**数据结构课程设计**

**-算术表达式转换**

# 题目描述

项目说明：

算术表达式有前缀表示法、中缀表示法和后缀表示法等形式，其中中缀表达式为日常最常用的表达式，即二元运算符位于两个运算数中间。现需要对中缀表达式转换为后缀表达式。

格式要求：

1. 输入在一行中给出以空格分隔不同对象的中缀表达式，可包含+、-、\*、/以及左右括号，表达式不超过20个字符（不包括空格）。
2. 在一行中输出转换后的后缀表达式，要求不同对象（运算数，运算符号）之间以空格分隔，但是结尾不得有多余空格。

# 总体思路

从计算机本身处理算术表达式的方法出发，不同运算符均有不同的优先级，通过运算符栈和运算数栈不断对不同对象进栈、出栈实现运算。后缀表达式也可根据此思路得到以下规则：

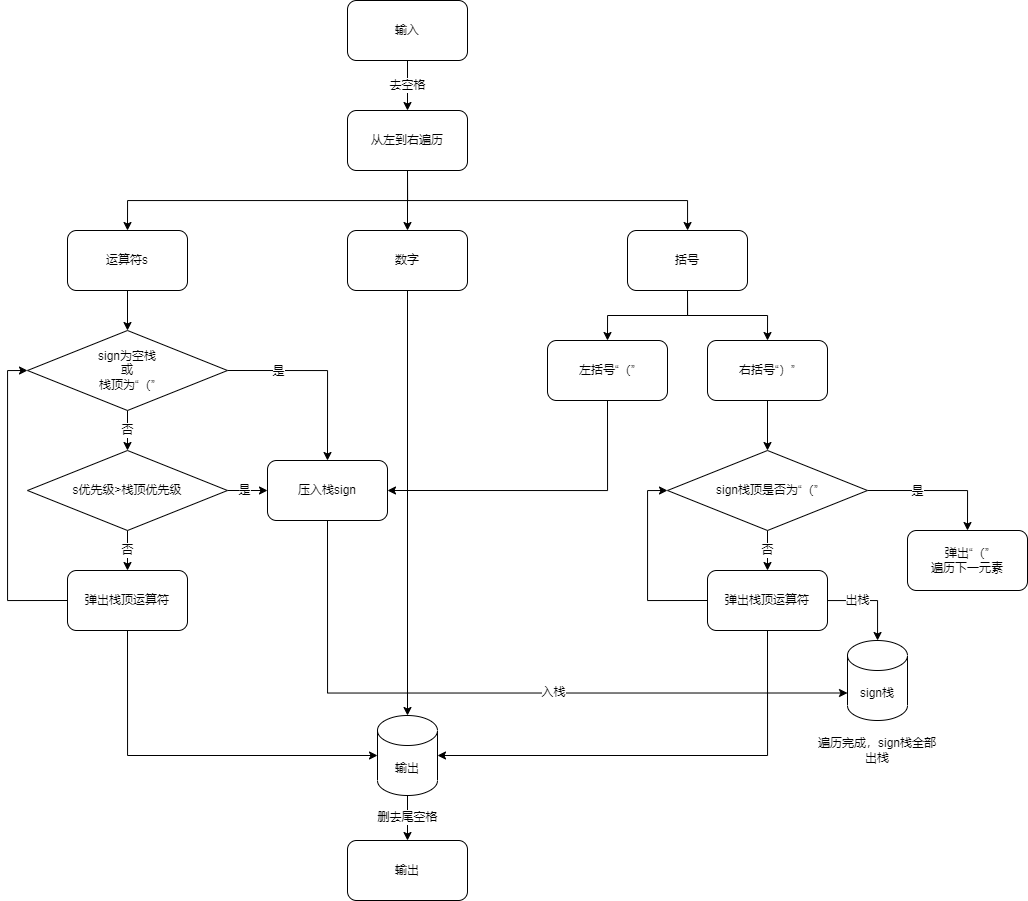
1. 规定“\*”和“/”的优先级最高，“+”和“-”的优先级其次，括号的优先级最小
2. 由于运算符均放在后面，遇到运算数直接输出
3. 左括号任何时间直接入栈；右括号出现，则出栈直到匹配到左括号
4. 运算符栈永远优先级高的压优先级低的
5. 若栈中仍有运算符，则全部出栈

# 类结构设计

mystack类，类似STL标准库中的vector

mystring类，类似STL标准库中的string

# 具体实现



## 输入处理

由于输入的算术表达式中含有空格，在后续处理时我们不希望有空格干扰，因此采取getchar()对单字符进行输入，遇到数字、小数点和六个运算符加入到string类型的变量s中，遇到回车则结束输入。

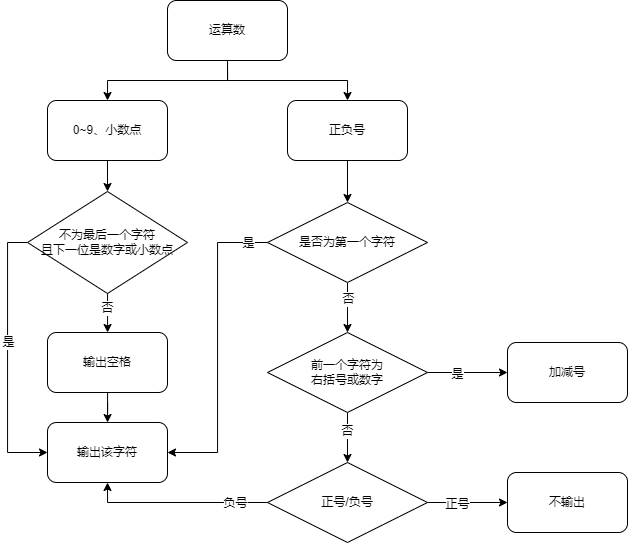
输入错误处理主要是以下两个方面：字符数小于20个和字符是否合法。若有错误输入则清空缓冲区和s及其他标记变量。

