数组

7.1 问题引入

在前面几章的解决方案中，设计了学生基本信息的存储方案。例如得分是这样来声明的：

```
int score;
```

然而，这单个的变量只能保存一个学生的得分信息，实际情况却是需要 30 名学生的信息。因此，不得不声明 30 个类似的变量来存储所有人的得分情况，它们的声明可能是这样的：

```
int score1;
int score2;
…
int score30;
```

那么在输入得分信息时，会用到这样一系列语句：

```
scanf("%d", &score1);
scanf("%d", &score2);
…
scanf("%d", &score30);
```

也许 30 条这样的语句并不算多。但如果参赛学生增多了，达到了数百个，那么还能用这样的方式吗？

也许读者会想到这样的方案，用一个循环输入得分，其代码如下：

```
for (i = 0; i < NUM; ++i) scanf("%d", &scorei);
```

然而不幸的是，这样做并不能达到目的：编译器不能自动将读者认为正确的组合 scorei 扩展为 score1、score2 等变量名，而是视它为单个变量名！这在逻辑上显然是不正确的。

仔细分析一下需要处理的这些变量，可以发现它们都拥有相同的特征：其一，这些数据具有明显的聚集性；其二，它们都用于描述不同对象的相同属性，有相同的数据类型，也有相同的处理方式。借用数学上的术语，它们是“齐次”的。

对于具有这样性质的数据，几乎所有的高级语言都提供了有效的解决方案，这就是**数组**（*array*）。简而言之，数组是具有相同属性的对象的有限有序集合。数组中，对象（称为“数组元素 *element*”）的数目是有限的，并且按顺序排列，每个对象（除首尾外）都有且仅有一个直接前驱，也有且仅有一个直接后继。数组元素使用相同的命名规则，这使得程序员可以用相同的方式处理这些元素，从而减少了代码的重复。

数组按照其组织方式可以分为一维数组、二维数组以及多维数组等。在本章中主要讨论一维和二维数组。

7.2 一维数组

7.2.1 一维数组的声明

一维数组的声明方式为：

基类型　　数组名[长度];

其中，基类型可以是任意合法类型；长度是一个整型常量表达式，说明了数组中有多少个元素；数组中的每一个元素都具有基类型。例如：

```
int a[5];
```

以上声明定义了一个数组变量，名字为 a，是一维的，有 5 个元素，每个元素都是整型的。

上述声明在声明一个数组的同时，还声明了一个数组类型，该类型可以描述为：

```
int [5]
```

可以看到，数组类型修饰符由两部分组成，并且分别出现在变量名的前后；相较之下，其他类型修饰符（函数除外）一般都出现在变量名的前面。

考察两个数组变量是否具有相同的类型（即可比较的）要看它们的 3 个指标：维数、每一维的长度、基类型。例如有如下声明。

```
int c[5], d[5], e[10];
float f[5];
```

那么 c 和 d 的类型是相同的、可比较的；而 c 和 f 则不同，d 和 e 也不同。

数组具有这样一些特性：

- 数组的基类型可以是任意合法的类型，包括内建类型和用户自定义类型，甚至仍是一个数组。
- 数组一旦声明，那么数组名就被视为**常量**，因此不能被更改。
- C 数组的长度是固定的，一旦确定就不能更改。
- 一维数组在内存中是顺序存放的，元素是两两相邻的。

例如数组 a，它的存储结构如图 7-1 所示。

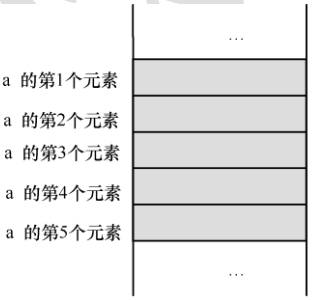

图 7-1　数组的顺序存储结构

可以看到，一维数组是一种线性结构，它主要用于存储按线性方式排列的数据。

数组名是一个常量，因此数组变量本身不是一个左值，所以单独的数组名不能出现在赋值号的左边。

> 对数组的整体赋值和输入、输出是错误的。例如：
>
> ```
> int a[10], b[10]; //ok
> b = a; //error, b 不是左值
> scanf("%d", a);   //error
> printf("%d", b);  //error
> ```
>
> 如果要对数组进行赋值或输入、输出，那么必须对单个的数组元素进行操作。

7.2.2　一维数组元素的使用

数组元素是组成数组的基本单元，同时也是一种变量，其标识方法为数组名后跟一个该元素在数组中的索引号。数组元素的表示形式为：

数组名[索引]

其中，索引（*index*）常称为“**下标**”，是元素在数组中的序号，可以是整型常量或整型表达式。C 语言规定，数组的下标从 0 开始计数。例如有声明：

```
int a[10];
int i = 2;
```

```
float b = 1.5;
```

