```cpp
// stack_test.cpp : Defines the entry point for the console application.
//
#include <iostream>
#include <stdio.h>
#include "ArrayStack.h"
#include "LinkedStack.h"
#include "StackApplication.h"

using namespace std;

void TestArrayStack();
void TestLinkedStack();

int main()
{

    //TestArrayStack();

    //TestLinkedStack();

    //cout<<endl;


    cout<<"Test if the symbol sequence is balance"<<endl;
    cout<<"==============Begin==================="<<endl;

    char e1[16]={'{','[','(',')','(','{','[',']','}',')',']','}'};
    char e2[16]={'{','[','(','(',')',')','(',']','}',')',']','}'};

    cout<<"Is "<<e1<< " Balance? "<<endl;
    cout<<"The result is "<<isBalance(e1)<<endl<<endl;

    cout<<"Is "<<e2<< " Balance? "<<endl;
    cout<<"The result is "<<isBalance(e2)<<endl;

    cout<<"==============End==================="<<endl<<endl;




    cout<<"Test Expression Evaluation"<<endl;
    cout<<"==============Begin==================="<<endl;

    char e3[]="5+7*3-(6*2+1)/10";
    //char e3[]="1.25*3.6+8000.9/15.2*(123-456*789)";

    cout<<"Expression Evaluation"<<endl;
    cout<<endl<<e3<< " = "<<ExpressionEvaluation(e3)<<endl;

    cout<<"==============End==================="<<endl;


    cout<<endl;
    system("pause");

    return 0;
}

void TestArrayStack()
{
    cout<<"========================="<<endl;
    cout<<"Array based stack"<<endl;
    cout<<"========================="<<endl;
    ArrayStack<char> c_stack;

    cout<<"Stack Push"<<endl;
    c_stack.push('H');
    cout<<"Top of the stack is:  ";
    cout<<c_stack.top()<<endl;


    cout<<"Stack Push"<<endl;
```

```
    c_stack.push('e');

    cout<<"Top of the stack is:  ";
    cout<<c_stack.top()<<endl;

    cout<<"Is stack empty?  ";
    cout<<c_stack.isEmpty()<<endl;

    cout<<"Stack Pop  ";
    cout<<c_stack.pop()<<endl;

    cout<<"Is stack empty?  ";
    cout<<c_stack.isEmpty()<<endl;

    cout<<"Stack Pop  ";
    cout<<c_stack.pop()<<endl;

    cout<<"Is stack empty?  ";
    cout<<c_stack.isEmpty()<<endl;
};

void TestLinkedStack()
{
    cout<<"========================"<<endl;
    cout<<"Linked List based stack"<<endl;
    cout<<"========================"<<endl;
    LinkStack<int> i_stack;
    cout<<"Stack Push"<<endl;
    i_stack.push(1);

    cout<<"Top of the stack is:  ";
    cout<<i_stack.top()<<endl;

    cout<<"Stack Push"<<endl;
    i_stack.push(2);

    cout<<"Top of the stack is:  ";
    cout<<i_stack.top()<<endl;

    cout<<"Stack Pop  ";
    cout<<i_stack.pop()<<endl;

    cout<<"Top of the stack is:  ";
    cout<<i_stack.top()<<endl;

    cout<<"Is stack empty?  ";
    cout<<i_stack.isEmpty()<<endl;

    cout<<"Stack Pop  ";
    cout<<i_stack.pop()<<endl;

    cout<<"Is stack empty?  ";
    cout<<i_stack.isEmpty()<<endl;
};
```

```
#ifndef ARRAY_STACK
#define ARRAY_STACK

const int SIZE=1024;

template <class T>
class ArrayStack
{
private:
    T buf[SIZE];
    int index;

public:

    ArrayStack():index(-1){};

    void push(T a)
    {
        buf[++index]=a;
    };

    T pop()
    {
        return buf[index--];
    };

    T top()
    {
        return buf[index];
    };

    bool isEmpty(){return -1==index;}

    bool isFull(){return (SIZE-1)==index;}
};

#endif
```

```
#ifndef LINKED_STACK
#define LINKED_STACK

template<class T>
class LinkStack
{
private:
    struct Node
    {
        T data;
        Node* next;
    }*head,*p;

public:
    LinkStack()
    {
        head=NULL;
        p=NULL;
    }

    ~LinkStack()
    {
        p=head;

        while(p)
        {
            head=p->next;

            delete p;

            p=head;
        }
    }

    void push(T a)
    {
        p=new Node;
        p->data=a;
        p->next=head;

        head=p;
        p=NULL;
    }

    T pop()
    {
        T a=head->data;

        p=head;
        head=head->next;

        delete p;
        p=NULL;

        return a;
    }

    T top()
    {
        return head->data;
    }

    bool isEmpty(){return NULL==head;}

};


#endif
```

```
#ifndef STACK_APPLICATION
#define STACK_APPLICATION

int isBalance(char *exprssion);

double ExpressionEvaluation(char* expression);

#endif
```

```cpp
#include <iostream>
#include "ArrayStack.h"
#include "LinkedStack.h"
#include "StackApplication.h"

using namespace std;

bool isOpen(char c)
{
    return ('{'==c || '['==c || '('==c);
}

bool isClose(char c)
{
    return ('}'==c || ']'==c || ')'==c);
}

bool isPair(char lc, char rc)
{
    return (rc-lc==1) || (rc-lc==2);
}

// Test if the symbol sequence is balance
// Return -1 if it is balance
// Else, return the position where unbalance
int isBalance(char *expression)
{

    //Make an empty stack
    ArrayStack<char> c_stack;
    //LinkStack<char> c_stack;
    int index=0;
    char ltmp,rtmp;

    //Read characters until end of file.
