项目说明文档

数据结构课程设计

——算数表达式求解

作 者 姓 名： 邓泉

学 号： 1953871

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

同济大学

Tongji Universit

目录

[1 分析](#_Toc24701351)

[1.1 项目要求](#_Toc24701352)

[1.2 功能分析](#_Toc24701353)

[2 设计](#_Toc24701354)

[2.1 数据结构设计](#_Toc495668157)

[2.2 类结构设计](#_Toc495668158)

[2.3 成员与操作设计](#_Toc495668159)

[2.4 主函数设计](#_Toc24701355)

[2.5 isValid()函数设计](#_Toc24701356)

[2.6 toPostOrder()函数设计](#_Toc24701357)

[3 实现](#_Toc24701359)

[3.1 主函数实现](#_Toc24701360)

[3.2 isValid()函数实现](#_Toc24701361)

[3.2.1 简单的判断](#_Toc24701362)

[3.2.2 对左右括号的判断](#_Toc24701363)

[3.2.3 单目表达式的判断](#_Toc24701364)

[3.2.4 双目表达式的判断](#_Toc24701365)

[3.3 toPostOrder()函数实现](#_Toc24701366)

[3.4 Caculator类成员函数的实现](#_Toc24701369)

[3.4.1函数描述](#_Toc24701370)

[3.4.2 函数代码](#_Toc24701371)

[4 测试](#_Toc24701372)

[4.1 功能测试](#_Toc24701373)

[4.1.1 测试1](#_Toc24701374)

[4.1.2 测试2](#_Toc24701375)

[4.1.3 测试3](#_Toc24701376)

[4.1.4 测试4](#_Toc24701377)

[4.1.4 测试5](#_Toc24701378)

[4.1.4 测试6](#_Toc24701379)

[4.2 非法表达式测试](#_Toc24701380)

[4.2.1 含有非法字符](#_Toc24701381)

[4.2.2 左右括号中内容为空](#_Toc24701382)

[4.2.3 左右括号不匹配](#_Toc24701383)

[4.2.4 等号位置错误](#_Toc24701384)

[4.2.5 单目运算符使用不合理](#_Toc24701385)

[4.2.6 双目运算符使用不合理](#_Toc24701386)

# 1 分析

## 背景分析

平时写的计算式一般是中缀表达式，而编译程序一般使用后缀表示求解表达式的值。

后缀表达式又称逆波兰表达式，明显的特点是：逆波兰表达式中没有括号，计算时将操作符之前的第一个数作为右操作数，第二个数作为左操作数，进行计算，得到的值继续放入逆波兰表达式中。

在日常生活中，我们常常使用中缀表达式进行计算，但后缀表达式更适合计算机。C++规定，一个表达式中相邻的两个操作符的计算次序为：优先级高的先计算；如果优先级相同，则自左向右计算；当使用括号时，从最内侧的括号开始计算。

由于中缀表达式有操作符的优先级问题，还有可加括号改变运算顺序的问题，所以对于编译程序来说，一般不使用中缀表示处理表达式。解决办法是用后缀表示（较常用）和前缀表示。因为后缀表示计算表达式的值只用一个栈，而前缀表示和中缀表示同时需要两个，所以编译程序一般使用后缀表达式求解表达式的值。

综上编写一个能够将中缀表达式转换为后缀表达式然后计算的程序具有广泛的应用价值。

## 功能分析

输入：输入在一行中给出以空格分隔不同对象的中缀表达式，可包含+, -, \*, /, -, ^以及左右括号，表达式不超过20个字符（不包括空格）。

输出：在一行中输出转换后的后缀表达式，要求不同对象（运算数，运算符号）之间以空格分隔，但是结尾不得有多余空格，最后完成计算。

支持包括加减，乘除取余，乘方和括号等操作符，其中优先级是等于<括号<加减<乘除取余<乘方。

能处理单目运算符：+或-。

首先,得对输入表达式进行判断,看它是否合理,如果不合理,要指出错误之处;如果是合理的,那么要计算这个表达式的结果并返回。

表达式是否错误根据下面几条进行判断：

1.出现了除了+-\* % / ^ ()=,数字以外的其他字符

2.()匹配不全 或者()里面为空

3.表达式没有结束之前出现了=,或者表达式最后不是=

4.+-不是开头,或者前方不是( 一目运算符

5.运算符前方或者后方无数字或者(),左边为),右边为(,避免(+) 二目运算符

# 2 设计

## **2.1 数据结构设计**

中缀表达式的计算、中缀表达式转为后缀表达式是经典的栈数据结构的应用。因此，我们使用栈对表达式进行保存和处理。栈具有先进后出（FILO）的特点，是一种使用广泛，特点鲜明的线性存储结构。

典型的栈数据结构应该具有Push()、getTop()和Pop()接口。Push()函数将元素压入栈顶，getTop()函数读取栈顶元素的值，而Pop()函数删除栈顶元素。

## **2.2 类结构设计**

## 2.2.1 LinkNode设计

LinkNode节点保存一个模板类型的元素data,并且有一个指针link指向下一个元素,用struct描述以便于访问。

## 2.2.2 LinkedStack设计

底层存储LinkNode类头节点以及指向栈顶元素的指针。为外界提供公有函数Push（），Pop（），getTop（），isEmpty（）。提供栈的基本功能。

## 2.2.3 Calculator设计

私有成员有操作数栈，私有成员函数有AddOperand()（进操作数栈）、get\_2\_Operands()（从栈中退出两个操作数）、DoOperator()（形成运算指令），公有成员函数Run()调用私有成员函数实现执行表达式计算。

