

Applications of Array Pointers (1A)

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

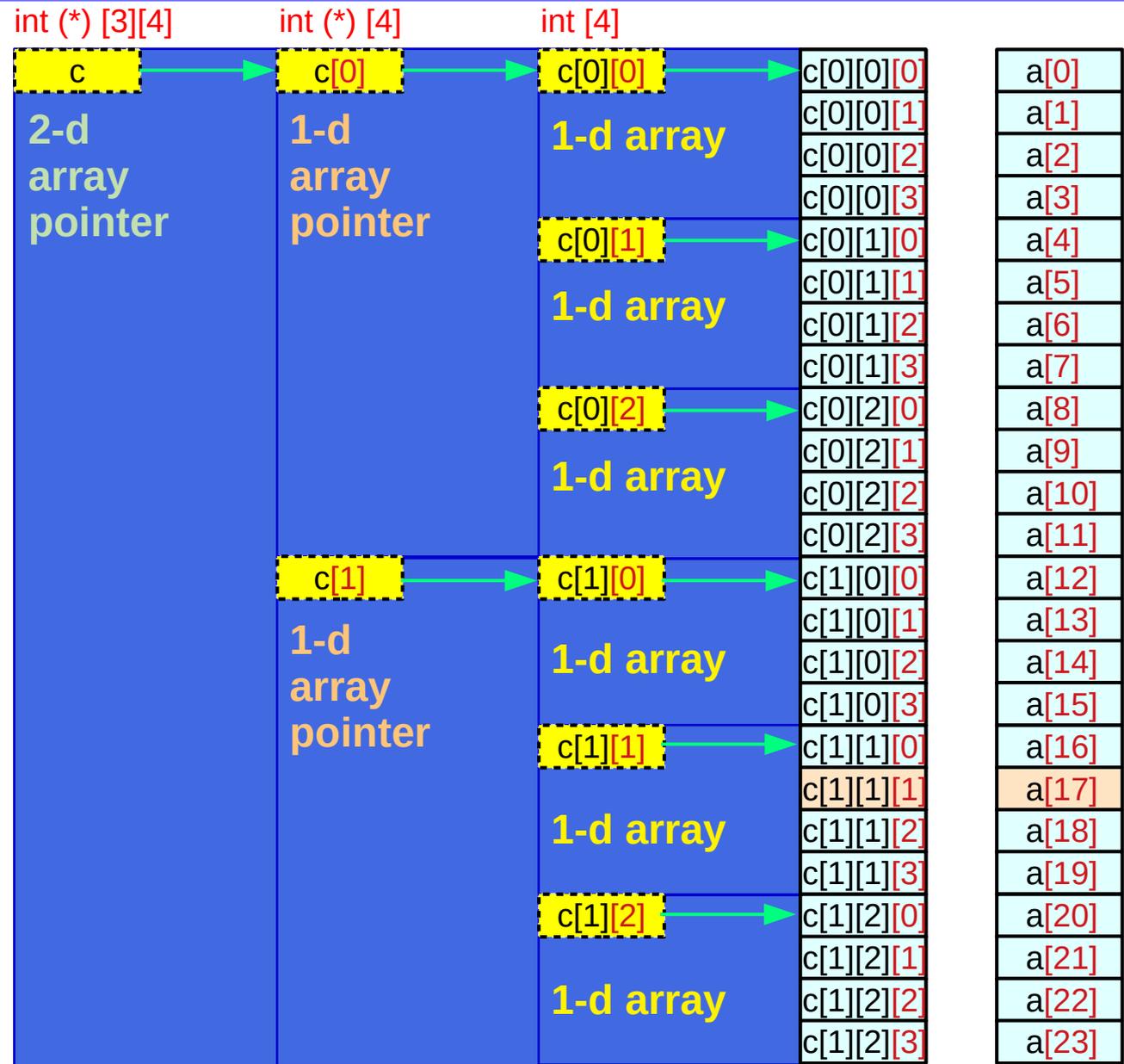
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is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

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

Virtual Array Pointers in Multi-dimensional Arrays

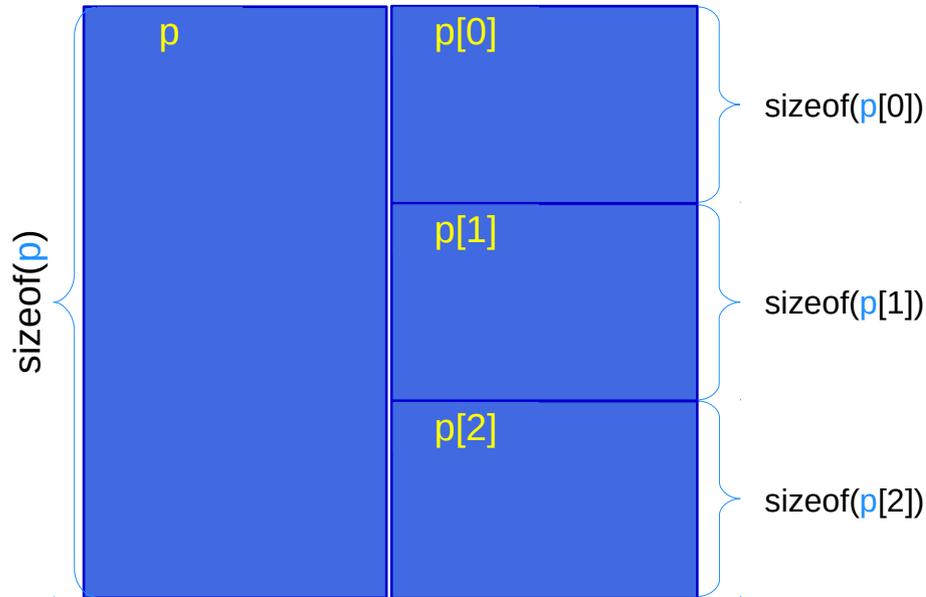
3-d array structure

- Hierarchical
- Nested Structure
- Virtual Array Pointers over
 - Contiguous
 - Linear Layout

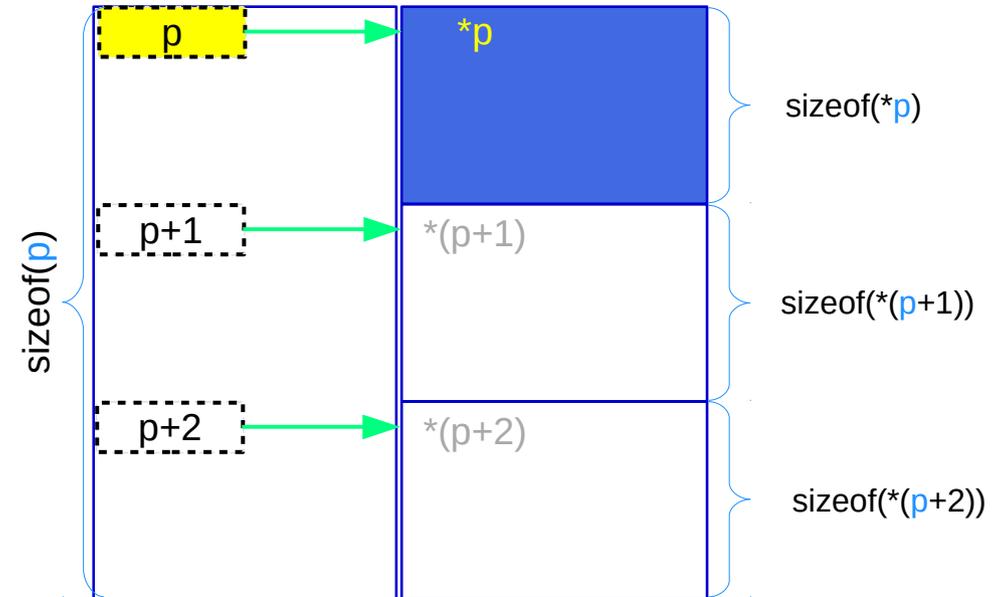


Array **p** and virtual array pointer **p**

Abstract data (array) **p**



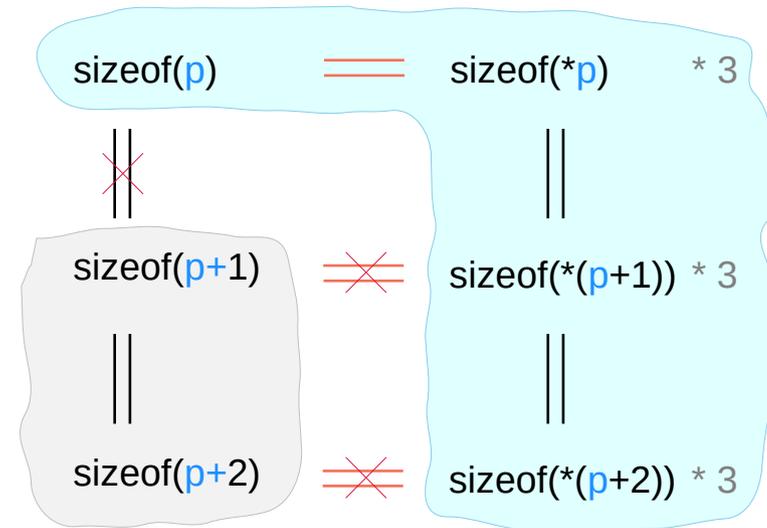
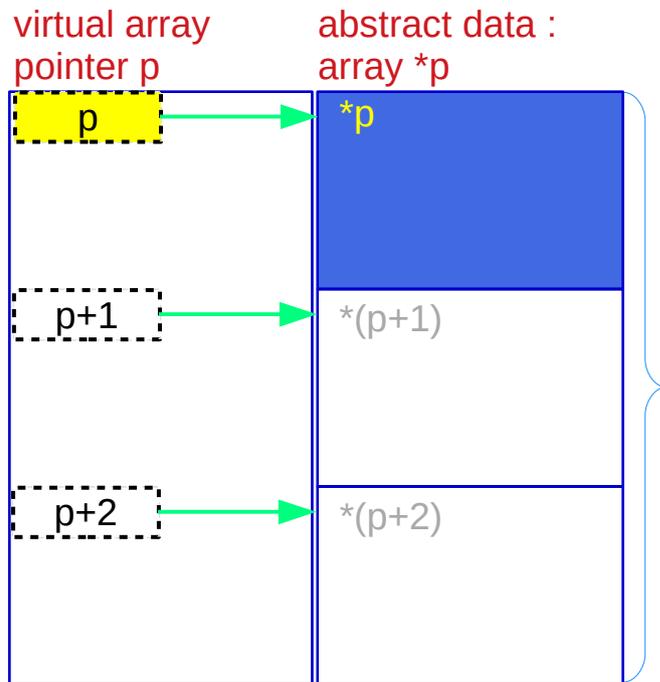
Virtual array pointer **p**



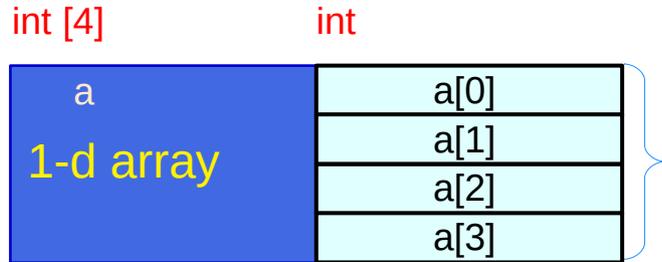
p is the name of an array and has a array pointer type but has a size of the array

p is a virtual array pointer

Virtual array pointer to abstract data

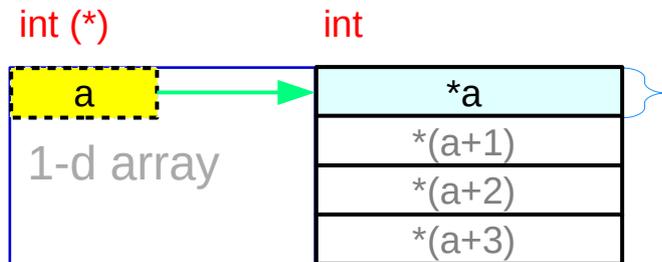


Array **a** and pointer **a**



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 and has a pointer type but has a size of the array

a is a virtual array pointer

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

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

`sizeof(b)`

`int [3] [4]`

`int [4]`

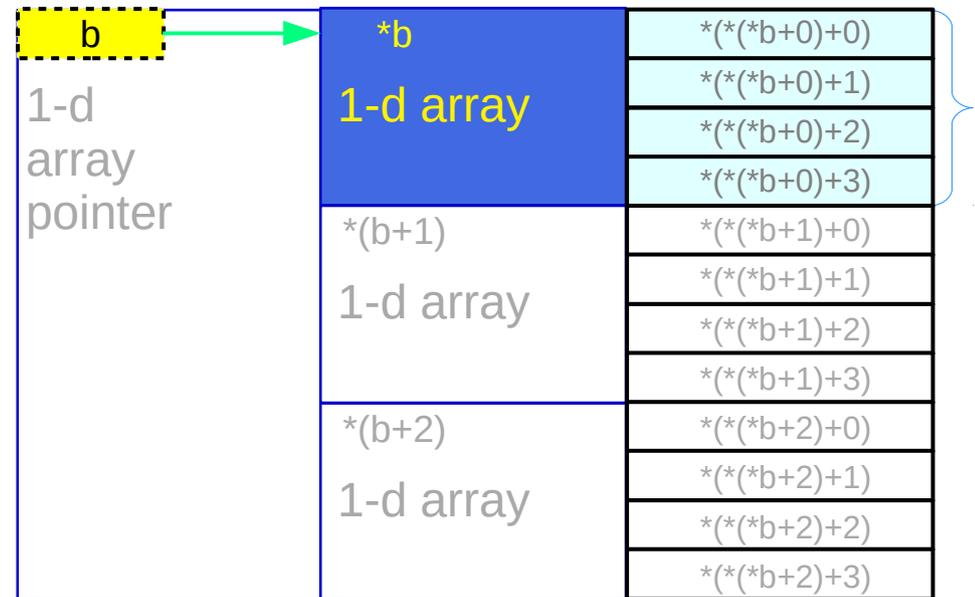


1-d array pointer **b** general pointer type

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

`int (*) [4]`

`int [4]`



b is the name of a 2-d array and has a 1-d array pointer type but has a size of the array

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 and has a 2-d array pointer type but has a size of the array

c is a virtual array pointer

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

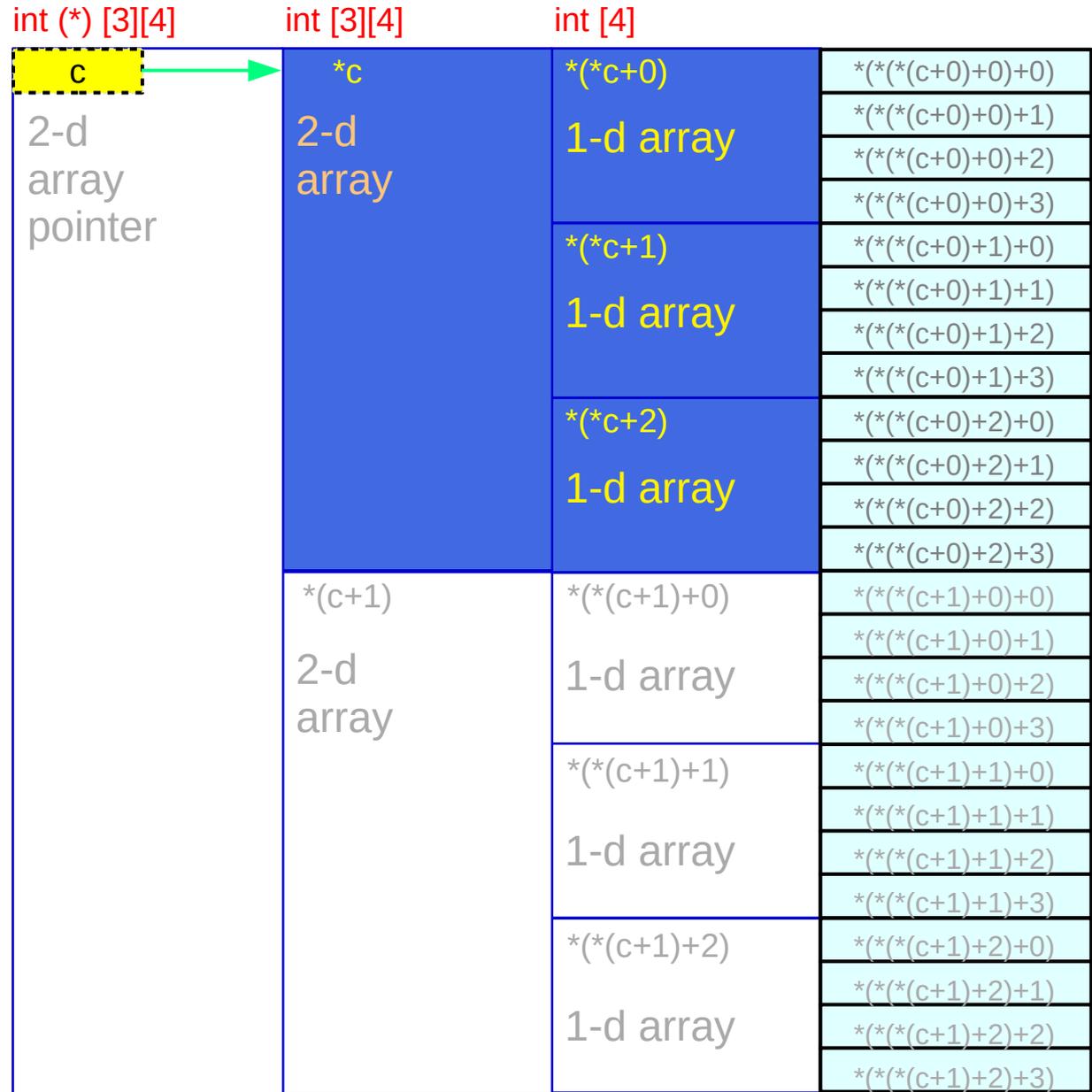
2-d array pointer c

general pointer type

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

c is the name of a 3-d array and has a 2-d array pointer type but has a size of the array

