|  |  |
| --- | --- |
| Assignment No. : | 8 |
| Title: | Implement C++ program for expression conversion as infix to postfix and its evaluation using stack. |
| Subject: | Data Structures Laboratory |
| Class: | S.Y. (C.s.E.) |
| Roll No.: |  |
| Assessment (Marks): |  |
| Signature and Date of Assessment: |  |

Experiment No.:8

Experiment Title: Implement C++ program for expression conversion as infix to postfixand its evaluation using stack. Objectives:

1. To understand the use of stack.
2. To understand expression conversion as infix to postfix using stack.
3. To understand the use of stack.

Problem Statement: Implement C++ program for expression conversion as infix to postfix and its evaluation using stack based on given conditions:

Operands and operator, both must be single character.lnput

Postfix expression must be in a desired format. Only and '/ ' operators are expected.

Outcomes:

Understanding the use of stack.

Understanding of expression conversion as infix to postfix using stack.

Understanding the postfix expression evaluation using stack,

Theory: Stack is mainly used for expression conversion and expression evaluation purpose.

In any programming language, if we want to perform any calcu lation orto frame a conditionetc.. we use a set of symbols to perform the task. These set of symbols makes an expression.

An expression can be defined as follows:

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 arithmeti' 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.

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

* Infix Expression  postfix Expression
* Prefix Expression

Infix Expression

In infix expression. operator is used in between operands. The general structure of an Infix expression is as follows...

Operand I Operator Operand2

Example: a + b

Postfix Expression

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

The general structure of Postfix expression is as followsm

Operand I Operand2 Operator

Example: a b +

Prefix Expression

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

The general structure of Prefix expression is as follows...Operator Operandl Operand2 Example: +a b

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

Infix to postfix conversion:

l. Scan through an expression, getting one token at a time.

1. Fix a priority level for each operator. For example, from high to low:
2. - (unary negation)

l. + - (subtraction)

1. lhus, high priority corresponds to high number in the table.

If the token is an operand, do not stack it. Pass it to the output.

1. If token is an operator or parenthesis, do the following:

 Pop the stack until you find a symbol of lower priority number than the current one. An incoming left parenthesis will be considered to have higher priority than any othersymbol. A left parenthesis on the stack will not be removed unless an incoming right parenthesis is found.

The popped stack elements will be written to output.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Ex ression | , Stack | | | Out ut |
| 2 |  | |  | 2 |
|  |  | | | 2 |
| 3 |  | | | 23 |
|  |  | | |  |
|  |  |  | |  |
| 2 |  | | |  |
|  |  |  | | 23•2 |
|  |  | | |  |
|  |  | | |  |
|  |  | | | 23214 |
|  |  | | |  |
|  |  | | |  |
| 3 |  | | | 23•21.tS3 |
|  |  | |  | n•n.ts••• |

Figure I : Infix to postfix conversion

ii. Stack the current symbol.

lii. If a right parenthesis is the current symbol, pop the stack down to (and including) the first left parenthesis. Write all the symbols except the left parenthesis to the output (i.e.,write the operators to the output).

iv. After the last token is read, pop the remainder of the stack and write any symbol(except left parenthesis) to output.

Postfix Evaluation:

Postfix Expression: 4 5+7 2 - \*

In following figure evaluation of postfix is given.

|  |  |  |  |
| --- | --- | --- | --- |
| 4 5 . 7 2 -  4  9 7 2 -  2  7  9 | 4 5 . 7 2 -  9 7 2 - 9 5 •  5  9 45 | 4 5 + 7 2 - | 9 7 2 - |

figure 2: Postfix Evaluation

Algorithm:

Algorithm for Infix to Postfix conversion:

Examine the next element in the input.

* 1. If it is operand, output it.
  2. If it is opening parenthesis, push it on stack.
  3. If it is an operator. then
     + 1. If stack is empty, push operator on stack.
       2. If the top of stack is opening parenthesis, push operator on stack
       3. If it has higher priority than the top of stack. push operator on stack.
       4. Else pop the operator from the stack and output it, repeat step 4

1. If it is a closing parenthesis. pop operators from stack and output them until an opening parenthesis is encountered. pop and discard the opening parenthesis.
2. If there is more input go to step I
3. If there is no more input. pop the remaining operators to output.

Algorithm for evaluation of postfix expression

l. Initialize an empty stack.

2. Repeat the following until the end of the expression is encountered:

* + - * + 2.1 Get next token (constant, variable, arithmetic operator) in the postfix expressron.
        + 2.2 If token is an operand. push it onto the stack.
        + If it is an operator, then
      1. Pop top two values from the stack.
      2. If stack does not contain two items, error due to a malformed postfix. Evaluation terminated.
      3. Apply the operator to these two values.
      4. Push the resulting value back onto the stack.
    1. When the end of expression encountered, its value is on top of the stack (and, in fact, must be the only value in the stack).

