

Convert Infix expression to Postfix expression

Write a program to convert an Infix expression to Postfix form.

**Infix expression:** The expression of the form “a operator b” (a + b) i.e., when an operator is in-between every pair of operands.  
**Postfix expression:** The expression of the form “a b operator” (ab+) i.e., When every pair of operands is followed by an operator.

Examples:

**Input:** A + B \* C + D  
**Output:** ABC\*+D+

**Input:** ((A + B) – C \* (D / E)) + F  
**Output:** AB+CDE/\*-F+

Why postfix representation of the expression?

The compiler scans the expression either from left to right or from right to left.  
Consider the expression: **a + b \* c + d**

- The compiler first scans the expression to evaluate the expression b \* c, then again scans the expression to add a to it.
- The result is then added to d after another scan.

The repeated scanning makes it very inefficient. Infix expressions are easily readable and solvable by humans whereas the computer cannot differentiate the operators and parenthesis easily so, it is better to convert the expression to postfix(or prefix) form before evaluation.

The corresponding expression in postfix form is **abc\*+d+**. The postfix expressions can be evaluated easily using a stack.

How to convert an Infix expression to a Postfix expression?

To convert infix expression to postfix expression, use the [stack data structure](#). Scan the infix expression from left to right. Whenever we get an operand, add it to the postfix expression and if we get an operator or parenthesis add it to the stack by maintaining their precedence.

Below are the steps to implement the above idea:

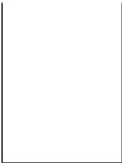
1. Scan the infix expression **from left to right**.
2. If the scanned character is an operand, put it in the postfix expression.
3. Otherwise, do the following
  - If the precedence and associativity of the scanned operator are greater than the precedence and associativity of the operator in the stack [or the stack is empty or the stack contains a ‘(’ ], then push it in the stack. [ ‘^’ operator is right associative and other operators like ‘+’, ‘-’, ‘\*’ and ‘/’ are left-associative].
    - Check especially for a condition when the operator at the top of the stack and the scanned operator both are ‘^’. In this condition, the precedence of the scanned operator is higher due to its right associativity. So it will be pushed into the operator stack.
    - In all the other cases when the top of the operator stack is the same as the scanned operator, then pop the operator from the stack because of left associativity due to which the scanned operator has less precedence.
  - 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 a ‘(’, push it to the stack.
5. If the scanned character is a ‘)’, pop the stack and output it until a ‘(’ is encountered, and discard both the parenthesis.
6. Repeat steps **2-5** until the infix expression is scanned.
7. Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.
8. Finally, print the postfix expression.

Illustration:

Follow the below illustration for a better understanding

Consider the infix expression **exp = “a+b\*c+d”**  
and the infix expression is scanned using the iterator **i**, which is initialized as **i = 0**.

**1st Step:** Here **i = 0** and **exp[i] = ‘a’** i.e., an operand. So add this in the postfix expression. Therefore, **postfix = “a”**.



postfix = "a"

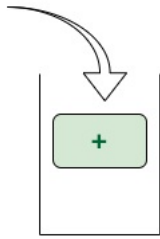
'a' is an operand. Add it in postfix expression

Add ‘a’ in the postfix

**2nd Step:** Here **i = 1** and **exp[i] = ‘+’** i.e., an operator. Push this into the stack. **postfix = “a”** and **stack = {+}**.

Add '+' into stack

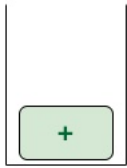
postfix = "a"



Stack is empty. Push '+' into stack

Push '+' in the stack

**3rd Step:** Now  $i = 2$  and  $\text{exp}[i] = 'b'$  i.e., an operand. So add this in the postfix expression. **postfix = "ab"** and **stack = {+, }**.



postfix = "ab"

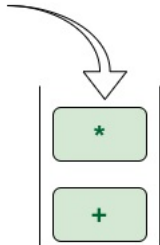
'b' is an operand. Add it in postfix expression

Add 'b' in the postfix

**4th Step:** Now  $i = 3$  and  $\text{exp}[i] = '*'$  i.e., an operator. Push this into the stack. **postfix = "ab"** and **stack = {+, \*}**.

Add '\*' into stack

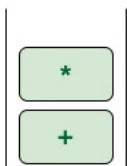
postfix = "ab"



\* has higher precedence. Push it into stack

Push '\*' in the stack

**5th Step:** Now  $i = 4$  and  $\text{exp}[i] = 'c'$  i.e., an operand. Add this in the postfix expression. **postfix = "abc"** and **stack = {+, \*}**.

