Sudoku Generator & Solver

Project Report

Lawaz Islam

C++ (modern, standard library) · Console application

Overview

This project is a console Sudoku application written in modern C++. It generates a random 9x9 Sudoku puzzle at a chosen difficulty, lets the user play it interactively, and can reveal the unique solution on request. It is a rewrite, in portable standard C++, of a Sudoku project I first worked on during an Object-Oriented Programming course, replacing the older Turbo C++ specific calls with standard library features so it compiles and runs on any current compiler.

Features

- Random puzzle generation with three difficulty levels (Easy, Medium, Hard), set by the number of revealed clues.
- Guaranteed unique solution: cells are only removed while the puzzle still has exactly one answer.
- Interactive play: place numbers by row, column, and value, with rule checking and protection of given cells.
- Reveal-solution and new-puzzle options at any time.

Object-Oriented Design

All state and logic live in a single Sudoku class. It holds the current puzzle grid, the full solution, and a record of which cells are fixed givens. Public methods expose a small, clear interface (generate, placeMove, isComplete, printPuzzle, printSolution), while the algorithms (validity checking, full-grid generation, solution counting, hole digging) are private helpers. This keeps the program state encapsulated and the main loop simple.

How It Works

Generation: an empty grid is filled completely using recursive backtracking. At each empty cell the digits 1 to 9 are tried in a randomized order, so each run produces a different valid grid. This completed grid is stored as the answer.

Creating the puzzle: cells are then removed one at a time in random order. After each removal the program counts the number of solutions (stopping early once it finds a second). If removing a cell would allow more than one solution, that cell is restored. Removal continues until only the target number of clues remains, which guarantees a single, fair puzzle.

Solving and validation: the same validity rule (no repeat in the row, column, or 3x3 box) drives both generation and the check on every user move. The solver itself is the same backtracking routine used to fill the grid.

How to Build and Run

```
g++ -std=c++17 -O2 sudoku.cpp -o sudoku
./sudoku
```

Sample Output

The screenshots below show the program in action: a freshly generated puzzle, a number placed during play, and the revealed solution. Replace each dashed box with the matching screenshot.

```
==== Sudoku Generator & Solver ====

Select difficulty: 1) Easy  2) Medium  3) Hard  (q to quit): 2

    1 2 3   4 5 6   7 8 9
  +-------+-------+-------+
A | . 8 2 | 1 . . | 9 3 6 |
B | . 7 9 | 8 . . | 5 1 4 |
C | 1 3 . | . 9 . | . . 2 |
  +-------+-------+-------+
D | . 1 3 | . . . | . . . |
E | 7 6 5 | 4 . . | . . 9 |
F | . 4 8 | . . 2 | . . . |
  +-------+-------+-------+
G | 8 . . | 5 . 4 | 1 . 7 |
H | . . . | . 7 . | . . . |
I | . . . | 2 8 1 | . 6 . |
  +-------+-------+-------+
```

Figure 1: A generated Medium-difficulty puzzle.

```
Options: (p) place a number, (s) show solution, (n) new puzzle, (q) quit: p
Enter row (A-I), column (1-9), value (1-9): A 1 5

    1 2 3   4 5 6   7 8 9
  +-------+-------+-------+
A | 5 8 2 | 1 . . | 9 3 6 |
B | . 7 9 | 8 . . | 5 1 4 |
C | 1 3 . | . 9 . | . . 2 |
  +-------+-------+-------+
D | . 1 3 | . . . | . . . |
E | 7 6 5 | 4 . . | . . 9 |
F | . 4 8 | . . 2 | . . . |
  +-------+-------+-------+
G | 8 . . | 5 . 4 | 1 . 7 |
H | . . . | . 7 . | . . . |
I | . . . | 2 8 1 | . 6 . |
  +-------+-------+-------+
```

