例 8.5 输入 3 个整数 a,b,c,要求按由大到小的顺序将它们输出。用函数实现。

解题思路：采用例 8.3 的方法在函数中改变这 3 个变量的值。用 swap 函数交换两个变量的值,用 exchange 函数改变这 3 个变量的值。

编写程序：

```
#include <stdio.h>
int main()
{ void exchange(int *q1, int *q2, int *q3);          //函数声明
  int a,b,c,*p1,*p2,*p3;
  printf("please enter three numbers:");
  scanf("%d,%d,%d",&a,&b,&c);
  p1=&a;p2=&b;p3=&c;
  exchange(p1,p2,p3);
  printf("The order is:%d,%d,%d\n",a,b,c);
  return 0;
}

void exchange(int *q1, int *q2, int *q3)             //定义将 3 个变量的值交换的函数
{void swap(int *pt1, int *pt2);                      //函数声明
 if(*q1<*q2) swap(q1,q2);                            //如果 a<b,交换 a 和 b 的值
 if(*q1<*q3) swap(q1,q3);                            //如果 a<c,交换 a 和 c 的值
 if(*q2<*q3) swap(q2,q3);                            //如果 b<c,交换 b 和 c 的值
}

void swap(int *pt1, int *pt2)                        //定义交换 2 个变量的值的函数
 {int temp;
  temp=*pt1;                                         //换 *pt1 和 *pt2 变量的值
  *pt1=*pt2;
  *pt2=temp;
 }
```

运行结果：

```
please enter three numbers:20,-54,67
The order is:67,20,-54
```

程序分析：exchange 函数的作用是对 3 个数按大小排序,在执行 exchange 函数过程中,要嵌套调用 swap 函数,swap 函数的作用是对两个数按大小排序,通过调用 swap 函数(最多调用 3 次)实现 3 个数的排序。

请读者自己画出如图 8.6 那样的图,仔细分析变量的值变化的过程。

思考：main 函数中的 3 个指针变量的值(也就是它们的指向)改变了没有。

8.3 通过指针引用数组

8.3.1 数组元素的指针

一个变量有地址,一个数组包含若干元素,每个数组元素都在内存中占用存储单元,它们都有相应的地址。指针变量既然可以指向变量,当然也可以指向数组元素(把某一元素的

地址放到一个指针变量中)。**所谓数组元素的指针就是数组元素的地址。**

可以用一个指针变量指向一个数组元素。例如:

```
int a[10]={1,3,5,7,9,11,13,15,17,19};        //定义 a 为包含 10 个整型数据的数组
int * p;                                      //定义 p 为指向整型变量的指针变量
p=&a[0];                                      //把 a[0]元素的地址赋给指针变量 p
```

以上是使指针变量 p 指向 a 数组的第 0 号元素,见图 8.8。

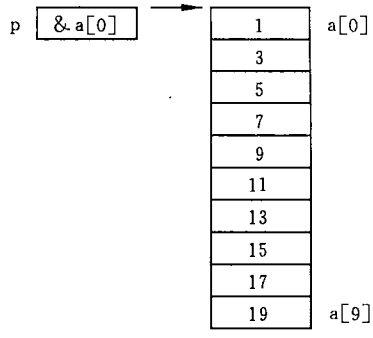

图 8.8

引用数组元素可以用**下标法**(如 a[3]),也可以用**指针法**,即通过指向数组元素的指针找到所需的元素。使用指针法能使目标程序质量高(占内存少,运行速度快)。

在 C 语言中,数组名(不包括形参数组名,形参数组并不占据实际的内存单元)代表数组中首元素(即序号为 0 的元素)的地址。因此,下面两个语句等价:

```
p=&a[0];    //p 的值是 a[0]的地址
p=a;        //p 的值是数组 a 首元素(即 a[0])的地址
```

注意:数组名不代表整个数组,只代表数组首元素的地址。上述"p=a;"的作用是"把 a 数组的首元素的地址赋给指针变量 p",而不是"把数组 a 各元素的值赋给 p"。

在定义指针变量时可以对它初始化,如:

```
int * p=&a[0];
```

它等效于下面两行:

```
int * p;
p=&a[0];    //不应写成 * p=&a[0];
```

当然定义时也可以写成

```
int * p=a;
```

它的作用是将 a 数组首元素(即 a[0])的地址赋给指针变量 p(而不是赋给 * p)。

8.3.2 在引用数组元素时指针的运算

在引用数组元素时常常会遇到指针的算术运算。有人会提出问题:对数值型数据进行算术运算(加、减、乘、除等)的目的和含义是清楚的,而在什么情况下需要用到对指针型数据的算术运算呢?其含义是什么?

