Programming Assignment #1: Su-Do-Kode

COP 3502, Fall 2015

Due: Wednesday, September 2, 11:59 PM via Webcourses

Abstract

In this programming assignment, you will write a solution checker for Sudoku puzzles. This assignment is intended to get you thinking in C again, as well as to acclimate you to the style of assignment submissions and grading you will encounter in this course.

By completing this assignment, you will gain hands-on experience using the Linux command line. In particular, you will use a *diff* tool to check for very precise file output matching.

Attachments

sample_input1.txt, sample_input2.txt, sample_output1.txt, sample_output2.txt

^ ***Note:*** *These files might look weird if you open them in Windows using Notepad, because I created them in Linux, which handles end-of-line characters a bit differently from Notepad. Namely, it might look like the files are just one long line of characters. To get around that, you can open these text files in Code::Blocks, or transfer them to Eustis and use the* cat *command to print them to the screen.*

Deliverables

sudokode.c

^ ***Note:*** *Your filename must match this one exactly, including spelling and capitalization. You should test your code on Eustis, but submit via Webcourses.*

1. Problem Statement

Dave has become addicted to Sudoku, the latest puzzle craze in all the newspapers and bookstands. In case you don't know, a Sudoku is a simple number puzzle played on a 3x3 grid of 3x3 subgrids. Below is an example:

Initial Puzzle

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

Solution

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

The object of Sudoku is to place numbers 1 through 9 in the empty spaces such that no row, column, or 3x3 subgrid has any number more than once. An interesting property of Sudoku puzzles is that there is always only one possible solution, and it can always be determined using logic, without the need for guessing.

Although Dave is wild about Sudoku, he still comes up with incorrect solutions sometimes. Dave is tired of being made fun of by his more Sudoku savvy friends, so he has asked you to write a program to check his solution for him.

Dave would like to give you his completed Sudoku puzzle solutions to have you determine which ones are correct, and which are invalid. For a Sudoku solution to be correct, every row, column, and 3x3 subgrid of the puzzle must have each digit (1 through 9) exactly once.

2. Input and Output Format Specifications

Input will be tested from standard input. That means your program should read the input using scanf(), which will allow the graders to type input manually, or to redirect an input file to your program at the command line using, e.g.:

```
./sudokode.exe < sample_input1.txt
```

Each input file will contain multiple Sudoku solutions to check. The input will begin with a single integer, *n*, on a line by itself. Following that line will be *n* sets of 9 lines, each containing 9 space-separated digits. Each of these digits will be in the range of 1 through 9, inclusive. Each set of 9 lines of 9 digits represents one of Dave's potential Sudoku puzzle solutions.

For each Sudoku solution, print a single line with either the word “YES” or the word “NO” (in all caps, without the quotes). **Note that if your output is not capitalized correctly, or has extraneous spaces or punctuation, the output will be considered incorrect.** You can avoid losing points for this by using a *diff* tool (as described in labs and in Section 3, “Compilation and Testing (Linux/Mac Command Line),” below) to check that your output matches the sample output files included with this assignment exactly.

For posterity, here is a sample input and output for the assignment:

```
Sample Input                 Sample Output
2                            YES
3 5 7 6 4 8 9 1 2            NO
2 1 6 5 3 9 7 4 8
9 4 8 7 1 2 5 3 6
5 2 1 4 8 6 3 9 7
4 6 3 1 9 7 2 8 5
7 8 9 3 2 5 1 6 4
6 3 2 9 7 4 8 5 1
1 7 4 8 5 3 6 2 9
8 9 5 2 6 1 4 7 3
2 6 3 8 4 7 1 5 9
5 1 4 9 3 6 2 7 8
9 8 7 1 2 5 3 6 4
6 4 5 3 8 2 9 1 7
1 3 9 5 7 4 8 2 6
8 7 2 6 1 9 5 4 3
6 5 8 7 9 1 6 3 2
7 9 1 2 6 3 4 8 5
3 2 6 4 5 8 7 9 1
```

3. Compilation and Testing (Linux/Mac Command Line)

To compile at the command line:

```
gcc sudokode.c
```

By default, this will produce an executable file called `a.out` that you can run by typing:

```
./a.out
```

If you want to name the executable something else, use (for example):

```
gcc sudokode.c -o sudokode.exe
```

...and then run the program using:

```
./sudokode.exe
```

To redirect one of our sample input files to your program via standard input, use file redirection like so:

```
./sudokode.exe < sample_input1.txt
```

Running the program could potentially dump a lot of output to the screen. If you want to redirect your output to a text file in Linux, it's easy. Just run the program using the following:

```
./sudokode.exe < sample_input1.txt > my_output1.txt
```

This will create a file called `my_output1.txt` that contains the output from your program.

Linux has a helpful command called `diff` for comparing the contents of two files, which is really helpful here since we've provided sample output files. You can see whether your output matches ours exactly by typing, e.g.:

```
diff my_output1.txt sample_output1.txt
```

If the contents of `whatever.txt` and `sample_output1.txt` are exactly the same, `diff` won't have any output. It will just look like this:

```
seansz@eustis:~$ diff my_output1.txt sample_output1.txt
seansz@eustis:~$ _
```

If the files differ, it will spit out some information about the lines that aren't the same. For example:

```
seansz@eustis:~$ diff my_output1.txt sample_output1.txt
2c2
< NO
---
> YES
seansz@eustis:~$ _
```

4. Deliverables

Submit a single source file, named `sudokode.c`, via Webcourses. Be sure to include your name and PID as a comment at the top of your source file.

5. Grading

The expected scoring breakdown for this programming assignment is:

50%	Correct Output for Test Cases (strict *diff* matching will be enforced)
30%	Implementation Details (manual inspection of your code)
10%	Comments and Whitespace (employ good style)
10%	Compiles on Eustis (your program must compile, and it must have the correct filename in order to do so; capitalization and spelling matter!)

Your program must compile and run to receive credit. Programs that do not compile will receive an automatic zero, unless the compilation error is related to problems transitioning from Windows/Mac to our Linux server, in which case only a 10% penalty will assessed.

Your grade will be based largely on your program's ability to compile and produce the *exact* output expected. Even minor deviations (such as capitalization errors or extraneous spaces) in your output will cause your program's output to be marked as incorrect, resulting in severe point deductions. Please be sure to follow all requirements carefully and test your program throughly.

Note that although we have included sample input and output files to get you started with testing the functionality of your code, we encourage you to develop your own test cases, as well. Ours are by no means comprehensive. We will use more elaborate test cases when grading your submission.