## **2.3 成员与操作设计**

## 2.3.1 **链表结点类**LinkNode

template <class T>

struct LinkNode

{

public:

//构造函数

LinkNode(const T& value, LinkNode<T>\* next = NULL) : data(value), link(next) {}

LinkNode(LinkNode<T>\* next = NULL) : link(next) {}

T data;

LinkNode<T>\* link;

};

## 2.3.2 **链式栈类** LinkedStack

template <class T>

class LinkedStack

{

public:

//构造函数

LinkedStack()

{

top = NULL;

}

//析构函数

~LinkedStack()

{

makeEmpty();

}

//清空栈的内容

void makeEmpty()

{

LinkNode<T>\* del;

while (top != NULL)

{

del = top;

top = top->link;

delete del;

}

}

//进栈

void Push(const T& x)

{

top = new LinkNode<T>(x, top);

assert(top != NULL); //创建新结点失败退出

}

//退栈（删除栈顶结点，不需返回栈顶元素的值）

bool Pop()

{

if (!isEmpty())

{

LinkNode<T>\* p = top;

top = top->link;

delete p;

return true;

}

return false;

}

//退栈（删除栈顶结点，并返回栈顶元素的值）

bool Pop(T& x)

{

if (!isEmpty())

{

LinkNode<T>\* p = top;

top = top->link;

x = p->data;

delete p;

return true;

}

return false;

}

//判栈空

bool isEmpty()

{

return top == NULL;

}

//返回栈顶元素的值

bool getTop(T& x)

{

if (isEmpty())

{

return false;

}

x = top->data;

return true;

}

private:

LinkNode<T>\* top;

};

## 2.3.3 计算器类 **Calculator**

class Calculator

{

public:

//执行表达式计算

double Run(string& postExpr);

private:

LinkedStack<double> s; //栈

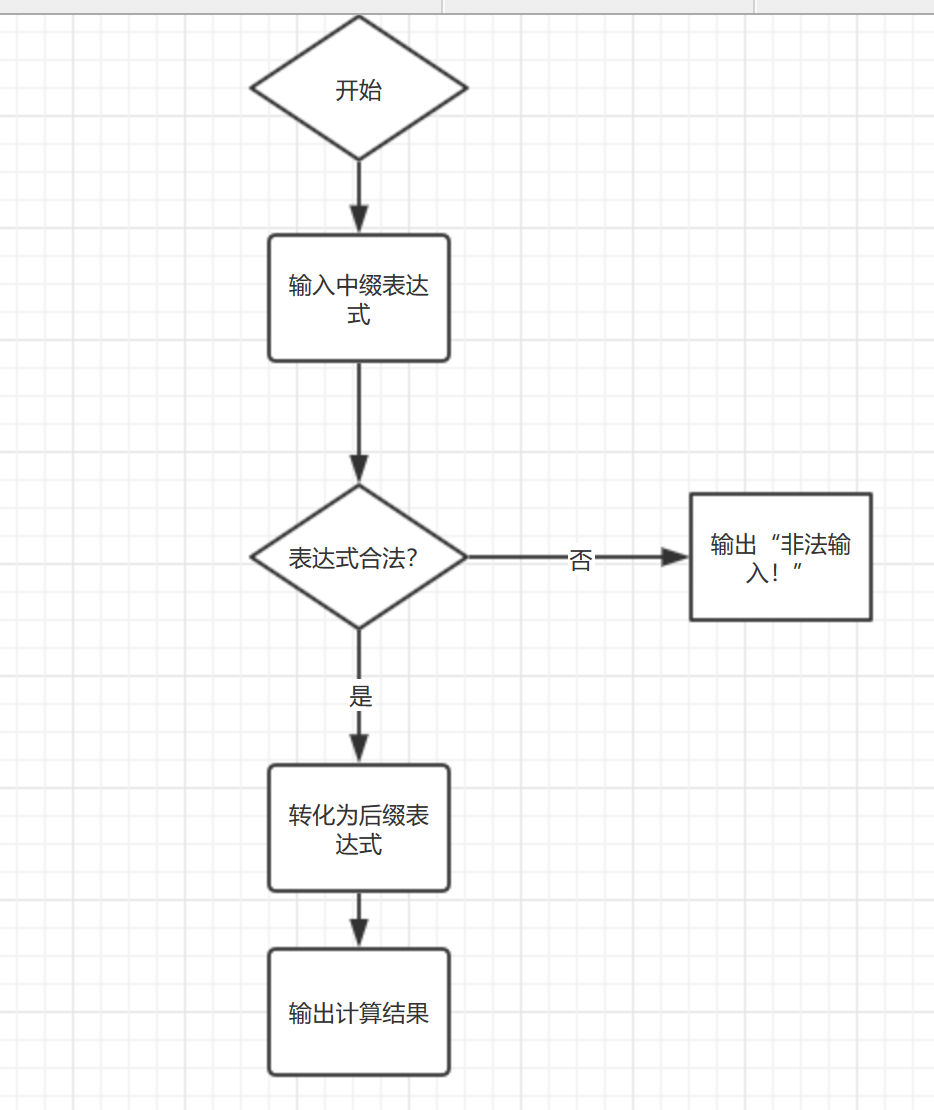
void AddOperand(double value); //进操作数栈

bool get\_2\_Operands(double& left, double& right); //从栈中退出两个操作数

void DoOperator(char op); //形成运算指令，进行计算

};

