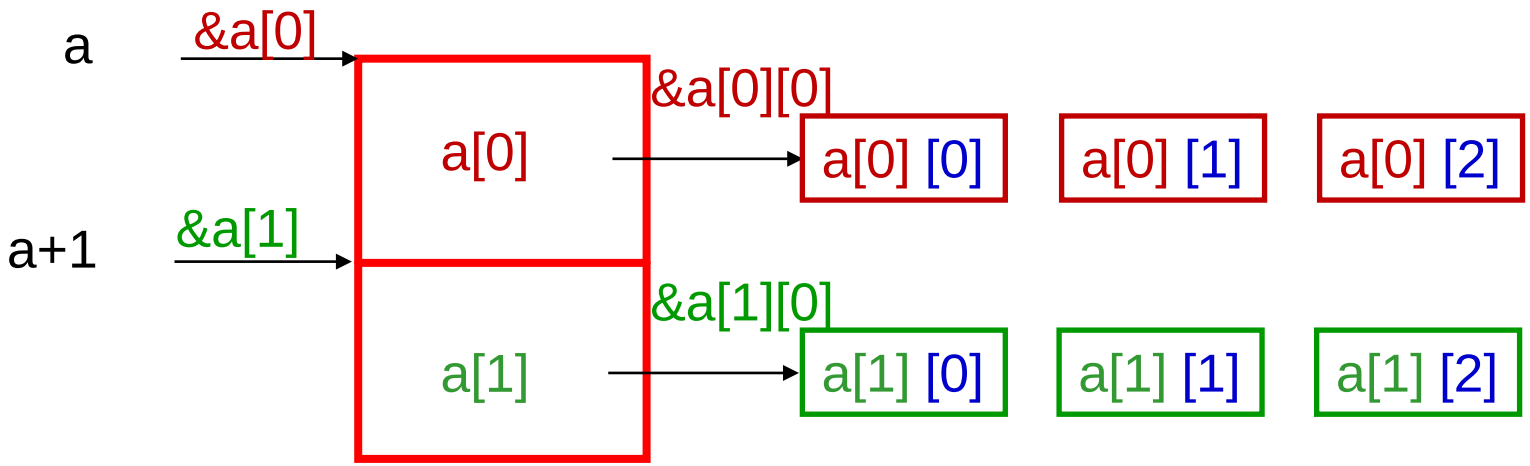
换个角度看二维数组

- 将二维数组a看成一维数组，有2个“int[3]型”元素




```
int a[2][3];
```

a[i][j]
((a+i)+j)

a代表二维数组的首地址，第0行的地址，行地址

a+i代表第i行的地址

但并非增加i个字节！

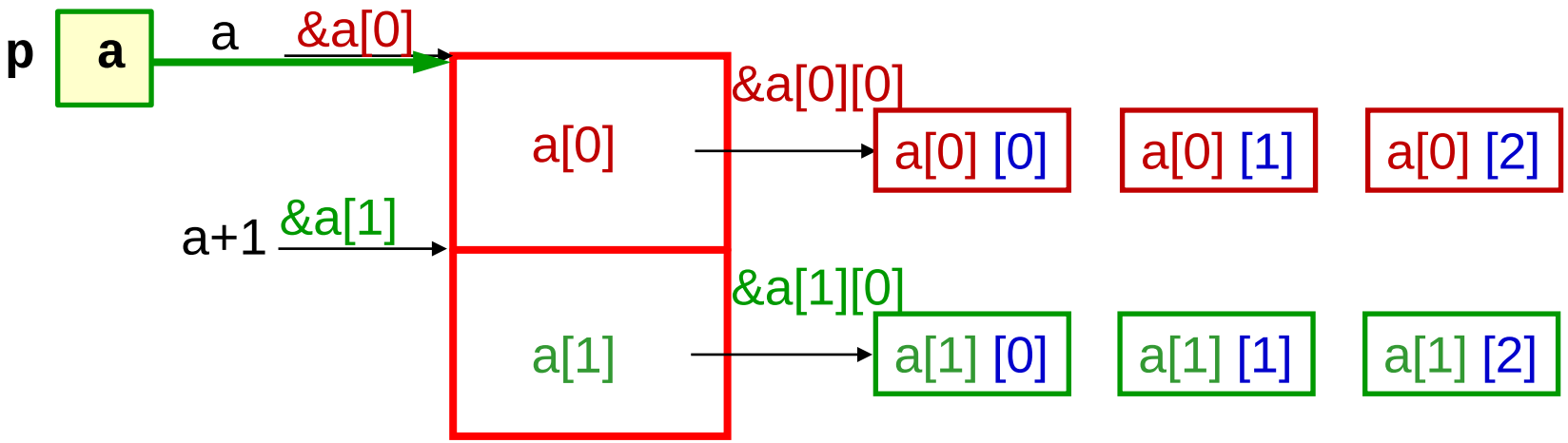
a[i] ↔ *(a+i)
&a[i] ↔ (a+i)

