数组和指针的区别，C语言数组和指针的区别

在 C 语言中，对数组的引用总是可以写成对指针的引用，而且也确实存在一种指针和数组定义完全相同的上下文环境。因此，给大家带来指针和数组应该是可以互换的错觉，大家也会自然地归纳并假定在所有的情况下数组和指针都是等同的。实际上，并非所有情况下都是如此。

简单地讲，数组就是数组，指针就是指针，它们之间没有任何关系，只是经常穿着相似的衣服来迷惑你罢了。因此，“数组和指针是相同的”这种说法是危险的，是不完全正确的。

回顾前面对于左值和右值的讨论，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（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。

这里需要注意的是，由于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，因此，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。示例代码如下所示：

```
1. char a[6]="hello";
2. ...
3. c=a[i];
```

对于上面的示例代码，在定义数组 a 时，编译器就在某个地方保存了 a 的首元素的首地址，这里假设地址为 10000（即 a 是一个地址，编译器会为数组 a 分配一个空间，但不会为 a 本身分配空间），如图 1 所示。

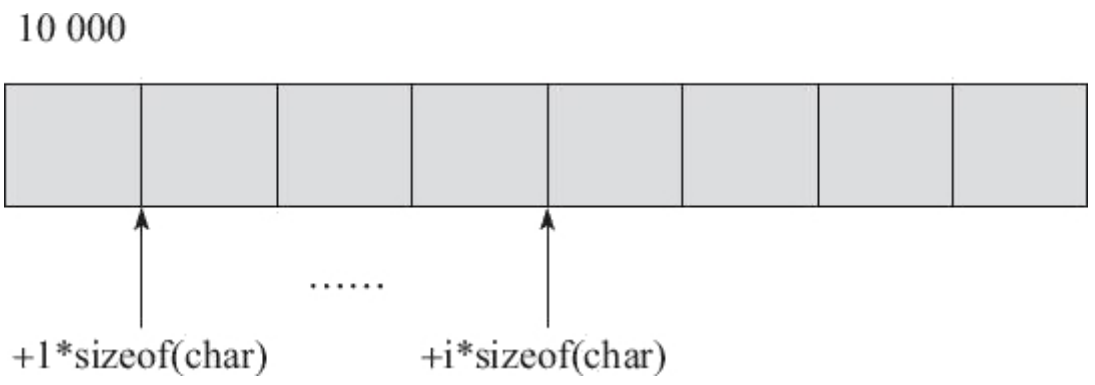

图 1 对数组下标的引用

现在，要取 a[i] 的内容可以分为如下两步进行：

1. 计算 a[i] 的地址：10000+i*sizeof(char)。
2. 取 10000+i*sizeof(char) 地址上的内容。

除了图 1 之外，我们还可以通过下面一段汇编代码来清楚地看见其执行步骤（Microsoft Visual Studio 2010 的 Debug 模式）：

```
1. char a[6]="hello";
2. 00A13718  mov       eax，dword ptr [string "hello"  (0A157A0h)]
3. 00A1371D  mov       dword ptr [ebp-10h]，eax
4. 00A13720  mov       cx，word ptr ds：[0A157A4h]
5. 00A13727  mov       word ptr [ebp-0Ch]，cx
6. char a0=a[0];
7. 00A1372B  mov       al，byte ptr [ebp-10h]
8. 00A1372E  mov       byte ptr [ebp-19h]，al
9. char a1=a[1];
10. 00A13731  mov       al，byte ptr [ebp-0Fh]
11. 00A13734  mov       byte ptr [ebp-25h]，al
12. char a2=a[2];
```

相反，对指针变量而言，编译器要为之分配一个空间。在对一个指针变量进行引用的时候（比如（*p）），编译器首先需要得到 p 的地址，从中取值，然后把得到的值作为地址，再取值。示例代码如下所示：

```
1. char *p="hello";
2. ...
3. c=*p;
```

对比前面的数组，它的执行步骤如图 2 所示。

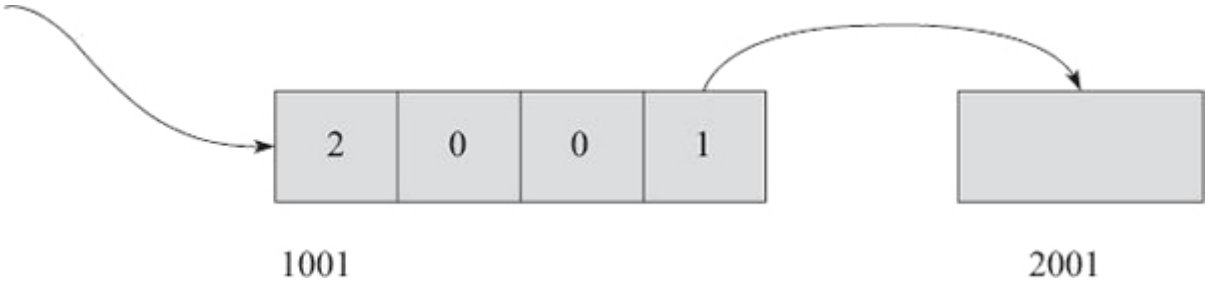

图 2 对指针的引用

1. 这里假设 p 的地址为 1001，取出地址 1001 上的内容 2001。
2. 再取出地址 2001 上的内容。

汇编代码如下（Microsoft Visual Studio 2010 的 Debug模式）：

```
1. 00D313D5  mov       eax，dword ptr [p]
2. 00D313D8  mov       cl，byte ptr [eax]
```

