自由函数计算器

广州致远电子有限公司 李兵

摘 要：根据表达式中基本元素的特征，将数学表达式分为14种基本元素，采用Poland表达式的计算方法，首先拆分表达式，然后将中缀表达式转化为后缀表达式，最后逐步进行计算，直至输出结果。增加对大数值计算自定义运算符的支持。

关键词：计算器；控制台；波兰表达式；

Abstract**：**This article introduced

Key words：Calculator;Console;Poland Expression;

目录

[**1** 引言 3](#_Toc333557276)

[**2** 表达式分析 3](#_Toc333557277)

[**3** 总体方案 4](#_Toc333557278)

[**3.1** 分步计算 4](#_Toc333557279)

[**3.2** 二叉树计算 4](#_Toc333557280)

[**3.3** 波兰表达式计算 5](#_Toc333557281)

[**3.4** 方案比较 6](#_Toc333557282)

[**4** 数据结构 6](#_Toc333557283)

[**4.1** 常用操作符 6](#_Toc333557284)

[**4.2** 自定义操作符 7](#_Toc333557285)

[**4.3** 变量 7](#_Toc333557286)

[**4.4** 命令 8](#_Toc333557287)

[**4.5** 数值型操作数 9](#_Toc333557288)

[4.5.1 大数值类BigDouble 9](#_Toc333557289)

[4.5.2 精度无限型大数值类BigDouble 9](#_Toc333557290)

[4.5.3 数值型操作数存储 10](#_Toc333557291)

[**4.6** 表达式基本元素队列 10](#_Toc333557292)

[**5** 字符串输入 10](#_Toc333557293)

[**6** 拆分表达式 11](#_Toc333557294)

[**6.1** 标记负号 11](#_Toc333557295)

[**6.2** 提取操作符 12](#_Toc333557296)

[**6.3** 提取变量型操作数 12](#_Toc333557297)

[**6.4** 提取数值型操作数 12](#_Toc333557298)

[**6.5** 空格处理 13](#_Toc333557299)

[**6.6** 一行语句多条表达式 13](#_Toc333557300)

[**7** 中缀表达式转化为后缀表达式 13](#_Toc333557301)

[**7.1** 转化规则 13](#_Toc333557302)

[**8** 后缀表达式计算 13](#_Toc333557303)

[**8.1** 求值表达式 14](#_Toc333557304)

[**8.2** 赋值表达式 14](#_Toc333557305)

[**8.3** double到BigDouble类型的切换 14](#_Toc333557306)

[**9** 命令系统 14](#_Toc333557307)

[**10** 自定义操作符 15](#_Toc333557308)

[**10.1** 自定义运算符的定义 15](#_Toc333557309)

[**10.2** 自定义运算符的计算 15](#_Toc333557310)

[**11** 结语 15](#_Toc333557311)

# 引言

在日常生活中，经常会用到计算器，微软自带的计算器，需要一步一步的计算，运算效率低下，并且不能存储变量。Matlab面世之后，即使是很小的计算题，我们都宁愿使用Matlab，这种草稿纸式的使用方式，给我们带来极大的方便。但是Matlab软件是在太大，在性能普通的计算机上，需要很长的启动时间，影响用户的使用效率。

本文设计的表达式计算器，则刚好可以解决这一难题，可以完成基本的表达式的计算，保证和Matlab兼容的输入方式，增加对大数据和自定义运算符的支持，并且软件比较小，启动快，使用方便。

# 表达式分析

表达式可以分为操作数和操作符两大类。

操作符按功能分为常用运算符、函数、用户自定义运算符，括号，按操作数个数又分为单目操作符和双目操作符。其中单目操作符又分为左操作符和右操作符。

操作数按功能分为数字、常量、用户自定义变量，按数据类型分为通用数据类型和用户自定义数据类型。

在本程序中，由于要求进行大数值的计算，因此这里自定义大数值类BigDouble类型，保持程序的稳定性和运行效率，正常情况下使用double类型进行计算，只有在double无法表示时才使用BigDouble类型。

