Introduction:

Sudoku is a popular puzzle where a player completes a grid (see below) based on some (but not all) cells provided. The challenge is to find the values of the unknown cells.

Rules:

A traditional Sudoku puzzle is either a 4x4, 9x9 or 16x16 grid. In the case of a 9x9 numeric puzzle, a solved puzzle has the following three properties:

• Every column contains the symbols 1–9 exactly once
• Every row contains the symbols 1–9 exactly once
• Each of the 3x3 minigrids contains the symbols 1–9 exactly once.

Figure 1: A Sudoku puzzle (From Will Shortz, Sudoku: Easy to Hard, St. Martin’s Griffin, 2005).
Over the next two weeks you will develop a non-recursive Sudoku solver based on sequence container data structures of your choice. Unlike the previous lab, we will allow you to use the C++ STL (vector, list, etc.). Note, however, that the choice of container may impact computational efficiency (see in class exercise re: circular linked lists).

All valid solutions will receive full credit for the puzzle portion of your grade. **If you decide to implement a simple recursive solver, however, you will not receive credit for data structure-linked portions.** Please see the accompanying rubric.

**Input format for lab:**

A puzzle text file will consist of 9 lines, each of which contains 9 integer values separated by a space. The integer values are either in the 1–9 (filled) or 0 (empty). Here is the representation of the previous page’s puzzle as an example:

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

**Part 1: Implementing a solver (start in lab but you will not finish in 50 mins)**

We recommend you store the Sudoku puzzle using a 2D array/vector. It may be slightly easier to represent each cell of the puzzle as a simple class or struct, since you may choose to use a non-vector container to store possible values for the algorithms below.

We will require you to implement scanning/elimination. Specifically, at the start all 9 values are possible and you repeatedly eliminate possible values based on the current state of the row/column/minigrid. If only a single value is left, fill the square with that value and restart the process.

To solve the medium and hard puzzles you need to also implement the “singleton” algorithm described below. For example, suppose that a puzzle has the row:

```
| 1 | A | 5 | 9 | 6 | 7 | 2 | B | C |
```

and that we know that A can only be 3, 4 or 8, B can only be 3 or 4, and C can only be 3 or 4 (based on other information). The value 8 is a singleton because it only appears in one cell, and therefore cell A must be an 8.

These two algorithms (single possibility and singleton) will be sufficient to solve all three puzzles used for grading. Note it is also possible that a solution with bugs may solve only one puzzle we will use.
There are a few online sites with sample Sudoku puzzles and a solver you can use for testing. This is one students’ used previously:

http://www.sudokuwiki.org/sudoku.htm

**Part 2: Putting it all together**

1. The final output out your solver should be the same as the input, with all 0s correctly replaced. Please send it to the screen using cout for grading.

2. Use (and debug if necessary) your solver to solve the easy and medium puzzles posted on the course website and Piazza.

3. In your usual report, add a paragraph on the strategies in your solver, what you think worked well, what did not work, and a general summary of the lab/post-lab. This will primarily be used to give feedback and evaluate the code.

**BONUS:** Your code will also be run on two hard difficulty problems. If it produces the correct result, you will receive 2 extra credit points per problem (max of 4 total).

**Rubric that will be used for grading**

**Part 1: 35 points**

--------

+2 Makefile correctly compiles code
+4 interface (.h) and implementation commented well (2 pts each)
Requested functionality present. Correctness of solver will be judged later using puzzles
- solve member function(s) that fills out the puzzle (3 pts)
- print functionality that displays the final result. (3 pts)
+4 uses fopen/streams properly for input
+5 correctly uses a sequence container to store the remaining possibilities
+10 functionality that eliminate values based on the current row/column/minigrid.
+10 functionality such that if only a single value is left, the square is filled in with that value and the process is restarted

**Part 2: Finishing up 40/pts**

--------

+20 solves the easy problem correctly and displays to the screen (from Piazza)
+20 solves the medium problem correctly and displays to the screen (from Piazza)

**Lab Report: 20 points**

--------

+4 Explains how the user uses the program.
+6 Explains how the program works internally.
+5 Explains how the program was verified.
+5 Text on what you think worked well, what did not work, and a general summary of the “how” the solver was implemented. This is effectively more detail than usual on how the program works, but from the programmer perspective.