--INFIX TO POSTFIX---

Code =>

```
#include <iostream>
#include <algorithm>
using namespace std;
class Node
{
public:
    int data;
    Node *next;
};
class Stack
{
public:
    Node *top = NULL;
    bool empty();
    void push(int);
    void pop();
};
bool Stack ::empty()
{
    if (top == NULL)
    {
        return true;
    }
    else
    {
        return false;
    }
}
void Stack ::push(int new_data)
{
    Node *new_node = new Node();
    new_node->data = new_data;
    new_node->next = top;
    top = new_node;
}
void Stack ::pop()
{
    if (empty())
    {
```

```cpp
        cout << "Stack is Empty" << endl;
    }
    else
    {
        Node *temp = top;
        top = top->next;
        delete (temp);
    }
}

int precedence(char c)
{
    if (c == '^')
    {
        return 3;
    }
    else if (c == '/' || c == '*')
    {
        return 2;
    }
    else if (c == '+' || c == '-')
    {
        return 1;
    }
    else
    {
        return -1;
    }
}

// The main function to convert infix expression to postfix expression
void infixToPostfix(string infix)
{

    Stack st;
    string postfix;

    for (int i = 0; i < infix.length(); i++)
    {
        char c = infix[i];

        // If the scanned character is an operand, add it to output
string.
```

```
        if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >=
'0' && c <= '9'))
        {
            postfix += c;
        }
        // If the scanned character is an ‘(‘, push it to the stack.
        else if (c == '(')
        {
            st.push('(');
        }
        // If the scanned character is an ‘)’, pop and to output string
from the stack until an ‘(‘ is encountered.
        else if (c == ')')
        {
            while (st.top->data != '(')
            {
                postfix += st.top->data;
                st.pop();
            }
            st.pop();
        }
        //If an operator is scanned
        else
        {
            while (!st.empty() && precedence(infix[i]) <=
precedence(st.top->data))
            {
                postfix += st.top->data;
                st.pop();
            }
            st.push(c);
        }
    }

    // Pop all the remaining elements from the stack
    while (!st.empty())
    {
        postfix += st.top->data;
        st.pop();
    }

    cout << postfix << endl;
}
```

```
int main()
{
    string str;
    cout<<"Enter the infix expression  :  ";
    cin>>str;
    cout<<"Postfix Expression  :  ";
    infixToPostfix(str);
    return 0;
}
```

```
Output 1 :
Enter the infix expression  :  a+b
Postfix Expression  :  ab+

Output 2 :
Enter the infix expression  :  (A+B)*(C+D)
Postfix Expression  :  AB+CD+*

Output 3:
Enter the infix expression  :  A+B*C+D
Postfix Expression  :  ABC*+D+

Output 4 :
Enter the infix expression  :  a+b-c+d
Postfix Expression  :  ab+c-d+

Output 5 :
Enter the infix expression  :  a+b*(c^d-e)^(f+g*h)-i
Postfix Expression  :  abcd^e-fgh*+^*+i-
```

