Balance Parathesis

```
#include<iostream>
#include<stack>
#include<string>
using namespace std;

bool ArePair(char opening,char closing)
{
    if(opening == '(' && closing == ')') return true;
    else if(opening == '{' && closing == '}') return true;
    else if(opening == '[' && closing == ']') return true;
    return false;
}
bool AreParanthesesBalanced(string exp)
{
    stack<char>  S;
    for(int i =0;i<exp.length();i++)
    {
        if(exp[i] == '(' || exp[i] == '{' || exp[i] == '[')
            S.push(exp[i]);
        else if(exp[i] == ')' || exp[i] == '}' || exp[i] == ']')
        {
            if(S.empty() || !ArePair(S.top(),exp[i]))
                return false;
            else
                S.pop();
        }
    }
    return S.empty() ? true:false;
}

int main()
{
    string expression;
    cout<<"Enter an expression:  ";
    cin>>expression;
    if(AreParanthesesBalanced(expression))
        cout<<"Balanced\n";
    else
        cout<<"Not Balanced\n";
}
```

```
Enter an expression:  [{}(){}]
Balanced

Process returned 0 (0x0)   execution time : 9.848 s
Press any key to continue.
```

```
Enter an expression:  [{()()]
Not Balanced

Process returned 0 (0x0)   execution time : 9.931 s
Press any key to continue.
```

Evolution of Postfix expression

```
#include<iostream>
#include<cmath>
#include<stack>
using namespace std;
float scanNum(char ch) {
   int value;
   value = ch;
   return float(value-'0');
}
int isOperator(char ch) {
   if(ch == '+'|| ch == '-'|| ch == '*'|| ch == '/' || ch == '^')
      return 1;
   return -1;
}
int isOperand(char ch) {
   if(ch >= '0' && ch <= '9')
      return 1;
   return -1;
}
float operation(int a, int b, char op) {
   if(op == '+')
      return b+a;
   else if(op == '-')
      return b-a;
   else if(op == '*')
      return b*a;
   else if(op == '/')
      return b/a;
   else if(op == '^')
      return pow(b,a);
   else
      return INT_MIN;
}
float postfixEval(string postfix) {
   int a, b;
   stack<float> stk;
   string::iterator it;
   for(it=postfix.begin(); it!=postfix.end(); it++) {
      if(isOperator(*it) != -1) {
```

```
            a = stk.top();
            stk.pop();
            b = stk.top();
            stk.pop();
            stk.push(operation(a, b, *it));
         }else if(isOperand(*it) > 0) {
            stk.push(scanNum(*it));
         }
      }
      return stk.top();
}
int main() {
      string post = "20 50 3 6 + * * 300 / 2 -";
      cout << "The result is : "<<postfixEval(post);
}
```

C:\Users\ASUS\Desktop\temp.exe

```
The result is : 28

Process returned 0 (0x0)   execution time : 9.105 s
Press any key to continue.
```

Evolution of Prefix expression

```
#include<iostream>
#include<cmath>
#include<stack>
using namespace std;
float scanNum(char ch) {
      int value;
      value = ch;
      return float(value-'0');
}
int isOperator(char ch) {
      if(ch == '+'|| ch == '-'|| ch == '*'|| ch == '/' || ch == '^')
         return 1;
      return -1;
}
int isOperand(char ch) {
      if(ch >= '0' && ch <= '9')
         return 1;
      return -1;
}
float operation(int a, int b, char op) {
      if(op == '+')
         return b+a;
```

```
   else if(op == '-')
      return b-a;
   else if(op == '*')
      return b*a;
   else if(op == '/')
      return b/a;
   else if(op == '^')
      return pow(b,a);
   else
      return INT_MIN;
}
float prefixEval(string prefix) {
   int a, b;
   stack<float> stk;
   string::iterator it;
   for(it=prefix.end(); it!=prefix.begin(); it++) {
      if(isOperator(*it) != -1) {
         a = stk.top();
         stk.pop();
         b = stk.top();
         stk.pop();
         stk.push(operation(a, b, *it));
      }else if(isOperand(*it) > 0) {
         stk.push(scanNum(*it));
      }
   }
   return stk.top();
}
int main() {
   string pre = "* + 2 - 2 1 / - 4 2 + - 5 3 1";
   cout << "The result is : "<<prefixEval(post);
}
```

C:\Users\ASUS\Desktop\temp.exe

```
The result is : 2

Process returned 0 (0x0)   execution time : 7.657 s
Press any key to continue.
```

