C语言高级程序设计

卢有杰 吴炜煜 编

清华大学出版社

C语言高级程序设计

卢有杰　吴炜煜　编

清华大学出版社

内 容 简 介

本书通过大量实例介绍运用C语言编程的特点和技术。大多数实例使用了结构和指针，并用了一定的篇幅来说明C语言数据结构。本书共组织了九个专题：C语言数据结构；动态存储管理；C的存储方式；C语言同操作系统的接口设计；混合语言接口；将PASCAL和BASIC程序改编成C程序；C语言统计程序设计；表达式句法分析和求值；效率、移植和调试。其材料大部分取自专家的著作，一部分是编者应用C语言编制软件的实例，读者可直接引用。

本书是为已具有C语言基础知识的软件工作人员、大学高年级学生、研究生以及各方面应用C语言进行软件开发的程序员编写的。

版权所有，翻印必究。
本书封面贴有清华大学出版社
激光防伪标志，无标志者不得销售。

(京)新登字158号

C语言高级程序设计
卢有杰　吴炜煜　编
责任编辑　范素珍
☆
清华大学出版社出版
北京　清华园
北京市昌平县第一排版厂排版
中国科学院印刷厂印装
新华书店总店科技发行所发行
☆
开本：787×1092 1/16　印张：15　字数：373千字
1991年9月第1版　1995年11月第7次印刷
印数：55001—58000
ISBN 7-302-00880-9/TP·319
定价：12.80元

前　言

C 语言是一种通用的程序设计语言，既适合应用程序设计，又适合系统程序设计。C 语言在操作系统、软件工具、图象处理、数值分析、人工智能、数据库管理系统设计等许多方面都得到了广泛的应用。C 语言有许多突出的优点，它的丰富的数据类型、灵活的数据流控制、多种多样的运算符、利于模块化的函数，增强了语言的表达能力，提高了编程效率，有利于充分发挥当前计算机硬件的潜在功能。C 语言比其它高级语言更具汇编语言特点，比汇编语言更具有可读性，比其它语言的可移植性都好。所有这些优点吸引了越来越多的人们应用 C 语言设计软件。

C 语言在我国的普及程度还很不够，适合我国读者使用的教材与其它语言相比，还不算多。特别是结合实际应用，讲解 C 语言编程技术和解决应用问题的程序表达技巧，以及有效地利用计算机系统硬件和软件资源的书就更不多见。然而学习了 C语言程序设计基础知识的人，要能够做到用 C 来进行应用软件设计，就十分需要进行这方面的学习。本书正是为已具有 C语言基础知识的软件工作人员、大学高年级学生、研究生以及各方面应用 C 语言进行软件开发的程序员编写的。

本书通过实例向读者介绍运用 C 语言编程的技术和特点。对于 C 语言中的指针、结构，许多初学者往往感到难以理解和运用，使用起来不大顺手。本书中的实例大都使用了结构和指针，读者通过这些程序员所编制的程序，便可悟其真谛。又如，书中用了一定的篇幅来说明 C语言数据结构的算法，在实例及其说明中，使抽象的概念寓于具体的应用程序中，对于未专门学过数据结构的人来说，可作为学习数据结构的材料，对于处理相应的数据结构问题，是较好的“汤药成方”。由于这本书的主要宗旨是帮助读者实现从 C 语言基础知识到 C语言高级程序员的过渡，所以围绕这个主要宗旨组织了九个专题进行介绍。书中的材料大多取自专家的著作，一部分是本书编者应用 C语言编制软件的实例，它们大多数都通过编译和运行，读者可以将它们直接地或者稍加修改后用在自己的相应程序中。

本书第一、二、三、八、九章由卢有杰编，第四、五、六、七章由吴炜煜编。清华大学软件开发中心程渝荣副教授对全书初稿进行了审校，清华大学计算机系张钹教授对本书曾给予热情的关怀和指导，在此一并表示衷心感谢。