那么 a[0]、a[i]、a[2+3]都是合法的数组元素表示；而 a[b]就是错误的。

可以这样来理解数组元素：a[i]是一个变量的名字，它的类型是整型。既然是一个变量，那么它就是一个左值，可以出现在赋值号的两边。

> ⓘ 由于 C 数组的下标是从 0 开始的，因此下标的最大值应该是数组的长度减 1。如上面定义的数组 a，它的最大下标是 9。下标值超过了 9 或者小于 0 的现象称为“下标越界”，例如 a[10]。但很遗憾的是，因为 C 语言没有把数组当作一个完整的对象来对待，而只是把它当作一个内存块，所以 C 语言中没有检测下标是否越界的机制。因此，在使用数组时，一定要注意下标是否超出了界限。

【例 7-1】数组元素的使用：输入和输出数组元素。

```
//7-1.c
#include <stdio.h>

int main()
{
    int a[10];
    int i;

    printf("Please input 10 integers:\n");
    for (i = 0; i < 10; ++i)
        scanf("%d", &a[i]); //注意这种用法

    printf("The elements are following:\n");
    for (i = 0; i < 10; ++i)
        printf("a[%d]=%d\n", i, a[i]);

    return 0;
}
```

程序的运行结果是：

```
Please input 10 integers:
2 6 5 9 0 7 4 8 1 3↙
The elements are following:
a[0]=2
a[1]=6
a[2]=5
a[3]=9
a[4]=0
a[5]=7
a[6]=4
a[7]=8
a[8]=1
a[9]=3
```

从例中可以看到，数组的连续存储结构和统一的命名规则使得对它的处理和循环结构紧密相关。一般地，处理 *n* 维数组需要 *n* 重循环。

一般地，将数组元素从头到尾访问一遍的过程称为“数组的遍历（*traverse*）”。遍历数组是经常用到的一项数组操作，在后面的例子中都可以看到它的应用。

7.2.3 一维数组的初始化

在前面的例子中可以看到，给数组元素赋予初始值是用输入或者赋值的方法完成的。这有时显得很费事。C 语言提供了更直接地为数组赋初值的方法，这就是数组的初始化。

数组初始化是指在数组声明时给数组元素赋予初值。数组初始化是在编译阶段进行的。其语法为：

基类型　数组名[常量表达式]={值列表};

其中，值列表是用逗号隔开的值序列，每个值的类型都应该与数组的基类型一样。例如：

```
int a[10]={ 0,1,2,3,4,5,6,7,8,9 };
```

这个初始化将 0 给了 a[0]，1 给了 a[1]，并以此类推。

C 语言对数组的初始化有以下几点规定。

（1）对于自动（即局部的非静态）数组，如果未对其进行初始化，那么所有元素的值都是未知的，因此不能直接使用。不能想当然地认为编译器会自动将所有元素都初始化为 0。

（2）对于全局和静态数组，编译器会自动将所有元素初始化为 0。

（3）可以只初始化部分元素。当值列表的长度小于数组长度时，只初始化前面部分元素。例如：

```
int a[10]={0,1,2,3,4};
```

表示只初始化了 a[0]～a[4]这 5 个元素，而后 5 个元素被自动初始化为 0。这意味着，一旦有元素被初始化，那么编译器会自动用 0 将剩余的元素初始化。

（4）如果在初始化时省略了数组的长度说明，那么数组的长度就被定为值列表的长度。例如：

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

则数组 a 的长度被编译器定为 5。

1. 值列表长度大于数组指定长度是个错误。例如：

```
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

2. 试图用单值将数组所有元素初始化为相同值是办不到的。例如：

```
int a[10] = 1;
```

而 int a[10] = {1}是正确的，但只将 a[0]初始化为 1，其余元素置 0。

【例 7-2】找出一个数组中的最大值并输出该值。

【解题思路】这个问题可以归类为数组的查找问题。查找也是一种遍历，其基本操作就是比较。假设数组为 a，那么可以这么做：声明变量 max，令 max=a[0]，然后在一个循环中将其与数组中的其他元素 a[i]逐一比较，只要 a[i]比 max 大，就将 max 的值替换为 a[i]的值。经过一次扫描后，max 的值一定是数组中最大的那个元素的值。

```
//7-2.c
#include <stdio.h>

#define LEN 9

int main()
{
    int a[LEN] = {33, 77, 99, 44, 66, 88, 22, 55, 11};
    int max = a[0];
    int i;
```

