

## 一、指针数组

- 1.概念
- 2.用法
  - (1) 案例一
  - (2) 案例二

## 二、数组指针

- 1.概念
  - (1) 引子
  - (2) 写法
  - (3) 辨析
  - (4) 总结
  - (5) 案例
- 2.&数组名VS数组名

## 三、数组指针的使用

- 1.案例
  - (1) 案例一
    - 方法1
    - 方法2
    - 方法3
  - (2) 案例二
    - 方法1
    - 方法2
    - 总结
  - (3) 案例三
    - 方法1
    - 方法2
    - 方法3
- 2.回顾总结

# 一、指针数组

在初识指针那一节，已经介绍了指针数组，这里再强调一下。

指针这一章，可以从[初识指针](#)开始看起。

## 1.概念

指针数组是 `数组`，用来存放指针的。

比如：

```
int arr[10]={0};    //整型数组，数组用来存放整型(int)
char ch[5]={0};     //字符数组， 数组用来存放字符(char)
int* parr[4];       //指针数组，数组用来存放整型指针(int*)
char* pch[5];       //指针数组，数组用来存放字符指针(char*)
```

又比如：

```
int* arr1[10]; //整型指针的数组（arr1数组里面有10个元素，每个元素是int*类型）
char* arr2[4]; //一级字符指针的数组（arr2数组里面有4个元素，每个元素是char*类型）
char* *arr3[5]; //二级字符指针数组（arr3数组里面有5个元素，每个元素是char**类型）
```

## 2.用法

### (1) 案例一

指针数组有什么用呢？

既然指针数组是用来存放指针的数组，那么最能想到的用法，就是**将一些变量的地址存放在一个数组里面**。

比如，现在这里有这么多整型变量：

```
int a=10;
int b=20;
int c=30;
int d=40;
```

可以将它们的 **地址**（地址的类型是int\*）存放在**整型指针数组**里面：

```
int* arr[4]={&a,&b,&c,&d}; //该指针数组里面存放的是整型指针(int*)
```

按 **F10** 调试：（打开监视，分别输入&a, &b, &c, &d）

监视 1		
搜索(Ctrl+E) 🔍 ⏪ ⏩ 搜索深度: 3		
名称	值	类型
▲ &a	0x00000011b8ff964 {10}	int *
	10	int
▲ &b	0x00000011b8ff984 {20}	int *
	20	int
▲ &c	0x00000011b8ff9a4 {30}	int *
	30	int
▲ &d	0x00000011b8ff9c4 {40}	int *
	40	int

同样可以输入arr：（可以看到arr数组里面存放的内容及地址）

名称	值	类型
&a	0x00000011b8ff964 {10}	int *
&b	0x00000011b8ff984 {20}	int *
&c	0x00000011b8ff9a4 {30}	int *
&d	0x00000011b8ff9c4 {40}	int *
arr	0x00000011b8ff9e8 {0x00000011b8ff964 {10}, 0x00000011b8f...	int *[4]
arr[0]	0x00000011b8ff964 {10}	int *
arr[1]	0x00000011b8ff984 {20}	int *
arr[2]	0x00000011b8ff9a4 {30}	int *
arr[3]	0x00000011b8ff9c4 {40}	int *

以上我们可以看到，arr里面存放的确实是a,b,c,d的地址。

既然是地址，那么该地址解引用之后就可以得到原来存储的值。

用一个循环来输出：

```
int i=0;
for(i=0;i<4;i++){
    printf("%d ",*(arr[i]));
}
```

如下结果：

test.c
指针进阶 (1)
(全局范围)

```

79
80 int main() {
81     int a = 10;
82     int b = 20;
83     int c = 30;
84     int d = 40;
85     int* arr[4] = { &a,&b,&c,&d };
86     int i = 0;
87     for (i = 0; i < 4; i++) {
88         printf("%d ", *(arr[i]));
89     }
90     return 0;
91 }

```

Microsoft Visual Studio 调试控制台
10 20 30 40
E:\C网代码\指针进阶
要在调试停止时自动关闭
按任意键关闭此窗口...

案例一代码（供大家学习使用）：

```

int main(){
    int a=10;
    int b=20;
    int c=30;
    int d=40;
    int* arr[4]={&a,&b,&c,&d};
    int i=0;
    for(i=0;i<4;i++){
        printf("%d ",*(arr[i]));
    }
    return 0;
}

```

## (2) 案例二

上面的用法是最原始，最初级的用法。

下面再来看一个高级用法。

现在有三个整型数组：

```

int arr1[]={1,2,3,4,5};
int arr2[]={2,3,4,5,6};
int arr3[]={3,4,5,6,7};

```

 我想把arr1，arr2，arr3存起来。

这三个数组名，是**首元素地址**。

arr1就是1的地址，arr2就是2的地址，arr3就是3的地址。（三个数组名分别是三个整型的地址）

既然三个数组名分别是三个**整型**的**地址**，那么想要将这三个**数组名**存放起来，就需要一个**整型指针**数组。

假设这个整型指针数组名为parr，它的类型就是**int\***。

那么，就可以这样来写：

```

int* parr[]={arr1,arr2,arr3};

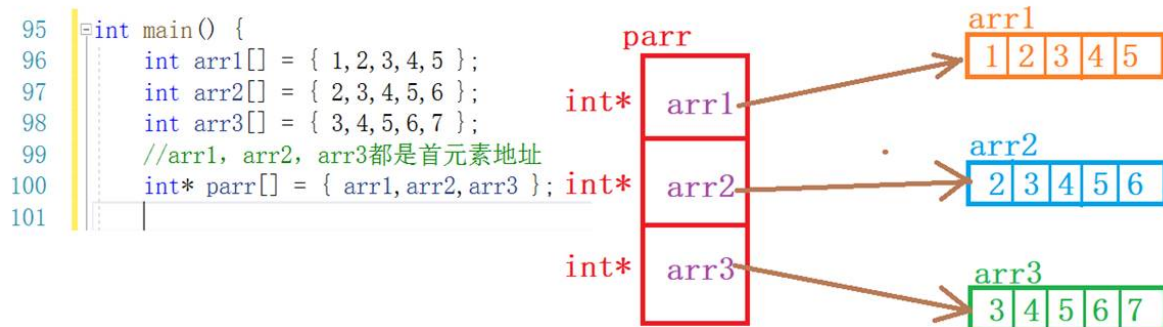
```

