数组指针和指针数组的区别

数组指针（也称行指针）

定义 int (*p)[n];

()优先级高，首先说明 p 是一个指针，指向一个整型的一维数组，这个一维数组的长度是 n，也可以说是 p 的步长。也就是说执行 p+1 时，p 要跨过 n 个整型数据的长度。

如要将二维数组赋给一指针，应这样赋值：

```
int a[3][4];
int (*p)[4]; //该语句是定义一个数组指针，指向含 4 个元素的一维数组。
 p=a;       //将该二维数组的首地址赋给 p，也就是 a[0]或&a[0][0]
 p++;       //该语句执行过后，也就是 p=p+1;p 跨过行 a[0][]指向了行 a[1][]
```

所以数组指针也称指向一维数组的指针，亦称行指针。

指针数组

定义 int *p[n];

[]优先级高，先与 p 结合成为一个数组，再由 int*说明这是一个整型指针数组，它有 n 个指针类型的数组元素。这里执行 p+1 时，则 p 指向下一个数组元素，这样赋值是错误的：p=a；因为 p 是个不可知的表示，只存在 p[0]、p[1]、p[2]...p[n-1],而且它们分别是指针变量可以用来存放变量地址。但可以这样 *p=a; 这里*p 表示指针数组第一个元素的值，a 的首地址的值。

如要将二维数组赋给一指针数组:

```
int *p[3];
int a[3][4];
p++; //该语句表示 p 数组指向下一个数组元素。注：此数组每一个元素都是一个指针
for(i=0;i<3;i++)
p[i]=a[i]
```

这里 int *p[3] 表示一个一维数组内存放着三个指针变量，分别是 p[0]、p[1]、p[2]

所以要分别赋值。

这样两者的区别就豁然开朗了，数组指针只是一个指针变量，似乎是 C 语言里专门用来指向二维数组的，它占有内存中一个指针的存储空间。指针数组是多个指针变量，以数组形式存在内存当中，占有多个指针的存储空间。

还需要说明的一点就是，同时用来指向二维数组时，其引用和用数组名引用都是一样的。

比如要表示数组中 i 行 j 列一个元素：

(p[i]+j)、(*(p+i)+j)、(*(p+i))[j]、p[i][j]

优先级：()>[]>*

==========

一、指针数组和数组指针的内存布局

初学者总是分不出指针数组与数组指针的区别。其实很好理解：

指针数组：首先它是一个数组，数组的元素都是指针，数组占多少个字节由数组本身的大小决定，每一个元素都是一个指针，在 32 位系统下任何类型的指针永远是占 4 个字节。它是“储存指针的数组”的简称。

数组指针：首先它是一个指针，它指向一个数组。在 32 位系统下任何类型的指针永远是占 4 个字节，至于它指向的数组占多少字节，不知道，具体要看数组大小。它是“指向数组的指针”的简称。

下面到底哪个是数组指针，哪个是指针数组呢：

A)