---END--

--INFIX TO PREFIX---

Code =>

```
#include <iostream>
#include <algorithm>
using namespace std;
class Node
{
public:
    int data;
    Node *next;
};
class Stack
{
public:
    Node *top = NULL;
    bool empty();
    void push(int);
    void pop();
};
bool Stack ::empty()
{
    if (top == NULL)
    {
        return true;
    }
    else
    {
        return false;
    }
}
void Stack ::push(int new_data)
{
    Node *new_node = new Node();
    new_node->data = new_data;
    new_node->next = top;
    top = new_node;
}
void Stack ::pop()
{
    if (empty())
    {
```

```cpp
        cout << "Stack is Empty" << endl;
    }
    else
    {
        Node *temp = top;
        top = top->next;
        delete (temp);
    }
}

int prec(char c)
{
    if (c == '^')
    {
        return 3;
    }
    else if (c == '/' || c == '*')
    {
        return 2;
    }
    else if (c == '+' || c == '-')
    {
        return 1;
    }
    else
    {
        return -1;
    }
}

bool isOperator(char c)
{
    return (!isalpha(c) && !isdigit(c));
}

string infixToPostfix(string infix)
{
    infix = '(' + infix + ')';
    int l = infix.size();
    Stack st;
    string output;

    for (int i = 0; i < l; i++)
    {

        // If the scanned character is an
        // operand, add it to output.
        if (isalpha(infix[i]) || isdigit(infix[i]))
            output += infix[i];

        // If the scanned character is an
        // '(', push it to the stack.
        else if (infix[i] == '(')
            st.push('(');

        // If the scanned character is an
        // ')', pop and output from the stack
        // until an '(' is encountered.
        else if (infix[i] == ')')
        {
            while (st.top->data != '(')
            {
                output += st.top->data;
                st.pop();
            }

            // Remove '(' from the stack
            st.pop();
        }

        // Operator found
        else
        {
            if (isOperator(st.top->data))
            {
                if (infix[i] == '^')
                {
                    while (prec(infix[i]) <= prec(st.top->data))
                    {
                        output += st.top->data;
                        st.pop();
                    }
                }
                else
                {
                    while (prec(infix[i]) < prec(st.top->data))
                    {
                        output += st.top->data;
                        st.pop();
                    }
                }

                // Push current Operator on stack
                st.push(infix[i]);
            }
        }
    }
    while (!st.empty())
    {
        output += st.top->data;
        st.pop();
    }
    return output;
}

string infixToPrefix(string infix)
{
    /* 1. Reverse String
       2. Replace ( with ) and vice versa
       3. Get Postfix
       4. Reverse Postfix
    */
    int l = infix.size();

    // Reverse infix
    reverse(infix.begin(), infix.end());

    // Replace ( with ) and vice versa
    for (int i = 0; i < l; i++)
    {

        if (infix[i] == '(')
        {
            infix[i] = ')';
        }
        else if (infix[i] == ')')
        {
            infix[i] = '(';
        }
    }

    string prefix = infixToPostfix(infix);

    // Reverse postfix
    reverse(prefix.begin(), prefix.end());

    return prefix;
}

int main()
{
    string str;
    cout << "Enter the infix expression  :  ";
    cin >> str;
    cout << "Prefix Expression  :  " << infixToPrefix(str);
    return 0;
}
```

```
Output 1 :
Enter the infix expression  :  A+B*C+D
Prefix Expression  :  ++A*BCD

Output 2 :
Enter the infix expression  :  (A+B)*(C+D)
Prefix Expression  :  *+AB+CD

Output 3 :
Enter the infix expression  :  (A-B/C)*(A/K-L)
Prefix Expression  :  *-A/BC-/AKL

Output 4 :
Enter the infix expression  :  a+b-c
Prefix Expression  :  -+abc

Output 5 :
Enter the infix expression  :  x+y*z/w+u
Prefix Expression  :  ++x/*yzwu
```

--END--

--PREFIX EVALUATION--

Code =>

```
#include <iostream>
#include <algorithm>
using namespace std;
class Node
{
public:
    int data;
    Node *next;
};
class Stack
{
public:
    Node *top = NULL;
    bool empty();
    void push(int);
    void pop();
};
bool Stack ::empty()
{
    if (top == NULL)
    {
        return true;
    }
    else
    {
        return false;
    }
}
void Stack ::push(int new_data)
{
    Node *new_node = new Node();
    new_node->data = new_data;
    new_node->next = top;
    top = new_node;
}
void Stack ::pop()
{
    if (empty())
    {
        cout << "Stack is Empty" << endl;
    }
    else
    {
        Node *temp = top;
        top = top->next;
        delete (temp);
    }
}
bool isOperand(char c)
{
    // If the character is a digit then it must
    // be an operand
    return isdigit(c);
}

double evaluatePrefix(string s)
{
    Stack st;

    for (int j = s.size() - 1; j >= 0; j--)
    {

        // Push operand to Stack
        // To convert s[j] to digit subtract
        // '0' from s[j].
        if (isOperand(s[j]))
        {
            st.push(s[j] - '0');
        }
        else
        {

            // Operator encountered
            // Pop two elements from Stack
            double o1 = st.top->data;
            st.pop();
            double o2 = st.top->data;
            st.pop();

            // Use switch case to operate on o1
            // and o2 and perform o1 O o2.
            switch (s[j])
            {
            case '+':
                st.push(o1 + o2);
                break;
            case '-':
                st.push(o1 - o2);
                break;
            case '*':
                st.push(o1 * o2);
                break;
            case '/':
                st.push(o1 / o2);
                break;
            }
        }
    }

    return st.top->data;
}
int main()
{
    string str;
    cout << "Enter the prefix expression  :  ";
    cin >> str;
    cout << "Answer  : " << evaluatePrefix(str) << endl;
    return 0;
}
```

