第8章 数组

在第 2 章，我们已经使用了一些简单的一维数组。本章我们将深入探讨数组，探索一些更加高级的数组话题如多维数组、数组和指针以及数组的初始化等。

8.1 一维数组

在讨论多维数组之前，我们还需要学习很多关于一维数组的知识。首先让我们学习一个概念，它被许多人认为是 C 语言设计的一个缺陷。但是，这个概念实际上以一种相当优雅的方式把一些完全不同的概念联系在一起的。

8.1.1 数组名

考虑下面这些声明：

```
int  a;
int  b[10];
```

我们把变量 a 称为标量，因为它是个单一的值，这个变量的类型是一个整数。我们把变量 b 称为数组，因为它是一些值的集合。下标和数组名一起使用，用于标识该集合中某个特定的值。例如，b[0]表示数组 b 的第 1 个值，b[4]表示第 5 个值。每个特定值都是一个标量，可以用于任何可以使用标量数据的上下文环境中。

b[4]的类型是整型，但 b 的类型又是什么？它所表示的又是什么？一个合乎逻辑的答案是它表示整个数组，但事实并非如此。在 C 中，在几乎所有使用数组名的表达式中，数组名的值是一个指针常量，也就是数组第 1 个元素的地址。它的类型取决于数组元素的类型：如果它们是 int 类型，那么数组名的类型就是“指向 int 的常量指针”；如果它们是**其他类型**，那么数组名的类型就是“指向**其他类型**的常量指针”。

请不要根据这个事实得出数组和指针是相同的结论。数组具有一些和指针完全不同的特征。例如，数组具有确定数量的元素，而指针只是一个标量值。编译器用数组名来记住这些属性。只有当数组名在表达式中使用时，编译器才会为它产生一个指针常量。

注意这个值是指针常量，而不是指针变量。你不能修改常量的值。你只要稍微回想一下，就

会认为这个限制是合理的：指针常量所指向的是内存中数组的起始位置，如果修改这个指针常量，唯一可行的操作就是把整个数组移动到内存的其他位置。但是，在程序完成链接之后，内存中数组的位置是固定的，所以当程序运行时，再想移动数组就为时已晚了。因此，数组名的值是一个指针常量。

只有在两种场合下，数组名并不用指针常量来表示——就是当数组名作为 sizeof 操作符或单目操作符&的操作数时。sizeof 返回整个数组的长度，而不是指向数组的指针的长度。取一个数组名的地址所产生的是一个指向数组的指针（指向数组的指针在第 8.2.2 节和第 8.2.3 节讨论），而不是一个指向某个指针常量值的指针。

现在考虑下面这个例子：

```
int     a[10];
int     b[10];
int     *c;
...
c = &a[0];
```

表达式&a[0]是一个指向数组第 1 个元素的指针。但那正是数组名本身的值，所以下面这条赋值语句和上面那条赋值语句所执行的任务是完全一样的：

```
c = a;
```

这条赋值语句说明了为什么理解表达式中的数组名的真正含义是非常重要的。如果数组名表示整个数组，这条语句就表示整个数组被复制到一个新的数组。但事实上完全不是这样，实际被赋值的是一个指针的拷贝，c 所指向的是数组的第 1 个元素。因此，像下面这样的表达式：

```
b = a;
```

是非法的。你不能使用赋值符把一个数组的所有元素复制到另一个数组。你必须使用一个循环，每次复制一个元素。

考虑下面这条语句：

```
a = c;
```

c 被声明为一个指针变量，这条语句看上去像是执行某种形式的指针赋值，把 c 的值复制给 a。但这个赋值是非法的：记住！在这个表达式中，a 的值是个常量，不能被修改。

8.1.2 下标引用

在前面声明的上下文环境中，下面这个表达式是什么意思？

```
*( b + 3 )
```

首先，b 的值是一个指向整型的指针，所以 3 这个值根据整型值的长度进行调整。加法运算的结果是另一个指向整型的指针，它所指向的是数组第 1 个元素向后移 3 个整数长度的位置。然后，间接访问操作访问这个新位置，或者取得那里的值（右值），或者把一个新值存储于该处（左值）。

这个过程听上去是不是很熟悉？这是因为它和下标引用的执行过程完全相同。我们现在可以解释第 5 章所提到的一句话：除了优先级之外，下标引用和间接访问完全相同。例如，下面这两个表达式是等同的：

```
array[subscript]
*( array + ( subscript ) )
```

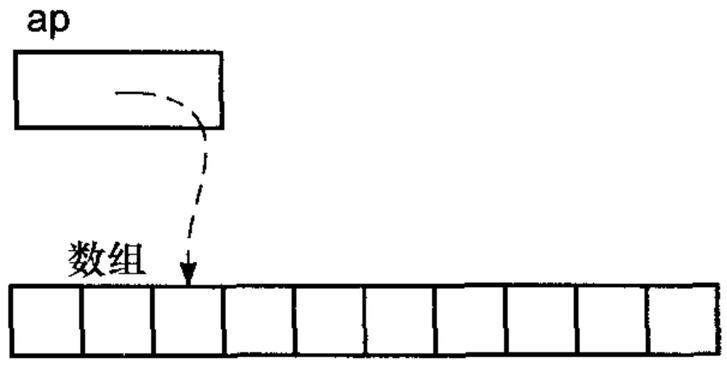
既然你已知道数组名的值只是一个指针常量，你可以证明它们的相等性。在那个下标表达式中，子表达式 subscript 首先进行求值。然后，这个下标值在数组中选择一个特定的元素。在第 2 个表达式中，内层的那个括号保证子表达式 subscript 像前一个表达式那样首先进行求值。经过指针运算，加法运算的结果是一个指向所需元素的指针。然后，对这个指针执行间接访问操作，访问它指向的那个数组元素。

在使用下标引用的地方，你可以使用对等的指针表达式来代替。在使用上面这种形式的指针表达式的地方，你也可以使用下标表达式来代替。

这里有个小例子，可以说明这种相等性。

```
int        array[10];
int        *ap = array + 2;
```

记住，在进行指针加法运算时会对 2 进行调整。运算结果所产生的指针 ap 指向 array[2]，如下所示：



在下面各个涉及 ap 的表达式中，看看你能不能写出使用 array 的对等表达式。

Ap　　这个很容易，你只要阅读它的初始化表达式就能得到答案：array+2。另外，&array[2]也是与它对等的表达式。

ap　　这个也很容易，间接访问跟随指针访问它所指向的位置，也就是 array[2]。你也可以这样写：(array+2)。

ap[0]　　“你不能这样做，ap 不是一个数组！”如果你是这样想的，你就陷入了“其他语言不能这样做”这个惯性思维中了。记住，C 的下标引用和间接访问表达式是一样的。在现在这种情况下，对等的表达式是*(ap+(0))，除去 0 和括号，其结果与前一个表达式相等。因此，它的答案和上一题相同：array[2]。

ap+6　　如果 ap 指向 array[2]，这个加法运算产生的指针所指向的元素是 array[2]向后移动 6 个整数位置的元素。与它对等的表达式是 array+8 或&array[8]。

*ap+6　　小心！这里有两个操作符，哪一个先执行呢？是间接访问。间接访问的结果再与 6 相加，所以这个表达式相当于表达式 array[2]+6。

*(ap+6)　　括号迫使加法运算首先执行，所以我们这次得到的值是 array[8]。注意这里的间接访问操作和下标引用操作的形式是完全一样的。

ap[6]　　把这个下标表达式转换为与其对应的间接访问表达式形式，你会发现它就是我们刚刚完成的那个表达式，所以它们的答案相同。

&ap　　这个表达式是完全合法的，但此时并没有对等的涉及 array 的表达式，因为你无法预测编译器会把 ap 放在相对于 array 的什么位置。

ap[-1]　　怎么又是它？负值的下标！下标引用就是间接访问表达式，你只要把它转换为那种形式并对它进行求值。ap 指向第 3 个元素（就是那个下标值为 2 的元素），所以使用偏移量-1 使我们得到它的前一个元素，也就是 array[1]。

ap[9]　　这个表达式看上去很正常，但实际上却存在问题。它对等的表达式是 array[11]，但问题是这个数组只有 10 个元素。这个下标表达式的结果是一个指针表达式，但它所指向的位置越过了数组的右边界。根据标准，这个表达式是非法的。但是，很少有编译器能够检测到这类错误，所以程序能够顺利地继续运行。但这个表达式到底干了些什么？标准表示它的行为是未定义的，但在绝大多数机器上，它将访问那个碰巧存储于数组最后一个元素后面第 2 个位置的值。你有时可以通过请求编译器产生程序的汇编语言版本并对它进行检查，从而推断出这个值是什么，但你并没有统一的办法预测存储在这个地方的到底是哪个值。因此，这个表达式将访问（或者，如果作为左值，将修改）某个任意变量的值。这个结果估计不是你所希望的。

最后两个例子显示了为什么下标检查在 C 中是一项困难的任务。标准并未提出这项要求。最早的 C 编译器并不检查下标，而最新的编译器依然不对它进行检查。这项任务之所以很困难，是因为下标引用可以作用于任意的指针，而不仅仅是数组名。作用于指针的下标引用的有效性既依赖于该指针当时恰好指向什么内容，也依赖于下标的值。

结果，C 的下标检查所涉及的开销比你刚开始想象的要多。编译器必须在程序中插入指令，证实下标表达式的结果所引用的元素和指针表达式所指向的元素属于同一个数组。这个比较操作需要程序中所有数组的位置和长度方面的信息，这将占用一些空间。当程序运行时，这些信息必须进行更新，以反映自动和动态分配的数组，这又将占用一定的时间。因此，即使是那些提供了下标检查的编译器通常也会提供一个开关，允许你去掉下标检查。

这里有一个有趣的，但同时也有些神秘和离题的例子。假定下面表达式所处的上下文环境和前面的相同，它的意思是什么呢？

```
2[array]
```

它的答案可能会令你大吃一惊：它是合法的。把它转换成对等的间接访问表达式，你就会发现它的有效性：

```
*( 2 + ( array ) )
```

内层的那个括号是冗余的，我们可以把它去掉。同时，加法运算的两个操作数是可以交换位置的，所以这个表达式和下面这个表达式是完全一样的

```
*( array + 2 )
```

也就是说，最初那个看上去颇为古怪的表达式与 array[2]是相等的。

这个诡异技巧之所以可行，缘于 C 实现下标的方法。对编译器来说，这两种形式并无差别。但是，你绝不应该编写 2[array]，因为它会大大影响程序的可读性。

8.1.3　指针与下标

如果你可以互换地使用指针表达式和下标表达式，那么你应该使用哪一个呢？和往常一样，这里并没有一个简明答案。对于绝大多数人而言，下标更容易理解，尤其是在多维数组中。所以，在可读性方面，下标有一定的优势。但在另一方面，这个选择可能会影响运行时效率。

假定这两种方法都是正确的，下标绝不会比指针更有效率，但指针有时会比下标更有效率。

为了理解这个效率问题，让我们来研究两个循环，它们用于执行相同的任务。首先，我们使用下标方案将数组中的所有元素都设置为 0。

```
int       array[10], a;
for ( a = 0; a < 10; a +=1 )
          array[a] = 0;
```

为了对下标表达式求值，编译器在程序中插入指令，取得 a 的值，并把它与整型的长度（也就是 4）相乘。这个乘法需要花费一定的时间和空间。

现在让我们再来看看下面这个循环，它所执行的任务和前面的循环完全一样。

```
int     array[10], *ap;
for( ap = array; ap < array + 10; ap++ )
         *ap = 0;
```

