2020/10/14 - Sudoku

13 Tháng Mười 2020 10:26 CH

SYNOPSIS

- Go over Lab 5.

LAB 5

- On Canvas, go to the "Lab 5" assignment and read the "lab5.html" file attached. "All" lab details will be there.

- There is only one part, but my guide splits it into THREE just so you stand a chance. Trust me.

- There is template code, like usual. Unlike lab4, it comes with the copy script this time. :)

SUBMISSION COMMAND

```
- tar -cvf lab5.tar Sudoku.cpp
```

BREAKING IT DOWN

- 3 PARTS:
  1.) Setup template, read txt, error checking
  2.) Sudoku Recursive Solve (Naïve) + "elapsed" function
  3.) We can do better... much better.

PART 1: SETUP TEMPLATE, READ TXT, ERROR CHECKING

- There is a class named omega. Do nothing with it for now...

- There is a function called elapsed. Comment it out for now...

- There are two functions named time_elapsed. Delete them. They aren't used.

- There is a class named sudoku. The following is done for you:
  - Constructor
  - solve() (Partially)
  - solve(...) (Pseudo-code)
  - display
  - read (Partially)
  - write (Both)

- For this "part 1", implement (in suggested order):
  - read
    - error_check_value(bool)
    - error_check_uniqueness()
      - check_row(int, int)
      - check_col(int, int)
      - check_square(int, int, int)

      OPTIONAL, BUT HIGHLY RECOMMENDED

  (indents = helper functions)

- Modify sudoku class for those functions. Both "error_check..." and "check_..." functions return a bool.

☆ READ

- Make a bool called "error" at the top and set it to FALSE. This variable shall be set to TRUE if an error occurs.
- Make an int called "line" and set it to 1 by default. After each iteration of the while loop, increment it.
    - You don't need getline. Assume 3 ints per line.
- In the while loop, check if i or j are not between 0 and 8.
    - If either are out, set "error" to TRUE and print an error message to stderr either via cerr or fprintf.
- At end of while loop, if "error" is TRUE, simply exit via "exit(0)".
- Call display() to print your amazing puzzle out.
- Call error_check_value. If it returns TRUE, set "error" to TRUE. (It takes a bool. Pass in TRUE.)
- Call error_check_uniqueness. If it returns TRUE, set "error" to TRUE.
- Once again, check if "error" is TRUE. If so, exit via "exit(0)".

★ ERROR_CHECK_VALUE

- bool error_check_value(bool zero_valid);
- Have a bool named error. It functions like in read.
- Loop through every cell, check if their values are within range. Valid range depend on zero_valid:
    - TRUE: 0 - 9 (For an unsolved puzzle)
    - FALSE: 1 - 9 (For a solved puzzle)
- If any cell goes out of range, set "error" to TRUE and print error to stderr via cerr or fprintf.
- Simply return error.

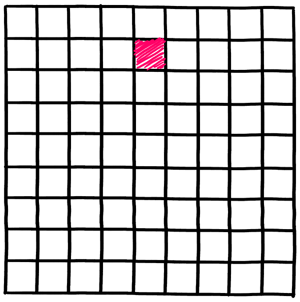
★ ERROR_CHECK_UNIQUENESS (AND CHECK_*)

- bool error_check_uniqueness();
- Many ways to do this.
- Make 3 functions:
    - bool check_row(int r, int v);
    - bool check_col(int c, int v);
    - bool check_square(int i, int j, int v);
- Their objectives are simple (should they choose to accept it):
    - bool check_row(int r, int v);
        - Check all cells in row r for value v. Return FALSE if v occurs in row r more than once. TRUE otherwise.

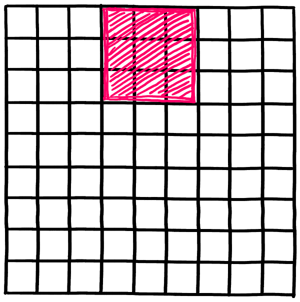
- `bool check_col(int c, int v);`
  - Check all cells in column c for value v. Return FALSE if v occurs in column c more than once. TRUE otherwise.

- `bool check_square(int i, int j, int v);`
  - Check all cells in inner 3x3 square located at (i, j) for value v. Return FALSE if v occurs in this inner square more than once. TRUE otherwise.

  - When I say inner 3x3 square, I mean
    - Assume (i, j) = (1, 4). Thus, here in RED:



    - Check this region for duplicates:



```
si = (i / 3) * 3   // start i
sj = (j / 3) * 3   // start j
```

- For every cell (i, j), run these three helper functions.
  - Run only if `game[i][j] > 0` btw...
- If ANY of them fail, a non-unique value was found. Print to error via `cerr` or `fprintf`.