由于编者水平所限，书中难免有不当或错误之处，恳请读者批评指正，以便再版时加以改进。

编　者

一九九〇年八月于清华大学

目　录

此为试读,需要完整PDF请访问: www.ertongbook.com

第一章　C语言数据结构

1.1　排序和查找

在计算机科学领域，排序和查找非常重要。数据一般先经过排序，再进行查找。排序和查找例程不但用在编译程序、解释程序和操作系统中，而且用在大多数的数据库程序中。

1.1.1　排序

排序是将一组数据按递增或递减的顺序排列。具体说来，具有n个元素的线性表i，经过排序，其元素间应满足下列关系：

$$i_1 <= i_2 <= \cdots <= i_n$$

即使许多C编译器在标准库中提供了标准的qsort（　）函数，仍然有必要对排序进行认真的研究，以求彻底理解。其原因有三：第一，qsort（　）太一般化了，并非在所有情况下都有效。第二，因为qsort（　）利用参数对不同类型的数据进行操作，所以运行起来就不如只对一种类型数据排序时快。第三，在某些特殊情况下，qsort(　)所用的快速排序算法并不是最好的算法。

排序算法可分为两种类型：用于内部随机文件的数组排序和用于外部顺序文件的排序。其中第一种对微机用户最有用，我们将重点讨论。

数组排序中数组的所有元素都可以随时取用，即数组的任何元素在任何时候都可以同其它元素进行比较和交换。而在顺序文件中，任何时候都只能取用其中的一个元素。由于存在这种差别，两种数据的排序方法相差也就很大。一般情况下，在对数据排序时，数据中的一小部分要充当排序关键字。排序关键字是比较的基准。必须交换时，整个数据结构都要交换。例如，在投递地址表中可用邮政编码ZIP字段充当关键字，但在交换时，该记录中其它字段，如姓名和地址等都要同ZIP一起改变位置。为简单起见，下面介绍的各种排序方法的实例都以字符数组的排序为重点，请读者将这些方法灵活运用到其它类型的数据结构中。

1．排序算法的分类

对数组排序的方法一般有三种：（1）交换排序，（2）选择排序，（3）插入排序。

假定我们有一副纸牌，若进行交换排序，则先将纸牌摊在桌子上，正面朝上，然后开始把不顺的牌的位置换过来，直到整副牌由无序变为有序。若进行选择排序，则先把牌摊在桌子上，找出点数最小的牌，并从牌堆中抽出，然后再从留在桌子上的牌中挑出最小的，并把它放在刚才已拿在手中的那张牌的后面，这样一直进行到所有的牌都拿到手中。由于总是从留在桌子上的纸牌中挑出最小的，所以当这个过程结束时，手中的牌也就排了序。若进行插入排序，则将纸牌放在手中，每次抽出一张，放在桌子上重新排

列，自始至终将所抽出的牌插入正确的位置。这样，当手中的牌抽完时，整副牌也就在桌上排好了序。

2。排序算法的优劣评价

在上述三种排序方法中，每一种都有多个不同的算法。每种算法都有其优点。要判断它们的优劣，必须回答下列问题：

- 一般情况下该算法对数据排序的速度如何？
- 何时最快？何时最慢？
- 该算法是否有自然的或不自然的性质？
- 该算法是否用相同的关键字重新排列诸元素？

算法的快慢至关重要。数组排序的速度同所需的比较和交换次数有直接关系，而交换所需的时间更多。若是经常遇到最好和最坏的情形，则最快和最慢的运行时间就是不可忽视的指标。

如果排序算法所排的对象开始时就已有序，则运行起来最省力，当对象的有序性逐渐变差时运行逐渐困难，而当对象开始时为逆序，则运行起来最费劲，就说该算法表现了自然的性质。排序算法运行时的困难程度取决于必须执行的比较和换位的次数。

要体会用相等的关键字重新排列元素的重要性。设想某个数据库要根据主关键字和次关键字排序，例如，投递地址表以ZIP编码为主关键字，而以同一个 ZIP 编码的最后一个名字为次关键字。当新地址加入投递地址表时，该表要重新排序，但又不希望重新排列次关键字。为了保证做到这一点，排序算法务必不要具有等值的主关键字。

以下各段对各类排序的代表性方法进行分析、研究，并比较它们的优缺点和效率。

3．上推排序法（或称汽泡排序法）

人们最熟悉的排序法是上推排序法。这个方法最简单，然而也是最差的排序法之一。

上推排序使用的方法是交换排序。此法就是反复进行比较，必要时交换相邻的元素。下面是最简单的上推排序法程序：

```
/*上推排序法*/
void bubble(item, count)
char *item;
int count;
{
  register int a, b;
  register char t;
  for(a=1; a<count; a++)
    for(b=count-1; b>=a; --b)
    {if(item[b-1]>item[b])
        /*交换元素*/
      {t=item[b-1];
       item[b-1]=item[b];
       item[b]=t;
      }
    }
}
```