 可以探讨一下它们的**内存结构**。

parr数组里面，存了三个数组名（arr1，arr2，arr3），类型都是**int\***。

通过三个数组名，可以分别找到该数组名对应的首元素地址。（比如通过数组名arr1，找到arr1数组首元素1的地址）

如下图：



有了这样一个指针数组，我们就能很好的将三个数组维护起来。

一个一维数组（`parr`），维护了三个一维数组（`arr1`，`arr2`，`arr3`）。

❓如何通过`parr`数组输出所有整型元素？

只要我们能够找到对应每个数组（`arr1`，`arr2`，`arr3`）首元素的地址，就能输出每个元素。

比如，我们能够拿到`arr1`数组首元素1的地址，就能输出12345。

①可以先通过一个for循环，拿到`parr`数组里面的内容，即`arr1`，`arr2`，`arr3`（数组名即首元素地址）。

```
int i=0;
for(i=0;i<3;i++){
    parr[i];
    // parr[0]-->arr1首元素地址
    // parr[1]-->arr2首元素地址
    // parr[2]-->arr3首元素地址
}
```

②既然拿到了对应数组（`arr1`，`arr2`，`arr3`）首元素地址，从该地址往后输出，就可以输出每个数组的元素了。

比如，`parr[0]` 拿到`arr1`首元素地址，那么 `parr[0]+0` 就是数组`arr1`第一个元素1的地址，解引用，就能输出这个数1。

`parr[0]+1` 就是第二个元素2的地址，解引用，就能输出这个数2。

那么，利用循环，就可以挨个输出所有元素了。

```
int j=0;
for(j=0;j<5;j++){
    *(parr[i]+j);
    /*(parr[0]+0) --> 1
    /*(parr[0]+1) --> 2
}
```

③将上面两个循环整合起来，就是：

```
int i=0;
for(i=0;i<3;i++){
    int j=0;
    for(j=0;j<5;j++){
        printf("%d ",*(parr[i]+j));
    }
    printf("\n");    //换行
}
```

输出看一下：

```
94
95 int main() {
96     int arr1[] = { 1, 2, 3, 4, 5 };
97     int arr2[] = { 2, 3, 4, 5, 6 };
98     int arr3[] = { 3, 4, 5, 6, 7 };
99     //arr1, arr2, arr3都是首元素地址
100    int* parr[] = { arr1, arr2, arr3 };
101    int i = 0;
102    for (i = 0; i < 3; i++) {
103        int j = 0;
104        for (j = 0; j < 5; j++) {
105            printf("%d ", * (parr[i] + j));
106        }
107        printf("\n");
108    }
109    return 0;
110 }
```

Microsoft Visual

```
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
```

E:\C网课代码\...  
要在调试停止时  
按任意键关闭以

总结：通过 `parr[i]` 找到存储的每个数组首元素地址，通过 `parr[i]+j` 找到每个元素的地  
址，解引用找到每个元素。

案例二所有代码：（供大家学习使用）

```
int main(){
    int arr1[]={1,2,3,4,5};
    int arr2[]={2,3,4,5,6};
    int arr3[]={3,4,5,6,7};
    int* parr[]={arr1,arr2,arr3};
    int i=0;
    for(i=0;i<3;i++){
        int j=0;
        for(j=0;j<5;j++){
            printf("%d ",*(parr[i]+j));
        }
    }
}
```

```
printf("\n");
}
return 0;
}
```

## 二、数组指针

### 1.概念

数组指针是 指针，用来存放数组的地址。

#### (1) 引子

我们已经熟悉，

整型指针：`int* pint`，能够指向整型数据的指针。可以存放整型的地址。

浮点型指针：`float* pf`，能够指向浮点型数据的指针。可以存放浮点型的地址。

字符指针：`char* pc`，能够指向字符数据的指针。可以存放字符的地址。

那 数组指针 应该是：**能够指向数组的指针。可以存放数组的地址。**

#### (2) 写法

经过上面的分析，可以得出：

**数组指针--指向数组的指针 -- 用于存放数组的地址**

```
int arr[10] = { 0 };
arr -- 首元素地址
&arr[0] -- 首元素地址
&arr -- 数组的地址
```

`&arr` 是数组的地址。既然是数组的地址，就可以存放 到 数组指针 中。

#### 举例

将下面数组的地址(&arr)存起来。

```
int arr[10]={1,2,3,4,5,6,7,8,9,10};
```

既然要存放 数组的地址，就要有一个 数组指针。

定义一个指针变量p，里面存放数组的地址。

🚗 那该如何来写这个数组指针变量呢？

①这样写行吗：

```
int* p=&arr;
```

这里的p是一个整型指针，整型指针里面存放的是整型元素，不能存放一个数组的地址。这样表示不行！

②这样呢：

```
int* p[10]=&arr;
```

这里的p是一个指针数组，`int*`表示该数组里面存放的是整型指针类型的元素，p为数组名，[10]表示数组里面有10个元素。

为什么它是数组而不是指针？因为“[]”优先级比“\*”高，p就是一个数组，里面存放指针。这样表示也不行！

**我现在想表示的是一个指针!!! 里面存放的是地址。**

③那我们这样写：

```
int(*p)[10]=&arr;
```

既然第二种写法，“[]”的优先级比“\*”高，导致p先与“[]”结合，成为了数组。

那我们可以用小括号，将\*p包裹起来，让p先与\*号结合，使p成为一个指针！

- P前面是\*，说明p是一个指针（定义变量的时候，\*不是解引用操作），指向什么呢？

除去\*p不看，剩下的是“int[10]”数组类型。

- [10]说明p指向的是一个数组，数组里面有10个元素。
- int说明数组里面存放的是整型元素。

所以p是一个指针，指向的是一个数组，数组里面有10个元素，每个元素是int类型。

④综上所述，下面的p变量就有能力存放arr数组的地址。

```
int arr[10]={1,2,3,4,5,6,7,8,9,10}; //arr数组里面有10个元素，每个元素是int类型
int(*p)[10]=&arr;
//P为数组指针--> 是一个指针，用来存放数组的地址。指向的是一个数组，数组里面有10个元素，每个元素是int类型
```

