

数据结构实验报告

实验名称： 实验 4——题目 2：表达式求值（科学计算器）

学生姓名：王思恢

班 级： 2016211110

班内序号： 03

学 号： 2016210271

日 期： 2018 年 1 月 7 日

1. 实验要求

本实验要求设计一个算术四则运算表达式求值的简单计算器，通过编写实现相应的功能，熟悉相关数据结构的函数和应用，增强综合运用数据结构知识解决实际问题的能力，初步了解和体会文本分析与编译的原理。

在本次实验中，我不满足于完成简单的四则运算表达式求值功能。相反地，我在思考这样一个问题：我们常见的数学表达式，无论多么复杂，其中一般来说都主要有两种“语法”成分，一种是操作符，一种是操作数。操作符根据其性质不同，可以分成单目、双目，可以分成中缀（如常见的算术运算符+、-、*、/）、前缀（如 $\sin(x)$ ）、后缀（如 $n!$ ），还可以分成整数参数的运算符（比如阶乘）和实数参数的运算符（比如 $\log(x)$ ），此外，不同的运算符还具有结合性、优先级等属性。那么，运用面向对象的程序设计思想，能否设计一些结构体或类，来一般性地刻画操作符的属性，规定一般意义上任意属性的操作符在表达式运算中的行为特点呢？这就是我在本次程序设计中要回答的核心问题。

在本次程序设计之前，我已经了解到，表达式的本质是一棵语法树，我们可以用栈的办法，相当于后根序遍历整棵语法树，就可以求出表达式的值，该方法的具体实现的代码在数据结构课程的实验指导书中已经详细给出。因而在本次程序设计中，我希望尝试一种不同的思路。我希望模仿人们在手工求解表达式时那种自然而然的做法，即从左到右地读取表达式，遇到括号时先把最内层括号中的值求出来，然后按照运算符的优先级次序，先把具有最高级优先级的阶乘运算值求出来，再把优先级次高的乘方运算值求出来，然后再依次对乘除运算、加减运算进行处理。在优先级相同的运算中，则一般按照从左到右的方式顺序求解。我在本次程序设计中，对于表达式求值过程的算法，就是以人们手工求解时的算法为蓝本设计的。

我为本次程序设计设定的小目标是实现一个简单的科学计算器，除加、减、乘、除外，它还应该能完成乘方、阶乘、取模等运算，它应该能处理整数对象和浮点数对象，并且具有一元和二元函数的计算功能。此外，该计算器应该具备初步的容错能力（比如把 .5 识别为 0.5， $5\sin(x)$ 识别为 $5*\sin(x)$ ），并且能够报告它在语法分析中发现的错误是什么。

在这样的思路和目标指引之下，我此次的程序设计可以大致分为三个层次：

1.底层的数据结构：定义了操作数与操作符的结构体 `term`，定义了函数的结构体 `function`，规定了操作符与函数可能具备的属性。

定义了表达式的结构体 `expression`，由 `term` 节点构成的链表 `termList` 和操作符的优先级队列 `OprQueue` 两部分组成，实际上规定了不同属性的操作符在表达式求值运算中的具体行为。

定义了函数名的匹配树 `matchTree`（本质是平衡二叉排序树），用于在字符串中识别函数名并给出相应的函数指针。

可以说，底层数据结构的设计，规定了一般的操作符与操作数的属性，以及它们的与表达式求值有关的方法。

2.文本分析：在实现底层数据结构的类设计得初见雏形时，我已经能够人为地在表达式 `expression` 的链表中增添操作数与操作符的节点，并且求出这个表达式的值了。该过程主要由结构体 `expression` 的 `push` 方法（本质上是 `termList` 的 `push` 方法和 `OprQueue` 的 `push` 方法，即链表的插入与队列的入队）和 `calculate` 方法（本质上是 `termList` 的 `mergeAdjacent` 方法和 `OprQueue` 的 `pop` 方法，即专为表达式求值设计的链表的临近节点合并的方法和优先级队列出队的方法）实现。而下一个层次的任务，就是理解数学表达式的某种模式规则或者说是语法结构，提供一种字符串的分析归约方法，把表示数学表达式的字符串“翻译”为用 `push`、`calculate`、`pop`、`mergeAdjacent` 的语言表达的对于底层数据结构的操作。

在本程序中，文本分析的工作主要由计算器 `calculator` 类的成员函数 `calculate` 实现。该函数的特点是，从左到右读取字符串，当识别到在特定上下文关系中的合法字符时，调用相关辅助函数，一次性地将表示一个语法单元的字符模式串归约到底。具体而言，分别由 `readNumber`、`readOperator`、`readFunction` 三个辅助函数来处理数字、符号和字母（其中 `readFunction` 由两个辅助函数 `match` 和 `readParameter` 来实现，分别用于匹配函数名和归约匹配参数）。遇到括号时，在 `calculate` 的主函数中通过对自身的递归调用实现了对括号的处理。

这样做的好处是，使得主要的功能函数 `calculate` 的结构比较清晰，和初步设计时写出的伪代码比较相似，可读性强。各子功能代码段相对独立成模块化，便于调试、修改和维护。

更重要的是，一次性将一个语法单元归约到底的做法使得我的设计思考难度大大下降，因为，当 `readFunction` 一次性将 `func(x,y)` 的字符串归约为数值，遇到括号时的递归调用一次性地将内层表达式归约为数值之后，我发现从本质上说表达式的字符串分析只需要处理四种合法字符，即数字（和`.'`）、字母（函数的起始字符）、符号和`'('`（递归开始的标志），而递归完毕得到的表达式将是一个只有操作符与操作数，没有函数与括号的特别容易求解的情形，这大大降低了我的设计难度。相比之下，我的最初一版的程序设计中规定了大量的状态，我需要根据前后两个甚至多个状态的变化情况来做出语法分析并进行相应的操作，不但费时费力，而且无法解决函数的参数本身为一个复杂的数学表达式的情况以及复合函数的情况下的表达式求值问题。对于我而言，想到一次性归约到底的方法，就为任意复杂的数学表达式的求值问题的求解扫清了障碍。

3.辅助信息的输出:

在将输入的字符串翻译为底层的类的具体行为的同时,我还希望字符串的分析能够为用户提供反馈信息,具体来说,是为用户提供调试信息,就像编译器分析用户的代码时,不会因为输入的错误而运行崩溃,反而能够为用户提供对于错误的分析信息那样。

在本程序中,错误信息的识别与输出的机制相对简单。具体而言,就是在 `calculator` 类中加入一个表示是否发生错误的 `err` 变量作为标记,再加入一个变量 `errCode` 来说明错误的类型代码。`calculator` 类中的数据成员 `errMsg` 储存了可能发生的各种错误对应的错误信息,发生错误时,只要根据 `errCode` 输出 `errMsg` 数组中对应的信息即可。

一般来说,输出错误的机制是:在主要功能函数 `calculate` 的读取字符串的 `while` 循环中,先根据字符类型决定跳转到哪个辅助函数进行语法归约,当相应的辅助函数在语法分析中发现错误时,将 `calculator` 类的 `err` 标记置为 `true`,根据错误类型给出 `errCode` 值并终止分析。在每次 `while` 循环的末尾, `calculate` 函数会检查 `err` 是否被置为 `true`,若是,则打印出当前遍历到的出错字符的位置信息、错误码 `errCode` 以及错误的具体信息 `errMsg[errCode]`。

在具体代码的实现中,错误信息的输出遇到的主要技术难题是对于函数嵌套调用和递归调用的处理。由于 `calculate` 函数打印的出错字符的位置信息是由遍历变量 `char ch[i]` 的 `i` 提供,当递归调用的内层 `calculate` 函数在循环结束时输出错误信息时,打印的将是子字符串中出错字符的位置信息。解决这个问题的方法是在 `calculate` 函数的参数表中加入 `offset` 参量,以表示子字符串在原字符串中的相对位置,从而计算出出错字符在原字符串中的位置信息。

另一个问题是,在嵌套调用和递归调用时,错误信息会反复输出。其中嵌套调用的情况尤其容易令人产生困惑。在本程序中,嵌套调用的情况发生在函数的参数归约过程中,即 `calculate` 调用 `readFunction`, `readFunction` 调用 `readParameter`, `readParameter` 将由括号或逗号限定的部分字符串构造成子字符串,再交给 `calculate` 函数去归约求值。此时,为避免反馈信息的错误以及重复出现,主要需要做两件事,一是在 `readFunction` 和 `readParameter` 函数中也增加 `offset` 参量,使得 `offset` 参量在 `readParameter` 和 `calculate` 函数之间能够双向传递,二是在 `readFunction` 函数的返回值上做文章,即凡是嵌套调用时发现错误的,返回值为 `-2`,表示嵌套函数已经打印过错误信息,原函数可以不必再次打印,而直接终止;凡是非嵌套调用时发现错误的,返回值为 `-1`,表示错误信息未打印,需要原函数进行处理。

总的来说,错误处理方面没有什么特别重要的设计思想,反而倒像是由一系列小的技巧组成的,然而正是这一部分,在后期调试过程中花费了我最多的时间。

为辅助错误处理,本程序中定义了一些轻量级的结构体用来表示函数运行的结果是否出错,其中包括结构体 `result`、`errorInfo` 等等。结构体 `result` 由 `value` 和 `error` 两个成员构成,分别表示求值结果和是否出错。

另外,在函数名匹配的函数 `match` 中,由于 `match` 的功能是在平衡二叉排序树中匹配字符串并返回包含对应函数指针的操作符 `term`,因而本程序中形式上地

写了一个形式为 `double Fail(double,double)` 的函数，当 `match` 函数返回的 `term` 的函数指针指向 `Fail` 函数时，表示匹配出错。

总的来说，本程序中的错误信息反馈方式比较的多变，可以说这是我在编程之初没有系统化设计，随意性较大造成的。

此外，本程序中用到的队列是自行定义实现的。在该队列的实现中，我尝试了这样一种想法，即队列一次性地申请一块连续的内存空间，当队列发生溢出时，再一次性申请一块内存空间，也就是将顺序表作为节点连成链表的想法。

在本次程序设计中，用到的主要的类与函数列举如下：

操作符与操作数的 `term` 结构体

名称	功能与备注
<code>union val</code>	用于储存整形、浮点型数值以及四种类型的函数指针，这样的写法是为了避免定义类的复杂的继承关系
<code>enum attribute</code>	表明 <code>term</code> 的类型信息，通过 <code>attribute</code> 的值来判断 <code>union val</code> 中哪个成员是有效值
<code>enum category</code>	有 5 个取值 <code>None</code> 、 <code>L</code> 、 <code>R</code> 、 <code>LR</code> 、 <code>RR</code> ，用于说明操作符与操作数怎样归并
<code>int position</code>	说明该语法单元在字符串中的定位，主要用于调试
<code>void print()</code>	打印该操作符或操作数的值，主要用于打印最终运算结果的数值。打印操作符的功能用于调试，也是为了使该结构体的设计更加完整，尚无实用意义

