第5单元　数　　组

数组（array）是一种聚合数据类型，它可以将相关的同类型数据按照某种顺序组织成一个整体。

5.1　一 维 数 组

5.1.1　数组类型的特征

例 5.1　一个小组有 5 位同学，为了用计算机处理这 5 位同学的成绩，在 C 语言中如何表示？

对于这样一个问题，已经具有前面介绍的 C 语言知识的人一定会说：使用下面的声明定义 5 个变量就可以了。

```
float a,b,c,d,e;
```

但这是一种不好的方法。因为这些简单变量无法反映变量之间的关系。假如一个程序中的所有量（例如学生成绩、学生年龄、学生身高、学生体重……）都用简单字母表示，就会造成阅读程序时的混乱。于是，有人想了一个见名知义的命名变量的方法：

```
double stuScore1, stuScore2, stuScore3, stuScore4, stuScore5;
```

这个办法比简单地使用一些字母好多了，但是它们在用于某些群体性数据时无法反映出个体与群体之间的联系，例如这个群体有多大，在这个群体中个体与个体之间有什么样的联系性等，都表现不出来。此外，命名的工作量很大，也不能发挥计算机高速处理的优势，不能使用重复结构进行计算处理（如进行求和）。为了对类似的情况进行有效管理和处理，高级计算机程序设计语言都提供了数组。

1. 数组类型的群体特征

数组类型具有以下 3 个群体性特征。

（1）数组的组成元素是同类型数据，这个类型也称为数组的元素类型或数组的基类型。

（2）数组的组成元素（element）具有逻辑顺序，这种逻辑顺序用下标表示。例如，一组学生成绩可以用数组 stuScore 存储，并分别用 stuScore[0]、stuScore[1]、stuScore[2]、stuScore[3]、stuScore[4]等表示其中每个学生的成绩。通常，被括在一对方括号中的数字称为下标，表示这组数据之间的逻辑顺序关系。

（3）数组中的各元素按照下标的顺序存储在一段连续内存单元中，所以数组也具有物理的顺序性。

2. 数组的个性化参数

数组的个性化参数有以下 3 点：

（1）具体的数据类型。

（2）具体数组的大小。

（3）具体数组的名字具有唯一性。

其中前两个参数是数组的存储细节，也是判断两个数组异同的依据，只有两个数组的元素类型和大小都相同时才认为它们是同类型的数组，但不是同一数组。

5.1.2 数组的定义

数组定义就是根据数组公共属性格式给出具体数组的个性化参数并进行命名的过程，其格式如下：

```
数组基类型 数组名 [数组长度表达式 ];
```

说明：

（1）数组声明中的一对方括号称为数组类型声明符，它表明所声明的名字是一种聚合数据类型的名字，即这种类型中可以存储多个元素，这些元素的类型相同并且具有顺序性。

（2）数组类型和数组长度表达式是由用户补充的存储细节。例如，前面介绍的存储 5 名学生成绩的数组可以定义为

```
double stuScore[5];
```

它表明这个名为 stuScore 的数组存储细节如下：

- 它所存储的元素类型是 double 类型。
- 它所存储的元素最大数目是 5。

（3）在 C 语言中，数组的长度是在编译时计算的，为此数组长度的定义必须使用编译时整数表达式——任何整数常量表达式，例如整数常量、整数宏、字符常量，以及编译时可以直接得到值的其他整数表达式。例如：

```
double stuScore[5];
```

也可以写成

```
double stuScore[2 + 3];                          //正确
```

或

```
#define N 5
double stuScore[N];                              //正确
```

采用宏定义数组长度的好处是比较灵活，如果以后想改变数组大小，仅在宏定义处修改即可。

（4）数组名不能作为左值表达式，因为数组是由若干独立的数组元素组成的，这些元

素不能作为一个整体被赋值。例如：

```
int x[5],y[5];
x = y;                          //错误
```

5.1.3 数组的初始化

在定义数组时，若仅仅声明了一个数组的名字，则这个数组中每个元素的值可能是不确定的。数组初始化就是在定义数组时给数组元素以确定的值。

1．数组全部初始化

数组的初始化用初始化列表进行。初始化列表是括在花括号中的一组表达式。例如：

```
double stuScore[5] = {87.6,90.7 + 8,76.5,65.4,56.7};
```

