

## **EXPRESSION CONVERSION & EVALUATION USING STACK**

### **Convert Infix to Postfix notation**

#### **What is infix notation?**

When the operator is written in between the operands, then it is known as **infix notation**. Operand does not have to be always a constant or a variable; it can also be an expression itself.

#### **For example,**

$$(p + q) * (r + s)$$

In the above expression, both the expressions of the multiplication operator are the operands, i.e., **(p + q)**, and **(r + s)** are the operands.

In the above expression, there are three operators. The operands for the first plus operator are p and q, the operands for the second plus operator are r and s. While performing the **operations on the expression, we need to follow some set of rules to evaluate the result**. In the above expression, addition operation would be performed on the two expressions, i.e., p+q and r+s, and then the multiplication operation would be performed.

#### **Syntax of infix notation is given below:**

**<operand> <operator> <operand>**

If there is only one operator in the expression, we do not require applying any rule. For example, 5 + 2; in this expression, addition operation can be performed between the two operands (5 and 2), and the result of the operation would be 7.

If there are multiple operators in the expression, then some rule needs to be followed to evaluate the expression.

If the expression is:

$$4 + 6 * 2$$

If the plus operator is evaluated first, then the expression would look like:

$$10 * 2 = 20$$

If the multiplication operator is evaluated first, then the expression would look like:

$$4 + 12 = 16$$

The above problem can be resolved by following the operator precedence rules. In the algebraic expression, the order of the operator precedence is given in the below table:

Operators	Symbols
Parenthesis	( ), { }, [ ]
Exponents	^
Multiplication and Division	*, /
Addition and Subtraction	+, -

The first preference is given to the parenthesis; then next preference is given to the exponents. In the case of multiple exponent operators, then the operation will be applied from right to left.

**For example:**

$$2^{2^3} = 2^8$$

$$= 256$$

After exponent, multiplication, and division operators are evaluated. If both the operators are present in the expression, then the operation will be applied from left to right.

The next preference is given to addition and subtraction. If both the operators are available in the expression, then we go from left to right.

The operators that have the same precedence termed as **operator associativity**. If we go from left to right, then it is known as left-associative. If we go from right to left, then it is known as right-associative.

**Problem with infix notation**

To evaluate the infix expression, we should know about the **operator precedence** rules, and if the operators have the same precedence, then we should follow the **associativity** rules. The use of parenthesis is very important in infix notation to **control the order** in which the operation to be performed. Parenthesis improves the readability of the expression. An infix expression is the most common way of writing expression, but it is not easy to parse and evaluate the infix expression without ambiguity. So, mathematicians and logicians studied this problem and discovered two other ways of writing expressions which are prefix and postfix. Both expressions do not require any parenthesis and can be parsed without ambiguity. It does not require operator precedence and associativity rules.

The following table briefly tries to show the difference in all three notations –

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
1	$a + b$	$+ a b$	$a b +$
2	$(a + b) * c$	$* + a b c$	$a b + c *$
3	$a * (b + c)$	$* a + b c$	$a b c + *$
4	$a / b + c / d$	$+ / a b / c d$	$a b / c d / +$
5	$(a + b) * (c + d)$	$* + a b + c d$	$a b + c d + *$
6	$((a + b) * c) - d$	$- * + a b c d$	$a b + c * d -$

## Postfix Expression

The postfix expression is an expression in which the operator is written after the operands. For example, the postfix expression of infix notation  $(2+3)$  can be written as  $23+$ .

Some key points regarding the postfix expression are:

- In postfix expression, operations are performed in the order in which they have written from left to right.
- It does not any require any parenthesis.
- We do not need to apply operator precedence rules and associativity rules.

## Conversion of infix to postfix

Here, we will use the stack data structure for the conversion of infix expression to prefix expression. Whenever an operator will encounter, we push operator into the stack. If we encounter an operand, then we append the operand to the expression.

