第5章　静 态 数 组

数组的存在提高了大规模数据处理的编程效率。例如，假设一个学校有 3 万个学生，现在我们需要分析和统计这些学生的成绩。如果不采用数组，那么我们可能需要定义 3 万个整数或浮点数类型的变量用来存储这些成绩。由这 3 万个变量的名称所形成的代码量可是一个不小的数字。如果采用数组，我们只需要一个数组变量就可以存储这些成绩，而且我们通过数组元素就可以获取每个学生的成绩。数组是由具有相同数据类型的元素按顺序排列组成的。数组元素在内存上占据连续的内存地址空间。每个数组元素相当于一个变量。数组元素的个数在数组定义时指定，而且不能改变。数组元素由数组变量名和数组下标指定，其中数组下标是大于或等于 0 的整数。使用数组需要注意数组的下标不能超过指定的范围。另外，还有下面两个注意事项。

> 注意事项
>
> （1）函数的返回类型不能是数组类型。
>
> （2）如果函数某个参数的数据类型是数组类型，那么这个参数的数据类型在编译时自动会被转换为相应的指针类型。

从严格上讲，数组分为静态数组和动态数组。静态数组在定义静态数组变量时直接就可以获得相应内存的空间，静态数组的内存空间由系统自动回收。动态数组则需要通过 malloc 等函数进行动态分配；在动态数组使用完毕之后，需要通过显式调用 free 等函数释放所占用的内存。系统不会自动回收动态数组所占用的内存。如果不回收动态数组所占用的内存，那么就称程序含有内存泄露错误。操作系统给每个程序的内存分配了不同的区域。静态数组通常与函数栈占用的是相同区域的内存。动态数组占用的内存不在这个区域。动态数组内存所在的区域称为堆。因此，如果静态数组占用的内存空间越大，那么通常就会使函数调用深度越小，因为函数调用深度越大，通常所需要的函数栈空间也会越大。如果需要较大的数组空间，通常采用动态数组。动态数组将在第 7 章进行讲解。本章主要介绍静态数组。

> 注意事项
>
> （1）静态数组在各类 C 语言程序相关文档中又被简称为数组。因此，在很多场合下，所谓的数组指的就是静态数组。在本章后续的内容中，基本上也直接用数组来代替静态数组。
>
> （2）这里的静态数组是相对于动态数组而言的。因此，在静态数组中“静态”的含义与在静态变量中“静态”的含义不同。不过，静态数组变量同样可以是静态变量，也可以不是静态变量。只有被关键字 static 修饰的静态数组变量才是静态变量，通常称为具有静态属性的静态数组变量。不被关键字 static 修饰的静态数组变量不是静态变量，通常称为不具有静态属性的静态数组变量。

5.1　一 维 数 组

本节介绍的一维数组实际上是一种静态数组。一旦定义了一维数组变量，系统就会直接分配相应的数组内存空间。不带初始化的单个一维数组变量定义格式如下：

数组元素的数据类型 变量名[*数组元素的个数*];

其中，数组元素的个数必须是一个大于 0 的整数类型常数，不能是变量，甚至连具有 const 属性的变量也不可以。下面给出一些合法的数组变量定义示例：

```
   int a[3];                                                    // 1
   int b[3];                                                    // 2
#define D_N 10                                                  // 3
   int c[D_N];                                                  // 4
```

下面给出一些不合法的数组变量定义示例：

```
   int n=10;                                                    // 1
   int a[n]; // 数组元素的个数不能是变量                          // 2
   int b[0]; // 数组元素的个数必须大于 0                          // 3
   const int DC_n = 10;                                         // 4
   int c[DC_n]; // 数组元素的个数不能是具有 const 属性的变量      // 5
   int d(10); // 界定数组元素个数的括号应当是方括号，不能是圆括号 // 6
   int e{10}; // 界定数组元素个数的括号应当是方括号，不能是大括号 // 7
```

