一、问题描述

使用计算机的方式读取一个字符串写成的表达式，并计算出结果。其中，要考虑到算式中运算优先级、正负号、小数点、括号的处理，以此作为计算器的基础，制作一个带有界面的计算器。

二、问题分析

作为生活中的一个常见问题，生活中人们对于表达式的处理方式是先算优先级高再算优先级低的运算，其本质就是跳过了低优先级运算，等待一个更高优先的运算，如果下一个运算的优先级更低，则先计算当前优先级的运算。生活中对于括号的处理则是，先找到最内括号，即内部没有括号的括号，计算出结果，并主观上记忆或进行替换，再按照正常方式继续计算。这为本实验提供了一个算法的入手点。

三、输入输出样例

1）运算优先级的可靠性

输入：3+2\*2^3

输出：19

2）括号及小数点的可靠性

输入：3.5\*(2.3+2)^2

输出：64.715

3）首位为负号的可靠性

输入：-3.25\*3

输出：-9.75

4) 多重括号的可靠性

输入：((1+1)\*2+(2-3)\*3)\*2

输出：2

四、设计思路

采用堆栈的方式解决问题，把字符串一个一个的放入栈中，若读取为操作数，压入数据栈NS中，若读取为更高级运算符，压入运算符号栈OS中，若读取为低级运算符，则进行计算，计算之后压入NS栈。全部运算完成输出结果。

本实验中采用的做法是先对表达式进行处理，处理为一个链表，每个结点由三个部分组成：分别是操作数部分，运算符号部分，以及指针部分。作为操作数，把这个结点的运算符号部分设置为‘0’，以便于识别其结点数据类型。把表达式的数（包括负数、小数）存在一个结点中，把运算符号存在结点中，即形成了一个由数和运算符号交叉的链表。

对于括号，本实验采用了递归算法，即先找到最内的一个括号（里面不再存在括号），把里面的表达式优先计算出结果，并存入一个数据结点中，再利用链表的运算，插入到原链表中，解决了括号优先计算的问题。并对于括号中只有一个负数的情况有了很好的解决，避免出现如处理“1+(-3)”这类问题时符号与数字个数不匹配而产生的问题。这也是本实验使用链表而不用数组处理的原因，即使用数组不便于括号内的结果的替换，而造成一系列的麻烦。

五、数据结构

OS栈，用于存储运算符号的栈

NS栈，用于存储操作数的栈

Input[N]，存储用户输入的表达式

Negative\_flag 用于判断首位是否为负号，首位是负号则为1，反之为0

Dot\_flag 用于判断读到了小数点，读到为1，反之为0

Dot\_num 用于记小数点后的位数

六、算法描述

main()

head←Change(input); //将字符串转换成链表

ans←Calculate(head);

Print ans;

Return

其中Change(input)为转换函数，把字符串转换为链表

Change(input)

GETNODE(head);

Negative\_flag←0;

if input[0]=’-’ //解决首位是正号或者负号的情况

negative\_flag←1;

Eat(input); //吃掉下一位

Else if input[0]=’+’

Negative\_flag←0;

Eat(input);

End(if)

//解决小数点问题

For i=0 to input[i]<>’\0’

If input[i]是数字

{

r←0;

dot\_flag←0;

dot\_num←0;

while input[0]是数字||input[0]=’.’

If input[0]是数字&&dot\_flag=0 //处理整数部分

{

w←input[0];

Eat(input);

r←r\*10+Exchange\_n(w);

}

else if input[0]=’.’

