项目说明文档

数据结构课程设计

——表达式转换

作 者 姓 名： 杨滕超

学 号： 2151298

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

目 录

[1 项目概述 4](#_Toc495668153)

[1.1 项目背景](#_Toc495668154) 4

[1.2 项目功能](#_Toc495668155) 4

[1.3 项目分析](#_Toc495668154) 4

[2 项目设计](#_Toc495668156) 4

[2.1 总体设计](#_Toc495668157) 4

[2.2 类设计](#_Toc495668158) 5

[2.2.1 List类](#_Toc495668163) 5

[2.2.2 Stack类](#_Toc495668163) 5

[2.2.3 String类](#_Toc495668163) 6

[3 项目实现](#_Toc495668161) 8

[3.1 总体实现](#_Toc495668162) 8

[3.1.1 总体实现流程图 8](#_Toc495668167)

[3.1.2 总体实现思路](#_Toc495668167) 8

[3.1.3 总体实现主要代码](#_Toc495668167) 8

[3.2 判断表达式合法函数 1](#_Toc495668166)0

[3.2.1 判断表达式合法函数流程图 1](#_Toc495668167)0

[3.2.2 判断表达式合法函数思路](#_Toc495668167) 10

[3.2.3 判断括号是否匹配思路](#_Toc495668168) 11

[3.2.4 判断括号是否匹配主要代码](#_Toc495668168) 11

[3.2.5 判断是否存在非法字符思路](#_Toc495668168) 11

[3.2.6 判断是否存在非法字符主要代码](#_Toc495668168) 11

[3.2.7 判断操作符冗余思路](#_Toc495668168) 12

[3.2.8 判断操作符冗余主要代码](#_Toc495668168) 12

[3.2.9 判断操作符缺少思路](#_Toc495668168) 12

[3.2.10 判断操作符缺少主要代码](#_Toc495668168) 12

[3.3 判断字符类型 1](#_Toc495668170)3

[3.3.1 判断字符类型思路](#_Toc495668171) 13

[3.3.2 判断字符类型主要代码 1](#_Toc495668171)3

[3.4 获得操作符的优先级 1](#_Toc495668174)4

[3.4.1 获得操作符的优先级思路 1](#_Toc495668175)4

[3.4.2 获得操作符的优先级主要代码 1](#_Toc495668176)4

[3.5 将中缀表达式转换为后缀表达式 1](#_Toc495668174)5

[3.5.1 将中缀表达式转换为后缀表达式流程图 1](#_Toc495668175)5

[3.5.2 将中缀表达式转换为后缀表达式思路 1](#_Toc495668175)5

[3.5.3 将中缀表达式转换为后缀表达式主要代码 1](#_Toc495668176)6

[3.6 计算后缀表达式的值](#_Toc495668174) 19

[3.6.1 计算后缀表达式的值流程图](#_Toc495668175) 19

[3.6.2 计算后缀表达式的值思路](#_Toc495668175) 19

[3.6.3 计算后缀表达式的值主要代码 2](#_Toc495668176)0

[4 项目测试](#_Toc495668161) 22

[4.1 输入测试 2](#_Toc495668174)2

[4.1.1 正确输入表达式](#_Toc495668176) 22

[4.1.2 过多空格](#_Toc495668176) 22

[4.1.3 输入为空 2](#_Toc495668176)2

[4.1.4 括号不匹配](#_Toc495668176) 22

[4.1.5 操作符确实](#_Toc495668176) 22

[4.1.6 操作符冗余](#_Toc495668176) 22

[4.1.7 非法字符 2](#_Toc495668176)2

[4.2 后缀表达式转换测试](#_Toc495668174) 23

[4.2.1 正整数运算](#_Toc495668176) 23

[4.2.2 负整数运算](#_Toc495668176) 23

[4.2.3 小数运算](#_Toc495668176) 23

[4.2.4 嵌套括号运算](#_Toc495668176) 23

[4.2.5 单目运算](#_Toc495668176) 23

1 项目概述

* 1. 项目背景

计算是计算机发明之初一个重要的功能，而中缀表达式是符合人类阅读习惯的一种表达式的形式，它容易被人类识别，但对于计算的工作并不友好，于是产生了波兰表达式和逆波兰表达式，即前缀表达式和后缀表达式，这是一种将运算符前置或后置的表达式，由此可以极大地方便计算机的计算。

随着科技的日益进步，人们对计算的要求与量级越来越高，从古时候的结绳计数到当今计算机超高量级的快速计算，无疑是计算机为人类减少了计算这种重复繁重的工作，使得人类拥有更多的时间投入与脑力思考中。

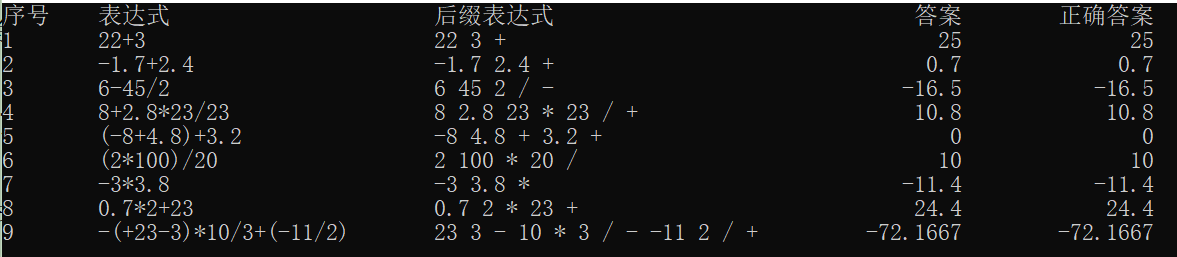
本项目正是从基本的表达式转换，将中缀表达式转为逆波兰表达式，为我们理解里计算机计算的过程打下基础。