`termList` 结构体（以 `term` 为数据成员的双向链表）

名称	功能与备注
<code>void push(const term& x)</code>	将字符串分析归并得到的语法单元 <code>term</code> 插入到链表尾部，延长表达式
<code>errorInfo mergeAdjacent(termNode* t)</code>	根据 <code>termNode</code> 中数据成员 <code>term</code> 的 <code>category</code> 属性，确定操作符节点怎样与左边和（或）右边的操作数节点进行合

	并、求值
void print()	打印链表,即依次打印节点 term 的信息

优先级队列 OprQueue

名称	功能与备注
queue<termNode*>	OprQueue 由四个队列 queue<termNode*>组成,分别保存各级运算符的指针序列,利用 OprQueue 可以像“书签”一样找到下一个需要进行计算的操作符

结构体 expression

表达式结构体 expression 将 termList 与 OprQueue 封装在了一起。

名称	功能与备注
void push	该函数可以带各种参数,从 int, double, char 到函数指针 double(*) (double,double)等等均可。用于构造各种 term 对象,将其加入到 termList 中,同时,对于操作符,根据其运算优先级将其指针加入到对应的 OprQueue 的队列中
errorInfo calculate()	按照优先级从高到低依次读取 OprQueue 中的 zeroOrder、firstOrder、secondOrder、thirdOrder 队列,将队列元素出队(相当于同级运算从左到右依次进行),将出队的 termNode*指针作为参数传递给 termList 的 mergeAdjacent 函数,实现操作符与相邻操作数的归并求值
void print()	打印 termList,用于中间过程的调试或最终结果的输出

函数 function 结构体

名称	功能与备注
----	-------

char tag[nameLength]	存储函数名
term opr	用 term 对象储存对应的函数指针及函数的特征分类（attribute 和 category）

结构体 matchTree（以 function 为数据成员的平衡二叉排序树）：

名称	功能与备注
void createNode(treeNode*& t,function f[],int lower,int upper)	递归地根据排好序的数组元素构造平衡二叉树，用于创建关于函数名的平衡二叉排序树，便于函数名的搜索匹配

calculator 类

名称	功能与备注
calculator()	在构造函数中建立计算器所支持的函数的函数名的平衡排序二叉树
result calculate(char* ch,int offset=0)	分析字符串，求出结果值或返回错误
void readNumber(char* ch,int& i,expression& e,bool omit=false)	用于一次性归约整数和浮点数，将得到的值 push 到 expression e 中。omit=true 时处理用.5 表示 0.5 这样的省略情况
int readFunction(char* ch,int& i,expression& e,int offset)	一次性匹配归约 f(x,y,...)的模式串，将求出的值 push 到表达式 e 中。具体实现分为函数名匹配和参数归约两步，分别由辅助函数 match 和 readParameter 实现
void readOperator(char* ch,int& i,expression& e)	判断运算符和相邻字符的上下文关系是否合法，合法时将算符插入链表并加入队列
term match(char* ch,int& i)	在 matchTree 中搜索匹配函数名，返回包含相应函数指针的 term 对象
int readParameter(char* ch,int& i,expression& e,term* t,int begin,int offset)	检验参数数量与 term 对象属性所规定的是否一致，调用 calculate 函数对参数表达式求值，将参数值带入函数指针求出函数值

本程序的运行效果如下：

```
Scientific Calculator
Operators supported: + - * / ^ % ! ()
Functions supported: abs acos asin atan cos exp ln log pow sin sqrt tan
Enter '@' to quit this program.
5.24^(3log(2.1+(1.6+0.4)!*pow(sin(0.1),cos(sin(0.1))))(2.13/145.7)+tan(cos(sin(1)))/(log(sqrt(4!))))
-6.7777066
(sin(0.65)^2+cos(0.65)^2+2)!
-6
1/15*(exp(1/15)+exp(2/15)+exp(3/15)+exp(4/15)+exp(5/15)+exp(6/15)+exp(7/15)+exp(8/15)+exp(9/15)+exp(10/15)
+exp(11/15)+exp(12/15)+exp(13/15)+exp(14/15)+exp(15/15))
-1.776194243
sin(log(2))+cos(pow(log(1.5(3+2)(1.6+4.4)!),cos(1/3.14))+17%13)
-0.6494967973
(sin(asin(0.5))*2+5)!
-720
1/sqrt(2*3.14159)exp(-1.65^2/2)
-0.1022649678
sin(1/1)+sin(sin(1/2))+sin(sin(sin(1/3)))+sin(sin(sin(sin(1/4))))
-1.858703395
5.14%2
Position:1 Error Code:9
Function int (int,int) can not take a parameter of type double.
sin(1.2,2,3)
Position:10 Error Code:17
Too many parameters.
```

从图中可以看出，本程序能够解析各种复杂的表达式，并对支持的各种函数进行运算。可以看到，本程序能够容许一定程度的省略或者语法上的不规范。本程序能够支持某种形式的类型转化，但同时也能够严格地执行类型检查。本程序能够给出错误发生的正确位置和反馈信息。

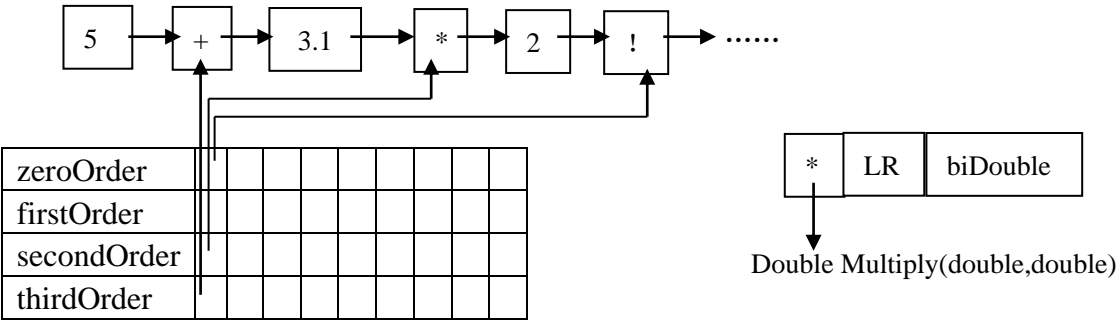
2. 程序分析

2.1 存储结构

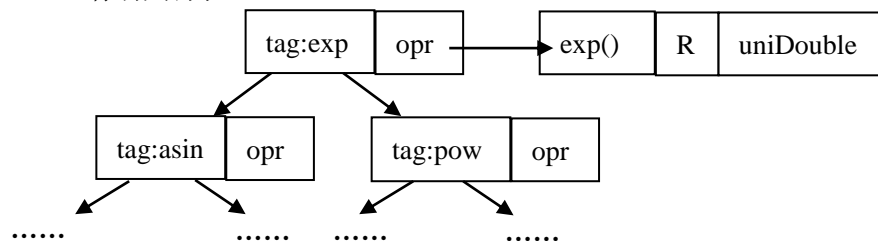
结构体 term 的存储结构：

union val	int ival	操作数
	double dval	
	double(*ptr1)(double,double)	操作符（函数指针）
	double(*ptr2)(double)	
	int(*ptr3)(int,int)	
	int(*ptr4)(int)	
attribute	Int,Float,biDouble,uniDouble,biInt,uniInt	
category	None,L,R,LR,RR	

结构体 expression 的存储结构：



结构体 matchTree 存储结构:



2.2 关键算法分析

1. 关键算法:

(1) 用 mergeAdjacent 函数实现 termList 中的操作符与邻近操作数节点的合并。

mergeAdjacent 函数有两个功能，一是对链表节点的删除和重新连接，二是对链表节点的值进行重新确定。其实现步骤大致如下:

①通过参数 termNode* t 读取对应操作符 term 的分类信息 category，根据 category 值为 L、R、LR、RR 分别跳到不同的分支处理。

②用指针 begin 和 end 指向删除节点后链表断点前后的两个待归并节点。用指针 del 和指针 del2 指向待删除节点。

③根据 category 值决定读取操作符节点左边或右边的操作数，进行类型检查，将操作数代入操作符 term 对象的函数指针中进行运算。

④检查运算结果与整数数值的差异是否在一定阈值以下，决定运算结果的类型，将结果值保存在 begin 节点中。

⑤删除中间节点，将 begin 与 end 相连，若 end 为空，重新设定 tail 指针的指向。

(2) 表达式 expression 的 push 与 calculate 方法。

push 方法的两个功能分别是，把操作符或操作数插入链表尾部，将操作符按照优先级加入队列。

其步骤为:

①根据待插入对象的类型，构造临时 term 对象，确定其 category 值和 attribute 值，对于操作符，赋予它相应的函数指针。

②将临时对象插入到链表中。

③对于操作符，将其根据优先级加入到相应的队列中。

calculate 方法的功能是，从优先级队列中读取操作符，用 mergeAdjacent 函数进行操作符与操作数的求值运算。

其步骤为:

①当第零级运算符队列不空时，令队列元素相继出队。由于读取字符串时是按照从左到右的顺序，即入队顺序为从左到右，所以现在出队的运算符也是按照从左到右的顺序。

②将出队 termNode* 指针作为参数传递给 mergeAdjacent 函数，完成运算。

③当第零级运算符队列为空时，相继读取第一级、第二级、第三级运算符队列，重复①、②的步骤。

(3) 构建排序二叉树，匹配函数名的方法。

用数组构建排序二叉树的步骤为:

①写好一个 function 数组，数组中 function 对象的 tag 即函数名按照字典序排好。这样，就相当于得到了二叉排序树的中根序遍历序列。

②通过二叉排序树 `matchTree` 的构造函数，调用辅助函数 `createNode`。

③对于给定的数组下界 `lower` 和数组上界 `upper`，取其中值 `mid`，用数组中坐标为 `mid` 的元素构造根节点。

④对于左子树，令下界为 `lower`，上界为 `mid-1`，对于右子树，令下界为 `mid+1`，上界为 `upper`，递归调用 `createNode` 函数，重复③的步骤。

由此得到的二叉树不仅是排序二叉树，而且是具有平衡性的平衡排序二叉树。

在 `match` 函数中，遍历字符串，匹配函数名的方法：

①设定二叉树节点指针指向二叉树的根节点，设定遍历坐标 `j=0`，即从节点 `tag` 的坐标 0 处开始匹配，记录原字符串起始比较位置坐标 `orig`。

②若原字符串 `ch[i]>tag[j]`，令二叉树节点指针指向原节点的右孩子节点，将 `i` 回溯为 `orig`，将 `j` 回溯为 0，重新开始字符串匹配过程。若原字符串 `ch[i]<tag[j]`，令二叉树节点指针指向原节点的左孩子节点，将 `i` 回溯为 `orig`，将 `j` 回溯为 0，重新开始字符串匹配过程。

③若二叉树节点指针值为 `NULL`，匹配失败，跳出循环。

④若 `ch[i]==tag[j]`，将 `i` 与 `j` 的值分别加 1，继续匹配过程。

⑤若原字符串的字母串部分刚好结束，`tag[j]` 在 `j++` 后刚好为 `'\0'`，说明匹配成功，用匹配成功的 `tag` 对应的 `function` 对象的函数指针及属性值构造新的操作符 `term` 对象，作为 `match` 函数的返回值。