### (3) 辨析

我们再来辨析一下：

①案例一



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

p1首先和 [] 结合，是一个**数组**。数组里面10个元素。每个元素是“int\*”类型，p1是一个存放指针的数组。

## ②案例二

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

p2首先和 \* 结合，是一个**指针变量**。指针指向数组，数组里面有10个元素，每个元素是int类型。

## (4) 总结



(1) int(\*p)[10]

p先和 \* 结合，说明p是一个指针变量，然后指向的是一个大小为10个整型的数组。

所以p是一个指针，指向一个数组，叫数组指针。

(2) 注意

[] 的优先级高于 \*，所以必须加上 () 来保证p先和 \* 结合。

## (5) 案例

再来举两个例子，巩固一下上边学的内容。

### □ 案例一

这里我想将数组arr的地址存进数组指针pa中，这里pa的类型如何表示呢？

```
char* arr[5];
```

大家可以自己写一下试试。

正确写法如下：

```
char* arr[5]; //arr数组里面有5个元素，每个元素是char*类型
char* (*pa)[5] = &arr; //将数组地址(&arr)取出来，放进数组指针pa里面
```

**解释：**

- pa是个指针变量名，pa前面的 \* 表示它是一个指针。
- [5] 表示pa指向的数组是5个元素的。
- char\* 表示pa指向数组的元素类型是char\*。
- arr这个数组有5个元素，每个元素类型是 char\*。pa有能力指向这个数组，5个元素，每个元素类型是 char\*。

## □ 案例二

这里我想将数组arr2的地址存进数组指针pa2中，这里pa2的类型如何表示呢？

```
int arr2[10]={0};
```

大家可以再去试试。

正确写法如下：

```
int arr2[10]={0};
int (*pa2)[10]=&arr2;
```

解释：

- 首先pa2是用来存放arr2的地址，所以pa2是一个指针。指针表示为：`*pa2`。
- arr2是一个数组，里面存放10个元素，每个元素是int类型。所以pa2指针也需要有10个int类型的空间来存放。即：`int [10]`。

以后需要写数组指针的时候，千万别写错了。

## 2.&数组名VS数组名

这个部分之前说[数组](#)的时候讲解过，这里再拿出来讲解一下。

对于下面的数组：

```
int arr[10];
```

❓ arr 和 &arr 分别是啥结果？

我们知道 arr 是数组名，数组名表示**数组首元素的地址**。

那 &arr 到底是啥？这里取出的是**数组的地址**。

不妨先来看一下它们分别输出的地址，来看一段代码：

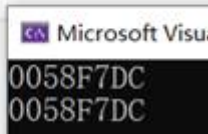
```
int main(){
    int arr[10]={0};
    printf("%p\n",arr);
    printf("%p\n",&arr);
    return 0;
}
```

输出结果：

```

26 int main() {
27     int arr[10] = { 0 };
28     printf("%p\n", arr);
29     printf("%p\n", &arr);
30     return 0;
31 }

```



可见，数组名和&数组名打印的地址是一样的。

难道两个是一样的吗？

我们将它们的地址分别加一，再看一段代码：

```

int main(){
    int arr[10]={0};
    printf("arr=%p\n",arr);
    printf("&arr=%p\n",&arr);
    printf("arr+1=%p\n",arr+1);
    printf("&arr+1=%p\n",&arr+1);
}


```

输出结果：

```

26 int main() {
27     int arr[10] = { 0 };
28     printf("arr=%p\n", arr);
29     printf("&arr=%p\n", &arr);
30
31     printf("arr+1=%p\n", arr+1);
32     printf("&arr+1=%p\n", &arr + 1);
33     return 0;
34 }

```



我们来算一算地址变化：

```

arr=005DF8FC
&arr=005DF8FC
arr+1=005DF900
&arr+1=005DF924

```

① &数组名 --> &数组名+1

跳过一个数组（10个元素，每个元素为整型，一个整型4个字节，一个数组就是10\*4=40个字节）

005DF8FC --> 005DF924

$$\begin{array}{r} 924 \\ - 8FC \\ \hline \end{array}$$

28  $\rightarrow 16+4-C=20-12=8$   
 $\rightarrow 16+1-F=17-15=2$

$$8 \times 16^0 + 2 \times 16^1 = 8 + 32 = 40$$

② 数组名 --> 数组名+1

跳过一个**元素** (一个元素是整型, 一个整型4个字节)

005DF8FC --> 005DF900

$$\begin{array}{r} 900 \\ - 8FC \\ \hline \end{array}$$

04  $\rightarrow 16-C=16-12=4$   
 $\rightarrow 15-F=0$

$$4 \times 16^0 = 4$$

根据上面的代码我们可以发现：

其实 `&arr` 和 `arr`，虽然值是一样的，但是意义是不一样！

### 📦 总结

实际上，`&arr`表示的是“**一整个数组首元素的地址**”，而不是数组单个首元素的地址。

`&arr+1`，跳过整个数组的大小。所以 `&arr+1` 相对于 `&arr` 的差值是40。

所以上面定义数组指针的时候，就可以将数组的地址（`&arr`）交给一个指针。

即：

```
int arr[10]={1,2,3,4,5,6,7,8,9,10};
int (*p)[10]=&arr;
```

## 三、数组指针的使用

那数组指针是怎么使用的呢？

### 1.案例

#### (1) 案例一

看一段代码：

```
int main(){
    int arr[10]={1,2,3,4,5,6,7,8,9,0};
    return 0;
}
```