* 1. 项目功能

用户输入想转换的中缀表达式，程序将检测表达式是否正确，若正确则输出相应的逆波兰表达式，否则输出提示表达式输入有误。

其中支持加减乘除+-\*/、括号()、单目运算符+-

程序示例：



* 1. 项目分析

1. 满足基本运算的要求，可以实现加减乘除、单目运算+-、括号的转换，并且包括多种数字的处理，其中有正数、小数、负数等等。
2. 当用户输入错误的表达式时，具有相应的错误提示而非程序崩溃。通过转换得到的逆波兰表达式计算其对应的值，若计算成功则输出转换结果，否则将输出错入输入的提示。
3. 利用数据结构Stack栈，实现较高效率的转换，对于比较复杂的表达式，能在尽可能小的时间复杂度转换出结果，时间复杂度为O(n)。

2 项目设计

2.1 总体设计

考虑到转换的算法需要用到先进先出的结构，我们可以想到使用栈，对于输入数据的不确定性以及栈满问题的考虑，我们决定使用链式栈，具有空间上的灵活性。

对于用户输入的数据，我们用String保存，将此转入转换函数，可以较为容易的使用。

对于操作符是否入栈或出栈，我们需要获得操作符的优先级，因此设置获得优先级的函数。

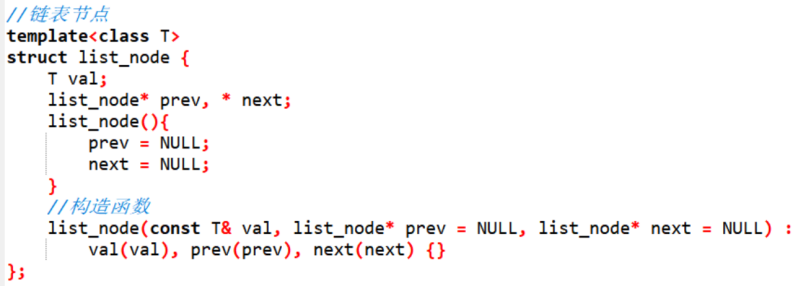
同时需要判断String中某个字符时数字或是操作符，因此需要该判断函数。

对于括号的输入，我们需要判断左右括号是否匹配，考虑到输入计算机的括号应为()圆括号，因此我们只对其考虑。

2.2 类设计

2.2.1 List类

·链表结点类设计：其中包括需要储存的数据，前驱节点和后继节点。如下图所示：



·链表类设计：数据成员包括，头结点，尾结点以及记录节点个数数据。

主要功能(与本题有关)包括：

·//返回链表大小inline int size() const;

·//判断链表是否为空inline bool empty() const;

·//清空链表void clear();

**·//最后添加元素void emplace\_back(const T& x)；**

**·//最后删除元素void pop\_back();**

**·//开头添加元素void emplace\_front(const T& x);**

**·//开头删除元素void pop\_front();**

·//插入void insert(const List<T>::list\_iterator& it, const T& x);

·//删除void remove(const List<T>::list\_iterator& it);

·//返回迭代器的查找typename List<T>::list\_iterator find(const T& x);

·//迭代器begin inline List<T>::list\_iterator begin();

·//迭代器end inline List<T>::list\_iterator end();

2.2.2 Stack类

栈是一种运算受限的线性表，它仅仅允许在栈顶进行元素的插入与删除操作，遵循先进后出的原则。在实现中，分为顺序栈和链式栈，考虑到顺序栈会出现栈满、申请空间未知导致的麻烦等缺点，我们的Stack类采用链式栈的方式构建。其中的有点包括申请空间灵活、不存在栈满的情况。

以链表为基础，我们构造链式栈。由于上述链表List类设置了头尾指针，并且是双链表，因此无论在链表的头或尾，增加或删除一个结点的时间复杂度均为O(1)，因此我们以链表尾作为栈顶。

该类主要成员函数如下：

//判断栈是否为空

inline bool empty() const {

return data.empty();

}

//获得栈中元素的个数

inline int size() const {

return data.Size;

}

//清空栈

void clear() {

data.clear();

}

//向栈顶添加元素

void emplace(const T& x) {

data.emplace\_back(x);

}

//删除栈顶元素

void pop() {

data.pop\_back();

}

//查询栈顶元素

T& top() {

return data.getTail()->val;

}

2.2.3 String类

字符串作为使用频率较高的一种数据结构，在程序发挥着巨大的作用。C++string类中配套的多种函数方便我们对字符串的处理，例如：比较、连接、替换、搜索等等。将其完全实现实现任务艰巨。因此我们实现其中的主要功能，以完成对主要的程序的要求。

该类主要成员函数如下：

·//空构造函数String();

·//赋值构造函数String(const String& str);

·//字符串构造函数String(const char\* str);