尽管这里并不存在下标，但还是存在乘法运算。请仔细观察一下，看看你能不能找到它。

现在，这个乘法运算出现在 for 语句的调整部分。1 这个值必须与整型的长度相乘，然后再与指针相加。但这里存在一个重大区别：循环每次执行时，执行乘法运算的都是两个相同的数（1 和 4）。结果，这个乘法只在编译时执行一次——程序现在包含了一条指令，把 4 与指针相加。程序在运行时并不执行乘法运算。

这个例子说明了指针比下标更有效率的场合——当你在数组中 1 次 1 步（或某个固定的数字）地移动时，与固定数字相乘的运算在编译时完成，所以在运行时所需的指令就少一些。在绝大多数机器上，程序将会更小一些、更快一些。

现在考虑下面两个代码段：

```
a = get_value();
array[a] = 0;
```

```
a = get_value();
*( array + a ) = 0;
```

两边的语句所产生的代码并无区别。a 可能是任何值，在运行时方知。所以两种方案都需要乘法指令，用于对 a 的值进行调整。这个例子说明了指针和下标的效率完全相同的场合。

8.1.4　指针的效率

前面我曾说过，指针有时比下标更有效率，**前提是它们被正确地使用**。就像电视上说的那样，你的结果可能不同，这取决于你的编译器和机器。然而，程序的效率主要取决于你所编写的代码。和使用下标一样，使用指针也很容易写出质量低劣的代码。事实上，这个可能性或许更大。

为了说明一些拙劣的技巧和一些良好的技巧，让我们看一个简单的函数，它使用下标把一个数组的内容复制到另一个数组。我们将分析这个函数所产生的汇编代码，我们选择了一种特定的编译器，它在一台使用 Motorola M68000 家族处理器的计算机上运行。我们接着将以不同的使用指针的方法修改这个函数，看看每次修改对结果目标代码有什么影响。

在开始这个例子之前，要注意两件事情。首先，你编写程序的方法不仅影响程序的运行时效率，而且影响它的可读性。不要为了效率上的细微差别而牺牲可读性，这点非常重要。对于这个话题，我后面还要深入探讨。

其次，这里所显示的汇编语言显然是 68000 处理器家族特有的。其他机器（和其他编译器）可能会把程序翻译成其他样子。如果你需要在你的环境里取得最高效率，你可以在你的机器（和编译器）上试验我在这里所使用的各种方法，看看各种不同的源代码惯用法是如何实现的。

首先，下面的声明用于所有版本的函数。

```
#define SIZE    50
int     x[SIZE];
int     y[SIZE];
int     i;
int     *p1, *p2;
```

这是函数的下标版本。

```
void
try1()
{
        for(i = 0; i < SIZE; i++)
                x[i] = y[i];
}
```

这个版本看上去相当直截了当。编译器产生下列汇编语言代码。

```
00000004   42b90000 0000      _try1:    clrl    _i
0000000a   6028                         jra     L20
0000000c   20390000 0000      L20001:   movl    _i,d0
00000012   e580                         asll    #2,d0
00000014   207c0000 0000                movl    #_y,a0
0000001a   22390000 0000                movl    _i,d1
00000020   e581                         asll    #2,d1
00000022   227c0000 0000                movl    #_x,a1
00000028   23b00800 1800                movl    a0@(0,d0:L),a1@(0,d1:L)
0000002e   52b90000 0000                addql   #1,_i
00000034   7032               L20:      moveq   #50,d0
00000036   b0b90000 0000                cmpl    _i,d0
0000003c   6ece                         jgt     L20001
```

让我们逐条分析这些指令。首先，包含变量 i 的内存位置被清除，也就是实现赋值为零的操作。然后，执行流跳转到标签为 L20 的指令，它和接下来的一条指令用于测试 i 的值是否小于 50。如果是，执行流跳回到标签为 L20001 的指令。

标签为 L20001 的指令开始了循环体。i 被复制到寄存器 d0，然后左移 2 位。之所以要使用移位操作，是因为它的结果和乘 4 是一样的，但它的速度更快。接着，数组 y 的地址被复制到地址寄存器 a0。

现在继续执行前面对 i 的几个计算操作，但这次结果值置于寄存器 d1。然后数组 x 的地址置于地址寄存器 a1。

带复杂操作数的 movl 指令执行实际任务：a0+d0 所指向的值被复制到 a1+d1 所指向的内存位置。然后 i 的值增加 1，并与 50 进行比较，看看是否应该继续循环。

提示：

编译器对表达式 i*4 进行了两次求值，你是不是觉得它有点笨？因为这两个表达式之间 i 的值并没有发生改变。是的，这个编译器确实有点旧，它的优化器也不是很聪明。现代的编译器可能会表现得好一点，但也未必。和编写差劲的源代码，然后依赖编译器产生高效的目标代码相比，直接编写良好的源代码显然更好。但是，你必须记住，效率并不是唯一的因素，通常代码的简洁性更为重要。

一、改用指针方案

现在让我们用指针重新编写这个函数。

```
void
try2()
{
        for( p1 = x, p2 = y; p1 - x < SIZE;
                *p1++ = *p2++;
}
```

我用指针变量取代了下标。其中一个指针用于测试，判断何时退出循环，所以这个方案不再需要计数器。

```
00000046    23fc0000 00000000  _try2:    movl    #_x,_p1
            0000
00000050    23fc0000 00000000             movl    #_y,_p2
            0000
0000005a    601a                          jra     L25
0000005c    20790000 0000      L20003:    movl    _p2,a0
00000062    22790000 0000                 movl    _p1,a1
00000068    2290                          movl    a0@,a1@
0000006a    58b90000 0000                 addql   #4,_p2
00000070    58b90000 0000                 addql   #4,_p1
00000076    7004               L25:       moveq   #4,d0
00000078    2f00                          movl    d0,sp@-
0000007a    20390000 0000                 movl    _p1,d0
00000080    04800000 0000                 subl    #_x,d0
00000086    2f00                          movl    d0,sp@-
00000088    4eb90000 0000                 jbsr    ldiv
0000008e    508f                          addql   #8,sp
00000090    7232                          moveq   #50,d1
00000092    b280                          cmpl    d0,d1
00000094    6ec6                          jgt     L20003
```

和第 1 个版本相比，这些变化并没有带来多大的改进。需要复制整数并增加指针值的代码减少了，但初始化代码却增加了。用于代替乘法的移位指令不见了，而且执行真正任务的 movl 指令不再使用索引。但是，用于检查循环结束的代码却增加了许多，因为两个指令相减的结果必须进行调整（在这里是除以 4）。除法运算是通过把值压到堆栈上并调用子程序 ldiv 实现的。如果这台机器具有 32 位除法指令，除法运算可能会完成得更有效率。

二、重新使用计数器

让我们试试另一种方法。

```
void
try3()
{
        for( i = 0, p1 = x, p2 = y; i < SIZE; i++ )
                *p1++ = *p2++;
}
```

我重新使用了计数器，用于控制循环何时退出，这样可以去除指针减法，并因此缩短目标代码的长度。

```
0000009e    42b90000 0000      _try3:    clrl    _i
000000a4    23fc0000 00000000             movl    #_x,_p1
            0000
000000ae    23fc0000 00000000             movl    #_y,_p2
            0000
000000b8    6020                          jra     L30
000000ba    20790000 0000      L20005:    movl    _p2,a0
000000c0    22790000 0000                 movl    _p1,a1
000000c6    2290                          movl    a0@,a1@
```

```
000000c8    58b90000 0000                   addql   #4,_p2
000000ce    58b90000 0000                   addql   #4,_p1
000000d4    52b90000 0000                   addql   #1,_i
000000da    7032                L30:        moveq   #50,d0
000000dc    b0b90000 0000                   cmpl    _i,d0
000000e2    6ed6                            jgt     L20005
```

在这个版本中，用于复制整数和增加指针值以及控制循环结束的代码要短一些。但在执行间接访问之前，我们仍需把指针变量复制到地址寄存器。

三、寄存器指针变量

我们可以对指针使用寄存器变量，这样就不必复制指针值。但是，它们必须被声明为局部变量。

```
Void
Try4()
{
        register int *p1, *p2;
        register int i;

        for( i = 0, p1 = x, p2 = y; i < SIZE, i++ )
                *p1++ = *p2++;
}
```

这个变化带来了较多的改进，并不仅仅是消除了复制指针的过程。

```
000000f0    7e00                _try4:      moveq   #0,d7
000000f2    2a7c0000 0000                   movl    #_x,a5
000000f8    287c0000 0000                   movl    #_y,a4
000000fe    6004                            jra     L35
00000100    2adc                L20007:     movl    a4@+,a5@+
00000102    5287                            addql   #1,d7
00000104    7032                L35:        moveq   #50,d0
00000106    b087                            cmpl    d7,d0
00000108    6ef6                            jgt     L20007
```

注意，指针变量一开始就保存于寄存器 a4 和 a5 中，我们可以使用硬件的地址自动增量模型（这个行为非常像 C 的后缀++操作符）直接增加它们的值。初始化和用于终止循环的代码基本未作变动。这个版本的代码看上去更好一些。

四、消除计数器

如果我们能找到一种方法来判断循环是否终止，但并不使用开始所提到的那种会引起麻烦的指针减法，我们就可以消除计数器。

```
Void
try5()
{
        register int *p1, *p2;

        for( p1 = x, p2 = y; p1 < &x[SIZE]; )
                *p1++ = *p2++;
}
```

这个循环并没有使用指针减法来判断已经复制了多少个元素，而是进行测试，看看 p1 是否到达源数组的末尾。从功能上说，这个测试应该和前面的一样，但它的效率应该更高，因为它不必执行减法运算。而且，表达式&x[SIZE]可以在编译时求值，因为 SIZE 是个数字常量。下面是它的结果：

```
0000011c    2a7c0000 0000      _try5:    movl    #_x,a5
00000122    287c0000 0000                movl    #_y,a4
00000128    6002                         jra     L40
0000012a    2adc               L20009:   movl    a4@+,a5@+
0000012c    bbfc0000 00c8      L40:      cmpl    #_x+200,a5
00000132    65f6                         jcs     L20009
```

这个版本的代码非常紧凑，速度也很快，完全可以与汇编程序员所编写的同类程序相媲美。计数器以及相关的指令不见了。比较指令包含了表达式_x+200，也就是源代码中的&x[SIZE]。由于 SIZE 是个常量，所以这个计算可以在编译时完成。这个版本的代码是我们在这个机器上所能获得的最紧凑的代码。

五、结论

我们可以从这些试验中学到什么呢？

1. 当你根据某个固定数目的增量在一个数组中移动时，使用指针变量将比使用下标产生效率更高的代码。当这个增量是 1 并且机器具有地址自动增量模型时，这点表现得更为突出。

2. 声明为寄存器变量的指针通常比位于静态内存和堆栈中的指针效率更高（具体提高的幅度取决于你所使用的机器）。

3. 如果你可以通过测试一些已经初始化并经过调整的内容来判断循环是否应该终止，那么你就不需要使用一个单独的计数器。

4. 那些必须在运行时求值的表达式较之诸如&array[SIZE]或 array+SIZE 这样的常量表达式往往代价更高。

提示：

现在，我们必须对前面这些例子进行综合评价。仅仅为了几十微秒的执行时间，是不是值得把第 1 个非常容易理解的循环替换成最后一个被某读者称为“莫名其妙”的循环呢？偶尔，答案是肯定的。但在绝大多数情况下，答案是不容置疑的“否”。在这种方法中，为了一点点运行时效率，它所付出的代价是：程序难于编写在前，难于维护在后。如果程序无法运行或者无法维护，它的执行速度再快也无济于事。

