Journal of Wuhan Polytechnic

一种应用数组计算复杂算术表达式的算法设计

陈亭志，程利民

（武汉职业技术学院 机电工程学院，湖北 武汉 430074）

摘 要：算术表达式的计算是C语言程序设计中经典数学问题，计算机在解决这个问题时，一般是用逆波兰算法，把中缀表达式转化为后缀表达式，然后再使用栈计算实现，这种算法利用递归的思想，并结合栈复杂数据结构才能实现。设计一种算法，通过将运算符进行优先级排序，找到了复杂计算问题的简化模型，只需要应用循环和数组就可以实现复杂算术表达式的计算。

关键词：逆波兰；算术表达式计算；数组；优先级

中图分类号：TP301.5　　**文献标识码**：A　　**文章编号**：1671-931X（2021）06-0102-07

DOI：10.19899/j.cnki.42-1669/Z.2021.06.020

C语言程序设计是工科类专业大学生普遍开设的应用性非常强的基础课程，该课程不仅培养学生对C语言知识点的掌握能力，更能训练学生细致耐心的编程态度、工程问题与数学模型转化能力，以及逻辑思维能力和计算机思维能力，特别是应用计算机思维解决实际问题的能力，为后续专业课程的学习奠定扎实基础[1]。

一、问题的提出

计算机如何计算一个表达式呢，比如-35+12*6/3-4，计算机没有人脑那么智能，需要编程人员告诉具体步骤，因此需要设计一个算法，可以自动分析表达式的优先级和词串，然后根据词串和优先级，一步步解读这个表达式，实现计算机和人脑一样的计算功能。

本文就是用C语言的基础知识：数组、循环语句以及判断语句，实现计算机能快速计算从键盘输入的任意表达式，比如5+6*(3+4)-24/(5+3)，并正确输出这个表达式的结果，如图1所示：

```
5+6*(3+4)-24/(5+3)
5+6*(3+4)-24/(5+3)=44
```

图1　算术表达式运算结果示例

二、一般算法——逆波兰算法

一般在解决这个问题时，多数人是先把普通表达式转化为逆波兰表达式来实现的。人类最熟悉的一种表达式1+2，(1+2)*3，3+4 *2+4等等都是中缀表示法。对于人们来说，也是最直观的一种求值方式，先算括号里的，然后算乘除，最后算加减，但是，计算

收稿日期：2021-04-15

基金项目：2020年武汉职业技术学院课题“基于输出式学习理论的在线课程有效教学”（项目编号：2020YJ002）。

作者简介：陈亭志（1981- ），女，湖北咸宁人，武汉职业技术学院机电工程学院副教授，研究方向：机电一体化、职业教育教学设计；程利民（1965- ），男，湖北武汉人，武汉职业技术学院机电工程学院副教授，研究方向：机电一体化。

机处理中缀表达式却并不方便，因为没有一种简单的数据结构可以方便从一个表达式中间抽出一部分算完结果，再放进去，然后继续后面的计算[2]。

例如，a+b 就是一种中缀表达式，写成后缀表达式就是 ab+。再如 2+3*4+4 的逆波兰式是 234*+4+，(2+3)*4+4 的逆波兰式是 23+4*4+，(2+3)*(4+4) 的逆波兰式是 23+44+*，1 + 2 * (4 – 3) + 6/2 的逆波兰式是 1243–*+62/+，(a+b)*c–(a+b)/e 的逆波兰式是 ab+c*ab+e/–。

将看似简单的中序表达式转换为复杂的逆波兰式的原因就在于这个简单是相对人类的思维结构来说的，对计算机而言中序表达式是非常复杂的结构。相对的，逆波兰式在计算机看来却是比较简单易懂的结构。因为计算机普遍采用的内存结构是栈式结构，它执行先进后出的顺序[3]。

