**SUDOKU PROJECT**

**1: PROBLEM STATEMENT**

**In this project, we are to create 2 parts of a program, one to create a sudoku grid and one to solve it. The grid will be 9x9 and follows the same rules as standard sudoku game.**

**2: REQUIREMENTS**

**A: ASSUMPTIONS**

**- the grid is 9x9 and 3x3 sub-grids**

**- format of the input is correct**

**- 0 will be treated as invalid**

**- user will input input and output file names**

**B: REQUIREMENTS**

**- The program will read data to output filename entered by user and will check for file existence then output sudoku grid into the file. program will use a random generator to generate a different game each time the program is launched.**

**- program will start with a welcome screen displaying the message “ Welcome to your Sudoku game”**

**- user will be prompted “Please enter name of file where the game will be outputted to: “ and program will check for existence. If file name does not exist, display error message “[file name] filename is not valid or does not exist, program will now terminate” and terminate the program. If file exists, game grid will be outputted to that file.**

**- once validity for existence is checked, program will ask for the difficulty level of the game “ choose the game difficulty you want:**

**- choices will be easy (1), medium (2) and hard (3), program will then check to make sure that a valid choice is been picked, if not error message “ invalid choice, please choose a valid #: “, program will cycle until user makes a valid choice.**

**- program will have 17 hints for hard, 21 hints for med and 25 hints for easy.**

**- program will then use a random generator to make a full grid, then it will remove a random set of numbers based on the difficulty level chosen using the following rules:**

**1. generate a full grid of numbers (fully filled in). Cannot randomly generate numbers to fill in the grid. Need to make sure that the numbers are positioned on the grid following the Sudoku rules. To do so, use the sudoku solver algorithm applied to an empty grid. Need to add a random element to this solver algorithm to make sure that a new grid is generated every time it is run.**

**2. from the completed full grid, remove 1 value at a time.**

**3. for each value removed, run the sudoku solver algorithm to see if the grid can still be solved and to count the number of solutions it leads to.**

**4. if the resulting grid only has one solution, carry on the process from step 2. If not need to put the value taken away back in the grid.**

**5. repeat the same process from step 2 several times using a different value each time to try to remove additional numbers, resulting in a more difficult grid to solve.**

**- program will then output the unsolved grid to the file user had chosen.**

**The solver part of the program will run after creator part is done.**

**- user will also be promoted “ please enter name of file where the sudoku grid is: ” input the output file name and if it does not exist terminate the program.**

**- if input file is validated, program will validated data as it is being read from the file, if data is not between 1 and 9, error message “Invalid data, program will now terminate” and stop the program.**

**- program will go through and try to solve the game, if successful, program will test for other possible solutions.**

**- if no solution exist, program will display “ grid can not be solved” and terminate the program.**

**- if unique solution or multiple solution, display all solutions.**

**-ALL ACTIONS ARE PRINTED TO SCREEN AND SAVED TO AN OUTPUT FILE**

**3: DECOMPOSITION DIAGRAM**

**4: TEST STRATEGY**

**-use valid data**

**-use invalid data**

**5: TEST PLAN V1**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Test Strategy** | **Test Number** | **Description** | **Input** | **Expected Output** | **Actual Output** | **Pass/Fail** |
| Invalid data | 1 | Use invalid file name for input file | Doesnotexist.txt | “Error, file does not exist” |  |  |
| Invalid data | 2 | Use valid file that exists and invalid difficulty option | Fileexist.txt and use difficulty F | Error message” invalid difficulty please try again” |  |  |
| Valid data | 3 | Use valid file that exists and valid difficulty option | Fileexist.txt and use difficulty M | Program will run and process data and move on to solver side |  |  |
| Invalid data | 4 | Input invalid name for solver to process unsolved grid | Doesnotexist.txt | Error message “ File does not exist, program will now terminate” |  |  |
| Invalid Data | 6 | Input a correct unsolved grid name and modify data to have invalid #s (higher than 9 | Fileexist.txt | Error message “file contains invalid data, program will now terminate.” |  |  |
| Valid data | 7 | Correct and valid data inputted into file | Valid data including input file name | Program will process unsolved data and will show results if grid is solvable |  |  |

**Solver:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Test Strategy** | **Test Number** | **Description** | **Input** | **Expected Output** | **Actual Output** | **Pass/Fail** |
| Invalid data | 1 | Use file that does not exist | Doesnotexist.txt | “input filename is not valid or does not exist, program will now terminate” |  |  |
| Invalid data | 2 | Invalid File Element | -2 | ERROR Invalid File Element |  |  |
| Valid data | 3 | Invalid Column | InvalidColumn.txt | Error message “ program will now terminate” |  |  |
| Invalid data | 4 | Invalid Row | InvalidRow.txt | Error message “ program will now terminate” |  |  |
| Invalid Data | 5 | Invalid Sub Grid | InvalidSubGrid.txt | Error message “ program will now terminate” |  |  |
| Valid data | 6 | Correct and valid data inputted into file | Valid data including input file name | Program will process unsolved data and will show results if grid is solvable |  |  |

**6: INITIAL ALGORITHM FOR CREATOR**

**- Create a program that will start by asking user for a file name where sudoku grid will be sent to**

**- user will be prompted to choose difficulty of game easy, medium and hard.**

**- program will try to open the file and if it does not open it will output error message and terminate the program**

**- if file opens.. on the back end we will have a switch algorithm to take user input and process it accordingly**

**- sudokuCreator will create a 9x9 2d array and will create the full sudoku grid then based on the user input it will use the removeNumber algorithm to remove random numbers from the grid numbers to be left are 25 for easy, 21 for medium and 17 for hard.**

**- we will use mt19937 random generator to ensure randomness in the sudoku grid**

**- use a for loop to make sure we have separate squares for 3x3 sub grids**

**- 3 functions one to check validity of number going into grid. Functions are checkColumn, checkRow and checkSubgrid**

**- function will run to combine all checks together as numbers are being inputted**

**- function will then remove numbers based on user input of difficulty level.**

**- function will then call copy function to copy the unsolved grid into a file and display both full grid and unsolved grid to screen**