## 2.4 主函数设计



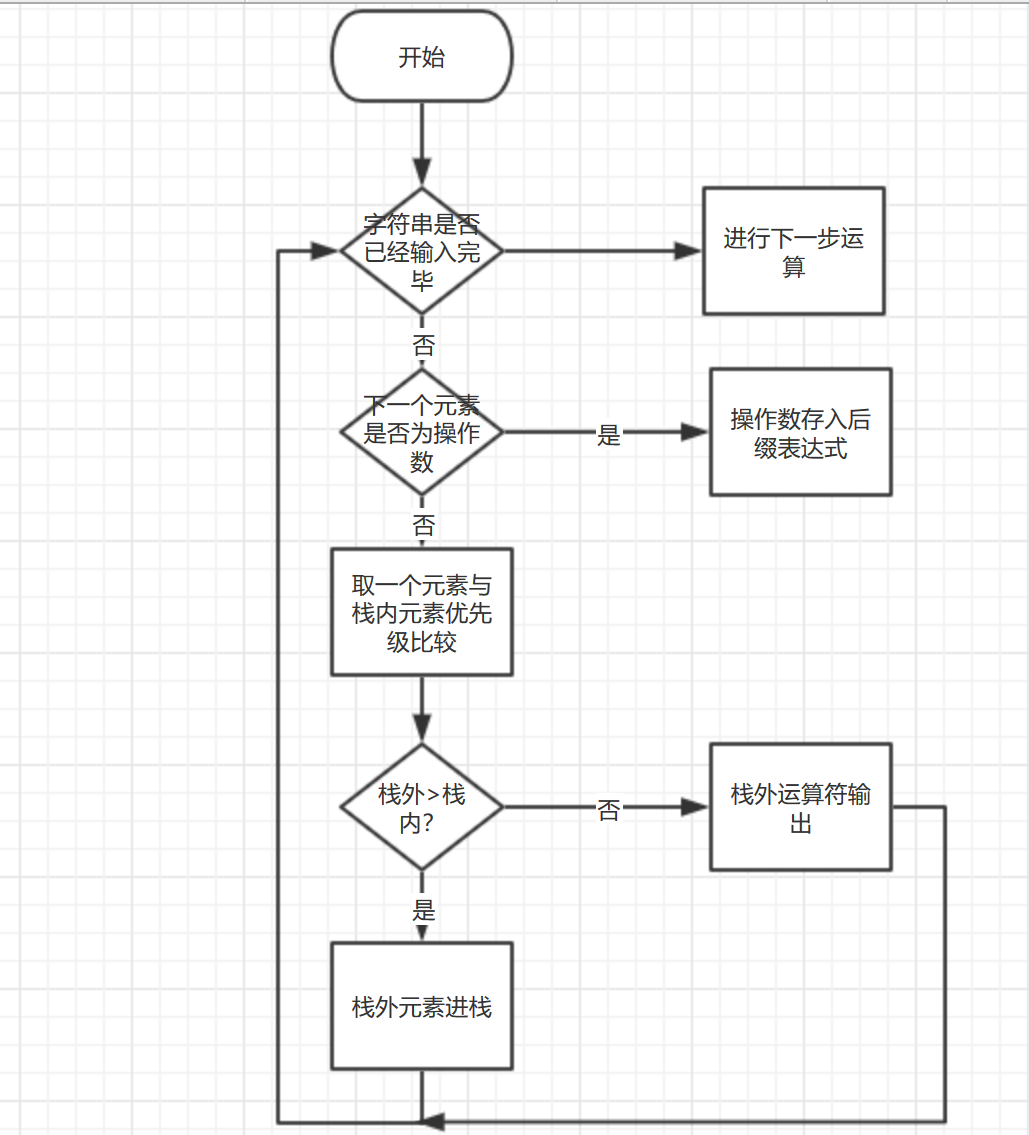
系统接受一个中缀表达式作为输入，调用函数判断表达式是否合理，如果合理，经过处理得到相应的后缀表达式，并计算该表达式，输出结果。

## **2.5 isValid()函数设计**

通过循环查看表达式每一个字符来判断表达式是否符合上面几条，如果不合法，就输出对应的提示信息，并且函数返回false，表示表达式有误，直到所有的条件都满足了，才返回true。

通过许多条件限制来判断是否是合理的表达式，如：左括号右边都没出现右括号；检测到了右括号后，在该处左边的右括号数量大于左括号数量；以及左括号右括号之中没有东西等。随后，当+-作为单目运算符，或者双目运算符的相关讨论也需要进行比较细致的分类讨论。再对其他运算符进行讨论是否合理。

## **2.6 toPostOrder()函数设计**



该函数目的是将之前的那个已经检验过合理性的表达式转为后续表达式，因为后序表达式能直接的计算。大致思路是：用栈存储运算符（char类型），然后遍历传入的字符串，分情况讨论，例如处理一目运算符+-（正负），处理数字，处理左右括号，处理其他运算符等。准备一个string存储后序表达式，如果是数字直接放入，如果是其他，则要小心的判断优先级，当遇到的优先级不断增大时，压入栈中，如果优先级相同,则将所有小于等于他优先级的运算符弹出,再在栈中加入这个运算符，以就能保证后续表达式优先级正确。

