

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.

Multi-dimensional Array Pointers

$(n-1)$ -d array pointer to a n -d array

`int a[4];` **1-d** array
`int (*p);` **0-d** array pointer ($p = a$)

`int b[4][2];` **2-d** array
`int (*q)[2];` **1-d** array pointer ($q = b$)

`int c[4][2][3];` **3-d** array
`int (*r)[2][3];` **2-d** array pointer ($r = c$)

`int d[4][2][3][4];` **4-d** array
`int (*s)[2][3][4];` **3-d** array pointer ($s = d$)

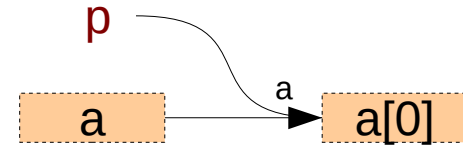


the 1st dimension can be accessed by incrementing $(n-1)$ -d array pointer

n -d array name and $(n-1)$ -d array pointer

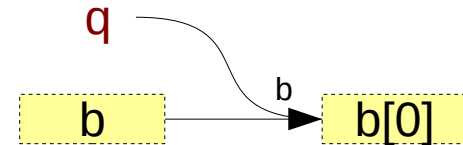
```
int a[4];  
int (*p);
```

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



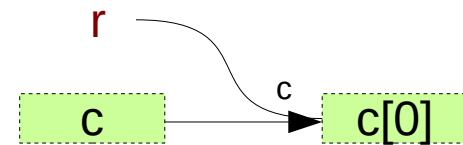
```
int b[4][2];  
int (*q)[2];
```

```
q = &b[0];  
q = b;
```



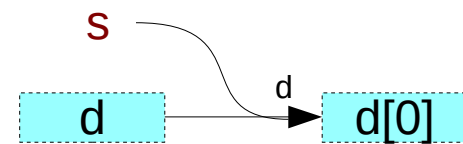
```
int c[4][2][3];  
int (*r)[2][3];
```

```
r = &c[0];  
r = c;
```



```
int d[4][2][3][4];  
int (*s)[2][3][4];
```

```
s = &d[0];  
s = d;
```



the 1st dimension can be accessed by incrementing $(n-1)$ -d array pointer

n-d array pointer to a *n*-d array

`int a [4] ;` **1-d** array
`int (*p) [4];` **1-d** array pointer (`p = &a`)

`int b [4][2];` **2-d** array
`int (*q) [4][2];` **2-d** array pointer (`q = &b`)

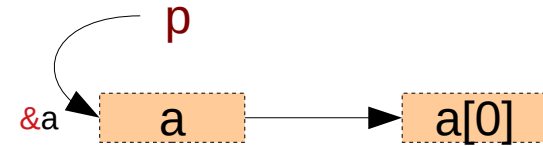
`int c [4][2][3];` **3-d** array
`int (*r) [4][2][3];` **3-d** array pointer (`r = &c`)

`int d [4][2][3][4];` **4-d** array
`int (*s) [4][2][3][4];` **4-d** array pointer (`s = &d`)

n-d array name and *n*-d array pointer

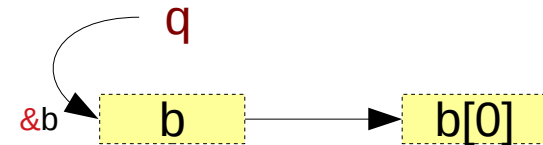
```
int a [4];  
int (*p) [4];
```

```
p = &a;
```



```
int b [4][2];  
int (*q) [4][2];
```

```
q = &b;
```



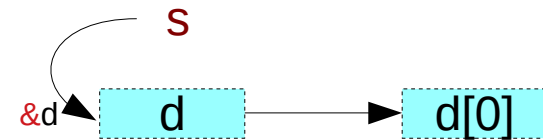
```
int c [4][2][3];  
int (*r) [4][2][3];
```

```
r = &c;
```

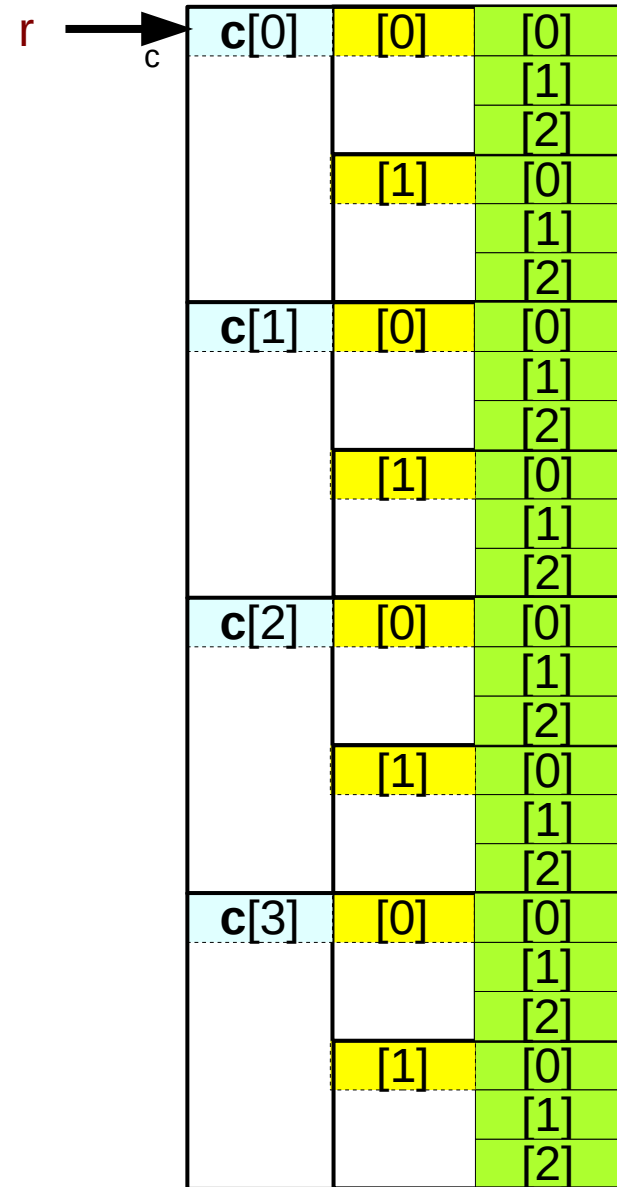
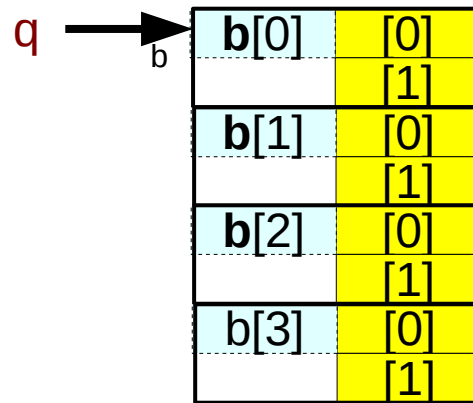
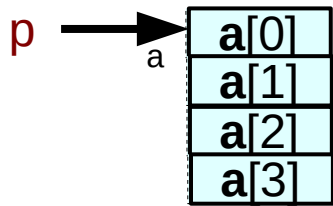


```
int d [4][2][3][4];  
int (*s) [4][2][3][4];
```

```
s = &d;
```

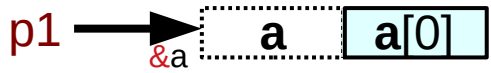


multi-dimensional array pointers

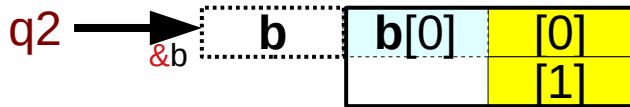


- `int a[4];` **1-d array**
- `int (*p);` **0-d array pointer**
- `int b[4][2];` **2-d array**
- `int (*q)[2];` **1-d array pointer**
- `int c[4][2][3];` **3-d array**
- `int (*r)[2][3];` **2-d array pointer**
- `int d[4][2][3][4];` **4-d array**
- `int (*s)[2][3][4];` **3-d array pointer**

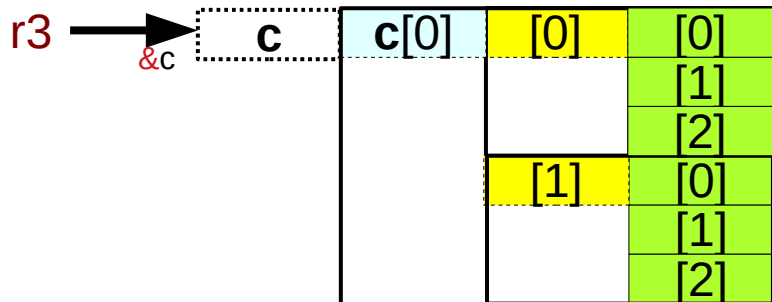
Initializing *n-d* array pointers



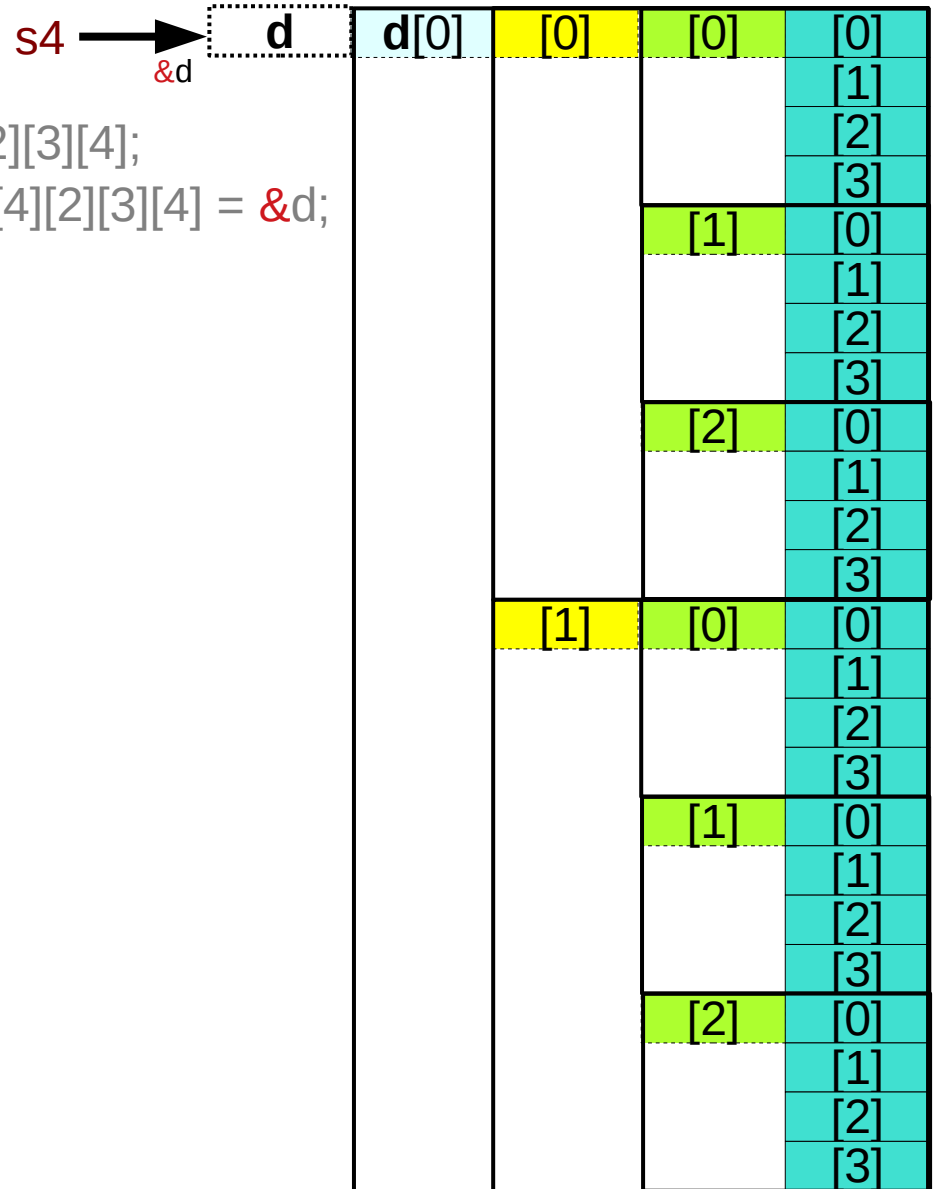
```
int a[4];
int (*p1)[4] = &a;
```



```
int b[4][2];
int (*q2)[4][2] = &b;
```