{

w←input[0];

Eat(input);

Dot\_flag←1;

Dot\_num←1;

}

Else if input[0]是数字&&dot\_flag=1 //处理小数部分

{

w←input[0];

r←r+Exchange\_n(w)\*pow(0.1, dot\_num);

dot\_num++;

Eat(input);

}

If(negative\_flag=1) //有负号时取反

{

r←-r;

negative\_flag←0;

}

GETNODE(newnode);

Newnode->number=r;

Newnode->operate=’0’

Tail->next←newnode;

tail←newnode;

}

Else //如果是运算符号，将其存入结点

{

GETNODE(newnode);

Newnode->operate←input[0];

Tail->next←newnode;

tail←newnode;

Eat(input);

}

End(i)

//在最后加上分号，就不需要用户输入分号了

GETNODE(newnode);

Newnode->operate←’;’;

Tail->next←newnode;

tail←newnode;

tail->next←nil;

return head;

其中，Exchange\_n函数用于将数字字符转化为数值

Exchange\_n(x)

n←x-48;

return n;

Eat函数用于吃掉字符数组的第一位

Eat(input)

{

For j=0 to strlen(input)-2

Input[j] ←input[j+1];

End(j)

Input[strlen(input)-1] ←’\0’;  
}

Calculate函数是将表达式计算为数值结果的函数，采用了递归算法处理括号

Calculate(head)

{

Searchflag(); //搜索出最内括号表达式

While Searchflag()=1 //说明有括号

{

h←括号内部表达式的链表

Calculate(h);

}

While t<>2

{

If t=0 then p←p->next;

If 结点中存有数值 then PUSH(NS, p->number);

Else if 结点中存有运算符号

{

w←p->operate;

s←TOP(OS);

if s>w then PUSH(OS,w); //新符号优先，入栈

else if s=’;’&&w=’;’ //都是分号说明运算结束，得出结果

{

answer←TOP(NS);

POP(NS);

Return answer;

t←2;

}

Else //如果旧符号优先或平权，则先进行旧符号的运算，存入NS栈

{

x←TOP(NS);

y←TOP(NS);

e←TOP(OS);

m←EXP(x, y, e);

PUSH(NS, m);

t←1;

}

}

}

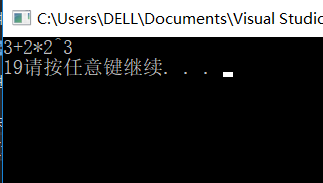
}

七、测试用例及结果说明

1）运算优先级的可靠性

输入：3+2\*2^3

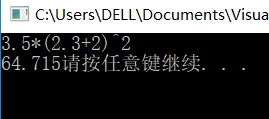
输出：19



2）括号及小数点的可靠性

输入：3.5\*(2.3+2)^2

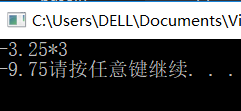
输出：64.715



3）首位为负号的可靠性

输入：-3.25\*3

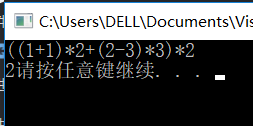
输出：-9.75



4) 多重括号的可靠性

输入：((1+1)\*2+(2-3)\*3)\*2

输出：2



上述测试用例说明本实验中实现了表达式的求值，并实现了括号、负号、小数点的运算。

八、设计及测试过程

本实验先是参考了《计算机软件技术基础》中对于表达式求解的算法，并加以改进，实现了更多的功能，本程序最开始使用了数组的数据结构，后改成了链表，这给递归的算法提供了便利。首先实现了基本的单个数字的运算优先级运算，然后进一步使多位数、负数、小数可以参与运算，最终使括号、多重括号得以实现。

九、评价与改进

本实验的优点是实现了比较多的功能，规避了一些可能会产生的错误，但是缺点也比较明显，算法方面还不够简介，代码上也过于重复，可读性较差。另外，本实验对于乱码的识别与处理上还可以加以考虑，使得程序的强壮性得以增强。

之后在本实验的基础上利用MFC做出了一个有界面的计算器，能够完成以上的运算。这个计算器的不足在于不能够使用键盘直接输入算式，而只能使用计算器的按键。另外，计算器不能显示中间结果，因为是使用的实验中程序的移植，不能够在用户输入的过程中显示结果，还可以改进。计算器的功能也不算完备，还有很多功能诸如平方、解方程等等待进一步开发。

十、心得体会

在做这个实验的过程就是一个不断尝试并否认错误的过程。本实验最开始是完全使用数组的，一直到做完了大部分工作到了括号才觉得数组不足以胜任递归。那么艰难的抉择就在于：是继续尝试使用数组还是全都改成链表。最终还是全盘来过，索性算法还是不变的，只是这种从数据结构全盘改掉的过程给了我一个深刻的教训。那就是一个程序的构思过程应该是完整的，而不是写点算点，写完一部分再去构思下一部分，很有可能在这之后遇到更大的困难。