上述过程由函数void read(mystring& s)实现，s为无空格的输入内容

## 中缀表达式转后缀表达式

该功能由函数mystring trans(mystring &infix)实现

### 数字类直接输出

1. 若为无符号的数字，继续直接输出即可。
2. 若带有正号或负号，进行如下特判：第一个字符就为“+”或“-”，则它一定是正负号；若加减号前一个字符为数字或右括号，则它一定为减号，且仅有这两种情况。由以上可以决定数字类直接输出，注意判断出“+”为正号，不用输出。
3. 数字类结束的判断：没有到算术表达式末尾，并且下一个字符不是数字类字符，则说明该数字输出结束，并再输出一个空格作为表达式分隔符。

该部分代码为：

1. //遇到数字直接输出，包括负数
2. **if** (infix[i] >= '0' && infix[i] <= '9'|| infix[i] == '.'//无符号数字
3. || i == 0 && infix[i] == '-'  //第一个字符为减号的一定是负号
4. || i > 0 && infix[i] == '-'
5. && infix[i - 1] != ')'
6. && (infix[i - 1] < '0' || infix[i - 1]>'9'))
7. { //"-"当减号用时，仅有前面是数字和后括号两种情况
8. suffix += infix[i];
9. //没到算术式末尾，并且下一个字符不是数字类字符，说明数字输出结束
10. **if** (i != (**int**)infix.size() + 1
11. && !(infix[i + 1] >= '0'
12. && infix[i + 1] <= '9' || infix[i + 1] == '.'))
13. suffix += " ";
14. }
15. //"+"与"-"同理，不过不用输出
16. **else** **if** (i == 0 && infix[i] == '+'
17. || i > 0 && infix[i] == '+' && infix[i - 1] != ')'
18. && (infix[i - 1] < '0' || infix[i - 1]>'9')) {
19. **continue**;
20. }

### 运算符处理

#### 当前运算符为左括号或当前符号栈为空时，直接入栈。

#### 右括号出现时，不断出栈直到匹配到左括号，并不输出括号。

#### 当前遍历到的运算符，若该运算符优先级大于当前栈顶优先级，则直接入栈，否则出栈到满足该运算符小于等于栈顶优先级。

#### 全部遍历完成后，运算符栈剩余符号全部出栈。

该部分代码为：

1. **if** (operStack.isEmpty()|| infix[i] == '(') {
2. operStack.push\_back(infix[i]);
3. }
4. //后括号出现，出栈直到前括号出栈，配成一对括号
5. **else** **if** (infix[i]==')') {
6. **while** (operStack.top() != '(') {
7. suffix = suffix + operStack.top() + " ";
8. operStack.pop\_back();
9. }
10. operStack.pop\_back();
11. }
12. //永远是当前栈中优先级高的压优先级低的，否则就出栈到这种情况
13. **else** {
14. **while** (priority(operStack.top()) >= priority(infix[i])) {
15. suffix = suffix + operStack.top() + " ";
16. operStack.pop\_back();
17. **if** (operStack.isEmpty()) {
18. **break**;
19. }
20. }
21. operStack.push\_back(infix[i]);
22. }

## 输出空格处理

所有空格均在数字输出完成或运算符出栈时紧随其后输出，最后一个运算符或运算数后的空格通过erase函数删除最后。

# 项目小结

1. 调试过程中的问题

在实际写代码的过程中，我先行使用STL库的相关容器，即vector和string，完成主体逻辑的编写，后对这两个类进行自定义编写替换。

主体逻辑中其中较为困难的是对正负号功能的区分，需要总结多种出现情况，即正负号作为数字符号时，仅有前一个字符为数字和右空格两种情况。

自定义类编写中，我在完成整个mystring后完全忽略了string类型最后应该要有尾零做结尾，因此开辟char数组空间时少了一位导致调试时很长时间不知道错误所在。

1. 性能分析

理论上使用了两个栈，即符号栈和数字栈，但由于本题求得是后缀表达式，需要压入数字栈中的内容每次入栈，最终以逆序输出得到后缀表达式，本质上与要入栈时直接输出无区别，因此不用数字栈而只使用了一个符号栈，则空间复杂度为常数级O(1)。因为只对表达式字符串进行了一次遍历，因此时间复杂度为O(n)。此题规定了最多20个运算符和运算数，因此对string类中初始化字符数组定长41可以节约增大字符数组大小的时间开销。

因为转换表达式中有对数字正负号的特判，且输入时忽略空格输入，因此输入时正负号是否与数字相邻（符号与数字无空格）无关紧要，对输入更具包容性

# 程序功能测试

1. 纯数字输入：123



1. 括号嵌套测试：( ( 2 + 3) \* 4 - ( 8 + 2 ) ) / 5



1. “+”“-”的正负号判断：-2 \* (+3 / -5) \* (+5 - +6)



1. 多位数的数字运算式：1314 + 25.5 \*12



1. 错误输入：

abc

123a + 9

64 % 5

1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11

