

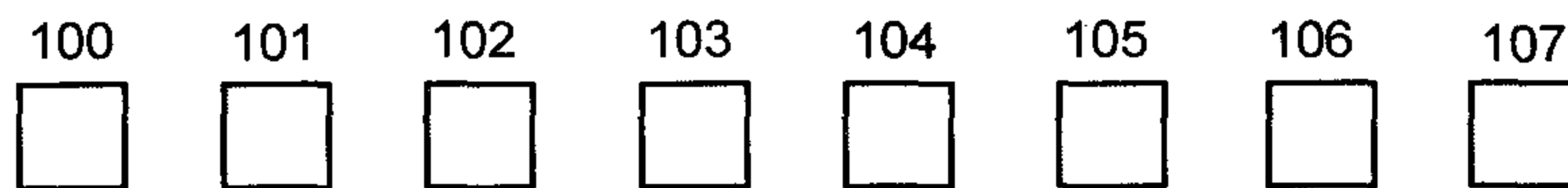
# 指针

是详细讨论指针的时候了，因为在本书的剩余部分，我们将会频繁地使用指针。你可能已经熟悉了本章所讨论的部分或全部背景信息。但是，如果你对此尚不熟悉，请认真学习，因为你对指针的理解将建立在这个基础之上。

## 6.1 内存和地址

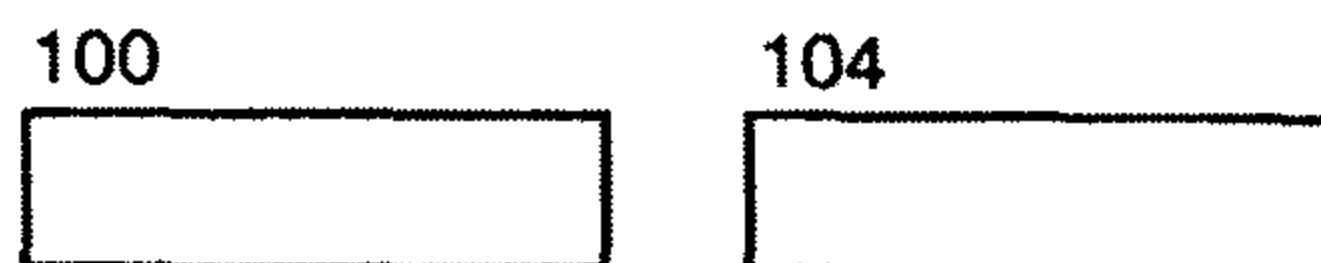
我在前面提到过，我们可以把计算机的内存看作是一条长街上的一排房屋。每座房子都可以容纳数据，并通过一个房号来标识。

这个比喻颇为有用，但也存在局限性。计算机的内存由数以亿万计的位（bit）组成，每个位可以容纳值 0 或 1。由于一个位所能表示的值的范围太有限，所以单独的位用处不大，通常许多位合成一组作为一个单位，这样就可以存储范围较大的值。这里有一幅图，展示了现实机器中的一些内存位置。



这些位置的每一个都被称为字节（byte），每个字节都包含了存储一个字符所需要的位数。在许多现代的机器上，每个字节包含 8 个位，可以存储无符号值 0 至 255，或有符号值 -128 至 127。上面这张图并没有显示这些位置的内容，但内存中的每个位置总是包含一些值。每个字节通过地址来标识，如上图方框上面的数字所示。

为了存储更大的值，我们把两个或更多个字节合在一起作为一个更大的内存单位。例如，许多机器以字为单位存储整数，每个字一般由 2 个或 4 个字节组成。下面这张图所表示的内存位置与上面这张图相同，但这次它以 4 个字节的字来表示。



由于它们包含了更多的位，每个字可以容纳的无符号整数的范围是从 0 至  $4294967295(2^{32}-1)$ ，可以容纳的有符号整数的范围是从  $-2147483648(-2^{31})$  至  $2147483647(2^{31}-1)$ 。

注意，尽管一个字包含了 4 个字节，它仍然只有一个地址。至于它的地址是它最左边那个字节的位置还是最右边那个字节的位置，不同的机器有不同的规定。另一个需要注意的硬件事项是边界对齐(boundary alignment)。在要求边界对齐的机器上，整型值存储的起始位置只能是某些特定的字节，通常是 2 或 4 的倍数。但这些问题是硬件设计者的事情，它们很少影响 C 程序员。我们只对两件事情感兴趣：

1. 内存中的每个位置由一个独一无二的地址标识。
2. 内存中的每个位置都包含一个值。

## 地址与内容

这里有另外一个例子，这次它显示了内存中 5 个字的内容。

100	104	108	112	116
112	-1	1078523331	100	108

这里显示了 5 个整数，每个都位于自己的字中。如果你记住了一个值的存储地址，你以后可以根据这个地址取得这个值。

但是，要记住所有这些地址实在是太笨拙了，所以高级语言所提供的特性之一就是通过名字而不是地址来访问内存的位置。下面这张图与上图相同，但这次使用名字来代替地址。

a	b	c	d	e
112	-1	1078523331	100	108

当然，这些名字就是我们所称的变量。有一点非常重要，你必须记住，名字与内存位置之间的关联并不是硬件所提供的，它是由编译器为我们实现的。所有这些变量给了我们一种更方便的方法记住地址——硬件仍然通过地址访问内存位置。

## 6.2 值和类型

现在让我们来看一下存储于这些位置的值。头两个位置所存储的是整数。第 3 个位置所存储的是一个非常大的整数，第 4、5 个位置所存储的也是整数。下面是这些变量的声明：

```
int    a = 112, b = -1;
float  c = 3.14;
int    *d = &a;
float  *e = &c;
```

在这些声明中，变量 a 和 b 确实用于存储整型值。但是，它声明 c 所存储的是浮点值。可是，在上图中 c 的值却是一个整数。那么到底它应该是哪个呢？整数还是浮点数？

答案是该变量包含了一序列内容为 0 或者 1 的位。它们可以被解释为整数，也可以被解释为浮点数，这取决于它们被使用的方式。如果使用的是整型算术指令，这个值就被解释为整数，如果使用的是浮点型指令，它就是个浮点数。

这个事实引出了一个重要的结论：不能简单地通过检查一个值的位来判断它的类型。为了判断值的类型（以及它的值），你必须观察程序中这个值的使用方式。考虑下面这个以二进制形式表示的 32 位值：

```
01100111011011000110111101100010
```

下面是这些位可能被解释的许多结果中的几种。这些值都是从一个基于 Motorola 68000 的处理器上得到的。如果换个系统，使用不同的数据格式和指令，对这些位的解释将又有所不同。

类 型	值
1 个 32 位整数	1735159650
2 个 16 位整数	26476 和 28514
4 个字符	glob
浮点数	$1.116533 \times 10^{24}$
机器指令	beg .+110 和 ble .+102

这里，一个单一的值可以被解释为 5 种不同的类型。显然，值的类型并非值本身所固有的一种特性，而是取决于它的使用方式。因此，为了得到正确的答案，对值进行正确的使用是非常重要的。

当然，编译器会帮助我们避免这些错误。如果我们把 c 声明为 float 型变量，那么当程序访问它时，编译器就会产生浮点型指令。如果我们以某种对 float 类型而言不适当的方式访问该变量时，编译器就会发出错误或警告信息。现在看来非常明显，图中所标明的值是具有误导性质的，因为它显示了 c 的整型表示方式。事实上真正的浮点值是 3.14。

6.3 指针变量的内容

让我们把话题返回到指针，看看变量 d 和 e 的声明。它们都被声明为指针，并用其他变量的地址予以初始化。指针的初始化是用&操作符完成的，它用于产生操作数的内存地址（见第 5 章）。

a	b	c	d	e
112	-1	3.14	100	108

d 和 e 的内容是地址而不是整型或浮点型数值。事实上，从图中可以容易地看出，d 的内容与 a 的存储地址一致，而 e 的内容与 c 的存储地址一致，这也正是我们对这两个指针进行初始化时所期望的结果。区分变量 d 的地址(112)和它的内容(100)是非常重要的，同时也必须意识到 100 这个数值用于标识其他位置（是...的地址）。在这一点上，房屋/街道这个比喻不再有效，因为房子的内容绝不可能是其他房子的地址。

在我们转到下一步之前，先看一些涉及这些变量的表达式。请仔细考虑这些声明。

```
int    a = 112, b = -1;
float  c = 3.14;
int    *d = &a;
float  *e = &c;
```

下面这些表达式的值分别是什么呢？

- a
- b
- c
- d

e

前 3 个非常容易：a 的值是 112，b 的值是-1，c 的值是 3.14。指针变量其实也很容易，d 的值是 100，e 的值是 108。如果你认为 d 和 e 的值分别是 112 和 3.14，那么你就犯了一个极为常见的错误。d 和 e 被声明为指针并不会改变这些表达式的求值方式：一个变量的值就是分配给这个变量的内存位置所存储的数值。如果你简单地认为由于 d 和 e 是指针，所以它们可以自动获得存储于位置 100 和 108 的值，那么你就错了。变量的值就是分配给该变量的内存位置所存储的数值，即使是指针变量也不例外。

## 6.4 间接访问操作符

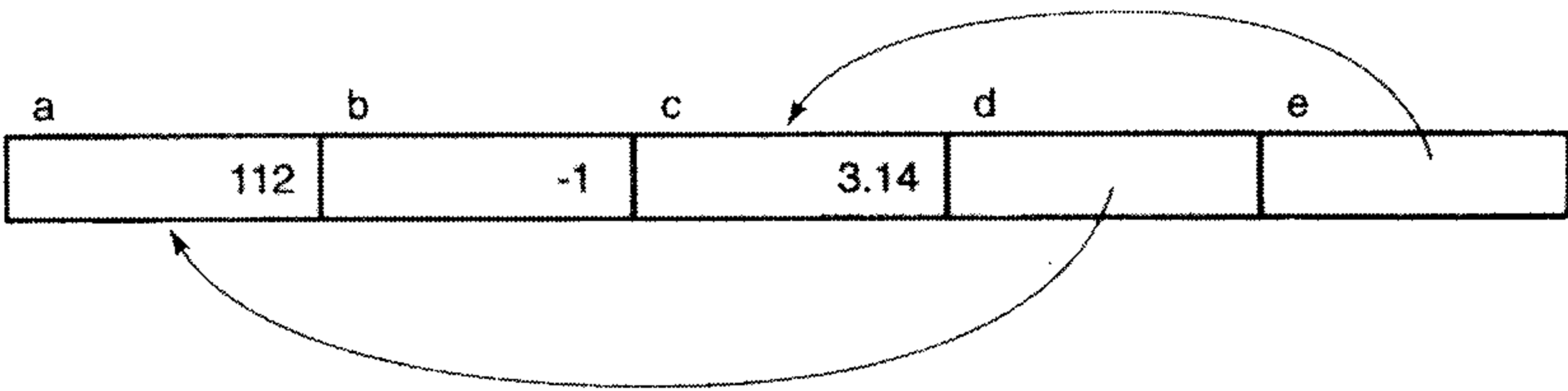
通过一个指针访问它所指向的地址的过程称为间接访问(indirection)或解引用指针(dereferencing the pointer)。这个用于执行间接访问的操作符是单目操作符\*。这里有一些例子，它们使用了前面小节里的一些声明。

