Challenges faced:-

1. Stack Implementation:

- Unlike higher-level languages that have built-in data structures like stacks, in C, creating a stack required manual management of array boundaries. Careful attention was needed to avoid stack overflow by ensuring the stack didn’t exceed its defined size, as well as handling stack underflow when popping from an empty stack.

- Managing the `top\_index` variable correctly was critical. Off-by-one errors (like incrementing or decrementing at the wrong time) could easily lead to incorrect results.

2. String Parsing and Number Conversion:

- Initially, the program could only handle single-digit numbers, as the input was parsed character by character. Modifying it to accept multi-digit numbers and floating-point numbers required using `atof` to convert strings into numbers and ensuring proper input validation.

- Ensuring that the input strings were correctly identified as either numbers or operators posed a challenge. Any invalid input or unsupported operators needed to be properly handled to avoid crashes or incorrect calculations.

4. Operator Precedence and Execution Order:

- RPN evaluates operations in the order they appear, without operator precedence. This required careful management of the stack, ensuring operands were pushed and popped in the correct order. Ensuring that operations were executed with the correct operands was a subtle but critical aspect of getting the final output right.

5. Edge Case Handling:

- Empty or Incomplete Expressions: Handling cases where insufficient operands were provided for the operators (e.g., `5 +`) was challenging. Proper error messages had to be displayed without crashing the program.

- Handling Extra Operands: Another issue was managing cases where there were too many operands and not enough operators (e.g., `5 6 7 +`), which needed clear reporting of an error.

6. Command-Line Arguments and Input Parsing:

- Since the input is passed as command-line arguments, the program needed to process each one individually. Ensuring that inputs like `5 6 +` worked as expected involved using `argc` and `argv[]` to loop through the input arguments and correctly interpret each one.0

- Variable Input Length: The program had to work with a variable number of inputs, which complicated the implementation compared to fixed-length inputs. This made the logic more dynamic and flexible but also added complexity to ensuring all input was processed correctly.

8. Testing and Debugging:

- Without modern debugging tools or error tracing from higher-level languages, debugging required printing intermediate values and checking where issues arose in the logic. This increased the time required for debugging and testing edge cases.

- Different Test Cases: I had to test various cases, from basic operations to more complex sequences, ensuring the stack handled all scenarios, including overflow, underflow, and operator precedence.

Test Runs and Observations

1. Input: 5 6 +

Output: Result: 11.00

2. Input: 5 1 2 + 4 \* + 3 -

Output: Result: 14.00

3. Input: 5 0 /

Output: `Error: Division by zero! Result: 0.00`

4. Input: 5 6 + -

Output: Error: Stack underflow!

5. Input: 5 +

Output: Error: Stack underflow!

The implementation of the RPN calculator in C required overcoming various challenges, including manual stack management, parsing input arguments, handling floating-point precision and robust error handling for division by zero and stack-related errors. Thorough testing confirmed that the program functions as expected, accurately evaluating expressions and gracefully handling errors.