9.6 C语言的多维数组

有些人声称 C 语言没有多维数组，这是不对的。ANSI C 标准在第 6.5.4.2 节以及第 69 号脚注上表示：

当几个“[]”修饰符连续出现时（方括号里面是数组的范围），就是定义一个多维数组。

9.6.1 但所有其他语言都把这称为“数组的数组”

那些人的意思是 C 语言没有像其他语言一样的多维数组，如 Pascal 或 Ada。在 Ada 中，可以如图 9-5 那样声明一个多维数组。

```
apples : array(0..10, 1..50) of real;

或者声明一个数组的数组：

type vector is array(1..50) of real;
orange : array(0..10) of vector;
```

Ada

图 9-5 Ada 的例子

但是不把苹果和梨混为一谈。在 Ada 中，多维数组和数组的数组是两个完全不同的概念。

Pascal 则采用了一种相反的方法。在 Pascal 中，数组的数组和多维数组是可以完全互换的，并且在任何时候都是等同的。在 Pascal 中，可以像图 9-6 那样声明和访问一个多维数组。

```
var M : array[a..b] of array[c..d] of char;
M[i][j] := c;

习惯上人们采用方便的简写形式：

var M : array[a..b, c..d] of char;
M[i, j] := c;
```

Pascal

图 9-6 Pascal 的例子

The Pascal User Manual and Report[1]清楚地说明了数组的数组与多维数组是等同的，两者可以互换。Ada 语言在这方面的限制更紧一些，它严格地维持了数组的数组和多维数组之间的区别。在内存中它们看上去是一样的，但在哪个类型具有兼容性以及可以被赋值给一个数组的数组的单独的行的问题上，两者存在明显的差别。这有点像在 int 和 float 之间选择变量的类型：所选择的类型最大限度地反映了底层的数据。在 Ada 中，当具有独立可变的下标时，如用笛卡尔坐标确定某一点的位置，一般会选择多维数组。当数据在层次上更加鲜明时，如某个数组具有[12]月[5]周[7]日这样的形式来代表某事物的每日记录，但有时也需要同时操纵整个星期或月时，一般选择数组的数组。

C 语言里面只有一种别的语言称为数组的数组的形式，但 C 语言称它为多维数组。

C 语言的方法多少有点独特：定义和引用多维数组惟一的方法就是使用数组的数组。尽管 C 语言把数组的数组当作是多维数组，但不能把几个下标范围如[i][j][k]合并成 Pascal 式的下标表达式风格如[i,j,k]。如果你清楚地明白自己在做什么，也介意产生不合规范的程序，可以把[i][j][k]这样的下标值计算为相应的偏移量，然后只用一个单一的下标[z]来引用数组。当然这不是一种值得推荐的做法。同样糟糕的是，像[i, j, k]这样的下标形式（由逗号分隔）是 C 语言合法的表达形式，只是它并非同时引用这几个下标（它实际上所引用的下标值是 k，也是就逗号表达式的值）。C 语言支持其他语言一般称作“数组的数组”的东西，但却称它为多维数组，这样就模糊了两者的边界，使许多人对两者混淆不清。（见图 9-7）

在 C 语言中，可以象下面这样声明一个 10×20 的多维字符数组：

```
char carrot[10][20];
```

或者声明一种看上去更像“数组的数组”形式：

```
typedef char vegetable[20];
vegetable carrot[10];
```

不论哪种情况，访问单个字符都是通过 carrot[i][j]的形式，
编译器在编译时会把它解析为*(*(carrot + i) + j)的形式。

图 9-7　数组的数组

尽管术语上称作“多维数组”，但 C 语言实际上只支持“数组的数组”。如果在你的思维模式中，把数组看作是一种向量（即某种对象的一维数组，它的元素可以是另一个数组），就能极大简化编程语言中这个相当复杂的领域。

