ENEE 140 Project 3: A Game of Sudoku

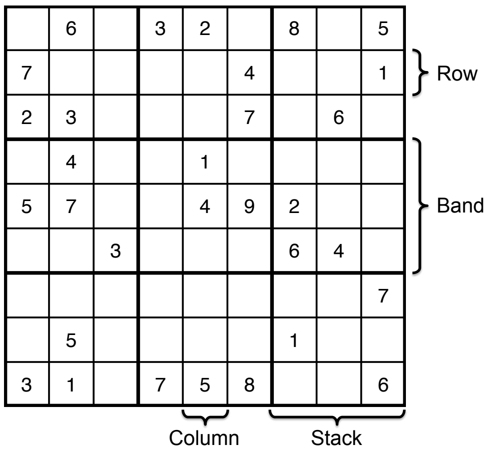
Posted: Tuesday, April 12, 2016
Due: Monday, May 09, 2016 at 11:59 pm

Project objectives

1. Learn how to use the random numbers in a complex program.
2. Cement your understanding of arrays and two-dimensional arrays.
3. Practice bitwise operations.
4. Master the use of file I/O.
5. Learn how to manipulate command line parameters.
6. Improve your programming skills and the ability to write correct and maintainable code.
7. Understand how to program games based on mathematical concepts.

Project description

In this project, you will write a program for generating Sudoku boards. In Sudoku, the player is presented with a 9×9 *board*, where some of the 81 *cells* are filled with *digits* (numbers between 1–9) and other cells are left *blank*. The player must fill the blank cells with digits so that each *column*, each *row*, and each of the nine 3×3 *boxes* that compose the board contains all of the digits from 1 to 9. In other words, no digit must be repeated in any row, column or box. Three boxes side-by-side make up a *band* and three boxes on top of each other make up a *stack*. The figure below illustrates a Sudoku board (note that the boxes have thicker borders):



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

Program structure

The functionality of the program will be implemented in three files. `enee140_sudoku.h` is a header file that includes function prototypes for your Sudoku library. `enee140_sudoku.c` is a C file where you define (implement) these functions, as well as any other helper functions you may find useful. `enee140_gen_sudoku.c` is the C file that includes the `main()` function of your program. In the `main()` function, you will read the arguments provided by the user on the command line and you will invoke (call) the functions declared in `enee140_sudoku.h` to provide the functionality requested. These files will be compiled together in an executable file called `enee140_gen_sudoku`.

Command line arguments

In this project, you will write a program that can generate a Sudoku board, save it to a file, read a previously saved board from a file, and print the board to the terminal in a human-friendly manner. This program will receive the file name and a parameter encoding the operations to be performed as arguments provided on the command line.

Note: This program should not print a menu. The user will choose different options through command-line arguments.

1 Read command-line arguments

Your first task is to read the command-line arguments provided by the user. The program will be launched as follows:

```
enee140_gen_sudoku filename options
```

where

- `filename` is the name of a file. You will use this file to save the board, to read the board or both, depending on the operations encoded in the next parameter.
- `options` is a positive integer that specifies the operations to be performed. You must interpret this parameter as described in Step 2.

2 Interpret the `options` argument

Your second task is to determine the operations requested by the user and to invoke the corresponding functions from your Sudoku library. The user options will be encoded in the binary representation of the `options` argument. Recall that, on the GRACE computers, unsigned integers are represented using 32 bits (b_0, b_1, ... b_{31}). Each bit can be either 0 or 1, and the value of the unsigned integer is computed using the following formula: $U = \sum_{i=0}^{31} = b_i \cdot 2^i$.