我之前从未写过这么长的代码，也是第一次尝试使用多文件，更是首次学习与尝试了MFC制作应用界面，这个自己搜索资源加以筛选利用的过程给了我很多感触。最后，我想我永远不会忘记，在十一长假的某个晚上，当我做出了所有的这些工作，看着自己写的计算器按照自己想法完成各种“高难度动作”时，那份内心的喜悦。那是在冥思苦想了好多天之后的喜悦，是一行一行加断点改bug之后的喜悦，那是从走下显示器看着自己满是血丝的眼睛之后的快乐。诚然，写代码的过程是连续且艰苦的，并且不断地考验着程序员的耐心，随时都有一种摔掉电脑一走了之的冲动，但在一切都解决之后的那种快乐，却是任何娱乐都无法比及的。

我期待着下一次这样的快乐。

附：源程序

**main.cpp**

#include<iostream>

#include<cmath>

#include<cstring>

#include"calculate.h"

#include"base.h"

using namespace std;

#define N 100

node \*Change(char input[]);

/\*---------------------------------------------------------------

主函数

---------------------------------------------------------------\*/

int main()

{

double ans; //表示最终结果

char input[N];

node \*head;

cin>>input;

head = Change(input);//创建链表，并将input数组转化为链表，数值和运算符号分别存在不同的数据域

ans = Calculate(head);

cout<<ans;

system("pause");

return 0;

}

node \*Change(char input[])//创建链表，并将input数组转化为链表，数值和运算符号分别存在不同的数据域

{

node \*head, \*tail, \*newnode;

head = new node;//创建伪表头

tail = head;

int negative\_flag = 0;

if(input[0] == '-')//先判断首位符号的情况，若是负号则记录，最后吃掉符号

{

negative\_flag = 1;

Eat(input);//吃掉下一位

}

else if(input[0] == '+')

{

negative\_flag = 0;

Eat(input);

}

for(int i = 0; input[i] != '\0'; )

{

if(input[i] >= '0'&&input[i] <= '9')//把操作数存入结点

{

char w;

double r = 0; //r用来存储操作数，可能有很多位，需要判断

int dot\_flag = 0; //用来判断有无小数点，需要在循环外定义

int dot\_num = 0; //用来计小数位

while(input[0] >= '0'&&input[0] <= '9'||input[0] == '.')

{

if(input[0] >= '0'&&input[0] <= '9'&&dot\_flag == 0)//处理小数点前的数位

{

w = input[0]; //再读一位，作为个位

Eat(input);

r = r\*10+Exchange\_n(w);

}

else if(input[0] == '.')

{

w = input[0];

Eat(input); //吃掉小数点

dot\_flag = 1;

dot\_num = 1;

}

else if(input[0] >= '0'&&input[0] <= '9'&&dot\_flag == 1)//处理小数点后面的数位

{

w = input[0];

r = r+Exchange\_n(w)\*pow(0.1, dot\_num);

dot\_num++;

Eat(input);

}

}

if(negative\_flag == 1) //有负号时取反

{

r = -r;

negative\_flag = 0;

}

//如果下一位是运算符，将r存入结点

newnode = new node;

newnode -> number = r;

newnode -> operate = '0';//这样一来operate域里如果是0，就说明这个节点中存储的是数值

tail -> next = newnode;

tail = newnode;

}

else //如果是运算符号，将其存入结点

{

newnode = new node;

newnode -> operate = input[0];

tail -> next = newnode;

tail = newnode;

Eat(input);

}

}

//在最后加上分号，这样用户就不需要输入分号了

newnode = new node;

newnode -> operate = ';';

tail ->next = newnode;

tail = newnode;

tail -> next =NULL;

return head;

}

**Calculate.h**

struct node //定义带表头的链表

{

double number;//操作数

char operate;//运算符号

struct node \*next;

};

double Calculate(node \*head);

