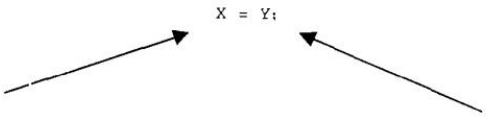
### 4.3.1 数组和指针是如何访问的

本节我们讲述对数组的引用和对指针的引用有何不同之处。首先需要注意的是“地址 y”和“地址 y 的内容”之间的区别。这是一个相当微妙之处，因为在大多数编程语言中我们用同一个符号来表示这两样东西，由编译器根据上下文环境判断它的具体含义。以一个简单的赋值为例，见图 4-1。




在这个上下文环境里，符号 X 的含义是 X 所代表的地址。

这被称为左值。

左值在编译时可知，左值表示存储结果的地方。

在这个上下文环境里，符号 Y 的含义是 Y 所代表的地址的内容。

这被称为右值。

右值直到运行时才知。如无特别说明，右值表示“Y 的内容”。

C 语言引入了“可修改的左值”这个术语。它表示左值允许出现在赋值语句的左边。这个奇怪的术语是为与数组名区分，数组名也用于确定对象在内存中的位置，也是左值，但它不能作为赋值的对象。因此，数组名是个左值但不是可修改的左值。标准规定赋值符必须用可修改的左值作为它左边一侧的操作数。用通俗的话说，只能给可以修改的东西赋值。

出现在赋值符左边的符号有时被称为左值（由于它位于“左手边”或“表示地点”），出现在赋值符右边的符号有时则被称为右值（由于它位于“右手边”）。编译器为每个变量分配一个地址（左值）。这个地址在编译时可知，而且该变量在运行时一直保存于这个地址。相反，存储于变量中的值（它的右值）只有在运行时才可知。如果需要用到变量中存储的值，编译器就发出指令从指定地址读入变量值并将它存于寄存器中。

这里的关键之处在于每个符号的地址在编译时可知。所以，如果编译器需要一个地址（可能还需要加上偏移量）来执行某种操作，它就可以直接进行操作，并不需要增加指令首先取得具体的地址。相反，对于指针，必须首先在运行时取得它的当前值，然后才能对它进行解除引用操作（作为以后进行查找的步骤之一）。图 A 展示了对数组下标的引用。

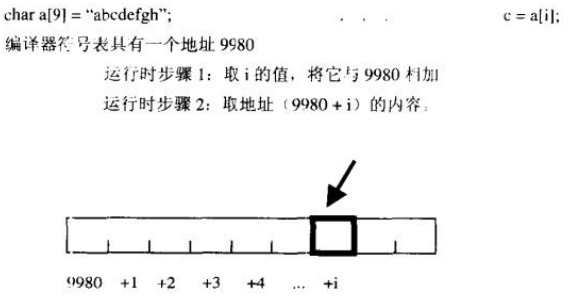



图 A　数组的下标引用

这就是为什么 extern char a[]与 extern char a[100]等价的原因。这两个声明都提示 a 是一个数组，也就是一个内存地址，数组内的字符可以从这个地址找到。编译器并不需要知道数组总共有多少长，因为它只产生偏离起始地址的偏移地址。从数组提取一个字符，只要简单地从符号表显示的 a 的地址加上下标，需要的字符就位于这个地址中。

相反，如果声明 extern char *p, 它将告诉编译器 p 是一个指针（在许多现代的机器里它是个四字节的对象），它指向的对象是一个字符。为了取得这个字符，必须得到地址 p 的内容，把它作为字符的地址并从这个地址中取得这个字符。指针的访问要灵活得多，但需要增加一次额外的提取，如图 B 所示。

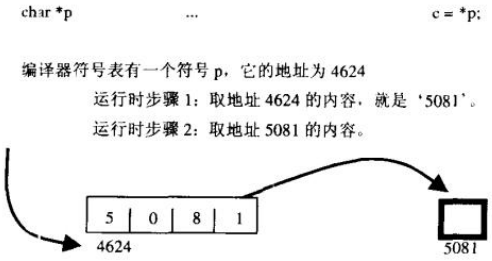



图 B　对指针的引用



### 4.3.2　当你“定义为指针，但以数组方式引用”时会发生什么

现在让我们看一下当一个外部数组的实际定义是一个指针，但却以数组的方式对其引用时，会引起什么问题。需要对内存进行直接的引用（如图 A 所示），但这时编译器所执行的却是对内存进行间接引用（如图 B 所示）。之所以会如此，是因为我们告诉编译器我们拥有的是一个指针，如图 C 所示。

