Submitted By:- Ravi Pandey (075BCT065)

INTRODUCTION

Stack

A stack is a basic data structure that can be logically thought of as a linear structure represented by a real physical stack or pile, a structure where insertion and deletion of items takes place at one end called top of the stack. The basic concept can be illustrated by thinking of your data set as a stack of plates where you can only take the top item off the stack in order to remove things from it. This structure is used all throughout programming.

The basic implementation of a stack is also called a LIFO (Last In First Out) to demonstrate the way it accesses data, since as we will see there are various variations of stack implementations.

There are basically three operations that can be performed on stacks. They are

1) inserting an item into a stack (push).

2) deleting an item from the stack (pop).

3) displaying the contents of the stack (peek or top).

Infix notation:

In infix expression, operators are written in-between their operands. This is the usual way we write expressions. An expression such as A * (B + C) / D is infix.

Infix notation needs extra information to make the order of evaluation of the operators clear: rules built into the language about operator precedence and associativity, and brackets ( ) to allow users to override these rules.

Postfix notation (also known as "Reverse Polish notation"):

Operators are written after their operands. The infix expression given above is equivalent to A B C + * D /

The order of evaluation of operators is always left-to-right, and brackets cannot be used to change this order. Because the "+" is to the left of the "*" in the example

above, the addition must be performed before the multiplication. Operators act on values immediately to the left of them.

Prefix notation (also known as "Polish notation"):

Operators are written before their operands. The expressions given above are equivalent to / * A + B C D

As for Postfix, operators are evaluated left-to-right and brackets are superfluous. Operators act on the two nearest values on the right. I have again added (totally unnecessary) brackets to make this clear:

(/ (* A (+ B C)) D)

1. Perform push and pop operations in stack.

A)Push Operation

The algorithm to perform the push operation in stack is given below:

```
Push(Stack, MaxN, Data,Top)
   If Top=N-1
      Print “Stack Overflow!”  and return null
   Set Top=Top+1
   Stack[Top]=Data
   Return
```

B)Pop Operation

The algorithm to perform the pop operation in stack is given below:

```
Pop(Stack, Item, Top)
   If TOP= -1   //stack is empty
      Print “Underflow!”  and return null
   Set Item=Stack[Top]
   Set Top=Top-1
   Return
```

Program Code

```
#include <iostream>
using namespace std;
template <class t>
```

```
class Stack
{
  const int max;// =5;
  int top=-1;
  t* stack;

  public:
  Stack(int maxN):max(maxN)
  {
      top=-1;
      stack= new t[max];
  }
  ~Stack()
  {
      delete[] stack;
  }
  bool isFull()
  {
    if(top==max-1)
        return true;
    return false;
  }
  bool isEmpty()
  {
    if(top==-1)
        return true;
    return false;
  }
  t pop()
  {
      t item;
      if(!isEmpty())
      {
          item = stack[top];
          top--;
      }
      else
      cout<<"Stack is empty!\nYou must add an item to proceed.";
      return item;
  }
  void push(t data)
  {
      if(!isFull())
      {
          top++;
```

```
          stack[top]=data;
      }
      else
      cout<<"Stack is Full!\nYou must pop an item to proceed.";
  }
};
int main()
{
    int max;
    cout<<"Enter maximum stack capacity: ";
    cin>>max;
    Stack<int> s(max);
    int item;
    char choice,op;
    do
    {
    cout<<"Which operation do you want to perform?\n1)Push\n2)Pop";
    cin>>choice;
    switch(choice)
    {
        case '1':
        {
            cout<<"Enter number to be pushed:";
            cin>>item;
            s.push(item);
            break;
        }
        case '2':
        {

            cout<<s.pop()<<" Popped";
            break;
        }
        default:
        {
            cout<<"Invalid Operation!";
            op='y';
        }
    }
    cout<<"Do you want to continue?(Y or N)";
    cin>>op;
    } while (toupper(op)=='Y');
    return 0;
}
```