现在要将数组arr里面的内容全部打印出来。

如何打印呢？

#### 方法1

```
int arr[10]={1,2,3,4,5,6,7,8,9,0};
int (*pa)[10]=&arr;    //将数组arr的地址赋值给数组指针变量pa
```

既然数组指针指向的是数组，那数组指针中存放的应该是数组的地址。

我们知道，`pa` 是数组指针，指向arr数组，里面存的是数组arr的地址。

既然拥有了数组arr的地址，解引用就可以拿到这个数组。（就相当于拿到了一个数组的数组名）

即：

```
*pa //数组的地址解引用，就拿到了这个数组
```

咱们现在拿到了数组名 `*pa`，要去找数组内每个元素，就可以按照正常数组元素的访问来拿到每个元素。即：

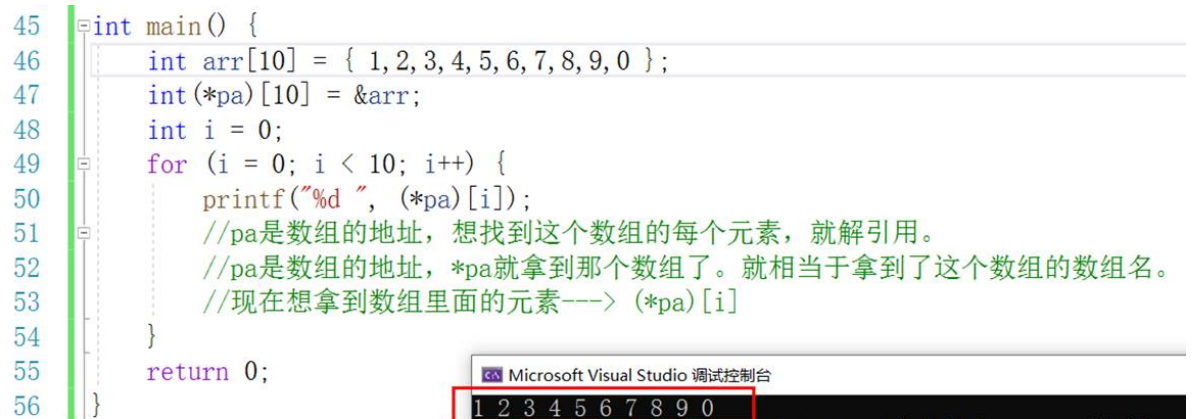
```
(*pa)[0]; //数组的第一个元素
```

再利用for循环，就可以输出数组的每个元素。

整体代码如下：

```
int main(){
    int arr[10]={1,2,3,4,5,6,7,8,9,0};
    int (*pa)[10]=&arr; //将数组arr的地址赋值给数组指针变量pa
    int i=0;
    for(i=0;i<10;i++){
        printf("%d ",(*pa)[i]);
    }
    return 0;
}
```

输出看一下：



```
45 int main() {
46     int arr[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
47     int(*pa)[10] = &arr;
48     int i = 0;
49     for (i = 0; i < 10; i++) {
50         printf("%d ", (*pa)[i]);
51         //pa是数组的地址，想找到这个数组的每个元素，就解引用。
52         //pa是数组的地址，*pa就拿到那个数组了。就相当于拿到了这个数组的数组名。
53         //现在想拿到数组里面的元素---> (*pa)[i]
54     }
55     return 0;
56 }
```

Microsoft Visual Studio 调试控制台

```
1 2 3 4 5 6 7 8 9 0
E:\C网代码\指针进阶(2)\Debug\指针进阶二.exe (进程 14
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->
```

## 方法2

还有一种输出方法。

上面说了，`*pa` 就等于找到了这个数组 (`*pa==arr`)。

`arr`是数组名，数组名是首元素地址。

`arr+i` 就是从`arr`数组首元素的位置向后移动 `i`，指向下标为`i`的元素。

再次解引用，就是第`i`个元素了。即：

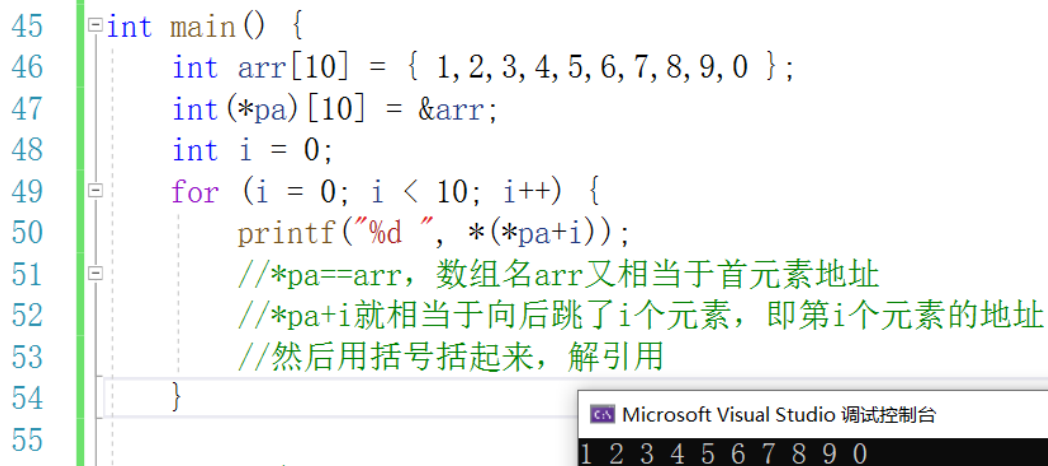
```
*(arr+i)
```

循环，即可输出所有元素。

整体代码如下：

```
int main(){
    int arr[10]={1,2,3,4,5,6,7,8,9,0};
    int (*pa)[10]=&arr; //将数组arr的地址赋值给数组指针变量pa
    int i=0;
    for(i=0;i<10;i++){
        printf("%d ",*(*pa+i));
    }
    return 0;
}
```

输出看一下：



```
45 int main() {
46     int arr[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
47     int(*pa)[10] = &arr;
48     int i = 0;
49     for (i = 0; i < 10; i++) {
50         printf("%d ", *(*pa+i));
51         // *pa==arr, 数组名arr又相当于首元素地址
52         // *pa+i就相当于向后跳了i个元素, 即第i个元素的地址
53         // 然后用括号括起来, 解引用
54     }
55 }
```

Microsoft Visual Studio 调试控制台

1 2 3 4 5 6 7 8 9 0

### 方法3

上面两种方法越写越别扭。

这种多简单：

```
int main(){
    int arr[10] = { 1,2,3,4,5,6,7,8,9,0 };
    int* p = arr; //arr为数组名, 交给一个p指针。这个指针指向第一个元素。
    int i = 0;
    for (i = 0; i < 10; i++) {
        //p表示第一个元素的地址, p+i表示往后的元素的地址
        //找到后面的元素, 就解引用即可: *(p + i)
        printf("%d ", *(p + i));
    }
    return 0;
}
```

