**Program 3: Sudoku**

**Definitions**

Sudoku is a type of puzzle that originated in Japan. It consists of a block of 9x9 numbers ranging from 1-9, which are further divided into 3x3 blocks. In order to solve the puzzle, all empty boxes must be filled so that there are no duplicate numbers in a single row, column, or block. An example of a Sudoku puzzle is pictured below.

A picture containing text, crossword puzzle

Description automatically generated

**Specification**

**Purpose:** The purpose of the program is to implement a set of classes that represent a Sudoku puzzle. The classes will then be used to take a Sudoku puzzle as input and solve the puzzle via a backtracking algorithm.

**Assumptions:** Boxes that already have numbers in them are constant and not allowed to change. Empty boxes can only be filled with numbers ranging from 1-9. No duplicate numbers are allowed in the same row, column, or 3x3 box. Depending on input, there may be no solution to the puzzle if there is a problem in the way the constant numbers are set up. ADD to this

**Special Cases:** There may be no solution to the puzzle that exists, depending on input. Ensure that the input only consists of characters (0-9) and that the number of characters is 81, or else give an error message.

**Input/Output:** The input will be in the form of a line of 81 ASCII characters which will be used to build the puzzle. The output will be the solved Sudoku puzzle with all boxes filled with the correct numbers. If there is no solution possible, a message will be outputted to the user.

An example output would look like:

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

**Testing:** Testing will be performed incrementally on each class. Unit testing will be performed first and then integrated testing once all unit tests pass. The integrated tests will be set up so that various puzzles are created and tested for a solution.

**Error Handling:** The only error handling that will be done in this program is if no solution is possible. FINISH this

**Design**

The program will consist of four classes. They are as follows: Puzzle class and the Square class (which will be nested within the Puzzle class). The Puzzle will have a backtracking algorithm that will be named Solve(). “**Backtracking** is a technique based on algorithm to solve problem. It uses recursive calling to find the solution by building a solution step by step increasing values with time. It removes the solutions that doesn't give rise to the solution of the problem based on the constraints given to solve the problem.”

**Square Class:** The square class will represent each box within the 9x9 puzzle. It will hold an integer value (integers 1-9).

Private:

**int value:** integer ranging from 0-9 with 0 indicating that the square is not fixed but variable

**bool fixed = true:** if fixed is true, then the square is not allowed to change its value

Public:

**Square() :** constructor that sets the default value of value to -1.If value is set to 0, bool fixed is true.

**~Square():** destructor that destroys the object

**int getValue():** returns the current value of the square object

**bool setValue(int newValue):** sets the value to the newValue regardless if it is the correct value for that square. PRE: value with original value POST: value is now assigned with the newValue entered. This funct-ion returns true if the value has been changed.

**void setFixed(bool newFixed):** sets the bool fixed to either true or false based on the newFixed input

**bool getFixed():** returns true if fixed is true and false if it is false

**Puzzle Class:** The puzzle class will consist of a 2-D array of squares. The array will have 9 rows and 9 columns.

Private:

**int numOfVariables:** current number of empty squares

**int numEmptyVariables:** starting number of empty squares

**Square puzzleGrid [9][9]:** 2D array that holds all 81 squares and represents the Sudoku game

**int get(int row, int col):** returns the value of the square at the given row and column

**bool set(int row, int col):** sets the value at the given row and column if the value is allowed at that location

returns true if the value was set. This function calls isSafe() to check if setting the value is safe

**bool isSafe(int row, int col, int value):** determines whether the squares value is safe at the given location. Returns true if the value is allowed at the given location.

**bool numberInBox(int row, int col, int value):** returns true if the value exists in the same box as the square at the given location

**bool numberInCollumn(int col, int value):** returns true if the value exists in the same column as the square at the given location

**bool numberInRow(int row, int value):** returns true if the value exists in the same row as the square at the given location

**bool isVariableEmpty(int row, int col):** returns true if the square is empty

**bool findNextEmpty(int &row, int &col):** goes through the puzzle to find first empty and returns true if found

Public:

**Puzzle():** constructor that initializes the class member variables

**~Puzzle():** destroys the puzzle object

**bool Solve():** attempts to fill the empty squares in the 2D array and solve the Sudoku puzzle. The function is gone over in more detail in the non-trivial implementations section.

