

# Applications of Array Pointers (1A)

---

Copyright (c) 2010 - 2020 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Please send corrections (or suggestions) to [youngwlim@hotmail.com](mailto:youngwlim@hotmail.com).  
This document was produced by using LibreOffice.

---

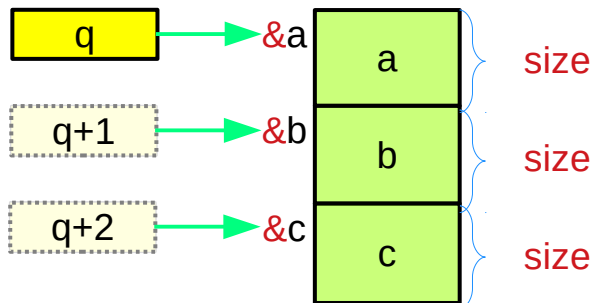
# Virtual Array Pointers in Multi-dimensional Arrays

# Pointers to various data types

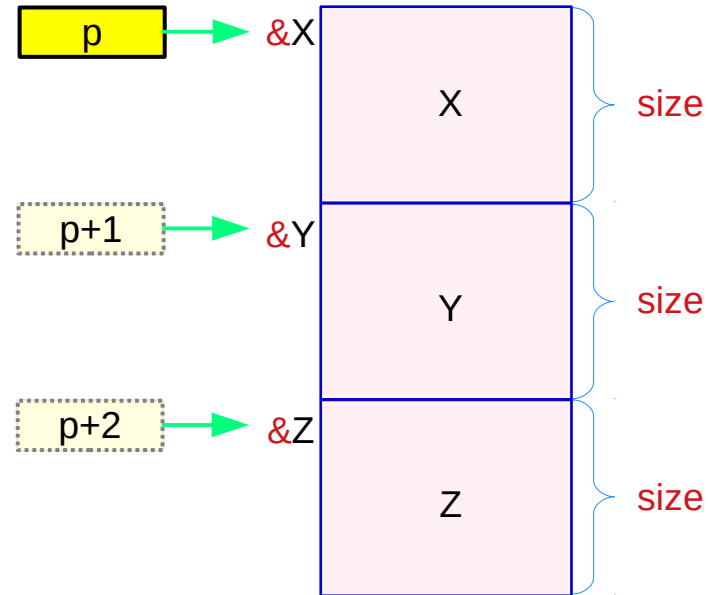
**int \*p;**      **int i, j, k;**



**double \*q;**      **double a, b, c;**



**T \*p;**      **T X, Y, Z;**



pointer

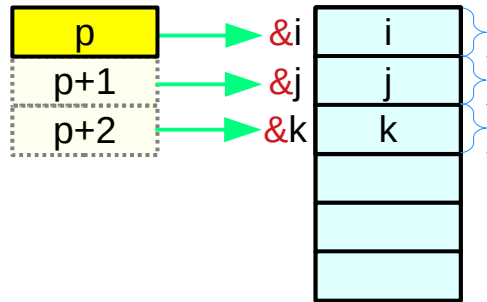
abstract data

# Pointers to primitive data

**int \*p;**

**int i, j, k;**

**sizeof(int) = 4 bytes**



size  
size  
size

$\text{size} = \text{sizeof}(i) = \text{sizeof}(*p)$   
 $\text{size} = \text{sizeof}(j) = \text{sizeof}(*(p+1))$   
 $\text{size} = \text{sizeof}(k) = \text{sizeof}(*(p+2))$

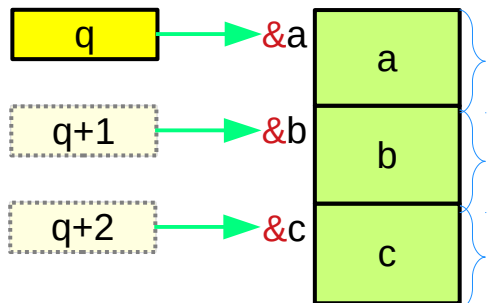
$\neq \text{sizeof}(p)$   
 $\neq \text{sizeof}(p+1)$   
 $\neq \text{sizeof}(p+2)$

pointer size  
4 or 8 bytes

**double \*q;**

**double a, b, c;**

**sizeof(double) = 8 bytes**



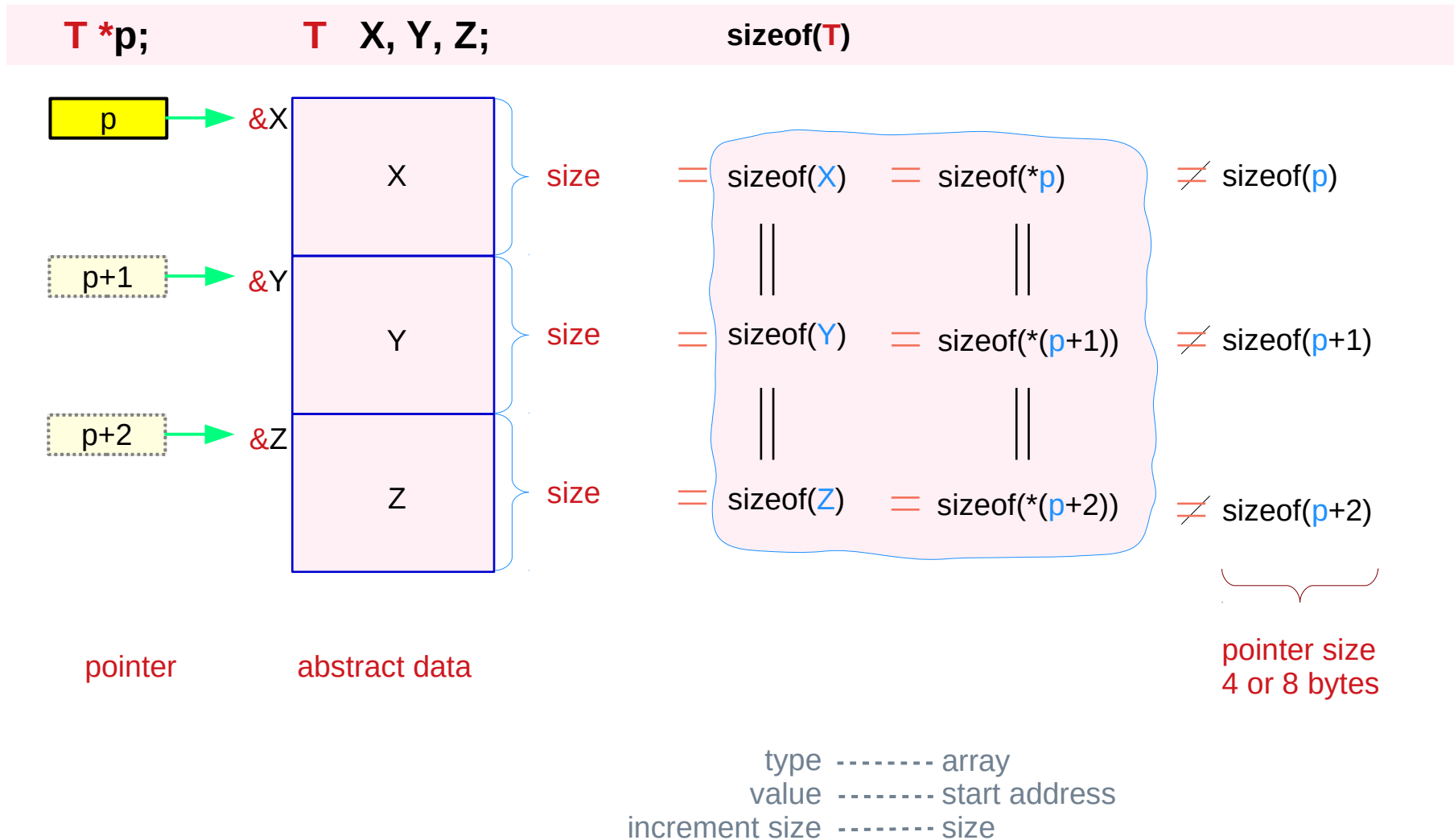
size  
size  
size

$\text{size} = \text{sizeof}(a) = \text{sizeof}(*q)$   
 $\text{size} = \text{sizeof}(b) = \text{sizeof}(*(q+1))$   
 $\text{size} = \text{sizeof}(c) = \text{sizeof}(*(q+2))$

$\neq \text{sizeof}(q)$   
 $\neq \text{sizeof}(q+1)$   
 $\neq \text{sizeof}(q+2)$

pointer size  
4 or 8 bytes

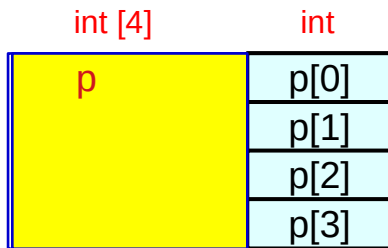
# Pointers to abstract data



# Virtual pointers in an array of integers

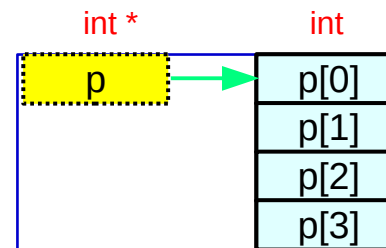
```
int p[3];
```

**p** is an abstract data (array)



- p** is the name of an array
- p** has the size of the whole array
- p** has an array type (abstract data)

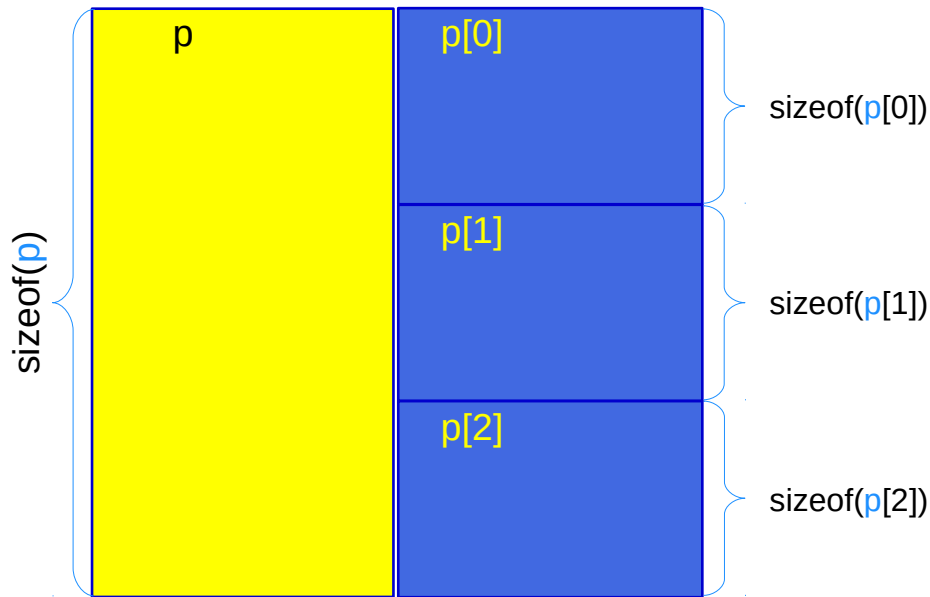
**p** can also be viewed as a pointer



- p** also has pointer characteristics
- p** has the value of the starting address
- p** is a virtual pointer

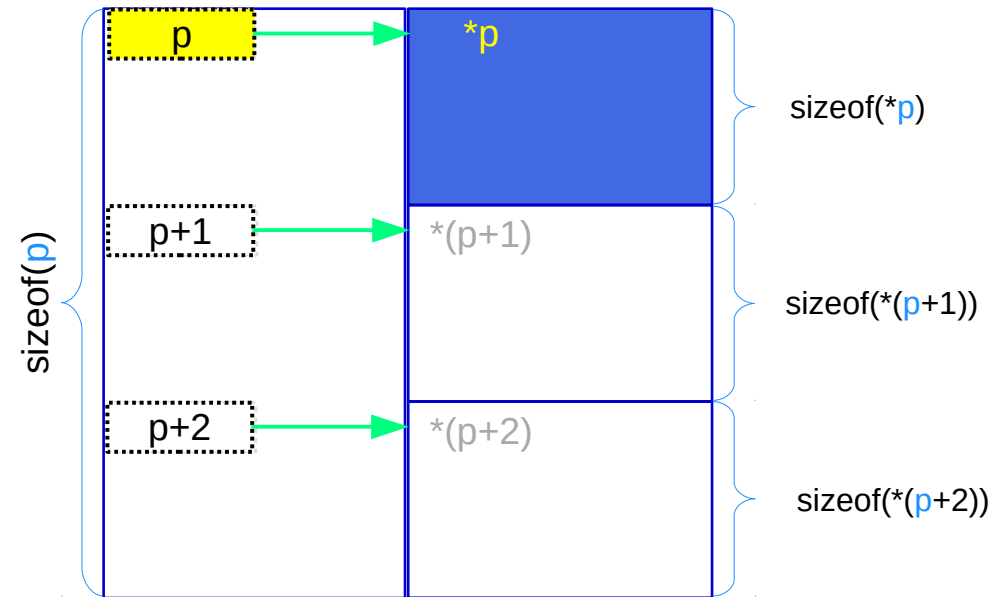
# Virtual pointers in an array of abstract data

## Abstract data array p



- p** has an array type (abstract data element)
- p** is the name of an array
- p** has the size of the whole array

## Virtual pointer p

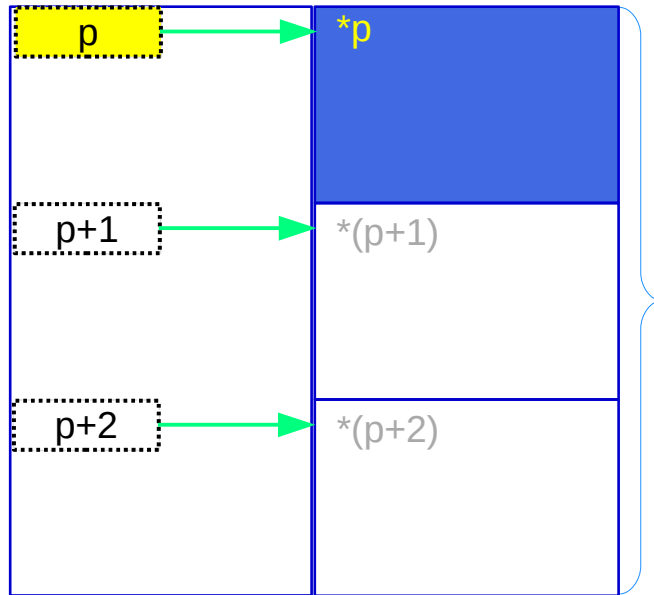


- p** also has a pointer type
- p** has the value of the starting address
- p** is a virtual array pointer

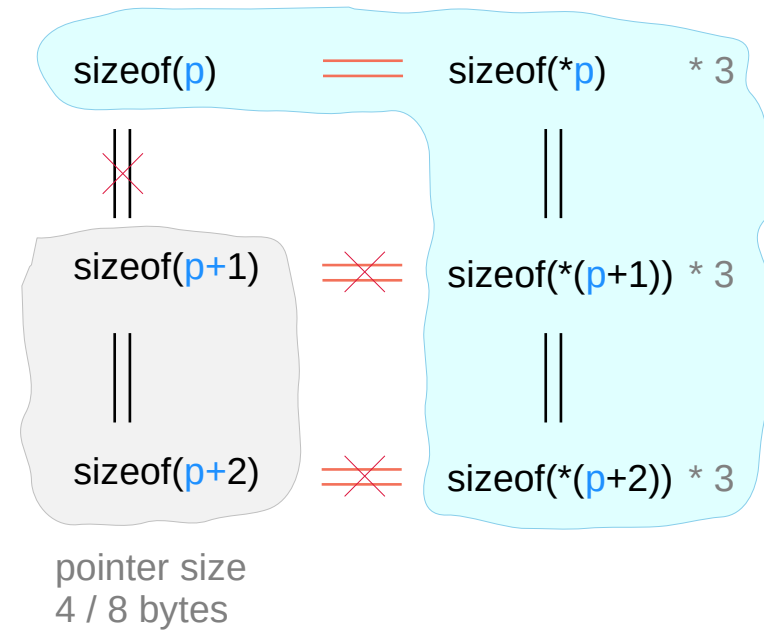


# Virtual pointer to abstract data

virtual pointer p    abstract data \*p

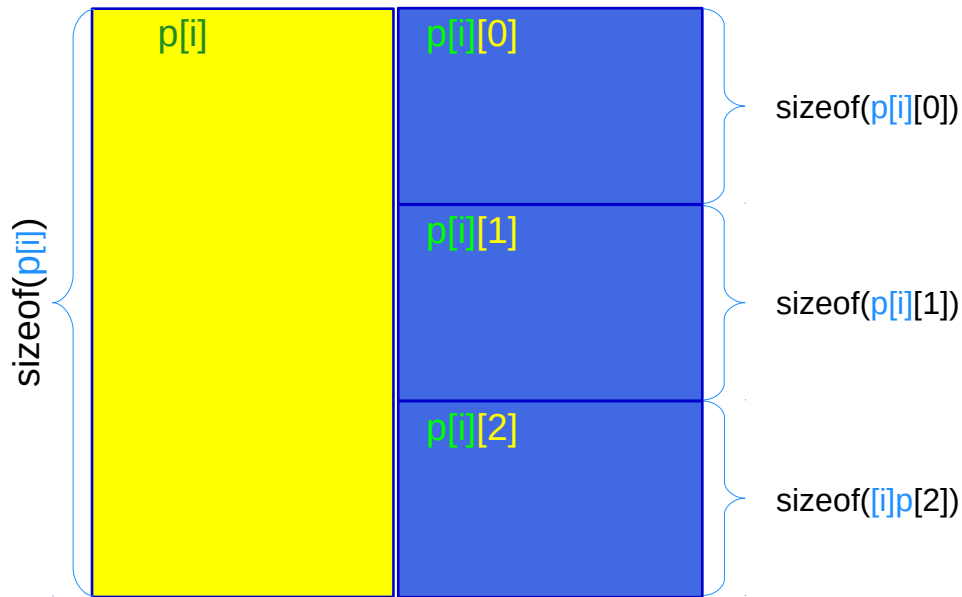


whole array size



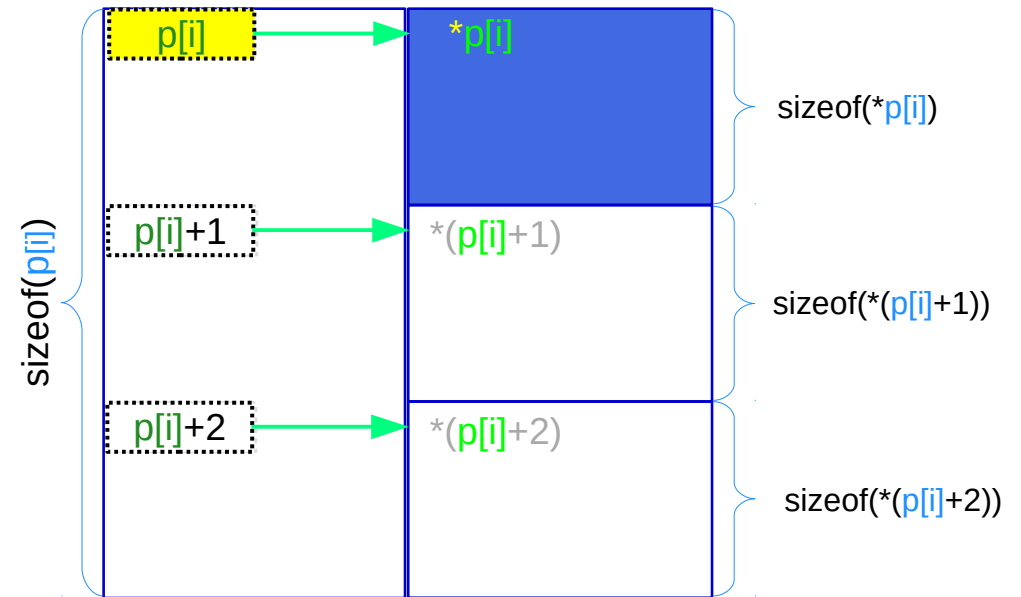
# Virtual array pointers in a multi-dimensional array

## Abstract data (array) $p[i]$



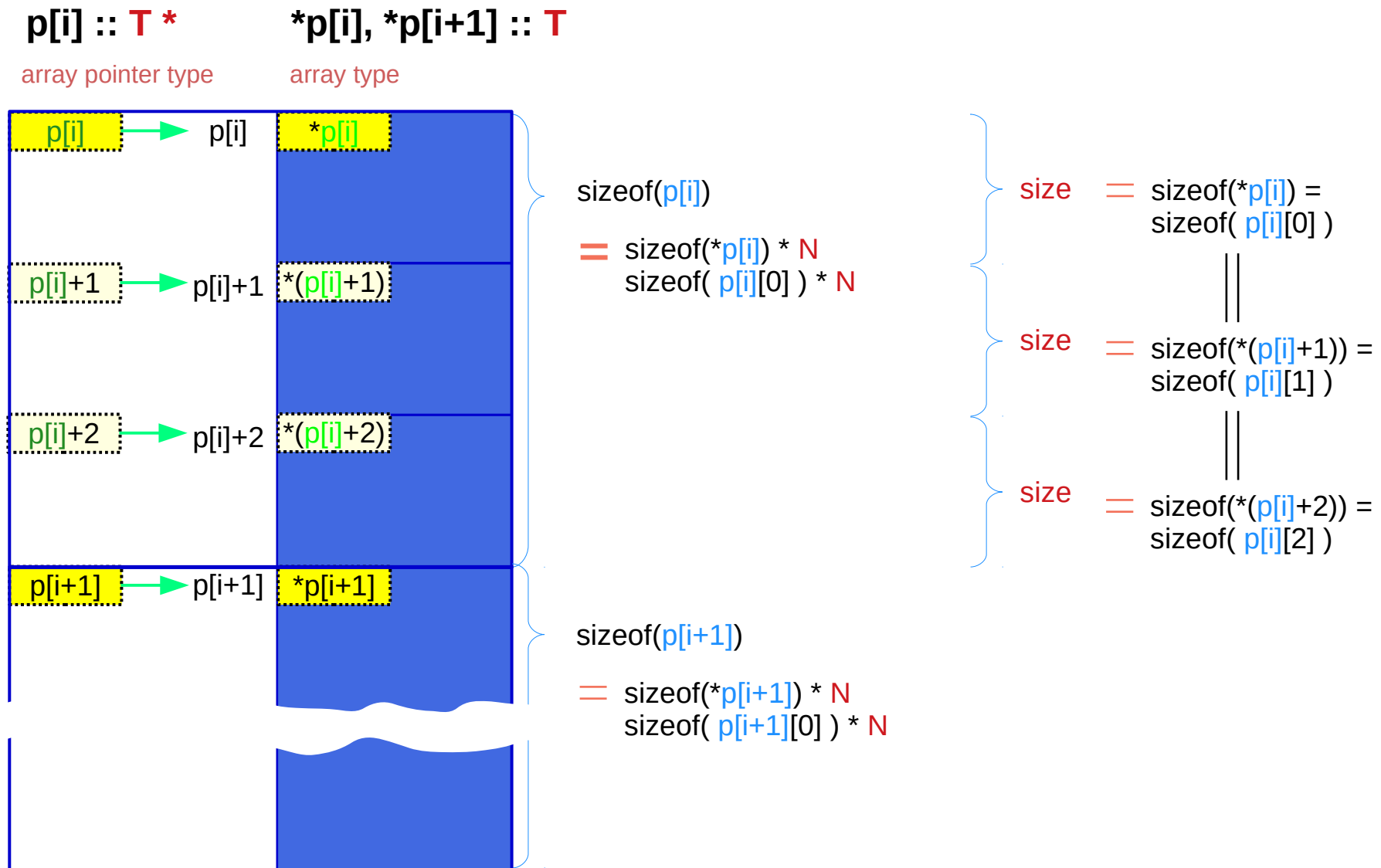
$p[i]$  has an array type (abstract data)  
 $p[i]$  is the name of an array  
 $p[i]$  has the size of the whole array

## Virtual array pointer $p[i]$

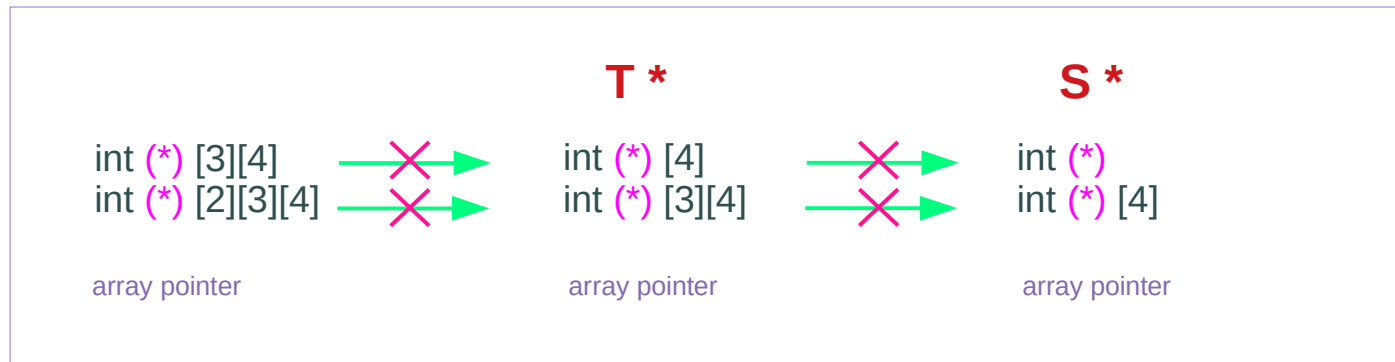
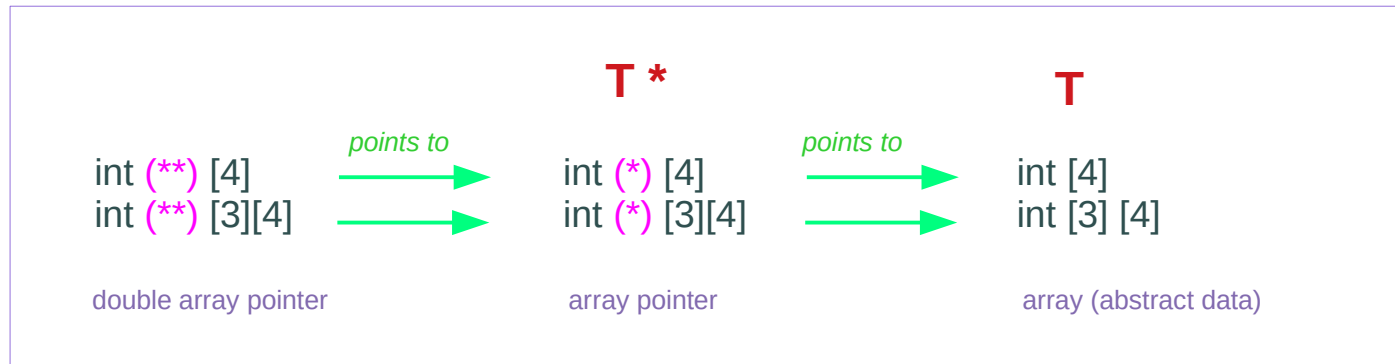