基于以上认识，将表达式中可能出现的基本元素分为14种类型，用一个枚举类型来表示。其中表达式结尾符(分号)本应归类到运算符中，为了使用方便，自己作为一类。变量赋值时，变量可能第一次出现，并没有值，归为Undefine（未定义）一类。常用函数中只有pow是二元操作函数，不再单列出一类，只在计算时单独处理。

enum KindType{

OperandDouble, /\*\*< double型操作数 \*/

OperandBigDouble, /\*\*< BigDouble型操作数 \*/

ConstVariable, /\*\*< 常量 \*/

VariableDouble, /\*\*< double型变量 \*/

VariableBigDouble, /\*\*< BigDouble型变量 \*/

Undefine, /\*\*< 未定义变量 \*/

UnayOprator, /\*\*< 一元运算符 \*/

BinOperator, /\*\*< 二元运算符 \*/

UserDefineUnayOprator, /\*\*< 用户自定义一元运算符 \*/

UserDefineBinOprator, /\*\*< 用户自定义二元运算符 \*/

LeftParen, /\*\*< 左括号 \*/

RightParen, /\*\*< 右括号 \*/

Function, /\*\*< 函数 \*/

EndExpression, /\*\*< 表达式结尾符 \*/

};

# 总体方案

## 分步计算

第一种方案的设计思路是将用户自定义运算符、变量、函数、括号、四则运算表达式分步处理，每次只处理一种类型并将计算结果替换成字符串，然后进行下一类型的处理，直至数学表达式处理结束，输出结果。思路简单，但是需要多次扫描字符串进行解码，运算效率低下。



图 1 分步计算流程

## 二叉树计算

第二种设计方案是使用二叉树构建表达式树。表达式树有操作数和操作符构成，其中函数当做二元或者一元操作符处理，变量当做操作数处理。

图 2 表达式树



构建表达式树时，将操作数作为二叉树的叶子节点，操作符作为内部节点。对于每个操作符左子树中包含左子树中的所有简单操作数和操作符，而右子树中包含右子树中的所有操作数和操作符。

图 3 二叉树计算流程



## 波兰表达式计算

第三种设计方案是使用波兰表达式来求解相应的表达式的值。

将表达式的操作符写在操作数之前或者之后的表示方法称为波兰表示法，二者分别称为前缀式和后缀式，常用的表达式的书写方法，操作符在操作数之间，称为中序式。如a+b化为后缀表达式为ab+。处理过程如图 4。

首先将数学表达式拆分成基本元素，然后将中缀表达式转化为后缀表达式，最后按照后缀表达式的运算规则，从左到右依次计算，直至输出结果。本程序采用第三种方法实现。

图 4 波兰表达式计算流程



使用波兰表达式进行求解，第一步要将表达式拆分成各个基本元素，第二步对拆分成基本元素的中缀表达式进行排序，将其转化为后缀表达式，第三步从左到右依次计算，直到遇到表达式结尾符，输出结果。

按照面向对象的思想，构建一个波兰表达式的类，添加三个处理函数，分别完成以上三个过程。

void BreakExpress(string sExpress); /\* 将表达式拆分成基本元素 \*/

void Infix2Postfix(); /\* 排序，将中缀式转化为后缀式 \*/

bool Postfix2Number(Variable& AnsVariable); /\* 后缀式求值 \*/

图 5 波兰表达式计算样例



## 方案比较

第一种方案逻辑上易于理解，最容易想到，但是需要多次扫描字符串来进行解码，运算效率低下，编程量较大。

第二和第三种方案是比较成熟的解决方案，具有固定的计算顺序和规则，不需要对表达式进行重复扫描来进行解码，并且经过初步翻译之后，不需要再考虑括号和优先级，表达式的求值可以获得非常高的效率。