⑥若匹配不成功，构造一个函数指针指向自定义的 `Fail` 函数的 `term` 对象作为返回值。

(4) `calculator` 类的 `calculate` 函数。

`calculate` 函数的实现步骤：

①读取字符串 `ch`，首先判断字符串 `ch` 的第一个字符是否为 `'\0'`，若是，返回 0 值。

②若不然，构造一个 `expression` 对象 `e`。顺序读取字符串 `ch` 中的字符。判断字符的类型，决定调用相应的归约函数，在传参时，把原字符串中的坐标信息 `i` 和 `expression` 对象 `e` 的引用传入，使得归约函数能够将归约结果插入 `expression` 的链表和队列。也使得归约函数能够控制对于字符串的遍历。

③对于不同类型的字符用不同的归约函数处理之后，坐标 `i` 值被归约函数更新，`err` 值与 `errCode` 值也得到了更新，检查 `err` 是否被置为 `true`，若是，返回一个 `error` 值为 `true` 的 `result` 对象，结束程序。

④若不然，检查 `i` 值被归约函数更新后，`ch[i]` 是否为 `'\0'`，若不然，调用下一个归约函数进行归约。可以看出，对于语法正确的简单（此处简单指不含括号即不需递归）表达式，`calculate` 函数的循环执行历程是：在每个语法单元子串的串首，由 `calculate` 判断字符类型，执行相应的归约函数，归约函数完成归约后，将 `i` 放置到下一个语法单元子串的串首，由此循环往复。

⑤`ch[i]=='\0'` 时，跳出循环，对表达式 `e` 执行 `calculate` 函数，然后打印出计算结果。

(5) 归约数字串的 `readNumber` 函数。

`readNumber` 函数的实现步骤：

①顺序读取数字串，将临时归约值保存在变量 `ival` 中，执行 `ival=ival*10+ch[i]-'0'`；

②遇到第一个非数字字符时，对其类型进行讨论。若为其他语法单元的合法

字符，退出 `readNumber` 函数返回 `calculate` 的主循环，对于省略乘号的情况，补充插入一个 '*' 运算符进入 `expression e` 的链表和队列。对于非法字符，置 `err` 值为 `true`，根据错误类型设定 `errCode` 值，退出 `readNumber` 函数。对于该字符为 '.' 的情况，继续检查 `ch[i+1]` 的字符类型。

③对于整数部分为 0 时省略的情况，应跳过①②直接从③开始。若小数点后第一个字符不是数字，设定错误码，返回 `calculate`。若不然，将小数部分值归约到变量 `dval` 中。

④数字串结束时，进行类似②中的语法验证，语法非法时设置错误码，合法时补足省略的算符，将整数部分值 `ival` 和小数部分值 `dval` 相加，用得到的值构造 `term` 对象，插入链表，返回 `calculate`。

(6) 归约运算符的 `readOperator` 函数。

`readOperator` 的步骤如下：

①先检查该运算符是否出现在表达式的首位。若首位出现运算符，只有 '+' 与 '-' 为合法，其余返回错误。对于 '+'、'-' 分别检验其后面的字符是否属于合法的语法单元，若不然返回错误。对于合法字符，对于 '+' 可不做任何处理直接跳过，对于 '-' 本程序在其前面先插入一个操作数数值为 0 的节点，再插入该 '-' 节点。

②对于不是出现在表达式首位的运算符，根据其是否是 '!' 进行分类讨论。由于 '!' 和其左边的数值进行归并，可以认为 '!' 是一个数字，这就允许其后面为空，或者出现数字（需补足省略的 '*'）、字母、运算符、括号四种情况。当 '!' 后面出现运算符时，本程序要求该运算符不能为 '!'。对于其他算符，还要再读取下一位字符，看是不是和运算符连接的许可语法单元。由于除数字之外，括号和函数都会优先归并为数值，所以运算符右边出现数字、括号、函数语法单元中任何一个的许可首字符都是可以接受的。

③对于不是出现在表达式首位的运算符，若其不是 '!'，则其后面必须为数字性质的语法单元，且不能为空。检验运算符后面的字符是否符合要求，决定是否返回错误码。

(7) 归约函数的 `readFunction` 函数。

`readFunction` 函数的步骤如下：

①对于字母串，执行 (3) 中过程，对字母串进行匹配。

②若匹配失败，终止语法分析，输出错误。若匹配成功，保存匹配得到的函数指针。进入参数匹配过程。

③根据匹配得到的函数的性质，记录函数要求的参数数目。建立实际读取字符串时读到的参数的计数器，建立左括号和右括号的计数器。当右括号数目多于左括号时，参数归并过程结束。当左右括号数相等且读到分隔符 ',' 时，令参数计数器加 1。把括号与括号或括号与逗号间的子串传递给 `calculate` 进行求值，将所求参数值传递给②中得到的函数指针，将函数部分归约为一个数值。

(8) `calculate` 函数遇到括号时的递归调用。

①建立左右括号计数器，当右括号计数刚好比左括号计数多 1 时，构建从左括号到此处的子串，递归调用 `calculate`，以求得内层括号中表达式的值。

②若提前遇到字符串终止，设置错误码。若递归调用的 `calculate` 函数返回的 `result` 对象的 `error` 属性为 `true`，构建一个 `error` 属性为 `true` 的 `result` 对象作为返回值，依次结束上一层 `calculate` 的执行。

2. 代码详细分析：

(1) 用 `mergeAdjacent` 函数实现 `termList` 中的操作符与邻近操作数节点的合

并。

`mergeAdjacent` 是本程序中代码长度最长的单一函数。其思想是简单的，其代码的复杂性主要来自分类讨论以及对于相关数据结构的操作的复杂。

①通过参数 `termNode* t` 读取对应操作符 `term` 的分类信息 `category`，根据 `category` 值为 L、R、LR、RR 分别跳到不同的分支处理。

```
errorInfo mergeAdjacent(termNode* t){
    termNode* begin;//删除节点后的断点起点节点指针
    termNode* end; //删除节点后的断点终点节点指针
    termNode* del;//待删除的左边指针
    termNode* del2=NULL; //待删除的右边指针，只删除一个节点时，不对
其进行操作
    double val_double;//用来保存归并后的浮点数值
    int val_int; //用来保存归并后的整数值
    int i1,i2;//用来保存整形参数的数值
    double d1,d2;//用来保存浮点型参数的数值
    //以上完成了对于各辅助变量的初始化
    switch((t->data).category){//对于读入参数t对应的操作符term的属性进行
讨论
```

②用指针 `begin` 和 `end` 指向删除节点后链表断点前后的两个待归并节点。用指针 `del` 和指针 `del2` 指向待删除节点。

下面以 `category` 值为 LR 的情况为例来说明问题。

case `term::LR`://`category` 值为 LR 时，需要将操作符与其左右两边的数值节点合并为一个节点

```
begin=t->prev;
end=t->next->next;//设置右边节点的下一个节点为断点后的终端节点
del=t;//设置待删节点指针
del2=t->next;
```

③根据 `category` 值决定读取操作符节点左边或右边的操作数，进行类型检查，将操作数代入操作符 `term` 对象的函数指针中进行运算。

if((t->data).attribute==`term::biDouble`){//对于 `double(double,double)` 类型的函数，对参数进行类型检查

```
if(begin->data.attribute==term::Int&&del2->data.attribute==term::Int){
    i1=begin->data.val.ival;
    i2=del2->data.val.ival;//左右两个数值节点都为整形变量
    val_double=((t->data).val.ptr1)(i1,i2);//运用操作符 t 对应的函数指针对参
数进行计算
}
else if(begin->data.attribute==term::Int&&del2->data.attribute==term::Float){//
分别对于两个参数为整形、浮点型的四种情况进行讨论
    i1=begin->data.val.ival;
```

```

        d1=del2->data.val.dval;//根据类型情况读取 union 中的 ival 值或 dval 值
        val_double=((t->data).val.ptr1)(i1,d1);//代入函数指针进行计算
    }
    else if(
    .....
    else{
        errorInfo tmp;//未通过参数的类型检查，返回错误信息
        tmp.errCode=11;
        if(begin->data.attribute!=term::Int&&begin->data.attribute!=term::Float){
            tmp.position=begin->data.position;//确定出错参数是哪一个，返回其位置信息
        }
        else if(del2->data.attribute!=term::Int&&del2->data.attribute!=term::Float){
            tmp.position=del2->data.position;
        }
        return tmp;
    }
}

```

④检查运算结果与整数数值的差异是否在一定阈值以下，决定运算结果的类型，将结果值保存在 begin 节点中。

```

begin->data.category=term::None;//将运算结果归并在 begin 节点中
if(equal(val_double,(int)val_double)){//结果近似为整数时
    begin->data.attribute=term::Int;//设定结果值的类型为整形
    begin->data.val.ival=(int)val_double;//保存结果值
}
else{
    begin->data.attribute=term::Float;//否则设定结果值类型为浮点型
    begin->data.val.dval=val_double;//保存结果值
}
}

```

⑤删除中间节点，将 begin 与 end 相连，若 end 为空，重新设定 tail 指针的指向。

更改链表节点的数据之后，再对链表的结构进行操作

```

begin->next=end;//令断点前后两个节点相连
if(end!=NULL){//front 指针不会改变，但 tail 指针可能作为 del 或 del2 被删除
    end->prev=begin;//确认 end 为实体对象，才设定其 prev 值
}
else{//end 非实体对象，说明 tail 将被删除，此时重新设定 tail 的指向
    tail=begin;
}
delete del;//将待删节点删除
--count;//减小链表节点计数
if(del2!=NULL){//当 category 值为 LR 或 RR 时，需要删除两个节点

```

```

        delete del2;
        --count;
    }

```

(2) 表达式 `expression` 的 `push` 与 `calculate` 方法。

`push` 方法的步骤:

①根据待插入对象的类型,构造临时 `term` 对象,确定其 `category` 值和 `attribute` 值,对于操作符,赋予它相应的函数指针。

`expression` 的 `push` 函数被重载多次,下面以其中一个重载函数为例说明问题。

`void push(double(*p)(double,double),int pos){//将函数指针为 p 的操作符加入链表和队列`

```

    term* tmp=new term;//建立临时 term 对象
    tmp->attribute=term::biDouble;//确定 attribute 值
    tmp->val.ptr1=p;//赋予函数指针
    tmp->category=term::RR;//确定 category 值
    tmp->position=pos;
    .....