$p[i]$  also has an array pointer type  
 $p[i]$  has the value of the starting address  
 $p[i]$  is a virtual array pointer

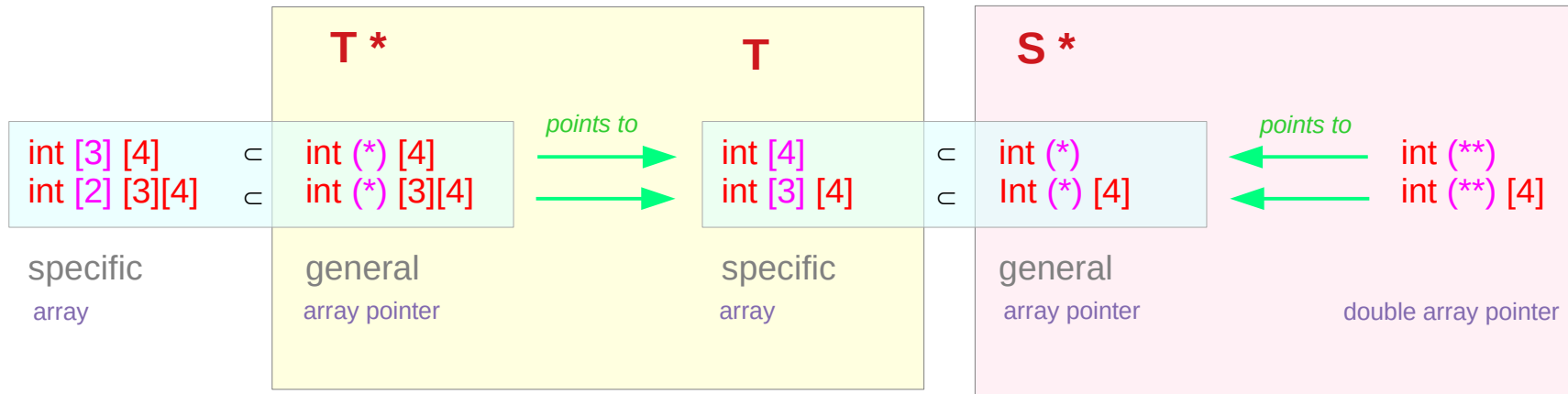
# Virtual pointers to a sub array



# Array pointer types



# Array pointer types point to array types



# Virtual array pointers pointing to arrays

```
typedef int (*T1) [4];  
typedef int (*T1) [3][4];
```

int (\*) [4]  
int (\*) [3][4]  
general

```
typedef int T2[4];  
typedef int T2[3][4];
```

int [4]  
int [3] [4]  
specific

```
T1 a;  
T2 b;
```

T1 is a pointer type  
T2 is an array type  
T1 has one more dimension than T2

```
a = &b;  
*a = b;
```

a references b

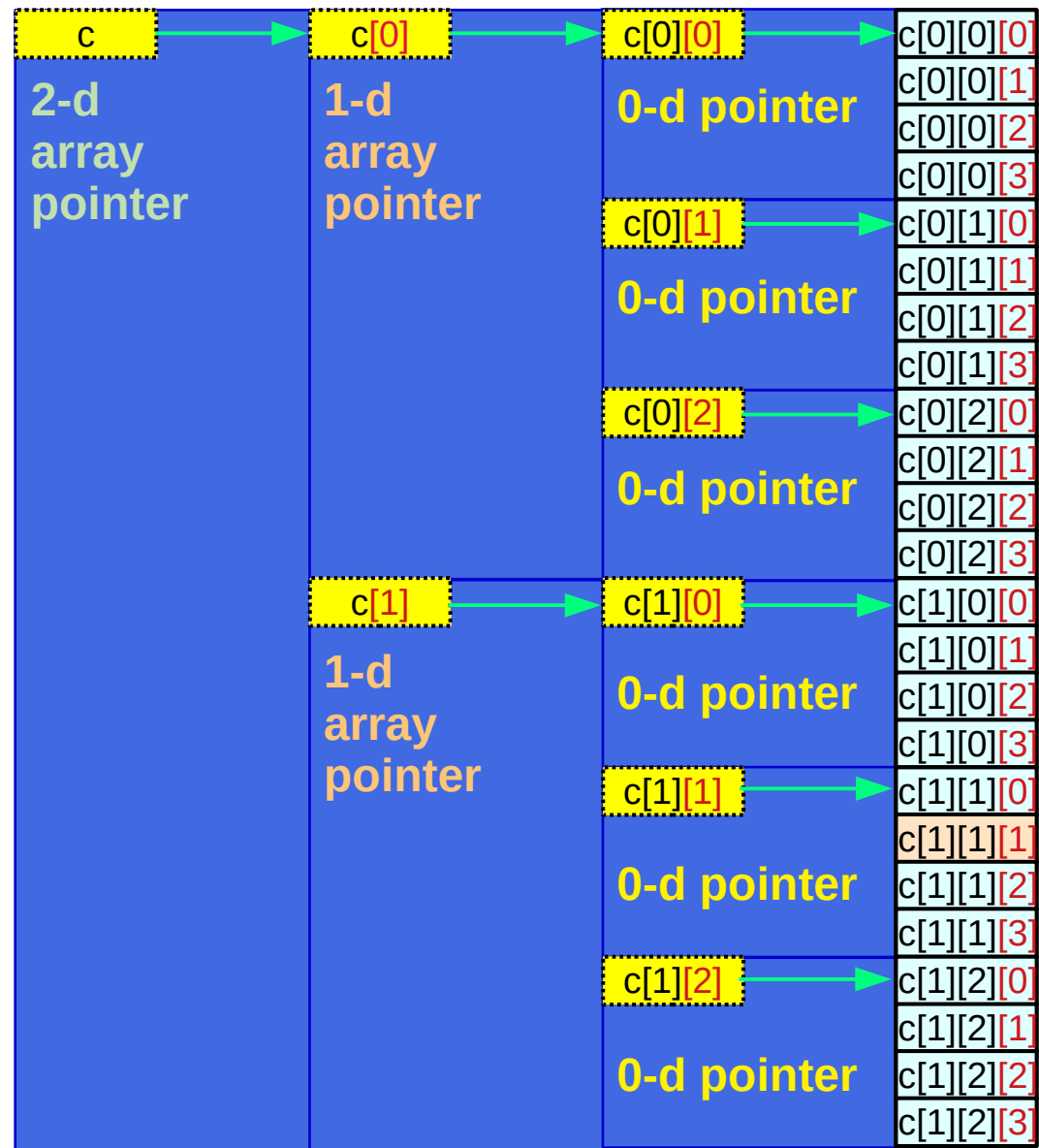
b is the dereference of a

# 3-d array structure – pointer representation

```
int c[2][3][4];
```

```
*(***(c +i) +j) +k)
```

- Hierarchical
- Nested Structure
- Virtual Array Pointers to abstract data (subarrays)
- Contiguous and Linear Data Layout
- Row Major Order



# 3-d array structure – abstract data representation

```
int c[2][3][4];
```

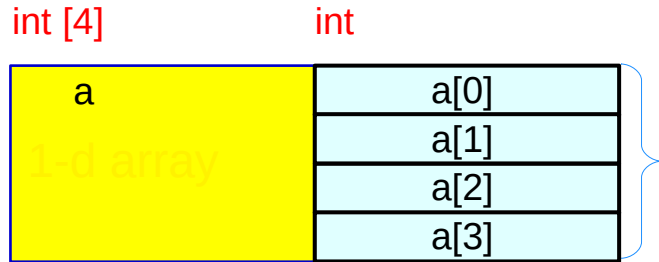
```
((c [i])[j])[k]
```

- Hierarchical
- Nested Structure
- Virtual Array Pointers to abstract data (subarrays)
- Contiguous and Linear Data Layout
- Row Major Order

c 3-d array name	c[0] 2-d array name	c[0][0] 1-d array name	c[0][0][0] c[0][0][1] c[0][0][2] c[0][0][3]
		c[0][1] 1-d array name	c[0][1][0] c[0][1][1] c[0][1][2] c[0][1][3]
		c[0][2] 1-d array name	c[0][2][0] c[0][2][1] c[0][2][2] c[0][2][3]
	c[1] 2-d array name	c[1][0] 1-d array name	c[1][0][0] c[1][0][1] c[1][0][2] c[1][0][3]
		c[1][1] 1-d array name	c[1][1][0] c[1][1][1] c[1][1][2] c[1][1][3]
		c[1][2] 1-d array name	c[1][2][0] c[1][2][1] c[1][2][2] c[1][2][3]

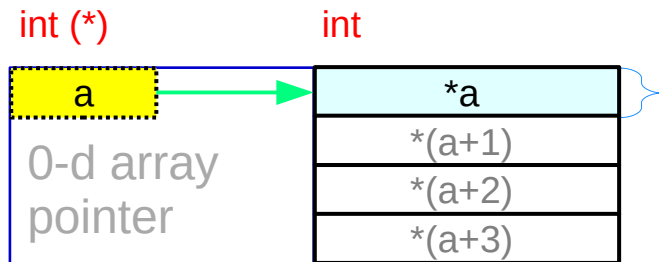


# Array **a** and pointer **a**



**1-d array **a****    specific array type

$\text{sizeof}(a)$



**pointer **a****    general pointer type

$\text{sizeof}(a) = \text{sizeof}(*a) * 4$

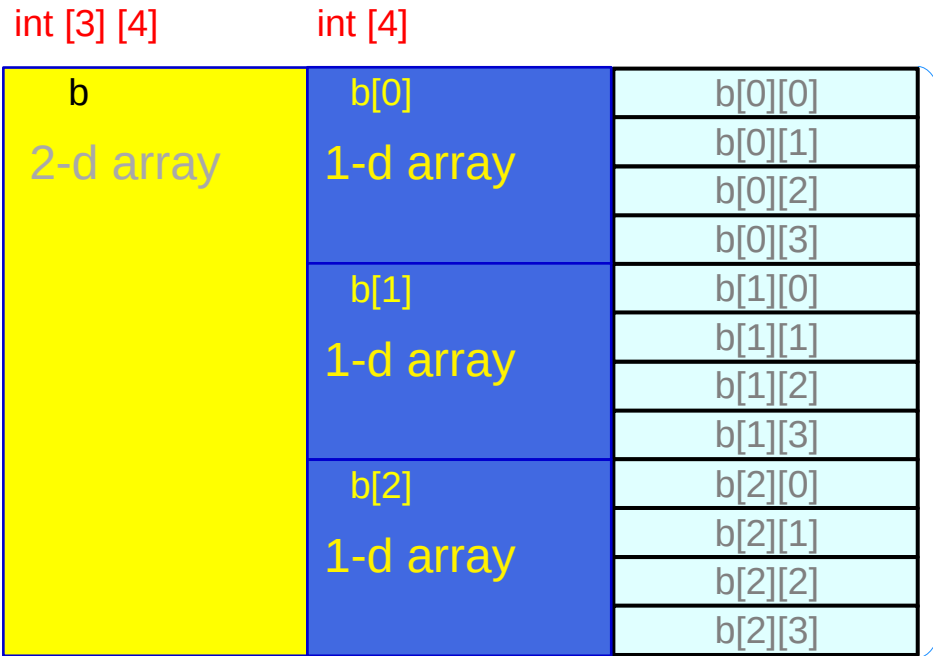
- a** is the name of a 1-d array
- a** also has a pointer type
- a** has the size of the array
- a** has the value of the starting address

**a** is a virtual array pointer

# Array **b** and pointer **b**

**2-d array **b**** specific array type

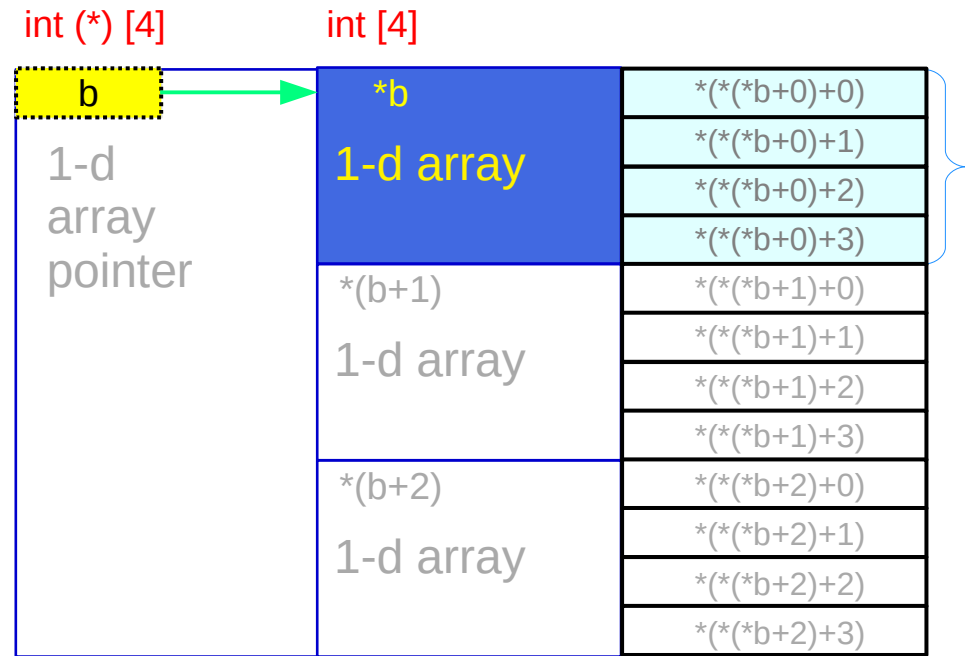
`sizeof(b)`



**b** is the name of a 2-d array  
**b** has the size of the array

**1-d array pointer **b**** general pointer type

`sizeof(b) = sizeof(*b) * 3`



**b** also has a 1-d array pointer type  
**b** has the value of the starting address

**b** is a virtual array pointer

# Array c

## 3-d array c

specific array type

sizeof(c)

c is the name of a 3-d array  
 c has the size of the array

int [2][3][4]	int [3][4]	int [4]		
c 3-d array	c[0] 2-d array	c[0][0] 1-d array	c[0][0][0]	
			c[0][0][1]	
			c[0][0][2]	
				c[0][0][3]
	c[0][1] 1-d array		c[0][1][0]	
			c[0][1][1]	
			c[0][1][2]	
				c[0][1][3]
	c[0][2] 1-d array		c[0][2][0]	
			c[0][2][1]	
			c[0][2][2]	
				c[0][2][3]
c[1] 2-d array	c[1][0] 1-d array		c[1][0][0]	
			c[1][0][1]	
			c[1][0][2]	
			c[1][0][3]	
c[1][1] 1-d array		c[1][1][0]		
		c[1][1][1]		
		c[1][1][2]		
			c[1][1][3]	
c[1][2] 1-d array		c[1][2][0]		
		c[1][2][1]		
		c[1][2][2]		
			c[1][2][3]	

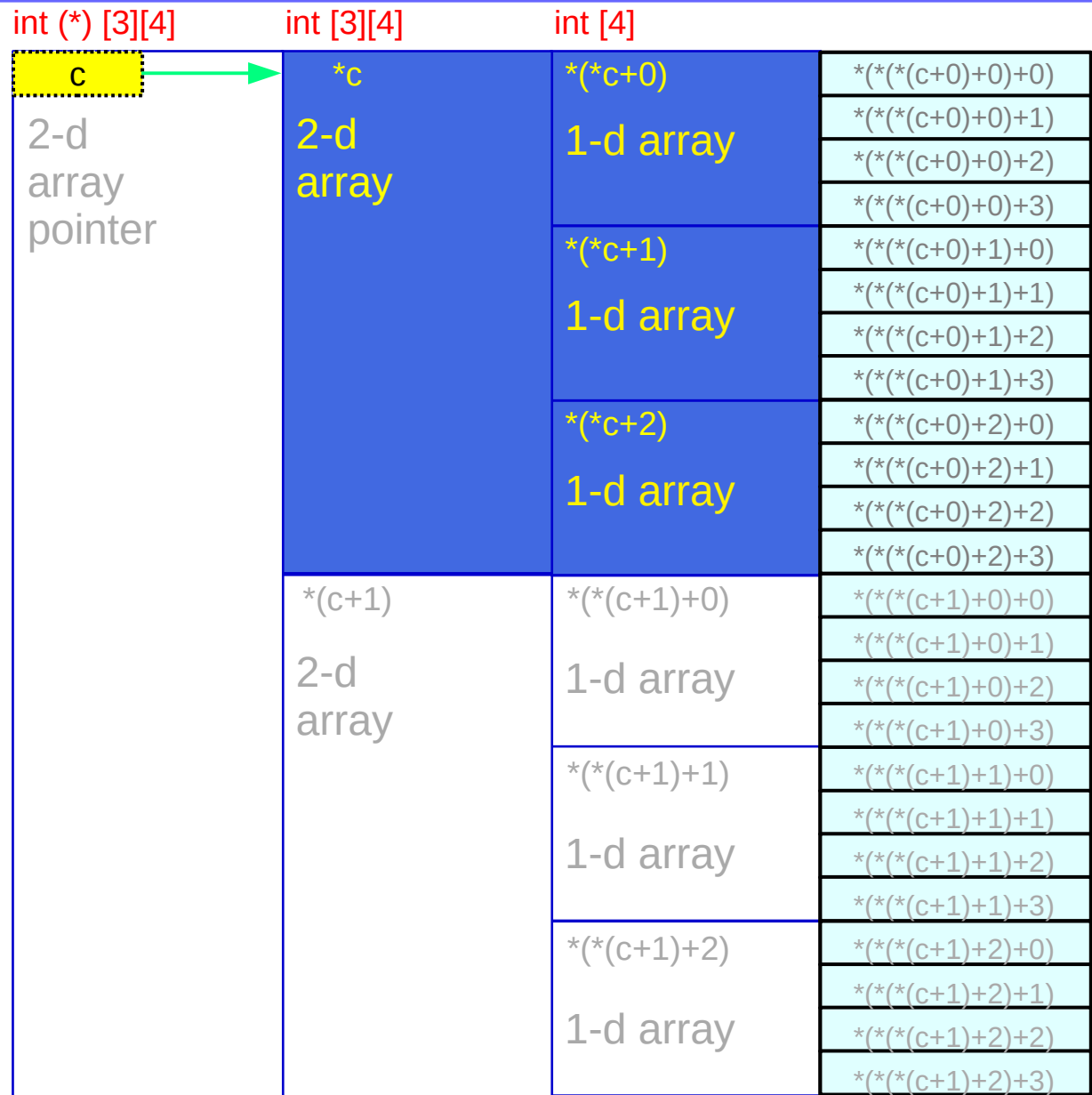
# Pointer c

## 2-d array pointer c

general pointer type

$\text{sizeof}(c) = \text{sizeof}(*c) * 2$

- c also has a 2-d array pointer type
- c has the value of the starting address
- c is a virtual array pointer



# Types of virtual array pointers in a 3-d array

```
int c[2][3][4];
```

`c[i][j][k]`

`c[i][j]`  
[k]

`c[i]`  
[j] [k]

`c`  
[i] [j] [k]

int

int [4]  
[k]

int [3][4]  
[j] [k]

int [2][3][4]  
[i] [j] [k]

array type (name)

int

int (\*)  
[k]

int (\*)[4]  
[j] [k]

int (\*)[3][4]  
[i] [j] [k]

array pointer type

# Sizes of virtual array pointers in a 3-d array

```
int c[2][3][4];
```

$$\text{sizeof}(c[i][j][k]) = \text{sizeof}(\text{int})$$

$$\text{sizeof}(c[i][j])_{[k]} = \text{sizeof}(\text{int}) * 4_{[k]}$$

$$\text{sizeof}(c[i])_{[j] [k]} = \text{sizeof}(\text{int}) * 3 * 4_{[j] [k]}$$

$$\text{sizeof}(c)_{[i] [j] [k]} = \text{sizeof}(\text{int}) * 2 * 3 * 4_{[i] [j] [k]}$$

# Address values of virtual array pointers in a 3-d array

```
int c[2][3][4];
```

$c[i][j][k] = \&c[i][j][k]$

$c[i][j]+k = \&c[i][j][0] + k * \text{sizeof}(c[i][j][k])$        $\text{sizeof}(*c[i][j]) = \text{sizeof}(c[i][j][0]) = \text{sizeof}(\text{int})$

$c[i]+j = \&c[i][0][0] + j * \text{sizeof}(c[i][j])$        $\text{sizeof}(*c[i]) = \text{sizeof}(c[i][0]) = \text{sizeof}(\text{int}) * 4$   
[k]

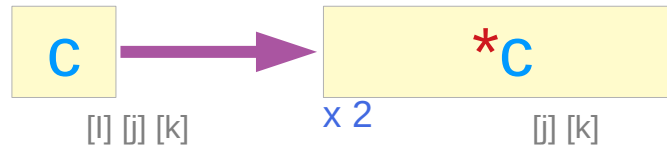
$c+i = \&c[0][0][0] + i * \text{sizeof}(c[i])$        $\text{sizeof}(*c) = \text{sizeof}(c[0]) = \text{sizeof}(\text{int}) * 3 * 4$   
[j] [k]

# Types in a multi-dimensional 3-d array

```
int c [2][3][4];
```

abstract data `int [2] [3][4]`

array pointer `int (*) [3][4]`

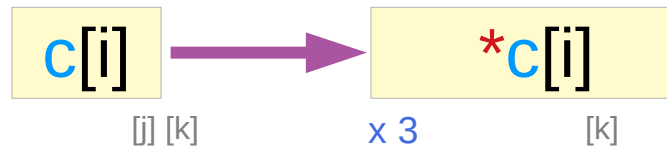


`int [3] [4]` abstract data

`int (*) [4]` array pointer

abstract data `int [3] [4]`

array pointer `int (*) [4]`

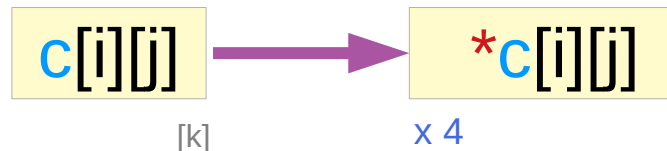


`int [4]` abstract data

`int (*)` array pointer

abstract data `int [4]`

array pointer `int (*)`

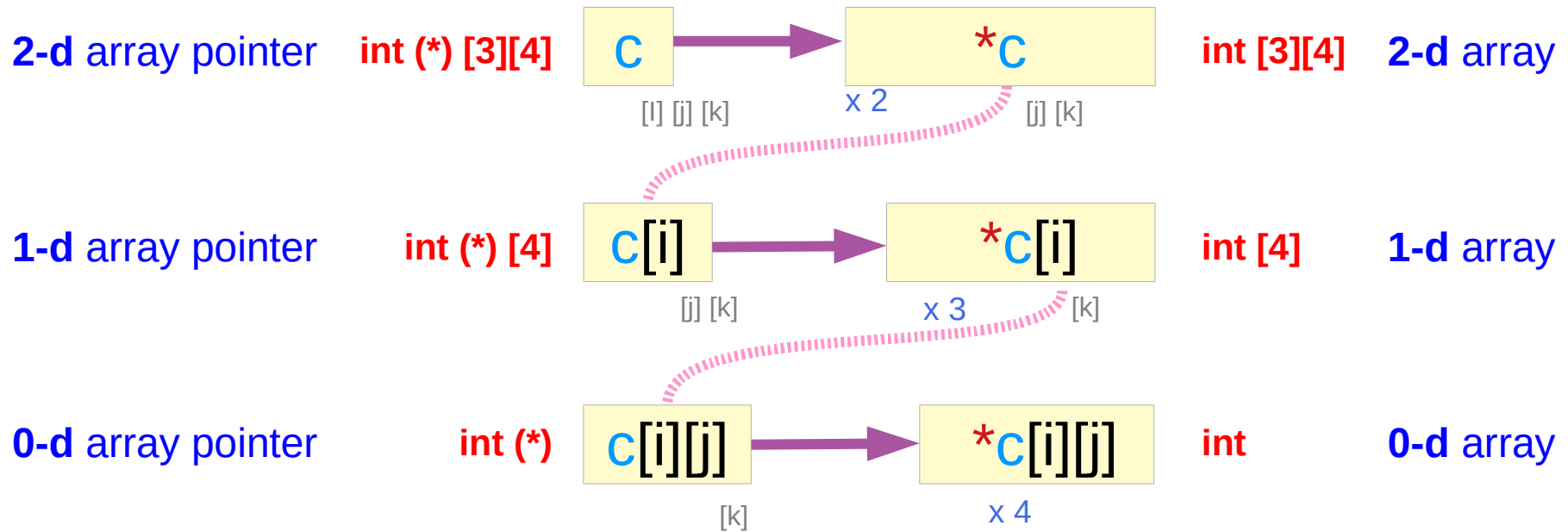


`int` primitive data



# Virtual array pointers and abstract data

```
int c [2][3][4];
```



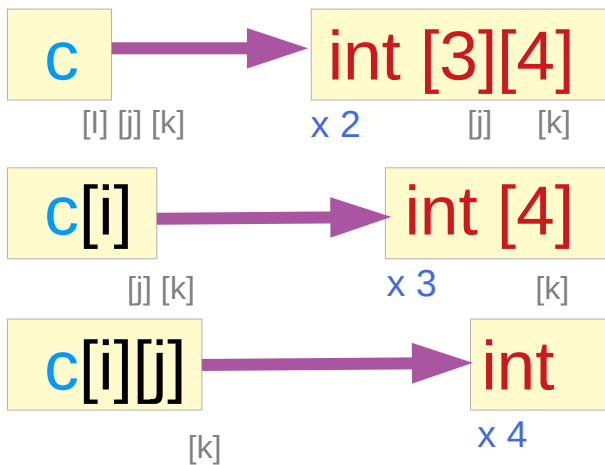
all these pointers are virtual, and take no actual memory locations

exploiting the **contiguity** of allocated memory locations

# Abstract data sizes

```
int c [2][3][4];
```

the size of a pointer type is fixed  
Here, the sizes of virtual pointers are shown  
i.e, the sizes of different abstract data types



```
sizeof( c) = sizeof(int [2][3][4])
sizeof(*c) = sizeof(int [3][4])

sizeof( c[i]) = sizeof(int [3][4])
sizeof(*c[i]) = sizeof(int [4])

sizeof( c[i][j]) = sizeof(int [4])
sizeof(*c[i][j]) = sizeof(int)
```

all are sizes of arrays

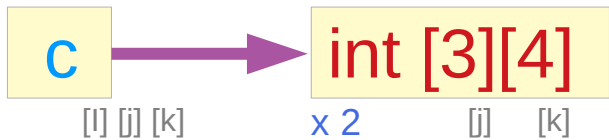
`c`, `c[i]`, `c[i][j]` are virtual array pointers  
and they are also abstract data (arrays)

when sizes are considered,  
view them as abstract data (arrays)

