第5章 数　组

主要知识点

- ◆ 数组概述
- ◆ 一维数组
- ◆ 二维数组
- ◆ 字符数组
- ◆ 字符串处理
- ◆ 数组与指针

前面章节中介绍了很多基本数据类型，如整型、浮点型及字符型等。使用这些类型的变量来处理数据，可以解决一些简单的问题。但在实际应用中，需要处理的数据往往是批量数据（如100个学生的成绩），这些批量数据不仅数据量较大，而且数据之间还可能存在着某种关系（如两门课程的成绩属于同一个学生），此时，使用数组则显得更加方便且有效。

本章主要介绍在C语言中怎样使用数组来处理批量数据。

5.1 数组概述

与整型等简单数据类型相似，数组也可以理解为一种用来定义变量的数据类型，只是用数组定义的变量，更便于存放批量数据。

【实例5-1】 输入5个学生的课程成绩，计算并输出最高分。

通过前面的学习，可以提出以下思路。

思路一：定义变量x和max，先输入第一个成绩给x，并将其赋值给max，即认为只有一个数时最大值就是这个数。再构造一个执行循环体4次的循环结构，依次输入并判断第2～5个数（后4个数）。具体在循环体中完成以下操作：输入一个成绩给x，并执行语句“if(max＜x) max＝x;”，同时更新max的值。最后在循环结束后输出max的值。对应的程序代码如下：

```
/*实例5-1*/
#include<stdio.h>
int main()
{
    int x, max, i;
    scanf("%d", &x);
    max = x;
    for(i = 1; i<5; i++)
```

```
    {
        scanf("%d", &x);
        if(max < x)
            max = x;
    }
    printf("The max score is %d\n", max);
    return 0;
}
```

在前面的章节中,思路一描述了类似问题的常见解决方式。但若问题稍加复杂化,在前面的基础上如何实现继续输出所有低于该最高分的学生成绩?通过分析可以发现,完成这个任务,首先需要将所有学生的成绩保存,而思路一中的变量 x 只能保存一个学生的成绩,那么有没有更好的解决方法呢?

C 语言程序设计中,在处理类似数据量较大的问题时,使用数组是一种常见的方法。那么什么是数组呢?

(1) 数组可以理解为由相同类型的数据组成的数据集合。数组中的每一个数据称为数组的元素,不同的数组有不同的名称,也有不同的元素个数。如 5 个学生的成绩就是一个数组,在 C 语言中可以用语句"int a[5];"来定义,其中 int 表示数组中元素的数据类型为整型,a 表示数组的名称,5 表示数组 a 中的元素个数。

(2) 数组中的元素是有顺序的。为了方便,在定义数组后,每一个元素都分配到一个顺序号(从 0 开始计数),称为这个元素在该数组中的下标。如语句"int a[5];"定义了一个含有 5 个元素的数组 a,那到底是哪 5 个呢?它们依次为 a[0],a[1],a[2],a[3]和 a[4],可以看出第 1 个元素的下标为 0。

有了数组的概念后,再回到实例 5-1,通过分析可得以下思路。

思路二:定义一个包含 5 个元素的数组 a 和简单变量 max。依次输入第 1 个成绩给 a[0],输入第 2 个成绩给 a[1]……一直到输入第 5 个成绩给 a[4]。先将第一个成绩 a[0]赋值给 max,即认为只有一个成绩 a[0]时,最大值就是 a[0]。再构造一个执行循环体 4 次的循环结构,使用语句"if(max < a[i]) max=a[i];"依次判断第 2～5 个数(后 4 个元素值,i 为元素下标,从 1 取到 4,即表示第 2～5 个元素),同时更新 max 的值。对应的程序代码如下:

```
/* 实例 5-1 */
#include <stdio.h>
int main()
{
    int i, a[5];                    /* 定义了 5 个元素的数组 a 表示 5 个学生成绩 */
    int max;                        /* max 用来表示最高成绩 */
    for(i = 0; i < 5; i++)
        scanf("%d", &a[i]);
    max = a[0];
    for(i = 1; i < 5; i++)
    {
        if(max < a[i])
            max = a[i];
    }
    printf("The max score is %d\n", max);
```

```
    return 0;
}
```

在上述程序的第一个循环中，i 的值从 0 取到 4 时，a[i]也从 a[0]变到 a[4]。若还想在此基础上继续输出所有低于最高成绩的学生成绩，只需在原程序的 printf 语句后添加以下程序段。

```
for(i = 0; i < 5; i++)
    if(a[i] < max)
        printf("%d\n", a[i]);
```

可见，将数组与循环相结合，可以方便、有效地处理批量数据，大大提高了工作效率。

5.2 一维数组

一维数组是最简单的数组，它的数组元素只需要用数组名加一个下标就能确定，如前面介绍的学生成绩数组就是一个一维数组。

5.2.1 一维数组的定义

和其他简单变量相同，要使用数组，就必须先在程序中定义它。因为数组是一个有顺序的同种类型数据的数据集合，那么在定义数组时就需要已知或指定该数组的名称、数组中元素的个数以及元素的类型。如下面的数组定义：

```
float b[10];
```

表示定义了一个单精度浮点型数组，数组名称为 b，数组中有 10 个浮点型元素。

定义一维数组的一般形式为：

```
类型说明符  数组名[常量表达式];
```

说明：

(1) 类型说明符统一说明数组中各个元素的类型。

(2) 数组名的命名规则和一般变量名的命名规则相同。

(3) 数组名后必须是一对方括号[]，不能是其他括号，而且方括号中为常量表达式，表示数组的元素个数，即数组的长度，不能包含变量，例如，可以这样定义：

```
#define N 5
int a[N];
```

但下面的数组定义是错误的：

```
int n = 5;
int a[n];
```

因为在第一种定义方式中使用的 N 是一个常量(符号常量)，而第二种定义方式中使用的 n 则是一个整型的变量。在 Visual C++ 6.0 环境中编译时，也会出现相应的错误提示信息。

(4) 通过语句“int a[5];”定义的一维数组a包含5个数组元素,依次为a[0],a[1],a[2],a[3]和a[4],注意下标是从0开始的,不存在a[5]这样的数组元素,而且这5个元素在内存中是按下标顺序依次对应分配内存空间的(见图5-1)。

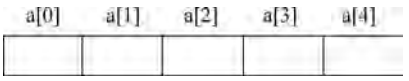

图5-1 一维数组的存储形式

5.2.2 一维数组的引用

在C语言中,变量被定义之后就可以被使用。数组也一样,在被定义之后,引用数组中的元素成为我们较关注的问题。注意:在C语言中,不能一次调用数组的所有元素,只能逐个引用数组的元素。引用一维数组元素的一般形式为:

```
数组名[下标值]
```

虽然数组元素的表示形式(必须标示下标,并置于“[]”中)与普通变量的表示形式(只有变量名)不同,但其使用方法与普通变量完全相同,对普通变量进行的输入、输出、引用和赋值等操作,对数组元素同样适用。如下列程序段中的语句均为合法语句:

```
int a, b, c[5];
scanf("%d", &a);
scanf("%d", &c[0]);
b = a + c[0];
printf("%d + %d = %d\n", a, c[0], b);
```

说明:

(1) 数组元素的下标必须为整型变量或整型表达式,否则编译环境会提示出错。

(2) 数组元素的下标取值范围为0～N－1(假设N为数组长度),编译环境一般不检测数组下标是否超出此范围。

(3) 定义数组时的“数组名[常量表达式]”和引用数组元素时的“数组名[下标]”形式相同,但含义不同,例如:

```
int a[5];        /*a[5]与类型 int 相结合,表示定义一个包含5个元素的整型数组a*/
a[2]=8;          /*a[2]表示前面已经定义的数组a中序号为2(第3个)的元素*/
```

【实例5-2】 已知包含6个元素的数组a,将其各元素依次赋值为0,1,2,3,4,5,并逆序输出各元素的值。

分析: 先给6个元素赋值,过程为a[0]=0,a[1]=1,…,a[5]=5,可以发现赋值过程是一个a[i]=i(i为0～5)的循环。逆序输出过程为:先输出a[5],再输出a[4],接着输出a[3]……一直到输出a[0],可以发现逆序输出过程也是一个输出a[i](i从5～0)的循环。

程序代码如下:

```
/*实例5-2*/
#include<stdio.h>
int main()
{
    int i, a[6];
```

```
    for(i = 0; i < 6; i++)
        a[i] = i;
    for(i = 5; i >= 0; i -- )
        printf("a[ %d] =  %d\n", i, a[i]);
    return 0;
}
```

说明：

(1) for 语句中用 i＋＋和 i－－分别来实现循环变量的递增和递减运算，进而控制数组元素的顺序和逆序引用。

(2) 语句“printf("a[%d] = %d\n", i, a[i]);”则使用双引号中的格式字符串，控制输出的格式，方便结合数组的引用格式进行程序的理解。

程序运行结果如图 5-2 所示。

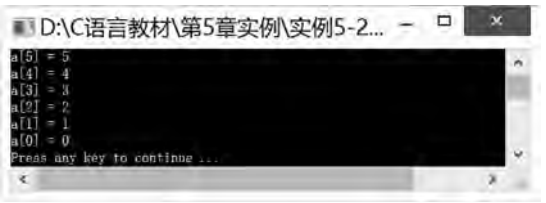

图 5-2　数组逆序输出结果

可见，在对一维数组进行操作时，操作的过程往往被抽象为关于数组元素下标规律性变化的子过程，使得一维数组常与循环结构相结合，达到有效处理数据的目的。

5.2.3　一维数组的初始化

和普通变量一样，数组也可以在定义的同时给其包含的数组元素赋值，即数组的初始化。一维数组初始化的一般形式为：

```
类型说明符　数组名[常量表达式] = {初始化列表};
```

根据需要，一维数组的初始化方法主要有以下几种。

(1) 在定义数组时对数组的全部元素赋初值。例如：

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

将数组中每个元素的初值按顺序放在一对花括号内，数据间用逗号分隔。经过上面的初始化语句后，数组 a 中各元素的值如图 5-3 所示。

元素	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
值	0	1	2	3	4	5	6	7	8	9

图 5-3　数组 a 中各元素的值

(2) 在定义数组时只对数组的部分元素赋初值。例如：

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

定义的数组a中包含10个元素,但花括号内只提供5个初值,这表示只给数组a的前5个元素赋初值,而后5个元素系统自动赋初值为0。经过上面的初始化语句后,数组a中各元素的值如图5-4所示。

元素	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
值	0	1	2	3	4	0	0	0	0	0

图5-4　数组a中各元素的值

(3) 如果想将一个数组中的元素全部初始化为0,可以写成:

```
int a[10] = {0,0,0,0,0,0,0,0,0,0};
```

或

```
int a[10] = {0};     /* 大括号中的0赋给第1个元素,其余元素均取系统默认值0 */
```

(4) 在对全部数组元素赋初值时,由于初始化列表中的数据个数已经确定,中括号中的数组长度可以省略。例如:

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

也可以写成:

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

在第2种写法中,系统会自动根据花括号中的数据个数得出当前数组的数组长度。

5.2.4　一维数组的程序举例

【实例5-3】 输入10个整数到数组a,计算并输出下标为偶数的元素之和。

分析:先依次输入10个整数赋值给数组a中的元素a[0],a[1],…,a[9],并将下标为偶数的元素,即a[0],a[2],a[4],a[6],a[8]求和,为了方便可使用循环结构实现。

程序代码如下:

```
/* 实例5-3 */
#include <stdio.h>
int main()
{
    int i, a[10], sum = 0;
    printf("Please input 10 numbers:\n");
    for(i = 0; i < 10; i++)
    scanf("%d", &a[i]);
    for(i = 0; i < 10; i += 2)
    sum += a[i];
    printf("sum = %d\n", sum);
    return 0;
}
```

说明:从语句"scanf("%d", &a[i]);"可知,引用的一维数组元素和常规变量的使用方法类似。

程序运行结果如图 5-5 所示，其中第 8 行和第 9 行又等同于如下的程序段：

```
for(i = 0; i < 5; i ++)
    sum += a[2 * i];
```

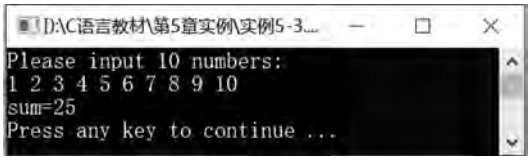

图 5-5　数组 a 下标为偶数的元素之和

试想本例改为求数组 a 下标为奇数的元素之和，将如何实现？

【实例 5-4】 利用数组求 Fibonacci 数列的前 15 项，并按每行 5 个数的格式输出。

分析：Fibonacci 数列满足以下数学公式：

$$F_n = \begin{cases} 1, & n = 1,2 \\ F_{n-1} + F_{n-2}, & n > 2 \end{cases}$$

在用数组(设为 Fib)存放该数列的前 15 项时，可根据公式建立对应关系：Fib[0]＝1，Fib[1]＝1，Fib[2]＝Fib[1]＋Fib[0]，Fib[3]＝Fib[2]＋Fib[1]，…，Fib[14]＝Fib[13]＋Fib[12]，即在 Fib[0]＝1，Fib[1]＝1 的情况下，其他元素的值可根据 Fib[i]＝Fib[i－1]＋Fib[i－2](i 为 2～14)求取。另在输出时要实现每行 5 个数，只需在输出下标为 4(第 5 个)、9(第 10 个)、14(第 15 个)的元素后输出换行符'\n'即可。