如 1 + 2 * (4 – 3) + 6/2 的逆波兰式是 1243–*+62/+，计算过程如下：

*+62/+　　// 中心运算符是 –，先计算 43–=4–3

=12(4–3)*+62/+　　//(4–3) 的结果入栈，用 A(=1) 表示

=12A*+62/+ // 中心运算符是 *，再计算 2A*=2*A

=12*A+62/+　　//2*A 的结果入栈，用 B(=2) 表示

=1B+62/+　　// 中心运算符是 +，再计算 1B+=1+B

=(1+B)62/+　　//1+B 的结果入栈，用 C(=3) 表示

=C62/+　　// 中心运算符是 /，再计算 62/=6/2

=C(6/2)+　　//6/2 的结果入栈，用 D(=3) 表示

=CD+　　// 中心运算符是 +，再计算 CD+=C+D

=C+D// 最后结果是 6

从上面分析可以看出用逆波兰法来实现一个表达式的四则运算有如下优点[4]：

第一，当有操作符时就计算，因此表达式并不是从左至右整体计算，而是每次由中心向外计算一部分，这样在复杂运算中就很少导致操作符错误。

第二，堆栈自动记录中间结果，这就是为什么逆波兰计算器能容易对任意复杂的表达式求值。与普通科学计算器不同，它对表达式的复杂性没有限制。

第三，逆波兰表达式中不需要括号，用户只需按照表达式顺序求值，让堆栈自动记录中间结果；同样的，也不需要指定操作符的优先级。

四、本文设计的算法——“三层数组间接寻址法”

从上面的分析可见，用逆波兰方法可以很好解决四则运算问题，不过这种方法需要将表达式由中缀表达式转化为后缀表达式，然后再使用栈计算。这两步要实现，需要用到堆栈，对于只学了 C 语言基础知识的学生，很难看懂这些代码，而且程序的代码长度有三四百行。

为了不用堆栈也可以实现四则运算，本文设计了一种巧妙的算法，定义了三个数组，分别存放要计算的表达式字符串、表达式中每个字符对应的优先级、计算过程的中间结果，通过一个算法找到了这三个数组之间的逻辑关系，实现了复杂计算器的计算，程序长度不到百行，提高了程序的执行效率，也体现了数组在实际编程中的巨大作用。

从逆波兰算法分析，要实现四则混合运算的功能，程序需要克服四个问题：

如何实现优先级计算，比如 –35+12*6/3–4，先计算负号‘–’，然后是乘号‘*’；

如何分割字符串，比如 –35+12*6/3–4，–35 是一个整体，12 是一个整体，即能识别多位数、运算符和错误字符，也就是词素化；

如何保存中间结果，比如上式中的 –35；

如何读取中间数据，然后根据优先级一步步计算出结果。比如 –35+12*6/3–4，先计算第一大步得到 –35，保存到中间数据，然后计算第二大步得到 12*6 也保存到中间数据，直到最后的结果。

（一）问题 1：实现字符串的优先级排序编号

为了实现根据运算符优先级来计算，程序设计了一个功能，将输入的运算字符串，根据字符运算优先级进行编码，这样在计算的时候就可以根据这个编码确定计算的先后顺序，见表 1。

表 1　运算符优先排序表

运算符	+–	* /	+–	0–9.	间接数据	无效符	结束符号
含义	正负号	乘除号	加减号	数字和小数点	运算中间的过程数据	多位数后面的字符	
优先级排序	14	13	12	4	3	1	0

为实现这个功能，在程序里定义两个字符型数组 char sr_ay[50] 和 char fz_ay[50]。

char sr_ay[50] 用来存放输入的字符串表达式，char fz_ay[50] 用来存放这个字符串表达式每个字符的优先级。

简单起见，先设定输入的字符串只含有正负号，加减乘除和数字 0–9、小数点，另外在计算过程中保存的中间数据称为间接数据，一个多位数后面的数字认为是无效符，比如 123 后面的 23 就是无效字符，23 和 1 构成一个数字，结束符号是数组 fz_ay[50] 的结束标志。

优先级根据 C 语言运算符的优先级顺序，约定

见表 1，正负号是单目运算符，优先级最高为 14，乘除符号的优先级是 13，加减号的优先级是 12，数字 0–9 的优先级是 4，间接数据的优先级为 3，无效符的优先级为 1，结束符号优先级最低为 0。

假设输入的字符串 1 为 sr_ay[50]=“–35+12*6/3–4”

则这个字符串对应的优先级 fz_ay[50]=[14 04 01 12 04 01 13 04 13 04 12 04 00]

