**研究性课题：中缀改后缀，后缀改中缀**

**余航**

**要求：**

1. P344-16

编写一个程序将一个后缀表达式转换成相应的完全括号化的中缀表达式。

要求：明确设计思路、画出完整流程图、运行结果正确

２．改进中缀转后缀算法，支持浮点数中缀表达式转后缀表

　　 示算法。

要求：在现有算法基础上修改，保持现有算法结构不变。

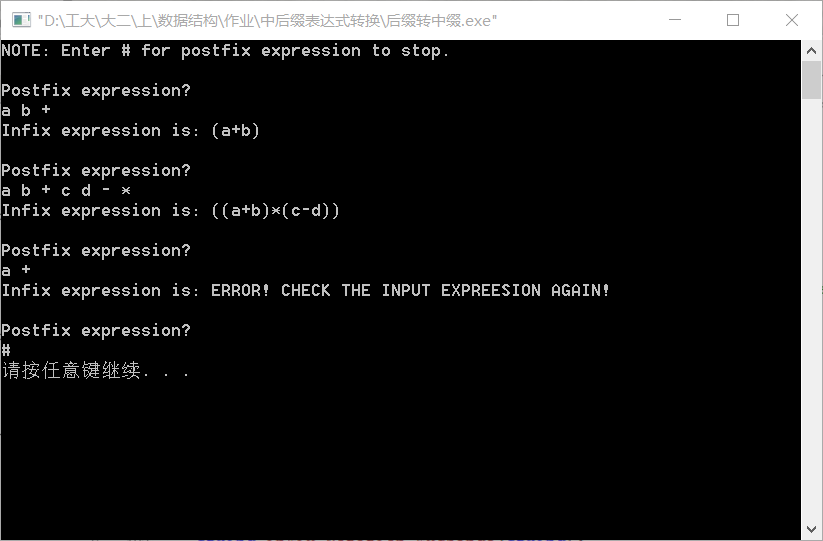
　　 明确设计思路、画出完整流程图、运行结果正确

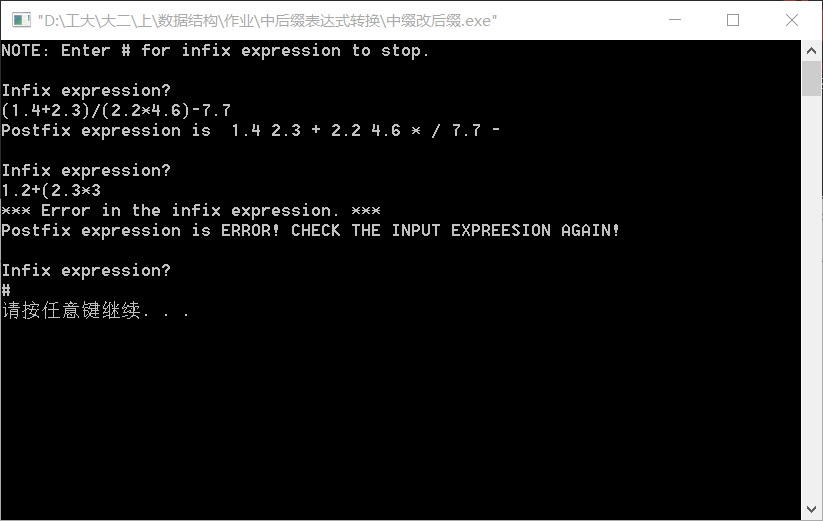
**分析：**

1. 后缀转换为中缀表达式，运用栈的方式，栈中储存的是表达式，遇到操作符则将栈中最上方的两个元素取出结合，并在两边加上括号，当所有的输入都取完时，在栈中应只剩下一个元素，这个元素便是最终的中缀表达式。
2. 中缀改后缀表达式的源代码在书中便有，要支持浮点数，重点能够在于读取到浮点“.”且能够将其在原来的整数部分和小数部分之间没有空隙的显示出来。

**运行截图：**

1、



2、

**源代码：**

1、

//Postfix expression converse to infix expression.

//Created by Yule, Nov\_14\_2015

//==============================================================================

// This program is to change the postfix expression into infix expression.

//

//CAUTION:

// The inputted expression should only include letter and operator(+,-,\*,/,%),

// no number or other symbol should appear in the expression.

// If happened, the output will report error.

//==============================================================================

#include<iostream>

#include<string>

using namespace std;

// Given the size of the stored stack for expression.

const int SIZE = 100;

// Create a Stack class for string.

class StrStack

{

public:

StrStack(){count = -1;} // Initialize the count -1 for convenience.

bool full() const;

bool empty() const;

void push(char sTemp);

void push(string sTemp);

string top();

bool sec\_empty() const;

string sec\_top();

void pop();

int size();

void display();// only use for test.

private:

// the stack data store place is based on a string array.

string StackString[SIZE];

// count the amouont of the elements.

int count;

};

//Function: Judge whether the stack is full.

bool StrStack::full() const

{

if (count == SIZE-1){

cout<<"The stack is full!"<<endl;

return 1;

}

return 0;

}

//Function: Judge whether the stack is empty.

bool StrStack::empty() const

{

if (count == -1){

cout<<"The stack is empty!"<<endl;

return 1;

}

return 0;

}

//Function: Element grabbed from the inserted line is in char type.

// push the gotten character into the stack.

void StrStack::push(char s)

{

if (!full()){

count += 1;

StackString[count] = s;

}

}

//Function: Element grabbed from following operating part is in string type.

// push the gotten string into the stack.

void StrStack::push(string s)

{

if (!full()){

count += 1;

StackString[count] = s;

}

}

//Function: Show the top element in the stack.

string StrStack::top()

{

if (!empty())

return StackString[count];

}

// Especially for changeing into infix expression,

// the operation will always handle two elements in the stack the same time.

