算数表达式求解

作 者 姓 名： 李翠琪

学 号： 1751022

指 导 教 师： 张颖 \_\_

学院、 专业： 软件工程 \_\_

同济大学

Tongji University

目录

[1 分析 1](#_Toc24701351)

[1.1 项目要求 1](#_Toc24701352)

[1.2 功能分析 1](#_Toc24701353)

[2 设计 2](#_Toc24701354)

[2.1 主函数设计 2](#_Toc24701355)

[2.2 isValid()函数设计 2](#_Toc24701356)

[2.3 toPostOrder()函数设计 2](#_Toc24701357)

[2.4 caculatePostOrder ()函数设计 3](#_Toc24701358)

[3 实现 3](#_Toc24701359)

[3.1 主函数实现 3](#_Toc24701360)

[3.2 isValid()函数实现 4](#_Toc24701361)

[3.2.1 简单的判断 4](#_Toc24701362)

[3.2.2 对左右括号的判断 4](#_Toc24701363)

[3.2.3单目表达式的判断 5](#_Toc24701364)

[3.2.4 双目表达式的判断 6](#_Toc24701365)

[3.3 toPostOrder()函数实现 6](#_Toc24701366)

[3.3.1函数描述 6](#_Toc24701367)

[3.3.2 函数代码 7](#_Toc24701368)

[3.4 caculatePostOrder（）的实现 9](#_Toc24701369)

[3.4.1函数描述 9](#_Toc24701370)

[3.4.2 函数代码 10](#_Toc24701371)

[4 测试 11](#_Toc24701372)

[4.1 功能测试 11](#_Toc24701373)

[4.1.1 测试1 11](#_Toc24701374)

[4.1.2 测试2 11](#_Toc24701375)

[4.1.3 测试3 12](#_Toc24701376)

[4.1.4 测试4 12](#_Toc24701377)

[4.1.4 测试5 13](#_Toc24701378)

[4.1.4 测试6 13](#_Toc24701379)

[4.2 非法表达式测试 14](#_Toc24701380)

[4.2.1 含有非法字符 14](#_Toc24701381)

[4.2.2 左右括号中内容为空 14](#_Toc24701382)

[4.2.3 左右括号不匹配 14](#_Toc24701383)

[4.2.4 等号位置错误 15](#_Toc24701384)

[4.2.5 正负号不合理 15](#_Toc24701385)

[4.2.6 双目运算符使用不合理 16](#_Toc24701386)

# 1 分析

## 1.1 项目要求

从键盘上输入中缀算数表达式，包括括号，计算出表达式的值。

程序对所有输入的表达式作简单的判断，如表达式有错，能给出适当的提示。

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-。

## 1.2 功能分析

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 主函数设计

先输入表达式，调用函数判断表达式是否合理，如果合理，首先将之转换为后续表达式，输出后续表达式的字符串，并且调用函数计算后续表达式，并且打印后续表达式的结果。

（因为之后需要用到栈，所以程序还出现了Stack.h，与Node.h，因为之前的题目也用过，所以这里直接复用之前已经实现的了，不在重新说明。）

## 2.2 isValid()函数设计

大致就是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。

主要的难度就是对（）合理性的判断，于是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。

## 2.3 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。这样就能保证后续表达式优先级正确了。在每个元素之间，还加入了一个空格（如12 5 +），这样可以避免出现将之读为125的bug。大致就是这样转换，但是还有许多细节来保证转换正确，这将在函数实现中进行细致讨论。

## 2.4 caculatePostOrder ()函数设计

该函数通过传入一个后续表达式,输出一个int值,代表表达式的结果。主要设计思路是用栈存储数值，遍历后续表达式，如果是数字就把整个数字压入栈中（如遇到19）会把19压入，而不是压入1和9。如果遍历到了操作符，那么就将栈顶两个元素取出并且计算，再压入栈中（因为这时是后续表达式，后续表达式已经把优先级等都考虑好了，所以只要根据符号采取特定的方法计算即可）。遍历完之后将栈顶元素返回，这个即为表达式的值。