程序代码如下：

```
/ * 实例 5-4 * /
# include < stdio.h >
int main()
{
    long Fib[15] = {1,1};                    / * 前面两个元素的值为 1 * /
    int i;
    for(i = 2; i < 15; i++)
        Fib[i] = Fib[i - 1] + Fib[i - 2];    / * 求取 Fib[2]～Fib[14]的值 * /
    for(i = 0; i < 15; i++)
    {
        printf(" % 6ld", Fib[i]);
        if((i + 1) % 5 == 0)                 / * 控制输出换行的位置 * /
            printf("\n");
    }
    return 0;
}
```

说明：

(1) 使用数组对 Fibonacci 数列进行存储，并将数组名命名为 Fib 以增加程序的可读性。

(2) 程序中 Fib[i－1]和 Fib[i－2]表示数列中相对当前位置前两项的值，其中 i－1 和 i－2 分别表示对应数组元素的下标，因此，可从直观上判断 i 从大于或等于 2 的位置开始，

这也是 for(i=2; i<15; i++)中 i=2 的部分原因。

程序运行结果如图 5-6 所示。

图 5-6 Fibonacci 数列的前 15 项

5.3 二维数组

一维数组中的元素只需通过一个下标即可唯一确定,但在实际应用中,有很多数据需要通过两个或多个下标才能唯一确定。如表 5-1 中要表示"3 班第 2 个学生的成绩"就需要两个下标(班级编号和学号)唯一确定,这种用两个下标唯一确定一个元素的数组称为二维数组。可以发现,二维数组表示的就是一个矩阵,数组中的每个元素都需要通过在矩阵中所处的行号和列号来确定。同理还可以有三维甚至多维数组,它们的概念和用法基本类似。本节只介绍二维数组,在熟练掌握一维和二维数组后,多维数组的使用很容易通过类推而得到。

表 5-1 学生成绩表

班级编号	学号			
	1	2	3	4
1	79	87	85	94
2	88	80	91	78
3	83	82	93	86
4	98	70	84	78
5	95	94	75	85

5.3.1 二维数组的定义

由于二维数组表示一个矩阵,因此在定义二维数组时,除了要指定二维数组的名称,还需要指定其行数和列数。定义二维数组的一般形式为:

```
类型说明符  数组名[常量表达式 1][常量表达式 2];
```

例如:

```
float b[3][4];
```

定义了一个 float 类型的二维数组,数组名称为 b,此数组第 1 维有 3 个元素(3 行),第 2 维有 4 个元素(4 列),且每一维的长度分别置于一对方括号中,该数组中共有 3×4 个元素,即

```
b[0][0], b[0][1], b[0][2], b[0][3]
b[1][0], b[1][1], b[1][2], b[1][3]
```

```
b[2][0], b[2][1], b[2][2], b[2][3]
```

可见，二维数组定义语句中“常量表达式 1”的值表示该矩阵的行数，“常量表达式 2”的值则表示该矩阵的列数。位于同一行的数组元素具有相同的行下标，而位于同一列的数组元素具有相同的列下标，因此，二维数组又可以理解为一个特殊的一维数组，如上述数组 b 就可以看作一个包含三个元素(b[0]，b[1]和 b[2])的一维数组，只不过这三个元素(b[0]，b[1]和 b[2])中的每个元素又是一个包含四个元素的一维数组，如下所示：

```
b[0]——b[0][0], b[0][1], b[0][2], b[0][3]
b[1]——b[1][0], b[1][1], b[1][2], b[1][3]
b[2]——b[2][0], b[2][1], b[2][2], b[2][3]
```

变量被定义后将被存放在内存中，C 语言中二维数组的元素在内存中是按行存放的，即在内存中先顺序存放第 1 行的元素，再顺序存放第 2 行的元素……直到最后一行元素存放完为止。如图 5-7 所示为前面定义的二维数组 b 在内存中的存储情况。

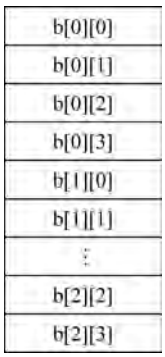

图 5-7　二维数组的存储形式

5.3.2　二维数组的引用

引用二维数组元素的一般形式为：

```
数组名[行下标值][列下标值]
```

例如，b[1][3]表示数组 b 中序号为 1 的行(即第 2 行)中序号为 3 的列(即第 4 列)的元素。下标应该是整型表达式，与一维数组一样，可以对二维数组元素进行输入、输出、引用和赋值等操作。例如：

```
int m = 0, n = 1, c[2][3];
scanf("%d%d", &c[m][n], &c[n][m]);
c[0][0] = c[m][n] + c[n][m];
printf("The first element is %d\n", c[0][0]);
```

说明：在引用二维数组的元素时，行或列的下标值均应在已定义的数组大小范围内。例如，语句“int a[3][3];”定义了一个 3×3 的二维数组 a，它的行或列下标的取值范围均为 0～2，所以数组 a 中不存在 a[3][3]这样的元素。

【实例 5-5】 从键盘输入一个 3×4 的整型矩阵给二维数组 a，并输出该矩阵的所有元素。

分析：可采用先行后列的顺序进行输入(或输出)，并且通过一维数组的操作可知，在正确定义变量 j 的情况下，输入第 1 行元素的程序段可描述如下：

```
for(j = 0; j < 4; j++)
    scanf("%d", &a[0][j]);
```

输入第 2 行元素的程序段可描述如下：

```
for(j = 0; j < 4; j++)
```

```
    scanf("%d", &a[1][j]);
```

通过观察可以发现,输入第i行元素的程序段可描述为:


```
for(j=0; j<4; j++)
    scanf("%d", &a[i][j]);
```

而此程序中i的取值范围为0～2,整理后的程序代码如下:

```
/*实例5-5*/
#include<stdio.h>
int main()
{
    int a[3][4], i, j;
    printf("Please input the matrix:\n");
    for(i=0; i<3; i++)
        for(j=0; j<4; j++)
            scanf("%d", &a[i][j]);
    printf("The matrix data are:\n");
    for(i=0; i<3; i++)
    {
        for(j=0; j<4; j++)
            printf("%5d", a[i][j]);
        printf("\n");
    }
    return 0;
}
```

说明:从语句"scanf("%d", &a[i][j]);"可知,二维数组元素的引用与一维数组元素的类似,二维数组主要多了一个表示维度的下标。

程序运行结果如图5-8所示。试想本实例中若采用先列后行的顺序进行输入(或输出),将如何求解?程序中语句"printf("\n");"的作用是什么,去掉后对输出结果有何影响?

图5-8 二维矩阵的输入和输出

可见,与一维数据类似,对二维数组的操作也常会与循环结构相结合,只是二维数组用两个下标来确定元素,因此,对二维数组的操作一般会结合二重循环结构。二重循环语句中具体该如何嵌套取决于对二维数组操作时是先行后列还是先列后行。实例5-5的程序中对二维数组元素进行的输入和输出均为先行后列的顺序进行。

5.3.3 二维数组的初始化

二维数组的初始化方法主要有以下几种。

(1) 通过嵌套一维数组初始化列表的方法顺序对二维数组的全部元素赋初值。例如：

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

这种方法将二维数组中每一行元素的初值置于花括号中，比较直观和方便地对程序代码进行检错，也被称为分行赋值方法。

(2) 因为二维数组元素在内存中是按行存储的，所以方法(1)中的初始化语句也可以写作：

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

与方法(1)相比，方法(2)更容易出现由于数据多而导致数据遗漏或错位等情况。

(3) 在定义数组时，只对二维数组的部分元素赋初值。例如：

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

这种赋值方法相当于：

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

系统自动将没有初值的元素赋予初值 0。又如：

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

在执行上述语句后，数组 b 的所有元素如下：

```
1  0  0  0
2  4  0  0
7  0  0  0
```

可以看出，这种初始化方法比较适合于非 0 元素少的数组，通过罗列少量的非 0 元素，减少数据输入的工作量。

(4) 给二维数组部分元素赋初值时，又有如下形式：

```
int c[3][4] = {1, 2, 4, 7 };
```

通过执行上述语句后，数组 c 的所有元素如下：

```
1  2  4  7
0  0  0  0
0  0  0  0
```

与方法(3)比较后可见，在对部分元素赋初值时，写或不写表示一行元素的花括号将代表着不同的意义。

在二维数组初始化时，也存在省略数组长度的情况，但只能省略第 1 维的长度(总行数)，第 2 维的长度(总列数)不能省略，具体情况如下。

(1) 对全部元素赋初值。例如：

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

或

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

均等价于

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

(2) 对部分元素赋初值,但在初始化列表中列出了全部行。例如:

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

等价于

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

5.3.4 二维数组的程序举例

【实例 5-6】 输入一个 4×4 的整型矩阵,计算并输出主对角线的元素之和。

分析:在二维数组中主对角线元素的特点是行下标和列下标相等,在 4×4 的矩阵(假设名称为 a)中这些元素为 a[0][0],a[1][1],a[2][2],a[3][3],观察后可知,要求这些元素的和(设为 sum),只需重复执行语句"sum += a[i][i];",其中的取值范围 i 为 0～3。

```
/* 实例 5-6 */
#include <stdio.h>
int main()
{
    int a[4][4], i, j, sum = 0;
    printf("Please input a 4 * 4 matrix:\n");
    for(i = 0; i < 4; i++)
        for(j = 0; j < 4; j++)
            scanf("%d", &a[i][j]);
    for(i = 0; i < 4; i++)
        sum += a[i][i];
    printf("sum = %d\n", sum);
    return 0;
}
```

说明:二维数组表示的方形矩阵中,主对角线元素的行、列下标是相等的,从元素引用 a[i][i]中即可表达出该相等关系。

程序运行结果如图 5-9 所示。

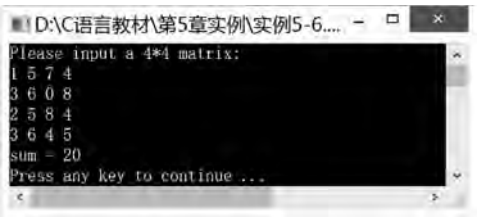

图 5-9 矩阵主对角线之和

由于二维数组可以表示矩阵,所以关于矩阵的运算也常通过二维数组来实现。本例若改为求次对角线上的元素之和,将如何实现?

【实例 5-7】 输入 3 个学生 5 门课(3×5)的课程成绩到二维数组 score 中,计算并输出

每个学生的平均成绩(保留两位小数)。

分析：已知一个 3×5 的二维数组 a 用来存放输入的 3 个学生 5 门课的课程成绩，且各个成绩的数组元素的对应关系如表 5-2 所示。

表 5-2　二维数组 a 的元素和 3 个学生 5 门课的课程成绩数据对应关系

学生编号	课程编号				
	0	1	2	3	4
0	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
1	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
2	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]

可以看出，第 1 个学生的平均成绩可通过(a[0][0]＋a[0][1]＋a[0][2]＋a[0][3]＋a[0][4])/5 计算，编程时常结合循环语句描述为 sum＋＝a[0][j]，j 从 0 取到 4，再用 sum/5 实现。而通过观察可知，计算第 2 个学生的平均成绩时只需将第 1 个学生计算过程中的 a[0][j]改成 a[1][j]即可，同理在计算第 3 个学生时改成 a[2][j]，通过整理整个计算过程可描述为 sum＝0，sum ＋＝ a[i][j]，j 从 0 取到 4，为内层循环控制变量，ave[i]＝sum/5，i 从 0 取到 2，为外层循环控制变量。

程序代码如下：

```
/*实例 5-7*/
#include<stdio.h>
int main()
{
    int i, j;
    float score[3][5], ave[3], sum=0;
    printf("Please input the scores:\n");
    for(i=0; i<3; i++)
        for(j=0; j<5; j++)
            scanf("%f", &score[i][j]);
    for(i=0; i<3; i++)
    {
        sum=0;
        for(j=0; j<5; j++)
            sum += score[i][j];
        ave[i] = sum/5;
    }
    printf("The averages of student are:\n");
    for(i=0; i<3; i++)
        printf("student %d: %.2f\n", i+1, ave[i]);
    return 0;
}
```

说明：程序中的 float ave[3]用于存放 3 个学生的平均成绩，可以进一步提高程序的可读性。

程序运行结果如图 5-10 所示。

在用二维数组解决问题时，按行顺序处理还是按列顺序处理是经常遇到的问题，在有些问题中按什么顺序都可以，但有些问题则不然，如实例 5-7 中就需要按行顺序处理，试想若按列顺序处理后计算结果代表什么意义。

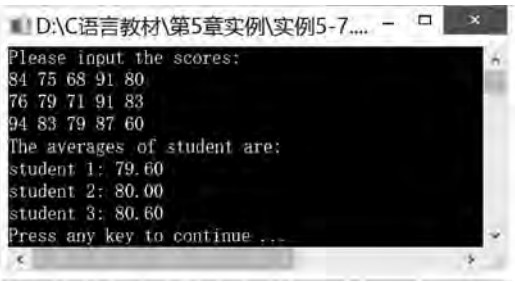

图 5-10　3个学生的平均成绩

5.4　字符数组

字符数组是元素类型为字符类型的数组。与数值型数据相比,字符型数据不仅应用广泛,更有其自身的特点,为了方便掌握,本节专门对字符数组加以讨论。

5.4.1　字符数组和字符串

通过前面的内容可知,字符类型(char)是C语言中的一种基本数据类型,通过该数据类型说明的字符变量可用来存放单个的字符数据。字符常量是用一对单引号括起来的单个字符,如'H'。

字符数组是用来存放由多个字符组成的有限字符序列(即字符类型的批量数据)。对字符数组的定义、初始化和引用等操作可通过与其他类型数组相同的方式进行。

字符串是由0个或多个字符组成的特殊的有限字符序列。其中有0个字符的字符串称为“空字符串”或“空串”,反之则称为“非空字符串”。字符串常量是用一对双引号括起来的多个字符,如"Hello"。