小 启 发

C 语言中的数组就是一维数组

当提到 C 语言中的数组时，就把它看作是一种向量(vector)，也就是某种对象的一维数组，数组的元素可以是另一个数组。

9.6.2　如何分解多维数组

必须仔细注意多维数组是如何分解为几个单独的数组的。如果我们声明如下的多维数组：

```
int apricot[2][3][5];
```

可以按图 9-8 所示的任何一种方法为它在内存中定位。

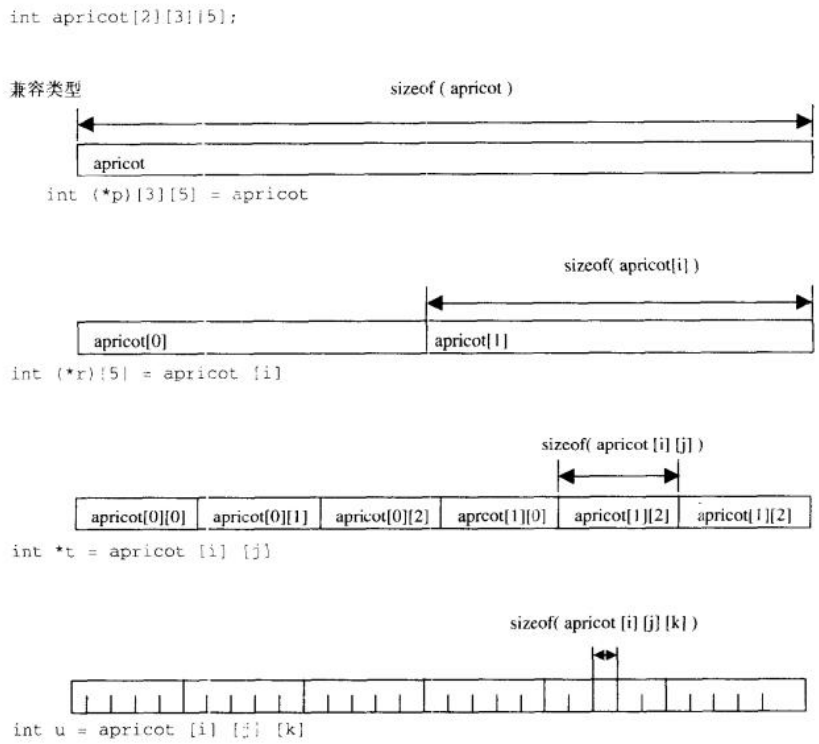


图 9-8　多维数组的存储

正常情况下，赋值发生在两个相同的类型之间，如 int 与 int、double 与 double 等。在图 9-8 中，可以看到在“数组的数组的数组”中的每一个单独的数组都可以看作是一个指针。这是因为在表达式中的数组名被编译器当作“指向数组第一个元素的指针”(第 242 页的规则 1)。换句话说，不能把一个数组赋值给另一个数组，因为数组作为一个整体不能成为赋值的对象。可以把数组名赋值给一个指针，就是因为这个“在表达式中的数组名被编译器当作一个指针”的规则。

指针所指向的数组的维数不同，其区别会很大。使用上面例子中的声明：

```
r++;
t++;
```

将会使 r 和 t 分别指向它们各自的下一个元素（两者所指向的元素本身都是数组)。它们所增

长的步长是很不相同的，因为 r 所指向的数组元素的大小是 t 所指向的数组的元素大小的三倍。

编程挑战

数组万岁！

使用下面的声明：

```
int apricot[2][3][5];

int (*r)[5] = apricot[0];
int *t = apricot[0][0];
```

编写一个程序，打印出 r 和 t 的十六进制初始值（使用 printf 的%x 转换符，打印十六进制值），对这两个指针进行自增(++)操作，并打印它们的新值。

在运行程序之前，预测一下指针每次增长的步长是多少字节，可参考图 9-8。

9.6.3 内存中数组是如何布局的

在 C 语言的多维数组中，最右边的下标是最先变化的，这个约定被称为“行主序”。由于“行/列主序”这个术语只适用于恰好是二维的多维数组，所以更确切的术语是“最右的下标先变化”。绝大部分语言都采用了这个约定，但 Fortran 却是一个主要的例外，它采用了“最左的下标先变化”，也就是“列主序”。在不同的下标变化约定中，多维数组在内存中的布局也不相同。事实上，如果把一个 C 语言的矩阵传递给一个 Fortran 程序，矩阵就会被自动转置——这是一个非常厉害的邪门密技，偶尔真还会用到。

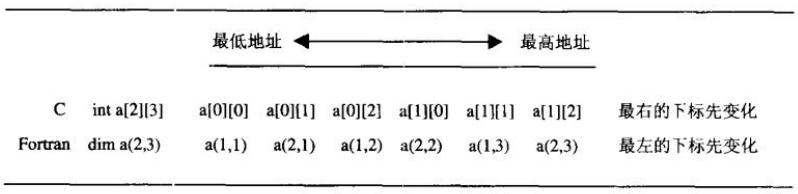


图 9-9 行主序 vs.列主序

C 语言中多维数组最大的用途是存储多个字符串。有人指出“最右边的下标先变化”在这方面具有优势（每个字符串中相邻的字符在内存中也相邻存储）。但在“最左边的下标先变化”的多维数组（如 Fortran）中，情况并不如此。