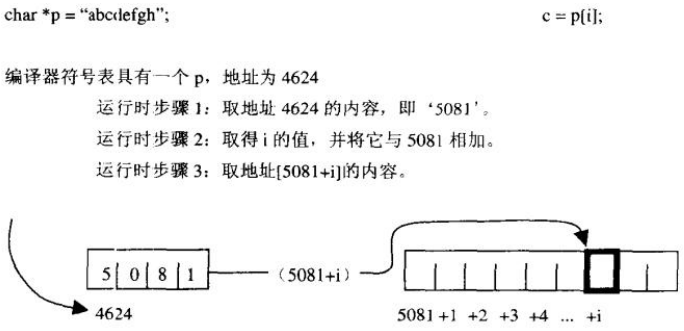



图 C　对指针进行下标引用

对照图 C 的访问方式：

```
char *p = "abcdefgh"; ... p[3]
```

和图 A 的访问方式：

```
char a[] = "abcdefgh"; ... a[3]
```

在这两种情况下，都可以取得字符'd'，但两者的途径非常不一样。

当书写了 extern char *p，然后用 p[3]来引用其中的元素时，其实质是图 A 和图 B 访问方式的组合。首先，进行图 B 所示的间接引用。然后，如图 A 所示用下标作为偏移量进行直接访问。更为正式的说法是，编译器将会：

1. 取得符号表中 p 的地址，提取存储于此处的指针。
2. 把下标所表示的偏移量与指针的值相加，产生一个地址。
3. 访问上面这个地址，取得字符。

编译器已被告知 p 是一个指向字符的指针（相反，数组定义告诉编译器 p 是一个字符序列）。p[i]表示“从 p 所指的地址开始，前进 i 步，每步都是一个字符（即每个元素的长度为一个字节）”。如果是其他类型的指针（如 int 或 double 等），其步长（每步的字节数）也各不相同。

既然把 p 声明为指针，那么不管 p 原先是定义为指针还是数组，都会按照上面所示的三个步骤进行操作，但是只有当 p 原来定义为指针时这个方法才是正确的。考虑一下 p 在这里被声明为 extern char *p;而它原先的定义却是 char p[10];这种情形。当用 p[i]这种形式提取这个声明的内容时，实际上得到的是一个字符。但按照上面的方法，编译器却把它当成是一个指针，把 ACSII 字符解释为地址显然是牛头不对马嘴。如果此时程序当掉，你应该额手称庆。否则的话，它很可能会污染程序地址空间的内容，并在将来出现莫名其妙的错误。

## 4.5 数组和指针的其他区别

比较数组和指针的另外一个方法就是对比两者的特点，见表 4-1。

表 4-1 数组和指针的区别

| 指　针 | 数　组 |
| --- | --- |
| 保存数据的地址 | 保存数据 |
| 间接访问数据，首先取得指针的内容，把它作为地址，然后从这个地址提取数据。<br>如果指针有一个下标[I]，就把指针的内容加上 I 作为地址，从中提取数据 | 直接访问数据，a[I]只是简单地以 a-I 为地址取得数据 |

续表

| 指　针 | 数　组 |
| --- | --- |
| 通常用于动态数据结构 | 通常用于存储固定数目且数据类型相同的元素。 |
| 相关的函数为 malloc(), free()。 | 隐式分配和删除 |
| 通常指向匿名数据 | 自身即为数据名 |

数组和指针都可以在它们的定义中用字符串常量进行初始化。尽管看上去一样，底层的机制却不相同。

定义指针时，编译器并不为指针所指向的对象分配空间，它只是分配指针本身的空间，除非在定义时同时赋给指针一个字符串常量进行初始化。例如，下面的定义创建了一个字符串常量（为其分配了内存）：

```
char *p = "breadfruit";
```

注意只有对字符串常量才是如此。不能指望为浮点数之类的常量分配空间，如：

```
float *pip = 3.141;     /* 错误！无法通过编译。 */
```

在 ANSI C 中，初始化指针时所创建的字符串常量被定义为只读。如果试图通过指针修改这个字符串的值，程序就会出现未定义的行为。在有些编译器中，字符串常量被存放在只允许读取的文本段中，以防止它被修改。