表达式	右值	类型
a	112	int
b	-1	int
c	3.14	float
d	100	int *
e	108	float *
*d	112	int
*e	3.14	float

d 的值是 100。当我们对 d 使用间接访问操作符时，它表示访问内存位置 100 并察看那里的值。因此，\*d 的右值是 112——位置 100 的内容，它的左值是位置 100 本身。

注意上面列表中各个表达式的类型：d 是一个指向整型的指针，对它进行解引用操作将产生一个整型值。类似，对 float \*进行间接访问将产生一个 float 型值。

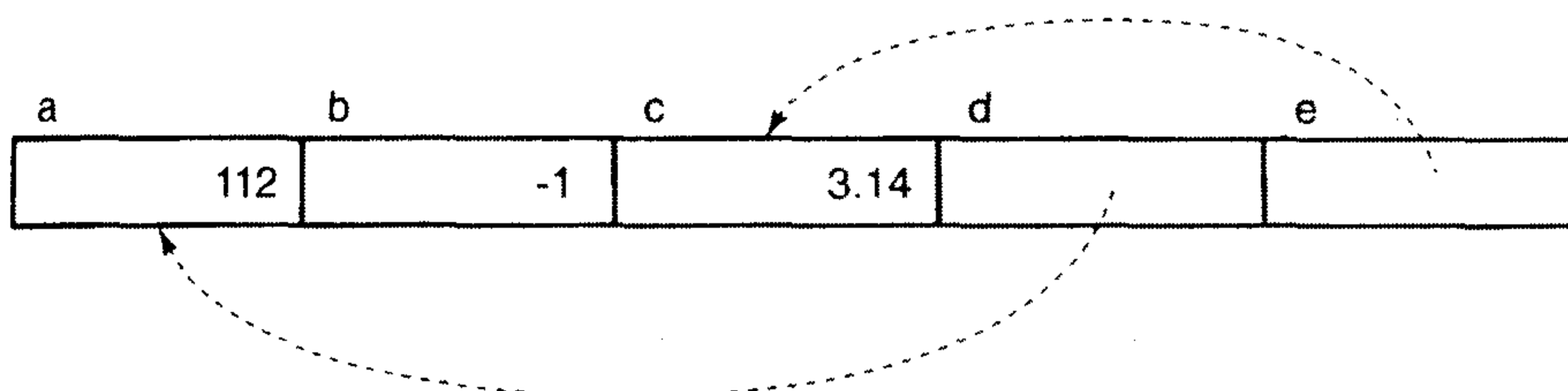
正常情况下，我们并不知道编译器为每个变量所选择的存储位置，所以我们事先无法预测它们的地址。这样，当我们绘制内存中的指针图时，用实际数值表示地址是不方便的。所以绝大部分书籍改用箭头来代替，如下所示：



但是，这种记法可能会引起误解，因为箭头可以会使你误以为执行了间接访问操作，但事实上它并不一定会进行这个操作。例如，根据上图，你会推断表达式 d 的值是什么？

如果你的答案是 112，那么你就被这个箭头误导了。正确的答案是 a 的地址，而不是它的内容。但是，这个箭头似乎会把你的注意力吸引到 a 上。要使你的思维不受箭头影响是不容易的，这也是问题所在：除非存在间接引用操作符，否则不要被箭头所误导。

下面这个修正后的箭头记法试图消除这个问题。



这种记法的意图是既显示指针的值，但又不给你强烈的视觉线索，以为这个箭头是我们必须遵从的路径。事实上，如果不对指针变量进行间接访问操作，它的值只是简单的一些位的集合。当执行间接访问操作时，这种记法才使用实线箭头表示实际发生的内存访问。

注意箭头起始于方框内部，因为它表示存储于该变量的值。同样，箭头指向一个位置，而不是存储于该位置的值。这种记法提示跟随箭头执行间接访问操作的结果将是一个左值。事实也的确如此，我们在以后将看到这一点。

尽管这种箭头记法很有用，但为了正确地使用它，你必须记住指针变量的值就是一个数字。箭头显示了这个数字的值，但箭头记法并未改变它本身就是个数字的事实。指针并不存在内建的间接访问属性，所以除非表达式中存在间接访问操作符，否则你不能按箭头所示实际访问它所指向的位置。

## 6.5 未初始化和非法的指针

下面这个代码段说明了一个极为常见的错误：

```
int    *a;
...
*a = 12;
```

这个声明创建了一个名叫 `a` 的指针变量，后面那条赋值语句把 12 存储在 `a` 所指向的内存位置。

**警告：**

但是究竟 `a` 指向哪里呢？我们声明了这个变量，但从未对它进行初始化，所以我们没有办法预测 12 这个值将存储于什么地方。从这一点看，指针变量和其他变量并无区别。如果变量是静态的，它会被初始化为 0；但如果变量是自动的，它根本不会被初始化。无论是哪种情况，声明一个指向整型的指针都不会“创建”用于存储整型值的内存空间。

所以，如果程序执行这个赋值操作，会发生什么情况呢？如果你运气好，`a` 的初始值会是个非法地址，这样赋值语句将会出错，从而终止程序。在 UNIX 系统上，这个错误被称为“段违例（segmentation violation）”或“内存错误（memory fault）”。它提示程序试图访问一个并未分配给程序的内存位置。在一台运行 Windows 的 PC 上，对未初始化或非法指针进行间接的访问操作是一般保护性异常（General Protection Exception）的根源之一。

对于那些要求整数必须存储于特定边界的机器而言，如果这种类型的数据在内存中的存储地址处在错误的边界上，那么对这个地址进行访问时将会产生一个错误。这种错误在 UNIX 系统中被称为“总线错误（bus error）”。

一个更为严重的情况是：这个指针偶尔可能包含了一个合法的地址。接下来的事很简单：位于



那个位置的值被修改，虽然你并无意去修改它。像这种类型的错误非常难以捕捉，因为引发错误的代码可能与原先用于操作那个值的代码完全不相干。所以，在你对指针进行间接访问之前，必须非常小心，确保它们已被初始化！

## 6.6 NULL 指针

标准定义了 NULL 指针，它作为一个特殊的指针变量，表示不指向任何东西。要使一个指针变量为 NULL，你可以给它赋一个零值。为了测试一个指针变量是否为 NULL，你可以将它与零值进行比较。之所以选择零这个值是因为一种源代码约定。就机器内部而言，NULL 指针的实际值可能与此不同。在这种情况下，编译器将负责零值和内部值之间的翻译转换。

NULL 指针的概念是非常有用的，因为它给了你一种方法，表示某个特定的指针目前并未指向任何东西。例如，一个用于在某个数组中查找某个特定值的函数可能返回一个指向查找到的数组元素的指针。如果该数组不包含指定条件的值，函数就返回一个 NULL 指针。这个技巧允许返回值传达两个不同片段的信息。首先，有没有找到元素？其次，如果找到，它是哪个元素？

### 提示：

尽管这个技巧在 C 程序中极为常用，但它违背了软件工程的原则。用一个单一的值表示两种不同的意思是件危险的事，因为将来很容易无法弄清哪个才是它真正的用意。在大型的程序中，这个问题更为严重，因为你不可能在头脑中对整个设计一览无余。一种更为安全的策略是让函数返回两个独立的值：首先是个状态值，用于提示查找是否成功；其次是个指针，当状态值提示查找成功时，它所指向的就是查找到的元素。

对指针进行解引用操作可以获得它所指向的值。但从定义上看，NULL 指针并未指向任何东西。因此，对一个 NULL 指针进行解引用操作是非法的。在对指针进行解引用操作之前，你首先必须确保它并非 NULL 指针。

### 警告：

如果对一个 NULL 指针进行间接访问会发生什么情况呢？它的结果因编译器而异。在有些机器上，它会访问内存位置零。编译器能够确保内存位置零没有存储任何变量，但机器并未妨碍你访问或修改这个位置。这种行为是非常不幸的，因为程序包含了一个错误，但机器却隐匿了它的症状，这样就使这个错误更加难以寻找。

在其他机器上，对 NULL 指针进行间接访问将引发一个错误，并终止程序。宣布这个错误比隐藏这个错误要好得多，因为程序员能够更容易修正它。

### 提示：

如果所有的指针变量（而不仅仅是位于静态内存中的指针变量）能够被自动初始化为 NULL，那实在是件幸事，但事实并非如此。不论你的机器对解引用 NULL 指针这种行为作何反应，对所有的指针变量进行显式的初始化是种好做法。如果你已经知道指针将被初始化为什么地址，就把它初始化为该地址，否则就把它初始化为 NULL。风格良好的程序会在指针解引用之前对它进行检查，这种初始化策略可以节省大量的调试时间。

## 6.7 指针、间接访问和左值

涉及指针的表达式能不能作为左值？如果能，又是哪些呢？对表 5.1 优先级表格进行快速查阅后可以发现，间接访问操作符所需要的操作数是个右值，但这个操作符所产生的结果是个左值。

让我们回到早些时候的例子。给定下面这些声明。

```
int a;
int *d = &a;
```

考虑下面的表达式：

表 达 式	左 值	指定位置
a	是	a
d	是	d
*d	是	a

指针变量可以作为左值，并不是因为它们是指针，而是因为它们是变量。对指针变量进行间接访问表示我们应该访问指针所指向的位置。间接访问指定了一个特定的内存位置，这样我们可以把间接访问表达式的结果作为左值使用。在下面这两条语句中，

```
*d = 10 - *d;
d = 10 - *d;    ← ???
```

第 1 条语句包含了两个间接访问操作。右边的间接访问作为右值使用，所以它的值是 **d** 所指向的位置所存储的值（**a** 的值）。左边的间接访问作为左值使用，所以 **d** 所指向的位置（**a**）把赋值符右侧的表达式的结果作为它的新值。

第 2 条语句是非法的，因为它表示把一个整型数量（ $10 - *d$ ）存储于一个指针变量中。当我们实际使用的变量类型和应该使用的变量类型不一致时，编译器会发出抱怨，帮助我们判断这种情况。这些警告和错误信息是我们的朋友，编译器通过产生这些信息向我们提供帮助。尽管被迫处理这些信息是我们很不情愿干的事情，但改正这些错误（尤其是那些不会中止编译过程的警告信息）确实是个好主意。在修正程序方面，让编译器告诉你哪里错了比你以后自己调试程序要方便得多。调试器无法像编译器那样准确地查明这些问题。

**K&R C:**

当混用指针和整型值时，旧式 C 编译器并不会发出抱怨。但是，我们现在对这方面的知识知道得更透彻一些了。把整型值转换为指针或把指针转换成整型值是极为罕见的，通常这类转换属于无意识的错误。