如果需要在一条语句中定义多个一维数组变量，相邻的变量之间用逗号分开。例如：

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

这时，这些数组的元素数据类型完全相同。如果需要定义不同元素数据类型的数组变量，那么应当采用多条语句分别定义这些数组变量。

数组元素在使用上有点类似变量，即可以获取数组元素的值或者给数组元素赋值。使用数组元素也称为访问数组元素。访问一维数组元素的格式如下：

数组变量名[*数组元素下标*]

其中，数组元素下标是一个整数表达式，其有效范围是从 0 开始到（数组元素的个数−1）。如果数组元素下标的值不在其有效范围内，则称为数组元素下标越界。例如，假设定义了数组变量“int a[3]”，那么有效的数组元素分别是 a[0]、a[1]和 a[2]。这三个元素相当于三个整数类型的变量。这里给出无效的数组元素示例: a[−2]、a[−1]、a[3]和 a[4]。在正常情况下，都不应当使用这些无效的数组元素。

注意事项：

（1）数组元素下标是从 0 开始的。

（2）在编写程序时，在正常情况下应当保证不要出现数组元素下标越界。如果出现数组元素下标越界并且因此修改了非预期的内存空间，就会出一些非预期的错误。在 C 语言中，编译和运行都有可能不检查数组元素下标是否越界。因为这种错误通常比较难以调试，所以在编写程序使用数组元素下标时要特别小心。

下面给出数组元素的使用示例：

```
int a[3];                                                    // 1
a[0] = 1;                                                    // 2
a[1] = 2;                                                    // 3
a[2] = a[0]+a[1];                                            // 4
printf("a[2]=%d, *(a+2)=%d。\n", a[2], *(a+2));              // 5
```

通过上面的示例可以看出每个数组元素相当于一个变量，可以获取它的值，也可以给它赋值。上面示例第 5 行“*(a+2)”给出另外一种访问一维数组元素方式，其具体格式为：

*(*数组变量名*+*整数表达式*)

其中，整数表达式的值就是数组元素下标。数组变量名与该整数表达式相加，得到的是相应数组元素的地址；然后，通过*运算符就可以访问该数组元素。在上面示例第 5 行中，“*(a+2)”与“a[2]”实际上是等价的，其中“a+2”是数组元素 a[2]的地址。因此，在上面示例第 4 行的代码“a[2] = a[0]+a[1];”也可以写成

```
*(a+2) = a[0]+a[1];
```

这两者在功能上是等价的。

注意事项

在 C 语言中，地址是以字节为单位进行计数的，但每个数组元素占用的内存大小不一定就是一字节。因此，虽然数组元素在内存当中是连续的，但是相邻数组元素之间地址不一定相差 1 字节。例如，设定义了数组“int a[3]”，并且设 sizeof(int)=4，那么 a[2]的地址比 a[1]的地址大 4 字节。

下面给出示例进一步介绍数组元素的地址。

```
int a[3];                                                    // 1
printf("&(a[0])=%p, &(a[1])=%p, &(a[2])=%p。\n",             // 2
   &(a[0]), &(a[1]), &(a[2]));                               // 3
printf("a=%p, a+1=%p, a+2=%p。\n", a, a+1, a+2);             // 4
printf("((a+2)-(a+1))=%d。\n", ((a+2)-(a+1)));               // 5
printf("((int)(a+2)-(int)(a+1))=%d。\n",                     // 6
   ((int)(a+2)-(int)(a+1)));                                 // 7
```

下面给出这个示例的一个运行的结果。在不同的运行环境下，具体结果可能会有所不同。

```
&(a[0])=0012FF54, &(a[1])=0012FF58, &(a[2])=0012FF5C。
a=0012FF54, a+1=0012FF58, a+2=0012FF5C。
```

```
((a+2)-(a+1))=1。
((int)(a+2)-(int)(a+1))=4。
```