·//大小为size的字符串构造函数String(const char\* str, int size);

·//字符串长度int length()const;

·//重载=，StirngString& operator=(const String& str);

·//重载=，字符串String& operator=(const char\* str);

·//重载=，一个字符 String& operator=(const char& ch);

·//重载比较函数friend int strCmp(const String& str1, const String& str2);

·//重载< > <= >= friend bool operator<(const String& str1, const String& str2);

friend bool operator>(const String& str1, const String& str2);

friend bool operator<=(const String& str1, const String& str2);

friend bool operator>=(const String& str1, const String& str2);

·//重载[]inline char& operator[](const int index);

·//重载== bool operator==(const String& str);

bool operator==(const char\* str);

·//重载+ String operator+(const String& str);

String operator+(const char\* str);

·//重载+= String& operator+=(const String& str);

String& operator+=(const char\* str);

String& operator+=(const char ch);

·//返回字符串类型char\* c\_str();

·//模式匹配 int find(const char\* str);

int find(String& str);

·//删除所有的某个字符void erase(char ch);

·//翻转void reverse();

·//清空void clear();

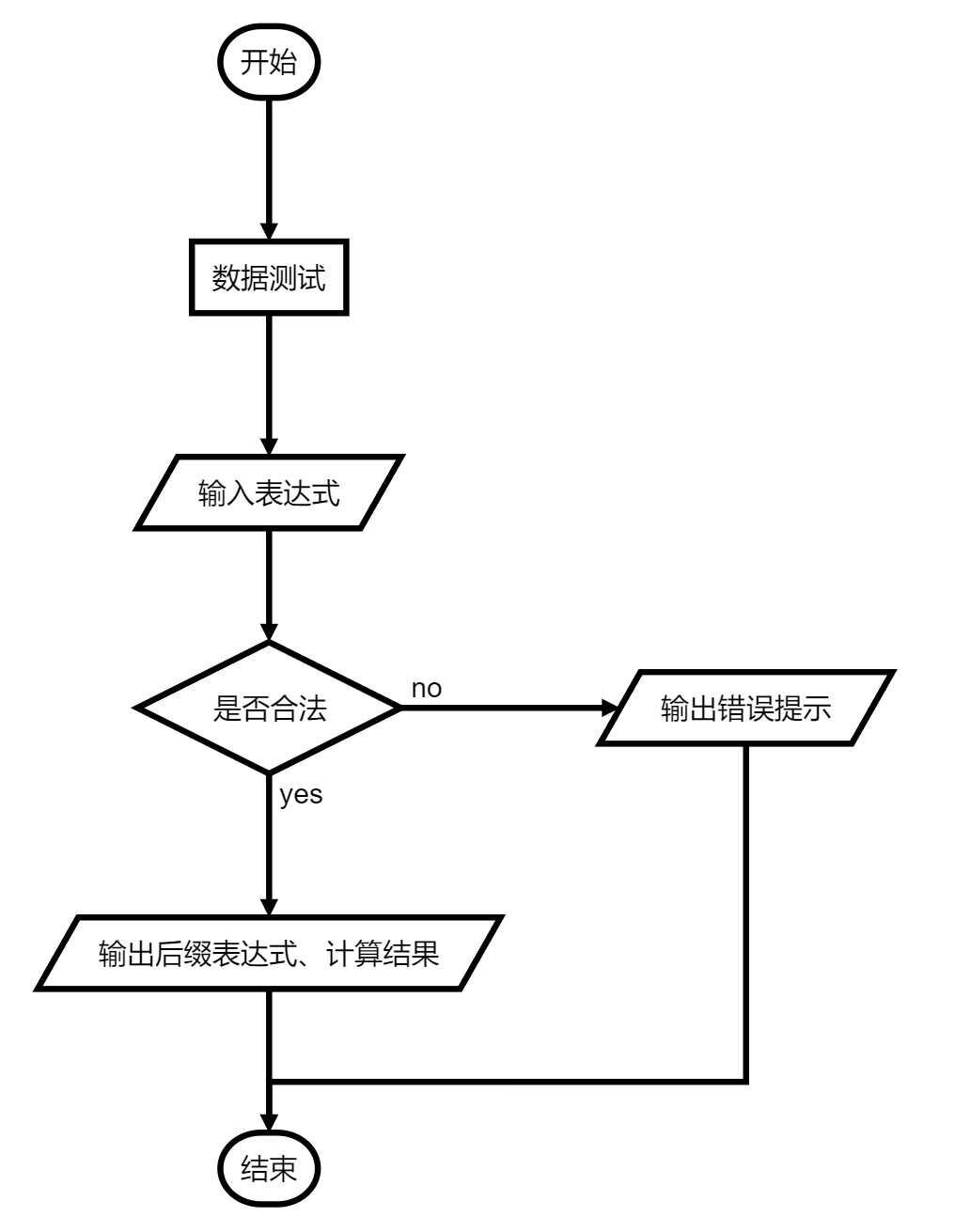
·//删掉最后一个void pop\_back();

·//获得最后一个字符char back() const;