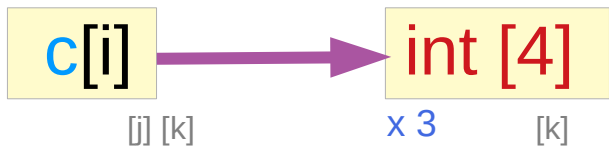
# Virtual array pointer sizes and abstract data sizes

```
int c [2][3][4];
```

$$\text{size of a virtual array pointer} = \text{size of the pointed abstract data type} * \text{the number of such types}$$



$$\text{sizeof}( c ) = \text{sizeof}( *c ) * 2$$



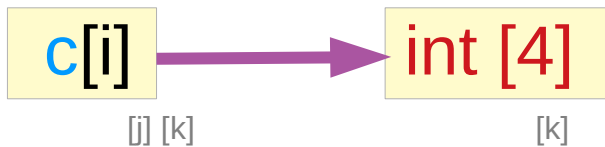
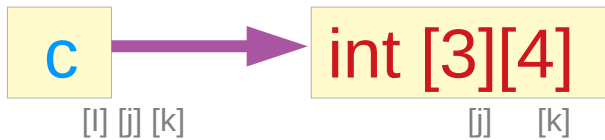
$$\text{sizeof}( c[i] ) = \text{sizeof}( *c[i] ) * 3$$



$$\text{sizeof}( c[i][j] ) = \text{sizeof}( *c[i][j] ) * 4$$

# Sizes of array pointer types

`int c [2][3][4];`



not real array pointers  
virtual array pointers



`c` `int (*)[3][4]` = sizeof(c)  
 sizeof(int (\*) [3][4]) = pointer size  $\neq$  sizeof(c)

`c[i]` `int (*) [4]` = sizeof(c[i])  
 sizeof(int (\*) [4]) = pointer size  $\neq$  sizeof(c[i])

`c[i][j]` `int [4]` = sizeof(c[i][j])  
 sizeof(int [4]) = pointer size  $\neq$  sizeof(c[i][j])



4 bytes for 32-bit machines  
8 bytes for 64-bit machines

# Virtual array pointer increment size

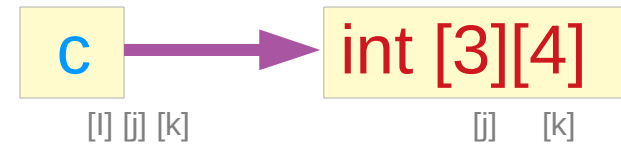
```
int c [2][3][4];
```

c points to a **2-d** array  
increment size: `sizeof(int[2][3][4])`

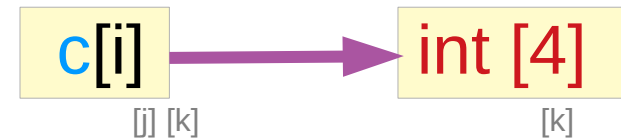
c[i] points to an **1-d** array  
increment size: `sizeof(int[3][4])`

c[i][j] points to an integer  
increment size: `sizeof(int[4])`

int (\*) [3][4]



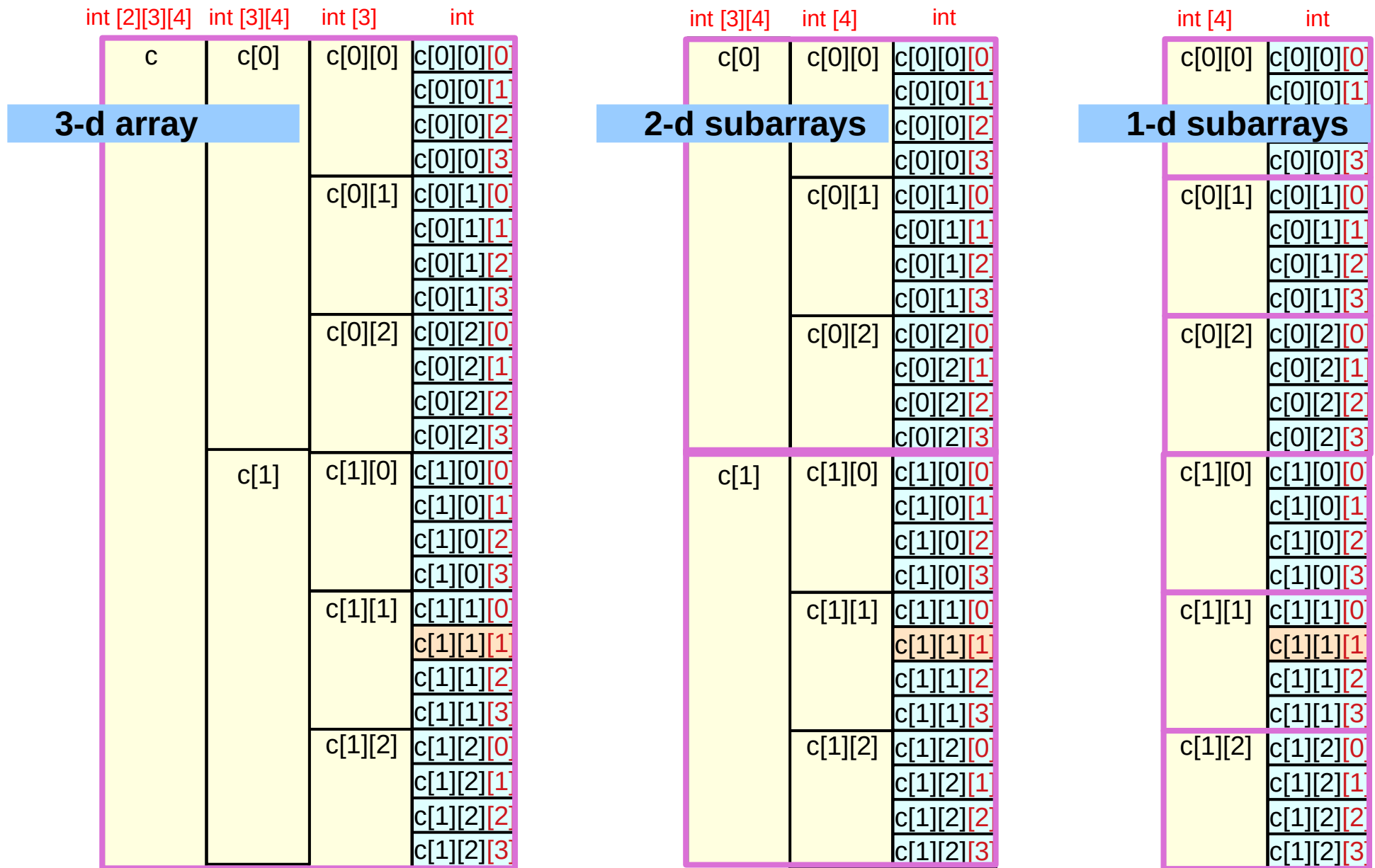
int (\*) [4]



int (\*)



# Subarrays in a 3-d array



# Virtual array pointer c, c[0], c[0][0] – types and sizes

## Types – array pointers

**int (\*) [3][4]**      **int [3][4]**

array pointer

array (abstract data)

c

c[0]  
c[1]

**int (\*) [4]**      **int [4]**

array pointer

array (abstract data)

c[0]

c[0][0]  
c[0][1]  
c[0][2]

**int [4]**      **int**

array pointer

array (abstract data)

c[0][0]

c[0][0][0]  
c[0][0][1]  
c[0][0][2]  
c[0][0][3]

## Sizes – abstract data

sizeof(c)  
sizeof(int [2][3][4])  
sizeof(int) \* 2 \* 3 \* 4

sizeof(int [2][3][4]) = 96  
sizeof(int (\*)[3][4]) = 4 / 8

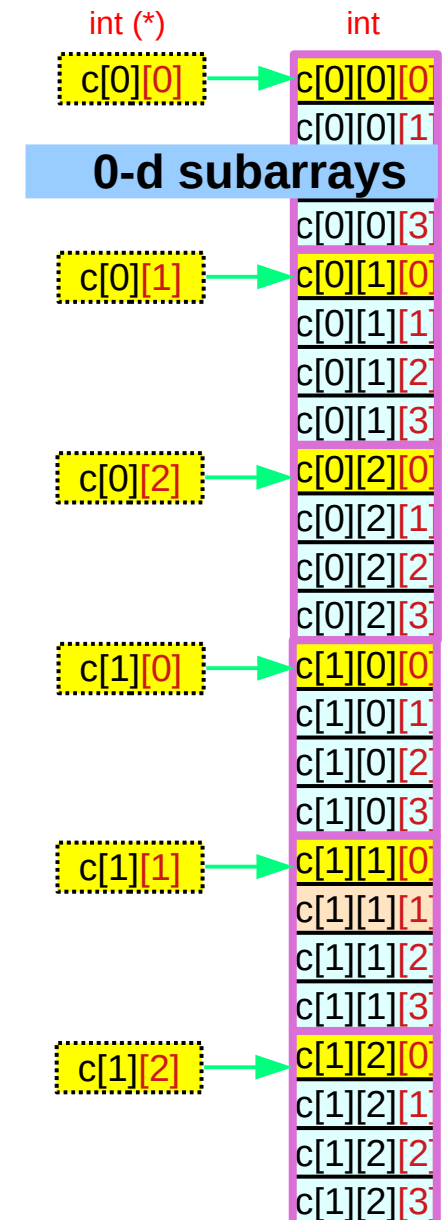
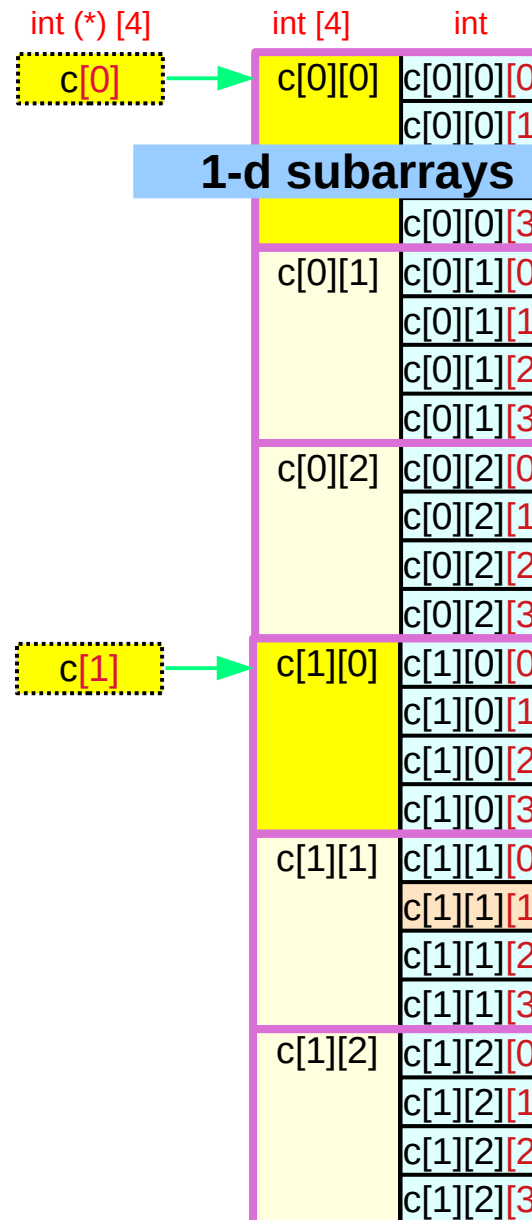
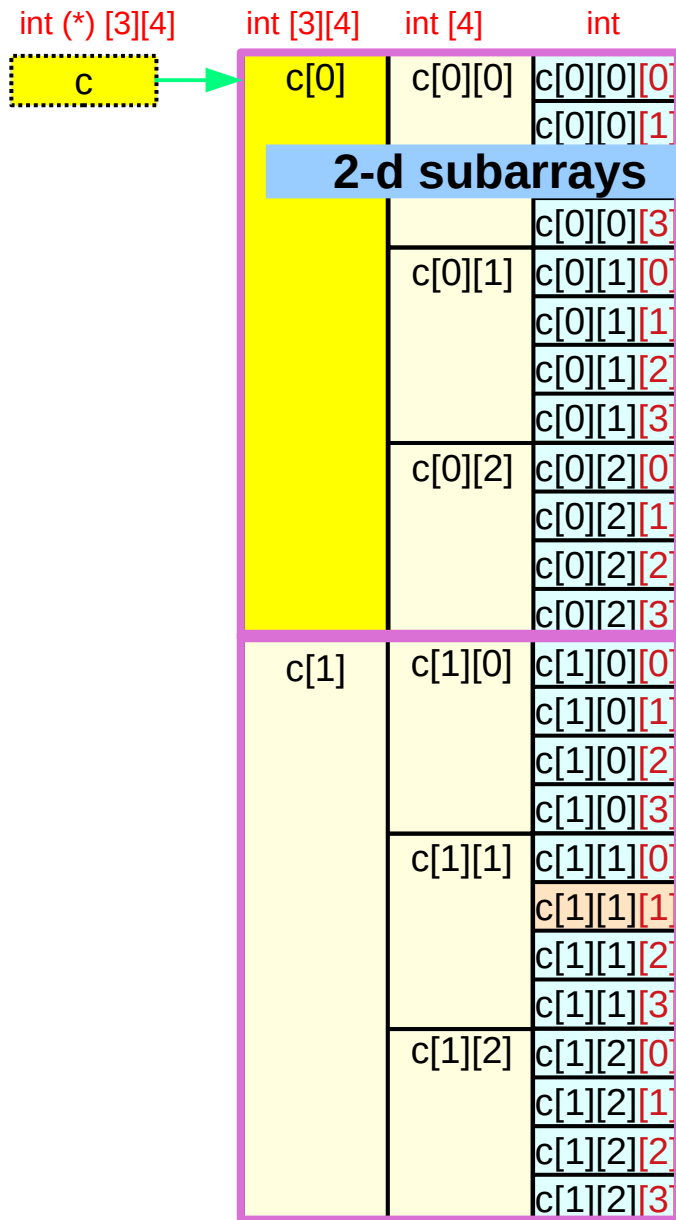
sizeof(c[0])  
sizeof(int [3][4])  
sizeof(int) \* 3 \* 4

sizeof(int [3][4]) = 48  
sizeof(int (\*)[4]) = 4 / 8

sizeof(c[0][0])  
sizeof(int [4])  
sizeof(int) \* 4

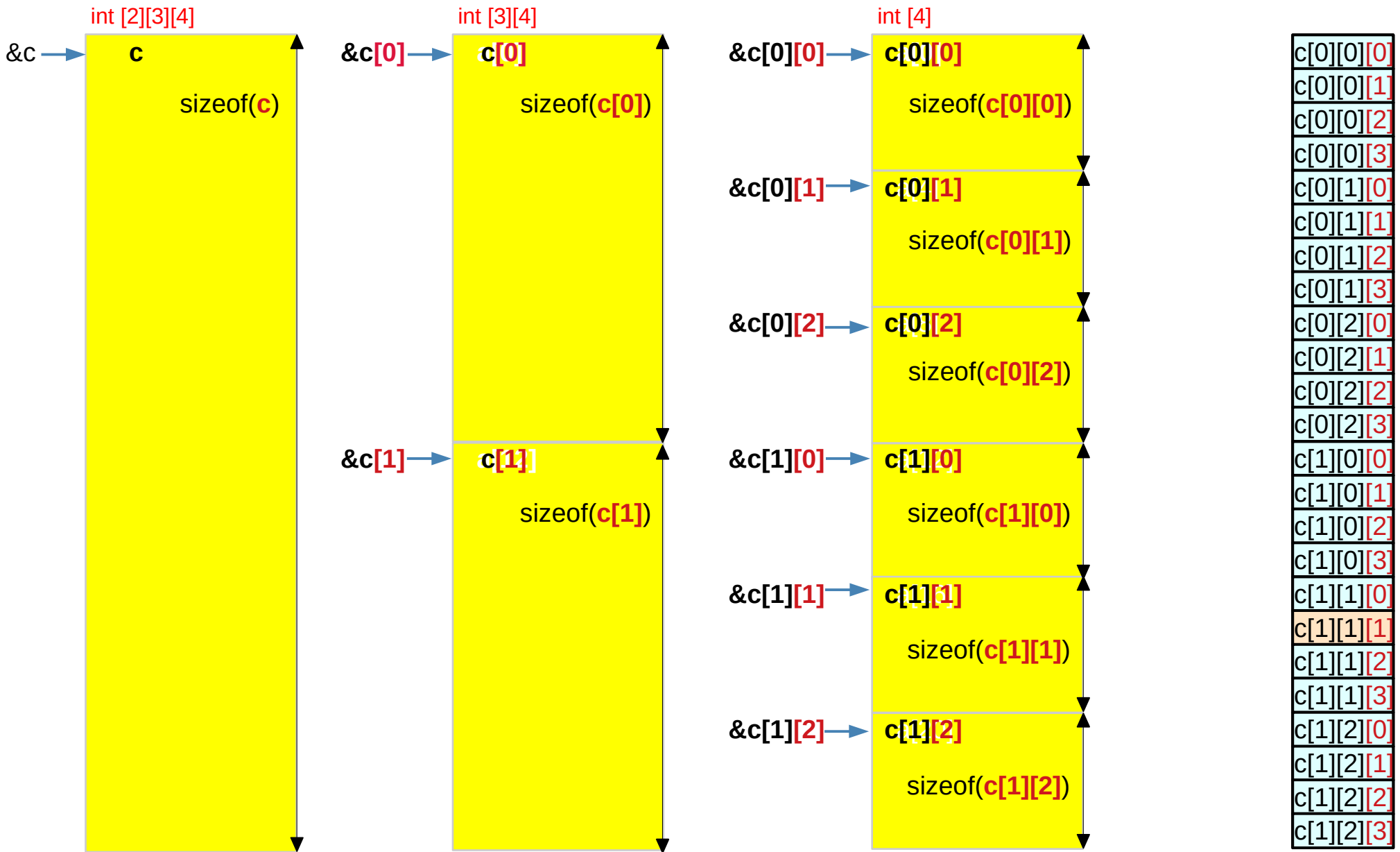
sizeof(int [4]) = 16  
sizeof(int (\*) ) = 4 / 8

# Pointers to subarrays in a 3-d array





# Abstract Data $c$ , $c[i]$ , $c[i][j]$ – start addresses and sizes



# Virtual array pointers – types, sizes, and values

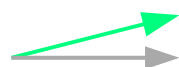
**int c[2][3][4];**

**c[i][j]**

**c[i][j][0]**

type

int [4]  
int (\*)



int  
int

- abstract data type
- array pointer type

size

sizeof(c[i][j]) =

sizeof(c[i][j][0]) \* 4

= sizeof(int) \* 4

value (address)

c[i][j] =

&c[i][j][0]

**int c[2][3][4];**

**c[i]**

**c[i][0]**

type

int [3][4]  
int (\*)[4]



int [4]  
int (\*)

- abstract data type
- array pointer type

size

sizeof(c[i]) =

sizeof(c[i][0]) \* 3

= sizeof(int) \* 4 \* 3

value (address)

c[i] =

&c[i][0][0]

**int c[2][3][4];**

**c**

**c[0]**

type

int [2][3][4]  
int (\*)[3][4]



int [3][4]  
int (\*)[4]

- abstract data type
- array pointer type

size

sizeof(c) =

sizeof(c[0]) \* 2

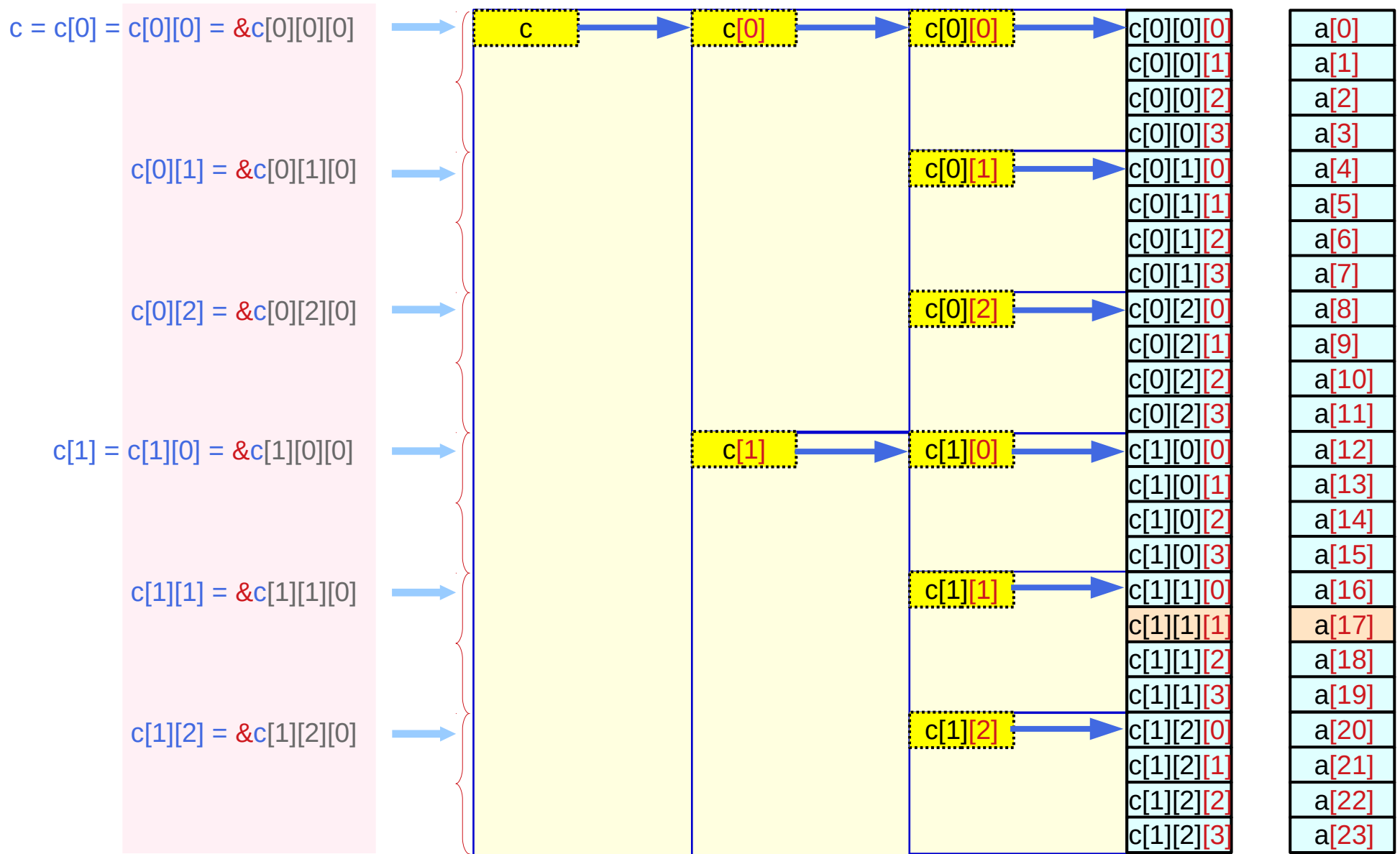
= sizeof(int) \* 4 \* 3 \* 2

value (address)

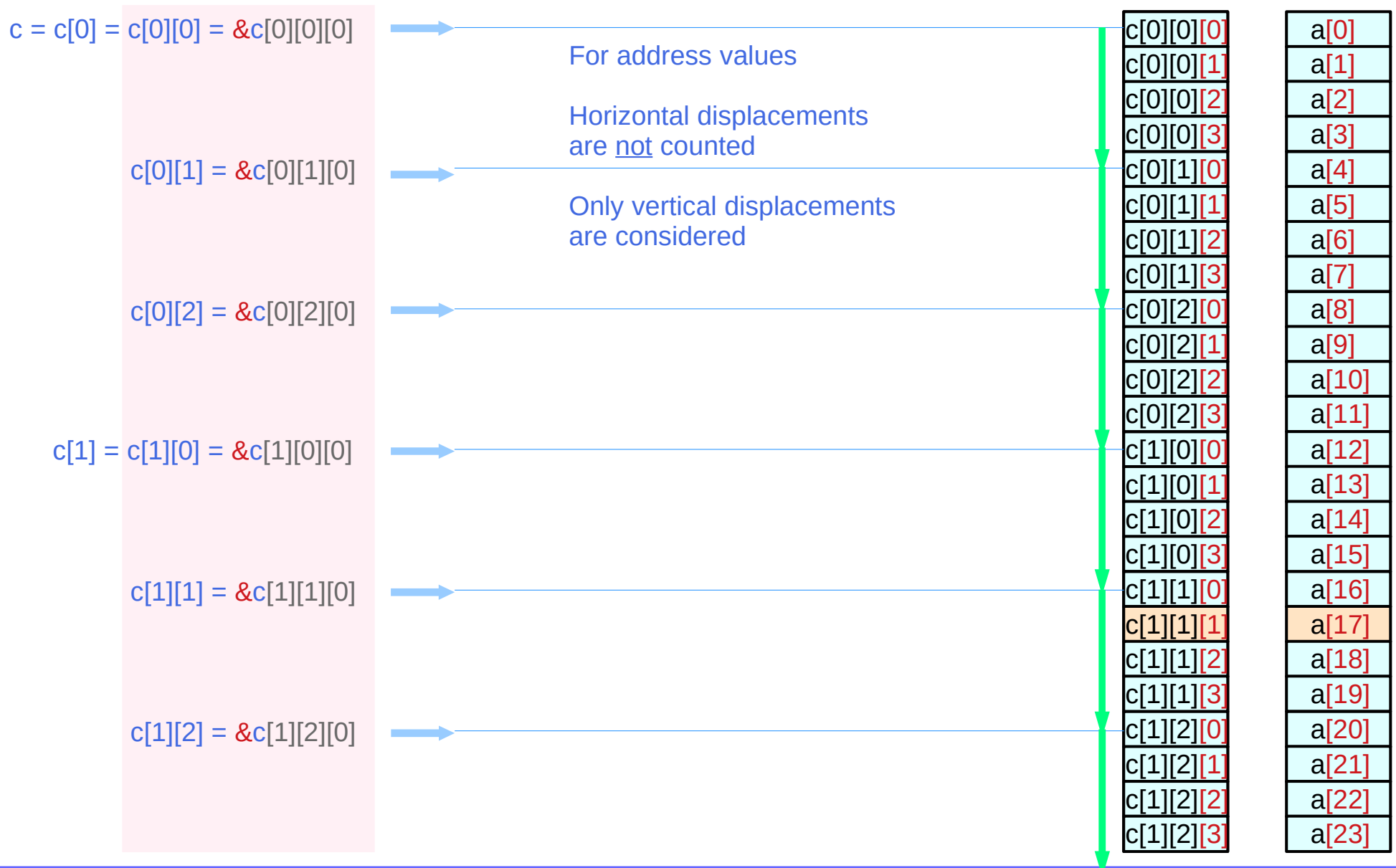
c =

&c[0][0][0]