# 3 实现

## 3.1 主函数实现

先让用户输入表达式，然后判断合理性，合理的话再打印转换成为后续表达式的字符串，随后打印表达式的结果。

int main()

{

string inExpr; //输入的中序表达式

Calculator CALC;

char choice;

do

{

cout << "请输入表达式：" << endl;

getline(cin, inExpr);

if (isValid(inExpr))

{

string postExpr = toPostOrder(inExpr);

cout << "后序表达式为:" << postExpr << endl;

double result = CALC.Run(postExpr);

cout << "运算结果为：" << result << endl;

}

cout << "是否继续（y,n）？";

cin >> choice;

cin.get(); //用于吃掉空格

} while (choice == 'y');

system("pause");

return 0;

}

## 3.2 isValid()函数实现

### 3.2.1 简单的判断

对表达式进行遍历（对运算符和左右括号的判断也在遍历表达式的时候进行，只不过为了分节而在下面几节再叙述），如果出现了表达式意外的字符，即既不是数字又不是+-\*/%^=(),则立即提示并且返回。如果表达式没有结束之前出现了=,或者表达式最后不是=，也提示并且返回。

if (!(isDigit(expr[i]) || isOp(expr[i]) || expr[i] == '=' || expr[i] == '(' || expr[i] == ')'))

{

cerr << "表达式中含有非法字符!" << endl;

return false;

}

if (i != N - 1)

{

if (expr[i] == '=')

{

cerr << "表达式非末尾端出现了 = ！" << endl;

return false;

}

}

else

{

if (expr[i] != '=')

{

cerr << "表达式没有以 = 结尾！" << endl;

return false;

}

}

### 3.2.2 对左右括号的判断

对于每一个左括号，如果它右边位置都没出现右括号则提示并返回，这样能确保有足够的右括号，但是右括号可能多了，于是需要对右括号进行判断。

对于每一个右括号，从该位置计算，如果左边的左括号数量小于了右括号的数量（该右括号也算上），那么是不合理的，则提示并返回。这样对于每一个右括号，都保证它左边有充分的左括号（或许多了，但是经过上面的条件，则符合条件的左右括号应该正好完全匹配）。

还判断了空（）的情况，即左括号右括号之中没有东西，那么也视为不合理，则提示并返回。

if (expr[i] == ')')

{

int leftCount = 0, rightCount = 0; //left记录（的个数, right记录）的个数

for (int j = 0; j <= i; j++)

{

if (expr[j] == ')')

{

rightCount++;

}

else if (expr[j] == '(')

{

leftCount++;

}

}

if (rightCount > leftCount)

{

cerr << "未找到左括号匹配右括号!" << endl;

return false;

}

}

if (expr[i] == '(')

{

if (expr.find(')', i) == -1)

{

cerr << "未找到右括号匹配左括号！" << endl;

return false;

}

}

if (expr[i] == ')' && i - 1 >= 0 && expr[i - 1] == '(')

{

cerr << "表达式中存在()内为空！" << endl;

return false;

}

### 3.2.3单目表达式的判断

如果这两个符号是表达式的第一个元素，或者它左边是左括号（这里假定合理的表达式中单目表达式都用括号括起来，如（-1）），则是单目表达式，代表的是正负。而是单目运算符时，他右边可以是一个子表达式（判断条件是它右边第一个元素是左括号），或者是一个数（判断条件是右边第一个字符大于‘0’并且小于‘9’）。否则不合理，则提示并返回。

如果出现的=-不是单目运算符正负，那么再通过左右条件判断是否是合理的双目运算符，即为加减。

//单目

if ((expr[i] == '+' || expr[i] == '-') && (i == 0 || expr[i - 1] == '(')) //单目运算符只可能出现在开头或者（右边

{

if (!(i + 1 < N && (expr[i + 1] == '(' || isDigit(expr[i + 1])))) //后面没有接操作数或者 (

{

cerr << "单目运算符 + - 不合理 ！" << endl;

return false;

}

}

### 3.2.4 双目表达式的判断

首先判断双目表达式的左右两边不是空的，因为双目表达式需要接受两个操作数。然后判断它左右两边是否是子表达式，或者是数字。如果不是这两者之一，就是不合理的双目表达式，则不合理，则提示并返回。

//双目

else if (isOp(expr[i]))

{

if (!(i - 1 >= 0 && i + 1 < N && ((expr[i - 1] == ')' || isDigit(expr[i - 1])) && (expr[i + 1] == '(' || isDigit(expr[i + 1])))))

{

cerr << "双目运算符 + - \* / % ^ 不合理!" << endl;

return false;

}

}

## 3.3 toPostOrder()函数实现

3.3.1 用一个map将运算符和他对应的优先级绑定（用数字表示，数字越大优先级越高）。并且准备一个栈存储运算符字符，一个字符串存储整个后续表达式。

unordered\_map<char, int> priority; //map数组存储运算符的优先级

priority['='] = 1;

priority['('] = 2;

priority[')'] = 2;

priority['+'] = 3;

priority['-'] = 3;

priority['\*'] = 4;

priority['/'] = 4;

priority['%'] = 5;

priority['^'] = 6;

string postExpr; //存储后序表达式的字符串

LinkedStack<char> s; //运算符栈

开始遍历整个原始的表达式字符串，根据字符的不同采取不同的操作：

3.3.2 如果是单目运算符=-(正负),(判断是他们的条件是他们是第一个字符或者前方是左括号),如果是+,不处理,如果是-,那么就相当于是0-目标数,例如-5,那么后续表达式就加入0 5 -。

if ((expr[i] == '+' || expr[i] == '-') && (i == 0 || expr[i - 1] == '(')) //单目运算符