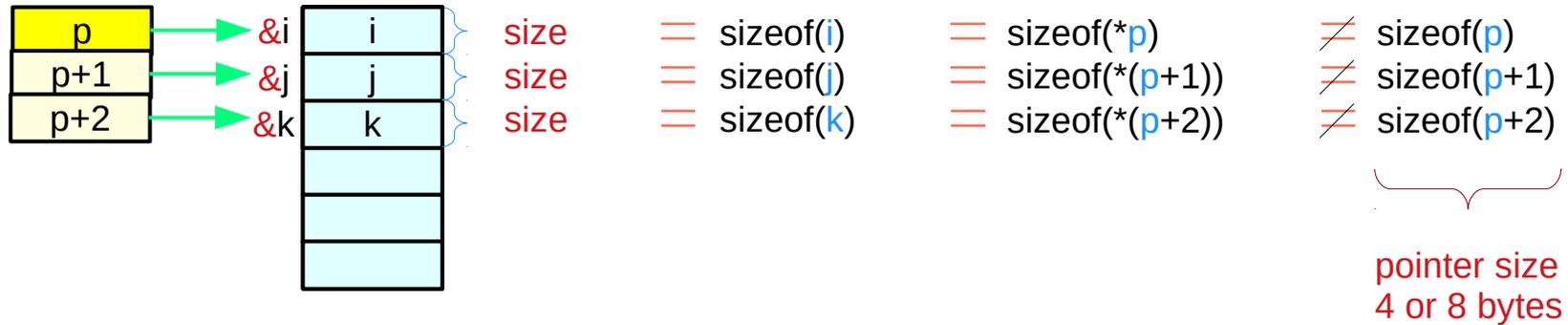
c is a virtual array pointer



Pointers to primitive data

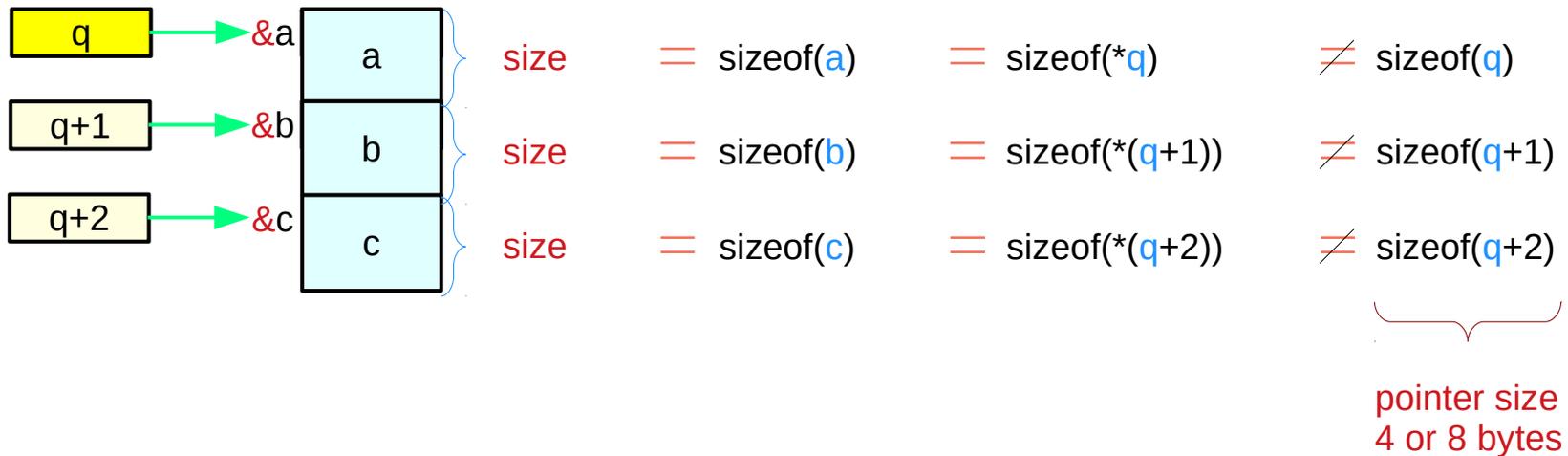
int *p;

int i, j, k;



double *q;

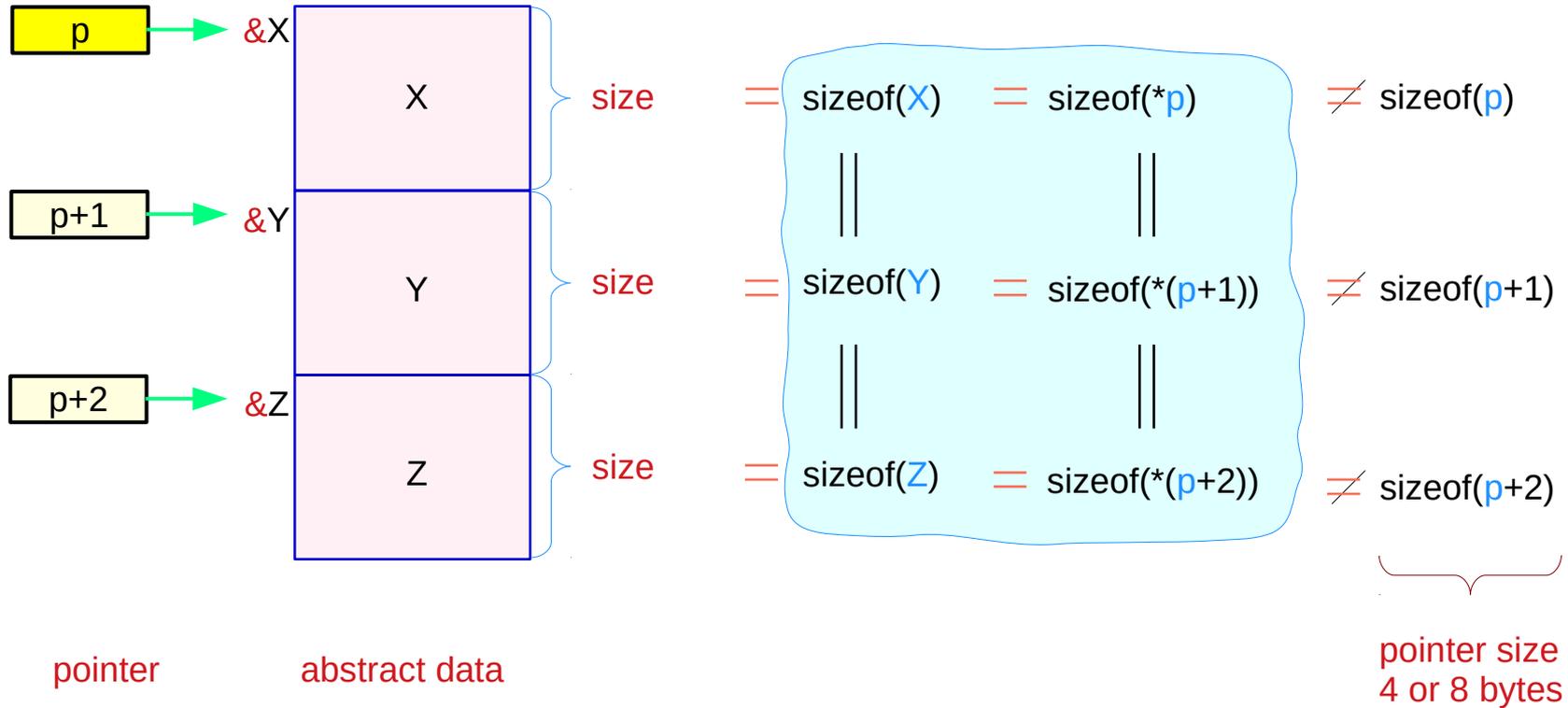
double a, b, c;



Pointers to abstract data

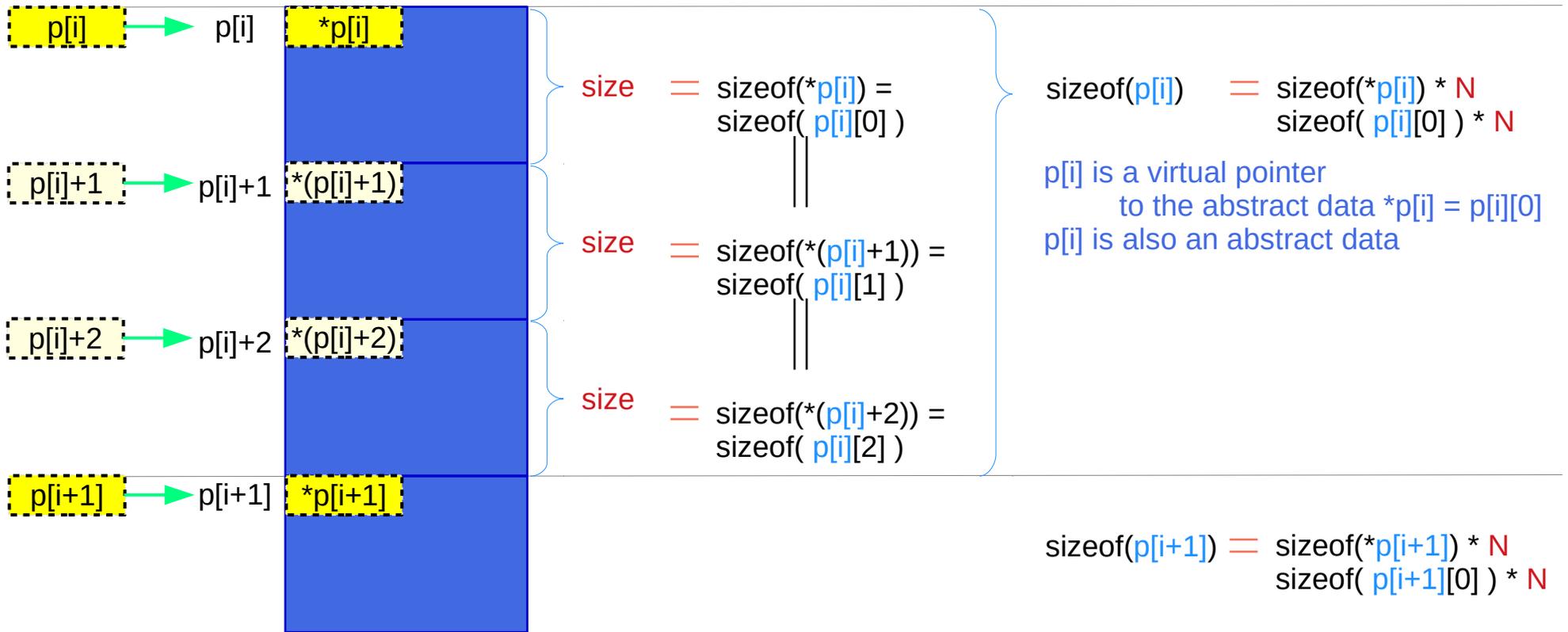
T *p;

T X, Y, Z;



Virtual pointers in a multi-dimensional array

$p[i] :: T1$ $*p[i], *p[i+1] :: T2$

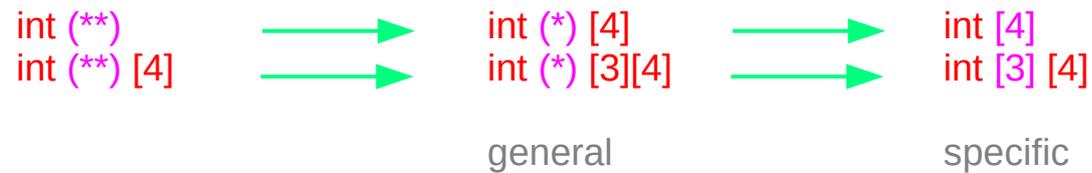
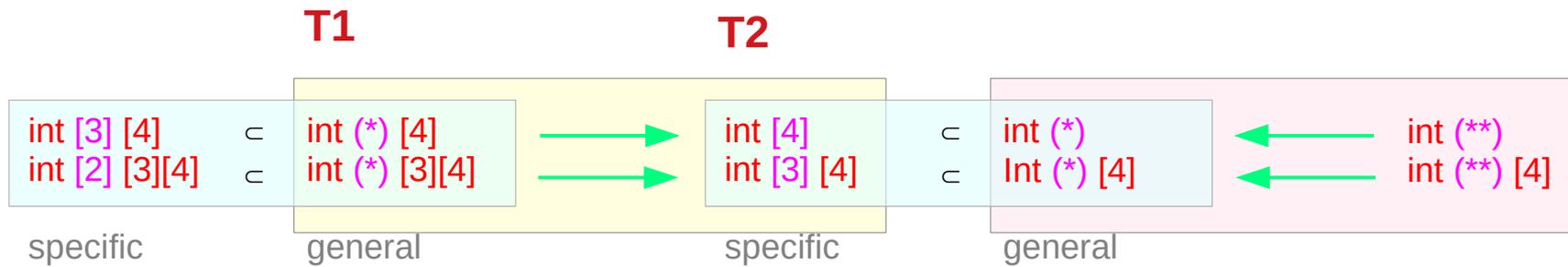


T1 **T2**

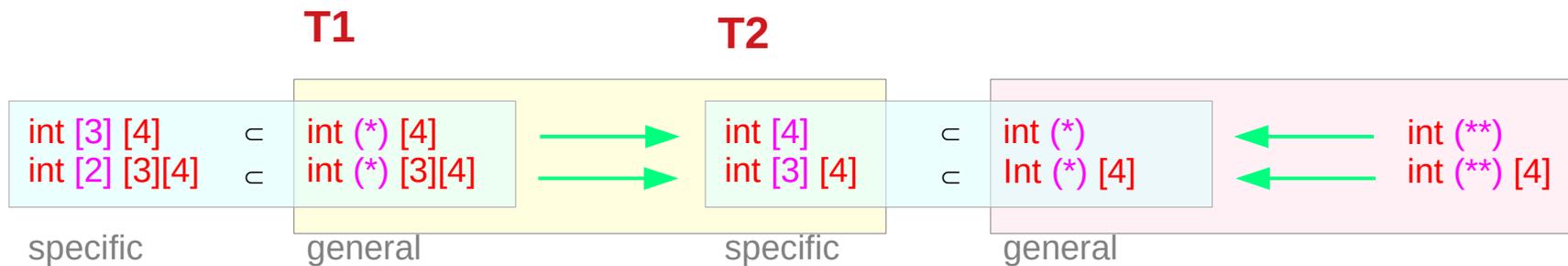
$\text{int} (*) [4]$ $\text{int} [4]$ $\subset \text{int} (*)$

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

Virtual pointers in a multi-dimensional array



Virtual pointers in a multi-dimensional array



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

T1 a;

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

T2 b;

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

general

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

specific

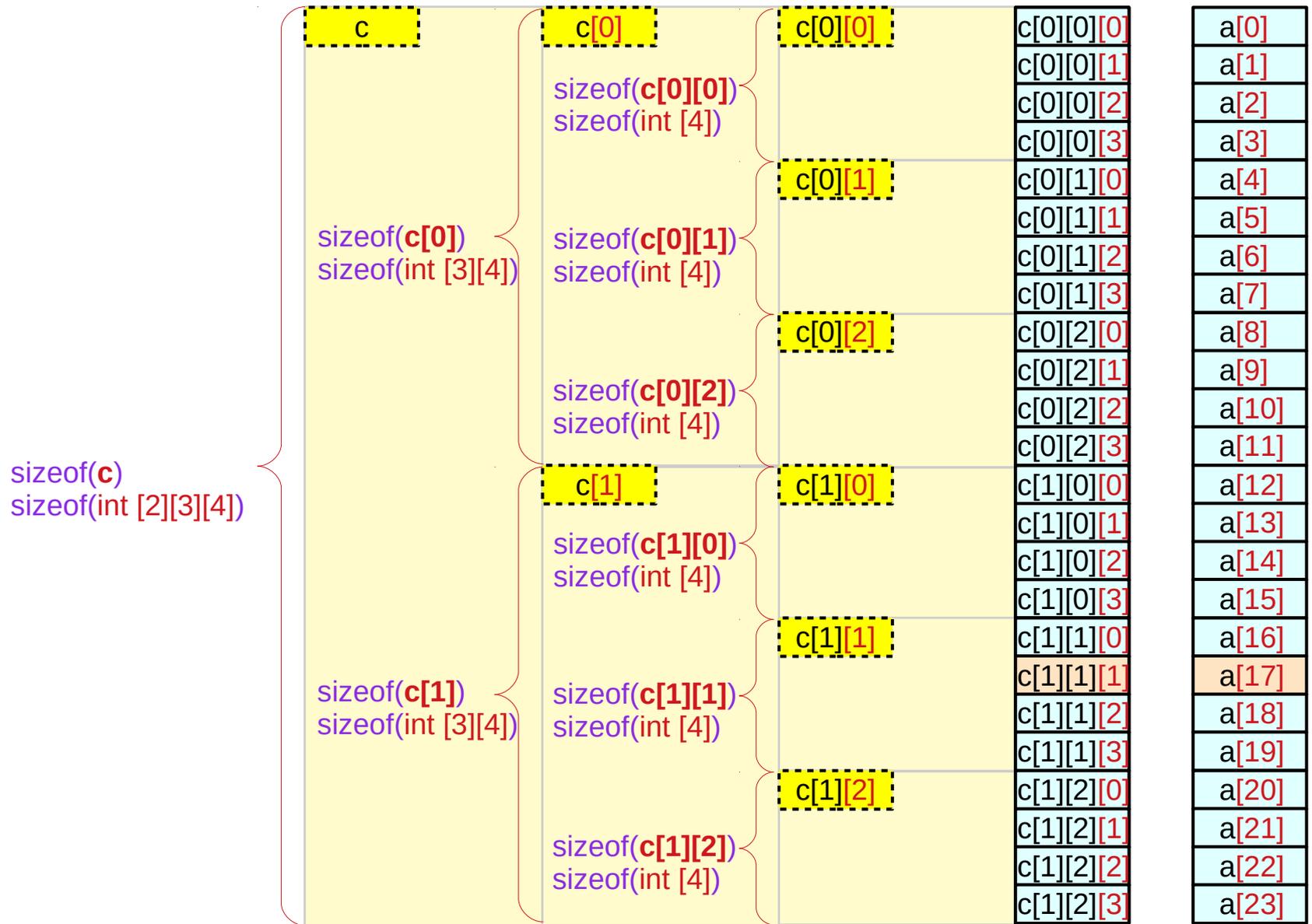
T1 references T2
T2 is a dereference of T1