3 项目实现

3.1 总体实现

3.1.1 总体实现流程图



3.1.2 总体实现思路

首先输出9个测试样例，等待用户输入表达式，判断表达式是否合法，若表达式非法，则输出输入错误提示，反之输出正确后缀表达式以及计算结果。

3.1.3 总体实现主要代码

//加减乘除小数负数测试

show("22+3", 25);

show("-1.7+2.4", 0.7);

show("6-45/2", -16.5);

show("8+2.8\*23/23", 10.8);

show("(-8+4.8)+3.2", 0);

show("(2\*100)/20", 10);

show("-3\*3.8", -11.4);

show("0.7\*2+23", 24.4);

show("-(+23-3)\*10/3+(-11/2)", -72.1667);

//自己表达式测试

cout << endl << "请输入表达式: " << endl;

char buffer[defaultSizeString] = { '\0' };

String temp;

//可用于空格输入

fgets(buffer, defaultSizeString, stdin);

temp = buffer;

temp.pop\_back();//去掉最后的\n

if (temp == "\0")

cout << "输入为空!" << endl;

else{

ans.clear();//清空之前

//得到返回值

int isValidRes = valid(temp);

if (isValidRes == 0){

convert(temp);//求后缀表达式

Pair<double, bool> res = cal(ans);

if (res.second == false)//计算不成功

cout << "表达式不规范，请重新输入" << endl;

else{

cout << "后缀表达式为：" << ans << endl;

cout << "计算结果为：" << res.first << endl;

}

}

else if (isValidRes == 1)

cout << "表达式不规范，左右括号不匹配，请重新输入!" << endl;

else if (isValidRes == 2)

cout << "表达式不规范，缺少操作符，请重新输入!" << endl;

else if (isValidRes == 3)

cout << "表达式不规范，存在非法字符，请重新输入!" << endl;

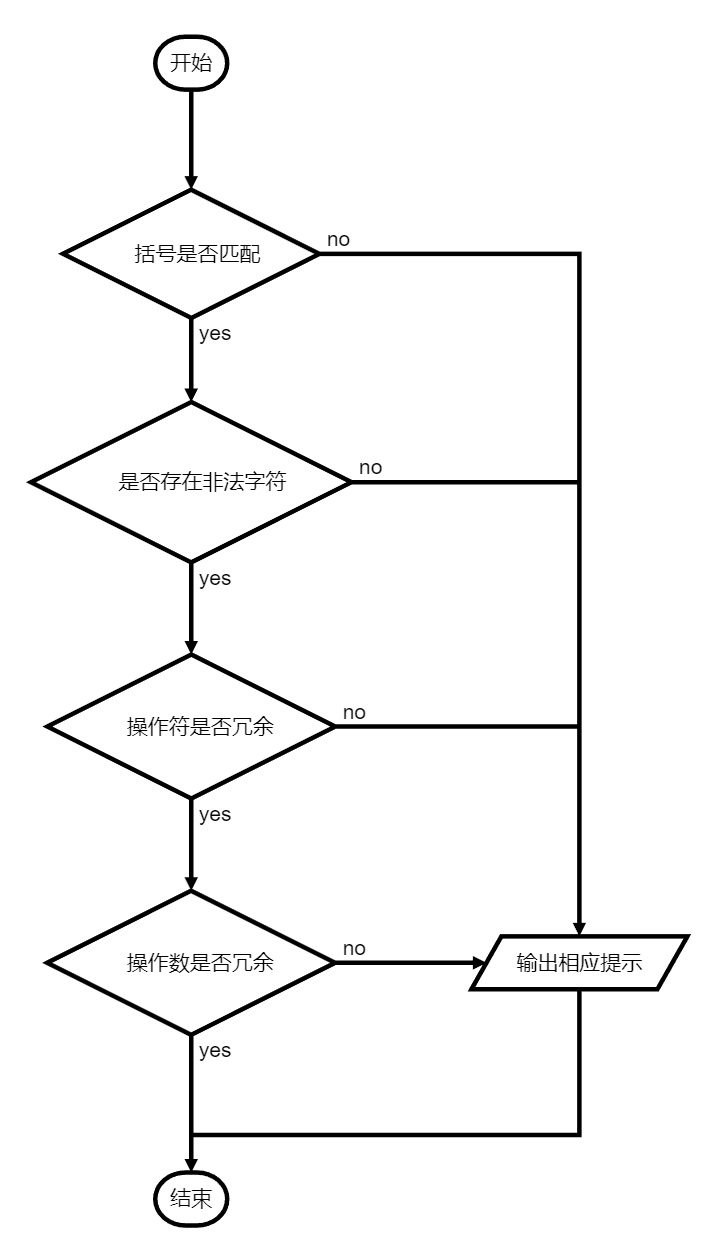
else if (isValidRes == 4)

cout << "表达式不规范，操作符冗余，请重新输入" << endl;

}

3.2 判断表达式合法函数

3.2.1 判断表达式合法函数流程图



3.2.2 判断表达式合法函数思路

功能：括号是否配对，操作符是否少了函数

返回值： 0表示没有问题，1括号不匹配，2缺少操作符，3存在非法字符，4操作符冗余

3.2.3 判断括号是否匹配思路

首先判断左右括号是否匹配，具体过程为：设置计数变量cnt初始值为0，遇到左括号计数变量++cnt，模拟左括号入栈个数，遇到右括号，首先判断cnt是否为零，即之前是否有左括号与之匹配，确定cnt > 0后，才--cnt，模拟左括号出栈，最后判断cnt是否为零，若为负数，说明右括号过多；若为正数，说明左括号过多；恰好等于零，说明左右括号匹配。

