cs280-ab-recursion

This assignment will exercise your skills in following and implementing a simple recursive algorithm. The task is to implement a solution to the popular puzzle game Sudoku. From Wikipedia: "The objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 sub-grids that compose the grid (also called"boxes","blocks","regions", or"sub-squares") contains all of the digits from 1 to 9."

Here's a sample initial configuration of a 9x9 Sudoku board[1]

Instead of simply hard-coding the algorithm to manipulate a 9x9 board, we are going to extend the algorithm so that it can handle boards of any size (e.g. 9x9, 16x16, 25x25, etc.). There are many ways to solve a Sudoku puzzle, but, as it turns out, a recursive solution is the simplest to implement. (It is not the most efficient, though.) Your solution to this exercise will involve a technique called backtracking. Backtracking is a popular technique that computers use to solve problems by systematically evaluating all possible combinations. This backtracking algorithm is also similar to how one would solve a maze. The interface to the algorithm is simply a single call to a method of the class Sudoku named Solve:

```
void Sudoku::Solve() {
// An example sequence of operations may be:
// 1. Initialize some private data.
// 2. Call a recursive place_value method to place 1 (or A) at the
//    starting point, index 0 (row 1, column 1) or whatever the first
//    empty cell is.
// 3. Send MSG_FINISHED_OK if a solution was found, otherwise send
//    MSG_FINISHED_FAIL.
}
```

There is no return and the client will know whether or not a solution was found by receiving the appropriate message. Within your code, the Solve method should call a private recursive function whose task it is to place the value in the specified position (row/column). Name this function place_value or something similar and call it to put the value in the first cell. After the function places the value at a particular position, it must check to see if there are more empty cells left and take appropriate action. There are four possible situations:

Situation	Action
All cells have a value	The algorithm has successfully found the solution. Stop searching and send the appropriate message.
Cells are still open	Call place_value recursively to place a value in the next open cell.
Cells are still open and you can NOT place a value in the current cell. (The value is already the maximum.)	Remove the current value by "backtracking" to the previous cell that you placed a value, increment that value, and continue.
Cells are still open and you can NOT place any more values because you've exhausted all possibilities.	The algorithm was unsuccessful at finding the solution. Stop searching and send the appropriate message.

The algorithm continues until you successfully place a value in every cell on the board or you've tried all possible combinations and cannot place a value in all cells. A solution will have the number of "moves" or "guesses" that were required to find the solution.

Since the goal of this assignment is for you to demonstrate that you can correctly implement a specific algorithm, you will be allowed to use anything you want from the STL. You don't have to use it, but if you like, you may. Even then, you may only need vector to help with your implementation.

[1] /docs/9x9-example.png

Messages

You need to make sure that you are sending the appropriate message at the correct time. The parameters depend on the message being sent. See the header file for details.

- When you start the algorithm (the client calls Solve), you will send `MSG_STARTING`.
- After you place a value on the board, you will send `MSG_PLACING`.
- After removing a value from the board, you will send `MSG_REMOVING`.
- You will send `MSG_ABORT_CHECK` immediately before you place a value or remove a value. If this call returns true, you will terminate the search.
- If, after placing a value you have filled the board, you will send `MSG_FINISHED_OK` and terminate the search.
- If you do not find a solution after exhaustively checking, you will send `MSG_FINISHED_FAIL`.

The SUDOKU_CALLBACK function

Providing a callback function to the constructor enables you to receive notifications at any point during the algorithm's execution. This allows you to separate the algorithm logic from other programming logic (e.g. debugging code or visualization code.) This separation of logic is crucial when developing software for any non-trivial project. The types and order of the messages are evident by looking at the sample output from the sample driver. One of your goals is to match the output exactly. (This will determine whether or not you followed the specifications correctly.) At various times during the algorithm's execution, you must call the function that was provided by the client. The callback function requires 8 parameters, but all of them may not be used during a callback. (This is similar to how the lparam and wparam of a Windows message may go unused.) The return value is only used by one particular type of callback.

Note: Be sure not to reset the board after the solution is found. The driver will call your `Sudoku::GetBoard()` after the solution is found and it expects the board to still be valid. Also, you are not implementing the callback function. It is implemented and passed as an argument to you from the driver and you are to call it from your code with the required arguments (depending on the message type).

You will find that callback functions such as this can be a very helpful aid when debugging your code. By providing a callback mechanism, you can inspect data from outside of the algorithm. Graphical drivers that I use often make use of a callback function, which then keeps the algorithm very simple. You will find additional information on the web page for this assignment. Finally, PLEASE read through the driver and look at the output generated. As always most of your questions regarding the meaning of the counters will be answered by watching the output that is generated.

place_value function

This function takes a position (row/column or an index) and is called everytime you want to place a value on the board.

The function is initially called from the public `Solve` method with the position as the first empty cell (from the upper-left) on the board.

You'll need to add the appropriate calls to the callback function in various places in your code.

Pseudocode: (assume 9x9)

1. Place a value at the specified location. (The first time you will be placing the value 1 in the first empty position)
2. Increment the number of moves taken.
3. Increment the current move number.
4. If there is no conflict with any neighboring values then
    - If this was the last position left on the board:
        - The algorithm has finished, nothing left to do but to perform the proper callback.

- If there are more empty cells:
  - call `place_value` recursively with the location of the next empty cell. (This will put you at step #1 again.)