### Rules for the conversion from infix to postfix expression

1. Print the operand as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator on to the stack.
3. If the incoming symbol is '(', push it on to the stack.
4. If the incoming symbol is ')', pop the stack and print the operators until the left parenthesis is found.
5. If the incoming symbol has **higher precedence than the top of the stack, push it on the stack.**
6. If the incoming symbol has **lower precedence than the top of the stack, pop and print the top of the stack.** Then test the incoming operator against the new top of the stack.
7. If the incoming operator has the same precedence with the top of the stack then use the associativity rules. If the associativity is from left to right then pop and print the top of the stack then push the incoming operator. If the associativity is from right to left then push the incoming operator.
8. At the end of the expression, pop and print all the operators of the stack.

Let's understand through an example.

**Infix expression:  $K + L - M * N + (O \wedge P) * W / U / V * T + Q$**

Input Expression	Stack	Postfix Expression
K		K
+	+	
L	+	K L

-	-	$KL+$
M	-	$KL+M$
*	- *	$KL+M$
N	- *	$KL+MN$
+	+	$KL+MN^*$ $KL+MN^*-$
(	+(	$KL+MN^*-$
O	+(	$KL+MN^*-O$
^	+( ^	$KL+MN^*-O$
P	+( ^	$KL+MN^*-OP$
)	+	$KL+MN^*-OP^{\wedge}$
*	+ *	$KL+MN^*-OP^{\wedge}$
W	+ *	$KL+MN^*-OP^{\wedge}W$
/	+ /	$KL+MN^*-OP^{\wedge}W^*$
U	+ /	$KL+MN^*-OP^{\wedge}W^*U$
/	+ /	$KL+MN^*-OP^{\wedge}W^*U/$
V	+ /	$KL+MN^*-OP^{\wedge}W^*U/V$
*	+ *	$KL+MN^*-OP^{\wedge}W^*U/V/$

T	+ *	KL+MN*-OP^W*U/V/T
+	+	KL+MN*-OP^W*U/V/T* KL+MN*-OP^W*U/V/T*+
Q	+	KL+MN*-OP^W*U/V/T*Q
		KL+MN*-OP^W*U/V/T*+Q+

The final postfix expression of infix expression  $(K + L - M * N + (O^P) * W / U / V * T + Q)$  is  $KL+MN*-OP^W*U/V/T*+Q+$ .

1. Convert  $A * (B + C) * D$  to postfix notation = **ABC+\*D\***
2. Convert  $(2-3+4)*(5+6*7)$  to postfix notation = **23-4+567\*+\* = 141**

## Parsing Expressions

As we have discussed, it is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed.

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

## Precedence

When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example –

$$a + b * c \Rightarrow a + (b * c)$$

As multiplication operation has precedence over addition,  $b * c$  will be evaluated first. A table of operator precedence is provided later.

## Associativity

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression  $a + b - c$ , both  $+$  and  $-$  have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both  $+$  and  $-$  are left associative, so the expression will be evaluated as

$$(a + b) - c.$$

Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) –

Sr.No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication ( * ) & Division ( / )	Second Highest	Left Associative
3	Addition ( + ) & Subtraction ( - )	Lowest	Left Associative

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example –

In  $a + b * c$ , the expression part  $b * c$  will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for  $a + b$  to be evaluated first, like  $(a + b) * c$ .

## Algorithm to evaluate the postfix expression

### Postfix Evaluation Algorithm

We shall now look at the algorithm on how to evaluate postfix notation –

Step 1 – scan the expression from left to right  
Step 2 – if it is an operand push it to stack  
Step 3 – if it is an operator pull operand from stack and perform operation  
Step 4 – store the output of step 3, back to stack  
Step 5 – scan the expression until all operands are consumed  
Step 6 – pop the stack and perform operation

**Let's understand the above algorithm through an example.**

Infix expression:  $2 + 3 * 4$

We will start scanning from the left most of the expression. The multiplication operator is an operator that appears first while scanning from left to right. Now, the expression would be:

Expression =  $2 + 3 \times 4$

=  $2 + 12$

Again, we will scan from left to right, and the expression would be:

Expression =  $2 \ 12 \ +$

= 14

### Evaluation of postfix expression using stack.

- Scan the expression from left to right.
- If we encounter any operand in the expression, then we push the operand in the stack.
- When we encounter any operator in the expression, then we pop the corresponding operands from the stack.
- When we finish with the scanning of the expression, the final value remains in the stack.