很显然，对数组而言，指针的访问要灵活得多，但需要增加一次额外的提取。

通过上面的阐述，相信你对数组与指针之间的区别已经有了一定的了解，现在继续看下面两种情况。

定义为数组，声明为指针

这里需要特别强调一下变量的声明与定义之间的区别。简单地讲，声明就是告诉编译器存在着这么一个变量，但是编译器并不会为它分配任何内存（即并不实现它）。而定义则是实现这个变量，真正在内存（堆或栈）中为此变量分配空间。定义只能出现一次，而声明可以出现多次。

理解了这些内容，来看下面的示例代码：

```
1. /*文件1：f1.c*/
2. int a[3]={0，1，2};
3. /*文件2：f2.c*/
4. int main(void)
5. {
6.     extern int *a;
7.     printf ("%d\n", a[0]);
```

```
8.     return 0;
9. }
```

在上面的示例代码中，我们在 f1.c 中定义了一个数组 a，在 f2.c 中声明了一个指针变量 a。其中，语句“extern int*a”告诉编译器：a 这个名字已经在别的文件中被定义了，下面的代码使用的名字 a 是别的文件定义的。因此，编译器此时一定不会做什么分配内存的事，因为它就是声明，仅仅表明下面的代码引用了一个符号。

但是，当你声明“extern int*a”时，编译器理所当然地认为 a 是一个指针变量，在 32 位系统下，占 4 字节。这 4 字节保存了一个地址，这个地址上保存的是 int 类型的数据。虽然在 f1.c 中，编译器知道 a 是一个数组，但是在 f2.c 中，编译器并不知道这点。大多数编译器是按文件分别编译的，编译器只按照本文件中声明的类型来处理。所以，虽然 a 在 f1.c 中实际大小为 12 字节，但是在 f2.c 中，编译器认为 a 只占 4 字节。

由此可见，“extern int*a”这种声明方法是完全错误的，它会按照指针的方法来引用数组。同时，在一些编译器中也会给出报错提示，如在 Microsoft Visual Studio 2010 中将给出错误提示：“error C2372：'a'：redefinition；different types of indirection”。

正确的声明方法应该是：

```
1. extern int a[];
2. /*或者*/
3. extern int a[3]
```

示例代码如下所示:

```
1. int main(void)
2. {
3.     extern int a[];
4.     printf ("%d\n", a[0]);
5.     return 0;
6. }
```

这里的 extern int a[] 与 extern int a[3] 是等价的。因为这只是声明，不分配空间，所以编译器无需知道这个数组有多少个元素。这两个声明都告诉编译器：a 是在别的文件中被定义的一个数组，a 同时代表着数组 a 的首元素的首地址，也就是这块内存的起始地址。数组内任何元素的地址都只需要知道这个地址就可以计算出来。

定义为指针，声明为数组

根据上面的分析不难看出，如果在文件 1 中定义为指针，而在文件 2 中声明为数组，也同样会发生错误，示例代码如下所示：

```
1. /*文件3：f3.c*/
2. char *pa = "hello";
3. /*文件4：f4.c*/
4. int main(void)
5. {
6.     extern char pa[];
7.     printf ("%c\n", pa[0]);
8.     return 0;
9. }
```

同样，在 Microsoft Visual Studio 2010 将给出错误提示：“error C2372：'pa'：redefinition；different types of indirection”。因为本来针对数组需要对内存进行直接引用，但是这时编译器所执行的却是对内存进行间接引用。

正确的声明方法应该是：

`extern char *pa`

由此可见，声明与定义应该完全相匹配。同时，这也证明了“数组和指针是相同的”这种说法的错误性。数组和指针在编译器处理的时候是不同的，在运行时的表示形式也不一样。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，你应该根据情况进行选择。表 3 列出了指针与数组的常见区别。

指针	数组
保存数据的地址，任何存入指针变量 p 的数据都会被当作地址来处理	保存数据，数组名 a 代表的是数组首元素的首地址，&a 是整个数组的首地址
间接访问数据，首先取得指针变量 p 的内容，把它当做地址，然后从这个地址提取数据或向这个地址写入数据。 指针可以以指针的形式访问 "*(p+i)" 也可以以下标的形式访问 "p[i]"。但其本质都是先取 p 的内容后加上“i*sizeof(类型)”字节作为数据的真正地址。	直接访问数据，数组名 a 是整个数组的名字，数组内每个元素并没有名字。只能通过"具名+匿名"的方式来访问其某个元素，不能把数组当一个整体进行读写操作。 数组可以以指针的形式访问"*(a+i)"，也可以以下标的形式访问"a[i]"。但其本质都是 a 所代表的数组首元素的首地址加上"i*sizeof(类型)"字节来作为数据的真正地址
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

需要 malloc 和 free 等相关的函数进行内存分配	隐式分配和删除
通常指向匿名数据	自身即为数组名