Sudoku Solver

Version: 2.5

Due Date: April 5th 2013

Summary:

- For this assignment you will be writing a program to solve Sudoku puzzles.
- You are provided with a makefile, the ".h" files, and cell.cpp, and a master solution named master_solver.
- You are not to alter any of the provided files.
- You are expected to complete the following:
    1. The driver for the program (named driver.cpp)
    2. The ".cpp" implementation for sudoku.h (named sudoku.cpp)
    3. The UML diagram for the program
- Input will only ever consist of '.', the numbers from 1 to 9 inclusively, or potentially spaces. You are not responsible for checking to make sure input is valid.
- There are no restrictions on the potential input – you are to account for every possible initial marking of the cells.
- Your program is to either produce a solved Sudoku puzzle for the corresponding input or an error message stating that there is no solution.
- In cases where correct input can produce multiple valid solutions, your implementation is to provide only one of the valid solutions and exit with a value of 0.
- If the input does not have a solution you are to print a descriptive error message, to standard error, that begins with the word "ERROR" and exit the program with a value of 0.
- Your program must use recursion to implement the backtracking method
- You may only implement and use the methods described below to solve a puzzle
- Your program should have no memory leaks.

Glossary:

Box:

- Refers to one of the 9, 3x3 matrices, found in a Sudoku puzzle

Candidate:

- A possible value of a cell

Determined Candidate:

- The value of the only possible candidate in a solved cell

Failure:

- A guess that resulted in an incorrect puzzle

Finalize a Cell:

- The process of removing all impossible candidates from a solvable cell and leaving it with only one remaining candidate.

Puzzle:

- The 9x9 grid of a Sudoku puzzle

Solvable Cell:

- A cell for which we can finalize

Solved Cell:

- A cell that has only one possible candidate remaining.

Solved Puzzle:

- A solution to the given Sudoku puzzle

Background:

Sudoku is a number based puzzle game. The objective of Sudoku is to fill in every cell in the 9x9 grid with a number from 1 to 9. Valid solutions to a Sudoku puzzle comply with the following rules:

- Cell values given in the initial puzzle must not be changed.
- Each row must contain the numbers from 1 to 9
- Each column must contain the numbers from 1 to 9
- Each of the nine 3x3 boxes must contain the numbers from 1 to 9

Below are two example Sudoku puzzles and a corresponding solution to each puzzle:

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

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

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

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

Algorithm:
Using the above rules here are 5 methods that you are to use to solve any Sudoku puzzle.

Method 1 – Cell Refinement:

- When a cell is solved (there is only one possible candidate remaining) we finalize the cell and eliminate the possibility of that candidate from all cells in the same row, column, and box (3x3 grouping of cells). If eliminating this candidate solves another cell then we repeat this method on the solvable cell.

Method 2 – Row Check:

- Utilizing the rule “each row must contain the numbers from 1 to 9” we can potentially solve more cells. Specifically, any cell with a candidate not found in the other cell in the same row can be solved for that candidate. For such cells we apply Method 1 on the solvable cell.

Method 3 – Colum Check:

- Method 3 works the same way as Method 2 but for columns. For each solvable cell, we apply Method 1 on the solvable cell.

Method 4 – Box Check:

- Method 4 works the same way as Method 2 but for boxes. For each solvable cell, we apply Method 1 on the solvable cell.

Method 5 – Backtracking:

- Methods 1 through 4 will not always produce a solved Sudoku puzzle. For these situations we must guess what the correct candidate of a particular cell is so that we can progress.
- To do this we will implement a recursive backtracking algorithm. If an incorrect guess is made the backtracking algorithm will allow us to return to the state of the puzzle before we make a guess, from here we can make a new guess.
- These guesses will either lead to a solved puzzle, a failure, or a state that requires another guess to be made.
- In the case of a failure we will (in order):
    1. Free all memory from the puzzle that caused the failure.
    2. Return a value indicating that our guess was a failure.
    3. Make a new guess on the cell with the least number of possible candidates.
- To reduce the number of potential incorrect guesses, first search for the cell that contains the least possible candidates, then make a guess on the candidate with the smallest value in the found cell.
- In the case where the input given has no solution you are to produce an error message beginning with the word “ERROR” that details the error and exit your program after with a value of 0.

