p2A: Sudoku Basic

Published

GOALS **OVERVIEW** **SPECS** **HINTS** **REQUIREMENTS** **SUBMITTING**

Announcements:

- 9/15 Released - Read Today and review the data files and check_board.c starter source code file we provide.

Learning GOALS

- able to keep a formal written log of progress on a project
- gain experience writing C programs
- gain experience solving problems using a lower-level, non-object oriented language
- able to create and use pointers and dynamically created 2D arrays on the heap
- use address arithmetic to access array elements for 1D and 2D arrays on stack or heap
- read data from a file into your program
- able to trace and explain code that reads lines from a file
- able to trace and explain code that parses (interprets) lines read from a file

For this project Part A, your focus will be on reading a data file, pointers, creating and filling arrays, and using address arithmetic to access elements of the array.

Continue documenting your work sessions in your **Work Log (https://canvas.wisc.edu/courses/359992/pages/work-log?wrap=1)** and when finished submit a downloaded **log-p2a.pdf** with your project files.

OVERVIEW

Sudoku is a puzzle where the player solves a puzzle by placing the digits 1-9 in cell locations such that there are no duplicate digits in any row, column, or box (sub-array) (**see Sudoku Wikipedia (https://en.wikipedia.org/wiki/Sudoku)**). The puzzle starts with a partially filled board and the player places values that do not violate any of the board rules until the board is solved. There are many online web and mobile apps where you can try this puzzle for yourself. There are also many coded solutions available which are not allowed for you to search, view, or use for reference in any way.

Why solve such a solved problem as a beginning C programmer? Simply because such programs are good practice for you to build your C program development skills. This is similar to a carpenter building their own toolbox as one of their early carpentry projects. It is not the solution that has value, rather it is the practice and skills that you gain as you develop your solution that are critical. To help ensure that students do gain skills they will need later in this course and in other program development work, we have specific requirements that you must follow for each assignment. Failure to understand

and follow any assignment's requirements will certainly lose points for you, and in some cases you may lose 1/2 or more of the available points for this assignment. Be sure to read this and all assignments carefully to understand what is allowed and what is required.

For this assignment, you'll be completing the **C** program in **check_board.c**, which processes a text file that contains a current puzzle state, represented as a 2D grid of digits 1 to n, where n is the size of the board and is an integer in the range 1 to 9. **Your task in p2A is to verify if the contents of a file represent a valid current state for each row and each column of the board. If the file exists your program must display valid or invalid and then exit with return 0 indicating the program worked.**

The puzzle size will be generalized to use a grid of **n** rows and **n** columns, where n is an integer from 1 to 9. This will require you to work with a dynamically allocated 2D array (heap allocation). The first value in the input file will be the value of the board size. Note the total number of digits on the puzzle board will be n*n.

A key objective of this assignment is for you to practice using pointers. To achieve this, **you are not allowed to use indexing (square brackets[ ]) to access any array elements** used in your code. Instead, you are required to use address arithmetic and dereferencing to access all elements of all arrays. Submitting a solution **using only indexing brackets to access array elements will result in a 50% reduction of your score**. Submitting a solution **using indexing in some but not all access of array elements will result in a 10-50% deduction**.

You're welcome to develop the solution in phases. You could first code a solution that uses indexing. Once you have that solution working, you can replace all indexing with pointer arithmetic before final testing and submission. If you do use this approach, make sure to replace all accesses to array elements to use address arithmetic and dereferencing to avoid a penalty of -50% of all points. This means there is a 30pt penalty for using brackets [ ] instead of pointer address arithmetic and notation. DO NOT USE [] for your array accesses on this program.

You're 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 line or few lines of code are added, you test it to ensure it works as desired before adding the next increment of code. Bugs in your code are easier to find since they're more likely to be in the few lines of new code rather than the code you've already tested.

SPECIFICATIONS

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

```
/p/course/cs354-deppeler/public/code/p2A/check_board.c
```

The skeleton has several functions some of which have been completed or partially completed. You may add your own functions if you wish. You must dynamically allocate memory to hold a 2D array for given board size.