# 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，最后让用户选择是否结束程序（不这样的话运行exe文件输入完之后就会退出，强制设定退出时间也不太好，所以让用户自己选择）。

int main()  
{  
 while(true)  
 {  
 string expr,isContinue;  
 cout << "输入表达式：" << endl;  
 getline(cin,expr);  
 if(isValid(expr))  
 {  
 string postOrderedExpr=toPostOrder(expr);  
 cout <<"后序表达式为:"<< postOrderedExpr<<endl;  
 int result = caculatePostOrder(postOrderedExpr);  
 cout<<result<<endl;  
 }  
 cout<<"任意键继续,n退出？";  
 getline(cin,isContinue);  
 if(isContinue=="n")  
 {  
 break;  
 }  
 }  
 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(expr[i]!='+'&&expr[i]!='-'&&expr[i]!='\*'&&expr[i]!='/'&&expr[i]!='%'&&expr[i]!='^'&&  
 expr[i]!='('&&expr[i]!=')'&&expr[i]!='='&& !(expr[i]<='9'&&expr[i]>='0'))  
{  
 cout<<"表达式含有非法字符"<<endl;  
 return false;  
}

if(i==N-1 && expr[i]!='=')  
{  
 cout << "不以=结尾"<<endl;  
 return false;  
}  
if(i!=N-1 && expr[i]=='=')  
{  
 cout << "表达式中出现了="<<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]=='(')  
{  
 if(expr.find(')', i)==-1)  
 {  
 cout<<"没有足够的右括号匹配左括号"<<endl;  
 return false;  
 }  
}  
if(expr[i]==')')  
{  
 int leftCount = 0, rightCount = 0;//()的个数  
 for (int j = 0; j <= i; ++j)  
 {  
 if(expr[j]=='(')  
 {  
 ++leftCount;  
 }  
 else if(expr[j]==')')  
 {  
 ++rightCount;  
 }  
 }  
 if(leftCount<rightCount)  
 {  
 cout<<"没有足够的左括号匹配右括号"<<endl;  
 return false;  
 }  
}  
if(expr[i]=='(' &&i+1<N && expr[i+1]==')')  
{  
 cout<<"出现了空()"<<endl;  
 return false;  
}

### 3.2.3单目表达式的判断

之所以单独讨论，是因为要以单目运算符，双目运算符来分别讨论。

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。而这样他们和其他运算符判断方式一样，在下一小节统一叙述。

if(expr[i]=='+' || expr[i]=='-')  
{  
 if(i==0 || expr[i-1]=='(')//满足第四点 单目运算符  
 {  
 if(i+1<N && ((expr[i+1]=='(')||(expr[i+1]<='9'&&expr[i+1]>='0')))//合理了的  
 {  
 }  
 else  
 {  
 cout<<"+-后面没有数字或者("<<endl;  
 return false;  
 }  
 }  
 //i+1存在,并且左边是)或者数字,并且右边是(或者数字  
 else if(i+1<N && (expr[i-1]==')' || (expr[i-1]<='9'&&expr[i-1]>='0'))&& ((expr[i+1]=='(')||(expr[i+1]<='9'&&expr[i+1]>='0')))//i-1肯定>=0,因为上面的条件  
 {  
  
 }  
 else  
 {  
 cout<<"不合理的+-"<<endl;  
 return false;  
 }  
}

### 3.2.4 双目表达式的判断

判断很简单，首先，要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。

if(expr[i]=='^'||expr[i]=='%'||expr[i]=='\*'||expr[i]=='/')  
{  
 if(i-1>=0 && i+1<N && ((expr[i-1]==')' || (expr[i-1]<='9'&&expr[i-1]>='0'))&& ((expr[i+1]=='(')||(expr[i+1]<='9'&&expr[i+1]>='0'))))  
 {  
 //合格  
 }  
 else  
 {  
 cout<<"不合理的\*/^%"<<endl;  
 return false;  
 }  
}

## 3.3 toPostOrder()函数实现

### 3.3.1函数描述

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。

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：

1.如果是一目运算符=-(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 -,这里注意两点,一是,作为负号,优先级是最高的,所以不用考虑其他符号,直接加入即可。二是需要小心的加入空格，如这里必须要是0 5 -，不能是05 -。

2.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，避免出现原本代表两个数的后续表达式（10 5 +）变成了错误的（105+）。

3.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。

4.处理非）的其他字符，因为是按顺序处理下来的，代表前方已经处理了一目运算符，左括号，数字，所以这里的 ！=‘）’ 代表了所有其他的双目运算符。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。

