```
/*Implement stack as an abstract data type using singly linked list and use this ADT for conversion of
infix expression to postfix*/

#include <iostream>

#include <stack>

#include <string>


using namespace std;


// Node class for the singly linked list

class Node {

public:

    char data;

    Node* next;


    Node(char value) : data(value), next(NULL) {}

};


// Stack implemented using a singly linked list

class Stack {

private:

    Node* top;


public:

    Stack() : top(NULL) {}


    bool isEmpty() {

        return top == NULL;

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

    char pop() {
        if (isEmpty()) {
            cerr << "Stack is empty." << endl;
            return '\0';
        }

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

    char peek() {
        if (isEmpty()) {
            cerr << "Stack is empty." << endl;
            return '\0';
        }
        return top->data;
    }
};
```

```
// Function to check if a character is an operand
bool isOperand(char c) {
    return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z');
}

// Function to get the precedence of operators
int getPrecedence(char op) {
    switch (op) {
        case '^':
            return 3;
        case '*':
        case '/':
            return 2;
        case '+':
        case '-':
            return 1;
        default:
            return -1;
    }
}

// Function to convert infix expression to postfix expression
string infixToPostfix(const string& infix) {
    string postfix;
    Stack stack;

    for (char c : infix) {
        if (isOperand(c)) {
            postfix += c;
        } else if (c == '(') {
            stack.push(c);
        } else if (c == ')') {
            while (!stack.isEmpty() && stack.peek() != '(') {
                postfix += stack.pop();
            }
            stack.pop(); // Pop '(' from the stack
        } else { // Operator
            while (!stack.isEmpty() && getPrecedence(c) <= getPrecedence(stack.peek())) {
                postfix += stack.pop();
            }
            stack.push(c);
        }
    }

    while (!stack.isEmpty()) {
        postfix += stack.pop();
    }

    return postfix;
}

int main() {
    string infixExpression;
    cout << "Enter an infix expression: ";
    getline(cin, infixExpression);
    string postfixExpression = infixToPostfix(infixExpression);
    cout << "Postfix expression: " << postfixExpression << endl;


    return 0;
}
```