CAUTION: Do NOT create a 1D array of ints on the stack or heap for this work. Your 2D array must be a dyamically allocated 1D array of pointers to dynamically allocated 1D arrays of ints

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

```
./check_board <input_filename>
```

Where <input_filename> is the name of the file that contains the data representing the current state of the sudoku board. The format of the input file is as follows:

- The first line contains one positive integer *n* for the number of rows and columns of the board. The dimensions of the board are *n* x *n*. Correct sudoku board size is 9, but our boards may be any size from 1x1 to 9x9. (e.g. a 5x5 board would need the integers 1-5 in each row and column).
- Every line after the board size represents a row in the board, starting with the first row. You may assume there will be *n* such lines where each line has *n* numbers (columns) separated by commas.
- Each number in the board is either 0, or 1-n, where n is the size of the board. If the integer is 1-n, then it denotes the presence of a placed digit at that position. The integer zero, 0, represents an unmarked space. You must check that each integer is valid for the given board size. You may assume that if the first line is a valid board size 1-9, then the number of rows and the number of columns is correct for that board file.

For instance, the following example shows the file format that represents a valid 9 x 9 puzzle (n=9) and its solution board:

```
9
7,0,4,0,0,0,8,0,0
0,9,0,0,8,7,6,0,0
0,0,0,0,4,0,7,0,1
3,0,0,2,0,0,0,6,7
0,2,0,6,0,3,0,4,0
6,4,0,0,0,9,0,0,5
1,0,6,0,3,0,0,0,0
0,0,9,1,6,0,0,8,0
0,0,2,0,0,0,4,0,6

that corresponds to the following puzzle:
```

```
7|_|4|_|_|_|8|_|_
_|9|_|_|8|7|6|_|_
_|_|_|_|4|_|7|_|1
3|_|_|2|_|_|_|6|7
_|2|_|6|_|3|_|4|_
6|4|_|_|_|9|_|_|5
1|_|6|_|3|_|_|_|_
_|_|9|1|6|_|_|8|_
_|_|2|_|_|_|4|_|6

The solution for above puzzle:

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

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

Sample input files, named **boardN.txt** (and a couple **boardNsolution.txt** files) are provided in the p2A starter directory as shown in the example below:

```
    /p/course/cs354-deppeler/public/code/p2A/board1.txt
```

These test files are meant to help you start testing your program. They are not exhaustive tests for your program. You will want to create your own board files to test your program fully. We do not publish test files that are used to evaluate your program's correctness.

We've provided code in the skeleton that reads and parses the input file, which you'll use to construct a dynamically allocated 2D array representing the board.

If the file exists and can be read, the program must print either valid or invalid followed by a newline (only these two outputs in lowercase will be accepted). Print valid only if the input file contains a valid board configuration, otherwise print invalid. To determine if the input board configuration is valid, you'll need to iterate over each row and each column.

A valid board for p2A:

- has a size **n**, where n is an integer in the range 1 to 9, inclusive;
- has no duplicate integers (1-n) in any row, duplicate 0's mean multiple blank locations and this is allowed.
- has no duplicate integers (1-n) in any column, duplicate 0's mean multiple blank locations and this is allowed.
- is either not complete (not solved yet), solved, or it is invalid because some row or column contains duplicates (that are not 0's)
- Note: if you are familiar with Sudoku, you know that sub-arrays or "box areas" must be checked also.
  DO NOT CHECK for valid sub-array "boxes" for p2A.

The sample runs below shows the expected behavior of the program for the boards shown:

```
[deppeler@liederkranz] (33)$ ./check_board
Usage: ./check_board <input_filename>
[deppeler@liederkranz] (34)$ cat board1.txt
9
7,0,4,0,0,0,8,0,0
0,9,0.0,8,7,6,0,0
0,0,0,0,4,0,7,0,1
3,0,0,2,0,0,0,6,7
0,2,0,6,0,3,0,4,0
6,4,0,0,0,9,0,0,5
1,0,6,0,3,0,0,0,0
```