    while (*expression)
    {
        //If the character is an opening symbol, push it onto the stack.
        if (isOpen(*expression))
        {
            c_stack.push(*expression);
        }
        else
        {
            if(isClose(*expression))
            {
                //If it is a closing symbol and the stack is empty, report an error.
                if(c_stack.isEmpty())
                    return index;
                else
                {
                    //Otherwise, pop the stack
                    ltmp=c_stack.pop();
                    rtmp=*expression;

                    //If the symbol popped is not the corresponding opening symbol, then report an error.
                    if (!isPair(ltmp,rtmp))
                    {
                        return index;
                    }//if (!isPair(ltmp,rtmp))

                }//if(c_stack.isEmpty())

            }//if(isClose(*expression))
        }
        expression++;
        index++;

    }

    //At end of file, if the stack is not empty, report an error
    if (!c_stack.isEmpty())
```

```
        return 0;

    return -1;

}

//Priority table value
int PriorityTable(char c)
{
    switch (c)
    {
    case '+':
    case '-':
        return 1;

    case '*':
    case '/':
        return 2;

    case '(':
        return 3;

    case ')':
        return -1;
    }
}

//change a string to number
double Str2Num(char* s)
{
    double num=0;
    bool decimal=0;
    double exp=0.1;

    while(*s)
    {
        if (*s=='.')
        {
            decimal=1;
            s++;
            continue;
        }

        if (!decimal)
        {
            num=num*10+(*s-'0');
        }
        else
        {
            num=num+(*s-'0')*exp;
            exp=exp*0.1;
        }

        s++;

    }

    return num;
}

//test if a char is a number
bool isNum(char c)
{
    return (c>='0' && c<='9' || c=='.');
}

// test if a char is an operator
bool isOpt(char c)
{
    return (c=='+' || c=='-' || c=='*' || c=='/' || c=='(' || c==')') ;
}
```

```
// apply the operato to the two numbers
double compute(double a1, double a2, char opt)
{
    switch (opt)
    {
    case '+':
        return a1+a2;
    case '-':
        return a1-a2;
    case '*':
        return a1*a2;
    case '/':
        return a1/a2;
    }

}

// Expression Evaluation
// Assume the input string is correct
// and then return the value
double ExpressionEvaluation(char* expression)
{

    char buf[16];
    double num1,num2;
    char opt;

    ArrayStack<double> num_stack;
    ArrayStack<char> opt_stack;

    while (*expression)
    {

        //if a number is seen, push it into the number stack
        if (isNum(*expression))
        {
            int i=0;
            do
            {
                buf[i++]=*(expression++);

            }
            while (isNum(*expression));

            buf[i]=0;

            num_stack.push(Str2Num(buf));

        }//if (isNum(*expression))

        // if a operator is seen
        if (isOpt(*expression))
        {
            // if a right parenthesis is seen, pop until the corresponding left parenthesis
            // during the pop, do the computation
            if (*expression==')')
            {
                while(opt_stack.top()!='(')
                {
                    opt=opt_stack.pop();
                    num1=num_stack.pop();
                    num2=num_stack.pop();

                    // apply the operato to the two numbers
                    num_stack.push(compute(num2,num1,opt));

                }//while(opt_stack.pop()!='(')

                opt_stack.pop();//pop '( '

            }
            else
```

```
            {
                //If the stack is empty, or if the operator at the top has lower priority, push the operator.
                if (opt_stack.isEmpty() || PriorityTable(*expression)>PriorityTable(opt_stack.top()))
                {
                    opt_stack.push(*expression);
                }
                else
                {
                    //Else, pop entries from the stack until the top has lower priority.
                    while (!opt_stack.isEmpty() && (PriorityTable(*expression)<=PriorityTable(opt_stack.top())) &↙
    & opt_stack.top()!='(')
                    {
                        opt=opt_stack.pop();
                        num1=num_stack.pop();
                        num2=num_stack.pop();
                        num_stack.push(compute(num2,num1,opt));
                    }
                    //And then push the operator
                    opt_stack.push(*expression);

                }

            }

            expression++;

        }//if (isOpt(*expression))



    }//while (*expression)


    // after the reading is finished, pop all the remain operators.
    while (!opt_stack.isEmpty())
    {
        opt=opt_stack.pop();
        num1=num_stack.pop();
        num2=num_stack.pop();
        num_stack.push(compute(num2,num1,opt));
    }

    return num_stack.pop();
}
```