9.6.4 如何对数组进行初始化

在最简单的情况下，一维数组可以通过把初始值都放在一对花括号内来完成初始化。如果在数组的定义里未标明它的长度，C 语言约定按照初始化值的个数来确定数组的长度。

```
float  banana[5] = { 0.0, 1.0, 2.72, 3.14, 25.625 };
float  honeydew[] = { 0.0, 1.0, 2.72, 3.14, 25.625 };
```

只能够在数组声明时对它进行整体的初始化。之所以存在这个限制，并没得过硬的理由。

多维数组可以通过嵌套的花括号进行初始化：

```
short cantaloupe[2][5] = {
  {10, 12, 3, 4, -5},
  {31, 22, 6, 0, -5},
};
int rhubarb[][3] = { {0, 0, 0}, {1, 1, 1}, };
```

注意，可以在最后一个初始化值的后面加一个逗号，也可以省略它。同时，也可以省略最左边下标的长度(也只能是最左边的下标)，编译器会根据初始化值的个数推断出它的长度。

如果数组的长度比所提供的初始化值的个数要多，剩余的几个元素会自动设置为 0。如果元素的类型是指针，那么它们被初始化为 NULL；如果元素的类型是 float，那么它们被初始化为 0.0。在流行的 IEEE 754 标准浮点数实现中(IBM PC 和 Sun 系统都使用了这个标准)，0.0 和 0 的位模式是完全一样的。

编程挑战

检查位模式

写一个简单的程序，检查在你的系统中，浮点数 0.0 的位模式是否与整型数 0 的位模式相同。

下面是一种初始化二维字符串数组的方法：

```
char vegetables[][9] = { "beet",
                         "barley",
                         "basil",
                         "broccoli",
                         "beans" };
```

一种有用的方法是建立指针数组。字符串常量可以用作数组初始化值，编译器会正确地把各个字符存储于数组中的地址。因此：

```
char *vegetables[] = { "carrot",
                       "celery",
                       "corn",
                       "cilantro",
                       "crispy fried patatoes" };  /* 没问题 */
```

注意它的初始化部分与字符“数组的数组”初始化部分是一样的。只有字符串常量才可以初始化指针数组。指针数组不能由非字符串的类型直接初始化：

```
int *weights[] = {              /* 无法成功编译 */
                  {1, 2, 3, 4, 5},
                  {6, 7},
                  {8, 9, 10}
                 };             /* 无法成功编译 */
```

如果想用这种方法对数组进行初始化，可以创建几个单独的数组，然后用这些数组名来初始化原先的数组。

```
int row_1[] = {1, 2, 3, 4, 5, -1};  /* -1 是行结束标志 */
int row_2[] = {6, 7, -1};
int row_3[] = {8, 9, 10, -1};

int *weight[] = {
                  row_1,
                  row_2,
                  row_3
};
```

10.1 多维数组的内存布局

多维数组在系统编程中并不常用。所以，毫不奇怪的是，C 语言并未像其他语言所要求的那样定义了详细的运行时程序来支持这个特性。对于某些结构如动态数组，程序员必须使用指针显式地分配和操纵内存，而不是由编译器自动完成。另外还有一些结构（作为参数的多维数组），在 C 语言中并没有一般的形式来表达。本章将讲述这些主题。现在，每个人都已经熟悉了多维数组在内存中的布局。如果我们具有以下声明：

```
char pea[4][6];
```

有些人把二维数组看作是排列在一张表格中的一行行的一维数组，如图 10-1 所示。

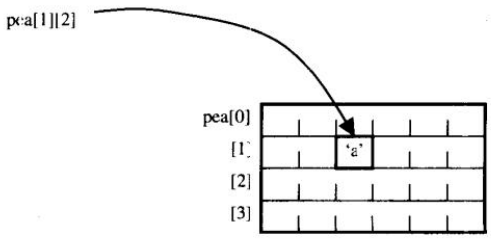


图 10-1 假想中的二维数组内存布局

事实上系统绝不允许程序按照这种方式存储数据。单个元素的存储和引用实际上是以线性形式排列在内存中的，如图 10-2 所示。

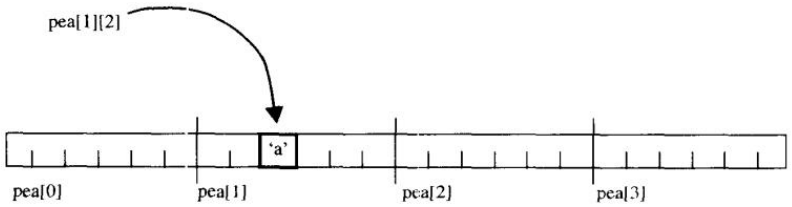


图 10-2　实际上的二维数组内存布局

数组下标的规则告诉我们如何计算左值 pea[i][j]，首先找到 pea[i]的位置，然后根据偏移量[j]取得字符。因此，pea[i][j]将被编译器解析为

```
*(*(pea + i) + j)
```

