3.3.4　数组和指针相关的运算符

以下介绍数组和指针相关的运算符。

▲解引用

单目运算符*被称为**解引用**。

运算符*将指针作为操作数，返回指针所指向的对象或者函数。只要不是返回函数，运算符*的结果都是左值。

从运算符*的操作数的类型中仅仅去掉一个指针后的类型，就是运算符*返回的表达式的类型（参照图 3-13）。

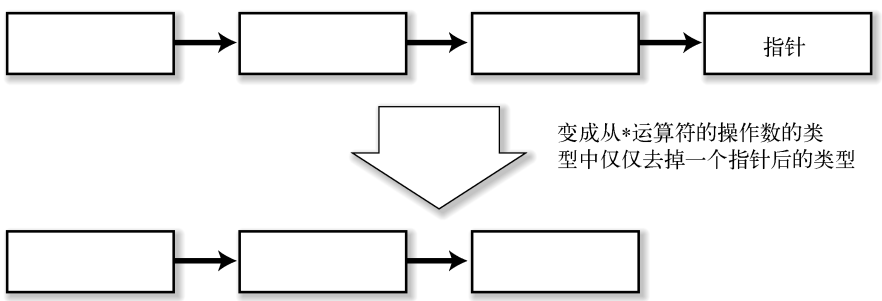

图 3-13　使用解引用而发生的类型的变化

▲地址运算符

单目运算符&被称为**地址运算符**。

&将一个左值作为操作数，返回指向该左值的指针。对左值的类型加上一个指针，就是&运算符的返回类型（参照图 3-14）。

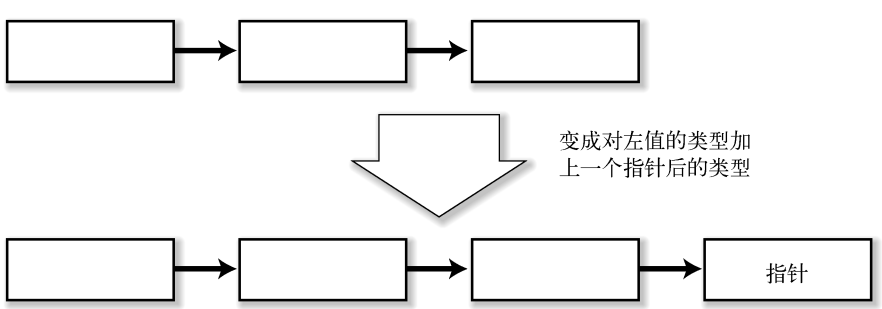

图 3-14　使用地址运算符而发生的类型的变化

地址运算符不能将非左值的表达式作为操作数。

▲下标运算符

后置运算符[]被称为**下标运算符**。

[]将**指针**和整数作为操作数。

```
p[i]
```

是

```
*(p + i)
```

的语法糖，除此以外没有任何其他意义。

对于声明为 int a[10]的数组，使用 a[i]的方式进行访问的时候，由于 a 在表达式中，因此它可以被解读成指针。所以，你可以通过下标运算符访问数组（将指针和整数作为操作数）。

归根结底，p[i]这个表达式就是*(p + i)，所以下标运算符返回的类型是，从 p 的类型去掉一个指针的类型。

▲->运算符

在标准中，似乎并没有定义->运算符的名称，现实中有时它被称为“箭头运算符”。

通过指针访问结构体的成员的时候，会使用->运算符。

```
p->hoge;
```

是

```
(*p).hoge;
```

的语法糖。

利用*p 的*，从指针 p 获得结构体的实体，然后引用成员 hoge。

3.3.5 多维数组

在 3.2.5 节中，我们提到了 **C 语言中不存在多维数组**。

那些看起来像多维数组的其实是“数组的数组”。

这个“多维数组”（山寨货），通常使用 hoge[i][j]的方式进行访问。让我们来看一看这个过程中究竟发生了什么。

```
int hoge[3][5];
```