```
    for (i = 0; i < LEN; ++i)
        if (a[i] > max) max = a[i];

    printf("The array is:\n");
    for (i = 0; i < LEN; ++i)
        printf("%4d", a[i]);

    printf("\nmax = %d\n", max);

    return 0;
}
```

程序的运行结果为：

```
The array is:
  33  77  99  44  66  88  22  55  11
max = 99
```

7.2.5 一维数组的应用

在很多的算法和程序中都会用到一维数组，而其中的经典就是数组的查找和排序。

1．数组查找

数组查找是指在数组中查找指定的元素及其位置。根据数组元素有无顺序，查找的算法有很大不同。这里先介绍顺序查找法。

【例 7-5】给定一个数组和一个关键值，查找这个关键值在数组中的位置。

【解题思路】当数组中的元素没有按某种规则形成递增或递减的序列时，顺序查找法可能是唯一的方法。其基本思想是：将指定的关键值与数组中的元素逐一比较，直到相等为止。如果扫描完整个数组都没有找到，则查找失败。如果找到，则返回元素的下标，否则返回负值。下面是程序代码。

```
//7-5.c
#include <stdio.h>

int find(int key, int a[], int len);

int main()
{
    int a[8] = { 12, 23, 34, 45, 56, 67, 78, 89 };
    int i;
    int key = 45, result;

    printf("The array is:\n");
    for (i = 0; i < 8; ++i)
        printf("%4d", a[i]);
    putchar('\n');

    result = find(key, a, 8);
    if (result == -1)
        printf("Not found key %d.\n", key);
    else
```

```
        printf("Found key %d@%d\n", key, result);

    return 0;
}

int find(int key, int a[], int len)
{
    int i;

    for (i = 0; i < len; ++i)
        if (key == a[i]) return i;

    return -1;
}
```

程序的运行结果是：

```
The array is:
  12  23  34  45  56  67  78  89
Found key 45@3
```

另一次运行结果为：

```
The array is:
  12  23  34  45  56  67  78  89
Not found key 99.
```

顺序查找法非常简单，可以在任何场合使用。但它有一个较大的缺陷，就是效率低。这里简单计算一下算法的效率。假设数组的长度为 *n*，那么在最好情况下只需 1 次查找就可以了，而在最坏情况下需要 *n* 次比较，平均比较次数是(1+*n*)/2。很明显，在 *n* 不大的情况下没有什么问题，但如果 *n* 很大的时候，算法的效率就会很低。

> 在衡量算法的效率时，常用“时间复杂度”这一概念。时间复杂度的度量可以粗略地用算法中主要步骤重复执行的次数来计算。一般情况下，执行次数是循环次数或者数组长度 *n* 的函数，那么时间复杂度可以表示为
>
> $O(f(n))$
>
> 其中，$f(n)$只取函数的最高次项，而去掉其他的低次项及系数。
>
> 在【例 7-5】中，$f(n)=(1+n)/2$，因此算法的时间复杂为是 $O(n)$。这是一个线性函数。

2．数组排序

顺序查找法的时间复杂度不理想，因此一些更好的查找算法被开发出来。但这些算法都有一个前提，就是数组是排好序（*sorted*）的，意思是数组的元素已按某种规则按一定顺序排列。所以，在介绍这些算法之前，先来看看数组的排序。

数组排序的方法有很多，这里介绍最简单的冒泡排序法。

【例 7-6】冒泡排序法

【解题思路】顾名思义，冒泡排序法的得名来源于模拟了水中气泡上升的过程：大的气泡上升得快，小的则慢，这些气泡就形成了一列从小到大排列的顺序，这称为“升序（*ascending*）”，反之称为“降序（*descending*）”。本例中只介绍升序算法。

冒泡排序法的基本思想是：从头到尾扫描数组，并在扫描过程中，比较紧邻的两个元素。如果大值在前，则交换两个元素的值，以使大值后移。然后依次类推。可以确定，当一次扫描结束

后，数组中的最大值一定被移到了数组的尾部。那么，这个数组就已经有序了吗？

答案显然不是。一次扫描只能使最大值到了尾部，而不能保证前面的元素是按升序排列的。因此，这样的扫描和移动还要进行多次。但要进行多少次呢？

这里分析一下每次扫描的情况。假设数组长度为 n，那么第一次扫描需要处理 n 个元素，进行 $n-1$ 次比较。此后，最大值已经移到尾部。那么，第 2 次扫描只需处理前面 $n-1$ 个元素，而不必再处理最后一个了。这样，第 2 次扫描的比较次数就是 $n-2$。依次类推，每次扫描处理的元素和比较次数都比上一次的少 1。归纳起来，第 i 次扫描需要处理 $n-i+1$ 个元素，进行 $n-i$ 次比较。最后一趟需要处理的元素是 2 个，比较次数是 1 次。一旦所有元素处理完，数组将呈现出升序排列状态。

总结一下需要扫描和比较的次数为：

（1）第一次扫描处理的元素个数是 n，最后一次是 2 个，因此扫描次数是 $n-1$ 次；

（2）在第 i 次扫描中，处理元素个数是 $n-i+1$，因此比较次数是 $n-i$ 次。

可以看到，多次扫描需要一个循环，而每次扫描的比较也需要一个循环。因此，完整的排序需要一个双重循环来进行：外层循环需要重复 $n-1$ 次，并且是个减量循环；而第 i 次的内层循环是增量循环，要重复 i 次。

根据上面的描述，可以设计出这样的算法。

设有数组 a，其长度为 N，那么

（1）令 scanPhase = N－1；

（2）令 i = 0；

（3）if a[i] > a[i+1] then 交换 a[i]和 a[i+1]的值；

（4）i = i+1;

（5）if i < scanPhase then 转到第 3 步；

（6）否则，scanPhase = scanPhase－1;

（7）if scanPhase >=1then 转到第 2 步；

（8）否则，程序结束。

算法的流程图如图 7-2 所示。根据算法，可以写出下面的程序。

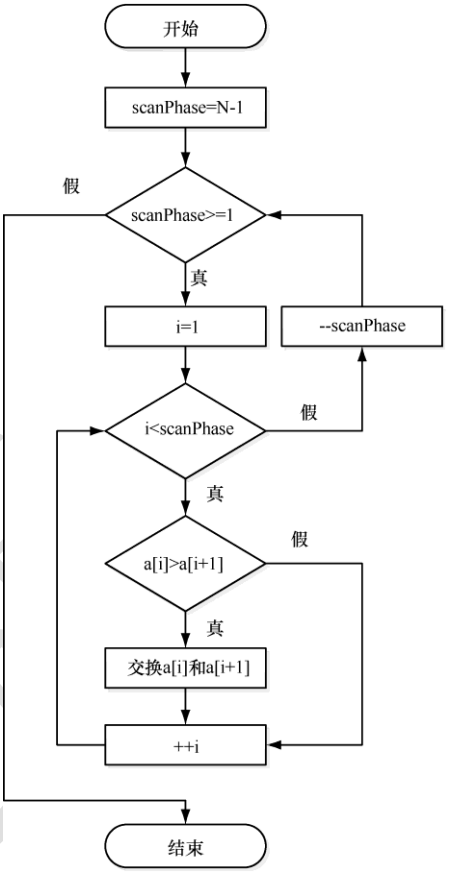

图 7-2　冒泡排序法的流程图