数组也可以用字符串常量进行初始化：

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

与指针相反，由字符串常量初始化的数组是可以修改的。其中的单个字符在以后可以改变，比如下面的语句：

```
strncpy(a, "black", 5);
```

就将数组的值修改为“blackberry”。

第 9 章讨论指针和数组可以等同的情况，并讨论了为什么有时它们可以相等，其中的机理是怎样的。第 10 章描述了一些基于指针的使用数组的高级技巧。如果能坚持读完那一章，那么，关于数组方面的知识，仅仅是你忘掉的内容也可能比许多 C 程序员总共知道的内容还要多。

指针是 C 语言中最难正确理解和使用的部分之一，可能只有声明的语法比它更烦了。然而，它们也是 C 语言中最重要的部分之一。专业 C 程序员必须熟练掌握 malloc()函数，并且学会用指针操纵匿名内存。

## 9.1 什么时候数组与指针相同

第 4 章着重强调了数组和指针并不一致的绝大多数情形。本章的开始部分就是讲述可以把它们看作是相同的情形。在实际应用中，数组和指针可以互换的情形要比两者不可互换的情形更为常见。让我们分别考虑“声明”和“使用”（使用它们传统的直接含义）这两种情况。

声明本身还可以进一步分成 3 种情况：

- 外部数组(external array)的声明。
- 数组的定义（记住，定义是声明的一种特殊情况，它分配内存空间，并可能提供一个初始值）。
- 函数参数的声明。

所有作为函数参数的数组名总是可以通过编译器转换为指针。在其他所有情况下（最有趣的情况就是“在一个文件中定义为数组，在另一个文件中声明为指针”，第 4 章已有所描述），数组的声明就是数组，指针的声明就是指针，两者不能混淆。但在使用数组（在语句或表达式中引用）时，数组总是可以写成指针的形式，两者可以互换。图 9-1 对这些情况作了总结。

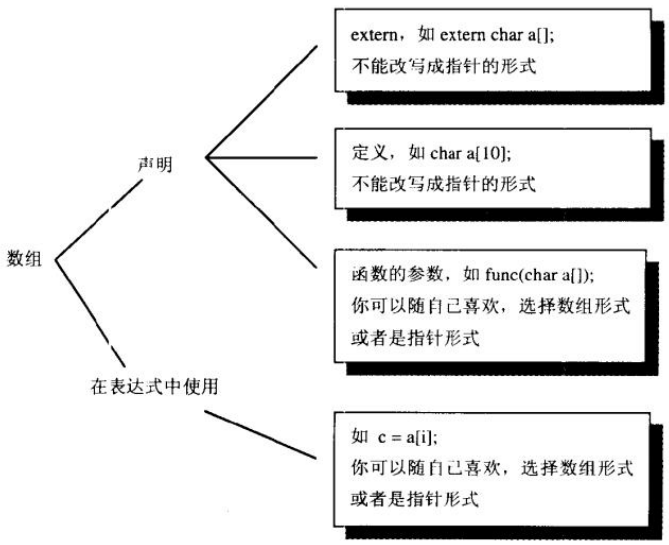



图 9-1 什么时候数组和指针相同

然而，数组和指针在编译器处理时是不同的，在运行时的表示形式也是不一样的，并可能产生不同的代码。对编译器而言，一个数组就是一个地址，一个指针就是一个地址的地址。你应该根据情况做出选择。

## 9.2 为什么会发生混淆

为什么人们会错误地认为数组和指针是可以完全互换的呢？这是因为他们阅读了标准的参考文献！

*The C Programming Language*，第二版，Kernighan & Ritchie，第 99 页的底部是：

As format parameters in a function definition（作为函数定义的形式参数），

然后翻到第 100 页，紧接前句：

```
      char s[];
and（和）
      char* s;
are equivalent（是一样的；）...
```

呜呼！真是不幸，这么重要的一句话竟然在 K&R 第二版中被分印在两页上！人们在阅读后一句话时，很容易忘掉它的前面还有一句“作为函数定义的形式参数”（也就是说它只限于这种情况），尤其是整句话的重点在于“数组下标表达式总是可以改写为带偏移量的指针表达式”。