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

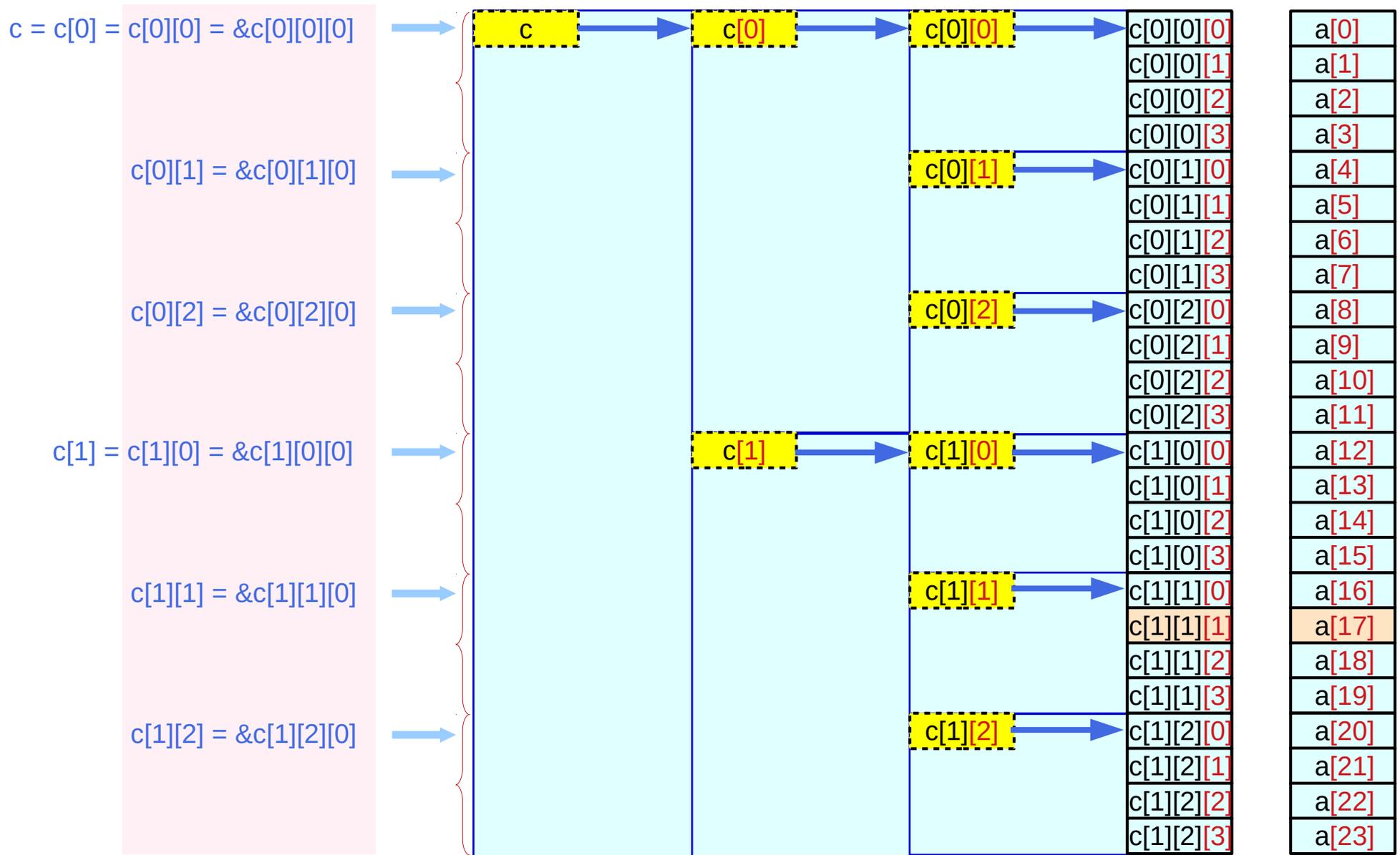
Virtual array pointers – types, sizes, and values

int c[2][3][4];	c[i][j]	c[i][j][0]	
type	int [4] int (*)	int int	<ul style="list-style-type: none"> 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 (*)	<ul style="list-style-type: none"> 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]	
int c[2][3][4];	c	c[0]	
type	int [2][3][4] int (*)[3][4]	int [3][4] int (*)[4]	<ul style="list-style-type: none"> abstract data type array pointer type
size	sizeof(c) =	sizeof(c[0]) * 2	= sizeof(int) * 4 * 3 * 2
value (address)	c =	&c[0]	

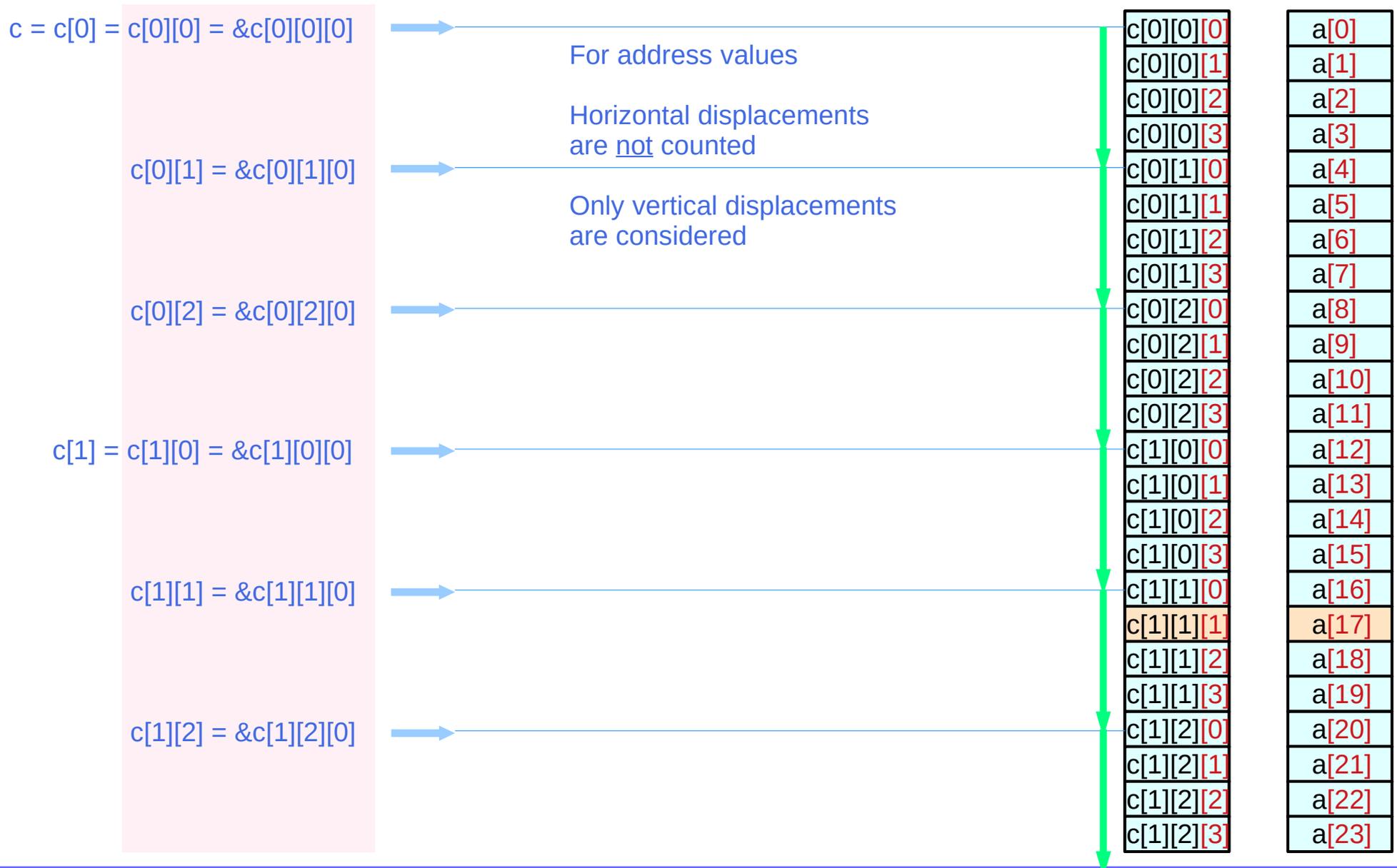
virtual array pointers c , $c[i]$, $c[i][j]$ – sizes



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

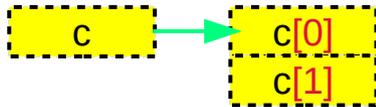
$c = c[0] = c[0][0] = \&c[0][0][0]$ means \rightarrow
 $c[0][1] = \&c[0][1][0]$ means \rightarrow
 $c[0][2] = \&c[0][2][0]$ means \rightarrow
 $c[1] = c[1][0] = \&c[1][0][0]$ means \rightarrow
 $c[1][1] = \&c[1][1][0]$ means \rightarrow
 $c[1][2] = \&c[1][2][0]$ means \rightarrow

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

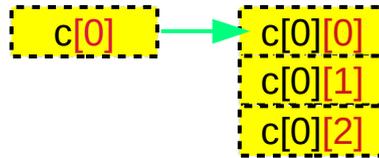
Virtual array pointer c, c[0], c[0][0] – types and sizes

Types – array pointers

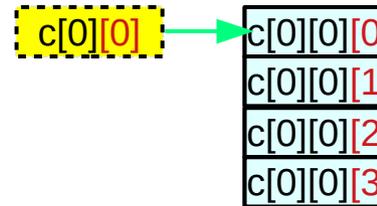
`int (*) [3][4]`



`int (*) [4]`



`int [4]`



Sizes – abstract data

`sizeof(c)`

`sizeof(c[0]) * 2`

`sizeof(c[0][0]) * 2 * 3`

`sizeof(c[0][0][0]) * 2 * 3 * 4`

`sizeof(int [2][3][4])`

`sizeof(int [2][3][4]) = 96`

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

`sizeof(c[0])`

`sizeof(c[0][0]) * 3`

`sizeof(c[0][0][0]) * 3 * 4`

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

`sizeof(int [3][4]) = 48`

`sizeof(int (*)[4]) = 4 / 8`

`sizeof(c[0][0])`

`sizeof(c[0][0][0]) * 4`

`sizeof(int [4])`

`sizeof(int [4]) = 16`

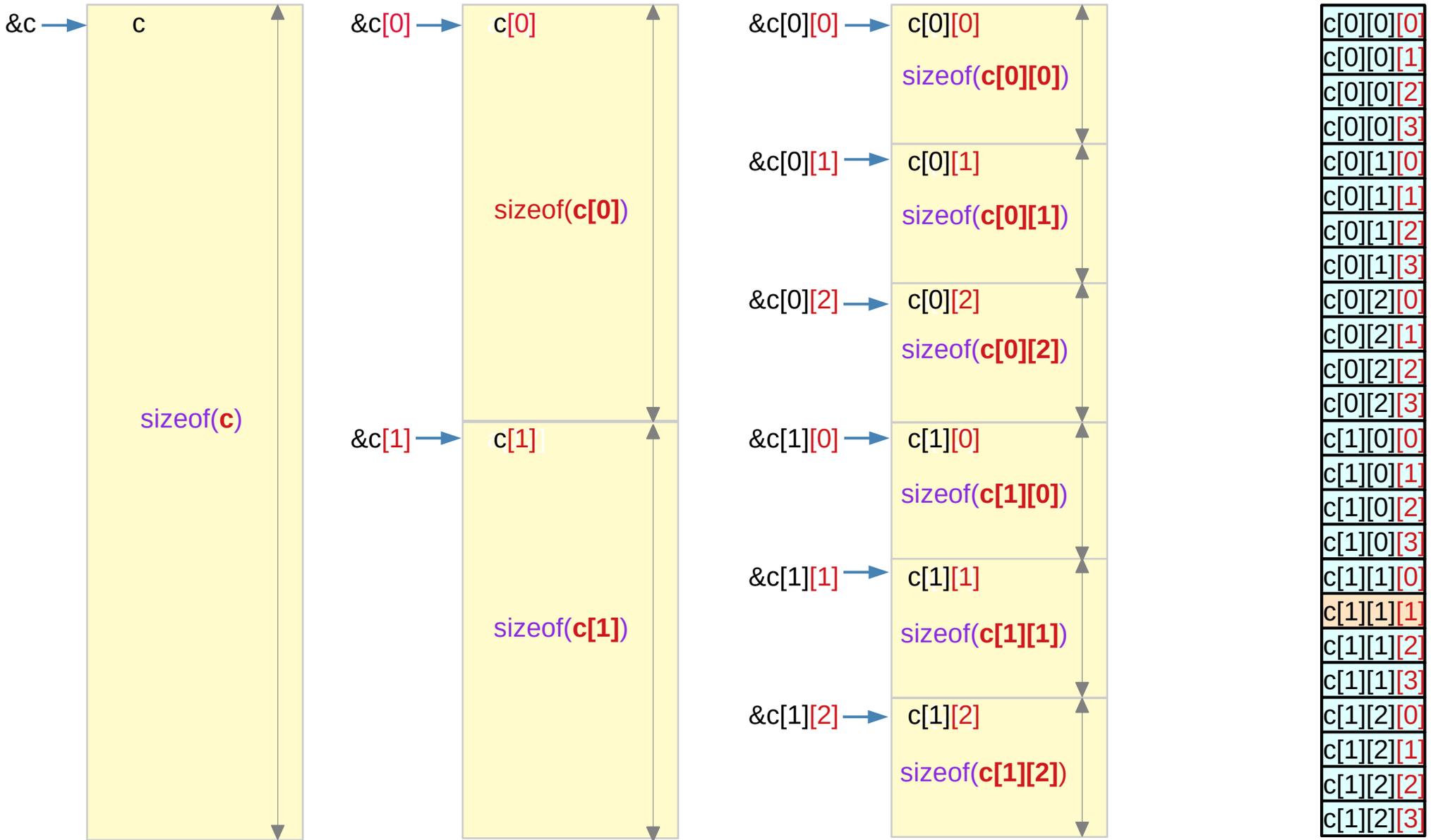
`sizeof(int (*) = 4 / 8`

`sizeof(c[0][0][0])`

`sizeof(int)`

`sizeof(int) = 4`

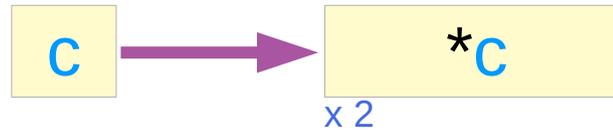
Abstract Data c , $c[i]$, $c[i][j]$ – start addresses and sizes



Types in a multi-dimensional array

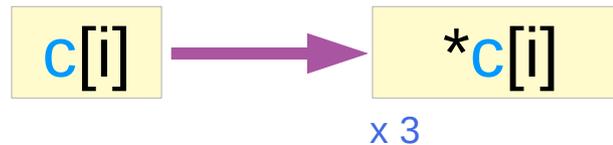
```
int c [2][3][4];
```

abstract data `int [2] [3][4]`
array pointer `int (*) [3][4]`



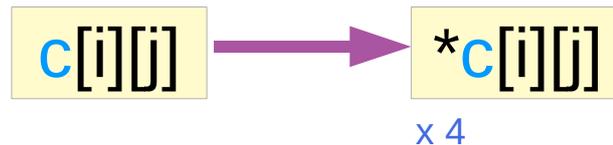
`int [3] [4]` abstract data
`int (*) [4]` array pointer

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



`int [4]` abstract data
`int (*)` array pointer

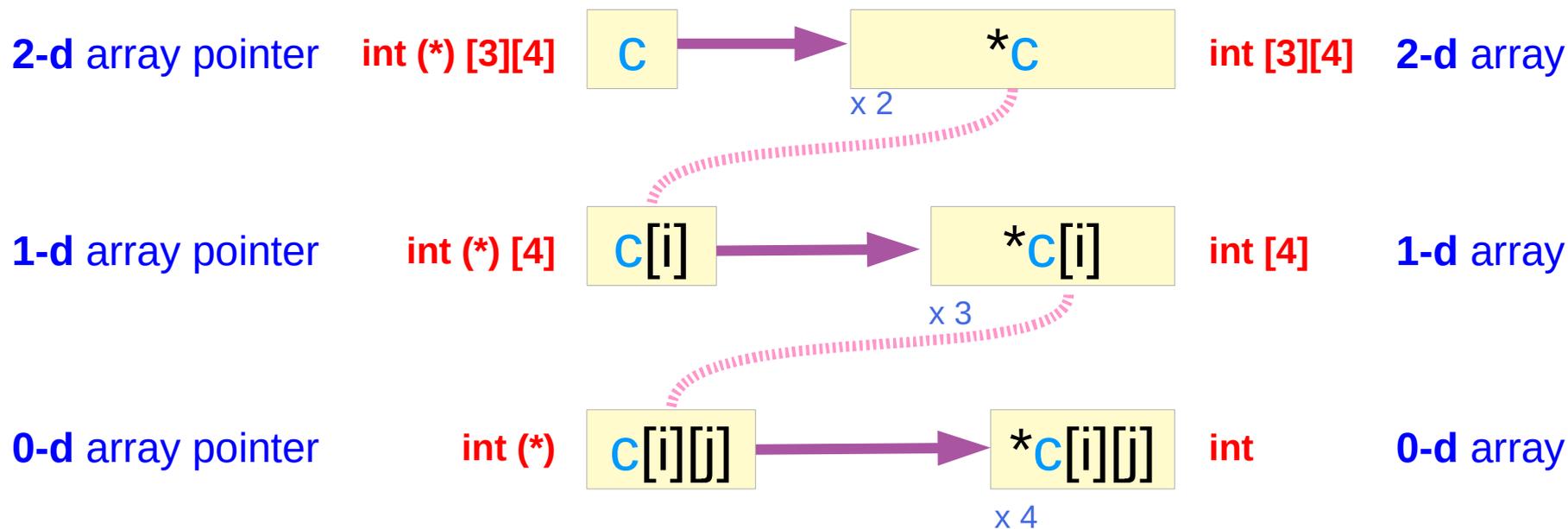
abstract data `int [4]`
array pointer `int (*)`