The values of the *least significant* 9 bits (b_0–b_8) in the binary representation of the `options` argument indicate the operations requested, as follows:

b_0: Read or generate the board. If this bit is 0, generate a new board by invoking the function from Step 4 (the `blanks` parameter should be a randomly generated number between 31 and

61). If this bit is 1, read a board from the file indicated by `filename` by invoking the function from Step 12, then check its validity by invoking the function from Step 3. If the file does not exist or cannot be read, if it does not contain a Sudoku board or if the Sudoku board therein is not valid, print an appropriate error message and exit the program.

b_1: If this bit is 1, permute rows within bands by invoking the function from Step 5.

b_2: If this bit is 1, permute the bands of the board by invoking the function from Step 6.

b_3: If this bit is 1, permute columns within stacks by invoking the function from Step 7.

b_4: If this bit is 1, permute the stacks of the board by invoking the function from Step 8.

b_5: If this bit is 1, flip the board along main diagonal by invoking the function from Step 9.

b_6: If this bit is 1, flip the board along minor diagonal by invoking the function from Step 10.

b_7: If this bit is 1, print board in a human-friendly manner to the standard output by invoking the function from Step 11.

b_8: If this bit is 1, save the board to the file indicated by `filename` by invoking the function from Step 13.

Note that several operations may be requested. For example, if `options` is 453 (111000101 in binary), you must read the board from `filename` ($b_0 = 1$), permute the bands ($b_2 = 1$), flip the cells along the minor diagonal ($b_6 = 1$), print the resulting board ($b_7 = 1$), and then save it back to `filename` ($b_8 = 1$). If the user requests several operations, perform them in the order listed above (first b_0, then b_1, then b_2, etc.).

If a function that you have not implemented is requested print the following error message (replace `XXXX` with the name of the function):

```
Function XXXX not implemented.
```

Sudoku board generation functionality

In your program, use a two-dimensional array to represent Sudoku boards:

```
int board[9][9];
```

An array element that corresponds to a filled cell on the Sudoku board holds an integer between 1–9. An array element that corresponds to a blank cell holds a 0.

You are now ready to start implementing the functions from your Sudoku library (Steps 3–13). These functions should be declared in `enee140_sudoku.h` and implemented in `enee140_sudoku.c`. In some cases, you may find that breaking down the functionality into smaller *helper functions*, which may be re-used for several operations, will make the implementation easier.

3 Check board validity

Implement a function that checks whether a board is valid:

```
int is_valid_board(int board[9][9]);
```

The function should return 0 if the `board` parameter violates any of the Sudoku rules and 1 if the board is valid. You have to check four constraints:

- Each cell holds a number between 1–9 (for the filled cells) or 0 (for the blanks).
- No number is repeated in any row.
- No number is repeated in any column.
- No number is repeated in any box.

4 Generate new Sudoku board

Implement a function with the following prototype

```
void new_sudoku_board(int board[9][9], int blanks);
```

This function should generate a valid Sudoku board by starting from a canonical board with no blanks. Then, the function should blank out some of the resulting cells. You may start from the following canonical board, which is created by generating the sequence of ordered numbers between 1 and 9 and copying them to rows 1, 4, 7, 2, 5, 8, 3, 6, 9 of the board, each time shifting the sequence one position to the left:

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

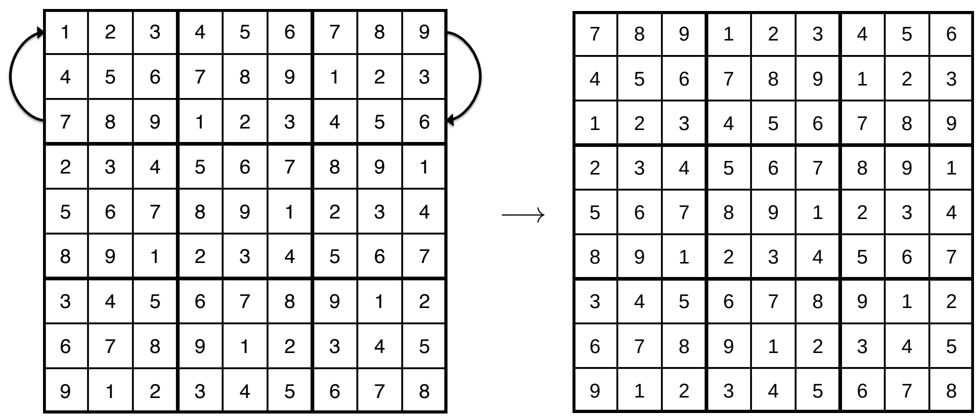
First, analyze this board to verify that it satisfies the rules of Sudoku. To generate a new board, assign these values to the `board` parameter of the function. Then, blank out several cells on this board by randomly selecting a row `i` and a column `j` and by setting `board[i][j] = 0`. The number of blank cells on the resulting board should equal the `blanks` parameter.