3.2.4 判断括号是否匹配主要代码

while (i < len){

if (str[i] == '(')//如果是左括号

++cnt;//模拟入栈

else if (str[i] == ')'){//右括号与之前的左括号配对

if (cnt == 0)//没有左括号与之匹配

return 1;

--cnt;//模拟出栈

}

++i;

}

3.2.5 判断是否存在非法字符思路

接着判断是否存在非法符号，除了小数点需要特殊判断，其他符号按照字符依次检查即可。关于小数点的判断，遇到小数点后，需要检查前一个与后一个字符为数字，才判断为合法。

3.2.6 判断是否存在非法字符主要代码

//检查是否有非法符号

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

if (str[i] == ' ')//空格ok

++i;

else if (isDigit(str[i]))//数字ok

++i;

else if (isOp(str[i]))//操作符也ok

++i;

else if (str[i] == '.' && i - 1 >= 0

&& isDigit(str[i - 1])

&& i + 1 < len && isDigit(str[i + 1]))//是小数点并且前后都有数字

++i;

else//不允许上述之外的东西存在了

return 3;

}

3.2.7 判断操作符冗余思路

之后判断操作符是否冗余。设置一个布尔变量，初始值为false。若遍历遇到+-\*/%，则检查布尔变量是否已经true，是则返回4，说明两个操作符之间没有数字，否则赋值为true，继续循环。

3.2.8 判断操作符冗余主要代码

bool isOperator = false;

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

if (str[i] == '+' || str[i] == '-'

|| str[i] == '/' || str[i] == '\*' || str[i] == '%'){

if (isOperator)

return 4;

isOperator = true;

}

else if (str[i] != ' ')

isOperator = false;

}

3.2.9 判断操作符缺少思路

最后判断是否出现操作符缺少的情况。我们利用bool变量标记上一个遍历到的是否为数字，当前时数字并且上一个(除去空格)仍然是数字，说明其间缺少操作符。

3.2.10 判断操作符缺少主要代码

bool isNum = false;

while (i < len){

//是数字或者是有符号的数字

if (isDigit(str[i]) || ((str[i] == '+' || str[i] == '-') && i + 1 < len && isDigit(str[i + 1]))){

if (str[i] == '+' || str[i] == '-')

++i;//跳到数字取

isNum = true;

++i;

}

else if (isOp(str[i]){)//是操作符标记为不是数字

isNum = false;

++i;

}

else if (str[i] == ' '){

if (isNum){//之前是数字

int next = i + 1;

for (; next < len; ++next){

if (isDigit(str[next]))//是数字，两个数字之间没有符号

return 2;

else if (isOp(str[next]))//是操作符跳出循环

break;

else//忽略空格，以及其他非+-\*/的符号

continue;

}

i = next;//跳到上述过程最后找的地方

}

else//空格前面不是数字跳过

++i;

}

else//其他跳过

++i;

}

3.3 判断字符类型函数

3.3.1 判断字符类型函数思路

需要判断遍历到的字符为数字还是操作符，因此设置函数判断。考虑到该函数简单常用，我们将其设置为内联函数。

3.3.2 判断字符类型函数主要代码

//判断是数字

inline bool isDigit(const char& ch){

return ch >= '0' && ch <= '9';

}

//判断是操作符

inline bool isOp(const char& ch) {

return ch == '(' || ch == '\*' || ch == '/'

|| ch == '%' || ch == '+' || ch == '-'

|| ch == ')';

}

3.4 获得操作符的优先级

3.4.1 获得操作符优先级思路

其中入栈之前，左括号最高，乘除模第二，加减次之，其他最低。

3.4.2 获得操作符优先级主要代码

//操作符优先级

inline int grade(String& ch) {

if (ch == "(")

return 5;

else if (ch == "\*" || ch == "/" || ch == "%")

return 4;

else if (ch == "+" || ch == "-")

return 3;

