8.2 指针的声明和使用

指针是 C 语言中非常有特色的一种数据类型，也是一种非常灵活的机制。指针在一定程度上给了程序员直接访问内存的权限，从而使问题的解决变得直截了当。但也正是这种放权，又引出了另外一些问题，比如潜在的权限滥用等。不过，利弊权衡之下，还是希望能拥有这样的权利，以方便问题的解决。下面就来详细讨论有关指针的问题。

在学习指针之前，请读者一定要牢记这一点：**指针就是一个地址**。这对后续的学习有极大的帮助。

8.2.1 指针变量的声明

为了说明指针是一个什么样的地址，先来复习一下在第 2 章中学过的关于内存的知识。假设有声明

```
int grade = 3;
```

那么变量 grade 的内存映像就如图 8-1 所示。

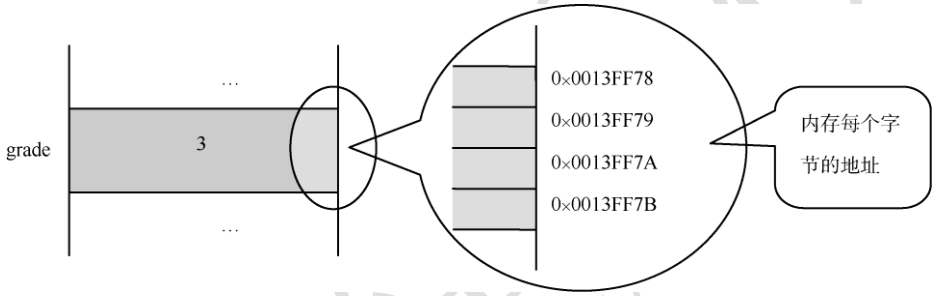


图 8-1 变量在 32 位机上的内存映像

从图中可以看到，grade 占了 4 个字节的内存单元，每一个字节都有一个唯一编号，这个编号就是该字节单元的**地址**（*address*）。而对于整数 grade，其首字节的地址 0x0013FF78 就是它的地址。

从字面上来看，变量的地址就是一个 unsigned int 型值。但是在程序中不能直接使用这个常量，况且也不能保证程序每次运行时 grade 都在同一地址上。不过，这个地址总是可以获取的，C 语言就通过一个指针变量来保存它。

```
int *p;
p = &grade;
```

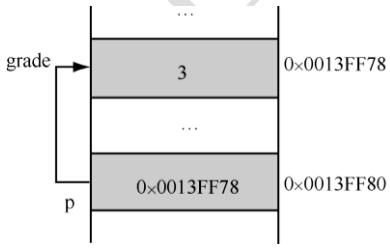


图 8-2 指针和变量

声明 int *p 中，运算符“*”指明了变量 p 不是一个普通的整型变量，而是一个指针。赋值语句 p = &grade 也不是普通的值的复制，而是通过取地址运算符“&”将 grade 单元的地址存储到变量 p 中。按图 8-1，p 的值就是 0x0013FF78。一旦经过这样的赋值，指针 p 就指向了变量 grade。图 8-2 示意了这种关系。

从图 8-2 还可以读到另外的信息。指针变量 p 也是一个普通的内存变量，也有自己的地址（此例中是 0x0013FF80）。p 的大小能够容纳一个地址。在 32 位的机器上，一个地址的大小是 32 位的，所以 p 单元的大小也是 32 位，恰好是一个 int 类型的大小。但是，这不能说明二者是可以相互操作的，因为还有类型的限制。

指针变量的形式化定义为：

基类型 *变量名;