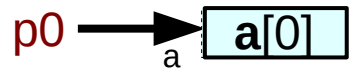


```
int c[4][2][3];
int (*r3)[4][2][3] = &c;
```

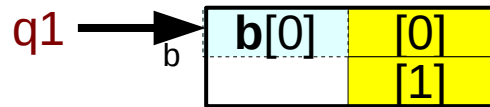


```
int d[4][2][3][4];
int (*s4)[4][2][3][4] = &d;
```

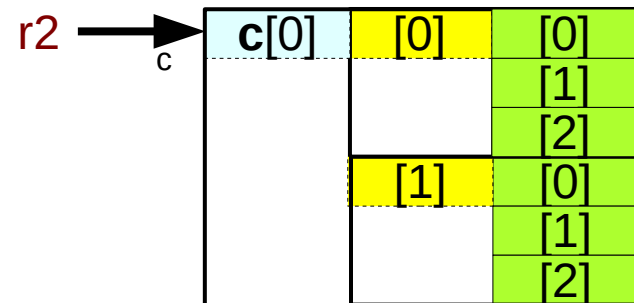
Initializing $(n-1)$ -d array pointers



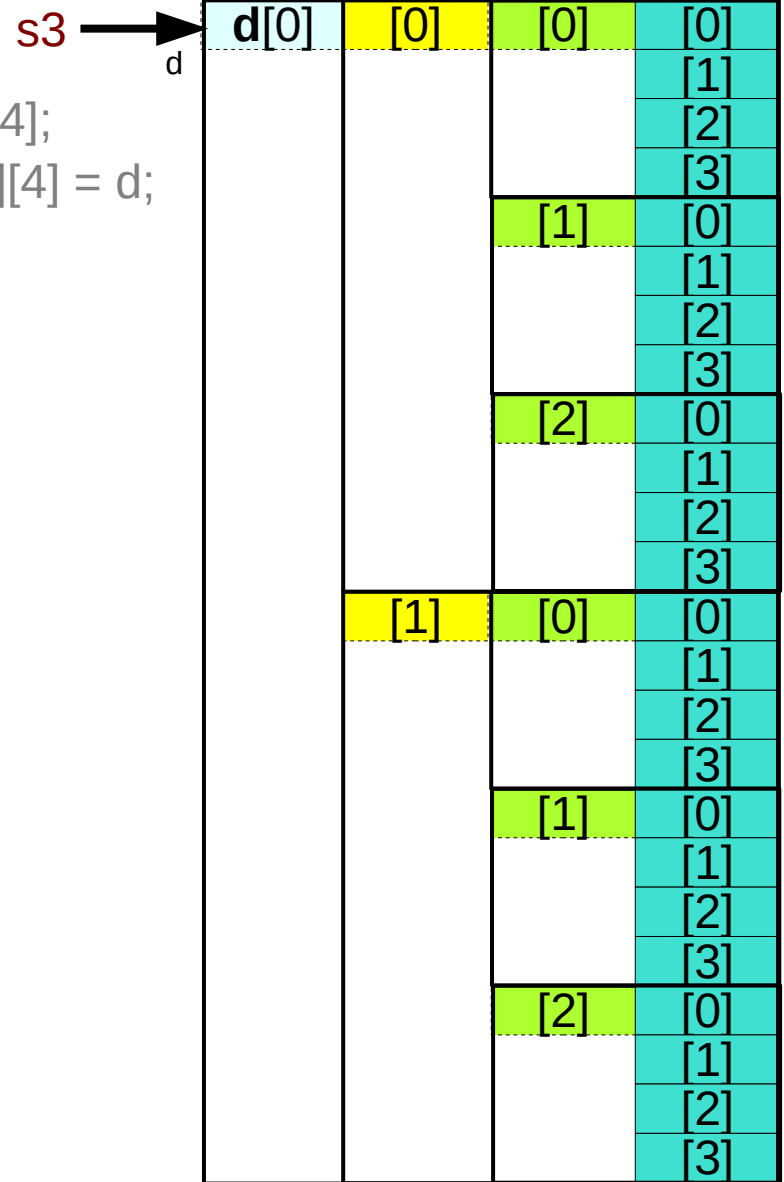
```
int a[4];
int (*p0) = a;
```



```
int b[4][2];
int (*q1)[2] = b;
```



```
int c[4][2][3];
int (*r2)[2][3] = c;
```



```
int d[4][2][3][4];
int (*s3)[2][3][4] = d;
```

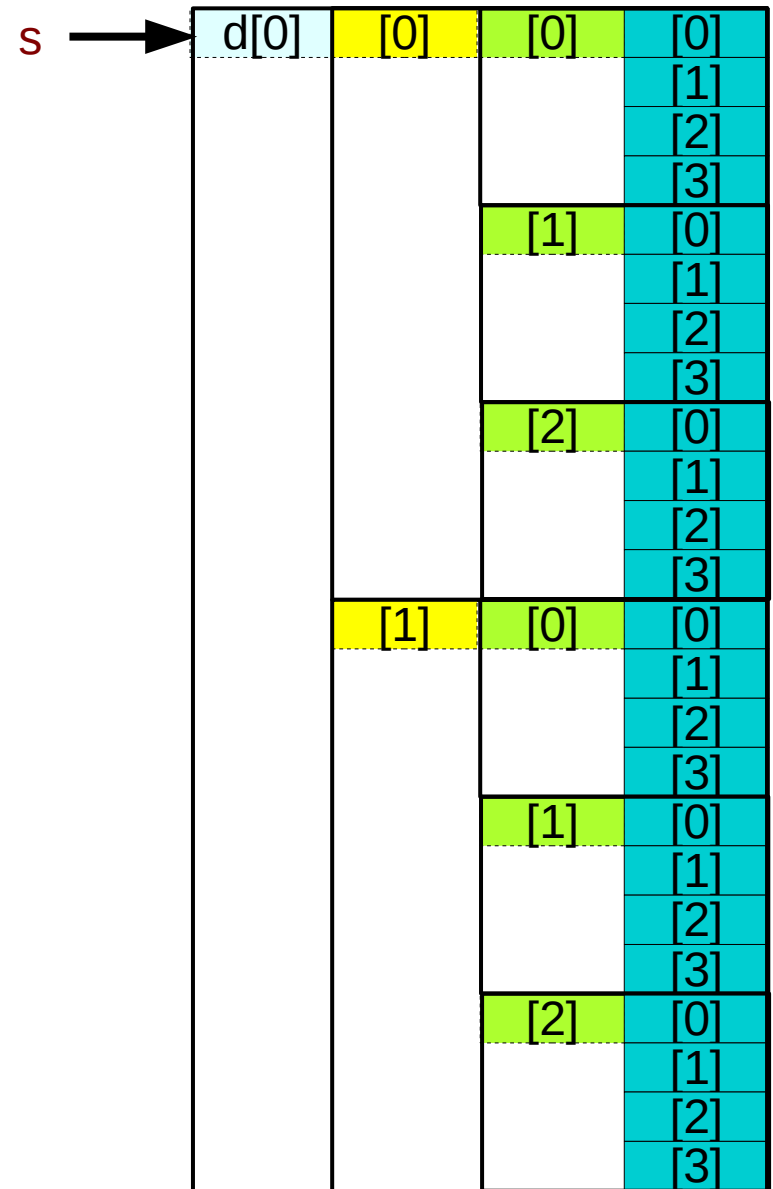
array pointers to multi-dimensional subarrays

```
int d[4][2][3][4];
int (*s)[2][3][4];
```

d	4-d array name	d[4][2][3][4]
	3-d array pointer	(*p)[2][3][4]
d[i]	3-d array name	d[i][2][3][4]
	2-d array pointer	(*q)[3][4]
d[i][j]	2-d array name	d[i][j][3][4]
	1-d array pointer	(*r)[4]
d[i][j][k]	1-d array name	d[i][j][k][4]
	0-d array pointer	(*s)

i,j,k are specific index values

i = [0..3], j = [0..1], k = [0..2]



Initializing array pointers to multi-dimensional subarrays

```
int d[4][2][3][4];  
int (*s)[2][3][4];
```

<code>d</code>	4-d array name 3-d array pointer	<code>d[4][2][3][4]</code> <code>(*p)[2][3][4]</code>	<code>p[i][j][k][l]</code> <code>int (*p)[2][3][4] = d;</code>
<code>d[i]</code>	3-d array name 2-d array pointer	<code>d[i][2][3][4]</code> <code>(*q)[3][4]</code>	<code>q[j][k][l]</code> <code>int (*q)[3][4] = d[i];</code>
<code>d[i][j]</code>	2-d array name 1-d array pointer	<code>d[i][j][3][4]</code> <code>(*r)[4]</code>	<code>r[k][l]</code> <code>int (*r)[4] = d[i][j];</code>
<code>d[i][j][k]</code>	1-d array name 0-d array pointer	<code>d[i][j][k][4]</code> <code>(*s)</code>	<code>s[l]</code> <code>int (*s) = d[i][j][k];</code>

`i = [0..3], j = [0..1], k = [0..2]`

Passing multidimensional array names

```
int a[4];  
int (*p);
```

call
funa(a, ...);

prototype
void **fun**a(int (*p), ...);

```
int b[4][2];  
int (*q)[2];
```

call
funb(b, ...);

prototype
void **fun**b(int (*q)[2], ...);

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

call
func(c, ...);