1）a包含2个元素a[0],a[1]

2）a[0],a[1]又分别是一个一维数组，包含3个元素

指向二维数组的行指针

- 将二维数组a看成一维数组，有2个“int[3]型”元素




```
int a[2][3];
```

- 若要让一个指针指向它，则应定义为
  - `int (*p)[3];`//行指针，基类型“int[3]型”
  - `p = a;` //&a[0]指向第0行的“int[3]型”元素
- `int *p[3];`//指针数组

int (*p)[3];
那么p + 1，是跳3个int，不是像``int *p;"那样

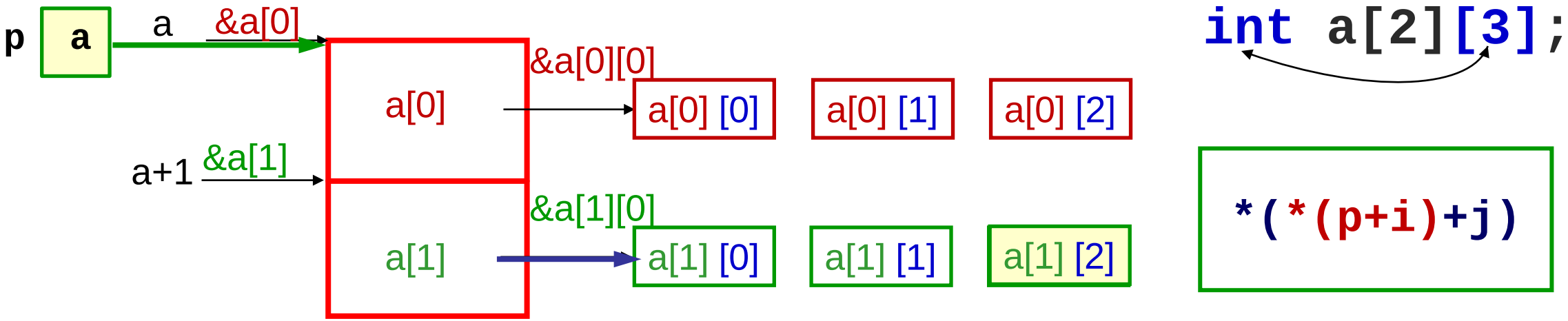
对于``int *p;"，p + 1只跳1个int

前者的基类型是3个int
后者的基类型是1个int

对比：
1. void OutputArray(int (*p)[N], int m, int n)
2. void OutputArray(int p[][N], int m, int n)

指向二维数组的行指针

- 将二维数组a看成一维数组，有2个“int[3]型”元素



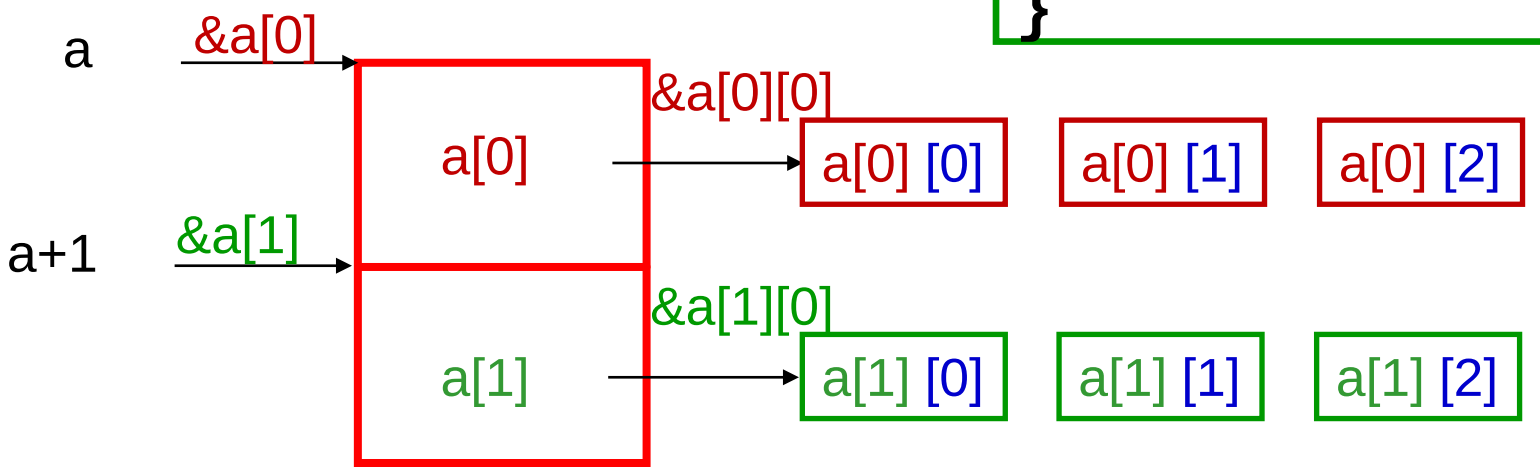
- p+i 指向第i行的“int[3]型”元素，即&a[i]
- *(p+i)，即a[i]，第i行的数组名，即指向第i行第0列的int型元素
- *(p+i)+j 指向第i行第j列的int型元素
- *(*(p+i)+j) 取出第i行第j列的内容（int型元素的值），即a[i][j]