**int size():** returns the number of squares that are variable (have the value 0)

**int numEmpty():** returns the current number of empty squares

**friend ostream &operator<<(ostream &os, const Puzzle &puzzle):** outputs the puzzle in a certain format pictured below in the non-trivial method implementation

**friend istream &operator>>(istream &in, Puzzle &puzzle):** takes input as 81 numbers and sets the squares’ values to the numbers in the 2D array

**Non-trivial methods implementation:**

**Solve()**  
  Move to next square without a value via findNextEmpty(int &row, int&col)  
 if (row, column) past end of puzzle, return success  
 **foreach** value from 1 through 9  
 **if** value is legal, set square to it  
 **if** Solve(next row and column) succeeds  
 return success  
 **endif**  
 erase square value  
 **endif**  
 **endfor**  
 return failure  
**end** Solve

**Friend ostream &operator<<(ostream &os, const Puzzle &puzzle):**

loop through columns then increment row

if the column is at 2 or 5, include “|” at the end of the cout statement

if at last column in row then add an endl

or else print out the value as normal

if the row is 2 or 5, include "------+-----+------" at the end of second or fifth row.

Return ostream os

**Example output for the ostream function is pictured below:**

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

**friend istream &operator>>(istream &in, Puzzle &puzzle):**

string numbers;

cout << “Enter the puzzle”;

in >> numbers;

if (numbers.length() != 81){

cerr << “Invalid puzzle entered”;

}

Else {

For(i: 0-numbers.length){

If(numbers are not between 0 and 9) cerr << “Incorrect puzzle entered”;

{

Else increment number of variables if number is 0

Loop through columns and rows

Set the squares in the 2D array value to the numbers from input

Return the istream in

**Input for the istream is pictures below:**

423751968759683124168249357945362871872915436316478295537194682691827543284536014

420751968759683124168049357945362871872915436316478295537194682691827543284536014

420701908709683124168049307945362071872915036316478295507194682601827503284506014

**Implementation Plan**

**Unit Test:**

Individual tests will be performed on the following functions in the order of:

**Square:**

1. Initialize a square
2. setValue()
   1. setValue to different types of integers
3. getValue()
   1. should return the values that the square was set to
4. setFixed()
   1. set to true and print
   2. set to false and print
5. getFixed()
   1. should return what fixed was set to

**Puzzle:**

1. Initialize a puzzle
2. Operator>>()
   1. Take input as a Sudoku puzzle
   2. Input a bunch of different kinds of puzzles including ones that are not possible to solve or do not have the correct number of variables
3. Get()
   1. Try to get different squares values based on their row and column location
4. Set()
   1. Test setting the value to stuff it should not be and also with values that are allowed for that location
5. isSafe()
   1. will test the following methods
      1. numberInCol
      2. numberInRow
      3. numberInBox
   2. test with all possible combinations
      1. ex: number is in col, but not row or box
      2. number is in all three etc.
6. isVariableEmpty()
   1. should return true if the squares value is 0
   2. test both for true or false
7. size()
   1. should return the number of current empty squares
   2. try setting a few that are empty to valid positions and then test size() again
8. numEmpty()
   1. should return the number of squares that started out empty
9. Operator<<()
   1. test the output of a sudoku puzzle

**Integration Tests:**

A comprehensive test will be run that will just run the solve() method

1. Initialize a puzzle
2. Take input as a Sudoku puzzle
   1. Input a bunch of different kinds of puzzles including ones that are not possible to solve or do not have the correct number of variables
3. Run the solve method on the various types of puzzles

**Works Cited:** [**https://canvas.uw.edu/courses/1494706/pages/program-3-description-solving-sudoku-with-backtracking?module\_item\_id=13974828**](https://canvas.uw.edu/courses/1494706/pages/program-3-description-solving-sudoku-with-backtracking?module_item_id=13974828)

<https://www.tutorialspoint.com/introduction-to-backtracking>