```
0,0,9,1,6,0,0,8,0
0,0,2,0,0,0,4,0,6
[deppeler@liederkranz]] (35)$ ./check_board board1.txt
valid
[deppeler@liederkranz]] (36)$ ./check_board board1solution.txt
valid
[deppeler@liederkranz] (37)$ cat board4.txt
9
0,0,0,0,4,0,0,0,2
1,0,0,0,0,9,4,8,0
0,4,0,8,0,3,0,0,0
0,0,2,0,1,4,5,0,0
7,6,0,0,0,0,0,2,4
0,0,9,2,7,0,1,0,0
0,0,0,3,0,1,0,7,0
0,9,3,5,0,0,0,0,1
5,0,0,0,6,4,0,0,0
[deppeler@liederkranz]] (38)$ ./check_board board4.txt
invalid
```

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 some of these library functions to write this program and will see that some of them are already used in our code. You do 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. **DO USE fgets()** **DO NOT USE gets()**
- **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 output 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 character. In this program the input file for a square has every row represented as columns 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.

When you're ready or needing to step through your code, you're ready to try the built-in **GDB debugger** tool. Add the **-g** option to your build (compile) command to make it possible to use **gdb** with your executable.

1. `gcc check_board.c -g -m32 -std=gnu99 -o check_board`
2. **`gdb ./check_board`**
3. See GDB Tutorial YouTube Video (https://canvas.wisc.edu/courses/359992/pages/gdb-tutorial-youtube-video) for more information.

REQUIREMENTS

- Your program must dynamically allocate a 2D array to store the puzzle board.
- Your program must use address arithmetic and dereferencing to access the array elements.
- Your program must print an error message, as shown in the sample runs above, and then call exit(1) if the user invokes the program without any arguments, or with two or more arguments.
- If invoked correctly, your program must operate as specified and only provide **valid** or **invalid** as output.
  - There must be no other output.
  - Tip: if any action causes an error return or other, then the output is invalid.
  - Your program must check the return values for all calls to any library functions, malloc(), fopen(), and fclose(). Handle errors by printing an error message and exiting.
- Your program must follow style guidelines as given in the Style Guide (https://canvas.wisc.edu/courses/359992/pages/programming-style-guide).
- Your program must follow commenting guidelines as given in the Commenting Guide (https://canvas.wisc.edu/courses/359992/pages/program-commenting-guide).
- We will compile your programs with **gcc -Wall -m32 -std=gnu99** on the Linux lab machines. So, your programs must compile there, and without warnings or errors.
- Your program must properly free up all dynamically allocated memory.
  - We will use **valgrind** to check for memory leaks and ensure that your programs shows no leaks or faults.
  - See **man valgrind** for how to use this tool to check for memory leaks in your executable programs.
    - **valgrind --leak-check=full --error-exitcode=1 ./check_board board.txt**

SUBMITTING & VERIFYING

SUBMISSION FOR p2A HAS BEEN ENABLED.

Leave plenty of time before the deadline to complete the two steps for submission found below. Work that is not submitted prior to the DUE Date and Time but is submitted before the availability date and time is marked LATE by Canvas. LATE work will be graded, but and it will incur the LATE penalty.

No submissions or updates to submissions are accepted after the assignment's availability period has passed.

1.) Submit only the files listed below under Project p2A in Assignments on Canvas. Do not zip, compress, or submit your file in a folder.

- **check_board.c**
- **log-p2a.pdf (must be an accurate reflection of your work on p2A)**

Repeated Submission: You may resubmit your work repeatedly so we strongly encourage you to use Canvas to store a backup of your current work.

Note: If you resubmit a file with the same name as a previous submission, Canvas will modify your file name by appending a hyphen and a number (e.g., **check_board-1.c**). This is expected and does not cause any problems for us or any penalty for you. The files you submit files may not have -1 when selected for uploading to Canvas, your file(s) must always be named as shown.