通过上面的示例，我们可以看到通过取地址运算符&也可以获取数组元素的地址。“&(a[0])”与“a”获取的地址是相同的，“&(a[1])”与“a+1”获取的地址是相同的，“&(a[2])”与“a+2”获取的地址是相同的。在上面运行环境中，sizeof(int)=4。如果把 a+1 和 a 的地址当作整数，那么它们在数值上相差 4=0x0012FF58−0x0012FF54。如果把 a+2 和 a+1 的地址当作整数，那么它们在数值上相差 4=0x0012FF5C−0x0012FF58。从运行结果“((a+2)−(a+1))=1”上，我们可以看出数组元素的地址之间可以做减法运算，但它们不是转换为整数进行运算，运算结果是数组元素下标之差。如果先将这两个地址转换为整数，再进行减法运算，那么应当写成“((int)(a+2)−(int)(a+1))”。从上面运算结果上看，其结果等于 4。

> 注意事项
>
> 在 C 语言中，在定义数组变量之后，就不能给数组变量赋值。例如下面的代码是非法的。

```
int a[3];                                                    // 1
int b[3];                                                    // 2
a = b;      // 非法的赋值操作                                 // 3
```

我们还可以通过 sizeof 运算获取数组变量占用的内存空间大小、数组元素占用的内存空间大小以及数组元素个数。下面给出代码示例。

```
int a[3];                                                    // 1
printf("sizeof(a)=%d。\n", (int)(sizeof(a)));                // 2
printf("sizeof(a[0])=%d。\n", (int)(sizeof(a[0])));          // 3
printf("sizeof(a)/sizeof(a[0])=%d。\n",                      // 4
   (int)(sizeof(a)/sizeof(a[0])));                           // 5
printf("sizeof(a)/sizeof(int)=%d。\n",                       // 6
   (int)(sizeof(a)/sizeof(int)));                            // 7
```

下面给出这个示例的一个运行的结果。在不同的运行环境下，具体结果可能会有所不同。

```
sizeof(a)=12。
sizeof(a[0])=4。
sizeof(a)/sizeof(a[0])=3。
sizeof(a)/sizeof(int)=3。
```

从上面的运行结果可以看出，数组变量占用的内存空间大小=数组元素占用的内存空间大小×数组元素个数，即数组变量占用的内存空间是其数组元素占用的内存空间的总和。反过来，数组元素个数=sizeof(数组变量)/sizeof(任意一个数组元素)=sizeof(数组变量)/sizeof(数组元素的数据类型)。

在定义数组变量的同时还可以给数组元素赋初值。这又称为通过初始化定义数组变量。这有两种形式，其中第一种是不显式地指定数组元素个数的初始化形式，具体格式如下：

```
数组元素的数据类型 变量名[ ]={由各个数组元素的初值表达式组成的列表};
```

其中，初值表达式的个数不能小于 1，初值表达式一定要与数组元素的数据类型相匹配，在初值表达式中可以包含变量，在相邻的初值表达式之间用逗号分开，最终数组元素个数就是这些初值表达式的个数。下面给出一些定义示例：

```
int a[ ] = {1, 2}; // 结果: 元素个数=2; a[0]=1; a[1]=2                    // 1
int n = 10;                                                              // 2
int b[ ] = {n, n*2, n*3}; // 结果: 元素个数=3; b[0]=10; b[1]=20; b[2]=30   // 3
```

第二种通过初始化定义数组变量的形式是**指定数组元素个数的初始化形式**，具体格式如下：

数组元素的数据类型 变量名[*数组元素的个数*]={*由各个数组元素的初值表达式组成的列表*};

第二种形式比第一种形式多了指定数组元素个数。这两种形式在其他部分的格式上都一样。数组元素的个数必须是一个大于 0 的整数类型常数，不能是变量，甚至连具有 const 属性的变量也不可以。在上面定义中，指定的数组元素个数一定要大于或等于初值表达式的个数，并且初值表达式的个数不能小于 1；否则，通不过编译，即会出现编译错误。如果指定的数组元素个数大于初值表达式的个数，那么数组最后剩余的元素的值都会初始化为 0 或相当于 0 的值。下面给出一些合法的定义示例：

```
int a[2] = {1, 2}; // 结果: a[0]=1; a[1]=2                    // 1
int b[3] = {10, 20}; // 结果: b[0]=10; b[1]=20; b[2]=0        // 2
int c[3] = {0}; // 结果: c[0]=0; c[1]=0; c[2]=0               // 3
```

