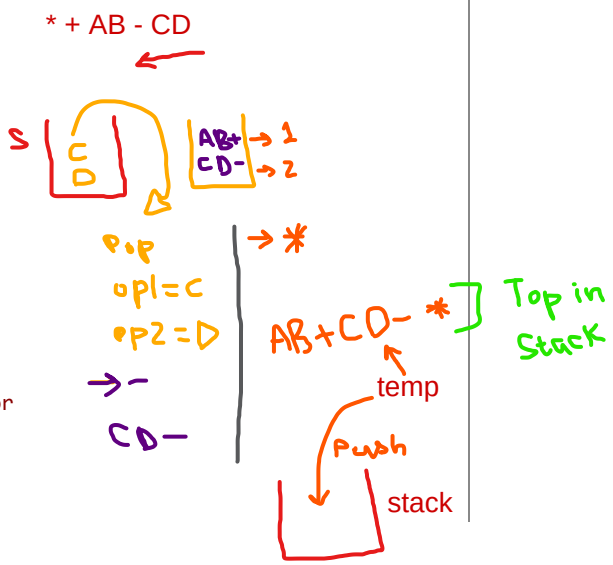
- **STACKS**

Algorithm for Prefix to Postfix:

- Read the Prefix expression in reverse order (from right to left)
- If the symbol is an operand, then push it onto the Stack
- If the symbol is an operator, then pop two operands from the Stack
  Create a string by concatenating the two operands and the operator after them.
  string = operand1 + operand2 + operator
  And push the resultant string back to Stack
- Repeat the above steps until end of Prefix expression.

```
                                        convert prefix to postfix
#include <iostream>
#include <stack>
using namespace std;

// function to check if character is operator or not
bool isOperator(char x)
{
      switch (x) {
      case '+':
      case '-':
      case '/':
      case '*':
                return true;
      }
      return false;
}

// Convert prefix to Postfix expression
string preToPost(string pre)
{

      stack<string> s;
      // length of expression
      int length = pre.size();

      // reading from right to left
      for (int i = length - 1; i >= 0; i--)
      {
                // check if symbol is operator
                if (isOperator(pre[i]))
                {
                          // pop two operands from stack
                          string op1 = s.top();
                          s.pop();
                          string op2 = s.top();
                          s.pop();

                          // concat the operands and operator
                          string temp = op1 + op2 + pre[i];

                          // Push string temp back to stack
                          s.push(temp);
```



```
                }

                // if symbol is an operand
                else {

                        // push the operand to the stack
                        s.push(string(1, pre[i]));
                }
        }

        // stack contains only the Postfix expression
        return s.top();
}

// Driver Code
int main()
{
        string pre = "*-A/BC-/AKL";
        cout << "Postfix : " << preToPost(pre);
        return 0;
}
```