假设输入的字符串 2 为 sr_ay[50]=“ –200+23*5”

则这个字符串对应的优先级 fz_ay[50]=[14 04 01 01 12 04 01 13 04 00]

算术表达式优先级数组示例见表 2 所示。

表 2　算术表达式优先级数组

输入字符串 1	sr_ay[50]=“ – 35+12*6/3–4”												
对应优先级	fz_ay[50]=[14 04 01 12 04 01 13 04 13 04 12 04 00]												
数组下标 nn	0	1	2	3	4	5	6	7	8	9	10	11	12
sr_ay[nn]	–	3	5	+	1	2	*	6	/	3	–	4	
sr_ay 含义	负号	数字	无效符	加号	数字	无效符	乘号	数字	除号	数字	减号	数字	结束符
fz_ay[nn]	14	04	01	12	04	01	13	04	13	04	12	04	0
输入字符串 2	sr_ay[50]=“ –200+2*–5”												
对应优先级	fz_ay[50]=[14 04 01 01 12 04 13 14 04 00]												
数组下标 nn	0	1	2	3	4	5	6	7	8	9			
sr_ay[nn]	–	2	0	0	+	2	*	–	5				
sr_ay 含义	负号	数字	无效符	无效符	加号	数字	乘号	负号	数字	结束符			
fz_ay[nn]	14	04	01	01	12	04	13	14	04	0			

那如何编写程序实现优先级的排序呢？

假如通过函数 gets(sr_ay) 从键盘输入一列字符串到数组 sr_ay，nn 表示这个数组的下标，ff=sr_ay[nn] 表示键盘输入的每个字符，通过分析，对数组 ff 里每个数据进行读取，只有三种可能性，具体分析如下：

第一种可能性是乘除。通过 if 语句的条件判断是否是乘除，只要是其中任何一个，条件成立，则优先级序号是 13，程序实现如下，注意这里的单引号不能省，表示是字符：

```
if(ff=='*'||ff=='/'){ fz_ay[nn]=13; }      // * / %，双目运算，优先级 =13
```

第二种可能性是’+’或’–’。这两个符号可能是正负号也可能是加减号，因此要分开判断。如果出现在 sr_ay 数组的第一个，就是正负号，优先级序号为 14；数组的非第一个字符，又分两种情况，如果前面那个字符是小数点或数字，则是加减符号优先级序号 13，其他情况则是正负号，优先级为 14。程序实现如下。

```
else if(ff=='+'||ff=='-'){
if(nn==0){ fz_ay[nn]=14; }  // 开头 +– 号，为单目运算符，优先级 =14
else{
if(sr_ay[nn-1]=='.'||(sr_ay[nn-1]>='0'&&sr_ay[nn-1]<='9')){
fz_ay[nn]=12; }   // . 0––9 后的 +– 为双目，优先级 =12
else{ fz_ay[nn]=14; }  // 其他情况下为单目运算符，优先级 =14
}
}
```

第三种可能性是数字或小数点，如果是有效数字，如 3+4 中数字 3 或数字 4，优先级序号是 4；如果是非有效数如 32+4 中的数字 2，优先级序号是 1，说明是无效符，程序如下：

```
else if((ff>='0'&&ff<='9')||ff=='.'){   // 首数字转为 4，其余数字转为 1
if(nn==0){ fz_ay[nn]=4; }  // 第一个字符为数字，是有效数字，优先级设为 4
else{   // 非首字符的判断方法
if((sr_ay[nn-1]>='0'&&sr_ay[nn-1]<='9')||sr_ay[nn-1]=='.'||fz_ay[nn-1]==4){
fz_ay[nn]=1; }   // 前面一个字符是数字或小数点，说明无效数字
else{ fz_ay[nn]=4; }  // 其他情况说明是有效数字
}
}
```

