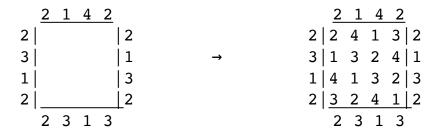
Skyscraper Puzzle Solver

This project implements a **solver for the Skyscraper puzzle** (also known as "Towers" or "Skyscrapers"). It is written in **C**, adheres to **Norminette rules**, and uses backtracking with pruning to efficiently find valid solutions.

About the Puzzle

The Skyscraper puzzle is a logic-based grid puzzle.

- The puzzle consists of a **4x4 grid**.
- Each cell must contain a number 1–4, representing the height of a building.
- Each number must appear **exactly once in every row and column** (like Sudoku).
- Clues are given along the edges: they specify how many buildings are **visible** from that side.
 - O Taller buildings hide smaller ones behind them.
 - Example: for row {2, 4, 1, 3}, looking from the left you see 2 buildings (2, then 4, and then 4 hides anything placed after it).



Works How It Works

The solver uses:

- Input validation (validate_input_condition, validate border conditions) to ensure correct and solvable puzzle input.
- **Permutation generation** (permute, get_arrays) to build candidate rows/columns that satisfy visibility rules.
- Validation functions (no_duplicates_in_rows, no_duplicates_in_columns, verification_left_right) to enforce puzzle rules.
- **Backtracking** (solve) to explore column-by-column, pruning invalid partial solutions early.

Usage

1. Compilation

cc -Wall -Wextra -Werror *.c -o skyscraper

2. Running

The program expects **one string argument** containing 16 numbers (1—4) separated by spaces. These numbers represent the border clues in this order:

```
TOP(4 \rightarrow) BOTTOM(4 \rightarrow) LEFT(4 \downarrow) RIGHT(4 \downarrow)
```

"col1top col2top col3top col4top col1bottom col2bottom col3bottom col4bottom row1left row2left row3left row4left row1right row2right row4right"



./skyscraper "2 1 4 2 2 3 1 3 2 3 1 2 2 1 3 2"

Example Output

2 4 1 3

1 3 2 4

4 1 3 2

3 2 4 1

If no solution exists or the input is not valid, the program prints:

Error

Project Structure

- skyscraper.c → entry point, solver execution, backtracking algorithm (main, solve, field is a solution)
- validations.c → input and rule validation functions
- **get_arrays.c** → row/column generation (permute, get arrays)
- helpers.c, field_helpers.c, get_arrays_helpers.c → utility functions and argument parsing (swap, reverse, display_field, grab_ints_from_str etc.)



- Clear separation of concerns (input validation, row/column generation, solver).
- Uses backtracking with pruning → efficient search.
- Prints either the **solution grid** or Error.
- Full compliance with **Norminette** (no long functions, 4 arguments maximum each, etc.).
- Only write(), malloc(), and free() were allowed to be used