{

if (expr[i] == '-')

{

postExpr += "0 ";

EnDigit(expr, postExpr, ++i); //将 - 后面的操作数读入

postExpr += "- ";

}

}

3.3.3 如果是数字，那么用另外的指针记录该位置，该指针循环往后直到找到第一个非数字字符，把整个数字都截取出来加入到后序表达式中，再再后续表达式后面加上一个空格。

//从expr的i位置读入操作数,如果输入大于0-9的数字也能正确处理

void EnDigit(const string& expr, string& postExpr, int& i)

{

int j = i + 1;

while (true)

{

if (j < (int)expr.length() && isDigit(expr[j]))

{

j++;

}

else

{

break;

}

}

postExpr += expr.substr(i, j - i); //数字的字符串

postExpr += " ";

i = j - 1; //更新i值

}

3.3.4 处理左括号（，左括号就直接压入栈中，将来作为一个弹栈的标志。

else if (expr[i] == '(') //左括号

{

s.Push(expr[i]);

}

3.3.5 处理双目运算符。首先，如果栈为空的话，直接将该运算符压入栈中，而如果栈不为空，则需要与栈顶元素对比，将所有大于等于该运算符优先级的栈元素依次取出栈，放入后续表达式中，这样就能保证优先级大的总是优先计算，而大于等于的都取出来是因为如果是等于的话，代表前后的优先级是相同的，但是为了从左往右计算，所以需要把左边那个优先级相等的运算符先取出来放入后续表达式中。当确保栈中没有大于等于该运算符的元素后，再将该运算符放入栈中。

else //运算符

{

if (s.isEmpty()) //栈为空,直接压入即可

{

s.Push(expr[i]);

}

else

{

//与栈顶运算符比较优先级

char top; s.getTop(top);

if (priority[expr[i]] > priority[top])

{

s.Push(expr[i]);

}

else

{

while (!s.isEmpty() && priority[expr[i]] <= priority[top])

{

postExpr += top;

postExpr += " ";

s.Pop();

if (!s.isEmpty())

{

s.getTop(top); //栈顶变化，需要更新

}

}

s.Push(expr[i]);

}

}

}

}

3.3.6 处理右括号）运算符，到这一步时，只需要不断弹栈，不断将栈中运算符加入后续表达式中直到遇到左括号（为止，因为在处理其他运算符时运算符已经按照相应的顺序在栈中或者在后续表达式中摆放好了，所以直接这样处理即可。

else if (expr[i] == ')') //右括号

{

char top; s.getTop(top);

while (top != '(')

{

postExpr += top;

postExpr += " ";

s.Pop();

s.getTop(top);

}

s.Pop(); //弹出(

}

3.3.7 循环结束后，已经将所有的数字和大部分运算符放入了后续表达式中，但是最后还会有少部分的运算符存在栈中（除非表达式最后一个字符是右括号）），所以此时要把栈中剩余所有元素放入后续表达式中。最后，将后续表达式返回即可。

//剩下的运算符一起加上

while (!s.isEmpty())

{

char top; s.getTop(top);

if (top != '=')

{

postExpr += top;

postExpr += " ";

}

s.Pop();

}

return postExpr;

}

## 3.4 **Caculate类成员函数的实现**

### **3.4.1函数描述**

用Run()执行运算。

传入一个后续表达式，将计算的结果以double类型返回。

用栈存储double类型数字。对后续表达式进行遍历，如果是数字，同样，得按照和前面一样的方法，将整个数字取出来，如23等，将数字放入栈中。如果是运算符，根据运算符的不同，采取相应的操作，但其实操作是很类似的，都是从栈中取出两个数字，用前面的数采用相应的运算符和后面的数进行计算，并且将运算符重新加入栈中。

**3.4.2 代码实现**

**1. 从操作数栈中取出两个操作数**

bool Calculator::get\_2\_Operands(double& left, double& right)

{

if (s.isEmpty())

{

cerr << "缺少右操作数！" << endl;

return false;

}

s.Pop(right);

if (s.isEmpty())

{

cerr << "缺少左操作数！" << endl;

return false;

}

s.Pop(left);

return true;

}

1. **操作数的值value进入栈**

void Calculator::AddOperand(double value)

{

s.Push(value);

}

1. **取两个操作数，根据操作符op形成运算指令并计算**

void Calculator::DoOperator(char op)

{

double left, right, value;

int result = get\_2\_Operands(left, right); //取两个操作数

if (result)

{

switch (op)

{

case'^':

value = pow(left, right); AddOperand(value); break;

case'\*':

value = left \* right; AddOperand(value); break;

case'%':

value = (int)left % (int)right; AddOperand(value); break;

case'/':

if (right == 0)

{

cerr << "除数为零!" << endl;

s.makeEmpty();

}

else

{

value = left / right;

AddOperand(value);

}

break;

case'+':

value = left + right; AddOperand(value); break;

case'-':

value = left - right; AddOperand(value); break;