前已反复说明指针就是地址。对地址进行赋值运算是没有问题的,但是对地址进行算术运算是什么意思呢?显然对地址进行乘和除的运算是没有意义的,实际上也无此必要。那么,能否进行加和减的运算?答案是:在一定条件下允许对指针进行加和减的运算。

那么,在什么情况下需要而且可以对指针进行加和减的运算呢?回答是:当指针指向数组元素的时候。譬如,指针变量 p 指向数组元素 a[0],我们希望用 p+1 表示指向下一个元素 a[1]。如果能实现这样的运算,就会对引用数组元素提供很大的方便。

在**指针指向数组元素时**,可以对指针进行以下运算:

加一个整数(用＋或＋＝),如 p＋1;

减一个整数(用－或－＝),如 p－1;

自加运算,如 p＋＋,＋＋p;

自减运算,如 p－－,－－p。

两个指针相减,如 p1－p2(只有 p1 和 p2 都指向同一数组中的元素时才有意义)。

分别说明如下:

(1) 如果指针变量 p 已指向数组中的一个元素,则 **p＋1 指向同一数组中的下一个元素,p－1 指向同一数组中的上一个元素**。注意:执行 p＋1 时并不是将 p 的值(地址)简单地加 1,而是加上一个数组元素所占用的字节数。例如,数组元素是 float 型,每个元素占 4 个字节,则 p＋1 意味着使 p 的值(是地址)加 4 个字节,以使它指向下一元素。p＋1 所代表的地址实际上是(p＋1)×d,d 是一个数组元素所占的字节数(在 Visual C++ 6.0 中,对 int 型,d＝4;对 float 和 long 型,d＝4;对 char 型,d＝1)。若 p 的值是 2000,则 p＋1 的值不是 2001,而是 2004。

有的读者问:系统怎么知道要把这个 1 转换为 4,然后与 p 的值相加呢?在定义指针变量时要指定基类型,如:

```
float *p;                                    //指针变量 p 的基类型为 float
```

现在 p 指向 float 型的数组元素,在执行＋＋p 时,系统会根据 p 的基类型为 float 型而将其值加 4,这样,p 就指向 float 型数组的下一个元素。

如果 p 原来指向 a[0],执行＋＋p 后 p 的值改变了,在 p 的原值基础上加 d,这样 p 就指向数组的下一个元素 a[1]。

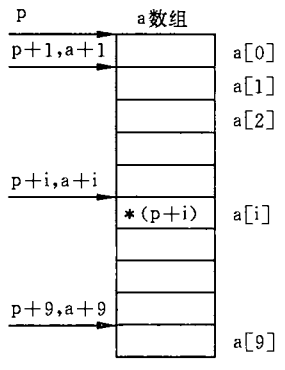

图 8.9

(2) 如果 p 的初值为 &a[0],则 p＋i 和 a＋i 就是数组元素 a[i]的地址,或者说,它们指向 a 数组序号为 i 的元素,见图 8.9。这里需要注意的是 a 代表数组首元素的地址,a＋1 也是地址,它的计算方法同 p＋1,即它的实际地址为(a＋1)×d。例如,p＋9 和 a＋9 的值是 &a[9],它指向 a[9],如图 8.9 所示。

(3) *(p＋i)或*(a＋i)是 p＋i 或 a＋i 所指向的数组元素,即 a[i]。例如,*(p＋5)或*(a＋5)就是 a[5]。即*(p＋5),*(a＋5)和 a[5]三者等价。实际上,在编译时,对数组元素 a[i]就是按*(a＋i)处理的,即按数组首元素的地址加上相对位移量得到要找的元素的地址,然后找出该单元中的内容。若数组 a 的首元素的地址为 1000,设数组为 float 型,则 a[3]的地址是这样计算的:1000＋3×4＝1012,然后从 1012 地址所指向的 float 型单元取出元素的值,即 a[3]的值。可以看出,[]实际上是**变址**运算符,即将 a[i]按 a＋i 计算地址,然后找出此地址单元中的值。