`int` primitive data

Virtual array pointers and abstract data

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

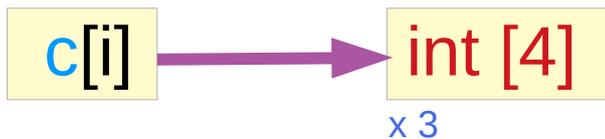
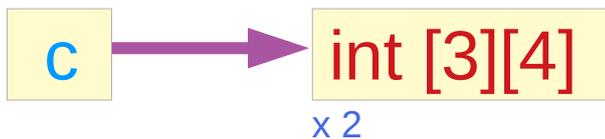


all these pointers are virtual, and take no actual memory locations

exploiting the **contiguity** of allocated memory locations

Abstract Data Sizes

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



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)

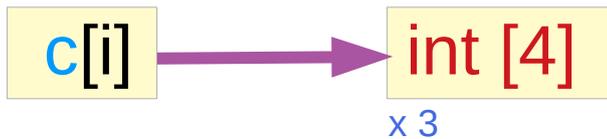
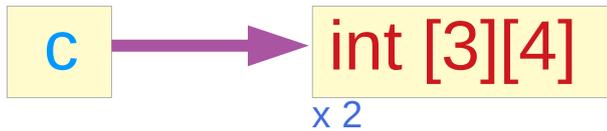
the size of a pointer type is fixed
Here, the sizes of virtual pointers are shown
i.e, the sizes of different abstract data types

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

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];
```

```
c → int [3][4]
```

```
c[i] → int [4]
```

```
c[i][j] → int
```

not real array pointers
virtual array pointers



```
c int (*)[3][4]  
sizeof(int (*) [3][4]) = pointer size ≠ sizeof(c)
```

```
c[i] int (*) [4]  
sizeof(int (*) [4]) = pointer size ≠ sizeof(c[i])
```

```
c[i][j] int [4]  
sizeof(int [4]) = pointer size ≠ sizeof(c[i][j])
```

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

Hierarchical nested array pointers

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

c points to a **2-d** array
increment size: $\text{sizeof(int)} * 2 * 3 * 4$

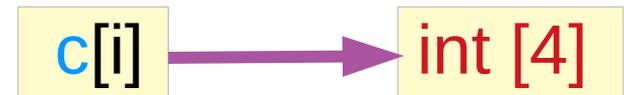
c[i] points to an **1-d** array
increment size: $\text{sizeof(int)} * 3 * 4$

c[i][j] points to an integer
increment size: $\text{sizeof(int)} * 4$

int (*) [3][4]



int (*) [4]



int (*)



Types of array pointers in a 3-d array

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

<code>c[i][j][k]</code>	:: int	int
<code>c[i][j]</code>	:: int _[k] ^[4]	int (*) _[k]
<code>c[i]</code>	:: int _{[j] [k]} ^{[3] [4]}	int (*) _{[j] [k]} ^[4]
<code>c</code>	:: int _{[j] [k]} ^{[2] [3] [4]}	int (*) _{[j] [k]} ^{[3] [4]}
	array type (name)	array pointer type

Sizes of array pointers in a 3-d array

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

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

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

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

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

Address values of 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]

Summary of 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 $c + i$

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

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 $c[i] + j$

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

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 $c[i][j] + k$

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

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

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

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



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

$$\begin{aligned}c[i][j][k] &= *(c[i][j] + k) \\ c[i][j] &= *(c[i] + j) \\ c[i] &= *(c + i)\end{aligned}$$

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

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

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

General requirements

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



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

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

Pointer array approach

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

$c[i][j][k] :: \text{int}$
 $c[i][j] :: \text{int}^*$
 $c[i] :: \text{int}^{**}$

$c[i] \leftarrow \&b[i*3]$
 $b[j] \leftarrow \&a[j*4]$

Explicit
Arrays of pointers with
Multiple Indirection

N-dim Array approach

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

$c[i][j][k] :: \text{int}$
 $c[i][j] :: \text{int}[4]$
 $c[i] :: \text{int}^*[4]$

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

Implicit
Nested
Virtual Array Pointers

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

General requirements

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



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

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

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]
```

Implicit
Nested
Virtual Array Pointers



Using N-dimensional arrays

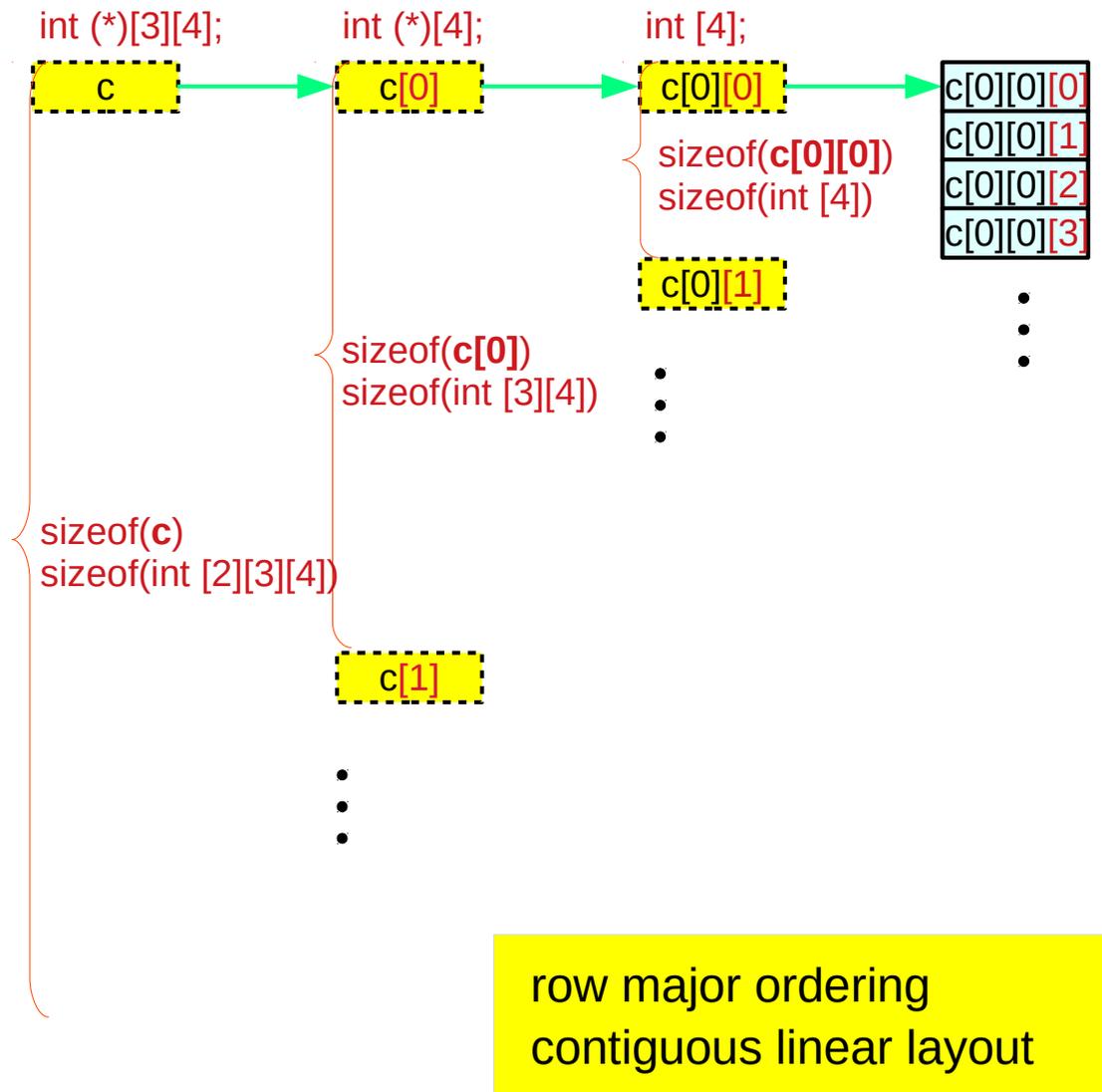
```
int c [2][3][4];
```



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

constraints

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



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

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

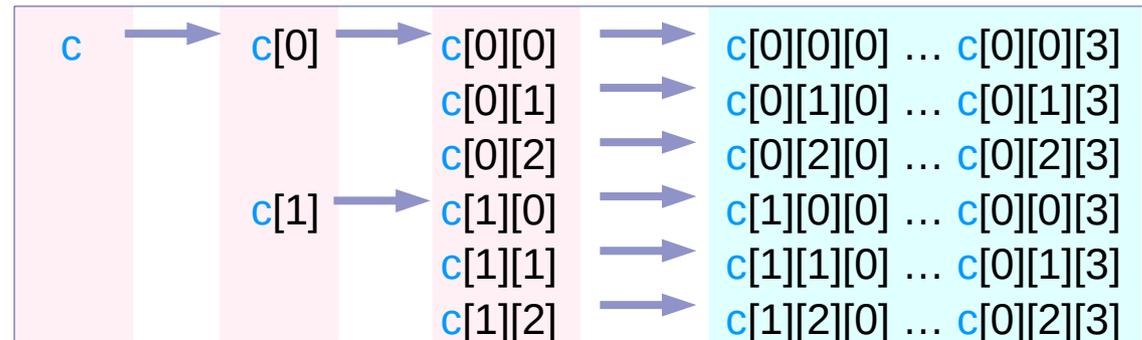
`&c[i][j][0] = c[i][j]`
`&c[i][0] = c[i]`
`&c[0] = c`

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

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

`c[i]` virtual 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 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 ... int`
`int (*) [3][4]` `int (*) [4]` `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 $c[i]$ and $c[i][j]$

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

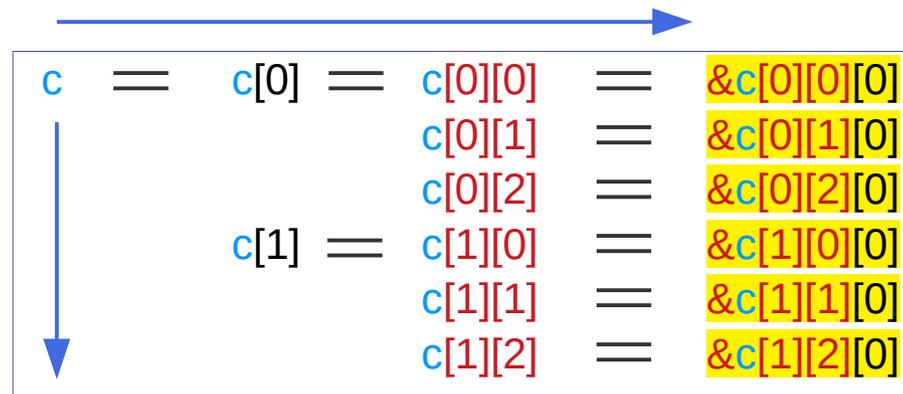
$\&c[i][j][0] = c[i][j]$
 $\&c[i][0] = c[i]$
 $\&c[0] = c$

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

$\text{int } c[2][3][4];$

virtual array pointers

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



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

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

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

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

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

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

$int\ c[2][3][4];$

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

append [0] to the right

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

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

$int\ (*)\ [4]$

$int\ [4]$

int

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

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

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

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

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

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

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

Finding sub arrays for the leading elements $c[i][j][0]$

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

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

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

$int\ c[2][3][4];$

```
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

```
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
```

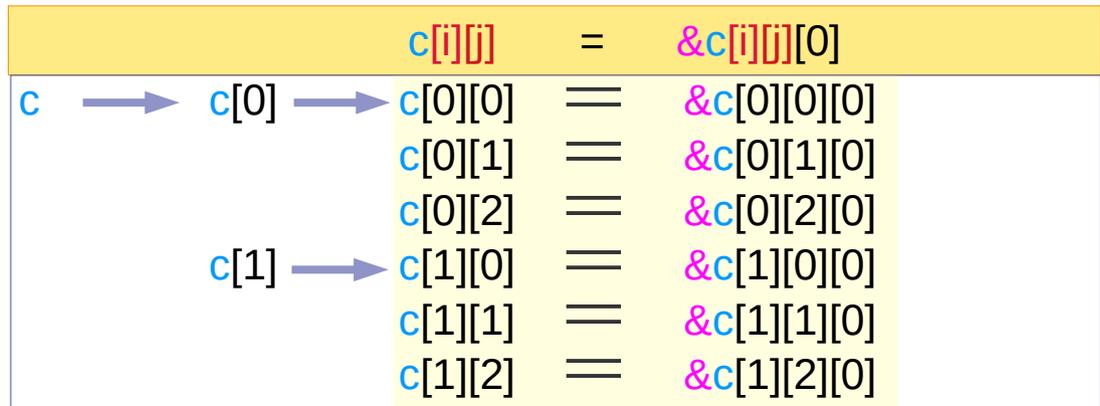
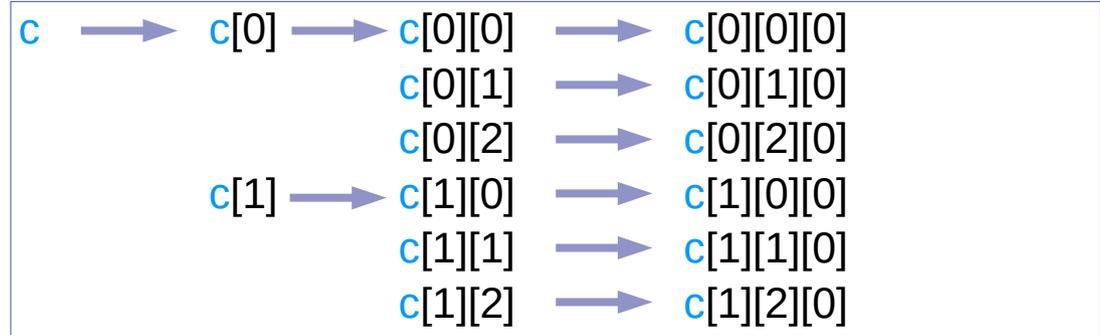
Pointer reference and dereference relationship

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

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

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

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



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

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

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

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

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

`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`

multi-dimensional arrays

```
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
```

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

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

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

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

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

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

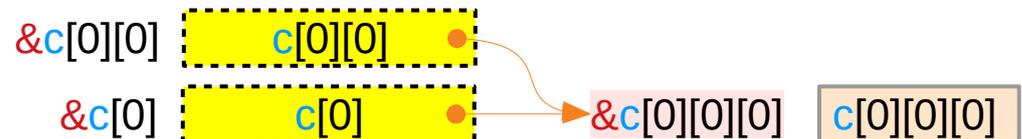
`value(c[0]) = &c[0][0][0]`

`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 $c[0][0]$

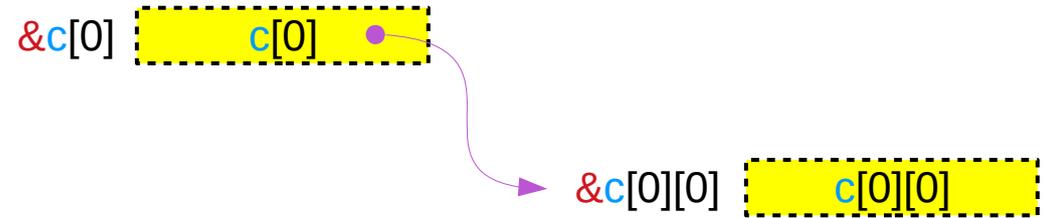
$c[i][j][k];$

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

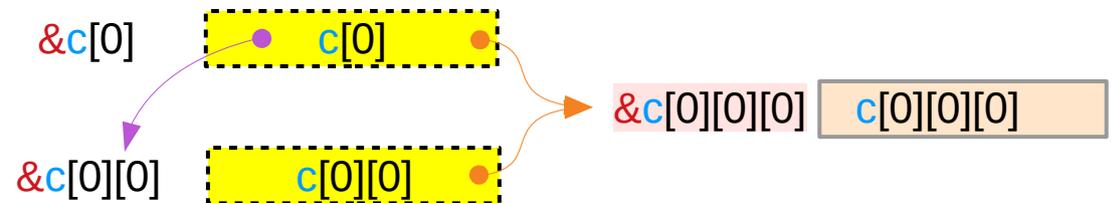
$\&c[i][j][k] = c[i][j] + k$
 $\&c[i][j] = c[i] + j$
 $\&c[i] = c + i$

$\text{int } c[2][3][4];$

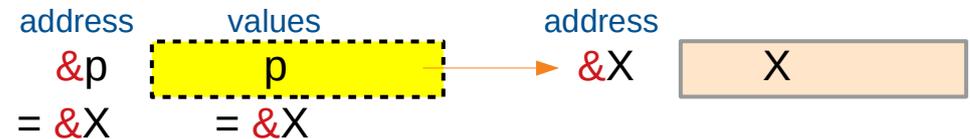
$c \rightarrow c[0] \rightarrow c[0][0] = \&c[0][0][0]$



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



A virtual pointer's address and value are the same



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

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

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

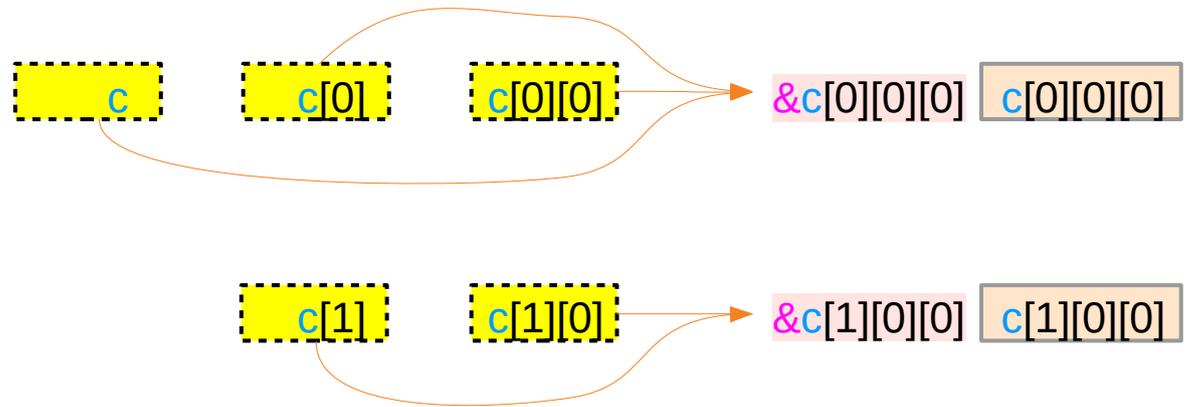
```
&c[i][j][k] = c[i][j]+k
&c[i][j]    = c[i]+j
&c[i]       = c+i
```

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