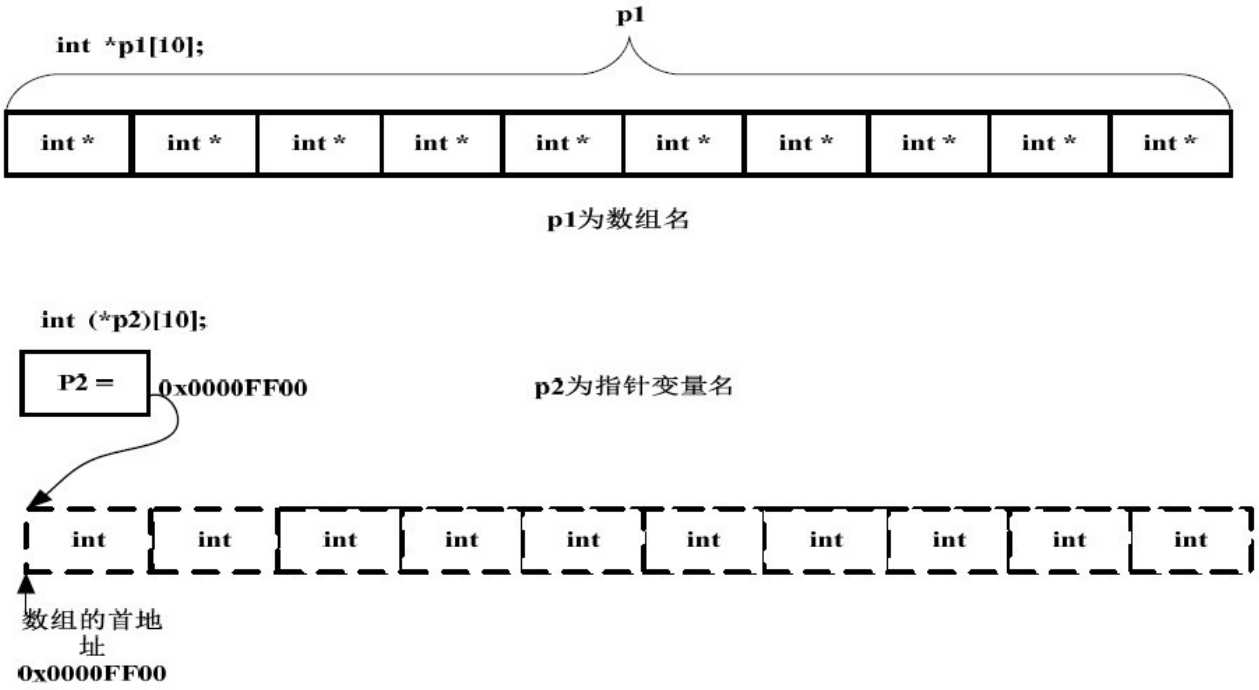
```
int *p1[10];
```

B)

```
int (*p2)[10];
```

每次上课问这个问题，总有弄不清楚的。这里需要明白一个符号之间的优先级问题。

“[]”的优先级比“*”要高。p1 先与“[]”结合，构成一个数组的定义，数组名为 p1，int *修饰的是数组的内容，即数组的每个元素。那现在我们清楚，这是一个数组，其包含 10 个指向 int 类型数据的指针，即指针数组。至于 p2 就更好理解了，在这里“（ ）”的优先级比“[]”高，“*”号和 p2 构成一个指针的定义，指针变量名为 p2，int 修饰的是数组的内容，即数组的每个元素。数组在这里并没有名字，是个匿名数组。那现在我们清楚 p2 是一个指针，它指向一个包含 10 个 int 类型数据的数组，即数组指针。我们可以借助下面的图加深理解：



二、int (*)[10] p2-----也许应该这么定义数组指针

这里有个有意思的话题值得探讨一下：平时我们定义指针不都是在数据类型后面加上指针变量名么？这个指针 p2 的定义怎么不是按照这个语法来定义的呢？也许我们应该这样来定义 p2：

```
  int (*)[10] p2;  //这样写的话，编译器会报错
```
int (*)[10]是指针类型，p2 是指针变量。这样看起来的确不错，不过就是样子有些别扭。其实数组指针的原型确实就是这样子的，只不过为了方便与好看把指针变量 p2 前移了而已。你私下完全可以这么理解这点。虽然编译器不这么想。^_^

假设有如下定义：

int a[1000];

int (*p)[100];

如果想把一维数组当作一个 10 x 100 的二维数组使用，那么可以进行如下强制转化：

p = (int (*)[100]) a;

将数组 a 的首地址转化为数组指针。

三、再论 a 和&a 之间的区别

既然这样，那问题就来了。前面我们讲过 a 和&a 之间的区别，现在再来看看下面的代码：

```
int main()
{
  char a[5]={'A','B','C','D'};
  char (*p3)[5] = &a;
  char (*p4)[5] = a;
  return 0;
}
```

上面对 p3 和 p4 的使用，哪个正确呢？p3+1 的值会是什么？p4+1 的值又会是什么？毫无疑问，p3 和 p4 都是数组指针，指向的是整个数组。&a 是整个数组的首地址，a 是数组首元素的首地址，其值相同但意义不同。在 C 语言里，赋值符号“=”号两边的数据类型必须是相同的，如果不同需要显示或隐式的类型转换。p3 这个定义的“=”号两边的数据类型完全一致，而 p4 这个定义的“=”号两边的数据类型就不一致了。左边的类型是指向整个数组的指针，右边的数据类型是指向单个字符的指针。在 Visual C++6.0 上给出如下警告：

```
  warning C4047: 'initializing' : 'char (*)[5]' differs in levels of indirection from 'char *'。
```