5.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。

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。

### 3.3.2 函数代码

string toPostOrder(const string & expr)  
{  
 unordered\_map<char,int> priority;//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 postOrderedExpr;  
 Stack<char> ops;//存储运算符的  
 for (int i = 0; i < expr.length(); ++i)  
 {  
 //处理一目运算符  
 if((expr[i]=='+' || expr[i]=='-')&&(i == 0 || expr[i - 1] == '('))//当作正负来使用时  
 {  
 if(expr[i]=='+')  
 {  
 //忽略他  
 }  
 else  
 {  
 postOrderedExpr += "0 ";  
 ++i;//将-后面的读入  
 postOrderedExpr += expr[i];  
 postOrderedExpr += " - ";  
 }  
  
 }  
 else if(expr[i]<='9'&&expr[i]>='0')//这里需要处理下,就是如果输入大于0-9的数字也能正确处理  
 {  
 int hi = i+1;  
 while (true)  
 {  
 if (hi < expr.length() && expr[hi] <= '9'&&expr[hi] >= '0')  
 {  
 ++hi;  
 }  
 else  
 {  
 break;  
 }  
  
 }  
 //hi指向的是数字后面的第一个非数字字符  
 //expr.substr(i, hi - i); 得到了数字的字符串  
 postOrderedExpr += expr.substr(i, hi - i);  
 postOrderedExpr += " ";//避免出现10+5->105+的问题  
 i = hi - 1;  
 }  
 else if(expr[i]=='(')  
 {  
 ops.push(expr[i]);  
 }  
 else if(expr[i]!=')')//除)外的运算符都压入  
 {  
 if(ops.empty())//栈为空,简单压入即可  
 {  
 ops.push(expr[i]);  
 }  
 else  
 {  
 //需要与栈顶运算符比较优先级  
 int topStackPriority = priority[ops.top()];  
 if(priority[expr[i]]>topStackPriority)  
 {  
 ops.push(expr[i]);  
 }  
 else  
 {  
 //优先级相同,必将弹出之前所有小于等于他优先级的运算符,再加入这个运算符  
 while(!ops.empty() && priority[expr[i]]<=topStackPriority)  
 {  
 postOrderedExpr += ops.top();  
 postOrderedExpr += " ";  
 ops.pop();  
 if(!ops.empty())  
 {  
 topStackPriority = priority[ops.top()];  
 }  
 }  
  
 ops.push(expr[i]);  
 }  
 }  
 }  
 else if(expr[i]==')')  
 {  
 //表达式检验过,一定正确,所以可以直接这样用  
 while(ops.top()!='(')  
 {  
 postOrderedExpr += ops.top();  
 postOrderedExpr += " ";//避免出现10+5->105+的问题  
 ops.pop();  
 }  
 ops.pop();//把(弹出去  
 }  
 }  
 //剩下的运算符一起加上  
 while(!ops.empty())  
 {  
 if(ops.top()!='=')//不要=  
 {  
 postOrderedExpr += ops.top();  
 postOrderedExpr += " ";  
 }  
 ops.pop();  
 }  
 return postOrderedExpr;  
}

## 3.4 caculatePostOrder（）的实现

### 3.4.1函数描述

传入一个后续表达式，将计算的结果以int类型返回。

用栈存储int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。

因为一目运算符正负=-已经转换为空或者双目运算符，左右括号也去掉，表达式也是合理的了，所以只要这样简单的运算到后续表达式结束即可得到结果，而结果就是栈顶元素。

### 3.4.2 函数代码

（一些重复部分没放上来）

int caculatePostOrder(const string & postOrderedExpr)  
{  
 Stack<int> nums;  
 for (int i = 0; i < postOrderedExpr.length(); ++i)  
 {  
 if(postOrderedExpr[i]<='9'&&postOrderedExpr[i]>='0')  
 {  
 //还要考虑处理大于10的数字  
  
 int hi = i + 1;  
 while (true)  
 {  
 if (hi < postOrderedExpr.length() && postOrderedExpr[hi] <= '9'&&postOrderedExpr[hi] >= '0')  
 {  
 ++hi;  
 }  
 else  
 {  
 break;  
 }  
  
 }  
 //hi指向的是数字后面的第一个非数字字符  
 //postOrderedExpr.substr(i, hi - i); 得到了数字的字符串  
  
 nums.push(atoi(postOrderedExpr.substr(i, hi - i).c\_str()));  
  
 i = hi - 1;  
 }  
 else if(postOrderedExpr[i]=='^')  
 {  
 //num1为左,num2为右  
 int num2 = nums.top();  
 nums.pop();  
 int num1 = nums.top();  
 nums.pop();  
 nums.push(pow(num1, num2));  
 }  
 else if(postOrderedExpr[i]=='%')  
 {  
 //num1为左,num2为右  
 int num2 = nums.top();  
 nums.pop();  
 int num1 = nums.top();  
 nums.pop();  
 nums.push(num1%num2);  
 }  
  