本程序选择第三种方案，即采用波兰表达式的方案进行计算。

# 数据结构

根据以上表达式基本元素的分类，定义类型结构体。

## 常用操作符

struct Operator{

char cName[6]; /\*\*< 运算符名字 \*/

KindType kind; /\*\*< 运算符类型 \*/

int iPriority; /\*\*< 运算符优先级 \*/

}; 操作符类型，包含常用运算符，函数，括号，表达式结尾符，但是不包含自定义数据类型。

本程序支持的常用操作符如下,

表 1 常用操作符

|  |  |  |
| --- | --- | --- |
| 操作符 | 优先级 | 类型 |
| 表达式结尾符 ; | 9 | EndExpression |
| ( | 8 | LeftParen |
| ) | 8 | RightPare |
| 所有一元操作符 # ~ ! | 7 | UnayOprator |
| ^ | 7 | BinOperator |
| 所有函数 | 7 | Function |
| \* / % | 6 | BinOperator |
| + - | 5 | BinOperator |
| == != | 4 | BinOperator |
| < <= > >= | 3 | BinOperator |
| && || | 2 | BinOperator |
| = | 1 | BinOperator |
| , | 0 | BinOperator |

其中#是负号的替代形式，将负号和减号视为两个不同的运算符。

使用一个结构体数组存储所有的Operator,作为常用操作符字典。

const Operator OperatorDic[OPERATORMOST] = {……};

## 自定义操作符

自定义操作符，需要用户定义如何进行计算，那么当前的运算符就应是自定义运算符的一个子集，因此自定义操作符相对于常用操作符多了一个string类型表达式，计算机靠这个表达式存储自定义操作符的运算过程，同时为了区分左右操作数，将左右操作数的变量名一起作为参数存储到结构体里。

struct UserDefineOperator{

char cOperator; /\*\*< 运算符 \*/

KindType kind; /\*\*< 用户自定义运算符的类型 \*/

int iPriority; /\*\*< 用户自定义运算符的优先级 \*/

char cLeftOperand[33]; /\*\*< 用户自定义运算符的左操作数 \*/

char cRightOperand[33]; /\*\*< 用户自定义运算符的右操作数 \*/

string sExpress; /\*\*< 用户自定义运算符的表达式 \*/

};

用户自定义运算符使用向量存储，长度自动扩展，避免空间的浪费，同时无须限制自定义运算符的数目。

vector<UserDefineOperator>UserOperatorVector; /\*\*< 存储用户自定义运算符 \*/

## 变量

用户定义变量和常量都需要存储变量值和变量名。变量的值可能会在计算中从一个double类型切换到BigDouble类型，因此Variable中含有一个double类型和一个BigDouble类型，由类型kind来确定哪个是有效的。

变量名的要求和C++保持一致，只能是字母、数字、下划线，并且首字符不能是数字。

由于常量只有pi和e，不再单独定义一种类型，而是和用户自定义变量使用同一种类型。虽然浪费一定的空间，但是便于扩充。

struct Variable{

char cName[33]; /\*\*< 变量名 \*/

KindType kind; /\*\*< 变量类型 \*/

double dNumber; /\*\*< double型变量的值 \*/

BigDouble bdNumber; /\*\*< BigDouble型变量的值 \*/

};

常量个数一定，并且不允许被修改，使用一个结构体数组来存储，作为常量字典。支持方的常量如表2，区分大小写。

表 2 常量

|  |  |  |
| --- | --- | --- |
| 常量 | 类型 | 常量值 |
| pi | ConstVariable | 3.141592653589793 |
| e | ConstVariable | 2.718281828459046 |

const Variable VariableDic[CONSTVARIABLEMOST]= {

{"pi",ConstVariable,3.141592653589793,BigDouble(3.141592653589793)},

{"e" ,ConstVariable,2.718281828459046,BigDouble(2.718281828459046)},

};