```
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

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

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

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

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

```
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 *

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

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

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

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

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

equivalences

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

equivalences

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

equivalences

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

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

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

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

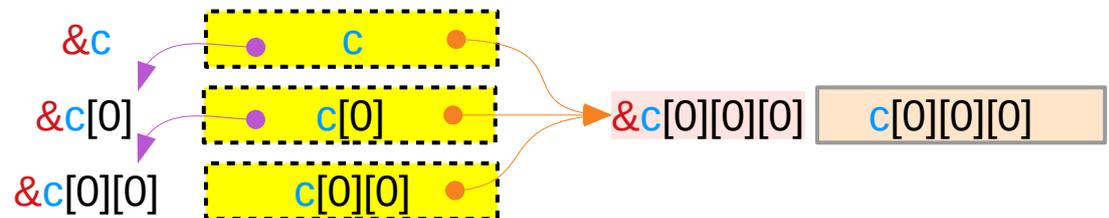
$\&c[i][j][k] = c[i][j] + k$
 $\&c[i][j] = c[i] + j$
 $\&c[i] = c + i$

$\text{int } c[2][3][4];$

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

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

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



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

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

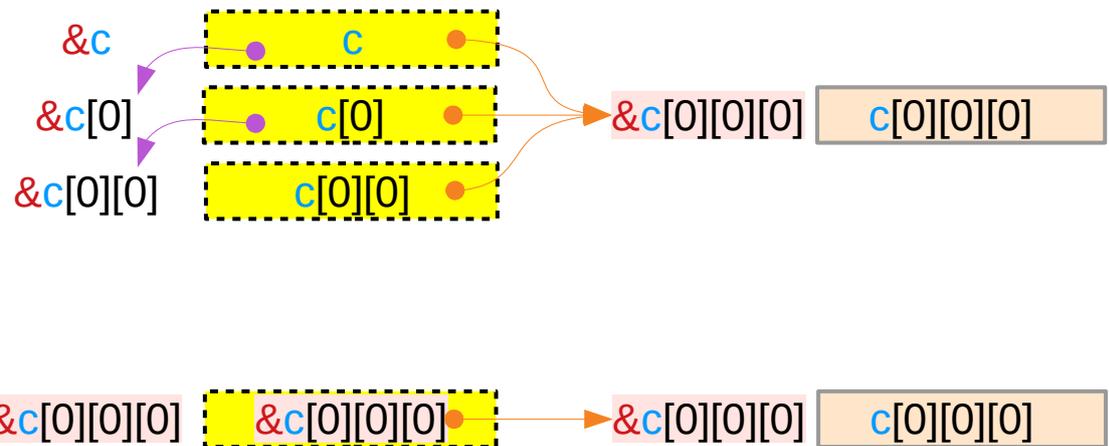
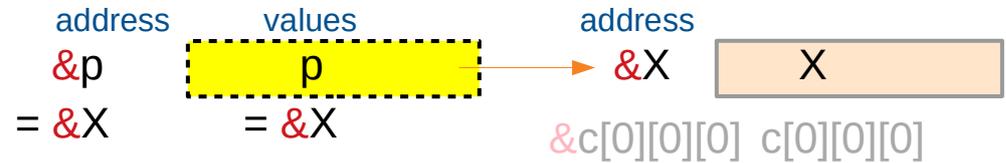
$\&c[i][j][0] = c[i][j]$
 $\&c[i][0] = c[i]$
 $\&c[0] = c$

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

$\text{int } c[2][3][4];$

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

A virtual pointer's address and value are the same



Leading elements and array pointers

`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]`

Array Pointers to `c[i][0][0]`

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

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

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

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

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

`&c[i][0][0] ≡ c[i][0]`

`&c[i][0] ≡ c[i]`

`&c[i] ≡ c+i`

virtual pointers:
 the address of a pointer is
 the same as its value

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

delete [0] from the right

<code>&c[0][0][0]</code>	<u><u>-[0]</u></u>	<code>c[0][0]</code>	<u><u>-[0]</u></u>	<code>c[0]</code>	<u><u>-[0]</u></u>	<code>c</code>
<code>&c[1][0][0]</code>	<u><u>-[0]</u></u>	<code>c[1][0]</code>	<u><u>-[0]</u></u>	<code>c[1]</code>		

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

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

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

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

$int\ c[2][3][4];$

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

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

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

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

delete [0] from the right