**Let's understand the evaluation of postfix expression using stack.**

**Example 1: Postfix expression:  $2 \ 3 \ 4 \ * \ +$**

Input	Stack	
2 3 4 * +	empty	Push 2
3 4 * +	2	Push 3
4 * +	3 2	Push 4
* +	4 3 2	Pop 4 and 3, and perform $4 \times 3 = 12$ . Push 12 into the stack.
+	12 2	Pop 12 and 2 from the stack, and perform $12 + 2 = 14$ . Push 14 into the stack.

The result of the above expression is 14.

**Example 2: Postfix expression:  $3 \ 4 \ * \ 2 \ 5 \ * \ +$**



Input	Stack	
3 4 * 2 5 * +	empty	Push 3
4 * 2 5 * +	3	Push 4
*2 5 * +	4 3	Pop 3 and 4 from the stack and perform $3*4 = 12$ . Push 12 into the stack
2 5 * +	12	Push 2
5 * +	2 12	Push 5
*+	5 2 12	Pop 5 and 2 from the stack and perform $5*2 = 10$ . Push 10 into the stack
+	10 12	Pop 10 and 12 from the stack and perform $10+12 = 22$ . Push 22 into the stack

The result of the above expression is 22.

**Example 3: Postfix expression: 5, 6, 2, +, \*, 12, 4, /, -**

Symbol Scanned	Stack
5	5
6	5, 6
2	5, 6, 2
+	5, 8
*	40
12	40, 12
4	40, 12, 4

/	40, 3
-	37

The result of the above expression is 37.

### A program to convert and evaluate infix notation to postfix notation

```

1. #include<stdio.h>
2. #include<string.h>
3.
4. //char stack
5. char stack[25];
6. int top = -1;
7.
8. void push(char item) {
9.     stack[++top] = item;
10.}
11.
12.char pop() {
13.    return stack[top--];
14.}
15.
16.//returns precedence of operators
17.int precedence(char symbol) {
18.
19.    switch(symbol) {
20.        case '+':
21.            case '-':
22.                return 2;
23.            break;
24.        case '*':
25.            case '/':
26.                return 3;
27.            break;
28.        case '^':
29.            return 4;
30.            break;
31.        case '(':
32.            case ')':

```

```

33.     case '#':
34.         return 1;
35.         break;
36.     }
37. }
38.
39. //check whether the symbol is operator?
40. int isOperator(char symbol) {
41.
42.     switch(symbol) {
43.         case '+':
44.         case '-':
45.         case '*':
46.         case '/':
47.         case '^':
48.         case '(':
49.         case ')':
50.             return 1;
51.             break;
52.         default:
53.             return 0;
54.     }
55. }
56.
57. //converts infix expression to postfix
58. void convert(char infix[], char postfix[]) {
59.     int i, symbol, j = 0;
60.     stack[++top] = '#';
61.
62.     for(i = 0; i < strlen(infix); i++) {
63.         symbol = infix[i];
64.
65.         if(isOperator(symbol) == 0) {
66.             postfix[j] = symbol;
67.             j++;
68.         } else {
69.             if(symbol == '(') {
70.                 push(symbol);
71.             } else {
72.                 if(symbol == ')') {
73.
74.                     while(stack[top] != '(') {
75.                         postfix[j] = pop();