其中，基类型可以是除枚举类型外的任意合法的类型，包括指针类型本身。

如果要同时声明多个指针变量，那么一定要在每一个指针变量名前都加上“*”号，例如：

```
int *q, *r, t;
```

在上面的声明中，变量 q 和 r 都是指针，而 t 却是一个普通整型变量。

> “*”运算符修饰的是变量，而非其基类型。一个好的建议是：将指针和非指针变量分开声明，以免发生理解上的错误。

一个指针变量的类型可以看作由两部分组成：

（1）首先，它是个指针；

（2）其次，该指针变量指向了一个特定类型的变量。

也就是说，指针受到双重约束。据此，可以这样来解释指针 p：p 是一个指针，它指向了一个整型变量 grade；而 grade 占据了 4 字节的内存单元，并且由于类型声明的限制，这 4 个字节中的每一位都是一个整数的一部分，不应该有其他解释。

> 一些初学者可能犯的一个错误就是使用 T1 类型的指针指向 T2 类型的变量。例如：
>
> ```
> int *ip = &grade;
> double *fp;
> fp = ip; //非法，因为类型不同
> ```
>
> 如果 fp 真指向了变量 grade 占据的内存，那么，由于 fp 的基类型是 double，因此它“认为”自己指向了一个浮点单元。这样一来，从指针 fp 的角度来观察原来 grade 占据的内存，它就被重新解释了，也就是说，从地址 0x0013FF78 开始的连续 8 个字节被当作了一个 double 数。而原来的 grade 单元只占据了 4 个字节！这是一个非常严重的隐患：当强行往 fp 指向的单元中存入一个 double 数时，除了占据 grade 单元的 4 字节外，还要延伸占据不知道属于哪个数据的单元，这可能导致灾难性的结果。所以，不同类型的指针之间的赋值是禁止的。

8.2.2 指针的使用

假设有声明。

```
int grade;
int *p;
```

为了要使用指针，首先必须让指针指向某个单元，这一点可以通过语句

```
p = &grade;
```

实现，其中“&”是取地址运算符。要想通过指针访问它指向的单元（称之为间接访问），可以使用表达式*p。例如：

```
*p = 100;
```

上述赋值语句的作用是：“把值 100 存入指针变量 p 指向的单元（也就是 grade）内”。它与语句：

```
grade = 100;
```

在效果上是等价的。其实，表达式*p 和 grade 是完全等效的。实际上，二者引用的是同一个对象。也就是说，表达式*p 的结果是个左值，可以出现在赋值号的左右两边。相较之下，类似于 a+b 这样的算术表达式的结果是个临时常量单元，不能作为左值对待。

容易让人混淆的是表达式 p 和*p。在图 8-2 中：

- 表达式 p 表示指针变量本身，它的值是 0x0013FF78，它的地址是 0x0013FF80；
- 表示式*p 表示了指针 p 指向的单元，也就是 grade，它的值是 100，它的地址是 0x0013FF78，也就是 p 的值。

【例 8-1】指针的初级使用。

```
//8-1.c
#include <stdio.h>
```

```
int main()
{
    int a = 11111;
    int *p;

    p = &a;
    printf("a=%d\n", a);
    printf("*p=%d\n", *p);
    printf("&a=0x%p, p=0x%p\n", &a, p); //输出变量 a 的地址

    return 0;
}
```

程序运行的结果是：

```
a=11111
*p=11111
&a=0x0022FF18, p=0x0022FF18
```

1. %p 格式的意思是将参数以指针（地址）的形式输出；2. 在每次运行时输出的地址值可能不同。

在使用指针时需要注意这一点：指针变量在使用前必须被初始化。这可以通过声明时的初始化或者赋值来完成。未初始化的指针不指向任何单元，此时使用指针可能会引起灾难性的结果。

【例 8-2】使用未初始化的指针存在安全隐患。

```
//8-2.c
#include <stdio.h>

int main()
{
    char *p;

    *p ='A'; //这个赋值操作将会引起一个运行时错误。
    putchar(*p);

    return 0;
}
```

例 8-2 在用 VC9 编译时会得到如下编译结果：

```
8-2.c(8) : warning C4700: local variable 'p' used without having been initialized
```