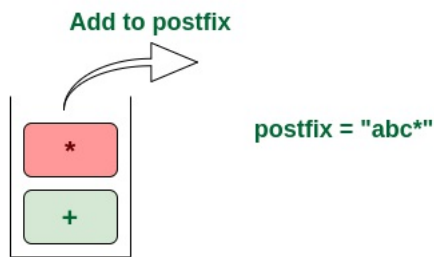


postfix = "abc"

'c' is an operand. Add it in postfix expression

Add 'c' in the postfix

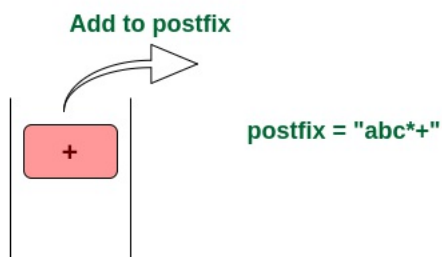
**6th Step:** Now  $i = 5$  and  $\text{exp}[i] = '+'$  i.e., an operator. The topmost element of the stack has higher precedence. So pop until the stack becomes empty or the top element has less precedence. '\*' is popped and added in postfix. So **postfix = "abc\*"** and **stack = {+, }**.



stack top has higher precedence than +

Pop '\*' and add in postfix

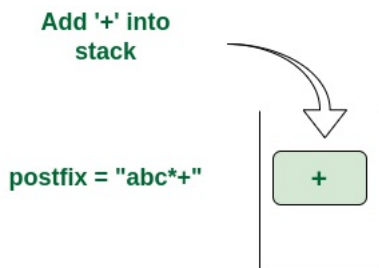
Now top element is '+' that also doesn't have less precedence. Pop it. **postfix** = "abc\*+".



'+' and stack top has same precedence

Pop '+' and add it in postfix

Now stack is empty. So push '+' in the stack. **stack** = {+}.



Stack is empty. Push '+' into stack

Push '+' in the stack

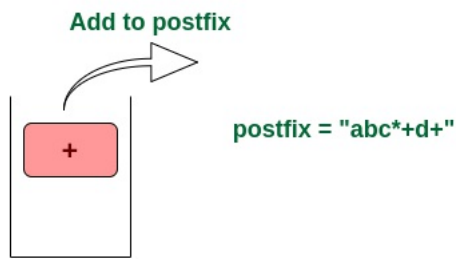
**7th Step:** Now  $i = 6$  and  $\text{exp}[i] = 'd'$  i.e., an operand. Add this in the postfix expression. **postfix** = "abc\*+d".



'd' is an operand. Add it in postfix expression

Add 'd' in the postfix

**Final Step:** Now no element is left. So empty the stack and add it in the postfix expression. **postfix** = "abc\*++".



**Nothing left. So pop all operators**

*Pop '+' and add it in postfix*

Below is the implementation of the above algorithm:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// 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(char s[]) {

    char result[1000];

    int resultIndex = 0;

    int len = strlen(s);

    char stack[1000];

    int stackIndex = -1;

    for (int i = 0; i < len; 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[resultIndex++] = c;
    }

    // If the scanned character is an '(', push it to the stack.
    else if (c == '(') {
        stack[++stackIndex] = c;
    }

    // If the scanned character is an ')', pop and add to the output string from the stack
    // until an '(' is encountered.
    else if (c == ')') {
        while (stackIndex >= 0 && stack[stackIndex] != '(') {
            result[resultIndex++] = stack[stackIndex--];
        }
        stackIndex--; // Pop '('
    }

    // If an operator is scanned
    else {
        while (stackIndex >= 0 && (prec(s[i]) < prec(stack[stackIndex]) ||
            prec(s[i]) == prec(stack[stackIndex]) &&
            associativity(s[i]) == 'L')) {
            result[resultIndex++] = stack[stackIndex--];
        }
        stack[++stackIndex] = c;
    }
}

// Pop all the remaining elements from the stack
while (stackIndex >= 0) {
    result[resultIndex++] = stack[stackIndex--];
}

result[resultIndex] = '\0';
printf("%s\n", result);
}

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

    // Function call
    infixToPostfix(exp);

    return 0;
}

```

## Output

abcd^e-fgh\*+^\*+i-

**Time Complexity:** O(N), where N is the size of the infix expression

**Auxiliary Space:**  $O(N)$ , where  $N$  is the size of the infix expression