按行指针访问二维数组元素

- 逐行查找→逐列查找

```
int (*p)[3];
p = a;
```

```
for (i=0; i<m; i++) //行下标变化
{
  for (j=0; j<n; j++)//列下标变化
  {
     printf("%d", *(*(p+i)+j));
  }
}
```

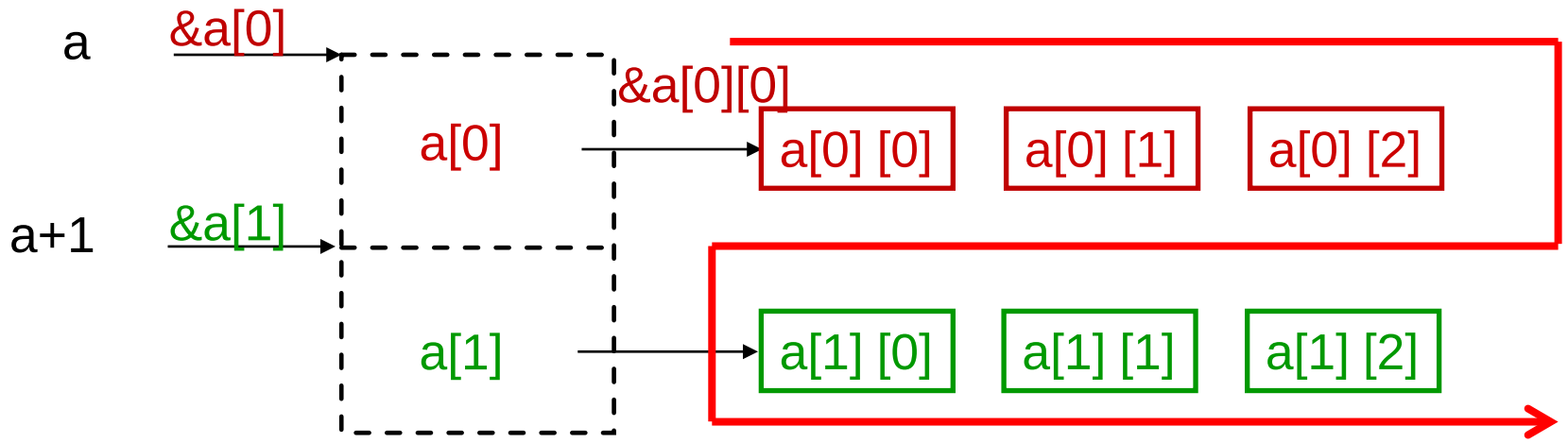


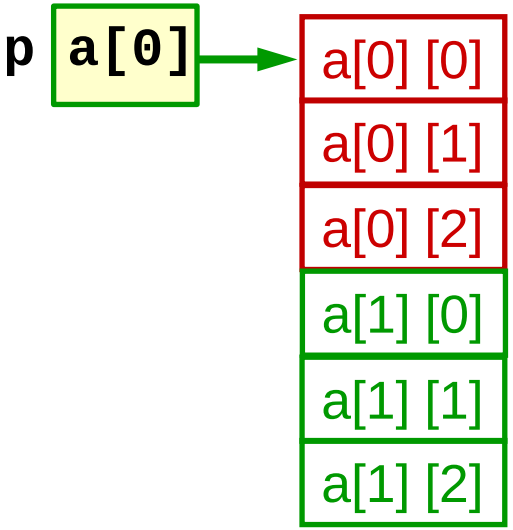

```
int a[2][3];
```

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

再换一个角度看二维数组

- 将二维数组a看成一维数组，有6个int型元素





- 若要让一个指针指向它，则应定义为

```
p = a[0];
p = *a;
p = &a[0][0];
```

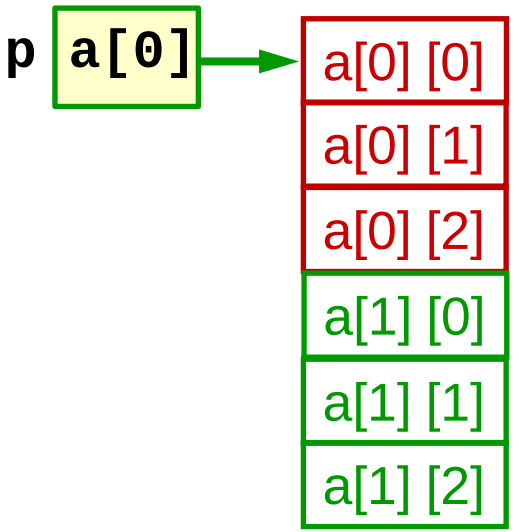
- `int *p;` //列指针，基类型是int型
