

Evaluation of expression using stacks

What is an Expression?

In any programming language, if we want to perform any calculation or to frame a condition etc., we use a set of symbols to perform the task. These set of symbols makes an expression.

An expression is a collection of operators and operands that represents a specific value.

In above definition, **operator** is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.,

Operands are the values on which the operators can perform the task. Here operand can be a direct value or variable or address of memory location.

Expression Types

Based on the operator position, expressions are divided into THREE types. They are as follows...

1. **Infix Expression**
2. **Postfix Expression**
3. **Prefix Expression**

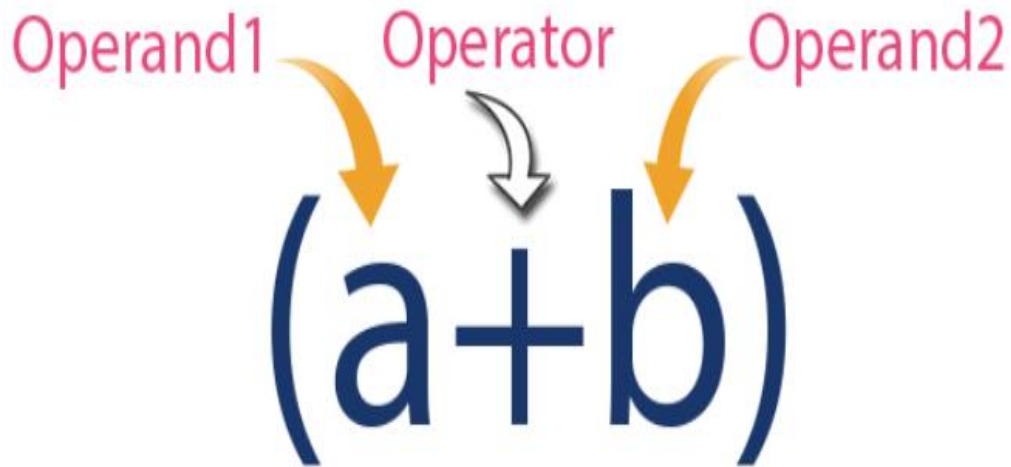
Infix Expression

In infix expression, operator is used in between the operands.

The general structure of an Infix expression is as follows...

Operand1 Operator Operand2

Example



Parsing Infix expressions

In order to parse any expression, we need to take care of two things, i.e., **Operator precedence** and **Associativity**. Operator precedence means the precedence of any operator over another operator. For example:

$A + B * C \rightarrow A + (B * C)$

As the multiplication operator has a higher precedence over the addition operator so $B * C$ expression will be evaluated first. The result of the multiplication of $B * C$ is added to the A.

Precedence order

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

Associativity means when the operators with the same precedence exist in the expression. For example, in the expression, i.e., **A + B - C**, '+' and '-' operators are having the same precedence, so they are evaluated with the help of associativity. Since both '+' and '-' are left-associative, they would be evaluated as **(A + B) - C**.

Associativity order

Operators	Associativity
\wedge	Right to Left
$*, /$	Left to Right
$+, -$	Left to Right

Let's understand the associativity through an example.

$$1 + 2 * 3 + 30 / 5$$

Since in the above expression, * and / have the same precedence, so we will apply the associativity rule. As we can observe in the above table that * and / operators have the left to right associativity, so we will scan from the leftmost operator. The operator that comes first will be evaluated first. The operator * appears before the / operator, and multiplication would be done first.

$$1 + (2 * 3) + (30 / 5)$$

$$1 + 6 + 6 = 13$$

Postfix Expression

In postfix expression, operator is used after operands. We can say that "**Operator follows the Operands**".

The general structure of Postfix expression is as follows..

Operand1 Operand2 Operator

Example



Prefix Expression

In prefix expression, operator is used before operands. We can say that **"Operands follows the Operator"**.

The general structure of Prefix expression is as follows...

Operator Operand1 Operand2

Example



Every expression can be represented using all the above three different types of expressions. And we can convert an expression from one form to another form like **Infix to Postfix**, **Infix to Prefix**, **Prefix to Postfix** and vice versa.

Conversion of Infix to Postfix

Algorithm for Infix to Postfix

Step 1: Consider the next element in the input.

Step 2: If it is operand, display it.

Step 3: If it is opening parenthesis, insert it on stack.

Step 4: If it is an operator, then

- If stack is empty, insert operator on stack.
- If the top of stack is opening parenthesis, insert the operator on stack
- If it has higher priority than the top of stack, insert the operator on stack.
- Else, delete the operator from the stack and display it, repeat Step 4.

Step 5: If it is a closing parenthesis, delete the operator from stack and display them until an opening parenthesis is encountered. Delete and discard the opening parenthesis.