**7: INITIAL ALGORITHM FOR SOLVER**

**- user will be prompted to input file name where the unsolved sudoku puzzle is located**

**- program will then use function backtracking that will call on 3 functions from creator to check numbers as they are going into grid. Functions are checkColumn, checkRow and checkSubgrid. Function will run until grid is filled or it can’t be filled anymore.**

**- program will then call on copy function from creator to copy file into user chosen file and display it to the screen as well. If it can’t be solved, an error will be generated and program will be terminated.**

**8: CODE FOR CREATOR AND SOLVER**

**CREATOR:.**

//Source.cpp Source File for Sudoku Generator Class

#include "Header.h"

#include <fstream>

#include <string>

#include <ostream>

#include <vector>

#include <iomanip>

#include <sstream>

#include <iostream>

using namespace std;

//Description: main function is the program director that starts and runs the program

//Pre-condition: n/a

//Post-condition: program runs

int main() {

Sudoku sudoku;

ofstream out;

string outFile;

ofstream hintFile;

string hint;

cout << "Where Would You Like the Answer to be Stored?" << endl;

cin >> outFile;

out.open(outFile);

if (!out.is\_open())

{

cerr << "File does not exist, program will now terminate";

return 0;

}

cout << "Where Would You Like the Puzzle to be Stored?" << endl;

cin >> hint;

hintFile.open(hint);

if (hintFile.is\_open())

{

cerr << "File does not exist, program will now terminate";

return 0;

}

sudoku.generatePuzzle();

sudoku.drawHintLineSeparate(hint);

cout << endl;

sudoku.solve();

sudoku.drawLineSeparate(outFile);

out.close();

hintFile.close();

return 0;

}

//Definitions.cpp Souce File For Sudoku Class

#include "Header.h"

#include <fstream>

#include <string>

#include <ostream>

#include <vector>

#include <iomanip>

#include <sstream>

#include <iostream>

using namespace std;

typedef mt19937 RandGenerator;

RandGenerator random;

//Description: this will create an 2 random 2d sudoku board of size 9x9

//Pre-condition: none

//Post-condition: Initializes 2D 9x9 array created and ready for input

Sudoku::Sudoku() {

for (int i = 0; i < SIZE; ++i)

for (int j = 0; j < SIZE; ++j) {

sudokuBoard[i][j] = 0;

finalBoard[i][j] = 0;

}

change = 0;

random.seed(time(nullptr));

}

//Description: checks subgrid for validity of number given from sudokugenerator

//Pre-condition: number is given from possiblesubgrid

//Post-condition: return false if invalid or true if valid number

bool Sudoku::checkSubGrid(int number, int x, int y) {

int mid[2] = { (x \* 3 + 1), (y \* 3 + 1) };

for (int i = mid[0] - 1; i <= mid[0] + 1; ++i) {

for (int j = mid[1] - 1; j <= mid[1] + 1; ++j) {

if (finalBoard[i][j] == 0)

return false;

if (number == finalBoard[i][j])

return true;

}

}

return false;

}

//Description: checks number against numbers in column to determine validity

//Pre-condition: generatepuzzle is active and going through cycle and possible column is sending integers for checks

//Post-condition: return true if number is valid or false if not

bool Sudoku::checkColumn(int number, int x) {

for (int i = 0; i < SIZE; ++i) {

if (finalBoard[i][x] == 0)

break;

if (finalBoard[i][x] == number)

return true;

}

return false;

}

//Description: generate possible integers to go into checksubgrid where it will check for validity

//Pre-condition: generatepuzzle is active and going through cycle

//Post-condition:

vector<bool> Sudoku::possibleSubGrid(int x, int y) {

vector<bool> check(9, true);

int mid[2] = { x \* 3 + 1, y \* 3 + 1 };

for (int i = mid[0] - 1; i <= mid[0] + 1; ++i) {

for (int j = mid[1] - 1; j <= mid[1] + 1; ++j) {

if (this->sudokuBoard[i][j] != 0)

check[this->sudokuBoard[i][j] - 1] = false;

}

}

return check;

}

//Description: generate possible integers to go into checkcolumn where it will check for validity

//Pre-condition: generatepuzzle is active and going through cycle

//Post-condition: return false if invalid and true if it is possible integer for row

vector<bool> Sudoku::possibleColumn(int x) {

vector<bool> check(9, true);

for (int i = 0; i < SIZE; ++i) {

if (this->sudokuBoard[i][x] != 0)

check[this->sudokuBoard[i][x] - 1] = false;

}

return check;

}

//Description: generate possible integers to go into check where it will check for validity

//Pre-condition: generatepuzzle is active and going through cycle

//Post-condition: return false if invalid and true if it is possible integer for row

vector<bool> Sudoku::possibleRow(int i) {

vector<bool> check(9, true);

for (int j = 0; j < SIZE; ++j) {

if (this->sudokuBoard[i][j] != 0)

check[this->sudokuBoard[i][j] - 1] = false;

}

return check;

}

//Description: this is the core of the program. calls other functions and creates the puzzle

//Pre-condition: 2 vectors created from sudoku()

//Post-condition: final board is completed

void Sudoku::generatePuzzle() {

vector<bool> numbers(9, true);

for (int i = 0; i < SIZE; ++i) {

vector<bool> row = numbers;

int j = 0;

while (j < SIZE) {

if (this->possibilities[i][j].size() == 0) {

this->possibilities[i][j] = row;

}

if (possiblesCount(this->possibilities[i][j]) == 0) {

this->possibilities[i][j].resize(0);

--j;

this->possibilities[i][j][finalBoard[i][j] - 1] = false;

row[finalBoard[i][j] - 1] = true;

finalBoard[i][j] = 0;

}

else {

uniform\_int\_distribution<uint32\_t> tempGenerator(0, possiblesCount(this->possibilities[i][j]) - 1);

int number = tempGenerator(random) + 1;

int pickedNumber = 0;

while (true) {

if (this->possibilities[i][j][pickedNumber])

--number;

if (number == 0)

break;

++pickedNumber;

}

if (checkSubGrid(pickedNumber + 1, i / 3, j / 3) || checkColumn(pickedNumber + 1, j)) {

this->possibilities[i][j][pickedNumber] = false;