- `p = a[0];` //*(a+0)+0即&a[0][0]，指向第0行第0列的int型元素

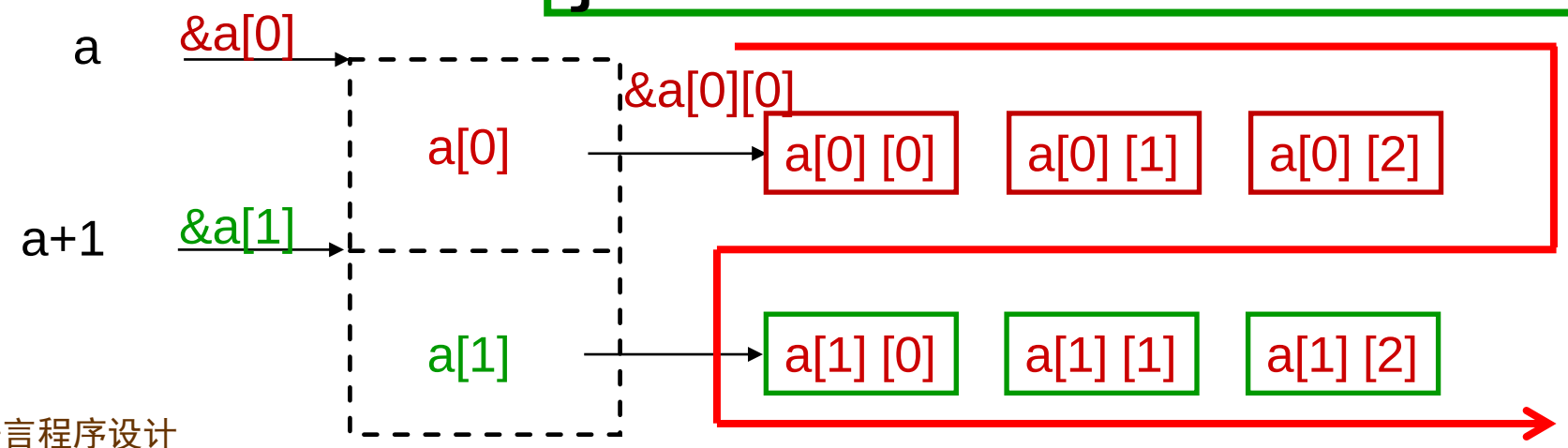
按列指针访问二维数组元素

- 根据相对偏移量逐列查找

```
int *p;
p = &a[0][0];
```

```
for (i=0; i<m; i++)
{
  for (j=0; j<n; j++)
  {
    printf("%d", *(p+i*n+j));
  }
}
```






```
*(p+i*n+j)
p[i*n+j]
```

数组名a
对应的指针类型为：int (*p)[N]

到目前为止的指针类型有：
int *p;
int (*p)[N];

二维数组作函数参数

```
void InputArray(int p[][N], int m, int n)
{
  int i, j;
  for (i=0; i<m; i++)
  {
     for (j=0; j<n; j++)
     {
        scanf("%d", &p[i][j]);
     }
  }
}
```