Step 6: If there is more input, go to Step 1.

Step 7: If there is no more input, delete the remaining operators to output.

Example: Suppose we are converting $3*3/(4-1)+6*2$ expression into postfix form.

Following table shows the evaluation of Infix to Postfix:

Expression	Stack	Output
3	Empty	3
*	*	3
3	*	33
/	/	33*
(/(33*
4	/(33*4
-	/(-	33*4
1	/(-	33*41
)	-	33*41-
+	+	33*41-/
6	+	33*41-/6
*	+*	33*41-/62
2	+*	33*41-/62
	Empty	33*41-/62*+

So, the Postfix Expression is **33*41-/62*+**

Example: Program for Infix to Postfix Conversion

```
#include <stdio.h>
#include <ctype.h>
#define SIZE 50
char s[SIZE];
int top=-1;
push(char elem)
{
    s[++top]=elem;
    return 0;
}
char pop()
{
```

```

    return(s[top--]);
}
int pr(char elem)
{
    switch(elem)
    {
        case '#': return 0;
        case '(': return 1;
        case '+':
        case '-': return 2;
        case '*':
        case '/': return 3;
    }
    return 0;
}
void main()
{
    char infx[50], pofx[50], ch, elem;
    int i=0, k=0;
    printf("\n\nEnter Infix Expression: ");
    scanf("%s",infx);
    push('#');
    while( (ch=infx[i++]) != '\0')
    {
        if( ch == '(')
            push(ch);
        else
            if(isalnum(ch))
                pofx[k++]=ch;
            else
                if( ch == ')')
                {
                    while( s[top] != '(')

```

```

        pofx[k++]=pop();
        elem=pop();
    }
    else
    {
        while( pr(s[top]) >= pr(ch) )
            pofx[k++]=pop();
        push(ch);
    }
}
while( s[top] != '#' )
    pofx[k++]=pop();
pofx[k]='\0';
printf("\n\n Given Infix Expression: %s ",infx);
printf("\n Postfix Expression: %s",pofx);
}

```

Output :

```

DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, P
Enter Infix Expression: a*b/(c-d)-e*f

Given Infix Expression: a*b/(c-d)-e*f
Postfix Expression: ab*cd-/ef*_

```

```

DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0,
Enter Infix Expression: 3*3/(4-1)-8*2

Given Infix Expression: 3*3/(4-1)-8*2
Postfix Expression: 33*41-/82*_

```


Prefix Expression

In prefix expression, operator is used before operands. We can say that "**Operands follows the Operator**".

Prefix notation is a notation for writing arithmetic expressions in which the operands appear after their operators.

Algorithm:

Reverse the given expression and Iterate through it, one character at a time

1. If the character is an operand, push it to the operand stack.
2. If the character is an operator,
 1. pop the operand from the stack, say it's s1.
 2. pop another operand from the stack, say it's s2.
 3. perform **(s1 operator s2)** and push it to stack.
3. Once the expression iteration is completed, The stack will have the final result. pop from the stack and return the result.

Example:

Postfix: +54

Output: 9

Explanation: Infix expression of the above prefix is: 5+ 4 which resolves to 9

Example #1: + - 20 * 3 4 1

+ - 20 * 3 4 **1**

The first character scanned is "**1**", which is an operand, so push it to the stack.



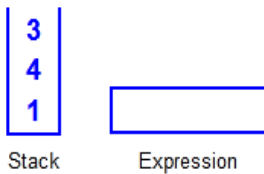
+ - 20 * 3 **4** 1

The next character scanned is "**4**", which is an operand, so push it to the stack.



+ - 20 * **3** 4 1

The next character scanned is "**3**", which is an operand, so push it to the stack.



+ - 20 * ***** 3 4 1

The next character scanned is "*****", which is an operator, so pop its two operands from the stack. Pop **4** from the stack for the left operand and then pop **3** from the stack to make the right operand.

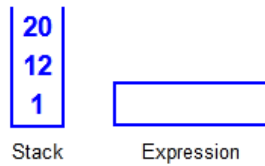


Next, push the result of **3 * 4** (**12**) to the stack.



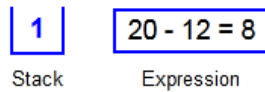
+ - 20 * 3 4 1

The next character scanned is "20", which is an operand, so push it to the stack.



+ - 20 * 3 4 1

The next character scanned is "-", which is an operator, so pop its two operands from the stack. Pop 12 from the stack for the left operand and then pop 20 from the stack to make the right operand.

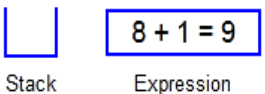


Next, push the result of 20 - 12 (8) to the stack.