# Virtual array pointer $c$ , $c[i]$ , $c[i][j]$ – values (addresses)



# Virtual array pointer $c$ , $c[i]$ , $c[i][j]$ – vertical displacement



# Virtual array pointer c, c[i], c[i][j] – values and types

$c = c[0] = c[0][0] = \&c[0][0][0]$  means  $\rightarrow$   
 $c[0][1] = \&c[0][1][0]$  means  $\rightarrow$   
 $c[0][2] = \&c[0][2][0]$  means  $\rightarrow$   
 $c[1] = c[1][0] = \&c[1][0][0]$  means  $\rightarrow$   
 $c[1][1] = \&c[1][1][0]$  means  $\rightarrow$   
 $c[1][2] = \&c[1][2][0]$  means  $\rightarrow$

$value(c) = value(c[0]) = value(c[0][0]) = value(\&c[0][0][0])$ $type(c) \neq type(c[0]) \neq type(c[0][0]) = type(\&c[0][0][0])$ $int (*) [3][4] \quad int (*) [4] \quad int * \quad int *$
$value(c[0][1]) = value(\&c[0][1][0])$ $type(c[0][1]) = type(\&c[0][1][0])$ $int * \quad int *$
$value(c[0][2]) = value(\&c[0][2][0])$ $type(c[0][2]) = type(\&c[0][2][0])$ $int * \quad int *$
$value(c[1]) = value(c[1][0]) = value(\&c[1][0][0])$ $type(c[1]) \neq type(c[1][0]) = type(\&c[1][0][0])$ $int (*) [4] \quad int * \quad int *$
$value(c[1][1]) = value(\&c[1][1][0])$ $type(c[1][1]) = type(\&c[1][1][0])$ $int * \quad int *$
$value(c[1][2]) = value(\&c[1][2][0])$ $type(c[1][2]) = type(\&c[1][2][0])$ $int * \quad int *$

# Summary of virtual array pointers in a 3-d array

$$c[i] \equiv *(c + i)$$

int (\*) [3][4] 2-d array pointer `c`  
 int [2] [3][4] 3-d array name `c`

address value `c + i`

`&c[0][0][0] + i * sizeof(*c)`  
`&c[0][0][0] + i * sizeof(c[0])`  
`&c[0][0][0] + i * 4 * 3 * 4`

leading elements

`c[0][0][0]`

$$c[i][j] \equiv *(c[i] + j)$$

int (\*) [4] 1-d array pointers `c[i]`  
 Int [3] [4] 2-d array names `c[i]`

address value `c[i] + j`

`&c[i][0][0] + j * sizeof(*c[i])`  
`&c[i][0][0] + j * sizeof(c[i][0])`  
`&c[i][0][0] + j * 4 * 4`

leading elements

`c[0][0][0]`

`c[1][0][0]`

$$c[i][j][k] \equiv *(c[i][j] + k)$$

int (\*) 0-d array pointers `c[i][j]`  
 int [4] 1-d array names `c[i][j]`

address value `c[i][j] + k`

`&c[i][j][0] + k * sizeof(*c[i][j])`  
`&c[i][j][0] + k * sizeof(c[i][j][0])`  
`&c[i][j][0] + k * 4`

leading elements

`c[0][0][0]`  
`c[0][1][0]`  
`c[0][2][0]`  
`c[1][0][0]`  
`c[1][1][0]`  
`c[1][2][0]`

# Sub-array properties in multi-dimensional arrays

`int c [2][3][4];` → 3-d access `c [i][j][k]`

2-d array pointer	<code>c</code>	<code>int (*) [3][4]</code>
1-d array pointers	<code>c[i]</code>	<code>int (*) [4]</code>
0-d array pointers	<code>c[i][j]</code>	<code>int (*)</code>

# Hierarchical Sub-arrays in a 3-d array

```
int c [L][M][N];
```

```
c [i][j][k]
```

left-to-right associativity

Array Names and Types

Pointers to hierarchical sub-arrays

c	[i]	[j][k]
c[i]	[j]	[k]
c[i][j]	[k]	

c	3-d array names	int (*) [M][N]	2-d array pointer
c[i]	2-d array names	int (*) [N]	1-d array pointer
c[i][j]	1-d array names	int (*)	0-d array pointer



# Associativity and Equivalence Relations

left-to-right associativity

$$((c[i])[j])[k]$$

$$\equiv$$

left-to-right associativity

$$*(*(*(c+i)+j)+k)$$

$$X[n]$$

$$\equiv$$

$$*(X+n)$$

given  $c[i][j]$

$$c[i][j][k]$$

$$\equiv$$

$$*(c[i][j]+k)$$

for all  $k$

given  $c[i]$

$$c[i][j]$$

$$\equiv$$

$$*(c[i]+j)$$

for all  $j$

given  $c$

$$c[i]$$

$$\equiv$$

$$*(c+i)$$

for all  $i$

# Requirements for the expression $c[i][j][k]$

for a given  $c[i][j]$ , for all  $k$

$$c[i][j][k] = *(c[i][j] + k)$$

for a given  $c[i]$ , for all  $j$

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

for a given  $c$ , for all  $i$

$$c[i] = *(c + i)$$

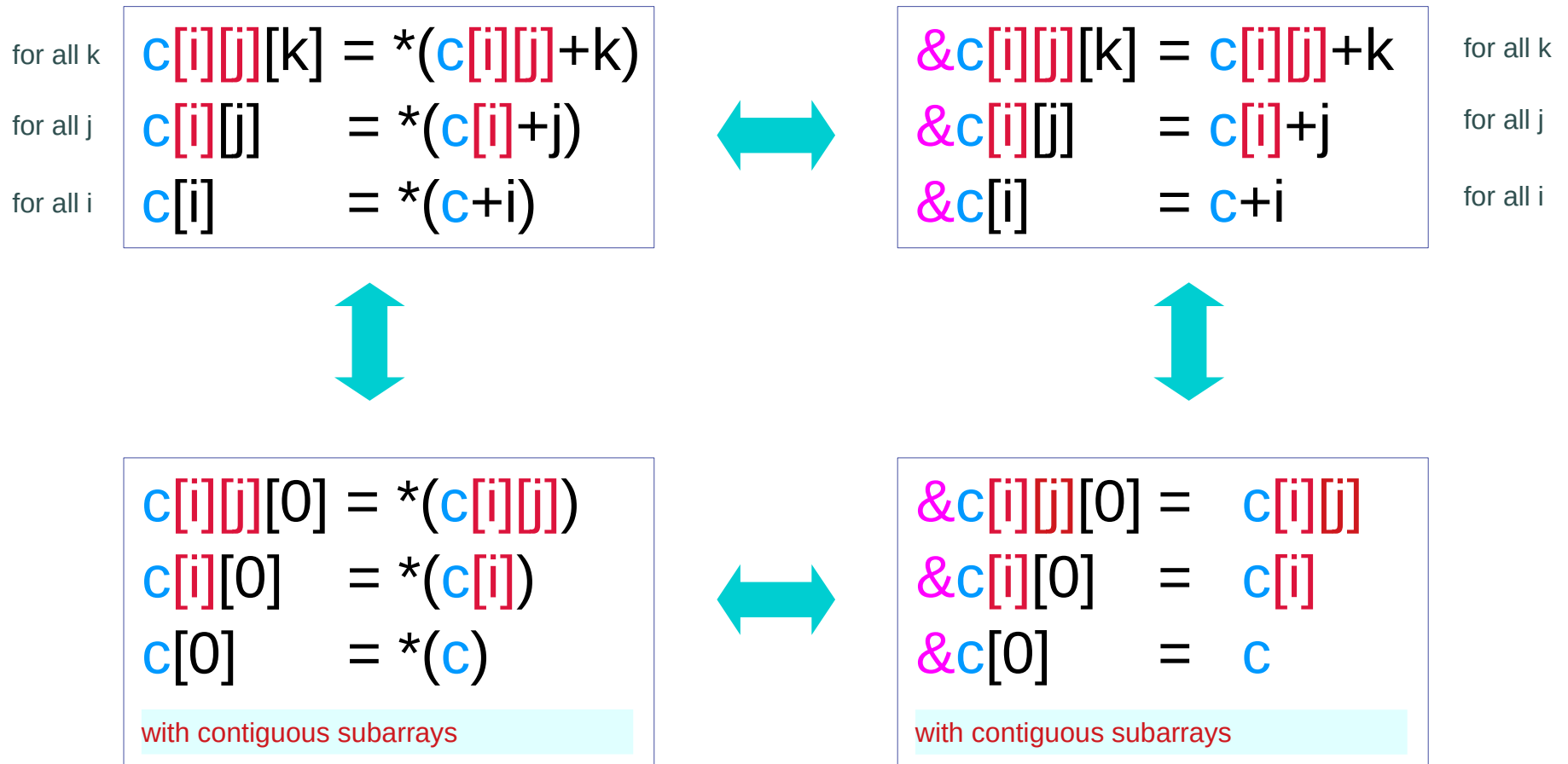
## 3 contiguity requirements

for a given  $c[i][j]$ , contiguous  $c[i][j][k]$

for a given  $c[i]$ , contiguous  $c[i][j]$

for a given  $c$ , contiguous  $c[i]$

# Equivalent requirements for the expression $c[i][j][k]$



# 3-d access pattern $c[i][j][k]$

## General requirements

$c[i][j][k]$



$\&c[i][j][k] = c[i][j] + k$  for all  $k$   
 $\&c[i][j] = c[i] + j$  for all  $j$   
 $\&c[i] = c + i$  for all  $i$

## Pointer array approach

```
int** c[2];  
int* b[2*3];  
int c[2*3*4];
```

```
c[i][j][k] :: int  
c[i][j]    :: int *  
c[i]       :: int **
```

```
c[i] ← &b[i*3]  
b[j] ← &a[j*4]
```

with contiguous subarrays

**Explicit**  
Arrays of pointers with  
Multiple Indirection

## N-dim Array approach

```
int c[2][3][4];
```

```
c[i][j][k] :: int  
c[i][j]    :: int [4]  
c[i]       :: int (*) [4]
```

```
c[i][j] ← &c[i][j][0]  
c[i]    ← &c[i][0][0]  
c       ← &c[0][0][0]
```

with contiguous subarrays

**Implicit**  
Nested  
Virtual Array Pointers

# 3-d access pattern $c[i][j][k]$ – array pointer approach

## General requirements

$c[i][j][k]$



$\&c[i][j][k] = c[i][j] + k$  for all  $k$   
 $\&c[i][j] = c[i] + j$  for all  $j$   
 $\&c[i] = c + i$  for all  $i$



## N-dim array approach

```
int c[2][3][4];
```

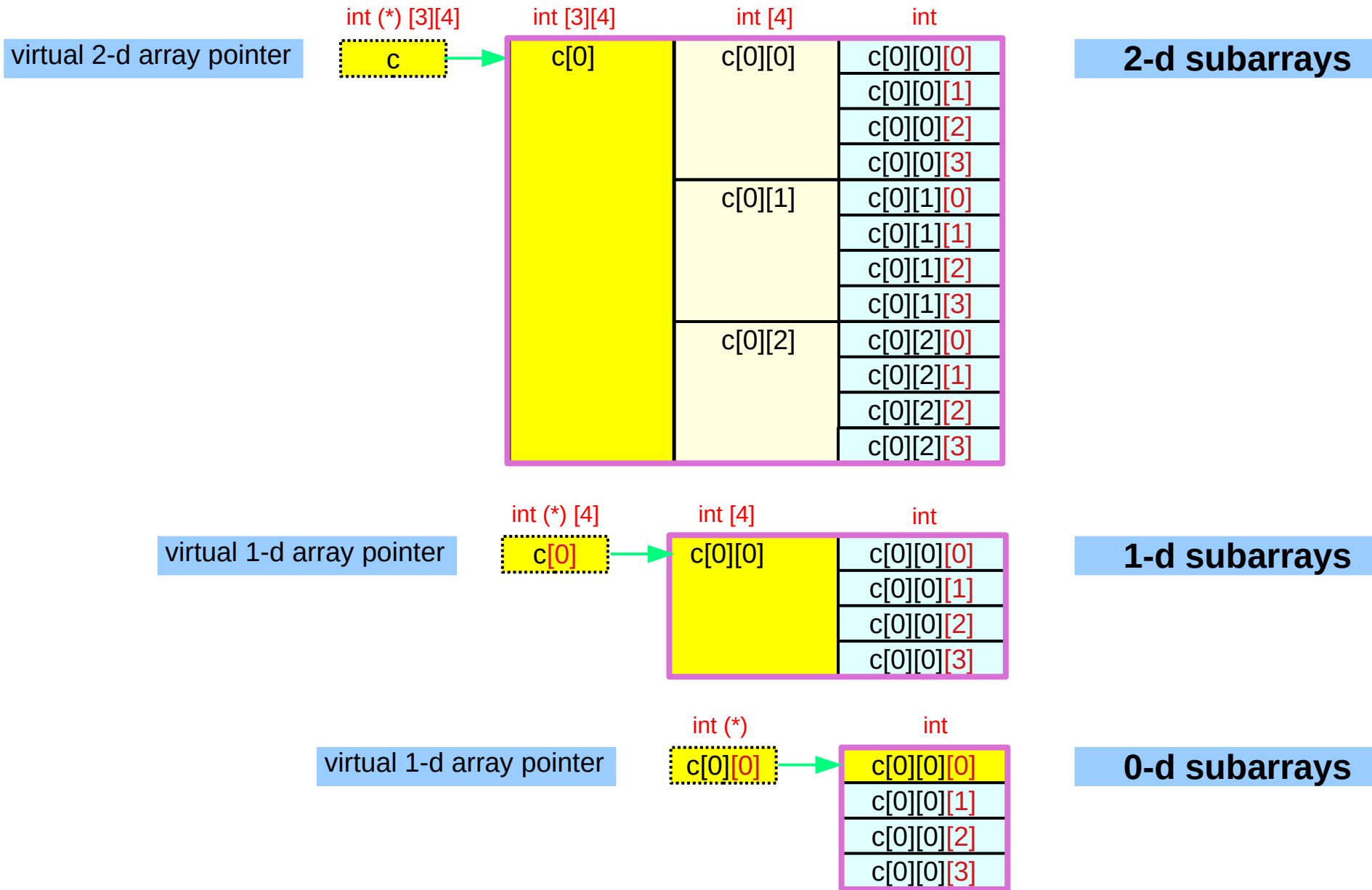
```
c[i][j][k] :: int  
c[i][j]   :: int [4]  
c[i]      :: int (*) [4]  
c         :: int (*) [3][4]
```

```
c[i][j] ← &c[i][j][0]  
c[i]    ← &c[i][0][0]  
c       ← &c[0][0][0]
```

with contiguous subarrays

**Implicit  
Nested  
Virtual Array Pointers**

# Pointers to subarrays in a 3-d array



# Using N-dimensional arrays

```
int c [2][3][4];
```

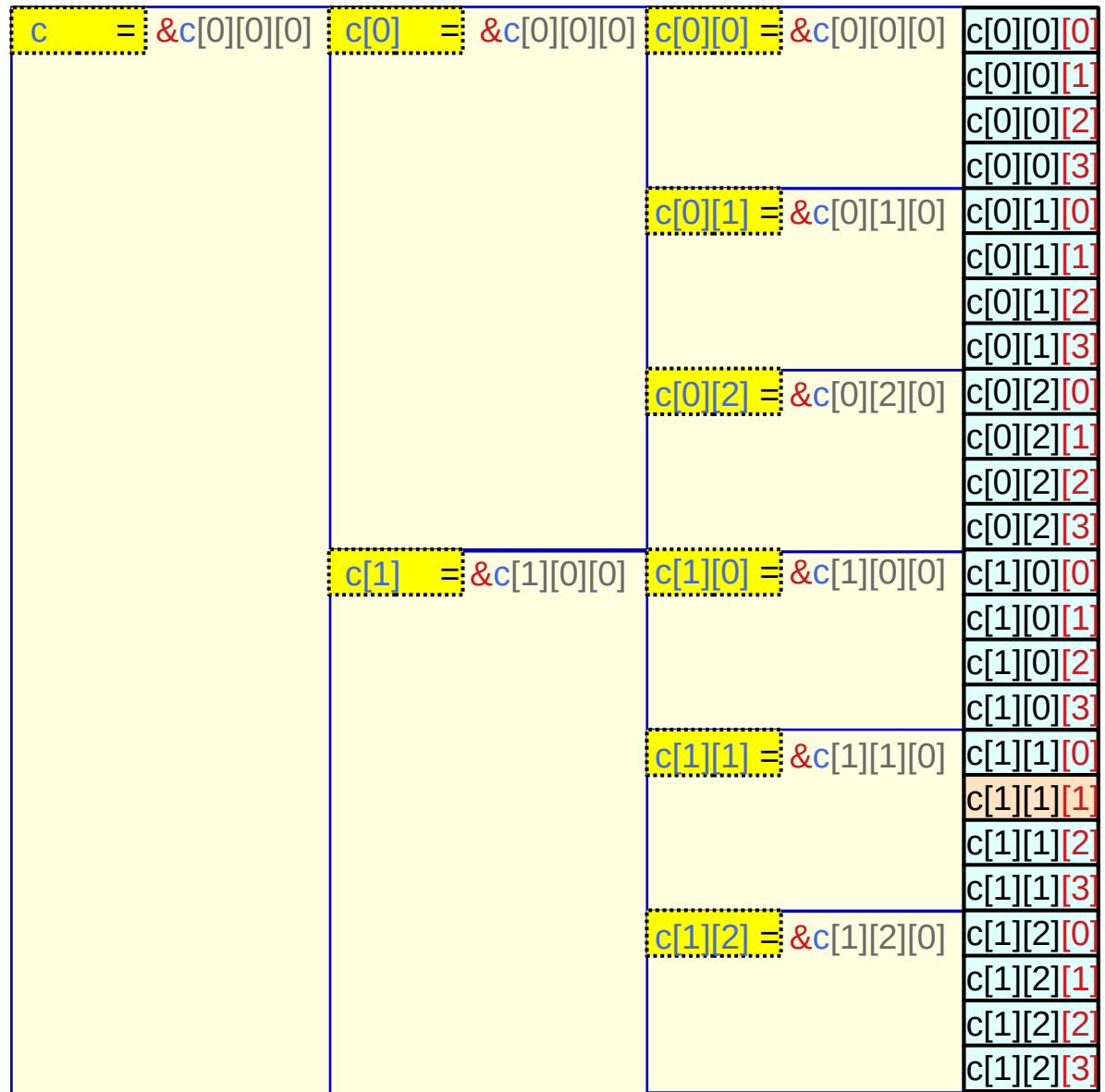


```
c [i][j][k];
```

## virtual assignments

```
c ← &c[0][0][0]
c[i] ← &c[i][0][0]
c[i][j] ← &c[i][j][0]
```

row major ordering  
contiguous linear layout



# Virtual assignments

```
int c [2][3][4];
```



```
c [i][j][k];
```

## virtual assignments

```
c ← &c[0][0][0]  
c[i] ← &c[i][0][0]  
c[i][j] ← &c[i][j][0]
```

row major ordering  
contiguous linear layout

```
c ← (int (*)[3][4]) &c[0][0][0]  
c[i] ← (int (*)[4]) &c[i][0][0]  
c[i][j] ← (int *) &c[i][j][0]
```

type casts      address values

if c, c[i], c[i][j] were real pointer variables,  
type casts would be needed



# Types of `c[i]` and `c[i][j]`

```
int c [2][3][4];
```

```
&c[i][j][0] = c[i][j]
&c[i][0]    = c[i]
&c[0]      = c
```

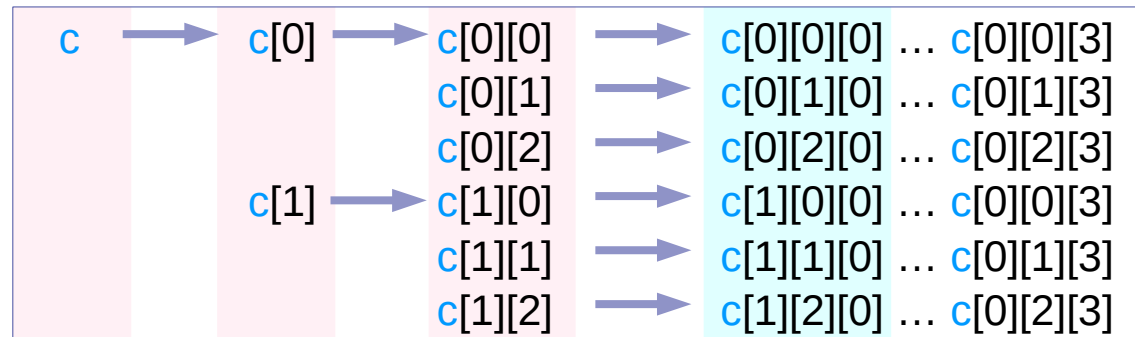
with contiguous subarrays

```
&c[i][j][k] = c[i][j]+k for all k
&c[i][j]    = c[i]+j   for all j
&c[i]       = c+i     for all i
```

```
c [i][j][k]
```

`c[i]` virtual 1-d array pointer of the type `int (*) [4]`  
`c[i][j]` : the name of 1-d array with 4 integers `int [4]`

`c[i][j]` (virtual 0-d array) pointer of the type `int (*)`  
`c[i][j][k]` : an element of a 4-integer array `int`



```
int [2] [3][4]  int [3] [4]      int [4]
int (*) [3][4]  int (*) [4]      int (*)
int             ...             int
int             ...             int
```

pointers to a 2-d array    pointers to a 1-d array    1-d array names    leading element of 4-integer array

# Values of $c$ , $c[i]$ , $c[i][j]$

```
int c [2][3][4];
```

```
&c[i][j][0] = c[i][j]  
&c[i][0] = c[i]  
&c[0] = c
```

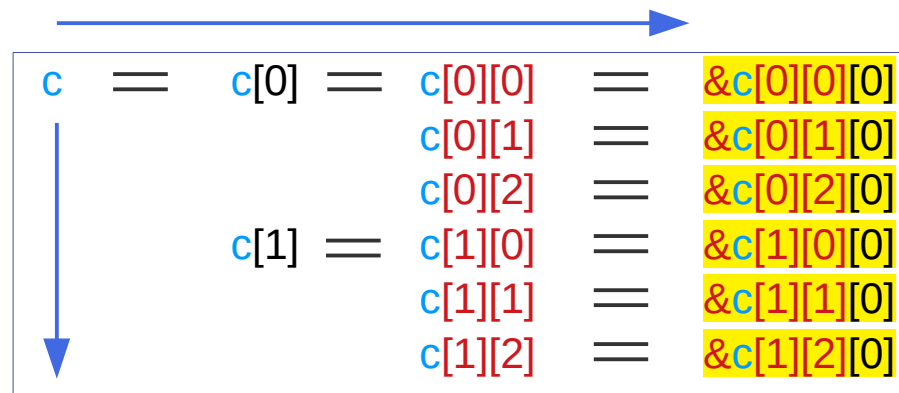
with contiguous subarrays

```
&c[i][j][k] = c[i][j]+k for all k  
&c[i][j] = c[i]+j for all j  
&c[i] = c+i for all i
```

```
c [i][j][k]
```

**virtual** array pointers

in each row in the following figure  
have the same address value



**Horizontal displacements** are not counted  
only **vertical displacements** are considered  
for address values

**virtual assignments**

```
c[i][j] = &c[i][j][0]  
c[i] = &c[i][0][0]  
c = &c[0][0][0]
```

# Finding address values of $c$ , $c[i]$ , $c[i][j]$

```
int c [2][3][4];
```

```
&c[i][j][0] = c[i][j]
&c[i][0]    = c[i]
&c[0]       = c
```

with contiguous subarrays

```
&c[i][j][k] = c[i][j]+k for all k
&c[i][j]    = c[i]+j   for all j
&c[i]       = c+i     for all i
```

```
c [i][j][k]
```

virtual assignments

```
c[i][j] = &c[i][j][0]
c[i]    = &c[i][0][0]
c       = &c[0][0][0]
```

append  $[0]$  to the right

$c$	$\xrightarrow{+[0]}$	$c[0]$	$\xrightarrow{+[0]}$	$c[0][0]$	$\xrightarrow{+[0]}$	$\&c[0][0][0]$
				$c[0][1]$	$\xrightarrow{+[0]}$	$\&c[0][1][0]$
				$c[0][2]$	$\xrightarrow{+[0]}$	$\&c[0][2][0]$
		$c[1]$	$\xrightarrow{+[0]}$	$c[1][0]$	$\xrightarrow{+[0]}$	$\&c[1][0][0]$
				$c[1][1]$	$\xrightarrow{+[0]}$	$\&c[1][1][0]$
				$c[1][2]$	$\xrightarrow{+[0]}$	$\&c[1][2][0]$

$\text{int (*) [3][4]}$

$\text{int (*) [4]}$

$\text{int [4]}$

$\text{int}$

$c[0][0][0]$  :  
leading  
elements  
of  $c$

$c[i][0][0]$  :  
leading  
elements  
of  $c[i]$

$c[i][j][0]$  :  
leading  
elements  
of  $c[i][j]$

$\&c[0][0][0]$

$\&c[0][0][0]$

$\&c[1][0][0]$

$\&c[0][0][0]$   
 $\&c[0][1][0]$   
 $\&c[0][2][0]$   
 $\&c[1][0][0]$   
 $\&c[1][1][0]$   
 $\&c[1][2][0]$

# Finding sub-array names with the address $\&c[i][j][0]$

```
int c [2][3][4];
```

```
&c[i][j][0] = c[i][j]
&c[i][0]    = c[i]
&c[0]       = c
```

with contiguous subarrays

```
&c[i][j][k] = c[i][j]+k for all k
&c[i][j]    = c[i]+j   for all j
&c[i]       = c+i     for all i
```

```
c [i][j][k]
```

virtual assignments

```
c[i][j] = &c[i][j][0]
c[i]    = &c[i][0][0]
c       = &c[0][0][0]
```