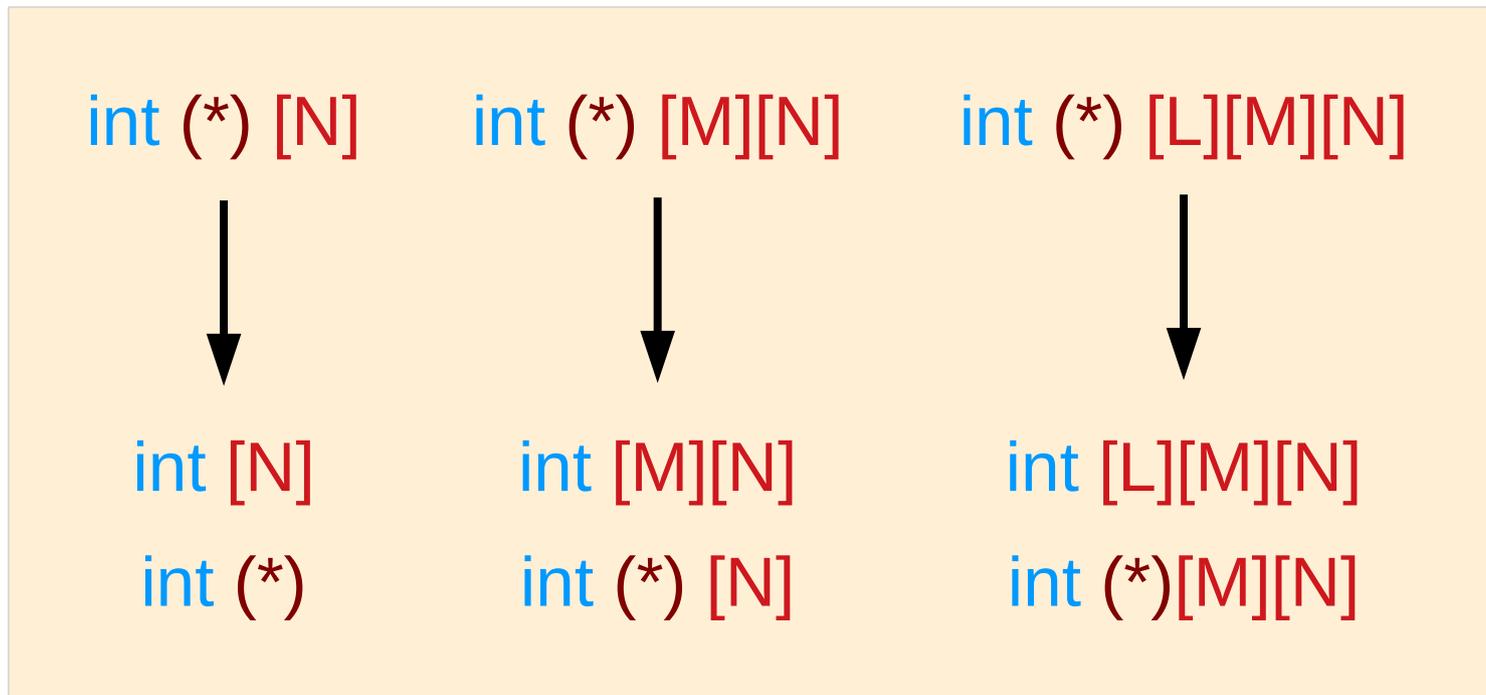
$\&c[0][0][0]$	$\underline{\underline{-[0]}}$	$c[0][0]$	$\underline{\underline{-[0]}}$	$c[0]$	$\underline{\underline{-[0]}}$	c
$\&c[0][1][0]$	$\underline{\underline{-[0]}}$	$c[0][1]$				
$\&c[0][2][0]$	$\underline{\underline{-[0]}}$	$c[0][2]$				
$\&c[1][0][0]$	$\underline{\underline{-[0]}}$	$c[1][0]$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

Contiguity Constraints

c [i][j][k];

Virtual Array Pointers and Contiguity

Using array pointers

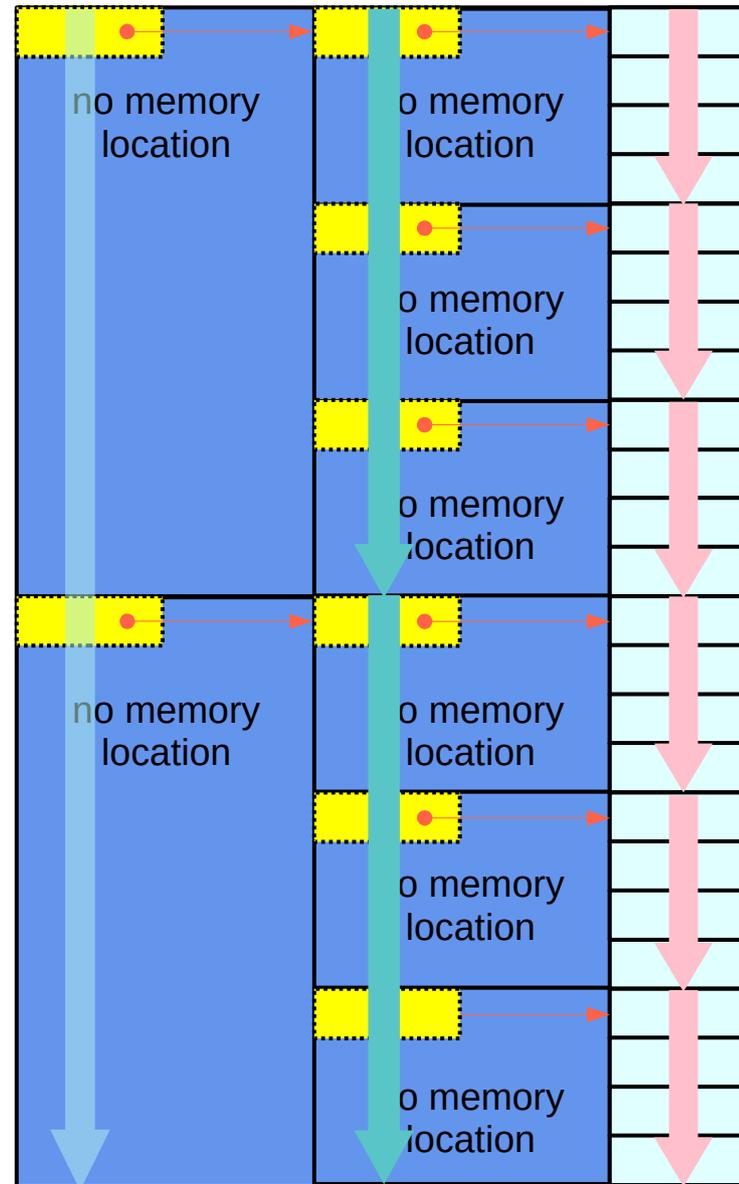


Array Pointer

Array Name

Array Pointer

Array pointer approach – contiguity constraints



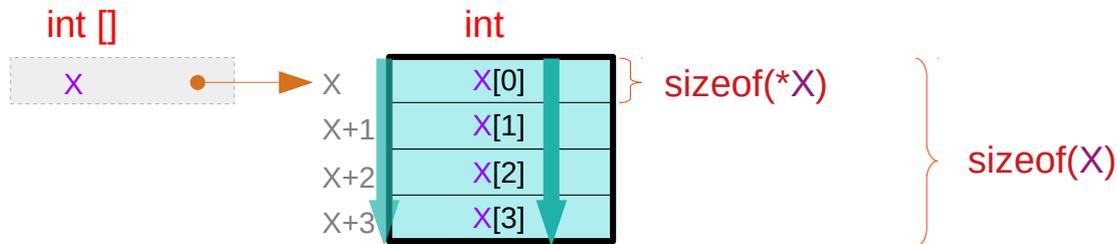
Array **Pointer** Approach
(**pointer to arrays**)

Equivalence and contiguity (1)

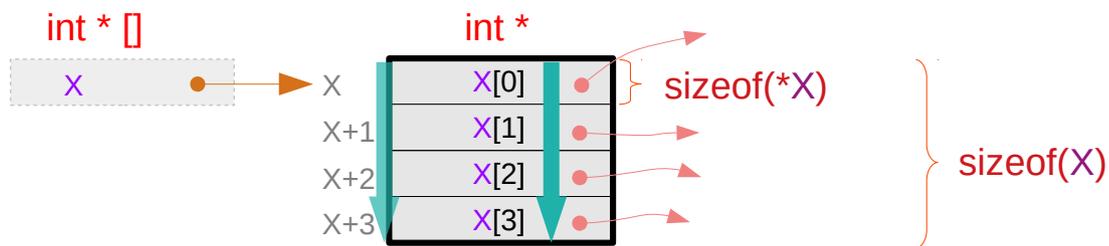
consecutive address consecutive data

$$*(\mathbf{X+n}) \equiv \mathbf{X[n]}$$

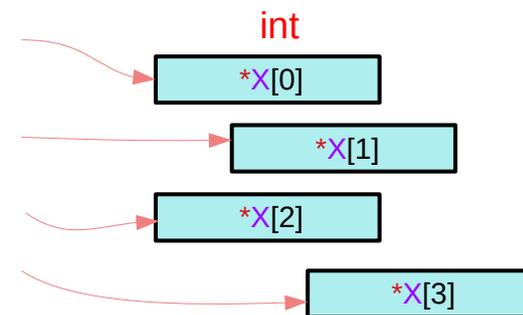
contiguous index : n



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



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



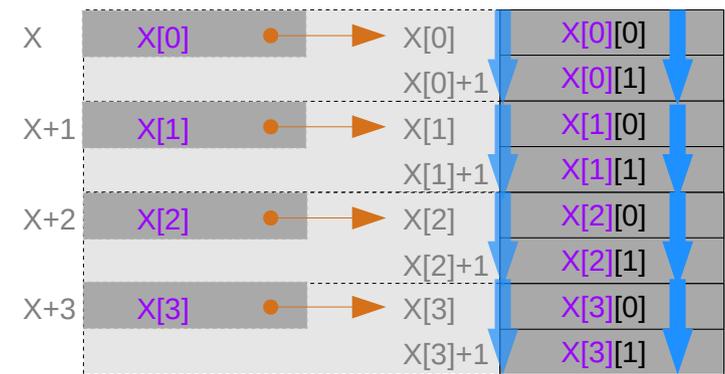
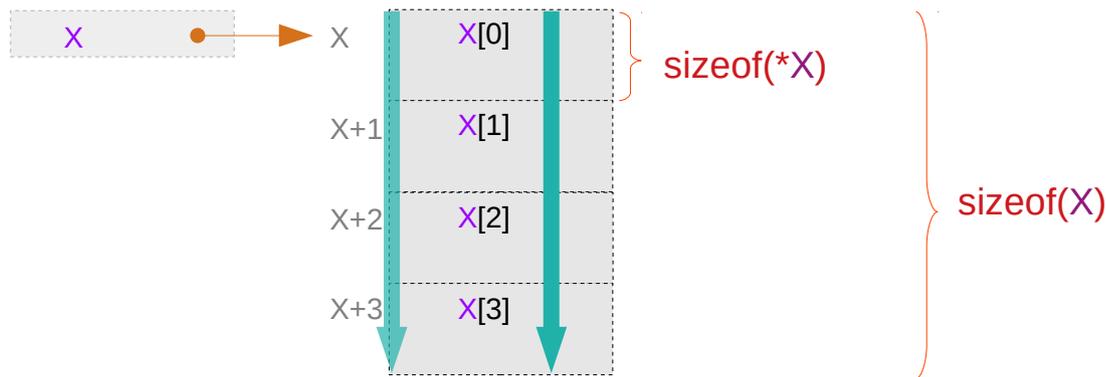
Equivalence and contiguity (2)

consecutive address consecutive data

$$*(\mathbf{X+n}) \equiv \mathbf{X[n]}$$

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

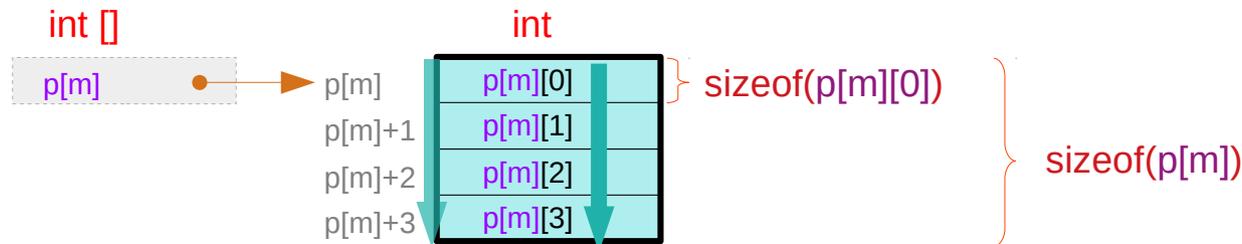
Equivalence for a given p[m] (1)

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

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

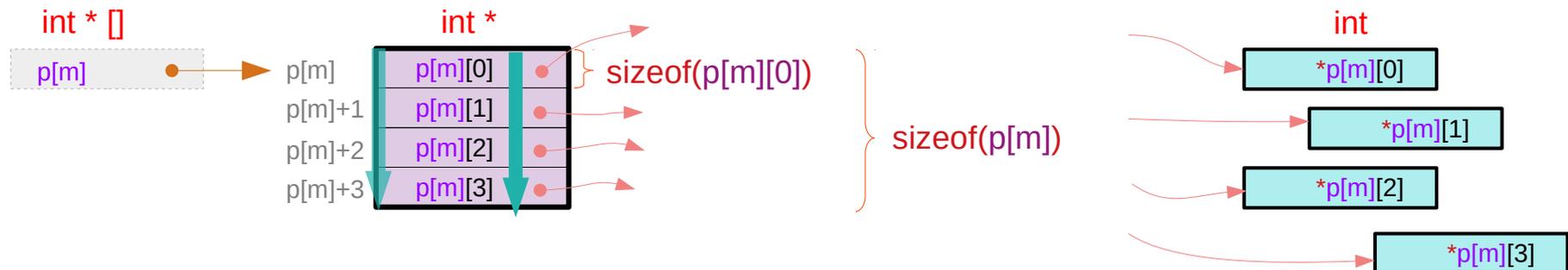
int p[M][4]; contiguous p[m][n] for a given p[m] : **primitive types**

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



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

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

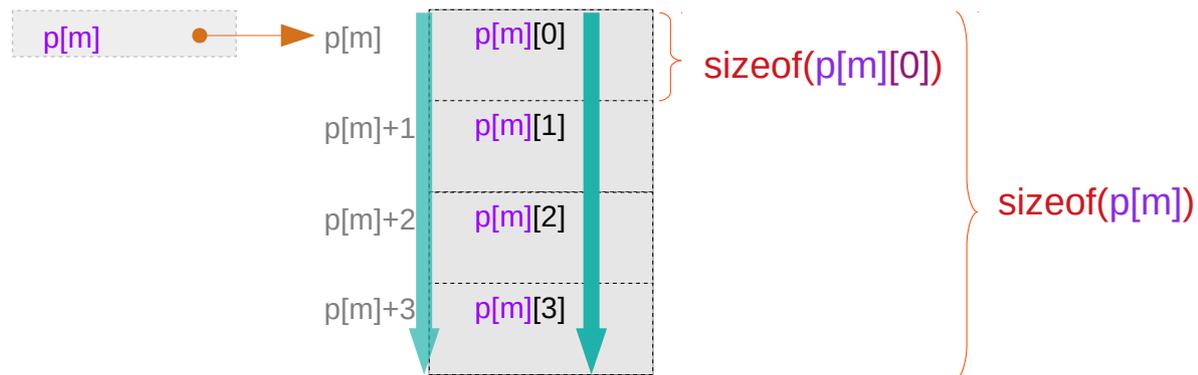


Equivalence for a given p[m] (2)

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

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

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



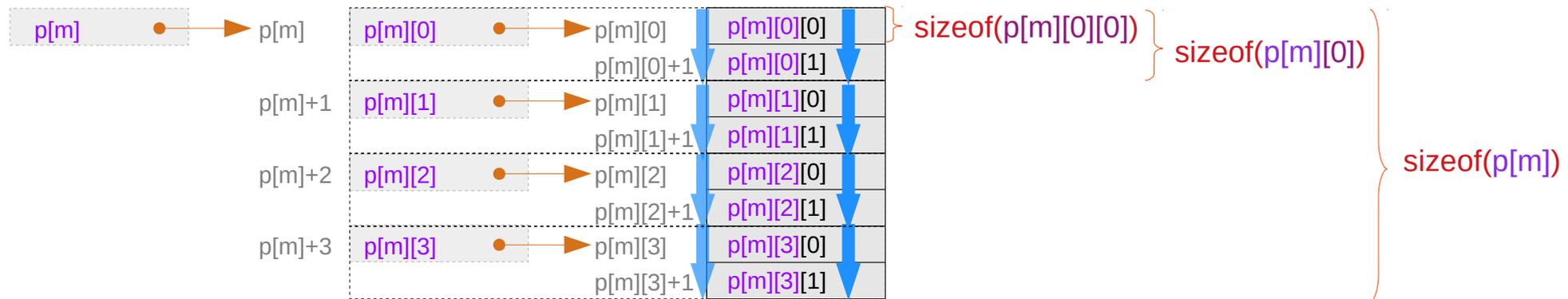
can be recursively applied

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



Contiguity constraints in multi-dimensional arrays

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

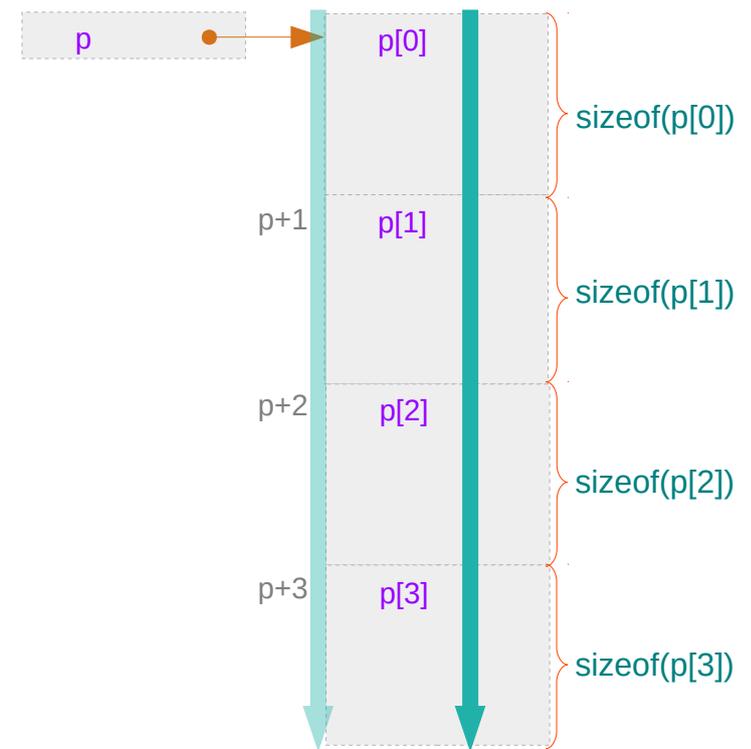
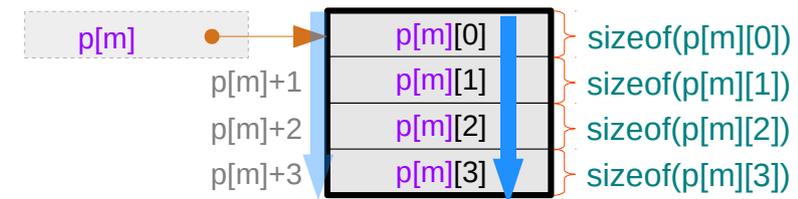
for a given $p[m]$, thus for a given p and m ,
 $p[m][n]$'s must be contiguous for all n .
 $p[m][0], p[m][1], \dots, p[m][N-1]$

contiguous index : n

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

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

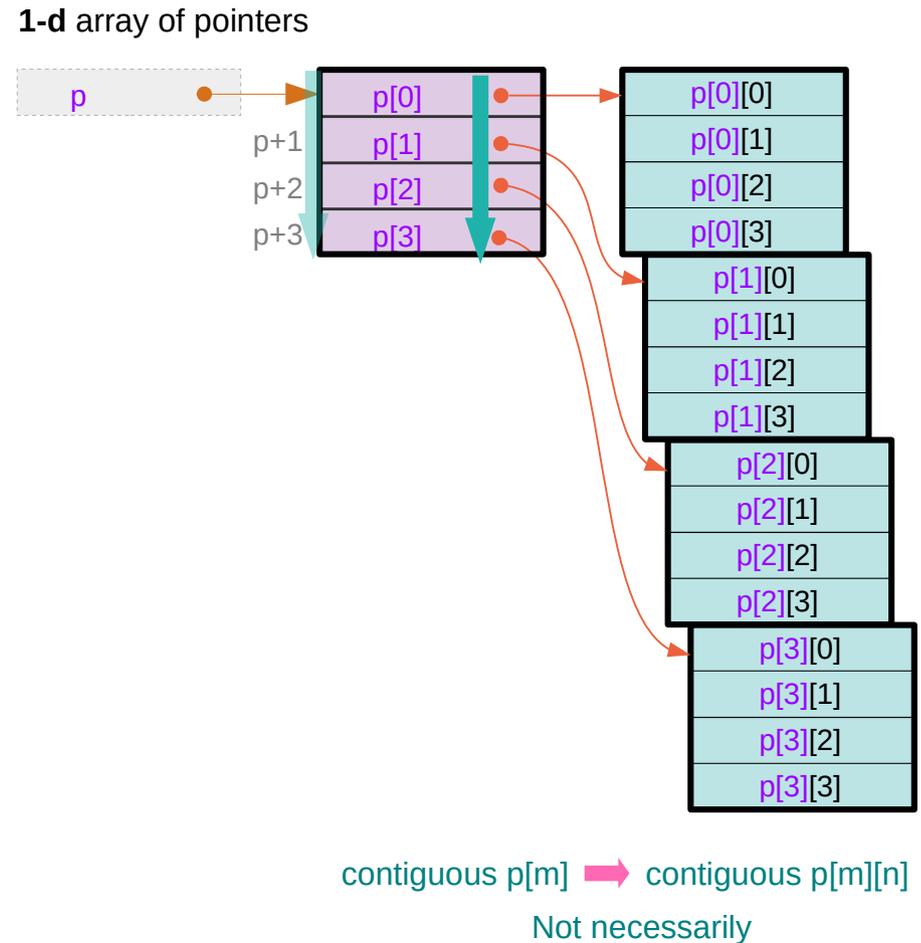
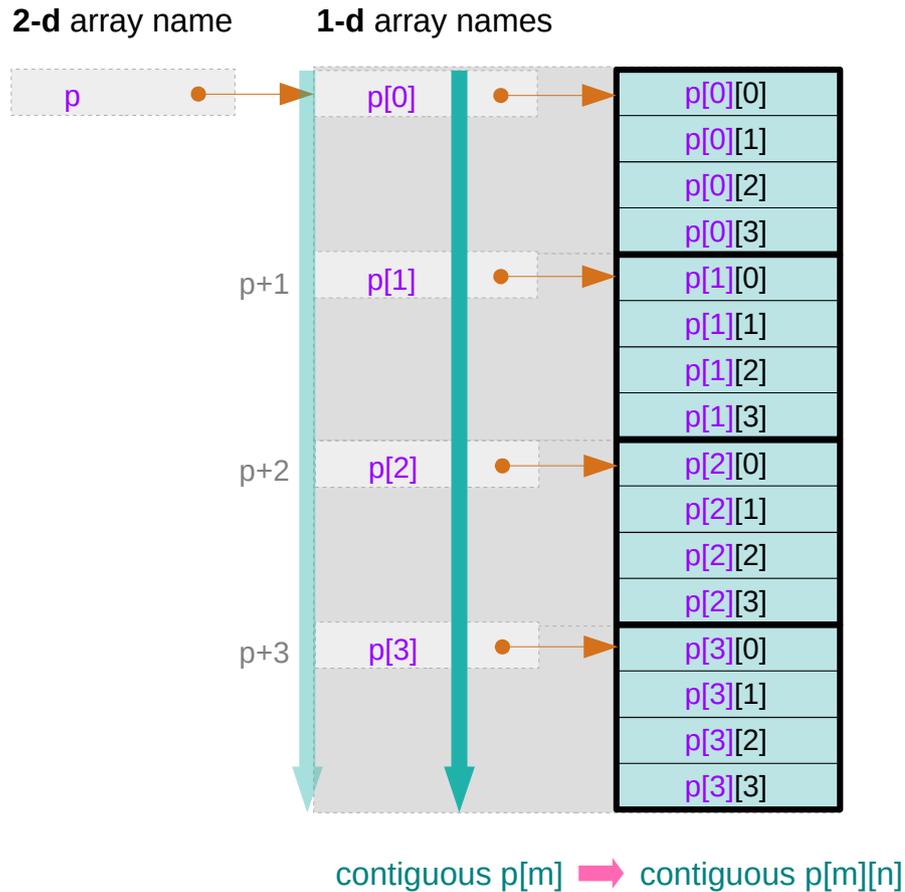
contiguous index : m



Contiguity constraints for p

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

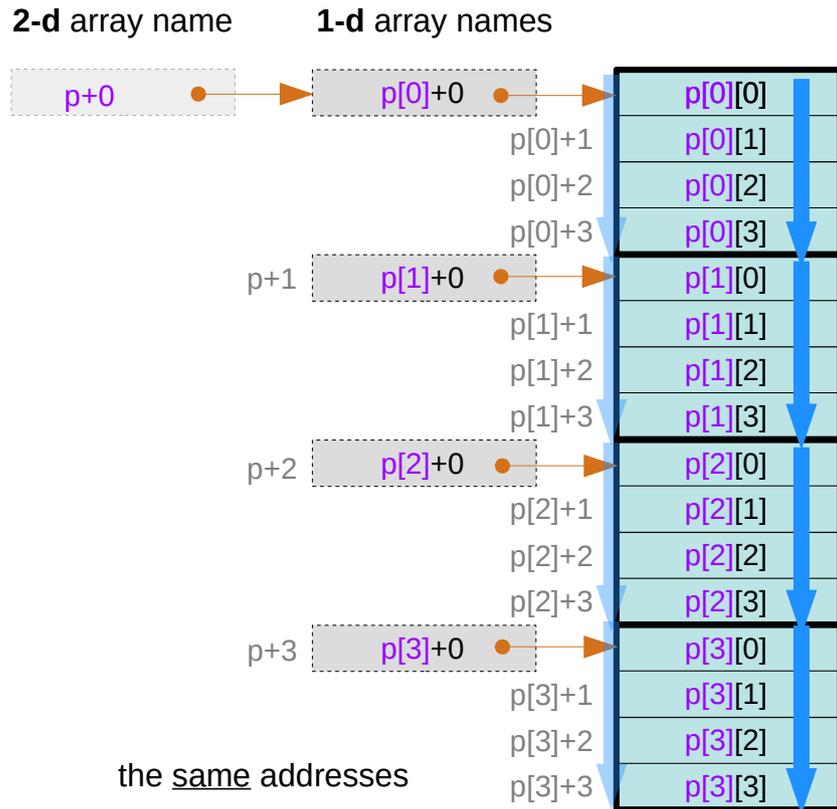
for a given p contiguous index : m



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

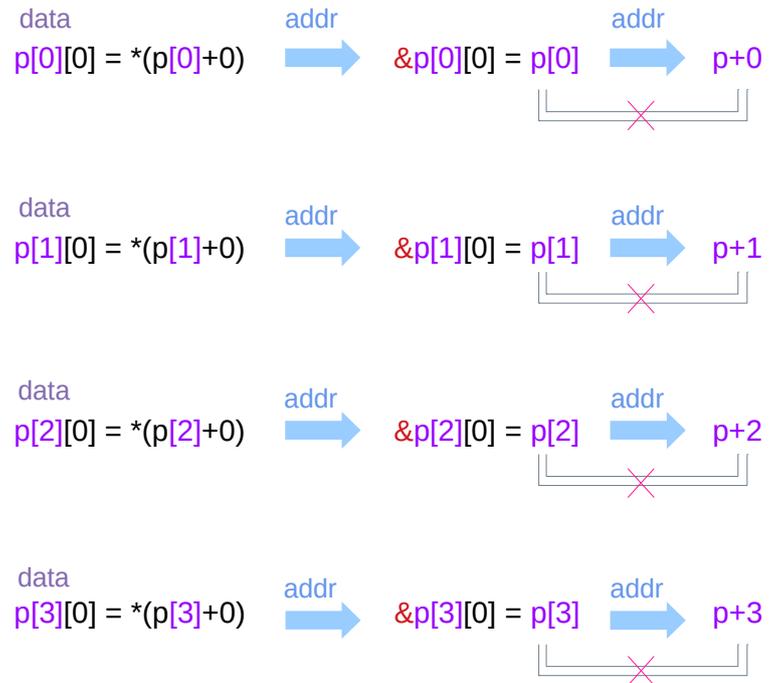
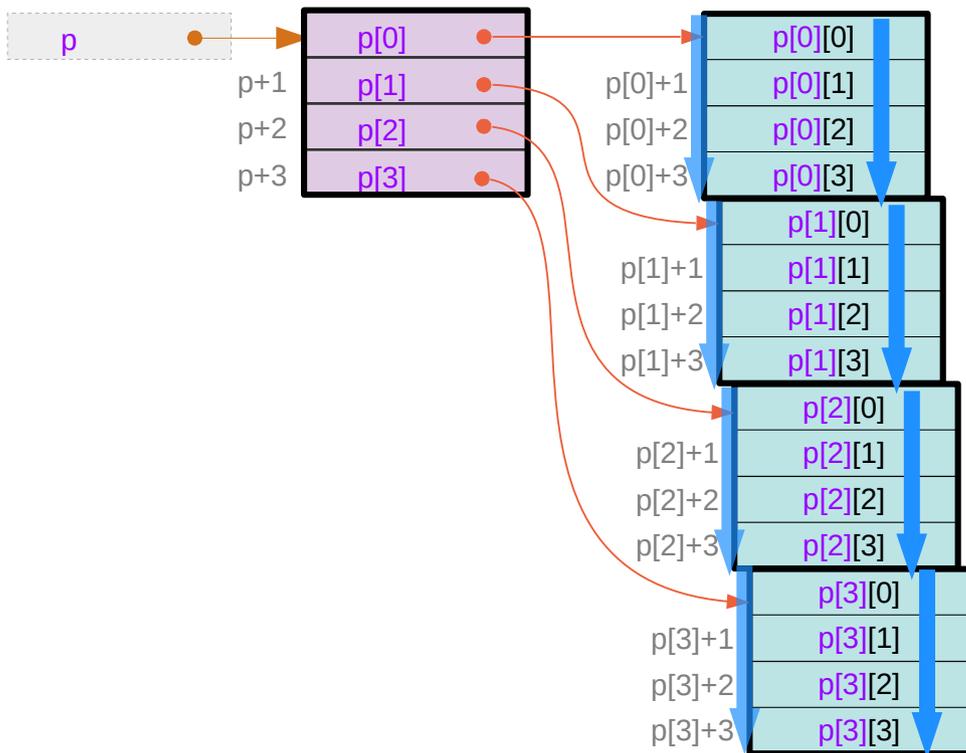
virtual array pointer ↔ no real memory locations

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] ;
```