(4) 如果指针变量 p1 和 p2 都指向同一数组,如执行 p2－p1,结果是 p2－p1 的值(两个地址之差)除以数组元素的长度。假设,p2 指向实型数组元素 a[5],p2 的值为 2020;p1 指向 a[3],其值为 2012,则 p2－p1 的结果是(2020－2012)/4＝2。这个结果是有意义的,表示 p2 所指的元素与 p1 所指的元素之间差 2 个元素。这样,人们就不需要具体地知道 p1 和 p2 的值,然后去计算它们的相对位置,而是直接用 p2－p1 就可知道它们所指元素的相对距离。

两个地址不能相加，如 p1＋p2 是无实际意义的。

8.3.3 通过指针引用数组元素

根据以上叙述，引用一个数组元素，可以用下面两种方法：

(1) **下标法**，如 a[i]形式；

(2) **指针法**，如 *(a＋i)或 *(p＋i)。其中 a 是数组名，p 是指向数组元素的指针变量，其初值 p＝a。

例 8.6 有一个整型数组 a，有 10 个元素，要求输出数组中的全部元素。

解题思路：引用数组中各元素的值有 3 种方法：(1)下标法，如 a[3]；(2)通过数组名计算数组元素地址，找出元素的值；(3)用指针变量指向数组元素。分别写出程序，以资比较分析。

编写程序：

(1) 下标法。

```
#include <stdio.h>
int main()
 {int a[10];
  int i;
  printf("please enter 10 integer numbers:");
  for(i=0;i<10;i++)
    scanf("%d",&a[i]);
  for(i=0;i<10;i++)
    printf("%d ",a[i]);                    //数组元素用数组名和下标表示
  printf("%\n");
  return 0;
 }
```

运行结果：

```
please enter 10 integer numbers:0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

(2) 通过数组名计算数组元素地址，找出元素的值。

```
#include <stdio.h>
int main()
 {int a[10];
  int i;
  printf("please enter 10 integer numbers:");
  for(i=0;i<10;i++)
    scanf("%d",&a[i]);
  for(i=0;i<10;i++)
    printf("%d ",*(a+i));                  //通过数组名和元素序号计算元素地址，再找到该元素
  printf("\n");
  return 0;
 }
```

运行结果：与(1)相同。

程序分析：第 9 行中(a＋i)是 a 数组中序号为 i 的元素的地址，*(a＋i)是该元素的值。

第 7 行中用 &a[i]表示 a[i]元素的地址，也可以改用(a+i)表示，即：

```
scanf("%d",a+i);
```

读者可以上机试一下。

(3) 用指针变量指向数组元素。

```
#include <stdio.h>
int main()
 {int a[10];
  int *p,i;
  printf("please enter 10 integer numbers:");
  for(i=0;i<10;i++)
    scanf("%d",&a[i]);
  for(p=a;p<(a+10);p++)
    printf("%d ",*p);                          //用指针指向当前的数组元素
  printf("\n");
  return 0;
}
```

运行结果：与(1)相同。

程序分析：第 8 行先使指针变量 p 指向 a 数组的首元素(序号为 0 的元素，即 a[0])，接着在第 9 行输出 *p，*p 就是 p 当前指向的元素(即 a[0])的值。然后执行 p++，使 p 指向下一个元素 a[1]，再输出 *p，此时 *p 是 a[1]的值。余类推，直到 p=a+10，此时停止执行循环体。

第 6，7 行可以改为