输出看一下：

```

45 int main() {
46     int arr[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
47     int* p = arr; // arr为数组名，交给一个p指针。这个指针就指向第一个元素
48     int i = 0;
49     for (i = 0; i < 10; i++) {
50         // p表示第一个元素的地址，p+i表示往后的元素的地址
51         // 找到后面的元素，就解引用即可：*(p + i)
52         printf("%d ", *(p + i));
53     }
54 }

```

Microsoft Visual Studio 调试控制台

1 2 3 4 5 6 7 8 9 0

🌱 可以看到，第三种方法，特别简单。

前面两种方法，将这个题目复杂化了。其实数组指针并不是这样使用的。

## (2) 案例二

❓ 所以数组指针究竟什么时候使用呢？

一般情况下，**数组指针要在二维数组以上使用**，才方便一些。

比如这里有一个二维数组：

```
int arr[3][5] = {{1, 2, 3, 4, 5}, {2, 3, 4, 5, 6}, {3, 4, 5, 6, 7}};
```

想要把数组的元素打印输出。该怎么办呢？

### 方法1

定义一个输出函数print1，将arr数组传递过去：

```
print1(arr, 3, 5); //将数组名，行数，列数传递过去
```

提一嘴，因为遍历数组，需要知道行和列，所以需要将数组的行数与列数传递过去。

在print1函数中，通过两个for循环，来遍历数组。即：

```

void print1(int arr[3][5], int x, int y) {
    int i = 0;
    int j = 0;
    for (i = 0; i < x; i++) {
        for (j = 0; j < y; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n"); //换行
    }
}

int main() {
    int arr[3][5] = { {1, 2, 3, 4, 5}, {2, 3, 4, 5, 6}, {3, 4, 5, 6, 7} }; //定义一个二维数组
    //我想把这个二维数组打印一下
}

```



```
    print1(arr,3,5); //封装一个函数，把数组名和行列传过去
    return 0;
}
```

输出结果：

```
78 void print1(int arr[3][5], int x, int y) {
79     int i = 0;
80     int j = 0;
81     for (i = 0; i < x; i++) {
82         for (j = 0; j < y; j++) {
83             printf("%d ", arr[i][j]);
84         }
85         printf("\n");
86     }
87 }
88 int main() {
89     int arr[3][5] = { {1, 2, 3, 4, 5}, {2, 3, 4, 5, 6}, {3, 4, 5, 6, 7} }; //定义一个二维数组
90     //我想把这个二维数组打印一下
91     print1(arr, 3, 5); //封装一个函数，把数组名和行列传过去
92     return 0;
93 }
```

Microsoft Visual Studio 调试控制台

1	2	3	4	5
2	3	4	5	6
3	4	5	6	7

未找到相关问题

## 方法2

```
int arr[3][5]={1,2,3,4,5},{2,3,4,5,6},{3,4,5,6,7};
```

既然arr是数组名，那么arr就是 首元素地址（除了两种特例）。

? 对于刚才写的二维数组来说，它的首元素是什么呢？

是1吗？不是的，这里的“1”是数组第一行第一个元素。

```
int arr[3][5] = { {1, 2, 3, 4, 5}, {2, 3, 4, 5, 6}, {3, 4, 5, 6, 7} };
```