从结果上看，由于没有致命错误，因此还是生成了可执行代码。但在调试模式下运行它，则会弹出类似于图 8-3 的出错信息。

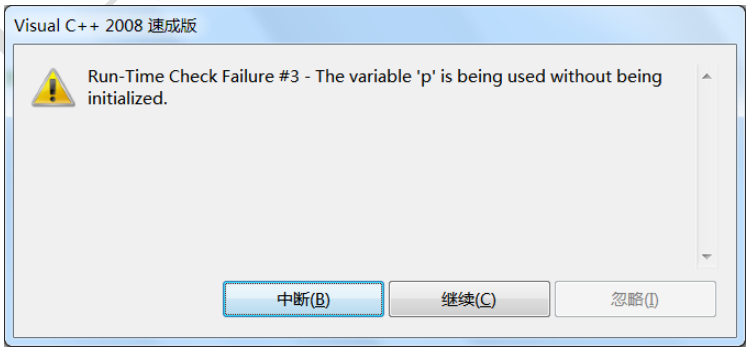


图 8-3　未初始化的指针引起一个异常

gcc 不会对【例 8-2】程序报出错误，甚至能正常运行。但这并不能说明程序是正确的。

初始化指针时，尽量在定义时确定它指向的变量，例如：

```
int *p = &grade;
```

或者如果当前还不能确定指针的指向，那么就使用语句：

```
int *p = NULL;
```

符号常量 NULL 在头文件 stdio.h 中被预定义，其值是 0，其含义是“空”。一个使用指针的好习惯是：在使用它之前，先检测它是否为空。

【例 8-3】检测空指针。

```
//8-3.c
#include <stdio.h>

int main()
{
     char *p = NULL;

     if (p != NULL) //可以写成 if (p)，但这看起来难以阅读
     {
          *p = 'A';
          printf("%c\n", *p);
     }
     else
          fprintf(stderr, "Uninitialized pointer.\n");

     return 0;
}
```

运行结果如下：

```
Uninitialized pointer.
```

> 程序中使用了 fprintf()和预定义变量 stderr。fprintf()函数的功能非常类似于 printf()。不过 printf()是向标准输出（一般是显示终端）打印，而 fprintf()是向指定的文件打印。变量 stderr 代表一个标准出错流。在一般情况下，标准出错流被定向到显示终端。

8.2.3 const 作用于指针

除了修饰普通变量，const 关键字也可用于修饰指针变量。由于指针的类型包含基类型和“*”两个修饰符，因此 const 也可以分别或同时修饰这两部分。

1．指向常量的指针变量

设有如下声明：

```
const int *pc;
```

在声明中，指针 pc 的基类型是 const int，因此这里定义了一个指向常量的指针变量。也就是说，从指针 pc 的角度来看，它指向的那个整数是个常量，不能被修改。因此，语句

```
*pc = 1;
```

是错误的。

因为声明中的 const 修饰符约束的是关键字 int 而非标识符 pc，所以指针 pc 本身仍然是个变量，可以指向任何一个整型变量。例如：

```
int a = 1, b = 2;
const int *pc;
pc = &a; //ok
pc = &b; //ok
```

此例中，无论 pc 指向变量 a 还是 b，表达式*pc 都被视为常量。因此，类似于

```
*pc = 1;
```

```
++*pc;
```

这样的语句都是错误的，因为它们都试图（从 pc 的角度出发）去修改常量。不过，相信读者已经注意到了这个事实：变量 a 和 b 都不是常量，它们是可以被任意修改的变量。

