Scott Elliott
Design

Drawing Fractals Using Recursion

Specification

Purpose and Problem Statement

The primary problem to be solved is the generation of a Koch curve of arbitrary detail (level) based on a user-defined initial line segment. The program must output a valid Postscript program to standard output (stdout), which can then be saved to a PDF and viewed.

Clarification Changes:

- **Koch Functionality**: Implement as a class rather than standalone functions
- **Input Handling**: Use std::stringstream for type-safe argument parsing with full validation
- **Memory Safety**: Implement destructor and disabled copying to prevent resource leaks
- **Exception Safety**: Wrap recursive calls in try-catch for proper resource cleanup
- **PostScript Compatibility**: Add bounding box and scaling for reliable viewing

Assumptions:

- Euclidean plane provides sufficient coordinate representation
- Recursion remains feasible up to level 8 given typical stack sizes
- Double-precision floating-point provides adequate mathematical precision
- Users understand command-line interfaces and have PostScript viewing capability
- Standard PostScript interpreters will correctly render the output format

Requirements and Input/Output

The table below summarizes the formal requirements for the program's interface and performance.

Category	**Detail**	**Explanation**
Input	Five command-line arguments: koch x1 y1 x2 y2 level.	The program must validate that all five arguments are present and correctly formatted (coordinates as doubles, level as a non-negative integer).
Output	A plain text Postscript program to stdout.	The output must begin with the %!PS-Adobe-2.0 header and end with stroke and showpage commands to ensure the file is viewable.
Performance	Time complexity is O(4^level).	This exponential growth means the number of operations quadruples with every increase in the level. A practical limit of level ≤ 8 is recommended to avoid excessive runtime and file size.

Algorithm Overview: Recursive Turtle Graphics

The program uses a recursive, turtle-graphics approach. The Koch class encapsulates the entire process, including the drawing mechanism.

1. **Initialization**: The main function parses the input and creates a Koch object. The Koch::generateCurve method calculates the initial length and angle of the level 0 segment.
2. **Drawing Mechanism**: A private nested class, TurtleHelper, is initialized with the starting point and angle. This helper class is responsible for translating the turtle's movements (turn, draw) into Postscript commands (rlineto).
3. **Recursion** (drawKoch): Core logic implemented in private recursive drawKoch(level, length).
    - **Base Case (Level 0):** If the recursion reaches level = 0, the TurtleHelper is instructed to draw a single line segment of the given length.
    - **Recursive Step (Level > 0):** The function calls itself four times with level-1 and a new length of length/3. Between these calls, the TurtleHelper is instructed to turn by specific angles (+60°, -120°, +60°) to create the characteristic Koch curve "bump".

Error Handling and Edge Cases: Robust error handling is critical for a command-line utility. The program must detect and report errors before attempting to generate the curve.

Special Case	**Handling Approach**	**Rationale**
Identical endpoints	Output minimal valid PS without drawing	Prevent division-by-zero and empty drawings
Negative level values	Reject during input validation	Maintain mathematical validity
Extreme coordinates	Rely on PS interpreter handling	Avoid coordinate system limitations
Maximum recursion	Practical level ≤ 8 limit	Prevent stack overflow
Floating-point precision	Accept minor rendering artifacts	Balance precision with performance

Input Validation

- **Incorrect argument count**: "Error: Incorrect number of arguments."
- **Non-numeric input**: "Error: Invalid numeric input for argument '[arg]'."
- **Negative level**: "Error: Curve level must be non-negative integer."

