数组在什么时候会转换为指针

数组名的本意是表示一组数据的集合，它和普通变量一样，都用来指代一块内存，但在使用过程中，数组名有时候会转换为指向数据集合的指针（地址），而不是表示数据集合本身，这在前面的例子中已经被多次证实。

数据集合包含了多份数据，直接使用一个集合没有明确的含义，将数组名转换为指向数组的指针后，可以很容易地访问其中的任何一份数据，使用时的语义更加明确。

C 语言标准规定，当数组名作为数组定义的标识符（也就是定义或声明数组时）、sizeof 或 & 的操作数时，它才表示整个数组本身，在其他的表达式中，数组名会被转换为指向第 0 个元素的指针（地址）。

数组和指针的关系颇像诗和词的关系，它们都是一种文学形式，有不少共同之处，但在实际的表现手法上又各有特色。

再谈数组下标[ ]

C 语言标准还规定，数组下标与指针的偏移量相同。通俗地理解，就是对数组下标的引用总是可以写成“一个指向数组的起始地址的指针加上偏移量”。假设现在有一个数组 a 和指针变量 p，它们的定义形式为：

```
1. int a = {1, 2, 3, 4, 5}, *p, i = 2;
```

读者可以通过以下任何一种方式来访问 a[i]：

p = a; p[i];	p = a; *(p + i);	p = a + i; *p;

对数组的引用 a[i] 在编译时总是被编译器改写成`*(a+i)`的形式，C 语言标准也要求编译器

必须具备这种行为。

取下标操作符`[ ]`是建立在指针的基础上，它的作用是使一个指针和一个整数相加，产生出一个新的指针，然后从这个新指针（新地址）上取得数据；假设指针的类型为`T *`，所产生的结果的类型就是`T`。

取下标操作符的两个操作数是可以交换的，它并不在意操作数的先后顺序，就像在加法中 3+5 和 5+3 并没有什么不一样。以上面的数组 a 为例，如果希望访问第 3 个元素，那么可以写作`a[3]`，也可以写作`3[a]`，这两种形式都是正确的，只不过后面的形式从不曾使用，它除了可以把初学者搞晕之外，实在没有什么实际的意义。

a[3] 等价于 *(a + 3)，3[a] 等价于 *(3 + a)，仅仅是把加法的两个操作数调换了位置。

使用下标时，编译器会自动把下标的步长调整到数组元素的大小。数组 a 中每个元素都是 int 类型，长度为 4 个字节，那么`a[i+1]`和`a[i]`在内存中的距离是 4（而不是 1）。

数组作函数参数

C 语言标准规定，作为“类型的数组”的形参应该调整为“类型的指针”。在函数形参定义这个特殊情况下，编译器必须把数组形式改写成指向数组第 0 个元素的指针形式。编译器只向函数传递数组的地址，而不是整个数组的拷贝。

这种隐式转换意味着下面三种形式的函数定义是完全等价的：

```
1. void func(int *parr){ ..... }
2. void func(int arr[]){ ..... }
3. void func(int arr[5]){ ..... }
```

在函数内部，arr 会被转换成一个指针变量，编译器为 arr 分配 4 个字节的内存，用 sizeof(arr) 求得的是指针变量的长度，而不是数组长度。要想在函数内部获得数组长度必须额外增加一个参数。

参数传递是一次赋值的过程，赋值也是一个表达式，函数调用时不管传递的是数组名还是数组指针，效果都是一样的，相当于给一个指针变量赋值。

把作为形参的数组和指针等同起来是出于效率方面的考虑。数组是若干类型相同的数据的集合，数据的数目没有限制，可能只有几个，也可能成千上万，如果要传递整个数组，无论在时间还是内存空间上的开销都可能非常大。而且绝大部分情况下，我们其实并不需要整个数组的拷贝，我们只想告诉函数在那一时刻对哪个特定的数组感兴趣。

关于数组和指针可交换性的总结

1) 用 a[i] 这样的形式对数组进行访问总是会被编译器改写成（或者说解释为）像 *(a+i) 这样的指针形式。

2) 指针始终是指针，它绝不可以改写成数组。你可以用下标形式访问指针，一般都是指针作为函数参数时，而且你知道实际传递给函数的是一个数组。

3) 在特定的环境中，也就是数组作为函数形参，也只有这种情况，一个数组可以看做是一个指针。作为函数形参的数组始终会被编译器修改成指向数组第一个元素的指针。

3) 当希望向函数传递数组时，可以把函数参数定义为数组形式（可以指定长度也可以不指定长度），也可以定义为指针。不管哪种形式，在函数内部都要作为指针变量对待。