**软件信条**

**什么时候数组和指针是相同的**

C 语言标准对此作了如下说明：

规则 1. 表达式中的数组名（与声明不同）被编译器当作一个指向该数组第一个元素的





指针[1]（具体释义见 ANSI C 标准第 6.2.2.1 节）。

规则 2. 下标总是与指针的偏移量相同（具体释义见 ANSI C 标准第 6.3.2.1 节）。

规则 3. 在函数参数的声明中，数组名被编译器当作指向该数组第一个元素的指针（具体释义见 ANSI C 标准第 6.7.1 节）。

简而言之，数组和指针的关系颇有点像诗和词的关系：它们都是文学形式之一，有不少共同之处，但在实际的表现手法上又各有特色。下面几个小节将详细描述这几个规则的实际含义。

### 9.2.1 规则 1："表达式中的数组名"就是指针

上面的规则 1 和规则 2 合在一起理解，就是对数组下标的引用总是可以写成"一个指向数组的起始地址的指针加上偏移量"。例如，假如我们声明：

```
int a[10], *p, i = 2;
```

就可以通过以下任何一种方法来访问 a[i]：

```
p = a;
p[i];
```

```
p = a;
*(p + i);
```

```
p = a + i;
*p;
```

事实上，可以采用的方法更多。对数组的引用如 a[i]在编译时总是被编译器改写成*(a+i)的形式。C 语言标准要求编译器必须具备这个概念性的行为。也许遵循这个规则的捷径就是记住方括号[]表示一个取下标操作符，就像加号表示一个加法运算符一样。取下标操作符取一个整数和一个指向类型 T 的指针，所产生的结果类型是 T，一个在表达式中的数组名于是就成了指针。你只要记住：在表达式中，指针和数组是可以互换的，因为它们在编译器里的最终形式都是指针，并且都可以进行取下标操作。就像加法一样，取下标操作符的操作数是可以交换的（它并不在意操作数的先后顺序，就像在加法中 3+5 和 5+3 并没有什么不一样）。这就是为什么在一个 a[10]的声明中下面两种形式都是正确的：

```
a[6] = ...;
6[a] = ...;
```

在实际的产品代码中，上面第二种形式从来不曾使用。确实，它除了可以把新手搞晕之外，实在没有什么实际意义。

编译器自动把下标值的步长调整到数组元素的大小。如果整型数的长度是 4 个字节，那么 a[i+1]和 a[i]在内存中的距离就是 4（而不是 1）。对起始地址执行加法操作之前，编译器会负责计算每次增加的步长。这就是为什么指针总是有类型限制，每个指针只能指向一种类型的原因所在——因为编译器需要知道对指针进行解除引用操作时应该取几个字节，以及每个下标的步长应取几个字节。

### 9.2.2 规则 2：C 语言把数组下标作为指针的偏移量

把数组下标作为指针加偏移量是 C 语言从 BCPL（C 语言的祖先）继承过来的技巧。在人们的常规思维中，在运行时增加对 C 语言下标的范围检查是不切实际的。因为取下标操作只是表示将要访问该数组，但并不保证一定要访问。而且，程序员完全可以使用指针来访问数组，从而绕过下标操作符。在这种情况下，数组下标范围检测并不能检测所有对数组的访问的情况。事实上，下标范围检测被认为并不值得加入到 C 语言中。

还有一种说法是，在编写数组算法时，使用指针比使用数组"更有效率"。

这个颇为人们所接受的说法在通常情况下是错误的。使用现代的产品质量优化的编译器，一维数组和指针引用所产生的代码并不具有显著的差别。不管怎样，数组下标是定义在指针的基础上的，所以优化器常常可以把它转换为更有效率的指针表达形式，并生成相同的机器指令。让我们再看一下数组/指针这两种方案，并把初始化从循环内部的访问中分离出来。

```
int a[10], *p, i;
```

变量 a[i]可以用图 9-2 所示的各种方法来访问，效果完全一样。

即使编译器使用的是较原始的翻译方法，两者产生不一样的代码，用指针迭代一个一维数组常常也并不比直接使用下标迭代一个一维数组来得更快。不论是指针还是数组，在连续的内存地址上移动时，编译器都必须计算每次前进的步长。计算的方法是偏移量乘以每个数组元素占用的字节数，计算结果就是偏移数组起始地址的实际字节数。步长因子常常是 2 的乘方（如 int 是 4 个字节，double 是 8 个字节等），这样编译器在计算时就可以使用快速的左移位运算，而不是相对缓慢的加法运算。一个二进制数左移 3 位相当于它乘以 8。如果数组中的元素的大小不是 2 的乘方（如数组的元素类型是一个结构），那就不能使用这个技巧了。