// So need another judging function to judge wether could take two elements.

//Function: Judge wether the stack has two elements insides.

bool StrStack::sec\_empty() const

{

if (count == 0)

return 1;

return 0;

}

//Function: Get the element just below the top.

string StrStack::sec\_top()

{

if (!sec\_empty()){

return StackString[count-1];

}

}

//Function: Pop out the top element.

void StrStack::pop()

{

if (!empty()){

count -= 1;

}

}

//Function: Show the size of stored elements.

int StrStack::size()

{

return count+1;

}

//Function: Display all the elements in the stack one time.

// Only use for test.

void StrStack::display()

{

if (!empty()){

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

cout<<StackString[i]<<endl;

}

}

//Function: Operater function.

string inFix\_Notation\_Exchange(string exp)

{

char token;

StrStack opStack; // Create a string stack.

bool ju = 1;

int i = 0;

while(ju && i<exp.length()){

token = exp[i];

// Only push the letter into the stack.

if ((token>='A'&&token<='Z') || (token>='a'&&token<='z'))

opStack.push(token);

else{

// Other symbol go into switch sentence.

// Except +,-,\*,/,%, other symbol will report error.

switch(token)

{

case ' ' : break;

case '+' :

case '-' :

case '\*' :

case '/' :

// Judge wether there exists two elements,

// if so, link two elements with the operator and

// add brackets both sides. Then pop out two original elements,

// push the new string into the stack.

case '%' : if (!opStack.sec\_empty()){

string temp = '(' + opStack.sec\_top() + token + opStack.top() + ')';

opStack.pop();

opStack.pop();

opStack.push(temp);

}

else{

ju = 0;

}

break;

default : ju = 0;

}

}

i++;

}

// After all character has been token, there should be only one element in the stack,

// and that is the result to output.

if (opStack.size()==1 && ju){

return opStack.top();

}

else

return "ERROR! CHECK THE INPUT EXPREESION AGAIN!";

}

int main()

{

string inFix\_Notation\_Exchange(string);

string postfixExp;

cout<<"NOTE: Enter # for postfix expression to stop.\n";

while(1){

cout<<"\nPostfix expression? "<<endl;

getline(cin, postfixExp);

// Input "#" to get out.

if (postfixExp == "#")

break;

cout<<"Infix expression is: "<<inFix\_Notation\_Exchange(postfixExp)<<endl;

}

return 0;

}

2、

//Infix expression converse to postfix expression.

//Created by Yule, Nov\_10\_2015

//==============================================================================

#include<iostream>

#include<string>

#include<cassert>

#include<cctype>

#include<stack>

using namespace std;

// Function to operate the original infix expression to postfix expression.

// Parameter: infix expression. Output: postfix expression.

string postfix(string exp)

{

char token, topToken;

stack<int> opStack; // Create a stack to store the sign.

string postfixExp;

const string BLANK = " ";

// deal every character by order

for (int i=0; i<exp.length(); i++){

token = exp[i];

// Switch the sign, decide wether go to the stack.

switch(token)

{

// token is blank, ignore this.

case ' ' : break;

// token is '('

// Push it into the stack unconditionally.

case '(' : opStack.push(token);

break;

// token is ')'

// Judge wether there exist any sign in the stack initially,

// If the stack include some signs, pop out the signs continuously

// until meet a '('. Pop out this '(' without displaying it.

// If topToken didn't meet a '(' until the stack was empty,

// there must be a error.It will be discussed at the back.

case ')' : for (;;){

assert(!opStack.empty());

topToken = opStack.top();

opStack.pop();

if (topToken == '(')

break;

postfixExp.append(BLANK + topToken);

}

break;

// token is '\*', '/', '+' or '-'

// If the stack is empty, or the token has a higher priority than

// the top sign in the stack, push this token to the stack.

// If not, display the sign in the stack until fit the assumption.

case '+' :

case '-' :

case '\*' :

case '/' :

case '%' : for (;;){

if (opStack.empty() ||

opStack.top() == '(' ||

// '\*' and '/' have higher priority than '+' and '-'.

(token=='\*' || token=='/' || token=='&') &&

(opStack.top()=='+' || opStack.top()=='-')){

opStack.push(token);

break;

}

else{

topToken = opStack.top();

opStack.pop();

postfixExp.append(BLANK + topToken);

}

}

break;

// If token is not fit to conditions ahead, it must be a number.

// Append the number to the output string.

default : postfixExp.append(BLANK + token);

for (;;){

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// In the task require, this number may be a floating point number,

// so except the digit, character'.' should also include in the judgement sentence.

// It prevents the blank appear in front of the '.'.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

if (exp[i+1]!='.' && !isalnum(exp[i+1]))

break;

i++;

token = exp[i];

postfixExp.append(1, token);

}

}

}

// Judge wether there exists any sign in the stack.

for (;;){

if (opStack.empty()) // If there is no sign in the stack, get out.

break;

topToken = opStack.top();

opStack.pop();

// If the left sign is '(', the original expression must have some mistakes,

// for '(' and ')' must appear in pairs.

if (topToken != '(')

postfixExp.append(BLANK + topToken);

else{

cout<<"\*\*\* Error in the infix expression. \*\*\*\n";

return "ERROR! CHECK THE INPUT EXPREESION AGAIN!";

}

}

return postfixExp;

}

int main()

{

string postfix(string);

string infixExp;

cout<<"NOTE: Enter # for infix expression to stop.\n";

for (;;){

cout<<"\nInfix expression? "<<endl;

getline(cin, infixExp);

if (infixExp == "#") // Use '#' as the mark of exit.

break;

cout<<"Postfix expression is "<<postfix(infixExp)<<endl;

}

return 0;

}