Solving Sudoku With Backtracking

Specifications

**Problem statement:**

For this assignment, I will be creating a sudoku puzzle. This sudoku puzzle is made up of 81 squares, so a 9x9 puzzle. The puzzle is then further broken down into a 3x3 blocks. The values in each row, column, and 3x3 block will be integers form 1 through 9 and each row, column, and 3x3 block cannot have a repeating value. The puzzle will be made up of a 2-dimensional array with each row/column index representing a square in the puzzle. A square object from the square class will hold the numerical int value to be inserted into the puzzle and will indicate if it is *fixed* meaning that it is given at the start of the puzzle or if the value is *variable* meaning that it is a value to be added into the puzzle through a backtracking algorithm. A value is inserted into the puzzle by moving to each empty square and checking If the value appropriately fits there through a set() method. From there, there are some other basic methods like size() which returns the size of the puzzles variable values, numEmpty() which returns the numbers of squares that are empty, get() which returns the int value of a square at a certain location etc. These methods are further described in the design section.

**Assumptions:**

* I am assuming the overloaded stream output operator will be defined as a friend as it needs access to the puzzle class to print the puzzle.

**Program Inputs:**

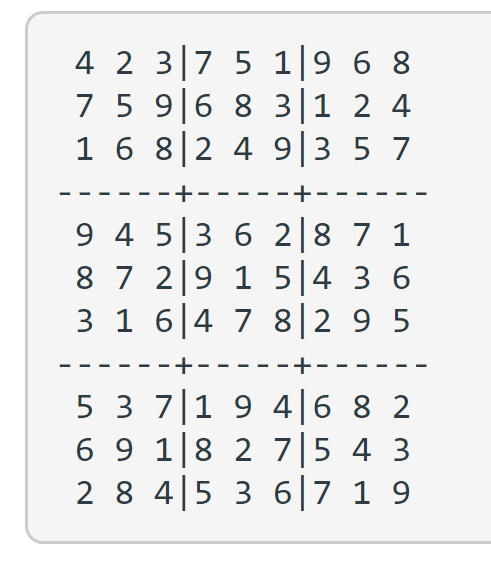
The input will be read from cin using an overloaded stream input operator. Which will intake a sequence of 81 numerical digits. The numbers will be read one by one and correspond to upper left square of the first row and will work its way to the right and will then work its way to the very left square of the next row and so on. The specification and algorithm of this operator is further described in the design section.

Example input:



**Program Outputs:**

An example output is shown to the right and is printed with cout using an overloaded stream output operator. Each 3x3 square is divided using dashes lines ( | and - ). Each row and column are lined up, so it visually make sense.



Design

For this program, I will have 3 files: **Sudoku.h**, **Sudoku.cpp**, and **Main.cpp**. Sudoku.h will contain all the headers for the necessary classes and methods. Sudoku.cpp will contain the implementation of everything defined in Sudoku.h and Main.cpp will be used to test methods, inputs, and outputs along the way.

I will create 2 classes in Sudoku.h: **class Puzzle** and **class Square**. The Square class will be nested within the puzzle class.

Note: I have not decided if a destructor Is necessary as I do not think I need to call *new* anywhere but that will be accounted for if I end up needing to.

**Square Class:**

**struct Square contains:**

* int value which is the number represented in each square of the puzzle.
* int array[rowSize][columnSize]; which is a 2-dimensional array which represent the squares and hold the value of each square.
* Int rowSize const; which is the number of rows the puzzle has. This value is inserted into the array.
* Int columnSize const; which is the number of columns the puzzle has. This value is also inserted into the array.
* Array will need to be set to null at construction time as it has no values yet.

**+ int getValue();**

* This method returns a value associated to a square.
* Parameters: none. It is already connected to a singular square.
* Returns: int which is the value of the square. If the square has no value set then It returns -1.

**+ void setValue(int rowIndex, int columnIndex, int newValue);**

* This method takes value and sets it to the square. it discards constraints among values in the same row, column, or 3x3 block.
* Parameters: an int representing the value to be set at the specific location.
* Returns: nothing it Is void.

**Puzzle Class:**

**istream& operator>>(istream& is);**

* Overloaded istream input operator. Reads the sequence of numbers and parses them so that they are separated. The fillPuzzle() method is then called which fills the puzzle with its corresponding values
* Parameters: intakes istream&
* Returns**:** overloaded input operator

**+ void fillPuzzle(int number);**

* This method inserts each number starting from the top left square and working its way to the end of the row and moving to the next. If the number is 0, meaning it is variable, then it sets the specific square to null instead of a 0.
* Parameters: an int representing the number to be added into the puzzle.
* Returns nothing.

**+ friend ostream& operator<<( ostream& os, const Puzzle& puzzle);**

* This is an overloaded ostream output operator and is declared as friend because it needs access to the puzzle class’s elements to print the puzzle.
* Parameters: intakes ostream& and Puzzle& which is declared const as it cannot be changed when printing,
* Returns: an overloaded output operator which can print the whole puzzle in the correct format described above.

**+int get(int rowIndex, int columnIndex);**

* This method takes a row and column location in the 2-dimensional array and returns the current int value at that location.
* Parameters: int values that represent the index of the row and column in the 2- dimensional array. This is considered the location/square in the puzzle we are trying to get the value for.
* Returns: an int representing the value at the specific location. if there is no value set at that location, returns -1.