}

else {

finalBoard[i][j] = pickedNumber + 1;

row[pickedNumber] = false;

++j;

}

}

}

}

copy();

}

//Description: Determines if Single Square needs to be changed and possibilities for it

//Pre-condition: 2D array is initialized and has valid data

//Post-condition: Changes number if necessary

void Sudoku::singleSolved(int x, int y, int number) {

int mid[2] = { (x / 3) \* 3 + 1, (y / 3) \* 3 + 1 };

for (int i = mid[0] - 1; i <= mid[0] + 1; ++i) {

for (int j = mid[1] - 1; j <= mid[1] + 1; ++j) {

this->possibilities[i][j][number] = false;

}

}

for (int i = 0; i < SIZE; ++i) {

this->possibilities[i][y][number] = false;

this->possibilities[x][i][number] = false;

}

this->change++;

this->sudokuBoard[x][y] = number + 1;

}

//Description: Checks square if needs to be solved and calls singleSolved if necessary

//Pre-condition:Board is initialized with valid data

//Post-condition: Once square is determined to need solving, calls singleSolved to change number on

//board

int Sudoku::checkSingles() {

int solved = 0;

for (int i = 0; i < SIZE; ++i) {

for (int j = 0; j < SIZE; ++j) {

if (this->sudokuBoard[i][j] == 0 && possiblesCount(this->possibilities[i][j]) == 1) {

for (int x = 0; x < 9; ++x) {

if (this->possibilities[i][j][x]) {

solved++;

singleSolved(i, j, x);

break;

}

}

}

}

}

return solved;

}

//Description: function checks temp input to see if it can be validated and be put into board

//Pre-condition: integer was passed on to this function for check

//Post-condition: passes on whether the integer is valid or not

void Sudoku::checkUnknown(int x, int y) {

int mid[2] = { (x / 3) \* 3 + 1, (y / 3) \* 3 + 1 };

int temp[SIZE] = { 0 };

for (int i = mid[0] - 1; i <= mid[0] + 1; ++i)

for (int j = mid[1] - 1; j <= mid[1] + 1; ++j)

if ((i != x) && (j != y)) {

for (int k = 0; k < 9; ++k) {

if (this->possibilities[i][j][k]) temp[k]++;

}

}

for (int i = 0; i < SIZE; ++i) {

if (temp[i] == 1) {

singleSolved(x, y, i);

}

}

fill\_n(temp, SIZE, 0);

int column[SIZE] = { 0 };

for (int i = 0; i < SIZE; ++i) {

for (int j = 0; j < 9; ++j) {

if (this->possibilities[x][i][j]) temp[j]++;

if (this->possibilities[i][y][j]) column[j]++;

}

}

for (int i = 0; i < SIZE; ++i) {

if (temp[i] == 1) {

singleSolved(x, y, i);

}

else if (column[i] == 1) {

singleSolved(x, y, i);

}

}

}

//Description: function will attempt to solve the hint grid and will go through and check to solution after complete to make sure it solved correctly

//Pre-condition: sudoku hint file is ready

//Post-condition: sudoku is solved and will

bool Sudoku::solve() {

this->change = 0;

for (int i = 0; i < SIZE; ++i) {

for (int j = 0; j < SIZE; ++j) {

this->possibilities[i][j] = vector<bool>(9, false);

}

}

while (true) {

int undo = this->change;

vector<bool> column[SIZE];

vector<bool> row[SIZE];

for (int j = 0; j < SIZE; ++j) {

column[j] = possibleColumn(j);

}

for (int i = 0; i < SIZE; ++i) {

row[i] = possibleRow(i);

for (int j = 0; j < SIZE; ++j) {

vector<bool> subGrid = possibleSubGrid(i / 3, j / 3);

for (int k = 0; k < 9; ++k) {

this->possibilities[i][j][k] = (row[i][k] &&

column[j][k] &&

subGrid[k]);

}

}

}

if (checkSingles() == 0) {

for (int i = 0; i < SIZE; ++i) {

for (int j = 0; j < SIZE; ++j) {

if (this->sudokuBoard[i][j] != 0) continue;

checkUnknown(i, j);

}

}

}

if (undo == this->change)

break;

}

return solutionCheck();

}

//Description: copies boards to a temp location and tries to solve the hint board

//Pre-condition: board is created and passed on from generateboard

//Post-condition: solved or not solved

void Sudoku::copy() {

int x[DELETE];

int y[DELETE];

while (true) {

for (int i = 0; i < SIZE; ++i) {

for (int j = 0; j < SIZE; ++j) {

this->sudokuBoard[i][j] = this->finalBoard[i][j];

}

}

uniform\_int\_distribution<uint32\_t> temp(0, 8);

for (int i = 0; i < DELETE; ++i) {

x[i] = temp(random);

y[i] = temp(random);

while (this->sudokuBoard[x[i]][y[i]] == 0) {

x[i] = temp(random);

y[i] = temp(random);

}

this->sudokuBoard[x[i]][y[i]] = 0;

}

if (this->solve())

break;

}

for (int i = 0; i < DELETE; ++i) {

this->sudokuBoard[x[i]][y[i]] = 0;

}

}

//Description: this will check the solution file once completed and sent from solve() before sending it to file to display

//Pre-condition: solve is finished processing and solving sudokuhintboard file

//Post-condition: return true if solved correctly. if unsolved return false

bool Sudoku::solutionCheck() {

for (int i = 0; i < SIZE; ++i) {

for (int j = 0; j < SIZE; ++j) {

if (this->sudokuBoard[i][j] != this->finalBoard[i][j]) {

return false;

}

}

}

return true;

}

//Discription: function checks to see if count is valid during generating sudoku board to make sure its in the 9x9 grid

//Pre-condition: generateboard is active and cycling through to create board

//Post-condition: returns # per cycles it goes through to make sure # is within range of board

int Sudoku::possiblesCount(vector<bool> tryOut) {

int numberChecker = 0;

for (int i = 0; i < tryOut.size(); ++i) {

if (tryOut[i]) ++numberChecker;

}

return numberChecker;