这时，声明语句中可以省略数组大小，由编译器按照初始化列表中的数据个数确定数组的大小。例如：

```
double stuScore[] = {87.6,98.7,76.5,65.4,56.7};
```

2．数组开始部分元素初始化

当一个数组的初始化列表比数组短时，则后面的元素按照静态方式默认初始化：对于int类型，被初始化为0；对于double类型，被初始化为0.0。例如：

```
double stuScore[5] = {87.6,98.7};
```

相当于使用{87.6,98.7,0.0,0.0,0.0}进行初始化。

显然，当只有部分元素的初始化列表时不可以省略数组大小，利用这一特点可以很方便地将一个数组初始化为全零。例如：

```
double stuScore[5] = {0.0};
```

就将每个元素都初始化为0.0。

3．数组元素的下标指定初始化

下标是对于数组元素的标号，其起始值为0，最大值为数组大小减1。C99允许用下标对指定元素初始化。例如：

```
double stuScore[5] = {[3] = 76.5, [1] = 98.7, [0] = 87.6};
```

这个数组的大小为5，其下标取值为0、1、2、3、4。在这个声明中，指定对下标为3、1和0的元素分别初始化为76.5、98.7和87.6。其他元素被默认初始化为0。显然，这时不再需要按照顺序。若省略数组大小，则编译器默认数组大小为下标最大值 +1，例如：

```
double stuScore [ ] ={[4] = 56.7, [1] = 98.7, [3] =65.4};
```

默认数组大小为 4 + 1 = 5。

下面的数组定义也是正确的：

```
int a[] = {0,1,2,[0] = 3,4,5};
```

它将元素 0 初始化了两次，即先初始化为 0，后又初始化为 3。

5.1.4 下标变量

1. 下标变量的概念

一个数组被定义之后就可以用下标变量对其元素随机访问了。下标变量用数组名加上括在方括号中表示逻辑序号的整数表达式表示，这个表达式称为下标表达式，简称下标（subscripting）。例如在定义了数组 stuScore 以后可以用 stuScore[0]、stuScore[1]、stuScore[2]等表示该数组的各个下标变量。

用下标变量可以随机地访问数组中的任何一个元素，对其赋值或引用其值。

代码 5-1 打印一组学生成绩。

```
#include <stdio.h>
int main (void){
    double stuScore[] = {87.6,98.7,76.5,65.4,56.7};          //定义一个数组存储学生成绩
    printf ("该组学生的成绩为:\n");
    for(int i = 0; i < sizeof stuScore/sizeof stuScore[0]; ++ i){
        printf("%.1lf,", stuScore[i]);
    }
    printf("\n", stuScore[i]);
    return 0;
}
```

说明：

（1）sizeof 是 C 语言中的一个操作符，可以用来计算一个表达式或一种类型所占用（或所需要）的内存字节数。所以表达式 sizeof stuScore 的值是数组 stuScore 占用的内存字节数，sizeof stuScore[0]的值是数组元素 stuScore[0]占用的内存字节数。二者相除，得到的是数组 stuScore 的元素个数。这种写法可以不考虑小组成员到底有多少人，有几个成绩就有多少人。而下面的写法必须先数有几个成绩，增加了出错的几率。

```
for (int i = 0; i < 5; ++ i){
    printf("%.1f,", stuScore[i]);
}
```

（2）在输出格式字段“%.1f”中用“.1”表示小数部分只保留 1 位。

运行该程序，输出结果如下。

```
该组学生的成绩为:
87.6,98.7,76.5,65.4,56.7
```

2. 使用下标变量应注意的事项

（1）在数组声明中，数组名后紧靠的一对方括号称为数组定义符，它要求为一个整数常量表达式。在下标变量中，数组名后紧靠的一对方括号称为下标操作符，它要求为整数表达式，不要求必须是常量，并执行对数组取下标操作或称对数组索引（indexing）操作。C 语言规定数组下标的取值始终从 0 开始，所以一个长度为 N 的数组的下标是从 0 到 $N-1$。