还好，这里虽然给出了警告，但由于&a 和 a 的值一样，而变量作为右值时编译器只是取变量的值，所以运行并没有什么问题。不过我仍然警告你别这么用。

既然现在清楚了 p3 和 p4 都是指向整个数组的，那 p3+1 和 p4+1 的值就很好理解了。

但是如果修改一下代码，把数组大小改小点，会有什么问题？p3+1 和 p4+1 的值又是多少呢？

```
int main()
{
  char a[5]={'A','B','C','D'};
  char (*p3)[3] = &a;
  char (*p4)[3] = a;
  return 0;
}
```

甚至还可以把代码再修改，把数组大小改大点：

```
int main()
{
  char a[5]={'A','B','C','D'};
  char (*p3)[10] = &a;
  char (*p4)[10] = a;
  return 0;
}
```

这个时候又会有什么样的问题？p3+1 和 p4+1 的值又是多少？

上述几个问题，希望读者能仔细考虑考虑，并且上机测试看看结果。

测试结果:

(1).char (*p2)[5]=a;必须使用强制转换,如:char (*p2)[5]=(char (*)[5])a;

(2).把数组大小改变,都会编译不通过,提示:

```
error C2440: 'initializing' : cannot convert from 'char (*)[5]' to 'char (*)[3]'
error C2440: 'initializing' : cannot convert from 'char (*)[5]' to 'char (*)[10]'
```

(3).把以上程序测试代码如下:

```
int main()
{
  char a[5]={'a','b','c','d'};
  char (*p1)[5]= &a;
  char (*p2)[5]=(char (*)[5])a;

  printf("a=%d\n",a);
  printf("a=%c\n",a[0]);
  printf("p1=%c\n",**p1);
  printf("p2=%c\n",**p2);
  printf("p1+1=%c\n",**(p1+1));
  printf("p2+1=%c\n",**(p2+1));

  return 0;
}
```

输出:

```
a=1638208
a=a
p1=a
p2=a
p1+1=?
p2+1=?
Press any key to continue
```

结论:
根据指针类型及所指对象,表示指针大小,每次加 1,表示增加指针类型大小的字节.----后面还会有解释说明.

四、地址的强制转换

先看下面这个例子：

```
struct Test
{
  int Num;
  char *pcName;
  short sDate;
  char cha[2];
  short sBa[4];
}*p;
```

假设 p 的值为 0x100000。如下表表达式的值分别为多少？
  p + 0x1 = 0x___ ?
  (unsigned long)p + 0x1 = 0x___?
  (unsigned int*)p + 0x1 = 0x___?
我相信会有很多人一开始没看明白这个问题是什么意思。其实我们再仔细看看，这个知识点似曾相识。一个指针变量与一个整数相加减，到底该怎么解析呢？

还记得前面我们的表达式“a+1”与“&a+1”之间的区别吗？其实这里也一样。指针变量与一个整数相加减并不是用指针变量里的地址直接加减这个整数。这个整数的单位不是 byte 而是元素的个数。所以：p + 0x1 的值为 0x100000+sizof（Test）*0x1。至于此结构体的大小为 20byte，前面的章节已经详细讲解过。所以 p +0x1 的值为：0x100014。

(unsigned long)p + 0x1 的值呢？这里涉及到强制转换，将指针变量 p 保存的值强制转换成无符号的长整型数。任何数值一旦被强制转换，其类型就改变了。所以这个表达式其实就是一个无符号的长整型数加上另一个整数。所以其值为：0x100001。

(unsigned int*)p + 0x1 的值呢？这里的 p 被强制转换成一个指向无符号整型的指针。所以其值为：0x100000+sizof（unsigned int）*0x1，等于 0x100004。

上面这个问题似乎还没啥技术含量，下面就来个有技术含量的：在 x86 系统下，其值为多少？

```
intmain()
{
  int a[4]={1,2,3,4};
  int *ptr1=(int *)(&a+1);//指向 a 数组后面的内存单元，&a+1 表示向后移 16 个存储单元
  int *ptr2=(int *)((int)a+1);//表示 a 的存储单元的地址增加一个字节
  printf("%x,%x",ptr1[-1],*ptr2);//ptr1[-1]其实指向的是 a 数组的最后一个单元，*ptr1 则表示 a 数组的地址后移一个字节之后的 4 个连续存储单元所存储的值
  return 0;
}
```

