**Experiment Number: 2b**

**Title**: Conversion of Infix Expression to Postfix Expression.

**Problem Statement**: Write a C program to implement the conversion from infix to postfix expression using stack.

**Algorithm:**

Let, X be an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

1. Push ‘(‘onto Stack, and add ‘)’ to the end of X.

2. Scan X from left to right and repeat Step 3 to 6 for each element of

X until the Stack is empty.

3. If an operand is encountered, add it to Y.

4. If a left parenthesis is encountered, push it onto Stack.

5. If an operator is encountered, then:

* + Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the same precedence as or higher precedence than operator.
  + Add operator to Stack.

[End of If]

6. If a right parenthesis is encountered, then:

-- Repeatedly pop from Stack and add to Y each operator (on the top

of Stack) until a left parenthesis is encountered.

-- Remove the left Parenthesis.

[End of If]  
 [End of If]

7. END.

**Code:**

#include<stdio.h>

#include<ctype.h>

#include<string.h>

#include<stdlib.h>

#include<stdbool.h>

#define MAX 100

char stack[MAX];

int top=-1,j=0;

void push(char item);

char pop(void);

bool isOperator(char token);

int precedance(char token);

void infixtopostfix(char[],char[]);

int main()

{

char infix[MAX],postfix[MAX];

int i;

printf("\nEnter the infix expression:");

gets(infix);

push('(');

infixtopostfix(infix,postfix);

puts(postfix);

return 0;

}

void push(char item)

{

if(top>=MAX-1)

printf("\nStack is full!! Overflow");

else

{

top++;

stack[top]=item;

}

}

char pop(void)

{

char item;

if(top<0)

{

printf("\nStack is Empty!! Underflow");

exit(1);

}

else

{

item=stack[top];

top--;

return(item);

}

}

bool isOperator(char token)

{

if(token=='+'||token=='-'||token=='\*'||token=='/'||token=='^')

return true;

return false;

}

int precedance(char token)

{

if(token=='^')

return 3;

else if(token=='\*'||token=='/')

return 2;

else if(token=='+'||token=='-')

return 1;

else

return 0;

}

void infixtopostfix(char infix[],char postfix[])

{

int i=0;

char token, stacktop;

strcat(infix,")");

while(infix[i]!='\0')

{

token=infix[i];

if(isdigit(token)||isalpha(token))

{

postfix[j]=token;

j++;

}

else if(isOperator(token))

{

stacktop=pop();

while((isOperator(stacktop))&&(precedance(stacktop)>=precedance(token)))

{

postfix[j]=stacktop;

stacktop=pop();

j++;

}

push(stacktop);

push(token);

}

else if(token=='(')

push(token);

else if(token==')')

{

stacktop=pop();

while(stacktop!='(')

{

postfix[j]=stacktop;

stacktop=pop();

j++;

}

}

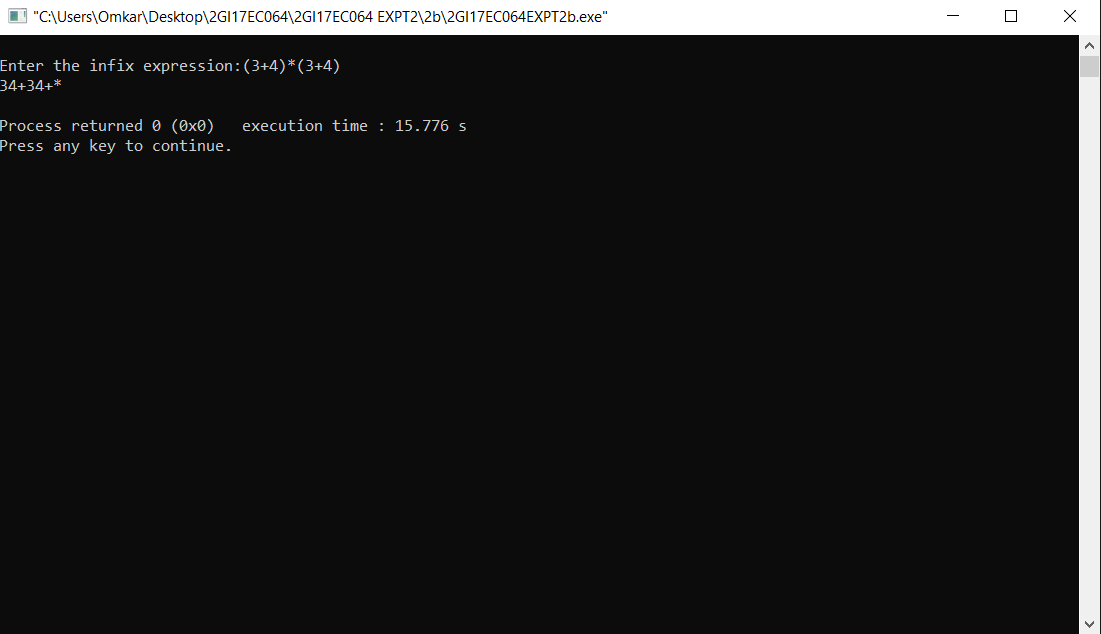
i++;

}

postfix[j]='\0';

}

**Output:**



**Analysis/Limitations:**

There are following limitations of above implementation.

1) It supports only 4 binary operators ‘+’, ‘\*’, ‘-‘and ‘/’. It can be extended for more operators by adding more switch cases.

2) The allowed operands are only single digit operands. The program can be extended for multiple digits by adding a separator like space between all elements (operators and operands) of given expression.