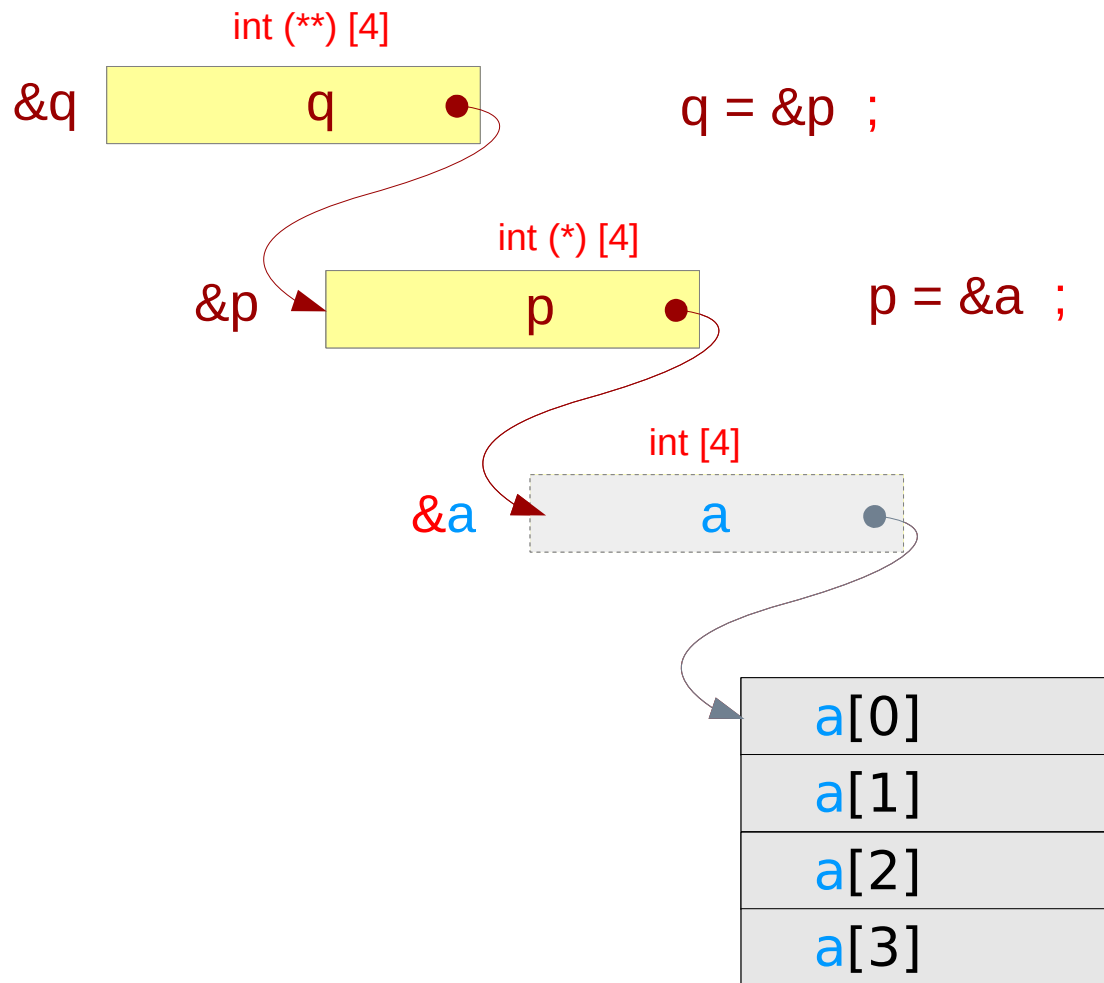
prototype
void **func**(int (*r)[2][3], ...);

```
int d[4][2][3][4];  
int (*s)[2][3][4];
```

call
fund(d, ...);

prototype
void **fund**(int (*s)[2][3][4], ...);

Double pointer to a 1-d array – a variable view (p, q)

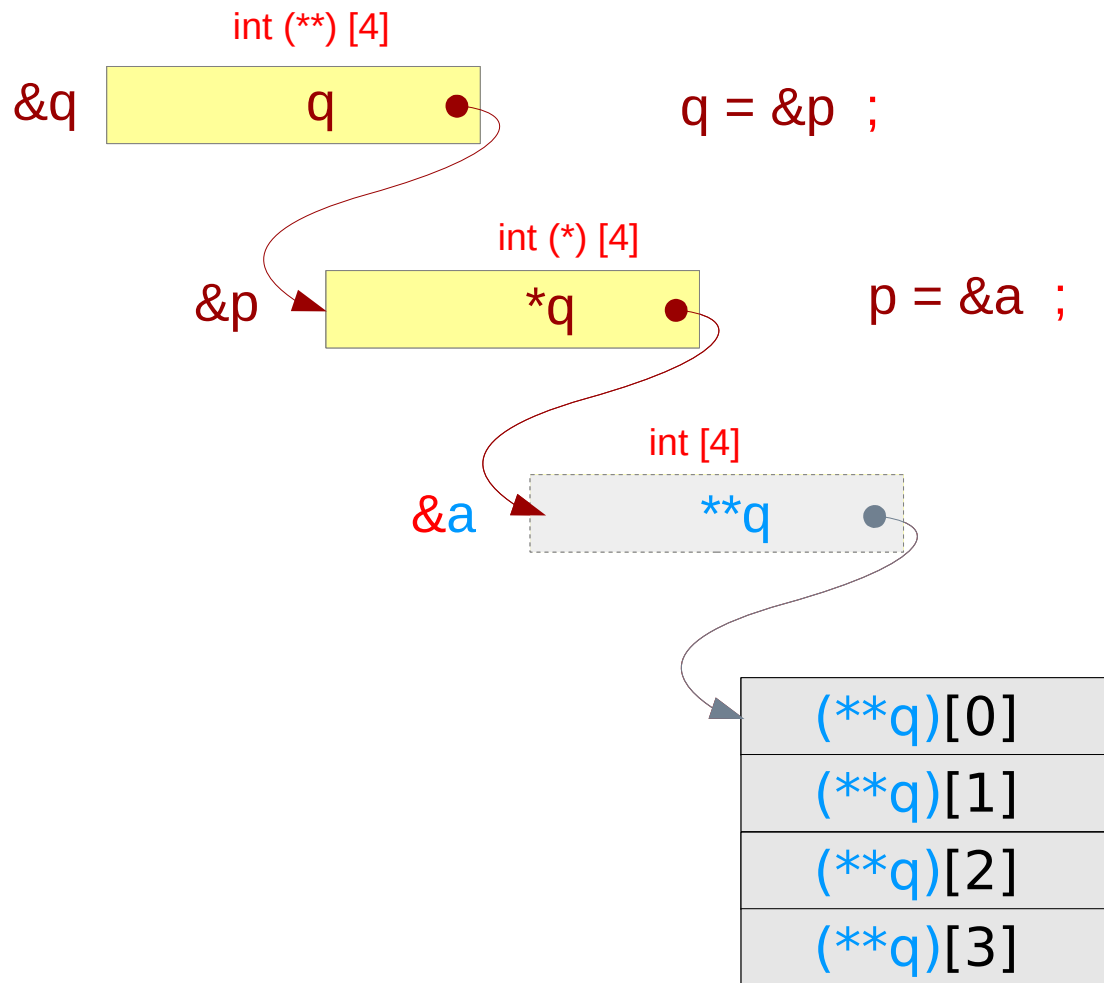


```
int a[4] ;  
int (*p) [4] = &a ;  
int (**q) [4] = &p ;
```

```
➡ p = &a ;
```

```
➡ q = &p ;
```

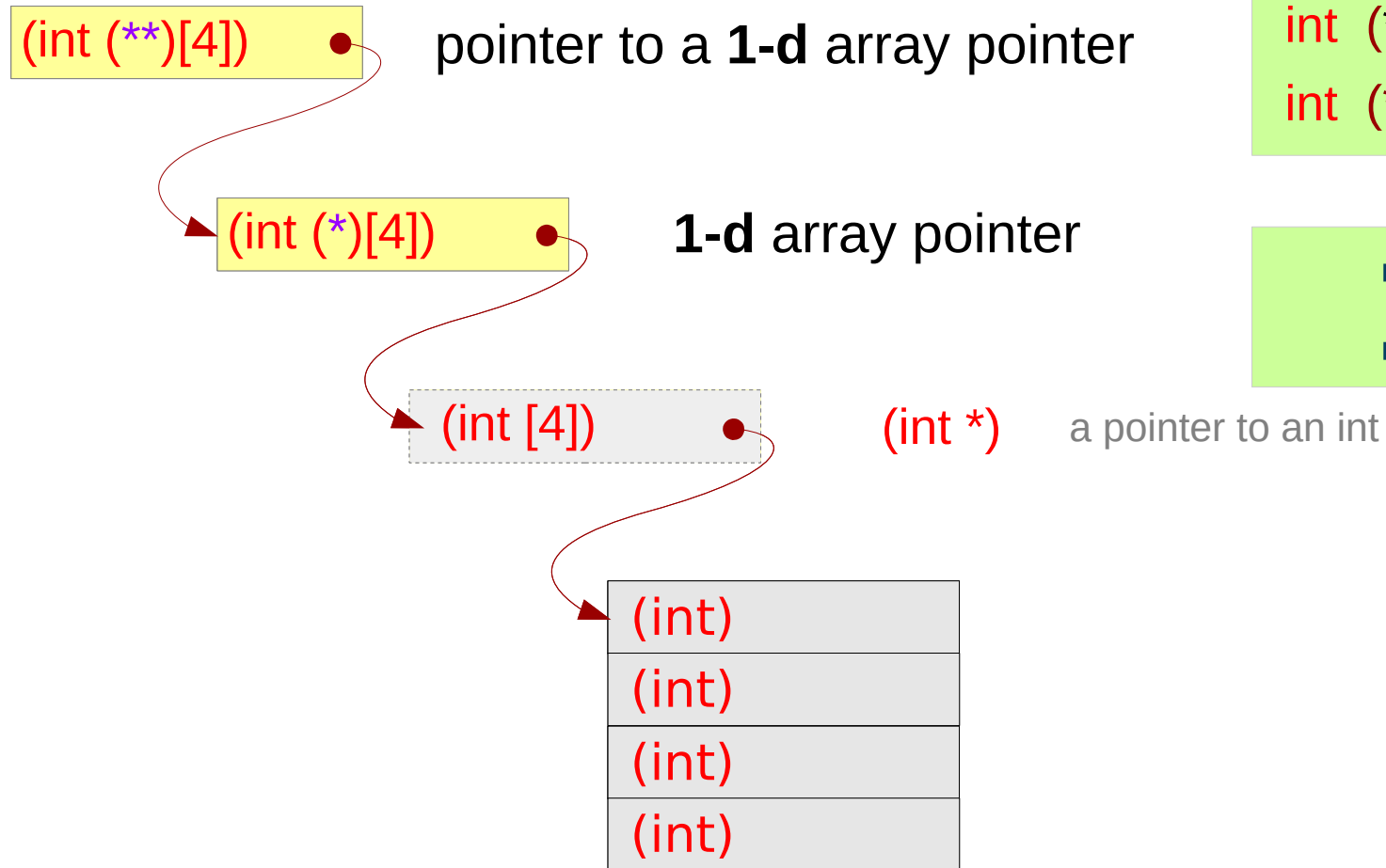
Double pointer to a 1-d array – a variable view (q)



```
int a[4] ;  
int (*p) [4] = &a ;  
int (**q) [4] = &p ;
```

```
➔ p = &a ;  
➔ q = &p ;
```

