a number (e.g., verify_magic-1.c) and these Canvas modified names are accepted for grading.

Repeated Submission: You may resubmit your work repeatedly until the deadline has passed. **We strongly encourage you to use Canvas as a place to store a back up copy of your work.** If you resubmit, you must resubmit all of your work rather than updating just some of the files.