对于上面这个“数组的数组”，使用 hoge[i][j]这样的方式进行访问（参照图 3-15）。

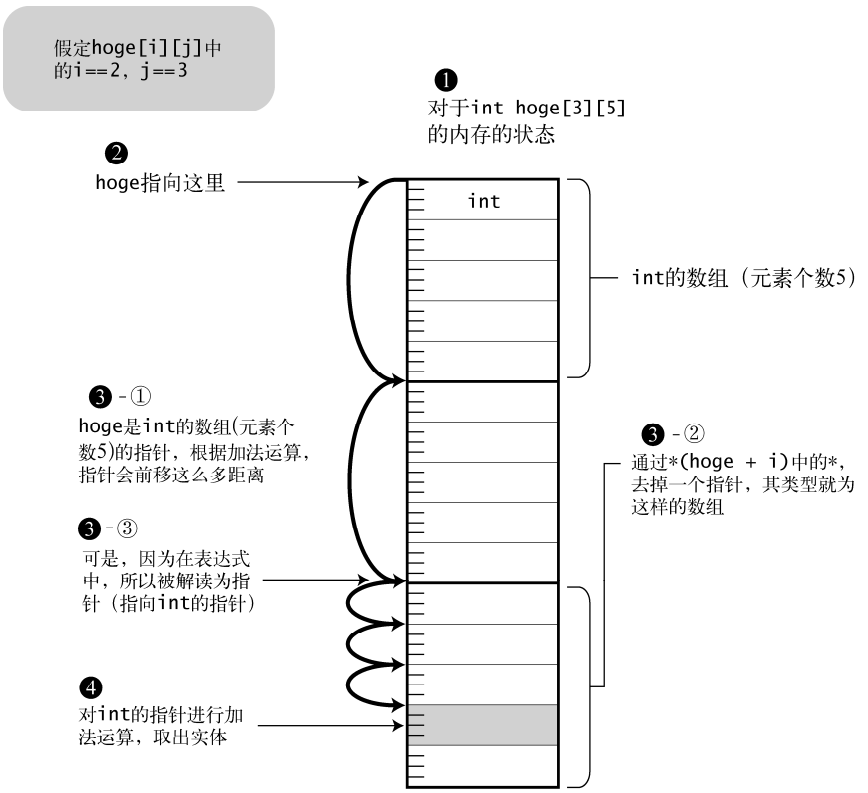

图 3-15　访问多维数组

❶ hoge 的类型为“int 的数组（元素个数 5）的数组（元素个数 3）”。

❷ 尽管如此，**在表达式中数组可以被解读成指针**。因此，hoge 的类型为“指向 int 的数组（元素个数 5）的指针”。

❸ hoge[i]是*(hoge + i)的语法糖。

① 给指针加上 i，就意味着指针前移它指向的类型×i 的距离。hoge 指向的类型为“int 的数组（元素个数 5）”，因此，hoge + i 让指针前移了 sizeof(int[5])×i 的距离。

② 通过*(hoge + i)中的*，去掉一个指针，*(hoge + i)的类型就是“指向 int 的数组（元素个数 5）”。

③ 尽管如此，由于在表达式中，数组可以解读成指针，所以*(hoge + i)的最终类型为“指向 int 的指针”。

❹ (*(hoge + i))[j]和*((*(hoge + i)) + j)其实是相等的，因此，(*(hoge + i))[j]就是“对指向 int 的指针加上 j 后得到的地址上的内容”，其类型为 int。

某些语言中，使用 array[i，j]这样的写法来支持多维数组*。

* Pascal 就支持这样的写法，但是，Pascal 的“多维数组”只不过是“数组的数组”的语法糖。

在 C 中，因为没有多维数组，所以使用“数组的数组”来代替，这样倒也没有什么问题。可是，如果反过来，只有多维数组，而没有“数组的数组”，事情就麻烦了。

比如，将某人一年之中每天的工作时间使用下面这个“数组的数组”来表现*，

* 数组中，月和日是从 0 开始的，只有在输出的时候才可修正。2 月等情况下，数组的元素就会显得冗余，但是不会产生问题。

```
int working_time[12][31];
```

在这里，如果开发一个根据一个月的工作时间计算工资的函数，可以像下面这样将某月的工作时间传递给这个函数，

```
calc_salary(working_time[month]);
```

calc_salary 的原型像下面这样：

```
int calc_salary(int *working_time);
```

这种技巧只有通过“数组的数组”才能实现，多维数组是无能为力的。

补充 运算符的优先级

C 语言中有数量众多的运算符，其优先级分 15 个级别。

和其他语言相比，C 语言的优先级别还是非常多的。很多 C 的参考书，都像表 3-5 这样记载了运算符优先级的内容。

其中，对于优先级“最高”的是()，有相当多的人抱有以下观点：

> 当需要改变原本语法规定的优先级、强制地设定自己需要的优先级的时候，程序员们会使用()。因此，()具有最高的优先级是理所当然的。

这是一种**误解**。

如果()可以这样理解，还有必要特地将它记载在优先级的表中吗？

这个表中的()，（正如 *K&R* 中记述的那样）代表着调用函数的运算符。此时的优先级是指，对于 func(a, b)这样的表达式，func 和()之间的关联强度。

表3-5　运算符优先顺序表（摘录于*K&R* p.65）

运　算　符	连接规则
() [] -> .	从左往右
! ~ ++ -- + - * & (*type*) sizeof	从右往左
* / %	从左往右
+ -	从左往右
<< >>	从左往右
< <= > >=	从左往右
== !=	从左往右
&	从左往右
^	从左往右
¦	从左往右
&&	从左往右
¦¦	从左往右
?:	从右往左
= += -= *= /= %= &= ^= ¦= <<= >>=	从右往左
,	从左往右

注：进行单目运算的+、-、和*的优先级高于进行双目运算的+、-、和*。

此外，我们经常看到类似于下面这样的编码：

```
*p++;
```

究竟是对 p 进行加法运算？还是对 p 所指向的对象（*p）进行加法运算？关于这一点，很多书是这样解释的：

> 尽管*和++的优先级相同，但由于连接规则是从右往左，所以 p 和++先进行连接。因此，被进行加法运算的不是*p，而是 p。

就连 *K&R* 自身也是这样说明的。其实这种说法不太恰当。

根据 BNF（Backus-Naur Form）规则，C 语言标准定义了语法规则，其中也包含了运算符的语法规则。

关于 BNF，由于超出了本书的范围，在此就不做说明了。根据 BNF，后置的++比前置的++和*等运算符的优先级高，()和[]的优先级相同*。

也就是说，关于*p++的运算符优先级，

> 后置的++比*的优先级高，因此，被进行加法运算的不是*p，而是 p。

从语法上来看，这种说法才是比较合理的。

* 参照 *K&R* 中的标准 6.3.2 和 6.3.3。