Introduction

Sudoku is a logic-based, combinatorial number-placement puzzle. The goal of Sudoku is to fill a 9×9 grid with digits such that each column, each row, and each of the nine 3×3 subgrids contain all of the digits from 1 to 9. This assignment involves designing and implementing a complete Sudoku application in C++ that supports puzzle generation based on difficulty, interactive manual solving, and an automatic solver using backtracking.

This project highlights the use of object-oriented programming (OOP) concepts in C++, such as classes, encapsulation, and modular design, to build an interactive and intelligent console-based application.

Sudoku Rules

1. **Use Numbers 1 to 9**: Fill the grid using only digits from 1 to 9.
2. **Each Row Unique**: Every row must contain each number from 1 to 9 without repetition.
3. **Each Column Unique**: Every column must contain each number from 1 to 9 without repetition.
4. **Each 3×3 Box Unique**: Each 3×3 subgrid must contain each number from 1 to 9 without repetition.
5. **No Guessing Required**: A valid Sudoku puzzle has a logical solution without guessing.
6. **Only One Solution**: A proper Sudoku puzzle has exactly one unique solution.

1. Main Menu Loop

Algorithm:

1. Display the menu:
    - Enter Sudoku Puzzle
    - Generate Sudoku Puzzle
    - Solve Sudoku
    - Manual Entry
    - Exit
2. Take user input for menu choice.
3. Based on choice, invoke appropriate class method.

---

2. Enter Sudoku Puzzle (enterSudokuPuzzle)

Algorithm:

1. Loop through each cell (i,j) from (0,0) to (8,8).
2. Ask user to input a value for each cell.
3. If input is -1:
    - If at least 2 values entered → exit early.
    - Else → prompt to continue.

4. If input is invalid (not 0–9) → ask again.
5. If valid and safe → set the value.
6. Else → reject the input due to conflict (row/col/box).

---

3. Manual Entry (manualEntryToSolve)

Algorithm:

1. Repeatedly ask user to enter (row, col, num) or -1 to finish.
2. On each input:
    - Check if the cell is already filled.
    - Check if placing number is safe.
    - If safe, place it and print the grid.
3. If user enters -1:
    - If puzzle is complete → success.
    - Else → ask if the user wants to solve automatically.

---

4. Generate Sudoku Puzzle (generateSudoku)

Algorithm:

1. Ask user for difficulty (Easy/Medium/Hard).
2. Set number of **clues** accordingly.
3. Reset grid to empty.
4. Fill diagonal 3x3 boxes with random numbers.
5. Recursively fill the rest of the grid (fillGrid()).
6. Remove cells while ensuring the puzzle has only **one solution** (countSolutions()).

---

5. Fill Diagonal Boxes (fillDiagonal)

Algorithm:

1. For each diagonal 3x3 box (at (0,0), (3,3), (6,6)):
    - Shuffle numbers 1–9.
    - Place them randomly in the 3x3 block.

---

6. Recursive Sudoku Generator (fillGrid)

Algorithm:

1. Find an empty cell.
2. Shuffle numbers 1–9.
3. For each number:
    - If placing is safe, place it.
    - Recursively call fillGrid.

- If recursive call fails → backtrack.

---

7. Sudoku Solver (solveSudoku)

Algorithm:

1. Find an empty cell.
2. Try placing numbers 1–9:
   - If safe, place number.
   - Recursively try solving the rest.
   - If solving fails, backtrack and remove the number.
3. Count each backtrack for analysis.

---

8. Check Validity (isSafe)

Algorithm:

1. Check the row for the number.
2. Check the column for the number.
3. Check the 3x3 box for the number.
4. Return true if number is not found in any of the above.

---

9. Count Solutions (countSolutions)

Algorithm:

1. Recursive backtracking like solveSudoku.
2. Each time a full grid is reached, increment solution count.
3. Stop and return early if more than 1 solution is found.

---

10. Print Grid (printGrid)

Algorithm:

1. For each row:
   - Print | separators for 3x3 blocks.
   - Print number or _ for empty.
2. Print horizontal separators every 3 rows.