Double pointer to a 1-d array – a type view



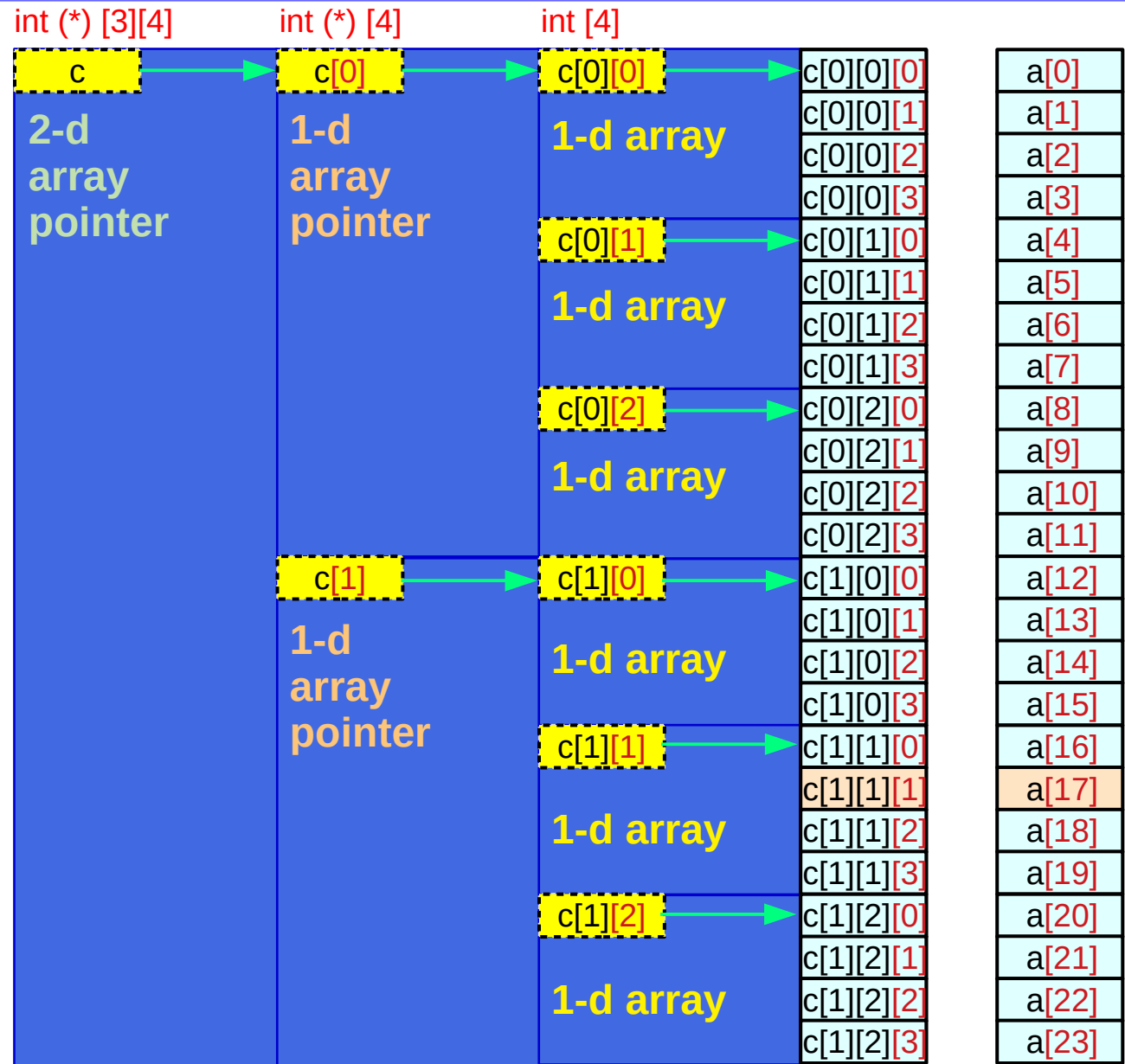
```
int a[4] ;  
int (*p) [4] = &a ;  
int (**q) [4] = &p ;
```

```
➔ p = &a ;  
➔ q = &p ;
```

Virtual Array Pointers in Multi-dimensional Arrays

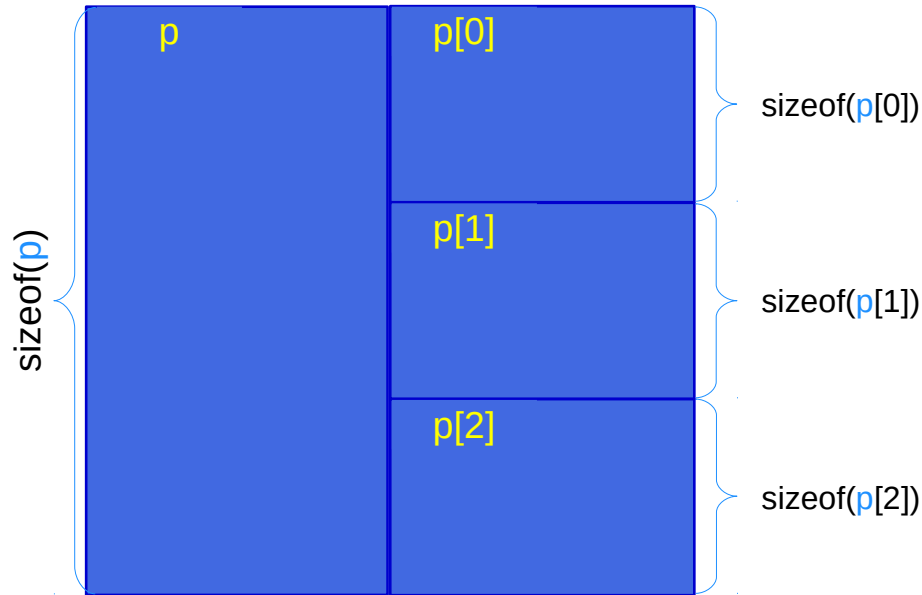
3-d array structure

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

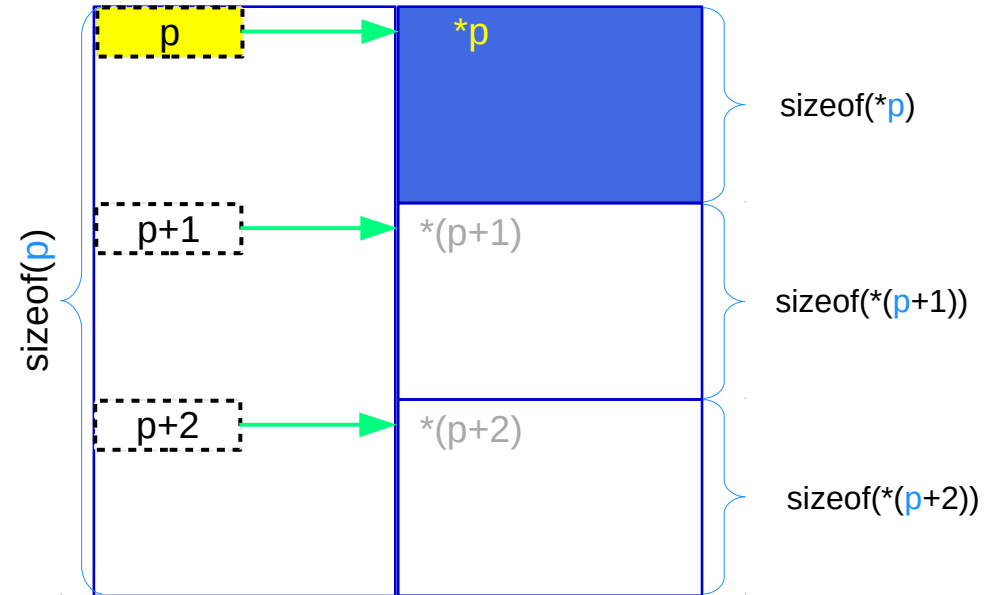


Array **p** and virtual array pointer **p**

Abstract data (array) **p**



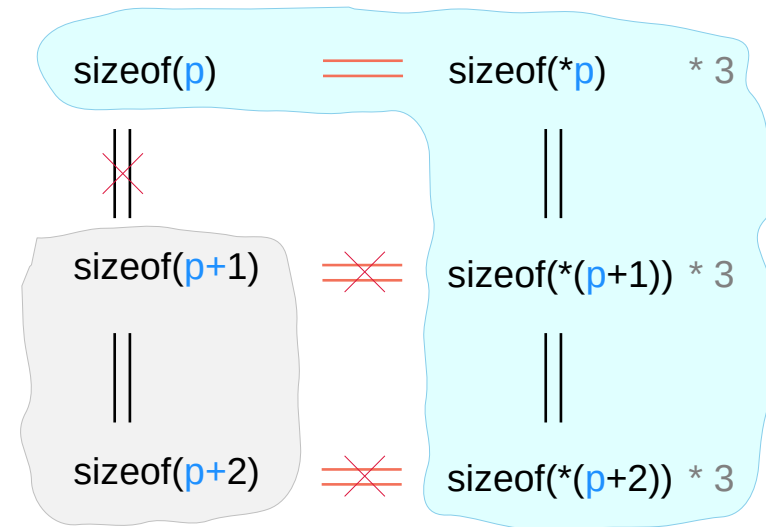
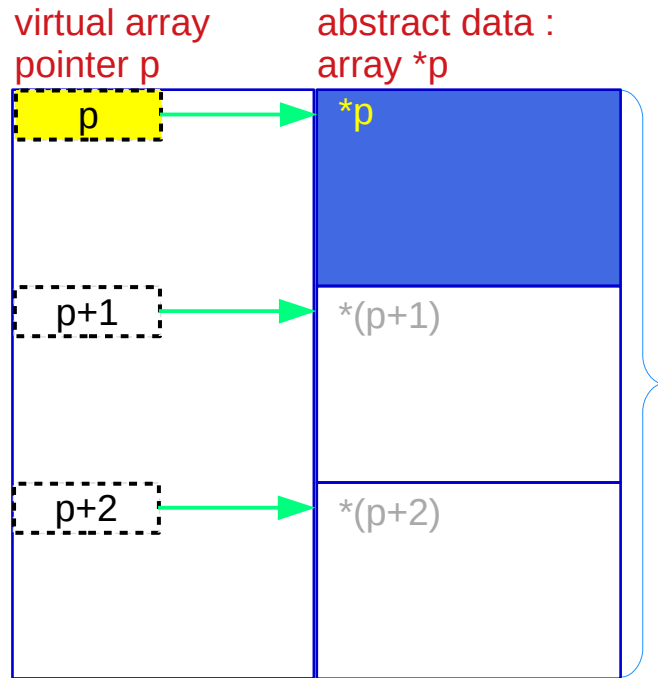
Virtual array pointer **p**



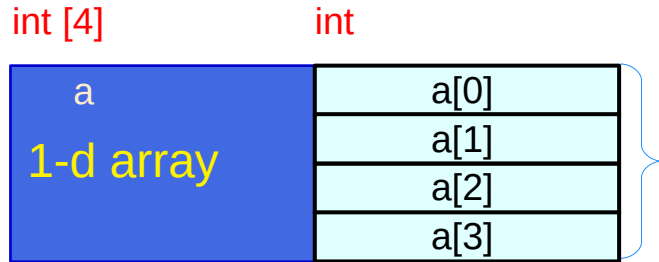
p is the name of an array and has a array pointer type but has a size of the array

p is a virtual array pointer

Virtual array pointer to abstract data

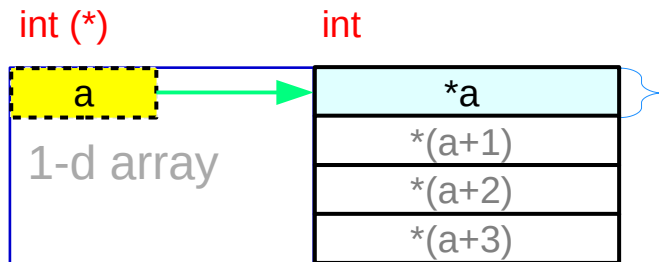


Array **a** and pointer **a**



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

$\text{sizeof}(a)$



pointer **a** general pointer type

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

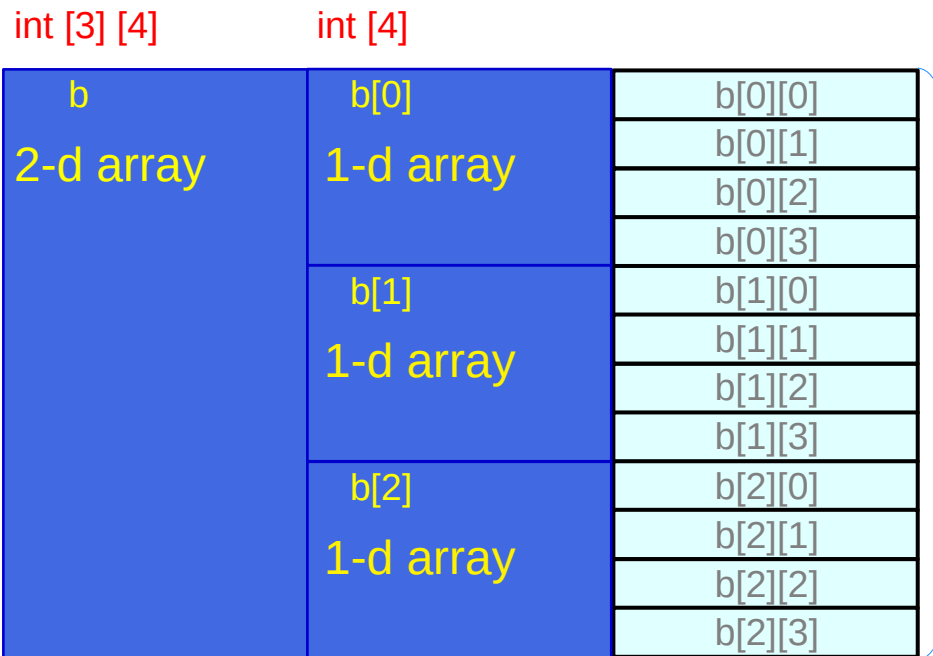
a is the name of a 1-d array and has a pointer type but has a size of the array