一般情况下，指向常量的指针都会指向某个常量。例如：

```
const int a = 1;
const int *pc = &a;
```

【例 8-4】指向常量的指针变量。

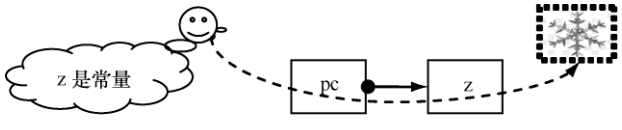
```
//8-4.c
#include <stdio.h>

int main()
{
    const int x = 100;
    const int y = 200;
    int z = 300;
    const int *pc;

    printf("x = %d, y = %d, z = %d\n", x, y, z);

    pc = &x;
    printf("pc = &x, *pc = %d\n", *pc);

    pc = &y;
    printf("pc = &y, *pc = %d\n", *pc);

    pc = &z;
    printf("pc = &z, *pc = %d\n", *pc);

    z = 400;
    //*pc = 400; //这条语句将会引起一个编译错误

    return 0;
}
```

程序的运行结果如下：

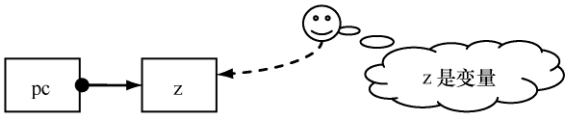
```
x = 100, y = 200, z = 300
pc = &x, *pc = 100
pc = &y, *pc = 200
pc = &z, *pc = 300
```

从代码中我们可以看到，指针 pc 确实是个变量，它可以被修改为指向任何一个整数（因为它的基类型是整数），包括常量和变量。并且一旦它指向了一个变量，如代码中的 z，那么如果通过 pc 来访问 z，则*pc 被视为一个常量，因而对*pc 的改变是非法的；然而，直接对 z 改变却是合法的。图 8-4 解释了这种现象。

为了避免理解上的错误，一般情况下，指向常量的指针变量都会指向本身就被 const 约束的常量。



(a) 通过pc来访问z，z就是一个常量



(b) 直接访问z，z就是一个变量

图 8-4　指向常量的指针与变量的关系

放开代码中的注释，会得到如下编译错误（VC9）：

```
error C2166: l-value specifies const object
```

2．指向变量的常量指针

如有声明

```
int * const cp;
```

那么，const 修饰的是一个基类型为 int *的变量 cp。也就是说，声明了一个常量指针 cp，它本身是不能被改变的；但 cp 指向的整型变量可以被任意修改。可以看出，指向变量的常量指针具有“从一而终”的特性。图 8-5 示意了这种特性。

【例 8-5】指向变量的常量指针。

```
//8-5.c
#include <stdio.h>

int main()
{
    int x = 100;
    int y = 200;
    int * const cp = &x; //常量指针必须初始化

    printf("x = %d, y = %d\n", x, y);
    printf("cp = &x, *cp = %d\n", *cp);

    *cp = y;
    printf("*cp = y, *cp = %d\n", *cp);

    //cp = &y; //这条语句将会引起一个编译错误

    return 0;
}
```