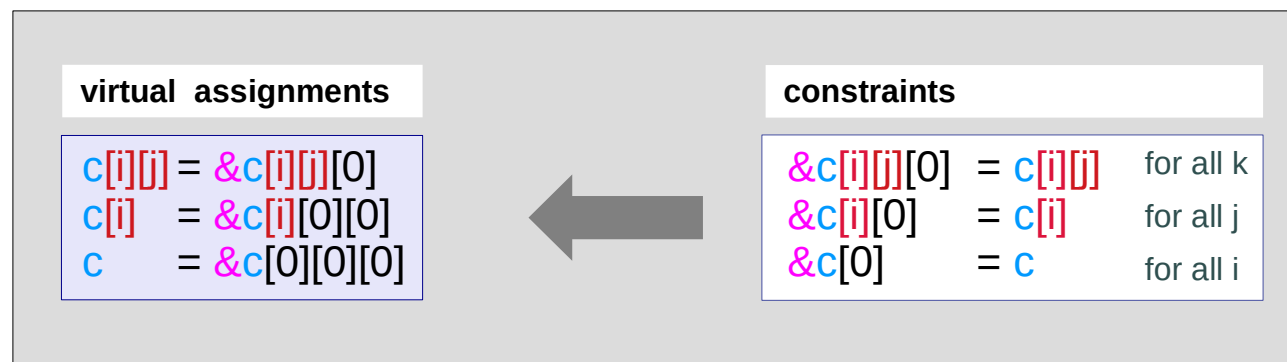
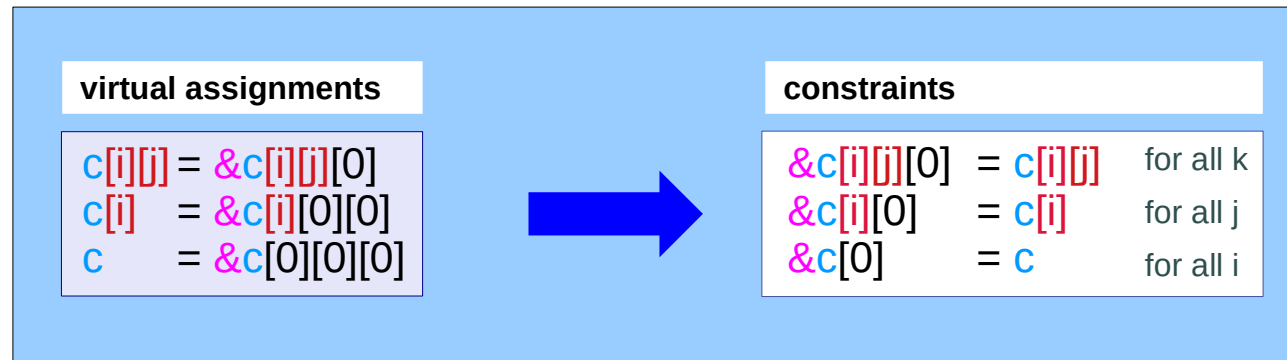
delete [0] from the right

$\&c[0][0][0]$	$\underline{\underline{-[0]}}$	$c[0][0]$	$\underline{\underline{-[0]}}$	$c[0]$	$\underline{\underline{-[0]}}$	$c$
$\&c[0][1][0]$	$\underline{\underline{-[0]}}$	$c[0][1]$				
$\&c[0][2][0]$	$\underline{\underline{-[0]}}$	$c[0][2]$				
$\&c[1][0][0]$	$\underline{\underline{-[0]}}$	$c[1][0]$	$\underline{\underline{-[0]}}$	$c[1]$		
$\&c[1][1][0]$	$\underline{\underline{-[0]}}$	$c[1][1]$				
$\&c[1][2][0]$	$\underline{\underline{-[0]}}$	$c[1][2]$				

int                      int [4]                      int (\*) [4]                      int (\*) [3][4]

$c[0][0][0]$  is the leading element of  $c[0][0]$ ,  $c[0]$ ,  $c$   
 $c[0][1][0]$  is the leading element of  $c[0][1]$   
 $c[0][2][0]$  is the leading element of  $c[0][2]$   
 $c[1][0][0]$  is the leading element of  $c[1][0]$ ,  $c[1]$   
 $c[1][1][0]$  is the leading element of  $c[1][1]$   
 $c[1][2][0]$  is the leading element of  $c[1][2]$

## multi-dimensional arrays



# Pointer reference relationship

```
int  c [2][3][4];
```

```
&c[i][j][0] = c[i][j]
&c[i][0]    = c[i]
&c[0]       = c
```

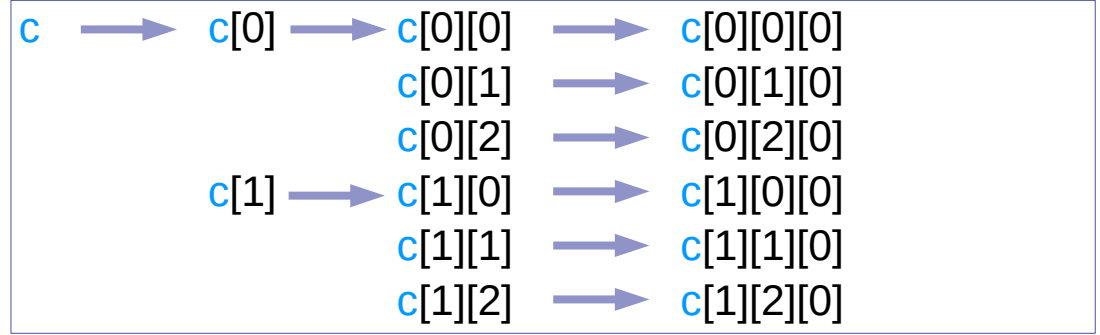
with contiguous subarrays

```
&c[i][j][k] = c[i][j]+k for all k
&c[i][j]    = c[i]+j   for all j
&c[i]       = c+i      for all i
```



virtual assignments

```
c[i][j] = &c[i][j][0]
c[i]    = &c[i][0][0]
c       = &c[0][0][0]
```



$c[i][j]$	=	$\&c[i][j][0]$
$c[0][0]$	=	$\&c[0][0][0]$
$c[0][1]$	=	$\&c[0][1][0]$
$c[0][2]$	=	$\&c[0][2][0]$
$c[1][0]$	=	$\&c[1][0][0]$
$c[1][1]$	=	$\&c[1][1][0]$
$c[1][2]$	=	$\&c[1][2][0]$

$c[i]$	=	$\&c[i][0]$
$c[0]$	=	$\&c[0][0]$ = $\&c[0][0][0]$
$c[1]$	=	$\&c[1][0]$ = $\&c[1][0][0]$

$c$	=	$\&c[0]$
$c$	=	$\&c[0]$ = $\&c[0][0][0]$

$c$	=	$\&c[0]$	=	$\&c[0][0][0]$
$c$	=	$\&c[0]$	=	$\&c[0][0][0]$

# General requirements for `c[i][j][k]`

```
int c [2][3][4];
```

```
&c[i][j][0] = c[i][j]
&c[i][0]    = c[i]
&c[0]      = c
```

with contiguous subarrays

```
&c[i][j][k] = c[i][j]+k for all k
&c[i][j]    = c[i]+j   for all j
&c[i]       = c+i      for all i
```



virtual assignments

```
c[i][j] = &c[i][j][0]
c[i]    = &c[i][0][0]
c       = &c[0][0][0]
```

`c[i][j]` virtual array pointer of the type `int (*)`  
`c[i][j][0]` : leading element of a 4-integer array `int`

```
*(c[0][0]+0) = c[0][0][0]
*(c[0][1]+0) = c[0][1][0]
*(c[0][2]+0) = c[0][2][0]
*(c[1][0]+0) = c[1][0][0]
*(c[1][1]+0) = c[1][1][0]
*(c[1][2]+0) = c[1][2][0]
```

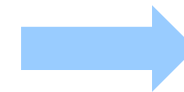
`c[0][0]` is the address of `c[0][0][0]`  
`c[0][1]` is the address of `c[0][1][0]`  
`c[0][2]` is the address of `c[0][2][0]`  
`c[1][0]` is the address of `c[1][0][0]`  
`c[1][1]` is the address of `c[1][1][0]`  
`c[1][2]` is the address of `c[1][2][0]`

`c[i]` virtual array pointer of the type `int (*) [4]`  
`c[i][j]` : a 4-element 1-d array name `int [4]`

```
*(c[0]+0) = c[0][0]
*(c[1]+0) = c[1][0]
```

`c[0]` is the address of `c[0][0]`  
`c[1]` is the address of `c[1][0]`

```
c[i][j] = &c[i][j][0]
c[i]    = &c[i][0][0]
c       = &c[0][0][0]
```



```
&c[i][j][0] = c[i][j]
&c[i][0]    = c[i]
&c[0]      = c
```

# General requirements for `c[i][j][k]`

```
int c [2][3][4];
```

```
&c[i][j][0] = c[i][j]
&c[i][0]    = c[i]
&c[0]      = c
```

with contiguous subarrays

```
&c[i][j][k] = c[i][j]+k for all k
&c[i][j]    = c[i]+j    for all j
&c[i]       = c+i       for all i
```



virtual assignments

```
c[i][j] = &c[i][j][0]
c[i]    = &c[i][0][0]
c       = &c[0][0][0]
```

```
int (*)      c[i][j] = (int (*)) &c[i][j][0]
int (*) [4]  c[i]    = (int (*) [4]) &c[i][0][0]
int (*) [3][4] c      = (int (*) [3][4]) &c[0][0][0]
```

Pointer  
Types

```
int [4]      c[i][j] size = 4*4
int [3][4]   c[i]    size = 3*4*4
int [2][3][4] c      size = 2*3*4*4
```

```
c[i][j][0] stride = 4*4
c[i][0][0] stride = 3*4*4
c[0][0][0] stride = 2*3*4*4
```

Pointer  
Types

```
c[i][j][k] 4 integers
c[i][j][k] 3*4 integers
c[i][j][k] 2*3*4 integers
```



# General requirements for `c[i][j][k]`

```
int c [2][3][4];
```

```
&c[i][j][0] = c[i][j]
&c[i][0]    = c[i]
&c[0]      = c
```

with contiguous subarrays

```
&c[i][j][k] = c[i][j]+k for all k
&c[i][j]    = c[i]+j   for all j
&c[i]       = c+i      for all i
```



virtual assignments

```
c[i][j] = &c[i][j][0]
c[i]    = &c[i][0][0]
c       = &c[0][0][0]
```

```
int [4]
int [3][4]
int [2][3][4]
```

```
c[i][j] size = 4*4
c[i]    size = 3*4*4
c       size = 2*3*4*4
```

Pointer  
Types

```
c[i][j][0] stride = 4*4
c[i][0][0] stride = 3*4*4
c[0][0][0] stride = 2*3*4*4
```

```
c[i][j][k] k=0..3 4 integers
c[i][j][k] j=0..2, k=0..3 3*4 integers
c[i][j][k] i=0..1, j=0..2, k=0..3 2*3*4 integers
```

```
c[i][j]+1 size = 4*4 4 integers away
c[i]+1    size = 3*4*4 3*4 integers away
c+1       size = 2*3*4*4 2*3*4 integers away
```

# General requirements for $c[i][j][k]$

```
int c [2][3][4];
```

```
&c[i][j][0] = c[i][j]
&c[i][0]    = c[i]
&c[0]      = c
```

with contiguous subarrays

```
&c[i][j][k] = c[i][j]+k for all k
&c[i][j]    = c[i]+j   for all j
&c[i]       = c+i      for all i
```



virtual assignments

```
c[i][j] = &c[i][j][0]
c[i]    = &c[i][0][0]
c       = &c[0][0][0]
```

int	$\&c[i][j][0]$	=	$c[i][j]$	int (*)
int [4]	$\&c[i][0]$	=	$c[i]$	int (*) [4]
int [3][4]	$\&c[0]$	=	$c$	int (*) [3][4]
	Abstract Data Types		Pointer Types	

int (*)	$c[i][j]$	=	$\&c[i][j][0]$	int
int (*) [4]	$c[i]$	=	$\&c[i][0]$	int [4]
int (*) [3][4]	$c$	=	$\&c[0]$	int [3][4]
	Pointer Types		Abstract Data Types	

int (*)	$c[i][j]$	=	(int *)	$\&c[i][j][0]$
int (*) [4]	$c[i]$	=	(int (*) [4])	$\&c[i][0][0]$
int (*) [3][4]	$c$	=	(int (*) [3][4])	$\&c[0][0][0]$
	Pointer Types			

int [4]	$c[i][j]$	size = $4*4$	$c[i][j][0]$	stride = $4*4$
int [3][4]	$c[i]$	size = $3*4*4$	$c[i][0][0]$	stride = $3*4*4$
int [2][3][4]	$c$	size = $2*3*4*4$	$c[0][0][0]$	stride = $2*3*4*4$
	Pointer Types			

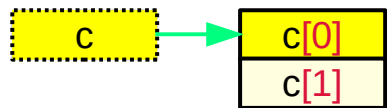
# Virtual array pointer c, c[0], c[0][0] – types and sizes

## Types – array pointers

**int (\*) [3][4]**    **int [3][4]**

array pointer

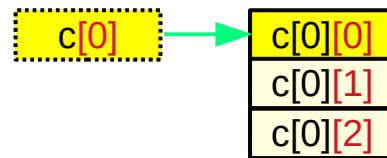
array (abstract data)



**int (\*) [4]**    **int [4]**

array pointer

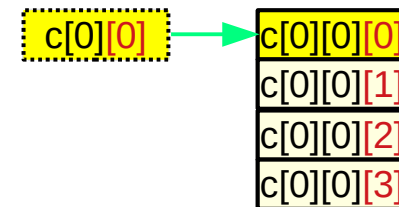
array (abstract data)



**int [4]**    **int**

array pointer

array (abstract data)



## Sizes – abstract data

sizeof(**c**)  
sizeof(**int [2][3][4]**)  
sizeof(**int**) \* 2 \* 3 \* 4

sizeof(**c[0]**)  
sizeof(**int [3][4]**)  
sizeof(**int**) \* 3 \* 4

sizeof(**c[0][0]**)  
sizeof(**int [4]**)  
sizeof(**int**) \* 4

sizeof(**int [2][3][4]**) = 96  
sizeof(**int (\*)[3][4]**) = 4 / 8

sizeof(**int [3][4]**) = 48  
sizeof(**int (\*)[4]**) = 4 / 8

sizeof(**int [4]**) = 16  
sizeof(**int (\*)**) = 4 / 8

# General requirements for `c[i][j][k]`

```
int c [2][3][4];
```

```
&c[i][j][0] = c[i][j]
&c[i][0]    = c[i]
&c[0]      = c
```

with contiguous subarrays

```
&c[i][j][k] = c[i][j]+k for all k
&c[i][j]    = c[i]+j   for all j
&c[i]       = c+i     for all i
```



virtual assignments

```
c[i][j] = &c[i][j][0]
c[i]    = &c[i][0][0]
c       = &c[0][0][0]
```

`int (*) [3][4]`

array pointer



`int (*) [3][4]`

array pointer



`int [3][4]`

array (abstract data)



2 `c[i]`'s

Contiguous  $2 \times 3 \times 4$  integers

`int (*) [4]`

array pointer



`int [4]`

array (abstract data)



3 `c[0][j]`'s

Contiguous  $3 \times 4$  integers

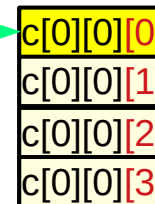
`int (*)`

array pointer



`int`

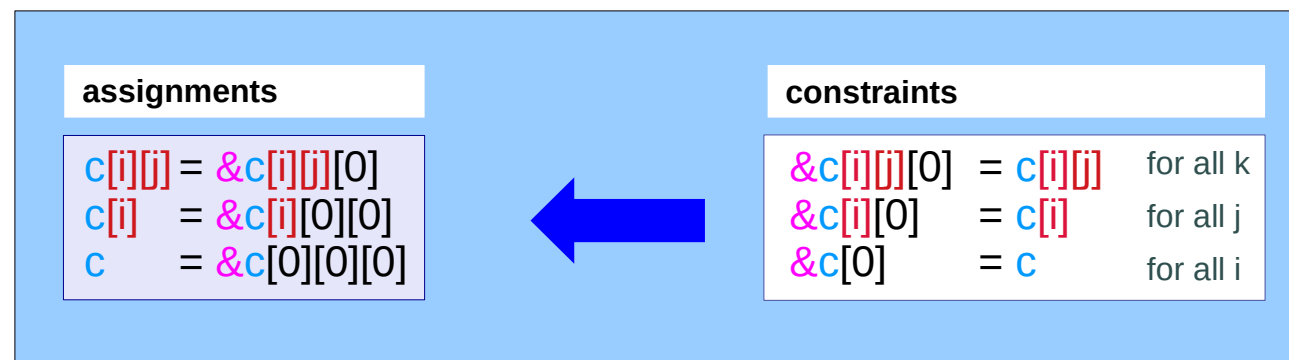
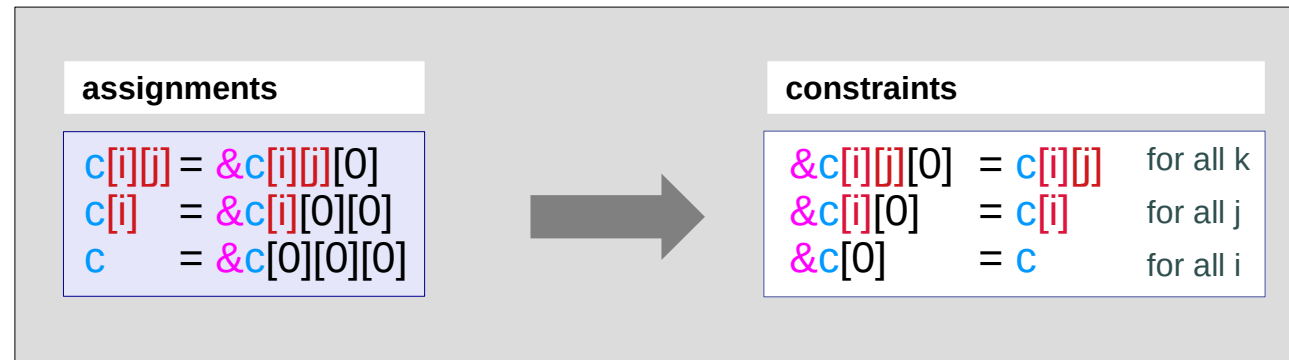
array (abstract data)



4 `c[0][0][k]`'s

Contiguous 4 integers

## multi-dimensional arrays



# c[0] = c[0][0] relation

```
int c [2][3][4];
```

```
&c[i][j][0] = c[i][j]
&c[i][0]    = c[i]
&c[0]      = c
```

with contiguous subarrays

```
&c[i][j][k] = c[i][j]+k for all k
&c[i][j]    = c[i]+j    for all j
&c[i]       = c+i       for all i
```



virtual assignments

```
c[i][j] = &c[i][j][0]
c[i]    = &c[i][0][0]
c       = &c[0][0][0]
```

```
c == c[0] == c[0][0] == &c[0][0][0]
```

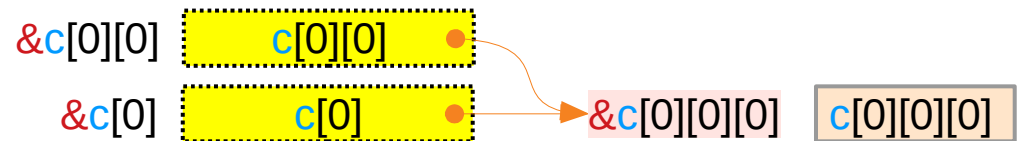
```
value(c[0]) = &c[0][0][0]
```

```
value(c[0][0]) = &c[0][0][0]
```

```
type(c[0]) = int (*)[4]
```

```
type(c[0][0]) = int [4]
```

```
c[0] = c[0][0] means
value(c[0]) = value(c[0][0])
```



```
c[i][j] = &c[i][j][0]
c[i]    = &c[i][0][0]
c       = &c[0][0][0]
```

# Addresses and Values of `c[0]` and `c[0][0]`

```
int c [2][3][4];
```

```
&c[i][j][0] = c[i][j]
&c[i][0]    = c[i]
&c[0]       = c
```

with contiguous subarrays

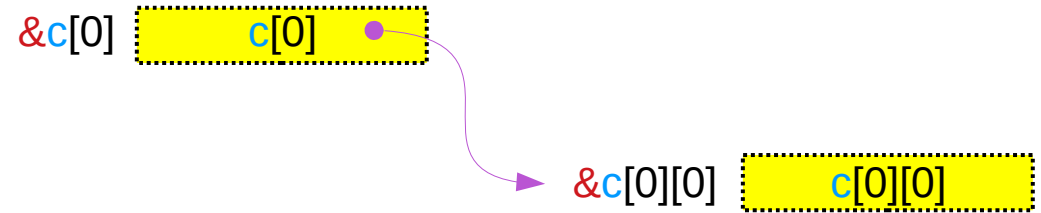
```
&c[i][j][k] = c[i][j]+k for all k
&c[i][j]    = c[i]+j   for all j
&c[i]       = c+i      for all i
```



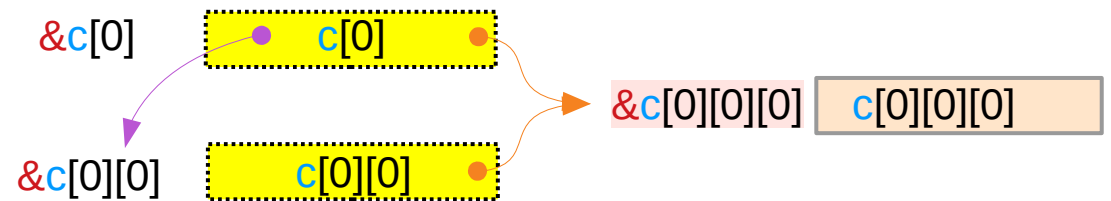
virtual assignments

```
c[i][j] = &c[i][j][0]
c[i]    = &c[i][0][0]
c       = &c[0][0][0]
```

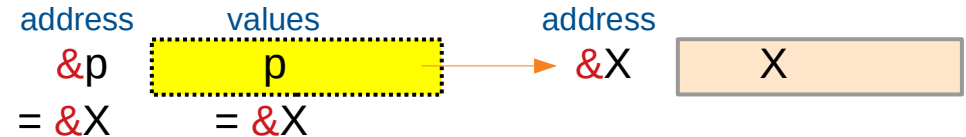
`c` → `c[0]` → `c[0][0]` = `&c[0][0][0]`



`c` = `c[0]` = `c[0][0]` = `&c[0][0][0]`



A virtual pointer's address and value are the same



# c[0] and c[0][0] point to the same c[i][0][0]

```
int c [2][3][4];
```

```
&c[i][j][0] = c[i][j]
&c[i][0]    = c[i]
&c[0]      = c
```

with contiguous subarrays

```
&c[i][j][k] = c[i][j]+k for all k
&c[i][j]    = c[i]+j   for all j
&c[i]       = c+i      for all i
```

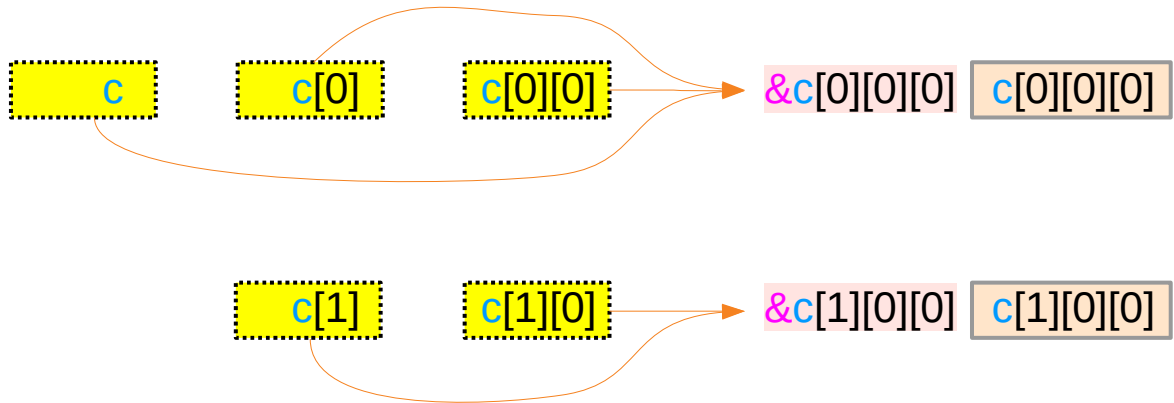


virtual assignments

```
c[i][j] = &c[i][j][0]
c[i]    = &c[i][0][0]
c       = &c[0][0][0]
```

```
c = c[0] = c[0][0] = &c[0][0][0]
int(*)[3][4] int(*)[4] int(*) int
```

```
c[1] = c[1][0] = &c[1][0][0]
int(*)[4] int(*) int
```



These virtual pointers have different types but the same value (address)



# &c[i][0] and &c[i][0][0] – equivalence relations

```
int c [2][3][4];
```

```
&c[i][j][0] = c[i][j]
&c[i][0]    = c[i]
&c[0]       = c
```

with contiguous subarrays

```
&c[i][j][k] = c[i][j]+k for all k
&c[i][j]    = c[i]+j   for all j
&c[i]       = c+i      for all i
```



virtual assignments

```
c[i][j] = &c[i][j][0]
c[i]    = &c[i][0][0]
c       = &c[0][0][0]
```

int(\*)[3][4] int(\*)[4] int(\*) int \*

```
c = c[0] = c[0][0] = &c[0][0][0]
&c &c[0] &c[0][0]
```

equivalences

```
c ≡ &c[0],
c[0] ≡ &c[0][0]
c[0][0] ≡ &c[0][0][0]
```

```
c[1] = c[1][0] = &c[1][0][0]
&c[1] &c[1][0]
```

equivalences

```
c[1] ≡ &c[1][0]
c[1][0] ≡ &c[1][0][0]
```

Horizontal displacements are not counted  
only vertical displacements are considered  
for address values

equivalences

```
c ≡ &c[0],
c[i] ≡ &c[i][0]
c[i][0] ≡ &c[i][0][0]
```

# $c[i] = \&c[i]$ and $c[i][0] = \&c[i][0]$

```
int c [2][3][4];
```

```
&c[i][j][0] = c[i][j]  
&c[i][0] = c[i]  
&c[0] = c
```

with contiguous subarrays

```
&c[i][j][k] = c[i][j]+k for all k  
&c[i][j] = c[i]+j for all j  
&c[i] = c+i for all i
```

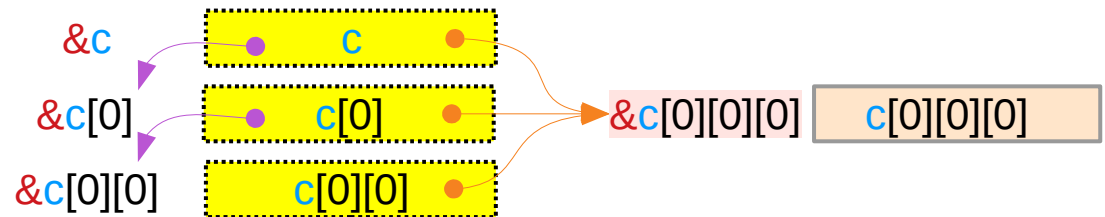


virtual assignments

```
c[i][j] = &c[i][j][0]  
c[i] = &c[i][0][0]  
c = &c[0][0][0]
```

```
c = c[0] = c[0][0] = &c[0][0][0]  
|| || ||  
&c = &c[0] = &c[0][0]
```

```
c[1] = c[1][0] = &c[1][0][0]  
|| ||  
&c[1] = &c[1][0]
```