其中item是指向字符的指针，count是数组元素的个数。

上推排序函数bubble（ ）有两个循环套。既然数组中有count个元素，那么外层循环对数组扫描count－1次，这就保证了在最坏的情况下，当函数结束时每个元素都放在适当的位置上。内层循环进行实际上的比较和交换（若把上面的上推排序法稍稍改进一下，就会在无交换进行时结束。但那样一来，就会在内层循环每走一圈时多做一次比较）。

该上推排序函数可用来将字符数组由小到大排序。例如，下面的程序将从键盘上输入的字符串排序：

```
void bubble( )
/*将键盘输入的字符串排序*/
main( )
{
    char s[80];
    printf("\t请输入字符串：");
    gets(s);
    bubble(s, strlen (s));
    printf("\t字符串排序后是：%s\n",s);
}
```

为了让读者看清上推排序函数的执行过程，下面列出对数组dcab排序时要走的几步：

开始时　d c a b
第一步　a d c b
第二步　a b d c
第三步　a b c d

在分析各种排序法的优劣时，必须计算在最好、一般和最差三种情况下所进行的比较和交换的次数。上推排序函数比较的次数总是一样的，因为这两层for循环套，不管排序的对象原来排序的情况怎样，都要进行指定次数的循环。也就是说，上推排序法总要做$\frac{1}{2}(n^2-n)$次比较，其中n是要排序的对象元素的个数。

至于交换次数，当对象原本已有序时为交换次数0，这是最好的情况。一般情况下为$\frac{3}{4}(n^2-n)$，最差时为$\frac{3}{2}(n^2-n)$。

当元素个数很大时，上推排序法很不好用，因为执行时间同比较和交换的次数成正比。读者可以对上推排序法做些改进，让它运行得快些。上推排序法有个怪癖：未在自己应在位置的元素若在后端，例如decab数组中的a，经过一步就可以找到它应在的位置；而若在前端，例如d，要回到自己应在的位置，则需很多步。因此，不要总是在同一个方向读数组，可以颠倒读的方向，以便使偏离自己应在位置很远的元素快些回到自己应在的位置上。经过改进的上推排序函数列在下面。该函数叫做鲨鱼函数，因为它象鲨鱼一样从数组的一端换到另一端。

```
/*鲨鱼排序是对上推排序的改进*/
void shaker(item, count)
```


此为试读，需要完整PDF请访问：www.ertongbook.com

```
char *item;
int count;
{
  register int a,b,c,d;
  char t;
  c=1;
  b=count-1;
  d=count-1;
  do {
    for(a=d; a>=c; --a)
    { if(item[a-1]>item[a])
      { t=item[a-1];
        item[a-1]=item[a];
        item[a]=t;
        b=a;
      }
    }
    c=b+1;
    for(a=c; a<d+1; ++a)
    { if(item[a-1]>item[a])
      { t=item[a-1];
        item[a-1]=item[a];
        item[a]=t;
        b=a;
      }
    }
    d=b-1;
  } while(c<=d);
}
```

鲨鱼排序法对上推排序法并没有太大改进，因为执行的次数仍是n^2数量级。

4. 选择排序

在选择排序方法中，第一步选出最小元素，将其与第一个元素交换。第二步从剩下的n－1个元素中选出最小元素，将其与第二个元素交换，如此下去，直到剩下最后两个元素。例如，对数组 bdac 作选择排序时要进行如下几步：

开始时　b d a c

第一步　a d b c

第二步　a b d c

第三步　a b c d

选择排序函数如下：

```
/*选择排序*/
void select(item, count)
char *item;
```

```
int count;
{
   register int a, b, c;
   char t;
   for(a=0; a<count-1; ++a)
   { c=a;
     t=item[a];
     for(b=a+1; b<count; ++b)
      { if(item[b]<t)
       { c=b;
         t=item[b];
       }
      }
     item[c]=item[a];
     item[a]=t;
   }
}
```