通过上述分析可见，通过两个数组以及 if–if else 语句里面嵌套 if–else 语句，可以实现将输入的字符串根据约定的优先级排序得到每个字符的优先级排

序。完整程序如图 2 所示，输入不同的运算字符串得到不同的优先级结果，如图 3 所示。

```
main()
{ char sr_ay[50]; char fz_ay[50];                    // 输入数组    每个字符串对应的优先级
  unsigned char nn,ff;
//=============================================================================== 整理输入字符串，分解优先级
  begin: gets(sr_ay);                                // 从键盘输入计算式
  nn=0;                                              // 分解字符串为优先级，便于计算
  while((ff=sr_ay[nn])!=0){
    if(ff=='*'||ff=='/'){ fz_ay[nn]=13; }            // * / %,  双目运算，优先级=13
    else if(ff=='+'||ff=='-'){
      if(nn==0){ fz_ay[nn]=14; }                     // 开头+-号，为单目运算符，优先级=14
      else{
        if(sr_ay[nn-1]=='.'||(sr_ay[nn-1]>='0'&&sr_ay[nn-1]<='9')){
          fz_ay[nn]=12; }                            // .  0--9后的 +-为双目，优先级=12
        else{ fz_ay[nn]=14; }                        // 其它情况下为单目运算符，优先级=14
      }
    }
    else if((ff>='0'&&ff<='9')||ff=='.'){            // 首数字转为4，其余数字转为1
      if(nn==0){ fz_ay[nn]=4; }
      else{
        if((sr_ay[nn-1]>='0'&&sr_ay[nn-1]<='9')||sr_ay[nn-1]=='.'){    //||fz_ay[nn-1]==4
          fz_ay[nn]=1; }                             // 非首数字=1
        else{ fz_ay[nn]=4; }                         // 首数字=4
      }
    }
    ++nn;
  }
  fz_ay[nn]=0;                                       // 添加结束符
  printf("sr_ay=%s\n",sr_ay);   printf("fz_ay=");
  for(nn=0;fz_ay[nn]!=0;++nn)
  {
    printf("%3d",fz_ay[nn]);}
    printf("\n") ;                    // 浮点数打印
  goto begin;
}
```

图 2　字符串优先级排序程序

```
-200+2*-5
sr_ay=-200+2*-5
fz_ay= 14  4  1  1 12  4 13 14  4
-35+12*6/3-4
sr_ay=-35+12*6/3-4
fz_ay= 14  4  1 12  4  1 13  4 13  4 12  4
```

图 3　优先级排序程序运行效果

（二）问题 2：分割语素

通过前文的分析，当一个语素由多个字符组成，后面字符的优先级编码是 1，如图 4 所示，输入字符串里面有两个两位数 35 和 12，所以优先级数组里有两个元素值是 1。那如何将是一个整体的数据读出来呢？可以通过一个函数 atof(sr_ay+nn)，将 sr_ay 字符串中 nn 开始的数据读出来。atof() 函数的名字来源于 ascii to floating point numbers 的缩写，它会扫描参数 str 字符串，跳过前面的空白字符（例如空格，tab 缩进等），直到遇上数字或正负符号才开始做转换，而再遇到非数字或字符串结束时（'\0'）才结束转换，并将结果返回。例如：

如果 char sr_ay[50]=" - 35+12*6/3-4"

nn=1 时，atof(sr_ay+nn) =35；

nn=4 时，atof(sr_ay+nn) =12；

输入字符串 1	sr_ay[50]= " -35+12*6/3-4"												
对应优先级	fz_ay[50]=[14	04	01	12	04	01	13	04	13	04	12	04	00]
数组下标 nn	0	1	2	3	4	5	6	7	8	9	10	11	12
sr_ay[nn]	-	3	5	+	1	2	*	6	/	3	-	4	
sr_ay 含义	负号	数字	无效符	加号	数字	无效符	乘号	数字	除号	数字	减号	数字	结束符
fz_ay[nn]	14	04	01	12	04	01	13	04	13	04	12	04	0

图 4　无效符示例

（三）问题 3：中间数据的保存和查找

这里引入了一个数组 js_ay[10] 来保存计算的中间数据，这个数组是本算法最巧妙的地方，这个数组的每一个值和都是计算过程的中间数据，按先后顺序存放，比如第一个中间数据保存在 js_ay[0]，第二个中间数据保存在 js_ay[1]，以此类推。在计算过程中需要中间数据的时候，就到这个数组里读取。