a is a virtual array pointer

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

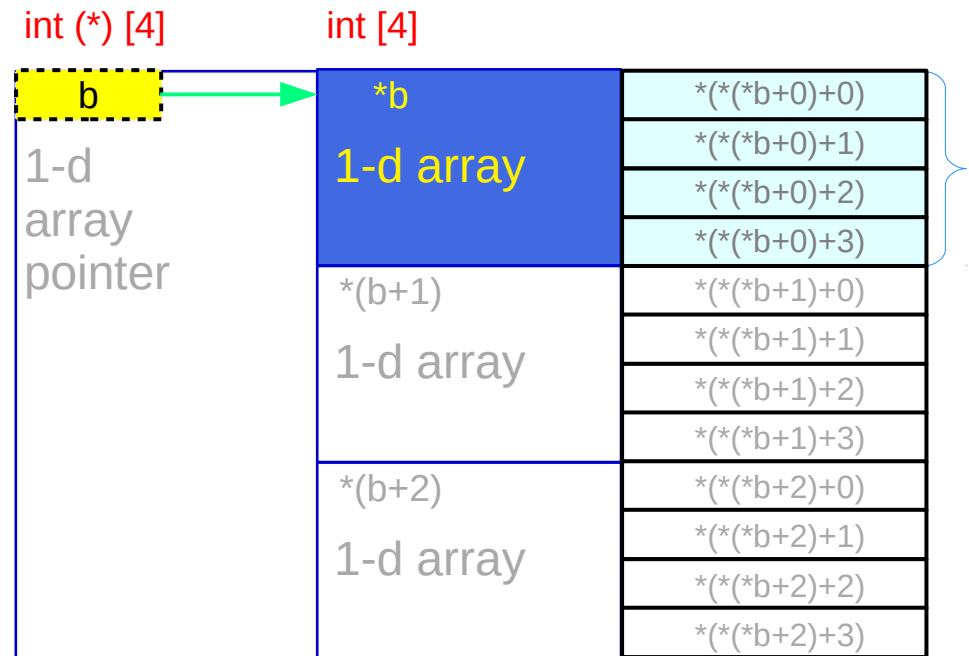
2-d array **b** specific array type

`sizeof(b)`



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

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



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

Pointer c

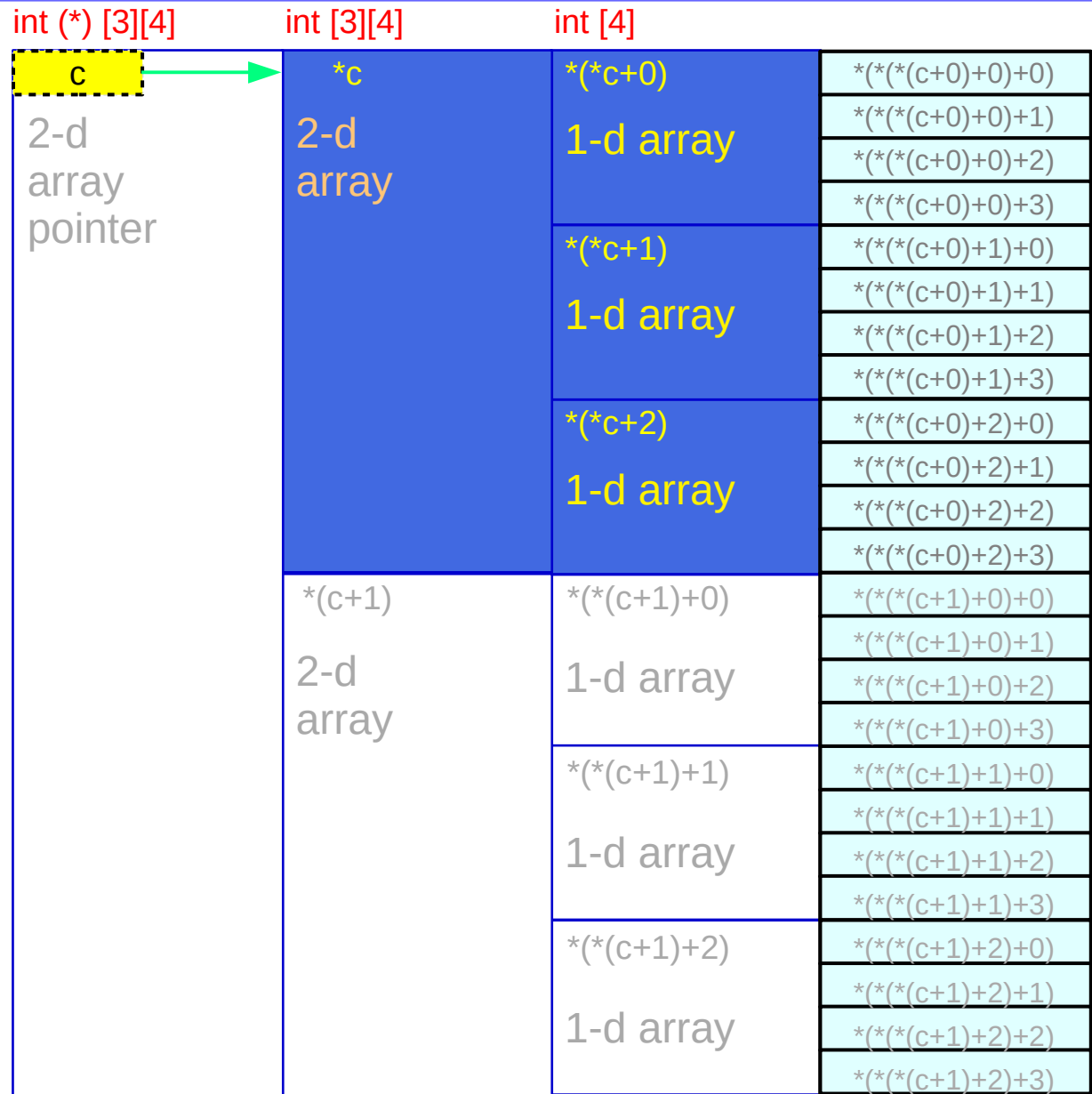
2-d array pointer c

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

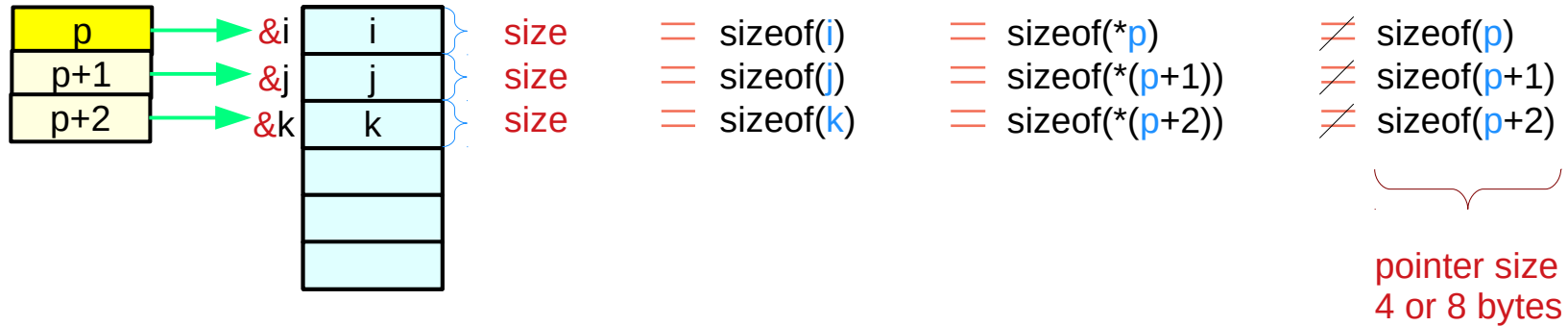
c is a virtual array pointer



Pointers to primitive data

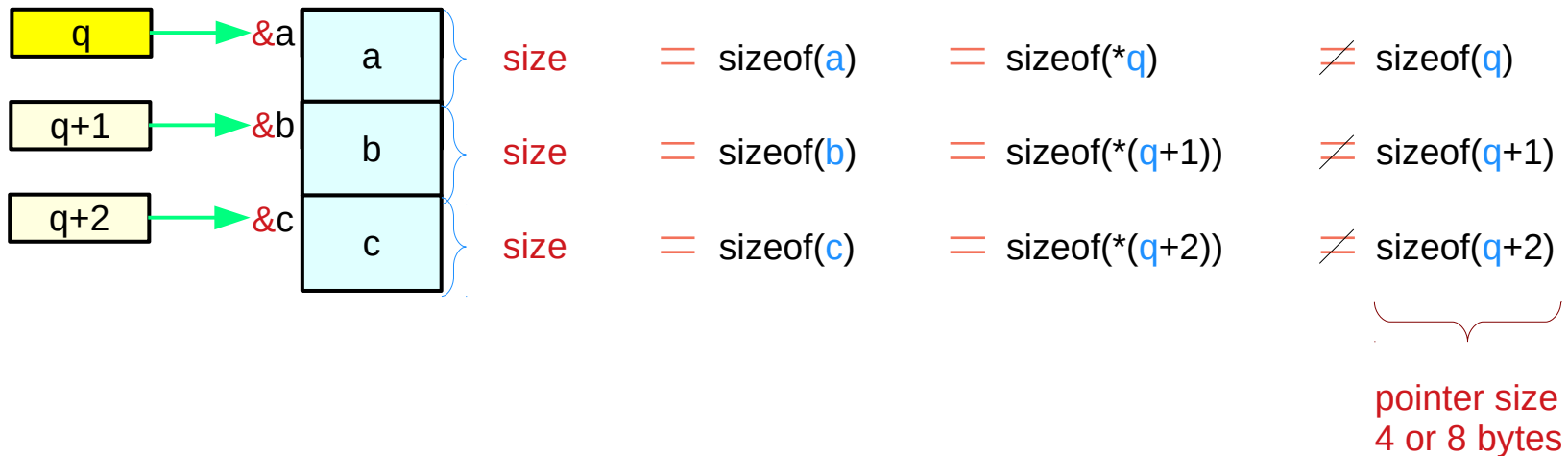
int *p;

int i, j, k;



double *q;