+ - 20 * 3 4 1

The next character scanned is "+", which is an operator, so pop its two operands from the stack. Pop 1 from the stack for the left operand and then pop 8 from the stack to make the right operand.



Next, push the result of 8 + 1 (9) to the stack.



Since we are done scanning characters, the remaining element in the stack (9) becomes the result of the prefix evaluation.

prefix notation: + - 20 * 3 4 1

Result: 9

Postfix Expression

This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfixed** to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.

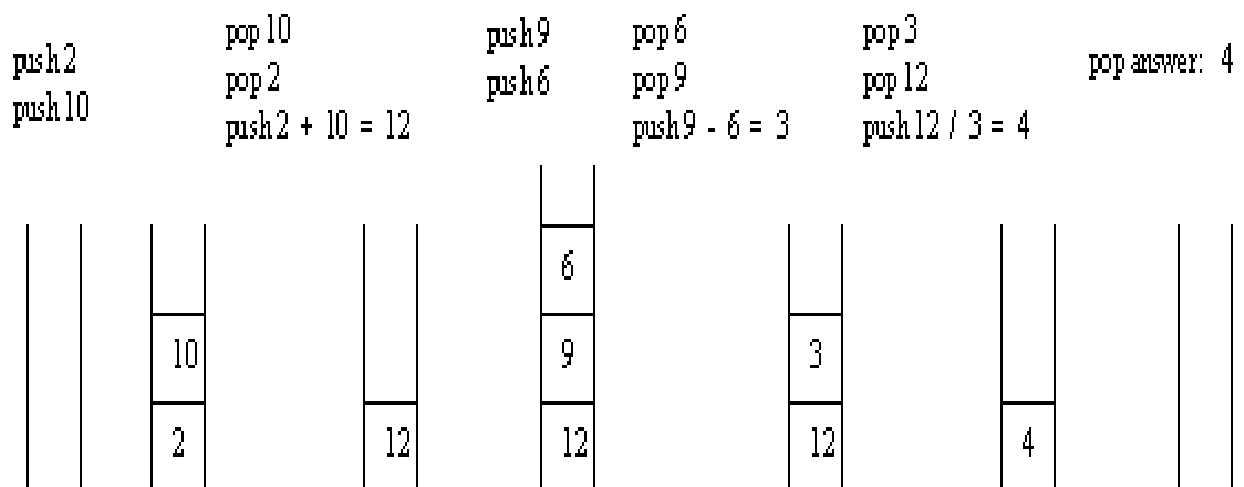
Algorithm for Evaluation of Postfix Expression

Create an empty stack and start scanning the postfix expression from left to right.

- If the element is an **operand**, **push it into the stack**.
- If the element is an **operator O**, **pop twice and get A and B respectively**. Calculate **BOA** and **push it back to the stack**.
- When the expression is ended, the value in the stack is the final answer.

Evaluation of a postfix expression using a stack is explained in below example:




2 10 + 9 6 - /



Infix Expression **(5 + 3) * (8 - 2)**

Postfix Expression **5 3 + 8 2 - ***

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations		Evaluated Part of Expression
Initially	Stack is Empty		Nothing
5	push(5)		Nothing
3	push(3)		Nothing

+

```
value1 = pop()
value2 = pop()
result = value2 + value1
push(result)
```



```
value1 = pop(); // 3
value2 = pop(); // 5
result = 5 + 3; // 8
Push( 8 )
```

(5 + 3)

8

push(8)



(5 + 3)

2

push(2)



(5 + 3)

