Infix To Postfix Conversion Using Stack

One of the applications of Stack is in the conversion of arithmetic expressions in high-level programming languages into machine readable form. As our computer system can only understand and work on a binary language, it assumes that an arithmetic operation can take place in two operands only e.g., **A+B, C*D,D/A** etc. But in our usual form an arithmetic expression may consist of more than one operator and two operands e.g. **(A+B)*C(D/(J+D))**.

These complex arithmetic operations can be converted into polish notation using stacks which then can be executed in two operands and an operator form.

Infix Expression

It follows the scheme of **<operand><operator><operand>** i.e. an <operator> is preceded and succeeded by an <operand>. Such an expression is termed infix expression. E.g., **A+B**

Postfix Expression

It follows the scheme of **<operand><operand><operator>** i.e. an <operator> is succeeded by both the <operand>. E.g., **AB+**

Algorithm to convert Infix To Postfix

1. Scan the infix expression from left to right.

2. If the scanned character is an operand, output it.

3. Else,

1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '('), push it.

2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

4. If the scanned character is an '(', push it to the stack.

5. If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.

6. Repeat steps 2-6 until infix expression is scanned.

7. Print the output

8. Pop and output from the stack until it is not empty.

Let’s take an examples to better understand the algorithm

Infix Expression	Postfix Expression
A + B * C + D	A B C * + D +
(A + B) * (C + D)	A B + C D + *
A * B + C * D	A B * C D * +
A + B + C + D	A B + C + D +

Infix Expression	Postfix Expression

Examples of Infix-to-Postfix Conversion

Infix expression: a+b*c-d/e*f

Token	operator stack → top	postfix string
A		a
+	+	
B		ab
*	+*	
C		ABC
-	+	ABC*
		ABC*+
	-	
D		ABC*+d
/	-/	
E		ABC*+de
*	-	ABC*+de/
	-*	
F		ABC*+de/f
	-	ABC*+de/f*
		ABC*+de/f*-

Infix expression: (a+b*c-d)/ (e*f)

Token	operator stack → top	postfix string
(	(	
a		a
+	(+	
b		ab
*	(+*	
c		abc
-	(+	ABC*
	(	ABC*+
	(-	
d		abc*+d
)		ABC*+d-
/	/	
(	/(	
e		abc*+d-e
*	/(*	
f		abc*+d-ef
)	/(	abc*+d-ef*

Infix Expression: **A+ (B*C-(D/E^F)*G)*H**, where **^** is an exponential operator.

Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack , that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack , that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

Resultant Postfix Expression: ABC*DEF^/G*-H*+

Advantage of Postfix Expression over Infix Expression

An infix expression is difficult for the machine to know and keep track of precedence of operators. On the other hand, a postfix expression itself determines the precedence of operators (as the placement of operators in a postfix expression depends upon its precedence).Therefore, for the machine it is easier to carry out a postfix expression than an infix expression.

```c
#include<stdio.h>
#include<ctype.h>

char stack[100];
int top = -1;

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

char pop()
{
    if(top == -1)
        return -1;
    else
        return stack[top--];
}

int priority(char x)
{
    if(x == '(')
        return 0;
    if(x == '+' || x == '-')
        return 1;
    if(x == '*' || x == '/')
        return 2;
    return 0;
}

int main()
{
    char exp[100];
    char *e, x;
    printf("Enter the expression : ");
    scanf("%s",exp);
    printf("\n");
    e = exp;
```

```
    while(*e != '\0')
    {
        if(isalnum(*e))
            printf("%c ",*e);
        else if(*e == '(')
            push(*e);
        else if(*e == ')')
        {
            while((x = pop()) != '(')
                printf("%c ", x);
        }
        else
        {
            while(priority(stack[top]) >= priority(*e))
                printf("%c ",pop());
            push(*e);
        }
        e++;
    }

    while(top != -1)
    {
        printf("%c ",pop());
    }return 0;
}
```