那这个中间数据是如何得来并保存的呢？

简单起见，假如输入表达式是 2+3*5，写出这三个数组：

sr_ay[50]= 2 + 3 * 5 ，数组下标用 nn 表示，

fz_ay[50]=[4 12 4 13 4]，数组下标用 nn 表示，

js_ay[10]=[?]，数组下标用 hcwz 表示，初值 =0。

第一步先找到输入字符串 sr_ay 中最高优先级序号及该字符在数组的位置，分别保存在 yxj 和 nn 中，这个例子中 yxj=13，nn=3；

第二步寻找优先级别最高运算符右侧的数据，找到后把优先级最高那个位置的优先级由 13 修改成

1，表示这个运算符计算了，变成无效字符，也就是 fz_ay[3]=1，那么现在 fz_ay[50]=[4 12 4 1 4]；

第三步寻找优先级别最高运算符左侧的数据，找到后把左侧数据的优先级也修改成 1，变成 1，表示这个数据计算了变成无效字符，在这个例子中也就是 fz_ay[2]=1，那么现在 fz_ay[50]=[4 12 1 1 4]；

第四步找到了右侧数据，也找到了左侧数据，就可以进行计算了，在这个例子中也就是 3*5=15，这是第一个中间数据；

第五步是保持中间结果到相应数组。首先把右侧数据的优先级修改为 3，表示这里有一个间接数据，那么现在 fz_ay[50]=[4 12 1 1 3]，fz_ay[4]=3，下标是 4，那对应的 sr_ay[4] 修改为 hcwz，hcwz 表示 js_ay[] 的下标，再把中间数据 15 存入 js_ay[hcwz]，这样就把第一个中间数据保存到了 js_ay[0] 中。

一轮以后：

sr_ay[50]= 2 + 3 * 0，数组下标是 nn，

fz_ay[50]=[4 12 1 1 3]，数组下标是 nn，

js_ay[10]=[15]，数组下标是 hcwz=0，

这样在下一轮计算过程中，fz_ay[50]=[4 12 1 1 3]，优先级最高的是 12，也先读取右边的数据，发现是 1 表示是无效数据，继续往右边找直到是 3 或是 4，3 表示间接数据，4 是直接数据，在这个例子里是一个中间数据（因为 fz_ay[4]=3），根据相同的下标 4 找到 sr_ay[4]=0，而这个 0 就是中间数组的下标，再找到 js_ay[0]=15，就找到了右边的数据是 15，然后再去找左边的数据，也是一样的算法，因为 fz_ay[0]=4，左边数据是一个直接数，通过函数 atof(sr_ay+0) 计算出来为 2，于是得到表达式 2+15，最终结果是 17。

（四）复杂计算器的实现

一个复杂的计算器表达式，中间的计算过程会有多步，本质上每一步都是先找到优先级最高的运算符，然后再分别找到这个运算符右边的数据和左边的数据，就可以计算出这一步的结果。从 4.3 可以看出，不管是左边的数据还是右边的数据，有两种可能性，要么是直接数据要么是中间数据，如果 fz_ay[nn]=3，表示这个数是中间数据，再通过 nn 找到 sr_ay[nn]，而 js_ay[sr_sy[nn]] 就是这个中间数据；fz_ay[nn]=4，表示这个数是直接数据，再通过函数 atof(sr_ay+nn) 得到直接数据。因此本文算法就是设计数组的嵌套，用一个数组值是另一个数组的下标，实现了中间数据的逻辑关联。整个程序流程图如图 5 所示：

