中缀表达式转化为后缀表达式

用的是自己的顺序栈

#include<iostream>

#include<string>

#include<iterator>

using namespace std;

const int MAXSIZE = 10000;//最大的栈空间为1万

template<class ElemType>

class SqStack {

private:

ElemType\* top;//栈顶指针，在栈空的时候和栈底指针一起指向栈的第一个位置，栈为非空时指向栈顶元素的上面的一个位置

ElemType\* base;//栈底指针，始终指向栈的第一个位置，不管栈中有多少个元素

int maxSize;//用于标记栈中最多可以存放的元素数，同时也是判断栈满不满的标志之一，因为这是顺序栈，所以

//在使用它之前要先申请一块空间

public:

//构造函数:

SqStack() {

maxSize = MAXSIZE;

base = new ElemType[MAXSIZE];

top = base;

}

SqStack(int n) {

base = new ElemType[n];

top = base;

maxSize = n;

}

//析构函数:

~SqStack() {

StackDestory();

//cout << "调用了析构函数" << endl;

}

//拷贝构造函数:

SqStack(const SqStack& SqS) {

int L = SqS.getLength();

maxSize = SqS.max\_size();

base = new ElemType[maxSize];

top = base;

ElemType\* bs = SqS.getBase();

int i;

for (i = 0; i < L; ++i) {

\*(base + i) = \*(bs + i);

}

top = base + i;

}

//拷贝赋值运算符:

SqStack<ElemType>& operator=(const SqStack<ElemType>& SqS);

//销毁栈:

void StackDestory() {

top = base;

delete[] base;

}

//得到栈底指针:

ElemType\* getBase() const { return base; }

//得到栈顶指针:

ElemType\* getTop\_pointer()const { return top; }

//返回目前栈的元素个数:

int getLength() const;

//返回栈的最大空间:

int max\_size()const { return maxSize; }

//判断栈是否为空:

bool isEmpty() const;

//判断栈是否为满栈:

bool isFull()const;

//用e返回栈顶元素:

bool getTop(ElemType& e) const;

//入栈:

bool push(ElemType& e);

//出栈:

bool pop(ElemType& e);

//遍历栈:

void traverse()const;

// 栈空间加倍:

void doubleSpace();

};

//拷贝赋值运算符:

template<class ElemType>

SqStack<ElemType>& SqStack<ElemType>::operator=(const SqStack<ElemType>& SqS) {

const int L = SqS.getLength();

maxSize = SqS.max\_size();

base = new ElemType[maxSize];

top = base;

ElemType\* bs = SqS.getBase();

int i;

for (i = 0; i < L; ++i) {

\*(base + i) = \*(bs + i);

}

top = base + i;

return \*this;

}

//返回目前栈的元素个数:

template<class ElemType>

int SqStack<ElemType>::getLength() const {

return top - base;

}

//判断栈是否为空:

template<class ElemType>

bool SqStack<ElemType>::isEmpty() const {

if (base == top) return true;

else return false;

}

//判断栈是否为满栈:

template<class ElemType>

bool SqStack<ElemType>::isFull()const {

if (top - base == maxSize) { return true; }

else return false;

}

//用e返回栈顶元素:

template<class ElemType>

bool SqStack<ElemType>::getTop(ElemType& e) const {

if (base == top) { return false; }

else {

ElemType\* t = top;

e = \*(--t);

return true;

}

}

//入栈:

template<class ElemType>

bool SqStack<ElemType>::push(ElemType& e) {

if (top - base == maxSize) { return false; }

else {

\*top++ = e;

return true;

}

}

//出栈:

template<class ElemType>

bool SqStack<ElemType>::pop(ElemType& e) {

if (base == top) { return false; }

else {

e = \*(--top);

if (top - base == 0) {//为了使得当栈中的元素只有一个往出弹后，其实没有真正的弹出，只是指针移位，

//不过底部的指针是不会移位的，所以也就相当于这个值没有被弹出，这里选择采取将空间释放再

//重新申请空间的做法来实现这种特殊情况下的真正的弹出

delete[]base;

base = new ElemType[maxSize];

top = base;

//cout << "栈已被弹空" << endl;

}

return true;

}

}

template<class ElemType>

void SqStack<ElemType>::traverse()const {//遍历顺序栈

if (top == base) { return; }

else {

const int L = this->getLength();

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

cout << \*(base + i) << ' ';

}

}

}

// 栈空间加倍:

template<class ElemType>

void SqStack<ElemType>::doubleSpace() {

ElemType\* tmp\_base = base;

base = new ElemType[maxSize \* 2];

int L = this->getLength();

int i;

for (i = 0; i < L; ++i) {

\*(base + i) = \*(tmp\_base + i);

}

top = base + i;

maxSize \*= 2;

}

inline bool isOperator(char op) {//运算符辨识

if (op == '+' || op == '-' || op == '\*' || op == '/' || op == '(' || op == ')') {

return true;

}

else return false;