For conversion of expression from infix to postfix: Input: (A + B)

Output: A B

For evaluation of postfix expression:

Input: A B \*

Enter A: 4

Enter B: 5

Output: 20

Conclusion: Thus, we implemented expression conversion as infix to postfix and its evaluation using stack.

Questions: 3. What is an expression?

* + 1. Explain different types of expression.
    2. Explain the role of stack for Infix to postfix conversion.
    3. Explain Postfix evaluation.

Code:-

/\* This program converts infix expression to postfix expression. This  program assume that there are Five operators:

(\*, /, +, -,^) in infix expression and operands can be of single-digit only. This program will not work for fractional numbers. Further this program does not check whether infix expression is  valid or not in terms of number of operators and operands.\*/

#include<stdlib.h>      /\* for exit() \*/

#include<ctype.h>     /\* for isdigit(char ) \*/

#include<string.h>

#include<iostream>

#define SIZE 100

using namespace std;

char stack[SIZE];

int top = -1;

void push(char item)

{

    if(top == SIZE-1)

        cout<<"\nStack Overflow.";

    else

    {

        top = top+1;

        stack[top] = item;

    }

}

 char pop()

{

    char item ;

    if(top ==-1)

    {

        cout<<"stack under flow: invalid infix expression";

        getchar(); /\* underflow may occur for invalid expression where ( and ) are not matched \*/

        exit(1);

    }

    else

    {

        item = stack[top];

        top = top-1;

        return item;

    }

}

/\* define function that is used to determine whether any symbol is operator or not (that is symbol is operand) this function returns 1 if symbol is operator else return 0 \*/

int is\_operator(char symbol)

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

      return 1;

    else

        return 0;

}

/\* define a function that is used to assign precedence to operator. Here ^ denotes exponent operator. In this function we assume that higher integer value means higher precedence \*/

int precedence(char symbol)

{   if(symbol == '^')/\* exponent operator, highest precedence\*/

        return 3;

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

        return 2;

    else if(symbol == '+' || symbol == '-')          /\* lowest precedence \*/

            return 1;

    else

        return 0;

}

void InfixToPostfix(char infix\_exp[], char postfix\_exp[])

{

    int i, j;

    char item;

    char x;

    push('(');                               /\* push '(' onto stack \*/

    strcat(infix\_exp,")");                  /\* add ')' to infix expression \*/

    i=0;

    j=0;

    item=infix\_exp[i];         /\* initialize before loop\*/

    while(item != '\0')        /\* run loop till end of infix expression \*/

    {

        if(item == '(')

            push(item);

        else if( isdigit(item) || isalpha(item))

        {

            postfix\_exp[j] = item;              /\* add operand symbol to postfix expr \*/

            j++;

        }

        else if(is\_operator(item) == 1)        /\* means symbol is operator \*/

        {

            x=pop();

            while(is\_operator(x) == 1 && precedence(x)>= precedence(item))

            {

                postfix\_exp[j] = x;                  /\* so pop all higher precendence operator and \*/

                j++;

                x = pop();                       /\* add them to postfix expresion \*/

            }

            push(x);

            /\* because just above while loop will terminate we have poppped one extra item

            for which condition fails and loop terminates, so that one\*/

            push(item);                 /\* push current oprerator symbol onto stack \*/

        }

        else if(item == ')')         /\* if current symbol is ')' then \*/

        {

            x = pop();                   /\* pop and keep popping until \*/

            while(x != '(')                /\* '(' encounterd \*/

            {

                postfix\_exp[j] = x;

                j++;

                x = pop();

            }

        }

        else

        { /\* if current symbol is neither operand not '(' nor ')' and nor

            operator \*/

            cout<<"\nInvalid infix Expression.\n";        /\* the it is illegeal  symbol \*/

            getchar();

            exit(1);

        }

        i++;

        item = infix\_exp[i]; /\* go to next symbol of infix expression \*/

    } /\* while loop ends here \*/

    if(top>0)

    {

        cout<<"\nInvalid infix Expression.\n";        /\* the it is illegeal  symbol \*/

        getchar();

        exit(1);

    }

    postfix\_exp[j] = '\0'; /\* add sentinel else puts() fucntion \*/

    /\* will print entire postfix[] array upto SIZE \*/

}

int main()

{

    char infix[SIZE], postfix[SIZE];

    cout<<"\n\t ASSUMPTION: The infix expression contains single letter variables and single digit constants only.\n";

    cout<<"\n\t\t Enter Infix expression : ";

    cin>>infix; //(a+b)\*(c-d)

    InfixToPostfix(infix,postfix); //call

    cout<<"\n\t\t Postfix Expression: ";

    cout<<postfix;

    return 0;

}

Output:-

ASSUMPTION: The infix expression contains single letter variables and single digit constants only.

Enter Infix expression : a\*(b+c)

Postfix Expression: abc+\*