用户自定义变量个数未知，使用一个向量存储，变量个数可以任意增加。

vector<Variable>VariableVector; /\*\*< 存储用户定义变量 \*/

## 命令

一个完整的控制台软件，出于用户的需求，需要添加相关的命令，比如，查看变量、清除变量、清屏、自定义运算符。定义命令类型，用于保存所有的命令、含义和用法。

struct Command{

char cName[10]; /\*\*< 命令名 \*/

string sCommandDetail; /\*\*< 命令含义 \*/

string sExample; /\*\*< 命令用法示例 \*/

};

使用一个结构体数组存储所有命令，作为一个命令字典。

const Command CommandDic[COMMANDMOST]={

{"help" , "Display help information" , "help whos \nhelp" },/\*\*< 查看帮助 \*/

{"whos" , "Lists all the variables" , "whos" },/\*\*< 查看所有变量 \*/

{"who" , "List current variables." , "who variableName" },/\*\*< 查看指定变量 \*/

{"clc" , "Clear command window." , "clc" },/\*\*< 清屏 \*/

{"clear", "Removes all variables" , "clear variableName \nclear" },/\*\*< 清除变量 \*/

{"operator" ,"Define user's Operator" , "operator $ a$b 1/2\*pi\*a\*b 4"}, /\*\*< 定义运算符 \*/

{"copyright", "Display copyright declaration" , "copyright" },/\*\*< 版权声明 \*/

{"history" , "Display history expression." , "history" },/\*\*< 历史记录 \*/

{"exit", "Exit from the console" , "exit" }, /\*\*< 退出程序 \*/

{"test" , "Auto test the programm" , "test" },/\*\*< 自动测试 \*/

};

## 数值型操作数

本程序支持大数值计算，因此自定义了大数值类BigDouble作为中用户定义数据类型，参与计算，double是BigDouble的一个子集。但是为了提高程序的稳定性和效率，保留double数据类型，并作为运算的默认数据类型，只有当double溢出或者表达式中出现double无法表示的数值时，才切换到BigDouble类型进行计算。

### 大数值类BigDouble

构建大数值类BigDouble，一种方法是使用科学计数法，Z = x\*10^y,假如基数x使用double类型，指数y使用int类型，则可表示的数的大小是1.79\*10^308\*(10^(2^31))，这个数大小在理论上可以达到10^900000000,而且在运算时性能几乎没有太大损失，因为加减乘除等仍可以转化为对基数和指数的加减乘除。缺点是有效位数只能达到double类型的精度即16位，适合于要求进行大数据运算，但是对精度要求不是特别高的场合。

为了使得构建的类便于使用，添加了多种构造函数，可以使用常用数据类型double、int、bool以及char\*、string等直接构造BigDouble类型，其中对于超大数值（超过308位）只能使用char\*或string类型直接构造BigDouble。

BigDouble( bool bNumber);

BigDouble( int iNumber);

BigDouble(double dNumber);

BigDouble(string NumberStrStr);

BigDouble( char\* NumberStr);

BigDouble(double dBaseNumber , int iIndexNumber);

首先要规定BigDouble的标准形式，要求小数点左侧只有一个有效数字，此时为BigDouble的标准形式。

对常用的运算符进行重载，按照科学计数法的计算规则，进行加减时，首先将指数化为一直，然后对基数进行加减。进行乘除时，对基数进行乘除运算，对指数进行加减运算。进行逻辑运算时，0为假，非0为真。

由于大数值采用的是损失精度的存储方式，因此进行三角函数计算没有意义，不支持三角函数运算、取模、取反运算。

### 精度无限型大数值类BigDouble

参照IEEE754标准，使用一个基数向量vector<int>BaseVector，一个指数int iIndex，和一个符号int iSign表示一个高精度的浮点数，构建一个10000进制的大浮点数，其中基数向量中存储大数值的每一位有效数字。

