10.4 编程实践：计算器

我们以计算器程序作为编程实践题目。希望通过这次编程实践，达到这样三个目的：第一、对本门课的知识和技能做全面的巩固和大幅度的提高；第二、掌握编写大型程序的基本技能；第三、为将来学习算法和数据结构打下坚实的基础。请读者务必高度重视这个编程实践！如果你能认认真真地按照要求一步一步地做下去，那么你的编程能力将会大大提高，你会发现本门课如此的简单，你一切都懂了！正所谓实践出真知。

请编写一个计算器程序，基本要求如下：用户从键盘输入一个表达式，程序以字符串形式接收、并计算表达式的值。构成表达式的合法字符包括阿拉伯数字、+(加)、−(减)、∗(乘)、/(除)、圆括号、正负号和小数点。

10.4.1 中缀表达式转后缀表达式

中缀、前缀和后缀表达式

表达式由操作数、运算符和分隔符组成，例如$(a+b)*c$，其中a、b、c是操作数，$+$ 和$*$ 是运算符，括号是用来改变优先级的分隔符。我们称它们为单词。本程序只涉及$+$、$-$、$*$、$/$四种运算，这些都是二元运算。也就是说，一个运算符带两个操作数。我们书写表达式时，一般把运算符放在两个操作数的中间，例如$a+b$，表示将a 和b 相加，我们称这种表达式为中缀表达式。虽然中缀表达式看起来十分自然，但是计算机处理起来并不方便。1929 年，波兰数学家Lukasiewicz 提出一种把运算符放在两个操作数前面的表示法，例如$+ab$，表示做加法操作，两个操作数是a和b。我们称这种表达式为前缀表达式，又称波兰式。如果把运算符写在两个操作数的后面，例如$ab+$，那么就得到一种后缀表达式，又称逆波兰式。

转换算法

由于计算机求后缀表达式的值要方便得多，这就涉及到中缀表达式到后缀表达式的转换问题。在转换过程中，需要使用一个栈，用来临时存放运算符和分隔符。转换步骤如下：

1. 从中缀表达式中取一个单词。
2. 如果是操作数，把这个操作数输出到后缀表达式中。

3. 如果是运算符，把它和栈顶元素的优先级作比较。如果它的优先级大于栈顶元素的优先级，那么就直接进栈；否则，就把栈顶元素弹出并输出到后缀表达式，重复这个操作，直到栈顶元素的优先级小于该运算符的优先级为止，然后该运算符进栈。
4. 如果是左括号，直接进栈。
5. 如果是右括号，弹出栈顶元素并依次输出到后缀表达式，直到弹出左括号为止，左括号不输出。
6. 重复执行1-5，直到把中缀表达式的所有单词处理完毕为止。
7. 把栈中的运算符依次弹出并输出到后缀表达式。

运算符的优先级

关于运算符的优先级，如表10.1所示，加减同级、乘除同级、乘除的优先级固然高于加减。此外，为了使程序设计方便，可以规定左括号的优先级低于加减；还可以增设一个栈底标志#，并规定其优先级为最低。

表 10.1 运算符的优先级

运算符	#	(	+	−	*	/
优先级	0	1	2	2	3	3

我们通常认为括号具有更高的优先级，那么这里的左括号的优先级为什么比加减还低呢？当左括号入栈后，应该先计算括号内的式子，因此栈内的其他运算符都不予考虑。这时栈顶的左括号把栈内其他运算符隔开，作为括号底部(左部)的标志，其作用类似于栈底标志#，此时尚不知右括号在哪里。中缀表达式中左括号后的第一个运算符必须保证进栈，而运算符进栈的条件是其优先级高于栈顶元素的优先级，因此我们规定栈内左括号的优先级低于加减运算符的优先级。事实上，规定左括号的优先级和增设栈底标志都不是必须的，这样做只是为了使程序设计更方便。只有当你亲手实现这个程序时才能明白这样做的好处。