但是（这正是关键所在！），“pea[i]”的意思将随 pea 定义的不同而变化。我很快将解释这个表达式，但首先让我们看一下 C 语言中最常见最重要的数据结构：指向字符串一维指针数组。

10.2　指针数组就是 Iliffe 向量

可以通过声明一个一维指针数组，其中每个指针指向一个字符串[1]来取得类似二维字符数组的效果。这种形式的声明如下：

```
char *pea[4];
```

软件信条

注意声明的语法

注意 char *turnip[23]把“turnip”声明为一个具有 23 个元素的数组，每个元素的类型是一个指向字符的指针（或者一个字符串——单纯从声明中无法区分两者）。可以假想它两边加上了括号——(char *)tunip[23]。这跟从左至右读时看上去的样子（一个指向“具有 23 个字

符类型元素的数组”的指针）不一样。这是因为下标方括号的优先级比指针的星号高。关于声明语法的分析，第 3 章已经作了详细介绍。

用于实现多维数组的指针数组有多种名字，如“Iliffe 向量”、“display”、或“dope 向量”。display 在英国也用来表示一个指针向量，用于激活一个在词法上封闭的过程的活动记录（作为“一个静态结点后面跟一个链表”的替代方案）。这种形式的指针数组是一种强大的编程技巧，在 C 语言之外取得了广泛的应用。图 20-3 显示了这样的结构。

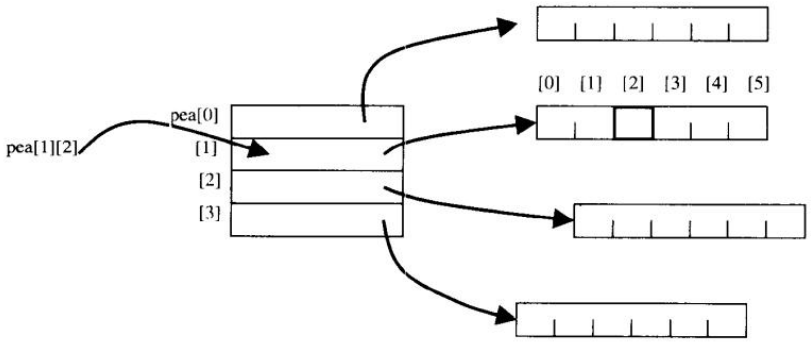


图 10-3　指向字符串的指针数组

这种数组必须用指向为字符串而分配的内存的指针进行初始化，可以在编译时用一个常量初始值，也可以在运行时用下面这样的代码进行初始化：

```
for(j = 0; j <= 4; j++)
   pea[j] = malloc(6);
```

另一种方法是一次性地用 malloc 分配整个 x×y 个数据的数组，

```
malloc(row_size * column_size * sizeof(char) );
```

然后，使用一个循环，用指针指向这块内存的各个区域。整个数组保证能够存储在连续的内存中，即按 C 用于分配静态数组的次序。它减少了调用 malloc 的维护性开销，但缺点是当处理完一个字符串时无法单独将其释放。

软件信条

当你看见 squash[i][]这样的形式时，你不知道它是怎样被声明的！

两个下标的二维数组和一维指针数组所存在的一个问题是：当你看到 squash[i][j]这样的引用形式时，你并不知道 squash 是声明为：

```
int squash[23][12];   /* int 类型的二维数组 */
```

或是

```
int *squash[23];      /* 23 个 int 类型指针的 Iliffe 向量 */
```

或是

```
int **squash;         /* int 类型的指针的指针 */
```

或甚至是

```
int (*squash)[12];   /*  类型为 int 数组（长度为 12）的指针 */
```

这有点类似在函数内部无法分辨传递给函数的实参究竟是一个数组还是一个指针。当然，基于同样的理由：作为左值的数组名被编译器当作是指针。

在上面几种定义中，都可以使用如 squash[i][j]这样的形式，尽管在不同的情况中访问的实际类型并不相同。

与数组的数组一样，一个 Iliffe 向量中的单个字符也是使用两个下标来引用数组中的元素（如 pea[i][j]）。指针下标引用的规则告诉我们 pea[i][j]被编译器解释为：

```
*(*(pea + i) + j)
```