```

```

76.         j++;
77.     }
78.
79.     pop(); //pop out (.
80. } else {
81.     if(precedence(symbol)>precedence(stack[top])) {
82.         push(symbol);
83.     } else {
84.
85.         while(precedence(symbol)<=precedence(stack[top])) {
86.             postfix[j] = pop();
87.             j++;
88.         }
89.
90.         push(symbol);
91.     }
92. }
93. }
94. }
95. }
96.
97. while(stack[top] != '#') {
98.     postfix[j] = pop();
99.     j++;
100. }
101.
102. postfix[j]='\0'; //null terminate string.
103. }
104.
105. //int stack
106. int stack_int[25];
107. int top_int = -1;
108.
109. void push_int(int item) {
110.     stack_int[++top_int] = item;
111. }
112.
113. char pop_int() {
114.     return stack_int[top_int--];
115. }
116.
117. //evaluates postfix expression
118. int evaluate(char *postfix){

```

```

119.
120. char ch;
121. int i = 0, operand1, operand2;
122.
123. while( (ch = postfix[i++]) != '\0') {
124.
125.     if(isdigit(ch)) {
126.         push_int(ch-'0'); // Push the operand
127.     } else {
128.         //Operator,pop two operands
129.         operand2 = pop_int();
130.         operand1 = pop_int();
131.
132.         switch(ch) {
133.             case '+':
134.                 push_int(operand1+operand2);
135.                 break;
136.             case '-':
137.                 push_int(operand1-operand2);
138.                 break;
139.             case '*':
140.                 push_int(operand1*operand2);
141.                 break;
142.             case '/':
143.                 push_int(operand1/operand2);
144.                 break;
145.         }
146.     }
147. }
148.
149. return stack_int[top_int];
150. }
151.
152. void main() {
153.     char infix[25] = "1*(2+3)", postfix[25];
154.     convert(infix, postfix);
155.
156.     printf("Infix expression is: %s\n", infix);
157.     printf("Postfix expression is: %s\n", postfix);
158.     printf("Evaluated expression is: %d\n", evaluate(postfix));
159. }

```

160. If we compile and run the above program, it will produce the following result –

161. Output

162. Infix expression is:  $1*(2+3)$

163. Postfix expression is:  $123+*$

164. Result is: 5

## Convert infix to prefix notation

### What is Prefix notation?

A prefix notation is another form of expression but it does not require other information such as precedence and associativity, whereas an infix notation requires information of precedence and associativity. It is also known as **polish notation**. In prefix notation, an operator comes before the operands. The syntax of prefix notation is given below:

**<operator> <operand> <operand>**

**For example**, if the infix expression is  $5+1$ , then the prefix expression corresponding to this infix expression is  $+51$ .

**If the infix expression is:**

**$a * b + c$**

↓

**$*ab+c$**

↓

**$+*abc$  (Prefix expression)**

**Consider another example:**

**$A + B * C$**

**First scan:** In the above expression, multiplication operator has a higher precedence than the addition operator; the prefix notation of  $B*C$  would be  $(*BC)$ .

**$A + *BC$**

**Second scan:** In the second scan, the prefix would be:

+A \*BC

In the above expression, we use two scans to convert infix to prefix expression. If the expression is complex, then we require a greater number of scans. We need to use that method that requires only one scan, and provides the desired result. If we achieve the desired output through one scan, then the algorithm would be efficient. This is possible only by using a stack.

### Rules for the conversion of infix to prefix expression:

1. First, reverse the infix expression given in the problem.
2. Scan the expression from left to right.
3. Whenever the operands arrive, print them.
4. If the operator arrives and the stack is found to be empty, then simply push the operator into the stack.
5. If the incoming operator has **higher precedence than the TOP of the stack, push the incoming operator into the stack.**
6. If the incoming operator has **the same precedence with a TOP of the stack, push the incoming operator into the stack.**
7. If the incoming operator has **lower precedence than the TOP of the stack, pop, and print the top of the stack.** Test the incoming operator against the top of the stack again and **pop the operator from the stack till it finds the operator of a lower precedence or same precedence.**
8. If the incoming operator has the **same precedence with the top of the stack and the incoming operator is ^, then pop the top of the stack till the condition is true. If the condition is not true, push the ^ operator.**
9. When we reach the end of the expression, pop, and print all the operators from the top of the stack.
10. If the operator is ')', then push it into the stack.
11. If the operator is '(', then pop all the operators from the stack till it finds ) opening bracket in the stack.
12. If the top of the stack is ')', push the operator on the stack.
13. At the end, reverse the output.