```
//7-6.c
#include <stdio.h>

void sort(int a[], int len)
{
    int scanPhase, i;

    for (scanPhase = len - 1; scanPhase >= 1; --scanPhase)
        for (i = 0; i < scanPhase; ++i)
            if (a[i] > a[i+1])
            {
                int t;
                t=a[i];
                a[i]=a[i+1];
                a[i+1]=t;
            }
}

int main()
{
    #define LEN 10
    int a[LEN];
    int i;

    printf("Please input 10 integers:\n");
    for (i = 0; i < LEN; ++i) scanf("%d", &a[i]);

    sort(a, LEN);

    printf("The sorted array is:\n");
    for (i = 0; i < LEN; ++i) printf("%4d", a[i]);

    return 0;
}
```

程序的运行结果如下：

```
Please input 10 integers:
8 5 7 2 9 0 1 4 3 6
The sorted array is:
   0   1   2   3   4   5   6   7   8   9
```

冒泡排序算法用了双重循环，其循环次数为(n−1)+(n−2)+…+1=n(n−1)/2，因此其时间复杂度为 $O(n^2)$。

3．更高效的查找算法

对于已排序的数组，有更高效的算法实现查找，其中较为常见的是折半查找法。

【例 7-7】折半查找法。

【解题思路】在介绍折半查找法之前，先请读者做一个小游戏。请读者邀请一位同伴并请他/她想一个 1～100 之间的数，然后读者去猜这个数是多少。规则是读者没有猜中时同伴只能回答“大了”或“小了”，直到猜中为止。请读者构思一下，如何才能尽快地猜到这个数呢？

假设同伴想的数是 83。读者可能会用这样的方法进行：

（1）50--->小了（那么下一次猜测区间一定在 51～100 之间）

（2）74--->小了（那么下一次猜测区间一定在 75～100 之间）

（3）86--->大了（那么下一次猜测区间一定在 75～85 之间）

（4）81--->小了（那么下一次猜测区间一定在 82～85 之间）

（5）83--->猜中！

从上述过程可以看到，每次读者猜测的数都是猜测区间的中点。这样可以确保最多 5 次就可以猜中。而靠随机猜中的几率就小得多了。

上面的猜数游戏指明了快速查找的思路。实际上，1～100 构成了一个有序递增序列，和题目给出的数组相似。据此，可以设计出这样的查找算法：对于一个已排序（这点很重要）的数组，假设其为升序排列，首先将查找的起点设为数组的第一个元素，终点设为最后一个元素；然后求这两个位置的中点，将要查找的关键字 key 与中点位置的该元素进行比较。比较的结果有 3 种。

- 如果 key 等于该元素，则查找成功。
- 如果 key 小于该元素，那么 key 可能的位置一定在数组的前半段。那么将查找范围缩小到前半段，重复同样的方法：起点还是原来的起点，但终点由中点代替，依此类推，直到找到为止；或者在仅剩一个元素时还没有找到，那么查找失败。
- 如果 key 大于该元素，那么就在数组的后半段查找，其方法与小于的情况类似。

根据上述规则，可以制定以下算法。

设有长度为 *n* 的数组 a，那么：

（1）定义 begin、mid、end 分别为数组头、中点、尾元素的下标；

（2）令 begin = 0，end = n – 1；

（3）if begin>end then 查找失败，转到第 8 步；

（4）令 mid = (begin + end) / 2；

（5）if key == a[mid] then 查找成功，结束；

（6）else if key < a[mid] then end = mid – 1，转到第 3 步；

（7）else key > a[mid] then begin = mid + 1，转到第 3 步；

（8）结束。

算法的流程图如图 7-3 所示。这种每次将查找区间缩减为上一次的一半的算法称为“折半查找法”，其编码如下：