# $c[i] = \&c[i]$ and $c[i][0] = \&c[i][0]$

```
int c [2][3][4];
```

```
&c[i][j][0] = c[i][j]  
&c[i][0] = c[i]  
&c[0] = c
```

with contiguous subarrays

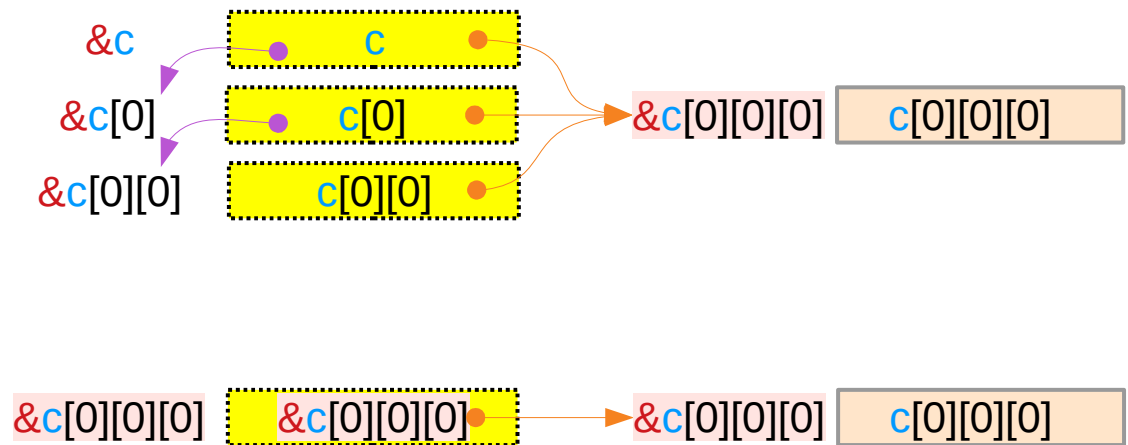
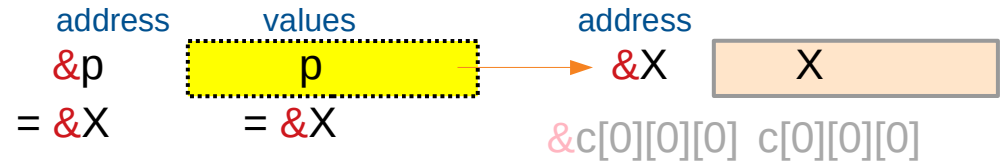
```
&c[i][j][k] = c[i][j]+k for all k  
&c[i][j] = c[i]+j for all j  
&c[i] = c+i for all i
```



virtual assignments

```
c[i][j] = &c[i][j][0]  
c[i] = &c[i][0][0]  
c = &c[0][0][0]
```

A virtual pointer's address and value are the same



**c[i]**

---

# Array Pointers to $c[i][0][0]$

$$\begin{aligned} &\&c[i][j][0] &= c[i][j] \\ &\&c[i][0] &= c[i] \\ &\&c[0] &= c \end{aligned}$$

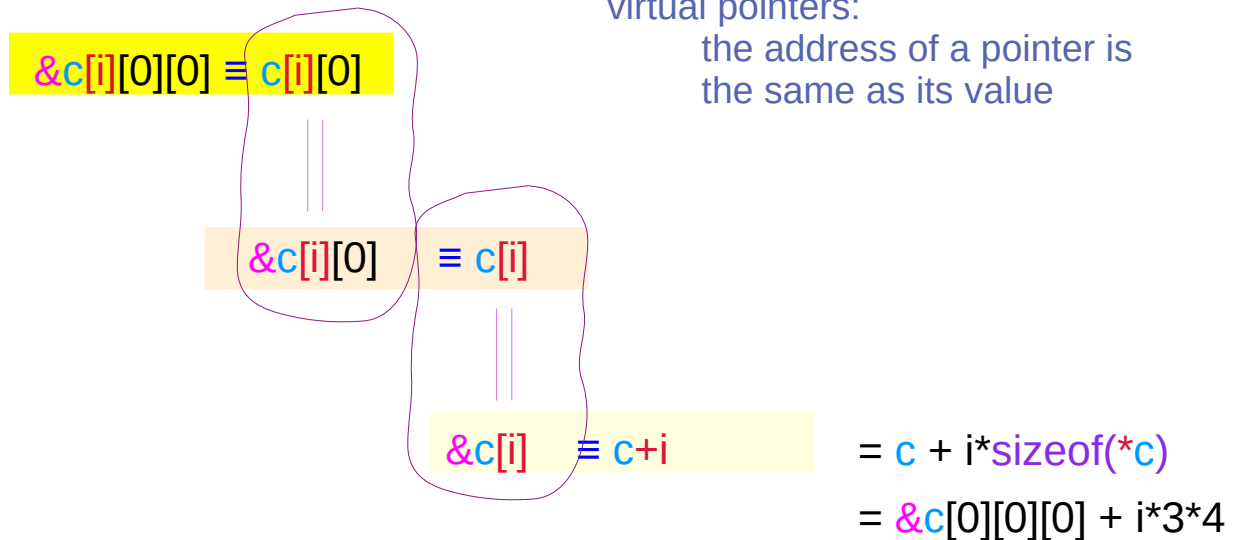
with contiguous subarrays

$$\begin{aligned} &\&c[i][j][k] &= c[i][j] + k && \text{for all } k \\ &\&c[i][j] &= c[i] + j && \text{for all } j \\ &\&c[i] &= c + i && \text{for all } i \end{aligned}$$



assignments

$$\begin{aligned} c[i][j] &= \&c[i][j][0] \\ c[i] &= \&c[i][0][0] \\ c &= \&c[0][0][0] \end{aligned}$$



delete [0] from the right

$$\begin{array}{l} \&c[0][0][0] \quad \underline{\underline{-[0]}} \quad c[0][0] \quad \underline{\underline{-[0]}} \quad c[0] \quad \underline{\underline{-[0]}} \quad c \\ \&c[1][0][0] \quad \underline{\underline{-[0]}} \quad c[1][0] \quad \underline{\underline{-[0]}} \quad c[1] \end{array}$$

# Array Pointers to $c[i][j][0]$

```
&c[i][j][0] = c[i][j]
&c[i][0]    = c[i]
&c[0]      = c
```

with contiguous subarrays

```
&c[i][j][k] = c[i][j] + k for all k
&c[i][j]   = c[i] + j   for all j
&c[i]      = c + i      for all i
```



assignments

```
c[i][j] = &c[i][j][0]
c[i]    = &c[i][0][0]
c       = &c[0][0][0]
```

$\&c[i][j][0] \equiv c[i][j]$

$\&c[i][j] \equiv c[i] + j$

$= c[i] + j * \text{sizeof}(*c[i])$   
 $= c + i * \text{sizeof}(*c) + j * 4$   
 $= \&c[0][0][0] + i * 3 * 4 + j * 4$

delete [0] from the right

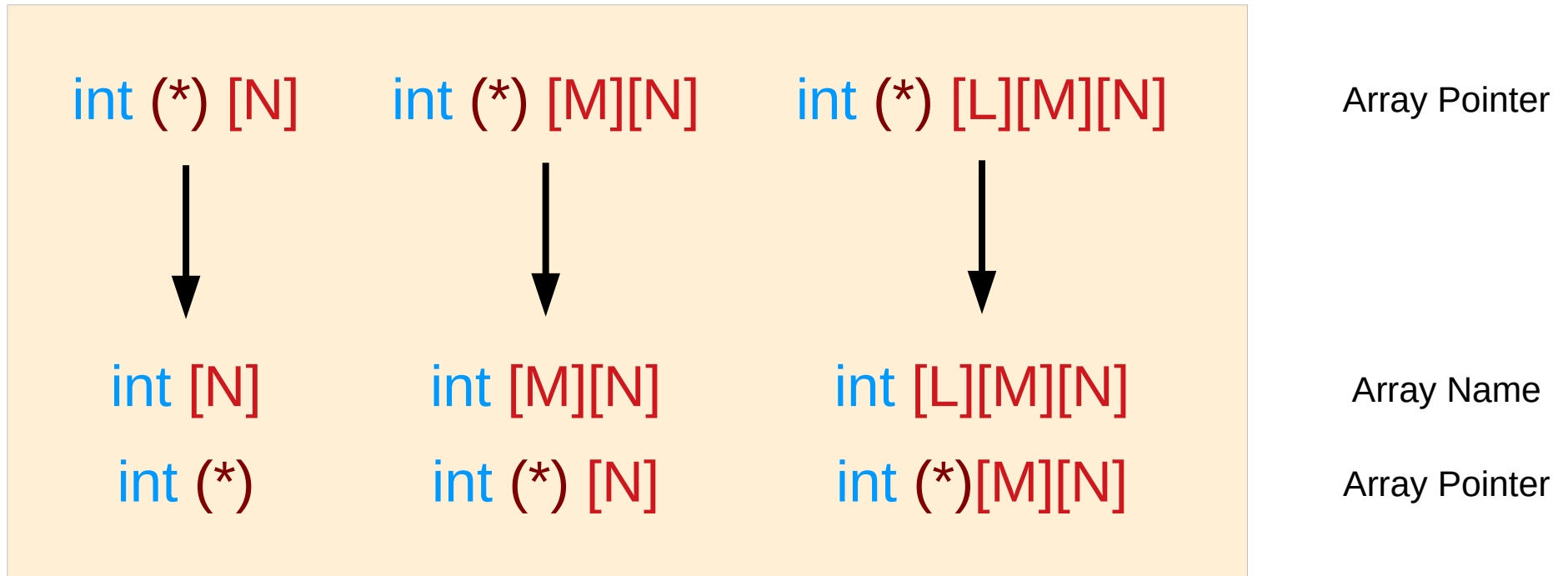
$\&c[0][0][0]$	$\underline{\underline{-[0]}}$	$c[0][0]$	$\underline{\underline{-[0]}}$	$c[0]$	$\underline{\underline{-[0]}}$	$c$
$\&c[0][1][0]$	$\underline{\underline{-[0]}}$	$c[0][1]$				
$\&c[0][2][0]$	$\underline{\underline{-[0]}}$	$c[0][2]$				
$\&c[1][0][0]$	$\underline{\underline{-[0]}}$	$c[1][0]$	$\underline{\underline{-[0]}}$	$c[1]$		
$\&c[1][1][0]$	$\underline{\underline{-[0]}}$	$c[1][1]$				
$\&c[1][2][0]$	$\underline{\underline{-[0]}}$	$c[1][2]$				

# Contiguity Constraints

c [i][j][k];

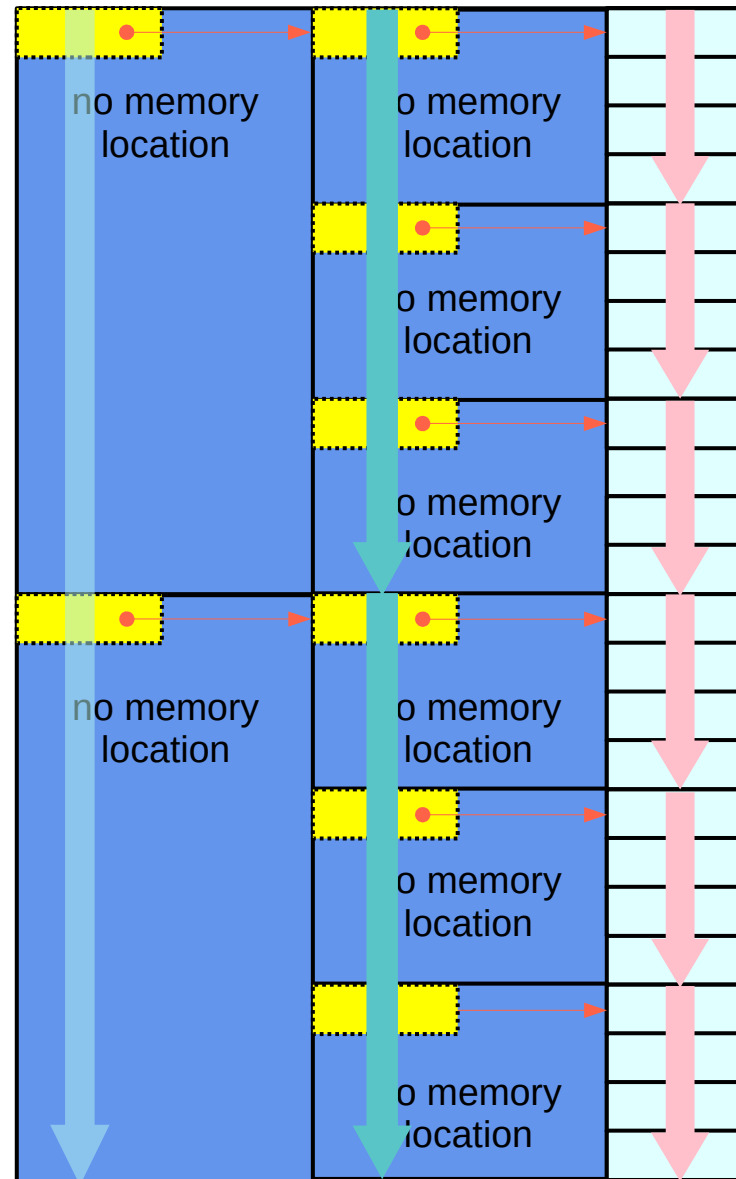
Virtual Array Pointers and Contiguity

# Using array pointers



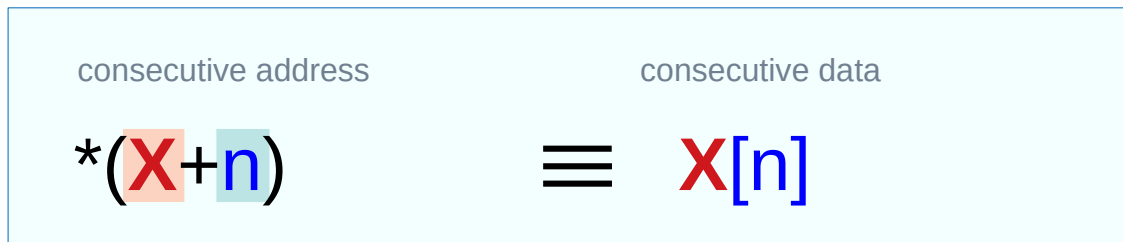


# Array pointer approach – contiguity constraints

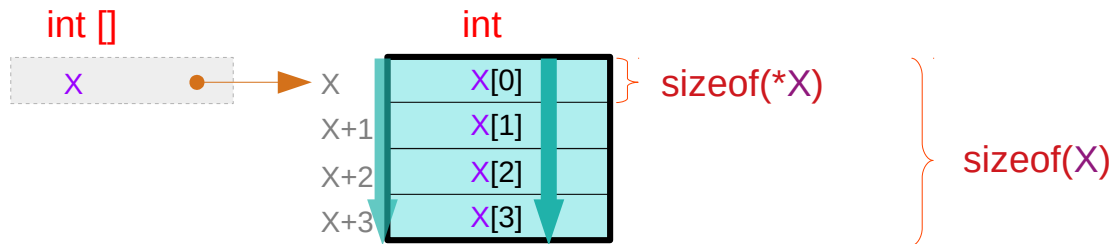


**Array Pointer Approach**  
(pointer to arrays)

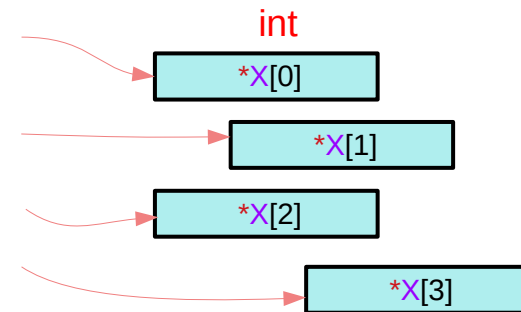
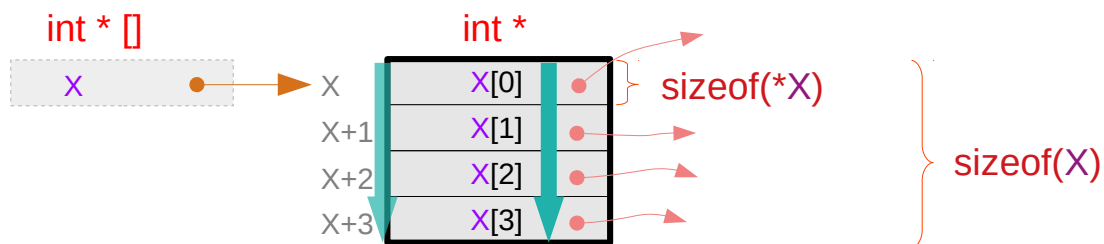
# Equivalence and contiguity (1)



contiguous index : n



`int X[4]`; contiguous `X[i]` for a given `X` : **primitive types**



`int * X[4]`; contiguous `X[i]` for a given `X` : **pointer types**

# Equivalence and contiguity (2)

consecutive address

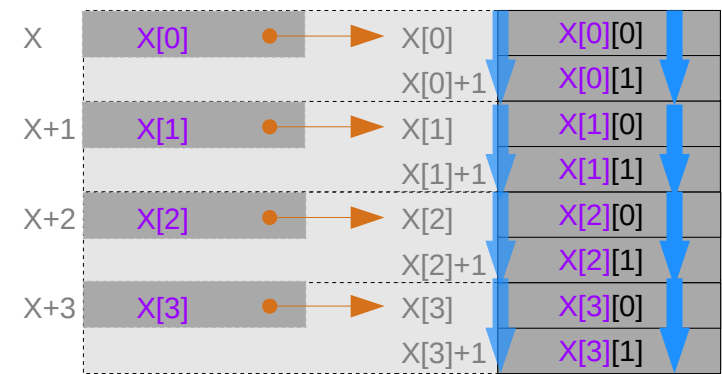
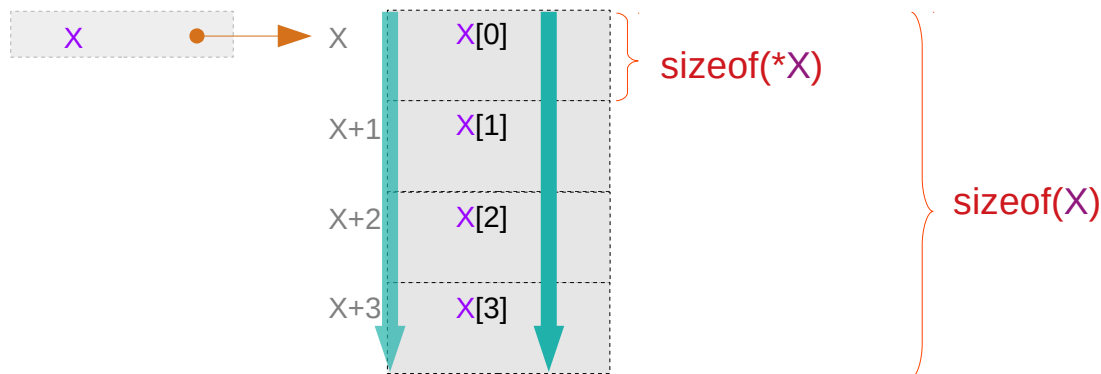
$*(X+n)$

consecutive data

$\equiv X[n]$

contiguous index : n

can be recursively applied



**atype \* X[4];** contiguous  $X[i]$  for a given  $X$  : **abstract data types**

# Recursive applications of equivalences

By definition, contiguous memory locations are assumed

consecutive address		consecutive data
$*(X+n)$	$\equiv$	$X[n]$

contiguous index : n

$*(p[m]+n)$	$\leftrightarrow$	$p[m][n]$
$(*(p+m))[n];$	$\leftrightarrow$	$p[m][n];$

$X = p[m]$  contiguous index : n

$X = p$  contiguous index : m

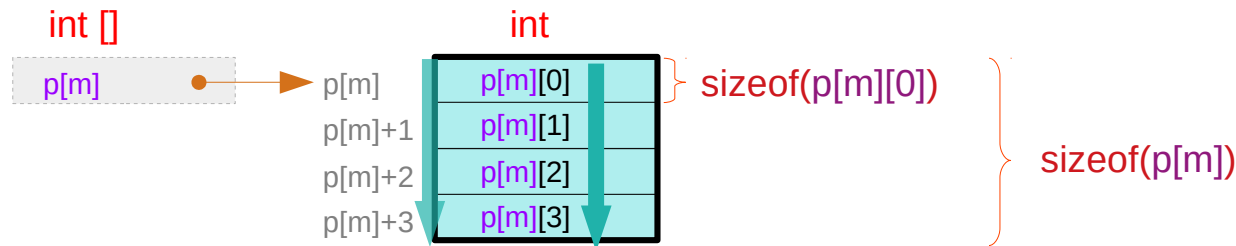
# Equivalence for a given p[m] (1)

$$*(p[m]+n) \iff p[m][n]$$

for a given  $p[m]$  contiguous index :  $n$

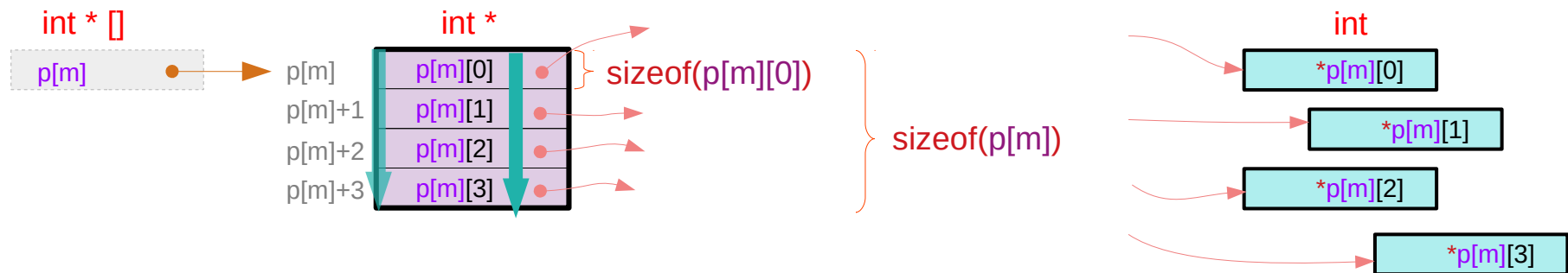
**int p[M][4];** contiguous p[m][n] for a given p[m] : **primitive types**

$m = 0, 1, \dots, M-1$



**int \* p[M][4];** contiguous p[m][n] for a given p[m] : **pointer types**

$m = 0, 1, \dots, M-1$

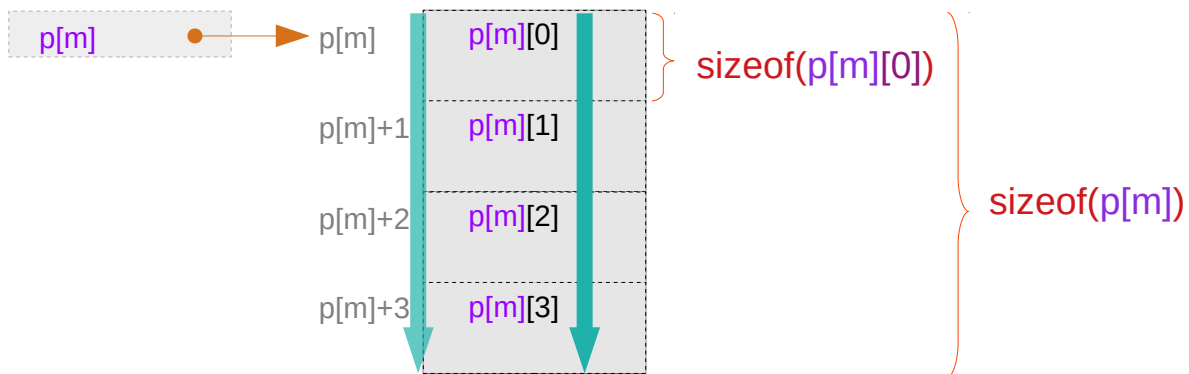


# Equivalence for a given p[m] (2)

$$*(p[m]+n) \iff p[m][n]$$

for a given `p[m]` contiguous index : `n`

`atype * p[M][4];` contiguous `p[m][n]` for a given `p[m]` : **abstract data types** `m = 0, 1, ..., M-1`



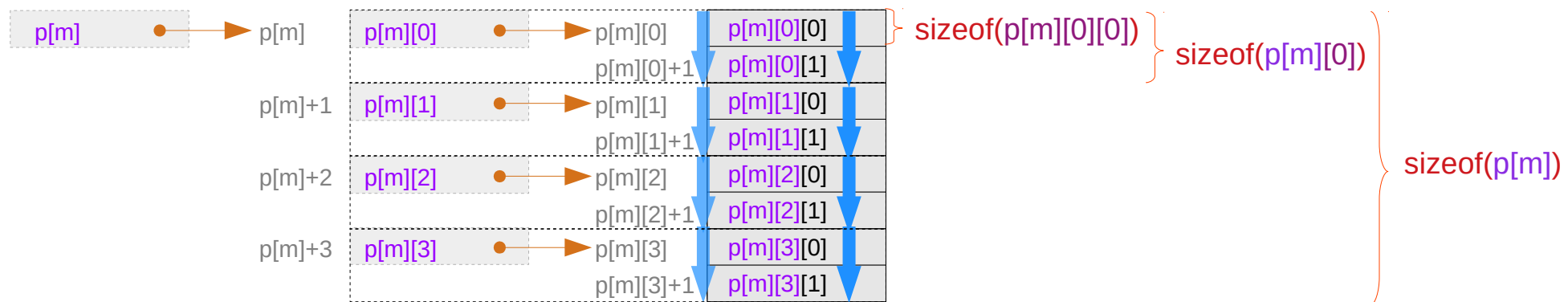
can be recursively applied

# Equivalence for a given p[m][n]

$$*(p[m][n]+k) \iff p[m][n][k]$$

for a given `p[m][n]` contiguous index : `k`

`atype * p[M][4][2];` contiguous `p[m][n][k]` for a given `p[m][n]` : **abstract data types** `m = 0, 1, ..., M-1`