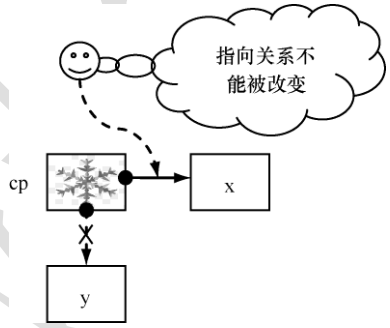


图 8-5　常量指针与变量的关系

一旦确立就不能被改变

程序的运行结果是：

```
x = 100, y = 200
cp = &x, *cp = 100
*cp = y, *cp = 200
```

放开代码中的注释将会得到一个与【例 8-5】中一样的编译错误。

3．指向常量的常量指针

指向常量的常量指针被声明为：

```
const int a = 100;
const int * const cpc = &a;
```

声明非常清楚地表明了指针本身以及它指向的单元都是常量，二者都不能被改变。

8.3　指针的运算

指针是一个地址，因此它能参与所有地址相关的运算。典型的运算是赋值和比较。本节除了讨论上述两种运算外，还将讨论其他运算。

8.3.1　指针的赋值运算

假设有声明：

```
int a = 100, b = 200;
```

```
int *p = &a, *q = &b;
```

那么，与指针相关的赋值操作有两种，如例 8-6。

【例 8-6】指针的赋值操作。

```
//8-6.c
#include <stdio.h>

int main()
{
    int a = 100, b = 200;
    int *p = &a, *q = &b;

    printf("a=%d, b=%d\n", a, b);
    printf("*p=%d, *q=%d\n", *p, *q);

    *p = *q;
    printf("After *p=*q:\na=%d, b=%d\n", a, b);
    printf("*p=%d, *q=%d\n", *p, *q);

    a = 100; b = 200;  //恢复初始值
    p = q;
    printf("After p=q:\na=%d, b=%d\n", a, b);
    printf("*p=%d, *q=%d\n", *p, *q);

    return 0;
}
```

程序的运行结果如下：

```
a=100, b=200
*p=100, *q=200
After *p=*q:
a=200, b=200
*p=200, *q=200
After p=q:
a=100, b=200
*p=200, *q=200
```

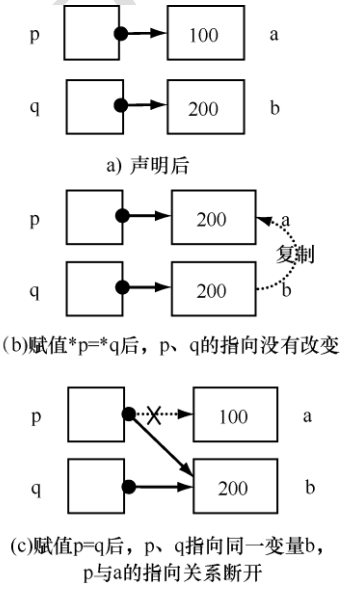


图 8-6　指针的赋值操作

程序中，赋值语句

```
*p = *q
```

的含义非常明确，就是将指针 q 指向单元的内容复制到指针 p 指向的单元，它等价于

```
a = b;
```

而赋值语句

```
p = q;
```

则容易让人混淆。这句话的真实含义是将变量 q 的内容复制到变量 p 中。由于 q 是一个地址，因此这条赋值语句使变量 p 和 q 装有相同的地址。换句话说，就是指针 p 和 q 指向了相同的单元，而指针 p 与原指向的变量 a 的联系也就断开了。图 8-6 示意了这两种赋值的不同。

8.3.2　指针的比较运算

两个地址的比较具有实际意义，因此，C 语言的所有关系运算符都可以作用于指针。

- 相等与不等。表达式 p==q 和 p!=q 分别表示了指针 p 和 q 是否存储了相同的地址，也就是说，是否指向了相同的单元；
- 比较大小。例如，p>q 的意思是地址 p 是否大于地址 q，也就是 p 指向的单元在内存中的位置是否在 q 指向的单元之后。其他大小比较与此类似。

与赋值相似，表达式 p>q 和*p>*q 具有完全不同的含义。

> 指针的比较只能在两个指针（地址）之间进行，将指针与其他类型的值进行比较是没有意义的。

8.3.3 指针的算术运算

用于指针的算术运算只有加法和减法两种，包括自加和自减。

1．指针和整数的加减运算

这里仅以加法运算为例。指针进行加法运算的形式化定义为：

指针变量 + 整型表达式

++指针变量

表达式得到的结果仍是一个地址值，可以用另一个指针变量来接收。例如有声明

```
int *p;
```

那么表达式 p + 2 的意思是从地址 p 开始的单元向后数第 2 个单元的地址。需要特别说明的是，这里的单元不是以字节为单位，而是以 p 的基类型的大小为单位，此例中就是一个整数的大小，一般为 4 字节。因此，如果 p=0x0013FF78，那么 p + 2 得到的结果是：

```
0x0013FF78 + 2 * 4 = 0x0013FF80
```