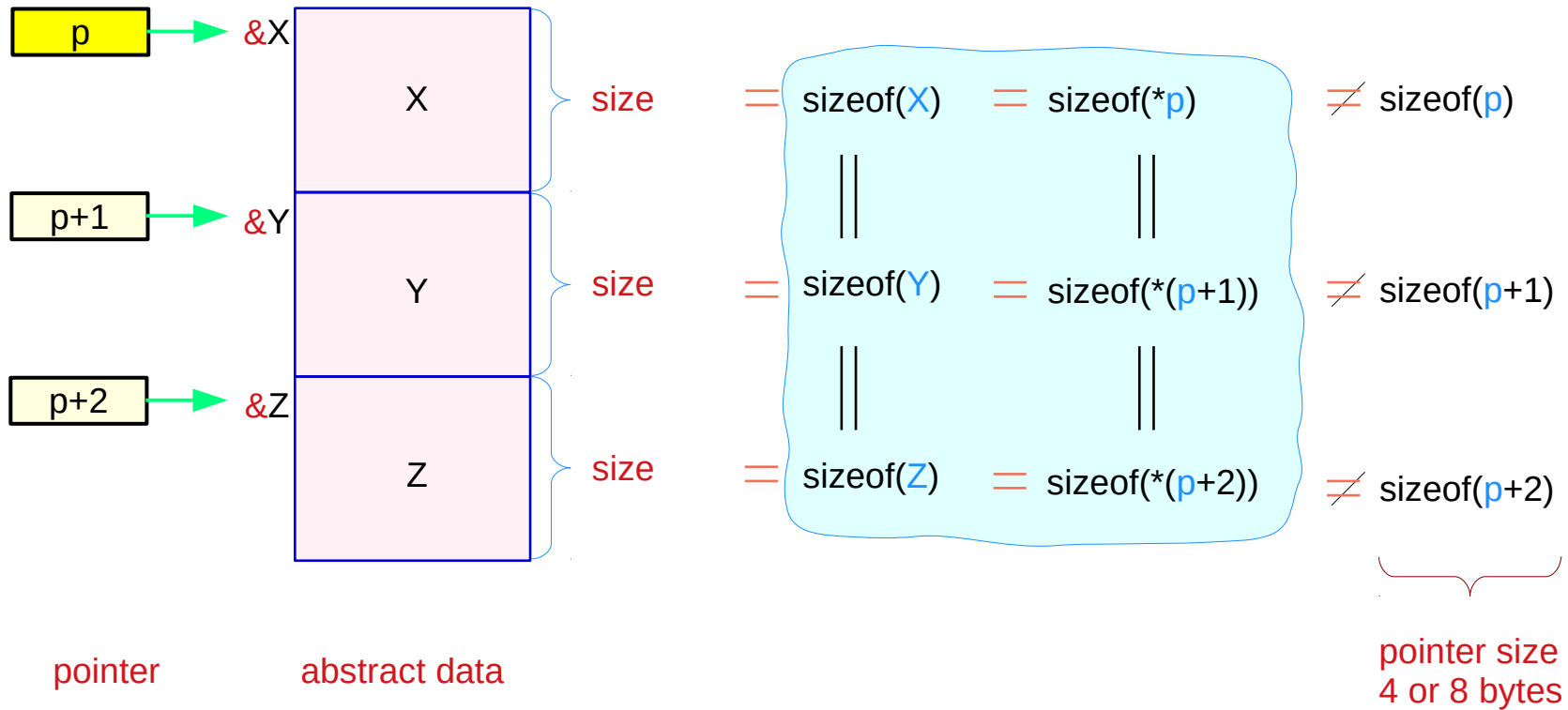
double a, b, c;



Pointers to abstract data

T *p;

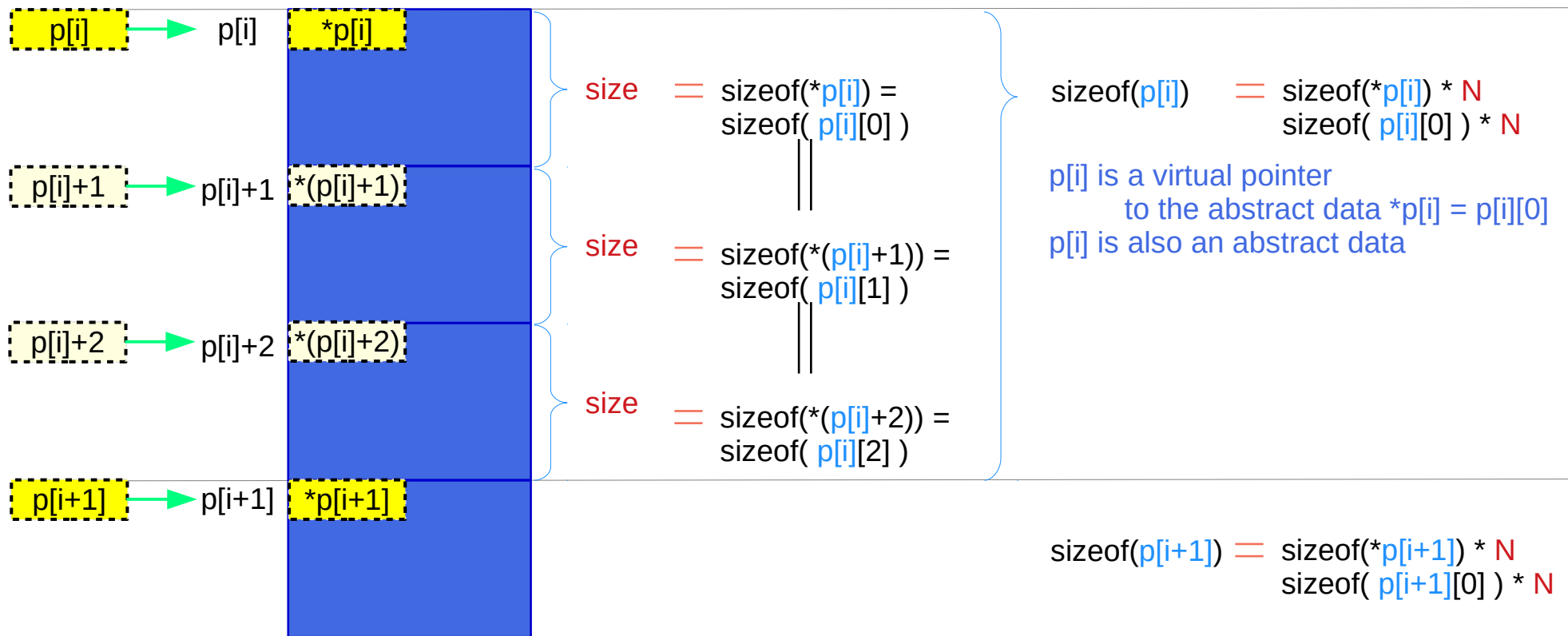
T X, Y, Z;



type ----- array
 value ----- start address
 increment size ----- size

Virtual pointers in a multi-dimensional array

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

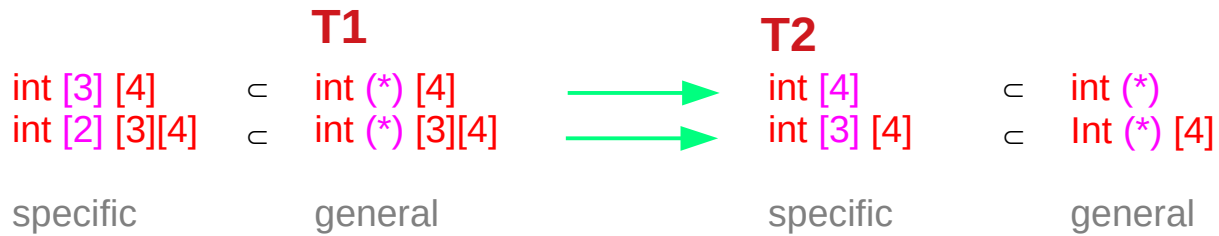


T1
 $\text{int} (*) [4]$
 $\text{int} (*) [3][4]$

T2
 $\text{int} [4]$
 $\text{int} [3][4]$

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

Virtual pointers in a multi-dimensional array



```
typedef int (*T1) [4];  
typedef int (*T1) [3][4];  
  
typedef int T2[4];  
typedef int T2[3][4];
```

T1 a;

T2 b;

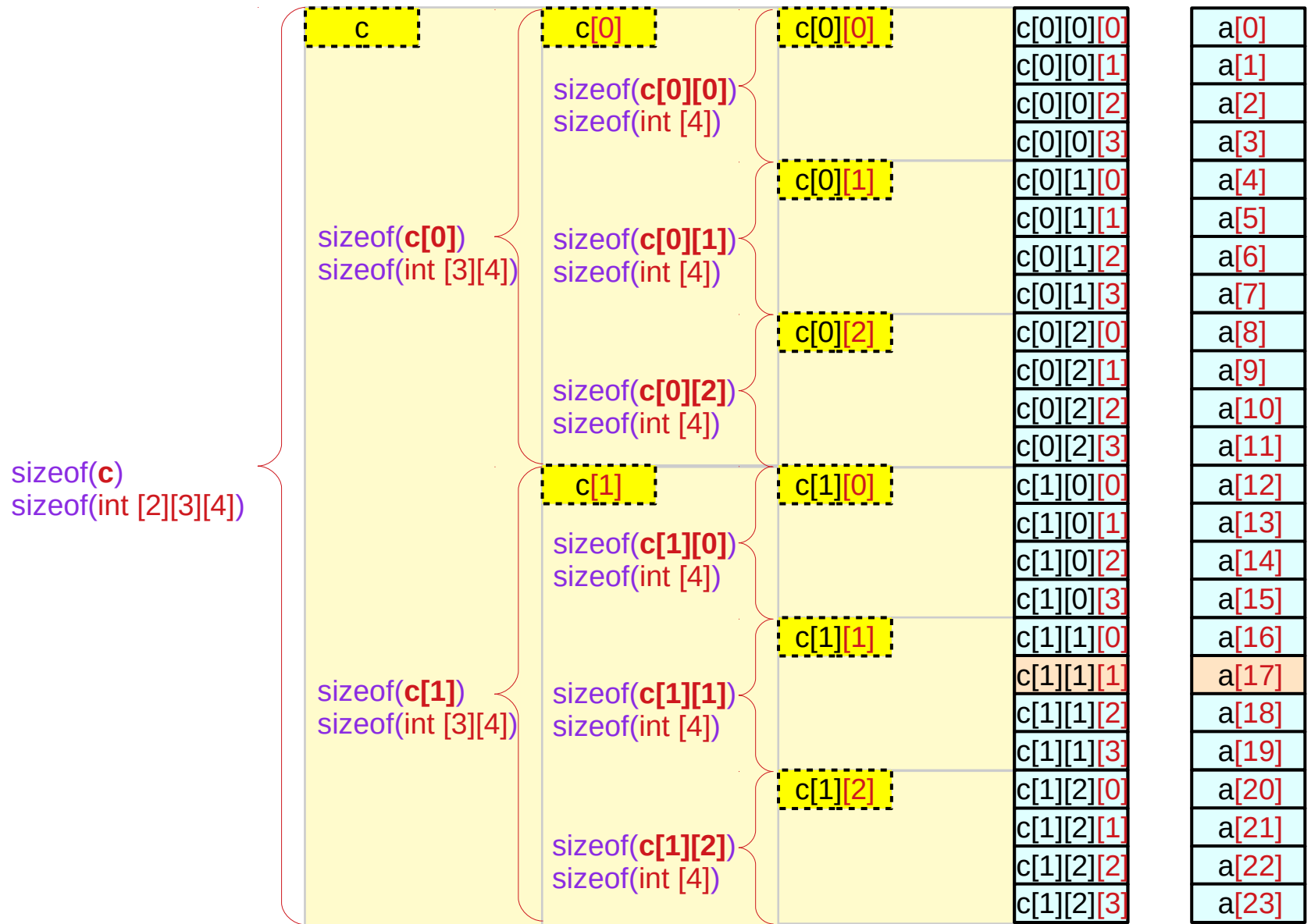
T1 references T2
T2 is a dereference of T1