Output 1:

```
Enter the prefix expression  :  +23
Answer  : 5
```

Output 2:

```
Enter the prefix expression  :  *+23+56
Answer  : 55
```

Output 3:

```
Enter the prefix expression  :  +9*26
Answer  : 21
```

Output 4:

```
Enter the prefix expression  :  +++1234
Answer  : 10
```

--END--

--POSTFIX EVALUATION---

Code =>

```
#include <bits/stdc++.h>
using namespace std;
class Node
{
public:
    float data;
    Node *next;
};
class Stack
{
public:
    Node *top = NULL;
    bool empty();
    void push(int);
    void pop();
};
bool Stack ::empty()
{
    if (top == NULL)
    {
        return true;
    }
    else
    {
        return false;
    }
}
void Stack ::push(int new_data)
{
    Node *new_node = new Node();
    new_node->data = new_data;
    new_node->next = top;
    top = new_node;
}
void Stack ::pop()
{
    if (empty())
    {
        cout << "Stack is Empty" << endl;
```

```cpp
    }
    else
    {
        Node *temp = top;
        top = top->next;
        delete (temp);
    }
}
float scanNum(char ch)
{
    int value;
    value = ch;
    return float(value - '0'); //return float from character
}
int isOperator(char ch)
{
    if (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^')
    {
        return 1; //character is an operator
    }
    else
    {
        return -1; //not an operator
    }
}
int isOperand(char ch)
{
    if (ch >= '0' && ch <= '9')
    {
        return 1; //character is an operand
    }
    else
    {
        return -1; //not an operand
    }
}
float operation(int a, int b, char op)
{
    //Perform operation
    if (op == '+')
    {
        return b + a;
    }
    else if (op == '-')
    {
        return b - a;
    }
    else if (op == '*')
    {
        return b * a;
    }
    else if (op == '/')
    {
        return b / a;
    }
    else if (op == '^')
    {
        return pow(b, a); //find b^a
    }
    else
    {
        return INT_MIN; //return negative infinity
    }
}
float postfixEval(string postfix)
{
    int a, b;
    Stack st;
    for (int i = 0; i < postfix.length(); i++)
    {
        //read elements and perform postfix evaluation
        if (isOperator(postfix[i]) != -1)
        {
            a = st.top->data;
            st.pop();
            b = st.top->data;
            st.pop();
            st.push(operation(a, b, postfix[i]));
        }
        else if (isOperand(postfix[i]) > 0)
        {
            st.push(scanNum(postfix[i]));
        }
    }
    return st.top->data;
}
```

```
int main()
{
    string str;
    cout << "Enter the PostFix Expression : ";
    cin >> str;
    cout << "Answer: " << postfixEval(str);
    return 0;
}
```

```
Output 1:
Enter the PostFix Expression : 231*+9-
Answer: -4

Output 2:
Enter the PostFix Expression : 651*+3+
Answer: 14

Output 3:
Enter the PostFix Expression : 45+2-
Answer: 7

Output 4:
Enter the PostFix Expression : 47+2+5+
Answer: 18
```

---END---