选择10000为进制的理由，int表示范围-10^9~10^9，以10000为进制时，每一位进行乘法，加法，减法时，int型都不会溢出；假如进制更小如10进制，则使用的内存会成倍上升，查找和排序等复杂度为O(2)的程序耗时也会上升；假如使用更大的进制，需要对溢出进行特别处理，使得运算变得复杂。

数值的标准形式，假设BaseVector中的每一个元素都是0-9999之间，向量的头部和尾部均不是零（当向量中就一个元素时，允许为0），指数只能是4的倍数，此时即为标准形式。

为了保证程序中两个BigDouble类的兼容性，二者采用完全一样的接口函数，只是实现的方式不同。

运算符重载，进行加减运算时，由于存在符号位，因此需要进行符号位判断，转化为同号数字的加减运算，对于减法，还要进行大小判断，使得总是大数减小数。然后将指数化为一致，假如不一致则对指数大的数的BaseVector进行移位，每移一位相当于乘以10000，指数减去4，直到指数一致，进行逐位计算，然后归一化。

对于乘法运算，移位，相乘，求和。

对于除法，使用递归调用的方式。但是此法非常的消耗堆栈，一般运算到小数点后500位堆栈就会溢出，需要将递归改为迭代的方式。

本程序有两个版本，最初出于性能考虑，选择第一种方法，并对常用的运算符进行了重载，虽能处理超大的数，但是精度有限，后来又编写了有效位超大的大数值类，用于提高运算结果的精度，为了保持兼容，第二次构建的BigDouble类完全按照第一版的接口，实现了第一版所有的接口函数，并对第一版用到的运算符进行重载，保证，第二版在主程序不加改动，只需替换掉BigDouble类就可以运行。

### 数值型操作数存储

数值型操作数使用向量来存储，在Poland类型定义两个vector，分别存储double型数值和BigDouble型数值，个数不限。

vector<double>DoubleVector; /\*\*< 存储double型操作数 \*/

vector<BigDouble>BigDoubleVector; /\*\*< 存储BigDouble型操作数 \*/

## 表达式基本元素队列

由于表达式的基本元素属于不同的类型，不同的类型只能存储在不同的数组或者向量中，因此还需要两个索引队列存储每个基本元素的位置，一个队列存储基本元素类型，由此可知基本元素存储在哪个数组或者向量中，另一个队列存储在数组或者向量中的下标。由二者可以去相应的数组或者向量中取出基本元素。切记第二个队列不能存储指针，因为vector向量的基本思想是每次动态申请相同大小的内存，当内存不够用时再申请一块更大的内存，并将当前的元素拷贝到新的内存中，因此如果第二个队列存储的是指针的话，指向的值一直在变化。

vector<int>IndexVector; /\*\*< 存储表达式元素的下标 \*/

vector<KindType>KindTypeVector; /\*\*< 存储表达式元素的类型 \*/

当由中缀表达式转化为后缀表达式时，并没有改变元素的存储位置，而是改变它的两个索引队列，同样使用两个向量vector存储后缀表达式的索引。

vector<int>PostfixIndexVector; /\*\*< 存储后缀表达式元素的下标 \*/

vector<KindType>PostfixKindTypeVector; /\*\*< 存储后缀表达式元素的类型 \*/

# 字符串输入

由于无法预知用户输入的表达式的长度，因此表达式的读入不能使用char型数组，太长会浪费空间，太短又会限制用户的输入长度。输入时类型采用c++的string类型，可以不限用户输入的长度。

由于允许用户为了美观在操作数和操作符之间插入空格，因此字符串的输入允许出现空格，不能使用标准输入流对象cin，需要使用getline函数或者getchar。

string sExpress;

getline(std::cin , sExpress);