在C语言中没有字符串类型,字符串都是存放在字符数组中,以字符数组的形式表示的。但是与一般的字符序列不同,字符串的末尾一般都要有一个'\0'字符(即ASCII码值为0的字符,又称为空字符),用于表示字符串的结束,除此之外该字符不产生其他附加操作。因此,将字符串中第一个'\0'之前的有效字符个数称为该字符串的长度。其中字符串常量末尾的'\0'字符一般为隐藏的,由编译系统自动添加,而字符串变量(即用来存储字符串的字符数组)中的字符串结束符'\0'需要手工添加。例如:

(1) 字符串常量"Hello"一共包含6个字符,依次为'H','e','l','l','o','\0',其中最后的'\0'字符在字符串中是隐藏的,由系统自动添加,因此该字符串的长度为5。

(2) 字符串常量"He\0llo"中一共包含7个字符,依次为'H','e','\0','l','l','o','\0',末尾的'\0'字符同样是隐藏的,由系统自动添加,但该字符串的长度却为2,因为字符串在遇到字符序列中的第一个'\0'字符就已经结束。

(3) 在语句“char c[5]={'H','e','l','l','o'};”中,数组c中存放的字符序列不能作为字符串处理,因为该字符数组中存储的字符序列不存在字符串结束标志'\0',要表示字符串"Hello"可使用语句“char c[6]={'H','e','l','l','o' , '\0'};”。

总之,在C语言中,字符数组中既可以存储一般的字符序列又可以存储字符串,但在存

储字符串时还要在字符串的结束位置多存储一个字符串结束标志'\0',否则计算机将不知道字符串在什么地方结束。一个一维字符数组可用于存放一个字符串,多个字符串则可以存放于二维字符数组中,即每行存放一个字符串。

5.4.2 字符数组的定义和初始化

字符数组的定义形式与前面介绍的其他类型数组的定义相同。

1. 字符数组的定义

定义字符数组的一般形式为:

```
char 数组名[常量表达式];
char 数组名[常量表达式 1][常量表达式 2];
```

例如:

```
char a[5];                  /* 定义了一维字符数组 a,它有 5 个元素 */
char b[3][5];               /* 定义了 3×5 的二维字符数组,它有 3 行 5 列共 15 个元素 */
```

2. 字符数组的初始化

(1) 在定义字符数组时按顺序给字符数组的所有元素赋初值。例如:

```
char  c[12]={ 'H', 'o', 'w', ' ', 'a', 'r', 'e', ' ', 'y', 'o', 'u', '\0' };
```

此时又等价于

```
char  c[]={ 'H', 'o', 'w', ' ', 'a', 'r', 'e', ' ', 'y', 'o', 'u', '\0' };
```

其中,' '表示一个空格字符,即用单引号将一个空格字符括起来。执行上述语句后数组元素的取值情况如图 5-11 所示。

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]	c[8]	c[9]	c[10]	c[11]
H	o	w		a	r	e		y	o	u	\0

图 5-11 字符数组 c 的元素取值

(2) 在定义字符数组时按顺序只给字符数组的部分元素赋初值。例如:

```
char  a[8]={ 'T', 'h', 'a', 'n', 'k', 's' };
```

在此情况下,系统自动将不提供初值的元素赋为空字符'\0'。执行上述语句后数组元素的取值情况如图 5-12 所示。

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
T	h	a	n	k	s	\0	\0

图 5-12 字符数组 a 的元素取值

(3) 由于字符数组可以用来存放字符串,为了方便,可以直接使用字符串常量为字符数组进行初始化。

```
char  a[8]={ "Program" };
```

等价于

```
char  a[8] = "Program";
```

可见,在用字符串常量直接对字符数组初始化时,初值化列表的花括号可以省略。

说明:

① 在用字符串常量直接对字符数组初始化时,表示数组长度常量表达式可以省略,此时数组的长度为字符串的长度加1。例如:

```
char a[ ] = "Morning";
```

上面的语句中,字符数组a的长度为8,因为除了字符串中的7个字符外,系统会自动将字符串结束标志'\0'也存储在数组中,数组a的存储情况如图5-13所示。

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
M	o	r	n	i	n	g	\0

图5-13　数组c的实际存储情况

② 在用二维字符数组存储字符串时也可使用本方法进行初始化。例如:

```
char  c[3][10] = {"Apple", "Orange", "Mandarin"};
```

在执行上述初始化语句后,二维字符数组c的实际存储情况如图5-14所示。

A	p	p	l	e	\0	\0	\0	\0	\0
O	r	a	n	g	e	\0	\0	\0	\0
M	a	n	d	a	r	i	n	\0	\0

图5-14　二维字符数组c的实际存储情况

5.4.3　字符数组的输入和输出

字符数组的输入和输出方法不止一种,具体如下。

1. 逐个字符输入和输出

此方法和其他类型数组元素的输入输出方法类似,通过与循环结构相结合,对数组中的元素逐个进行输入和输出,只不过字符的格式字符串为"%c"。例如:

程序一(运行结果如图5-15所示):

```
#include <stdio.h>
int main()
{
    int i;
    char str[10];
    for(i = 0; i < 10; i++)
        scanf("%c", &str[i]);

    for(i = 0; i < 10; i++)
        printf("%c", str[i]);
    return 0;
}
```

程序二(运行结果如图5-16所示):

```
#include <stdio.h>
int main()
{
    int i;
    char str[10];
    for(i = 0; i < 9; i++)
        scanf("%c", &str[i]);
    str[i] = '\0';      /*i的值为9*/
    for(i = 0; i < 10; i++)
        printf("%c", str[i]);
    return 0;
}
```

图 5-15　程序一的运行结果

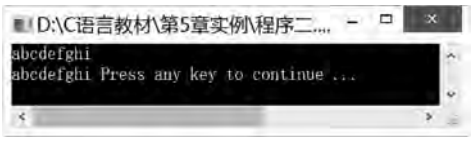

图 5-16　程序二的运行结果

程序一用于逐个输入 10 个字符到字符数组 str 中，然后逐个输出这 10 个字符。

在实际应用中，字符数组往往被用于存放字符串，而要使程序一的数组 str 也存放从外部输入的字符串，就需要添加字符串结束符'\0'，可将其修改为程序二的方式。

通过对比两个程序的代码及运行结果可见：

(1) 在字符数组中存放字符串时，需要在字符串尾部对应的位置提供一个元素(在此为 str[9])用于存放一个字符串结束符'\0'。

(2) 字符串结束符'\0'不能以"%c"格式从外部输入，只能进行手工赋值，如语句"str[i] = '\0';"。

(3) 字符'\0'在用"%c"格式输出时表现为一个空格。从输出结果的格式中可以看出，程序二在"Press any key to continue..."之前还输出了一个空格，而这个空格正是字符'\0'。

(4) 当字符数组用于存放字符串时，与循环结构相结合的输入和输出方法还不够灵活。程序二中语句"char str[10];"表明数组 str 中可以存放小于或等于 9 个有效字符的字符串，而根据当前程序二的输入方法，在程序运行时必须输入 9 个字符，少于 9 个字符时程序将一直运行在等待输入状态。为了更加方便地处理字符串，可用下面的方法进行字符数组的输入和输出。

2. 整个字符串输入和输出

此方法使用格式符"%s"，可一次性对数组元素进行输入和输出。例如以下程序：

```
/* 示例程序 */
#include <stdio.h>
int main()
{
    char str1[8], str2[8], str3[ ] = " Year!";
    printf("Please input two strings:\n");
    scanf("%s%s", str1, str2);                                    /* 输入两个字符串 */
    printf("str1 = %s, str2 = %s, str3 = %s\n", str1, str2, str3); /* 输出字符串 */
    return 0;
}
```

程序运行结果如图 5-17 所示。

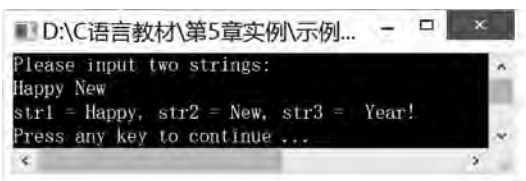

图 5-17　示例程序的运行结果

说明：

(1) 在用 scanf 和 printf 函数输入和输出字符串时，格式字符串"%s"对应的输出项为字符数组的名称，而非某个数组元素。如语句“scanf("%s%s", str1, str2);”不能写成语句“scanf("%s%s", str1[1], str2[1]);”。

(2) 在以"%s"格式输入字符串时，系统自动会在输入结束时添加字符'\0'到存储该字符串的字符数组中，因此，输入的字符串长度必须小于或等于该字符数组的长度减 1。例如执行下面的程序段时：

```
char str[8];
scanf("%s", str);
```

输入内容：Morning↙后，字符数组 str 的元素依次为'M','o','r','n','i','n','g','\0'。

(3) 在以"%s"格式输出字符串时，遇到第一个'\0'就结束输出，且不包括'\0'。例如下面的语句：

```
char str[ ] = "Good\0Morning!";
printf("%s", str);
```

执行后输出一个字符串"Good"，字符串结束符'\0'以后的字符不输出。

(4) 用 scanf 函数输入多个字符串时，系统会把空格、Enter 键或 Tab 键作为输入字符串之间的分隔符，因此，按"%s"格式输入的字符串中不能包含空格(具体见图 5-17)。

3. 使用字符串处理函数输入和输出

为了方便，在 C 函数库中提供了专门输入和输出字符串的函数：gets 和 puts，解决了用"%s"格式输入时不能输入空格等问题。和其他输入和输出函数一样，使用这两个函数时，应在程序的开始位置加上文件包含预处理命令：

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

(1) gets 函数。

gets 函数用于输入字符串，其使用的一般形式为：

```
gets(字符数组名称);
```

其作用是从键盘输入一个以 Enter 键结束的字符串(可以包括空格和 Tab 键)，在输入结束时，系统自动将结尾的 Enter 键转化为'\0'并存入字符串尾部。例如执行下面的程序段时：

```
char s[10];
gets(s);
```

输入内容：Mary↙后，数组 s 的前 5 个元素依次为'M','a','r','y','\0'，剩余的 5 个元素没有被赋初值。

(2) puts 函数。

与 gets 函数相对应，puts 函数用于输出字符串，其使用的一般形式为：

```
puts(字符串常量或字符数组名称)
```

其作用是将一个字符串(以'\0'结束的字符序列)输出到屏幕，输出时将字符串结束标志'\0'转换成'\n'，即输出字符串内容后再换行。例如执行语句“puts("Oh, My God!");”后的输

出结果如图 5-18 所示。

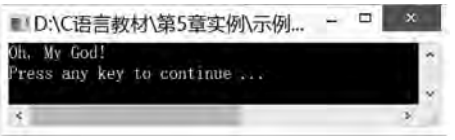

图 5-18　示例语句的输出结果

5.4.4　常用字符串处理函数

除了字符串的输入和输出，C 函数库中还提供了一些专门用来处理字符串的函数，如计算字符串的长度等。使用这些函数时，应在程序的开始位置加上文件包含预处理命令：

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

以下对常见的字符串处理函数进行介绍。

1. strlen 函数——测试字符串(string)长度(length)的函数

strlen 函数的一般格式如下：

```
strlen(字符串常量或字符数组名称)
```

函数返回值的数据类型是整型，返回值的大小表示被测试字符串的长度(不包括'\0')。若字符串为空串，则返回值为 0。例如：

```
char  str[10] = "Mary";
printf("%d", strlen(str));
```

程序段执行后的输出结果为：

```
4
```

说明：这里字符数组 str 的长度(即方括号中的数字)为 10，而字符串的长度为 4。

2. strcat 函数——字符串(string)连接(catenate)函数

strcat 函数的一般格式如下：

```
strcat(字符数组 1, 字符数组 2 或字符串常量)
```

strcat 函数的主要功能为将字符数组 2 或字符串常量连接到字符数组 1 的后面，形成一个新的字符数组，且保存在字符数组 1 中。函数的返回值为字符数组 1 的地址。例如：

```
char  str1[20] = "Micro";
char  str2[ ] = "soft";
printf("%s", strcat(str1,str2));
```

程序段执行后的输出结果为：

```
Microsoft
```

说明：使用 strcat 函数连接两个字符串时，字符数组 1 的存储空间必须足够大，足以存放连接后的新字符串。连接时，字符串 1 后面的'\0'取消，只在新字符串最后保留'\0'。

3. strcpy 函数——字符串(string)复制(copy)函数

strcpy 函数的一般格式如下：

```
strcpy(字符数组 1, 字符数组 2 或字符串常量)
```

strcpy 函数的主要功能为将字符数组 2 或字符串常量中的内容复制到字符数组 1 中，连同字符串结束标志'\0'也一起复制，字符数组 1 中原来的内容被覆盖。例如：

```
char  str1[] = "Chinese", str2[] = "English";
strcpy(str1, str2);
strcpy(str2, "French");
puts(str1);
puts(str2);
```

程序段执行后的输出结果为：

```
English
French
```

说明：不能用赋值语句将字符数组(或字符串)直接赋值给另一个字符数组。例如，下面的写法是错误的：

```
s1 = s2;
s2 = "Windows";
```

错误的原因是 s1、s2 都是字符数组名，而 C 程序中数组一经定义，数组名就代表了该数组的首地址，在程序执行过程中始终不变，它是一个常量。常量是不能出现在赋值表达式左边的。