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

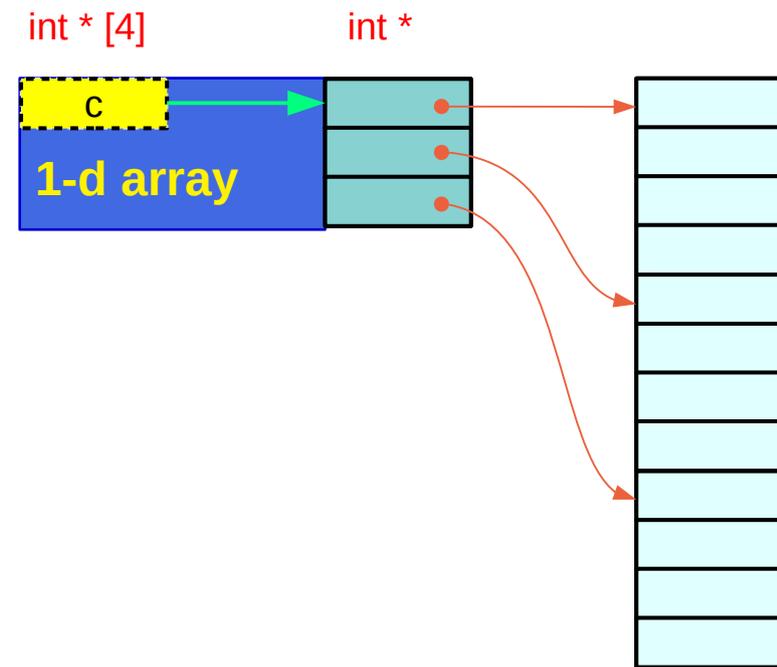
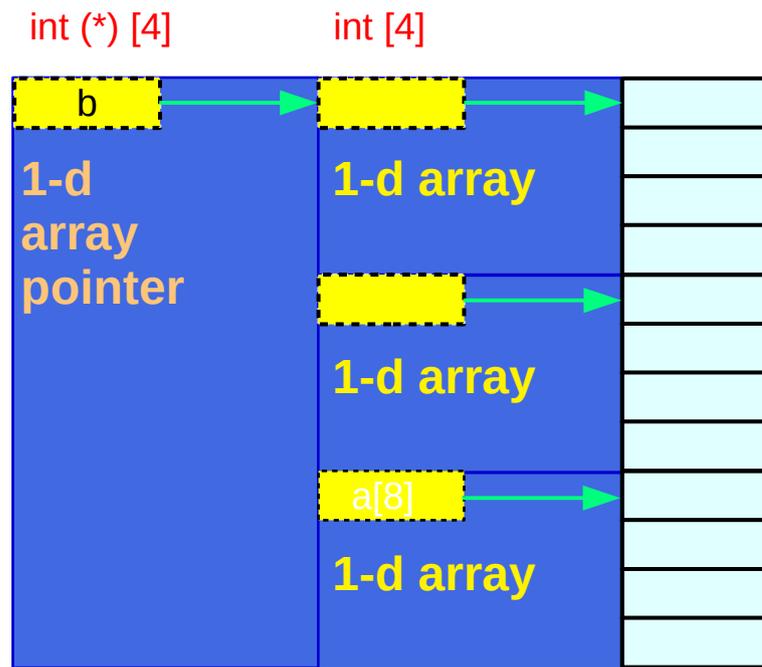
$c[0], c[1], \dots, c[M-1]$
are contiguous

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

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

a set of assignments of pointers
are necessary for this contiguity

Pointer Arrays vs Array Pointers



`int (*b)[N] ;`

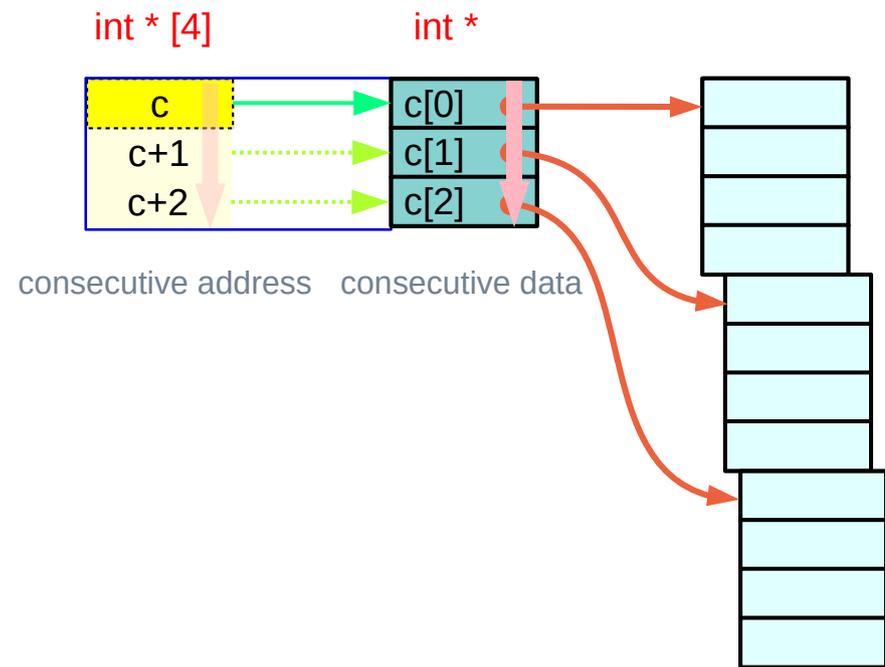
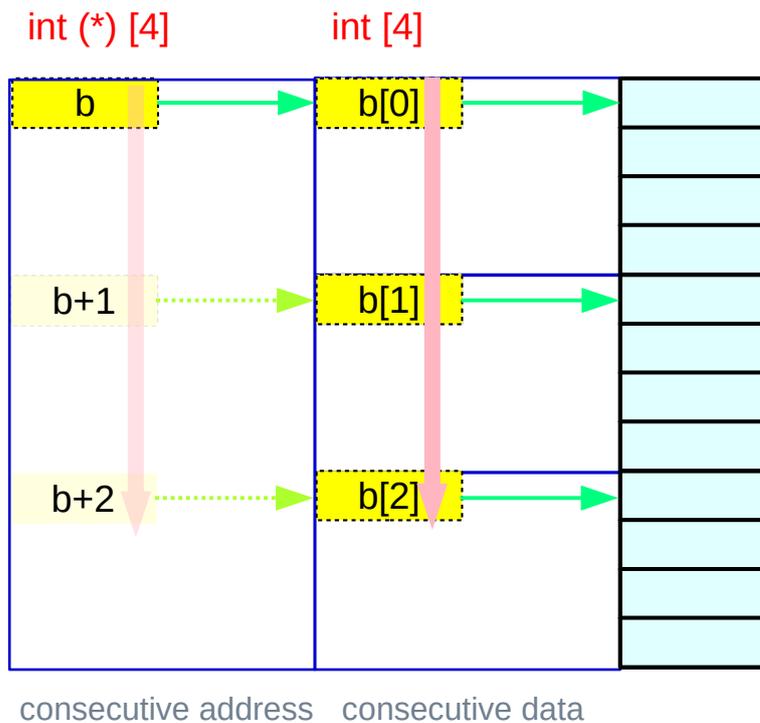
`int * c[M] ;`

with proper assignments

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

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

Pointer Arrays vs Array Pointers



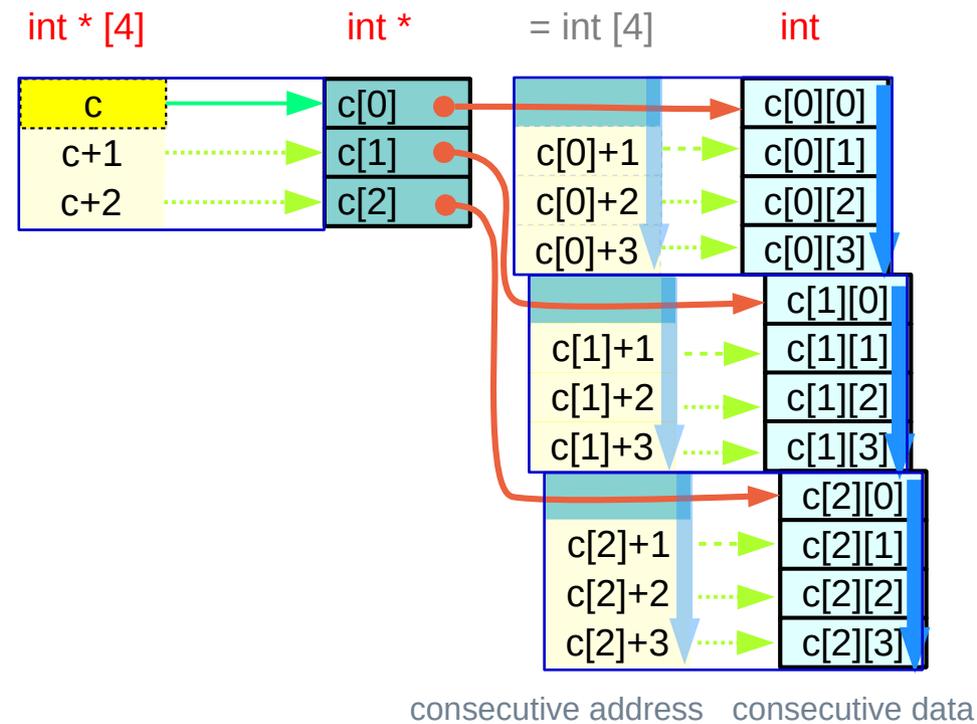
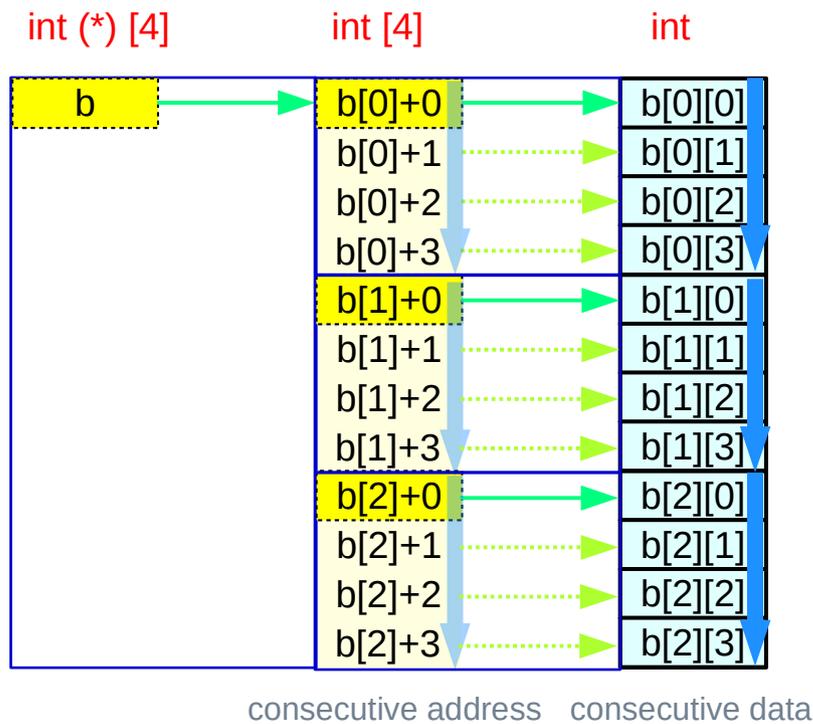
```
int (*b)[N] ;
```

$*(b+m)$ \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] ;`

`*(b+m) ↔ b[m]`
`*(b[m]+n) ↔ b[m][n]`

`int * c[M] ;`

with proper assignments

`*(c+m) ↔ c[m]`
`*(c[m]+n) ↔ c[m][n]`

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)

```

• •
• •
• •

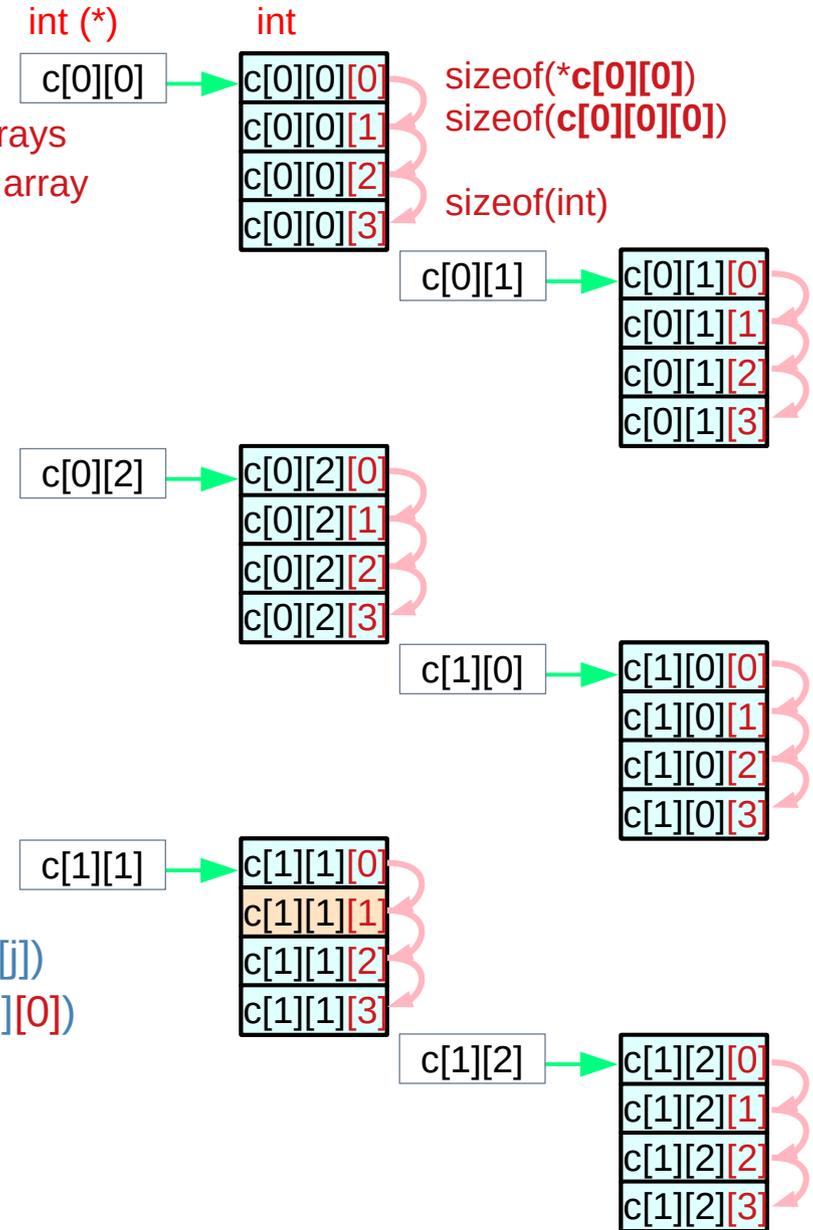
contiguous 1-d
array elements

$c[i][j]$
int [4] 4 contiguous 0-d arrays
int * points to the 1st 0-d array
int 0-d array

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