在进行字符串的处理时，常常需要申请新的内存空间，简单的做法时，估计用户的大概需求，申请一块稍大于需求的空间，但是此方法，空间浪费明显，并会限制用户输出的字符串的长度，且不易扩充。为了不限制用户输入的字符串的长度，本程序在遇到有内存需求时，统一使用C++关键字new申请动态内存，可以按照需要的大小进行内存申请，避免了空间的浪费，且无需限制用户的输入长度，扩展性好，缺点是动态内存申请，速度较慢，运算速度略微降低。在内存使用结束之后要及时使用关键字delete进行内存回收，避免内存泄露。

# 拆分表达式

表达式中含有操作数和操作符，首先识别操作符，然后任意另个操作符之间或者表达式开头和末尾则认为是操作数。

图 6 拆分表达式流程



## 标记负号

负号和减号具有不同的优先级，不同的运算特性，但是计算机并不能区分减号和负号。

第一种方法是遇到“-”时，如果前面是加减，就在前面加上0，变成0-，如果前面是乘除，就将负号前移，直到遇到加减号。本程序第一版使用此法处理，对于只有含四则运算的表达式还好处理，假如表达式中含有所有常用的操作符，括号，函数等，逻辑上会非常的复杂，很容易出错或者疏漏。

第二种方法是，将二者作为两个操作符来处理，他们具有不同的优先级和运算特性。在开始运算之前，首先扫描表达式，如果“-”出现在表达式开头、运算符、函数、左括号之后，则认为是负号，使用另一个符号“#”代替，以后遇到#就当做负号处理。此法使用起来逻辑上容易理解，并且易于实现。

使用函数来void MarkNegative(char \* cExpress);完成

## 提取操作符

由于存在&&等两个字符的操作符和多个字符的函数，因此使用char型数组来存储操作符的名称。

在搜索操作符时，假如从字符串的任意下标开始进行全词匹配，效率会很低。为了提高效率，对于中的每一个字符，首先与操作符中的首字符比较，若相同，再进行全词匹配。

有些操作符如 == 和 =，假如先匹配=，那么 == 就永远不会被匹配到。为了解决操作符覆盖的问题，直接改变存储顺序，将可能被覆盖的操作符放在前面。例如将 == 存储在 = 之前，= 首先尝试匹配==，然后再尝试匹配=，这样就不会出现被覆盖的问题。

## 提取变量型操作数

提取两个相邻操作符之间的字符串，字符串如果长度非零，则可能是一个操作数，首先判定该操作数是否一个变量。

首先验证字符串是否是一个合格的变量名，要求只能有字母、数字、下划线，并且首字符不能是数字。

如果是合格的变量名，然后搜索常量字典VariableDic，如果是常量，将类型和下标存入KindTypeVector和IndexVector；如果不是常量，搜索存储变量的向量VariableVector，如果是用户自定义变量则将类型和下标存入KindTypeVector和IndexVector；否则，则是一个未定义变量，类型就是Undefine，在VariableVector中存入新的变量，并将类型和下标存入KindTypeVector和IndexVector。

## 提取数值型操作数

程序支持十六进制、十进制、八进制、二级制等多种进制，要求添加不同进制数字的前缀后者后缀。首先根据前缀后缀的格式判断操作数的进制。

表 3 支持的数据格式

|  |  |
| --- | --- |
| 进制 | 格式 |
| 十六进制 | 0x前缀或者h后缀 |
| 十进制 | 无前缀后缀或后缀d |
| 八进制 | 后缀o |
| 二进制 | 后缀b |

前缀和后缀均不区分大小写，多种进制均支持整型和浮点型型输入，如0xff.ff。当十六进制的整型使用后缀h时，如果首字符又是字母，如AB3H，完全符合变量名的规范，因此会被当成变量处理，整型十六进制输入时，如果会产生歧义，则需要在后面加上点，如AB3.H，或者使用0x前缀，以免产生歧义。

将多进制数值字符串转化为数值，首先扫描字符串，确定小数点的位置，如不存在，则默认小数点在字符串末尾的下一位，这样就将整型和浮点型统一处理。将每一位的值乘以该位的权值，求和即可得到数值。