转换实例

表10.2显示了中缀表达式转后缀表达式的过程。

表 10.2 $a+(b*c-d)/e$ 转后缀表达式

输入	栈(左为底)	输出后缀表达式	说 明
初始	#		设置栈底标志#，其优先级最低。
a	#	a	操作数直接输出。
$+$	# $+$	a	$+$的优先级高于#，直接入栈。
$($	# $+$ $($	a	左括号直接入栈。
b	# $+$ $($	$a\ b$	操作数直接输出。
$*$	# $+$ $($ $*$	$a\ b$	$*$的优先级高于$($，直接入栈。
c	# $+$ $($ $*$	$a\ b\ c$	操作数直接输出。
$-$	# $+$ $($ $-$	$a\ b\ c\ *$	弹出并输出$*$后，$-$入栈。
d	# $+$ $($ $-$	$a\ b\ c\ *\ d$	操作数直接输出。
$)$	# $+$	$a\ b\ c\ *\ d\ -$	弹出栈顶元素，直到弹出左括号。
$/$	# $+$ $/$	$a\ b\ c\ *\ d\ -$	$/$的优先级高于$+$，直接入栈。
e	# $+$ $/$	$a\ b\ c\ *\ d\ -\ e$	操作数直接输出。
结束	#	$a\ b\ c\ *\ d\ -\ e\ /\ +$	弹出所有运算符，并依次输出。

正负号的处理

我们既用±表示正负号，又用±表示加减运算符，这就要求程序必须能够区分。通过仔细观察，你会发现可以这样判定：

1. 如果±作为第一个字符出现，那么它一定是正负号。
2. 如果±紧跟在左括号的后面，那么它一定是正负号。
3. 除上述两种情况外，其他的±都是加减运算符。

那么怎样处理正负号呢？一个简单的办法是在正负号的前面加零，这样正负号就变成加减运算符了。例如：

$$-1.2*(+2.3) \implies 0-1.2*(0+2.3)$$

词法分析和算术表达式的转换

在上述对中缀表达式到后缀表达式转换算法的描述中，我们用一个字符表示一个单词，这显然是以词法分析为基础的。在实际的表达式中，不管是操作数还是运算符，都很可能是由多个字符组成的[6]。词法分析要做的事，就是把组成表达式的单词提取出来。中缀表达式是以字符串形式输入的，后缀表达式一般也是以字符串的形式输出的，因此词法分析提取的单词一般也是字符串形式。这里不详细介绍词法分析，请读者自己完成。

[6]例如C语言中的==、++和sizeof都是由多个字符组成的运算符。

本程序只涉及算术表达式，当把中缀形式的算术表达式转成后缀形式时，为了区分单词，应该加入自定义的分隔符。例如，$1.73 + 2.72 * 3.14$ 转后缀表达式时，如果不加分隔符，将输出

$1.732.723.14 * +$

而如果加入空格做分隔符，则输出

$1.73\ \ 2.72\ \ 3.14\ \ *\ \ +$

10.4.2 表达式求值

数字串转浮点数

表达式求值的最基本操作是把字符串形式的操作数转成浮点形式。对于单个数字字符，只要减去字符0的ASCII值即可得到相应的数值。而对于数字字符串，则需要把各位上的数字乘以权值再求和。例如，

$$\text{“123.45”} \Longrightarrow 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 4 \times 10^{-1} + 5 \times 10^{-2} = 123.45$$

当用程序实现时，一般从左到右扫描字符串，整数部分和小数部分的转换不便于统一，一般分别计算。转换算法如下：

```
ConvertToFloat(s)
 1  i=0;
 2  value=0;
 3  while s[i]!='\0' and s[i]!='.', do
 4      value=value*10+(s[i]-'0');
 5      i=i+1;
 6  if s[i]=='\0', then return value;
 7  i=i+1;
 8  weight=0.1;
 9  while s[i]!='\0', do
10      value=value+(s[i]-'0')*weight;
11      weight=weight/10;
12      i=i+1;
13  return value;
```