* +AB - CD

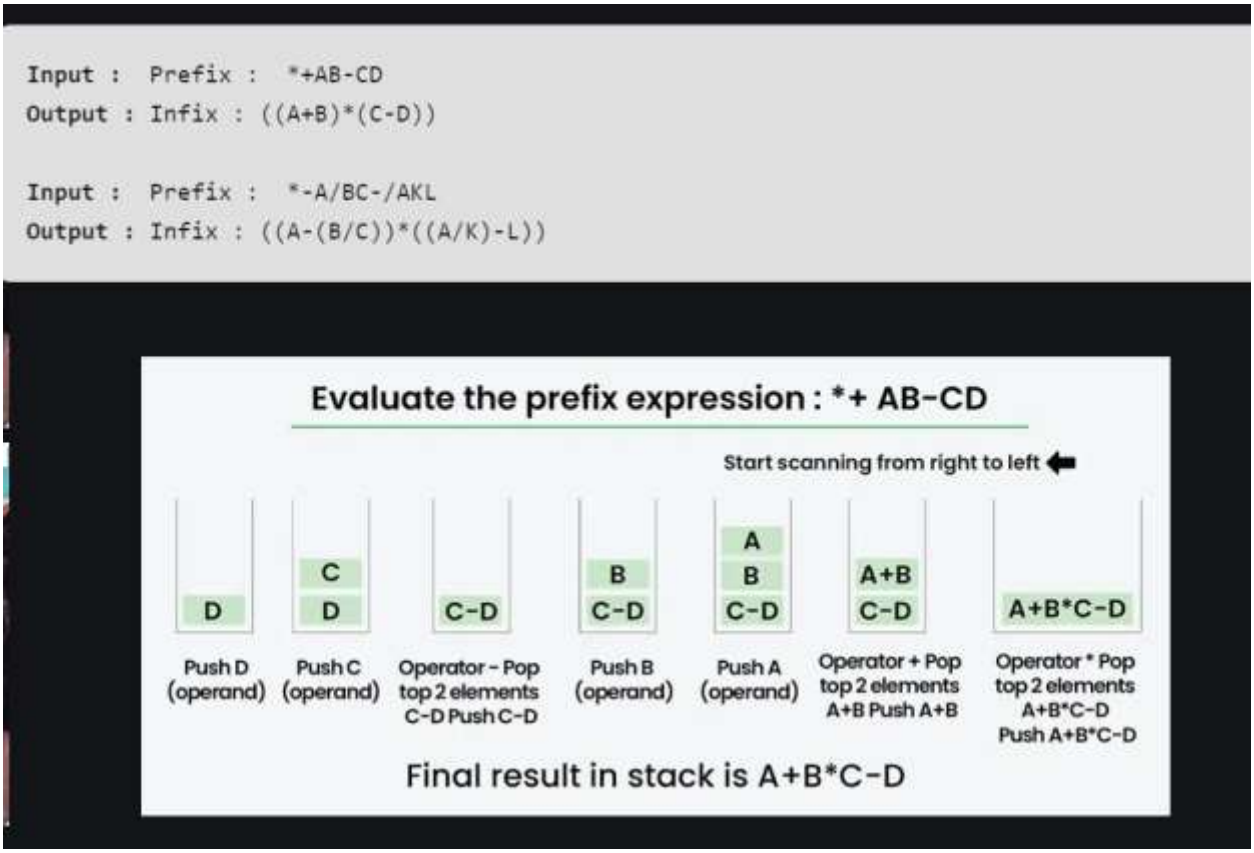
AB + CD- *

Output:

```
Postfix : ABC/-AK/L-*
```

Prefix to Infix

```
Input :  Prefix :  *+AB-CD
Output : Infix : ((A+B)*(C-D))

Input :  Prefix :  *-A/BC-/AKL
Output : Infix : ((A-(B/C))*((A/K)-L))
```



Algorithm for Prefix to Infix:

- Read the Prefix expression in reverse order (from right to left)
- If the symbol is an operand, then push it onto the Stack
- If the symbol is an operator, then pop two operands from the Stack
  Create a string by concatenating the two operands and the operator between them.
  string = (operand1 + operator + operand2)
  And push the resultant string back to Stack
- Repeat the above steps until the end of Prefix expression.
- At the end stack will have only 1 string i.e resultant string