Pseudo Code:
The following is the pseudo code for the recursive backtracking algorithm you must implement in your driver to solve Sudoku puzzles.

```
Sudoku* solve (Sudoku* puzzle) {
     Method 2;
     Method 3;
     Method 4;

     //finalize all solvable cells and eliminate the determined
     //candidate form all related cells.
     Method 1;

     if (failure)
          return NULL;
     else if (puzzle is solved)
          return puzzle;
     else if (no more solvable cells) {
          //Create a new puzzle with a cell finalized to the
          //guessed candidate and eliminate the guessed
          //candidate from the original puzzle.
          newpuzzle = guess (puzzle);
          solve (newpuzzle);
          if (the guess was a failure)
               return solve (puzzle);
          return newpuzzle;
     }
     else
          return solve (puzzle);
}
```

Implementation/Purpose:
Cell Class:

- The purpose of this class is to store the possible candidates of a given cell.
- Do not modify the cell.h or cell.cpp file in any way.
- Note: candidates[0] corresponds to what whether or not the value 1 is a candidate for the cell. Furthermore, candidates[1] corresponds to whether or not the value 2 is a candidate for the cell not value 1.

Sudoku Class:

- This class is meant to represent an actual Sudoku puzzle. It will be used to implement methods 1 to 4.
- The class contains a 9x9 2D array of pointers to Cells.
- The sudoku.h file is not to be modified in any way.

Driver:

- The driver for your program is will recursively implement the backtracking method as well as the calls to the other methods in the algorithm as detailed above.

Input:

- Your program is to accept input from standard input
- Input will only ever consist of '.', the numbers from 1 to 9 inclusively, or potentially spaces. You are not responsible for checking to make sure input is valid.
- '.' represents a cell for which we are not give the finalized value
- A space is not guaranteed to be in the input, nor is it guaranteed that the input will be formatted in the ways presented below
- The only other valid input is a number from 1 to 9
- The following are examples of the input format you may be given:

```
. . . . . 4 . 2 8
4 . 6 . . . . . 5
1 . . . 3 . 6 . .
. . . 3 . 1 . . .
. 8 7 . . . 1 4 .
. . . 7 . 9 . . .
. . 2 . 1 . . . 3
9 . . . . . 5 . 7
6 7 . 4 . . . . .
OR
.....4.284.6.....51...3.6.....3.1....87...14....7.9.....2.
1...39.....5.767.4.....
```

Output:

- Your program should always begin by printing the input as follows and then a new line in the following way:

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

- If there is no solution, you are to print an appropriate error message to standard error beginning with the word "ERROR" and exit the program with a value of 0.
- If there is a solution to the given input you are to print out the solved puzzle in the way shown above.
- For more detail on the output of the program you are to test your input against the provided master solution

Assignment Specifications:

a6p1: Create sudoku.cpp

Write the implementation for all functions found in the file suodku.h. Name this file sudoku.cpp.

Your sudoku.cpp file is to #include only the following:

- "sudoku.h"

Zip the following files into to a6p1.zip and submit the zip file to a6p1 on Marmoset:

- sudoku.cpp

a6p2: Create driver.cpp

Write the driver for the Sudoku solver. Your driver is to implement the algorithm described above. Name this file driver.cpp.

Your driver.cpp file is to #include only the following:

- <iostream>
- <cstdlib>
- "sudoku.h"

Zip the following files into to a6p2.zip and submit the zip file to a6p2 on Marmoset:

- sudoku.cpp
- driver.cpp

a6p3: Create the UML

Create the UML diagram for the program. Save your UML diagram to a PDF file and name the file SudokuUML.pdf.

Submit the following files to a6p3 on Marmoset:

- SudokuUML.pdf

Marking Scheme:

Question	Marks
a6p1	30
a6p2	50
a6p3	10
Style	10

~ contributed by Richard Bruce Wallace