下面给出一些不合法的定义示例：

```
int a[2] = { }; // 初值表达式的个数为 0                       // 1
int b[2] = {10, 20, 30}; // 数组元素个数小于初值表达式的个数   // 2
```

例程 5-1　数组作为函数形式参数被自动转换为指针。

例程功能描述：验证数组作为函数形式参数被自动转换为指针。

例程解题思路：我们先定义一个函数，它包含一个形式参数。我们让该参数的数据类型是数组类型。然后，我们在主函数中定义一个数组。我们让它们在形式上具有相同的数组类型，并在主函数中调用前面的函数，其实际参数就是在主函数中定义的那个数组。要区分数组与指针，我们只要通过运算 sizeof 获取它们占用的内存空间大小就可以了。如果是完全相同类型的数组，它们占用的内存空间大小应当完全相同。

下面给出按照上面思路编写的代码。例程代码由源程序代码文件“C_ArrayToPointer.c”组成，具体的程序代码如下。

```
// 文件名: C_ArrayToPointer.c；开发者: 雍俊海                    行号
#include <stdio.h>                                              // 1
#include <stdlib.h>                                             // 2
                                                                // 3
void gb_arrayShowSize(int a[10])                                // 4
```

```
{                                                                    // 5
   printf("在被调用函数中，sizeof(a)=%d, sizeof(a[0])=%d。\n",          // 6
      (int)(sizeof(a)), (int)(sizeof(a[0])));                        // 7
} // 函数 gb_arrayShowSize 结束                                       // 8
                                                                     // 9
int main(int argc, char* args[ ])                                    // 10
{                                                                    // 11
   int a[10];                                                        // 12
   printf("在主函数中，sizeof(a)=%d, sizeof(a[0])=%d。\n",              // 13
      (int)(sizeof(a)), (int)(sizeof(a[0])));                        // 14
   gb_arrayShowSize(a);                                              // 15
   system("pause"); // 暂停住控制台窗口                                // 16
   return 0; // 返回 0 表明程序运行成功                                // 17
} // main 函数结束                                                    // 18
```

可以对上面的代码进行编译、链接和运行。下面给出一个运行的结果示例。

```
在主函数中，sizeof(a)=40, sizeof(a[0])=4。
在被调用函数中，sizeof(a)=8, sizeof(a[0])=4。
请按任意键继续...
```

我们来分析上面的程序代码和运行结果。在上面代码第 12 行定义了数组变量 a。因为上面输出“sizeof(a[0])=4”，所以我们知道在该运行环境下 int 类型变量占用内存的字节数是 4，数组 a 的每个元素占用内存的字节数就是 4。因为数组 a 的元素个数是 10，所以数组 a 占用内存的字节数是 40=4×10。因此，上面输出 sizeof(a)=40。在函数 gb_arrayShowSize 的首部，虽然形式参数 a 的定义严格按照数组的定义格式，但变量 a 的实际数据类型是“int *”类型的指针，因为我们发现相应的输出是“sizeof(a)=8”。如果在函数 gb_arrayShowSize 中的变量 a 是数组类型，那么按照前面的分析，它占用的内存空间大小应当是 40。按照 C 语言标准，上面第 4 行代码与下面代码从编译结果的角度上看实际上是完全一样的。

```
void gb_arrayShowSize(int* a)
```

这里介绍上面代码第 15 行代码函数调用的参数传递过程。如图 5-1 所示，主函数的局部变量 a 与函数 gb_arrayShowSize 的形式参数变量 a 各自拥有不同的内存空间与不同的数据类型。因为函数 gb_arrayShowSize 的形式参数变量 a 的数据类型是指针，所以在进行参数传递时，将主函数的局部变量 a 的内存地址赋值给函数 gb_arrayShowSize 的形式参数变