4. strcmp 函数——字符串(string)比较(compare)函数

strcmp 函数的一般格式如下：

```
strcmp(字符串 1, 字符串 2)
```

strcmp 函数的主要功能为比较字符串 1 和字符串 2 的大小，并将比较结果返回。

C 语言中字符串的比较规则是对两个字符串中包含的有效字符从左向右依次逐个比较，直到出现不同的字符或遇到结束符'\0'为止。当出现不同的字符时，则返回这两个不同字符的 ASCII 码值之差，并认为这两个字符串不相等。反之，当两个字符串的全部字符都相同，则返回 0，并认为这两个字符串相等。返回值与比较结果的具体对应关系如下。

(1) 当字符串 1＝字符串 2 时，函数返回值为 0。

(2) 当字符串 1＞字符串 2 时，函数返回值为大于 0 的整数。

(3) 当字符串 1＜字符串 2 时，函数返回值为小于 0 的整数。

例如：

```
int r1, r2, r3;
char  s1[] = "Word", s2[] = "Excel";
r1 = strcmp(s1, s2);
r2 = strcmp("PowerPoint", s1);
r3 = strcmp(s1, "Word");
```

以上程序段运行后有 r1＞0，r2＜0，r3＝0。

说明：不能用关系运算符“>、>=、<、<=、＝＝”来比较两个字符串的大小。例如，以下描述是不合法的：

```
if(s1 > s2)
```

正确的写法为：

```
if(strcmp(s1, s2)> 0)
```

且判断两个字符串是否相等的表达式为：strcmp(s1,s2)＝＝0。

5. strupr 函数——字符串(string)转换为大写(uper)函数

strupr 函数的一般格式如下：

```
strupr(字符串)
```

strupr 函数的主要功能为将字符串中的所有小写字母转换成大写字母，其他字符保持不变。

6. strlwr 函数——字符串(string)转换为小写(lower)函数

strlwr 函数的一般格式如下：

```
strlwr(字符串)
```

strlwr 函数的主要功能为将字符串中的所有大写字母转换成小写字母，其他字符保持不变。

例如：

```
char  str1[6] = "CHinA";
printf("%s\n", strlwr(str1));
printf("%s\n", strupr(str1));
```

程序段执行后的输出结果为：

```
china
CHINA
```

5.4.5 字符数组的程序举例

【实例 5-8】 输入一个由字母组成的字符串(少于 20 个字符)，将其转换为纯大写字母后输出(不使用 strupr 函数)。

分析：定义一个一维数组用以存放外部输入的字符串，依次扫描并检测该一维数组中的每个元素，若检测到当前元素为小写字母，则将其转换为大写字母，一直扫描到字符串结束符'\0'为止，大写字母与小写字母的关系为：大写字母＝小写字母－32。

程序代码如下：

```
/* 实例 5-8 */
#include <stdio.h>
int main()
{
```

```
    char str[20];
    int i;
    printf("Input a string:\n");
    scanf("%s", str);
    for(i=0; str[i] != '\0'; i++)
        if(str[i] >= 'a' && str[i] <= 'z')
            str[i] -= 32;                    /*将小写字母转换为大写字母*/
    printf("str = %s\n", str);
    return 0;
}
```

说明：

(1) 语句“scanf("%s", str);”中的str前面不需要添加取地址符号&,因为str自身即表示地址,具体参考数组与指针部分的内容。

(2) 程序中用“str[i]!='\0'”作为循环条件来判断当前字符是否为'\0',由于字符'\0'的ASCII码值为0,因此,以上表达式亦可以写作“str[i]!=0”或“str[i]”,逻辑上表示字符串未结束。

程序运行结果如图5-19所示。

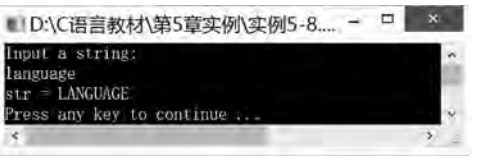

图5-19　小写字母转换为大写字母

【实例5-9】 输入任意三个字符串(每个字符串少于15个字母),按从小到大的顺序排序并输出。

分析：将三个字符串分别输入3×15的二维字符数组s中,假设输入后数组的存储形式如图5-20所示。

A	c	k	n	o	w	l	e	g	e	m	e	n	t	\0
C	a	l	c	u	l	a	t	o	r	\0				
C	o	n	c	e	i	v	a	b	l	e	\0			

图5-20　三个示例字符串的存储形式

可将数组的每一行看作一个一维数组,则三个一维数组依次为s[0],s[1],s[2],且将三个字符串排序的操作可用以下三步完成。

(1) 比较s[0]和s[1],如果s[0]>s[1],则交换s[0]和s[1]的内容。

(2) 比较s[0]和s[2],如果s[0]>s[2],则交换s[0]和s[2]的内容,此时s[0]的内容已经为三个字符串的最小值。

(3) 比较s[1]和s[2],如果s[1]>s[2],则交换s[1]和s[2]的内容。

程序代码如下：

```
/*实例5-9*/
#include <stdio.h>
```

```
#include <string.h>
int main()
{
    char sTemp[15], s[3][15];
    int i;
    printf("Input three strings:\n");
    for(i=0; i<3; i++)
        gets(s[i]);
    if(strcmp(s[0], s[1])>0)
    {
        strcpy(sTemp, s[0]); strcpy(s[0], s[1]); strcpy(s[1], sTemp);
    }
    if(strcmp(s[0], s[2])>0)
    {
        strcpy(sTemp, s[0]); strcpy(s[0], s[2]); strcpy(s[2], sTemp);
    }
    if(strcmp(s[1], s[2])>0)
    {
        strcpy(sTemp, s[1]); strcpy(s[1], s[2]); strcpy(s[2], sTemp);
    }
    printf("The strings sorted are:\n");
    for(i=0; i<3; i++)
        puts(s[i]);
    return 0;
}
```

说明：

(1) 变量 sTemp 表示临时字符串，用 temp 来参与命名，增加程序可读性。

(2) 程序中“strcpy(sTemp, s[0]); strcpy(s[0], s[2]); strcpy(s[2], sTemp);”与整型变量下的程序段“t=a;a=b;b=t;”类似，实现字符串 s[0]和 s[2]的内容互换。注意：这里是三条语句，因此，各 if 语句的花括号{}不能省略。

程序运行结果如图 5-21 所示。

图 5-21　三个字符串按从小到大的顺序排序

5.5　数组与指针

指针是 C 语言中的一种重要数据类型，通过前面的章节已对指针有了基本的认识，灵活地使用指针，可以方便、有效地组织和表示复杂的数据结构，本节主要介绍如何用指针访

问数组元素。

5.5.1 使用指针处理数组元素

在了解指针的基础上,本节将再结合数组使用,因为指针和数组有着密切的关系,几乎所有使用数据下标来实现的操作均可由指针来完成。

1. 使用指针处理一维数组

数组是由一组有序的、具有相同数据类型的数据构成的集合。在使用数组时可通过下标来访问其中的元素,且这些数组元素在内存中是按下标顺序连续存放的,换句话说,它们的地址也是按下标顺序连续的。如图 5-22 所示即为以下数组 a 对应的内存分布情况。

```
int a[5] = { 5, 10, 15, 20, 25 };
```

	a[0]	a[1]	a[2]	a[3]	a[4]	
…	5	10	15	20	25	…

图 5-22 数组 a 的存储形式

从图 5-22 中可以看出,如果知道元素 a[2]的地址 p,那么 p+1 就是 a[3]的地址,p-1 就是 a[1]的地址,而且除第一个元素 a[0]和最后一个元素 a[4]外,所有的元素都有这样的规律。对于当前元素 a[0],虽然 p+1 将得到 a[1]的地址,但 p-1 得到的地址已经超出数组 a 所分配的内存空间,对该内存空间的访问极有可能导致非法操作;同理,对于元素 a[4],对地址 p+1 指向的内存进行访问也可能导致非法操作。

因此,对于数据 a 中的非首尾元素 a[i],如果其指针为 p,均有 p+1 指向 a[i+1],p-1 指向 a[i-1];而对于数组的首尾元素,当 p+1 或 p-1 不超出数组分配内存范围时同样成立。

【实例 5-10】 使用指针访问数组元素。

具体程序代码如下:

```
/*实例 5-10*/
#include <stdio.h>
int main()
{
    int a[5] = { 10, 11, 12, 13, 14 };
    int* p0 = &a[0], *p3 = &a[3];                    //用 &a[0]取 a[0]的地址
    printf("%d, %d\n", a[0], *p0);                   //用 *p0 取 p0 所指向的变量
    printf("%d, %d\n", a[1], *(p0+1));
    printf("%d, %d, %d\n", a[2], *(p0+2), *(p3-1));
    return 0;
}
```

说明:语句"int* p0 = &a[0], *p3 = &a[3];"中定义了两个指针变量,即 int *p0, *p3;这里的第二个星号不能省略。

程序运行结果如图 5-23 所示。

函数中第 2 条语句将 a[0]的地址赋给指针 p0,将 a[3]的地址赋给指针 p3;紧接着执行第 1 条 printf()语句时,将输出 a[0]的值和 *p0 的值,而由于 p0 本来就指向 a0,故在此的

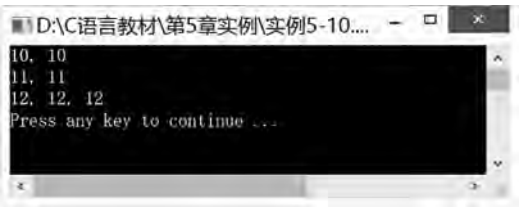

图 5-23　使用指针访问数组元素

a[0]和 * p0 是等价的，因此将输出两次 a[0]的值；同理，在执行第 2 条 printf()语句时，由于(p0＋1)是指向 a[1]的，最终也将输出两次 a[1]的值；对于第 3 条 printf()语句，由于指针 p3 是指向元素 a[3]的，所以 p3－1 是指向 a[2]的，*(p3－1)就表示 a[2]，故第 3 个 printf()函数将输出 3 次 a[2]的值。

通过实例 5-10 得知，在定义数组后，要使用指针来访问数组元素，只需已知其中某个元素的地址以及对应的下标即可。可以根据元素间的相对位置对当前指针进行“偏移”操作来得到要访问元素的地址，进而通过取内容运算符访问到该元素，如实例 5-10 中的 *(p0＋2)和 *(p3－1)。但相比之下，因为数组元素在内存中是连续存储的，因此更关注第一个元素的地址(又称为数组的首地址)，具体见实例 5-11。

【实例 5-11】 通过数组的首地址来访问数组元素。

具体程序代码如下：

```
/* 实例 5-11 */
#include <stdio.h>
int main()
{
    int a[5] = { 10, 11, 12, 13, 14 }, i;
    int* p0 = &a[0];                              //&a[0]为数组的首地址
    for(i = 0; i < 5; i ++)
        printf("%d\n", *(p0 + i));                //*(p0 + i)等价于 a[i]
    return 0;
}
```

程序运行结果如图 5-24 所示。

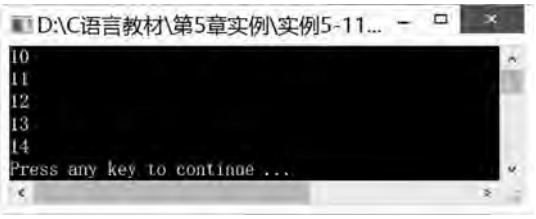

图 5-24　数组首地址的使用

在使用首地址后访问数组元素的编写方式：

```
printf("%d\n", *(p0 + i));
```

不使用地址而直接使用数组元素的编写方式：

```
printf("%d\n", a[i]);
```

两种编写方式非常类似,可以很容易地将一种方式改写为另一种方式。两种编写方式最大的区别在于使用指针时,需要添加类似"int * p0 = &a[0];"的语句来定义首地址,那么有没有更简略的方式呢?回答是肯定的。

其实,在了解指针及地址的知识后,在此需要对数组定义语句:

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

进行一个补充说明,在该语句中a除了表示数组的名称外,还表示数组的首地址,即

(1) a既表示包含5个整型元素的数组名称,又表示一个地址。

(2) a表示的是首地址,即a等价于&a[0]。

于是,实例5-11的代码又可改写为如下方式:

```
/*实例5-11*/
#include<stdio.h>
int main()
{
    int a[5] = { 10, 11, 12, 13, 14 }, i;
    int* p0 = a;                                    //a相当于&a[0]
    for(i=0; i<5; i++)
        printf("%d\n", *(p0+i));                    //*(p0+i)等价于a[i]
    return 0;
}
```

由语句"int * p0 = a;"中可知p0和a等价,进一步可以改写为如下方式:

```
/*实例5-11*/
#include<stdio.h>
int main()
{
    int a[5] = { 10, 11, 12, 13, 14 }, i;
    for(i = 0; i<5; i++)
        printf("%d\n", *(a+i));                     //*(a+i)等价于a[i]
    return 0;
}
```

可通过程序运行结果验证,它们确实有相同的功能,且第二种改写方式更为简略,更加接近于不用指针时的编写方式,当然程序中的语句"printf("%d\n",*(a+i));"还可写作:

```
printf("%d\n", p0[i]);                              //p0和a等价,p0[i]和a[i]等价
```

至此,我们对使用指针访问数组做如下总结。

(1) 数组名称也是数组的首地址,因此语句:

```
int a[5], *p;
p = &a[0];
```

等价于

```
int a[5], *p;
p = a;
```

或

```
int a[5], *p = a;
```

但不能写作：

```
int *p = a, a[5];
```

因为数组 a 需要先定义后使用，在执行 p＝a 时，a 还未定义。

(2) 在有语句“int a[5], *p＝a;”之后，描述下标为 i 的元素可有以下 4 种等价方法。

- a[i]
- p[i]
- *(a＋i)
- *(p＋i)