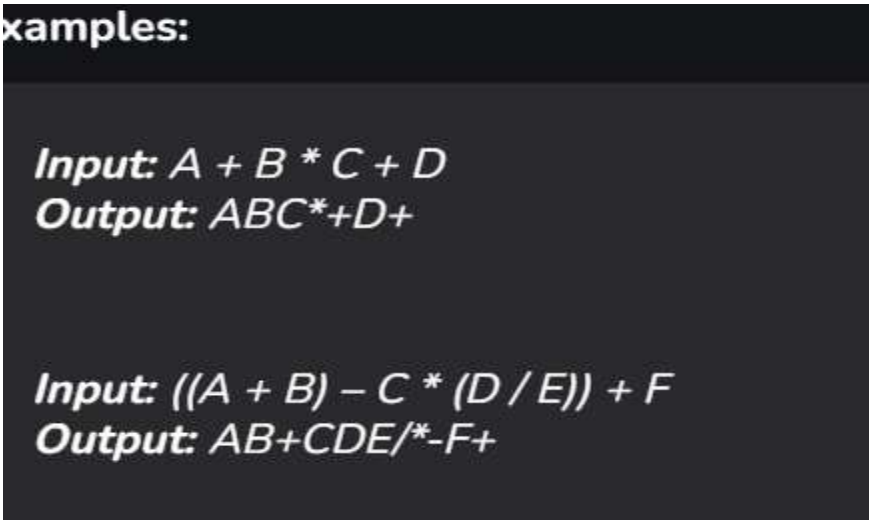
```
1.                                        convert prefix to Infix
2. #include <iostream>
3. #include <stack>
4. using namespace std;
5.
6. // function to check if character is operator or not
7. bool isOperator(char x) {
8.   switch (x) {
9.   case '+':
10.  case '-':
11.  case '/':
12.  case '*':
13.  case '^':
14.  case '%':
15.    return true;
16.  }
17.  return false;
18. }
19.
20. // Convert prefix to Infix expression
21. string preToInfix(string pre) {
22.   stack<string> s;
23.
24.   // length of expression
25.   int length = pre.size();
26.
27.   // reading from right to left
28.   for (int i = length - 1; i >= 0; i--) {
29.
30.     // check if symbol is operator

31.     if (isOperator(pre[i])) {
32.
33.       // pop two operands from stack
34.       string op1 = s.top();   s.pop();
35.       string op2 = s.top();   s.pop();
36.
37.       // concat the operands and operator

38.       string temp = "(" + op1 + pre[i] + op2 + ")";
39.
40.       // Push string temp back to stack
41.       s.push(temp);
42.     }
```

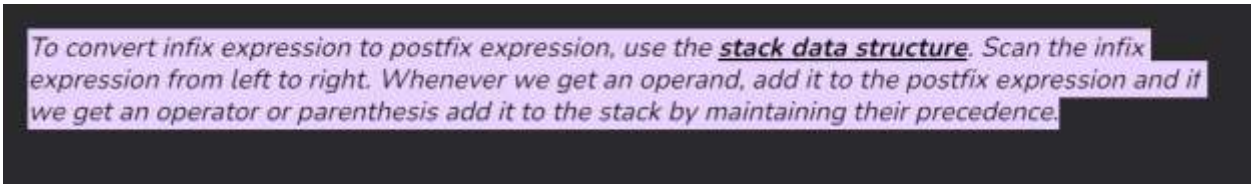
```
43.
44.     // if symbol is an operand
45.     else {
46.
47.       // push the operand to the stack
48.       s.push(string(1, pre[i]));
49.     }
50.   }
51.
52.   // Stack now contains the Infix expression
53.   return s.top();
54. }
55.
56. // Driver Code
57. int main() {
58.   string pre_exp = "*-A/BC-/AKL";
59.   cout << "Infix : " << preToInfix(pre_exp);
60.   return 0;
61. }
62.
```

Output:

Infix : ((A-(B/C))*((A/K)-L))