## 6.8 指针、间接访问和变量

如果你自以为已经精通了指针，请看一下这个表达式，看看你是否明白它的意思。

```
*&a = 25;
```

如果你的答案是它把值 25 赋值给变量 **a**，恭喜！你答对了。让我们来分析这个表达式。首先，

&操作符产生变量 `a` 的地址，它是一个指针常量（注意，使用这个指针常量并不需要知道它的实际值）。接着，`*`操作符访问其操作数所表示的地址。在这个表达式中，操作数是 `a` 的地址，所以值 25 就存储于 `a` 中。

这条语句和简单地使用 `a=25` 有什么区别吗？从功能上说，它们是相同的。但是，它涉及更多的操作。除非编译器（或优化器）知道你在干什么并丢弃额外的操作，否则它所产生的目标代码将会更大、更慢。更糟的是，这些额外的操作符会使源代码的可读性变差。基于这些原因，没人会故意使用像 `*&a` 这样的表达式。

## 6.9 指针常量

让我们来分析另外一个表达式。假定变量 `a` 存储于位置 100，下面这条语句的作用是什么？

```
*100 = 25;
```

它看上去像是把 25 赋值给 `a`，因为 `a` 是位置 100 所存储的变量。但是，这是错的！这条语句实际上是非合法的，因为字面值 100 的类型是整型，而间接访问操作只能作用于指针类型表达式。如果你确实想把 25 存储于位置 100，你必须使用强制类型转换。

```
*(int *)100 = 25;
```

强制类型转换把值 100 从“整型”转换为“指向整型的指针”，这样对它进行间接访问就是合法的。如果 `a` 存储于位置 100，那么这条语句就把值 25 存储于 `a`。但是，你需要使用这种技巧的机会是绝无仅有的！为什么？我前面提到过，你通常无法预测编译器会把某个特定的变量放在内存中的什么位置，所以你无法预先知道它的地址。用 `&`操作符得到变量的地址是很容易的，但表达式在程序执行时才会进行求值，此时已经来不及把它的结果作用字面值常量复制到源代码。

这个技巧唯一有用之处是你偶尔需要通过地址访问内存中某个特定的位置，它并不是用于访问某个变量，而是访问硬件本身。例如，操作系统需要与输入输出设备控制器通信，启动 I/O 操作并从前面的操作中获得结果。在有些机器上，与设备控制器的通信是通过在某个特定内存地址读取和写入值来实现的。但是，与其说这些操作访问的是内存，还不如说它们访问的是设备控制器接口。这样，这些位置必须通过它们的地址来访问，此时这些地址是预先已知的。

第 3 章曾提到并没有一种内建的记法用于书写指针常量。在那些极其罕见的需要使用它们的时候，它们通常写成整型字面值的形式，并通过强制类型转换转换成适当的类型<sup>1</sup>。

## 6.10 指针的指针

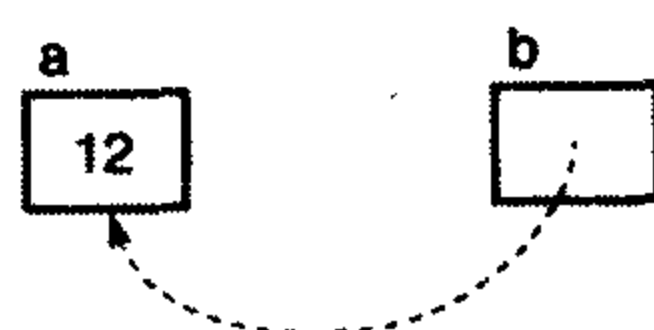
这里我们再稍微花点时间来看一个例子，揭开这个即将开始的主题的序幕。考虑下面这些声明：

```
int a = 12;
int *b = &a;
```

它们如下图所示进行内存分配：

<sup>1</sup> 在段式机器(segmented machine)的实现中，如 Intel 80x86，可能会提供一个宏(macro)来创建指针常量。这些宏把段地址和偏移地址组合转换为指针值。

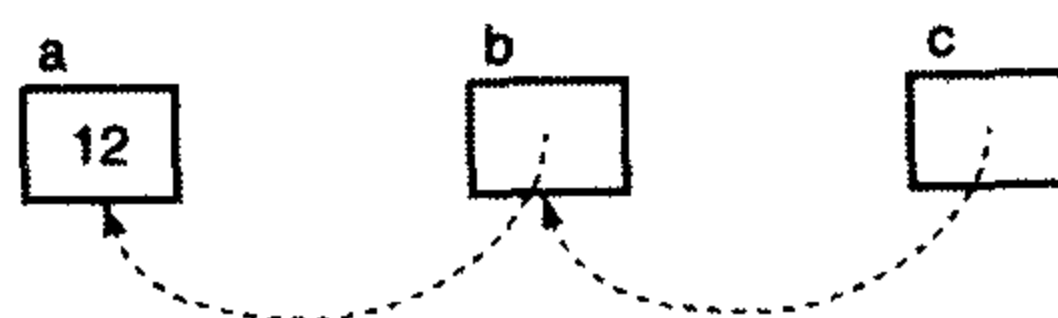




假定我们有了第 3 个变量，名叫 `c`，并用下面这条语句对它进行初始化：

```
c = &b;
```

它们在内存中的模样大致如下：



问题是：`c` 的类型是什么？显然它是一个指针，但它所指向的是什么？变量 `b` 是一个“指向整型的指针”，所以任何指向 `b` 的类型必须是指向“指向整型的指针”的指针，更通俗地说，是一个指针的指针。

它合法吗？是的！指针变量和其他变量一样，占据内存中某个特定的位置，所以用 `&` 操作符取得它的地址是合法的<sup>1</sup>。

那么这个变量是怎样声明的呢？声明

```
int **c;
```

表示表达式 `**c` 的类型是 `int`。表 6.1 列出了一些表达式，有助于我们弄清这个概念。假定这些表达式进行了如下这些声明。

```
int    a = 12;
int    *b = &a;
int    **c = &b;
```

表中唯一的新面孔是最后一个表达式，让我们对它进行分析。`*操作符具有从右向左的结合性`，所以这个表达式相当于 `*(*c)`，我们必须从里向外逐层求值。`*c` 访问 `c` 所指向的位置，我们知道这是变量 `b`。第 2 个间接访问操作符访问这个位置所指向的地址，也就是变量 `a`。指针的指针并不难懂，你只要留心所有的箭头，如果表达式中出现了间接访问操作符，你就随箭头访问它所指向的位置。

表 6.1 双重间接访问

表达式	相当的表达式
<code>a</code>	<code>12</code>
<code>b</code>	<code>&amp;a</code>
<code>*b</code>	<code>a, 12</code>
<code>c</code>	<code>&amp;b</code>
<code>*c</code>	<code>b, &amp;a</code>
<code>**c</code>	<code>*b, a, 12</code>

## 6.11 指针表达式

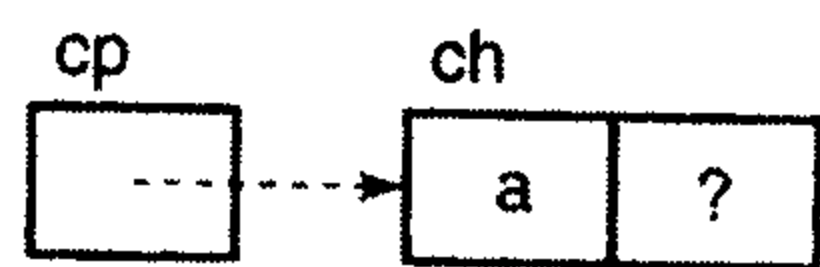
现在让我们观察各种不同的指针表达式，并看看当它们分别作为左值和右值时是如何进行求值的。有些表达式用得很普遍，但有些却不常用。这个练习的目的并不是想给你一本这类表达式的“烹

<sup>1</sup> 声明为 `register` 的变量例外。

调全书”，而是想让你完善阅读和编写它们的技巧。首先，让我们来看一些声明。

```
char ch = 'a';
char *cp = &ch;
```

现在，我们就有了两个变量，它们初始化如下：

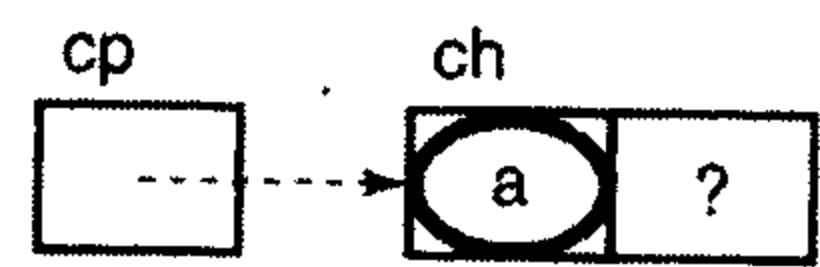


图中还显示了 ch 后面的那个内存位置，因为我们所求值的有些表达式将访问它（尽管是在错误情况下才会对它进行访问）。由于我们并不知道它的初始值，所以用一个问号来代替。

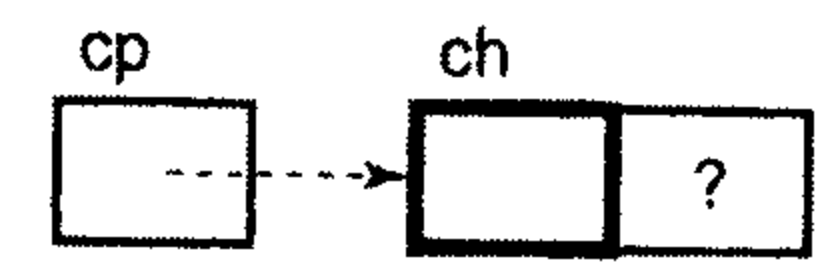
首先来个简单的作为开始，如下面这个表达式：

ch

当它作为右值使用时，表达式的值为'a'，如下图所示：



那个粗椭圆提示变量 ch 的值就是表达式的值。但是，当这个表达式作为左值使用时，它是这个内存的地址而不是该地址所包含的值，所以它的图示方式有所不同：



此时该位置用粗方框标记，提示这个位置就是表达式的结果。另外，它的值并未显示，因为它并不重要。事实上，这个值将被某个新值所取代。接下来的表达式将以表格的形式出现。每个表的后面是表达式求值过程的描述。

表达式	右值	左值
&ch		非法

作为右值，这个表达式的值是变量 ch 的地址。注意这个值同变量 cp 中所存储的值一样，但这个表达式并未提及 cp，所以这个结果值并不是因为它而产生的。这样，图中椭圆并不画于 cp 的箭头周围。第 2 个问题是，为什么这个表达式不是一个合法的左值？优先级表格显示&操作符的结果是个右值，它不能当作左值使用。但是为什么呢？答案很简单，当表达式&ch 进行求值时，它的结果应该存储于计算机的什么地方呢？它肯定会位于某个地方，但你无法知道它位于何处。这个表达式并未标识任何机器内存的特定位置，所以它不是一个合法的左值。