Conversion of Infix to Postfix

```
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include<string.h>
using namespace std;
```

```
#define SIZE 100
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 <0)
    {
        cout<<"stack under flow: invalid infix expression";
        exit(1);
    }
    else
    {
        item = stack[top];
        top = top-1;
        return(item);
    }
}
int is_operator(char symbol)
{
    if(symbol == '^' || symbol == '*' || symbol == '/' || symbol == '+' || symbol
 =='-')
    {
        return 1;
    }
    else
    {
    return 0;
    }
}
int precedence(char symbol)
{
    if(symbol == '^')
    {
        return(3);
    }
    else if(symbol == '*' || symbol == '/')
    {
```

```
        return(2);
    }
    else if(symbol == '+' || symbol == '-')
    {
        return(1);
    }
    else
    {
        return(0);
    }
}
void InfixToPostfix(char infix_exp[], char postfix_exp[])
{
    int i, j;
    char item;
    char x;

    push('(');
    strcat(infix_exp,")");

    i=0;
    j=0;
    item=infix_exp[i];

    while(item != '\0')
    {
        if(item == '(')
        {
            push(item);
        }
        else if( isdigit(item) || isalpha(item))
        {
            postfix_exp[j] = item;
            j++;
        }
        else if(is_operator(item) == 1)
        {
            x=pop();
            while(is_operator(x) == 1 && precedence(x)>= precedence(item))
            {
                postfix_exp[j] = x;
                j++;
                x = pop();
            }
            push(x);

            push(item);
        }
        else if(item == ')')
        {
            x = pop();
```

```
            while(x != '(')
            {
                postfix_exp[j] = x;
                j++;
                x = pop();
            }
        }
        else
        {
            operator */
            cout<<"\nInvalid infix Expression.\n";
            exit(1);
        }
        i++;
        item = infix_exp[i];
    }
    if(top>0)
    {
        Cout<<"\nInvalid infix Expression.\n";
        exit(1);
    }
    if(top>0)
    {
        cout<<"\nInvalid infix Expression.\n";
        exit(1);
    }
    postfix_exp[j] = '\0';
}

int main()
{
    char infix[SIZE], postfix[SIZE];
    cout<<"ASSUMPTION: The infix expression contains single letter variables and
single digit constants only.\n";
    cout<<"\nEnter Infix expression : ";
    gets(infix);

    InfixToPostfix(infix,postfix);
    cout<<"Postfix Expression: ";
    puts(postfix);

    return 0;
}
```

C:\Users\ASUS\Desktop\temp.exe

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

Enter Infix expression : A*(B+C/D)
Postfix Expression: ABCD/+*

Process returned 0 (0x0)   execution time : 7.493 s
Press any key to continue.
```

Conversion of Infix to Prefix

```
#include <iostream.h>
#include <string.h>
#include <ctype.h>
using namespace std;
const int MAX = 50 ;
class infix
{
    private :
        char target[MAX], stack[MAX] ;
        char *s, *t ;
        int top, l ;
    public :
        infix( ) ;
        void setexpr ( char *str ) ;
        void push ( char c ) ;
        char pop( ) ;
        void convert( ) ;
        int priority ( char c ) ;
        void show( ) ;
} ;
infix :: infix( )
{
    top = -1 ;
    strcpy ( target, "" ) ;
    strcpy ( stack, "" ) ;
    l = 0 ;
}
void infix :: setexpr ( char *str )
{
    s = str ;
    strrev ( s ) ;
    l = strlen ( s ) ;
    * ( target + l ) = '\0' ;
    t = target + ( l - 1 ) ;
}
void infix :: push ( char c )
{
    if ( top == MAX - 1 )
        cout << "\nStack is full\n" ;
    else
    {
        top++ ;
        stack[top] = c ;
    }
}
char infix :: pop( )
{
    if ( top == -1 )
    {
```

```
        cout << "Stack is empty\n" ;
        return -1 ;
    }
    else
    {
        char item = stack[top] ;
        top-- ;
        return item ;
    }
}
void infix :: convert( )
{
    char opr ;

    while ( *s )
    {
        if ( *s == ' ' || *s == '\t' )
        {
            s++ ;
            continue ;
        }

        if ( isdigit ( *s ) || isalpha ( *s ) )
        {
            while ( isdigit ( *s ) || isalpha ( *s ) )
            {
                *t = *s ;
                s++ ;
                t-- ;
            }
        }

        if ( *s == ')' )
        {
            push ( *s ) ;
            s++ ;
        }

        if ( *s == '*' || *s == '+' || *s == '/' ||
                *s == '%' || *s == '-' || *s == '$' )
        {
            if ( top != -1 )
            {
                opr = pop( ) ;

                while ( priority ( opr ) > priority ( *s ) )
                {
                    *t = opr ;
                    t-- ;
                    opr = pop( ) ;
                }
                push ( opr ) ;
```

```
                push ( *s ) ;
            }
            else
                push ( *s ) ;
            s++ ;
        }

        if ( *s == '(' )
        {
            opr = pop( ) ;
            while ( ( opr ) != ')' )
            {
                *t = opr ;
                t-- ;
                opr =  pop ( ) ;
            }
            s++ ;
        }
    }

    while ( top != -1 )
    {
        opr = pop( ) ;
        *t = opr ;
        t-- ;
    }
    t++ ;
}
int infix :: priority ( char c )
{
    if ( c == '$' )
        return 3 ;
    if ( c == '*' || c == '/' || c == '%' )
        return 2 ;
    else
    {
        if ( c == '+' || c == '-' )
            return 1 ;
        else
            return 0 ;
    }
}
void infix :: show( )
{
    while ( *t )
    {
        cout << " " << *t ;
        t++ ;
    }
}
```

```
int main( )
{
    char expr[MAX] ;
    infix q ;

    cout << "\nEnter an expression in infix form: " ;
    cin.getline ( expr, MAX ) ;

    q.setexpr( expr ) ;
    q.convert( ) ;

    cout << "The Prefix expression is: " ;
    q.show( ) ;
}
```

C:\Users\ASUS\Desktop\temp.exe

```
Enter an expression in infix form: A*(B+C/D)
The Prefix expression is:  * A + B / C D
Process returned 0 (0x0)   execution time : 15.601 s
Press any key to continue.
```