Hint: After randomly generating a cell to blank out, you should make sure that you haven't already added a blank in that cell.

Important: A proper Sudoku puzzle has a unique solution. This is **not a requirement** for this project. It is ok if, after removing some cells, the generated board has several solutions.

5 Permute rows within bands

If you permute the rows within the same band of a valid Sudoku board, you will obtain another valid board:



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

$\longrightarrow$

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

Implement a function with the following prototype:

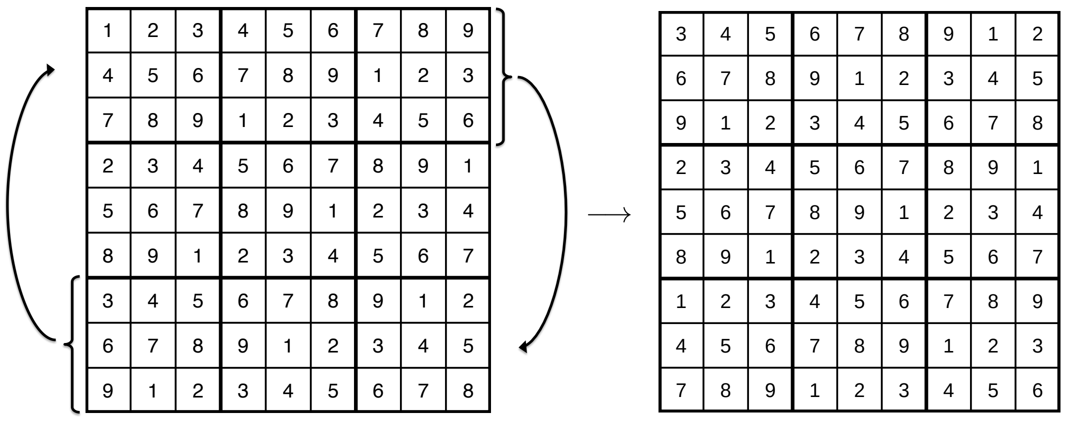
```
void transform_permute_rows(int board[9][9]);
```

The function should generate a random permutation of the row numbers and swap the rows of the board according to these permutations. Note that some permutations of the $\{1, 2, \ldots, 9\}$ sequence are invalid, because each row can only move within its own band. For example, the permutation $\{3, 2, 1, 4, 5, 6, 7, 8, 9\}$ illustrated above is valid, while the permutation $\{4, 2, 3, 1, 5, 6, 7, 8, 9\}$ is invalid.

Hint: Create a helper function that generates a random permutation of the $\{1 \ldots n\}$ sequence for any given n. Use permutations generated in this manner to determine how to rearrange the rows of the canonical board.

6 Permute bands

If you permute the three bands of a valid Sudoku board, you will obtain another valid board:



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

$\longrightarrow$

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

Implement a function with the following prototype:

```
void transform_permute_bands(int board[9][9]);
```

This function should generate a random permutation of the $\{1, 2, 3\}$ sequence and swap the bands of the board according to this permutation.

7 Permute columns within stacks

If you permute the columns within the same stack of a valid Sudoku board, you will obtain another valid board. Implement a function with the following prototype:

```
void transform_permute_columns(int board[9][9]);
```

Generate a random permutation of the column numbers, like in Step 5, and swap the columns of the board according to this permutation. Note that some permutations of the $\{1, 2, \ldots, 9\}$ sequence are invalid, because each column can only move within its own stack.

This functionality is optional; implement it if you want to receive bonus points.