Infix to PostFix

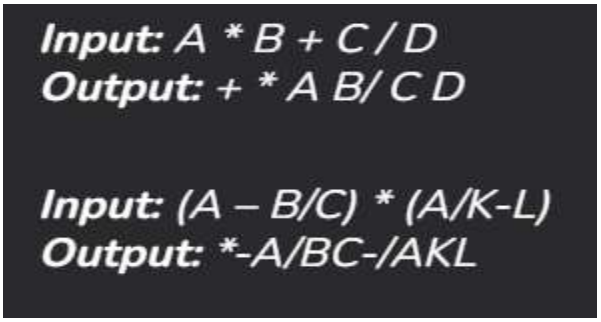


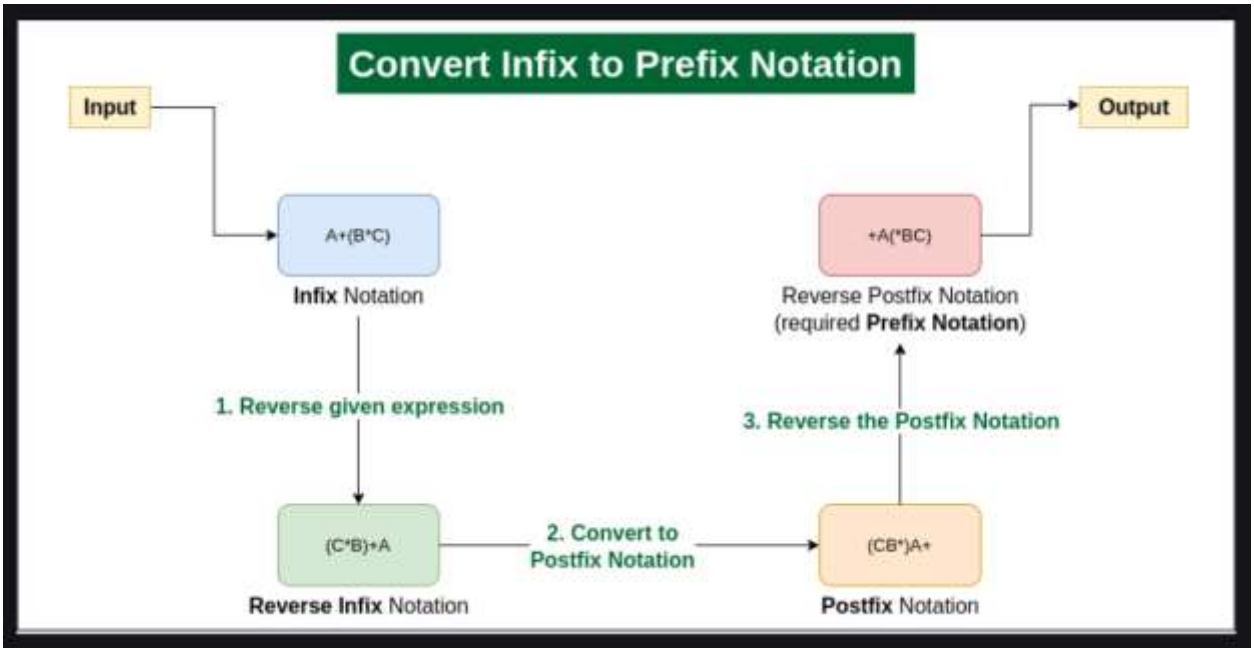


```
                            Infix to Post Fix
#include <iostream>
#include <stack>
     using namespace std;
using namespace std;

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

// Function to return associativity of operators
char associativity(char c) {
    if (c == '^')
        return 'R';
    return 'L'; // Default to left-associative
}

// The main function to convert infix expression
// to postfix expression
void infixToPostfix(string s) {
    stack<char> st;
    string result;

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

        // If the scanned character is
        // an operand, add it to the output string.
        if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9'))
            result += 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 add to the output string from the stack
        // until an ‘(‘ is encountered.
        else if (c == ')') {
            while (st.top() != '(')
            {
                result += st.top();
                st.pop();
            }
            st.pop(); // Pop '('
        }

        // If an operator is scanned
        else {
            while (!st.empty() && prec(s[i]) < prec(st.top()) ||
                   !st.empty() && prec(s[i]) == prec(st.top()) &&
                   associativity(s[i]) == 'L')
               {
                result += st.top();
                st.pop();
```

```
                }
            st.push(c);
        }
    }

    // Pop all the remaining elements from the stack
    while (!st.empty()) {

        result += st.top();
        st.pop();
    }

    cout << result << endl;
}

// Driver code
int main() {
    string exp = "a+b*(c^d-e)^(f+g*h)-i";

    // Function call
    infixToPostfix(exp);

    return 0;
}

Output: abcd^e-fgh*+^*+i-


Infix: a+b*c-d
Postfix: abc*+d-
```