注意：p 和 a 还是有区别的：a 为数组的首地址，在数组被定义时被系统统一分配，在整个程序运行时是不会变化的，因此是常量；p 为整型指针，可以指向任何一个整型数据，包括数组 a 中的某个数组元素，因此是变量。所以不能对 a 进行 a++等自增或自减操作，而 p 则可以，如执行 p++后，p 将指向数组中的下一个元素。

点拨：在语句“int a[5], *pA＝a;”和“float b[5], *pB＝b;”中，pA＋1 与 pB＋1 也是有区别的，因为指针变量的加减和指针本身的类型是相关的，由于 pA 为整型指针，pA＋1 代表 pA 向后移动一个 int“单位”(即 4 字节)后得到的地址值，而 pB＋1 时移动的“单位”为一个 float 型变量所占的字节数(即 4 字节)。

【实例 5-12】 利用指针来计算数组的最大值和最小值及其位置。

根据前面相应的数组实例分析，可知只需将相应部分的描述使用指针替换即可，具体程序代码如下：

```
/*实例 5-12*/
#include <stdio.h>
int main()
{
    int a[10], *p=a, *pMin, *pMax, i;
    printf("Input 10 numbers:\n");
    for(i=0; i<10; i++)
        scanf("%d", p + i);
    pMin = pMax = a;
    p = a + 1;
    for(i=1; i<10; i++, p++)
    {
        if(*p > *pMax)
            pMax = p;
        if(*p < *pMin)
            pMin = p;
```

```
    }
    printf("Min is a[%d], its value is %d\n", pMin - a, *pMin);
    printf("Max is a[%d], its value is %d\n", pMax - a, *pMax);
    return 0;
}
```


说明：

(1) 语句“scanf("%d", p+i);”中 p+i 相当于 a+i，即变量 a[i]的地址。

(2) 语句 i++，p++中相当于两个循环变量同步递增，以实现与数组元素的依次比较。

程序运行结果如图 5-25 所示。

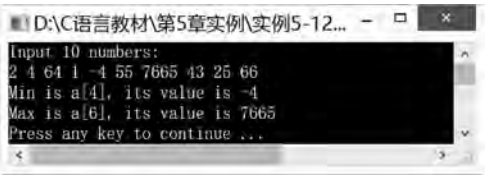

图 5-25　使用指针计算最大值和最小值及其位置

程序中主要分为数据输入、比较及输出三部分。在输入时，使用 p+i 的形式来获取当前元素的地址，并作为 scanf 函数中的参数传入，随着 i 的变化，p+i 总是得到当前元素的内存地址，使得输入的值能准确赋给相应的元素；在比较部分则使用 p++的方式，通过直接修改指针 p 的值，来完成指向当前元素的指针变换，使得在当前元素处理结束时，通过 p++将指向当前元素的指针 p 指向下一个元素，并进入下一次循环，一直到循环结束，且通过 *pMax，*pMin 来指代当前得到的最大值及最小值，通过与所有循环中的 *p 进行比较并更新后，得到的 pMax 和 pMin 即指向最大值和最小值所在的存储单元；在输出部分，为了得到最大值和最小值的下标值，直接使用类似于 pMax－a 和 pMin－a 的方式来计算指针 pMax(或 pMin)与数组的首地址 a 之间的元素个数，使得整个程序满足题目要求。

在使用类似 p++的指针运算时，还需要注意以下几种常见的指针运算相关表达式，更需要注意其中的运算次序。

(1) (*p)++：先取 *p 的值，再将该值加 1 并存入 p 指向的内存空间。

(2) ++(*p)：先取 *p 的值，将该值加 1 并存入 p 所指的内存空间后，再取 *p 的值。

(3) (*p)--：先取 *p 的值，再将该值减 1 并存入 p 指向的内存空间。

(4) --(*p)：先取 *p 的值，将该值减 1 并存入 p 所指的内存空间后，再取 *p 的值。

(5) *p++：先取 *p 的值，再执行 p++，使得 p 指向下一个元素的内存单元。

(6) *++p：先执行 p++，使得 p 指向下一个元素的内存单元，再取 *p 的值。

(7) *p--：先取 *p 的值，再执行 p--，使得 p 指向上一个元素的内存单元。

(8) *--p：先执行 p--，使得 p 指向上一个元素的内存单元，再取 *p 的值。

为了进一步了解以上几种表达式运算过程，可对照以下示例的运行结果。

```
/*示例程序*/
#include <stdio.h>
int main()
```

```
{
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, *p = a, n;
    n = *p++;
    printf("n = %d\n", n);
    n = *++p;
    printf("n = %d\n", n);
    n = (*p)++;
    printf("n = %d\n", n);
    n = (*p)++;
    printf("n = %d\n", n);
    return 0;
}
```

程序运行结果如图 5-26 所示。

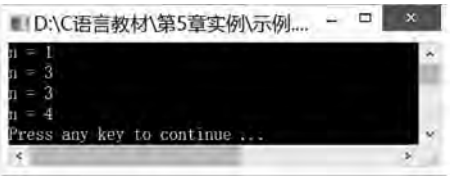

图 5-26　示例程序的运行结果

2. 使用指针处理多维数组

和一维数组类似，也可以将指针指向多维数组中的元素，通过该指针对多维数组的元素进行访问和处理，为了便于理解，在此以二维数组为例进行描述。

先回顾一下二维数组的一些特性，对于一个包含 3 行 4 列的二维数组定义语句：

```
float b[3][4];
```

可将其理解为一个包含三个复杂元素(b[0]，b[1]和 b[2])的一维数组，每个复杂元素对应二维数组的一行，每个复杂元素内部又包含四个较简单的 float 型元素，具体对应关系如下。

b[0]——b[0][0]，b[0][1]，b[0][2]，b[0][3]

b[1]——b[1][0]，b[1][1]，b[1][2]，b[1][3]

b[2]——b[2][0]，b[2][1]，b[2][2]，b[2][3]

且整个二维数组以先行后列的形式连续存放在内存中，如图 5-27 所示。

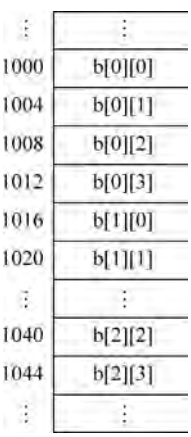

图 5-27　二维数组的存储

接下来我们来讨论二维数组中的指针情况，主要根据数组的名称，即数组的首地址来进行分析。

对于某一行(以 i 为下标)数据，它是一个以 b[i]为名称的一维数组，因此，b[i]本身就是一个指针，表示该行中第一个元素的地址，即 b[i]与 &b[i][0]或 *b[i]与 b[i][0]等价，通过对 b[i]进行加减操作即可访问到该行的其他元素，如 b[i]+2 表示元素 b[i][2]的地址，*(b[i]+2)即为元素 b[i][2]。

对于整个二维数组，共有三个元素且数组名称为 b，则 b 就是首地址，表示这三个元素

中第一个元素的地址,即b与&b[0]或*b与b[0]等价,又因为将b看作一维数组时该数组中的元素为二维数组中的一行,所以对b进行的加减操作将每行元素所占的字节数为"单位"进行,即b+i表示下标为i的行的地址,而该行的地址总是用首地址来表示,即b[i]与&b[i][0]等价,综合两种等价关系可知,*(*b)与*(b[0])和b[0][0]均等价,注意:在此的二维数组名称b前面有两个"*",当数组为三维数组或更多维数组时,类似的规律同样存在,当然该等价关系也可以表示为*b与b[0]和&b[0][0]均等价,它们表达相同的意思,只是在描述上有所不同。

以上描述中的结论也可通过对数组的内存分布情况分析得到,同时,由于存在较多的等价描述方式,同一功能又可写作多种方式。

【实例5-13】 使用指针的方式输入整型数组的元素值,计算并输出这些元素的和。

结合前面相应的数组实例分析,将相应部分的描述使用指针替换,具体程序代码如下:

```
/*实例5-13*/
#include<stdio.h>
int main()
{
    int b[3][4], i, j, sum = 0;
    printf("Please input a 3*4 matrix:\n");
    for(i=0; i<3; i++)
        for(j=0; j<4; j++)
            scanf("%d", b[i]+j);
    for(i=0; i<3; i++)
        for(j=0; j<4; j++)
            sum += *(b[i]+j);
    printf("sum = %d\n", sum);
    return 0;
}
```

说明: 表达式b[i]+j用于表示数组b中行下标为i、列下标为j的元素地址。

程序运行结果如图5-28所示。

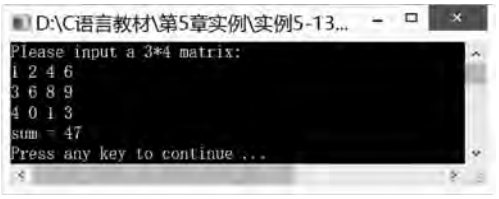

图5-28 借助数组首地址来计算数组元素的和

3. 由指针组成的数组

在使用指针方式处理数组元素时,按照习惯常将数组的名称(即数组的首地址)赋给某个指针变量,再使用该指针变量来进行处理。例如:

```
int a[5], *p = a;
```

语句中将一维数组的名称(即首地址)直接赋值给指针变量p,此时使得p和a几乎等价,在指定数组a的元素时可以直接使用p[i]代替a[i],p和a除了一个为变量、另一个为常量之

外，几乎没什么区别。但这种情况仅限于一维数组中。例如：

```
float array[2][3], *pTemp = array;
```

语句中将二维数组名称(即首地址)直接赋值给指针变量 pTemp，使得 pTemp 和 array 相等，但要注意此时的 pTemp 和 array 不等价，除了一个为变量、另一个为常量外，对它们做加减法运算时也有很大区别。对于首地址 array，执行 array+1 时，总是把 array 看作两个较复杂的元素来进行，每个复杂元素为数组 array 的一行，每行内包含三个 float 型变量，因此 array+1 时总是以这里的行为“单位”进行运算，整个地址向后偏移 4*3(其中 3 表示 3 个 float 型变量，4 表示每个 float 型变量占 4 字节)字节的内存空间；而对于指针 pTemp，它只是一个 float 型的指针变量，对其进行的加减法运算，如 pTemp+1，只会将 pTemp 指向的地址向后偏移 4 字节的内存空间。

因此，在使用指针指向数组并处理数组元素时，需要特别注意，最好将多维数组分解到一维数组，如将语句“int b[3][4];”中二维数组 b 的每一行看作一个元素，即整个数组由三个复杂元素 b[0]，b[1]和 b[2]组成，而这三个复杂元素就是一维数组，可通过操作一维数组的方式来操作它们，在方便理解的同时，降低直接操作多维数组中的复杂性。例如：

```
int b[3][4];
int *p0, *p1, *p2;
p0 = b[0];
p1 = b[1];
p2 = b[2];
```

可以看出，通过以上赋值后的指针变量 p0，p1，p2 已分别表示第一行、第二行以及第三行中一维数组的首地址，即相当于每行中一维数组的别名，可直接使用 p0[0]表示第一行中的第一个元素，p2[3]表示第 3 行的第 4 个元素等。通过观察上面示例代码可以发现，语句“int *p0，*p1，*p2;”相当于同时定义了三个相同类型的指针变量，或由三个相同类型的指针变量组成的集合，即一维数组，只是这里的数组元素类型为指针类型 int*，如此表示的数组称为指针数组。

定义指针数组与定义一般数组类似，只须在表示数据类型的部分添加相应的表示指针的“*”即可。如定义一维指针数组的一般形式如下：

```
数组类型  数组名称[长度];
```

其中，数组类型为所定义数组中各元素的数据类型，在此由于是指针类型，即可写为类似 int* 等形式，数组名称及长度与定义一般的一维数组相同。如语句“float *a[5];”即定义了一个 float *类型的一维数组。

通过使用一维指针数组后，可以对上面的程序进行改写如下：

```
int b[3][4], *p[3], i;
for(i = 0; i < 3; i ++)
    p[i] = b[i];
```

或直接写作对数组进行初始化的形式：

```
int b[3][4], *p[3] = { b[0], b[1], b[2] };
```

可见,通过将指针指向一维数组可以较简单地处理该一维数组中的元素,同理,如果数组为由语句"int a[3][4][5];"定义的三维数组,可以将其看作有三层,每层上有4行5列元素,那么,可以使用语句"int **pA[3][4];"定义一个二维指针数组来指向该三维数组中的每一行元素,其中第一个"*"表示将每一层看作一个元素,且指向该一维数组的指针,第二个"*"表示在一层的基础上将每层中的每一行看作一个元素后,指向某层中一维数组的指针。如此即可使用和上面类似的for语句进行指针赋值,使pA[0][0]指向数组a中第一层的第一行,pA[0][0]+2即为该行中第三个元素的地址,而pA[2][3]则指向第3层的第4行,pA[2][3]+4即为第3层第4行的第5个数,即数组的最后一个元素。为了叙述方便,在此仍以指向二维数组的一维指针数组来进行举例。

【实例5-14】 使用指针的方式输入整型数组的元素值,计算并输出这些元素的和。

具体程序代码如下:

```
/*实例5-14*/
#include<stdio.h>
int main()
{
    int b[3][4], i, j, sum = 0, *pB[3];
    for(i = 0; i<3; i++)
        pB[i] = b[i];
    printf("Please input a 3*4 matrix:\n");
    for(i = 0; i<3; i++)
        for(j = 0; j<4; j++)
            scanf("%d", &pB[i][j]);
    for(i = 0; i<3; i++)
        for(j = 0; j<4; j++)
            sum += *(pB[i]+j);
    printf("sum = %d\n", sum);
    return 0;
}
```

