```
/*Implement stack as an abstract data type using singly linked list and use this ADT for evaluation of
postfix and prefix expression. */
#include <iostream>
#include <stack>
#include <cmath>
#include <cstring>

using namespace std;

// Node class for singly linked list
class Node {
public:
   double data;
   Node* next;

   Node(double value) {
      data = value;
      next = NULL;
   }
};

// Stack ADT using singly linked list
class Stack {
private:
   Node* top;

public:
   Stack() {
    top = NULL;
}

void push(double value) {
    Node* newNode = new Node(value);
    newNode->next = top;
    top = newNode;
}

double pop() {
    if (isEmpty()) {
        cerr << "Stack is empty." << endl;
        return -1; // You can choose a different error value if needed
    }

    double value = top->data;
    Node* temp = top;
    top = top->next;
    delete temp;
    return value;
}

double peek() {
    if (isEmpty()) {
        cerr << "Stack is empty." << endl;
        return -1; // You can choose a different error value if needed
    }
    return top->data;
    }

    bool isEmpty() {
        return top == NULL;
    }

    ~Stack() {
        while (!isEmpty()) {
            pop();
        }
    }
};

// Function to evaluate postfix expression
double evaluatePostfix(const char* expression) {
    Stack stack;

    for (size_t i = 0; expression[i]; i++) {
        if (isdigit(expression[i])) {
            stack.push(expression[i] - '0');
        } else {
            double operand2 = stack.pop();
            double operand1 = stack.pop();

            switch (expression[i]) {
                case '+':
                    stack.push(operand1 + operand2);
                    break;
                case '-':
                    stack.push(operand1 - operand2);
                    break;
                case '*':
                    stack.push(operand1 * operand2);
                    break;
                case '/':
                    stack.push(operand1 / operand2);
                    break;
                case '^':
                    stack.push(pow(operand1, operand2));
                    break;
                default:
                    cerr << "Invalid operator: " << expression[i] << endl;
                    return -1;
            }
        }
    }

    return stack.pop();
}

// Function to evaluate prefix expression recursively
double evaluatePrefix(const char* expression, int& index) {
    if (expression[index] == '\0') {
        cerr << "Expression ended unexpectedly." << endl;
        return -1;
    }
    char token = expression[index++];
    if (isdigit(token)) {
        return token - '0';
    } else {
        double operand2 = evaluatePrefix(expression, index);
        double operand1 = evaluatePrefix(expression, index);

        switch (token) {
            case '+':
                return operand1 + operand2;
            case '-':
                return operand1 - operand2;
            case '*':
                return operand1 * operand2;
            case '/':
                return operand1 / operand2;
            case '^':
                return pow(operand1, operand2);
            default:
                cerr << "Invalid operator: " << token << endl;
                return -1;
        }
    }
}

int main() {
    cout << "Enter a postfix expression: ";
    char postfixExpression[100];
    cin.getline(postfixExpression, 100);

    double resultPostfix = evaluatePostfix(postfixExpression);
    cout << "Result of postfix evaluation: " << resultPostfix << endl;

    cout << "Enter a prefix expression: ";
    char prefixExpression[100];
    cin.getline(prefixExpression, 100);
    int index = 0;
    double resultPrefix = evaluatePrefix(prefixExpression, index);
    cout << "Result of prefix evaluation: " << resultPrefix << endl;

    return 0;
}
```