Postfix to Infix Conversion using Stack

Introduction

Postfix notation (Reverse Polish Notation) is a mathematical notation in which every operator follows all of its operands. Converting a postfix expression to infix requires maintaining the correct order of operations using a stack. This document explains the logic behind the given C++ code for postfix-to-infix conversion, along with a detailed dry run for better understanding.

Thought Process

1. **Use of Stack:**
   - A stack is used to store operands as we traverse the postfix expression.
   - If an operator is encountered, the top two operands are popped from the stack, and an infix expression is formed and pushed back into the stack.

2. **Iterate Through Postfix Expression:**
   - If a digit (operand) is encountered, push it onto the stack.
   - If an operator is encountered:
     - Pop the top two elements from the stack.
     - Form an infix expression with these two operands and the operator in between.
     - Push the new expression back onto the stack.

3. **Final Result:**
   - After processing the entire postfix expression, the stack will contain only one element, which is the fully converted infix expression.

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

// Function to evaluate the expression and return an infix string
string eval(string v1, string v2, char ch) {
    string s = "";
    s += v1;
    s.push_back(ch);
    s += v2;
    return s;
}

int main() {
    string s = "126+4*8/+3-"; // Postfix expression
    stack<string> val; // Stack to store operands and partial infix expressions

    for(int i = 0; i < s.length(); i++) {
        if(s[i] >= '0' && s[i] <= '9') { // If operand, push it to the stack
            val.push(to_string(s[i] - '0'));
        } else { // If operator, pop two operands from stack
            string v2 = val.top(); val.pop();
            char ch = s[i];
            string v1 = val.top(); val.pop();
            string ans = eval(v1, v2, ch); // Form infix expression
            val.push(ans); // Push back to stack
        }
    }

    // The final infix expression
    cout << "Infix Expression: " << val.top() << endl;
    return 0;
}
```

Dry Run

Example 1
Input: `126+4*8/+3-`

Step	Stack Content	Operation
1	1	Push 1
2	1, 2	Push 2
3	1, 2, 6	Push 6
4	1, 8	6 + 2 = 8
5	1, 8, 4	Push 4
6	1, 8, 16	4 * 4 = 16
7	1, 24	8 + 16 = 24
8	3	Push 3
9	21	24 - 3 = 21

Output: `((1+(2+6))+(4*8))/3-`

Conclusion

- The given approach efficiently converts a postfix expression to an infix expression using a stack.
- The `eval` function plays a crucial role in constructing the infix expression.
- The final result is obtained as a single element in the stack, representing the complete infix expression.

This document provides a comprehensive explanation of the logic, code, and dry runs, making it easy to understand the conversion process.