5. If there is a conflict:
   - If the value is less than 9:
     - Remove that value, increment it, and place it back in the cell and check for conflicts.
   - If the value is 9:
     - Backtrack by removing the 9.
     - Decrement the current move number.
     - Return from the function (it will return to the previous call and will likely be ready to try the next value in the previous cell)

The details are left for you to figure out. Of course, this is not the only way, or necessarily the "right" way to approach the algorithm. However, it is straight-forward and relatively simple. Feel free to modify or tweak the algorithm to suit your needs. Make sure to send the appropriate messages via the callback function so the driver can update the display properly.

Testing

A recommended sequence to test your code:

1. You should start with board3-3.txt[2]. It is an empty 3x3 board and you don't have to worry about pre-defined cells. Once you've successfully solved the empty 3x3 board, you should attempt board4-4.txt[3], which is an empty 4x4 board. It is even simpler (there are no backtracks) but will make sure that you can deal with different board sizes. You should be able to solve both boards in a fraction of a second.
2. Then try board3-6.txt[4], as it's pretty simple (45 moves, no backtracks). Then, you should attempt board3-0.txt[5], as it requires only 533 moves and has only 40 backtracks, but has some pre-defined cells.
3. Then, you should try board4-1.txt[6], which will finish in a few milliseconds.
4. Then, to test the 5x5 boards, you should try board5-1.txt[7] , which will also finish almost instantly.
5. Once you've tested these "easy" boards that require almost no time to run, you should try the boards that take considerably longer (e.g. board4-2.txt[8] and board5-2.txt[9]). The time required to solve the board is in a comment in the first line of each input file.
6. Finally, try the boards that have no solution, board3-8.txt[10] and board3-9.txt[11].

You will always start in the upper-left corner and work towards the bottom-right corner.

Be careful not to remove or change any of the original values that the board was started with.

The input to the Sudoku board uses a period `.` to represent empty cells. In your code, you will actually store an empty `(space)` character in place of the period. The periods make it easier to see where the empty cells are in the input.

Files Provided

The interface[12] .

[2] data/input/board3-3.txt
[3] data/input/board4-4.txt
[4] data/input/board3-6.txt
[5] data/input/board3-0.txt
[6] data/input/board4-1.txt
[7] data/input/board5-1.txt
[8] data/input/board4-2.txt
[9] data/input/board5-2.txt
[10] data/input/board3-8.txt
[11] data/input/board3-9.txt
[12] code/Sudoku.h

Support lib files[13] for parsing the command line in the driver. Realize that only the driver needs this API, not your code. Also, only Windows env will need this.

Sample driver[14] containing loads of test cases as usual.

There are a number of sample **inputs** and **outputs** in the **data** folder.

Some sample 9x9[15], 16x16[16] and 25x25[17] board images.

And also an animated gif[18] to provide some visuals of the operations.

Compilation:

These are some sample command lines for compilation. GNU should be the priority as this will be used for grading.

On Windows, you'll need all of the files from getopt.zip[19], to be present in your directory where you are working.) You don't need it with Cygwin (gcc/g++), Mac OS X, or Linux as the functionality is already present on those platforms.

GNU g++: (Used for grading)

```
g++ driver-sample.cpp Sudoku.cpp -o vpl_execution \
    -std=c++17 -Wall -Wextra -Wconversion -Wno-deprecated -pedantic
```

Microsoft: (Good to compile but executable not used in grading)

```
cl /Fems.exe driver.cpp Sudoku.cpp \
   /MT /W4 /O2 /EHsc /Za /WX /D_CRT_SECURE_NO_DEPRECATE getopt64.lib
```

Command line options

The driver sets up the output executable in a way that takes in command line options. Assuming the name of the executable is sudoku (or sudoku.exe). Typing sudoku at the command line will print this:

```
Usage: sudoku [options] input_file

Options:
 -a  --show_solution=X       show final board (0-OFF, 1-ON, default is ON).
 -b  --basesize=X            the size of the board (e.g. 3, 4, 5, etc. Default is 3)
 -c  --clear_screen          clear the screen between boards.
 -e  --show_boards_every=X   only show boards after every X iterations.
 -h  --help                  display this information.
 -m  --show_messages         show all messages (placing, removing, backtracking, etc.)
 -p  --pause                 pause between moves (press a key to continue).
 -s  --show_boards           show boards after placing/removing.
 -t  --symbol_type=X         the type of symbols to use, numbers or letters (0 is numbers, 1 is letters
 -w  --show_stats=X          show statistics after each move.
```

These options give you control over how much output to show. You can just show the initial and final boards or you can show the board after every move. This can potentially generate lots of output, but is essential for

[13] code/getopt.zip
[14] code/driver-sample.cpp
[15] 9x9-example.png
[16] 16x16-example.png
[17] 25x25-example.png
[18] docs/ani-9x9-full.gif
[19] code/getopt.zip

debugging your program. By pausing the program after every board, you can effectively single-step through the boards to see how your algorithm is working.

Sample input and output: (A `w` in the output name means it will show statistics. An `s` in the name means show all boards, and an `m` means to show all messages (starting, placing, removing, etc.) Because showing all boards produces a lot of output, it is only in the first example. Please work with the 9x9 example first to ensure that you are doing things correctly.

Deliverables

You must submit your program files (header and implementation file) by the due date and time to the appropriate submission page as described in the syllabus.

Sudoku.h

The header files. No implementation is allowed by the student, although the exception class is implemented here. The public interface must be exactly as described above.

Sudoku.cpp

The implementation file. All implementation goes here. You must document this file (file header comment) and functions (function header comments) using Doxygen tags as usual. Make sure you document all functions, even if some were provided for you.