

数据结构课程设计

——表达式转换

项目说明文档

作 者 姓 名： 韦世贸

学 号： 2351131

指 导 老 师： 软件学院

专 业： 软件工程

同济大学

Tongji University

1 介绍

1.1 背景介绍

算数表达式有前缀表示法，中缀表示法和后缀表示法等形式。日常使用的算术表达式是采用中缀表示法，即二元运算符位于两个运算数中间。但是在计算机中，使用后缀表示法更为普遍。

1.2 功能介绍

将输入的正确的中缀表达式正确的转换为后缀表达式，同时也增加了附加功能，计算该表达式的值。

2 设计

2.1 数据结构设计

该项目数据结构采用链式栈的形式（数据结构）进行算术表达式的存储，因为这样的数据结构方便与表达式计算的优先级相结合，可以省去很多不必要的操作，其次链式栈也相对于顺序栈方便不少。

2.2 类结构设计

就主要内容方面该程序包含了一个替代string类的mystring类，一个实现链式栈的模板类。可以用于表达式转换和计算表达式结果。

链式栈也定义了栈的基本操作：出栈，入栈，判断栈空，获取栈顶元素等操作。

2.3 成员与操作设计

字数原因，栈的部分就不做过多展示

迭代器类

class MyStringIterator //字符串迭代器类，可以使用下标访问元素，可以使用begin(),end()

{

public:

MyStringIterator(char\* ptr) : m\_ptr(ptr) {}

char& operator\*() const //通过迭代器访问和修改字符串中的字符

{

return \*m\_ptr;

}

MyStringIterator& operator++() //++前缀

{

++m\_ptr;

return \*this;

}

MyStringIterator operator++(int) //++后缀

{

MyStringIterator temp = \*this;

++m\_ptr;

return temp;

}

bool operator==(const MyStringIterator& other) const

{

return m\_ptr == other.m\_ptr;

}

bool operator!=(const MyStringIterator& other) const

{

return m\_ptr != other.m\_ptr;

}

char& operator[](size\_t index) const //下标访问

{

return \*(m\_ptr + index);

}

private:

char\* m\_ptr;

};

字符串类

class MyString

{

public:

MyString(const char\* cstr = 0); //默认构造函数

MyString(const MyString& str); //拷贝构造函数

~MyString() {}

MyString& operator=(const MyString& str); //拷贝赋值运算符重载

MyString(MyString&& str); //移动构造函数

MyString& operator=(MyString&& str); //移动赋值

MyString& operator+=(const MyString& other); //重载+=运算符 拼接string

MyString& operator+=(const char\* str); //拼接常量字符串

MyString& operator+=(const char ch); //拼接单个字符

char\* get\_c\_str()const

{

return m\_data;

}

size\_t get\_length()const

{

return strlen(get\_c\_str());

}

friend ostream& operator<<(ostream& os, const MyString& str); //重载流输出运算符

friend istream& operator>>(istream& is, MyString& str); //重载流输入运算符

MyStringIterator begin()

{

return MyStringIterator(m\_data);

}

MyStringIterator end()

{

return MyStringIterator(m\_data + strlen(m\_data));

}

char& operator[](size\_t index)

{

return m\_data[index];

}

const char& operator[](size\_t index) const

{

return m\_data[index];

}

private:

char\* m\_data;

};

3 实现

3.1 栈基本功能的实现

3.1.1 栈基本功能流程简介

栈的基本功能包括检查是否为空，弹出、压入栈，读取顶元素、以及置空栈的操作。

3.2 栈内外优先级功能的实现

3.2.1 栈内外优先级功能流程简介

因为不同的运算有不同的优先级所以需要创建栈内外的相同符号的不同优先级，从而进行优先级运算。在这里建立了一个option结构体来定义不同运算符的进栈出栈的优先级，同时定义‘#’运算符，这代表栈底元素和输入终止符

3.2.2 栈内外优先级功能核心代码

struct Option

{

char data; //只接收符号入栈

int icp; //栈外优先级

int isp; //栈内优先级

Option(char ch = ' ', int ISP = 0, int ICP = 0)

:data(ch), isp(ISP), icp(ICP)

{

switch (data)

{

//栈底符号和输入终止符号

case'#':

isp = 0;

icp = 0;

break;

//括号

case'(':

isp = 1;

icp = 6;

break;

case')':

isp = 6;

icp = 1;

break;

//高级运算符

case'\*':

case'/':

case'%':

isp = 5;

icp = isp - 1;

break;