表达式	右值	左值
cp		

你以前曾见到过这个表达式。它的右值如图所示就是 `cp` 的值。它的左值就是 `cp` 所处的内存位置。由于这个表达式并不进行间接访问操作，所以你不必依箭头所示进行间接访问。

表达式	右值	左值
&cp		非法

这个例子与 `&ch` 类似，不过我们这次所取的是指针变量的地址。这个结果的类型是指向字符的指针的指针。同样，这个值的存储位置并未清晰定义，所以这个表达式不是一个合法的左值。

表达式	右值	左值
*cp		

现在我们加入了间接访问操作，所以它的结果应该不会令人惊奇。但接下来的几个表达式就比较有意思。

表达式	右值	左值
*cp + 1		非法

这个图涉及的东西更多，所以让我们一步一步来分析它。这里有两个操作符。`*`的优先级高于`+`，所以首先执行间接访问操作（如图中 `cp` 到 `ch` 的实线箭头所示），我们可以得到它的值（如虚线椭圆所示）。我们取得这个值的一份拷贝并把它与 1 相加，表达式的最终结果为字符 `'b'`。图中虚线表示表达式求值时数据的移动过程。这个表达式的最终结果的存储位置并未清晰定义，所以它不是一个合法的左值。优先级表格证实 `+` 的结果不能作为左值。

在这个例子中，我们在前面那个表达式中增加了一个括号。这个括号使表达式先执行加法运算，就是把 1 和 `cp` 中所存储的地址相加。此时的结果值是图中虚线椭圆所示的指针。接下来的间接访问操作随着箭头访问紧随 `ch` 之后的内存位置。这样，这个表达式的右值就是这个位置的值，而它的左值是这个位置本身。

表达式	右值	左值
<code>*(cp + 1)</code>		

在这里我们需要学习很重要的一点。注意指针加法运算的结果是个右值，因为它的存储位置并未清晰定义。如果没有间接访问操作，这个表达式将不是一个合法的左值。然而，间接访问跟随指针访问一个特定的位置。这样，`*(cp+1)`就可以作用左值使用，尽管 `cp+1` 本身并不是左值。间接访问操作符是少数几个其结果为左值的操作符之一。

但是，这个表达式所访问的是 `ch` 后面的那个内存位置，我们如何知道原先存储于那个地方的是什么东西？一般而言，我们无法得知，所以像这样的表达式是非法的。本章的后面我将更为深入地探讨这个问题。

表达式	右值	左值
<code>++cp</code>		非法

`++`和`--`操作符在指针变量中使用得相当频繁，所以在这种上下文环境中理解它们是非常重要的。在这个表达式中，我们增加了指针变量 `cp` 的值。（为了让图更清楚，我们省略了加法）。表达式的结果是增值后的指针的一份拷贝，因为前缀`++`先增加它的操作数的值再返回这个结果。这份拷贝的存储位置并未清晰定义，所以它不是一个合法的左值。

表达式	右值	左值
<code>cp++</code>		非法

后缀`++`操作符同样增加 `cp` 的值，但它先返回 `cp` 值的一份拷贝然后再增加 `cp` 的值。这样，这个表达式的值就是 `cp` 原来的值的一份拷贝。

前面两个表达式的值都不是合法的左值。但如果我们在表达式中增加了间接访问操作符，它们就可以成为合法的左值，如下面的两个表达式所示。

表达式	右值	左值
<code>*++cp</code>		

这里，间接访问操作符作用于增值后的指针的拷贝上，所以它的右值是 `ch` 后面那个内存地址的值，而它的左值就是那个位置本身。

表达式	右值	左值
<code>*cp++</code>		

使用后缀++操作符所产生的结果不同：它的右值和左值分别是变量 `ch` 的值和 `ch` 的内存位置，也就是 `cp` 原先所指。同样，后缀++操作符在周围的表达式中使用其原先操作数的值。间接访问操作符和后缀++操作符的组合常常令人误解。优先级表格显示后缀++操作符的优先级高于\*操作符，但表达式的结果看上去像是先执行间接访问操作。事实上，这里涉及 3 个步骤：(1) ++操作符产生 `cp` 的一份拷贝，(2) 然后++操作符增加 `cp` 的值，(3) 最后，在 `cp` 的拷贝上执行间接访问操作。

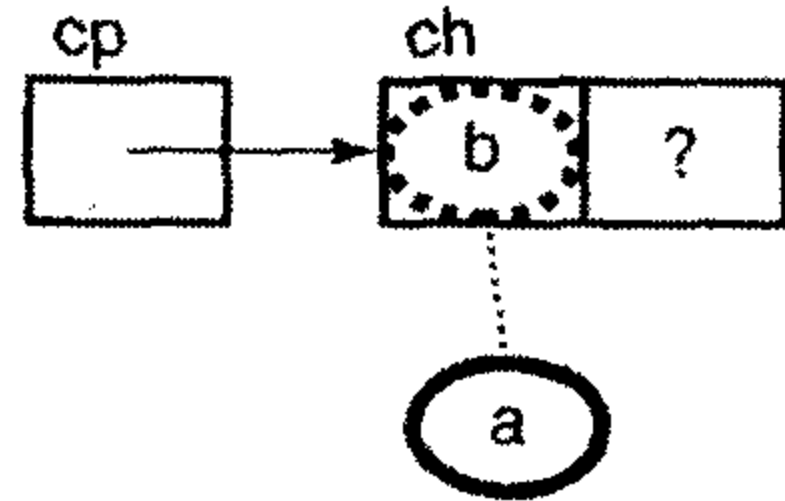
这个表达式常常在循环中出现，首先用一个数组的地址初始化指针，然后使用这种表达式就可以依次访问该数组的内容了。本章的后面显示了一些这方面的例子。

表达式	右值	左值
<code>++*cp</code>		非法

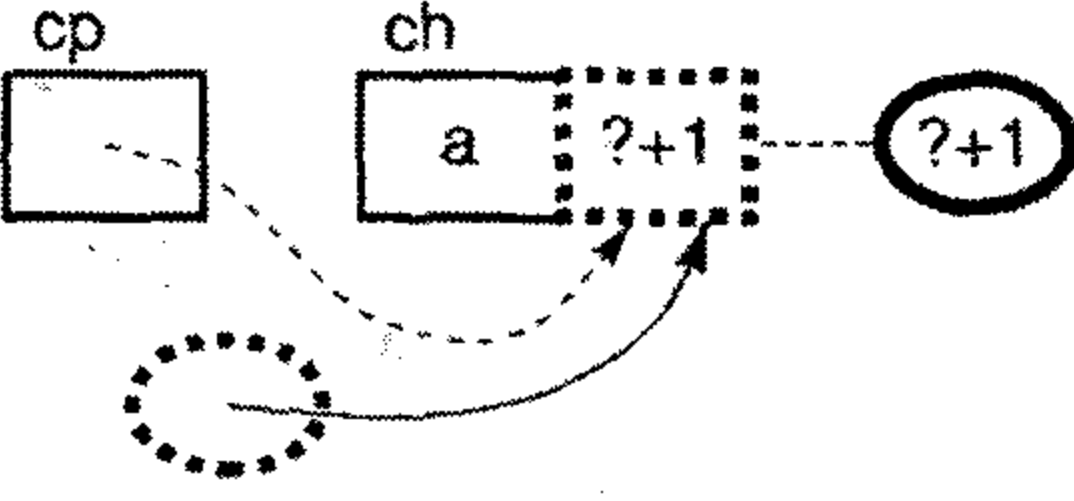
在这个表达式中，由于这两个操作符的结合性都是从右向左，所以首先执行的是间接访问操作。然后，`cp` 所指向的位置的值增加 1，表达式的结果是这个增值后的值的一份拷贝。

与前面一些表达式相比，最后 3 个表达式在实际中使用得较少。但是，对它们有一个透彻的理解有助于提高你的技能。

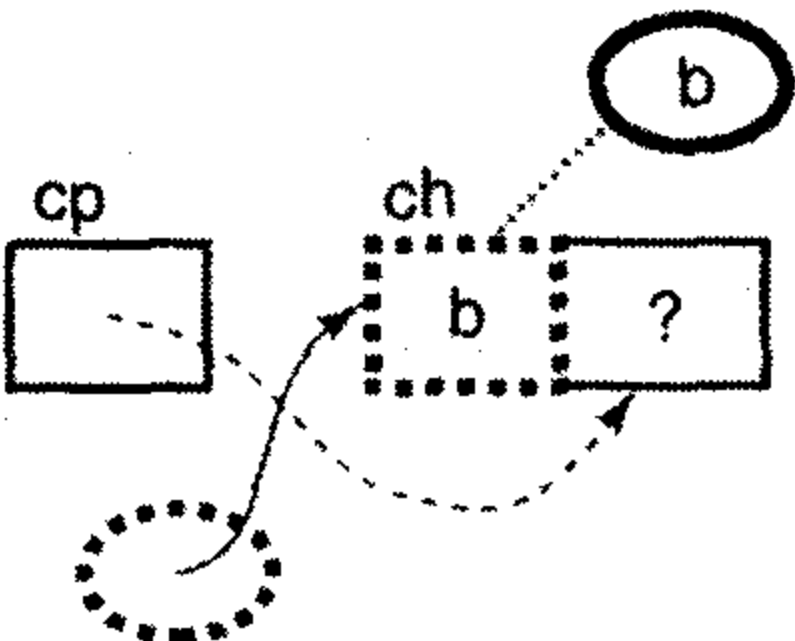


表达式	右值	左值
<code>(*cp)++</code>		非法

使用后缀++操作符，我们必须加上括号，使它首先执行间接访问操作。这个表达式的计算过程与前一个表达式相似，但它的结果值是 `ch` 增值前的原先值。

表达式	右值	左值
<code>++*++cp</code>		非法

这个表达式看上去相当诡异，但事实上并不复杂。这个表达式共有 3 个操作符，所以看上去有些吓人。但是，如果你逐个对它们进行分析，你会发现它们都很熟悉。事实上，我们先前已经计算了 `*++cp`，所以现在我们需要做的只是增加它的结果值。但是，让我们还是从头开始。记住这些操作符的结合性都是从右向左，所以首先执行的是 `++cp`。`cp` 下面的虚线椭圆表示第 1 个中间结果。接着，我们对这个拷贝值进行间接访问，它使我们访问 `ch` 后面的那个内存位置。第 2 个中间结果用虚线方框表示，因为下一个操作符把它当作一个左值使用。最后，我们在这个位置执行++操作，也就是增加它的值。我们之所以把结果值显示为 `?+1` 是因为我们并不知道这个位置原先的值。

表达式	右值	左值
<code>++*cp++</code>		非法

这个表达式和前一个表达式的区别在于这次第 1 个++操作符是后缀形式而不是前缀形式。由于它的优先级较高，所以先执行它。间接访问操作所访问的是 `cp` 所指向的位置而不是 `cp` 所指向位置后面的那个位置。