```

②将临时对象插入到链表中。

```
expr.push(*tmp);
```

③对于操作符,将其根据优先级加入到相应的队列中。

`opr.zeroOrder.push(expr.tail);//函数为最优先运算的对象,将其节点指针放入第零级优先队列中`

`calculate` 方法的步骤为:

①当第零级运算符队列不空时,令队列元素相继出队。由于读取字符串时是按照从左到右的顺序,即入队顺序为从左到右,所以现在出队的运算符也是按照从左到右的顺序。

②将出队 `termNode*` 指针作为参数传递给 `mergeAdjacent` 函数,完成运算。

```

while(!opr.zeroOrder.isEmpty()){//当第零级运算符队列不空时
    err=expr.mergeAdjacent(opr.zeroOrder.pop());//操作符依次出队,代入
mergeAdjacent 函数,进行求值运算
    if(err.errCode!=-1)return err;//mergeAdjacent 函数返回错误时,将错误信息
返回
}

```

③当第零级运算符队列为空时,相继读取第一级、第二级、第三级运算符队列,重复①、②的步骤。

即将 `while(!opr.zeroOrder.isEmpty())` 的条件改为

`while(!opr.firstOrder.isEmpty())`、`while(!opr.secondOrder.isEmpty())`,重复执行类似的代码。

(3) 构建排序二叉树，匹配函数名的方法。

用数组构建排序二叉树的步骤为：

①写好一个 function 数组，数组中 function 对象的 tag 即函数名按照字典序排好。这样，就相当于得到了二叉排序树的中根序遍历序列。

上述过程在 calculator 类的构造函数中完成，其代码大致为：

```
calculator(){
    tagCopy(functionTable[0].tag,"abs");//functionTable 数组的元素是一系列的
function 类的对象，tagCopy 是一个辅助函数，用于字符串的复制，这里将本程
序支持的所有函数的函数名依次复制到 function 对象的 tag 字符串中
    tagCopy(functionTable[1].tag,"acos");//数组元素的 tag 按照字典序升序排
列
    tagCopy(functionTable[2].tag,"asin");
    .....
    functionTable[0].opr.val.ptr2=fabs;//设定 function 对象的函数名 tag 后，设
置其实际对应的函数指针
    functionTable[0].opr.attribute=term::uniDouble;//规定其相应的函数属性
    functionTable[0].opr.category=term::R;
    functionTable[1].opr.val.ptr2=acos;
    functionTable[1].opr.attribute=term::uniDouble;
    functionTable[1].opr.category=term::R;
    functionTable[2].opr.val.ptr2=asin;
    functionTable[2].opr.attribute=term::uniDouble;
    functionTable[2].opr.category=term::R;
    .....
}
```

②通过二叉排序树 matchTree 的构造函数，调用辅助函数 createNode。

在 calculator 类的构造函数中，执行 matchTree 的构造函数，用数组构造二叉排序树。

```
fTree=new matchTree(functionTable,12);
```

在二叉排序树 matchTree 的构造函数的具体实现中，整棵树的构造依靠 createNode 函数的递归调用完成。

```
matchTree(function f[],int n){
    createNode(root,f,0,n-1);
}
```

③对于给定的数组下界 lower 和数组上界 upper，取其中值 mid，用数组中坐标为 mid 的元素构造根节点。

```
void createNode(treeNode*& t,function f[],int lower,int upper){
    int mid;
    if(lower<=upper){
        t=new treeNode;
        mid=(lower+upper)/2;//取中值
        tagCopy(t->func.tag,f[mid].tag);//用中值元素构造节点，设定节点的
```

tag 属性

性

针

```
t->func.opr.attribute=f[mid].opr.attribute;//设定 function 节点的其他属性

t->func.opr.category=f[mid].opr.category;
switch(t->func.opr.attribute){//分类讨论，设定 function 节点的函数指针

    case term::biDouble:
        t->func.opr.val.ptr1=f[mid].opr.val.ptr1;
        break;
    case term::uniDouble:
        t->func.opr.val.ptr2=f[mid].opr.val.ptr2;
        break;
    case term::biInt:
        t->func.opr.val.ptr3=f[mid].opr.val.ptr3;
        break;
    case term::uniInt:
        t->func.opr.val.ptr4=f[mid].opr.val.ptr4;
        break;
    default:
        cout<<"Failed to construct the function table.";
} //完成对于节点的构造和属性的设定
```

④对于左子树，令下界为 lower，上界为 mid-1，对于右子树，令下界为 mid+1，上界为 upper，递归调用 createNode 函数，重复③的步骤。

createNode(t->lch,f,lower,mid-1);//递归构建左子树

createNode(t->rch,f,mid+1,upper);//递归构建右子树

在 match 函数中，遍历字符串，匹配函数名的方法：

①设定二叉树节点指针指向二叉树的根节点，设定遍历坐标 j=0，即从节点 tag 的坐标 0 处开始匹配，记录原字符串起始比较位置坐标 orig。

```
term match(char* ch,int& i){
```

int j=0;//设置遍历 matchTree 对象 fTree 的节点 function 对象的 tag 字符串的遍历坐标

```
int orig=i;//记录字符串匹配的起始位置，以备后面的回溯
```

```
matchTree::treeNode* node=fTree->root;//从二叉排序树的根节点开始遍
```

历

②若原字符串 ch[i]>tag[j]，令二叉树节点指针指向原节点的右孩子节点，将 i 回溯为 orig，将 j 回溯为 0，重新开始字符串匹配过程。若原字符串 ch[i]<tag[j]，令二叉树节点指针指向原节点的左孩子节点，将 i 回溯为 orig，将 j 回溯为 0，重新开始字符串匹配过程。

③若二叉树节点指针值为 NULL，匹配失败，跳出循环。

```
while(node!=NULL){//遍历排序二叉树，匹配失败
```

if(ch[i]>node->func.tag[j]){//比较 ch[i]与 tag[j]的值, 决定向二叉树的左子树或右子树遍历

```

    i=orig;//字符串坐标回溯
    j=0;
    node=node->rch;
}
else if(ch[i]<node->func.tag[j]){
    i=orig;
    j=0;
    node=node->lch;
}

```

④若 ch[i]==tag[j], 将 i 与 j 的值分别加 1, 继续匹配过程。

```
else if(ch[i]==node->func.tag[j]){
```

```
    ++i;
```

++j;//相等时, 继续比较字符串的下一字符与当前节点 tag 的下一字符是否匹配

break;//跳至外层循环, 检查 i, j 更新后, 字符串的字母串是否结束, 当前节点的 tag 字符串是否遍历完毕, 若不然, 继续匹配过程, 否则, 进入下面的判断过程

```
}
```

⑤若原字符串的字母串部分刚好结束, tag[j]在 j++后刚好为'\0', 说明匹配成功, 用匹配成功的 tag 对应的 function 对象的函数指针及属性值构造新的操作符 term 对象, 作为 match 函数的返回值。

⑥若匹配不成功, 构造一个函数指针指向自定义的 Fail 函数的 term 对象作为返回值。

```
if(node==NULL){//字母串未结束, 二叉树已经遍历完毕, 说明匹配失败
```

```
    err=true;
```

```
    errCode=13;
```

```
    term tmp;//构造一个 term 对象
```

```
    tmp.val.ptr1=Fail;//其函数指针指向特别定义的 Fail 函数
```

```
    tmp.attribute=term::biDouble;//设定相关属性
```

```
    tmp.category=term::RR;
```

tmp.position=orig;//把匹配失败的字母串的第一个字母的位置作为错误信息的定位信息

```
    return tmp;
```

```
}
```

else if(!isAlphabet(ch[i])&&node->func.tag[j]=='\0'){//字母串结束时, tag 字符串也刚好遍历完毕, 说明函数名匹配成功

```
    term tmp;
```

tmp.attribute=node->func.opr.attribute;//用匹配成功的 function 节点构造操作符 term 对象


```

    tmp.category=node->func.opr.category;
    tmp.position=orig;
    switch(node->func.attribute){
        case term::biDouble://分类讨论，设定 term 的函数指针
            tmp.val.ptr1=node->func.opr.val.ptr1;
            break;
        case term::uniDouble:
            tmp.val.ptr2=node->func.opr.val.ptr2;
            break;
        case term::biInt:
            tmp.val.ptr3=node->func.opr.val.ptr3;
            break;
        case term::uniInt:
            tmp.val.ptr4=node->func.opr.val.ptr4;
            break;
        default:
            cout<<"Exception";
    }
    return tmp;
}

```

else{//其他情况下，说明字符串与函数名长度不符，比如函数表中有“sin”而输入的字母串是“sinc”或者函数表中有“sin”而输入的字母串是“si”，此时，也需要返回错误

```

    err=true;
    errCode=13;
    term tmp;
    tmp.val.ptr1=Fail;//同样地构造 term 对象，以 Fail 作为函数指针
    tmp.attribute=term::biDouble;
    tmp.category=term::RR;
    tmp.position=orig;
    return tmp;
}

```

(4) calculator 类的 calculate 函数。

calculate 函数的实现步骤：

①读取字符串 ch，首先判断字符串 ch 的第一个字符是否为'\0'，若是，返回 0 值。

```

err=false;//初始化错误码
errCode=-1;
if(ch[0]=='\0'){//对于空字符串，返回值 0
    result tmp;
    tmp.value=0;
    tmp.error=false;
    return tmp;
}

```

```
}
```

②若不然，构造一个 `expression` 对象 `e`。顺序读取字符串 `ch` 中的字符。判断字符的类型，决定调用相应的归约函数，在传参时，把原字符串中的坐标信息 `i` 和 `expression` 对象 `e` 的引用传入，使得归约函数能够将归约结果插入 `expression` 的链表和队列。也使得归约函数能够控制对于字符串的遍历。

```
int i=0;//初始化坐标信息
expression* e=new expression;//建立表达式
while(ch[i]!='\0'){//读到字符串尾之前
    if(isNumber(ch[i])){//对读到字符的类型分类讨论
        readNumber(ch,i,*e);//调用相应的归约函数，将遍历的位置信息 i 和
        表达式 e 的引用传入，便于归约函数对其进行更新和更改
    }
    else if(ch[i]=='.'){
        readNumber(ch,i,*e,true);
    }
    else if(isOperator(ch[i])){
        readOperator(ch,i,*e);
    }
    else if(isAlphabet(ch[i])){
        int rlt=readFunction(ch,i,*e,offset);
        .....
    }
}
```

③对于不同类型的字符用不同的归约函数处理之后，坐标 `i` 值被归约函数更新，`err` 值与 `errCode` 值也得到了更新，检查 `err` 是否被置为 `true`，若是，返回一个 `error` 值为 `true` 的 `result` 对象，结束程序。

```
if(err){//错误状态被归约函数置为 true
    cout<<"Position:"<<offset+i+1<<" Error
    Code:"<<errCode<<endl<<errMsg[errCode];//将归约函数设定的错误码 errCode 值
    输出，在错误信息数组中读取相应的错误信息进行输出。在递归调用 calculate
    函数时，需要通过字符串对母串的偏移量 offset 和字符串的坐标值 i 计算出错误位置在
    母串中的定位。+1 是为了符合人的习惯，将起始字符看做第 1 个而不是第 0 个
    字符
```

```
    result tmp;//构造 result 对象
    tmp.value=0;
    tmp.error=true;//表明错误发生
    return tmp;
}
```

④若不然，检查 `i` 值被归约函数更新后，`ch[i]` 是否为 `'\0'`，若不然，调用下一个归约函数进行归约。可以看出，对于语法正确的简单（此处简单指不含括号即不需递归）表达式，`calculate` 函数的循环执行历程是：在每个语法单元子串的串首，由 `calculate` 判断字符类型，执行相应的归约函数，归约函数完成归约后，将 `i` 放置到下一个语法单元子串的串首，由此循环往复。