8 Permute stacks

If you permute the three stacks of a valid Sudoku board, you will obtain another valid board. Implement a function with the following prototype:

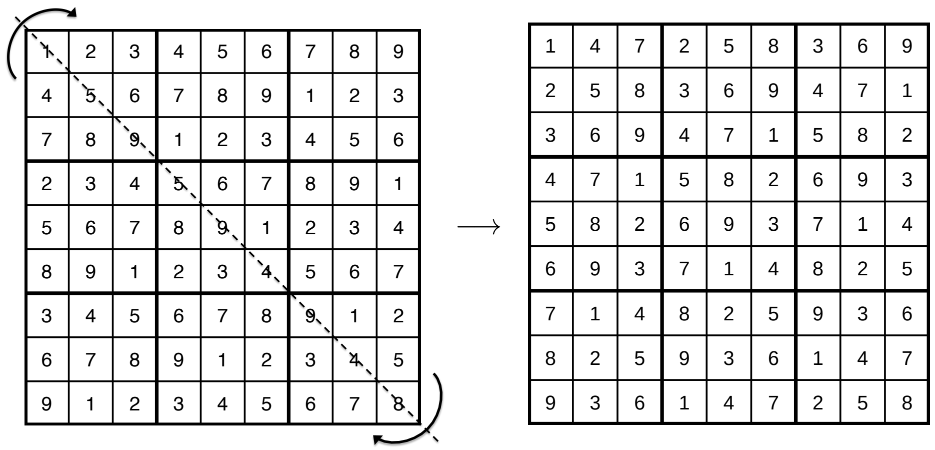
```
void transform_permute_stacks(int board[9][9]);
```

Generate a random permutation of the $\{1, 2, 3\}$ sequence, like in Step 6, and swap the stacks of the board according to this permutation.

This functionality is optional; implement it if you want to receive bonus points.

9 Flip cells along main diagonal

If you flip the cells of a valid Sudoku board along the main diagonal, you will obtain another valid board:



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

$\longrightarrow$

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

Implement a function with the following prototype:

```
void transform_flip_main_diagonal(int board[9][9]);
```

This function should flip cells in a Sudoku board along the main diagonal, as illustrated above. A cell `board[i][j]` is on the main diagonal if `i == j`. The cells of the resulting board should mirror the cells of the canonical board along the main diagonal. Note that this transformation is deterministic (it does not involve any random moves).

10 Flip cells along minor diagonal

If you flip the cells of a valid Sudoku board along the minor diagonal, you will obtain another valid board. Implement a function with the following prototype:

```
void transform_flip_minor_diagonal(int board[9][9]);
```

This function should flip the cells of `board` along the minor diagonal. The minor diagonal runs from the top-right corner to the bottom-left corner.

11 Print Sudoku board in a human-friendly manner

Implement a function with the following prototype:

```
int print_sudoku_board(int board[9][9]);
```

This function should print the board to the terminal screen in a human-friendly manner. The function should return 0 on success and -1 if the `board` argument includes any invalid numbers (other than 0–9). For example, the Sudoku board on page 1 should be printed as follows:

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

12 Read Sudoku board from a file

Implement a function with the following prototype:

```
int read_sudoku_board(const char file_name[], int board[9][9]);
```

This function should read a Sudoku board from a file, in the format described in Step 13. Ignore any characters after the 9th line in the file. The function should return 0 on success, -1 if any file

I/O error occurred, -2 if an invalid character (other than 1–9 and -) was encountered and -3 if the file format is invalid (e.g. not enough lines or lines too short).

13 Save Sudoku board to a file

Implement a function with the following prototype:

```
int write_sudoku_board(const char file_name[], int board[9][9]);
```

This function should save the board to a file. The function should return 0 on success, -1 if any file I/O error occurred and -2 if an invalid board was provided. In the output file, write a line for each row, with no spaces, and use a dash (the `'-'` character) to represent blanks. For example, the Sudoku board on page 1 should be saved as follows:

```
-6-32-8-5
7----4--1
23---7-6-
-4--1----
57--492--
--3---64-
--------7
-5----1--
31-758--6
```

Hint: If you have read the board from the same file (using the function from Step 12), make sure you close the file and re-open it before saving the board.

