

### Program 3: Sudoku

**Problem Statement:** Sudoku is a puzzle game where you have a board with a 9 by 9 arrangement of squares, subdivided into nine 3 by 3 blocks. Some of squares have predetermined values (numbers 1 - 9) while some are empty. The point of the game is to give values to those empty squares while following three simple constraints:

1. No row on the board has the same number twice
2. No column on the board has the same number twice
3. No 3 by 3 block on the board has the same number twice

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

Figure 1. Sudoku image by [The University of Melbourne](#). Puzzle by [Tim Stellmach](#) and solution by [Cburnett](#)

**Purpose:** The purpose of this program is to implement a set of classes to represent the Sudoku puzzle board and to create a basic solver using recursion with backtracking.

The program will read a string composed of 81 ASCII numbers 1-9 with unsolved spots to be indicated by a number 0. The program flow is: set up a 2D array puzzle board with the numbers contained in square objects, perform solving algorithm, and display output on “human-friendly” display.

**Multidimensional arrays:** Two-dimensional arrays are usually declared as `ObectType [int columnSize][int rowSize]`. In reality, it is really stored a “long one-dimensional array. It is stored in what’s called row-order, meaning by row.” (Zander, C++ arrays)

**Arrays as parameters:** In the case that an array is passed as a parameter, it is important to note that an array’s value is its address. It acts as if you are passing

it by reference, regardless of '&' use. In order to pass an array by value, you must use the term *const*. (Zander, Arrays as parameters to functions)

**Recursion:** Recursion will most play into checking the validity of a possible solution. For example, you want to check if 1 is a valid answer for a given location on the board. Say it is a valid input, you can move on to the next unsolved location. However, say that no value works for current location you are trying to solve, you would then need to backtrack and try a different value for the location where you initially entered 1. A simple pseudo-code outline of such an algorithm (from Program 3 Description) is provided:

```
Solve(row, column)
    Move to next square without a value
    if (row, column) past end of puzzle, return success
    foreach value from 1 through 9
        if value is legal, set square to it //tentative
            if solve(next row and column) succeeds
                return success // success
            endif
        erase square value // clear & try again
    endif
endfor
return failure //this triggers backtracking
end Solve
```

Input data			
Name	Description	Variable Type	Assumptions
unsolvedSudokuData	This will be 81 ASCII characters (numbers 0-9). Unprocessed, it may contain spaces and invalid characters. It will be read by cin (>>) in row-major order.	String	* May or may not contain special characters or spaces. * Should assume, the Sudoku passed are valid and can be solved.

An example input data may look like (from Program 3 Description):

```
423751968759683124168249357945362871872915436316478295537194
682691827543284536019
```

Output data			
Name	Description	Variable Type	Constraints
solvedSudokuPuzzle	This will be a "human-	String	* Should be solved.

e	friendly” display of the solved Sudoku puzzle. It is 11 lines of text with each line containing numbers in a row of puzzle separated by single spaces along with ‘ ’, ‘+’, and ‘-’ to separate the nine 3 by 3 blocks. Be outputted with cout (<<).		
---	---	--	--

An example output data may look like (from Program 3 Description):

```

4 2 3|7 5 1|9 6 8
7 5 9|6 8 3|1 2 4
1 6 8|2 4 9|3 5 7
-----+-----+-----
9 4 5|3 6 2|8 7 1
8 7 2|9 1 5|4 3 6
3 1 6|4 7 8|2 9 5
-----+-----+-----
5 3 7|1 9 4|6 8 2
6 9 1|8 2 7|5 4 3
2 8 4|5 3 6|7 1 9

```

## Error handling:

- The program must handle errors and display appropriate messages for each error.
- The program must verify that they are working with exactly 81 integers (numbers 1-9 and 0's to indicate empty spaces)
- The program must verify that each integer is properly constructed as a square in the correct location, to create a row-major grid that accurately resembles data.
- The final solution should contain all valid values and no unsolved squares.
- The “human-friendly” display should add readability without manipulating the values in the grid.

## Implementation & Plan:

Puzzle class		
Name & Description	Implementation plan	Test plan
<b>Operator&lt;&lt;:</b> Overload the stream input operator to read puzzles in a compact format containing 81	<pre> Friend istream&amp; operator&lt;&lt;(istream&amp; i, char[] string){     // create char[] stringCopy     // create new empty     Square[][] </pre>	<ul style="list-style-type: none"> <li>- A string containing any non-integer values should be ignored.</li> <li>- The entire string should</li> </ul>