同上推排序函数一样，选择排序函数外层循环要执行n－1次，而内层要执行$\frac{1}{2}$n次。这就是说，选择排序要做$\frac{1}{2}(n^2-n)$次比较，对于数目大的数组，这就太慢了。交换次数最好时是3(n－1)次，最坏时是$\frac{n^2}{4}+3(n-1)$次。

同上推排序法相比，比较次数相同，但交换次数要少得多。

5．插入排序

插入排序首先对数组的头两个元素排序。然后根据前两个元素排序的情况把第三个元素插入其应到的位置。第三步，根据前三个元素排序的情况把第四个元素插入其应到的位置。该过程一直进行到所有元素都排好序为止。例如，对于数组dcab，插入排序的各步如下：

开始时　d c a b

第一步　c d a b

第二步　a c d b

第三步　a b c d

插入排序函数如下：

```
/*直接插入排序*/
void insert(item, count)
char *item;
int count;
{
   register int a, b;
   char t;
```

```
for(a=1;a<count;++a)
 { t=item[a];
   b=a-1;
   while(b>=0&&t<item[b])
   { item[b+1]=item[b];
     b--;
   }
   item[b+1]=t;
 }
}
```

与上推排序和选择排序不同，插入排序的比较次数取决于排序对象开始时的排列情况。排序对象开始时若有序，则比较次数为n－1。如果开始时为逆序，则比较次数为$\frac{1}{2}(n^2+n-1)$，而平均次数为$\frac{1}{4}(n^2+n-2)$。

各种情况下的交换次数分别为：

最好的情况：$2(n-1)$

一般情况：$\frac{1}{4}(n^2+9n-10)$

最差的情况：$\frac{1}{2}(n^2+3n-4)$

因此，最坏情况下的交换次数同上推排序和选择排序一样糟糕，而一般情况下只是略好一些。但是，插入排序有两个优点。首先，它很自然：即当数组开始为有序时工作量最小，当数组开始为逆序时工作量最大。这样，插入排序适用于对象基本有序的情况；其次，它不改变相同关键字的顺序：即若对象用两个关键字进行了排序，那么在插入排序后，对象按这两个关键字都是有序的。

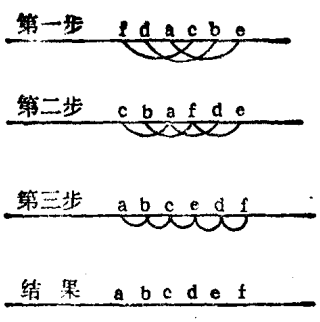
6．改进的排序

前面介绍的各种算法都有一个致命的缺点，即执行次数都是n^2数量级。对于量大的数据，排序过程将进行得很慢，有时会慢得无法使用。一个排序若花很长的时间，其根源在于所用的算法。虽然使用汇编码确实可以提高例程的执行速度，但是，所用的算法若不好，不管目标码进行过何种优化，速度都不会有较大改善。解决的办法只有一个，即选用较好的排序算法。

下面将分几步编写两种很好的排序函数。第一个是希尔排序，又叫缩小增量排序；第二个叫快速排序，一般认为它是最好的排序例程。

7．希尔（shell）排序

这种方法以它的创造者D.L.希尔命名。该法源于插入排序法，运行时逐步缩小增量。下面画出希尔排序法对数组fdacbe排序的过程：



首先对相隔三个位置的元素排序，然后对相隔两个位置的元素排序，最后对相邻的元素排序。

这种方法是否可产生好的结果，看起来并不明显，甚至还看不出它是在进行排序。实际上，它确实在进行排序，其结果也确实不错。该算法之所以效率高是由于每一步用到的元素少，或者用到的诸元素已在先前排好了序，因此每一步都提高了数据的有序性。

元素之间相隔的位置叫增量。各步所用的增量可以变化，唯一的要求是最后一步的增量必须为1。例如，增量序列9，5，3，1用起来就很好。下面的希尔排序函数就使用了这个序列。请避免使用2的幂做为增量序列，因为所涉及的数学复杂，而且还会降低排序算法的效率。

```
/*希尔排序*/
void shell(item, count)
char *item;
int count;
{
   register int i,j,k,s,w;
   char x, a[5];
   a[0]=9;
   a[1]=5;
   a[2]=3;
   a[3]=2;
   a[4]=1;
for(w=0;w<5;w++)
{ k=a[w];
  s=-k;
  for(i=k;i<count;++i)
 {x=item[i];
  j=i-k;
  if(s==0){ s=-k;
   s++;
   item[s]=x;
  }
```

```
        while(x<item[j] && j>=0 && j<=count)
        { item[j+k]=item[j];
          j=j-k;
        }
        item[j+k]=x;
        }
    }
}
```