⑤`ch[i]=='\0'`时,跳出循环,对表达式 `e` 执行 `calculate` 函数,然后打印出计算结果。

`errorInfo eInfo=e->calculate();`//对最终得到的表达式 `e` 进行计算,将返回值保存在 `eInfo` 中

`if(eInfo.errCode!=-1){`//如果对 expression `e` 执行 `calculate`(这里的 `calculate` 函数是 `expression` 类的 `calculate` 函数,而不是上面的 `calculator` 类的 `calculate` 函数)函数出错,输出错误信息,返回表示错误的 `result` 对象

`cout<<"Position:"<<offset+eInfo.position+1<<" Error Code:"<<eInfo.errCode<<endl<<errMsg[eInfo.errCode];`

`result tmp;`

`tmp.value=0;`

`tmp.error=true;`

`return tmp;`

`}`//若 expression `e` 的 `calculate` 函数执行未出错

`if(offset==0){`//对于主串调用的 `calculate` 函数, `offset` 为 0,对于递归调用的 `calculate` 函数, `offset` 不为 0。在这里,如果 `offset` 为 0,说明主串的表达式 `e` 成功进行了计算求值,此时输出最终结果,如果 `offset` 不为 0,则仅仅是某括号中的子串计算完毕,则不输出结果

`cout<<"=";`

`e->print();`

`}`

`result tmp;`//构造 `result` 对象

`if(e->expr.front->data.attribute==term::Float){`//读取 `e` 的计算结果,根据其类型信息,将 `e` 的值读取到 `result` 对象中

`tmp.value=e->expr.front->data.val.dval;`

`}`

`else if(e->expr.front->data.attribute==term::Int){`

`tmp.value=e->expr.front->data.val.ival;`

`}`

`tmp.error=false;`

`delete e;`

`return tmp;`

(5) 归约数字串的 `readNumber` 函数。

`readNumber` 函数的实现步骤:

①顺序读取数字串,将临时归约值保存在变量 `ival` 中,执行 `ival=ival*10+ch[i]-'0'`;

`void readNumber(char* ch,int& i,expression& e,bool omit=false){`

`int ival=0;`//这里设定 `ival` 为 0 是为了 `omit` 为 `true` 时的情况,即整数部分为 0 的情况

`int orig;`

`if(!omit){`

`orig=i;`//保存数字串语法单元的定位信息

`ival=ch[i]-'0';`

```

while(isNumber(ch[++i])){
    ival=ival*10+ch[i]-'0';//保存数字串的临时归约值
}

```

②遇到第一个非数字字符时，对其类型进行讨论。若为其他语法单元的合法字符，退出 readNumber 函数返回 calculate 的主循环，对于省略乘号的情况，补充插入一个 '*' 运算符进入 expression e 的链表和队列。对于非法字符，置 err 值为 true，根据错误类型设定 errorCode 值，退出 readNumber 函数。对于该字符为 '.' 的情况，继续检查 ch[i+1] 的字符类型。

```

if(ch[i]=='0'){//在数字型的语法单元后，字符串直接结束，这是合法的
    e.push(ival,orig);//将整形变量作为操作数加入表达式
    return;
}
else if(isOperator(ch[i])){//数字型语法单元后面跟着操作符，这是合法的
    e.push(ival,orig);
    return;
}
else if(isAlphabet(ch[i])){//数字型语法单元后面跟着函数，这是乘号省略的情况
    e.push(ival,orig);
    e.push('*',i);
    return;
}
else if(ch[i]==','){//数字后面出现逗号的情况，只可能在 readFunction 函数中被一次性归约，在这里出现，一定是不合法的
    err=true;
    errorCode=0;
    return;
}
else if(ch[i]==''){//处理左括号的代码已经一次性地将与左括号配对的右括号进行归约了，这里的括号是不合法的
    err=true;
    errorCode=1;
    return;
}
else if(ch[i]=='('){//数字型语法单元后面出现非数字的数字型语法单元，这是允许的，但需要在表达式 e 中补足省略的乘号
    e.push(ival,orig);
    e.push('*',i);
    return;
}
else if(ch[i]=='.'){//对于隔点符号，执行空语句之后跳到下面的处理小数部分的代码
    ;

```

```

    }
    else{//其他字符均为非法字符
        err=true;
        errCode=2;
        return;
    }

```

③对于整数部分为 0 时省略的情况，应跳过①②直接从③开始。若小数点后第一个字符不是数字，设定错误码，返回 `calculate`。若不然，将小数部分值归约到变量 `dval` 中。

```

    if(isNumber(ch[++i])){//如果小数点后第一位是数字
        if(omit)orig=i;//对于整数部分为 0 且省略整数部分的情况，用小数部分第
        一位作为该数字的定位信息
        double dval=0;
        double deci=1.0/10;
        do{
            dval+=(ch[i++]-'0')*deci;
            deci=deci/10;
        }while(isNumber(ch[i]));//归约小数部分值
        dval+=ival;
        e.push(dval,orig);//将整数部分与小数部分值加在一起，将数值加入表达式

```

④数字串结束时，进行类似②中的语法验证，语法非法时设置错误码，合法时补足省略的算符，将整数部分值 `ival` 和小数部分值 `dval` 相加，用得到的值构造 `term` 对象，插入链表，返回 `calculate`。

```

    if(ch[i]=='0'){//以数字结尾，合法
        return;
    }
    else if(isOperator(ch[i])){//数字+操作符，合法
        return;
    }
    else if(isAlphabet(ch[i])){//数字+函数，补足省略的乘号
        e.push('*',i);
        return;
    }
    else if(ch[i]==','){//数字+分隔号，对于一次性归约的算法不应在这里出现
    这种情况，非法
        err=true;
        errCode=0;
        return;
    }
    else if(ch[i]=='('){//数字+数字型语法单元，合法，补足乘号
        e.push('*',i);

```

```

        return;
    }
    else if(ch[i]==''){//一次性归约算法中不可能出现，非法
        err=true;
        errCode=1;
        return;
    }
    else if(ch[i]=='.'){//小数部分后再次出现小数点，非法
        err=true;
        errCode=3;
        return;
    }
    else{//其余字符均为非法
        err=true;
        errCode=2;
        return;
    }
}

```

(6) 归约运算符的 readOperator 函数。

readOperator 的步骤如下：

①先检查该运算符是否出现在表达式的首位。若首位出现运算符，只有‘+’与‘-’为合法，其余返回错误。对于‘+’、‘-’分别检验其后面的字符是否属于合法的语法单元，若不然返回错误。对于合法字符，对于‘+’可不做任何处理直接跳过，对于‘-’本程序在其前面先插入一个操作数数值为 0 的节点，再插入该‘-’节点。

```

if(e.expr.tail==NULL){//算符位于表达式首
    if(ch[i]!='+'&&ch[i]!='-'){//加号减号以外的都是非法的
        err=true;
        errCode=5;
        return;
    }
    else if(ch[i]=='+'){
        ++i;
        if(isNumber(ch[i])||isAlphabet(ch[i])||ch[i]=='.'||ch[i]=='('){//若加号之后
            出现的是数字型的语法单元，都是合法的，跳过该加号不做任何处理
            return;
        }
        else if(ch[i]=='0'){//对不合法情况进行分类，输出错误信息
            err=true;
            errCode=12;
            return;
        }
        else{
            err=true;

```

```

        errCode=6;
        return;
    }
}
else{//出现在表达式首的是减号
    ++i;
    if(isNumber(ch[i])||isAlphabet(ch[i])||ch[i]=='.'||ch[i]=='('){//若减号后面
出现的是数字型语法单元
        e.push(0,i-1);//将-x 视为 0-x，在前面补上 0
        e.push('-',i-1);
        return;
    }
    else if(ch[i]=='\0'){//其他情况下，分类输出错误信息
        err=true;
        errCode=12;
        return;
    }
    else{
        err=true;
        errCode=6;
        return;
    }
}
}

```

②对于不是出现在表达式首位的运算符，根据其是否是'!'进行分类讨论。由于'!'和其左边的数值进行归并，可以认为'!'是一个数字，这就允许其后面为空，或者出现数字（需补足省略的'*'）、字母、运算符、括号四种情况。当'!'后面出现运算符时，本程序要求该运算符不能为'!'。对于其他算符，还要再读取下一位字符，看是不是和运算符连接的许可语法单元。由于除数字之外，括号和函数都会优先归并为数值，所以运算符右边出现数字、括号、函数语法单元中任何一个的许可首字符都是可以接受的。

```

else{//该算符为'!'
    ++i;
    if(isNumber(ch[i])||isAlphabet(ch[i])||ch[i]=='.'||ch[i]=='('){
        e.push('!',i-1);
        e.push('*',i-1);
        return;
    }//如果下一个字符是数字型语法单元，合法，补足乘号
    else if(ch[i]=='\0'){以 x! 作为结尾，合法
        e.push('!',i-1);
        return;
    }
    else if(isOperator(ch[i])){x! 后面出现算符，需要分情况讨论
        if(ch[i]=='!'){x!!的情况不被允许

```

```

        err=true;
        errCode=7;
        return;
    }
    else{

        if(isNumber(ch[i+1])||isAlphabet(ch[i+1])||ch[i+1]=='.'||ch[i+1]=='('){//x!后面的
算符后面必须有数字型语法单元
            e.push('!',i-1);
            e.push(ch[i],i);
            ++i;
            return;
        }
        else if(ch[i+1]=='\0'){//否则，分类设置错误码
            err=true;
            errCode=12;
            return;
        }
        else{
            err=true;
            errCode=6;
            return;
        }
    }
}
else{
    err=true;
    errCode=6;
    return;
}
}

```

③对于不是出现在表达式首位的运算符，若其不是'!',则其后面必须为数字性质的语法单元，且不能为空。检验运算符后面的字符是否符合要求，决定是否返回错误码。

```
if(ch[i]!='!'){
```

```

    if(isNumber(ch[i+1])||isAlphabet(ch[i+1])||ch[i+1]=='.'||ch[i+1]=='('){//后面是数
字型语法单元时，合法
        e.push(ch[i],i);
        ++i;
        return;
    }
    else if(ch[i+1]=='\0'){//否则，分类输出错误码

```



```

        err=true;
        errCode=12;
        return;
    }
    else{
        err=true;
        errCode=6;
        return;
    }
}

```

(7) 归约函数的 readFunction 函数。

readFunction 函数的步骤如下：

①对于字母串，执行 (3) 中过程，对字母串进行匹配。

```

int readFunction(char* ch,int& i,expression& e,int offset){
    int orig=i;
    term func=match(ch,i);

```

②若匹配失败，终止语法分析，输出错误。若匹配成功，保存匹配得到的函数指针。进入参数匹配过程。

```

    if(func.val.ptr1==Fail){//匹配失败
        i=func.position;
        return -1;//非递归调用中的错误，返回-1，表示需要在 calculate 函数中输出错误信息
    }
    int rlt=readParameter(ch,i,e,&func,orig,offset);//若匹配成功，执行 readParameter 函数归约参数