（2）C 语言将字符作为整数处理，因此也可以用字符作为下标。但为了把字符放到合适的范围内，可以用 *a*[ch – 'a']表示 *a*[0]。

（3）C 语言标准没有规定要对下标范围进行检查。因此，当程序员使用了超出数组的最大下标值（最常见的是错将 *n* 当成下标最大值）时就会形成未定义操作。例如，对于图 5.1 所示的内存内容，代码 5-2 将是一个无限循环。

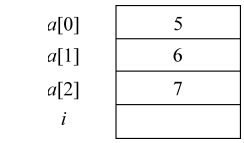

图 5.1　数组越界隐患示例

代码 5-2　数组越界造成的无限循环。

```
#define N 3
int a[N];
//…
for(int i = 1; i <= N; i ++ )
    a[i] = 0;
//…
```

当这段程序执行完 $i=1$、$i=2$ 时，会把 *a*[1]、*a*[2]都赋值为 0。而执行到 $i=3$ 时，会把 *a*[3]这个位置的内存也赋值为 0。但是内存中 *a*[3]的位置已经超出了数组 *a* 的范围，保存的是变量 *i* 的值，所以执行到 $i=3$ 时是把 *i* 赋值为 0。这样，就永远不会退出循环。

当然，若 *i* 不是正好保存在 *a*[3]的位置，就可能不会形成死循环了。不过这总是一个隐患，并已经成为黑客窃取信息的一种手段。

（4）下标表达式可以产生左值，而数组名不是左值。

（5）用户要注意下标表达式中的副作用。例如在表达式[*i* ++] = *a*[*i* ++] + 2 中，*i* 被修改了两次，称为实现定义行为。为避免发生这种行为，可以修改为 *a*[*i* ++] + = 2，或 *i* ++，*a*[*i*] = *a*[*i*] + 2，或 *i* ++，*a*[*i*] + = 2。

再看表达式 *a*[*i*] = *i* ++，虽然在这个表达式中对于 *i* 只修改了一次，但在修改 *a*[*i*]时要读取 *i* 的值，但无法保证读取的是哪个 *i* 值。实际上还是一个修改叠加问题。

类似的问题还有以下程序段：

```
int i = 0;
while(i < N)
    a[i] = b[i++ ];
```

5.1.5　变长数组与常量数组

1. 变长数组

C99 支持 VLA（variable-length array，变长数组），VLA 有以下特点。

（1）其长度不是编译时计算，而是运行时计算，因此长度可以用任意整型表达式定义。
（2）没有初始化式。
（3）一般用在 main()之外的其他函数中，每次调用时长度可以不同。

2. 常量数组

C 语言允许将数组定义为常量数组，使每个元素的值都不可再修改。定义常量数组的方法是在数组定义的开始处加上关键字 const，例如：

```
const char octalChars[] = { '0', '1', '2', '3', '4', '5', '6', '7'};
```

这说明，关键字 const 不仅可以保护数组元素不被修改，还可以保护任何一个变量的值不被修改。

5.2 排序与查找

排序（sorting）也称分类，其目的是将一组“无序”的记录序列调整为“有序”的记录序列。排序方法有很多，主要可以分为选择排序、插入排序、交换排序、归并排序等。不同的排序算法有不同的时间效率和空间效率（多用的存储空间），适合不同的情况。

在多个有序的或无序的数据元素中，通过一定的方法找出与给定关键字相同的数据元素的过程称为查找（search）。通常，查找的输入有一组数据（如一个数组）、一个关键字。查找的输出分两种情形，若查找成功，则输出查找到的数据；若查找结束，没有相同的关键字，则输出找不到的信息。

5.2.1 直接选择排序

1. 直接选择排序的基本思路

选择排序（selection sort）的基本思路是把序列分为两个部分，即已排序序列和未排序序列。开始前，已排序序列没有元素，未排序序列具有全部元素。若要进行升序排序每次就要从未排序序列中选择一个最小值放进已排序序列，直到把未排序序列中的元素都放进已排序序列为止。为了节省空间，可以按照下面的算法进行：首先从未排序序列中选择一个最小元素与第 1 个元素交换，然后把第 1 个元素作为已排序序列，把其余的元素作为未排序序列；接着再从未排序序列中选择一个最小元素与未排序序列的第 1 个元素交换，使已排序序列增加一个元素……；如此重复 $N-1$ 次，就把具有 N 个元素的序列排好序了。这种算法称为直接选择排序。图 5.2 为采用直接选择排序对初始序列进行降序排序的情况。

