Prefix Expression Evaluation

Problem Statement

Problem Statement:

Evaluate a given **prefix expression** using a **stack-based approach**.

Given Expression:

```
-+1/*+26483
```

This is a prefix expression where:

- Operators (`+, -, *, /`) appear **before** their operands.
- The goal is to **evaluate** this expression and obtain the correct result.

Logic Behind the Code

Logic Behind Prefix Evaluation:

1. **Traverse the string from right to left** (since it is a prefix expression).
2. If the character is a **digit**, push it onto the stack.
3. If the character is an **operator**:
   - Pop the **top two elements** from the stack.
   - Apply the operator on the two elements.
   - Push the result back onto the stack.
4. The final result will be at the top of the stack after processing the entire expression.

C++ Code

```
#include<iostream>
#include<stack>
#include<string>
using namespace std;

int solve(int v1, int v2, char ch) {
  if (ch == '+') return v1 + v2;
  else if (ch == '-') return v1 - v2;
```

```
  else if (ch == '*') return v1 * v2;
  else return v1 / v2;
}

int main() {
  string s = "-+1/*+26483";
  stack<int> st;

  for (int i = s.length() - 1; i >= 0; i--) {
    if (s[i] >= '0' && s[i] <= '9') {
      st.push(s[i] - '0');
    } else {
      int v1 = st.top();
      st.pop();
      int v2 = st.top();
      st.pop();
      char ch = s[i];
      int ans = solve(v1, v2, ch);
      st.push(ans);
    }
  }

  cout << st.top();
  return 0;
}
```

Dry Run Example

Dry Run Example:

Given Prefix Expression: **-+1/*+26483**

Step	Stack Content	Operation
Read '3'	[3]	Push 3

| Read '8' | [3, 8] | Push 8 |

| Read '4' | [3, 8, 4] | Push 4 |

| Read '6' | [3, 8, 4, 6] | Push 6 |

| Read '2' | [3, 8, 4, 6, 2] | Push 2 |

| Read '+' | [3, 8, 4, 8] | `2 + 6 = 8`, Push 8 |

| Read '*' | [3, 8, 32] | `8 * 4 = 32`, Push 32 |

| Read '/' | [3, 4] | `32 / 8 = 4`, Push 4 |

| Read '1' | [3, 4, 1] | Push 1 |

| Read '+' | [3, 5] | `1 + 4 = 5`, Push 5 |

| Read '-' | [2] | `5 - 3 = 2`, Push 2 |

Final Output: 2

Conclusion

Conclusion:

- **Prefix expressions** are evaluated by scanning **right to left**.

- A **stack** helps keep track of numbers and intermediate results.

- The **final result** is found at the **top of the stack** after evaluation.

This approach ensures **efficient** and **correct evaluation** of prefix expressions.

Short Notes

Short Notes:

- Prefix Expression: Operators appear **before** operands.

- Evaluation Order: **Right to Left**.

- Stack Usage: Helps store operands and intermediate results.

- Time Complexity: **O(n)** (where `n` is the length of the expression).

- Space Complexity: **O(n)** (in worst case when all are operands).