说明:语句"pB[i]=b[i];"将数组b中行下标为i(i取0到2,共3行)的首地址复制到pB[i]中,因此,pB[0]相当于第1行的首地址,以此类推。

程序运行结果如图5-29所示。可以发现,在通过一维指针数组来操作二维数组时,也可以用多种方式来引用二维数组。如实例5-13和实例5-14的代码中,b[i]+j与pB[i]+j均用来表示数组b中元素b[i][j]的地址。

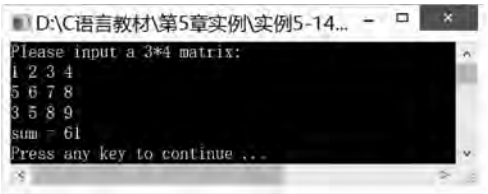

图5-29 借助指针来计算数组元素的和

总之,在了解数组的名称又是数组的首地址后,可以有多种访问数组元素的方式,且对于多维数组,可将该多维数组看作较复杂的一维数组进行处理。要了解更多关于多维数组

及多维指针等方面的知识，可参考相关资料及网络资源。

5.5.2 使用指针处理字符串

在使用指针访问一般类型的数组元素后，由于字符数组常用于存储字符串且在访问时总是对整个字符串进行操作，不是对单个字符数组的元素，在此对使用指针处理字符串进行单独介绍。

【实例 5-15】 使用指针简单操作字符串。

具体程序代码如下：

```
/* 实例 5-15 */
#include <stdio.h>
int main()
{
    char str[] = "Hello,everyone!";
    printf("%s\n", str);
    printf("%c\n", str[6]);
    return 0;
}
```

程序运行结果如图 5-30 所示。

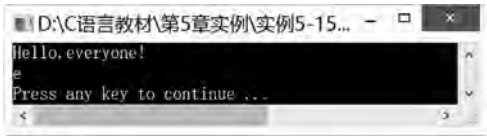

图 5-30 使用指针简单操作字符串

结合程序运行结果可知，程序中的 str 为字符数组的名称，又为字符串"Hello, everyone!"的首地址，而且字符串中的字符个数是由字符串结束符的位置来决定的，与字符数组 str 中的元素个数相关性较弱，因此，实例 5-15 的程序有如下等价形式：

```
/* 实例 5-15 */
#include <stdio.h>
int main()
{
    char str[] = "Hello,everyone!";
    char *pStr = str;
    printf("%s\n", pStr);
    printf("%c\n", pStr[6]);
    return 0;
}
```

或直接写为：

```
/* 实例 5-15 */
#include <stdio.h>
int main()
{
    char *pStr = "Hello,everyone!";
```

```
    printf("%s\n", pStr);
    printf("%c\n", pStr[6]);
    return 0;
}
```


总之,在使用字符数组表示字符串时,可通过字符类型的指针以更简单的形式来进行表达,具体如下。

(1) 语句:

```
char str[] = "Hello, World!";
```

等价于

```
char *str = "Hello, World!";
```

或

```
char *str;
str = "Hello, World!";
```

(2) 在对字符串进行输出时,如语句"printf("%s\n", pStr);"总是输出以 pStr 为首地址的字符串,如下面语句:

```
char *str = "Hello, World!";
str = str + 6;
printf("%s", str);
```

的输出结果为"World!",因为 printf()语句中的 str 为字符串"World!"的首地址。

(3) 语句:

```
char str1[] = "Hello, World!";
```

和

```
char *str2 = "Hello, World!";
```

的主要区别为:str1 为常量,而 str2 则为变量,不能对 str1 进行赋值运算,如进行类似"str1=str1+6;"的运算,但 str2 则可以。

【实例 5-16】 使用指针查找已知字符串中字符'a'的个数。

结合前面字符数组的相应实例分析,可知具体程序代码如下:

```
/*实例 5-16*/
#include <stdio.h>
int main()
{
    char *str = "I am a student";
    int n = 0;
    for(; *str != '\0'; str++)
        if(*str == 'a')
            n++;
    printf("The number of \'a\' is %d\n", n);
    return 0;
}
```

说明：

(1) 格式字符串"The number of \'a\' is %d\n"中通过\'表示单引号，因此，\'a\'所对应的输出为'a'。

(2) 描述 for(；*str!='\0'；str++)与字符串中的常规描述 for(i=0；str[i] !='\0'；i++)类似。

程序运行结果如图 5-31 所示。其中 *str 表示取 str 当前指向的字符。

图 5-31　使用指针查找已知字符串中元素的个数

5.5.3　数组与指针的程序举例

【实例 5-17】 输入 10 个整数到数组 a，计算下标为偶数的元素之和。

具体程序代码如下：

```
/* 实例 5-17 */
#include <stdio.h>
int main()
{
    int a[10], *p = a, i, sum;
    printf("Please input 10 numbers:\n");
    for(i = 0; i < 10; i ++)
        scanf("%d", p + i);
    sum = 0;
    for(i = 0; i < 10; i += 2, p += 2)
        sum += *p;
    printf("sum = %d\n", sum);
    return 0;
}
```

说明：程序段 i＋＝2，p＋＝2 用于实现数组中偶数下标 i 以及对应指针变量 p 的依次变换。

程序运行结果如图 5-32 所示。

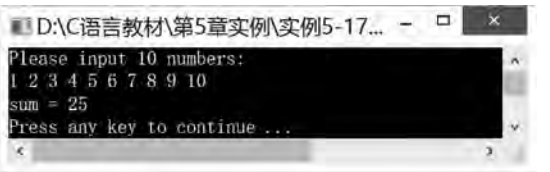

图 5-32　使用指针计算数组 a 中偶数下标的元素之和

【实例 5-18】 输入一个字符串(少于 80 个字符)，统计并输出小写元音字母的个数。

具体程序代码如下：

```
/* 实例 5-18 */
#include <stdio.h>
#include <string.h>
int main()
{
    char str[80], *p = str;
    int nCount = 0;
    printf("Please input a string:\n");
    gets(p);
    for(; *p != '\0'; p++)
        if(*p == 'a'||*p == 'e'||*p == 'i'||*p == 'o'||*p == 'u')
            nCount++;
    printf("nCount = %d\n", nCount);
    return 0;
}
```

说明：小写元音字母有 a、e、i、o、u，因此 if 语句中通过或运算符“‖”分别表达这 5 种可能。

程序运行结果如图 5-33 所示。

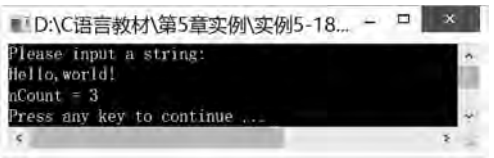

图 5-33　使用指针统计小写元音字母的个数

5.6　数组综合举例

在 C 程序中，数组将具有相同类型的批量数据有序组织，能方便、快速地解决许多实际问题，以下介绍几种典型的方法。

5.6.1　查找

查找即在同类型的批量数据(或记录集合，又称为“查找表”)中找出某个已知特征数据(或记录)的操作，若通过查找后找到该已知特征数据(或记录)，则称为查找成功，否则称为查找失败。

在实际生活中，到处都在进行查找操作，如在词典中查找某个特定的词；在课程表中查找一门想要选修的课程；在通讯录中查找某个朋友的电话号码等。根据查找表中数据之间的不同关系，人们提出不同的查找算法，下面介绍两种常用的查找方法。

1. 简单顺序查找

简单顺序查找的基础思想是从查找表的一端开始，按顺序逐个扫描表中的每一个元素，并将扫描到的元素与已知特征数据相比较。若当前元素满足已知的特征，则查找成功并结束查找；若在扫描完整个查找表都未找到符合特征的元素，则查找失败。

【实例 5-19】 编写程序完成以下功能：先输入由 10 个学生成绩组成的数组 a，再输入

一个成绩 x，在数组 a 中查找成绩 x，如果找到，则输出找到时的位置，否则输出“Not Found”。

程序流程如图 5-34 所示，程序代码如下：

```
/*实例 5-19*/
#include<stdio.h>
int main()
{
    int a[10], x, i;
    printf("Input 10 scores:\n");
    for(i=0; i<10; i++)
        scanf("%d", &a[i]);
    printf("Input x:\n");
    scanf("%d", &x);
    for(i=0; i<10; i++)
        if(a[i] == x)
            break;
    if(i<10)
        printf("%d is the %dth score.\n", x, i+1);
    else
        printf("Not Found\n");
    return 0;
}
```

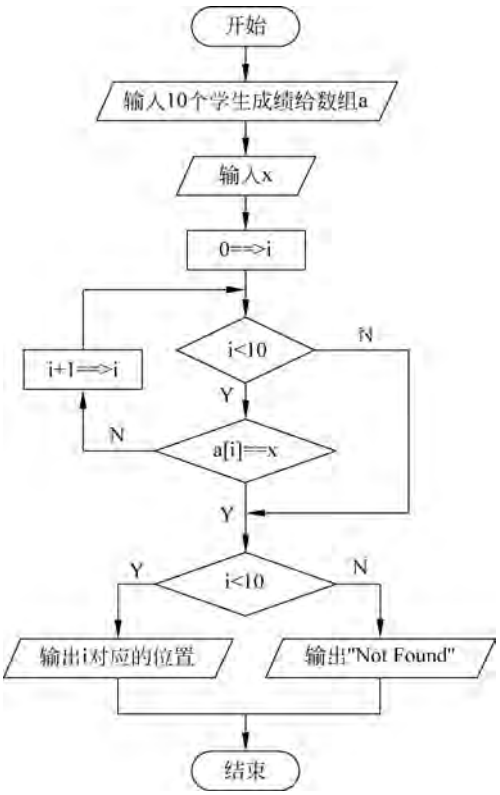

图 5-34　简单顺序查找算法的流程

说明：语句“printf("%d is the %dth score.\n", x, i+1);”用于输出查找的x在数组中的位置，由于实际位置从1开始计数，而数组下标则从0开始计数，因此，这里用i+1得到位置的值。

程序运行结果如图5-35所示。

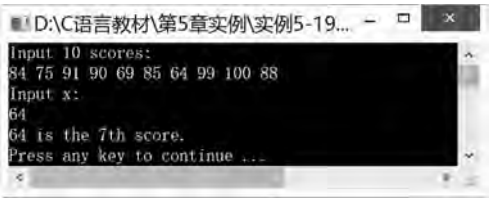

图5-35　简单顺序查找实例

2. 二分查找

二分查找又称为折半查找，是一种适合在已经排序后的数据序列(有序表)中进行查找的方法。其基本思想是在查找过程中采用跳跃的方式查找，即总是以有序表的中间位置为比较对象，根据比较结果来确定要查找的元素位于中间位置之前还是中间位置之后，从而将查找区间缩小一半，再取缩小后的查找区间的中间位置为新的比较对象，以此类推，具体步骤如下(假设有序表为升序排序)。

(1) 将整个有序表作为最初的查找区间。

(2) 确定查找区间的中间位置：mid=(left+right)/2，其中left为整个查找区间中第一个元素的位置，right为最后一个元素的位置。

(3) 将要查找的数据与中间位置(序号为mid)的数据进行比较：若相等，则查找成功并返回；若大于，则将右半个区域作为新的查找区间，回到步骤(2)继续进行折半查找；若小于，则将左半个区域作为新的查找区间，回到步骤(2)继续进行折半查找。

(4) 重复步骤(2)和步骤(3)，直到找到要查找的值(即查找成功)，或新的查找区间不存在(即left>right时查找失败)。

【实例5-20】 编写程序完成以下功能：按从小到大的顺序输入由10个学生成绩组成的数组a，再输入一个成绩x，在数组a中查找成绩x，如果找到则输出找到时的位置，否则输出“Not Found”。

程序流程如图5-36所示，程序代码如下：

```
/* 实例 5-20 */
#include <stdio.h>
int main()
{
    int a[10], x, i, mid, left, right, flag;
    printf("Input 10 scores:\n");
    for(i = 0; i<10; i++)
        scanf("%d", &a[i]);
    printf("Input another score:\n");
    scanf("%d", &x);
    flag = 0; left = 0; right = 9;
    while(left <= right)
```

```
    {
        mid = (left + right) / 2;
        if(x == a[mid])
        {
            flag = 1; break;
        }
        else if(x < a[mid])
            right = mid - 1;
        else
            left = mid + 1;
    }
    if(flag)                    /* 等价于 if(flag == 1) 或 if(flag != 0) */
        printf("%d is the %dth score.\n", x, mid + 1);
    else
        printf("Not Found.\n");
    return 0;
}
```