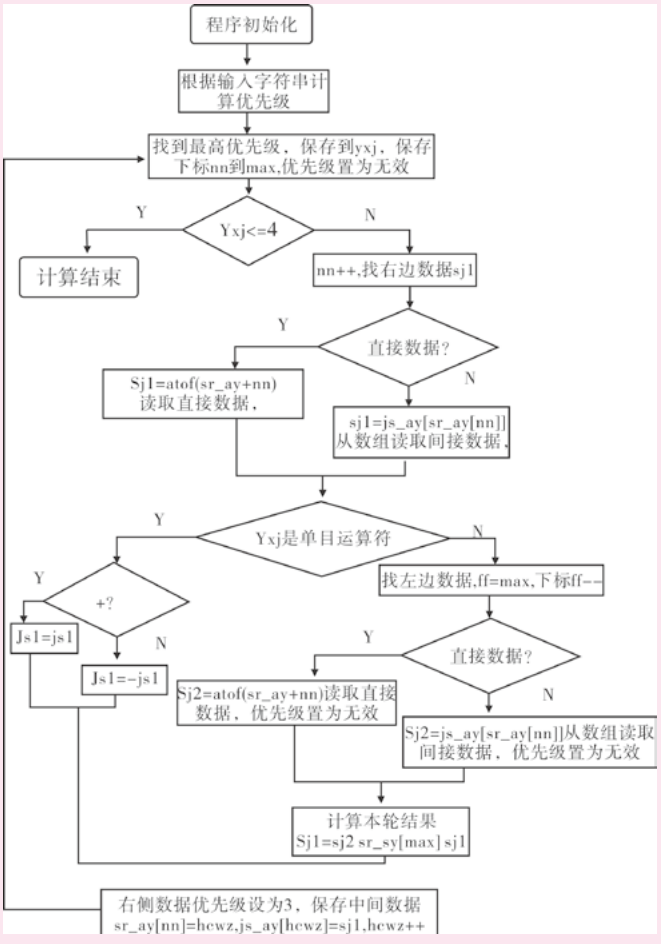

图 5　完整程序流程图

程序如图 6 所示：

```
hcwz=0;                                                           // 中间数据缓存数组地址=0
while(1){
  nn=0; yxj=0; while(fz_ay[nn]!=0){ if(fz_ay[nn]>yxj){ yxj=fz_ay[nn]; }  ++nn; }   // 找到当前最高优先级运算符
  nn=0; while(fz_ay[nn]!=yxj){ ++nn; }  fz_ay[nn]=1;  max=nn;      // 清优先级标志，找到最高优先级的位置
  if(yxj<=4){ break; }                                             // 没有运算符了，停止计算
  while(fz_ay[nn]!=3&&fz_ay[nn]!=4){ ++nn; }                       // 查找右边数据，nn=数据地址
 // ++nn;
  if(fz_ay[nn]==3){ sj1=js_ay[sr_ay[nn]]; }                        // 3--间接数据
  else if(fz_ay[nn]==4){ sj1=atof(sr_ay+nn); }                     // 4--直接数据,库函数，字符串变浮点数
  //------------------------------------------------------------ 单目运算
  if(yxj==14){                                                     // 单目+-运算
    switch(sr_ay[max]){
      case '-':  sj1=-sj1; break;                                  // 单目- 运算
      case '+':  break;                                            // 单目+ 运算
    }
  }
  //---------------------------------------------------------------- 双目运算
  else{
    ff=max;
    while(fz_ay[ff]!=3&&fz_ay[ff]!=4){ --ff; }                     // 找到左边的操作数
    if(fz_ay[ff]==3){ sj2=js_ay[sr_ay[ff]]; fz_ay[ff]=1; }         // 3=间接数据
    else if(fz_ay[ff]==4){ sj2=atof(sr_ay+ff); fz_ay[ff]=1; }      // 4=直接数据
    switch(sr_ay[max]){
      case '+': sj1=sj2+sj1; break;
      case '-': sj1=sj2-sj1; break;
      case '*': sj1=sj2*sj1; break;
      case '/': sj1=sj2/sj1;
    }
  }
  fz_ay[nn]=3; sr_ay[nn]=hcwz; js_ay[hcwz]=sj1; ++hcwz;            // 保存中间结果到相应数组
}
printf("%f\n",sj1);                                                // 浮点数打印
```

图 6　根据优先级计算的程序

五、两种算法对比结果

通过对两种算法进行对比，有如下结果：

第一，简单的四则运算，两种运算结果一致，如图 7 所示。

第二，笔者在程序中将各类函数通过数组增加到多种，包括取反、左移、右移、按位与、按位或等，如图 8 所示，同时加入各种三角函数，如图 9 所示，依本算法设计的程序通过测试，可以得到正确结果，图 10 是测试过程。而用逆波兰算法实现这些功能，程序需要修改很多，代码的复杂度和长度都远远大于本算法，如图 11 所示，逆波兰算法程序将近 300 行，而本算法的长度只有 140 行，而且功能更强大。

```
C:\Users\Administrator\Desktop\逆波兰.exe
8+(35/7.2-2)*6
result = 25.17

E:\陈亭志\C语言慕课\cyypt.exe
8+(35/7.2-2)*6
8+(35/7.2-2)*6=25.166668
```