当我们写出来这样一个二维数组的时候，假想情况是这样的：

1 2 3 4 5

2 3 4 5 6

3 4 5 6 7

现在我们数组名要进行传参。数组名又是首元素地址。数组首元素是什么？是1吗？不是！！！！

当我们要讨论这个数组名是谁的时候，数组名是首元素地址的时候，那首元素是谁，就是谁的地址。

🚗 怎么讨论呢？

- 当我们说，数组名是首元素地址。

如果是二维数组，**首先得把二维数组想象成一维数组。**

把arr想象成一维数组。

把第一行当成一个元素，第二行当成一个元素，第三行当成一个元素。每行只有一个元素。

`arr[3][5]` 其实就有三个元素。

- 又回到刚才说的，数组名就是首元素地址。

其实**第一行就是它的第一个元素，第一行就是数组的第一个地址。**

第一行是一个一维数组，这个数组有5个元素，每个元素是Int类型。即：`int[5]`。

**我们现在这个二维数组有三个元素（三行），每行由五个整型构成。**

---

这样我们就有另外一种写法：

还是写一个输出函数：

```
print2(arr, 3, 5);
```

第一种方法，我们是把 `arr` 当作数组传递上去的，所以当时是用数组（`int arr[3][5]`）接收的。

现在，我们将 `arr` 当作二维数组第一行的地址传递上去。

`Print2`函数传上去的第一个参数是第一行的地址(`arr`)，第一行是一个一维数组，所以传上去的是一维数组的地址。

**一维数组的地址**，应该放到“数组指针”里面去。

这个数组指针指向的不是二维数组，而是一个一维数组（第一行），这个一维数组，有五个元素，每个元素是整型。

那我们可以这样写：

```
int (*p)[5]
```

解释一下这行代码：

“`*p`”表示`p`为指针，“`[5]`”表示指针指向的是一个数组，5个元素，“`int`”表示每个元素是`int`类型。

此时`p`有能力指向二维数组第一行。

这样就将`arr`传递上去的第一行元素的地址接收了。

如下：

```
void print2(int (*p)[5],int x,int y){  
  
}
```

因为这是一个指向数组的指针，指向的是数组。

当我们指向整型的时候，加一，跳过一个整型；当指向一个数组的时候，加一，会跳过一个数组。

`P`指向的数组（第一行）是5个元素，我们加一就会跳到第二行，就会指向第二行。

因为跳过5个整型元素，加一就会指向下一行了。**每次加一，会跳过一行！**

所以怎么写呢？

- `(p+i)`表示跳过`i`行，就指向了下标为 `i` 的这一行（得到了第`i`行的地址），这时候我们解引用，就找到了这一行，即：

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

这时候，我们就拿到了这一行的数组名`arr[i]`，表示第`i`行的地址。

- 要找到具体某个元素，就要在这个地址上再加一个`j`。找到下标为`j`的这个元素的地址。即：

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

- 然后括起来，解引用，就找到了`i`行`j`列的元素了。即：

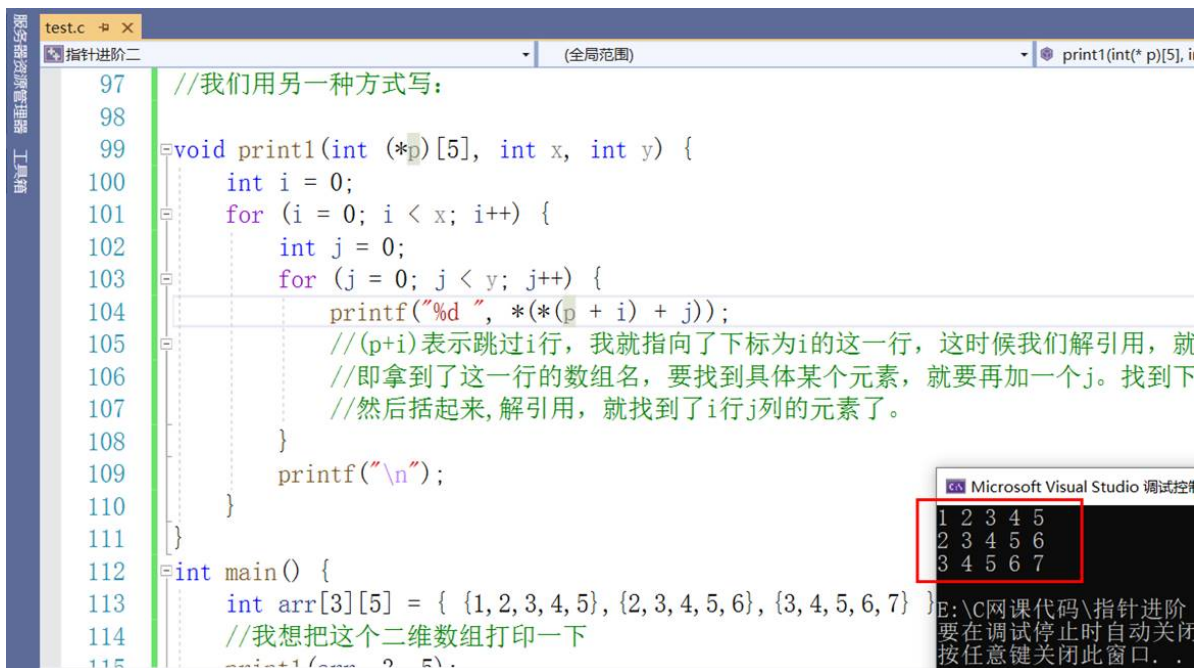
```
*(*(p+i)+j)
```

整合代码：

```
void print2(int (*p)[5], int x, int y) {
    int i = 0;
    for (i = 0; i < x; i++) {
        int j = 0;
        for (j = 0; j < y; j++) {
            printf("%d ", *(*(p + i) + j));
        }
        printf("\n");
    }
}

int main() {
    int arr[3][5] = { {1,2,3,4,5}, {2,3,4,5,6}, {3,4,5,6,7} };
    print2(arr, 3, 5);
    return 0;
}
```

输出看一下：



The screenshot shows the Visual Studio IDE with a C file named 'test.c'. The code defines a function `print1` that takes a 2D array `int (*p)[5]`, and two integers `x` and `y`. It uses nested loops to iterate through the array and print each element. Comments explain that `(p+i)` points to the `i`-th row, and `*(p+i)+j` points to the `j`-th element in that row. The `main` function initializes a 3x5 array `arr` with values `{1,2,3,4,5}`, `{2,3,4,5,6}`, and `{3,4,5,6,7}`, and calls `print1(arr, 3, 5)`. The output window shows the result: a 3x5 grid of numbers.

```
97 //我们用另一种方式写：
98
99 void print1(int (*p)[5], int x, int y) {
100     int i = 0;
101     for (i = 0; i < x; i++) {
102         int j = 0;
103         for (j = 0; j < y; j++) {
104             printf("%d ", *(*(p + i) + j));
105             // (p+i)表示跳过i行，我就指向了下标为i的这一行，这时候我们解引用，就
106             // 即拿到了这一行的数组名，要找到具体某个元素，就要再加一个j。找到下
107             // 然后括起来，解引用，就找到了i行j列的元素了。
108         }
109         printf("\n");
110     }
111 }
112 int main() {
113     int arr[3][5] = { {1,2,3,4,5}, {2,3,4,5,6}, {3,4,5,6,7} };
114     //我想把这个二维数组打印一下
115     print1(arr, 3, 5);
}
```

Microsoft Visual Studio 调试控制台

```
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
```

E:\C网课题代码\指针进阶  
要在调试停止时自动关闭  
按任意键关闭此窗口...

## 总结

第一种写法，参数是数组的形式。

第二种写法，参数是指针的形式。

接下来，咱们再来解释一遍这个代码：

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

① `p+i`：P表示指向了第一行（二维数组第一行，即第一个一维数组）的数组指针，加i表示跳过i行，即指向第i个一维数组（第i行）的数组指针。

② `*(p+i)`：解引用之后，就是第i个一维数组（第i行）的地址，也就是每一个一维数组的数组名。

③ `*(p+i)+j`：然后我们要拿到每一个一维数组（每一行）的里面的元素，先将刚才的一维数组的地址往后挪，加j，就是第i个一维数组的第j个元素的地址。

④ `*(*(p+i)+j)`：然后解引用，就是第i个一维数组（第i行）的第j个元素。

还可以这样写：`(*(p+i))[j]`：第i个一维数组第j个元素。

#### 注意

1. `p+i` 就是指针在二维数组中行的移动，`*(p+i)` 找到的是第i行这个一维数组的地址。
2. 数组的地址和数组首元素地址的大小是一样的。
3. `*(p+i)+j` 就是在这个一维数组里面移动，最后再取值。
4. 可以这么认为，数组指针中放着的是数组名，解引用后得到数组首地址。

## (3) 案例三

### 方法1

我们再来看一个例子：

```
int main() {
    int arr[10] = { 1,2,3,4,5,6,7,8,9,10 };
    int i = 0;
    int* p = arr; //我们把数组名交给了一个整型指针，数组名是首元素地址，首元素是整型。
    //arr就是整型数组的地址，放到整型指针中。
    for (i = 0; i < 10; i++) {
        printf("%d ", *(p + i)); //p为首元素地址，(p+i)是下标为i的元素的地址，解引用就找到这个元素了
    }
    return 0;
}
```