Infix to Prefix






```
1.                                  infix to prefix
2.  #include <iostream>
3.  #include <stack>
4.      #include <bits/stdc++.h>
5.  using namespace std;
6.
7.  // Function to check if the character is an operator
8.  bool isOperator(char c)
9.  {
10.      return (!isalpha(c) && !isdigit(c));
11. }
12.
13. // Function to get the priority of operators
14. int getPriority(char C)
15. {
16.      if (C == '-' || C == '+')
17.                return 1;
18.      else if (C == '*' || C == '/')
19.                return 2;
20.      else if (C == '^')
21.                return 3;
22.      return 0;
23. }
24.
25. // Function to convert the infix expression to postfix
26. string infixToPostfix(string infix)
27. {
28.      infix = '(' + infix + ')';

29.      int l = infix.size();

30.      stack<char> char_stack;
31.      string output;
32.
33.      for (int i = 0; i < l; i++) {
34.
35.                // If the scanned character is an
36.                // operand, add it to output.
37.                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] == '(')
                        char_stack.push('(');

                // If the scanned character is an
                // ')', pop and output from the stack
                // until an '(' is encountered.
                else if (infix[i] == ')') {
                        while (char_stack.top() != '(') {

                                output += char_stack.top();
                                char_stack.pop();
                        }

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

                // Operator found
                else {
                        if (isOperator(char_stack.top())) {
                                if (infix[i] == '^') {
                                        while (
                                                getPriority(infix[i])
                                                <= getPriority(char_stack.top())) {
                                                output += char_stack.top();
                                                char_stack.pop();
                                        }
                                }
                                else {
                                        while (
                                                getPriority(infix[i])
                                                < getPriority(char_stack.top())) {
                                                output += char_stack.top();
                                                char_stack.pop();
                                        }
                                }

                                // Push current Operator on stack
                                char_stack.push(infix[i]);
                        }
                }
        }

        while (!char_stack.empty()) {
                output += char_stack.top();
                char_stack.pop();
        }
        return output;
}

// Function to convert infix to prefix notation
string infixToPrefix(string infix)
{
        // Reverse String and replace ( with ) and vice versa
        // Get Postfix
        // Reverse Postfix
        int l = infix.size();

        // Reverse infix
```

```
 99.      reverse(infix.begin(), infix.end());
100.
101.      // Replace ( with ) and vice versa
102.      for (int i = 0; i < l; i++) {
103.
104.                if (infix[i] == '(') {
105.                          infix[i] = ')';
106.                }
107.                else if (infix[i] == ')') {
108.                          infix[i] = '(';
109.                }
110.      }
111.
112.      string prefix = infixToPostfix(infix);
113.
114.      // Reverse postfix
115.      reverse(prefix.begin(), prefix.end());
116.
117.      return prefix;
118. }
119.
120. // Driver code
121. int main()
122. {
123.      string s = ("x+y*z/w+u");
124.
125.      // Function call
126.      cout << infixToPrefix(s) << std::endl;
127.      return 0;
128. }
129.

Output: ++x/*yzwu
```

OUTPUT:
Infix: (A+B)*(C-D)
PreFix: *+AB-CD
PostFix: AB+CD-*

cout << infixToPostfix(s) << endl;

PostFix to PreFix

```
Input :  Postfix : AB+CD-*
Output : Prefix :  *+AB-CD
Explanation : Postfix to Infix : (A+B) * (C-D)
              Infix to Prefix :  *+AB-CD

Input :  Postfix : ABC/-AK/L-*
Output : Prefix :  *-A/BC-/AKL
Explanation : Postfix to Infix : ((A-(B/C))*((A/K)-L))
              Infix to Prefix :  *-A/BC-/AKL
```

Algorithm for Postfix to Prefix:

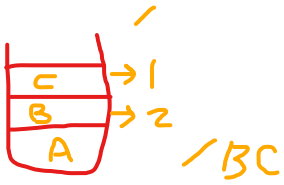
- *Read the Postfix expression from left to right*
- *If the symbol is an operand, then push it onto the Stack*
- *If the symbol is an operator, then pop two operands from the Stack*
  Create a string by concatenating the two operands and the operator before them.
  string = operator + operand2 + operand1
  And push the resultant string back to Stack
- *Repeat the above steps until end of Postfix expression.*

```
                                        convert postfix to prefix

#include<iostream>
#include<stack>
using namespace std;

// function to check if character is operator or not
bool isOperator(char x)
{
      switch (x) {
      case '+':
      case '-':
      case '/':
      case '*':
                return true;
      }
      return false;
}

// Convert postfix to Prefix expression
string postToPre(string post)
{
      stack<string> s;

      // length of expression
      int length = post.size();

      // reading from left to right
      for (int i = 0; i < length; i++) {

                // check if symbol is operator
                if (isOperator(post[i])) {

                          // pop two operands from stack
                          string op1 = s.top();
                          s.pop();
                          string op2 = s.top();
                          s.pop();

                          // concat the operands and operator
                          string temp = post[i] + op2 + op1;

                          // Push string temp back to stack
                          s.push(temp);
                }
```

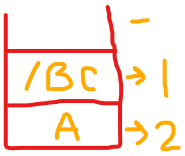
AB + CD - *
→

```
46.
47.                 // if symbol is an operand
48.                 else {
49.
50.                         // push the operand to the stack
51.                         s.push(string(1, post[i]));
52.                 }
53.         }
54.
55.         string ans = "";
56.         while (!s.empty()) {
57.                 ans += s.top();
58.                 s.pop();
59.         }
60.         return ans;
61. }
62.
63. // Driver Code
64. int main()
65. {
66.         string post_exp = "ABC/-AK/L-*";
67.
68.         // Function call
69.         cout << "Prefix : " << postToPre(post_exp);
70.         return 0;
71. }
72.
Output:

Prefix : *-A/BC-/AKL
```

return s.top();



Move two back



Postfix to Infix

```
Input : abc++
Output : (a + (b + c))

Input  : ab*c+
Output : ((a*b)+c)
```

We have already discussed Infix to Postfix. Below is algorithm for Postfix to Infix.

Algorithm

1.While there are input symbol left

...1.1 Read the next symbol from the input.

2.If the symbol is an operand

...2.1 Push it onto the stack.

3.Otherwise,

...3.1 the symbol is an operator.

...3.2 Pop the top 2 values from the stack.

...3.3 Put the operator, with the values as arguments and form a string.

...3.4 Push the resulted string back to stack.

4.If there is only one value in the stack

...4.1 That value in the stack is the desired infix string.

Below is the implementation of above approach:

```
// CPP program to find infix for
a given postfix.

#include <bits/stdc++.h>
using namespace std;

bool isOperand(char x)
{
return (x >= 'a' && x <= 'z') ||
                (x >= 'A' && x <= 'Z');
}

// Get Infix for a given postfix
// expression
string getInfix(string exp)
{
      stack<string> s;

      for (int i=0; exp[i]!='\0'; i++)
      {
                // Push operands
                if (isOperand(exp[i]))
                {
                string op(1, exp[i]);
                s.push(op);
                }

                // We assume that input is
                // a valid postfix and expect
                // an operator.
```

```
30.                 else
31.                 {
32.                           string op1 = s.top();
33.                           s.pop();
34.                           string op2 = s.top();
35.                           s.pop();
36.                           s.push("(" + op2 + exp[i] +
37.                                     op1 + ")");
38.                 }
39.       }
40.
41.       // There must be a single element
42.       // in stack now which is the required
43.       // infix.
44.       return s.top();
45. }
46.
47. // Driver code
48. int main()
49. {
50.       string exp = "ab*c+";
51.       cout << getInfix(exp);
52.       return 0;
53. }
54.
```

Output:

```
((a*b)+c)
```