}

}

else

{

s.makeEmpty();

}

}

1. **求一个后缀表达式的值，以字符‘=’结束**

double Calculator::Run(string& postExpr)

{

s.makeEmpty();

int N = postExpr.length();

for (int i = 0; i < N; i += 2)

{

char ch = postExpr[i];

if (ch == '^' || ch == '%' || ch == '\*' || ch == '/' || ch == '+' || ch == '-')

{

DoOperator(ch);

}

else

{

//考虑大于0-9的操作数

int j = i + 1;

while (true)

{

if (j < N && (postExpr[j] <= '9' && postExpr[j] >= '0'))

{

j++;

}

else

{

break;

}

}

s.Push(atoi(postExpr.substr(i, j - i).c\_str())); //postExpr.substr(i, j - i); 得到了数字的字符串

i = j - 1;

}

}

double value;

s.getTop(value);

return value;

}

# 

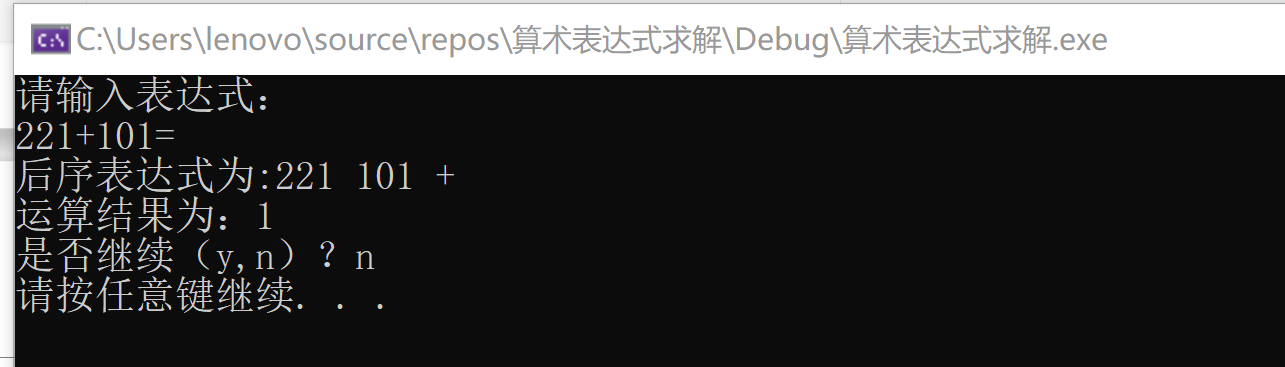
# 4 测试

## 4.1 功能测试

### 4.1.1 测试1

**测试用例**：221+101=

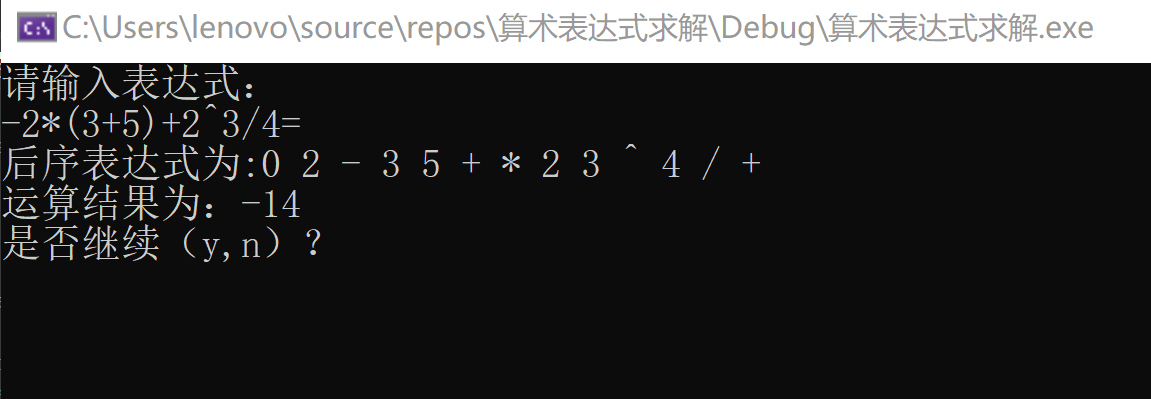
**实验结果：**

****

### 4.1.2 测试2

**测试用例：**-2\*(3+5)+2^3/4=

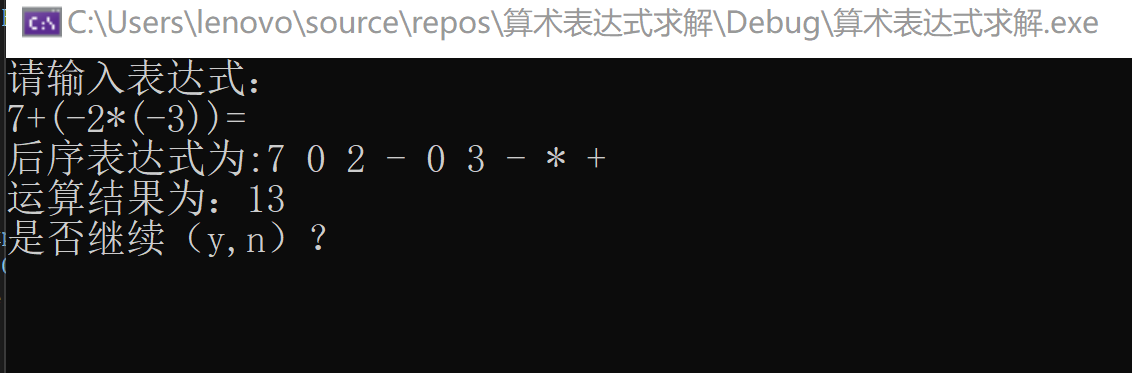
**实验结果：**

****

### 4.1.3 测试3

**测试用例：7**+(-2\*(-3))=

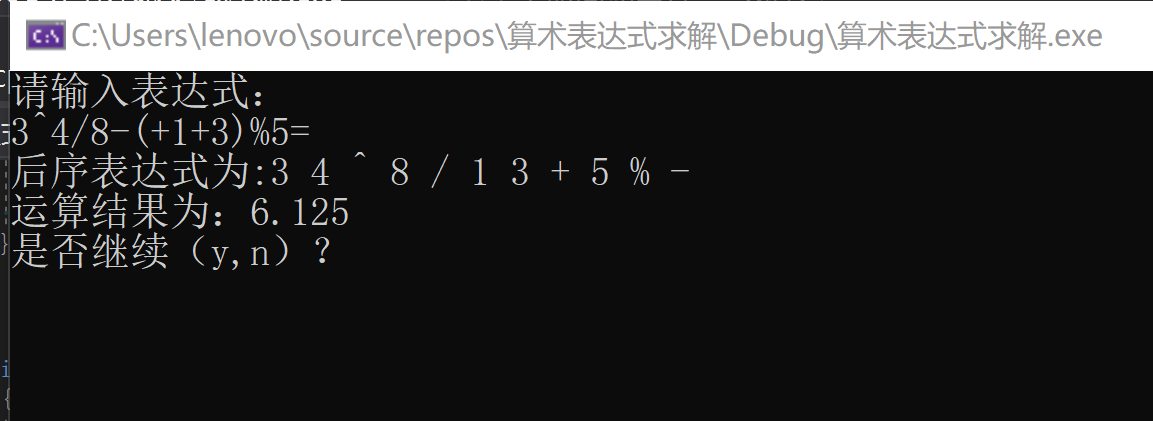