你很容易争辩说，经验丰富的 C 程序员在使用指针循环时不会遇到太大麻烦。但这个论断存在两个荒谬之处。首先，“不会遇到太大麻烦”实际上意味着“还是会遇到一些麻烦”。从本质上说，复杂的用法比简单的用法所涉及的风险要大得多。其次，维护代码的程序员可能并不如阁下经验丰富。程序维护是软件产品的主要成本所在，所以那些使程序维护工作更为困难的编程技巧应慎重使用。

同时，有些机器在设计时使用了特殊的指令，用于执行数组下标操作，目的就是为了使这种极为常用的操作更加快速。在这种机器上的编译器将使用这些特殊的指令来实现下标表达式，但编译器并不一定会用这些指令来实现指针表达式，即使后者也应该这样使用。这样，在这种机器上，下标可能比指针效率更高。

那么，比较这些试验的效率又有什么意义呢？你可能被迫阅读一些别人所编写的“莫名其妙”的代码，所以理解这类代码还是非常重要的。而且在某些场合，追求峰值效率是至关重要的，如那些必须对即时发生的事件作出最快反应的实时程序。但那些运行速度过于缓慢的程序也可以从这类技巧中获益。关键是你先要确认程序中哪些代码段占用了绝大部分运行时间，然后再把你的精力集中在这些代码上，致力于改进它们。这样，你的努力才会获得最大的收获。用于确认这类代码段的

技巧将在第 18 章讨论。

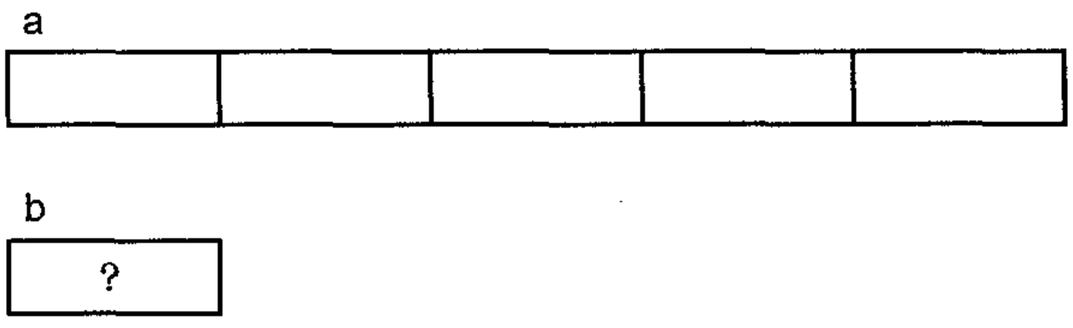
8.1.5 数组和指针

指针和数组并不是相等的。为了说明这个概念，请考虑下面这两个声明：

```
int    a[5];
int    *b;
```

a 和 b 能够互换使用吗？它们都具有指针值，它们都可以进行间接访问和下标引用操作。但是，它们还是存在相当大的区别。

声明一个数组时，编译器将根据声明所指定的元素数量为数组保留内存空间，然后再创建数组名，它的值是一个常量，指向这段空间的起始位置。声明一个指针变量时，编译器只为指针本身保留内存空间，它并不为任何整型值分配内存空间。而且，指针变量并未被初始化为指向任何现有的内存空间，如果它是一个自动变量，它甚至根本不会被初始化。把这两个声明用图的方法来表示，你可以发现它们之间存在显著不同。



因此，上述声明之后，表达式*a 是完全合法的，但表达式*b 却是非法的。*b 将访问内存中某个不确定的位置，或者导致程序终止。另一方面，表达式 b++可以通过编译，但 a++却不行，因为 a 的值是个常量。

你必须清楚地理解它们之间的区别，这是非常重要的，因为我们所讨论的下一个话题有可能把水搅浑。

8.1.6 作为函数参数的数组名

当一个数组名作为参数传递给一个函数时会发生什么情况呢？你现在已经知道数组名的值就是一个指向数组第 1 个元素的指针，所以很容易明白此时传递给函数的是一份该指针的拷贝。函数如果执行了下标引用，实际上是对这个指针执行间接访问操作，并且通过这种间接访问，函数可以访问和修改调用程序的数组元素。

现在我可以解释 C 关于参数传递的表面上的矛盾之处。我早先曾说过所有传递给函数的参数都是通过传值方式进行的，但数组名参数的行为却仿佛它是通过传址调用传递的。传址调用是通过传递一个指向所需元素的指针，然后在函数中对该指针执行间接访问操作实现对数据的访问。作为参数的数组名是个指针，下标引用实际执行的就是间接访问。

那么数组的传值调用行为又是表现在什么地方呢？传递给函数的是参数的一份拷贝（指向数组起始位置的指针的拷贝），所以函数可以自由地操作它的指针形参，而不必担心会修改对应的作为实参的指针。

所以，此处并不存在矛盾：所有的参数都是通过传值方式传递的。当然，如果你传递了一个指向某个变量的指针，而函数对该指针执行了间接访问操作，那么函数就可以修改那个变量。尽管初

看上去并不明显，但数组名作为参数时所发生的正是这种情况。这个参数（指针）实际上是通过传值方式传递的，函数得到的是该指针的一份拷贝，它可以被修改，但调用程序所传递的实参并不受影响。

程序 8.1 是一个简单的函数，用于说明这些观点。它把第 2 个参数中的字符串复制到第 1 个参数所指向的缓冲区。调用程序的缓冲区将被修改，因为函数对参数执行了间接访问操作。但是，无论函数对参数（指针）如何进行修改，都不会修改调用程序的指针实参本身（但可能修改它所指向的内容）。

注意 while 语句中的*string++表达式。它取得 string 所指向的那个字符，并且产生一个副作用，就是修改 string，使它指向下一个字符。用这种方式修改形参并不会影响调用程序的实参，因为只有传递给函数的那份拷贝进行了修改。

```
/*
** 把第 2 个参数中的字符串复制到第 1 个参数指定的缓冲区。
*/
void
strcpy( char *buffer, char const *string )
{
        /*
        ** 重复复制字符，直到遇见 NUL 字节。
        */
        while( (*buffer++ = *string++) != '\0' )
              ;
}
```

程序 8.1 字符串复制　　strcpy.c

提示：

关于这个函数，还有两个要点值得一提（或强调）。首先，形参被声明为一个指向 const 字符的指针。对于一个并不打算修改这些字符的函数而言，预先把它声明为常量有何重要意义呢？这里至少有三个理由。第一，这是一样良好的文档习惯。有些人希望仅观察该函数的原型就能发现该数据不会被修改，而不必阅读完整的函数定义（读者可能无法看到）。第二，编译器可以捕捉到任何试图修改该数据的意外错误。第三，这类声明允许向函数传递 const 参数。

提示：

关于这个函数的第 2 个要点是函数的参数和局部变量被声明为 register 变量。在许多机器上，register 变量所产生的代码将比静态内存中的变量和堆栈中的变量所产生的代码执行速度更快。这一点在早先讨论数组复制函数时就已经提到。对于这类函数，运行时效率尤其重要。它被调用的次数可能相当多，因为它所执行的是一项极为有用的任务。

但是，这取决于在你的环境中，使用 register 变量是否能够产生更快的代码。许多当前的编译器比程序员更加懂得怎样合理分配寄存器。对于这类编译器，在程序中使用 register 声明反而可能降低效率。请检查一下你的编译器的有关文档，看看它是否执行自己的寄存器分配策略[1]。

[1] 在写完这个提示之后，我似乎是遵循了自己的意见，去掉了函数千中的 register 声明，让编译器自己进行优化。同时，我还消除了函数中的局部变量。这个提示本身很有意义，但书上的这个例子并没有很好地展现这一点。

8.1.7 声明数组参数

这里有一个有趣的问题。如果你想把一个数组名参数传递给函数，正确的函数形参应该是怎样的？它是应该声明为一个指针还是一个数组？

正如你所看到的那样，调用函数时实际传递的是一个指针，所以函数的形参实际上是个指针。但为了使程序员新手更容易上手一些，编译器也接受数组形式的函数形参。因此，下面两个函数原型是相等的：

```
int  strlen( char *string );
int  strlen( char string[] );
```

这个相等性暗示指针和数组名实际上是相等的，但千万不要被它糊弄了！这两个声明确实相等，但只是在**当前这个上下文环境中**。如果它们出现在别处，就可能完全不同，就像前面讨论的那样。但对于数组形参，你可以使用任何一种形式的声明。

你可以使用任何一种声明，但哪个“更加准确”呢？答案是指针。因为实参实际上是个指针，而不是数组。同样，表达式 sizeof string 的值是指向字符的指针的长度，而不是数组的长度。

现在你应该清楚为什么函数原型中的一维数组形参无需写明它的元素数目，因为函数并不为数组参数分配内存空间。形参只是一个指针，它指向的是已经在其他地方分配好内存的空间。这个事实解释了为什么数组形参可以与任何长度的数组匹配——它实际传递的只是指向数组第 1 个元素的指针。另一方面，这种实现方法使函数无法知道数组的长度。如果函数需要知道数组的长度，它必须作为一个显式的参数传递给函数。

8.1.8 初始化

就像标量变量可以在它们的声明中进行初始化一样，数组也可以这样做。唯一的区别是数组的初始化需要一系列的值。这个系列是很容易确认的：这些值位于一对花括号中，每个值之间用逗号分隔。如下面的例子所示：

```
int   vector[5] = { 10, 20, 30, 40, 50 };
```

初始化列表给出的值逐个赋值给数组的各个元素，所以 vector[0]获得的值是 10，vector[1]获得的值是 20，其他类推。

静态和自动初始化

数组初始化的方式类似于标量变量的初始化方式——也就是取决于它们的存储类型。存储于静态内存的数组只初始化一次，也就是在程序开始执行之前。程序并不需要执行指令把这些值放到合适的位置，它们一开始就在那里了。这个魔术是由链接器完成的，它用包含可执行程序的文件中合适的值对数组元素进行初始化。如果数组未被初始化，数组元素的初始值将会自动设置为零。当这个文件载入到内存中准备执行时，初始化后的数组值和程序指令一样也被载入到内存中。因此，当程序执行时，静态数组已经初始化完毕。

但是，对于自动变量而言，初始化过程就没有那么浪漫了。因为自动变量位于运行时堆栈中，执行流每次进入它们所在的代码块时，这类变量每次所处的内存位置可能并不相同。在程序开始之前，编译器没有办法对这些位置进行初始化。所以，自动变量在缺省情况下是未初始化的。如果自

动变量的声明中给出了初始值，每次当执行流进入自动变量声明所在的作用域时，变量就被一条隐式的赋值语句初始化。这条隐式的赋值语句和普通的赋值语句一样需要时间和空间来执行。数组的问题在于初始化列表中可能有很多值，这就可能产生许多条赋值语句。对于那些非常庞大的数组，它的初始化时间可能非常可观。

因此，这里就需要权衡利弊。当数组的初始化局部于一个函数（或代码块）时，你应该仔细考虑一下，在程序的执行流每次进入该函数（或代码块）时，每次都对数组进行重新初始化是不是值得。如果答案是否定的，你就把数组声明为 static，这样数组的初始化只需在程序开始前执行一次。