- Simply return `error`.

PART 2: SUDOKU RECURSIVE SOLVE (NAÏVE) + "ELAPSED" FUNCTION

- Notice? There's two solve functions.
  - solve() - Recursion starter.
  - solve(...) - Actual recursion function.

- Arguments left ambiguous so you may choose your weapon of choice. However... it won't matter.

- Few choices:
  - bool solve(int cell_num)
    Makes recursive calls easier
  - bool solve(int i, int j)
    Makes calling more readable
  - bool solve(vector<int> cells, int c)
    What? Well, it's the same choice.
    If you want to survive part 3.

- Now that you made your "choice" (NUMBER 3, SERIOUSLY), we need a helper function called valid_values.
  - vector<int> valid_values(int i, int j)
  - Given a cell at (i, j) (game[i][j]), returns a vector<int> of all possible values that cell can be, given the current game board.
  - This is easier than it looks, if you wrote check_row, check_col, and check_square.
  - Back up game[i][j] into a temporary int.
  - Make a vector<int> to return.

    - Loop from 1 to 9 (inclusively), setting game[i][j] to each.
        - Run check_row, check_col, and check_square. If none fail, push that value to the vector.

    - Set game[i][j] to original value stored in temporary int.

- Note, we can store i and j into a single value and get those values back:
    - 2-to-1 = (i × 9) + j
    - 1-to-2 i = num / 9
      j = num % 9

★ SOLVE

- Make a bool called error.
    - It does exactly what you think it does.
- Make a vector<int> called cells.
    - Loop through 9×9 grid. Insert the numbers of the cells converted 2-to-1 by the equation above (i×9)+j into our newly created vector named cells... if game[i][j] is 0.

    - Yes, we are making a vector of cells that need to be filled in.

- //Unleash recursion like hell...

```
if (solve(cells, 0)) {
    display();
}
```

- C+P your error checking from read down below. But make sure it's error_check_value(FALSE) this time.

★ SOLVE (THE REAL ONE)

- bool solve(vector<int> cells, int c);

- In a nutshell, try all possible valid values for cell at cells[c] and go to the next cell. Recurse until DONE or FAILURE.

- First, check if c == cells.size().
  - Return TRUE if so. It means we solved it.

- Extract i and j from cells[c] using the 1-to-2 equation above.
  - i = cells[c] / 9
  - j = cells[c] % 9

- Grab the valid values for the cell via the valid_values helper function made earlier. Store it in a vector<int> values.

- If values.size() is 0, return FALSE. We messed up.

- Loop through all values in values:
  - Set game[i][j] to each value.
  - Try to solve(...) the next cell. Its return value matters:
    - TRUE: A solution was found! Return TRUE.
    - FALSE: Whatever it tried in those future calls clearly failed. Do nothing. Let the loop go to next iteration.
- Past the loop, we must assume failure.
  - Set game[i][j] to 0
  - Return false.
- Thought we were done? WRONG!
  - "Add code to the recursive solve function to ensure that the next cell considered has the lowest number of valid values of those left".
  - That's part 3... btw lol. Told ya to use vectors for the cells.

★ ELAPSED

- Notice const char *units[] global array.
- string elapsed(float duration, int i = 0);
- If duration < 0.1, recursive call with:
  - duration * 1000
  - i + 1
- If recursed, return the string the recursive call returns.

- If duration is larger, time to construct that string.
  - #include <sstream>
  - #include <iomanip>
  - Construct a string via ostringstream. Give it duration and units[i].
  - Return the string inside the ostringstream (it has a .str() function).

★ MAIN

- Change the lines where T0 and T1 are set to timer.get_sec().
- Modify that final cout statement to print elapsed, as well as seconds with fixed precision and 6 decimal places.

PART 3: WE CAN DO BETTER... MUCH BETTER.

- I hope you chose to write solve(...) with vectors, because it's going to be useful here.

☆ SOLVE (RECURSIVE ONE)

- We're going to modify the beginning.
- Just after the first check (which returns TRUE)... make two variables:
  - lowest_vv - lowest valid value
  - lowest_vv_i - index of cell w/ lowest valid value
- Loop from c (passed into function) to size of "cells", and call valid_values on every cell there. store the lowest one (by index) in lowest_vv_i, and store the return value from valid_values in lowest_vv.
- When that is done, you will have the index of the cell with the least possible valid values. SWAP cell[lowest_vv_i] with cell[c].
- That simple change shot my performance up by around 75x. It's a small change, but it really helps.