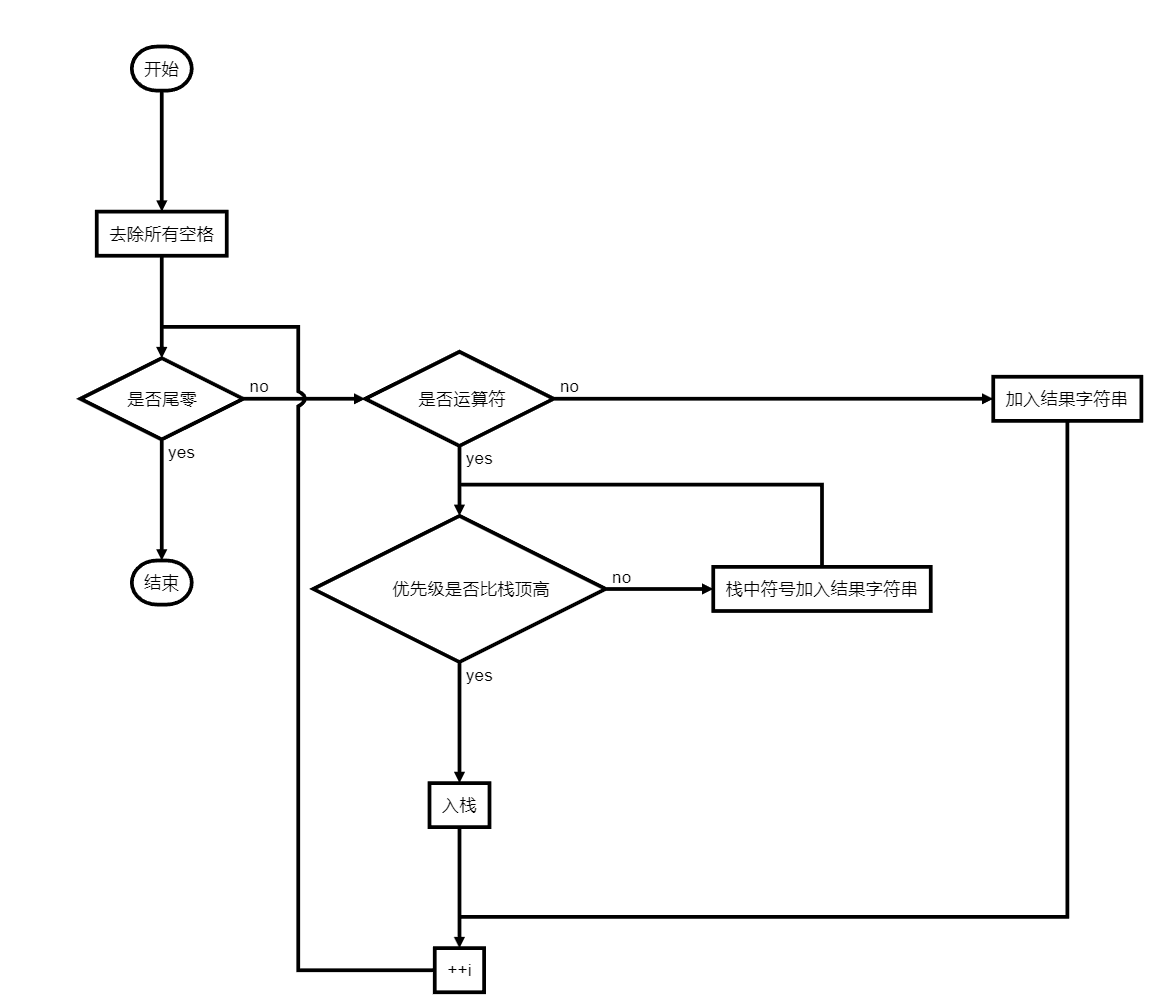
else

return 2;

}

3.5 将中缀表达式转换为后缀表达式

3.5.1 将中缀表达式转换为后缀表达式流程图



3.5.2 将中缀表达式转换为后缀表达式思路

这里需要两个栈，一个存放操作符，一个存放数字，根据优先级判断不断将操作符放入数字栈中，因此为了方便输出结果，数组栈用String代替，String同样具有back(查看最后一个字符)以及pop\_back(将最后一个字符删除)的两个函数，可以模拟栈的操作。

主要过程开始之前，我们将String中的空格全部删除，方便接下来的操作。这里的删除利用两个指针，遍历一遍即可完成删除所有的空格。

具体过程为：遍历中缀表达式的String类，遇到数字直接放入数字结果栈中，遇到操作符优先级高比操作符栈顶的操作符高的优先入栈，否则将栈中的操作符弹出加入数字栈中，知道当前遇到的操作符可以入栈为止。

其中的难点在于判断+与-究竟是数字前面表示正负的符号还是加减号还是单目运算符号。通过不断的探索与尝试，我们发现：-或+前面是右括号）或者数字时候，作为减号；后面是左括号（作为单目取反；其他情况则为负号。

另一个需要注意的地方为：在入栈之前左括号具有最高的优先级，而其入栈之后，优先级边的很低。

3.5.3 将中缀表达式转换为后缀表达式主要代码

//处理之前前去除所有空格

str.erase(' ');

while (i < len){

s = "";

//如果是操作符

if ((isOp(str[i]) && str[i] != '-' && str[i] != '+')

|| ((str[i] == '-' || str[i] == '+') && i - 1 >= 0 && str[i - 1] == ')')

|| ((str[i] == '-' || str[i] == '+') && i - 1 >= 0 && isDigit(str[i - 1]))

|| ((str[i] == '-' || str[i] == '+') && i + 1 < len && str[i + 1] == '('))//取反{

//栈中没有操作符直接放入

if (op.empty()){

s = str[i];

op.emplace(s);

}

else//栈中有操作符，进行比较

{

String Op1 = "", Op2 = "";

Op1 += str[i];

int grade1 = grade(Op1);

Op2 += op.top();

int grade2 = grade(Op2);

//栈中的左括号优先级最小

if (Op2 == "(")

grade2 = 0;

//左括号的直接入栈

if (Op1 == "(")

op.emplace(Op1);

//把括号中的符号放入num直到（

else if (Op1 == ")"){

while (!op.empty() && !(op.top() == "(")){

ans += op.top();

ans += ' ';

op.pop();

}

//弹出与之匹配的左括号

op.pop();

}

else{//加减乘除

while (!op.empty() && grade1 <= grade2){

ans += op.top();

ans += ' ';

op.pop();

if (op.empty())

break;

Op2 = "";