14 (5 Bonus Points) Allow the user to specify the difficulty level

Allow the user to provide an optional third parameter on the command line. This parameter should be a string, which specifies the difficulty level of the board generated as follows:

- `"easy"`: Generate a board with 31–45 blanks
- `"medium"`: Generate a board with 46–51 blanks
- `"hard"`: Generate a board with 52–61 blanks

When invoking the function from Step 4, generate a new board of the specified difficulty level.

15 (15 Bonus Points) Write a program that solves Sudoku puzzles

Write a separate program, called `enee140_solve_sudoku.c`, that reads a Sudoku board using the function from Step 12 and then computes a solution by finding the appropriate values to fill the blanks without violating any of the validity constrains described in Step 3. The program should work for any Sudoku board, not just the ones generated from the canonical board from Step 4. You may use all the functions implemented so far in `enee140_sudoku.c`; additionally, implement a function with the following prototype:

```
int solve_sudoku_board(int board[9][9]);
```

The function should return the number of board validity tests performed in order to find the solution. After the function returns, the `board` parameter should hold the solution.

Hint: You can build your solution incrementally, by filling blanks one by one. Try to place the 1–9 values, in order, on a blank position and check if the board is valid. If it is, advance to the next blank position; otherwise, try the next number. If you cannot place any number in the current blank, this means that the board cannot be solved with the numbers you have placed so far; return to the previous blank position and try the next number there. This strategy is called *backtracking*.

Project requirements

1. You must program in C and name your program files `enee140_sudoku.c`, `enee140_sudoku.h`, `enee140_gen_sudoku.c` (and, optionally, `enee140_solve_sudoku.c`). Templates for these programs are included at the end of this document (you do not have to use them, but they may provide some hints).
2. Your programs must compile on the GRACE UNIX machines using

   ```
   gcc enee140_sudoku.c enee140_gen_sudoku.c
   ```
3. Your programs must implement all the steps described above correctly.
4. Your programs must be readable to other programmers (e.g. the TAs).
5. Before you submit, you must archive all the programs you wrote into a single file, called `enee140_gen_sudoku.tar.gz`, using the `tar` command. For example, if all your program files are in the current directory, you can use the following UNIX command:

   ```
   tar cvfz enee140_gen_sudoku.tar.gz enee140_sudoku.c enee140_sudoku.h enee140_gen_sudoku.c
   ```

   Submit the `enee140_gen_sudoku.tar.gz` file using the following command:

   ```
       submit 2016 spring enee 140 AAAA 103 enee140_gen_sudoku.tar.gz
   ```

   Note: you must replace `AAA` with your own section number (0101, 0102, etc.)

Grading criteria

Correctness:	80%
Good coding style and comments:	20%
Late submission penalty:	-40% for the first 24 hours
	-100% for more than 24 hours
Program that does not compile on GRACE:	-100%
Wrong file names (other than `enee140_sudoku.c`, `enee140_sudoku.h`, `enee140_gen_sudoku.c`, `enee140_gen_sudoku.tar.gz`):	-100%
Bonus (difficulty level):	5%
Bonus (Sudoku solver):	15%

Program templates

You can start from the following templates (also available in the GRACE class public directory, at `public/projects/project3`).

Template for `enee140_sudoku.h`

```
/*
 * enee14_sudoku.h
 *
 *  Function prototypes for the Sudoku library.
 */

#ifndef SUDOKU_H_
#define SUDOKU_H_

int is_valid_board(int board[9][9]);

#endif /* SUDOKU_H_ */
```

Template for `enee140_sudoku.c`

```
/*
 * enee140_sudoku.c
 *
 *  Implementation of the Sudoku library.
 *
 */

#include "enee140_sudoku.h"
#include <stdio.h>
#include <stdlib.h>

// Check that the board is valid: each number between 1 and 9 must appear only
// once in each row, column and 3 x 3 box in the board. The board may include
// blank elements.
int
is_valid_board(int board[9][9])
{
}
```

Template for `enee140_gen_sudoku.c`

```
/*
 * enee140_gen_sudoku.c
 *
 *  Generate Sudoku boards.
 *
 */

#include "enee140_sudoku.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int
main(int argc, char *argv[])
{
	int board[9][9];

	return 0;
}
```