是不是觉得很熟悉？应该是这样。它和一个多维数组引用的分解形式完全一样，在许多 C 语言书中就是这样解释的。然而，这里存在一个很大的问题。尽管这两种下标形式在源代码里看上去是一样，而且被编译器解释为同一种指针表达式，但它们在各自的情况下所引用的实际类型并不相同。表 10-1 和表 10-2 显示了这种区别。

表 10-1　　一个数组的数组 char a[4][6]

char a[4][6]——一个数组的数组

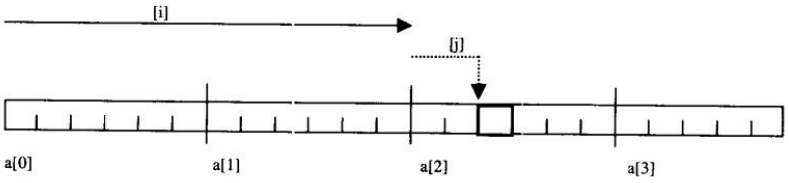
在编译器符号表中，a 的地址为 9980

运行时步骤 1：取 i 的值，把它的长度调整为一行的宽度（这里是 6），然后加到 9980 上

运行时步骤 2：取 j 的值，把它的长度调整为一个元素的宽度（这里是 1），然后加到前面所得出的结果上。

运行时步骤 3：从地址（9980+i*scale-factor1+j*scale-factor2）中取出内容。

a[i][j]



char a[4][6]的定义表示 a 是一个包含 4 个元素的数组，每个元素是一个 char 类型的数组（长度为 6）。所以查找到第 4 个数组的第 i 个元素（前进 i*6 个字节），然后找到数组中的第 j 个元素。

char *p[4]——一个字符串指针数组

在编译器的符号表中，p 的地址为 4624

运行时步骤 1：取 i 的值，乘以指针的宽度（4 个字节），并把结果加到 4624 上。

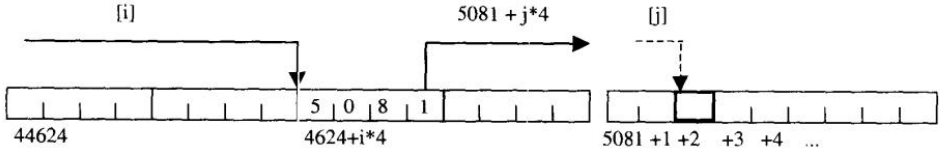
运行时步骤 2：从地址（4624+4*i）取出内容，为“5081”

运行时步骤 3：取 j 的值，乘以元素的宽度（这里是 1 个字节），并把结果加到 5081 上

运行时步骤 4：从地址（5081+j*1）取出内容

p[i][j]

p[i][j]



char *p[4]的定义表示 p 是一个包含 4 个元素的数组，每个元素为一个指向 char 的指针。所以除非指针已经指向字符（或字符数组），否则查找过程无法完成。假定每个指针都给定了一个值，那么查寻过程先找到数组的第 i 个元素（每个元素均为指针），取出指针的值，加上编移量 j，以此为地址，取出地址的内容。

这个过程之所以可行是因为第 9 章的规则 2：一个下标始终相当于指针的偏移量。因此，turnip[i]选择一个元素，也就是一个指针，然后使用下标[j]引用指针，产生*（指针+j），它所指向的是一个单字符。这仅仅是 a[2]和 p[2]的一种扩展，它们的结果都是一个字符，正如我们在前一章所见到的那样。

小 启 发

数组和指针参数是如何被编译器修改的

“数组名被改写成一个指针参数”规则并不是递归定义的。数组的数组会被改写为“数组的指针”，而不是“指针的指针”。

实 参		所匹配的形式参数	
数组的数组	`char c[8][10];`	`char(*)[10];`	数组指针
指针数组	`char *c[15];`	`char **c;`	指针的指针
数组指针（行指针）	`char (*c)[64];`	`char (*c)[64];`	不改变
指针的指针	`char **c;`	`char **c;`	不改变

你之所以能在 main()函数中看到 char **argv 这样的参数，是因为 argv 是个指针数组（即 char *argv[]）。这个表达式被编译器改写为指向数组第一个元素的指针，也就是一个指向指针的指针。如果 argv 参数事实上被声明为一个数组的数组（也就是 char argv[10][15]），它将被编译器改写为 char(*argv)[15](也就是一个字符数组指针)，而不是 char **argv。