**+bool set(int rowIndex, int columnIndex, int newValue);**

* This method takes a row and column location in the 2-dimensional array and sets the given value at that location. it has constraints so that each row, column, and 3x3 square do not have repeating values.
* Parameters: int values that represent the index of the row and column in the 2-dimensional array. This is considered the location/square in the puzzle the value is being set for. Also takes an int representing the value to be set at the specific location.
* Returns: a bool. Returns true if the value is legal for that square and does not repeat in the row, column, or block. In that case the value for that square is successfully set. Returns false if there is no spot to insert or if it fails.

**+bool solve(int row, int column);**

* This is a recursive method and uses backtracking to fill in the empty squares and complete the puzzle.
* Parameters: none
* Returns: bool which indicated if the puzzle was successfully solved. If for some reason the puzzle was not able to be solved it returns false and also displays an error message.

**+ int size() const;**

* this method counts and returns the total number of squares in the puzzle that are not fixed before the puzzle is filled in. Meaning that they contain the square is null. It is declared const as the value cannot change even after squares are filled.
* Parameters: none.
* Returns: an integer value of the total number of variable squares.

**Int numEmpty();**

* this method computes the number of squares in the puzzle that are currently empty.
* Parameters: none.
* Returns: int which represents the number of empty squares.

Algorithms

**+int getValue();**

**If**(value is not null)

return value;

**else** return -1;

**+ void setValue(int newValue);**

set value to newValue;

**+int get(int rowIndex, int columnIndex);**

**for**( i starting at 0 and iterating through the rows; i++)

**for**( j starting at 0 and iterating though columns; j++)

**if**(i == rowIndex && j == columnIndex)

break and return value at that index

**+bool set(int rowIndex, int columnIndex, int newValue);**

**for**( i starting at 0 and iterating through the rows; i++)

**for**( j starting at 0 and iterating though columns; j++)

**if**(i == rowIndex && j == columnIndex)

if(row does not have repeating value &&

column does not have repeating value &&

block does not have repeating value)

//this might be vague but I still need to figure out how this will translate in actual code

set the value of square at that index and return true;

otherwise return false;

**+ int size() const;**

Int count = 0;

**for**( i starting at 0 and iterating through the rows; i++)

**for**( j starting at 0 and iterating though columns; j++)

**if**(value equals null)

count++;

return count;

**Int numEmpty();**

Int count = 0;

**for**( i starting at 0 and iterating through the rows; i++)

**for**( j starting at 0 and iterating though columns; j++)

**if**(value equals null)

count++;

return count;

**+ void fillPuzzle(int number);**

for(row starting at zero and traversing to rowSize)

for(column starting at 0 and traversing to columnSize)

value[row][column] = number;

**+bool solve(int row, int column);**

**If** current square is not empty then move to next empty square – using 2 for loops

**Foreach** value from 1 – 9

**If**(value does not repeat in row, column, 3x3)

Set value to that specific square.

**If**(call method again with next row and column)

Return true; //success.

Return false; //failure.

Implementation plan

1. Creating the square class with its struct/constructor to create the 2 dimensional array and defining some things.
2. Implementing getValue() and setValue() which are simple algorithms and make sure they are properly getting and setting the correct values.
3. Nesting square class into puzzle class and changing the implementation in .cpp to Puzzle::Square
4. Implementing the ostream and istream operators and making sure the ostream operator matches the output shown above.
5. Making sure that istream is working properly and is able to break apart each number from the sequence.
6. I will have to implement fillPuzzle() alongside the istream operator to fill the puzzle with the individual numbers the istream breaks apart.
7. Will then work on implementing get() and set(). With these methods it is necessary to make sure I am getting and setting at the right row and column location. I can do this by stepping through and printing the puzzle to visually check that the right things are happening.
8. Then implementing size and numEmpty. These are pretty similar but I will need to assure that size does not change even after the puzzle is solved.
9. Lasty, I will implement solve() after I have the more basic functions working.

the structs and constructors and square class are pretty self-explanatory. I am going to cover the deeper test cases for the puzzle class.

solve() method I am going to be testing some cases:

* Trying to solve a filled puzzle – this should return false and should display an error message as the puzzle is already solved and completed.
* There is only one square remaining and the algorithm has hit base case. Checking while stepping over to see how the algorithm responds.
* Giving the puzzle bad input that cannot be solved – making sure it returns false and displays error message.
* I know that checking every 3x3 block is going to be tricky so I will pay extra attention to that by setting breakpoints and stepping over.
* Overall setting break points and checking how the puzzle is being traversed and how values are being set.

set() method tests:

* Setting a value for an index that already contains a fixed value – should return false and not do anything.
* setting a value in an empty square that is already in the column, row, or block – should not work and return false.
* Setting a value that does not yet exist in the row, column, or block. This should fill in the value and then return true.
* Setting a value in the middle of the puzzle and checking to see how the algorithm responds.

size() method tests:

* Calling size from main before the puzzle is touched and then calling it again after the puzzle has some solved values and checking to see if the return value stays the same.
* Making sure the size method is only counting squares that are empty which will be indicated as the square being set to null.