输出结果：

```
122 int main() {
123     int arr[10] = { 1,2,3,4,5,6,7,8,9,10 };
124     int i = 0;
125     int* p = arr; //我们把数组名交给了一个整型指针，数组名是首元素地址，首元素是整型。
126     //arr就是整型的地址，放到整型指针。
127     for (i = 0; i < 10; i++) {
128         printf("%d ", *(p + i)); //p为首元素地址，(p+i)为下标为i的元素的地址，解引用就找到这个元素了
129     }
130     return 0;
131 }
```

Microsoft Visual Studio 调试控制台

1 2 3 4 5 6 7 8 9 10

E:\C\网课代码\指针进阶(2)\Debug\指针进阶二.exe (进程 23764) 已退出，代码为 0。  
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动

## 方法2

再来想一下，

我们是把arr赋给了p，就说明arr和p是一回事。

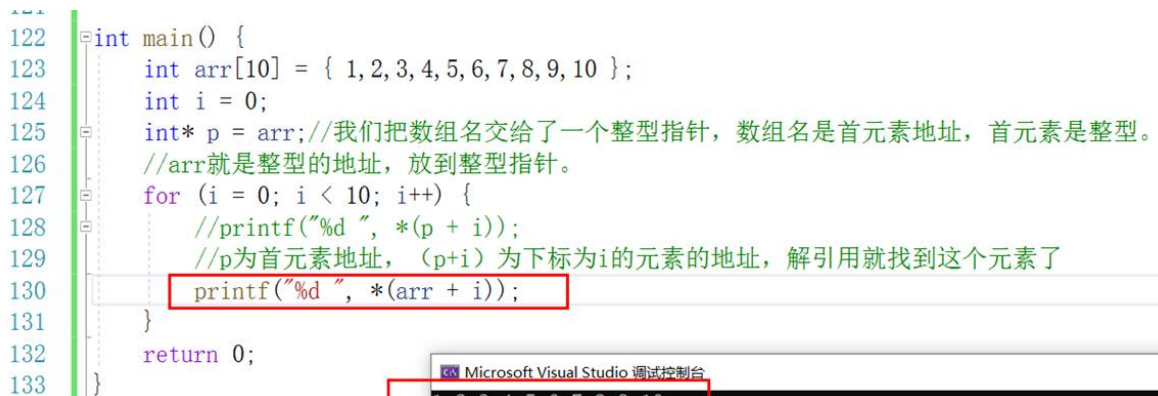
数组名就是首元素地址，首元素地址arr放入p里面，p+i与arr+i是一个道理。

那我们就可以把p+i改成arr+i。

如下：

```
int main() {
    int arr[10] = { 1,2,3,4,5,6,7,8,9,10 };
    int i = 0;
    int* p = arr;
    for (i = 0; i < 10; i++) {
        printf("%d ", *(arr + i));
    }
    return 0;
}
```

输出结果：



The screenshot shows the Visual Studio code editor with the C program from the previous block. The code is annotated with green comments: line 125 says '我们把数组名交给了一个整型指针，数组名是首元素地址，首元素是整型。' and line 126 says '//arr就是整型的地址，放到整型指针。'. Line 129 has a comment '//p为首元素地址，(p+i)为下标为i的元素的地址，解引用就找到这个元素了'. The line 'printf("%d ", \*(arr + i));' on line 130 is highlighted with a red box. Below the code, the 'Microsoft Visual Studio 调试控制台' (Debug Console) is open, showing the output '1 2 3 4 5 6 7 8 9 10', which is also highlighted with a red box. The console title bar indicates the file path 'E:\C网课代码\指针进阶(2)\Debug\指针进阶二.exe (进程 23420)' and its status '已退出' (Exited).

## 方法3

那我们就可以再写一种。

数组名本来可以这样来写：arr[i] --> 找下标为i的元素

**arr[i] == \*(arr+i) == \*(p+i) == p[i]**

p[i]就是访问以p为起始地址，下标为i的元素。

arr[i]也是一样，以arr为起始地址，访问下标为i的元素。



```
test.c - x
指针进阶二 (全局范围)
125 int* p = arr; //我们把数组名交给了一个整型指针，数组名是首元素地址
126 //arr就是整型的地址，放到整型指针。
127 for (i = 0; i < 10; i++) {
128     printf("%d ", *(p + i)); //p为首元素地址，(p+i)为下标为i的元
129 }
130 printf("\n");
131 for (i = 0; i < 10; i++) {
132     printf("%d ", *(arr + i));
133 }
134 printf("\n");
135 for (i = 0; i < 10; i++) {
136     printf("%d ", arr[i]);
137 }
138 printf("\n");
139 for (i = 0; i < 10; i++) {
140     printf("%d ", p[i]);
141 }
142 return 0;
```