8.1.9　不完整的初始化

在下面两个声明中会发生什么情况呢？

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

在这两种情况下，初始化值的数目和数组元素的数目并不匹配。第 1 个声明是错误的，我们没有办法把 6 个整型值装到 5 个整型变量中。但是，第 2 个声明却是合法的，它为数组的前 4 个元素提供了初始值，最后一个元素则初始化为 0。

那么，我们可不可以省略列表中间的那些值呢？

```
int       vector[5] = { 1, 5 };
```

编译器只知道初始值不够，但它无法知道缺少的是哪些值。所以，只允许省略最后几个初始值。

8.1.10　自动计算数组长度

这里是另一个有用技巧的例子。

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

如果声明中并未给出数组的长度，编译器就把数组的长度设置为刚好能够容纳所有的初始值的长度。如果初始值列表经常修改，这个技巧尤其有用。

8.1.11　字符数组的初始化

根据目前我们所学到的知识，你可能认为字符数组将以下面这种形式进行初始化：

```
char  message[] = { 'H', 'e', 'l', 'l', 'o', 0 };
```

这个方法当然可行。但除了非常短的字符串，这种方法确实很笨拙。因此，语言标准提供了一种快速方法用于初始化字符数组：

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

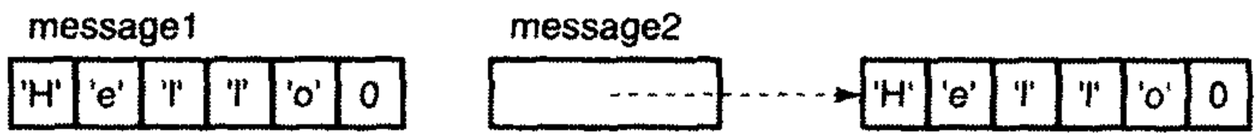
尽管它看上去像是一个字符串常量，**实际上并不是**。它只是前例的初始化列表的另一种写法。

如果它们看上去完全相同，你如何分辨字符串常量和这种初始化列表快速记法呢？它们是根据它们所处的上下文环境进行区分的。当用于初始化一个字符数组时，它就是一个初始化列表。在其他任何地方，它都表示一个字符串常量。

这里有一个例子：

```
char    message1[] = "Hello";
char    *message2 = "Hello";
```

这两个初始化看上去很像，但它们具有不同的含义。前者初始化一个字符数组的元素，而后者则是一个真正的字符串常量。这个指针变量被初始化为指向这个字符串常量的存储位置，如下图所示：



8.2 多维数组

如果某个数组的维数不止 1 个，它就被称为多维数组。例如，下面这个声明

```
int         matrix[6][10];
```

创建了一个包含 60 个元素的矩阵。但是，它是 6 行每行 10 个元素，还是 10 行每行 6 个元素？

为了回答这个问题，你需要从一个不同的视点观察多维数组。考虑下列这些维数不断增加的声明：

```
int         a;
int         b[10];
int         c[6][10];
int         d[3][6][10];
```

a 是个简单的整数。接下来的那个声明增加了一个维数，所以 b 就是一个向量，它包含 10 个整型元素。

c 只是在 b 的基础上再增加一维，所以我们可以把 c 看作是一个包含 6 个元素的向量，只不过它的每个元素本身是一个包含 10 个整型元素的向量。换句话说，c 是个一维数组的一维数组。d 也是如此：它是一个包含 3 个元素的数组，每个元素都是包含 6 个元素的数组，而这 6 个元素中的每一个又都是包含 10 个整型元素的数组。简洁地说，d 是一个 3 排 6 行 10 列的整型三维数组。

理解这个视点是非常重要的，因为它正是 C 实现多维数组的基础。为了加强这个概念，让我们先来讨论数组元素在内存中的存储顺序。

8.2.1 存储顺序

考虑下面这个数组：

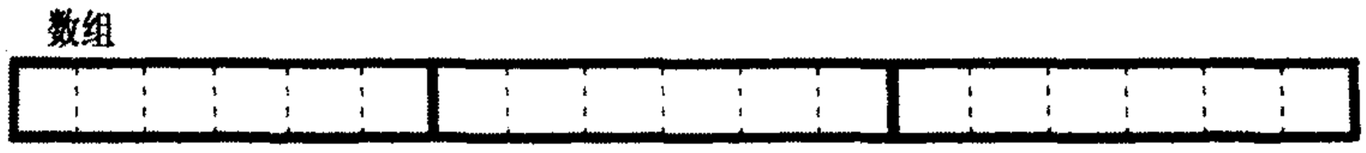
```
int         array[3];
```

它包含 3 个元素，如下图所示：

但现在假定你被告知这3个元素中的每一个实际上都是包含6个元素的数组，情况又将如何呢？下面是这个新的声明：

```
int         array[3][6];
```

下面是它在内存中的存储形式：



实线方框表示第 1 维的 3 个元素，虚线用于划分第 2 维的 6 个元素。按照从左到右的顺序，上面每个元素的下标值分别是：

```
0,0  0,1  0,2  0,3  0,4  0,5  1,0  1,1  1,2
1,3  1,4  1,5  2,0  2,1  2,2  2,3  2,4  2,5
```

这个例子说明了数组元素的存储顺序(storage order)。在 C 中，多维数组的元素存储顺序按照最右边的下标率先变化的原则，称为行主序(row major order)。知道了多维数组的存储顺序有助于回答一些有用的问题，比如你应该按照什么样的顺序来编写初始化列表的值。

下面的代码段将会打印出什么样的值呢？

```
int     matrix[6][10];
int     *mp;
...
mp = &matrix[3][8];
printf( "First value is %d\n", *mp );
printf( "Second value is %d\n", *++mp );
printf( "Third value is %d\n", *++mp );
```

很显然，第 1 个被打印的值将是 matrix[3][8]的内容，但下一个被打印的又是什么呢？存储顺序可以回答这个问题——下一个元素将是最右边下标首先变化的那个，也就是 matrix[3][9]。再接下去又轮到谁呢？第 9 列可是一行中的最后一列啦。不过，根据存储顺序规定，一行存满后就轮到下一行，所以下一个被打印的元素将是 matrix[4][0][1]。

这里有一个相关的问题。matrix 到底是 6 行 10 列还是 10 行 6 列？答案可能会令你大吃一惊——在某些上下文环境中，两种答案都对。

两种都对？怎么可能有两个不同的答案呢？这个简单，如果你根据下标把数据存放于数组中并在以后根据下标查找数组中的值，那么不管你把第 1 个下标解释为行还是列，都不会有什么区别。**只要你每次都坚持使用同一种方法**，这两种解释方法都是可行的。

但是，把第 1 个下标解释为行或列并不会改变数组的存储顺序。如果你把第 1 个下标解释为行，把第 2 个下标解释为列，那么当你按照存储顺序逐个访问数组元素时，你所获得的元素是按行排列的。另一方面，如果把第 1 个下标作为列，那么当你按前面的顺序访问数组元素时，你所得到的元素是按列排列的。你可以在你的程序中选择更加合理的解释方法。但是，你不能修改内存中数组元素的实际存储方式。这个顺序是由标准定义的。

8.2.2　数组名

一维数组名的值是一个指针常量，它的类型是“指向元素类型的指针”，它指向数组的第 1 个元素。多维数组也差不多简单。唯一的区别是多维数组第 1 维的元素实际上是另一个数组。例如，下面这个声明：

```
int     matrix[3][10];
```

[1] 这个例子使用一个指向整型的指针遍历存储了一个二维整型数组元素的内存空间。这个技巧被称为“flattening the array（压扁数组）”，它实际上是非法的，因此从某行移到下一行后就无法回到包含第 1 行的那个子数组。尽管它通常没什么问题，但有可能的话还是应该避免。

创建了 matrix，它可以看作是一个一维数组，包含 3 个元素，只是每个元素恰好是包含 10 个整型元素的数组。

matrix 这个名字的值是一个指向它第 1 个元素的指针，所以 matrix 是一个指向一个包含 10 个整型元素的数组的指针。

K&R C：

指向数组的指针这个概念是在相当后期才加入到 K&R C 中的，有些老式的编译器并没有完全实现它。但是，指向数组的指针这个概念对于理解多维数组的下标引用是至关重要的。

8.2.3 下标

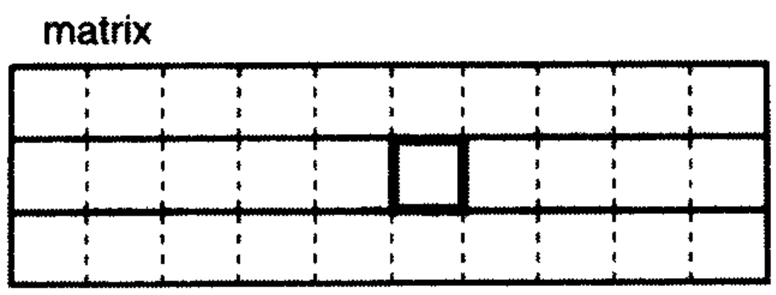
如果要标识一个多维数组的某个元素，必须按照与数组声明时相同的顺序为每一维都提供一个下标，而且每个下标都单独位于一对方括号内。在下面的声明中：

```
int        matrix[3][10];
```