 }  
 return nums.top();  
}

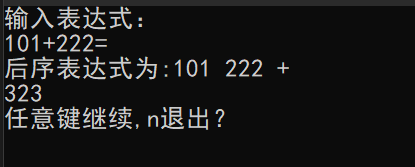
# 4 测试

## 4.1 功能测试

### 4.1.1 测试1

**测试用例**：101+222=

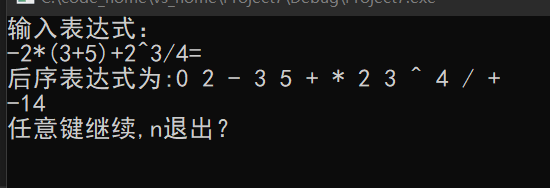
**实验结果：**



### 4.1.2 测试2

**测试用例：**-2\*(3+5)+2^3/4=

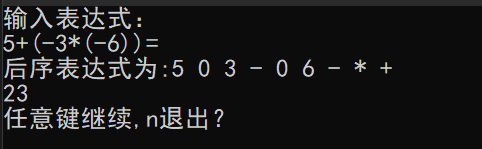
**实验结果：**



### 4.1.3 测试3

**测试用例：**5+(-3\*(-6))=

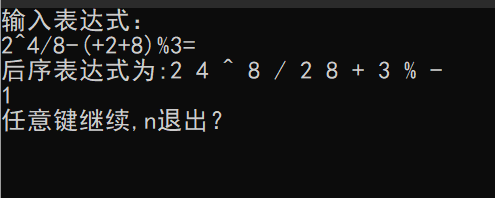
**实验结果：**



### 4.1.4 测试4

**测试用例：** 2^4/8-(+2+8)%3=

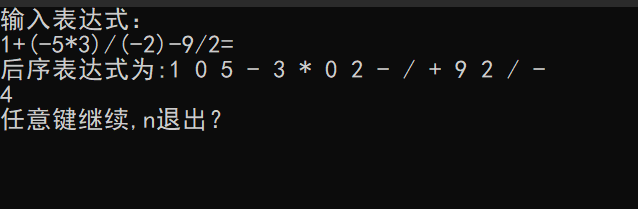
**实验结果：**



### 4.1.4 测试5

**测试用例：** 1+(-5\*3)/(-2)-9/2=

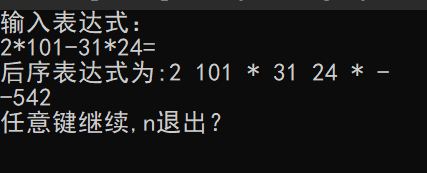
**实验结果：**



### 4.1.4 测试6

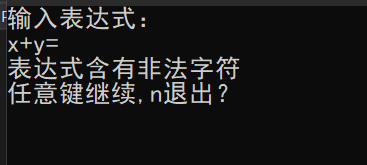
**测试用例：** 2\*101-31\*24=

**实验结果：**

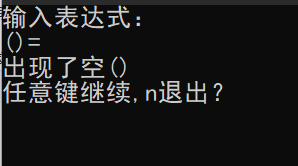


## 4.2 非法表达式测试

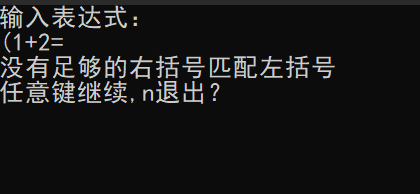
### 4.2.1 含有非法字符

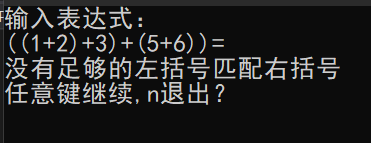


### 4.2.2 左右括号中内容为空

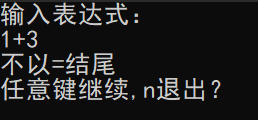


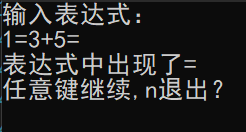
### 4.2.3 左右括号不匹配

****

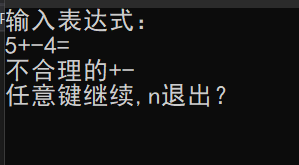


### 4.2.4 等号位置错误

****



### 4.2.5 正负号不合理



### 4.2.6 双目运算符使用不合理