6.12 实例

这里有几个例子程序，用于说明指针表达式的一些常见用法。程序 6.1 计算一个字符串的长度。

你应该不用自己编写这个函数，因为函数库里已经有了一个，不过它是个有用的例子。

```

/*
** 计算一个字符串的长度。
*/

#include <stdlib.h>

size_t
strlen( char *string )
{
    int    length = 0;

    /*
    ** 依次访问字符串的内容，计数字符数，直到遇见 NUL 终止符。
    */
    while( *string++ != '\0' )
        length += 1;

    return length;
}

```

### 程序 6.1 字符串长度

strlen.c

在指针到达字符串末尾的 NUL 字节之前，while 语句中 \*string++ 表达式的值一直为真。它同时增加指针的值，用于下一次测试。这个表达式甚至可以正确地处理空字符串。

#### 警告：

如果这个函数调用时传递给它的是一个 NULL 指针，那么 while 语句中的间接访问将会失败。函数是不是应该在解引用指针前检查这个条件？从绝对安全的角度讲，应该如此。但是，这个函数并不负责创建字符串。如果它发现参数为 NULL，它肯定发现了一个出现在程序其他地方的错误。当指针创建时检查它是否有效是合乎逻辑的，因为这样只需检查一次。这个函数采用的就是这种方法。如果函数失败是因为粗心大意的调用者懒得检查参数的有效性而引起的，那是他活该如此。

程序 6.2 和 6.3 增加了一层间接访问。它们在一些字符串中搜索某个特定的字符值，但我们使用指针数组来表示这些字符串，如图 6.1 所示。函数的参数是 strings 和 value，strings 是一个指向指针数组的指针，value 是我们所查找的字符值。注意指针数组以一个 NULL 指针结束。函数将检查

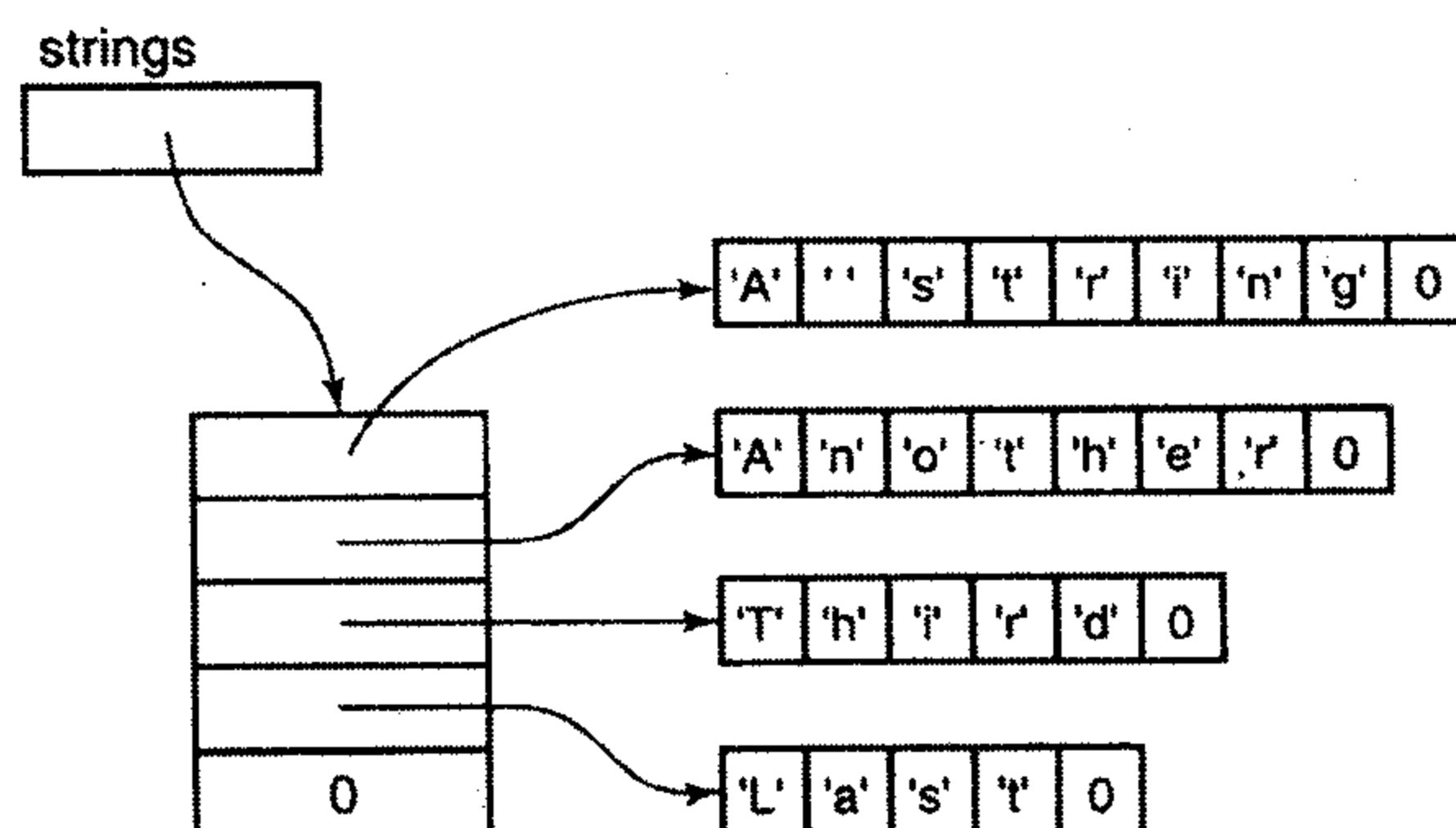


图 6.1 指向字符串的指针的数组

这个值以判断循环何时结束。下面这行表达式

```
while( ( string = *strings++ ) != NULL ) {
```

完成三项任务：(1) 它把 `strings` 当前所指向的指针复制到变量 `string` 中。(2) 它增加 `strings` 的值，使它指向下一个值。(3) 它测试 `string` 是否为 `NULL`。当 `string` 指向当前字符串中作为终止标志的 `NUL` 字节时，内层的 `while` 循环就终止。

```
/*
** 给定一个指向以 NULL 结尾的指针列表的指针，在列表中的字符串中查找一个特定的字符。
*/

#include <stdio.h>

#define TRUE 1
#define FALSE 0

int
find_char( char **strings, char value )
{
    char*string;      /* 我们当前正在查找的字符串 */

    /*
    ** 对于列表中的每个字符串 ...
    */
    while( ( string = *strings++ ) != NULL ){
        /*
        ** 观察字符串中的每个字符，看看它是不是我们需要查找的那个。
        */
        while( *string != '\0' ){
            if( *string++ == value )
                return TRUE;
        }
    }
    return FALSE;
}
```

## 程序 6.2 在一组字符串中查找：版本 1

s\_srchl.c

如果 `string` 尚未到达其结尾的 `NUL` 字节，就执行下面这条语句

```
if( *string++ == value )
```

它测试当前的字符是否与需要查找的字符匹配，然后增加指针的值，使它指向下一个字符。

程序 6.3 实现相同的功能，但它不需要对指向每个字符串的指针作一份拷贝。但是，由于存在副作用，这个程序将破坏这个指针数组。这个副作用使该函数不如前面那个版本有用，因为它只适用于字符串只需要查找一次的情况。

```
/*
** 给定一个指向以 NULL 结尾的指针列表的指针，在列表中的字符串中查找一个特定的字符。这个函数将破坏这些指针，所以它只适用于这组字符串只使用一次的情况。
*/

#include <stdio.h>
#include <assert.h>

#define TRUE 1
#define FALSE 0
```

```

int
find_char( char **strings, int value )
{
    assert( strings != NULL );

    /*
    ** 对于列表中的每个字符串 ...
    */
    while( *strings != NULL ){
        /*
        ** 观察字符串中的每个字符，看看它是否是我们查找的那个。
        */
        while( **strings != '\0' ){
            if( *(*strings)++ == value )
                return TRUE;
        }
        strings++;
    }
    return FALSE;
}

```

程序 6.3 在一组字符串中查找：版本 2

s\_srch2.c

但是，在程序 6.3 中存在两个有趣的表达式。第 1 个是 `**strings`。第 1 个间接访问操作访问指针数组中的当前指针，第 2 个间接访问操作随该指针访问字符串中的当前字符。内层的 `while` 语句测试这个字符的值并观察是否到达了字符串的末尾。

第 2 个有趣的表达式是  `*(*strings)++`。括号是需要的，这样才能使表达式以正确的顺序进行求值。第 1 个间接访问操作访问列表中的当前指针。增值操作把该指针所指向的那个位置的值加 1，但第 2 个间接访问操作作用于原先那个值的拷贝上。这个表达式的直接作用是对当前字符串中的当前字符进行测试，看看是否到达了字符串的末尾。作为副作用，指向当前字符串字符的指针值将增加 1。

## 6.13 指针运算

程序 6.1~6.3 包含了一些涉及指针值和整型值加法运算的表达式。是不是对指针进行任何运算都是合法的呢？答案是它可以执行某些运算，但并非所有运算都合法。除了加法运算之外，你还可以对指针执行一些其他运算，但并不是很多。

指针加上一个整数的结果是另一个指针。问题是，它指向哪里？如果你将一个字符指针加 1，运算结果产生的指针指向内存中的下一个字符。float 占据的内存空间不止 1 个字节，如果你将一个指向 float 的指针加 1，将会发生什么呢？它会不会指向该 float 值内部的某个字节呢？

幸运的是，答案是否定的。当一个指针和一个整数量执行算术运算时，整数在执行加法运算前始终会根据合适的大小进行调整。这个“合适的大小”就是指针所指向类型的大小，“调整”就是把整数值和“合适的大小”相乘。为了更好地说明，试想在某台机器上，float 占据 4 个字节。在计算 float 型指针加 3 的表达式时，这个 3 将根据 float 类型的大小（此例中为 4）进行调整（相乘）。这样，实际加到指针上的整型值为 12。

把 3 与指针相加使指针的值增加 3 个 float 的大小，而不是 3 个字节。这个行为较之获得一个指向一个 float 值内部某个位置的指针更为合理。表 6.2 包含了一些加法运算的例子。调整的美感在于

指针算法并不依赖于指针的类型。换句话说，如果 `p` 是一个指向 `char` 的指针，那么表达式 `p+1` 就指向下一个 `char`。如果 `p` 是个指向 `float` 的指针，那么 `p+1` 就指向下一个 `float`，其他类型也是如此。