表达式

```
matrix[1][5]
```

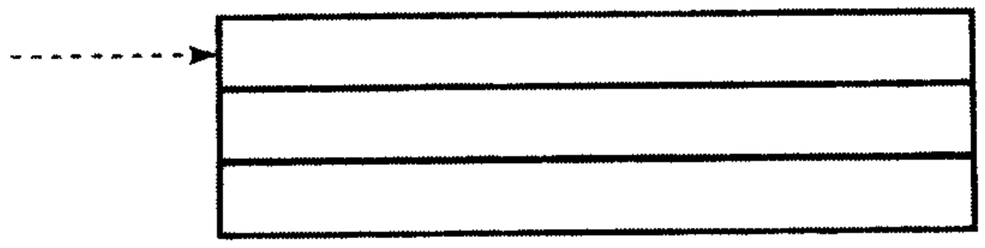
访问下面这个元素：



但是，下标引用实际上只是间接访问表达式的一种伪装形式，即使在多维数组中也是如此。考虑下面这个表达式：

```
matrix
```

它的类型是“指向包含 10 个整型元素的数组的指针”，它的值是：

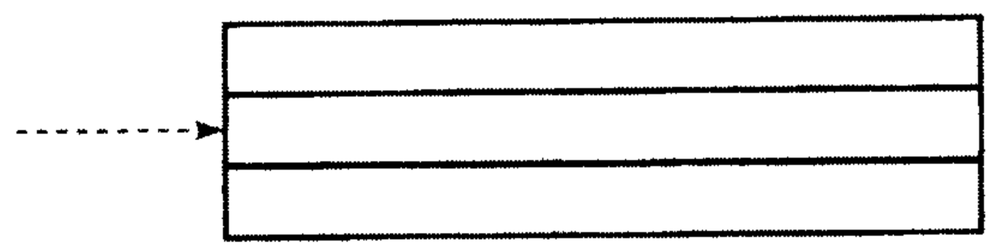


它指向包含 10 个整型元素的第 1 个子数组。

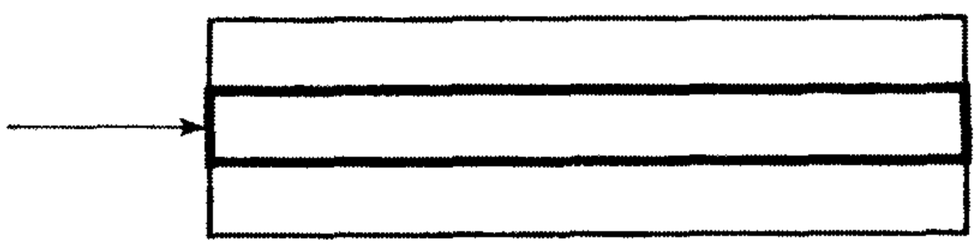
表达式

```
matrix + 1
```

也是一个“指向包含 10 个整型元素的数组的指针”，但它指向 matrix 的另一行：



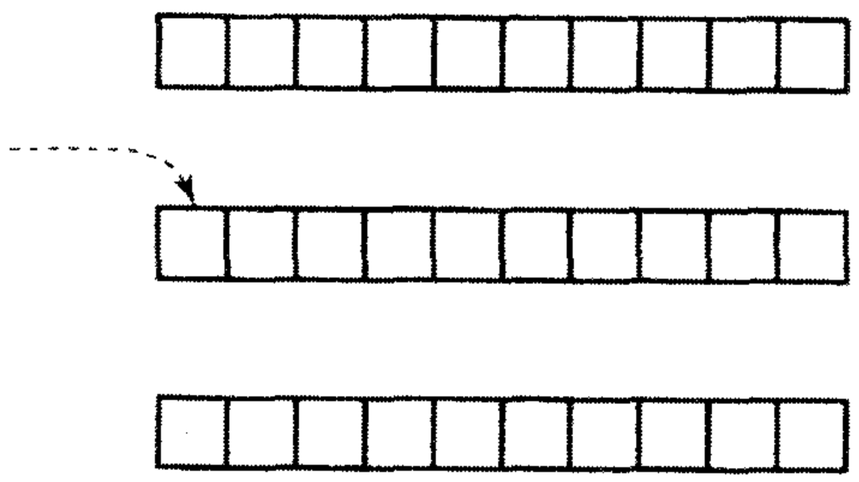
为什么？因为 1 这个值根据包含 10 个整型元素的数组的长度进行调整，所以它指向 matrix 的下一行。如果对其执行间接访问操作，就如下图随箭头选择中间这个子数组：



所以表达式

```
*(matrix + 1)
```

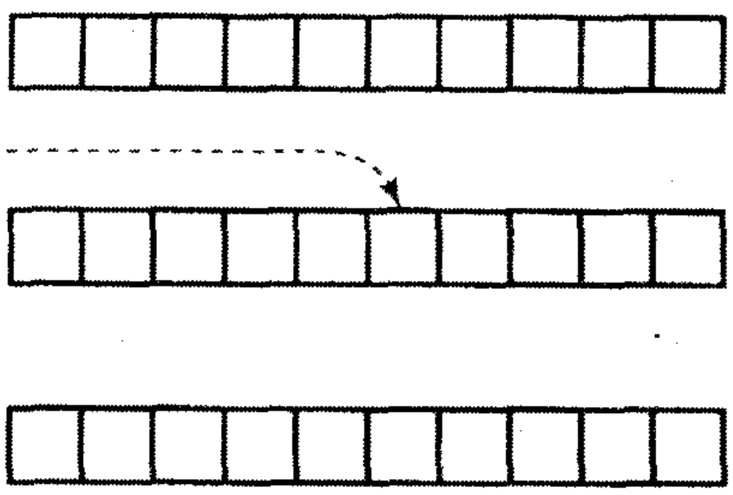
事实上标识了一个包含 10 个整型元素的子数组。数组名的值是个常量指针，它指向数组的第 1 个元素，在这个表达式中也是如此。它的类型是“指向整型的指针”，我们现在可以在下一维的上下文环境中显示它的值：



现在请拿稳你的帽子，猜猜下面这个表达式的结果是什么？

```
*( matrix + 1 ) + 5
```

前一个表达式是个指向整型值的指针，所以 5 这个值根据整型的长度进行调整。整个表达式的结果是一个指针，它指向的位置比原先那个表达式所指向的位置向后移动了 5 个整型元素。



对其执行间接访问操作：

```
*( *( matrix + 1 ) + 5 )
```

它所访问的正是图中的那个整型元素。如果它作为右值使用，你就取得存储于那个位置的值。如果它作为左值使用，这个位置将存储一个新值。

这个看上去吓人的表达式实际上正是我们的老朋友——下标。我们可以把子表达式*(matrix + 1)改写为 matrix[1]。把这个下标表达式代入原先的表达式，我们将得到：

```
*( matrix[1] + 5 )
```

这个表达式是完全合法的。matrix[1]选定一个子数组，所以它的类型是一个指向整型的指针。我们对这个指针加上 5，然后执行间接访问操作。

但是，我们可以再次用下标代替间接访问，所以这个表达式还可以写成：

```
matrix[1][5]
```

这样，即使对于多维数组，下标仍然是另一种形式的间接访问表达式。

这个练习的要点在于它说明了多维数组中的下标引用是如何工作的，以及它们是如何依赖于指向数组的指针这个概念。下标是从左向右进行计算的，数组名是一个指向第 1 维第 1 个元素的指针，所以第 1 个下标值根据该元素的长度进行调整。它的结果是一个指向那一维中所需元素的指针。间接访问操作随后选择那个特定的元素。由于该元素本身是个数组，所以这个表达式的类型是一个指向下一维第 1 个元素的指针。下一个下标值根据这个长度进行调整，这个过程重复进行，直到所有的下标均计算完毕。

警告：

在许多其他语言中，多重下标被写作逗号分隔的值列表形式。有些语言这两种形式都允许，但 C 并非如此：编写

```
matrix[4, 3]
```

看上去没有问题，但它的功能和你想象的几乎肯定不同。记住，逗号操作符首先对第 1 个表达式求值，但随即丢弃这个值。最后的结果是第 2 个表达式的值。因此，前面这个表达式与下面这个表达式是相等的。

```
matrix[3]
```

问题在于这个表达式可以顺利通过编译，不会产生任何错误或警告信息。这个表达式是完全合法的，但它的意思跟你想象的根本不同。

8.2.4 指向数组的指针

下面这些声明合法吗？

```
int        vector[10], *vp = vector;
int        matrix[3][10], *mp = matrix;
```

第 1 个声明是合法的。它为一个整型数组分配内存，并把 vp 声明为一个指向整型的指针，并把它初始化为指向 vector 数组的第 1 个元素。vector 和 vp 具有相同的类型：指向整型的指针。但是，第 2 个声明是非法的。它正确地创建了 matrix 数组，并把 mp 声明为一个指向整型的指针。但是，mp 的初始化是不正确的，因为 matrix 并不是一个指向整型的指针，而是一个指向整型数组的指针。我们应该怎样声明一个指向整型数组的指针的呢？

```
int   (*p)[10];
```

这个声明比我们以前见过的所有声明更为复杂，但它事实上并不是很难。你只要假定它是一个表达式并对它求值。下标引用的优先级高于间接访问，但由于括号的存在，首先执行的还是间接访问。所以，p 是个指针，但它指向什么呢？

接下来执行的是下标引用，所以 p 指向某种类型的数组。这个声明表达式中并没有更多的操作

符，所以数组的每个元素都是整数。

声明并没有直接告诉你 p 是什么，但推断它的类型并不困难——当我们对它执行间接访问操作时，我们得到的是个数组，对该数组进行下标引用操作得到的是一个整型值。所以 p 是一个指向整型数组的指针。

在声明中加上初始化后是下面这个样子：

```
int        (*p)[10] = matrix;
```

它使 p 指向 matrix 的第 1 行。

p 是一个指向拥有 10 个整型元素的数组的指针。当你把 p 与一个整数相加时，该整数值首先根据 10 个整型值的长度进行调整，然后再执行加法。所以我们可以使用这个指针一行一行地在 matrix 中移动。

如果你需要一个指针逐个访问整型元素而不是逐行在数组中移动，你应该怎么办呢？下面两个声明都创建了一个简单的整型指针，并以两种不同的方式进行初始化，指向 matrix 的第 1 个整型元素。

```
Int        *pi = &matrix[0][0];
int        *pi = matrix[0];
```

增加这个指针的值使它指向下一个整型元素。

警告：

如果你打算在指针上执行任何指针运算，应该避免这种类型的声明：

```
int        (*p)[] = matrix;
```

p 仍然是一个指向整型数组的指针，但数组的长度却不见了。当某个整数与这种类型的指针执行指针运算时，它的值将根据空数组的长度进行调整（也就是说，与零相乘），这很可能不是你所设想的。有些编译器可以捕捉到这类错误，但有些编译器却不能。

8.2.5　作为函数参数的多维数组

作为函数参数的多维数组名的传递方式和一维数组名相同——实际传递的是个指向数组第 1 个元素的指针。但是，两者之间的区别在于，多维数组的每个元素本身是另外一个数组，编译器需要知道它的维数，以便为函数形参的下标表达式进行求值。这里有两个例子，说明了它们之间的区别：

```
int        vector[10];
...
func1(vector);
```

参数 vector 的类型是指向整型的指针，所以 func1 的原型可以是下面两种中的任何一种：

```
void func1( int *vec );
void func1(int vec[] );
```

作用于 vec 上面的指针运算把整型的长度作为它的调整因子。

现在让我们来观察一个矩阵：

```
int        matrix[3][10];
...
func2( matrix );
```

这里，参数 matrix 的类型是指向包含 10 个整型元素的数组的指针。func2 的原型应该是怎样的

呢？你可以使用下面两种形式中的任何一种：

```
void func2( int (*mat)[10] );
void func2( int mat[][10] );
```

在这个函数中，mat 的第 1 个下标根据包含 10 个元素的整型数组的长度进行调整，接着第 2 个下标根据整型的长度进行调整，这和原先的 matrix 数组一样。

这里的关键在于编译器必须知道第 2 个及以后各维的长度才能对各下标进行求值，因此在原型中必须声明这些维的长度。第 1 维的长度并不需要，因为在计算下标值时用不到它。

在编写一维数组形参的函数原型时，你既可以把它写成数组的形式，也可以把它写成指针的形式。但是，对于多维数组，只有第 1 维可以进行如此选择。尤其是，把 func2 写成下面这样的原型是不正确的：

```
void func2( int **mat );
```

这个例子把 mat 声明为一个指向整型指针的指针，它和指向整型数组的指针并不是一回事。

8.2.6 初始化

在初始化多维数组时，数组元素的存储顺序就变得非常重要。编写初始化列表有两种形式。第 1 种是只给出一个长长的初始值列表，如下面的例子所示。

```
int matrix[2][3] = { 100, 101, 102, 110, 111, 112 };
```

多维数组的存储顺序是根据最右边的下标率先变化的原则确定的，所以这条初始化语句和下面这些赋值语句的结果是一样的：

```
matrix[0][0] = 100;
matrix[0][1] = 101;
matrix[0][2] = 102;
matrix[1][0] = 110;
matrix[1][1] = 111;
matrix[1][2] = 112;
```

第 2 种方法基于多维数组实际上是复杂元素的一维数组这个概念。例如，下面是一个二维数组的声明：

```
int        tow_dim[3][5];
```

我们可以把 tow_dim 看成是一个包含 3 个（复杂的）元素的一维数组。为了初始化这个包含 3 个元素的数组，我们使用一个包含 3 个初始内容的初始化列表：

```
int        two_dim[3][5] = { ★, ★, ★ };
```