每一位的值的获取，根据ASCII表，如果是数字型字符character，character – '0',即为该值，若是十六进制，会ABCDEF等字母型字符，tolower(character)-'a'+10即为该位的值。

权值计算，pow(进制，偏移)，如十六进制小数点前面第二位的权值为pow(16,1)。

## 空格处理

用户在输入表达式时，为了美观一般会在操作数和操作符之前插入一个或多个空格，此时程序应能够自动处理，但是空格若在操作数内部或者操作符内部会产生歧义，则需要报错。如1 + 2应是对的，但是1 = =2就是错的。

当提取两个操作符之间的操作数时，首先从两端开始搜索，剔除空格，直到遇到有效字符。然后从剔除两端空格之后的字符串中提取操作数。

## 一行语句多条表达式

为了提高使用效率，允许用户在每一条字符串语句中输入多个表达式，但是要求相邻两个表达式之间使用表达式结尾符分号进行分割。

在进行计算式，遇到表达式结尾符，则表示一条表达式计算完毕，输出结果。然后开始下一条表达式的计算。

# 中缀表达式转化为后缀表达式

表达式基本元素的顺序是靠索引的顺序的确定的，因此，将中缀表达式转化为后缀表达式实际上是对类型索引和下标索引的重新排序，对两个索引的操作要同步进行。

## 转化规则

新建一对空的索引堆栈，从输入索引向量的第一个元素开始扫描，

1、遇到操作数，立刻放入到输出索引向量中

2、遇到左括号，立刻放入索引堆栈中

3、遇到右括号，将索引堆栈中，匹配的左括号之后的操作符逐个出栈，放入输出索引向量中，将右括号直接丢弃

4、遇到操作符，将索引堆栈中优先级大于等于它的操作符出栈，放入到输出索引向量，直到遇到优先级小于它的操作符。常用运算符和用户自定义运算符同等处理。

5、遇到表达式结尾符，直接将索引堆栈中的所有操作符出栈，放入到输出索引向量。

6、特殊情况，赋值运算符是从右向左赋值，与普通运算符不同，遇到等号，不能将前面的等号出栈。

# 后缀表达式计算

赋值语句和计算语句区别比较大，因此见后缀表达式分为赋值表达式和求值表达式。对于赋值表达式，截取后面的计算部分，依然是求值表达式。因此对于赋值表达式先计算后面求值表达式的结果，然后进行赋值操作。允许在一个表达式中含有连续赋值语句，如a = b =1；

## 求值表达式

表达式的基本元素存储在索引向量中，根据类型可以确定存储所在的数组或者向量，根据下标可以取出该基本元素。

vector<int>PostfixIndexVector; /\*\*< 存储后缀表达式元素的下标 \*/

vector<KindType>PostfixKindTypeVector; /\*\*< 存储后缀表达式元素的类型 \*/

表达式求值时，分为两种情况，如果所有的基本元素中不存在BigDouble类型的数值或者变量，则整个计算过程使用double类型的堆栈存储临时变量，并输出double类型的计算结果。如果存在BigDouble类型的数值或者变量，则使用BigDouble类型的堆栈存储临时变量，输出BigDouble类型的计算结果。

