**Challenges Faced**

**1. Implementing Trigonometric, Logarithmic, and Exponential Functions**

- Unlike basic arithmetic operations, trigonometric (e.g., `sin`, `cos`), logarithmic (`log`), and exponential (`exp`) functions required including the `<math.h>` library. Each function also had specific input requirements, such as angles in radians for trigonometric functions, which took lots of time to learn and implement it carefully.

- Some functions, such as `log`, required checking for invalid inputs, managing I stuck what to do with -ve value as it is not defined but with the help of internet I am able to rectify. For example, attempting to compute the logarithm of a negative number or zero needed appropriate error handling to prevent crashes or undefined behavior.

**2. Managing Different Function Inputs**

- Trigonometric functions use "radian inputs", so ensuring users passed angles in the correct unit was important. A decision had to be made on whether to accept angles in degrees and convert them internally or expect radians directly from the user.

- Logarithmic and exponential operations often result in non-integer values. Handling these floating-point results accurately while managing stack operations was essential for maintaining the calculator's correctness.

**3. Handling Function-Related Edge Cases**

- Invalid Inputs: Logarithmic operations required validation to ensure only positive numbers were processed. Similarly, handling `exp` with very large inputs needed consideration to avoid overflow issues, which could lead to incorrect results or program crashes.

- Stack Behavior with Unary Functions: Trigonometric, logarithmic, and exponential functions are unary (they take a single operand). Ensuring that these functions correctly popped only one operand from the stack—without disrupting the overall stack state—was critical for smooth execution.

**4. Testing and Debugging Specialized Operations**

- Debugging these operations required printing intermediate stack values, especially for trigonometric and logarithmic results, where even small input errors could lead to large output differences.

- Extensive testing was required to cover both valid and invalid inputs, including edge cases like:

- "log(0)" and log(-x) (should display error messages).

- sin(π/2), exp(0), and log(1) (to ensure correct outputs).

- Large input values for `exp` to detect potential overflow.

**Test Runs and Observations**

Test Case 1-

Input: 2 exp log sin

Output: Result: 0.91

Test Case 2-

Input: 3 4 + 2 \* 9 / 1 log

Output: Result: 0.00

Test Case 3-

Input: -5 log 3 +

Output: Logarithm of zero or negative number!

Stack underflow!

Test Case 4-

Input: 5 6 7 +

Output: Too many operands remaining!

Test Case 5-

Input: 3.1416 sin 2 \*

Output: Result: 0.00

**Conclusion**

Extending the RPN calculator with trigonometric, logarithmic, and exponential operations added complexity, especially with handling special input cases and managing floating-point precision. Implementing these operations required additional error handling for invalid inputs, such as logarithms of negative numbers and overflow cases for exponential operations. Through thorough testing, the enhanced calculator was able to correctly compute these functions while gracefully managing edge cases and errors.