但是，该数组的每个元素实际上都是包含 5 个元素的整型数组，所以每个★的初始化列表都应该是一个由一对花括号包围的 5 个整型值。用这类列表替换每个★将产生如下代码：

```
int      two_dim[3][5] = {
         { 00, 01, 02, 03, 04 },
         { 10, 11, 12, 13, 14 },
         { 20, 21, 22, 23, 24 }
};
```

当然，我们所使用的缩进和空格并非必需，但它们使这个列表更容易阅读。

如果你把这个例子中除了最外层之外的花括号都去掉，剩下的就是和第 1 个例子一样的简单初始化列表。那些花括号只是起到了在初始化列表内部逐行定界的作用。

图 8.1 和图 8.2 显示了三维和四维数组的初始化。在这些例子中，每个作为初始值的数字显示了它的存储位置的下标值[1]。

```
Int       three_dim[2][3][5] = {
          {
                { 000, 001, 002, 003, 004 },
                { 010, 011, 012, 013, 014 },
                { 020, 021, 022, 023, 024 }
          },
          {
                { 100, 101, 102, 103, 104 },
                { 110, 111, 112, 113, 114 },
                { 120, 121, 122, 123, 124}
          }
};
```

图 8.1　初始化一个三维数组

```
int       four_dim[2][2][3][5] = {
          {
               {
                  { 0000, 0001, 0002, 0003, 0004 },
                  { 0010, 0011, 0012, 0013, 0014 },
                  { 0020, 0021, 0022, 0023, 0024 }
               },
               {
                   { 0100, 0101, 0102, 0103, 0104 },
                   { 0110, 0111, 0112, 0113, 0114 },
                   { 0120, 0121, 0122, 0123, 0124 }
               }
          },
          {
               {
                   { 1000, 1001, 1002, 1003, 1004 },
                   { 1010, 1011, 1012, 1013, 1014 },
                   { 1020, 1021, 1022, 1023, 1024 }
               },
               {
                   { 1100, 1101, 1102, 1103, 1104 },
                   { 1110, 1111, 1112, 1113, 1114 },
                   { 1120, 1121, 1122, 1123, 1124 }
               }
          }
};
```

图 8.2　初始化一个四维数组

提示：

既然加不加那些花括号对初始化过程不会产生影响，那么为什么要不厌其烦地加上它们呢？这里有两个原因。首先是它有利于显示数组的结构。一个长长的单一数字列表使你很难看清哪个

[1] 如果这些例子进行编译，那些以 0 开头的初始值实际上会被解释为八进制数值。我们在此不会理会它，只需要观察每个初始值的单独数字。

值位于数组中的哪个位置。因此，花括号起到了路标的作用，使你更容易确信正确的值出现在正确的位置。

其次，对于不完整的初始化列表，花括号就相当有用。如果没有这些花括号，你只能在初始化列表中省略最后几个初始值。即使一个大型多维数组只有几个元素需要初始化，你也必须提供一个非常长的初始化列表，因为中间元素的初始值不能省略。但是，如果使用了这些花括号，每个子初始列表都可以省略尾部的几个初始值。同时，每一维的初始列表各自都是一个初始化列表。

为了说明这个概念，让我们重新观察图 8.2 的四维数组初始化列表，并略微改变一下我们的要求。假定我们只需要对数组的两个元素进行初始化，元素[0][0][0][0]初始化为 100，元素[1][0][0][0]初始化为 200，其余的元素都缺省地初始化为 0。下面是我们用于完成这个任务的方法：

```
int     four_dim[2][2][3][5] = {
        {
                {
                        { 100 }
                }
        },
        {
                {
                        { 200 }
                }
        }
};
```

如果初始化列表内部不使用花括号，我们就需要下面这个长长的初始化列表：

```
int     four_dim[2][2][3][5] = { 100, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 200 };
```

这个列表不仅难于阅读，而且一开始要准确地把 100 和 200 这两个值放到正确的位置都很困难。

8.2.7 数组长度自动计算

在多维数组中，只有第 1 维才能根据初始化列表缺省地提供。剩余的几个维必须显式地写出，这样编译器就能推断出每个子数组维数的长度。例如：

```
int     two_dim[][5] = {
        { 00, 01, 02 },
        { 10, 11 },
        { 20, 21, 22, 23 }
};
```

编译器只要数一下初始化列表中所包含的初始值个数，就可以推断出最左边一维为 3。

为什么其他维的大小无法通过对它的最长初始列表的初始值个数进行计数自动推断出来呢？原则上，编译器能够这样做。但是，这需要每个列表中的子初始值列表至少有一个要以完整的形式出现（不得省略末尾的初始值），这样才能保证编译器正确地推断出每一维的长度。但是，如果我们要求除第 1 维之外的其他维的大小都显式提供，所有的初始值列表都无需完整。

8.3 指针数组

除了类型之外，指针变量和其他变量很相似。正如你可以创建整型数组一样，你也可以声明指

针数组。这里有一个例子：

```
int        *api[10];
```

为了弄清这个复杂的声明，我们假定它是一个表达式，并对它进行求值。

下标引用的优先级高于间接访问，所以在这个表达式中，首先执行下标引用。因此，api 是某种类型的数组（噢！顺便说一下，它包含的元素个数为 10）。在取得一个数组元素之后，随即执行的是间接访问操作。这个表达式不再有其他操作符，所以它的结果是一个整型值。

那么 api 到底是什么东西？对数组的某个元素执行间接访问操作后，我们得到一个整型值，所以 api 肯定是个数组，它的元素类型是指向整型的指针。

什么地方你会使用指针数组呢？这里有一个例子：

```
char     const    keyword[] = {
         "do",
         "for",
         "if",
         "register",
         "return",
         "switch",
         "while"
};
#define N_KEYWORD        \
         ( sizeof( keyword ) / sizeof( keyword[0] ) )
```

注意 sizeof 的用途，它用于对数组中的元素进行自动计数。sizeof(keyword)的结果是整个数组所占用的字节数，而 sizeof(keyword[0])的结果则是数组每个元素所占用的字节数。这两个值相除，结果就是数组元素的个数。

这个数组可以用于一个计算 C 源文件中关键字个数的程序中。输入的每个单词将与列表中的字符串进行比较，所有的匹配都将被计数。程序 8.2 遍历整个关键字列表，查找是否存在与参数字符串相同的匹配。当它找到一个匹配时，函数就返回这个匹配在列表中的偏移量。调用程序必须知道 0 代表 do，1 代表 for 等，此外它还必须知道返回值如果是-1 表示没有关键字匹配。这个信息很可能是通过头文件所定义的符号获得的。

```
/*
** 判断参数是否与一个关键字列表中的任何单词匹配，并返回匹配的索引值。如果未** 找到匹配，函数返回-1。
*/
#include <string.h>

int
lookup_keyword( char const * const desired_word,
    char const *keyword_table[], int const size )
{
         char const **kwp;

         /*
         ** 对于表中的每个单词 ...
         */
         for( kwp = keyword_table; kwp < keyword_table + size; kwp++ )
              /*
              ** 如果这个单词与我们所查找的单词匹配，返回它在表中的位置。
              */
              if( strcmp( desired_word, *kwp ) == 0 )
                   return kwp - keyword_table;
```

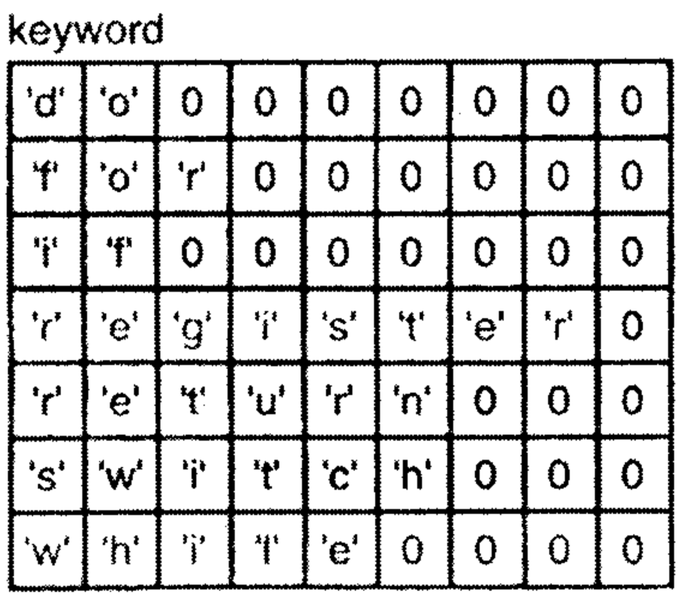
```
        /*
        ** 没有找到。
        */
        return -1;
}
```

程序 8.2 关键字查找 keyword.c

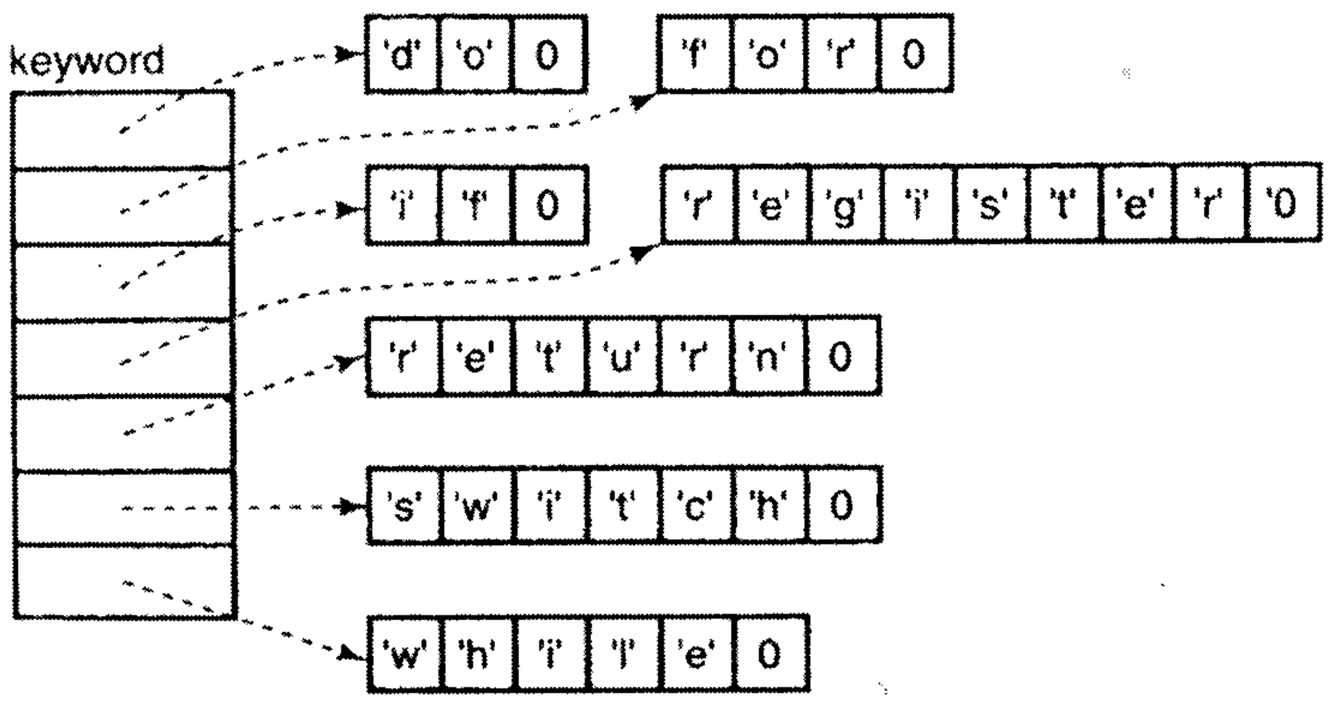
我们也可以把关键字存储在一个矩阵中，如下所示：

```
char    const   keyword[][9] = {
        "do",
        "for",
        "if",
        "register",
        "return",
        "switch",
        "while"
};
```