```
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 1st 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];
```

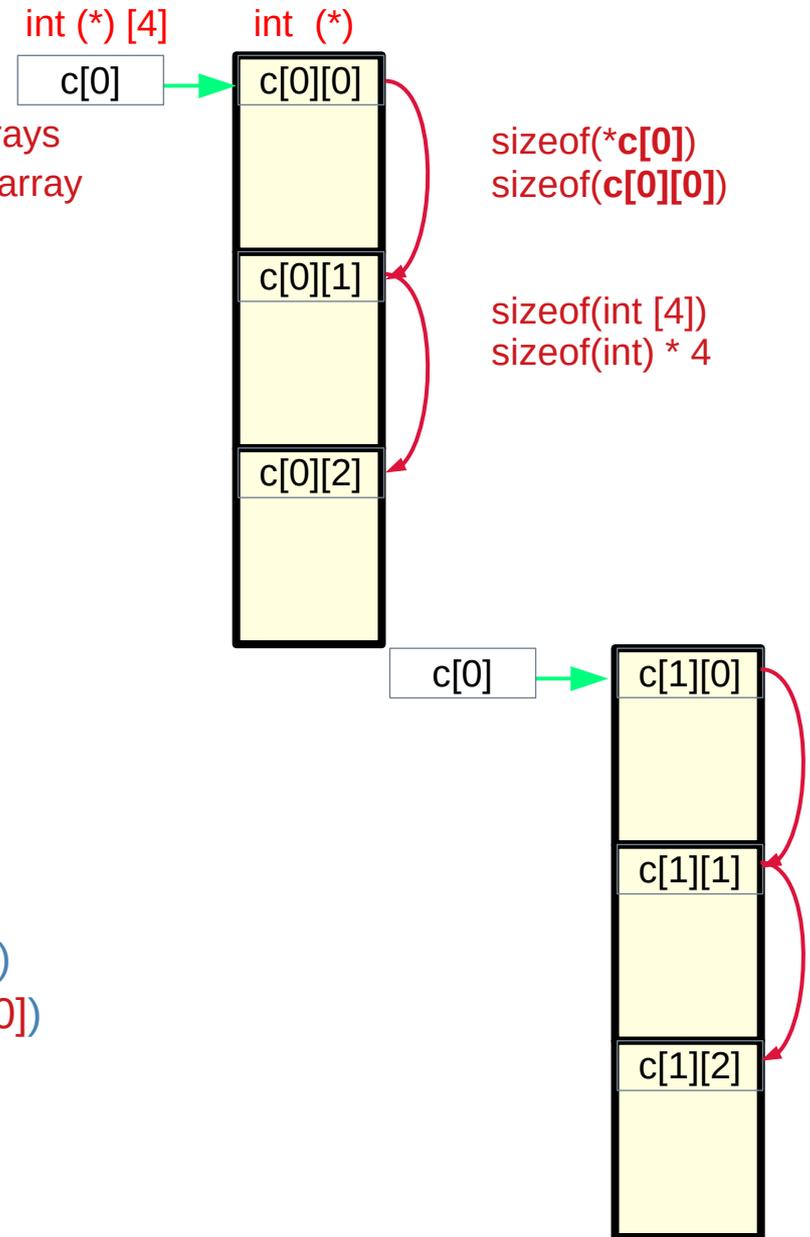
Address Value

$c[i] + j$

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

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

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



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 1st 2-d array
2-d array

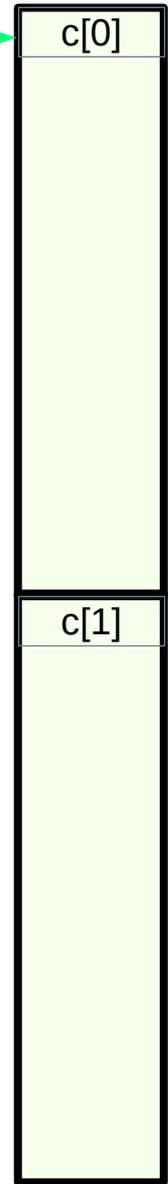
```
sizeof(c)           [i] [j] [k]
sizeof(c[i][j][k]) * 2 * 3 * 4
sizeof(int) * 2 * 3 * 4
```

```
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
```

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



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

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

Subarrays in a 3-d array

1-d subarrays

int [4]	int
c[0][0]	c[0][0][0]
	c[0][0][1]
	c[0][0][2]
	c[0][0][3]
c[0][1]	c[0][1][0]
	c[0][1][1]
	c[0][1][2]
	c[0][1][3]
c[0][2]	c[0][2][0]
	c[0][2][1]
	c[0][2][2]
	c[0][2][3]
c[1][0]	c[1][0][0]
	c[1][0][1]
	c[1][0][2]
	c[1][0][3]
c[1][1]	c[1][1][0]
	c[1][1][1]
	c[1][1][2]
	c[1][1][3]
c[1][2]	c[1][2][0]
	c[1][2][1]
	c[1][2][2]
	c[1][2][3]

2-d subarrays

int [3][4]	int [4]	int	
c[0]	c[0][0]	c[0][0][0]	
		c[0][0][1]	
		c[0][0][2]	
		c[0][0][3]	
	c[0][1]	c[0][1]	c[0][1][0]
			c[0][1][1]
			c[0][1][2]
			c[0][1][3]
	c[0][2]	c[0][2]	c[0][2][0]
			c[0][2][1]
			c[0][2][2]
			c[0][2][3]
c[1]	c[1][0]	c[1][0][0]	
		c[1][0][1]	
		c[1][0][2]	
		c[1][0][3]	
	c[1][1]	c[1][1]	c[1][1][0]
			c[1][1][1]
			c[1][1][2]
			c[1][1][3]
	c[1][2]	c[1][2]	c[1][2][0]
			c[1][2][1]
			c[1][2][2]
			c[1][2][3]

3-d array

int [2][3][4]	int [3][4]	int [3]	int	
c	c[0]	c[0][0]	c[0][0][0]	
			c[0][0][1]	
			c[0][0][2]	
			c[0][0][3]	
	c[0][1]	c[0][1]	c[0][1]	c[0][1][0]
				c[0][1][1]
				c[0][1][2]
				c[0][1][3]
	c[0][2]	c[0][2]	c[0][2]	c[0][2][0]
				c[0][2][1]
				c[0][2][2]
				c[0][2][3]
c[1]	c[1]	c[1][0]	c[1][0][0]	
			c[1][0][1]	
			c[1][0][2]	
			c[1][0][3]	
	c[1][1]	c[1][1]	c[1][1]	c[1][1][0]
				c[1][1][1]
				c[1][1][2]
				c[1][1][3]
	c[1][2]	c[1][2]	c[1][2]	c[1][2][0]
				c[1][2][1]
				c[1][2][2]
				c[1][2][3]

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

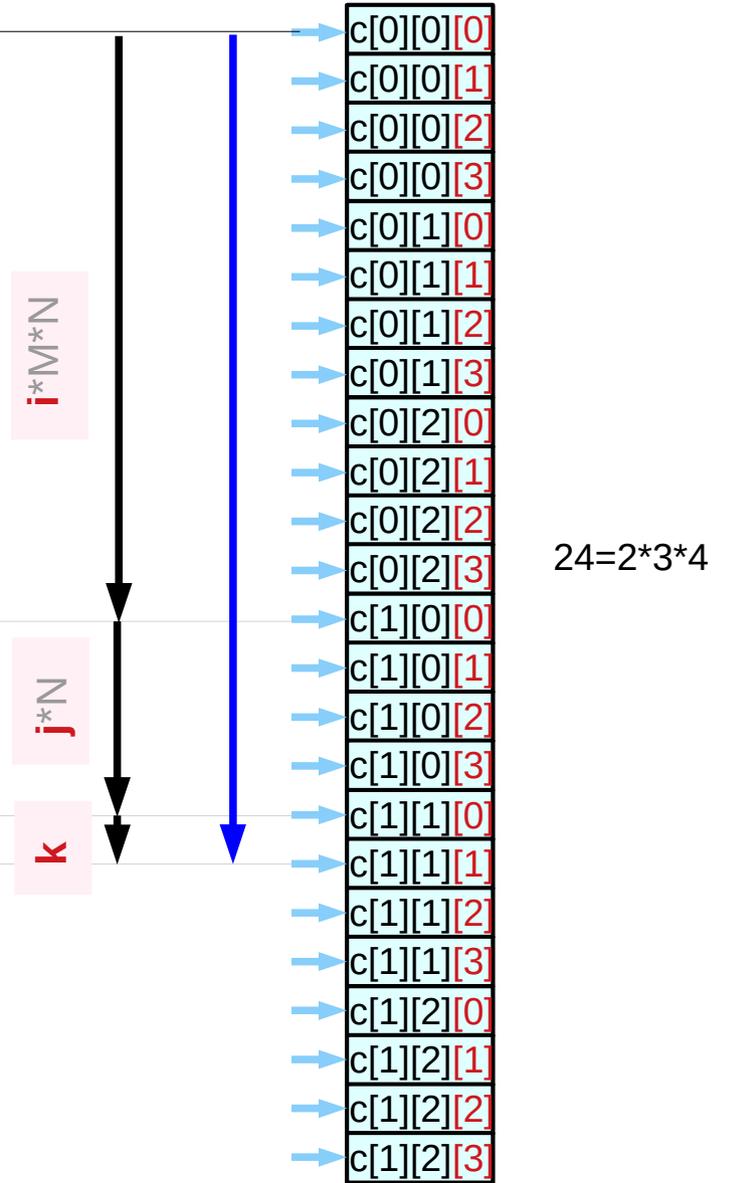
Offset Index 1 (i=1)

Offset Index 2 (j=1)

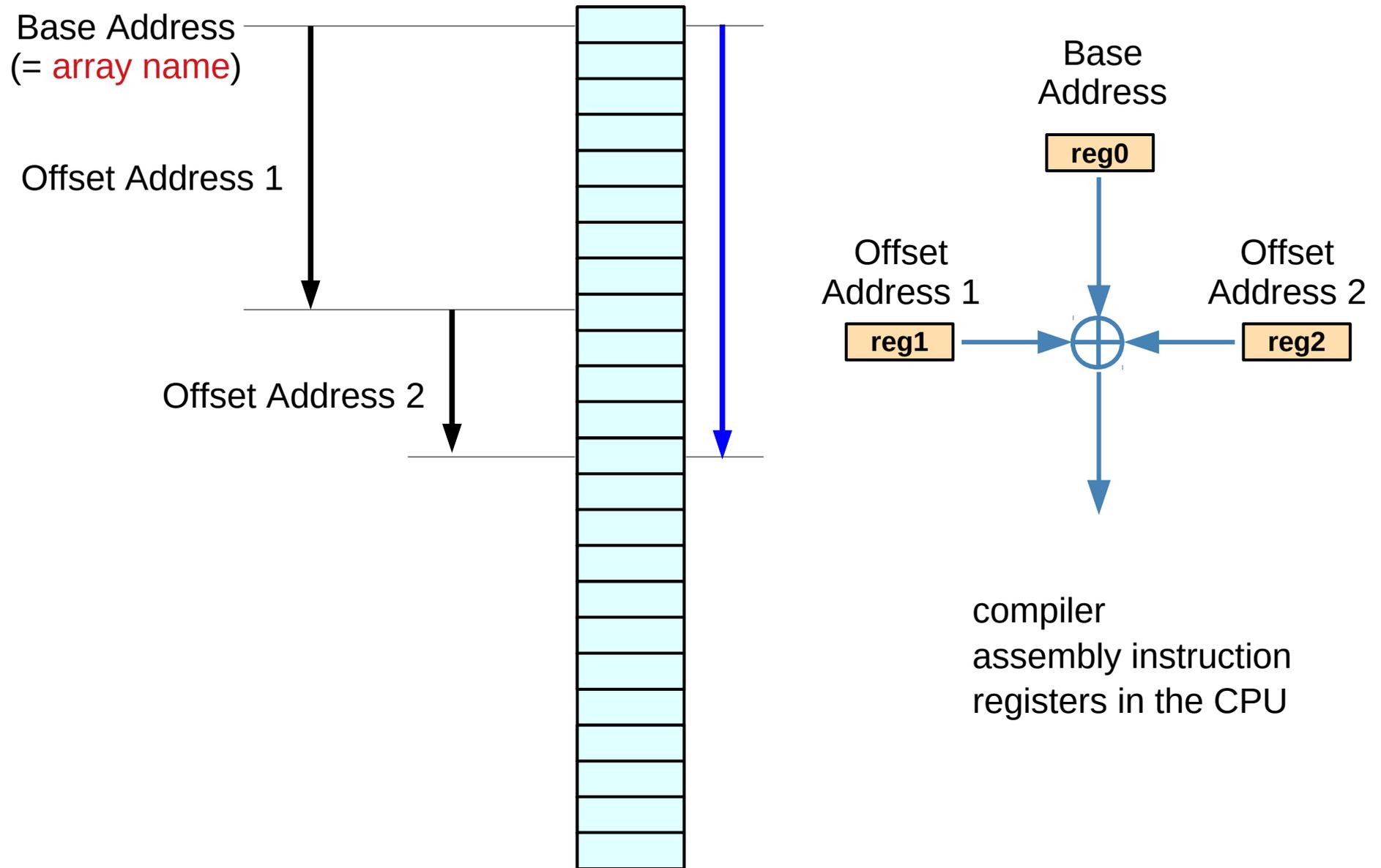
Offset Index 3 (k=1)

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

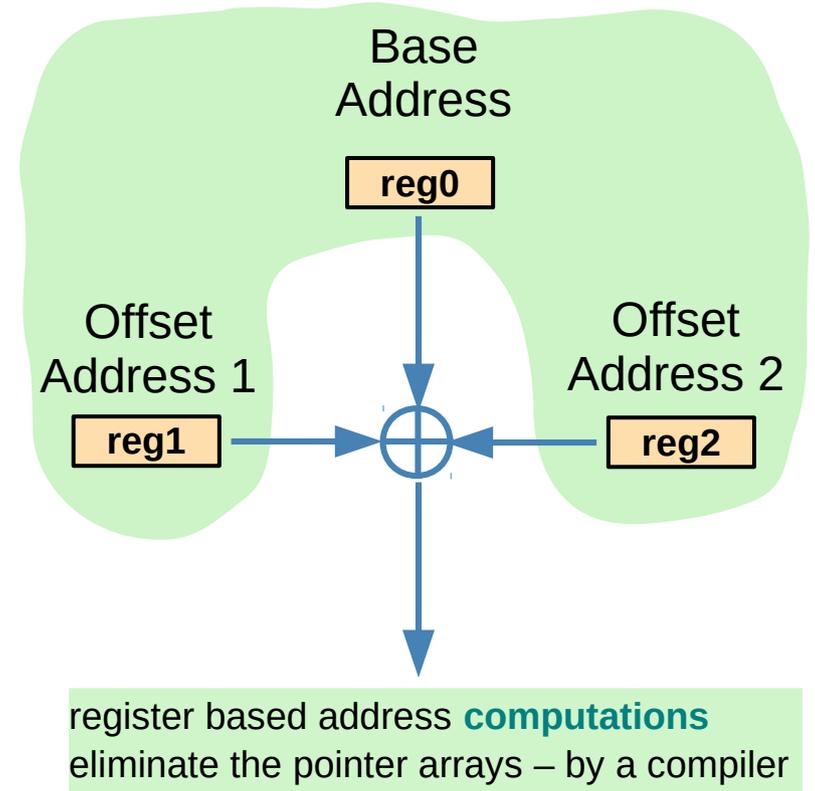
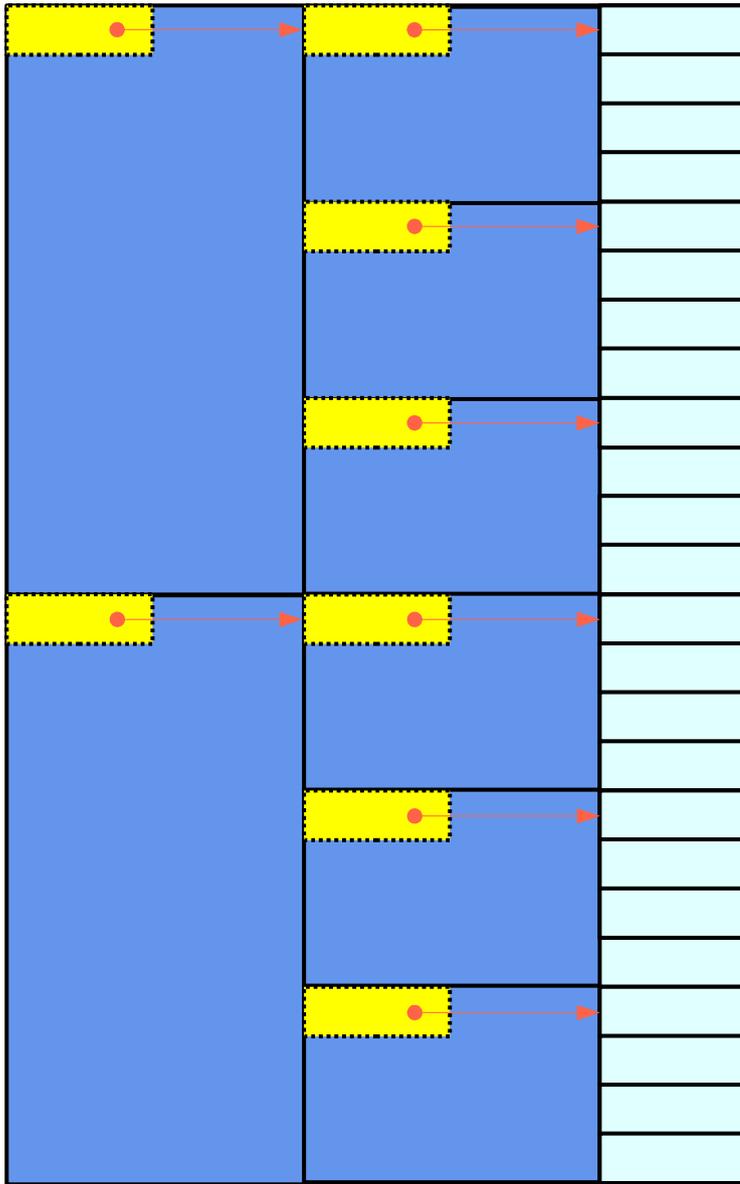
$$((i * M + j) * N + k)$$



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