而非想象的那样，等于 0x0013FF78 + 2 = 0x0013FF7A。

> 指针的加减运算是由编译器自动完成的。所以这里建议读者不要去进行手工计算，因为不同的环境定义了不同的整数单元的大小，所以手工计算的结果有可能换一种环境就是错误的。

现在来看看自加运算。假设有声明

```
int a, b;
int *p = &a;
```

那么表达式++p 完成的操作是：

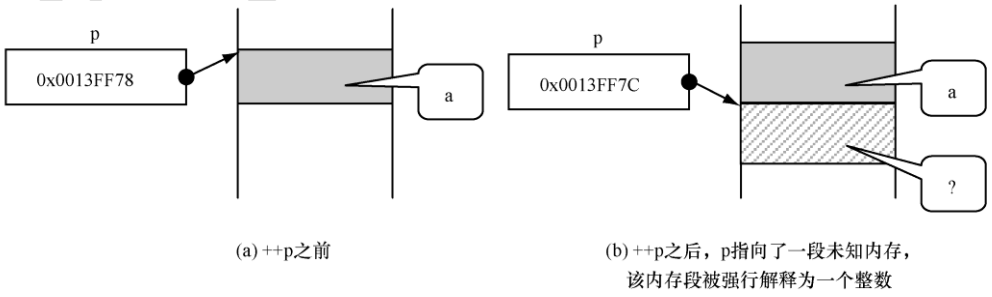
（1）指针 p 自身的值加上一个整型单元的大小，这样使 p 指向了紧接在 a 单元后面的那个整型单元；

（2）整个自加表达式的值是 a 后面那个整型单元的地址。

后缀自加表达式 p++具有类似的功能，只不过整个后缀表达式的值仍是 a 的地址。这非常符合++运算符作为前缀和后缀的不同特点。

相信读者已经注意到了上面描述中的这句话：“a 后面的单元”。现在问题来了：a 后面的那个单元是哪个单元呢？会是 b 吗？

是不是 b 取决于编译器的安排。但无论如何，我们可能已经意识到了危险的存在：++p 使 p 指针获得了另一个地址，该地址标示的单元在指针 p 基类型的约束下被强行解释成为一个整数单元。但事实上，这个单元是否是个整型单元完全是未知的！图 8-7 示意了这种情况。



(a) ++p之前　　(b) ++p之后，p指向了一段未知内存，该内存段被强行解释为一个整数

图 8-7　指针加法可能产生的结果

所以，一定要注意指针的加法潜在的危险性。如果要使用新地址的话，那么一定要清楚该地址是否在掌控之中。然而，程序员并不是总能掌控指针的这种行为，这为指针的使用带来了隐患。

请再来看看以下的运算。有声明

```
int a = 1;
```

```
int *p = &a;
```

那么表达式

（1）*p + 2

（2）*(p + 2)

的结果分别是什么呢?

在表达式（1）中，由于“*”运算符的优先级高于“+”运算符，因此“*”首先与 p 结合，得到结果 1，最终表达式的结果为 3。在表达式(2)中，由于()提升了“+”号的优先级，因此 p 首先与 2 相加，得到一个地址，而整个表达式的值是取该地址下的值。正如前面提到的那样，这个地址的可用性值得怀疑。

有了上面的经验后，读者能分辨出*p++和(*p)++的区别吗?

指针与整数的减法（含自减）与加法类似，这里就不再赘述了。

2．指针间的减法运算

两个指针之间相减是有意义的,它的结果是两个地址之间的以单元(而不是字节)计的间隔。例如有声明：

```
double a[10];
double *p=&a[0], *q=&a[9];
```