然而，迭代一个 int 数组是人们最容易想到的。如果一个经过良好优化的编译器进行代码分析，并把基本变量放在高速的寄存器中来确认循环是否继续，那么最终在循环中访问指针和数组所产生的代码很可能是相同的。

在处理一维数组时，指针并不见得比数组更快。C 语言把数组下标改写成指针偏移量的根本原因是指针和偏移量是底层硬件所使用的基本模型。

**数组访问**

```
for(i = 0; i < 10; i++)
        a[i] = 0;
```

**中间代码**

把左值（a）装入 R1（可以提到循环外）
把左值（i）装入 R2（可以提到循环外）
把[R2]装入 R3
如果需要，对 R3 的步长进行调整把 R1+R3 的结果装入 R4 中
把 0 存储到[R4]。

**指针备选方案 1**

```
p = a;
for(i = 0; i < 10; i++)
        p[i] = 0;
```

把左值（p）装入 R0（可以提到循环外）
把[R0]装入 R1（可以提到循环外）
把左值（i）装入 R2（可以提到循环外）
把[R2]装入 R3
如果需要，对 R3 的步长进行调整
把 R1+R3 的结果装入 R4 中
把 0 存储到[R4]。

**指针备选方案 2**

```
p = a;
for(i = 0; i < 10; i++)
        *(p + i) = 0;
```

与指针备选方案 1 相同
（想一想，为什么？）

**指针备选方案 3**

```
p = a;
for(i = 0; i < 10; i++)
        *p++ = 0;
```

把 p 所指对象的大小装入 R5
(可以提到循环外)
把左值（p）装入 R1（可以提到循环外）
把[R0]装入 R1
把 0 存储到[R1]
把 R5+R1 的结果装入 R1
把 R1 存储到[R0]

上面这些例子显示了不同的备选方案经过翻译后所产生的中间代码。如果采用优化措施，中间代码可能跟这里显示的不一样。R0、R1 等代表 CPU 的寄存器。在图 9-2 中，我们用

R0 存储 p 的左值　　R1 存储 a 的左值或 p 的右值

R2 存储 i 的左值　　R3 存储 i 的右值

[R0]表示间接载入或写入，其地址就是寄存器的内容（这是许多汇编语言所使用的一个普通概念）。“可以提到循环外”表示这个数据不会被循环修改，在每次循环时可不必执行该语句，可以加快循环的速度。

### 9.2.3 “作为函数参数的数组名”等同于指针

规则 3 也需要进行解释。首先，让我们回顾一下 *The C Programming Language* 中所提到的一些术语。

| 术　语 | 定　义 | 例　子 |
|---|---|---|
| 形参(parameter) | 它是一个变量，在函数定义或函数声明的原型中定义。又称“形式参数（formal parameter)” | int power(int base, int n);<br>base 和 n 都是形参 |
| 实参(argument) | 在实际调用一个函数时所传递给函数的值。又称“实际参数(actual paramenter)” | i = power(10, j);<br>10 和 j 都是实参。在同一个函数的多次调用时，实参可以不同 |

标准规定作为“类型的数组”的形参的声明应该调整为“类型的指针”。在函数形参定义这个特殊情况下，编译器必须把数组形式改写成指向数组第一个元素的指针形式。编译器只向函数传递数组的地址，而不是整个数组的拷贝。不过，现在让我们重点观察一下数组，隐性转换意味着三种形式是完全等同的。因此，在 my_function()的调用上，无论实参是数组还是真的指针都是合法的。

```
my_function(int *turnip)   { ... }
my_function(int turnip[])  { ... }
my_function(int turnip[200]) { ... }
```

## 9.3 为什么 C 语言把数组形参当作指针

之所以要把传递给函数的数组参数转换为指针是出于效率的考虑，这个理由常常也是对违反软件工程做法的辩解。Fortran 的 I/O 模型使用起来相当麻烦，因为它必须“有效地”复用现有的 IBM 704 汇编程序 I/O 库（尽管相当笨拙，而且已经过时)。全面的语义检查被可移植的 C 编译器所排斥，其理由很牵强，他们认为把 lint 程序作为一个单独的程序，“效率”会更高一些。大多数现代的 ANSI C 编译器在错误检查方面都作了增强，也算是对这个决定的不认同吧。