Output

```
Enter maximum stack capacity: 2
Which operation do you want to perform?
1)Push
2)Pop1
Enter number to be pushed:38
Do you want to continue?(Y or N)y
Which operation do you want to perform?
1)Push
2)Pop1
Enter number to be pushed:55
Do you want to continue?(Y or N)y
Which operation do you want to perform?
1)Push
2)Pop1
Enter number to be pushed:68
Stack is Full!
You must pop an item to proceed.Do you want to continue?(Y or N)
```

2. Convert :

Algorithm

a. Infix expression to postfix expression

1) START
2) Scan the infix expression form left to right
3) If the scanned character is an operand, then add it to postfix expression
4) The scanned character is an operator then:
    a) If the operator currently at the top of the stack has lower precedence than the operator read or the stack is empty then push the operator to the stack
    b) Else pop the stack and add the popped operator to postfix expression and push the read operator to the stack
5) If stack is not empty after reading all characters from infix expression then pop operators from stack to postfix expression until the stack is empty
6) Display the resulting postfix expression
7) END

b. Infix expression to prefix expression

1) START
2) Reverse the given infix expression
3) Interchange all '(' into ')' and vice-versa.
4) Convert the obtained expression into postfix expression
5) Reverse the obtained expression so that it becomes the required prefix expression.
6) END

Program Code

```cpp
#include<bits/stdc++.h>
using namespace std;

int precedence(char optr)
{
    if(optr=='^')
        return 3;
    else if(optr=='*' || optr=='/')
        return 2;
    else if (optr=='+' || optr=='-')
        return 1;
    else
        return -1;
}

string infixToPostfix(string Expr)
{
    stack<char> st;
    string postExpr;
    int length=Expr.length();

    st.push('N');

    for(int i=0;i<length;i++)
    {
        if((Expr[i]>='a' && Expr[i]<='z') || (Expr[i]>='0' && Expr[i]<='9'))
        postExpr+=Expr[i];

        else if(Expr[i]==')')
        {
            while(st.top()!='(' && st.top()!='N')
            {
                    postExpr=postExpr+st.top();
                    st.pop();

            }
            if(st.top()=='(')
            {
                st.pop();
            }

        }
        else if(Expr[i]=='(')
        st.push('(');
```

```
        else if (Expr[i]=='+' || Expr[i]=='-
' ||Expr[i]=='*' ||Expr[i]=='/' ||Expr[i]=='^')
        {
            if(precedence(Expr[i])>precedence(st.top()))
            st.push(Expr[i]);
            else
            {
            postExpr=postExpr+st.top();
            st.pop();
            st.push(Expr[i]);
            }
        }
    }
    while(st.top()!='N')
    {
        postExpr=postExpr+st.top();
            st.pop();
    }
    return postExpr;
}

string infixToPrefix(string Expr)
{
    reverse(Expr.begin(),Expr.end());
    for(int i=0;i<Expr.length();i++)
    {
        if (Expr[i] == '(')
            Expr[i] = ')';

        else if (Expr[i] == ')')
            Expr[i] = '(';

    }
    string revExpr = infixToPostfix(Expr);
    reverse(revExpr.begin(),revExpr.end());
    return revExpr;
}
Int main()
{
    string Expr;
    cout<<"Enter infix expression:";
    cin>>Expr;
    cout<<"\nPostfix: "<<infixToPostfix(Expr)<<"\nPrefix: "<<infixToPrefix(Expr);
    return 0;
}
```

Output

```
Enter infix expression:((a+b)*(c-d/e)^f)-g

Postfix: ab+cde/-f^*g-
Prefix: -*+ab^-c/defg
Process returned 0 (0x0)   execution time : 223.611 s
Press any key to continue.
```

3. Evaluate the postfix expression

Algorithm

1) START
2) Scan each character of the postfix expression from left to right
3) If operand is encountered, push it onto Stack
   [End If]
4) If operator is encountered
   i. A -> Top element and pop
   ii. B-> Next to Top element and pop
   iii. Evaluate B operator A
   iv. push result of B operator A onto Stack
5) Set result = stack[top]
6) END

