Imperial College of Science, Technology and Medicine
Department of Computing

MSc C++ Programming – Assessed Exercise No. 1

Due: see Scientia[1]

Problem Description

Sudoku[2] is a popular number-placement puzzle played on a 9 × 9 board. Starting with a partially completed board of the kind shown in Figure 1, the objective is to fill the board with digits so that each row of the board, each column of the board and each of the nine 3 × 3 sub-boards (outlined by the thick black lines in the figure) contain all of the digits from 1 to 9. Ideally Sudoku problems have just one unique solution.

Typically Sudoku players use logic to deduce what digit should be assigned to a particular board position. For example, one can readily work out that the highlighted square towards the bottom right of Figure 1 must contain the digit 1. This is because the lower-right 3 × 3 sub-board must contain a 1 and there is a 1 in the 7th and 8th rows and in the 7th and 9th columns already.

Your challenge here is to code a Sudoku puzzle solver. To aid you in developing your program, you are supplied with five test Sudoku boards: the "easy" one shown in Figure 1, one of "medium" difficulty, and three "mystery" boards.

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

Figure 1: A Sudoku puzzle board

Pre-supplied functions and files

You are supplied with some data files representing Sudoku boards: `easy.dat`, `medium.dat`, `mystery1.dat`, `mystery2.dat`, and `mystery3.dat`. We will be storing these boards in our

[2] A contraction of the Japanese *Suji wa dokushin ni kagiru* which translates as "The digits must be single".

program in a two-dimensional (9×9) array of characters. The solution to the "easy" Sudoku board is given in the data file `easy-solution.dat`.

You can use the command `cat` to inspect these files, e.g.

```
% cat easy.dat           % cat easy-solution.dat
...1.83..               697128345
24..5....               241653897
..8....61               538497261
..4..9..3               714269583
.6.....2.               865371429
3..8..1..               329845176
17....9..               176582934
....1..52               483916752
..27.4...               952734618
```

You are also supplied with some helper functions (with prototypes in `sudoku.h` and implementations in the file `sudoku.cpp`):

- `void load board(const char* filename, char board[9][9])` reads in characters from the file with name filename into the two-dimensional character array board.
- `void display board(char board[9][9])` displays the 2D character array `board` in a friendly layout familiar to Sudoku players. Row indices (in the form of letters 'A' to 'I') and column indices (in the form of digits '1' to '9') are included in the output to help with the identification of particular board positions.

To illustrate the use of the above functions, consider the code:

```
char board[9][9];
load_board("easy.dat", board);
display_board(board);
```

This results in the output:

```
    1   2   3   4   5   6   7   8   9
  +===========+===========+===========+
A |   :   :   | 1 :   : 8 | 3 :   :   |
  +---+---+---+---+---+---+---+---+---+
B | 2 : 4 :   |   : 5 :   |   :   :   |
  +---+---+---+---+---+---+---+---+---+
C |   :   : 8 |   :   :   |   : 6 : 1 |
  +===========+===========+===========+
D |   :   : 4 |   :   : 9 |   :   : 3 |
  +---+---+---+---+---+---+---+---+---+
E |   : 6 :   |   :   :   |   : 2 :   |
  +---+---+---+---+---+---+---+---+---+
F | 3 :   :   | 8 :   :   | 1 :   :   |
  +===========+===========+===========+
G | 1 : 7 :   |   :   :   | 9 :   :   |
  +---+---+---+---+---+---+---+---+---+
H |   :   :   |   : 1 :   |   : 5 : 2 |
  +---+---+---+---+---+---+---+---+---+
I |   :   : 2 | 7 :   : 4 |   :   :   |
  +===========+===========+===========+
```

You are also supplied with a main program in `main.cpp`.

Specific Tasks

1. Write a Boolean function `is_complete(board)` which takes a 9×9 array of characters representing a Sudoku board and returns `true` if all board positions are occupied by digits, and `false` otherwise. Note you do *not* need to check whether each digit is logically valid.

   For example, the code:

   ```
   load_board("easy.dat", board);
   cout << "Board is ";
   if(!is_complete(board)) {
       cout << "NOT ";
   }
   cout << "complete." << '\n';
   ```

   should display the output

   ```
   Loading Sudoku board from file 'easy.dat'... Success!
   Board is NOT complete.
   ```

   Similarly, the code:

   ```
   load_board("easy-solution.dat", board);
   cout << "Board is ";
   if(!is_complete(board)) {
       cout << "NOT ";
   }
   cout << "complete." << '\n';
   ```

   should display the output

   ```
   Loading Sudoku board from file 'easy-solution.dat'... Success!
   Board is complete.
   ```

2. Write a Boolean function `make_move(position, digit, board)` which attempts to place a digit onto a Sudoku board at a given position. Here `position` is a two-character string denoting row (A to I) and column (1 to 9) board coordinates (e.g. `"I8"`), `digit` is a character denoting the digit to be placed (from `'1'` to `'9'`), and board is a two-dimensional character array. If `position` is invalid (e.g. because the coordinates are out of range), or the placing of the digit at position is invalid (e.g. because it would result in two copies of the same digit in the same row), then the return value of the function should be `false`, and `board` should be unaltered. Otherwise, the return value of the function should be `true` and `board` should be updated to reflect the placing of digit at position.

