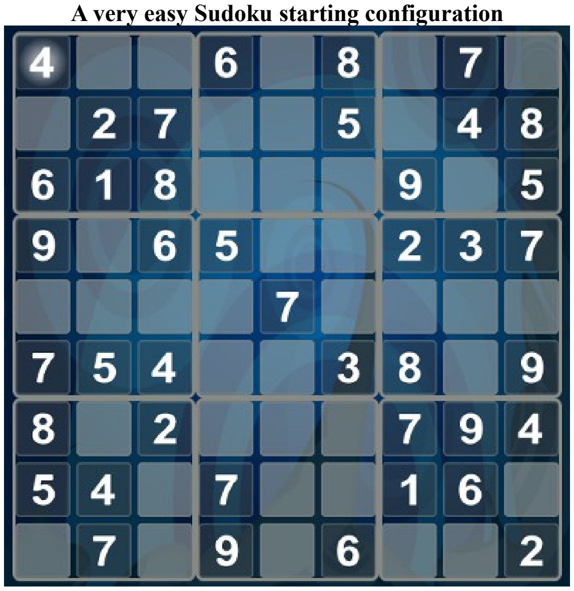
Programming Assignment #3

Due Wednesday, February 17, 2016 at 11:59 pm

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."

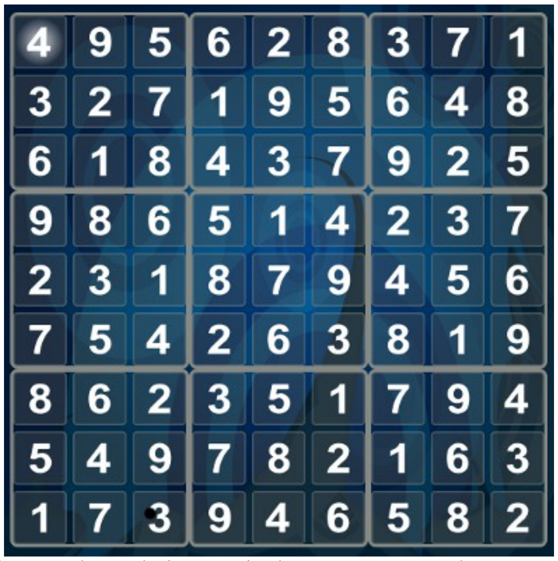
A very easy Sudoku starting configuration



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

An initial configuration of a Sudoku board. (`9x9.txt`)

Solution to the board on the left



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

The complete solution required 533 "guesses" and backtracked 40 times..

Details

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();
```

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. **More details are posted on the website.**

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 (I didn't use it), but if you like, you may. Even then, you may only need vector to help with your implementation.

More on the back ⟶

The 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.

```
typedef bool (*SUDOKU_CALLBACK)(
                          const Sudoku& sudoku, // the Sudoku board itself
                          const char *board,    // one-dimensional array of symbols
                          MessageType message,  // type of message
                          size_t move,          // the move number
                          int basesize,         // 3, 4, 5, etc. (for 9x9, 16x16, 25x25, etc.)
                          int index,            // index of the current cell (0-based)
                          char value,           // symbol (value) in current cell
                          int *dup_indexes      // index of each duplicate (maximum of 3)
                          );
```
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 in the driver and you are to call it from your code, passing the required arguments (depending on the message type).

	Parameter	Description
1	`const Sudoku& sudoku`	A reference to the Sudoku board itself.
2	`const char *board`	A pointer to the 1-D representation of the state of the board. Internally, you can store the board in any format you like, but it must be an array for the driver.
3	`MessageType message`	An integer (enum) representing the type of message. There are 6 message types defined in *Sudoku.h*. See the driver and website for more details.
4	`size_t move`	The current move number. *move* is the total move count since the algorithm began. Every time you place a value on the board, it is considered a move. See the output from the driver for details.
5	`int basesize`	3, 4, 5, etc. (for 9x9, 16x16, 25x25, etc.) You cannot hard-code these sizes into your code, but you can assume that you won't handle larger boards.
6	`int index`	The (0-based) index of the current cell. The index is based on the 1-D representation in the *board* parameter.
7	`char value`	The symbol (value) in the current cell that is being placed or removed.
7	`int *dup_indexes`	An array of 3 integers representing the indexes of 3 potential duplicates on the board. The array is sorted from low to high index. -1 means that there is no duplicate in that spot. There can only be 3 duplicates, one in the row, one in the column, and one "neighbor" in the sub-grid.

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. Most of your questions regarding the meaning of the counters will be answered by watching the output that is generated.

What to submit
You must submit your header file and implementation file (Sudoku.h and Sudoku.cpp) in a .zip file to the CS280 submission page.

Source files	Description
Sudoku.h	The header file for the Sudoku class. **No implementation is allowed in this file.** The public interface must be exactly as described above.
Sudoku.cpp	The implementation file. All implementation for the functions goes here. You must document the file (file header comment) and functions (function header comments) using Doxygen tags. Don't forget to include comments indicating why you are **#includ**ing certain header files.

Usual stuff
Your code must compile (using the compilers specified in this course) to receive credit. The code must be formatted as per the documentation on the website. Note that you must not submit any other files in the .zip file other than the 2 files specified. Details about what to submit and how are posted on the course website and in the syllabus.

Make sure your name and other info is on all documents.