表 6.2 指针运算结果

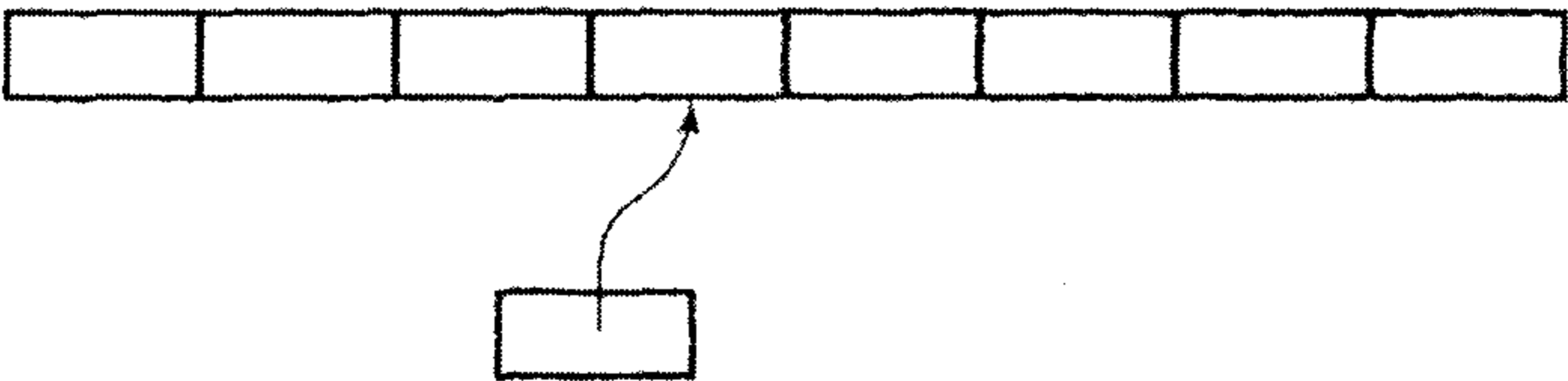
表达式	假定 <code>p</code> 是个指向...的指针	而且 <code>*p</code> 的大小是...	增加到指针的值
<code>p + 1</code>	<code>char</code>	1	1
	<code>short</code>	2	2
<code>p + 1</code>	<code>int</code>	4	4
	<code>double</code>	8	8
<code>p + 2</code>	<code>char</code>	1	2
	<code>short</code>	2	4
	<code>int</code>	4	8
	<code>double</code>	8	16

6.13.1 算术运算

C 的指针算术运算只限于两种形式。第 1 种形式是：

指针 ± 整数

标准定义这种形式只能用于指向数组中某个元素的指针，如下图所示。



并且这类表达式的结果类型也是指针。这种形式也适用于使用 `malloc` 函数动态分配获得的内存（见第 11 章），尽管翻遍标准也未见它提及这个事实。

数组中的元素存储于连续的内存位置中，后面元素的地址大于前面元素的地址。因此，我们很容易看出，对一个指针加 1 使它指向数组中下一个元素，加 5 使它向右移动 5 个元素的位置，依次类推。把一个指针减去 3 使它向左移动 3 个元素的位置。对整数进行扩展保证对指针执行加法运算能产生这种结果，而不管数组元素的长度如何。

对指针执行加法或减法运算之后如果结果指针所指的位置在数组第 1 个元素的前面或在数组最后一个元素的后面，那么其效果就是未定义的。让指针指向数组最后一个元素后面的那个位置是合法的，但对这个指针执行间接访问可能会失败。

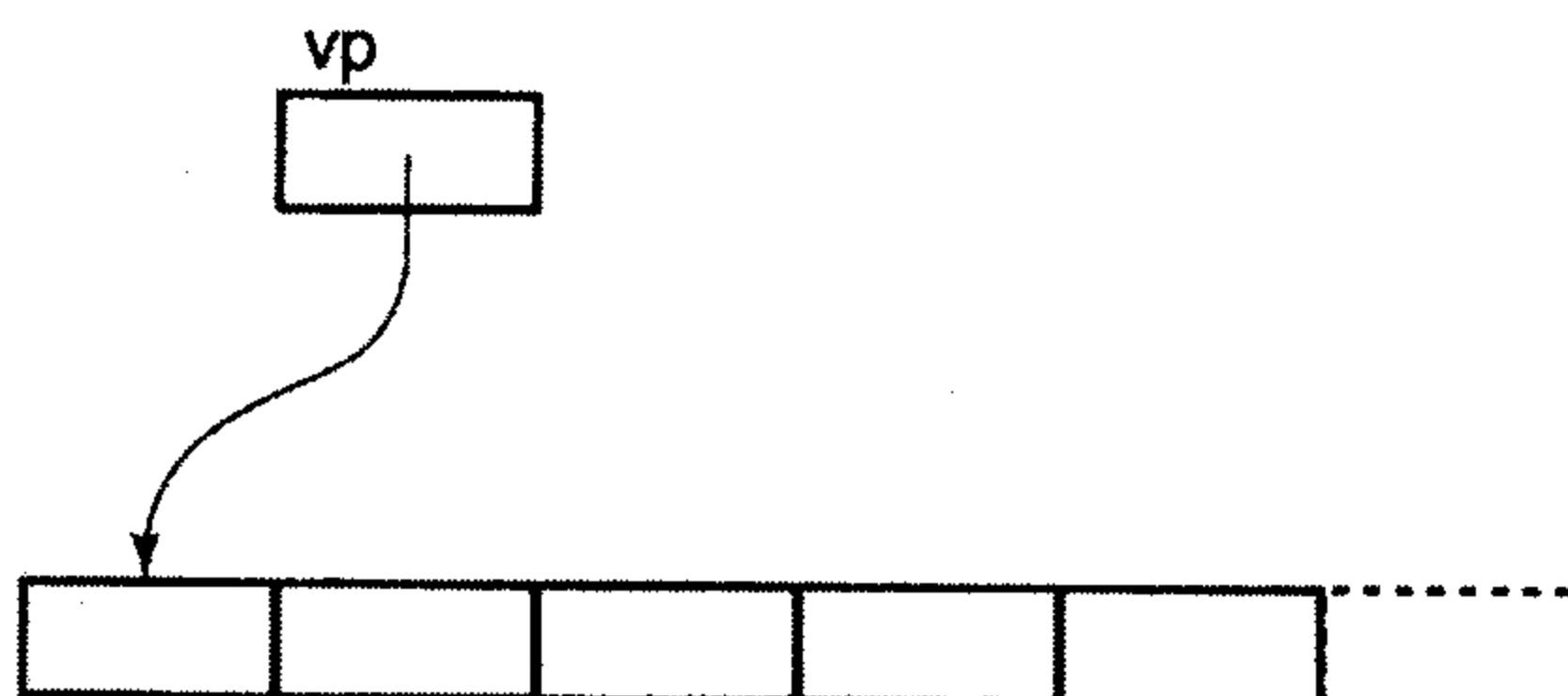
是该举个例子的时候了。这里有一个循环，把数组中所有的元素都初始化为零。（第 8 章将讨论类似这种循环和使用下标访问的循环之间的效率比较）。

```
#define N_VALUES      5
float  values[N_VALUES];
float  *vp;

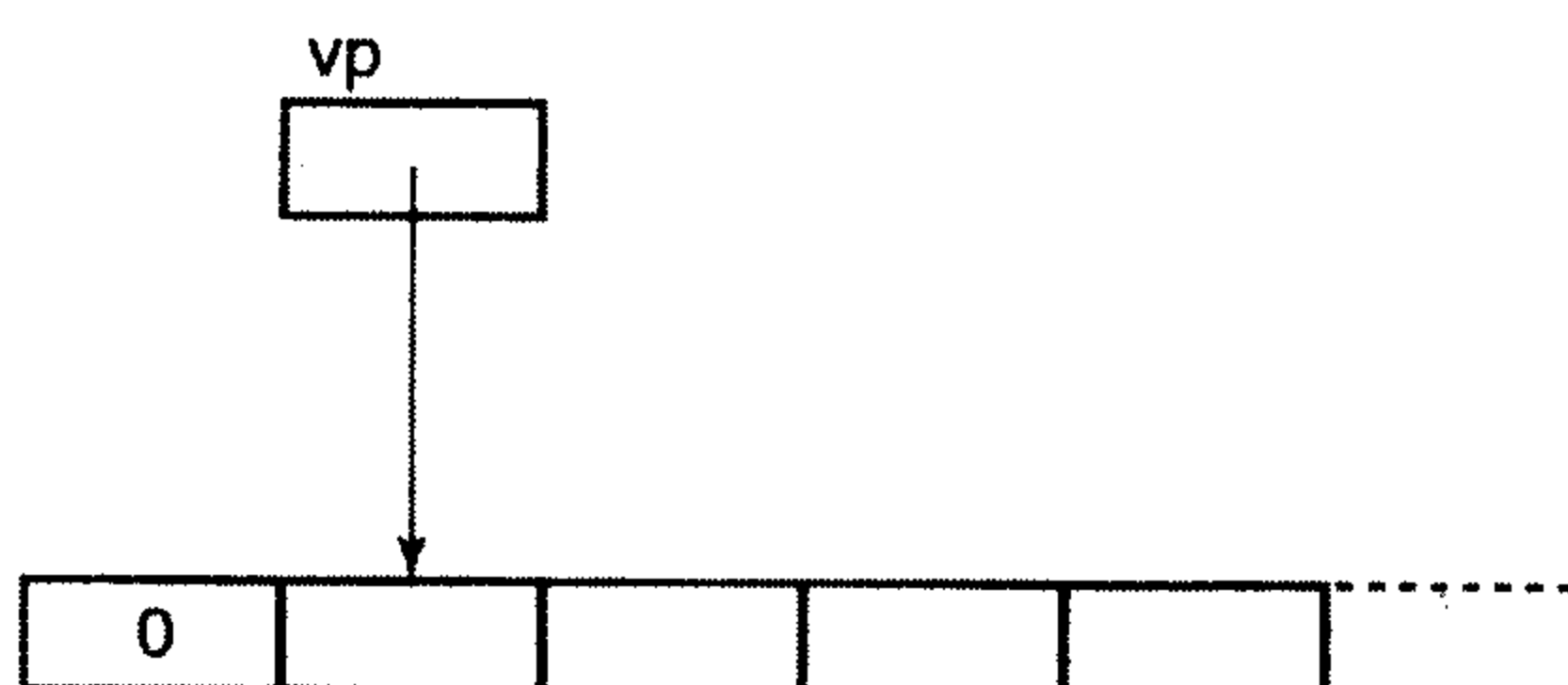
for( vp = &values[0]; vp < &values[N_VALUES]; )
    *vp++ = 0;
```