//普通运算符

case'+':

case'-':

isp = 3;

icp = isp - 1;

default:

break;

}

}

};

3.3 优先级比较功能的实现

3.3.1 优先级比较功能流程简介

优先级比较功能是为了更好地进行优先级比较运算而设置的，可以用于表达式转换和表达式的计算，但两者逻辑之间有一点小差别

表达式转换的比较

比较规则十分简单：栈顶元素和遍历到的元素进行比较，若目前的元素的入栈优先级大于栈顶元素的出栈优先级，目前元素入栈；若栈顶元素出栈优先级大于目前元素入栈优先级，则让栈顶元素出栈直到目前的元素的入栈优先级大于栈顶元素的出栈优先级或者栈空。若目前的元素的入栈优先级等于栈顶元素的出栈优先级，目前的运算符只有括号和输入终止符，这种情况下直接退栈不输出即可。

for (int i = 0; i < inLen; i++) {

if (input[i] != ' ') { //没有读到空格，往后继续读入直至读到空格

temp += input[i];

continue;

}

else { //读到空格，执行入栈或者输出操作

//非运算符

if ((temp[0] >= '0' && temp[0] <= '9') || temp.get\_length() >= 2)

{

if (temp[0] == '+') //正数前自动省略正号

for (int i = 1; i < int(temp.get\_length()); i++)

output += temp[i];

else

output += temp;

if (i != inLen - 1)

output += " ";

}

else {

Option ch(temp[0]);

if (changeStack.isEmpty()) {

changeStack.Push(ch);

}

else {

Option top;

changeStack.getTop(top);

while (ch.icp < top.isp) //入栈优先级大于出栈优先级

{

Option out;

changeStack.Pop(out);

output += out.data;

if (i != inLen - 1)

output += " ";

changeStack.getTop(top);

}

if (ch.icp > top.isp) //入栈优先级大于出栈优先级

changeStack.Push(ch);

if (ch.icp == top.isp) //退栈不输出

changeStack.Pop(top);

}

}

temp = ""; //最后置空temp字符串

}

}

while (!changeStack.isEmpty()) { //遍历结束后若栈内还有元素，则清空栈

Option out;

changeStack.Pop(out);

if (out.data != '#') //栈底符号不输出

output += out.data;

if (!changeStack.isEmpty())

output += " ";

}

计算时的比较

MyString temp;

for (int i = 0; i < outLen; i++) {

if (output[i] != ' ') {

temp += output[i];

continue;

}

else {

//非运算符

if ((temp[0] >= '0' && temp[0] <= '9') || temp.get\_length() >= 2)

calStack.Push(to\_beNum(temp));

//运算符

else

DoOperator(temp);

temp = ""; //最后置空temp字符串

}

}

3.4运算功能的实现

3.4.1运算功能流程介绍

后缀表达式的计算依托栈来进行，依靠运算优先级进行入栈出栈，同时还需要提取两个操作数的功能和按运算符进行运算的功能，提取操作数还需要将字符串形式的数字转化为浮点数。

3.4.2 运算功能核心代码

double to\_beNum(MyString& change)

{

double value = 0, calDecimal = 10;

bool isDecimal = false; //判断此时的转换是否为小数位的转换

// 使用迭代器遍历字符串

for (MyStringIterator it = change.begin(); it != change.end(); ++it) {

if (\*it == '-')

continue;

if (\*it == '.') {

isDecimal = true;

continue;

}

if (isDecimal) {

value += ((\*it - '0') \* 1.0 / calDecimal);

calDecimal \*= 10;

}

else

value = value \* 10 + (\*it - '0') \* 1.0;

}

if (change[0] == '-')

value = -value;

return value;

}

void DoOperator(MyString &option)

{

double value = 0, left = 0, right = 0;

if (get2Append(left, right)) {

switch (option[0])

{

case'+':

value = left + right;

break;

case'-':

value = left - right;

break;

case'\*':

value = left \* right;

break;

case'/':

if (right == 0) {

cout << "除数不能为0！" << endl;

return;

}

value = left / right;

break;

default:

break;