}

//Description: this will draw lines to separate board to make it more visible

//Pre-condition: completed sudoku board

//Post-condition: completed board with separated lines

void Sudoku::drawLineSeparate(string outFile) {

ofstream out;

out.open(outFile);

drawSudoku(this->sudokuBoard, outFile);

}

//Description: this will draw lines to separate board to make it more visible

//Pre-condition: completed sudoku board

//Post-condition: completed board with separated lines

void Sudoku::drawHintLineSeparate(string hint) {

ofstream hintFile;

hintFile.open(hint);

drawSudokuHint(this->sudokuBoard, hint);

}

//Description: function is called by drawsudoku and it prints 0 for none hint integers

//Pre-condition: drawsudokuhint is activated and creating hint file

//Post-condition: 0 for empty spots and hint number where needed

char Sudoku::fill(const int x) {

if (x == 0) {

return '0';

}

else {

return x + '0';

}

}

//Description: creates squares and outputs them to outfile

//Pre-condition: sudokuboard is created and ready to output to file

//Post-condition: squares created and output to file and to screen

void Sudoku::drawSudoku(int sudokuSquares[][SIZE], string outFile) {

ofstream out;

out.open(outFile);

for (int i = 0; i < SIZE; ++i) {

for (int j = 0; j < SIZE; ++j) {

cout << fill(sudokuSquares[i][j]) << " ";

out << fill(sudokuSquares[i][j]) << " ";

}

cout << endl;

out << endl;

}

}

//Description: function will create sudoku hint file by removing a predetermined number of integers from sudoku grid

//Pre-condition: sudoku grid is created and complete

//Post-condition: predetermined number of integers removed from sudoku grid and a new grid created with hints only

void Sudoku::drawSudokuHint(int sudokuSquares[][SIZE], string hint) {

ofstream hintFile;

hintFile.open(hint);

for (int i = 0; i < SIZE; ++i) {

for (int j = 0; j < SIZE; ++j) {

cout << fill(sudokuSquares[i][j]) << " ";

hintFile << fill(sudokuSquares[i][j]) << " ";

}

cout << endl;

hintFile << endl;

}

}

**SOLVER:**

//Program Name:Sudoku Solver

//Programmer Name:Erika Valle-Baird

//Description:Puzzle Solver that takes in a file containing hints, uses recursion

//to solve. Prints the answer and recursive solving tries to separate files.

//Date Created:08/07/2020

//Header.h Header File For SudokuSolver Class

#pragma once

#include <fstream>

#include <string>

#include <ostream>

#include <vector>

#include <iomanip>

#include <random>

#include <ctime>

#include <sstream>

#include <iostream>

#include <string>

#include <ostream>

using namespace std;

//Variable Declarations Used Throughout Program

typedef vector<int> numbers;

typedef vector<numbers> Game;

//Counters Declared and Initialized

static int subGridCounter = 0;

static int rowCounter = 0;

static int columnCounter = 0;

static int recursiveCounter = 0;

class SudokuSolver {

//Variable Declarations

Game SudokuBoard;

numbers checker;

vector<string> FileInput;

int rowCheck(int row);

int columnCheck(int col);

int subGridCheck(int row, int col);

public:

int getSubGridCalls();

int getRowCalls();

int getColumnCalls();

int getRecurrCalls();

int RecursionToSolve(int row, int col,string solving);

SudokuSolver(string hint);

int solveSudoku(string solving);

void print();

void printAnswerToFile(string answer);

void printRecursion(string solving, bool append);

};

//Description: Keeps track of calls to subGridCheck

//Pre-condition: Initialized to zero

//Post-condition: Allows main to get amount of calls

int SudokuSolver::getSubGridCalls() {

return subGridCounter;

}

//Description: Keeps track of calls to rowCheck

//Pre-condition: Initialized to zero

//Post-condition: Allows main to get amount of calls

int SudokuSolver::getRowCalls() {

return rowCounter;

}

//Description: Keeps track of calls to colCheck

//Pre-condition: Initialized to zero

//Post-condition: Allows main to get amount of calls

int SudokuSolver::getColumnCalls() {

return columnCounter;

}

//Description: Keeps track of calls the recurance solver makes

//Pre-condition: Initialized to zero

//Post-condition: Allows main to get amount of calls

int SudokuSolver::getRecurrCalls() {

return recursiveCounter;

}

//Description: Class Constructor to recieve file input

//Pre-condition: Input File exists

//Post-condition: Data is loaded onto the table if data is valid,

//if not error messages displayed

SudokuSolver::SudokuSolver(string hint) {

//Variable Declarations

ifstream hintFile;

hintFile.open(hint);

checker.resize(10);

int value;

string line;

while (getline(hintFile, line)) {

FileInput.push\_back(line);

}

hintFile.close();

print();

SudokuBoard.resize(9);

for (int i = 0; i < FileInput.size(); ++i) {

string input = FileInput[i];

istringstream fileStream(input);

for (int j = 0; j < 9; ++j) {

if (!(fileStream >> value)) {

cout << "ERROR With File Read" << endl;

exit(1);

}

if (value == 0 || (value > 0 && value <= 9)) {

SudokuBoard[i].push\_back(value);

}

else {

cout << "ERROR Invalid File Element" << endl;

exit(1);

}

}

}

for (int i = 0; i < 9; i++) {

if (!rowCheck(i)) {

cout << "ERROR Invalid Row" << endl;

exit(1);

}

if (!columnCheck(i)) {

cout << "ERROR Invalid Column" << endl;

exit(1);

}

}

for (int i = 0; i < 9; i += 3) {

for (int j = 0; j < 9; j += 3) {

if (!subGridCheck(i, j)) {

cout << "ERROR Invalid SubGrid" << endl;

exit(1);

}

}

}

}

//Description: Checks if viable solution in row

//Pre-condition: Board Initialized with valid values

//Post-condition: Returns valid or not to be placed in row

int SudokuSolver::rowCheck(int row) {

//Variable Declaration and initialization

int countRow = 1;

for (int i = 0; i <= 9; i++)

checker[i] = 0;

for (int i = 0; i < 9; i++) {

if (SudokuBoard[row][i] != 0) {

if (checker[SudokuBoard[row][i]])

return 0;

checker[SudokuBoard[row][i]] = 1;

