

**Experiment No 2:- Conversion of Infix to postfix expression using stack ADT**

**Aim:** To convert infix expression to postfix expression using stack ADT

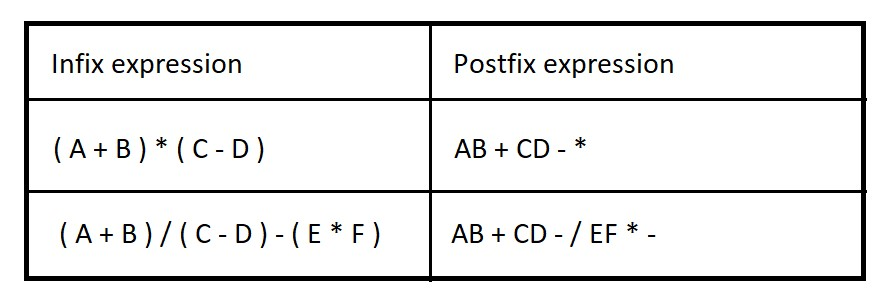
**Objective:**

1. Understand the use of stack
2. Understand how to import an ADT in an application program
3. Understand the instantiation of stack ADT in an application program
4. Understand how the member function of an ADT are accessed in an application program

**Theory:** To convert infix expression to postfix expression, we will use the stack data structure. By scanning the infix expression from left to right, when we will get any operand, simply add them to the postfix form, and for the operator and parenthesis, add them in the stack maintaining the precedence of them.

Infix:- An Infix expression is a mathematical expression in which operators are placed between the operands.

Postfix :- A postfix expression is a collection of operators and operands in which the operator is placed after the operands. That means, in a postfix expression the operator follows the operands.



**Algorithm:**

* Step 1: Scan the Infix Expression from left to right.
* Step 2: If the scanned character is an operand, append it with final Infix to Postfix string.
* Step 3: Else,
* Step 3.1: If the precedence order of the scanned operator is greater than the precedence order of the operator in the stack (or the stack is empty or the stack contains a ‘(‘ or ‘[‘ or ‘{‘), push it on stack.
* Step 3.2: 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.)
* Step 4: If the scanned character is an ‘(‘or ‘[‘ or ‘{‘, push it to the stack.
* Step 5: If the scanned character is an ‘)’or ‘]’ or ‘}’, pop the stack and and output it until a ‘(‘or ‘[ or ‘{‘respectively is encountered, and discard both the parenthesis.
* Step 6: Repeat steps 2-6 until infix expression is scanned.
* Step 7: Print the output
* Step 8: Pop and output from the stack until it is not empty.

**Code:**

#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);

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;

}

**Output:**



**Conclusion:**

* Infix expressions are readable and solvable by humans because of easily distinguishable order of operators, but compiler doesn't have integrated order of operators.
* Hence to solve the Infix Expression compiler will scan the expression multiple times to solve the sub-expressions in expressions orderly which is very in-efficient.
* To avoid this traversing, Infix expressions are converted to Postfix expression before evaluation.