```
for(p=a;p<(a+10);p++)
  scanf("%d",p);
```

用指针变量表示当前元素的地址。

3 种方法的比较：

① 例 8.6 的第(1)和第(2)种方法执行效率是相同的。C 编译系统是将 a[i]转换为 *(a+i)处理的，即先计算元素地址。因此用第(1)和第(2)种方法找数组元素费时较多。

② 第(3)种方法比第(1)、第(2)种方法快，用指针变量直接指向元素，不必每次都重新计算地址，像 p++这样的自加操作是比较快的。这种有规律地改变地址值(p++)能大大提高执行效率。

③ 用下标法比较直观，能直接知道是第几个元素。例如，a[5]是数组中序号为 5 的元素(注意序号从 0 算起)。用地址法或指针变量的方法不直观，难以很快地判断出当前处理的是哪一个元素。例如，例 8.6 第(3)种方法所用的程序，要仔细分析指针变量 p 的当前指向，才能判断当前输出的是第几个元素。有经验的专人员往往喜欢用第(3)种形式，用 p++进行控制，程序简洁、高效。初学者在开始时可用第(1)种形式，直观、不易出错。

注意：在使用指针变量指向数组元素时，有以下几个问题要注意：

(1) 可以通过改变指针变量的值指向不同的元素。例如，上述第(3)种方法是用指针变量 p 来指向元素，用 p++使 p 的值不断改变从而指向不同的元素 。

如果不用 p 变化的方法而用数组名 a 变化的方法(例如，用 a++)行不行呢？假如将上述第(3)种方法中的程序的第 8、9 两行改为

```
for(p=a;a<(p+10);a++)
    printf("%d", *a);
```

是不行的。因为数组名 a 代表数组首元素的地址，它是一个指针型常量，它的值在程序运行期间是固定不变的。既然 a 是常量，所以 a++是无法实现的。

(2) 要注意指针变量的当前值。请看下面的例子。

例 8.7 通过指针变量输出整型数组 a 的 10 个元素。

解题思路：用指针变量 p 指向数组元素，通过改变指针变量的值，使 p 先后指向 a[0]～a[9]各元素。

编写程序：

```
#include <stdio.h>
int main()
{ int *p,i,a[10];
  p=a;                                    //p 指向 a[0]
  printf("please enter 10 integer numbers:");
  for(i=0;i<10;i++)
    scanf("%d",p++);                      //输入 10 个整数给 a[0]～a[9]
  for(i=0;i<10;i++,p++)
    printf("%d ", *p);                    //想输出 a[0]～a[9]
  printf("\n");
  return 0;
}
```

运行结果：

```
please enter 10 numbers:0 1 2 3 4 5 6 7 8 9
0 1245052 1245120 4199177 1 4394640 4394432 2367460 1243068 2147340288
```

在不同的环境中运行时显示的数据可能与上面的有所不同。

程序分析：显然输出的数值并不是 a 数组中各元素的值。需要检查和分析程序。

有的人觉得上面的程序没有什么问题，即使已被告知此程序有问题，还是找不出问题出在哪里。问题出在指针变量 p 的指向。请仔细分析 p 的值的变化过程。指针变量 p 的初始值为 a 数组首元素(即 a[0])的地址，见图 8.10 中的①，但经过第 1 个 for 循环读入数据后，p 已指向 a 数组的末尾(见图 8.10 中②)。因此，在执行第 2 个 for 循环时，p 的起始值不是 &a[0]了，而是 a+10。由于执行第 2 个 for 循环时，每次要执行 p++，因此 p 指向的是 a 数组下面的 10 个存储单元(图 8.10 中以虚线表示)，而这些存储单元中的值是不可预料的。

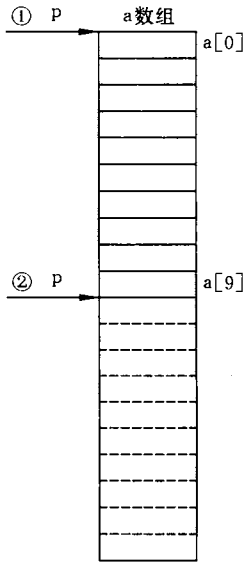

图 8.10

解决这个问题的办法是，只要在第 2 个 for 循环之前加一个赋值语句：

```
p=a;
```

使 p 的初始值重新等于 &a[0]，这样结果就对了。程序为

```
#include <stdio.h>
int main()
{ int i,a[10],*p=a;                    //p的初值是a,p指向a[0]
  printf("please enter 10 integer numbers:");
  for(i=0;i<10;i++)
    scanf("%d",p++);
  p=a;                                 //重新使p指向a[0]
  for(i=0;i<10;i++,p++)
    printf("%d ",*p);
  printf("\n");
  return 0;
}
```

运行结果：

```
please enter 10 integer numbers:0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