```
//7-7.c
#include <stdio.h>

int find(int key, int a[], int len);

int main()
{
    const int LEN = 9;
    int a[] = {11, 22, 33, 44, 55, 66, 77, 88, 99};
    int key = 77, result;

    result = find(key, a, LEN);
    if (result != -1) printf("Found key %d@%d.\n", key, result);
    else
       printf("Key %d not found.\n");

    return 0;
}

int find (int key, int a[], int len)
{
    int begin = 0, end = len - 1;
    int mid;

    while (begin<= end)
    {
        mid = (begin + end) / 2;
        if (a[mid] == key) return mid;

        if (a[mid] < key)
            begin = mid + 1;
        else
            end = mid - 1;
    }

    return -1;
}
```

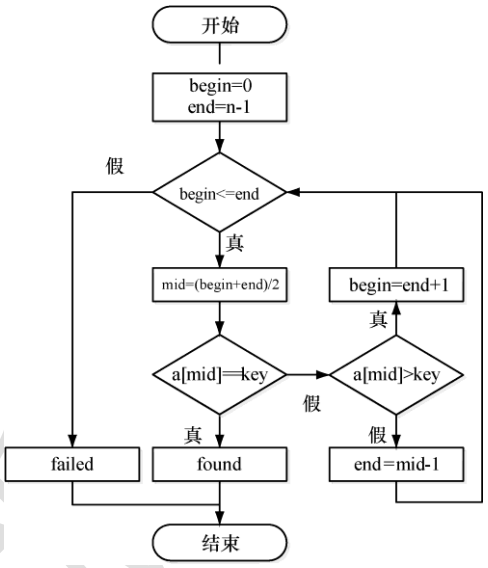

图 7-3　折半查找法的流程图

程序的运行结果如下：

```
Found key 77@6.
```

> 折半查找法是一种收敛很快的算法。由于每次查找区域的元素个数是上次的一半，因此，在最坏情况下，如果查找次数为 s，那么有 $2^s=n$，则 $s=\log_2 n$，也就是说，折半查找法的时间复杂度为 $O(\log_2 n)$。比起顺序查找法，它在效率上有明显的优势。但它也有一个缺陷，就是只能对已排序数组用这种方法查找。

7.3 二维数组

在日常工作中，常会用到二维表格来表示一种用行和列来表达的阵列，例如 Excel 中的工作表，数学计算中用到的行列式和矩阵等。在 C 语言中，也有相似的表示，这就是二维数组。

7.3.1 二维数组的声明和使用

二维数组声明的一般形式：

基类型 数组名[常量表达式 1][常量表达式 2];

其中，常量表达式 1 表示第一维（行）的长度，常量表达式 2 表示第二维（列）的长度。例如：

```
int a[3][4];
```

说明了一个 3 行 4 列的数组，数组名为 a，其基类型为整型。该数组的元素共有 3×4=12 个；该数组的类型是 int [3][4]。

要使用二维数组元素，可以这样引用：

数组名[行号][列号];

a[0][0]、a[i][j]都是对数组元素的合法引用。

与一维数组一样，形如 a[i][j]的元素可以被视为是一个整型变量。此外，二维数组的行号和列号都是从 0 开始的。

二维数组常用于表示一种二维结构，比如二维棋盘、课表等。图 7-4 示意了二维数组的结构。

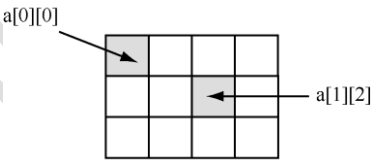

图 7-4 二维数组的结构

现在来看一个实例。

【例 7-9】打印输出如下的二维阵列。

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

【解题思路】凡是涉及二维阵列的场合都可以使用二维数组来处理。在本例中，可以设置一个 4 行 4 列的二维数组 a 来保存结果，它的声明如下：