integers (integers 0-9). It will pass the string into createGrid.	<pre>// call createGrid with stringCopy as parameter and checks if Square[][] contains 81 objects (with int values). // Square[][] as return }</pre>	total 81 characters. The program should print an error message and terminate if it fails to meet this requirement.
<b>CreateGrid:</b> Will take in a string of integers (81 characters in length) and then create a puzzle grid in row-major order.	<pre>Square[][] createGrid(char[] oCopy){     // for each character in copy     //remove special characters     // counts and sets startingVal corresponding to number of 0's present prior to solving any squares     // update class member     // return Square[][] }</pre>	<ul style="list-style-type: none"> <li>- Order, arrangement, and size of string should be maintained.</li> <li>- Each number should be constructed as a square, and take part as 1 unit of the 2d array.</li> </ul>
<b>Get:</b> Takes in x and y location on the grid (zero-based) and returns the value at that location.	<pre>int get(int x, int y) {     // return puzzle[x][y].getValue }</pre>	<ul style="list-style-type: none"> <li>- Entering an invalid location should print an error message.</li> </ul>
<b>Set:</b> Takes in x and y location as well as a value and attempts to set the value at that location.	<pre>bool get(int x, int y) {     // puzzle[x][y].setValue     // checks if this change is a valid move     // if valid, return true, else return false }</pre>	<ul style="list-style-type: none"> <li>- Entering an invalid location should print an error message.</li> <li>- If value is a legal value, it will update the value and the it will return true.</li> <li>- If it fails, it will return false.</li> </ul>
<b>operator&lt;&lt;:</b> This will display a "human friendly" grid via stream output. It should be 11 by 11 characters containing ' ', '+', and '-' to separate 3 by 3 blocks.	<pre>Friend ostream&amp; operator&lt;&lt;(ostream&amp; o, int[][] puzzle){     // create new Square[][]     // use nested forloop to iterate through 2d array, copying each value and adding special characters at appropriate locations     // prints new, modified 11x11 array, Square[][] }</pre>	<ul style="list-style-type: none"> <li>- The order and arrangement of the squares should remain unchanged when adding special characters.</li> </ul>
<b>Size:</b> Returns the number of variable entries in the puzzle (the values written in as parts of possible solution).	<pre>Int size(){     // returns startingVar, which was initialized in createGrid }</pre>	<ul style="list-style-type: none"> <li>- The size should be the same throughout the game.</li> </ul>
<b>numEmpty:</b> Returns the number of empty spaces.	<pre>Int numEmpty(){     // create new int counter     //iterates through puzzle[][] using nested forloop and increments count every time it gets a 0 value     // returns count }</pre>	<ul style="list-style-type: none"> <li>- Should correspond to the number of zeros still present on the board.</li> </ul>

<b>SolvePuzzle:</b> Performs recursive algorithm to solve puzzle. It will take in an unsolved puzzle (unformatted with 0's) and will return a solved puzzle. If it cannot solve the puzzle, it will return the original grid and print an error to tell the user that the puzzle was not able to be solved.	Please see recursive algorithm described in problem statement.	<ul style="list-style-type: none"> <li>- If the input puzzle (unsolved 2d array) has values that are not legal (a value appears more than once on row, column, or block), this method should not complete.</li> <li>- This will solve zeros from the most upper left corner to the lower right corner as it is implemented as row-major array.</li> </ul>
<b>IsLegal:</b> Checks to see if the move (takes in x and y coordinate along with value) is a valid move by making sure it meets constraints: no number appears twice in a given row, column, or 3 by 3 block. Returns true if legal and false if not.	<pre> Bool isLegal(int x, int y, and int value) {     // nested for loop to     check if column constraints     are met     // nested for loop to check     if row constraints are met     // check if 3x3 block     constraints are met     // if the value at the     given location violates any     constraints, it will return     false. Else it will return     true. }           </pre>	<ul style="list-style-type: none"> <li>- If it receives an invalid location, it should not continue to perform.</li> </ul>
<b>IsSolved:</b> Checking whether values are meeting constraints was performed in isLegal method. In this method, we will return true if numEmpty is zero and false if it is equal to or greater than 1.	<pre> Bool isSolved(){     // if numEmpty is greater     than zero, return false. Else     return true }           </pre>	<ul style="list-style-type: none"> <li>- A grid containing one or more 0's should return false.</li> </ul>

Square class		
Name & Description	Implementation	Test plan
<b>GetValue:</b> Returns the current value in the square	<pre> Int getValue(){     // returns value }           </pre>	<ul style="list-style-type: none"> <li>- All returns should be a single integer (0-9)</li> </ul>
<b>SetValue:</b> Takes an integer (integers 1-9) and sets the value to a given argument. However, this does not enforce any constraints.	<pre> Void setValue(){     // Changes value. }           </pre>	<ul style="list-style-type: none"> <li>- If a user tries to pass an integer greater than 9 or any special characters, it should not execute and print an error.</li> </ul>

# UML Diagrams:

Puzzle
<ul style="list-style-type: none"><li>- constraint: int</li><li>- puzzle: int[constraint][constraint]</li><li>- startingVar: int</li></ul>
<ul style="list-style-type: none"><li>+ operator&gt;&gt;();</li><li>+ createGrid(unsolvedSudokuData: String): int[][]</li><li>+ get(x:int, y:int): int</li><li>+ set(x:int, y:int, value:int): boolean</li><li>+ operator&lt;&lt;()</li><li>+ size(): int</li><li>+ numEmpty(): int</li><li>+ isLegal(x:int, y:int, value:int): Boolean</li><li>+ isSolved(grid: int[][]): Boolean</li><li>+ solvePuzzle(unsolvedBoard: int[][] ): int[][]</li></ul>

Square
<ul style="list-style-type: none"><li>- value: int</li><li>- flag: bool</li></ul>
<ul style="list-style-type: none"><li>+ getValue(value: int): int</li><li>+ setValue(input: int): void</li></ul>