```

③根据匹配得到的函数的性质，记录函数要求的参数数目。建立实际读取字符串时读到的参数的计数器，建立左括号和右括号的计数器。当右括号数目多于左括号时，参数归并过程结束。当左右括号数相等且读到分隔符','时，令参数计数器加 1。把括号与括号或括号与逗号间的子串传递给 calculate 进行求值，将所求参数值传递给②中得到的函数指针，将函数部分归约为一个数值。

```

int readParameter(char* ch,int& i,expression& e,term* t,int begin,int offset){
    if(ch[i]=='\0'){//处理函数名之后不为'('的各种非法情况
        err=true;
        errCode=14;
        return -1;
    }
    else if(ch[i]!='('){
        err=true;
        errCode=15;
        return -1;
    }

```

```

else{//进入函数的参数部分
    int lbr=0;//记录进入参数左括号后遍历的左括号数
    int rbr=0;//记录进入参数左括号后遍历的右括号数
    int paraNum;//记录匹配得到的函数应有的参数个数
    if(t->attribute==term::uniDouble||t->attribute==term::uniInt){
        paraNum=1;//通过 match 返回的 term 对象的属性获知函数应有的
参数个数
    }
    else if(t->attribute==term::biDouble||t->attribute==term::biInt){
        paraNum=2;
    }
    double para[paraNum];//根据应用的参数个数建立参数表数组
    int paraCnt;//对实际读取的参数个数建立计数器
    if(ch[i+1]==')'){//读取到 f()的情况下，实际参数计数为 0
        paraCnt=0;
    }
    else{//参数括号中不为空时
        int orig=i;
        paraCnt=1;
        while(rbr<=lbr){
            ++i;//遍历括号中字符串
            if(ch[i]=='('){//对左括号，右括号进行计数
                ++lbr;
            }
            else if(ch[i]==')'){
                ++rbr;
            }
        }
        if(ch[i]!='\0'){//未找到匹配的右括号，表达式提前终止
            err=true;
            errCode=19;
            return -1;
        }
        if(lbr==rbr&&ch[i]==','){//左括号计数与右括号计数相等时，
出现的分隔符是外层函数参数的分隔符，其他地方出现的逗号，是复合的其他函
数中的参数分隔符
            ++paraCnt;//更新实际参数计数
            if(paraCnt>paraNum){//参数计数多于实际应有参数个数
                时，返回错误信息
                err=true;
                errCode=17;
                return -1;
            }
            else{//未发现错误时，每经过一个符合条件的逗号，对于
括号与逗号，逗号与逗号分隔的参数表达式字符串进行归约求值

```

```

        result r=calculate(substr(ch,orig,i),offset+orig+1);
        if(!r.error){//若求值成功，更新参数数组中的参数值
            para[paraCnt-2]=r.value;
        }
        else{//若发生错误，返回值为-2，表示在递归调用
calculate 函数时已经输出过错误信息，此次返回上层 calculate 函数后不必再次输出错误信息

```

```

            return -2;
        }
        orig=i;
    }
}
}
result r=calculate(substr(ch,orig,i),offset+orig+1);//跳出右括号后，
对于最后一个逗号到括号之间的参数进行归约求值。或者对于函数唯一的参数进行
表达式的归约求值

```

```

        if(!r.error){//求值后的处理方式同上
            para[paraCnt-1]=r.value;
        }
        else{
            return -2;
        }
    }
    if(paraCnt<paraNum){//检查实际参数个数是否少于应有函数参数个
数

```

```

        err=true;
        errCode=16;
        return -1;
    }
    switch(t->attribute){//参数求值完毕，分情况讨论，将参数代入到对应的
函数指针中，求出函数值
        case term::biDouble:

```

```

        if(equal(t->val.ptr1(para[0],para[1]),(int)t->val.ptr1(para[0],para[1]))){
            e.push((int)t->val.ptr1(para[0],para[1]),begin);
        }//类型检查，若函数值与整数之差小于阈值，认为其为整形
        else{
            e.push(t->val.ptr1(para[0],para[1]),begin);
        }
        break;
        case term::uniDouble:
            if(equal(t->val.ptr2(para[0]),(int)t->val.ptr2(para[0]))){
                e.push((int)t->val.ptr2(para[0]),begin);
            }

```

```

        else{
            e.push(t->val.ptr2(para[0]),begin);
        }
        break;
    case term::biInt:
        e.push(t->val.ptr3(para[0],para[1]),begin);
        break;
    case term::uniInt:
        e.push(t->val.ptr4(para[0]),begin);
        break;
    default:cout<<"Error";
}
++i;//检查函数后面的语法单元
if(isNumber(ch[i])||isAlphabet(ch[i])||ch[i]=='.'||ch[i]=='('){//后面是数字
    型语法单元，合法，补足乘号
    e.push('*',i);
    return 0;
}
else if(isOperator(ch[i])||ch[i]=='\0'){//函数本身是数字型语法单元，以
    其作为结束，合法；后面出现算符，也属合法
    return 0;
}
else{//否则，处理非法字符
    err=true;
    errCode=18;
    return -1;
}
}
}
}

```

(8) calculate 函数遇到括号时的递归调用。

①建立左右括号计数器，当右括号计数刚好比左括号计数多 1 时，构建从左括号到此处的子串，递归调用 calculate，以求得内层括号中表达式的值。

②若提前遇到字符串终止，设置错误码。若递归调用的 calculate 函数返回的 result 对象的 error 属性为 true，构建一个 error 属性为 true 的 result 对象作为返回值，依次结束上一层 calculate 的执行。

```

else if(ch[i]=='('){
    int lbr=0;//设立左括号计数器
    int rbr=0;//设立右括号计数器
    int orig=i;
    while(rbr<=lbr){//右括号比左括号多 1 时，说明左右括号得到匹配
        ++i;
        if(ch[i]=='('){//在此之前，遍历字符串，对左右括号进行计数
            ++lbr;
        }
    }
}

```

```

        if(ch[i]==')'){
            ++rbr;
        }
        if(ch[i]=='\0'){//括号未配对，表达式已经结束
            err=true;
            errCode=1;
            i=orig;
            break;
        }
    }
    if(!err){
        result r=calculate(substr(ch,orig,i),orig+1);//对于括号中的子串递归调用 calculate，求出子表达式的值
        if(!r.error){//求值过程无误时，用将括号中子表达式的数值插入到 expression e 中
            if(equal(r.value,(int)r.value)){//与整数值极其接近的表达式的值视为整数值
                e->push((int)r.value,orig+1);
            }
            else{//否则，用浮点数值作为子表达式的值
                e->push(r.value,orig+1);
            }
            ++i;
            if(isNumber(ch[i])||isAlphabet(ch[i])||ch[i]=='('){//括号属于数字型的语法单元，对其归约求值之后，检查后面的语法单元，若后面也为数字型的语法单元，需在两者间补足一个省略的乘号
                e->push('*',i);
            }
        }
        else{
            result tmp;
            tmp.value=0;
            tmp.error=true;
            return tmp;
        }
    }
}

```

3.算法的时间、空间复杂度：

关于时间复杂度：对于不含任何括号的简单表达式，本程序从左到右扫描字符串一次就可以得到计算结果，因此对于规模为 n 的简单表达式，扫描字符串贡献的时间复杂度为 $O(n)$ 。

在扫描过程中，本程序将每个语法单元归约，加入链表，将运算符加入队列，可以认为表达式中语法单元的数目以及其中操作符的数目都是与 n 成正比的，因

此归约操作，也就是链表插入和队列入队的操作，其时间复杂度为 $O(n)$ 。

在表达式求值的过程中，链表节点的合并过程中，优先级队列中的指针为合并位点提供了“书签”，因而合并过程基本上不需要遍历链表，本质上是总共 k 个语法单元节点合并为 1 个语法单元节点的过程，其时间复杂度与 k 的规模有关，由于 k 与 n 大致成正比，所以链表合并的过程贡献了 $O(n)$ 的时间复杂度。优先级队列出队的过程的时间复杂度和操作符的数目 m 有关，由于 m 与 n 大致成正比，所以优先级队列出队的时间复杂度也为 $O(n)$ 。表达式求值的总体时间复杂度也为 $O(n)$ 。

在计算求值的过程中，由于具体的运算由头文件 `cmath` 中定义的函数实现，其实现细节不详，因此我不能准确估计其时间复杂度。然而从另一方面来看，阶乘运算虽然复杂，但对于单个的整形变量，由于整形变量是有上界的，所以阶乘运算对于单个整形变量的运算次数是有上界的。同理，假设 `sin`，`cos`，`exp` 等函数由幂级数的求和方式实现，由于浮点数的计算精度也是有限的，所以它们对于单个浮点数变量的运算次数也是有上界的。

由上面的分析可以看出，对于单个操作数，函数运算造成的运算开销存在上界 M ，这样，对于长度为 n 的表达式，由运算造成的时间复杂度最多为 $M*n$ ，即使 M 值很大，我们仍然认为表达式求值、函数运算对于时间复杂度的贡献为 $O(n)$ 。

基于类似的道理，我们认为排序二叉树的匹配过程也贡献了 $O(n)$ 的时间复杂度。

综上所述，对于简单的不含括号的表达式，本程序的时间复杂度为 $O(n)$ 。

对于含有参数和括号，需要递归或嵌套调用的情况，由于子表达式会在归约求值后作为一个操作数加入到主表达式的链表中，因此可以认为链表的插入、队列的入队、函数名的匹配、链表的合并、函数的求值这些操作的次数并没有增加。真正增加的是对于子字符串的遍历次数。由程序设计可知，对于括号中的子字符串，复制构造子串和递归函数的遍历使得程序对它的遍历次数增加了 2 次。对于一般的，括号的嵌套次数有确定的上界 M ，该上界值与表达式的规模 n 无关的情况，可以认为本程序的时间复杂度仍为 $O(n)$ 。

在最坏的情况下，整个表达式就是由 $(f(g(h(\dots))))$ 这样的括号嵌套组成的，在这样的情况下，括号的嵌套次数与 n 成正比，此时本程序的最坏时间复杂度为 $O(n^2)$ 。

综上，本程序的主要功能函数 `calculate` 的时间复杂度一般为 $O(n)$ ，最坏为 $O(n^2)$ 。

空间复杂度：

动态申请内存空间造成的空间复杂度来自链表与队列的延长，以及对于括号中的子字符串的复制。链表插入与队列入队造成的空间复杂度为 $O(n)$ 。对于括号，类似上面的讨论，如果括号的嵌套次数有一个与表达式规模 n 无关的上界，则子字符串的复制贡献的空间复杂度为 $O(n)$ ，在最坏情况下，括号的嵌套次数与 n 成正比，此时复制子字符串的空间复杂度为 $O(n^2)$ 。

关于由递归、嵌套调用带来的栈的空间复杂度：在没有括号，函数的参数为数值或没有括号的简单表达式的情况下，`calculate` 与 `readNumber`、`readOperator` 之间的调用返回关系的最大栈深度为 2，空间复杂度为 $O(1)$ 。`readFunction` 与 `match` 和 `readParameter` 之间的调用返回关系的最大栈深度为 2，空间复杂度为 $O(1)$ 。在存在括号嵌套的情况下，可以认为最坏情况下括号的嵌套次数与 n 成正比，此