```
int a[4][4];
```

其中每一维的长度为 4。

在为数组赋值之前，首先来观察阵列的变化规律。假设用变量 row、col 分别表示行号和列号。

（1）首先看第一行，此时 row=0。在这一行中，第 col 列的值是 col+1；

（2）再看第 2 行，此时 row=1。此行中，第 0 列的值是 5，刚好是第一行第 0 列的值加上长度 4，也就是等于 4+1；第 1 列的值是 6，等于 4+2，以此类推。那么第 row 行第 col 列的值是 row*4+col+1；

（3）经过推算，第 3、4 行的元素也具有如 2 中描述的规律。

这样，阵列中第 row 行第 col 列的值具有如下规律：

```
a[row][col] = row * 4 + col + 1
```

根据上述规律，可以编写如下代码。

```
//7-9.c
#include <stdio.h>

#define LEN 4

int main()
{
     int a[LEN][LEN];
     int row, col;

     for (row = 0; row < LEN; ++row)
          for (col = 0; col < LEN; ++col)
               a[row][col] = row * LEN + col + 1;

     for (row = 0; row < LEN; ++row)
     {
          for (col = 0; col < LEN; ++col) printf("%4d", a[row][col]);
          putchar('\n');
     }

     return 0;
}
```

程序的运行结果如下：

```
   1   2   3   4
   5   6   7   8
   9  10  11  12
  13  14  15  16
```

> 在打印二维数组时，使用域宽控制打印元素的间距和在打印一行后换行是常用的技巧。这样可以使打印的阵列美观。

7.3.2 二维数组的初始化

在前面的例子中使用计算或输入的方式为二维数组的每一个元素赋初值，这显然很麻烦。其实二维数组也可以像一维数组那样，在声明的时候进行初始化。

二维数组的初始化一般是按行分段进行的。例如有声明：

```
int a[5][3];
```

那么对其进行初始化的语法如下：

```
int a[5][3]={ {80,75,92},
              {61,65,71},
              {59,63,70},
              {85,87,90},
              {76,77,85}
              };
```

上述初始化还可以简写成：

```
int a[5][3]={ 80,75,92,61,65,71,59,63,70,85,87,90,76,77,85 };
```

编译器会根据指定的行数和列数来自动切分数据。

对于二维数组初始化还有以下说明：

（1）可以只初始化部分元素，未初始化的元素自动赋 0 值。例如

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

是对每一行的第一列元素赋初值，未初始化的元素赋 0 值。初始化后各元素的值为：

```
1  0  0
2  0  0
3  0  0
```

再如

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

初始化后各元素的值为：

```
0  1  0
0  0  2
3  0  0
```

（2）初始化时，行的长度可以不给出，但列的长度必须给出。例如：

```
int a[3][3]={1,2,3,4,5,6,7,8,9};
```

可以写为：

```
int a[][3]={1,2,3,4,5,6,7,8,9};
```

一般情况下，如果二维数组的列数是 col，而初始化值的个数是 m，那么该二维数组的行数就是(m+col-1)/col。上例中，a 的行数为(9+3-1)/3=3。

又例如有声明：

```
int b[][3]={1,2,3,4,5,6,7};
```

这里 col=3，m=7，那么数组 b 的行数就是(7+3-1)/3=3，其初始化情况如下：

```
1   2   3
4   5   6
7   0   0
```

【例 7-10】打印出杨辉三角。杨辉三角又称为 Pascal 三角，是形如下表的数表。

```
        1
        1   1
        1   2   1
        1   3   3   1
```

```
1   4   6   4   1
1   5   10  10  5   1
```

【解题思路】从数学的角度来讲，杨辉三角中第 n 行其实是二项式 $(x+y)^n$ 展开后各项的系数。设 $a_{i,j}$ 代表三角中第 i 行第 j 列的元素。可以观察到，除了第一列外，有

$a_{i,j}=a_{i-1,j}+a_{i-1,j-1}$

成立。据此，可以使用一个二维数组来保存计算结果。在初始化时，可以利用 C 语言提供的部分初始化功能，只将左上角的元素置为 1，而其他的元素都自动置为 0。三角中首列的元素比较特殊，因此可以用直接赋值的方式将它们置为 1。采用上面的计算公式，在输出时，第 i 行输出 i+1 个元素。

