//Assignment 2

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
// ================== Stack ADT using Singly Linked List ==================
template <typename T>
class Stack {
private:
   struct Node {
      T data;
      Node* next;
      Node(T val) : data(val), next(nullptr) {}
   };
   Node* topNode;
public:
   Stack() : topNode(nullptr) {}

   ~Stack() {
      while (!isEmpty()) pop();
   }
   void push(T value) {
      Node* newNode = new Node(value);
      newNode->next = topNode;
      topNode = newNode;
   }
   void pop() {
      if (isEmpty()) return;
      Node* temp = topNode;
      topNode = topNode->next;
      delete temp;
   }
   T top() {
      if (isEmpty()) throw runtime_error("Stack is empty");
      return topNode->data;
   }
   bool isEmpty() {
      return topNode == nullptr;
   }
```

```
};
// ================== Helper Functions ==================
bool isOperator(char c) {
   return (c == '+' || c == '-' || c == '*' || c == '/' || c == '^');
}
int precedence(char c) {
   if (c == '^') return 3;
   if (c == '*' || c == '/') return 2;
   if (c == '+' || c == '-') return 1;
   return -1;
}
// ================== Infix to Postfix ==================
string infixToPostfix(string infix) {
   Stack<char> st;
   string result;
   for (char c : infix) {
      if (isalnum(c)) {
         result += c;  // Operand directly to result
      }
      else if (c == '(') {
         st.push(c);
      }
      else if (c == ')') {
         while (!st.isEmpty() && st.top() != '(') {
            result += st.top();
            st.pop();
         }
         if (!st.isEmpty()) st.pop();  // remove '('
      }
      else if (isOperator(c)) {
         while (!st.isEmpty() && precedence(st.top()) >= precedence(c)) {
            if (c == '^' && st.top() == '^') break; // right-associative
            result += st.top();
            st.pop();
         }
         st.push(c);
      }
   }
```

```
    while (!st.isEmpty()) {
        result += st.top();
        st.pop();
    }
    return result;
}
// ================== Infix to Prefix ==================
string infixToPrefix(string infix) {
    // Reverse infix expression
    reverse(infix.begin(), infix.end());
    // Replace ( with ) and vice versa
    for (char &c : infix) {
        if (c == '(') c = ')';
        else if (c == ')') c = '(';
    }
    // Convert to postfix of the modified expression
    string postfix = infixToPostfix(infix);
    // Reverse postfix to get prefix
    reverse(postfix.begin(), postfix.end());
    return postfix;
}
// ================== Main ==================
int main() {
    string infix;
    cout << "Enter an infix expression: ";
    cin >> infix;
    try {
        string postfix = infixToPostfix(infix);
        string prefix = infixToPrefix(infix);
        cout << "Infix   : " << infix << endl;
        cout << "Postfix : " << postfix << endl;
        cout << "Prefix  : " << prefix << endl;
    }
    catch (exception &e) {
        cout << "Error: " << e.what() << endl;
    }

    return 0;
}
```

```
//OUTPUT
Enter an infix expression: a+b*c
Infix   : a+b*c
Postfix : abc*+
Prefix  : +a*bc
```