图 7　简单四则运算对比

```
// 13个运算符号 ( )  A-Z a-z  ~ + -  * / %  + -  << >>  &  ^  |              0-9.  间接数据  ( )  无效符  结束符
// 优先级排序    16     15      14     13     12    11    10  9  8  7  6  5    4      3       2     1       0
```

图 8　运算符增加到 13 种

```
main()
{ char sr_ay[200]; char fz_ay[200];              //输入字符串，   每个字符串对应的优先级
  char hs_ay[10][8]={"sin(","asin(","cos(","acos(","tan(","atan(","exp(","sqrt(","log(","log10("};   // 计算函数
  float js_ay[20]; float sj1,sj2;                                                                    // 辅助变量
```

图 9　增加各种计算函数

```
5>>3
5>>3=0
5>>3|20
5>>3|20=20
(35/7-2)*sin(30)
(35/7-2)*sin(30)=1.500000
8+(35/7-2)*sin(30)
8+(35/7-2)*sin(30)=9.500000
5>>3|20+8+(35/7-2)*sin(30)
5>>3|20+8+(35/7-2)*sin(30)=29.000000
```

图 10　本算法的测试过程

```
265      free(rpn->opera);             135    fz_ay[nn]=3; sr_ay[nn]=hcw
266      free(rpn);                    136  }
267      free(mark->opera);            137  if(dot==0){ printf("%f\n",sj
268      free(mark->prior);            138  else if(dot==1){ printf("%d\
269      free(mark);                   139  else if(dot==2){ printf("0x%
270                                    140  goto begin;
271      return 0;                     141  return 0;
272  }
```

图 11　逆波兰和本算法程序长度对比

六、结束语

本文设计的这种算法，通过将运算符进行优先级排序，找到了复杂计算问题的简化模型。即每一步先找优先级最高的运算符（假设是 X），然后找右边的数据（假设是 A）和左边的数据（假设是 B），那这一步的结果就是 BXA，将这个结果存入中间数据数组，为下一轮计算做准备。值得注意的是，在每一步找到 X、B、A 后，要将其优先级分别设为无效、无效和中间数据，这样在下一轮计算的时候就不会重复计算。而在找左边或右边的数据时有两种情况，要么是直接数据，要么是间接数据，直接数据通过函数 atof( )得到，间接数据通过数组的嵌套 js_ay[sr_sy[nn]] 得到。这样不管表达式有多长和多复杂，只要设置好表达式的优先级，都可以通过这个算法，重复每一步计算，最终得到结果。

参考文献：

[1] 李小玲,魏建国,袁继敏.新工科背景下基于OBE的《C语言程序设计》课程建设[J].攀枝花学院学报,2020,(5):103-107

[2] 李桂春.关于逆波兰表达式在程序设计中的应用[J].白城师范学院学报,2007,(12):58-62.

[3] 周欣.逆波兰式转换规则的推导过程[J].电脑编程技巧与维护,2016,(21):35-36.

[4] 杨忠.逆波兰表达式在VB中的算法与设计实现[J].制造业自动化,2011,(9):79-83.

[责任编辑：胡大威]

An Algorithm Design for Calculating Complicated Arithmetic Expression Using Array

Chen Tingzhi, Cheng Limin

(College of Mechanical and Electrical Engineering, Wuhan Polytechnic, Wuhan430074, China)

Abstract: The calculation of arithmetic expressions is a classic mathematical problem in C language programming. When a computer solves this problem, it usually uses the inverse Polish algorithm to convert the infix expression into a suffix expression, and then use the stack calculation to implement this algorithm. It can be realized only by using the idea of recursion and combining the complex data structure of the stack. In this paper, an algorithm is designed. By prioritizing operators, a simplified model of complex calculation problems is found. The calculation of complex arithmetic expressions can be realized by using loops and arrays.

Key words: inverse Polish; arithmetic expression calculate; array; priority