for 语句的初始部分把 `vp` 指向数组的第 1 个元素。

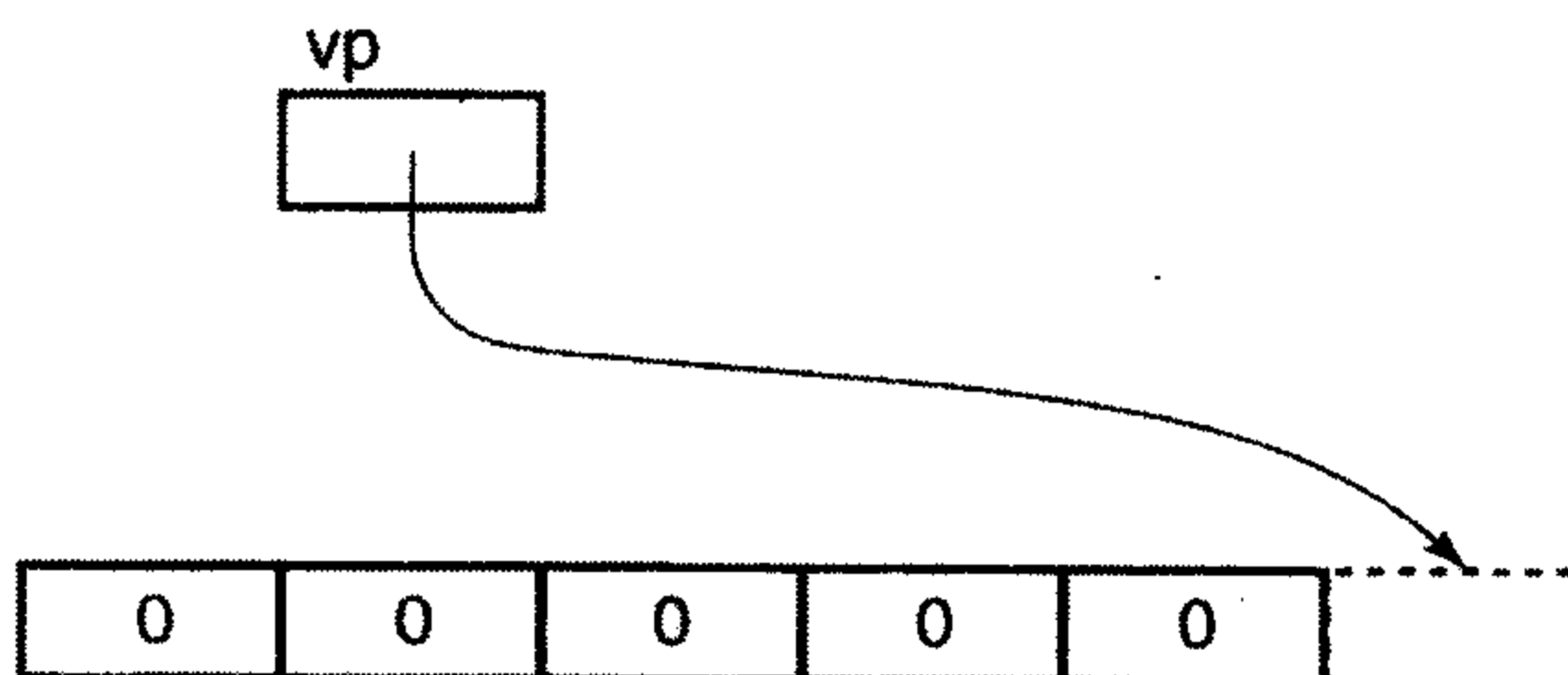




这个例子中的指针运算都是用++操作符完成的。增加值 1 与 float 的长度相乘，其结果加到指针 vp 上。经过第 1 次循环之后，指针在内存中的位置如下：



经过 5 次循环之后，vp 就指向数组最后一个元素后面的那个内存位置。



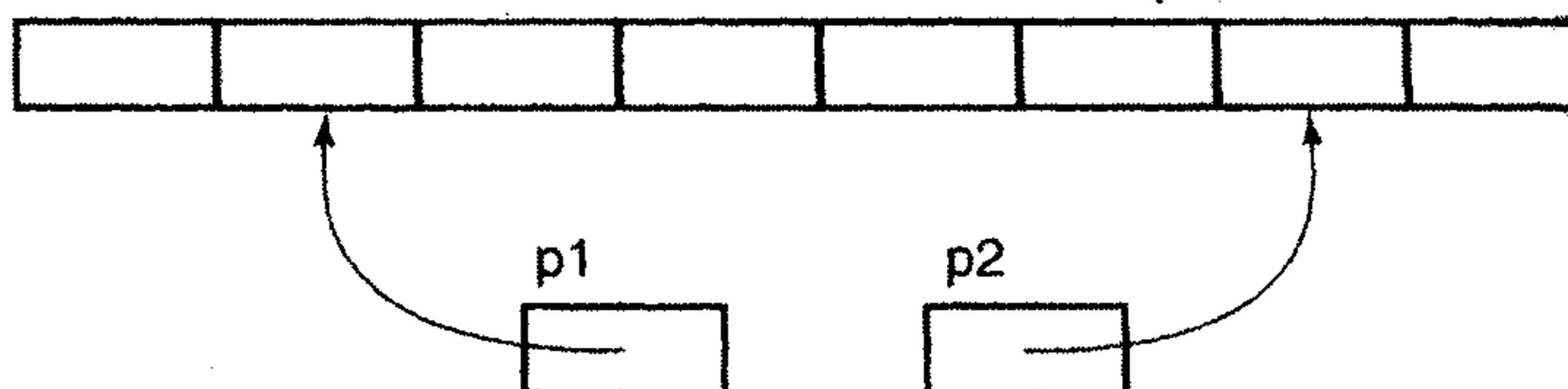
此时循环终止。由于下标值从零开始，所以具有 5 个元素的数组的最后一个元素的下标值为 4。这样，&values[N\_VALUES]表示数组最后一个元素后面那个内存位置的地址。当 vp 到达这个值时，我们就知道到达了数组的末尾，故循环终止。

这个例子中的指针最后所指向的是数组最后一个元素后面的那个内存位置。指针可能可以合法地获得这个值，但对它执行间接访问时将可能意外地访问原先存储于这个位置的变量。程序员一般无法知道那个位置原先存储的是什么变量。因此，在这种情况下，一般不允许对指向这个位置的指针执行间接访问操作。

第 2 种类型的指针运算具有如下形式：

指针 — 指针

只有当两个指针都指向同一个数组中的元素时，才允许从一个指针减去另一个指针，如下所示：



两个指针相减的结果的类型是 `ptrdiff_t`，它是一种有符号整数类型。减法运算的值是两个指针在内存中的距离（以数组元素的长度为单位，而不是以字节为单位），因为减法运算的结果将除以数组元素类型的长度。例如，如果 `p1` 指向 `array[i]` 而 `p2` 指向 `array[j]`，那么 `p2-p1` 的值就是 `j-i` 的值。

让我们看一下它是如何作用于某个特定类型的。假定前图中数组元素的类型为 `float`，每个元素占据 4 个字节的内存空间。如果数组的起始位置为 1000，`p1` 的值是 1004，`p2` 的值是 1024，但表达式 `p2-p1` 的结果值将是 5，因为两个指针的差值(20)将除以每个元素的长度(4)。

同样，这种对差值的调整使指针的运算结果与数据的类型无关。不论数组包含的元素类型如何，这个指针减法运算的值总是 5。

那么，表达式 `p1-p2` 是否合法呢？是的，如果两个指针都指向同一个数组中的元素，这个表达式就是合法的。在前一个例子中，这个值将是一5。

如果两个指针所指向的不是同一个数组中的元素，那么它们之间相减的结果是未定义的。就像如果你把两个位于不同街道的房子的门牌号码相减不可能获得这两所房子间的房子数一样。程序员无从知道两个数组在内存中的相对位置，如果不知道这一点，两个指针之间的距离就毫无意义。

#### 警告：

实际上，绝大多数编译器都不会检查指针表达式的结果是否位于合法的边界之内。因此，程序员应该负起责任，确保这一点。类似，编译器将不会阻止你取一个标量变量的地址并对它执行指针运算，即使它无法预测运算结果所产生的指针将指向哪个变量。越界指针和指向未知值的指针是两个常见的错误根源。当你使用指针运算时，必须非常小心，确信运算的结果将指向有意义的东西。

### 6.13.2 关系运算

对指针执行关系运算也是有限制的。用下列关系操作符对两个指针值进行比较是可能的：

<            <=            >            >=

不过前提是它们都指向同一个数组中的元素。根据你所使用的操作符，比较表达式将告诉你哪个指针指向数组中更前或更后的元素。标准并未定义如果两个任意的指针进行比较会产生什么结果。

然而，你可以在两个任意的指针间执行相等或不相等测试，因为这类比较的结果和编译器选择在何处存储数据并无关系——指针要么指向同一个地址，要么指向不同的地址。

让我们再观察一个循环，它用于清除一个数组中所有的元素。

```
#define N_VALUES      5
float  values[N_VALUES];
float  *vp;

for( vp = &values[0]; vp < &values[N_VALUES]; )
    *vp++ = 0;
```

`for` 语句使用了一个关系测试来决定是否结束循环。这个测试是合法的，因为 `vp` 和指针常量都

指向同一数组中的元素（事实上，这个指针常量所指向的是数组最后一个元素后面的那个内存位置，虽然在最后一次比较时，`vp` 也指向了这个位置，但由于我们此时未对 `vp` 执行间接访问操作，所以它是安全的）。使用 `!=` 操作符代替 `<` 操作符也是可行的，因为如果 `vp` 未到达它的最后一个值，这个表达式的结果总是假的。

现在考虑下面这个循环：

```
for( vp = &values[N_VALUES]; vp > &values[0]; )
    *--vp = 0;
```

它和前面那个循环所执行的任务相同，但数组元素将以相反的次序清除。我们让 `vp` 指向数组最后那个元素后面的内存位置，但在对它进行间接访问之前先执行自减操作。当 `vp` 指向数组第 1 个元素时，循环便告终止，不过这发生在第 1 个数组元素被清除之后。

有些人可能会反对像 `*--vp` 这样的表达式，觉得它的可读性较差。但是，如果对其进行“简化”，看看这个循环会发生什么：

```
for( vp = &values[N_VALUES - 1]; vp >= &values[0]; vp-- )
    *vp = 0;
```

现在 `vp` 指向数组最后一个元素，它的自减操作放在 `for` 语句的调整部分进行。这个循环存在一个问题，你能发现它吗？

**警告：**

在数组第 1 个元素被清除之后，`vp` 的值还将减去 1，而接下去的一次比较运算是用于结束循环的。但这就是问题所在：比较表达式 `vp >= &values[0]` 的值是未定义的，因为 `vp` 移到了数组的边界之外。标准允许指向数组元素的指针与指向数组最后一个元素后面的那个内存位置的指针进行比较，但不允许与指向数组第 1 个元素之前的那个内存位置的指针进行比较。

实际上，在绝大多数 C 编译器中，这个循环将顺利完成任务。然而，你还是应该避免使用它，因为标准并不保证它可行。你迟早可能遇到一台这个循环将失败的机器。对于负责可移植代码的程序员而言，这类问题简直是个恶梦。

## 6.14 总结

计算机内存中的每个位置都由一个地址标识。通常，邻近的内存位置合成一组，这样就允许存储更大范围的值。指针就是它的值表示内存地址的变量。

无论是程序员还是计算机都无法通过值的位模式来判断它的类型。类型是通过值的使用方法隐式地确定的。编译器能够保证值的声明和值的使用之间的关系是适当的，从而帮助我们确定值的类型。

指针变量的值并非它所指向的内存位置所存储的值。我们必须使用间接访问来获得它所指向位置存储的值。对一个“指向整型的指针”施加间接访问操作的结果将是一个整型值。

声明一个指针变量并不会自动分配任何内存。在对指针执行间接访问前，指针必须进行初始化：或者使它指向现有的内存，或者给它分配动态内存。对未初始化的指针变量执行间接访问操作是非法的，而且这种错误常常难以检测。其结果常常是一个不相关的值被修改。这种错误是很难被调试发现的。