Figure 2: Placing a number during interactive play.

```
Options: (p) place a number, (s) show solution, (n) new puzzle, (q) quit: s

    1 2 3   4 5 6   7 8 9
  +-------+-------+-------+
A | 5 8 2 | 1 4 7 | 9 3 6 |
B | 6 7 9 | 8 2 3 | 5 1 4 |
C | 1 3 4 | 6 9 5 | 8 7 2 |
  +-------+-------+-------+
D | 2 1 3 | 7 5 9 | 6 4 8 |
E | 7 6 5 | 4 1 8 | 3 2 9 |
F | 9 4 8 | 3 6 2 | 7 5 1 |
  +-------+-------+-------+
G | 8 2 6 | 5 3 4 | 1 9 7 |
H | 4 5 1 | 9 7 6 | 2 8 3 |
I | 3 9 7 | 2 8 1 | 4 6 5 |
  +-------+-------+-------+
```

Figure 3: The revealed solution after pressing 's'.

Possible Improvements

- Add a move counter and an optional timer per difficulty level.
- Track and display mistakes, or offer a hint that fills one correct cell.
- Add a simple scoring or quiz mode, and optionally a graphical interface.

Source Code (sudoku.cpp)

```
#include <iostream>
#include <vector>
#include <random>
#include <numeric>
#include <algorithm>

using Grid = std::vector<std::vector<int>>;   // 9x9, 0 means empty

class Sudoku {
public:
    Sudoku() : grid_(9, std::vector<int>(9, 0)),
               solution_(9, std::vector<int>(9, 0)),
               given_(9, std::vector<bool>(9, false)),
               rng_(std::random_device{}()) {}

    // Build a new puzzle. `clues` = how many numbers are revealed (lower = harder).
    void generate(int clues) {
        for (auto& row : grid_)     std::fill(row.begin(), row.end(), 0);
        fillFull(grid_);            // 1. make a complete, valid grid
        solution_ = grid_;          // 2. remember it as the answer
        digHoles(clues);            // 3. remove cells while keeping one solution
        for (int r = 0; r < 9; ++r)
            for (int c = 0; c < 9; ++c)
                given_[r][c] = grid_[r][c] != 0;   // lock the revealed cells
    }

    // Try to place value v at (r,c). Returns false if the cell is a given,
    // out of range, or the move breaks Sudoku rules.
    bool placeMove(int r, int c, int v) {
        if (r < 0 || r > 8 || c < 0 || c > 8 || v < 1 || v > 9) return false;
        if (given_[r][c]) return false;
        if (v != 0 && !isValid(grid_, r, c, v)) return false;
        grid_[r][c] = v;
        return true;
    }

    bool isComplete() const { return grid_ == solution_; }

    void printPuzzle()  const { print(grid_); }
    void printSolution() const { print(solution_); }

private:
    Grid grid_;
    Grid solution_;
    std::vector<std::vector<bool>> given_;
    std::mt19937 rng_;

    // Can value v legally go at (r,c) in grid g?
    bool isValid(const Grid& g, int r, int c, int v) const {
        for (int i = 0; i < 9; ++i)
            if (g[r][i] == v || g[i][c] == v) return false;   // row + column
        int br = (r / 3) * 3, bc = (c / 3) * 3;                 // top-left of 3x3 box
        for (int i = 0; i < 3; ++i)
            for (int j = 0; j < 3; ++j)
                if (g[br + i][bc + j] == v) return false;       // the box
        return true;
    }

    // Fill an empty grid with a complete, valid, randomized solution.
```

```
    bool fillFull(Grid& g) {
        for (int r = 0; r < 9; ++r) {
            for (int c = 0; c < 9; ++c) {
                if (g[r][c] != 0) continue;
                std::vector<int> nums(9);
                std::iota(nums.begin(), nums.end(), 1);     // 1..9
                std::shuffle(nums.begin(), nums.end(), rng_); // random order = random grid
                for (int v : nums) {
                    if (isValid(g, r, c, v)) {
                        g[r][c] = v;
                        if (fillFull(g)) return true;        // recurse forward
                        g[r][c] = 0;                          // backtrack
                    }
                }
                return false;   // no digit fits here, signal caller to backtrack
            }
        }
        return true;            // no empty cells left = solved