# Contiguity constraints in multi-dimensional arrays

$$*(p[m]+n) \iff p[m][n]$$

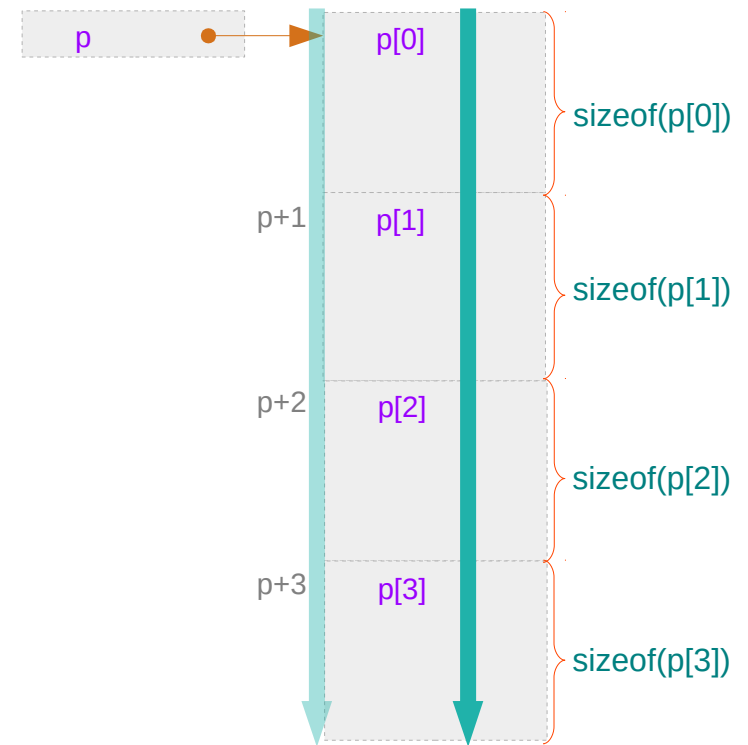
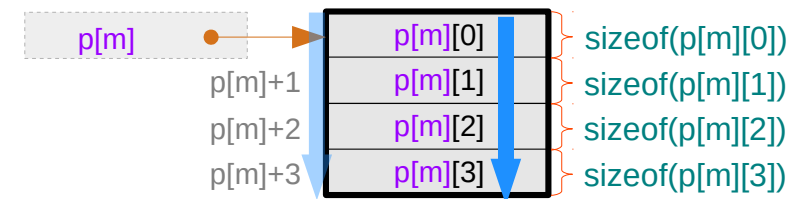
for a given  $p[m]$ , thus for a given  $p$  and  $m$ ,  
 $p[m][n]$ 's must be contiguous for all  $n$ .  
 $p[m][0], p[m][1], \dots, p[m][N-1]$

contiguous index :  $n$

$$*(p+m) \iff p[m]$$

for a given  $p$ ,  
 $p[m]$ 's must be contiguous for all  $m$ .  
 $p[0], p[1], \dots, p[M-1]$

contiguous index :  $m$

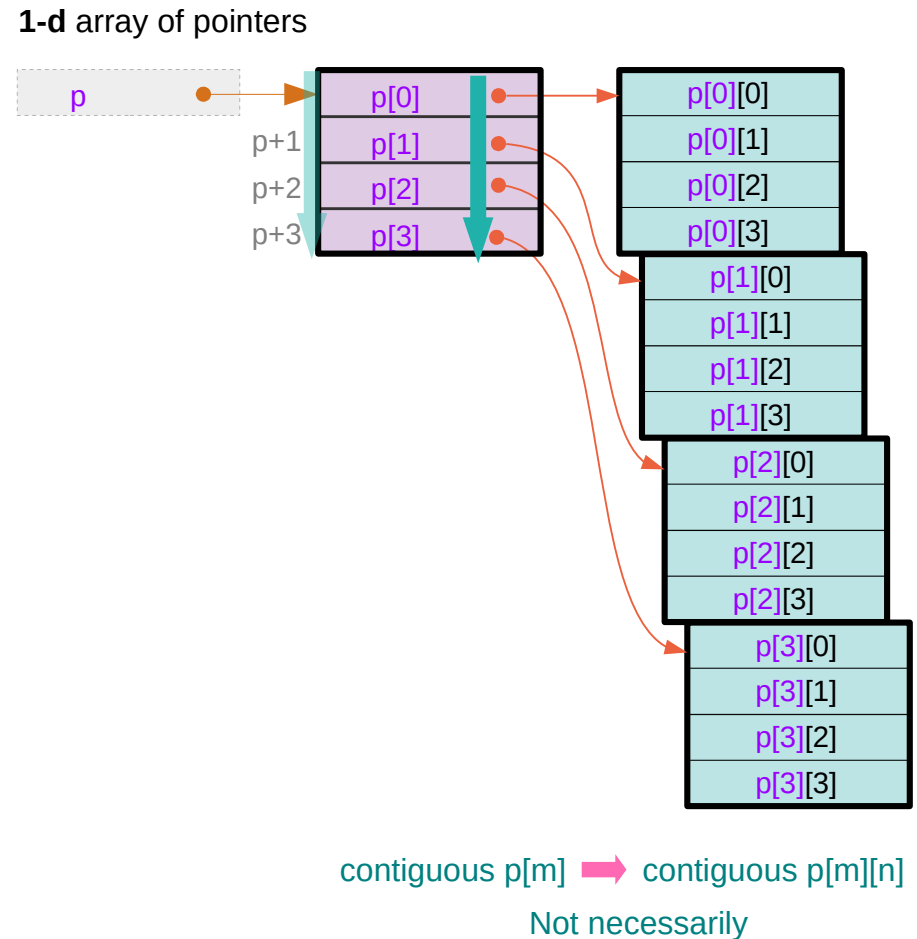
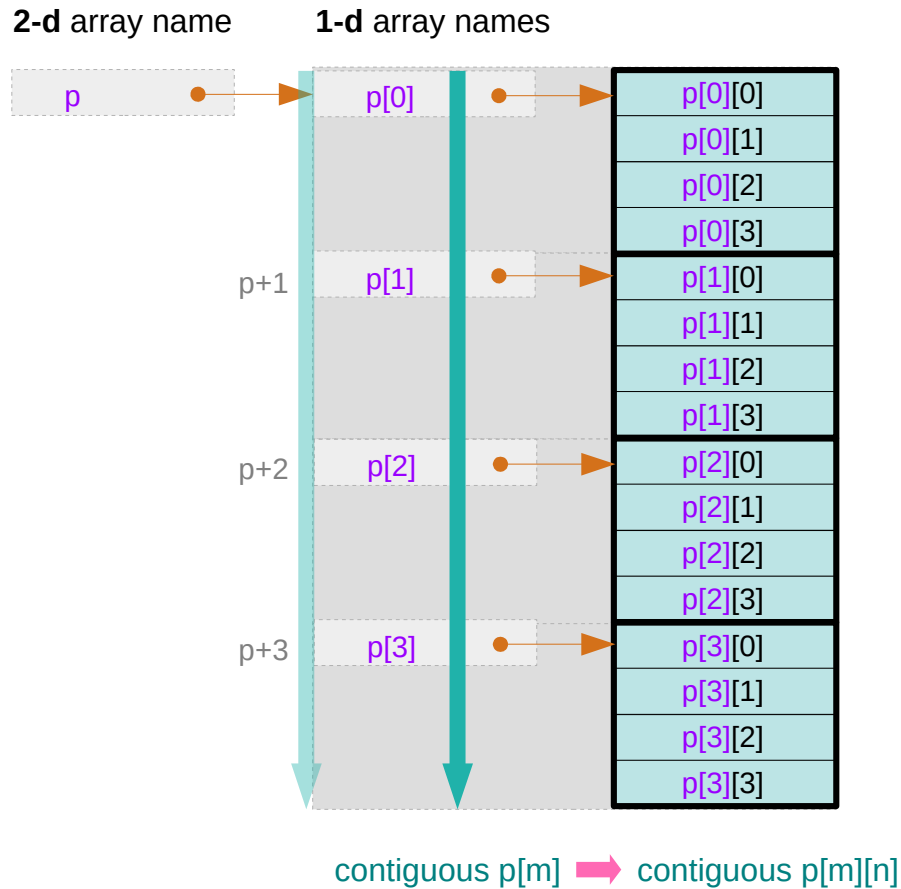




# Contiguity constraints for p

$$*(p+m) \iff p[m]$$

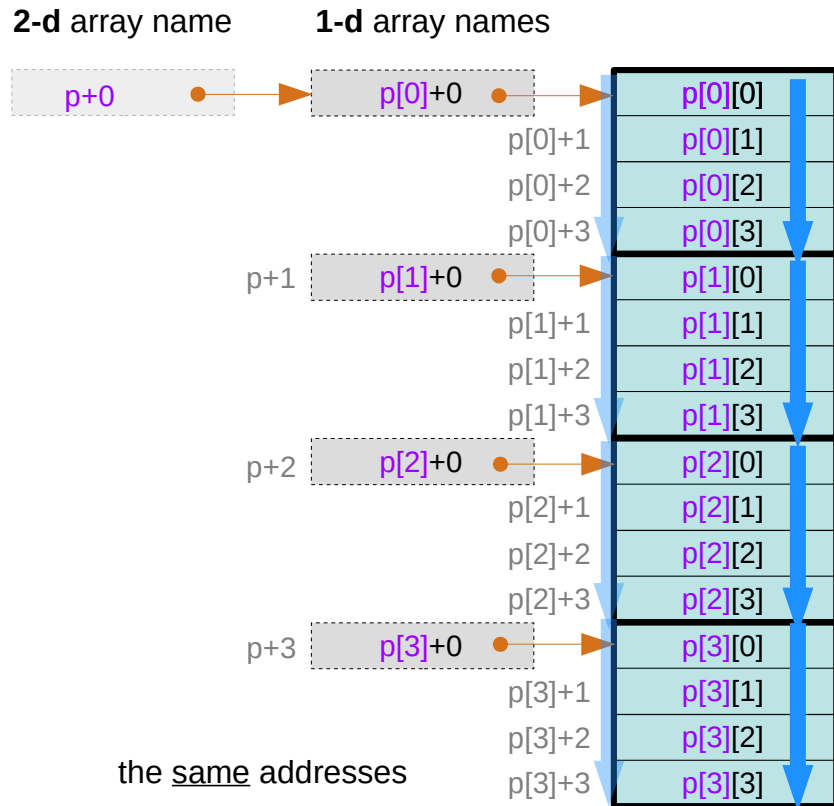
for a given  $p$  contiguous index :  $m$



# Contiguity constraints for p[m] – using array pointers

$$*(p[m]+n) \iff p[m][n]$$

for a given  $p[m]$  contiguous index :  $n$



$$p[0][0] = *(p[0]+0) \xrightarrow{\text{addr}} \underbrace{\&p[0][0] = p[0]}_{\text{addr}} \xrightarrow{\text{addr}} p+0$$

$$p[1][0] = *(p[1]+0) \xrightarrow{\text{addr}} \underbrace{\&p[1][0] = p[1]}_{\text{addr}} \xrightarrow{\text{addr}} p+1$$

$$p[2][0] = *(p[2]+0) \xrightarrow{\text{addr}} \underbrace{\&p[2][0] = p[2]}_{\text{addr}} \xrightarrow{\text{addr}} p+2$$

$$p[3][0] = *(p[3]+0) \xrightarrow{\text{addr}} \underbrace{\&p[3][0] = p[3]}_{\text{addr}} \xrightarrow{\text{addr}} p+3$$

the same addresses

contiguous  $p[m]$   $\rightarrow$  contiguous  $p[m][n]$

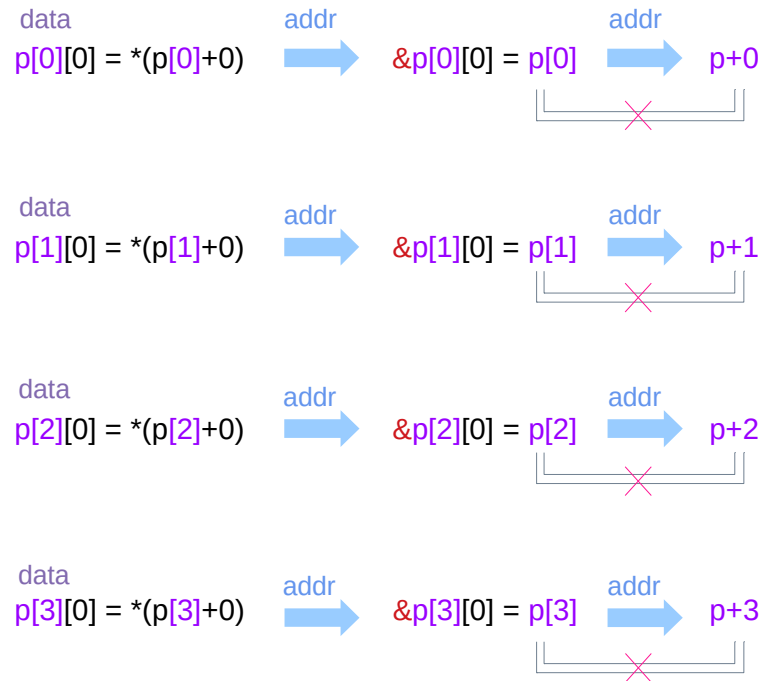
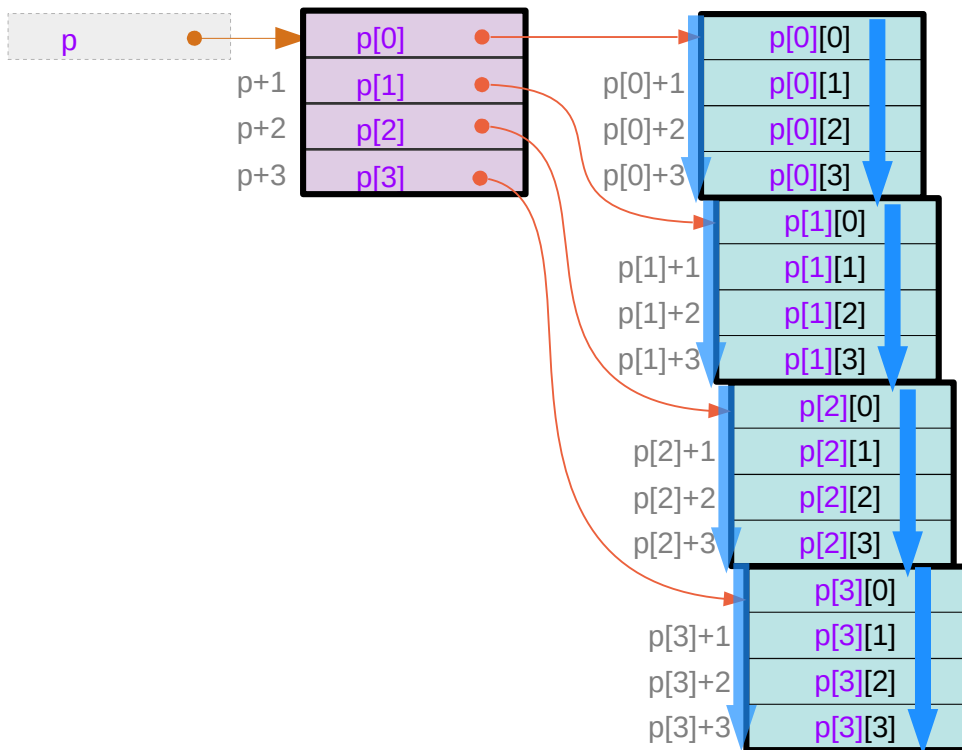
virtual array pointer  $\iff$  no real memory locations

# Contiguity constraints for p[m] – using pointer arrays

$$*(p[m]+n) \iff p[m][n]$$

for a given  $p[m]$  contiguous index :  $n$

1-d array of pointers



the different addresses

contiguous  $p[m]$   $\rightarrow$  contiguous  $p[m][n]$   
Not necessarily

# Contiguity constraints for 2-d arrays

```
int a[M][N] ;
```

$*(a+m) \leftrightarrow a[m]$

$a[0], a[1], \dots, a[M-1]$   
are contiguous

$*(a[m]+n) \leftrightarrow a[m][n]$

$a[m][0], a[m][1], \dots, a[m][N-1]$   
are contiguous

```
int (*b)[N] ;
```

$*(b+m) \leftrightarrow b[m]$

$b[0], b[1], \dots, b[M-1]$   
are contiguous

$*(b[m]+n) \leftrightarrow b[m][n]$

$b[m][0], b[m][1], \dots, b[m][N-1]$   
are contiguous

```
int * c[M] ;
```

$*(c+m) \leftrightarrow c[m]$

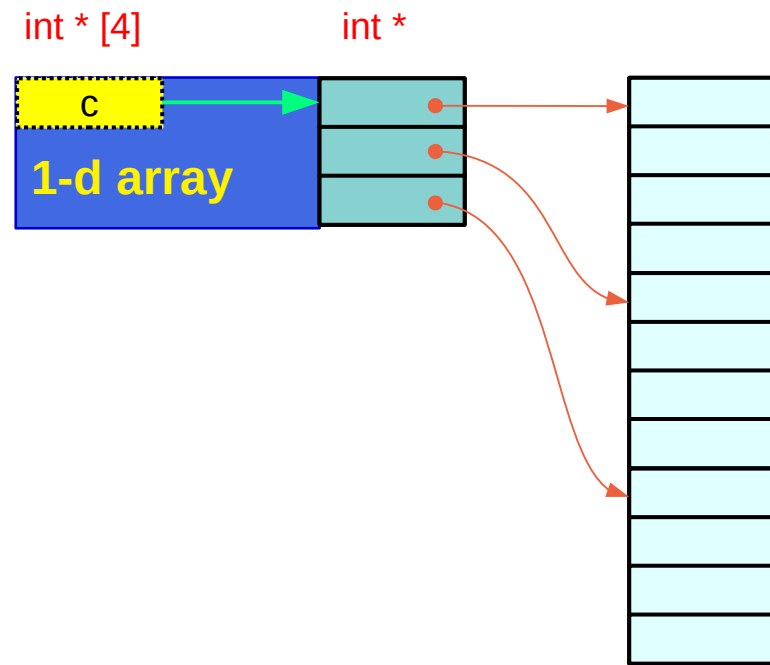
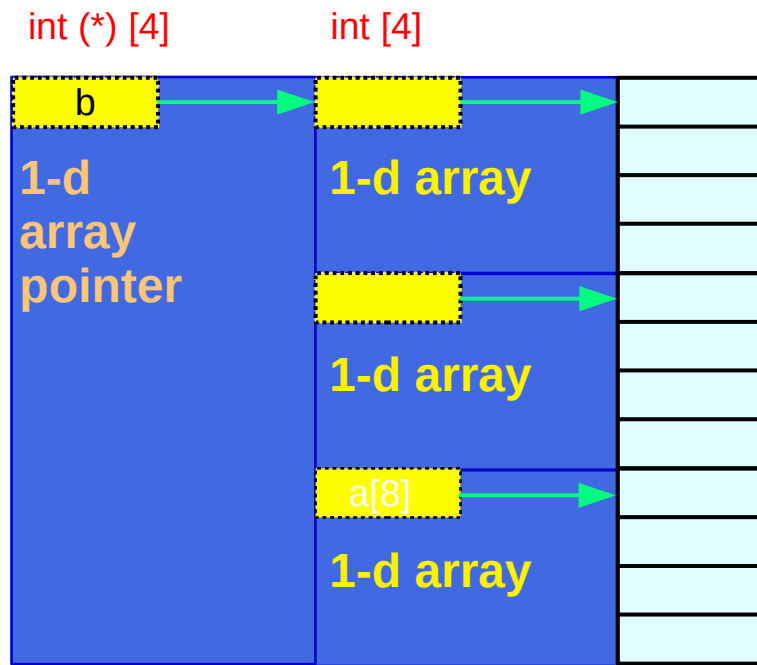
$c[0], c[1], \dots, c[M-1]$   
are contiguous

$*(c[m]+n) \leftrightarrow c[m][n]$

$c[m][0], c[m][1], \dots, c[m][N-1]$   
are contiguous

a set of assignments of pointers  
are necessary for this contiguity

# Pointer Arrays vs Array Pointers



`int (*b)[N] ;`

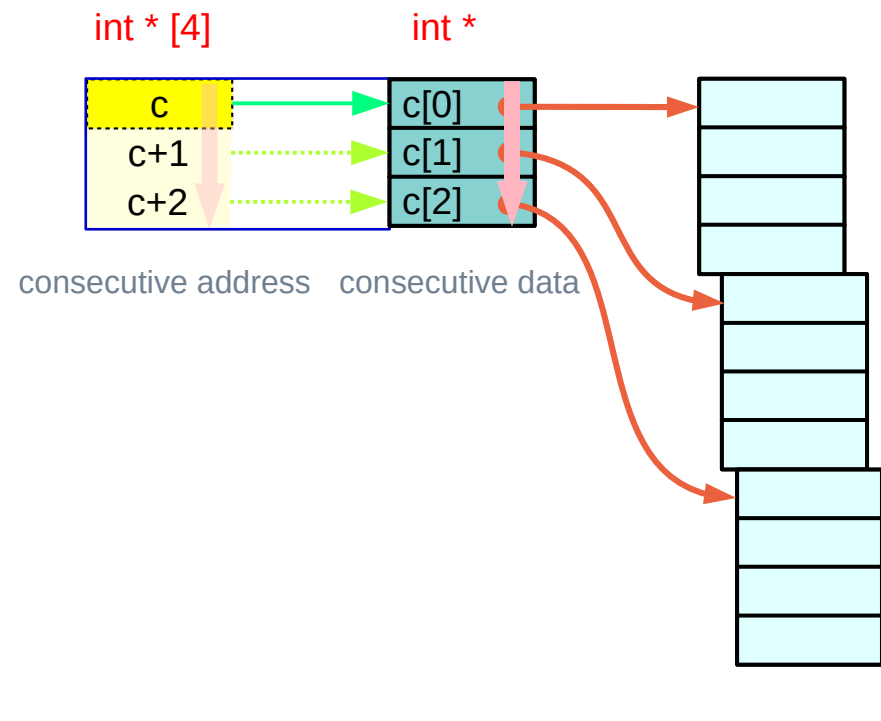
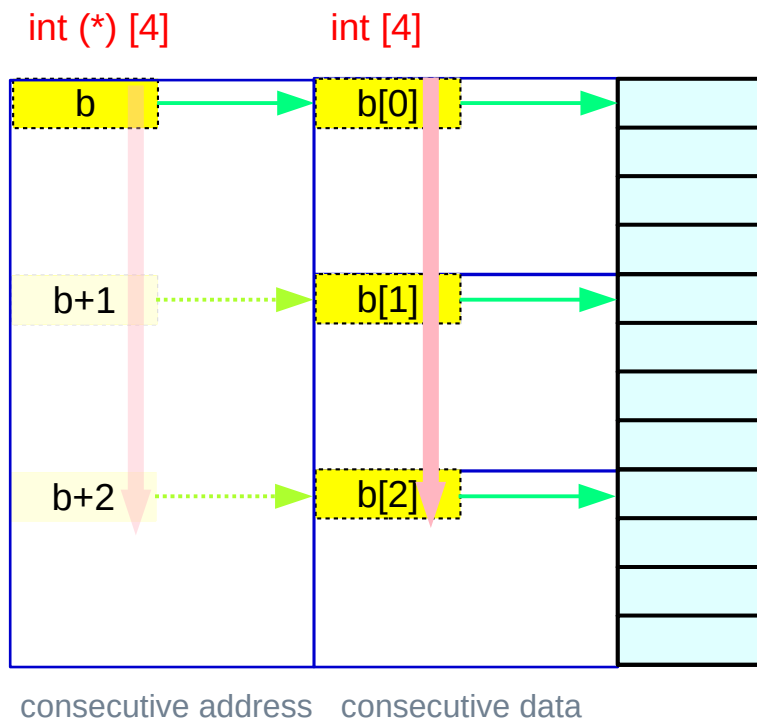
`*(b+m)`       $\longleftrightarrow$       `b[m]`  
`*(b[m]+n)`       $\longleftrightarrow$       `b[m][n]`

`int * c[M] ;`

with proper assignments

`*(c+m)`       $\longleftrightarrow$       `c[m]` or  
`*(c[m]+n)`       $\longleftrightarrow$       `c[m][n]`

# Pointer Arrays vs Array Pointers



`int (*b)[N] ;`

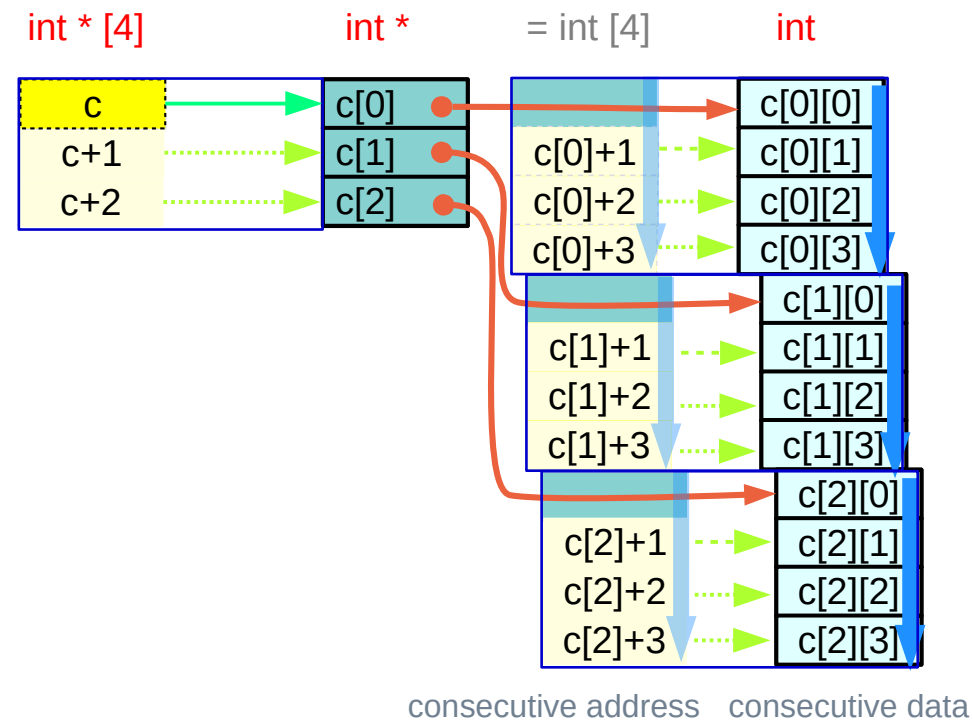
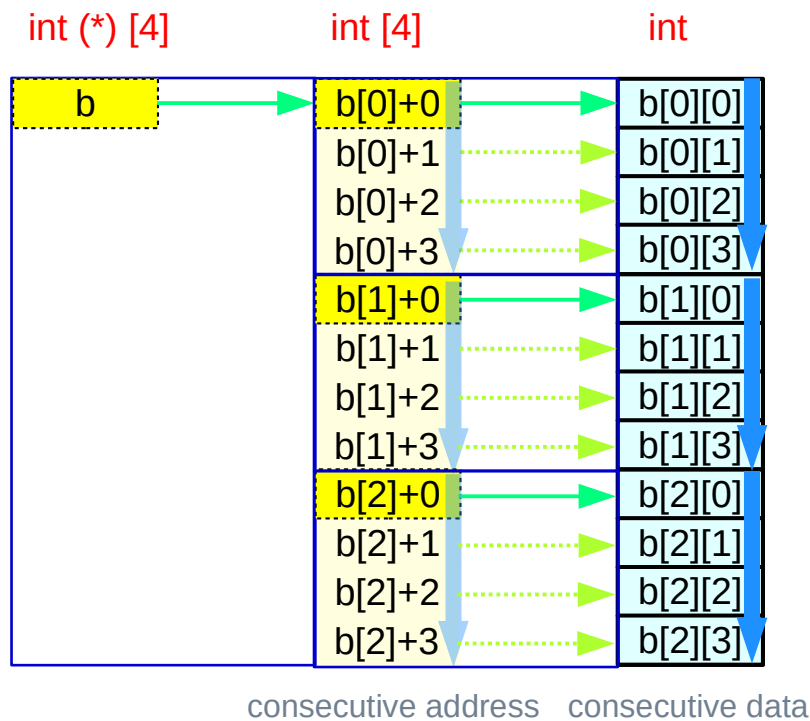
`*(b+m)`     $\longleftrightarrow$     `b[m]`  
`*(b[m]+n)`     $\longleftrightarrow$     `b[m][n]`