形参声明为二维数组，列数必须为常量

```
void OutputArray(int p[][N], int m, int n)
{
  int i, j;
  for (i=0; i<m; i++)
  {
     for (j=0; j<n; j++)
     {
        printf("%4d", p[i][j]);
     }
     printf("\n");
  }
}
```

```
InputArray(a, 3, 4);
OutputArray(a, 3, 4);
```

指向二维数组的行指针作函数参数

```
void InputArray(int (*p)[N], int m, int n)
{
  int i, j;
  for (i=0; i<m; i++)
  {
     for (j=0; j<n; j++)
     {
        scanf("%d", *(p+i)+j);
     }
  }
}
```

形参声明为二维数组的行指针，列数必须为常量

```
void OutputArray(int (*p)[N], int m, int n)
{
  int i, j;
  for (i=0; i<m; i++)
  {
     for (j=0; j<n; j++)
     {
        printf("%4d", *(*(p+i)+j));
     }
     printf("\n");
  }
}
```

```
InputArray(a, 3, 4);
OutputArray(a, 3, 4);
```

指向二维数组的列指针作函数参数

```
void InputArray(int *p, int m, int n)
{
  int i, j;
  for (i=0; i<m; i++)
  {
     for (j=0; j<n; j++)
     {
        scanf("%d", &p[i*n+j]);
     }
  }
}
```

形参声明为二维数组的列指针，列数可为变量

行指针：传入的参数为a
列指针：传入的参数为*a

```
InputArray(*a, 3, 4);
OutputArray(*a, 3, 4);
```

```
void OutputArray(int *p, int m, int n)
{
  int i, j;
  for (i=0; i<m; i++)
  {
     for (j=0; j<n; j++)
     {
        printf("%4d", p[i*n+j]);
     }
     printf("\n");
  }
}
```

问题：
p[i*N+j];
v.s.
p[i*n+j];

从for循环的角度去理解存储形式

用const保护你传给函数的数据

```
void OutputArray(const int p[][N], int m, int n)
{
  int i, j;
  for (i=0; i<m; i++)
  {
     for (j=0; j<n; j++)
     {
         printf("%4d", p[i][j]);
     }
     printf("\n");
  }
}
```

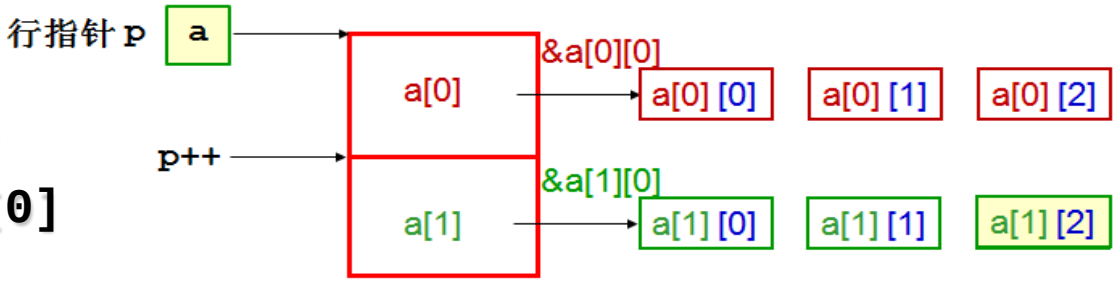
若只向函数传数据，则把形参定义为const，防止它被意外修改

```
void OutputArray(const int *p, int m, int n)
{
  int i, j;
  for (i=0; i<m; i++)
  {
     for (j=0; j<n; j++)
     {
         printf("%4d", p[i*n+j]);
     }
     printf("\n");
  }
}
```

小结

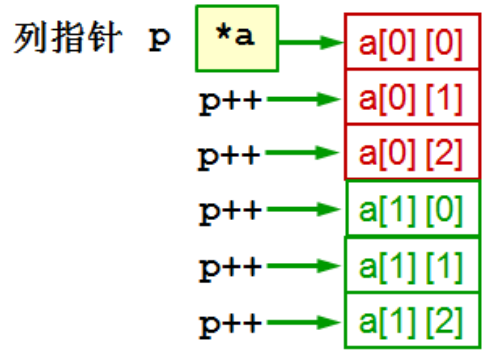
- 指针与二维数组间的关系的关键
  - 理解二维数组的行指针和列指针
    - 二维数组在内存中按行存储，但可以以两种方式看待它
    - 一个x型的指针指向x型的数据
    - a[i] ↔ *(a+i)

```
int (*p)[3];
p = a; //&a[0]
```



a[i][j] ←→ *(*(p+i)+j)

```
int *p;
p = *a; //a[0]
```



a[i][j] ⟷ *(p+i*n+j)