显然结果正确。

(1) 从例 8.7 可以看到，虽然定义数组时指定它包含 10 个元素，并用指针变量 p 指向某一数组元素，但是实际上指针变量 p 可以指向数组以后的存储单元。如果在程序中引用数组元素 a[10]，虽然并不存在这个元素(最后一个元素是 a[9])，但 C 编译程序并不认此为非法。系统把它按 *(a+10)处理，即先找出(a+10)的值(是一个地址)，然后找出它指向的单元(*(a+10))的内容。这样做虽然在编译时不出错，但运行结果不是预期的，应避免出现这样的情况。这是程序逻辑上的错误，这种错误比较隐蔽，初学者往往难以发现。在使用指针变量指向数组元素时，应切实保证指向数组中有效的元素。

(2) 指向数组的指针变量也可以带下标，如 p[i]。有些读者可能想不通，因为只有数组才能带下标，表示数组某一元素。带下标的指针变量是什么含义呢？当指针变量指向数组元素时，指针变量可以带下标。因为在程序编译时，对下标的处理方法是转换为地址的，对 p[i]处理成 *(p+i)，如果 p 是指向一个整型数组元素 a[0]，则 p[i]代表 a[i]。但是必须弄清楚 p 的当前值是什么？如果当前 p 指向 a[3]，则 p[2]并不代表 a[2]，而是 a[3+2]，即 a[5]。建议少用这种容易出错的用法。

(3) 利用指针引用数组元素，比较方便灵活，有不少技巧。在专业人员中常喜欢用一些技巧，以使程序简洁。读者在看别人写的程序时可能会遇到一些容易使人混淆的情况，要仔细分析。请分析下面几种情况(设 p 指向数组 a 的首元素(即 p=a))：

① 分析：

```
p++;
*p;
```

p++使 p 指向下一元素 a[1]。然后若再执行 *p，则得到下一个元素 a[1]的值。

② *p++;

由于++和 * 同优先级，结合方向为自右而左，因此它等价于 *(p++)。先引用 p 的值，实现 *p 的运算，然后再使 p 自增 1。

例 8.7 的第 2 个程序中最后一个 for 语句

```
for(i=0;i<10;i++,p++)
    printf("%d", *p);
```

可以改写为

```
for(i=0;i<10;i++)
    printf("%d", *p++);
```

作用完全一样。它们的作用都是先输出 *p 的值，然后使 p 值加 1。这样下一次循环时，*p 就是下一个元素的值。

③ *(p++)与*(++p)作用是否相同？不相同。前者是先取 *p 值，然后使 p 加 1。后者是先使 p 加 1，再取 *p。若 p 初值为 a(即 &a[0])，若输出 *(p++)，得到 a[0]的值，而输出 *(++p)，得到 a[1]的值。

④ ++(*p)。表示 p 所指向的元素值加 1，如果 p=a，则++(*p)相当于++a[0]，若 a[0]的值为 3，则在执行++(*p)(即++a[o])后 a[0]的值为 4。注意：是元素 a[0]的值加 1，而不是指针 p 的值加 1。

⑤ 如果 p 当前指向 a 数组中第 i 个元素 a[i]，则：

(p--)相当于 a[i--]，先对 p 进行“”运算(求 p 所指向的元素的值)，再使 p 自减。

(++p)相当于 a[++i]，先使 p 自加，再进行“”运算。

(--p)相当于 a[--i]，先使 p 自减，再进行“”运算。

将++和--运算符用于指针变量十分有效，可以使指针变量自动向前或向后移动，指向下一个或上一个数组元素。例如，想输出 a 数组的 100 个元素，可以用下面的方法：

```
p=a;
while(p<a+100)
printf("%d", *p++);
```

或

```
p=a;
while(p<a+100)
  {printf("%d", *p); p++;}
```

但如果不小心，很容易弄错。因此在用 *p++形式的运算时，一定要十分小心，弄清楚先取 p 值还是先使 p 加 1。对初学者不建议多用，但应当知道有关的知识。

8.3.4 用数组名作函数参数

在第 7 章中介绍过可以用数组名作函数的参数。例如：

```
int main()
{void fun(int arr[], int n);                    //对 fun 函数的声明
 int array[10];                                 //定义 array 数组
     ⋮
 fun(array,10);                                 //用数组名作函数的参数
 return 0;
```