这个声明和前面那个声明的区别在什么地方呢？第 2 个声明创建了一个矩阵，它每一行的长度刚好可以容纳最长的关键字（包括作为终止符的 NUL 字节）。这个矩阵的样子如下所示：



第 1 个声明创建了一个指针数组，每个指针元素都初始化为指向各个不同的字符串常量，如下所示：



注意这两种方法在占用内存空间方面的区别。矩阵看上去效率低一些，因为它的每一行的长度都被固定为刚好能容纳最长的关键字。但是，它不需要任何指针。另一方面，指针数组本身也要占

用空间，但是每个字符串常量占据的内存空间只是它本身的长度。

如果我们需要对程序 8.2 进行修改，改用矩阵代替指针数组，我们应该怎么做呢？答案可能会令你吃惊，我们只需要对列表形参和局部变量的声明进行修改就可以了，具体的代码无需变动。由于数组名的值是一个指针，所以无论传递给函数的是指针还是数组名，函数都能运行。

哪个方案更好一些呢？这取决于你希望存储的具体字符串。如果它们的长度都差不多，那么矩阵形式更紧凑一些，因为它无需使用指针。但是，如果各个字符串的长度千差万别，或者更糟，绝大多数字符串都很短，但少数几个却很长，那么指针数组形式就更紧凑一些。它取决于指针所占用的空间是否小于每个字符串都存储于固定长度的行所浪费的空间。

实际上，除了非常巨大的表，这些差别非常之小，所以根本不重要。人们时常选择指针数组方案，但略微对其作些改变：

```
char   const  *keyword[] = {
            "do",
            "for",
            "if",
            "register",
            "return",
            "switch",
            "while",
            NULL
};
```

这里，我们在表的末尾增加了一个 NULL 指针。这个 NULL 指针使函数在搜索这个表时能够检测到表的结束，而无需预先知道表的长度，如下所示：

```
for( kwp = keyword_table; *kwp != NULL; kwp++ )
```

8.4　总结

在绝大多数表达式中，数组名的值是指向数组第 1 个元素的指针。这个规则只有两个例外。sizeof 返回整个数组所占用的字节而不是一个指针所占用的字节。单目操作符&返回一个指向数组的指针，而不是一个指向数组第 1 个元素的指针的指针。

除了优先级不同以外，下标表达式 array[value]和间接访问表达式*(array+(value))是一样的。因此，下标不仅可以用于数组名，也可以用于指针表达式中。不过这样一来，编译器就很难检查下标的有效性。指针表达式可能比下标表达式效率更高，但下标表达式绝不可能比指针表达式效率更高。但是，以牺牲程序的可维护性为代价获得程序的运行时效率的提高可不是个好主意。

指针和数组并不相等。数组的属性和指针的属性大相径庭。当我们声明一个数组时，它同时也分配了一些内存空间，用于容纳数组元素。但是，当我们声明一个指针时，它只分配了用于容纳指针本身的空间。

当数组名作为函数参数传递时，实际传递给函数的是一个指向数组第 1 个元素的指针。函数所接收到的参数实际上是原参数的一份拷贝，所以函数可以对其进行操纵而不会影响实际的参数。但是，对指针参数执行间接访问操作允许函数修改原先的数组元素。数组形参既可以声明为数组，也可以声明为指针。这两种声明形式只有当它们**作为函数的形参时**才是相等的。

数组也可以用初始值列表进行初始化，初始值列表就是由一对花括号包围的一组值。静态变量（包括数组）在程序载入到内存时得到初始值。自动变量（包括数组）每次当执行流进入它们声明所

在的代码块时都要使用隐式的赋值语句重新进行初始化。如果初始值列表包含的值的个数少于数组元素的个数，数组最后几个元素就用缺省值进行初始化。如果一个被初始化的数组的长度在声明中未给出，编译器将使这个数组的长度设置为刚好能容纳初始值列表中所有值的长度。字符数组也可以用一种很像字符串常量的快速方法进行初始化。

多维数组实际上是一维数组的一种特型，就是它的每个元素本身也是一个数组。多维数组中的元素根据行主序进行存储，也就是最右边的下标率先变化。多维数组名的值是一个指向它第 1 个元素的指针，也就是一个指向数组的指针。对该指针进行运算将根据它所指向数组的长度对操作数进行调整。多维数组的下标引用也是指针表达式。当一个多维数组名作为参数传递给一个函数时，它所对应的函数形参的声明中必须显式指明第 2 维（和接下去所有维）的长度。由于多维数组实际上是复杂元素的一维数组，一个多维数组的初始化列表就包含了这些复杂元素的值。这些值的每一个都可能包含嵌套的初始值列表，由数组各维的长度决定。如果多维数组的初始化列表是完整的，它的内层花括号可以省略。在多维数组的初始值列表中，只有第 1 维的长度会被自动计算出来。

我们还可以创建指针数组。字符串的列表可以以矩阵的形式存储，也可以以指向字符串常量的指针数组形式存储。在矩阵中，每行必须与最长字符串的长度一样长，但它不需要任何指针。指针数组本身要占用空间，但每个指针所指向的字符串所占用的内存空间就是字符串本身的长度。

8.5 警告的总结

1. 当访问多维数组的元素时，误用逗号分隔下标。
2. 在一个指向未指定长度的数组的指针上执行指针运算。

8.6 编程提示的总结

1. 一开始就编写良好的代码显然比依赖编译器来修正劣质代码更好。
2. 源代码的可读性几乎总是比程序的运行时效率更为重要。
3. 只要有可能，函数的指针形参都应该声明为 const。
4. 在有些环境中，使用 register 关键字提高程序的运行时效率。
5. 在多维数组的初始值列表中使用完整的多层花括号能提高可读性。

8.7 问题

1. 根据下面给出的声明和数据，对每个表达式进行求值并写出它的值。在对每个表达式进行求值时使用原先给出的值（也就是说，某个表达式的结果不影响后面的表达式）。假定 ints 数组在内存中的起始位置是 100，整型值和指针的长度都是 4 个字节。

```
int     ints[20] = {
        10, 20, 30, 40, 50, 60, 70, 80, 90, 100,
        110, 120, 130, 140, 150, 160, 170, 180, 190, 200
};
(Other declarations)
int     *ip = ints + 3;
```

表达式	值	表达式	值
ints	_____	ip	_____
ints[4]	_____	ip[4]	_____
ints + 4	_____	ip + 4	_____
*ints + 4	_____	*ip + 4;	_____
*(ints + 4)	_____	*(ip + 4)	_____
ints[-2]	_____	ip[-2]	_____
&ints	_____	&ip	_____
&ints[4]	_____	&ip[4]	_____
&ints + 4	_____	&ip + 4	_____
&ints[-2]	_____	&ip[-2]	_____

2. 表达式 array[i+j]和 i+j[array]是不是相等？
3. 下面的声明试图按照从 1 开始的下标访问数组 data，它能行吗？

```
int    actual_data[ 20 ];
int    *data = actual_data - 1;
```

4. 下面的循环用于测试某个字符串是否是回文，请对它进行重写，用指针变量代替下标。

```
char    buffer[SIZE];
int     front, rear;
...
front = 0;
rear = strlen( buffer ) - 1;
while( front < rear ){
        if( buffer[front] != buffer[rear] )
                break;

        front += 1;
        rear -= 1;
}
if( front >= rear ){
        printf( "It is a palindrome!\n" );
}
```

5. 指针在效率上可能强于下标，这是使用它们的动机之一。那么什么时候使用下标是合理的，尽管它在效率上可能有所损失？
6. 在你的机器上编译函数 try1 至 try5，并分析结果的汇编代码。你的结论是什么？
7. 测试你对前一个问题的结论，方法是运行每一个函数并对它们的执行时间进行计时。把数组的元素增加到几千个，增加试验的准确性，因为此时复制所占用的时间远远超过程序不相关部分所占用的时间。同样，在一个循环内部调用函数，让它重复执行足够多的次数，这样你可以精确地为执行时间计时。为这个试验两次编译程序——一次不使用任何优化措施，另一次使用优化措施。如果你的编译器可以提供选择，请选择优化措施以获得最佳速度。

8. 下面的声明取自某个源文件：

```
int    a[10];
int    *b = a;
```

但在另一个不同的源文件中，却发现了这样的代码：

```
extern  int     *a;
extern  int     b[];
int     x, y;
...
x = a[3];
y = b[3];
```

请解释一下，当两条赋值语句执行时会发生什么？（假定整型和指针的长度都是 4 个字节。）

9. 编写一个声明，初始化一个名叫 coin_values 的整型数组，各个元素的值分别表示当前各种美元硬币的币值。

10. 给定下列声明

```
int       array[4][2];
```

请写出下面每个表达式的值。假定数组的起始位置为 1000，整型值在内存中占据 2 个字节的空间。

表达式	值
array	
array + 2	
array[3]	
array[2] - 1	
&array[1][2]	
&array[2][0]	

11. 给定下列声明

```
int       array[4][2][3][6];
```

表达式	值	X 的类型
array		
array + 2		
array[3]		
array[2] - 1		
array[2][1]		
array[1][0] + 1		
array[1][0][2]		
array[0][1][0] + 2		
array[3][1][2][5]		
&array[3][1][2][5]		

计算上表中各个表达式的值。同时，写出变量 x 所需的声明，这样表达式不用进行强制类型转换就可以赋值给 x。假定数组的起始位置为 1000，整型值在内存中占据 4 个字节的空间。

12. C 的数组按照行主序存储。什么时候需要使用这个信息？

13. 给定下列声明

```
int     array[4][5][3];
```

把下列各个指针表达式转换为下标表达式。

表达式	下标表达式
`*array`	
`*( array + 2 )`	
`*( array + 1) + 4`	
`*( *( array + 1) + 4 )`	
`*( *( *( array + 3 ) + 1 ) + 2 )`	
`*( *( *array + 1) + 2 )`	
`*( **array + 2 )`	
`**( *array + 1 )`	
`***array`	

14. 多维数组的各个下标必须单独出现在一对方括号内。在什么条件下，下列这些代码段可以通过编译而不会产生任何警告或错误信息？

```
int     array[10][20];
...
i = array[3,4];
```

15. 给定下列声明

```
unsigned int   which;
int            array[ SIZE ];
```

下面两条语句哪条更合理？为什么？

```
if(array[ which ] == 5 && which < SIZE ) ...
if( which < SIZE && array[ which ] == 5 )...
```

16. 在下面的代码中，变量 array1 和 array2 有什么区别（如果有的话）？

```
void function( int array1[10] ){
        int     array2[10];
        ...
}
```

17. 解释下面两种 const 关键字用法的显著区别所在。

```
void function( int const a, int const b[] ) {
```

18. 下面的函数原型可以改写为什么形式，但保持结果不变？

```
void function( int array[3][2][5] );
```

19. 在程序 8.2 的关键字查找例子中，字符指针数组的末尾增加了一个 NULL 指针，

这样我们就不需要知道表的长度。那么，矩阵方案应如何进行修改，使其达到同样的效果呢？写出用于访问修改后的矩阵的 for 语句。

8.8 编程练习

★ 1. 编写一个数组的声明，把数组的某些特定位置初始化为特定的值。这个数组的名字应该叫 char_value，它包含 3×6×4×5 个无符号字符。下面的表中列出的这些位置应该用相应的值进行静态初始化。