### Pseudocode of infix to prefix conversion

Function InfixtoPrefix( stack, infix)

infix = reverse(infix)

loop i = 0 to infix.length

**if** infix[i] is operand → prefix+= infix[i]

**else if** infix[i] is '(' → stack.push(infix[i])

**else if** infix[i] is ')' → pop and print the values of stack till the symbol ')' is not found

**else if** infix[i] is an operator(+, -, \*, /, ^) →

**if** the stack is empty then push infix[i] on the top of the stack.

Else →

If precedence(infix[i] > precedence(stack.top))

→ Push infix[i] on the top of the stack

**else if**(infix[i] == precedence(stack.top) && infix[i] == '^')

→ Pop and print the top values of the stack till the condition is **true**

→ Push infix[i] into the stack

**else if**(infix[i] == precedence(stack.top))

→ Push infix[i] on to the stack

Else **if**(infix[i] < precedence(stack.top))

→ Pop the stack values and print them till the stack is not empty and infix[i] < precedence(stack.top)

→ Push infix[i] on to the stack

End loop

Pop and print the remaining elements of the stack

Prefix = reverse(prefix)

**return**

### Conversion of Infix to Prefix using Stack

**K + L - M \* N + (O^P) \* W/U/V \* T + Q**

If we are converting the expression from infix to prefix, we need first to reverse the expression.

The Reverse expression would be:



$$Q + T * V/U/W * ) P^O(+ N*M - L + K$$

To obtain the prefix expression, we have created a table that consists of three columns, i.e., input expression, stack, and prefix expression. When we encounter any symbol, we simply add it into the prefix expression. If we encounter the operator, we will push it into the stack.

Input expression	Stack	Prefix expression
Q		Q
+	+	Q
T	+	QT
*	+*	QT
V	+*	QTV
/	+*/	QTV
U	+*/	QTVU
/	+*//	QTVU
W	+*//	QTVUW
*	+*//*	QTVUW
)	+*//*)	QTVUW
P	+*//*)	QTVUWP
^	+*//*)^	QTVUWP
O	+*//*)^	QTVUWPO

(	+*//*	QTVUWPO^
+	++	QTVUWPO^*//*
N	++	QTVUWPO^*//*N
*	++*	QTVUWPO^*//*N
M	++*	QTVUWPO^*//*NM
-	++-	QTVUWPO^*//*NM*
L	++-	QTVUWPO^*//*NM*L
+	+++	QTVUWPO^*//*NM*L
K	+++	QTVUWPO^*//*NM*LK
		QTVUWPO^*//*NM*LK+++

The above expression, i.e., QTVUWPO^\*//\*NM\*LK+---, is not a final expression. We need to reverse this expression to obtain the prefix expression.

**Final Prefix expression is : +++-KL\*MN\*//\*^OPWUVTQ**

## Evaluation of Prefix Expression using Stack

**Step 1:** Initialize a pointer 'S' pointing to the end of the expression.

**Step 2:** If the symbol pointed by 'S' is an operand then push it into the stack.

**Step 3:** If the symbol pointed by 'S' is an operator then pop two operands from the stack. Perform the operation on these two operands and stores the result into the stack.

**Step 4:** Decrement the pointer 'S' by 1 and move to step 2 as long as the symbols left in the expression.

**Step 5:** The final result is stored at the top of the stack and return it.

**Step 6:** End

**Let's understand the evaluation of prefix expression through an example.**

**Expression:** +, -, \*, 2, 2, /, 16, 8, 5

**First, we will reverse the expression given above.**

**Expression:** 5, 8, 16, /, 2, 2, \*, -, +

**We will use the stack data structure to evaluate the prefix expression.**

Symbol Scanned	Stack
5	5
8	5, 8
16	5, 8, 16
/	5, 2
2	5, 2, 2
2	5, 2, 2, 2
*	5, 2, 4
-	5, 2
+	7

**The final result of the above expression is 7.**

## Conversion of Prefix to Postfix expression

Here, we will see the conversion of prefix to postfix expression using a stack data structure.