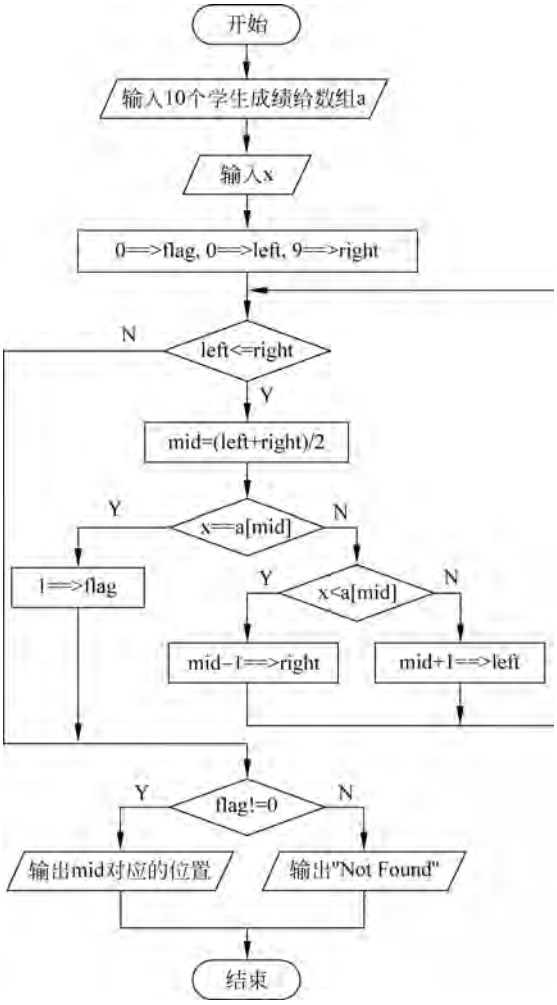

图 5-36　二分查找算法的流程

说明：

(1) 变量 flag 用于表示某种标记，在此表示是否查找到指定值，且在查找前先设置 flag=0，表示未找到指定值。

(2) 变量 left、right 分别用于表示查找区域左边元素下标和右边元素下标，变量 mid 则用于表示查找区域的中间位置，因此，通过 mid=(left+right)/2 来计算中间位置。以上的命名用于增加程序可读性。

程序运行结果如图 5-37 所示。

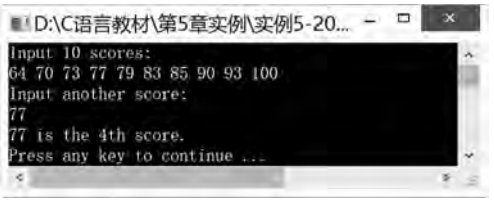

图 5-37　二分查找实例

可见，简单顺序查找在思想上简单易行，程序也容易实现，但与二分查找相比，二分查找算法充分利用了元素间的次序关系，通过跳跃查找的方式，有比较次数少、查找速度快等优点。但因为二分查找要利用元素间的次序关系，因此，它的通用性又不如简单顺序查找，只适合使用在已经排好序的数据序列中。关于查找方面的算法还有很多，有兴趣的读者可以查阅相关的参考资料。

5.6.2　排序

在二分查找时，需要将预先排好序的数据输入计算机中以供后续操作，那么能不能利用计算机完成排序操作呢？

在实际应用中，排序方法的使用非常广泛，如期末考试后的成绩排名；网上购物时，将感兴趣的商品按价格进行排序以供参考；在使用计算机文件时常会将文件按创建日期进行排序以方便文件查阅。它们的目的都是将由杂乱无章的数据元素组成的集合，按某个关键字重新整理排列成一个有序的数据序列。

排序的方法有很多，以下介绍两种常用的方法。

1. 冒泡排序

冒泡排序(BubbleSort)即在要排序的数据序列中，依次比较相邻的两个数，并将小数放在前面，大数放在后面，具体步骤如下(假设要对 N 个数进行排序，且 N 为 5 时的排序过程如图 5-38 所示)。

(1) 比较第 1 个数和第 2 个数，当第 1 个数大于第 2 个数时，调换两个数的位置，保持将小数放前，大数放后。

(2) 比较第 2 个数和第 3 个数，同样将小数放前，大数放后，如此继续，直至比较到第 N−1 个数和第 N 个数，并将小数放前，大数放后。至此称为第一趟排序结束，从图 5-38 中可以看出，通过第一趟排序将最大的数(图中为 9)放到第 N 个位置。

(3) 开始第二趟排序，仍从第 1 个数和第 2 个数开始比较，并将小数放前，大数放后，一直比较到第 N−2 个数和第 N−1 个数(第 N 个数已经是最大的数，不用再比较)，第二趟排

图 5-38　冒泡排序的排序过程

序结束，在第 N－1 个位置上得到一个新的最大数（在整个序列中是第二大的数，图中为 8）。如此继续，重复以上过程，直至最终完成排序，一共需要进行 N－1 趟排序。

若将待排序的数据序列看作从上到下存放，由于在排序过程中总是小数放前，大数放后，相当于气泡往上升，所以称为冒泡排序。

【实例 5-21】 任意输入 5 个整数，利用冒泡排序对其进行升序排序。

算法流程如图 5-39 所示，程序代码如下：

```
/*实例 5-21*/
#include <stdio.h>
int main()
{
    int a[5], i, j, temp;
    printf("Input 5 numbers:\n");
    for(j=0; j<5; j++)
        scanf("%d", &a[j]);
    for(i=5-1; i>0; i--)
    {
        for(j=0; j<i; j++)
        {
            if(a[j]>a[j+1])
            {
                temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
    printf("Numbers after sorted are:\n");
    for(i=0; i<5; i++)
        printf("%d\t", a[i]);
    printf("\n");
    return 0;
}
```

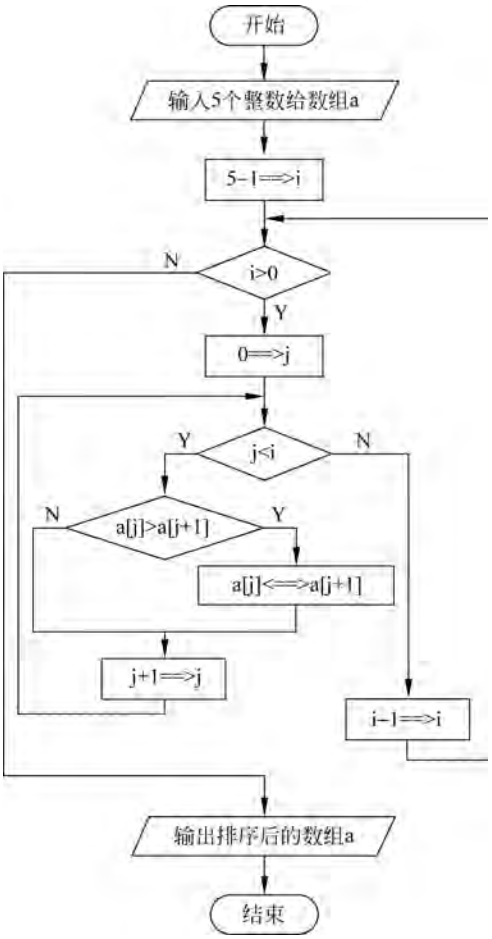

图 5-39　冒泡排序算法的流程

说明：冒泡排序的代表性操作是依次比较下标相邻的两个数组元素，并在题目要求的顺序条件时互换两个元素的值。

程序运行结果如图 5-40 所示。

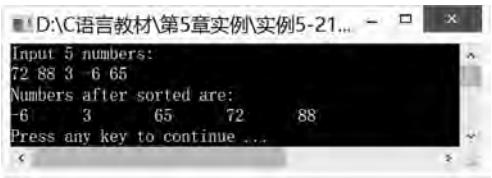

图 5-40　冒泡排序实例

2. 选择排序

选择排序即先在要排序的数据序列中选出最小的数，将最小数与第 1 个数据进行交换，然后再从第 2 个数开始的其余数据中选出次小的数与第 2 个数进行交换，以此类推，直到选出倒数第 2 小的数与倒数第 2 个数进行交换，至此所有数据均已排好序。具体步骤如下(假设要对 N 个数进行排序，且 N 为 5 时的排序过程如图 5-41 所示)。

(1) 第一趟排序：找出 N 个数中的最小值，并与第 1 个数进行交换。

(2) 第二趟排序：找出从第 2 个数到第 N 个数的最小值，并与第 2 个数进行交换。

(3) 以此类推，开始第三趟和第四趟排序，一直到第 N－1 趟排序找出第 N－1 个数到第 N 个数的最小值，并与第 N－1 个数进行交换，至此选择排序已经完成。

由于每次都是在多个数中选择一个最小的数进行操作，所以称为选择排序。

图 5-41　选择排序的排序过程

【实例 5-22】　任意输入 8 个整数，利用选择排序对其进行升序排序。

算法流程如图 5-42 所示。程序代码如下：

```
/* 实例 5-22 */
#include <stdio.h>
int main()
{
    int a[8], i, j, p, temp;
    printf("Input 8 numbers:\n");
    for(j = 0; j < 8; j++)
        scanf("%d", &a[j]);
    for(i = 0; i < 7; i++)
    {
        p = i;
        for(j = i + 1; j < 8; j++)
            if(a[p] > a[j])
                p = j;
        if(p != i)
        {
            temp = a[p];
            a[p] = a[i];
            a[i] = temp;
        }
    }
    printf("Numbers after sorted are:\n");
    for(i = 0; i < 8; i++)
        printf("%d\t", a[i]);
    printf("\n");
    return 0;
}
```

说明：选择排序的代表性操作是选择最大(或最小)值，并将其置于第一个(或最后一个)的位置，并以此往复选择第二大(或第二小)的值，并将其置于第二个(或倒数第二个)的位置，进而实现所有的值按大小要求排序。

程序运行结果如图 5-43 所示。

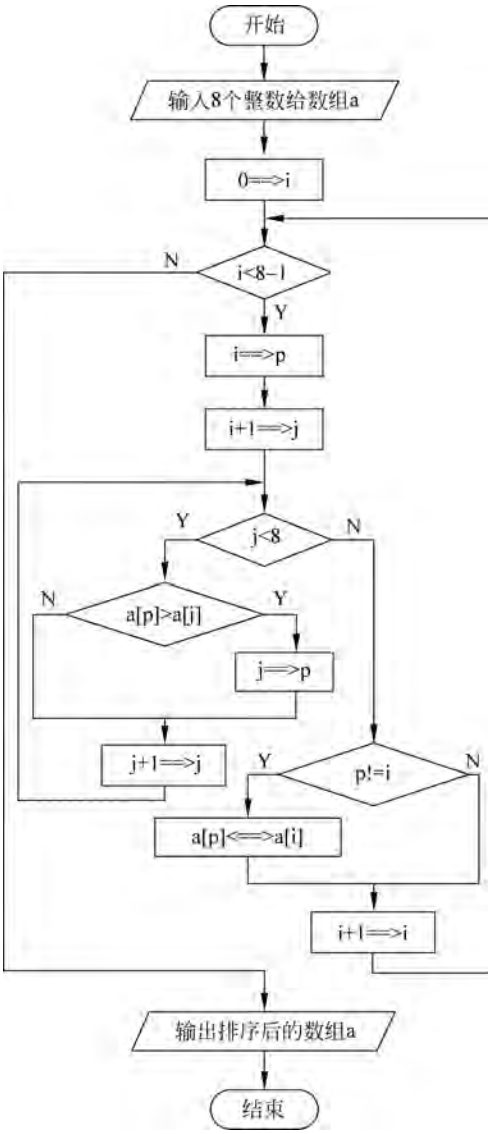

图 5-42　选择排序算法的流程

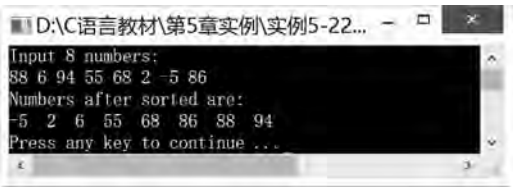

图 5-43　选择排序实例

5.6.3　求最大、最小值

求最大、最小值也是日常数据处理过程中经常进行的一种操作。如求取计算机成绩的

年级最高和最低分；求取一天中气温的最高和最低点等示例中都会涉及最大、最小值的求取操作。下面主要从一维数组和二维数组出发介绍相应的求取方法。

【实例 5-23】 输入 10 个学生成绩到数组 a，求取并输出最高分及其在数组 a 中的下标。

分析：输入 10 个学生成绩并将其保存在一维数组 a 中。定义一个中间变量 max 并将它的值取为第 1 个学生的成绩，定义另一个中间变量 index 并将它的值取为 0（第 1 个元素的下标）。依次扫描每个元素并将此元素的值与 max 的值进行比较，若当前元素的值比 max 大，则将此元素的值赋值给 max，同时将此元素的下标保存到 index 中，这样使得 max 的取值始终为当前扫描元素的最大值，而 index 则为当前扫描元素的最大值的下标。当数组中的所有元素都扫描并比较后，max 的取值即为所有分数中的最高分，相应的 index 中的取值则为此最高分在原数组中的下标。

程序代码如下：

```
/* 实例 5-23 */
#include <stdio.h>
int main()
{
    int a[10], max, index, i;
    printf("Input 10 scores:\n");
    for(i=0; i<10; i++)
        scanf("%d", &a[i]);
    max = a[0];
    index = 0;
    for(i=0; i<10; i++)                    /* 此题中也可写为 for(i=1; i<10; i++) */
        if(max<a[i])
        {
            max = a[i]; index = i;
        }
    printf("The max score is %d and its index is %d.\n", max, index);
    return 0;
}
```

说明：程序中 max 和 index 分别表示最大值及其下标，因此两个变量之间是相关的，若最大值发生变化，则对应的下标也同时发生变化，在进行更新时，需要 max 和 index 同时更新，如“max=a[0]; index=0;”以及“max=a[i]; index=i;”中的两条语句均需同时出现。