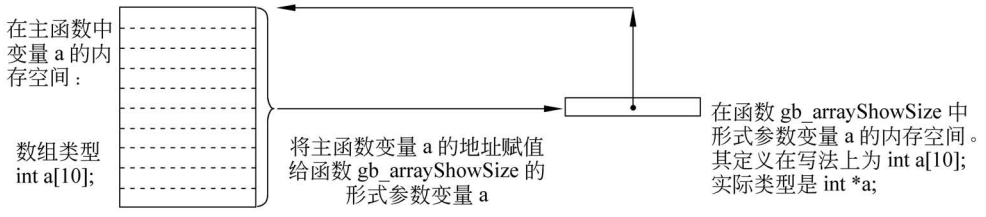

图 5-1　从数组类型实际参数到指针类型形式参数的函数调用参数传递过程示意图

量 a。这样，在函数 gb_arrayShowSize 内部，指针 a 指向了主函数变量 a 的内存空间。于是，在函数 gb_arrayShowSize 内部，我们也可以通过“a[下标]”或“*(a+下标)”等形式访问在主函数中数组 a 的数组元素。

例程 5-2　数组元素下标越界。

例程功能描述：对于含有变量的数组下标表达式，验证编译与运行都有可能不检查该下标的值是否会越界。

例程解题思路：我们定义一个数组，并访问下标越界的内存空间。为了防止因此而出现**程序崩溃**，我们让这个内存空间是一个已知变量的内存空间。下面给出按照这个思路编写的代码。例程代码由源程序代码文件“C_ArrayOutOfRangeMain.c”组成，具体的程序代码如下。

```
// 文件名：C_ArrayOutOfRangeMain.c；开发者：雍俊海                              行号
#include <stdio.h>                                                          // 1
#include <stdlib.h>                                                         // 2
                                                                            // 3
int main(int argc, char* args[ ])                                           // 4
{                                                                           // 5
    int a = 0;                                                              // 6
    int b[ ] = {1, 2, 3, 4};                                                // 7
    int n;                                                                  // 8
    n = &a - b;                                                             // 9
    printf("sizeof(int)=%d。\n", (int)(sizeof(int)));                       // 10
    printf("&a=%p, b=%p, &(b[%d])=%p。\n", &a, b, n, &(b[n]));              // 11
    printf("(int)(&a)-(int)b=%d。\n", (int)(&a)-(int)b);                    // 12
    printf("a=%d。\n", a);                                                  // 13
    b[n] = 1000;                                                            // 14
    printf("b[%d]=%d, b[0]=%d, b[3]=%d。\n", n, b[n], b[0], b[3]);          // 15
    printf("a=%d。\n", a);                                                  // 16
    system("pause"); // 暂停住控制台窗口                                    // 17
    return 0; // 返回 0 表明程序运行成功                                    // 18
} // main 函数结束                                                         // 19
```

可以在调试（Debug）模式下对上面的代码进行编译、链接和运行。下面给出在该模式下的一个**运行的结果**示例。

```
sizeof(int)=4。
&a=000000DF2475F4D4, b=000000DF2475F4F8, &(b[-9])=000000DF2475F4D4。
(int)(&a)-(int)b=-36。
a=0。
b[-9]=1000, b[0]=1, b[3]=4。
a=1000。
请按任意键继续...
```

可以在发布（Release）模式下对上面的代码进行编译、链接和运行。下面给出在该模式下的一个**运行的结果**示例。

```
sizeof(int)=4。
&a=000000CD6DCFFCC0, b=000000CD6DCFFCC8, &(b[-2])=000000CD6DCFFCC0。
(int)(&a)-(int)b=-8。
a=0。
b[-2]=1000, b[0]=1, b[3]=4。
a=1000。
请按任意键继续...
```