Outputs:

1. Generating and Solving Sudoku with Choice 2 and 3 respectively.

```
Menu:
1. Enter Sudoku Puzzle to Solve
2. Generate Sudoku Puzzle
3. Solve Sudoku
4. Manual Entry to Solve
5. Exit
Choice: 2
Select difficulty (1-Easy, 2-Medium, 3-Hard): 2

------------------------
| 4 3 2 | _ _ _ | 8 _ 5 |
| 1 _ _ | 5 8 _ | _ _ _ |
| _ _ 5 | 1 _ 2 | _ 6 7 |
------------------------
| _ _ 1 | 7 _ _ | 9 2 8 |
| 9 2 3 | _ 4 1 | _ _ 6 |
| _ 5 _ | 2 _ 6 | 1 4 3 |
------------------------
| 3 7 9 | 4 _ _ | 6 8 _ |
| 5 1 6 | _ _ _ | _ 9 4 |
| 2 _ 4 | _ 7 _ | 3 _ 1 |
------------------------
```

```
Menu:
1. Enter Sudoku Puzzle to Solve
2. Generate Sudoku Puzzle
3. Solve Sudoku
4. Manual Entry to Solve
5. Exit
Choice: 3

------------------------
| 4 3 2 | 9 6 7 | 8 1 5 |
| 1 6 7 | 5 8 4 | 2 3 9 |
| 8 9 5 | 1 3 2 | 4 6 7 |
------------------------
| 6 4 1 | 7 5 3 | 9 2 8 |
| 9 2 3 | 8 4 1 | 5 7 6 |
| 7 5 8 | 2 9 6 | 1 4 3 |
------------------------
| 3 7 9 | 4 1 5 | 6 8 2 |
| 5 1 6 | 3 2 8 | 7 9 4 |
| 2 8 4 | 6 7 9 | 3 5 1 |
------------------------
Solved using 41 backtracks.
```

2. Entering our own sudoku to get solved .

```
Menu:
1. Enter Sudoku Puzzle to Solve
2. Generate Sudoku Puzzle
3. Solve Sudoku
4. Manual Entry to Solve
5. Exit
Choice: 1
Enter the Sudoku puzzle (0 for empty cells, -1 to stop early after 2 values):
Cell (1,1): 2
Cell (1,2): 1
Cell (1,3): 1
Invalid! Conflict in row/col/box.
Cell (1,3): -1
Early exit.

------------------------
| 2 1 _ | _ _ _ | _ _ _ |
| _ _ _ | _ _ _ | _ _ _ |
| _ _ _ | _ _ _ | _ _ _ |
------------------------
| _ _ _ | _ _ _ | _ _ _ |
| _ _ _ | _ _ _ | _ _ _ |
| _ _ _ | _ _ _ | _ _ _ |
------------------------
| _ _ _ | _ _ _ | _ _ _ |
| _ _ _ | _ _ _ | _ _ _ |
| _ _ _ | _ _ _ | _ _ _ |
------------------------
```

3. Manual entry to solve sudoku.

```
Choice: 4

Enter row (1-9), column (1-9), number (1-9) or -1 to finish: 2 2 9
Invalid move! Conflict in row, column, or box.

Enter row (1-9), column (1-9), number (1-9) or -1 to finish: 2 2 6
Valid move. Current grid:

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

Enter row (1-9), column (1-9), number (1-9) or -1 to finish: -1
The puzzle is not complete. Solve it? (Y/N): y
```

```
Sudoku solved:


-------------------------
| 3 4 8 | 5 1 2 | 9 7 6 |
| 7 6 1 | 3 9 4 | 8 2 5 |
| 9 2 5 | 7 6 8 | 1 3 4 |
-------------------------
| 2 3 4 | 8 7 9 | 5 6 1 |
| 8 1 6 | 4 5 3 | 7 9 2 |
| 5 9 7 | 1 2 6 | 4 8 3 |
-------------------------
| 6 8 9 | 2 4 1 | 3 5 7 |
| 1 5 3 | 6 8 7 | 2 4 9 |
| 4 7 2 | 9 3 5 | 6 1 8 |
-------------------------
Solved using 1 backtracks.
```

Conclusion

This project demonstrates how C++ can be used to solve real-world problems involving logic, recursion, and object-oriented design. By implementing both the puzzle generator and solver, it provides a comprehensive understanding of Sudoku logic and computational problem-solving using backtracking. Features like manual solving and validation enrich user interactivity and improve understanding of constraints-based programming.