Microsoft Visual Studio 调试控制台

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
E:\C网课代码\指针进阶 (2) \Debug
要在调试停止时自动关闭控制台，请
按任意键关闭此窗口...
```

那我们案例二的代码中  $*(p+i)$  就可以写成  $p[i]$ 。

即：

$*(*(p+i)+j) == *(p[i]+j)$

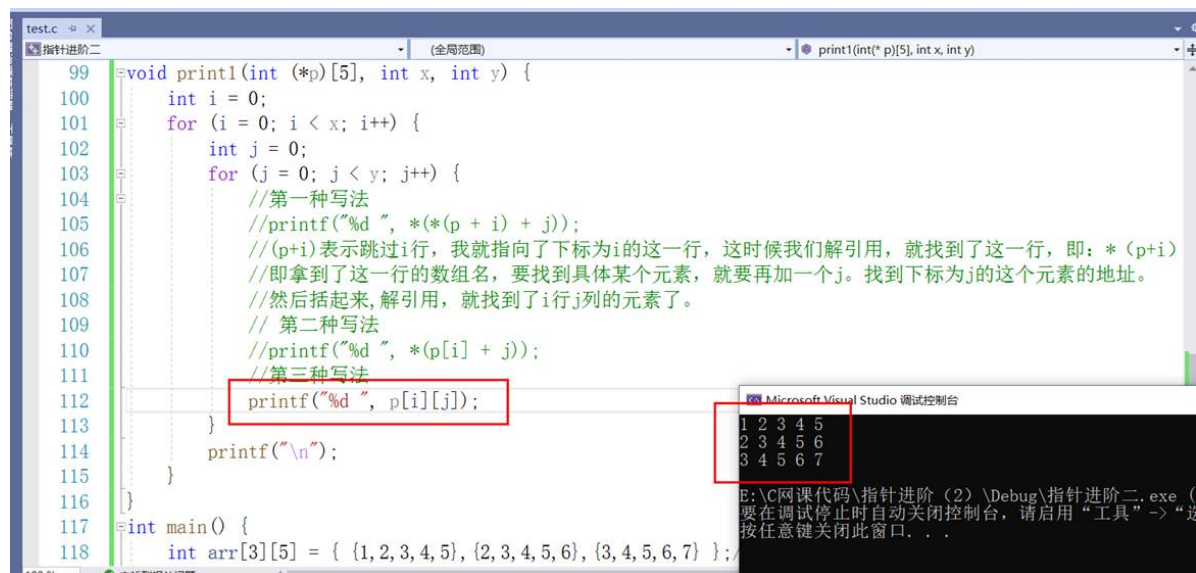
```
test.c - x
指针进阶二 (全局范围)
99 void print1(int (*p)[5], int x, int y) {
100     int i = 0;
101     for (i = 0; i < x; i++) {
102         int j = 0;
103         for (j = 0; j < y; j++) {
104             //printf("%d ", (*(p + i) + j));
105             // (p+i) 表示跳过i行，我就指向了下标为i的这一行，这时候我们解引用，就找到了这一行，即：*(p+i)
106             // 即拿到了这一行的数组名，要找到具体某个元素，就要再加一个j。找到下标为j的这个元素的地址。
107             // 然后括起来，解引用，就找到了i行j列的元素了。
108             printf("%d ", *(p[i] + j));
109         }
110     }
111     printf("\n");
112 }
113 int main() {
114     int arr[3][5] = { {1, 2, 3, 4, 5}, {2, 3, 4, 5, 6}, {3, 4, 5, 6, 7} };
115     //我想把这个二维数组打印一下
116     print1(arr, 3, 5);
117     //依然是这样传参的方式，arr为数组名，表示首元素地址，首元
118     //第一行是一个一维数组，那我们传上去的是一维数组的地址。
```

Microsoft Visual Studio 调试控制台

```
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
E:\C网课代码\指针进阶 (2) \Debug\指针进阶二.exe (
要在调试停止时自动关闭控制台，请启用“工具”->“
按任意键关闭此窗口...
```

那么又有一种写法：

$*(*(p+i)+j) == *(p[i]+j) == (*(p+i))[j] == p[i][j]$



## 2.回顾总结

好了，现在我们来回顾一下刚才学的几个概念：

(1) 例1

```
int arr[5];
```

`arr`是一个整型数组，数组有5个元素，每个元素是整型。

简而言之，就是：`arr`是由5个元素组成的整型**数组**。

(2) 例2

```
int* parr1[10];
```

`parr1`首先和 `[]` 结合，说明它是一个数组，数组10个元素，元素类型是 `int*`。`parr1`是一个**指针数组**。

(3) 例3

```
int (* parr2)[10];
```

`parr2`首先和 `*` 结合，说明它是一个指针，指针指向的是一个数组，数组有10个元素，每个元素的类型是整型。**`parr2`是数组指针**。

(4) 例4

```
int (* parr3[10])[5];
```

`parr3`首先和 `[]` 结合，说明它是一个数组，有10个元素。

对于一个数组来说，明确了它的数组名和它的元素个数，剩下的就是它的元素类型。

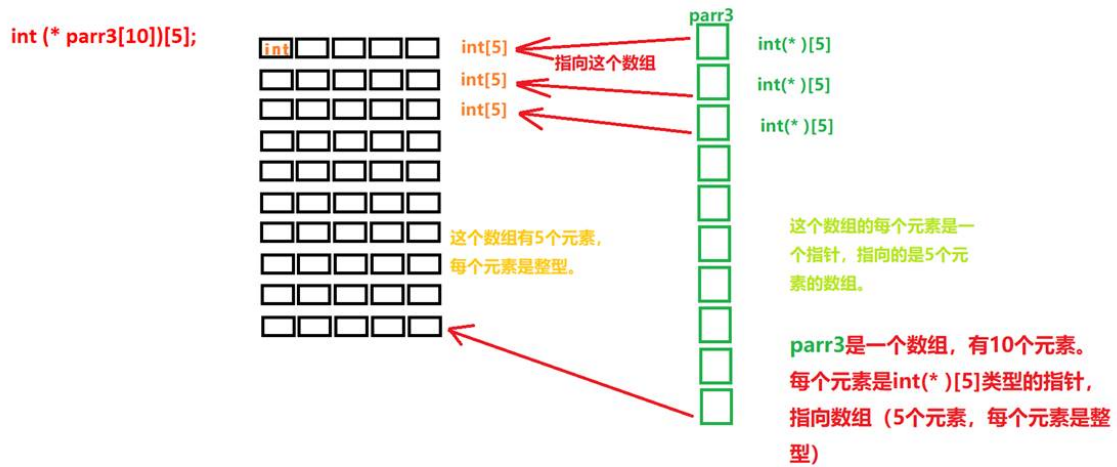
比如：`int arr[10]` -> `arr`为数组名，`[10]`表示它有10个元素，除了数组名和元素个数之外，剩下的`int`就是元素类型。



那么回到这个，剩下的 `int(*)[5]` 就是元素类型。和第三个题很像，是一个指针，指向数组的指针，即数组指针。

则：parr3是一个**数组**，该数组有10个元素，每一个元素是一个数组指针。该数组指针指向的数组有5个元素，每个元素是int类型。

我们来画一下它的大概图解：



---

欢迎关注，一位喜欢慢慢生活的博主。

自述文件



雨翼轻尘

110

3.3w

2.1w

原创内容

作者排名

粉丝数量



CSDN