Program Code

```
#include<bits/stdc++.h>
using namespace std;

int toDigit(char c)
{
    if(c>47 && c<58)
        return ( c - 48);
    else
        return -1;
}
bool isOperator(char symbol)
{
    if(symbol=='+' || symbol=='-' || symbol=='*' || symbol=='/' || symbol=='^')
    return true;
    return false;
}
int getRes(float a, float b, char op)
{
    switch(op)
```

```
            {
                case '+':
                    return b+a;
                case '-':
                    return b-a;
                case '*':
                    return b*a;
                case '/':
                    return b/a;
                case '^':
                    return pow(b,a);
                default:
                {
                cout<<"error!";
                exit(0);
                }
            }
}
int main()
{
    string  postfix;
    char symbol;
    int operand = 0;
    stack<int> opStack;
    opStack.push('N');
    int op1, op2;

    cout<<"\nEnter the postfix expression to evaluate [e.g. 10,20* ]  ";
    cin>>postfix;

     for(int i=0 , p = -1;i<postfix.length();i++)
     {
        symbol = postfix[i];
         if(toDigit(symbol) != -1)
         {
             if(i == postfix.length() -1)
             {
                 cout<<"\nInvalid expression!!";
                 exit(0);
             }
            p++;
         }
         else if(symbol == ',' || isOperator(symbol) && !isOperator(postfix[i-
1]))
         {
```

```
            for( ;p>=0;p--)
                operand += pow(10,p)*toDigit(postfix[i-p-1]);

            opStack.push(operand);
            operand = 0;
        }
        else if (isOperator(postfix[i-1])) {}
        else
        {
            cout<<"\nInvalid symbol in expression!!";
            exit(0);
        }

        if(isOperator(symbol))
        {
            op2 = opStack.top();
            opStack.pop();
            op1 = opStack.top();
            opStack.pop();
            opStack.push(getRes(op2,op1,symbol));
        }
    }
    int result = opStack.top();
    opStack.pop();

    if(opStack.top()=='N')
        cout<<postfix<<" = "<<result;
    else
        cout<<"\nInvalid expression";
    return 0;
}
```

Output

```
Enter the postfix expression to evaluate [e.g. 10,20* ]  12,3+34,22,2/-2^*5-
12,3+34,22,2/-2^*5- = 7930
Process returned 0 (0x0)   execution time : 504.617 s
Press any key to continue.
```

4. Check the paired parenthesis in mathematical expression.

Algorithm

1) START
2) Scan each character of the expression from left to right
3) If left parenthesis is encountered, push it onto Stack
4) If right parenthesis is encountered
    i. And stack is not empty i.e it contains one left parenthesis then pop it out.
    ii. And stack is empty then print “Unbalanced Parentheses” and exit
5) If stack is empty after scanning all the characters then
   Print “Balanced parentheses”
6) END

Program Code

```
#include <bits/stdc++.h>
using namespace std;

bool checkParan(string expr)
{
    stack <char> st;
    st.push('T');
    for(int i=0;i<expr.length();i++)
    {
        if(expr[i]=='(' || expr[i]=='{' || expr[i]=='[')
        st.push(expr[i]);
        else if(expr[i]==')')
        {
            if(st.top()!='T' && st.top()!='{' && st.top()!='[')
            st.pop();
            else
            return false;
        }
        else if(expr[i]=='}')
        {
            if(st.top()!='T' && st.top()!='(' && st.top()!='[')
            st.pop();
            else
            return false;
        }
        else if(expr[i]==']')
        {
            if(st.top()!='T' && st.top()!='(' && st.top()!='{')
            st.pop();
            else
            return false;
```

```
        }

    }
    if(st.top()=='T')
    return true;
    else
    return false;
}
int main()
{
    string str;
    cout<<"Enter expression:";
    cin>>str;
    if(checkParan(str))
    {
        cout<<"All paranthesis are paired";
    }
    else
    {
        cout<<"Unpaired paranthesis.";
    }
    return 0;
}
```

Outputs

```
Enter expression:[a+{(b/c)-d}+e*f]
All paranthesis are paired
Process returned 0 (0x0)   execution time : 47.891 s
Press any key to continue.
```

```
Enter expression:[a+{(b/c)-d+e*f]
Unpaired paranthesis.
Process returned 0 (0x0)   execution time : 56.907 s
Press any key to continue.
```