NULL 指针就是不指向任何东西的指针。它可以赋值给一个指针，用于表示那个指针并不指向任何值。对 NULL 指针执行间接访问操作的后果因编译器而异，两个常见的后果分别是返回内存位置零的值以及终止程序。

和任何其他变量一样，指针变量也可以作为左值使用。对指针执行间接访问操作所产生的值也是个左值，因为这种表达式标识了一个特定的内存位置。

除了 NULL 指针之外，再也没有任何内建的记法来表示指针常量，因为程序员通常无法预测编译器会把变量放在内存中的什么位置。在极少见的情况下，我们偶尔需要使用指针常量，这时我们可以通过把一个整型值强制转换为指针类型来创建它。

在指针值上可以执行一些有限的算术运算。你可以把一个整型值加到一个指针上，也可以从一个指针减去一个整型值。在这两种情况下，这个整型值会进行调整，原值将乘以指针目标类型的长度。这样，对一个指针加 1 将使它指向下一个变量，至于该变量在内存中占几个字节的大小则与此无关。

然而，指针运算只有作用于数组中其结果才是可以预测的。对任何并非指向数组元素的指针执行算术运算是非法的（但常常很难被检测到）。如果一个指针减去一个整数后，运算结果产生的指针所指向的位置在数组第一个元素之前，那么它也是非法的。加法运算稍有不同，如果结果指针指向数组最后一个元素后面的那个内存位置仍是合法（但不能对这个指针执行间接访问操作），不过再往后就不合法了。

如果两个指针都指向同一个数组中的元素，那么它们之间可以相减。指针减法的结果经过调整（除以数组元素类型的长度），表示两个指针在数组中相隔多少个元素。如果两个指针并不是指向同一个数组的元素，那么它们之间进行相减就是错误的。

任何指针之间都可以进行比较，测试它们相等或不相等。如果两个指针都指向同一个数组中的元素，那么它们之间还可以执行<、<=、>和>=等关系运算，用于判断它们在数组中的相对位置。对两个不相关的指针执行关系运算，其结果是未定义的。

## 6.15 警告的总结

1. 错误地对一个未初始化的指针变量进行解引用。
2. 错误地对一个 NULL 指针进行解引用。
3. 向函数错误地传递 NULL 指针。
4. 未检测到指针表达式的错误，从而导致不可预料的结果。
5. 对一个指针进行减法运算，使它非法地指向了数组第 1 个元素的前面的内存位置。

## 6.16 编程提示的总结

1. 一个值应该只具有一个意思。
2. 如果指针并不指向任何有意义的东西，就把它设置为 NULL。

## 6.17 问题

- ☞ 1. 如果一个值的类型无法简单地通过观察它的位模式来判断，那么机器是如何知道应

该怎样对这个值进行操纵的？

- 2. C 为什么没有一种方法来声明字面值指针常量呢？
- 3. 假定一个整数的值是 244。为什么机器不会把这个值解释为一个内存地址呢？
- 4. 在有些机器上，编译器在内存位置零存储 0 这个值。对 NULL 指针进行解引用操作将访问这个位置。这种方法会产生什么后果？
- 5. 表达式(a)和(b)的求值过程有没有区别？如果有的话，区别在哪里？假定变量 offset 的值为 3。

```
int    i[ 10 ];
int    *p = &i[ 0 ];
int    offset;

p += offset;    (a)
p += 3;         (b)
```

- 6. 下面的代码段有没有问题？如果有的话，问题在哪里？

```
int    array[ARRAY_SIZE];
int    *pi;

for(pi=&array[0];pi<&array[ARRAY_SIZE];)
    *++pi=0;
```

- 7. 下面的表显示了几个内存位置的内容。每个位置由它的地址和存储于该位置的变量名标识。所有数字以十进制形式表示。  
使用这些值，用 4 种方法分别计算下面各个表达式的值。首先，假定所有的变量都是整型，找到表达式的右值，再找到它的左值，给出它所指定的内存位置的地址。接着，假定所有的变量都是指向整型的指针，重复上述步骤。注意：在执行地址运算时，假定整型和指针的长度都是 4 个字节。

变量	地址	内容	变量	地址	内容
a	1040	1028	o	1096	1024
c	1056	1076	q	1084	1072
d	1008	1016	r	1068	1048
e	1032	1088	s	1004	2000
f	1052	1044	t	1060	1012
g	1000	1064	u	1036	1092
h	1080	1020	v	1092	1036
i	1020	1080	w	1012	1060
j	1064	1000	x	1072	1080
k	1044	1052	y	1048	1068
m	1016	1008	z	2000	1000
n	1076	1056			



	表达式	整型		整型指针	
		右值	左值地址	右值	左值地址
a.	m				
b.	v + 1				
c.	j - 4				
d.	a - d				
e.	v - w				
f.	&c				
g.	&e + 1				
h.	&o - 4				
i.	&( f + 2 )				
j.	*g				
k.	*k + 1				
l.	*( n + 1 )				
m.	*h - 4				
n.	*( u - 4 )				
o.	*f - g				
p.	*f - *g				
q.	*s - *q				
r.	*( r - t )				
s.	y > i				
t.	y > *i				
u.	*y > *i				
v.	**h				
w.	c++				
x.	++c				
y.	*q++				
z.	(*q)++				
aa.	*++q				
bb.	++*q				
cc.	*++*q				
dd.	++* (*q)++				

## 6.18 编程练习

- ★★★ 1. 请编写一个函数，它在一个字符串中进行搜索，查找所有在一个给定字符集中出现的字符。这个函数的原型应该如下：

```
char *find_char( char const *source,
                 char const *chars );
```

它的基本想法是查找 `source` 字符串中匹配 `chars` 字符串中任何字符的第 1 个字符，函数然后返回一个指向 `source` 中第 1 个匹配所找到的位置的指针。如果 `source` 中的所有字符均不匹配 `chars` 中的任何字符，函数就返回一个 `NULL` 指针。如果任何一个参数为 `NULL`，或任何一个参数所指向的字符串为空，函数也返回一个 `NULL` 指针。

举个例子，假定 `source` 指向 `ABCDEF`。如果 `chars` 指向 `XYZ`、`JURY` 或 `QQQQ`，函数就返回一个 `NULL` 指针。如果 `chars` 指向 `XRCQEF`，函数就返回一个指向 `source` 中 `C` 字符的指针。参数所指向的字符串是绝不会被修改的。

碰巧，C 函数库中存在一个名叫 `strpbrk` 的函数，它的功能几乎和这个你要编写的函数一模一样。但这个程序的目的是让你自己练习操纵指针，所以：

- 你不应该使用任何用于操纵字符串的库函数（如 `strcpy`, `strcmp`, `index` 等）。
- 函数中的任何地方都不应该使用下标引用。

- ★★★ 2. 请编写一个函数，删除一个字符串的一部分。函数的原型如下：

```
int del_substr( char *str, char const *substr )
```

函数首先应该判断 `substr` 是否出现在 `str` 中。如果它并未出现，函数就返回 0；如果出现，函数应该把 `str` 中位于该子串后面的所有字符复制到该子串的位置，从而删除这个子串，然后函数返回 1。如果 `substr` 多次出现在 `str` 中，函数只删除第 1 次出现的子串。函数的第 2 个参数绝不会被修改。

举个例子，假定 `str` 指向 `ABCDEFGH`。如果 `substr` 指向 `FGH`、`CDF` 或 `XABC`，函数应该返回 0，`str` 未作任何修改。但如果 `substr` 指向 `CDE`，函数就把 `str` 修改为指向 `ABFG`，方法是把 `F`、`G` 和结尾的 `NUL` 字节复制到 `C` 的位置，然后函数返回 1。不论出现什么情况，函数的第 2 个参数都不应该被修改。

和上题的程序一样：

- 你不应该使用任何用于操纵字符串的库函数（如 `strcpy`, `strcmp`，等）。
- 函数中的任何地方都不应该使用下标引用。

一个值得注意的是，空字符串是每个字符串的一个子串，在字符串中删除一个空子串字符串不会发生变化。

- 🔗 ★★★ 3. 编写函数 `reverse_string`，它的原型如下：

```
void reverse_string( char *string );
```

函数把参数字符串中的字符反向排列。请使用指针而不是数组下标，不要使用任何 C 函数库中用于操纵字符串的函数。提示：不需要声明一个局部数组来临时存储参数字符串。

- ★★★ 4. 质数就是只能被 1 和本身整除的整数。Eratosthenes 筛选法是一种计算质数的有效方法。这个算法的第 1 步就是写下所有从 2 至某个上限之间的所有整数。在算法的剩余部分，你遍历整个列表并剔除所有不是质数的整数。

后面的步骤是这样的。找到列表中的第 1 个不被剔除的数（也就是 2），然后将列表后面所有逢双的数都剔除，因为它们都可以被 2 整除，因此不是质数。接着，再回到列表的头部重新开始，此时列表中尚未被剔除的第 1 个数是 3，所以在 3 之后把每逢第 3 个数（3 的倍数）剔除。完成这一步之后，再回到列表开头，3 后面的下一个数是 4，但它是 2 的倍数，已经被剔除，所以将其跳过，轮到 5，将所有 5 的倍数剔除。这样依次类推、反复进行，最后列表中未被剔除的数均为质数。

编写一个程序，实现这个算法，使用数组表示你的列表。每个数组元素的值用于标记对应的数是否已被剔除。开始时数组所有元素的值都设置为 TRUE，当算法要求“剔除”其对应的数时，就把这个元素设置为 FALSE。如果你的程序运行于 16 位的机器上，小心考虑是不是需要把某个变量声明为 long。一开始先使用包含 1000 个元素的数组。如果你使用字符数组，使用相同的空间，你将会比使用整数数组找到更多的质数。你可以使用下标来表示指向数组首元素和尾元素的指针，但你应该使用指针来访问数组元素。

注意除了 2 之外，所有的偶数都不是质数。稍微多想一下，你可以使程序的空间效率大为提高，方法是数组中的元素只对应奇数。这样，在相同的数组空间内，你可以寻找到的质数的个数大约是原先的两倍。

- ★★ 5. 修改前一题的 Eratosthenes 程序，使用位的数组而不是字符数组，这里要用到第 5 章编程练习中所开发的位数组函数。这个修改使程序的空间效率进一步提高，不过代价是时间效率降低。在你的系统中，使用这个方法，你所能找到的最大质数是多少？
- ★★ 6. 大质数是不是和小质数一样多？换句话说，在 50 000 和 51 000 之间的质数是不是和 1 000 000 和 1 001 000 之间的质数一样多？使用前面的程序计算 0 到 1 000 之间有多少个质数？1 000 到 2 000 之间有多少个质数？以此每隔 1 000 类推，到 1 000 000（或是你的机器上允许的最大正整数）有多少个质数？每隔 1 000 个数中质数的数量呈什么趋势？