位置	值	位置	值	位置	值
1,2,2,3	'A'	1,1,1,1	' '	1,3,2,2	0xf3
2,4,3,2	'3'	1,4,2,3	'\n'	2,2,3,1	'\121'
2,4,3,3	3	2,5,3,4	125	1,2,3,4	'x'
2,1,1,2	0320	2,2,2,2	'\''	2,2,1,1	'0'

那些在上面的表中未提到的位置应该被初始化为二进制值 0（不是字符'0'）。**注意**：应该使用静态初始化，在你的解决方案中不应该存在任何可执行代码！

尽管并非解决方案的一部分，你很可能想编写一个程序，通过打印数组的值来验证它的初始化。由于某些值并不是可打印的字符，所以请把这些字符用整型的形式打印出来（用八进制或十六进制输出会更方便一些）。

注意：用两种方法解决这个问题，一次在初始化列表中使用嵌套的花括号，另一次则不使用，这样你就能深刻地理解嵌套花括号的作用。

★★ 2. 美国联邦政府使用下面这些规则计算 1995 年每个公民的个人收入所得税：

如果你的含税收入大于	但不超过	你的税额为	超过这个数额的部分
$0	$23,350	15%	$0
23 350	56,550	$3 502.50+28%	23 350
56 550	117,950	12 798.50+31%	56 550
117 950	256,500	31 832.50+36%	117 950
256 500	—	81 710.50+39.6%	256 500

为下面的函数原型编写函数定义：

```
float single_tax( float income );
```

参数 income 表示应征税的个人收入，函数的返回值就是 income 应该征收的税额。

★★ 3. 单位矩阵(identity matrix)就是一个正方形矩阵，它除了主对角线的元素值为 1 以后，其余元素的值均为 0。例如：

```
1  0  0
0  1  0
0  0  1
```

就是一个 3×3 的单位矩阵。编写一个名叫 identity_matrix 的函数，它接受一个 10×10 整型矩阵为参数，并返回一个布尔值，提示该矩阵是否为单位矩阵。

★★★ 4. 修改前一个问题中的 identity_matrix 函数，它可以对数组进行扩展，从而能够接受

任意大小的矩阵参数。函数的第 1 个参数应该是一个整型指针，你需要第 2 个参数，用于指定矩阵的大小。

★★★★★5. 如果 A 是个 x 行 y 列的矩阵，B 是个 y 行 z 列的矩阵，把 A 和 B 相乘，其结果将是另一个 x 行 z 列的矩阵 C。这个矩阵的每个元素是由下面的公式决定的：

$$Ci,j = \sum_{k=1}^{y} Ai,k \times Bk,j$$

例如：

$$\begin{bmatrix} 2 & -6 \\ 3 & 5 \\ 1 & -1 \end{bmatrix} \times \begin{bmatrix} 4 & -2 & -4 & -5 \\ -7 & -3 & 6 & 7 \end{bmatrix} = \begin{bmatrix} 50 & 14 & -44 & -52 \\ -23 & -21 & 18 & 20 \\ 11 & 1 & -10 & -12 \end{bmatrix}$$

结果矩阵中 14 这个值是通过 2×-2 加上-6×-3 得到的。

编写一个函数，用于执行两个矩阵的乘法。函数的原型应该如下：

```
void matrix_multiply( int *m1, int *m2, int *r,
          int x,  int y, int z );
```

m1 是一个 x 行 y 列的矩阵，m2 是一个 y 行 z 列的矩阵。这两个矩阵应该相乘，结果存储于 r 中，它是一个 x 行 z 列的矩阵。记住，你应该对公式作些修改，以适应 C 语言下标从 0 而不是 1 开始这个事实！

★★★★★ 6. 如你所知，C 编译器为数组分配下标时总是从 0 开始。而且当程序使用下标访问数组元素时，它并不检查下标的有效性。在这个项目中，你将要编写一个函数，允许用户访问“伪数组”，它的下标范围可以任意指定，并伴以完整的错误检查。

下面是你将要编写的这个函数的原型：

```
int array_offset ( int arrayinfo[], ... );
```

这个函数接受一些用于描述伪数组的维数的信息以及一组下标值。然后它使用这些信息把下标值翻译为一个整数，用于表示一个向量（一维数组）的下标。使用这个函数，用户既可以以向量的形式分配内存空间，也可以使用 malloc 分配空间，但按照多维数组的形式访问这些空间。这个数组之所以被称为“伪数组”是因为编译器以为它是个向量，尽管这个函数允许它按照多维数组的形式进行访问。

这个函数的参数如下：

参　数	含　义
arrayinfo	一个可变长度的整型数组，包含一些关于伪数组的信息。arrayinfo[0]指定伪数组具有的维数，它的值必须在 1 和 10 之间（含 10）。arrayinfo[1]和 arrayinfo[2]给出第 1 维的下限和上限。arrayinfo[3]和 arrayinfo[4]给出第 2 维的下限和上限，以此类推
...	参数列表的可变部分可能包含多达 10 个的整数，用于标识伪数组中某个特定位置的下标值。你必须使用 va_参数宏访问它们。当函数被调用时，arrayinfo[0]参数将会被传递

公式根据下面给出的下标值计算一个数组位置。变量 s_1,s_2 等代表下标参数 s_1,s_2 等。变量 lo_i 和 hi_i 代表下标 s_i 的下限和上限，它们来源于 arrayinfo 参数，其余各维依次类推。变量 loc 表示伪数组的目标位置，它用一个距离伪数组起始位置的整型偏移量表示。对于一维伪数组：

$$loc = s_1 - lo_1$$

对于二维伪数组：

$$loc = (s_1 - lo_1) \times (hi_2 - lo_2 + 1) + s_2 - lo_2$$

对于三维伪数组：

$$loc = [(s_1 - lo_1) \times (hi_2 - lo_2 + 1) + s_2 - lo_2] \times (hi_3 - lo_3 + 1) + s_3 - lo_3$$

对于四维伪数组：

$$loc = \{[(s_1 - lo_1) \times (hi_2 - lo_2 + 1) + s_2 - lo_2] \times (hi_3 - lo_3 + 1) + s_3 - lo_3\} \times (hi_4 - lo_4 + 1) + s_4 - lo_4$$

一直到第 10 维为止，都可以类似地使用这种方法推导出 loc 的值。

你可以假定 arrayinfo 是个有效的指针，传递给 array_offset 的下标参数值也是正确的。对于其他情况，你必须进行错误检查。可能出现的一些错误有：维的数目不处于 1 和 10 之间；下标小于 low 值；low 值大于其对应的 hign 值等。如果检测到这些或其他一些错误，函数应该返回-1。

提示：把下标参数复制到一个局部数组中。你接着便可以把计算过程以循环的形式编码，对每一维都使用一次循环。

举例：假定 arrayinfo 包含值 3,4,6,1,5,-3 和 3。这些值提示我们所处理的是三维伪数组。第 1 个下标范围从 4 到 6，第 2 个下标范围从 1 至 5，第 3 个下标范围从-3 到 3。在这个例子中，array_offset 被调用时将有 3 个下标参数传递给它。下面显示了几组下标值以及它们所代表的偏移量。

下标	偏移量	下标	偏移量	下标	偏移量
4, 1, -3	0	4, 1, 3	6	5, 1, -3	35
4, 1, -2	1	4, 2, -3	7	6, 3, 1	88

★★★ 7. 修改问题 6 的 array_offset 函数，使它访问以列主序存储的伪数组，也就是最左边的下标率先变化。这个新函数，array_offset2，在其他方面应该与原先那个函数一样。计算这些数组下标的公式如下所示。对于一维伪数组：

$$loc = s_1 - lo_1$$

对于二维伪数组：

$$loc = (s_2 - lo_2) \times (hi_1 - lo_1 + 1) + s_1 - lo_1$$

对于三维伪数组：

$$loc = [(s_3 - lo_3) \times (hi_2 - lo_2 + 1) + s_2 - lo_2] \times (hi_1 - lo_1 + 1) + s_1 - lo_1$$

对于四维伪数组：

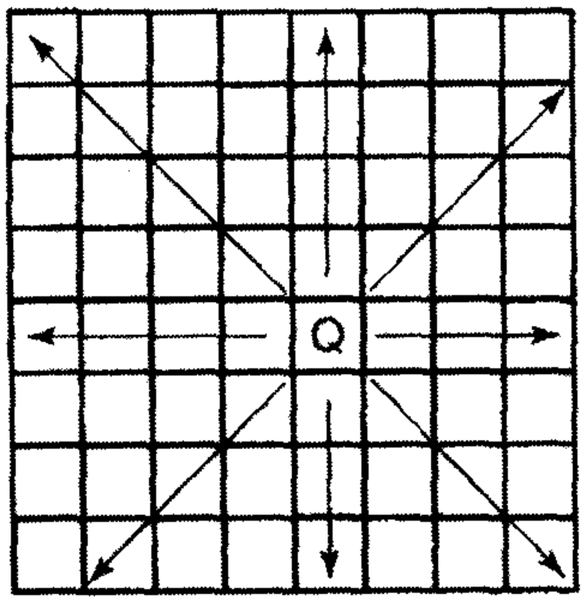
$$loc = \{[(s_4 - lo_4) \times (hi_3 - lo_3 + 1) + (s_3 - lo_3)] \times (hi_2 - lo_2 + 1) + s_2 - lo_2\} \times (hi_1 - lo_1 + 1) + s_1 - lo_1$$

一直到第 10 维为止，都可以类似地使用这种方法推导出 loc 的值。

例如：假定 arrayinfo 数组包含了值 3,4,6,1,5,-3 和 3。这些值提示我们所处理的是三维伪数组。第 1 个下标范围从 4 到 6，第 2 个下标范围从 1 至 5，第 3 个下标范围从-3 到 3。在这个例子中，array_offset 被调用时将有 3 个下标参数传递给它。下面显示了几组下标值以及它们所代表的偏移量。

下标	偏移量	下标	偏移量	下标	偏移量
4,1,-3	0	4,2,-3	3	4,1,-1	30
5,1,-3	1	4,3,-3	6	5,3,-1	37
6,1,-3	2	4,1,-2	15	6,5,3	104

★★★★★ 8. 皇后是国际象棋中威力最大的棋子。在下面所示的棋盘上，皇后可以攻击位于箭头所覆盖位置的所有棋子。



我们能不能把 8 个皇后放在棋盘上，它们中的任何一个都无法攻击其余的皇后？这个问题被称为八皇后问题。你的任务是编写一个程序，找到八皇后问题的所有答案，看看一共有多少种答案。

提示：如果你采用一种叫做**回溯法**(backtracking)的技巧，就很容易编写出这个程序。编写一个函数，把一个皇后放在某行的第 1 列，然后检查它是否与棋盘上的其他皇后互相攻击。如果存在互相攻击，函数把皇后移到该行的第 2 列再进行检查。如果每列都存在互相攻击的局面，函数就应该返回。

但是，如果皇后可以放在这个位置，函数接着应该递归地调用自身，把一个皇后放在下一行。当递归调用返回时，函数再把原先那个皇后移到下一列。当一个皇后成功地放置于最后一行时，函数应该打印出棋盘，显示 8 个皇后的位置。