### Rules for prefix to postfix expression using stack data structure:

- Scan the prefix expression from right to left, i.e., reverse.
- If the incoming symbol is an operand then push it into the stack.
- If the incoming symbol is an operator then pop two operands from the stack. Once the operands are popped out from the stack, we add the incoming symbol after the operands. When the operator is added after the operands, then the expression is pushed back into the stack.
- Once the whole expression is scanned, pop and print the postfix expression from the stack.

### Pseudocode for prefix to postfix conversion

1. Function PrefixToPostfix(string prefix)
2. 1. Stack s
3. 2. Loop: i = prefix.length-1 to 0
4. • **if** prefix[i] is operand ->
5. s.push(prefix[i])
6. • **else if** prefix[i] is operator->
7. op1 = s.top()
8. s.pop()
9. op2 = s.top()
10. s.pop()
11. exp = op1 + op2 + prefix[i]
12. s.push(exp)
13. End Loop
14. 3. Return s.top

Let's understand the conversion of Prefix to Postfix expression using Stack through an example.

If the expression is: \* - A / B C - / A K L

Symbols to be scanned	Action	Stack	Description
L	Push L into the stack	L	
K	Push K into the stack	L, K	
A	Push A into the stack	L, K, A	
/	Pop A from the stack Pop K from the stack Push A K / into the stack	L, A K /	Pop two operands from the stack, i.e., A and K. Add '/' operator after K operand, i.e., AK/. Push AK/ into the stack.
-	Pop A K / and L from the stack. Push (A K / L -) into the stack	A K / L -	Pop two operands from the stack, i.e., AK/ and L. Add '-' operator after 'L' operand.
C	Push C into the stack	AK/L-, C	
B	Push B into the stack	AK/L-, C, B	
/	Pop B and C from the stack. Push BC/ into the stack.	AK/L-, BC/	Pop two operands from the stack, i.e., B and C. Add '/' operator after C operator, i.e., BC/. Push BC/ into the stack.
A	Push A into the stack	AK/L-, BC/, A	
-	Pop BC/ and A from the stack. Push ABC/- into the stack.	AK/L-, ABC/-	Pop two operands from the stack, i.e., A and BC/. Add '-' operator after '/'.

*	Pop ABC/- and AK/L- from the stack. Push ABC/AK/L-* into the stack.	ABC/- AK/L-*	Pop two operands from the stack, i.e., ABC/-, and AK/L- . Add '*' operator after L and '-' operator, i.e., ABC/-AK/L-*.
---	---	-----------------	---

## Conversion of Postfix to Prefix expression

There are two ways of converting a postfix into a prefix expression:

1. Conversion of Postfix to Prefix expression manually.
  2. Conversion of Postfix to Prefix expression using stack.
- 1. The following are the steps required to convert postfix into prefix expression:**
- Scan the postfix expression from left to right.
  - Select the first two operands from the expression followed by one operator.
  - Convert it into the prefix format.
  - Substitute the prefix sub expression by one temporary variable
  - Repeat this process until the entire postfix expression is converted into prefix expression.

### 2. Conversion of Postfix to Prefix expression using Stack

The following are the steps used to convert postfix to prefix expression using stack:

- Scan the postfix expression from left to right.
- If the element is an operand, then push it into the stack.
- If the element is an operator, then pop two operands from the stack.
- Create an expression by concatenating two operands and adding operator before the operands.
- Push the result back to the stack.
- Repeat the above steps until we reach the end of the postfix expression.