}

//运算符级别比较函数，若ope1更高级，返回1，若ope2更高级，则返回-1，若二者同级（不考虑先后顺序），则返回0

//且不考虑括号运算符

inline int operator\_cmp(const char& ope1, const char& ope2) {

if (ope1 == '+' || ope1 == '-') {

if (ope2 == '+' || ope2 == '-')return 0;

else return -1;

}

else if (ope1 == '\*' || ope1 == '/') {

if (ope2 == '+' || ope2 == '-')return 1;

else return 0;

}

}

template<class ElemType>//infix：中缀表达式字符串；suffix：后缀表达式字符串

void infix\_to\_suffix(SqStack<ElemType>& operator\_S, string& infix, string& suffix) {

string::iterator it;

int appear = 0;

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

if (\*it == ' ') continue;

else if (\*it >= '0' && \*it <= '9') {//数据直接放到后缀表达式字符串suffix中

suffix += \*it;

if (it < infix.end() - 1) {//表明迭代器还没有指向中缀表达式的最后一个数

if (\*(it + 1) == ' ' || isOperator(\*(it + 1)) == true) {//用来区分一个数是否结束（有可能是多个数），主要看原来的

suffix += ' ';//中缀式序列中其下一位是不是空格或运算符

}

}

else suffix += ' ';//因为可以知道中缀表达式infix的最后一个字符是一个数，而在后缀表达式中，数字绝对不是最后一个

//字符，所以如果遇到了中缀式的最后一个，那么将这个数字放到后缀字符串suffix后，必须在其后面输出一个字符串

/\*string::iterator k = it + 1;

while (k != infix.end() && (\*k <= '9' && \*k >= '0') ) {

suffix += \*k;

++k;

}

suffix += ' ';

it = k - 1;\*/

//if (appear == 0) appear = 1;//表明已经出现了一次数字

}

else {//表明这是运算符

//if (it != infix.end() - 1&&appear==1) suffix += ' ';

if (operator\_S.isEmpty()) {//空栈直接压入，注意此时操作的是运算符栈

operator\_S.push(\*it);

}

else {//表明栈非空，需要进行一些比较

char e;//一个用于辅助执行pop操作和放入suffix字符串中的函数

if (\*it == '(') {//右括号直接压入栈，会与一个左括号相呼应，括号规定了运算符的运算顺序，所以要将其压入运算符栈

operator\_S.push(\*it);

}

else if (\*it == ')') {//遇到左括号涉及pop操作，直到栈顶遇到右括号为止

operator\_S.pop(e);//遇到了左括号，则开始弹出数据，处理括号中的内容

while (e != '(') {//注意：括号不会出现在后缀式表达式中

suffix += e;//此时的e必然是一个运算符

suffix += ' ';

operator\_S.pop(e);

}

}

else {//此时遇到的运算符是属于加减乘除的任意一类，涉及到与运算符栈中的元素的比较操作

operator\_S.getTop(e);//取得此时的栈顶元素

if (e == '(') {

operator\_S.push(\*it);

}

else if (operator\_cmp(\*it, e) == 1) {//表明\*it的级别更高，\*it直接入栈

//要注意，不要忘了右括号这种情况

operator\_S.push(\*it);

}

else {//涉及pop操作，一直pop到\*it级别与栈顶运算符级别超过为止或是栈空为止

suffix += e;//把运算级别较高的率先放到后缀字符串中

suffix += ' ';

operator\_S.pop(e);

operator\_S.getTop(e);

if (e == '(') {

operator\_S.push(\*it);

continue;

}

else {

while (!operator\_S.isEmpty() && operator\_cmp(\*it, e) != 1) {//如果栈不空且\*it的运算级别小于e

operator\_S.pop(e);//那么就一直往下找，直到栈空或者找到了运算级低于\*it的e，我们就跳出这个循环

suffix += e;

suffix += ' ';

operator\_S.getTop(e);

}

operator\_S.push(\*it);//再把\*it压入栈

}

}

}

}

}

}

if (!operator\_S.isEmpty()) {//一般来说，当it遍历完原中缀表达式后，运算符栈中必然不空，这个时候就要对运算符栈进行弹出，依次接到suffix后面

char e;//与前面的e作用相同

while (!operator\_S.isEmpty()) {

operator\_S.pop(e);

suffix += e;

suffix += ' ';

}//这么操作会使最后一个接到suffix后面的运算符后会多接入一个空格

}

//cout << "此时suffix的长度为:" << suffix.size() << endl;

if (suffix[suffix.size() - 1] == ' ') {

suffix.erase(suffix.size() - 1, 1);//删除最后那个多余的空格

}

//cout << "最终suffix的长度为:" << suffix.size() << endl;

}

int main() {

string infix;//中缀表达式字符串

string suffix = "";//后缀表达式字符串

getline(cin, infix);//注意可能含有空格

//cout << infix << endl;

SqStack<char> operator\_S;//运算符栈

infix\_to\_suffix(operator\_S, infix, suffix);

cout << suffix;

return 0;

}