把作为形参的数组和指针等同起来是出于效率原因的考虑。在 C 语言中，所有非数组形式的数据实参均以传值形式（对实参作一份拷贝并传递给调用的函数，函数不能修改作为实参的实际变量的值，而只能修改传递给它的那份拷贝）调用。然而，如果要拷贝整个数组，无论在时间上还是在内存空间上的开销都可能是非常大的。而且在绝大部分情况下，你其实并不需要整个数组的拷贝，你只想告诉函数在那一时刻对哪个特定的数组感兴趣。要达到这个目的，可以考虑的方法是在形参上增加一个存储说明符(storage specifier)，表示它是传值调用还是传址调用，Pascal 语言就是这样做的。如果采用“所有的数组在作为参数传递时都转

换为指向数组起始地址的指针，而其他的参数均采用传值调用”的约定，就可以简化编译器。类似地，函数的返回值绝不能是一个函数数组，而只能是指向数组或函数的指针。

有些人喜欢把它理解成除数组和函数之外的所有的C语言参数在缺省情况下都是传值调用，数组和函数则是传址调用。数据也可以使用传址调用，只要在它前面加上取地址操作符(&)，这样传递给函数的是实参的地址而不是实参的拷贝。事实上，取地址操作符的主要用途就是实现传址调用。“传址调用”这个说法从严格意义上说并不十分准确，因为编译器的机制非常清楚——在被调用的函数中，你只拥有一个指向变量的指针而不是变量本身。如果你取实参的地址或对它进行拷贝，就能体会到两者的差别。

### 数组形参是如何被引用的

图 9-3 展示了对一个下标形式的数组形参进行访问所需要的几个步骤。

```
func(char p[]);          ...          c = p[i];
func(char *p);           ...          c = p[i];
```

编译器符号表显示 p 可以取址，从堆栈指针 SP 偏移 14 个位置

运行时步骤 1：从 SP 偏移 14 个位置找到函数的活动记录，取出实参。

运行时步骤 2：取 i 的值，并与 5081 相加。

运行时步骤 3：取出地址(5081+i)的内容。

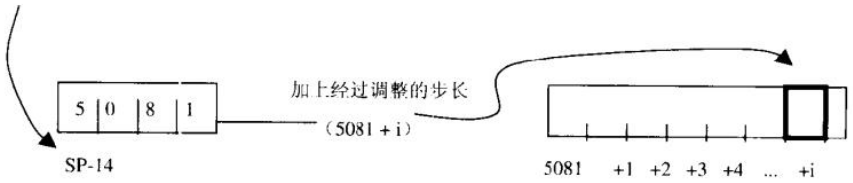



图 9-3　下标形式的数组形参是如何引用的

注意它和第 4 章图 C 一样，图 C 显示的是一个下标形式的指针是如何查找地址的。C 语言允许程序员把形参声明为数组（程序员打算传递给函数的东西）或者指针（函数实际所接收到的东西）。编译器知道何时形参是作为数组声明的，但事实上在函数内部，编译器始终把它当作一个指向数组第一个元素（元素长度未知）的指针。这样，编译器可以产生正确的代码，并不需要对数组和指针这两种情况作仔细区分。

不管程序员实际所写的是哪种形式，函数并不自动知道指针所指的数组共有多少个元素，所以必须要有个约定，如数组以 NUL 结尾或者另有一个附加的参数表示数组的范围。当然并不是每种语言都是这样做的，比如 Ada，它的每个数组都有一些附加信息，表示每个元素的长度、数组的维数以及下标范围。

在下列定义中：

```
func(int * tun:p) {  ...  }
```

或

```
func(int turnip[]) { ... }
```

或

```
func(int turnip[200]{ ... }
```

```
int my_int;   /* 数据定义 */
int *my_int_ptr;
int my_int_array[10];
```