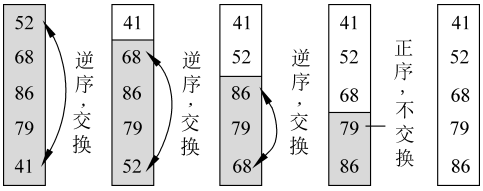

图 5.2 直接选择排序算法示例

2. 直接选择排序程序

代码 5-3 一个简单选择排序函数。

```
void seleSort(int a[],int size){
     int k,min,temp;
     for (int i = 0; i < size - 1; ++ i ){
          min = a[i];
          k = i;
          for (int j = i + 1; j < size; j  ++ )      //在后面的数列中选择比 min 小的元素
               if (min > a[j]){
                    min = a[j];                      //更新最小值
                    k = j;                           //记录当前最小值的位置
               }
          if(k  != i){                               //交换,形成已排序序列的最后一个元素

               temp = a[i]; a[i] = a[k]; a[k] = temp;
          }
    }
}
```

说明：在这个算法中记录了最小元素 min 和它的位置 k。略加分析可以看出，这两者是互相联系的。为此可以省略 min，得到下面的算法。

代码 5-4 修改后的简单选择排序函数。

```
void seleSort(int a[],int size){
     int k,temp;
     for (int i = 0; i < size - 1; ++ i){
          k = i;
          for (int j = i + 1; j < size; j ++ )       //在后面的数列中选择一个最小元素位置
               if (a[k] > a[j])
                    k = j;                           //记录当前最小值的位置
          if(k  != i){                               //交换,形成已排序序列的最后一个元素

               temp = a[i]; a[i] = a[k]; a[k] = temp;
          }
    }
}
```

3. 测试设计

输入：任意一个数列。

输出：已经排序的数列。

4. 测试

代码 5-5 代码 5-4 的测试程序。

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include  <time.h>
#define N 9
void dispAllElemenNumberts(int score[], int size);  //输出函数原型声明
void seleSort(int a[], int size);                   //选择排序函数原型声明

int main (void) {
   int stuScore[N];                                 //定义一个数组存储学生成绩

     srand (time (00));                             //用时间函数作为伪随机数序列种子
     for (int i = 0; i < N; ++ i){
          stuScore[i] = rand () % 101;              //产生一个[0,100]区间的随机数赋值给下标变量
     }
     printf("排序前的序列：");
     dispAllElemenNumberts(stuScore,N);             //显示所有元素值
     seleSort(stuScore,N);                          //排序
     printf("排序后的序列：");
     dispAllElemenNumberts(stuScore,N);             //显示所有元素值
     return 0;
}

void dispAllElemenNumberts(int a[],int size){
     for (int i = 0; i < size; ++ i)
          printf("%d,",a[i] );
     printf("\n");
}
```

一次测试结果如下。

```
排序前的序列:71,33,94,72,11,29,91,35,88.
排序后的序列:11,29,33,35,71,72,88,91,94.
```

5.2.2 冒泡排序

1. 冒泡排序算法的基本思路

冒泡排序（bubble sort）是一种典型的交换排序算法。它的基本思路是按一定的规则比较待排序序列中的两个数，如果是逆序，就交换这两个数；否则继续比较另外一对数，直到将全部数都排好为止。冒泡排序是通过对未排序序列中两个相邻元素的比较交换来实现排序过程。图 5.3 为用冒泡排序对数据序列[7,5,3,9,1]进行升序排序的过程，其基本算法是从待排序序列的一端开始，首先对第 1 个元素（7）和第 2 个元素（5）进行比较，当发现逆序时进行一次交换；接着对现在的第 2 个元素（7）和第 3 个元素（3）进行比较，当发现逆序时进行一次交换；如此下去，直到对第 n–1 个元素（9）和第 n 个元素（1）比较交换完为止。这时，最大的一个元素（9）便被“沉”到了最后一个元素的位置上，成为已排序序列中的一个元素，未排序序列成为[5,3,7,1]。接着再重新对这个未排序序列进行比较交换，将次大元素（7）“沉”到倒数第 2 个元素的位置上。如此操作，直到没有元素需要交换为止。

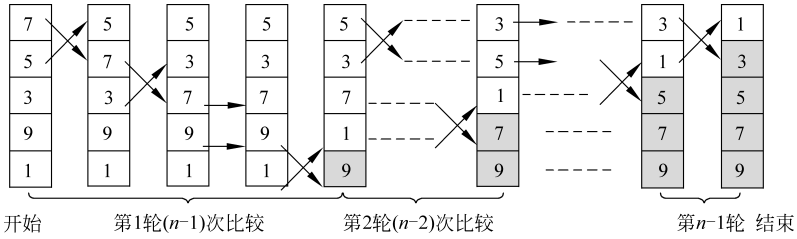

图 5.3　冒泡排序示例

2. 冒泡排序程序

代码 5-6　一个冒泡排序函数。