**Calculate.cpp**

#include<iostream>

#include<cmath>

#include<stack>

#include"base.h"

#include"calculate.h"

using namespace std;

int SearchFlag(node \*head, node \*\*pp, node \*\*qq);//搜索链表，看是否含有括号，若有括号，直接得出最内括号位置

double Calculate(node \*head)

{

node \*p = head -> next;

node \*q =p;

node \*(\*pp) =&p;

node \*(\*qq) =&q;

//先处理括号

while(SearchFlag(head, pp, qq) == 1)//此时p和q已经锁定了最内括号，其中q指向'('结点的前驱，p

{

node \*h, \*newnode, \*t;

p -> operate = ';';

t = p;

t = t -> next;

p -> next = NULL;

h = q -> next;//'('的结点相当于头结点

p = p -> next;

newnode = new node;

newnode -> number = Calculate(h);

newnode -> operate = '0';

q -> next = newnode;

newnode -> next = t;

}

stack<double> NS;//操作数组成的栈

stack<char> OS;//运算符号组成的栈

OS.push(';');//把分号压在最下面

int t = 0;//t=0时扫描下一个结点

double x,y,answer,m;

char e;

p = head;

while(t != 2)

{

if(t == 0)//p指向下一个结点

{

p = p -> next;

}

if(p -> operate == '0')//如果结点中存有数值

{

NS.push(p -> number);

}

else if(p -> operate != '0')//如果结点中存储的是符号

{

char w = p -> operate;

if(w == '+'||w == '-'||w == '\*'||w == '/'||w == '^'||w == ';')

{

char s = OS.top();

if(Exchange\_o(w) > Exchange\_o(s)) //以决定入栈还是运算

{ //若新符号优先，则入栈，继续读取下一位字符

OS.push(w);

t = 0;

}

else if(s == ';'&&w == ';') //若新符号为分号，此时处理结束，得到最终结果,并退出循环

{

answer = NS.top();

NS.pop();

return answer;

t = 2;

}

else //若旧符号优先或平权，则进行旧符号运算，并将结果压入NS栈

{

x = NS.top();

NS.pop();

y = NS.top();

NS.pop();

e = OS.top();

OS.pop();

m = Exp(x,y,e); //构成运算

NS.push(m);

t = 1; //继续处理之后的字符

}

}

}

}

}

int SearchFlag(node \*head, node \*\*pp, node \*\*qq)//搜索链表，看是否含有括号，若有括号，直接得出最内括号位置

{

int flag;

node \*p, \*q;

p = head -> next;

q = head;

while(p -> operate != ')'&&p -> next != NULL)

{

if(p -> operate == '(')

{

while(q -> next != p) //q应指向'('结点的前驱

{

q =q -> next;

}

}

p = p -> next;

}

if(p -> next == NULL&&head == q)

flag = 0;

else

flag = 1;

\*pp =p;

\*qq =q;

return flag;

}

**Base.h**

double Exchange\_n(char x); //将数字字符转化为数值

void Eat(char input[]); //吃掉下一位

int Exchange\_o(char x); //将运算符号字符转换为数值，用以比较运算级别

double Exp(double x, double y, char e); //得到一个运算

**Base.cpp**

#include<string>

double Exchange\_n(char x) //将数字字符转化为数值

{

int n;

n=x-48;

return n;

}

void Eat(char input[]) //吃掉下一位

{

for(int j = 0; j < strlen(input)-1; j++)

input[j]=input[j+1];

input[strlen(input)-1] = '\0';

}

int Exchange\_o(char x) //将运算符号字符转换为数值，用以比较运算级别

{

int n;

switch(x)

{

case ';':n=0;break;

case '+':case '-': n=1;break;

case '\*':case '/': n=2;break;

case '^': n=3;break;

default:n=-1;

}

return n;

}

double Exp(double x, double y, char e) //得到一个运算

{

switch(e)

{ //先弹出x，后弹出y，则运算时y在前

case '+':return (y+x);break;

case '-':return (y-x);break;

case '\*':return (y\*x);break;

case '/':return (y/x);break;

case '^':return pow(y,x);break;

}

}