Data Structures and Objects
CSIS 3700

Spring Semester 2020 — CRN 21212 / 21213

Project 2 — Sudoku Solver
Due date: Friday, March 16, 2020

Goal

Develop and implement a stack-based Sudoku puzzle solver.

Details

You are most likely familiar with the Sudoku puzzle game. It consists of a 9-by-9 grid; initially, some of the positions are filled with numbers and others are blank.

When solved, each row must have all of the integers 1 – 9, each column must have all of the integers 1 – 9 and each of the nine 3-by-3 blocks must have all of the integers 1 – 9.

One method to solve a Sudoku puzzle is trial-and-error. If a valid guess can be made, make it and repeat. If no valid guess can be made, go back to the previous guess and change it; if no other guesses can be made, go back to the guess before that. Continue until either all boxes are filled or all guesses are exhausted (which shouldn't happen because that means there is no solution.) Create a stack of integers which represent the location of cells being filled in. The top location on the stack represents the cell currently being filled in. Note that there will be at most 64 locations to track, since any Sudoku puzzle must have at least 17 cells filled in initially.

Note: If you're clever, you only need one integer to represent the location, not two.

Read the data from **`cin`**. Input consists of nine lines of nine characters. If a cell is filled in, its character will be a digit 1 – 9. If it is blank, its character is a period.

Pro tip: You only need a single **`char`** variable for the input; no strings necessary.

Once the data is read, use the following algorithm to solve the puzzle.

Algorithm 1 The main Sudoku algorithm

Preconditions *board* contains an unsolved Sudoku puzzle

Postconditions *board* contains a solved Sudoku puzzle

```
procedure SOLVE
    Select the best empty cell and place its location on the stack

    while true do
        Let (i, j) be the location on top of the stack

        Select the next valid choice for board[i][j]
        if no such choice exists then
            Mark board[i][j] as not filled in
            Pop the stack
            if the stack is empty then
                Return; the puzzle has no solution
            end if
            continue
        end if

        Select the best empty cell and place its location on the stack
        if no such cell exists then
            break                                        ▷ Puzzle is now solved
        end if
    end while

    Output the solution
end procedure
```

▹*Keeping track of choices*

You should use bit manipulation to keep track of information for each cell. All of the information for one cell can be kept in 14 bits:

- One bit to indicate if the cell has been filled in
- Nine bits to keep track of which digits you are allowed to place in the cell
- Four bits to hold the current choice for the cell

Although it's not strictly necessary, you can fit all of the information into the `short int` data type.

You'll want to use the masking operations at various points in the program to turn bits on and turn them off. You'll also want to use the left shift operation to look at the valid choices for a cell.

▹*Selecting the best empty cell*

Hypothetically, you can pick any empty cell for your next choice. However, to minimize the work the computer performs in backtracking, there is a preferred cell. The best cell to choose has the fewest valid choices for its digit.

The following algorithm selects the best empty cell and places its location on the stack.

Algorithm 2 Finding the best empty cell

Preconditions *board* contains an unsolved Sudoku puzzle

Postconditions the best location is pushed onto the stack
the best location is marked as filled in

```
procedure FINDBEST
    for each empty cell board[i][j] do          ▹ Initialize to allow all digits as choices
        Mark all digits as valid choices
    end for

    for each filled in cell board[i][j] do      ▹ Remove invalid choices
        for each unfilled cell in row i do
            Mark digit in board[i][j] as an invalid choice
        end for
        for each unfilled cell in column j do
            Mark digit in board[i][j] as an invalid choice
        end for
        for each unfilled cell in the 3x3 block containing board[i][j] do
            Mark digit in board[i][j] as an invalid choice
        end for
    end for

    Set low ← 10
    for each empty cell board[i][j] do
        Count 1-bits in valid choices for board[i][j]
        if count < low then
            low ← count
            i_best ← i
            j_best ← j
        end if
    end for

    if low = 10 then
        return false                            ▹ No empty cells remain
    end if

    Mark board[i_best][j_best] as filled in
    Push (i_best, j_best) onto the stack

    return true
end procedure
```

What to turn in

Turn in your source code and `Makefile`. If you are using an IDE, compress the folder containing the project and submit that.

Example 1

▹*Input*

```
4.......1
.2.7..59.
.98.437..
..1.2..7.
..53.49..
.8..9.2..
..798.12.
.52..7.3.
8.......7
```

▹*Output*

```
Solution: 476 259 381
          123 768 594
          598 143 762

          931 825 476
          265 374 918
          784 691 253

          347 986 125
          652 417 839
          819 532 647
```

Example 2

▹*Input*

```
.........
47..9..36
8...4...1
...2.4...
.25...48.
1.3...2.7
21.....48
....7....
.9.3.2.7.
```

▹*Output*

```
Solution: 361 827 954
          472 195 836
          859 643 721

          786 234 519
          925 716 483
          143 958 267

          217 569 348
          538 471 692
          694 382 175
```

Example 3

▹Input

```
..423...9
3.....27.
6..9.....
..3....54
4...86.9.
..9....26
1..7.....
9.....41.
..634...5
```

▹Output

```
Solution: 754 238 169
          391 654 278
          682 917 543

          863 129 754
          427 586 391
          519 473 826

          148 795 632
          935 862 417
          276 341 985
```

Example 4

▹Input

```
1.......9
.5.8...6.
....9.8.4
43..81...
6...2...1
...65..43
5.6.3....
.7...8.3.
2.......7
```

▹Output

```
Solution: 148 276 359
          759 843 162
          362 195 874

          437 981 625
          695 324 781
          821 657 943

          516 732 498
          974 518 236
          283 469 517
```