2.) Verify your submission to ensure it is complete and correct. If it is not complete or correct, you must 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.
  - To check your submitted files are correct:
    - you must go to your submission and download the files
    - and examine the contents. It is not enough to see the correct filename in your submission folder.
- **Make sure your file names, when selecting for upload, exactly match those listed above.**

Points	60
Submitting	a file upload
File Types	c and pdf

Due	For	Available from	Until
Sep 29, 2023	Everyone	Sep 15, 2023 at 12am	Oct 1, 2023 at 11:59pm

Project p2A sudoku (1)

Criteria	Ratings				Pts
1. Compiles without warnings or errors	**6 pts** **No warnings or errors**	**0 pts** **One or more errors or at least 5 warnings**			6 pts
2. Follows commenting and style guidelines	**6 to >0.0 pts** **Followed**	**0 pts** **Not followed**			6 pts
3. Implements CLA checking	**3 pts** **Meets specifications**	**2 pts** **Minor problem** Error message does not match specification	**1 pts** **Major problem** Incorrect argc check, no error message displayed, or does not exit	**0 pts** **Does not check CLAs**	3 pts
4. Checks return values of malloc() and fopen()	**6 pts** **Checks all**	**3 pts** **Checks some**	**0 pts** **Checks none**		6 pts
5. Closes all opened files - fclose()	**3 pts** **Closed**	**2 pts** **Some closed**	**0 pts** **None closed**		3 pts
Execution test: frees heap memory	**6 pts** **Freed**	**6 to >0 pts** **Not all freed**			6 pts
Execution test: board1.txt (provided)	**2 pts** **Correct result**	**0 pts** **Incorrect result**			2 pts
Execution test: board2.txt (provided)	**2 pts** **Correct result**	**0 pts** **Incorrect result**			2 pts
Execution test: board3.txt (provided)	**2 pts** **Correct result**	**0 pts** **Incorrect result**			2 pts
Execution test: board4.txt (provided)	**2 pts** **Correct result**	**0 pts** **Incorrect result**			2 pts

Criteria	Ratings		Pts
Execution test: valid1.txt (incomplete valid board)	**2 pts** **Displays valid**	**0 pts** **Incorrect result**	2 pts
Execution test: valid2.txt (incomplete valid board)	**2 pts** **Displays valid**	**0 pts** **Incorrect result**	2 pts
Execution test: valid3.txt (complete valid board)	**1 pts** **Displays valid**	**0 pts** **Incorrect result**	1 pts
Execution test: valid4.txt (complete valid board)	**1 pts** **Displays valid**	**0 pts** **Incorrect result**	1 pts
Execution test: valid5.txt (partial complete valid board)	**1 pts** **Displays valid**	**0 pts** **Incorrect result**	1 pts
Execution test: valid6.txt (partial complete valid board)	**1 pts** **Displays valid**	**0 pts** **Incorrect result**	1 pts
Execution test: valid7.txt (complete valid board)	**1 pts** **Displays valid**	**0 pts** **Incorrect result**	1 pts
Execution test: valid8.txt (complete valid board)	**1 pts** **Displays valid**	**0 pts** **Incorrect result**	1 pts
Execution test: invalid1.txt (invalid board size)	**1 pts** **Displays invalid**	**0 pts** **Incorrect result**	1 pts
Execution test: invalid2.txt (row has duplicates)	**1 pts** **Displays invalid**	**0 pts** **Incorrect result**	1 pts
Execution test: invalid3.txt (border row has duplicates)	**1 pts** **Displays invalid**	**0 pts** **Incorrect result**	1 pts

Criteria	Ratings				Pts
Execution test: invalid4.txt (column has duplicates)	**1 pts** **Displays invalid**		**0 pts** **Incorrect result**		1 pts
Execution test: invalid5.txt (border column has duplicates)	**1 pts** **Displays invalid**		**0 pts** **Incorrect result**		1 pts
Execution test: invalid6.txt (corners are duplicates)	**1 pts** **Displays invalid**		**0 pts** **Incorrect result**		1 pts
Execution test: invalid7.txt (interior have duplicates)	**1 pts** **Displays invalid**		**0 pts** **Incorrect result**		1 pts
Maintain Project Work Log Record Able to communicate via a written document, the work completed on a project. The details include what work was done, when, where, and with whom, the work was completed. threshold: 5.0 pts	**5 pts** **Meets Expectations**	**4 pts** **wrong filename**	**3 pts** **Insufficient detail**	**0 pts** **Incorrect or no submission.**	5 pts
					Total Points: 60