++countRow;

}

}

rowCounter = countRow;

return 1;

}

//Description: Checks if viable solution in column

//Pre-condition: Board Initialized with valid values

//Post-condition: Returns valid or not to be placed in sub column

int SudokuSolver::columnCheck(int col) {

//Variable Declaration and initialization

int columnCount = 1;

for (int i = 0; i <= 9; i++)

checker[i] = 0;

for (int i = 0; i < 9; i++) {

if (SudokuBoard[i][col] != 0) {

if (checker[SudokuBoard[i][col]])

return 0;

checker[SudokuBoard[i][col]] = 1;

++columnCount;

}

}

columnCounter = columnCount;

return 1;

}

//Description: Checks if viable solution in sub grid

//Pre-condition: Board Initialized with valid values

//Post-condition: Returns valid or not to be placed in sub grid

int SudokuSolver::subGridCheck(int x, int y) {

//Variable Declarations and initialization

int row, col;

int subGrid = 1;

for (int i = 0; i <= 9; i++)

checker[i] = 0;

for (row = x; row < x + 3; row++) {

for (col = y; col < y + 3; col++) {

if (SudokuBoard[row][col] != 0) {

if (checker[SudokuBoard[row][col]])

return 0;

checker[SudokuBoard[row][col]] = 1;

++subGrid;

}

}

}

subGridCounter = subGrid;

return 1;

}

//Description: Method that allows main to call and calls solving method

//Pre-condition: Called from main with output file name

//Post-condition: Calls method that uses recursion to solve

int SudokuSolver::solveSudoku(string solving) {

//Output File Declaration and open for writing

ofstream solveFile;

solveFile.open(solving, ios\_base::app);

return RecursionToSolve(0, 0, solving); }

//Description: Method to solve puzzle using recursion

//Pre-condition: Valid viable table, output file created

//Post-condition: Determines if puzzle can be solved. If it can

//Solves it and outputs recursive method to output file

int SudokuSolver::RecursionToSolve(int row, int col, string solving) {

//Variable Declarations and initialization

int recur = 1;

ofstream solveFile;

solveFile.open(solving, ios\_base::app);

while (row < 9 && SudokuBoard[row][col] != 0) {

col++;

if (col == 9) {

row++;

col = 0;

}

}

if (row == 9)

return 1;

for (int i = 1; i <= 9; i++) {

SudokuBoard[row][col] = i;

if (rowCheck(row) && columnCheck(col) && subGridCheck(row - row % 3, col - col % 3) && RecursionToSolve(row, col, solving)) {

return 1;

}

if (i % 3 == 0) {

++recur;

printRecursion(solving, true);

}

}

SudokuBoard[row][col] = 0;

recursiveCounter = recur;

return 0;

}

//Description: Print method to print input file and answer to screen

//Pre-condition: Input File exists and is valid, puzzle is sovable

//Post-condition: Input file and answer to puzzle is printed to screen for user

void SudokuSolver::print() {

for (int i = 0; i < 11; i++)

cout << "--";

cout << endl;

for (int i = 0; i < SudokuBoard.size(); i++) {

for (int j = 0; j < SudokuBoard[i].size(); j++) {

if (SudokuBoard[i][j] == 0)

cout << " ";

else

cout << SudokuBoard[i][j] << " ";

if (j == 2 || j == 5)

cout << "| ";

}

cout << endl;

if (i == 2 || i == 5) {

for (int k = 0; k < 11; k++)

cout << "--";

cout << endl;

}

}

cout << endl;

cout << endl;

}

//Description: Print method to print answer to puzzle to file

//Pre-condition: Output file exists

//Post-condition: Answer to puzzle written to output file

void SudokuSolver::printAnswerToFile(string answer) {

//File Declaration and Open

ofstream answerFile;

answerFile.open(answer);

for (int i = 0; i < 11; i++)

answerFile << "--";

answerFile << endl;

for (int i = 0; i < SudokuBoard.size(); i++) {

for (int j = 0; j < SudokuBoard[i].size(); j++) {

if (SudokuBoard[i][j] == 0)

answerFile << " ";

else

answerFile << SudokuBoard[i][j] << " ";

if (j == 2 || j == 5)

answerFile << "| ";

}

answerFile << endl;

if (i == 2 || i == 5) {

for (int k = 0; k < 11; k++)

answerFile << "--";

answerFile << endl;

}

}

answerFile << endl;

}

//Description: Print method to print Recursive Solving attempts while solving puzzle to file

//Pre-condition: Output file exists and is open for appending information

//Post-condition: Recursive Solving attempts is written to file

void SudokuSolver::printRecursion(string solving, bool append) {

//File Declaration and Open for writing

ofstream solveFile;

if (append) {

solveFile.open(solving, ios\_base::app);

}

else {

solveFile.open(solving);

}

for (int i = 0; i < 11; i++)

solveFile << "--";

solveFile << endl;

for (int i = 0; i < SudokuBoard.size(); i++) {

for (int j = 0; j < SudokuBoard[i].size(); j++) {

if (SudokuBoard[i][j] == 0)

solveFile << " ";

else

solveFile << SudokuBoard[i][j] << " ";

if (j == 2 || j == 5)

solveFile << "| ";

}

solveFile << endl;

if (i == 2 || i == 5) {

for (int k = 0; k < 11; k++)

solveFile << "--";

solveFile << endl;

}

}

solveFile << endl;

}

#include "Header.h"

using namespace std;

//Description: Function to display hint file to screen

//Pre-condition: Input file exists and is valid

//Post-condition: Calls Method to print

void hintFileToScreen(SudokuSolver display) {

cout << "Hint File: "<< endl;

display.print();

}

//Description: Function to display solved puzzle to screen

//Pre-condition: Output files exist and have been named via user

//Post-condition: Calls method to solve sudoku, prints to screen

//and output file

void solvedPuzzle(SudokuSolver display,string answer,string solving) {

ofstream solveFile;

solveFile.open(solving);

ofstream answerFile;

answerFile.open(answer);

if (!display.solveSudoku(solving)) {

cout << "ERROR Unable to Solve Puzzle! " << endl;

exit(0);