时栈的空间复杂度为 $O(n^2)$ 。

综上，本程序的主要功能函数 `calculate` 的空间复杂度一般为 $O(n)$ ，最坏为 $O(n^2)$ 。

2.3 其他

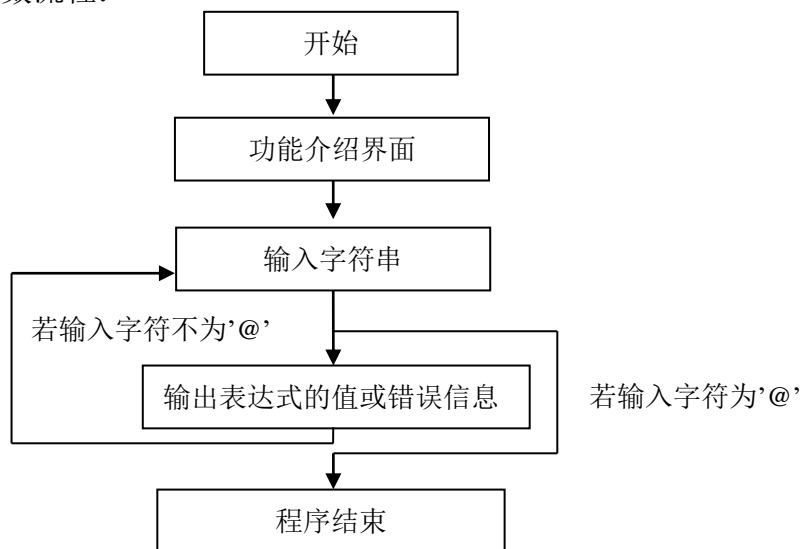
本程序用面向对象的程序设计思路，设计并规定了操作符、操作数、函数作为类的对象，在表达式求值过程中的方法和属性。这样做的好处是使得本程序的设计不受限于对于特定函数与运算符的处理过程，而是能够较好地兼容一般的操作符与函数的处理流程。

正因为这样，本程序支持的函数表具有可扩充的特性。用户完全可以自行定义函数，只要将它们按字典序添加到 `function` 数组中，给它们定义合适的属性，分配合适的函数指针，本程序就能够对它们进行匹配与运算。

可以说，本次程序设计让我面对更为复杂的表达式时有了一个基本的设计框架。完全可以设想，在以后的设计中，运算符号`+ - * /`也可以直接添加到函数匹配树中，由匹配树承担所有非数字字符的分析。为算符与函数设计的类如果能够对它们的操作属性与语法属性做出很好的定义与描述，那么表达式处理程序将能够支持更一般的运算符的识别与求值功能。再进一步，我们还可以让匹配树中包括算符和标签两类元素，其中标签用来表示数字是二进制或十进制，等等。而这一切的基础，就是面向对象的程序设计思路，就是设计良好的相关的类的实现。

3. 程序运行结果

1.测试主函数流程：



2.测试条件：

本程序在 Windows10 64 位系统，Dev C++ 5.11 编译环境下测试通过。

可以说，测试的过程就是发现错误并改进的过程。在测试之初，本程序已经能够对一般的带有括号和函数的表达式进行求值，经过多次测试验算，运算结果绝大多数是正确的。

另一方面，测试中也发现了一些问题，这些问题现在已经得到了改正。

一个是运行时抛出错误的问题，经排查，该问题是对于已经指向 NULL 的空指针进行操作造成的，现在，在相关语段已经添加了对于指针是否为 NULL 的判断语句，保证了程序只对非空指针进行操作。

运行时出现过输入“(5.2+2.8)!”，结果程序输出错误信息的情况，解决方案是在代码中增加了对于类型的检查和处理的功能，使得结果与整数值差距极小的数值被认为是整形变量。

运行时出现过错误信息打印多次和定位信息不正确的情况，这是由于函数的递归与嵌套调用造成的。对于这个问题的处理方法，在本文第一部分已有叙述。

运行时出现过其他错误信息不正确的情况，其中原因之一是在实现语法成分的归约的函数 readNumber、readOperator、readFunction 中，没有考虑到字符串突然中止，读取到'\0'的情况，在现有代码中，此类问题已经得到了解决。

运行时出现过输入(5+3)(2+6)，输出为 88 的情况。这是因为在之前编写的代码中，只有 readNumber、readOperator 和 readFunction 函数考虑了乘号的省略问题，处理括号内容的主体函数 calculate 反而没有考虑乘号省略问题。对于括号，往往只考虑了在进入左括号之前补足省略的乘号的问题，没有考虑跳出右括号后补足省略的乘号的问题。这样，当 calculate 函数把两个子表达式的值 8 与 8 分别求出并 push 到表达式 e 中之后，由于没有补上省略的乘号，这时表达式的算符队列为空，程序便认为计算结束，于是将表达式链表中节点的值顺序输出，这样就得到了 88 的结果。对此的解决办法是在 calculate 函数处理括号部分的代码中，当子表达式求值完毕后，检查到后面的字符元素为字母、括号或数字时，将省略的乘号添加到表达式中。

3.测试结论:

总的来说,本程序对于各种比较复杂的函数复合、括号嵌套的表达式都能计算出正确的数值。

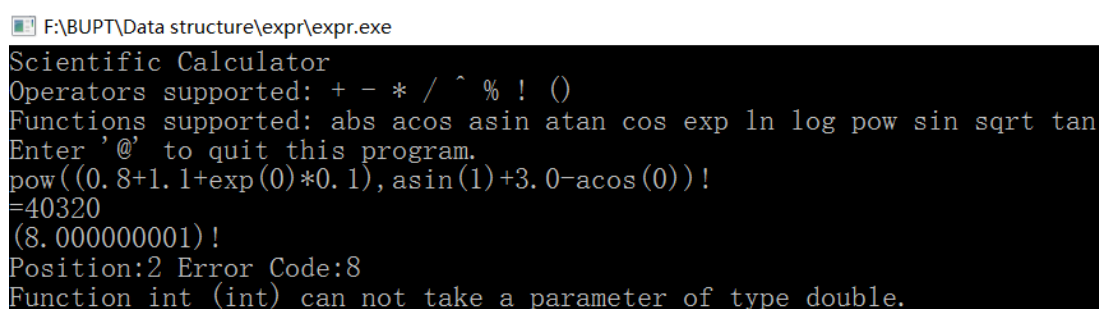
解决了测试中出现的种种问题后,现阶段本程序对于绝大多数表达式都能给出正确的数值结果,在大部分情况下,对于语法不合法的表达式能够输出符合设计预期的反馈信息。

另一方面,测试中发现的种种问题也表明,我在本程序设计之初没有特别重视空指针的问题,没有特别严谨地将不同语法元素之间的转化关系画出来,因而设计有一定的随意性,因而后来采取的补救措施也可能有局限性,或者出现解决了旧问题,又引入新问题的情况。需要针对本程序的设计弱点,多举出一些刁钻的样例,才能够进一步发现潜在的问题。

总之,本程序表达式求值的正确性高,反馈错误信息的准确率较高。对于语法正确的字符串,本程序能够准确求值,对于语法存在省略和错误的字符串,本程序可能存在着潜在的漏洞。

本程序实现了对于语法错误的分析,但没有检查运算具体的合法性。比如,对于 $1/0$ 或者负数开平方这样的运算,本程序的输出结果可能要视系统与编译器的特性而定(在本人的计算机上,这类非法运算的输出结果为 `inf` 或 `nan`)。另外,本程序没有检查整形变量是否溢出,因而,在含有阶乘 $n!$ 的表达式中,如果 n 值稍大,本程序会给出错误的结果,而不会给出提示信息。

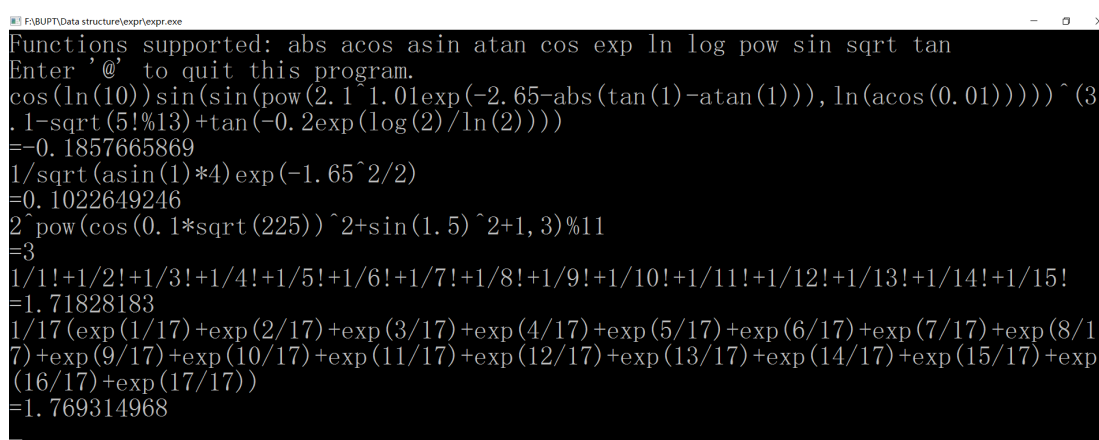
以下是本程序在一些样例下的运行结果。



```

F:\BUPT\Data structure\expr\expr.exe
Scientific Calculator
Operators supported: + - * / ^ % ! ()
Functions supported: abs acos asin atan cos exp ln log pow sin sqrt tan
Enter '@' to quit this program.
pow((0.8+1.1+exp(0)*0.1),asin(1)+3.0-acos(0))!
=40320
(8.000000001)!
Position:2 Error Code:8
Function int (int) can not take a parameter of type double.
  