`int * c[M] ;`

with proper assignments

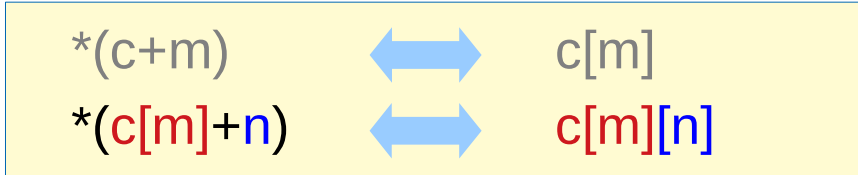
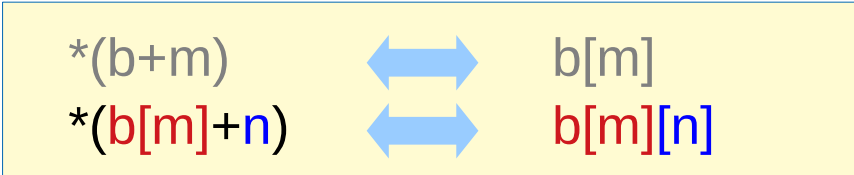
`*(c+m)`     $\longleftrightarrow$     `c[m]` or  
`*(c[m]+n)`     $\longleftrightarrow$     `c[m][n]`

# Pointer Arrays vs Array Pointers



```
int (*b)[N] ;
```

```
int * c[M] ;      with proper assignments
```



# Three contiguity constraints for 3-d arrays

## Pointer Array Approach (array of pointers)

$c[i][j][k]$        $\rightarrow$      $*(c[i][j] + k)$   
 $*(c[i][j] + k)$      $\rightarrow$      $*(*(c[i] + j) + k)$   
 $*(*(c[i] + j) + k)$   $\rightarrow$   $*(**(*c + i) + j) + k)$

contiguous **int**                                    **int**  
contiguous pointers to **int**                    **int \***  
contiguous double pointers to **int**           **int \*\***

the contiguity constraints are satisfied by allocating arrays of pointers

## Array Pointer Approach (pointer to arrays)

$c[i][j][k]$        $\rightarrow$      $*(c[i][j] + k)$   
 $*(c[i][j] + k)$      $\rightarrow$      $*(*(c[i] + j) + k)$   
 $*(*(c[i] + j) + k)$   $\rightarrow$   $*(**(*c + i) + j) + k)$

contiguous **0-d** arrays    **int**                                    **int**  
contiguous **1-d** arrays    **int [4]**                                    **int \***  
contiguous **2-d** arrays    **int [3][4]**                                **int (\*) [4]**

The contiguity constraints are satisfied by row major ordered linear data layout



# Contiguous array pointers $c[i][j][k] \equiv *(c[i][j] + k)$

```

c[0][0][0] = *(c[0][0] + 0)
c[0][0][1] = *(c[0][0] + 1)
c[0][0][2] = *(c[0][0] + 2)
c[0][0][3] = *(c[0][0] + 3)
c[0][1][0] = *(c[0][1] + 0)
c[0][1][1] = *(c[0][1] + 1)
c[0][1][2] = *(c[0][1] + 2)
c[0][1][3] = *(c[0][1] + 3)

```

• •  
• •

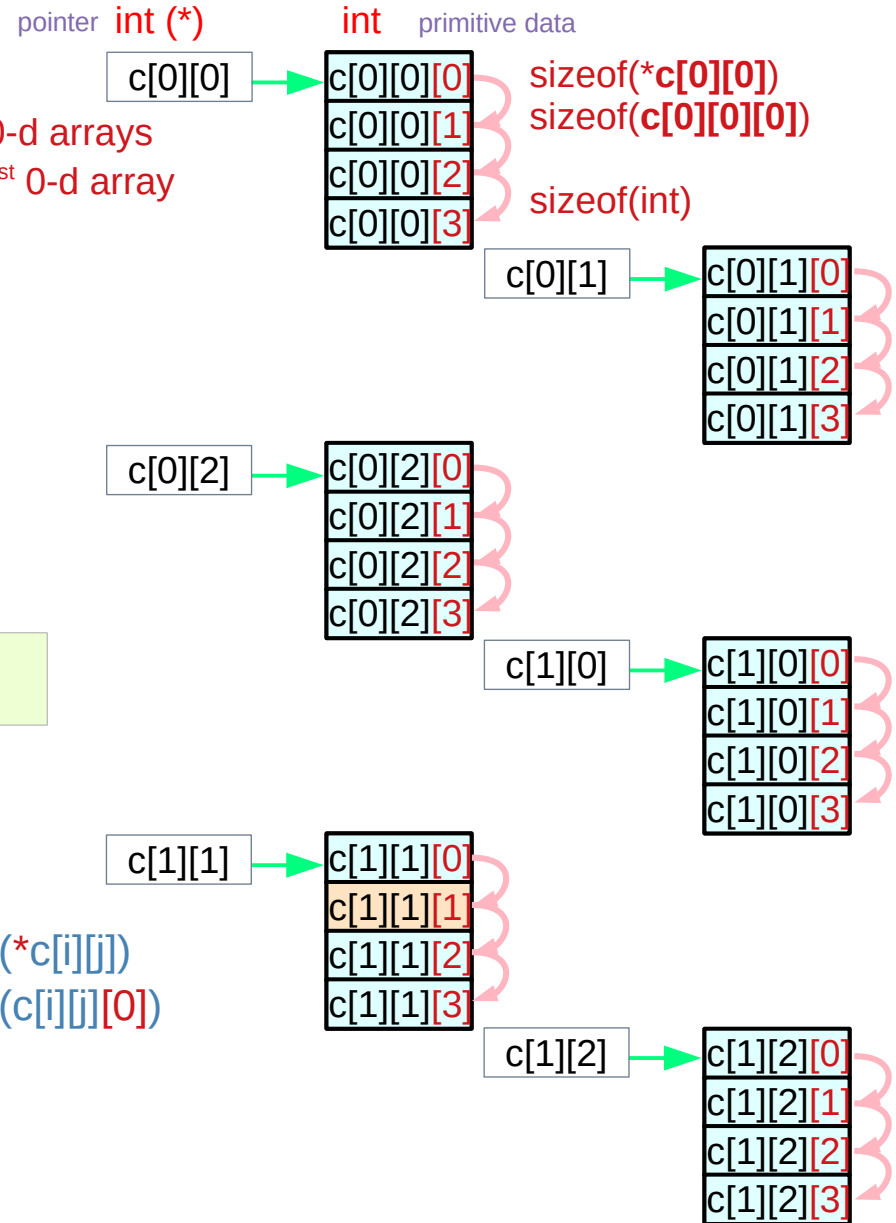
contiguous 1-d  
array elements

**c[i][j]**  
**int [4]**      4 contiguous 0-d arrays  
**int \***        points to the 1<sup>st</sup> 0-d array  
**int**            0-d array

`sizeof(c[i][j])`      [k]  
`sizeof(c[i][j][k]) * 4`  
`sizeof(int) * 4`

```
int c[2][3][4];
```

Address Value  
 $c[i][j] + k$   
 $\&c[i][j][0] + k * \text{sizeof}(*c[i][j])$   
 $\&c[i][j][0] + k * \text{sizeof}(c[i][j][0])$   
 $\&c[i][j][0] + k * 4$



# Contiguous array pointers $c[i][j] \equiv *(c[i] + j)$

```

c[0][0] = *(c[0] + 0)
c[0][1] = *(c[0] + 1)
c[0][2] = *(c[0] + 2)
c[1][0] = *(c[1] + 0)
c[1][1] = *(c[1] + 1)
c[1][2] = *(c[1] + 2)
    
```

**c[i]**  
**int [3][4]** 3 contiguous 1-d arrays  
**int (\*) [4]** points to the 1<sup>st</sup> 1-d array  
**int [4]** 1-d array

sizeof(c[i]) [j] [k]  
 sizeof(c[i][j][k]) \* 3 \* 4  
 sizeof(int) \* 3 \* 4

```
int c[2][3][4];
```

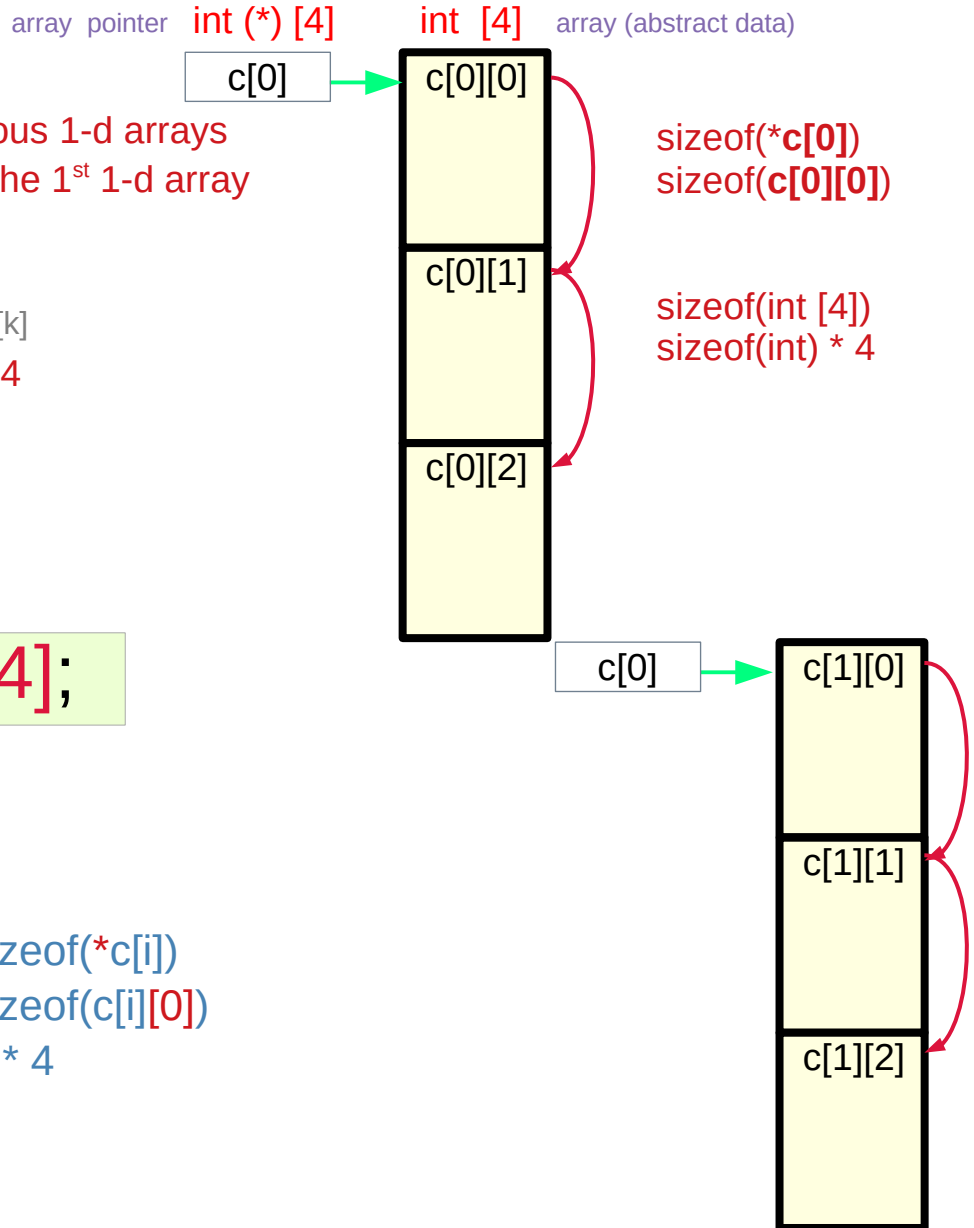
Address Value

$c[i] + j$

$\&c[i][0][0] + j * \text{sizeof}(*c[i])$

$\&c[i][0][0] + j * \text{sizeof}(c[i][0])$

$\&c[i][0][0] + j * 4 * 4$



# Contiguous array pointers $c[i] \equiv *(c + i)$

```
c[0] = *(c + 0)
c[1] = *(c + 1)
```

```
c
int [2][3][4]
int (*) [3][4]
int [3][4]
```

2 contiguous 2-d arrays  
points to the 1<sup>st</sup> 2-d array  
2-d array

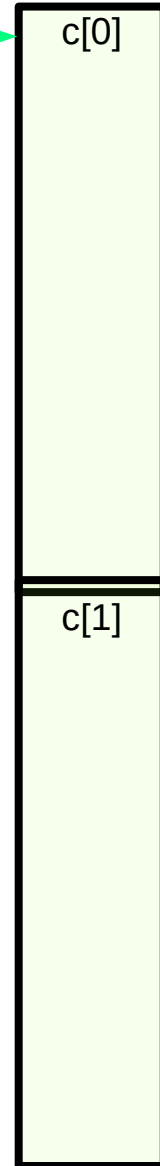
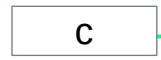
```
sizeof(c)           [i] [j] [k]
sizeof(c[i][j][k]) * 2 * 3 * 4
sizeof(int) * 2 * 3 * 4
```

```
int c[2][3][4];
```

Address Value

```
c + i
&c[0][0][0] + i * sizeof(*c)
&c[0][0][0] + i * sizeof(c[0])
&c[0][0][0] + i * 4 * 3 * 4
```

array pointer  $int (*) [3][4]$   $int [3][4]$  array (abstract data)



sizeof(\*c)  
sizeof(c[0])

sizeof(int [3][4])  
sizeof(int) \* 3 \* 4

# Contiguous linear layout

```
int c [L][M][N];
```

```
C [i][j][k];
```

L	M	N
i	j	k
$i * M * N$	$j * N$	k

Base Index = 0

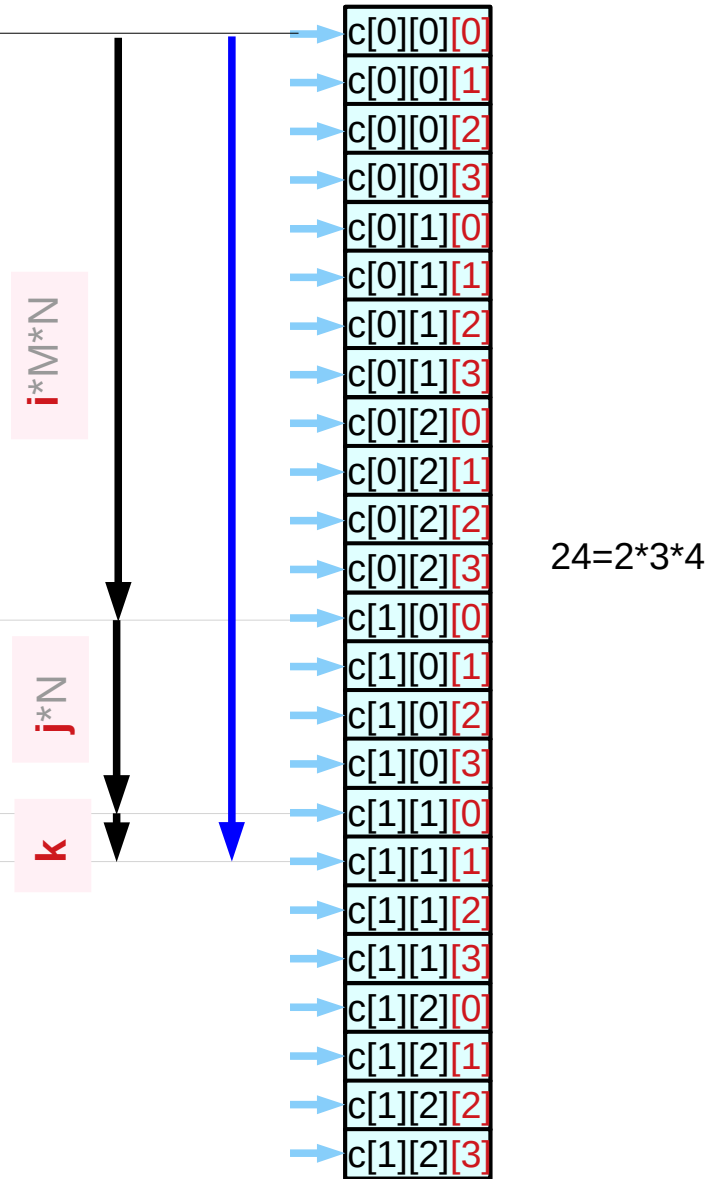
Offset Index 1 (i=1)

Offset Index 2 (j=1)

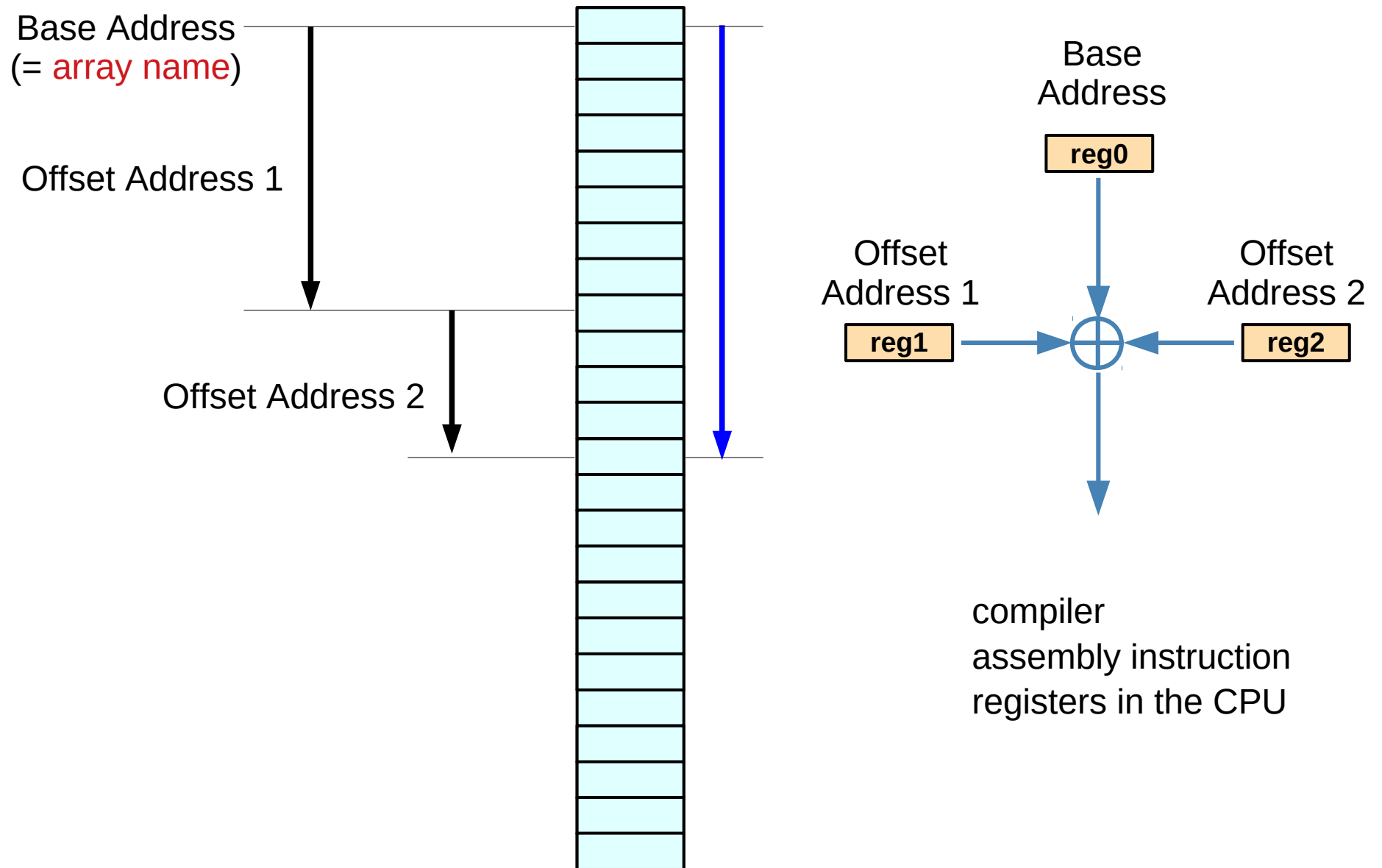
Offset Index 3 (k=1)

$$(i * M * N + j * N + k)$$

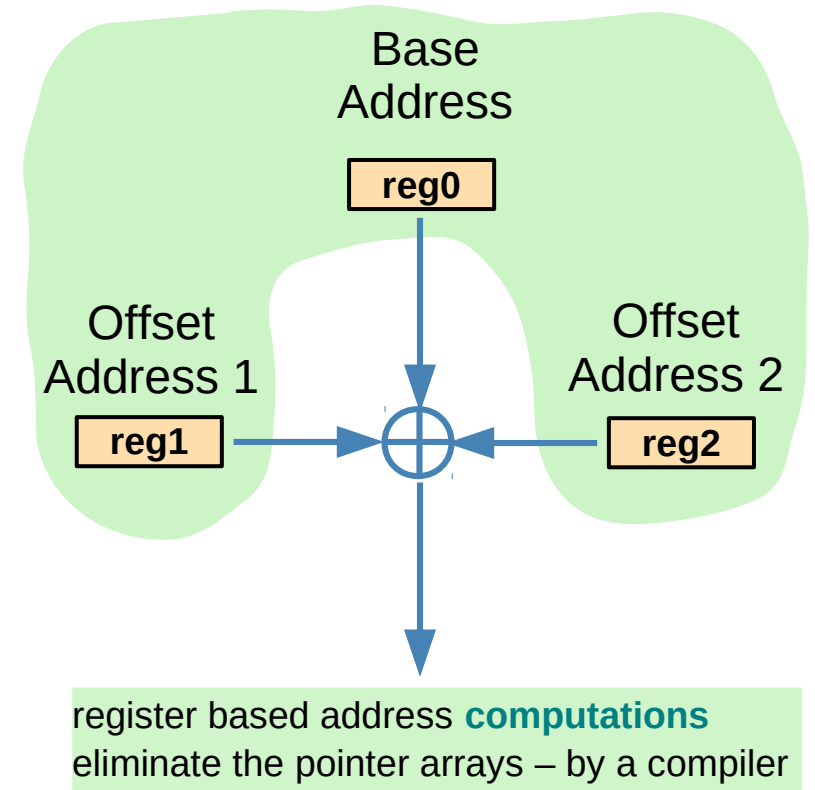
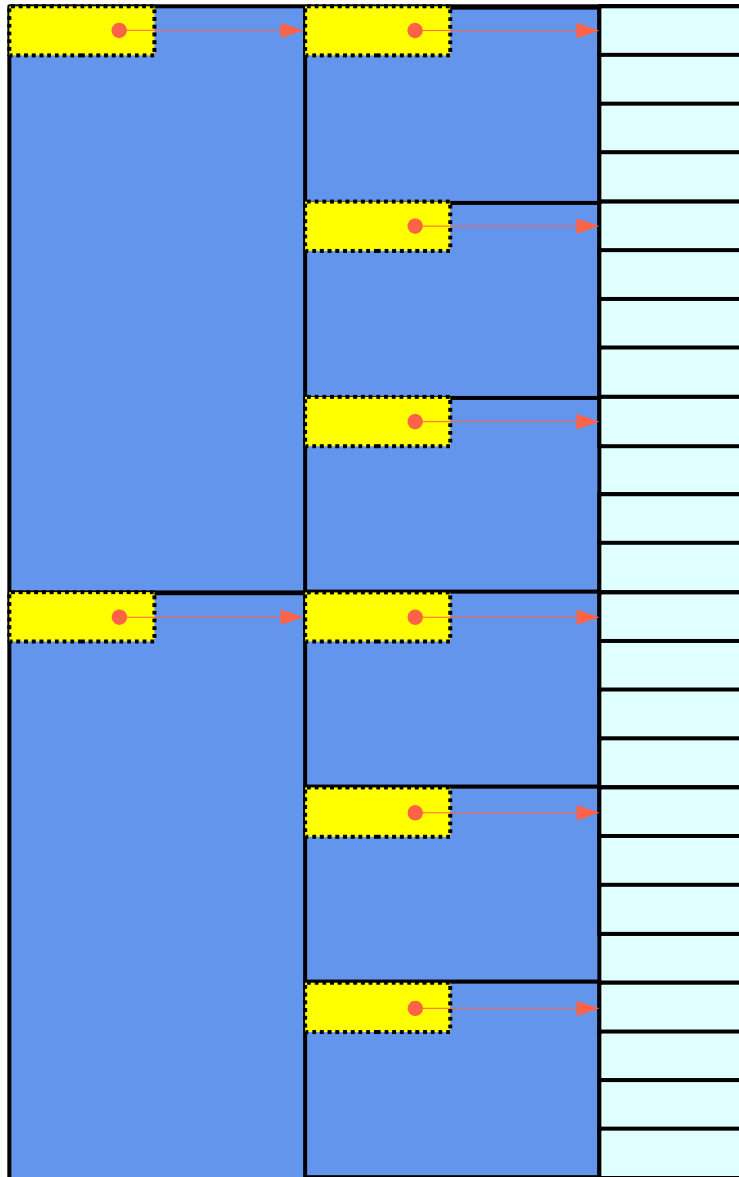
$$((i * M + j) * N + k)$$



# Base and Offset Addressing



# Array Pointer Approach



**Array Pointer Approach**  
**(pointer to arrays)**

## References

- [1] Essential C, Nick Parlante
- [2] Efficient C Programming, Mark A. Weiss
- [3] C A Reference Manual, Samuel P. Harbison & Guy L. Steele Jr.
- [4] C Language Express, I. K. Chun