}

cout << endl;

cout << " SOLVED! " << endl;

cout << endl;

display.print();

cout << endl;

display.printAnswerToFile(answer);

}

//Description: Main function for program

//Pre-condition: None

//Post-condition: Puzzle will be solved or not, information

//written to necessary output files, statistics determined

int main() {

//Variable Declarations

string hint;

string answer;

string solving;

ifstream hintFile;

ofstream answerFile;

ofstream solveFile;

int counterCalls;

int totalCalls;

cout << "What is the Name of the File that Contains the Hints: " << endl;

cin >> hint;

cout << endl;

cout << "Where Would You Like to Save the Answer File to: " << endl;

cin >> answer;

cout << endl;

cout << "Where Would You Like to Save the Recursion Solving File to: " << endl;

cin >> solving;

cout << endl;

hintFile.open(hint);

if (!hintFile.is\_open()) {

ofstream errorWrite;

errorWrite.open("error.txt");

cout << "ERROR NO SUCH FILE";

errorWrite << "ERROR NO SUCH FILE";

return 0;

}

answerFile.open(answer);

SudokuSolver solve(hint);

cout << endl;

cout << "Sudoku Hint File " << endl;

cout << endl;

hintFileToScreen(solve);

solvedPuzzle(solve,answer,solving);

cout << endl;

hintFile.close();

answerFile.close();

counterCalls = subGridCounter + rowCounter + columnCounter;

cout << "The Sudoku Solver Made " << subGridCounter << " Calls to subGridCheck " << endl;

cout << "The Sudoku Solver Made " << rowCounter << " Calls to rowCheck " << endl;

cout << "The Sudoku Solver Made " << columnCounter << " Calls to colCheck" << endl;

cout << "The Sudoku Solver Made " << counterCalls << " Total Calls to Methods For rowCheck, colCheck, and subGridCheck " << endl;

cout << endl;

cout << "The Sudoku Solver Made " << recursiveCounter << " Recursive Calls, Resulting in ";

totalCalls = counterCalls \* recursiveCounter;

cout << totalCalls << " Total Calls to Solve this Puzzle " << endl;

return 0;

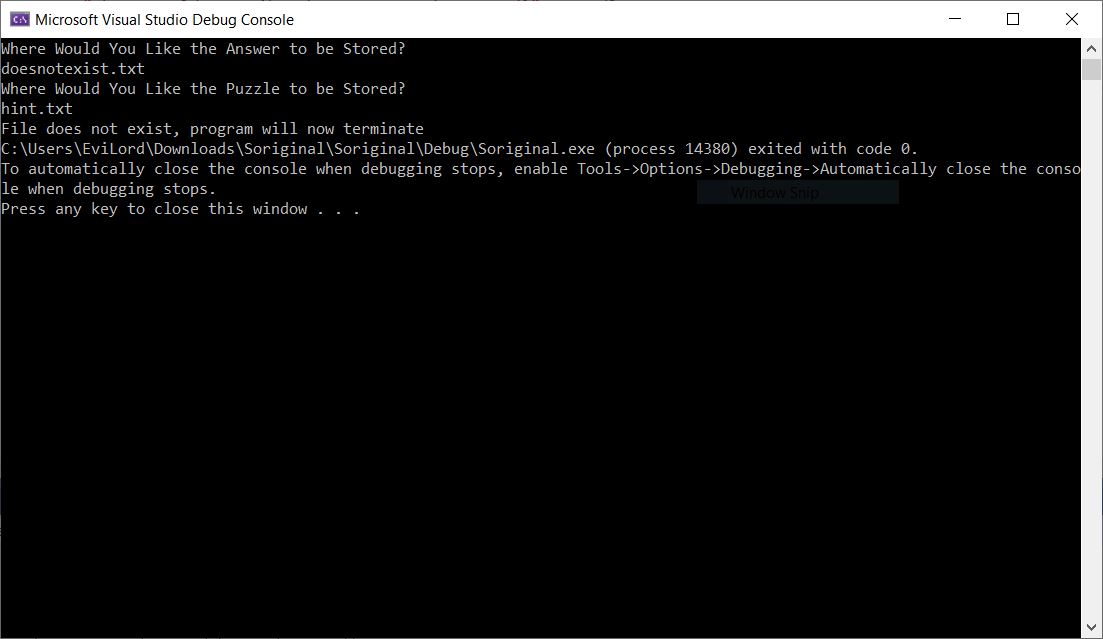
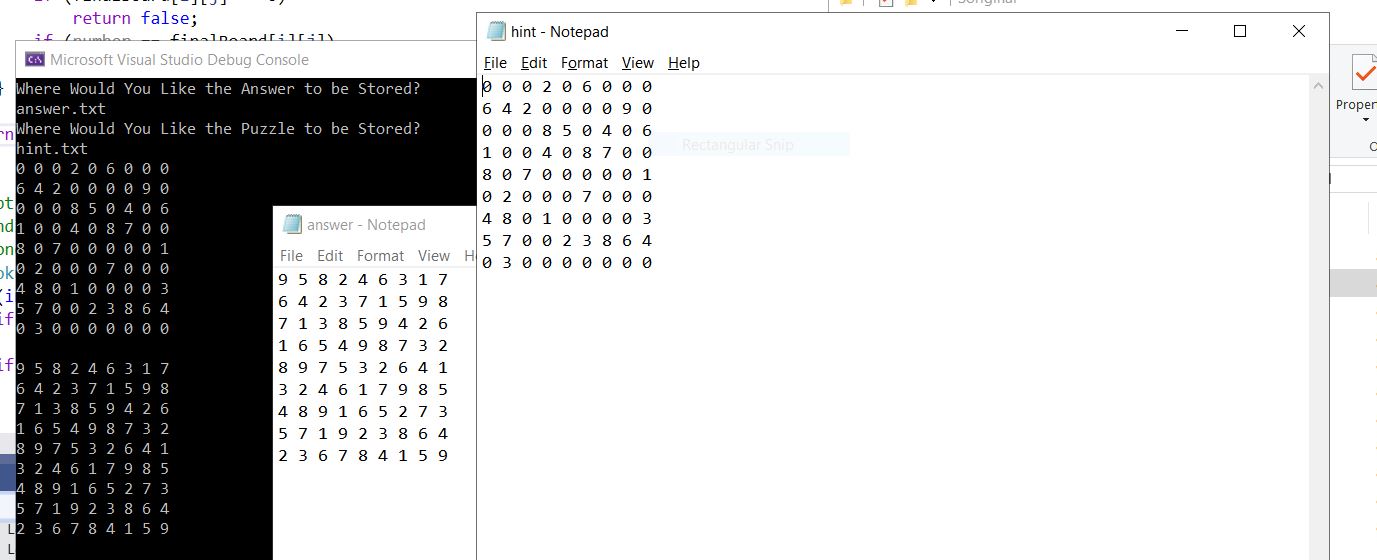
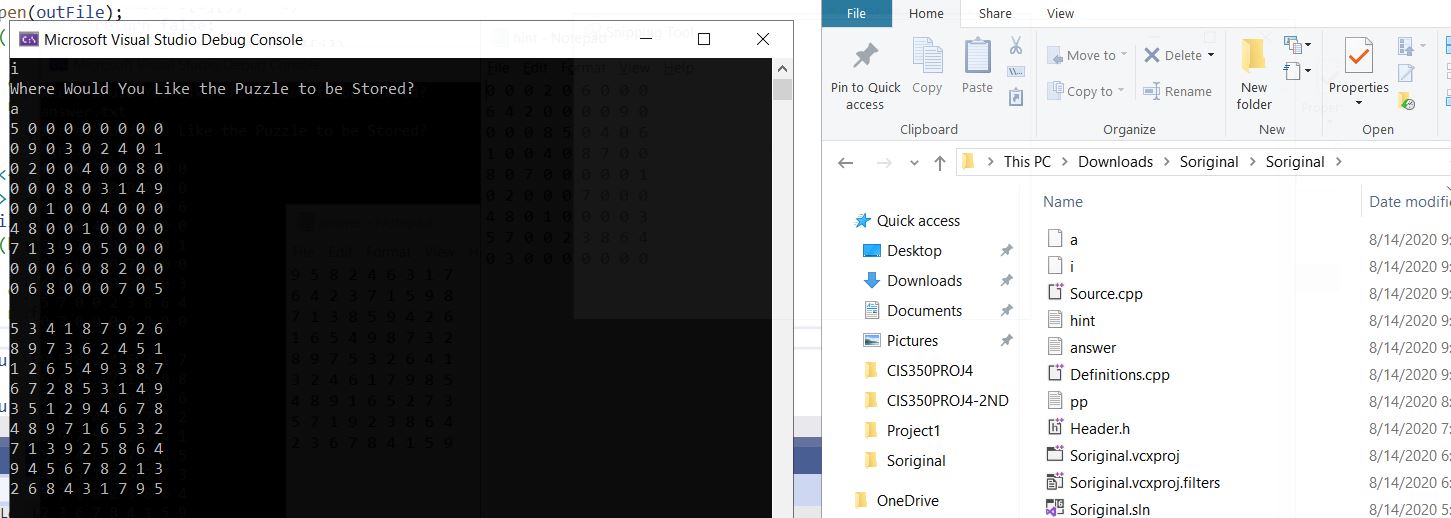
}