```

本例表明本程序可以进行类型检查,另一方面也可以把极其接近整数值的浮点数识别为整形变量。



```

F:\BUPT\Data structure\expr\expr.exe
Functions supported: abs acos asin atan cos exp ln log pow sin sqrt tan
Enter '@' to quit this program.
cos(ln(10))sin(sin(pow(2.11.01exp(-2.65-abs(tan(1)-atan(1))),ln(acos(0.01)))))^(3
.1-sqrt(5!%13)+tan(-0.2exp(log(2)/ln(2))))
=-0.1857665869
1/sqrt(asin(1)*4)exp(-1.65^2/2)
=0.1022649246
2^pow(cos(0.1*sqrt(225))^2+sin(1.5)^2+1,3)%11
=3
1/1!+1/2!+1/3!+1/4!+1/5!+1/6!+1/7!+1/8!+1/9!+1/10!+1/11!+1/12!+1/13!+1/14!+1/15!
=1.71828183
1/17(exp(1/17)+exp(2/17)+exp(3/17)+exp(4/17)+exp(5/17)+exp(6/17)+exp(7/17)+exp(8/1
7)+exp(9/17)+exp(10/17)+exp(11/17)+exp(12/17)+exp(13/17)+exp(14/17)+exp(15/17)+exp
(16/17)+exp(17/17))
=1.769314968
  
```

本例验证了本程序执行科学计算的表现,验证了本程序对于复合函数,括号嵌套的复杂表达式的求值情况,验证了本程序对于级数的计算与积分的近似计算,从一个侧面表明了计算的准确性。验证了本程序对于多种算符与函数的支持,

验证了本程序对于省略乘号的情况的支持。

```
Enter '@' to quit this program.
!2
Position:1 Error Code:5
Only '+' and '-' allowed at the beginning of the expression.
sin(1.2,3)
Position:8 Error Code:17
Too many parameters.
pow()
Position:4 Error Code:16
Too few parameter(s).
-(-(-(-(-(-5))))))!
=-120
pow(log((7+2)%4), sin(cos(tan(1))))
Position:35 Error Code:18
Illegal characters.
pow(log((7+2)%4), sin(cos(tan(1))))
Position:34 Error Code:19
Expression not finished.
5)(
Position:2 Error Code:1
')' not matched.
20!
=-2102132736
1/0
=inf
sqrt(-2)
=nan
sqr(3)
Position:1 Error Code:13
Function not defined.
(sin(1)cos(1)).5
=0.2273243567
```

本例验证了本程序对一些刁钻样例的处理情况。验证了本程序对于不合乎语法的输入的语法分析情况。检验了本程序对于非法运算的输出情况。从测试中看到，本程序对于整形变量的溢出没有进行处理，这是本程序的一个缺点。

4. 总结

1. 调试时出现的问题及解决的方法：

本程序设计初期遇到的一大障碍是，由于自行设计的队列与链表的设计不完善，在运行时会抛出 runtime error。而用 STL 中的队列与链表则不会出现这些问题。我对这一现象进行了思考，认为最可能是我自行设计的类的构造函数或者复制构造函数编写得不完善。在我的计算机上观察到的现象是，如果我用自定义的队列存储基本数据类型的变量，如 `queue<int>`，则在执行过程不会出错。如果我用自定义的队列存储自定义类型的变量，如 `queue<term>`，则在入队、出队等操作中常常是要么结果不正确，要么抛出 runtime error。而对于 `queue<term*>` 则不会出现错误。

在编写 calculator 类的 calculate 函数时，我遇到了类似的问题。如果我在

calculate 函数中写 expression e;则遇到递归调用 calculate 函数的时候, 程序输出的错误信息的定位信息常常不正确。而如果在 calculate 函数中写 expression* e=new expression;就消除了这个问题。

对于这个问题的深层次原因, 我还不能很好的解释。队列在入队和出队时, 隐性地用到了复制构造函数, 而我自定义的类没有实现复制构造函数, 这似乎是第一种情况下出错的原因。但这并不能解释 calculate 函数递归调用输出的信息出错的原因。将 queue<term>改为 queue<term*>, 规避了复制构造 term 的问题, 避免了运行时错误, 但我对于在第二种情况下, expression e 会出错, expression* e=new expression 则不会出错的原因仍不甚明了。

在程序设计之初, 我对于 expression 类进行调试时, 发现其 calculate 计算得到的值经常是不正确的, 通过打印中间变量的方法, 我最终发现这是因为编写代码方面的粗疏造成的。由于不熟悉 union 的使用方法, 我在求得的值储存在 union 的整形变量成员中的情况下, 打印的是 union 的浮点型成员的数值, 这就造成了运行结果的错误。

后期调试中出现的错误, 在 3.2 测试条件部分的叙述中已经提及。主要是通过输出中间变量, 准确定位出错的位置, 再对于特定的程序段重新进行思考和分析来解决这些问题。

2. 心得体会:

本次程序设计带给我成长上的收获。一开始本程序的设计进展缓慢, 但当我完成了 expression 类的设计, 人为地用 expression 类的 push 和 calculate 方法实现了表达式的求值的时候, 我感觉内在的热情点燃了。接下来我用了一个晚上就把剩下的主体工作完成了。我体会到了有些同学通宵玩游戏时体会到的那种兴奋。可以说本次程序设计实现了我自己的想法, 也极大地增进了我对于编程的兴趣。

从对于程序设计的理解方面看, 本次程序设计让我对于面向对象的程序设计思想有了更深的理解。我深深地感到, 在真正编程之前, 对于自己所要实现的一类进行通盘规划的重要性。一个设计良好的类能够清晰而准确地定义该类对象的属性及其在各种情况下的方法和行为。虽然设计这一部分的代码有点像在黑暗中凭着想象力摸索, 但这种探索的回报是很大的, 它可以让后面的代码编写变得相对轻松, 也可以很大地提升所编写的程序的可扩展性。这些好处, 并不是一句“提高了代码的重用性”就能够说清的。

数据结构方面的知识让我能够比较容易地从“工具箱”中选出所需要的, 作为自己设计的类所参考的结构。比如, term 的设计参考了第四章的广义表, termList 虽然有专为表达式打造的功能函数 mergeAdjacent, 但其基本结构仍然是第二章的链表。从左到右读取表达式, 并且从左到右进行运算这件事, 让我想到了第三章的队列。我将 function 类的设计和第七章的平衡二叉排序树的想法融合在一起, 使得我能够完成对于任意函数表的搜索匹配, 这使得我的程序能够支持更为一般的, 形式各异的自定义函数, 增加了我的程序的可扩展性, 而为此, 我只需编写一些思路清晰而简单的代码, 而无需大大增加代码的复杂程度, 我想, 这就是数据结构课程学习带给我的好处。

通过此次程序设计, 我还深深地体会到了编写伪代码的好处。实际上, 编写伪代码能够将漂浮在头脑中的想法落到实处, 让人在完成一些具体的工作的同时得到一种掌控感。有时, 正是在将一些细节落实的过程中, 我对于程序设计的整体思路又有了新的想法。对我个人而言, 编写伪代码时心态更加轻松, 容易写出头脑中的直觉, 而且可以说确实是“好记性不如烂笔头”, 落实在纸上, 本身就提

高了人的记忆力、综合力和思考能力。

编写伪代码帮我理清了设计思路，帮我把一个函数分解为不同的操作模块，最终我的 `calculate` 函数仍然有伪代码设计的痕迹。

平常的我不是一个特别重视细节，能够保存各种资料和草稿的人，然而像草稿一样编写的伪代码确实极大地提高了我的工作效率，让我在头脑没有那么清晰的时候，仍然能够比较准确有效率地将原先的想法实现。

3. 下一步的改进：

(1) 继续排除错误，首先是解决非法运算不提示、不报错的问题。

(2) 从时间复杂度的分析看，本程序耗时最大的部分是递归调用和子字符串复制、重复遍历的过程。如果把递归调用改成用栈实现，就会提升本程序的性能。

从设计思路上看，用栈实现并不十分困难，只要写一个 `stack<expression>`，每次读到左括号时新的空表达式 `expression` 入栈，每次读到右括号时栈顶表达式出栈，即可替代 `calculate` 函数的递归调用。然而另一方面，牵一发而动全身，用栈改写还需要一定的工作量，需要相应的改写、推敲细节和调试的过程。

(3) 从增加程序的可扩展性的角度看，下一步应当探索对于变长变量表的函数的求值的实现。这一功能的实际意义是统计求平均、求方差等等。

关于支持变长参数表函数的求值，我的设计思路是，为此类函数定义一个特定的 `attribute`，当 `readParameter` 函数读到此类 `attribute` 的 `term` 时，不检查参数个数的匹配问题，每逢逗号便归约一次参数，将归约得到的所有参数保存在数组中，遇到右括号时，将最后一个参数进行归约，保存到参数数组中，统计参数个数，将参数数组和参数个数传递给函数指针进行计算。

(4) 思考怎样让函数名支持的字符从仅限于字母变成字母、数字和其他字符混编。

(5) 思考怎样支持函数名以外的功能性字段，比如声明二进制，等等。

(6) 思考更复杂的语法系统，以支持 `f(1)` 到 `f(n)` 的求和以及积分的近似运算。最好是能让用户在程序界面上写几个字符和参数，就能够实现关于函数求值的一些循环、累加、累乘的操作。比如，用 `f(1: n)` 表示对于函数值的循环遍历，等等。这样就避免了用户每次编写代码来实现相应的效果。

(7) 思考怎样让表达式中允许出现变量，怎样让程序理解这些变量的赋值：理论上讲，可以让程序在字符串匹配出错的时候不是设置错误标记，而是建立一个自定义变量的二叉排序树，二叉树的每个节点表示一个自定义变量，该节点应当关联一个 `termNode` 指针队列，像书签一样地表示自定义变量每次出现的位置。当出现赋值表达式时，在二叉排序树中搜索到该自定义变量的节点，将队列中保存的全部位置替换为赋予的数值。

(8) 思考能否通过表达式解析实现解方程。

(9) 思考在计算器框架下，对于复数、多项式类的支持。

(10) 增加大数类，重新编写相关运算与函数的实现，函数考虑用幂级数进行近似计算，探索实现高精度计算的计算器。

(11) 令计算器将最近几次运算结果保存在 `ans` 数组中，可供用户直接使用。

5. 附录：

本文成文以后，我又尝试着为本程序添加了对于不特定个数的参数求和，求平均，求方差和标准差的功能。我初步实现对于一元、二元函数求数值积分近

似值的功能。另外，我又往本程序中添加了 4 个支持的函数 `cbrt`, `cosh`, `sinh`, `tanh`，并且发现和改正了 `match` 函数中的一个设计漏洞。

其初步测试结果如下：

对于统计功能的验证：

```
F:\BUPT\Data structure\exprimp\expr.exe
Scientific Calculator
Operators supported: + - * / ^ % ! ()
Functions supported: abs acos asin atan avg cbrt cos cosh exp
Enter '@' to quit this program.
avg(99.7,96.4,94.2,95.8,90.2,-4,3!)
=68.32857143
sum(99.7,96.4,94.2,95.8,90.2,-4,3!)
=478.3
var(99.7,96.4,94.2,95.8,90.2,-4,3!)
=2131.802381
sd(99.7,96.4,94.2,95.8,90.2,-4,3!)
=46.17144552
```

对于数值积分功能的验证（格式：函数名（积分下限：积分上限：步长，...））：

```
F:\BUPT\Data structure\exprimp\expr.exe
Scientific Calculator
Operators supported: + - * / ^ % ! ()
Functions supported: abs acos asin atan avg cbrt cos cosh exp
Enter '@' to quit this program.
cosh(0:1:0.001)
=1.174929751
cosh(0:1:0.00001)
=1.175213909
sinh(0:1:0.001)
=0.5424930795
sinh(0:1:0.00001)
=0.5430865108
tanh(0:1:0.001)
=0.4333999851
tanh(0:1:0.00001)
=0.4337846384
```

可见，在“下一步的改进”中的一些功能得到了初步的实现。当然，这一部分的功能还需要测试验证，目前只支持输入文法正确的字符串，对于错误的文法分析功能还不健全。

原有程序，作为功能稳定版，改进程序，作为功能增强版，分别在程序文件中提供。