```
void bubbleSort (int a[], int size) {
     int temp;
     for (int j = 0; j < size - 1; j  ++ )                    //总的比较交换轮数
          for (int i = 0; i < size - j; ++ i)                 //每轮中的比较交换次数
               if (a[i] > a[i + 1]) {
                    temp = a[i];
                    a[i] = a[i + 1];
                    a[i + 1] = temp;
               }
}
```

说明：

（1）在这个函数中，当 $j = 0$ 时，内层循环变量 i = size −1 后，if ($a[i] > a[i + 1]$)中的 $a[i + 1]$将越界。为了避免产生这种情况，可以使数组元素 $a[0]$空闲，数据从 $a[1]$开始存储。与此相对应，函数 bubbleSort ()中的循环也从 1 开始。

（2）在这个算法中，有可能出现某一轮的两两比较后不需要交换的情况。这种情况说明，所有元素的位置都是不需要变动的，就是一个已经排好序的序列了。到此为止，就不需要再进行后面的两两比较交换了。这样可以提高排序的效率。那么，如何判断一轮中有无交换呢？一个简单的方法是在进入一轮前设置一个交换标志（例如 exchange）为−1，只要进行了交换，就让交换标志为 1。这样，用这个交换标志就可以知道待排序序列是否已经全部有序。

代码 5-7　改进的冒泡排序函数。

```
void bubbleSort (int a[], int size) {
     int temp,exchange;
     for (int j = 1; j < size - 1; j  ++ ) {
          exchange = -1;                                      //进入每一轮前设置一个交换标志为-1
          for (int i = 1; i < size - j; ++ i)
               if (a[i] > a[i + 1]) {
                    temp = a[i];
                    a[i] = a[i + 1];
                    a[i + 1] = temp;
                    exchange = 1;                             //只要有交换,就使交换标志改变为 1
               }
```

```
        if (exchange == -1)                                    //交换标志若为-1,就返回
            return;
    }
}
```

3. 程序测试

测试程序与代码 5-5 基本相同，但要进行如下修改。

（1）修改排序函数及其声明。

（2）将数组元素 *a*[0]赋值为–1，因为学生成绩不可能为–1。

（3）将数组元素的输入部分的循环变量起始值修改为 1。

（4）将 dispAllElemenNumberts()函数中的循环变量起始值修改为 1。

代码 5-8 代码 5-7 的测试程序。

```
#include <stdio.h>
#include <stdlib.h>
#include  <time.h>
#define N 9
void dispAllElemenNumberts(int score[], int size);       //输出函数原型声明
void bubbleSort (int a[], int size) ;                    //冒泡排序函数原型声明


int main (void) {
    int stuScore[N];
    stuScore[0] = -1;

    srand (time (00));
    for (int i = 1; i < N; ++ i ){                       //修改循环变量起始值
        stuScore[i] = rand () % 101;
    }
    printf("排序前的序列：");
    dispAllElemenNumberts(stuScore,N);
    bubbleSort Sort(stuScore,N);                         //排序
    printf("排序后的序列：");
    dispAllElemenNumberts(stuScore,N);
    return 0;
}

void dispAllElemenNumberts(int a[],int size){
    for (int i = 1; i < size; ++ i)                      //修改循环变量起始值
        printf("%d, ",a[i] );
    printf("\n");
}
```

一次测试结果如下。

```
排序前的序列：38,9,26,92,51,39,45,17,94,
排序后的序列：9,17,26,38,39,45,51,92,94,
```