**Godavari College Of Engineering, Jalgaon.**

**Subject Name:** Data Structure. **Teacher Name:** Prof. S.S.Shete

**Practical No. : 2** **Date:**

**Class:** S.E **Roll No:**

**Title:** Write a program to evaluate a given postfix expression using stacks.

**Aim:** To evaluate a given postfix expression using stacks.

**Theory:**

The way to write arithmetic expression is known as a notation. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are −

* Infix Notation
* Prefix (Polish) Notation
* Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression. We shall learn the same here in this chapter.

**Infix Notation**

We write expression in infix notation, e.g. a - b + c, where operators are used in-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

**Prefix Notation**

In this notation, operator is prefixed to operands, i.e. operator is written ahead of operands. For example, +ab. This is equivalent to its infix notation a + b. Prefix notation is also known as Polish Notation.

**Postfix Notation**

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.

The following table briefly tries to show the difference in all three notations −

|  |  |  |  |
| --- | --- | --- | --- |
| Sr.No. | Infix Notation | Prefix Notation | Postfix Notation |
| 1 | a + b | + a b | a b + |
| 2 | (a + b) ∗ c | ∗ + a b c | a b + c ∗ |
| 3 | a ∗ (b + c) | ∗ a + b c | a b c + ∗ |
| 4 | a / b + c / d | + / a b / c d | a b / c d / + |
| 5 | (a + b) ∗ (c + d) | ∗ + a b + c d | a b + c d + ∗ |
| 6 | ((a + b) ∗ c) - d | - ∗ + a b c d | a b + c ∗ d - |

**Associativity**

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a + b − c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and − are left associative, so the expression will be evaluated as (a + b) − c.

Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) −

|  |  |  |  |
| --- | --- | --- | --- |
| Sr.No. | Operator | Precedence | Associativity |
| 1 | Exponentiation ^ | Highest | Right Associative |
| 2 | Multiplication ( ∗ ) & Division ( / ) | Second Highest | Left Associative |
| 3 | Addition ( + ) & Subtraction ( − ) | Lowest | Left Associative |

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example −

In a + b\*c, the expression part b\*c will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for a + b to be evaluated first, like (a + b)\*c.

## **Postfix Evaluation Algorithm**

We shall now look at the algorithm on how to evaluate postfix notation −

Step 1 − scan the expression from left to right

Step 2 − if it is an operand push it to stack

Step 3 − if it is an operator pull operand from stack and perform operation

Step 4 − store the output of step 3, back to stack

Step 5 − scan the expression until all operands are consumed

Step 6 − pop the stack and perform operation

**Program:**

#include <stdio.h>

#include <stdlib.h>

int top = 10;

struct node

{

char ch;

struct node \*next;

struct node \*prev;

} \*stack[11];

typedef struct node node;

void push(node \*str)

{

if (top <= 0)

printf("Stack is Full ");

else

{

stack[top] = str;

top--;

}

}

node \*pop()

{

node \*exp;

if (top >= 10)

printf("Stack is Empty ");

else

exp = stack[++top];

return exp;

}

void convert(char exp[])

{

node \*op1, \*op2;

node \*temp;

int i;

for (i=0;exp[i]!='\0';i++)

if (exp[i] >= 'a'&& exp[i] <= 'z'|| exp[i] >= 'A' && exp[i] <= 'Z')

{

temp = (node\*)malloc(sizeof(node));

temp->ch = exp[i];

temp->next = NULL;

temp->prev = NULL;

push(temp);

}

else if (exp[i] == '+' || exp[i] == '-' || exp[i] == '\*' || exp[i] == '/' ||

exp[i] == '^')

{

op1 = pop();

op2 = pop();

temp = (node\*)malloc(sizeof(node));

temp->ch = exp[i];

temp->next = op1;

temp->prev = op2;

push(temp);

}

}

void display(node \*temp)

{

if (temp != NULL)

{

display(temp->prev);

printf("%c", temp->ch);

display(temp->next);

}

}

int main()

{

char exp[50];

//clrscr();

printf("Enter the postfix expression :");

scanf("%s", exp);

convert(exp);

printf("\nThe Equivalant Infix expression is:- ");

display(pop());

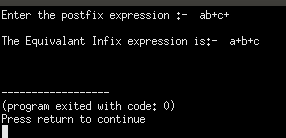
printf("\n\n");

//getch();

return 0;

}

**Output :**



**Conclusion:**