**实验结果：**

****

### 4.1.4 测试4

**测试用例：** 3^4/8-(+1+3)%5=

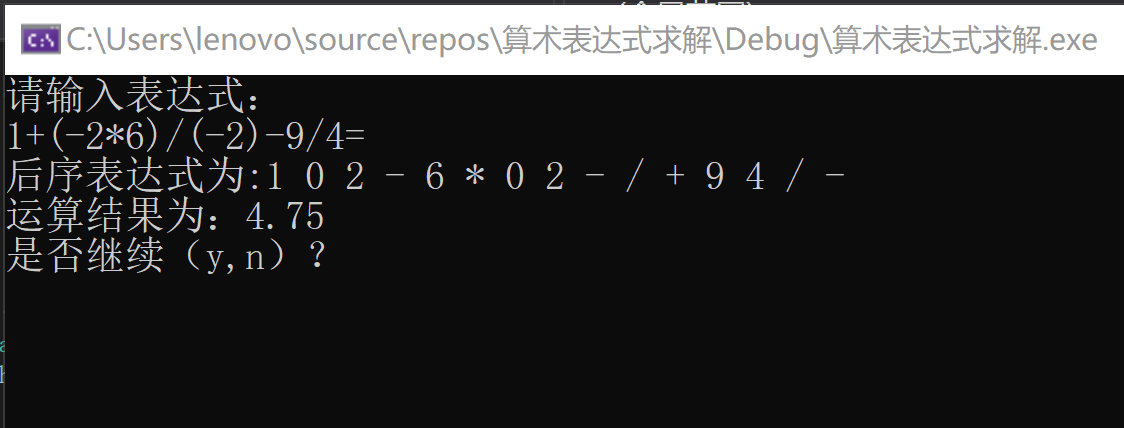
**实验结果：**

****

### 4.1.4 测试5

**测试用例：** 1+(-2\*6)/(-2)-9/4=

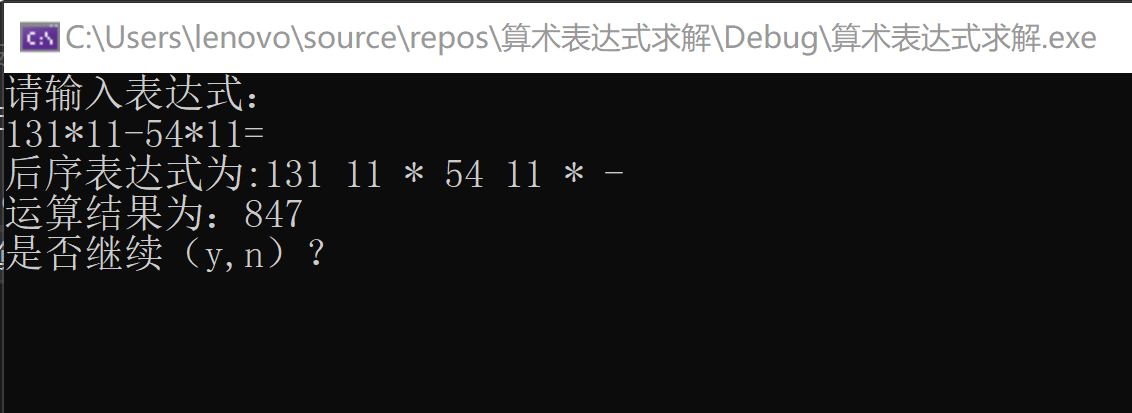
**实验结果：**

****

### 4.1.4 测试6

**测试用例：** 131\*11-54\*11=

**实验结果：**

****

## 4.2 非法表达式测试

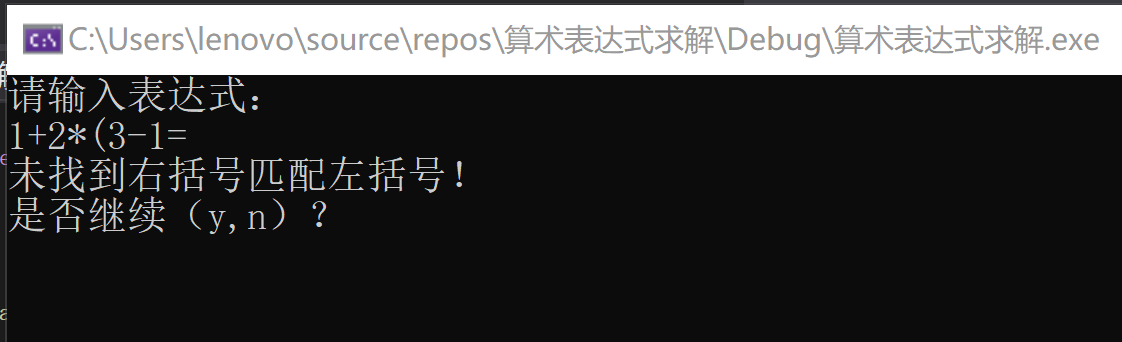
### 4.2.1 含有非法字符



### 4.2.2 左右括号中内容为空

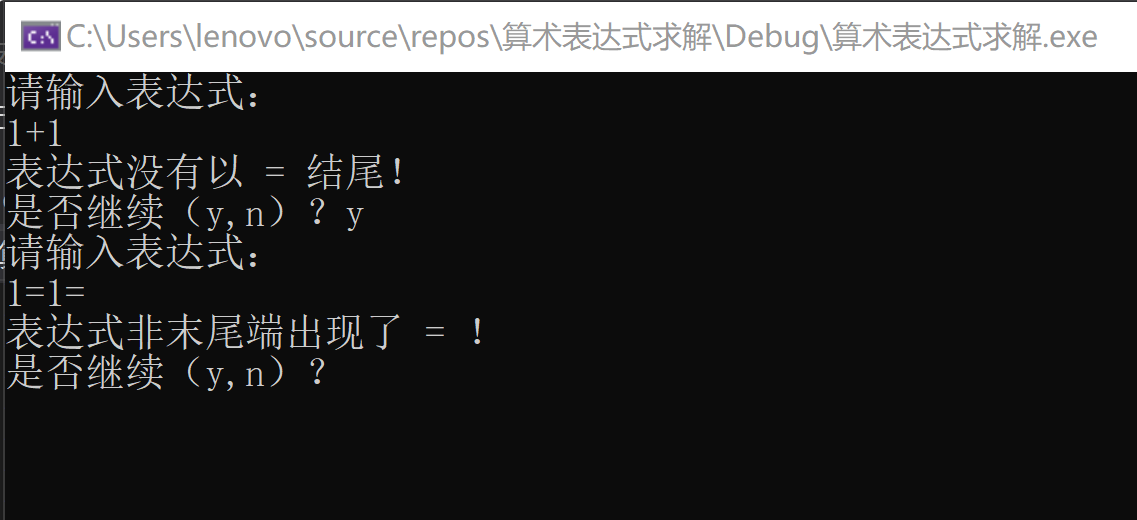


### 4.2.3 左右括号不匹配

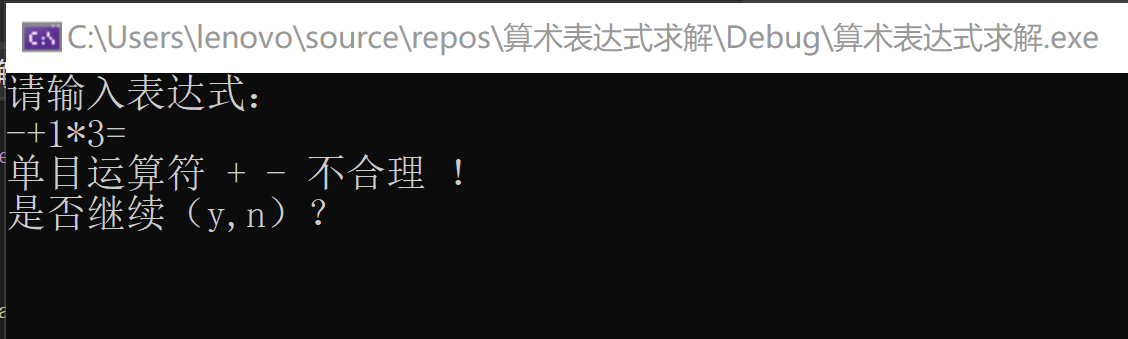




### 4.2.4 等号位置错误



### 4.2.5 单目运算符使用不合理



### 4.2.6 双目运算符使用不合理