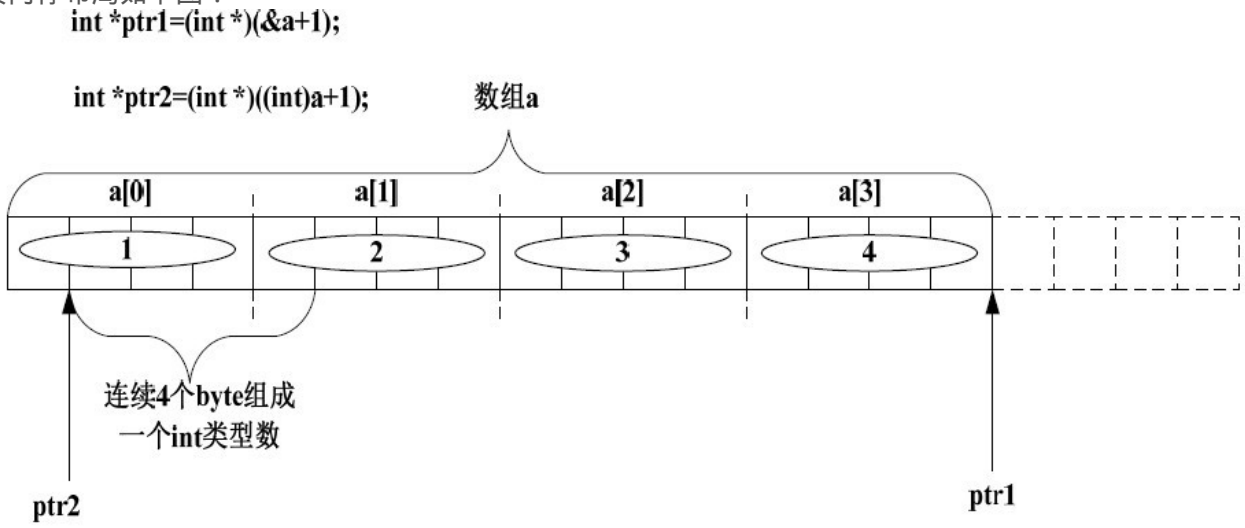
这是我讲课时一个学生问我的题，他在网上看到的，据说难倒了 n 个人。我看题之后告诉他，这些人肯

定不懂汇编，一个懂汇编的人，这种题实在是小 case。下面就来分析分析这个问题：

根据上面的讲解，&a+1 与 a+1 的区别已经清楚。

ptr1：将&a+1 的值强制转换成 int*类型，赋值给 int* 类型的变量 ptr，ptr1 肯定指到数组 a 的下一个 int 类型数据了。ptr1[-1]被解析成*(ptr1-1)，即 ptr1 往后退 4 个 byte。所以其值为 0x4。
ptr2：按照上面的讲解，(int)a+1 的值是元素 a[0]的第二个字节的地址。然后把这个地址强制转换成 int* 类型的值赋给 ptr2，也就是说*ptr2 的值应该为元素 a[0]的第二个字节开始的连续 4 个 byte 的内容。

其内存布局如下图：



好，问题就来了，这连续 4 个 byte 里到底存了什么东西呢？也就是说元素 a[0],a[1]里面的值到底怎么存储的。这就涉及到系统的大小端模式了，如果懂汇编的话，这根本就不是问题。既然不知道当前系统是什么模式，那就得想办法测试。大小端模式与测试的方法在第一章讲解 union 关键字时已经详细讨论过了，请翻到彼处参看，这里就不再详述。我们可以用下面这个函数来测试当前系统的模式。

```
int checkSystem()
{
  union check
  {
     int i;
     char ch;
  } c;
  c.i = 1;
  return (c.ch ==1);//如果当前系统为大端模式这个函数返回 0；如果为小端模式，函数返回 1。
}
```

如果当前系统为大端模式这个函数返回 0；如果为小端模式，函数返回 1。也就是说如果此函数的返回值为 1 的话，*ptr2 的值为 0x2000000。如果此函数的返回值为 0 的话，*ptr2 的值为 0x100。