我们来分析上面的程序代码和运行结果。上面运行结果第 1 行“sizeof(int)=4”表明在这个运行环境下每个整数类型变量占用的内存空间大小是 4 字节。从上面调试模式和发布模式运行结果上看，运算“n = &a−b”的结果并不相同，前者 n=−9，后者 n=−2。因此，从运行结果上看，我们可以推出这样的结论：不能通过变量 **a** 和 **b** 在代码上的相对位置推断出变量 **a** 和 **b** 的相对地址关系。上面第 6~8 行代码依次定义了变量 a、b 和 n，这三个变量的内存地址也不一定会满足递增或递减的关系。在调试模式下，编译器需要在内存中插入一些数据从而方便程序调试；在发布模式下，编译器需要进行代码优化，从而提高代码的执行效率。

在调试模式下，运算结果“(int)(&a)−(int)b=−36”表明变量 a 和 b 的地址实际上相差 36 字节。而运算“n = &a−b”执行的不是整数运算，而是整数指针运算，其结果得到的是这两个地址相差多少个整数数组元素。因此，这时，n=(−36)/sizeof(int)=(−36)/4=−9。

同样，在发布模式下，运算结果“(int)(&a)−(int)b=−8”表明变量 b 和 a 的地址实际上相差 8 字节。运算“n = &a−b”的结果是 n=(−8)/sizeof(int)= (−8)/4=−2。

从上面的运行结果，我们还可以看出在第 11 行，变量 a 与 b[n]拥有相同的地址，不管是在调试模式下，还是在发布模式下。在调试模式下，上面结果显示 b[n]在这时实际上是 b[−9]；在发布模式下，上面结果显示 b[n]在这时实际上是 b[−2]。无论是 b[−9]还是 b[−2]，其下标都超出了数组 b 的下标有效范围。从上面的运行结果，我们可以看到程序并没有报错，甚至连警告也没有。这就验证了本例程的结论，对于含有变量的数组下标表达式，编译与运行都有可能不检查该下标的值是否会越界。不过，如果随意让下标越界，有可能会导致程序崩溃等破坏性事件。在上面第 14 行代码中，因为变量 a 与 b[n]拥有相同的地址和相同的数据类型，所以我们可以通过修改 b[n]的值来修改变量 a 的值。结果变量 a 的值被改为 1000。不过，这种通过修改 b[n]的值来修改变量 a 的值的做法是非常危险的，这不是正常的 C 语言程序。

> ✍注意事项✍
>
> 不要通过数组下标越界来修改其他变量的值。因为编译器在将 C 语言代码转换为可执行代码时存在着代码优化，而数组下标越界的行为是编译器的非预期行为，所以有可能会造成一些非常奇怪的非预期结果。在上面例程中，如果我们删去第 15 行的代码，那么第 16 行代码“printf("a=%d。\n", a);”都有可能比第 14 行代码“b[n] = 1000;”先执行，因为编译器会误认为变量 a 与 b[n]在内存空间上是不相关的，即编译器会误认为这两条语句的执行顺序不影响最终结果，而先执行第 16 行代码很有可能会提高代码运行效率。如果这种情况发生，我们会发现非预期的结果：第 16 行代码“printf("a=%d。\n", a);”输出“a=0”。

例程 5-3　给数组元素初始化和赋值并输出结果。

例程功能描述：比较初始化和不初始化数组元素的结果，给数组元素赋值，并输出数组元素的值。

例程解题思路：定义两个数组变量，对其中一个进行初始化，另一个不进行初始化。输出这两个数组的各个元素的值。对这两个数组的元素进行赋值，并输出赋值的结果。

下面给出按照上面思路编写的代码。例程代码由 3 个源程序代码文件“C_ArrayDefineAndShow.h”“C_ArrayDefineAndShow.c”“C_ArrayDefineAndShowMain.c”组成，具体的程序代码如下。