那么 q-p 的结果是 9。实际上，将上面声明中的 double 换成任何类型其结果都是 9。

除上面讲到的运算之外，指针和整数的乘除法以及两个指针间的加、乘、除法是没有任何意义的。

8.4 指针和数组

用一种简单的观点来观察 C 语言的数组，可以发现其实它就是一个内存块。例如声明

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

数组 a 在 32 位机器中共占据 20 字节的内存块，而该内存块又被视为由 5 个单元构成，每 4 个字节单元可以容纳一个整数。

现在我们都已知道,内存可以由其地址来访问。因此,可以使用指针指向数组所在的内存块，从而方便地访问到数组中的每一个单元。

8.4.1 指向数组元素的指针

定义一个指向数组元素的指针与定义普通指针一样简单。设有如下数组和指针声明：

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

那么，要使 p 指向数组的某个元素，可以用这样的语句：

```
p = &a[0];
```

在上述语句中，由于[]运算符的优先级高于“&”运算符，所以[]首先与 a 结合，得到数组 a 的第一个元素，然后“&”作用于元素 a[0]上，最后将其地址赋值给指针 p。p 和 a 的关系如图 8-8 所示。

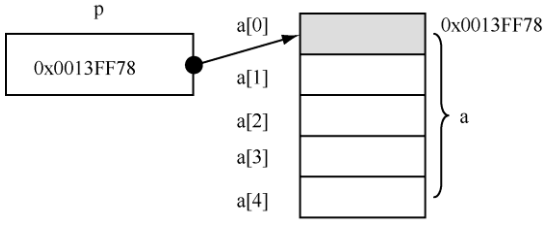


图 8-8 指向数组元素的指针

同样的道理，可以通过赋值使 p 指向数组 a 的任何一个元素，例如：

```
int i = 3;
p = &a[i];
```

从图 8-8 中还可以看到一个事实：因为数组的元素是连续存放的，所以可以认为这段内存是可以掌控的。那么一旦指针 p 指向了 a 的某一个元素，就可以通过对指针的算术运算来“移动”指针，让指针在数组元素间自由移动，这样就可以非常方便地访问数组元素。

【例 8-7】通过指针来访问数组元素。

```
//8-7.c
#include <stdio.h>

int main()
{
    int a[5] = { 11, 22, 33, 44, 55 };
    int *p;
    int i;

    p = &a[0];
    for (i = 0; i < 5; i++) printf("%4d", *p++);

    return 0;
}
```

程序的运行结果如下：

```
  11  22  33  44  55
```

请注意代码中的表达式*p++。“*”和“++”都是单目运算符，且具有相同的优先级和从右到左的结合性。所以变量 p 先与“++”运算符结合，完成指针的自加，然后返回 p 先前的地址值，最后与“*”结合，得到先前那个元素的值。例如，当 i=0 时，p 指向 a[0]，执行*p++后，这个表达式的值是 a[0]，而 p 却已经指向了 a[1]。

在例 8-7 中，当 i=5 循环结束时，p 已经指向了数组 a 以外的单元。此时直接使用 p 是相当危险的。请看例 8-8。

【例 8-8】指针移动超界。

```
//8-8.c
#include <stdio.h>

int main()
{
    int a[5];
    int *p;
    int i;

    p = &a[0];
    for (i = 0; i < 5; i++) scanf("%d", p++); //注意：p 前没有&

    for (i = 0; i < 5; i++) printf("%4d", *p++);

    return 0;
}
```

程序的运行结果为：

```
11 22 33 44 55
   022935322147307520   02293608
```

显然，这是不正确的结果。代码错在哪里呢？

这里分析一下输入时最后一次循环的结果。当 i = 4 时，p 指向 a[4]，scanf()执行后 i = 5，p 指向 a[4]后面的那个单元，而这个单元并不属于数组 a。那么后面的循环输出错误结果就很容易解释了：代码误将 a[4]后面的 5 个连续的单元当作了数组元素对待，而这 5 个单元里

的值是不定的。修正的方法是在输出循环前加上

```
p = &a[0];
```

这条语句。

> 遗憾的是，C 语言并没有提供一种内部机制来警告程序员指针已经超出了数组的边界。因此，在使用指针来操纵数组时，一定要非常谨慎地控制指针在数组中的移动。

在多数情况下，都会使指针指向数组的首元素。而数组的名字，如例中的 a，被视为一个常量，并且还代表数组首地址，也就是首元素的地址。因此，可以将

```
p = &a[0];
```