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

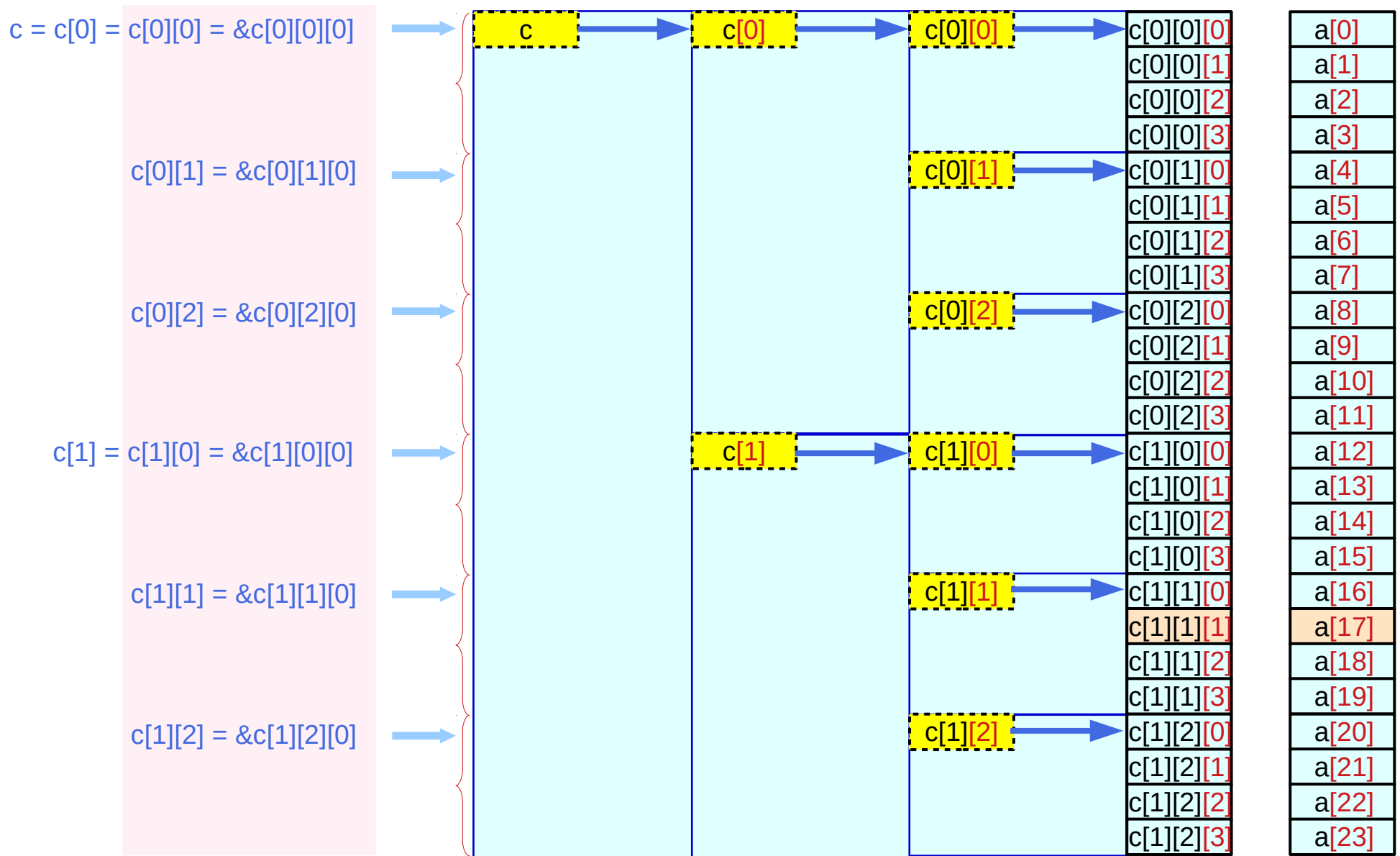
Virtual array pointers – types, sizes, and values

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]	

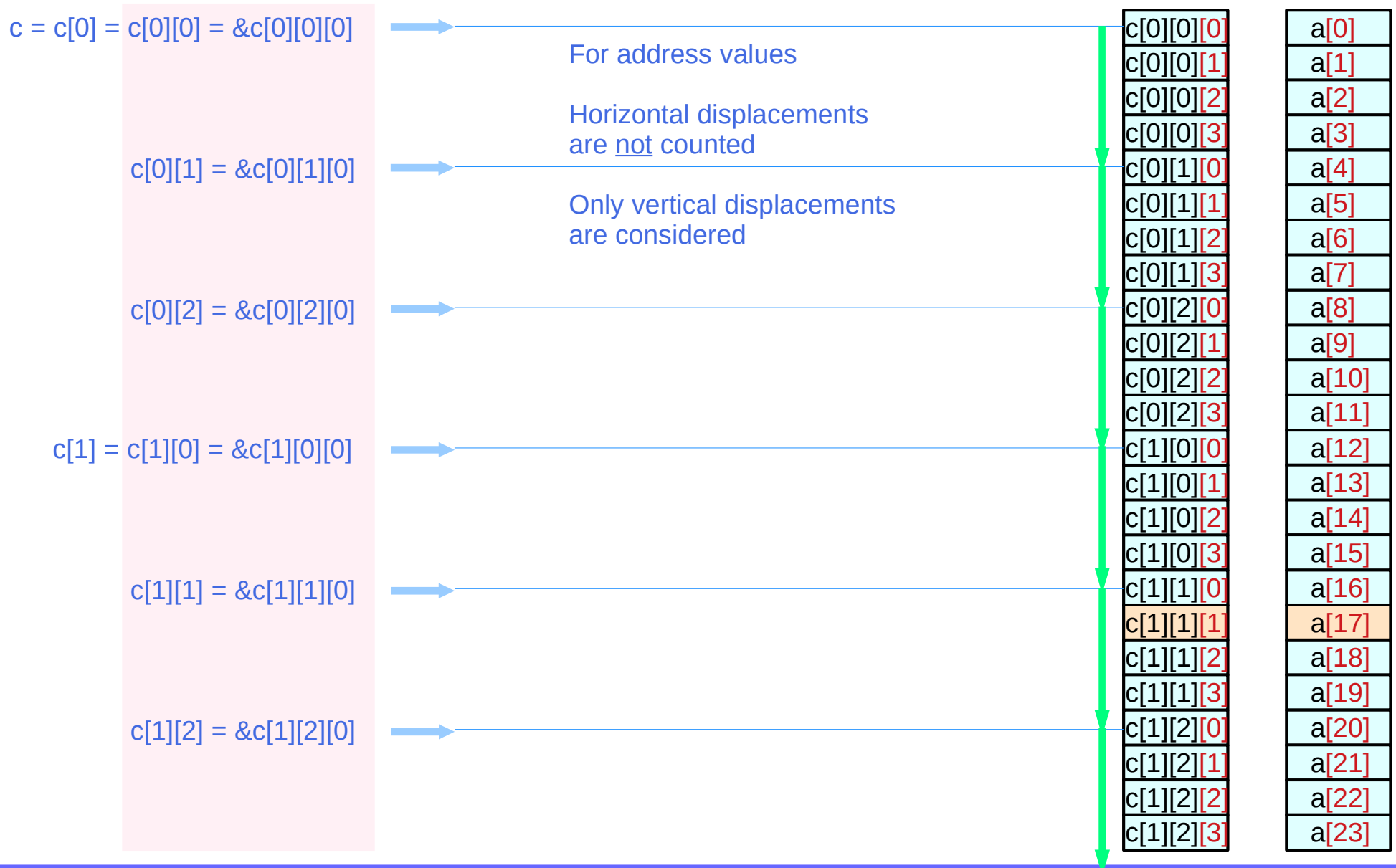
virtual array pointers c, c[i], c[i][j] – sizes



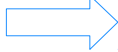
Virtual array pointer c, c[i], c[i][j] – values (addresses)




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





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

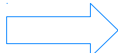
c = c[0] = c[0][0] = &c[0][0][0]  means

c[0][1] = &c[0][1][0]  means

c[0][2] = &c[0][2][0]  means

c[1] = c[1][0] = &c[1][0][0]  means

c[1][1] = &c[1][1][0]  means

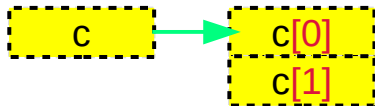
c[1][2] = &c[1][2][0]  means

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

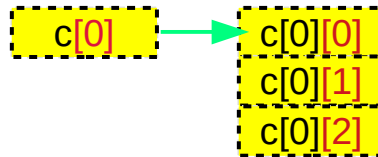
Virtual array pointer c, c[0], c[0][0] – types and sizes