   For example, the code:

   ```
   load_board("easy.dat", board);
   cout << "Putting '1' into I8 is ";
   if(!make_move("I8", '1', board)) {
       cout << "NOT ";
   }
   cout << "a valid move." << '\n';
   ```

should result in the output:

```
Loading Sudoku board from file 'easy.dat'... Success!
Putting '1' into I8 is a valid move.
```

and board cell I8 should be `'1'`.

3. Write a Boolean function `save_board(filename, board)` which outputs the two-dimensional character array `board` to a file with name `filename`. The return value should be `true` if the file was successfully written, and `false` otherwise.

   For example, the code:

```
load_board("easy.dat", board);
if(save_board("easy-copy.dat", board)) {
    cout << "Save board to 'easy-copy.dat' successful." << '\n';
}
else {
    cout << "Save board failed." << '\n';
}
cout << '\n';
```

   should result in the output:

```
Loading Sudoku board from file 'easy.dat'... Success!
Save board to 'easy-copy.dat' successful.
```

   with `easy-copy.dat` having identical contents to `easy.dat`.

4. Write a Boolean function `solve_board(board)` which attempts to solve the Sudoku puzzle in input/output parameter `board`. The return value of the function should be `true` if a solution is found, in which case `board` should contain the solution found. In the case that a solution does not exist the return value should be `false` and `board` should contain the original board.

   For full credit for this part, your function – or helper function if you choose to use one – should be recursive.

   For example, the code:

```
load_board("easy.dat", board);
if(solve_board(board)) {
    cout << "The 'easy' board has a solution:" << '\n';
    display_board(board);
}
else {
    cout << "A solution cannot be found." << '\n';
}
```

   should result in the output:

```
Loading Sudoku board from file 'easy.dat'... Success!
The 'easy' board has a solution:
    1   2   3   4   5   6   7   8   9
```

```
  +===========+===========+===========+
A | 6 : 9 : 7 | 1 : 2 : 8 | 3 : 4 : 5 |
  +---+---+---+---+---+---+---+---+---+
B | 2 : 4 : 1 | 6 : 5 : 3 | 8 : 9 : 7 |
  +---+---+---+---+---+---+---+---+---+
C | 5 : 3 : 8 | 4 : 9 : 7 | 2 : 6 : 1 |
  +===========+===========+===========+
D | 7 : 1 : 4 | 2 : 6 : 9 | 5 : 8 : 3 |
  +---+---+---+---+---+---+---+---+---+
E | 8 : 6 : 5 | 3 : 7 : 1 | 4 : 2 : 9 |
  +---+---+---+---+---+---+---+---+---+
F | 3 : 2 : 9 | 8 : 4 : 5 | 1 : 7 : 6 |
  +===========+===========+===========+
G | 1 : 7 : 6 | 5 : 8 : 2 | 9 : 3 : 4 |
  +---+---+---+---+---+---+---+---+---+
H | 4 : 8 : 3 | 9 : 1 : 6 | 7 : 5 : 2 |
  +---+---+---+---+---+---+---+---+---+
I | 9 : 5 : 2 | 7 : 3 : 4 | 6 : 1 : 8 |
  +===========+===========+===========+
```

5. Consider the following information about the mystery puzzle boards in `mystery1.dat`, `mystery2.dat`, and `mystery3.dat`:

   - One is a Sudoku board of "hard" difficulty.
   - One is a Sudoku board of "extremely hard" difficulty.
   - One is actually impossible to solve.

   Your task is to identify which mystery board matches each of the descriptions above. Summarise your findings in relation to the identification of the puzzles in the plain text file `findings.txt`.

What To Hand In

Place your function implementations in the file `sudoku.cpp` and corresponding function declarations in the file `sudoku.h`. Use the file `main.cpp` to test your functions. Summarise your findings related to task 5 in the file `findings.txt`. Create a `makefile` which compiles your submission into an executable file called `sudoku`. Submit your files via your GitLab repo on LabTS, and ensure that your submission has been recognised by Scientia. **Do not forget to test your final commit before submitting it.**

How You Will Be Marked

You will be assigned a mark according to:

- whether your program works or not,
- whether your program is clearly set out with adequate blank space and indentation,
- whether your program is adequately commented,
- whether you have used meaningful names for variables and functions, and
- whether you have used a clear, appropriate and logical design.

Note that submissions that do not compile on LabTS will be given 0 marks.

Hints

1. You will save time if you begin by studying the main program in `main.cpp`, the header file `sudoku.h`, the pre-supplied functions in `sudoku.cpp` and the given data files.

2. All the questions will be **much** easier if you exploit the pre-supplied helper functions.

3. Feel free to define any of your own helper functions which would help to make your code more elegant.

4. You are explicitly required to use recursion in your answer to Question 4. You are not obliged to use recursion in answering any other question.