Op2 += op.top();

grade2 = grade(Op2);

if (Op2 == "(")

grade2 = 0;

}

op.emplace(Op1);

}

}

++i;

}

//如果是数字

else if (isDigit(str[i]) || str[i] == '-' || str[i] == '+'){

//如果是作为负号

if (str[i] == '-')

s += str[i++];

else if(str[i] == '+')//正号直接跳过

++i;

//把数字连起来

while (isDigit(str[i]) && i < len){

s += str[i];

++i;

}

//如果有小数点

if (str[i] == '.'){

s += str[i];

++i;

while (isDigit(str[i]) && i < len){

s += str[i];

++i;

}

}

ans += s;

ans += ' ';

}

else//其他跳过

++i;

}

//剩下操作符都加入

while (!op.empty()){

ans += op.top();

ans += ' ';

op.pop();

}

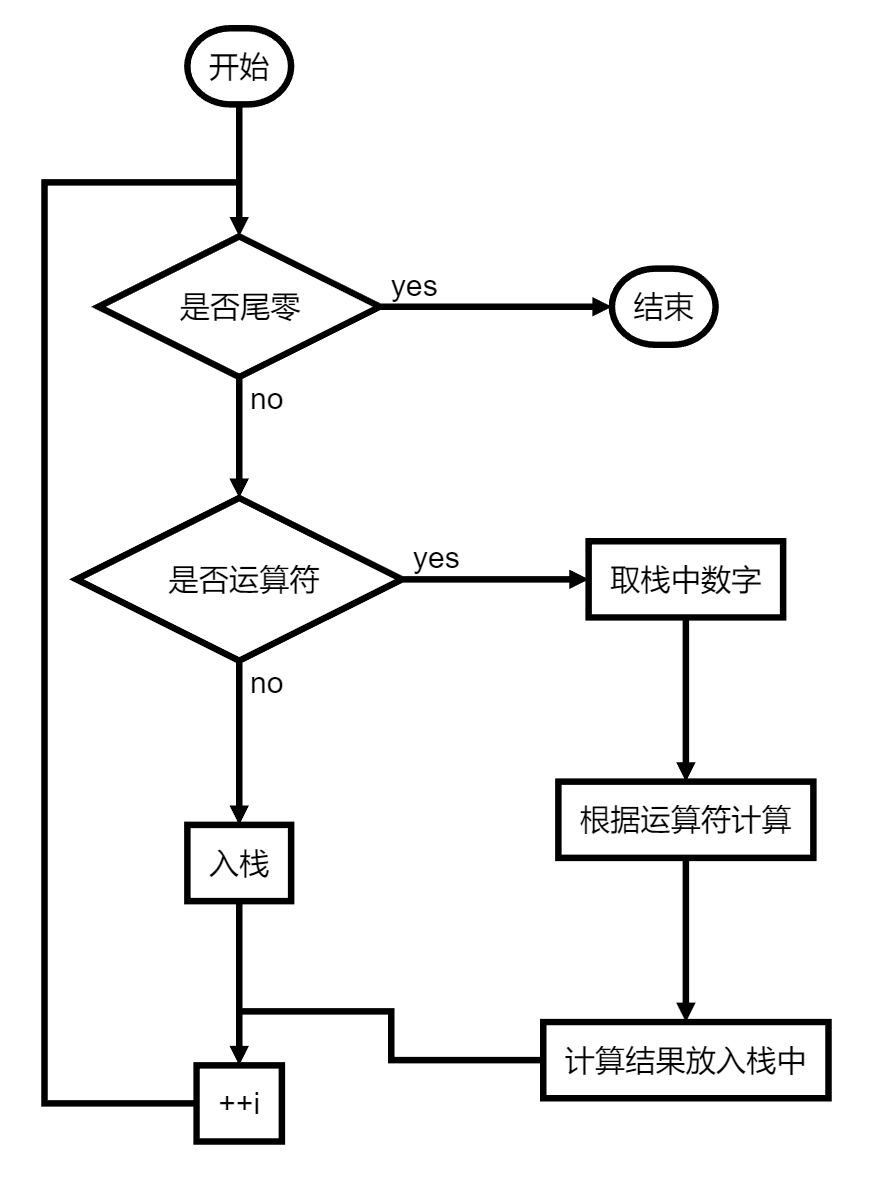
//删掉结尾空格

while (ans.back() == ' ')

ans.pop\_back();

3.6 计算后缀表达式的值

3.6.1 计算后缀表达式的值流程图



3.6.2 计算后缀表达式的值思路

需要一个栈辅助存放中间的运算结果。具体过程为：遇到数字入栈，遇到操作符则取出栈中的操作数，进行运算，再将运算结果放入栈中。

这时候需要判断需要操作数的个数，如果是遇到的操作符为-，并且栈中只剩下一个操作数，则说明为单目取反运算；若有两个操作数则依次取出并运算。这里对于减法与除法需要注意取出数字的次序。若栈中没有需要的操作数个数，则说明输入的表达式中操作符冗余，在此输出输入错误提示。