**9: TEST PLAN V2**

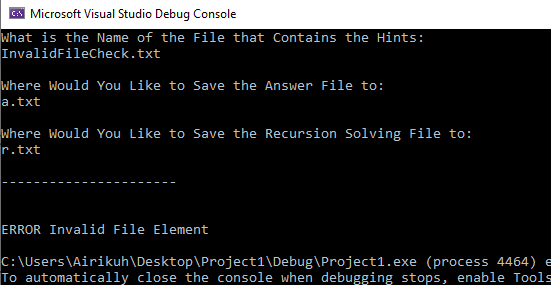
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Test Strategy** | **Test Number** | **Description** | **Input** | **Expected Output** | **Actual Output** | **Pass/Fail** |
| Invalid data | 1 | Use invalid file name for input file | Doesnotexist.txt | ~~“ERROR file does not exit”~~ | “input filename is not valid or does not exist, program will now terminate” | Pass |
| Invalid data | 2 | Use invalid file that does not exists before adding file check | Filedoesnotexist.txt | Error File does not exist, program will now terminate | Runs and creates files and sudoku board | Fail |
| Valid data | 3 | Use valid file that exists after adding file check | Filedoesnotexist.txt | “input filename is not valid or does not exist, program will now terminate” | “input filename is not valid or does not exist, program will now terminate” | Pass |

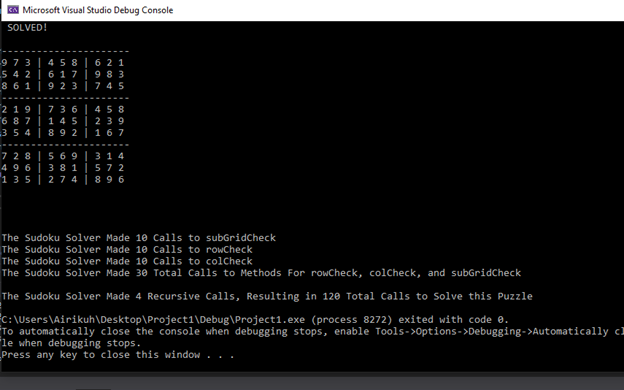
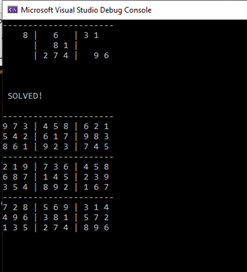
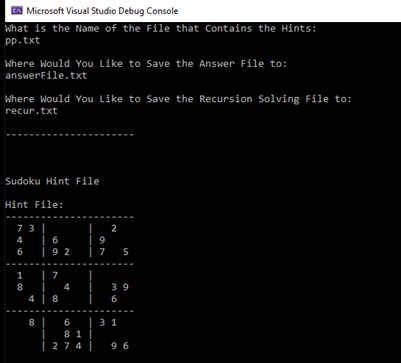
**Solver:**