Runtime Protection

- **Zero-length segments**: Skip drawing operations
- **Recursion depth**: Rely on practical level limits
- **Memory allocation**: Exception handling for turtle object creation

Output Assurance

- **PostScript validity**: Guaranteed header and termination commands
- **Resource cleanup**: RAII pattern for turtle object management
- **Error streams**: Descriptive messages to stderr, clean output to stdout

Architectural Design

Overview: The program employs a highly cohesive architecture centered on a single Koch class that manages both fractal generation and drawing output. This class orchestrates the recursive construction while delegating mechanical drawing to a private TurtleHelper class. The recursive algorithm efficiently builds complexity through geometric substitution at each level, transforming simple coordinates into intricate fractal patterns. This separation of concerns allows the main driver to remain simple while ensuring robust fractal generation.

```
classDiagram
```

```
    class driver {
        + main()
    }
    class Koch {
        - TurtleHelper* turtle
        + generateCurve()
        - drawKoch()
    }
    class TurtleHelper {
        - currentX, currentY, currentAngle
        + turn(), drawLine(), outputStroke(), outputShowpage()
    }

    Koch "1" *-- "1" TurtleHelper : contains (private)
    driver --> Koch : creates and calls
```

Relationship Description:

- **Containment**: Koch exclusively owns TurtleHelper through composition
- **Encapsulation**: TurtleHelper is private nested class, hidden from driver
- **Dependency**: Driver depends only on Koch's public interface

Class Specifications

Koch Class: This class orchestrates the fractal generation process from user coordinates to final output. It transforms mathematical specifications into recursive construction plans while managing the complete lifecycle. It knows fractal geometry but delegates drawing mechanics to its helper.

+generateCurve(level, x1, y1, x2, y2) (public): Main orchestrator that calculates initial parameters, manages turtle lifecycle, and initiates recursion

-drawKoch(level, length) (private): Recursive engine implementing the four-part Koch rule

TurtleHelper Class: This class handles the mechanical work of drawing, maintaining pen state and translating abstract movements into precise PostScript. It knows nothing about fractals—only about maintaining position and orientation; and executing drawing commands.

+TurtleHelper(x, y, angle) (public): Constructor initializes state and outputs PS header

+turn(degrees) (public): Modifies current angle with normalization

+drawLine(length) (public): Calculates displacement, updates position, outputs rlineto

+outputStroke(), **outputShowpage()** (public): PS termination commands

Implementation and Test Plan (bottom-up) strategy. Implementation Order focuses on verifying the low-level drawing mechanism before testing the high-level recursion.

Phase	**Components**	**Testing Approach**	**Validation Criteria**
1. TurtleHelper Foundation	Constructor, turn(), drawLine(), output commands	Manual PostScript inspection of simple shapes	Basic lines and angles render correctly in PS viewer
2. Koch Recursive Core	drawKoch() with base case (level 0) and recursive step	*Level 0*: single segment *Level 1*: four segments with 60° turns	Geometric patterns match mathematical expectations
3. System Integration	generateCurve() with angle/length calculation, driver.cpp argument parsing	End-to-end testing with various parameters and error conditions	Full workflow produces valid fractals, handles errors gracefully

Unit Testing: Each class tested in isolation before integration
Integration Testing: Incremental build from TurtleHelper → drawKoch → generateCurve → main
Validation: Manual verification of fractal patterns and PostScript syntax

Test Data Sets: specific test cases to ensure both unit-level functionality and overall correctness

Case	**Command Line Input**	**Expected Behavior**
Minimal Case	./koch 10 10 100 10 0	Tests the Base Case of the recursion, resulting in a single straight line segment.
Typical Case	./koch 72 360 504 360 1	Tests the Recursive Step once, verifying the four-segment output and the 60° turns.
Edge Case (Vertical)	./koch 100 100 100 400 1	Verifies that the initial angle calculation (atan2) correctly handles a vertical line segment.
Invalid Input	./koch 10 10 100 10 -1	Verifies the Error Handling in driver.cpp, ensuring the program exits gracefully with an error message.

References

AI assistance was provided by Claude (Anthropic's Claude Sonnet 4.5) for grammatical review, structural organization, and documentation formatting. All technical content, design decisions, algorithm specifications, and implementation strategies are the original work of Scott Elliott.

Original prompt to Claude: "Review this fractal drawing design for grammatical errors and suggest improvements to document structure.