你可以合法地使用下列任何一个实参来调用上面任何一个原型的函数。它们常常用于不同的目的：

**表 9-1　　数组/指针实参的一般用法**

| 调用时的实参 | 类　型 | 通常目的 |
|---|---|---|
| `func(&my_int);` | 一个整型数的地址 | 一个 int 参数的传址调用 |
| `func(my_int_ptr);` | 指向整型数的指针 | 传递一个指针 |
| `func(my_int_array);` | 整型数组 | 传递一个数组 |
| `func(&my_int_array[i]);` | 一个整型数组某个元素的地址 | 传递数组的一部分 |

相反，如果处于 func()函数内部，就没有一种容易的方法分辨这些不同的实参，因此也无法知道调用该函数是出于何种目的。所有属于函数实参的数组在编译时被编译器改写为指针。因此，在函数内部对数组参数的任何引用都将产生一个对指针的引用。图 9-3 显示了它的实际操作过程。

因此，很有意思的是，没有办法把数组本身传递给一个函数，因为它总是被自动转换为指向数组的指针。当然，在函数内部使用指针，所能进行的对数组的操作几乎跟传递原原本本的数组没有差别。只不过，如果想用 sizeof（实参）来获得数组的长度，所得到的结果不正确而已。

这样，在声明这样一个函数时，你就有了选择余地。可以把形参定义成数组，也可以定义成指针。不论你选择什么，编译器都会注意到该对象是一个函数参数的特殊情况，它会产生代码对该指针进行解除引用操作。

如果你想让代码看上去清楚明白，就必须遵循一定的规则！我们倾向于始终把参数定义为指针，因为这是编译器内部所使用的形式。如果名不副实，那就是一种很可疑的编程风格。但从另一方面看，有些人觉得 int table[]比 int *table 更能表达程序员的意图。table[]这种记法清楚地表明了 table 内里有好几个元素，提示函数会对它们都进行处理。

注意，有一样操作只能在指针里进行而无法在数组中进行，那就是修改它的值。数组名是不可修改的左值，它的值是不能改变的。见图 9-4（几个函数并排放在一起以便比较，它们都是同一个文件的一部分）。

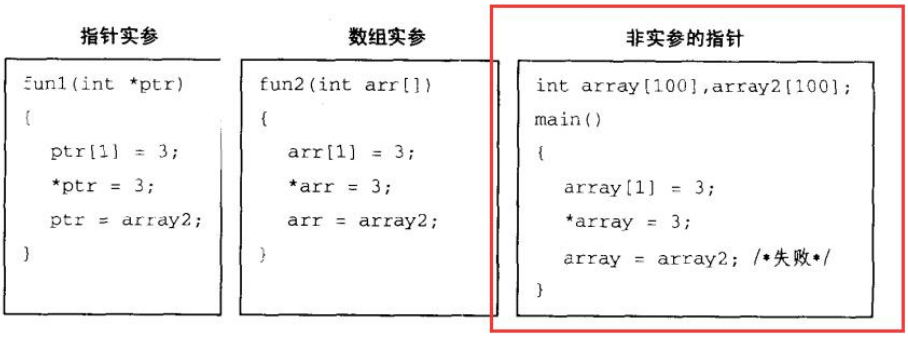



图 9-4　数组实参的有效操作

语句 array = array2;将引起一个编译时错误，错误信息是“无法修改数组名”。但是，arr = array2 却是合法的，因为 arr 虽然声明为一个数组但实际上却是一个指针。

## 9.5　数组和指针可交换性的总结

警告：在你阅读并理解前面的章节之前不要阅读这一节的内容，因为它可能会使你的脑力永久退化。

1. 用 a[i]这样的形式对数组进行访问总是被编译器“改写”或解释为像*(a+1)这样的指针访问。

2. 指针始终就是指针。它绝不可以改写成数组。你可以用下标形式访问指针，一般都是指针作为函数参数时，而且你知道实际传递给函数的是一个数组。

3. 在特定的上下文中，也就是它作为函数的参数（也只有这种情况），一个数组的声明可以看作是一个指针。作为函数参数的数组（就是在一个函数调用中）始终会被编译器修改成为指向数组第一个元素的指针。

4. 因此，当把一个数组定义为函数的参数时，可以选择把它定义为数组，也可以定义指针。不管选择哪种方法，在函数内部事实上获得的都是一个指针。

5. 在其他所有情况中，定义和声明必须匹配。如果定义了一个数组，在其他文件对它进行声明时也必须把它声明为数组，指针也是如此。