后缀表达式的计算

后缀表达式的求值算法如下：

1. 从后缀表达式读取一个单词。
2. 如果是操作数，把它转成浮点形式，然后进栈。

3. 如果是运算符，弹出两个操作数，做相应的运算，再把计算结果压入栈。
4. 重复1-3，直到把后缀表达式处理完毕。最后栈中只有一个数，即为表达式的值。

表10.3显示了后缀表达式“0.6 1.1 1.9 + 3.1 * +”的计算过程，它的中缀形式为“$0.6+(1.1+1.9)*3.1$”，栈中最后剩下的9.9即为表达式的值。

表 10.3 后缀表达式“0.6 1.1 1.9 + 3.1 * +”的计算过程

输入单词	栈(左边是栈底)	说 明
0.6	0.6	把字符串0.6转成浮点数，然后压入栈。
1.1	0.6 1.1	把字符串1.1转成浮点数，然后压入栈。
1.9	0.6 1.1 1.9	把字符串1.9转成浮点数，然后压入栈。
+	0.6 3.0	弹出1.9和1.1，相加，再把结果3.0压入栈。
3.1	0.6 3.0 3.1	把字符串3.1转成浮点数，然后压入栈。
*	0.6 9.3	弹出3.1和3.0，相乘，再把结果9.3压入栈。
+	9.9	弹出9.3和0.6，相加，再把结果9.9压入栈。

中缀表达式的直接计算

从上面的介绍可以看出，要计算一个中缀表达式的值，应该先转成后缀表达式，再计算后缀表达式的值。事实上，在把中缀表达式转后缀表达式的同时，可以完成计算，而不必输出后缀表达式。具体做法与上文所述基本相同，只是把这两步合成一步而已。下面给出简要的描述。

同样要使用两个栈，设用于存储操作数的栈为operandStack，用于存储运算符的栈为operatorStack。从左向右扫描中缀表达式，如遇操作数，压入operandStack；如遇运算符，压入operatorStack。当从operatorStack弹出运算符时，就从operandStack 弹出两个操作数，并做相应的运算，再把运算结果压入operandStack。这样就可以直接计算中缀表达式，而不必输出后缀表达式。

10.4.3 计算器程序的优化和增强

请读者在亲手实现上述计算器程序之后再来阅读这部分内容。

现在你已经实现了计算器的基本功能，然而在本书作者看来，尚不能据此给及格分数。本书作者在这里将带着你一步一步优化和增强你的计算器

程序，如果你认真去做了，那么你的编程水平将得到很大的提高。

规范化

检查你的源程序，是否做到了规范化？

1. 源程序是按照文件包含、常量定义、类型定义、函数声明、函数实现、主函数这样的顺序书写的吗？
2. 常量名是否全大写并用下划线连接单词？变量名是采用小驼峰表示法吗？自定义类型名是采用大驼峰表示法吗？函数名规范吗？这些名字能让其他程序员见名知义吗？
3. 在常量定义处、类型定义处、函数声明处、头文件中和一些不易懂的代码处是否有清晰的注释？
4. 该对齐的地方对齐了吗？该缩进的地方缩进了吗？花括号的位置正确吗？注意左花括号不另起行，右花括号收回缩进。一行上的代码超出页面了吗？如果是，要分多行书写。

这里只列出一些常见的格式问题，格式问题多种多样，不能一一赘述，请读者仔细检查。如果你的代码符合编程规范，那么你可以得60分了。

调试模式

为你的程序增加调试模式，参见10.1.2。

批量计算

请用你的计算器做批量计算。也就是说，用户把待计算的多个表达式输入到一个文件中，然后你的计算器对这个文件中的表达式逐个计算，并把计算结果输出到另一个文件中。这实际上是在考察你对文件的操作。

泛型栈

在计算表达式时，需要使用两个栈：一个是操作数栈，另一个是运算符栈。因为操作数栈存储的是浮点型元素，而运算符栈存储的是字符型元素，它们具有不同的数据类型，所以我们需要实现两个栈，而这两个栈的实现代码几乎是一样的，只是栈元素的类型不一样而已。请实现一个可以存储任意类型数据的泛型栈，参见8.5.2，并应用到你的计算器程序中。