后缀表达式计算过程如下

1、新建一个double类型或BigDouble类型的堆栈

2、从头开始扫面基本元素，当遇到操作数，可以是常量、变量、数值，将操作数的值压入堆栈

3、当遇到操作符，如果是单目运算符，则从堆栈中弹出一个操作数，计算后重新压入堆栈，如果是双目运算符，则从堆栈中探出两个操作数，计算后将结果压入堆栈。

4、直到遇到表达式结尾符，堆栈中的唯一一个元素就是计算结果。

## 赋值表达式

当表达式中存在等号时，则是赋值表达式，首先要计算赋值的层数。

将赋值表达式等号后面的部分作为一个求值表达式计算，然后将返回的值逐个赋值给等号前面的变量，并根据返回值修改被赋值变量的类型和数值。

变量的定义和修改均可以直接使用等号运算符。

当使用等号运算符时，等号左侧只能是变量名，不能是数值或者操作符。

## double到BigDouble类型的切换

根据IEEE754标准，double数据类型含有64位，其中符号位1位，指数位11位，尾数位52位。其中指数位全0和全1具有特殊含义，分别表示无穷大和无穷小。根据此特点可以提取double数据类型的第2至第12位，判断是否是全1，如果是的则表示超过double的表示范围。

为了保证程序的稳定性和运算速度，当double数据类型能够满足要求时，使用double数据类型，当计算的结果溢出时或者表达式中含有大数值则使用BigDouble类型计算。

# 命令系统

为了便于用户使用，添加一些常用的命令，包括查看变量的命令whos和who，清除变量的命令clear，清屏命令clc，版权声明命令copyright，help命令，历史记录命令history和自定义运算符命令operator，退出命令exit，自动测试命令test。

当用户输入一行字符串，首先检测是否是命令，如果是命令，将后面的参数分别取出放入参数列表。然后针对不同的命令进行处理。

其中test命令，是为了提高程序的测试效率，将以大批写好的计算表达式放入test.txt中，test命令则是逐行读取test.txt中的表达式，计算，然后将计算结果输出到屏幕。为了便于比对，本程序很多细节尽量保持和数学软件Matlab保持一致，因此除了大数值和自定义运算符，和matlab都是通用的。测试者可以直接打开Matlab，将test.txt中的表达式放入matlab进行计算，然后比对结果。

# 自定义操作符

本程序所指的自定义运算符是用户已知，但是程序员未知的运算符。要求常用运算符是自定义运算符的一个子集。

自定义运算符的存储，使用一个string类型的表达式存储自定义运算符的运算过程。

## 自定义运算符的定义

使用命令operator进行运算符的定义，例如定义优先级为8的$运算符进行求椭圆面积的运算，输入：operator $ a$b 1/2\*pi\*a\*b 7，命令系统会自动解析参数列表，从语句中解析出左操作数名a，右操作数名b，优先级7，运算表达式1/2\*pi\*a\*b，然后将各个部分分别存入自定义运算符结构体，然后将结构体存入向量。

## 自定义运算符的计算

仔细分析会发现，自定义运算符的计算本质上就是一个表达式的计算，但是首先需要获取左右操作数。

在后缀表达式求值时，当遇到操作符时，从堆栈中取出操作数，计算，然后将计算结果存入堆栈。如果遇到自定义操作符，构建临时Poland类的实例，取出操作数，为Poland类的实例添加变量即左右操作数，然后按照Poland类的求法，拆分表达式，中缀式转化为后缀式，表达式求值，即可求出自定义运算符的计算结果。

自定义运算符放到最后处理，是因为自定义运算符的处理和普通表达式的运算过程一致，遇到表达式则构建一个新的类进行处理。但是这样存在一个问题，加入运算中包含运算符，则不能引用，因为完全是一个新的类，一种可行的方式，是将原来的Poland实例拷贝到新实例中，这样能够引用原有的已定义的运算符。

# 结语

通过对表达式的分析，将表达式的基本元素分为14种基本类型，然后构建相应的数据结构，按照拆分表达式，将中缀式转化为后缀式，后缀表达式求值的顺序，对表达式进行求解。最终完成了整个程序的编写。最终版本的表达式计算器支持几乎所有的常见的运算符，并支持大数值的计算和自定义运算符。

参考文献

[1] [李兵]自由函数计算器需求文档

[2]C++程序设计

[3]C++ Primer Plus

[4] C++STL程序员开发指南

[5] 现代C++程序设计

[6]Visual Studio 2010 CSDN