    }

    // Count solutions up to `limit` (we only care whether it is exactly 1).
    int countSolutions(Grid& g, int limit) {
        for (int r = 0; r < 9; ++r) {
            for (int c = 0; c < 9; ++c) {
                if (g[r][c] == 0) {
                    int total = 0;
                    for (int v = 1; v <= 9; ++v) {
                        if (isValid(g, r, c, v)) {
                            g[r][c] = v;
                            total += countSolutions(g, limit);
                            g[r][c] = 0;
                            if (total >= limit) return total; // early exit
                        }
                    }
                    return total;
                }
            }
        }
        return 1;   // fully filled = one solution found
    }

    // Remove cells one by one, keeping the puzzle uniquely solvable,
    // until only `clues` numbers remain.
    void digHoles(int clues) {
        std::vector<int> cells(81);
        std::iota(cells.begin(), cells.end(), 0);
        std::shuffle(cells.begin(), cells.end(), rng_);
        int filled = 81;
        for (int idx : cells) {
            if (filled <= clues) break;
            int r = idx / 9, c = idx % 9;
            int backup = grid_[r][c];
            if (backup == 0) continue;
            grid_[r][c] = 0;
            Grid test = grid_;
            if (countSolutions(test, 2) != 1) {  // removal made it ambiguous
                grid_[r][c] = backup;            // put it back
            } else {
                --filled;
            }
        }
    }

    void print(const Grid& g) const {
        std::cout << "\n    1 2 3   4 5 6   7 8 9\n";
        std::cout << "  +-------+-------+-------+\n";
        for (int r = 0; r < 9; ++r) {
            std::cout << (char)('A' + r) << " | ";
            for (int c = 0; c < 9; ++c) {
                if (g[r][c] == 0) std::cout << ". ";
                else              std::cout << g[r][c] << ' ';
                if (c % 3 == 2) std::cout << "| ";
```

```
            }
            std::cout << '\n';
            if (r % 3 == 2) std::cout << "  +-------+-------+-------+\n";
        }
        std::cout << '\n';
    }
};

int difficultyToClues(char choice) {
    switch (choice) {
        case '1': return 45;   // easy
        case '2': return 36;   // medium
        case '3': return 28;   // hard
        default:  return 36;
    }
}

int main() {
    Sudoku game;
    std::cout << "==== Sudoku Generator & Solver ====\n";

    while (true) {
        std::cout << "\nSelect difficulty: 1) Easy  2) Medium  3) Hard  (q to quit): ";
        char d;
        if (!(std::cin >> d) || d == 'q') break;

        game.generate(difficultyToClues(d));
        game.printPuzzle();

        // Inner loop: play this puzzle.
        while (true) {
            std::cout << "Options: (p) place a number, (s) show solution, "
                         "(n) new puzzle, (q) quit: ";
            char opt;
            if (!(std::cin >> opt)) return 0;

            if (opt == 'q') return 0;
            if (opt == 'n') break;
            if (opt == 's') { game.printSolution(); continue; }
            if (opt == 'p') {
                std::cout << "Enter row (A-I), column (1-9), value (1-9): ";
                char rowCh; int col, val;
                std::cin >> rowCh >> col >> val;
                int row = std::toupper(rowCh) - 'A';
                if (game.placeMove(row, col - 1, val)) {
                    game.printPuzzle();
                    if (game.isComplete()) {
                        std::cout << "Solved! Well done.\n";
                        break;
                    }
                } else {
                    std::cout << "Invalid move (fixed cell or breaks the rules).\n";
                }
            }
        }
    }
    std::cout << "Goodbye.\n";
    return 0;
}
```