```
#include <iostream>
#include <string>
#include <math.h>
using namespace std;

// Define the maximum size of the stack
const int size = 100;

// Stack class for handling string operations
class Stack {
public:
    string stack[size]; // Array to hold stack elements
    int top; // Index for the top element

    // Constructor to initialize the stack
    Stack() {
        top = -1; // Stack is initially empty
    }

    // Check if the stack is full
    bool isFull() {
        return top == size - 1; // Full if top is at last index
    }

    // Check if the stack is empty
    bool isEmpty() {
        return top == -1; // Empty if top is -1
    }

    // Push a string onto the stack
```

```
    void push(string s) {
        if (!isFull()) { // Only push if not full
            top++;
            stack[top] = s; // Insert string at the top
        }
    }

    // Pop a string from the stack
    string pop() {
        if (!isEmpty()) { // Only pop if not empty
            return stack[top--]; // Return top element and decrement top
        }
        return ""; // Return empty string if stack is empty
    }

    // Peek at the top element of the stack without removing it
    string peek() {
        if (!isEmpty()) {
            return stack[top]; // Return the top element
        }
        return ""; // Return empty string if stack is empty
    }
};

// Manual implementation of stoi (string to int)
int stringToInt(const string& str) {
    int num = 0; // Initialize number to 0
    for (int i = 0; i < str.length(); i++) {
        num = num * 10 + (str[i] - '0'); // Convert character to integer
    }
```

```
    return num; // Return the integer value
}

// Manual implementation of to_string (int to string)
string intToString(int num) {
    string result = ""; // Initialize result string
    if (num == 0) return "0"; // Handle zero case
    while (num > 0) {
        result = char(num % 10 + '0') + result; // Convert integer to string
        num /= 10; // Reduce number
    }
    return result; // Return the string representation
}

// Check if a character is an operator
bool isOperator(char x) {
    return (x == '+' || x == '-' || x == '*' || x == '/' || x == '^'); // Return true if x is an operator
}

// Convert a prefix expression to infix
string prefixToInfix(string prefix) {
    Stack s; // Create a new stack
    for (int i = prefix.length() - 1; i >= 0; i--) { // Traverse the prefix expression from right to
left
        if (isOperator(prefix[i])) { // If the character is an operator
            string op1 = s.pop(); // Pop the first operand
            string op2 = s.pop(); // Pop the second operand
            string temp = "(" + op1 + prefix[i] + op2 + ")"; // Create a temporary infix expression
            s.push(temp); // Push the temporary expression back onto the stack
        }
        else {
```

```
            s.push(string(1, prefix[i])); // Push operand as string onto the stack
        }
    }
    return s.peek(); // Return the final infix expression
}

// Get precedence of operators
int precedence(char c) {
    if (c == '^') return 3; // Highest precedence for exponentiation
    if (c == '*' || c == '/') return 2; // Next highest for multiplication and division
    if (c == '+' || c == '-') return 1; // Lowest for addition and subtraction
    return -1; // Invalid character precedence
}

// Convert an infix expression to postfix
string infixToPostfix(string infix) {
    Stack s; // Create a new stack
    string postfix; // Initialize postfix expression
    for (int i = 0; i < infix.length(); i++) { // Traverse the infix expression
        char c = infix[i];
        if (isalnum(c)) { // If the character is alphanumeric, add it to postfix
            postfix += c;
        }
        else if (c == '(') {
            s.push(string(1, c)); // Push left parenthesis onto the stack
        }
        else if (c == ')') { // On right parenthesis, pop until left parenthesis is found
            while (!s.isEmpty() && s.peek() != "(") {
                postfix += s.pop(); // Pop and add to postfix
            }
```

```
            s.pop(); // Remove the left parenthesis from the stack
        }
        else if (isOperator(c)) { // If it's an operator
            while (!s.isEmpty() && precedence(s.peek()[0]) >= precedence(c)) {
                postfix += s.pop(); // Pop operators of higher or equal precedence
            }
            s.push(string(1, c)); // Push the current operator onto the stack
        }
    }
    while (!s.isEmpty()) { // Pop all remaining operators in the stack
        postfix += s.pop();
    }
    return postfix; // Return the final postfix expression
}

// Evaluate a postfix expression
int evaluatePostfix(string postfix) {
    Stack s; // Create a new stack for evaluation
    for (int i = 0; i < postfix.length(); i++) { // Traverse the postfix expression
        char c = postfix[i];
        if (isdigit(c)) { // If the character is a digit, push it onto the stack
            s.push(intToString(c - '0'));
        }
        else if (isOperator(c)) { // If it's an operator, pop two operands
            int op2 = stringToInt(s.pop()); // Pop the second operand
            int op1 = stringToInt(s.pop()); // Pop the first operand
            switch (c) { // Perform the operation based on the operator
                case '+': s.push(intToString(op1 + op2)); break;
                case '-': s.push(intToString(op1 - op2)); break;
                case '*': s.push(intToString(op1 * op2)); break;
```

```
            case '/': s.push(intToString(op1 / op2)); break;
            case '^': s.push(intToString(pow(op1, op2))); break;
          }
        }
    }
    return stringToInt(s.peek()); // Return the final evaluated result
}

// Main function to demonstrate conversion and evaluation
int main() {
    string prefix, infix, postfix;

    // Input and conversion from prefix to infix
    cout << "Enter prefix expression: ";
    cin >> prefix;
    cout << "Prefix to Infix: " << prefixToInfix(prefix) << endl;

    // Input and conversion from infix to postfix
    cout << "Enter infix expression: ";
    cin >> infix;
    postfix = infixToPostfix(infix);
    cout << "Infix to Postfix: " << postfix << endl;

    // Input and evaluation of postfix expression
    cout << "Enter postfix expression for evaluation: ";
    cin >> postfix;
    cout << "Postfix Evaluation: " << evaluatePostfix(postfix) << endl;

    return 0; // Exit the program
}
```