需要注意的一点是：如何将String遇到的数字转换为double型。需要考虑到对小数点以及负号的判断。这里对double的有效位数做出了限制，只取六位。

3.6.3 计算后缀表达式的值主要代码

while (i < len){

//有操作数才看符号，对负号还需要判断后面不是数字

if (!stk.empty() && (str[i] == '\*' || str[i] == '/'

|| str[i] == '+' || (str[i] == '-' && i + 1 < len && !isDigit(str[i + 1])))){

temp1 = stk.top();

stk.pop();

if (stk.empty()){//只剩下一个操作数

if (str[i] == '-' || str[i] == '+')//如果是单目 stk.emplace(-temp1);

else

return Pair<double, bool>(0, false);

}

else{//有两个操作数

temp2 = stk.top();

stk.pop();

switch (str[i]){

case '+':

stk.emplace(temp2 + temp1);

break;

case '-':

stk.emplace(temp2 - temp1);

break;

case '\*':

stk.emplace(temp2 \* temp1);

break;

case '/':

stk.emplace(temp2 / temp1);

break;

default:

break;

}

}

++i;

}

//是数字

else if (isDigit(str[i]) || (str[i] == '-' && i + 1 < len && isDigit(str[i + 1]))){

bool minus = false;

if (str[i] == '-'){//负数

minus = true;

++i;

}

double sum = 0, help = 0.1;

int cnt = 0;//记录位数

const int MAX\_N = 6;//有效数字取六位小数

while (isDigit(str[i])){

sum = sum \* 10 + str[i] - '0';

++i;

++cnt;

}

//如果有小数点

if (str[i] == '.'){

++i;//跳过这个小数点取后面的数字

while (isDigit(str[i]) && i < len && cnt <= MAX\_N){

sum += help \* ((double)str[i] - '0');

++i;

++cnt;

help /= 10;

}

}

if (minus)//有负号了就取反

sum = 0 - sum;

stk.emplace(sum);

}

//是空格

else

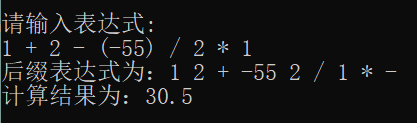
++i;

}

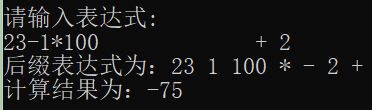
4 项目测试

4.1 输入测试

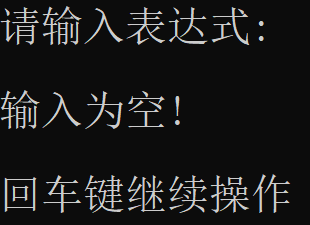
4.1.1 正确输入表达式



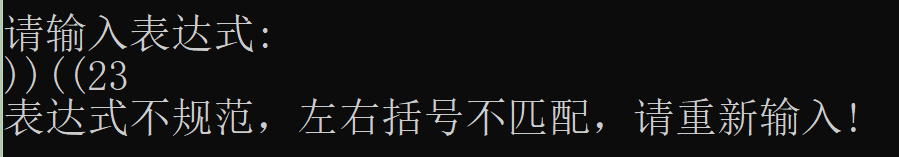
4.1.2 过多的空格



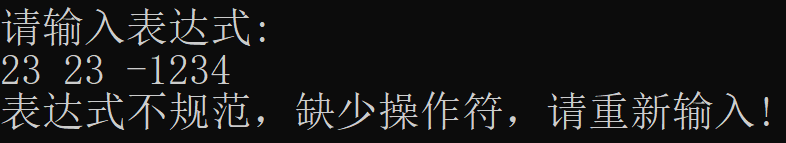
4.1.3 输入为空



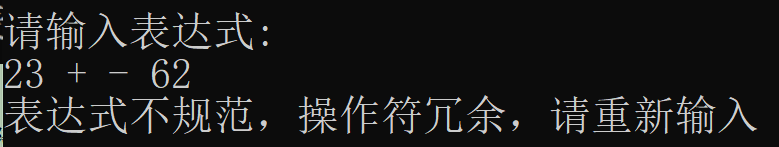
4.1.4 括号不匹配



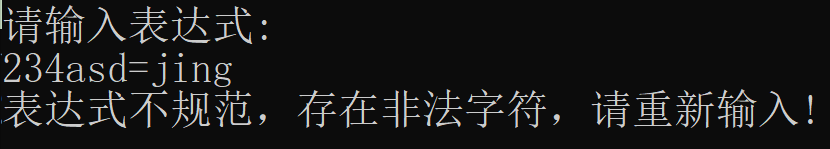
4.1.5 操作符缺失



4.1.6 操作符冗余

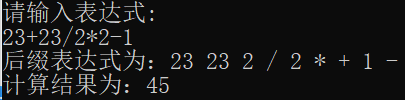


4.1.7 非法字符

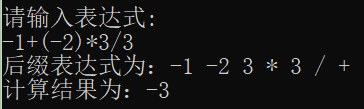


4.2 后缀表达式转换测试

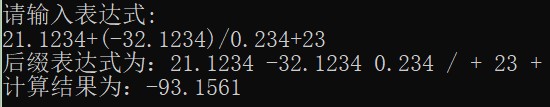
4.2.1 正整数运算



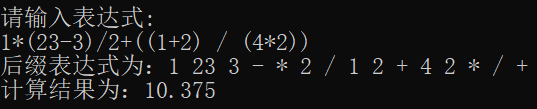
4.2.2 负整数运算



4.2.3 小数运算



4.2.4 嵌套括号运算



4.2.5 单目运算