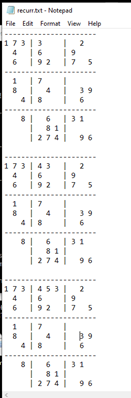
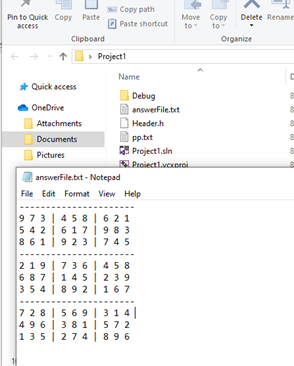
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Test Strategy** | **Test Number** | **Description** | **Input** | **Expected Output** | **Actual Output** | **Pass/Fail** |
| **Invalid data** | **1** | **Use file that does not exist** | **~~Doesnotexist.txt~~**  **InvalidFileCheck.txt** | **~~“input filename is not valid or does not exist, program will now terminate”~~**  **ERROR no such file** | **ERROR no such file** | **Pass** |
| **Invalid data** | **2** | **Invalid File Element** | **-2** | **ERROR Invalid File Element** | **ERROR Invalid File Element** | **Pass** |
| **Invalid data** | **3** | **Invalid Column** | **InvalidColumn.txt** | ~~Error message “ program will now terminate”~~  **ERROR Invalid Column** | **ERROR Invalid Column** | **PASS** |
| **Valid data** | **4** | **Invalid Row** | **InvalidRow.txt** | ~~Error message “ program will now terminate”~~  **ERROR Invalid Column** | **ERROR Row Column** | **PASS** |
| **Invalid Data** | **6** | **Invalid Sub Grid** | **InvalidSubGrid.txt** | ~~Error message “ program will now terminate”~~  **ERROR Invalid Sub Grid** | **ERROR Sub Grid Column** | **PASS** |
| **Valid data** | **7** | **Correct and valid data inputted into file** | **Valid data including input file name** | **Program will process unsolved data and will show results if grid is solvable** | **Processed and displayed proper output** | **PASS** |

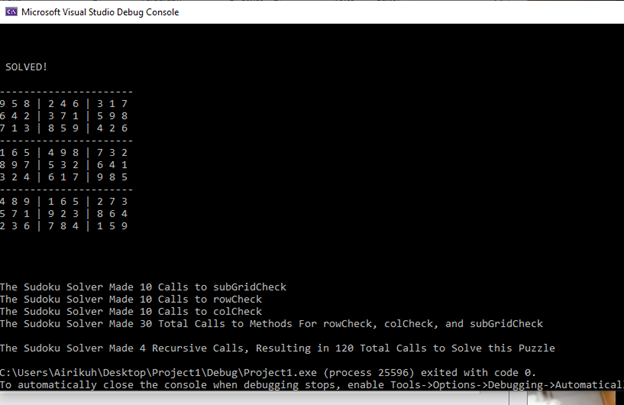
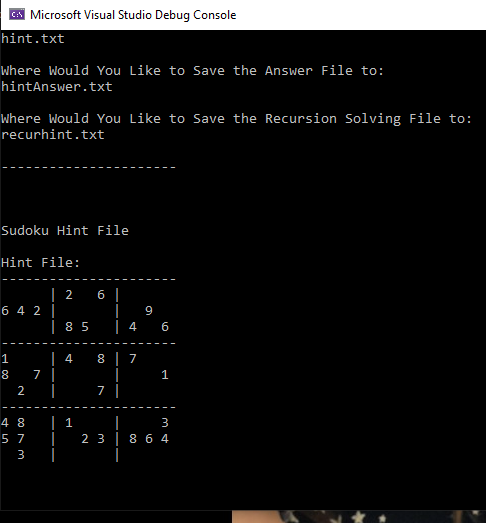
**CREATOR SCREENSHOTS**

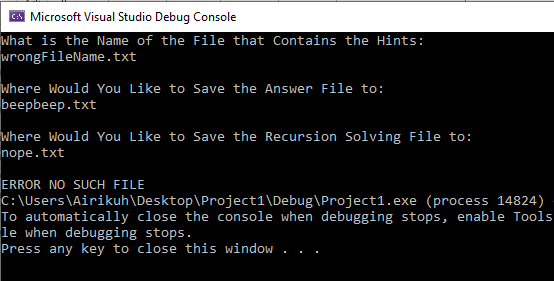
**SOLVER SCREENSHOTS**

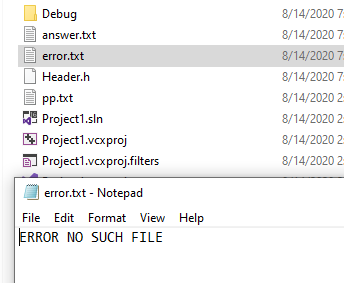
**6.**

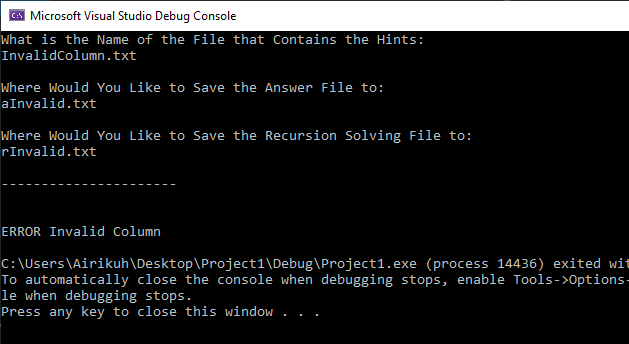
**7.**

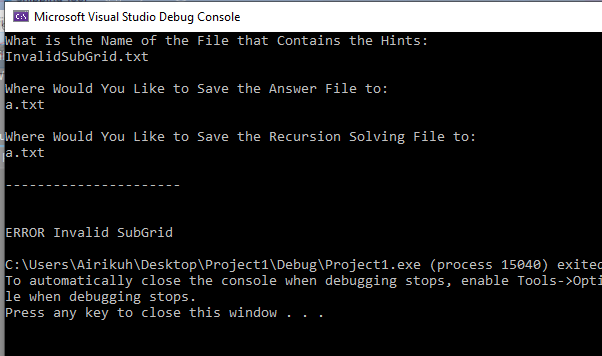




1.



3.

6.

4.