程序运行结果如图 5-44 所示。

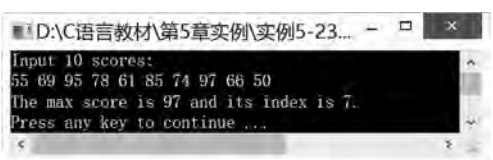

图 5-44 求一维数组的最大值实例

【实例 5-24】 输入 4 个学生三门课程的成绩到二维数组 a，求取并输出最高分、最低分及其各自在数组 a 中的下标。

分析：求取二维数组的最大、最小值与一维数组的类似，只须在扫描数组元素时从行和列两个方向考虑，在记录下标时也需要记录行和列两个方向的下标。

程序代码如下：

```
/*实例 5-24*/
#include <stdio.h>
int main()
{
    int a[4][3], i, j, max, min, rMax, cMax, rMin, cMin;
    printf("Input 12 scores:\n");
    for(i=0; i<4; i++)
        for(j=0; j<3; j++)
            scanf("%d", &a[i][j]);
    max = min = a[0][0];
    rMax = cMax = rMin = cMin = 0;
    for(i=0; i<4; i++)
        for(j=0; j<3; j++)
        {
            if(max<a[i][j])
            {
                max = a[i][j]; rMax = i; cMax = j;
            }
            if(min>a[i][j])
            {
                min = a[i][j]; rMin = i; cMin = j;
            }
        }
    printf("Max is a[%d][%d] = %d\n", rMax, cMax, max);
    printf("Min is a[%d][%d] = %d\n", rMin, cMin, min);
    return 0;
}
```

说明：

(1) 程序中 max 和 min 分别表示最大值和最小值；rMax 和 cMax 分别表示行(row)最大值的下标和列(column)最大值的下标；rMin 和 cMin 分别表示行最小值的下标和列最小值的下标，增加可读性。

(2) 语句“rMax=cMax=rMin=cMin=0;”使用连续赋值方式，同时对 4 个变量赋值。

程序运行结果如图 5-45 所示。

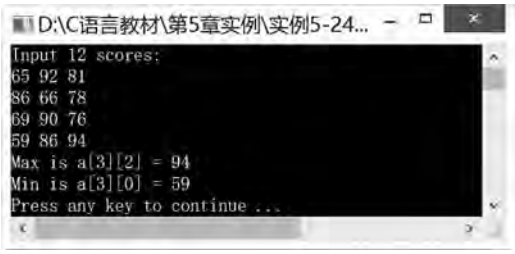

图 5-45　求二维数组的最大、最小值实例

5.6.4 其他

在处理批量数据时，数组还常被用于解决如下问题。

【实例 5-25】 任意输入一个字符串(少于 20 个字符)，将其按逆序存储后输出。

分析： 输入一个字符串将其存储在字符数组 str 中，将字符串中的第 1 个字符和最后 1 个字符(字符串结束符'\0'之前的字符)交换位置，再将字符串的第 2 个字符和倒数第 2 个字符交换位置，以此类推，直到最中间的两个字符交换位置后即可得到与原字符串逆序存储的字符串。图 5-46 描述了一个示例字符串的逆序存储的交换过程(交换 3 次后到达中间位置)。

程序代码如下：

```
/* 实例 5-25 */
#include <stdio.h>
#include <string.h>
int main()
{
    char s[20], i, nLen, temp;
    printf("Input a string:\n");
    scanf("%s", s);
    nLen = strlen(s);
    for(i = 0; i < nLen/2; i++)
    {
        /* 交换 s[i]和 s[nLen - i - 1]的值 */
        temp = s[i];
        s[i] = s[nLen - i - 1];
        s[nLen - i - 1] = temp;
    }
    printf("The inversed string is:\n%s\n", s);
    return 0;
}
```

数组元素	原始取值	第一次交换后	第二次交换后	第三次交换后
str[0]	P	m	m	m
str[1]	r	r	a	a
str[2]	o	o	o	r
str[3]	g	g	g	g
str[4]	r	r	r	o
str[5]	a	a	r	r
str[6]	m	P	P	P

图 5-46 示例字符串逆序存储的交换过程

说明：

(1) 交换第 i 个元素和倒数第 i 个元素时，它们的下标分别为 i 和 nLen－i－1。

(2) 整个交换过程只须进行到中间位置即可，因此 for 语句中 i < nLen/2 不能写为 i < nLen。

程序运行结果如图 5-47 所示。

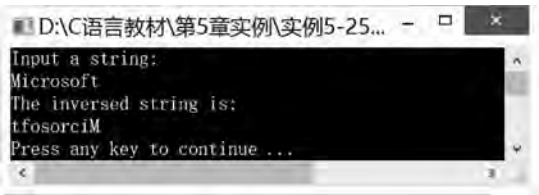

图 5-47 逆序存储字符串

【实例 5-26】 输入一个字符串(少于 20 个字符)，删除其中的数字字符并将删除后的字符串输出。

分析：输入一个字符串并将其存储在字符数组 str 中，依次扫描每个字符数组中的元素，直到字符串结束符'\0'为止，在扫描的同时检测当前元素是否为数字字符，若是则删除该字符，否则继续扫描并检测下一个元素。其中删除一个数字字符的过程为：将当前元素后面的所有元素都向前移动一个位置，此时当前元素会被其后面的第 1 个元素所覆盖，而其后面其他元素的相对位置均未发生变化，具体示例如图 5-49 所示。

程序代码如下：

```
/*实例 5-26*/
#include <stdio.h>
int main()
{
    char str[20];
    int i, j;
    printf("Input a string:\n");
    gets(str);
    for(i=0; str[i] != '\0';)
    {
        if(str[i] >= '0' && str[i] <= '9')
        {
            /*当前元素是数字字符时删除该元素*/
            for(j=i; str[j] != '\0'; j++)
                str[j] = str[j+1];
        }
        else
            i++;        /*当前元素不是数字字符时才继续检测下一个元素*/
    }
    printf("The new string is:\n");
    puts(str);
    return 0;
}
```

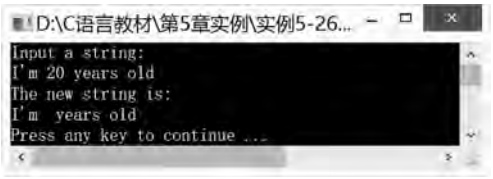

图 5-48　删除字符串中的数字字符

原始字符数组	P	3	r	o	g	r	a	m	'\0'
删除字符'3'后	P	r	o	g	r	a	m	'\0'	'\0'

图 5-49　删除数字字符示例

说明：

(1) 判断数字字符时，语句“if(str[i] >= '0' && str[i] <= '9')”用于表示'0'和'9'之间的字符，与 if(str[i] >=0&& str[i] <=9)不同，该语句用于表示数值范围为 0～9 的数。

(2) 循环语句“for(i=0; str[i]!='\0';)”中的 i++被写到其中的 else 语句后，表示对字符检测过程中，不是每次都要发生下标递增操作。

程序运行结果如图 5-48 所示。

【实例 5-27】 输入一个以 Enter 键结束的字符串(少于 80 个字符)，统计并输出其中数字、英文字符和其他字符的个数。

分析：定义 3 个起计数器作用的变量 digit、letter 和 other，分别用于存储数字、英文字符和其他字符的个数，并将它们的值初始化为 0。输入一个字符串将其存储在字符数组 str 中，依次扫描每个字符数组中的元素，在扫描的同时检测当前元素是数字还是英文字符，若两者都不是则认定为其他字符。若为数字，则 digit 的值加 1；若为英文字符则 letter 的值加 1；否则 other 的值加 1，一直扫描和检测到字符串结束符'\0'为止。

程序代码如下：

```
/* 实例 5-27 */
#include <stdio.h>
int main()
{
    char str[80];
    int i, digit = 0, letter = 0, other = 0;
    printf("Please input a string:\n");
    gets(str);
    for(i = 0; str[i] != '\0'; i++)
    {
        if(str[i] >= '0' && str[i] <= '9')
            digit++;
        else if((str[i] >= 'a' && str[i] <= 'z') ||(str[i] >= 'A' && str[i] <= 'Z'))
            letter++;
        else
            other++;
    }
    printf("The string has %d digits, %d letters and %d others\n", digit, letter, other);
    return 0;
}
```

程序运行结果如图 5-50 所示。

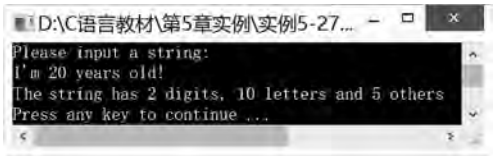

图 5-50　统计字符串中的特征字符

5.7　数组应用实例

数组在批量数据处理中有很重要的应用，本节通过学生成绩管理系统中的成绩处理实例对此进行示范。

【实例 5-28】 输入批量学生成绩数据（假设均为正整数且数量不超过 100 个）到数组中，并提供菜单选项能选择并完成以下操作。

（1）输入新成绩。

（2）求出最高、最低成绩。

（3）计算总成绩、平均成绩。

(4) 统计不及格的成绩个数。

(5) 对成绩进行排序。

分析：可通过搭建相应的菜单，选择并进入相应的操作步骤，而对于具体的步骤中，可参考5.6节中的代码进行程序完善。

程序代码如下：

```
/*实例5-28*/
#include<stdio.h>
int main()
{
    int a[100], nSel;
    while(1)
    {
        printf("              学生成绩管理系统              \n");
        printf("*****************菜单************\n");
        printf("*          1.输入新成绩                    *\n");
        printf("*          2.求最高、最低成绩              *\n");
        printf("*          3.计算总成绩、平均成绩          *\n");
        printf("*          4.统计不及格的成绩个数          *\n");
        printf("*          5.对成绩进行排序                *\n");
        printf("*          0.退出                          *\n");
        printf("***********************************\n");
        printf("请输入您的操作选择[0-5]:");
        scanf("%d", &nSel);
        if(nSel == 0)
        {
            printf("程序即将退出,谢谢使用!\n");
            return 0;
        }
        switch(nSel)
        {
        case 1:
            /*
            输入新成绩代码
            */
            break;
        case 2:
            /*
            求最高、最低成绩代码
            */
            break;
        case 3:
            /*
            计算总成绩、平均成绩代码
            */
            break;
        case 4:
            /*
            统计不及格的成绩个数代码
            */
```

```
            break;
        case 5:
            /*
            对成绩进行排序代码
            */
            break;
        default:
            printf("输入有误,请重新输入\n");
            break;
        }
    }
    return 0;
}
```

说明：完善以上所有注释处的功能后，程序会显得很复杂，在学习第 6 章后，可以将本程序中的大部分功能以函数形式进行实现，实现函数封装后的代码结构会更为清晰。

程序运行结果如图 5-51 所示。

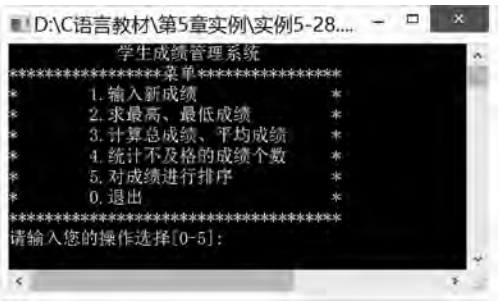

图 5-51　学生成绩管理系统菜单

本 章 小 结

数组是 C 语言程序设计中最常用的数据类型(或结构)。根据定义时的类型不同，数组可分为数值数组(整型数组，实型数组)、字符数组、指针数组以及结构数组等。

数组用于存储多个同类型的数据，常用于矩阵(一维、二维或多维)、字符(串)、集合等问题中的数据存储。

定义数组时，由类型说明符、数组名称、数组长度(数组元素个数)三部分组成。数组长度中“[]”的个数表示数组的维数，根据维数的不同，可将数组分为一维数组，二维数组以及多维数组。“[]”中的数值则表示当前维的数组长度。

已定义的数组通常会进行数组赋值、数组引用等操作。数组赋值可以在数组定义时进行初始化完成，也可通过输入函数或赋值语句进行。数组引用可以通过数组名称和下标的形式来描述所要引用的数组元素，且数组的元素引用通常与循环语句结合使用。

使用指针可以灵活访问数组中的元素。

使用数组存储数据，能方便、快速地解决许多实际问题，常见的如查找、排序、计数、统计等。

习　　题

1. 输入10个整数到数组a中,求出其中的最大值并将此最大值与数组的第1个元素进行交换,并将交换后的数组元素依次输出。

2. 输入一个2×8的整型数组a,对数组a中的每行元素分别做升序排序,并输出排序后的矩阵的值。

3. 打印如下所示的杨辉三角形(要求打印到第10行)。

```
1
1  1
1  2  1
1  3  3   1
1  4  6   4   1
1  5  10  10  5  1
⋮
```

4. 存储5×5的矩阵到二维数组a,并按如下所示的格式输出该矩阵(要求使用for循环语句)。

```
1    1    1    1    0
1    1    1    0   -1
1    1    0   -1   -1
1    0   -1   -1   -1
0   -1   -1   -1   -1
```

5. 按递增顺序输入10个整数到数组a,再输入一个整数x,并将x插入数组a中,且使得a中的元素仍为递增顺序,再将插入数据后的数组元素依次输出。

6. 输入以Enter键结束的字符串(少于20个字符)到字符数组s1,将s1的全部字符(到'\0'为止,包括'\0')复制到字符数组s2中(要求不用strcpy()函数)。

7. 任意输入一个字符串(少于10个字符),判断该字符串是否为回文("回文"即顺读和反读内容均相同的字符串,如121、abccba、X等)。

8. 将4×4矩阵的每行元素均除以该行上的主对角元素,并输出结果矩阵。

9. 输入10个学生的成绩到数组a,统计并输出优秀(大于或等于85)、及格(60～84)和不及格(小于60)的学生人数,并将各区间的人数输出。

10. 使用字符数组操作,根据输入的密码字符数组s(少于80个字符),完成电文破译工作,并输出原文和密码,具体要求如下。

(1) 电文字符a和密码字符b之间的加密规律为第1个字母变为第26个字母,第i个字母变为第(26－i＋1)个字母。如'A'变为'Z','B'变为'Y','C'变为'X',以此类推。

(2) 非字母字符保持不变。