读者可能已经注意到，内层的while循环有三个测试条件。x<item[j]是排序过程所必须的比较。j>=0和j<=count用来防止超越数组item的上下界。这三个附加的检查会在某种程度上降低希尔排序法的性能。

对希尔排序稍加修改，使用了称做 sentinels 的专用变量，它并不属于要排序的数组，而用来保存专用的结束值，以指明最小和最大的可能元素。这样，上下界检查也就不需要了。但是，使用sentinels时要对数据有具体的了解，因此也就限制了该排序函数的通用性。

当元素个数为n时，希尔排序所需的运行时间正比于$n^{1.2}$，与前面介绍的诸排序方法相比有了很大的改进。

8．快速排序

快速排序是由C.A.R Hoarse发明并命名的，被公认为是目前最好的排序算法。此法借助"交换"进行排序。

快速排序的基本思想是分割排序对象。一般的做法是先选择一个值，将其称为"比较数"然后将数组分为两部分，所有大于等于此"比较数"的元素放在一边，所有小于此"比较数"的元素都放在另一边。对分出的各部分重复这一过程直到整个数组完成排序。例如，假定数组为fedacb，而"比较数"为d，快速排序的第一步把此数组排列如下：

开始时 f e d a c b

第一步 b c a d e f

分成的两部分（bca和def）再以同样方式进行。该过程实质上是递归过程，要干净利落地实现快速排序必须使用递归算法。

可用两种方式选取中间"比较数"的值。此值可以随机选取，也可取自该数组的一小部分元素的平均值。要得到最优排序，最好选择在数组取值范围中间的那个值。然而，对大多数数据来讲，这一点不易做到。即使出现最坏的情况，即此值是最大的或最小的值，快速排序仍然可得到良好结果。

快速排序程序如下，选取数组的中间值做为"比较数"，这样的选法虽然并不能总是选到理想的值，但做起来简单、快、正确。

```
/*快速排序输入函数*/
void quick(item, count)
char *item;
int count;
{
   qs(item,0,count-1);
```

```
}
/*快速排序函数*/
void qs(item, left, right)
char *item;
int left, right;
{
   register int i,j;
   char x,y;
   i=left;
   j=right;
   x=item[(left+right)/2];
   do {
     while(item[i]<x&&i<right) i++;
     while(x<item[j]&&j>left) j--;
     if(i<=j)
     { y=item[i];
       item[i]=item[j];
       item[j]=y;
       i++;
       j--;
     }
   } while(i<=j);
   if(left<j) qs(item, left, j);
   if(i<right) qs(item, i, right);
}
```

其中quick（ ）实现对主要排序函数qs（ ）的调用。这样做虽然可以为item和count维持同一个共用接口，但意义不大。因为qs（ ）也可以利用三个参数直接调用。

快速排序进行的比较和交换次数分别约为nlogn和$\frac{n}{6}$logn，要比前面介绍过的所有排序法都好得多。

但是快速排序有一个不利的方面需要读者当心。如果分割后的各个部分的"比较数"恰好都是最大值时，快速排序就倒退而成"慢速排序"。当然，一般不会出现这种情况。尽管如此，读者在选择确定"比较数"方法时还是小心为妙。此数常常由要排的实际数据所确定。在很大的投递地址表问题中由于常常按ZIP编码排序，方法的选取很简单。ZIP编码分布相当均匀，用简单的代数函数就能确定合适的比较数。但是在某些数据库中排序关键字的值可能非常接近，甚至许多都具有同样的值，随机选取常常是最好的办法。常用而有效的方法是从分割后的部分中抽取三个元素，取其中间值。

9．排序方法的选取

通常快速排序因其快而为人们所乐用。然而当数据量小时（比如少于100个），快速排序递归调用所带来的附加负担可能会抵消巧妙算法所带来的好处。这时，别的简单排序法，甚至上推排序法都会比它好。

此为试读，需要完整PDF请访问：www.ertongbook.com