}

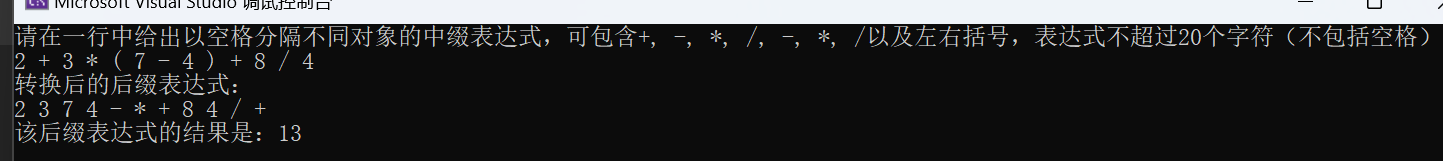
calStack.Push(value);

}

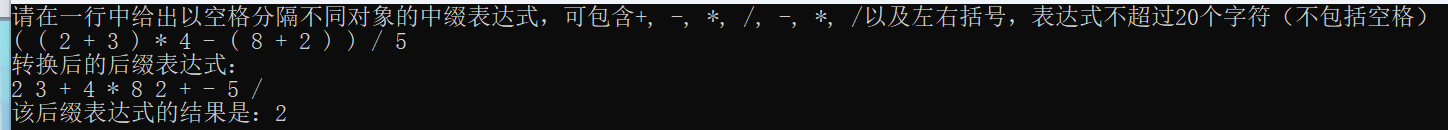
}

3.6 输出结果展示

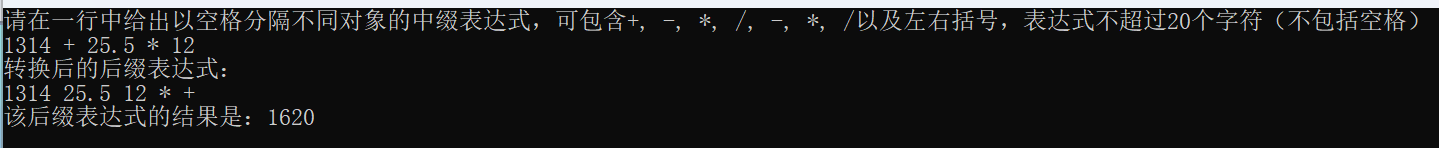
测试用例1：2 + 3 \* ( 7 - 4 ) + 8 / 4



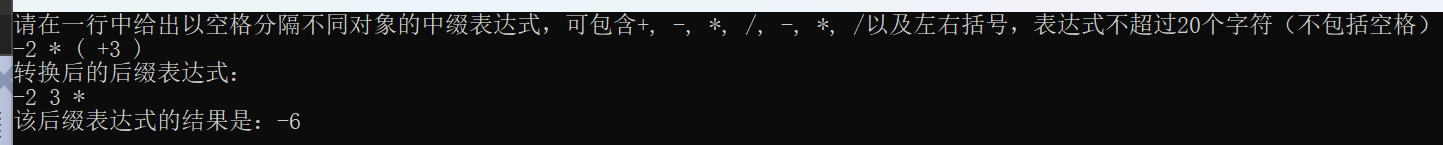
测试用例2：( ( 2 + 3 ) \* 4 - ( 8 + 2 ) ) / 5



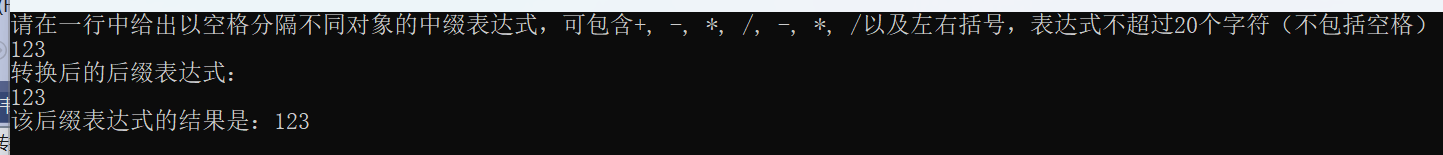
测试用例3：1314 + 25.5 \* 12



测试用例4：-2 \* ( +3 )



测试用例5：123



4 .设计小结

在这个表达式转换的程序中实现了一些基础功能，但是也有一些地方需要进行一定改进，虽然这个程序可以进行转换和计算并且无错误，但是对于一些极端情况的输入可能做的还不是很到位，而且计算模块只支持最普通的四则运算，想乘方这样的运算目前还没有完成，这也是之后需要考虑周全的地方。

但是通过这个程序我掌握了栈的基本创建方法，也学习到了链式栈以及顺序栈的优缺点，不仅如此，我还了解到了计算算术表达式的一般方法，总体来说还是比较有价值的。