// 文件名：C_ArrayDefineAndShow.h；开发者：雍俊海　　**行号**

```
#ifndef C_ARRAYDEFINEANDSHOW_H                                          // 1
#define C_ARRAYDEFINEANDSHOW_H                                          // 2
                                                                        // 3
extern void gb_arraySetLinear(int a[ ], int n, int start, int step);    // 4
extern void gb_arrayShow(char c, int a[ ], int n);                      // 5
extern void gb_arrayTest( );                                            // 6
                                                                        // 7
#endif                                                                  // 8
```

// 文件名：C_ArrayDefineAndShow.c；开发者：雍俊海　　**行号**

```
#include <stdio.h>                                                      // 1
#include <stdlib.h>                                                     // 2
#include <malloc.h>                                                     // 3
                                                                        // 4
void gb_arraySetLinear(int a[ ], int n, int start, int step)            // 5
{                                                                       // 6
   int i;                                                               // 7
   if ((a==NULL) || (n<=0))                                             // 8
      return;                                                           // 9
   a[0] = start;                                                        // 10
   for (i=1; i<n; i++)                                                  // 11
      a[i] = a[i-1] + step;                                             // 12
} // 函数 gb_arraySetLinear 结束                                        // 13
                                                                        // 14
void gb_arrayShow(char c, int a[ ], int n)                              // 15
{                                                                       // 16
   int i;                                                               // 17
   printf("%c 的地址是 0X%p。\n\t", c, a);                              // 18
   for (i=0; i<n; i++)                                                  // 19
      printf("%c[%d]=%d。", c, i, a[i]);                                // 20
   printf("\n");                                                        // 21
} // 函数 gb_arrayShow 结束                                            // 22
                                                                        // 23
void gb_arrayTest( )                                                    // 24
```

```
(                                                                    // 25
    int a[3] = {0};                                                  // 26
    int b[3];                                                        // 27
                                                                     // 28
    printf("在调用 gb_arraySetLinear 之前:\n");                      // 29
    gb_arrayShow('a', a, 3);                                         // 30
    gb_arrayShow('b', b, 3);                                         // 31
    gb_arraySetLinear(a, 3, 10, 1);                                  // 32
    gb_arraySetLinear(b, 3, 20, 2);                                  // 33
    printf("在调用 gb_arraySetLinear 之后:\n");                      // 34
    gb_arrayShow('a', a, 3);                                         // 35
    gb_arrayShow('b', b, 3);                                         // 36
} // 函数 gb_arrayTest 结束                                          // 37
```

// 文件名: C_ArrayDefineAndShowMain.c；开发者: 雍俊海　　行号

```
#include <stdio.h>                                                   // 1
#include <stdlib.h>                                                  // 2
#include "C_ArrayDefineAndShow.h"                                    // 3
                                                                     // 4
int main(int argc, char* args[ ])                                    // 5
{                                                                    // 6
    gb_arrayTest( );                                                 // 7
    gb_arrayTest( );                                                 // 8
    system("pause"); // 暂停住控制台窗口                             // 9
    return 0; // 返回 0 表明程序运行成功                             // 10
} // main 函数结束                                                  // 11
```

可以在调试（Debug）模式下对上面的代码进行编译、链接和运行。下面给出在该模式下的一个运行的结果示例。

```
在调用 gb_arraySetLinear 之前:
a 的地址是 0X000000C2A38FF958。
       a[0]=0。a[1]=0。a[2]=0。
b 的地址是 0X000000C2A38FF988。
       b[0]=-858993460。b[1]=-858993460。b[2]=-858993460。
在调用 gb_arraySetLinear 之后:
a 的地址是 0X000000C2A38FF958。
       a[0]=10。a[1]=11。a[2]=12。
b 的地址是 0X000000C2A38FF988。
       b[0]=20。b[1]=22。b[2]=24。
在调用 gb_arraySetLinear 之前:
a 的地址是 0X000000C2A38FF958。
       a[0]=0。a[1]=0。a[2]=0。
b 的地址是 0X000000C2A38FF988。
       b[0]=-858993460。b[1]=-858993460。b[2]=-858993460。
在调用 gb_arraySetLinear 之后:
```