```
//7-10.c
#include <stdio.h>

#define ROWNUM 10

int main()
{
    int Yhtri[ROWNUM][ROWNUM] = { {1}, {0} };
    int row, col;

    for (row = 1; row < ROWNUM; ++row)
    {
        Yhtri[row][0] = 1;
        for (col = 1; col <= row; ++col)
            Yhtri[row][col] = Yhtri[row-1][col] + Yhtri[row-1][col-1];
    }

    for (row = 0; row < ROWNUM; ++row)
    {
        for (col = 0; col <= row; ++col)
            printf("%4d", Yhtri[row][col]);
        putchar('\n');
    }

    return 0;
}
```

7.3.5 二维数组和一维数组的关系

在 C 语言中，二维数组的声明没有采用以下方式：

```
int a[3,4];
```

而是这样的：

```
int a[3][4];
```

从这个声明中可以解析出哪些信息呢?

在解析一个比较复杂的变量声明时，可以采用如下步骤。

（1）首先解析出变量名，此例中是 a；

（2）再看看 a 右面有没有其他修饰符。如果没有，那么 a 就是一个简单变量；如果有，那么这些修饰符一般是[]和()之一。此例中是[]，那么 a 就先和[3]结合。这充分说明 a 是一个一维数组，其长度为 3。

（3）此后去掉 a[3]，剩下的部分就是变量 a 的基类型。此例中剩下的部分是 int [4]。很明显，这是一个数组类型。这说明了数组 a 的每一个元素又是一个一维数组，该一维数组的长度为 4，其基类型为 int。

从上面的解析可以看到，C 的二维数组其实是数组的数组。基于此，对于数组 a 的使用有 3 种情况。

（1）a，这代表了整个二维数组，是个常量，其类型是 int[3][4];

（2）a[i]，代表了二维数组 a 的第 i 行，是一个一维数组的名字，也是个常量，其基类型是 int[4];

（3）a[i][j]，代表了二维数组 a 第 i 行第 j 列的元素，或者是一维数组 a[i]第 j 列的元素，是一个变量，其基类型是 int。

图 7-5 示意了上述情况。

需要特别说明的是，在实际使用中，并不深究二维数组到底是怎么构成的，而是把它整体视为一张二维表格。

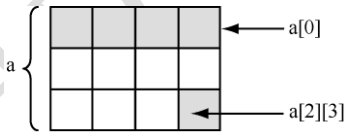

图 7-5　二维数组和一维数组的关系

7.4　字符数组

7.4.1　字符数组、字符串及其初始化

字符数组是用来存放字符数据的数组，它的每一个元素都是一个字符。例如：

```
char hello[5] = { 'H', 'e', 'l', 'l', 'o' };
```

其使用与一般的一维数组没有区别。但在编写程序时，字符数组常被当作字符串对待。读者已经知道，字符串有一个结尾标志，就是一个'\0'。因此，如果要把上述的 hello 当作字符串的话，那么必须在其结尾补上'\0'：

```
char hello[6] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

读者可能已经注意到了数组长度的变化。

上述的初始化方式有些繁琐，因此常使用下面的方法来声明和初始化字符数组：

```
char hello[] = "Hello";
```

在这种情况下，数组 hello 的长度为 6，其中计算了结尾符'\0'，但其表示的字符串长度却是 5。

如下的初始化因为指定了长度而没有包含'\0'：

```
char hello[5] = "Hello";
```

此时，hello 不能用作字符串。

【例 7-13】一维字符数组的使用。

```
//7-14.c
#include <stdio.h>

int main()
{
    char hello1[5] = { 'H','e', 'l', 'l', 'o' };
    char hello2[] = { 'H','e', 'l', 'l', 'o', '\0' };
    char hello3[5] = "Hello";
    char hello4[] = "Hello";
```

```
    printf("Hello1=%s\n", hello1);
    printf("Hello2=%s\n", hello2);
    printf("Hello3=%s\n", hello3);
    printf("Hello4=%s\n", hello4);

    return 0;
}
```

程序的运行结果如下：

```
Hello1=Hello 烫藟
Hello2=Hello
Hello3=Hello 烫藁 Hello
Hello4=Hello
```

从结果中可以看到，hello1、hello3 的输出结果中包含了一些不属于它们的奇怪字符。出现这种错误现象的原因是这两个字符串没有包含结尾标志‘\0’，所以 printf()函数不知道在正确的位置结束输出，而是继续顺着存储方向往后读取字符，直到碰巧遇到一个‘\0’为止。这证明了这两个字符数组不能用作字符串使用。

除了一维数组，还可以声明二维字符数组。例如：

```
char text[20][30];
```

其使用方式与整型、浮点型二维数组没有区别。

在实际应用中，二维字符数组常被视为字符串的数组。例如：

```
char trafficLight[][6] = { “red”, “green”, “amber” };
```

【例 7-14】字符串数组的使用。

```
//7-15.c
#include <stdio.h>

int main()
{
    char trafficLight[][6] = { "red", "green", "amber" };
    int i;

    for (i = 0; i < 3; ++i)
        printf("%s\n", trafficLight[i]);

    return 0;
}
```

程序的运行结果是：

```
red
green
amber
```

字符数组还常用作输入字符串时的缓冲区。缓冲区（*buffer*）是一片连续的存储区，常用于暂存数据。当然，这个缓冲区要求足够大，能够容纳输入的字符，同时还要容纳结尾标志‘\0’。