简化为：

```
p = a;
```

在这种情况下，可以说：p 指向了数组 a。但一定要非常清楚地认识到这一点：p 实际上是指向了数组 a 的首元素，表达式*p 得到结果是 a 的首元素的值，而非整个数组。实际上，指向数组的指针是存在的，但与上述的定义和用法完全不同。

实际上，C 编译器在处理数组时，是将数组名转换为指针来使用的。也就是说，例中的数组名 a 被当作了一个常量指针，可以认为 a 是这样声明的（只是认为，而不能显式声明）：

```
int * const a;
```

并且有 a≡ &a[0]。

有了上述的关系，那么在程序中，一旦指针 p 和 a 发生了关联，那么 p 和 a 就完全可以互相替代。例如，表达式 a[i]可以写成 p[i]；表达式 p + i 可以写成 a + i。

虽然如此，但 a 和 p 还是有一些重要的不同：

- a 是常量，而 p 是变量；
- a ≡ &a[0]，而 p 可以是 a 中任何一个元素的地址。例如：

```
int a[5] = { 11, 22, 33, 44, 55 };
int *p = &a[2];
```

那么 a[0] = 11，a[2] = 33，而 p[0] = 33，p[2] = 55。

【例 8-9】指针和数组的关系。

```
//8-9.c
#include <stdio.h>

int main()
{
    int a[5] = {11, 22, 33, 44, 55};
    int *p;
    int i;

    for (i = 0; i < 5; i++) printf("%4d", a[i]);
    putchar('\n');

    p = &a[2];
    for (i = 0; i < 3; i++) printf("%4d", p[i]);

    return 0;
}
```

程序的运行结果是：

```
  11  22  33  44  55
  33  44  55
```

数组 a 和指针 p 的关系如图 8-9 所示。

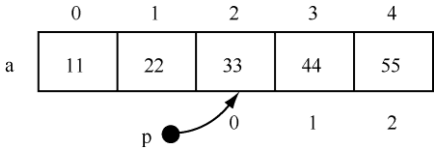


图 8-9　数组 a 和指针 p 的关系

【例 8-10】用指针实现排序数组的折半查找。假设数组已按升序排序。

【解题思路】在【例 7-7】折半查找中，使用了表示数组下标的 begin、end 和 mid3 个游标来指示查找的范围和中间点。在本例中，将使用 3 个指针 head、tail 和 mid 来代替上述游标。

```
//8-10.c
#include <stdio.h>

int find(int a[], int len, int key);

const int LEN = 5;

int main()
{
    int a[] = {11, 22, 33, 44, 55};

    if (find(a, LEN, 33) >0) printf("Found.");
    else printf("Not found.");

    return 0;
}

int find(int a[], int len, int key)
{
    int *head = a, *tail = a + len - 1; //头尾指针
    int *mid;

    while (head <= tail)
    {
        mid = head + (tail - head) / 2; //中间指针
        if (*mid == key) return 1;

        if (*mid < key) //说明待查数在 mid 指针的后面
            head = mid + 1;
        else                    //否则，在前面
            tail = mid - 1;
    }

    return -1; //没找到
}
```

程序的运行结果如下：

```
Found.
```

这里，解析一下 find()中的两个表达式。

（1）tail = a + len – 1。指针运算 a + len 得到的地址是数组 a 最后一个元素后面一个单元的地址，因此需要回退一个单元。

（2）head + (tail – head) / 2。有读者可能想，为什么不直接写成(head + tail) / 2 呢？前面已经提到过，两个指针的加法是没有意义的，而两个指针的减法却有实际意义，就是两个地址间隔了多少个单元。因此，只有采用这样间接的方法才能得到两个指针中点的地址。