-

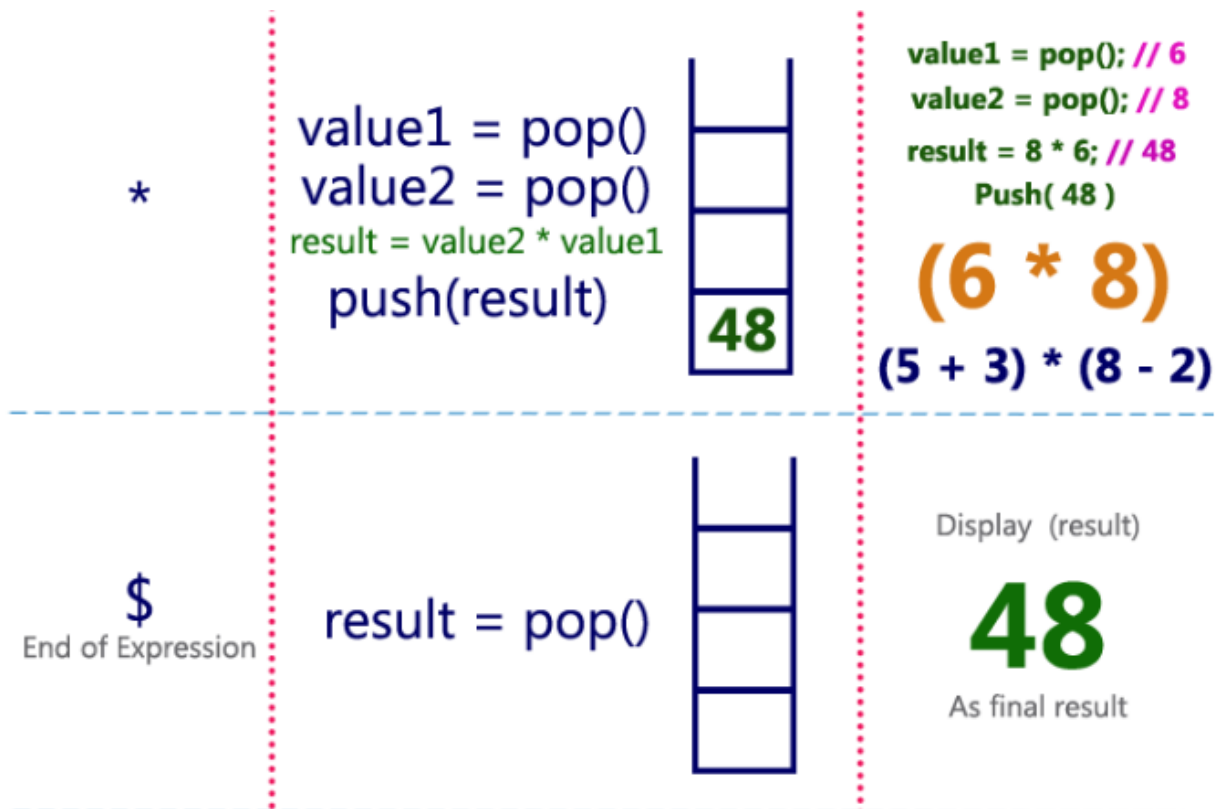
```
value1 = pop()
value2 = pop()
result = value2 - value1
push(result)
```



```
value1 = pop(); // 2
value2 = pop(); // 8
result = 8 - 2; // 6
Push( 6 )
```

(8 - 2)

(5 + 3) , (8 - 2)



Infix Expression **(5 + 3) * (8 - 2) = 48**

Postfix Expression **5 3 + 8 2 - * value is 48**

Example :

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

```
#define SIZE 10
```

```
int TOP=SIZE;
```

```
int stack[SIZE];
```

```
void push(int);
```

```
int pop();
```

```
void main()
{
    char postfix[100],c;
    int i=0,r1,r2,r,x;
    printf("Enter the Expression:");
    scanf("%s",postfix);
    while((c=postfix[i])!='\0')
    {
        switch(c)
        {
            case '+':
                r1=pop();
                r2=pop();
                r=r2+r1;
                push(r);
                break;

            case '-':
                r1=pop();
                r2=pop();
                r=r2-r1;
                push(r);
                break;

            case '*':
                r1=pop();
                r2=pop();
                r=r2*r1;
```



```

        push(r);
        break;

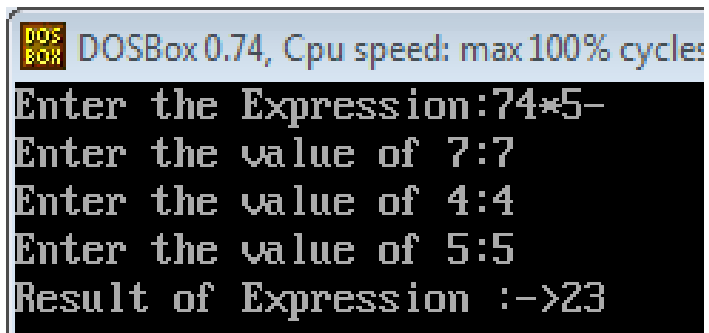
    case '/':
        r1=pop();
        r2=pop();
        r=r2/r1;
        push(r);
        break;

    default:printf("Enter the value of %c:",c);
        scanf("%d",&x);
        push(x);
    }
    i++;
}
r=pop();
printf("Result of Expression :->%d",r);
}
void push(int x)
{
    if(TOP==0)
        printf("Stack is full....");
    else
    {
        TOP=TOP-1;
        stack[TOP]=x;
    }
}

```

```
}  
int pop()  
{  
    int x;  
    if(TOP==SIZE)  
        printf("Stack is empty....");  
    else  
    {  
        x=stack[TOP];  
        TOP=TOP+1;  
    }  
    return(x);  
}
```

Output :



DOSBox 0.74, Cpu speed: max 100% cycles

```
Enter the Expression:74*5-  
Enter the value of 7:7  
Enter the value of 4:4  
Enter the value of 5:5  
Result of Expression :->23
```