7.4.2 字符串处理函数

C 语言提供了丰富的字符串处理函数，大致可分为字符串的输入、输出、合并、修改、比较、转换、复制、搜索几类。使用这些函数可大大减轻编程的负担。用于输入、输出的字符串函数，在使用前应包含头文件<stdio.h>；使用其他字符串函数则应包含头文件<string.h>。下面介绍

几个常用的函数。

1．字符串输出函数

函数原型：int puts(字符数组名)

功能：把字符数组中的字符串输出到显示器上。如果函数正确执行，则返回一个非负值；否则返回 EOF 表示失败。

2．字符串输入函数

函数原型：char * gets(字符数组名)

功能：从标准输入设备键盘上输入一个字符串。此函数的返回值是该字符数组的首地址。如果失败，则返回空指针 NULL。关于地址的概念我们将在“指针”(8.2)章节中详细讲解。

3．字符串连接函数

函数原型：char * strcat(字符数组 1，字符数组 2）

功能：连接两个字符数组中的字符串，将字符串 2 接到字符串 1 的后面，结果存放在字符数组 1 中，函数的返回值是字符数组 1 的地址。

使用这个函数要注意以下几点。

（1）字符数组 1 必须足够大，以便容纳连接后的新字符串。

（2）连接前，两个字符串的结尾都有一个‘\0’，连接时将字符串 1 后面的‘\0’取消，只在新串的最后保留一个‘\0’。

4．字符串复制函数

函数原型：char * strcpy(字符数组 1，字符数组 2)

功能：把字符数组 2 中的字符串复制到字符数组 1 中。串结束标志‘\0’也一同复制。函数的返回值是字符数组 1 的首地址。

5．字符串比较函数

函数原型：int strcmp(字符数组 1，字符数组 2)

功能：按照 ASCII 码顺序比较两个数组中的字符串，并由函数返回值返回比较结果：如果字符串 1 与字符串 2 相同，则返回值等于 0；否则为非 0。

C 语言不支持用条件运算符直接比较两个字符串，即“abc”==“ABC”在逻辑上是错误的，但在语法上是正确的，因为“abc”或“ABC”表示的是这两个字符串的首地址，永远不可能相等，这种比较没有意义。

6．求字符串长度

函数原型：int strlen(字符数组）

功能：测字符串的实际长度(不含字符串结束标志‘\0’)，并将其作为函数返回值返回。

【例 7-16】字符串函数的使用。

```
//7-17.c
```

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

int main()
{
     char str1[128];
     char str2[128];

     printf("Please type in two strings:\n");
     gets(str1);
     gets(str2);

     printf("The length of string 1 is %d.\n", strlen(str1));
     printf("The length of string 2 is %d.\n", strlen(str2));

     printf("Comparing...\n");
     if (strcmp(str1, str2) == 0)
          printf("identical!\n");
     else
          printf("different!\n");

     printf("Concating...\n");
     strcat(str1, str2);
     puts(str1);

     printf("Copying...\n");
     strcpy(str2, str1);
     puts(str2);

     return 0;
}
```

程序的运行结果是：

```
Please type in two strings:
I like milk↙
But I like coffee much more↙
The length of string 1 is 12.
The length of string 2 is 28.
Comparing...
different!
Concating...
I like milk But I like coffee much more.
Copying...
I like milk But I like coffee much more.
```

ⓘ 请读者注意 strcmp()函数的返回值：只有返回值等于 0 的情况下才说明两个字符串相同。

7.5 高维数组

三维以上的数组常称为“高维数组”，例如：

```
int a[4][5][6];
```

可以这样形象地理解这个三维数组：数组 a 有 4 层，每层是一个二维阵列，这个阵列有 5 行 6 列；数组 a 共有 4×5×6=120 个整型元素。

高维数组主要用于表示复杂的数学模型，是一个非常不容易理解和掌握的概念。这里举一个例子，用高维数组来表示一个书库，通过逐步递进的方式介绍高维数组的构成。

（1）如果用一个字符变量来表示书中的一个文字，那么可以声明成 char book;

（2）如果表示一行文字（假设一行最多 80 个文字），那么可以用：char book**[80]**;

（3）如果要表示一页（假设一页最多 50 行），那么可以用：char book**[50]**[80];

（4）如果表示 300 页的一本书，那么可以用：char book**[300]**[50][80];

（5）如果这本书放在可以放 20 本的书架的一层上，那么可以用

```
char book[20][300][50][80];
```

（6）如果这个书架有 5 层，那么可以用

```
char book[5][20][300][50][80];
```

（7）如果这个书架放在能容纳 50 个书架的房间中，那么可以用

```
char book[50][5][20][300][50][80];
```

（8）如果这个书库有 10 个这样的房间，那么可以用

```
char book[10][50][5][20][300][50][80];
```

可以看出，高维数组主要表示了一种超立体结构，而这种结构在日常生活中较少遇到。

正是由于高维数组的复杂结构，因此建议读者谨慎使用高维数组，本章也不再过多讲解。