## Pseudocode for the conversion of Postfix to Prefix

1. Function PostfixToPrefix(string postfix)
2. Stack s
3. Loop: i = 0 to postfix.length
4. **if** postfix[i] is operand ->
5. s.push(postfix[i])
- 6.
7. **else if** postfix[i] is operator->
8. op1 = s.top()
9. s.pop()
10. op2 = s.top()
11. s.pop()
12. expression = postfix[i] + op2 + op1
13. s.push(expression)
14.       end loop
15.       **return** s.top

**Let's understand the conversion of postfix to prefix expression using stack.**

**If the Postfix expression is given as:**

**AB + CD - \***

Symbol Scanned	Action	Stack	Description
A	Push A into the stack	A	
B	Push B into the stack	AB	
+	Pop B from the stack Pop A from the stack Push +AB into the stack.	+AB	Pop two operands from the stack, i.e., A and B. Add '+' operator before the operands AB, i.e., +AB.

C	Push C into the stack	+ABC	
D	Push D into the stack	+ABCD	
-	Pop D from the stack. Pop C from the stack. Push -CD into the stack	+AB - CD	Pop two operands from the stack, i.e., D and C. Add '-' operator before the operands CD, i.e., -CD.
*	Pop -CD from the stack. Pop +AB from the stack. Push *+AB - CD into the stack.	*+AB - CD	Pop two operands from the stack, i.e., -CD and +AB. Add '*' operator before +AB then the expression would become *+AB-CD.

The prefix expression of the above postfix expression is \*+AB-CD.

### Implementation of Postfix to Prefix conversion in C

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

#define MAX 20

char str[MAX], stack[MAX];
int top = -1;

void push(char c)
{
    stack[++top] = c;
}

char pop()
{
    return stack[top--];
}
```



```

// A utility function to check if the given character is operand
int checkIfOperand(char ch)
{
    return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');
}

//function to check if it is an operator
int isOperator(char x)
{
    switch (x) {
        case '+':
        case '-':
        case '/':
        case '*':
            return 1;
    }
    return 0;
}

void postfixToPrefix()
{
    int n, i, j = 0;
    char c[20];
    char a, b, op;

    printf("Enter the postfix expression\n");
    scanf("%s", str);

    n = strlen(str);

    for (i = 0; i < MAX; i++)
        stack[i] = '\0';
    printf("Prefix expression is:\t");

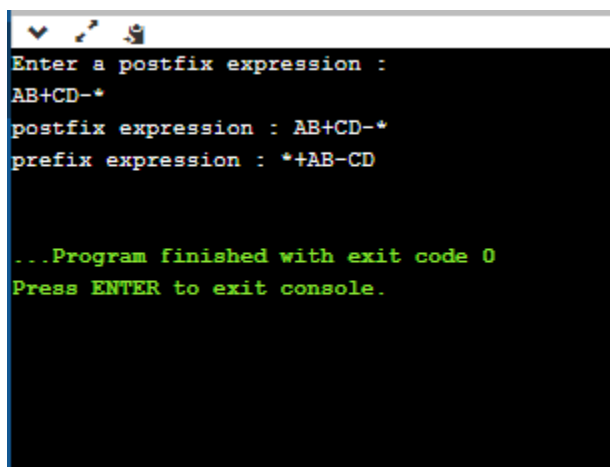
    for (i = n - 1; i >= 0; i--)
    {
        if (isOperator(str[i]))
        {
            push(str[i]);
        } else
        {
            c[j++] = str[i];
            while ((top != -1) && (stack[top] == '#'))
            {
                a = pop();
                c[j++] = pop();
            }
            push('#');
        }
    }
    c[j] = '\0';

    i = 0;
    j = strlen(c) - 1;
    char d[20];

```

```
while (c[i] != '\0') {  
    d[j--] = c[i++];  
}  
  
printf("%s\n", d);  
}  
int main()  
{  
    postfixToprefix();  
  
    return 0;  
}
```

## Output



A screenshot of a terminal window with a black background and green text. The window has a title bar with standard Linux window controls. The output shows the program prompting for a postfix expression, receiving 'AB+CD-\*', and then displaying the converted prefix expression '\*+AB-CD'. It concludes with a message about the program finishing with exit code 0 and a prompt to press ENTER to exit the console.

```
Enter a postfix expression :  
AB+CD-*  
postfix expression : AB+CD-*  
prefix expression : *+AB-CD  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```