Types – array pointers

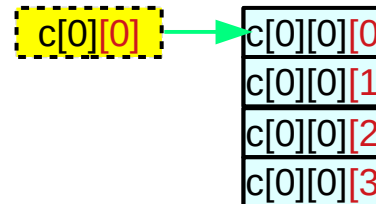
`int (*) [3][4]`



`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`

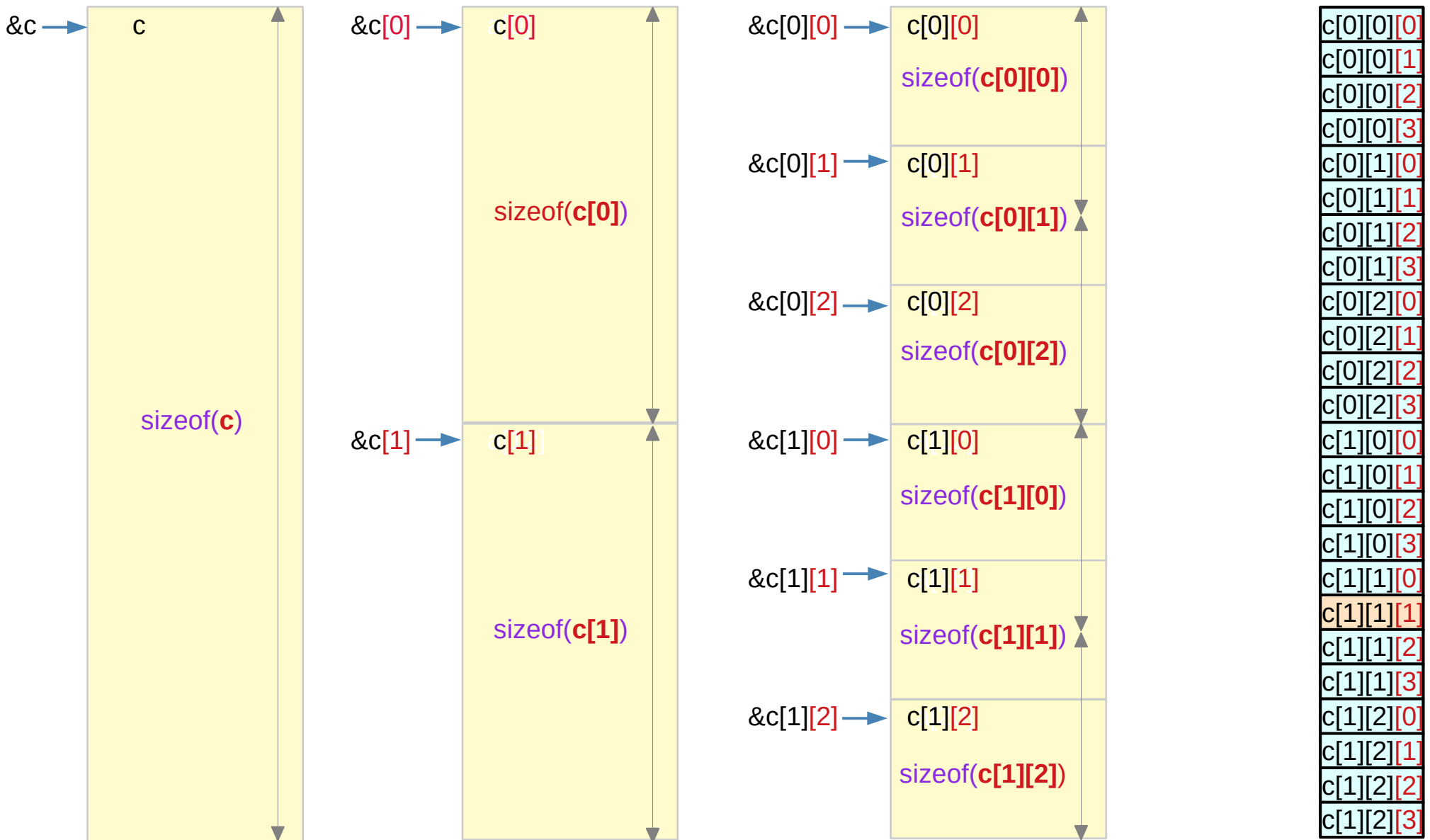
`sizeof(int (*) = 4 / 8`

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

`sizeof(int)`

`sizeof(int) = 4`

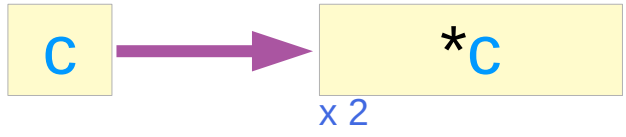
Abstract Data c , $c[i]$, $c[i][j]$ – start addresses and sizes



Types in a multi-dimensional array

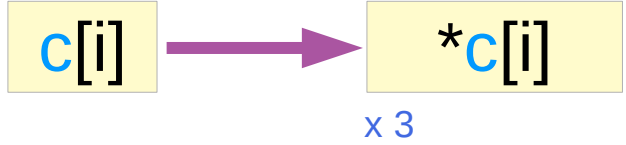
```
int c [2][3][4];
```

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



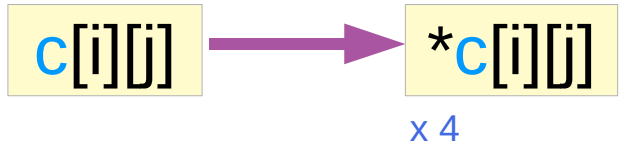
`int [3] [4]` abstract data
`int (*) [4]` array pointer

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



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

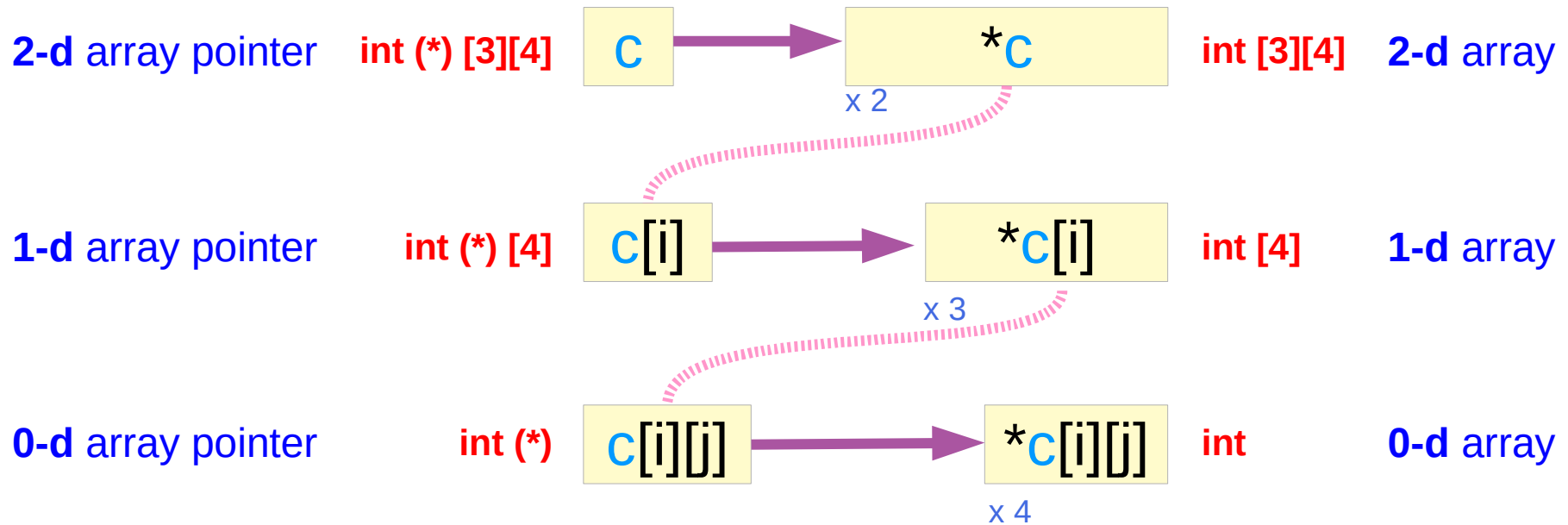
abstract data `int [4]`
array pointer `int (*)`



`int` primitive data

Virtual array pointers and abstract data

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



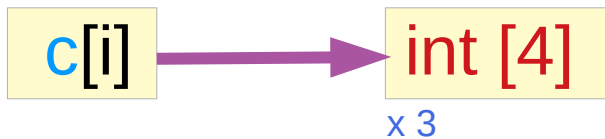
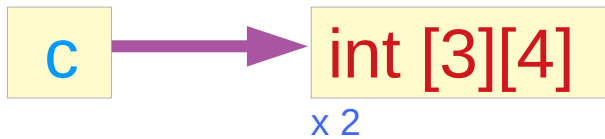
all these pointers are virtual, and take no actual memory locations

exploiting the **contiguity** of allocated memory locations

Abstract Data Sizes

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

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)
```

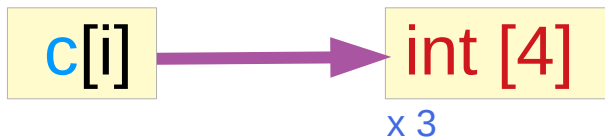
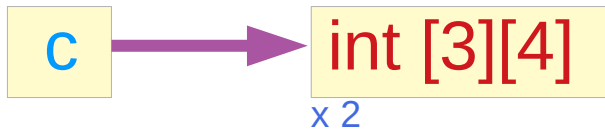
all are sizes of arrays

`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];
```

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: $\text{sizeof(int)} * 2 * 3 * 4$

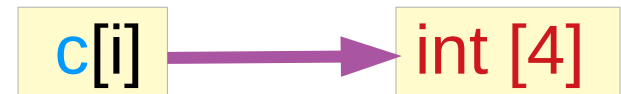
c[i] points to an **1-d** array
increment size: $\text{sizeof(int)} * 3 * 4$

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

int (*) [3][4]



int (*) [4]



int (*)



Sub-array properties in multi-dimensional arrays

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

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

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];
```

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

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

Explicit
Arrays of pointers with
Multiple Indirection

N-dim Array approach

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

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

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

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

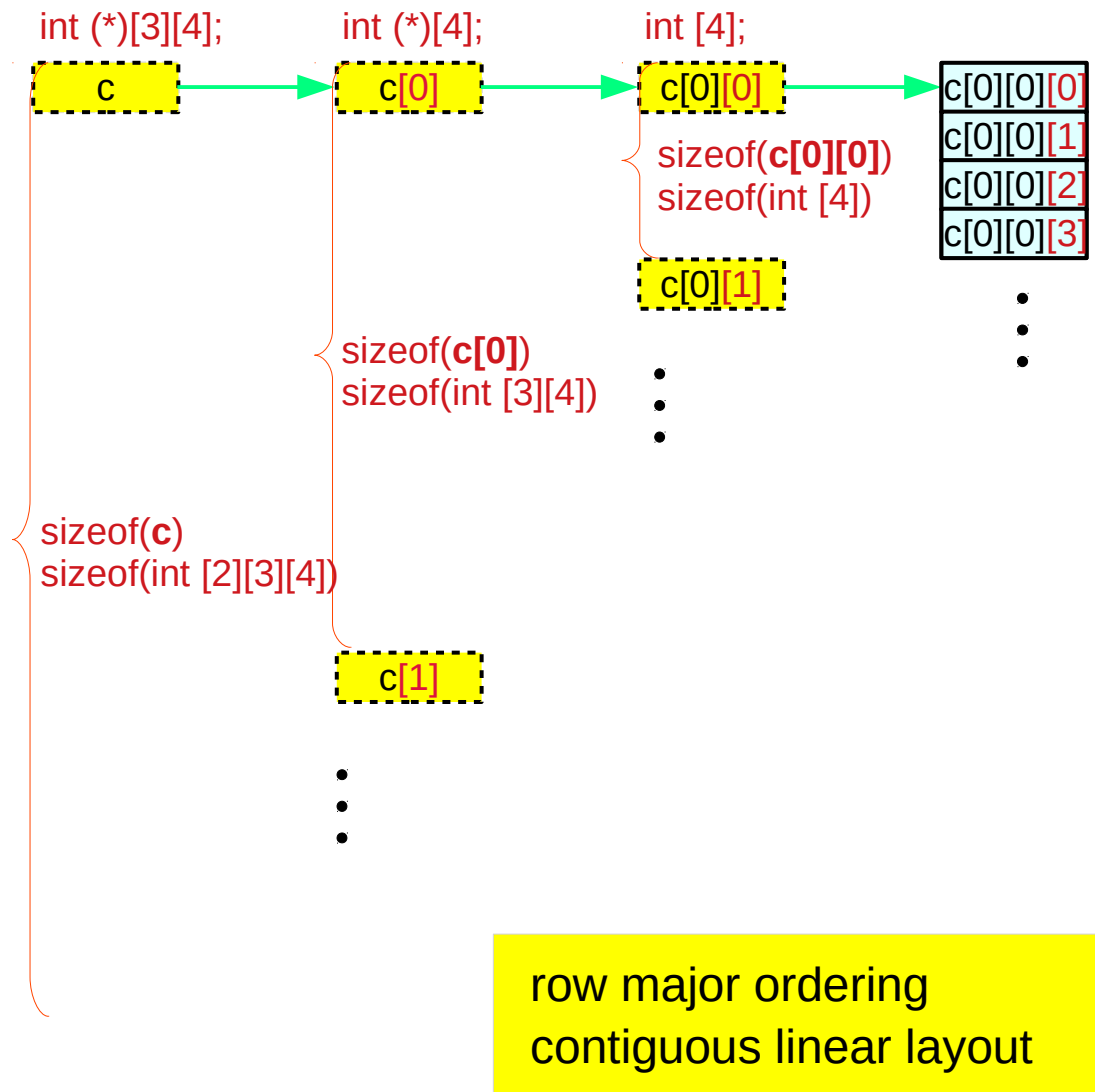
```
int c [2][3][4];
```



```
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];`

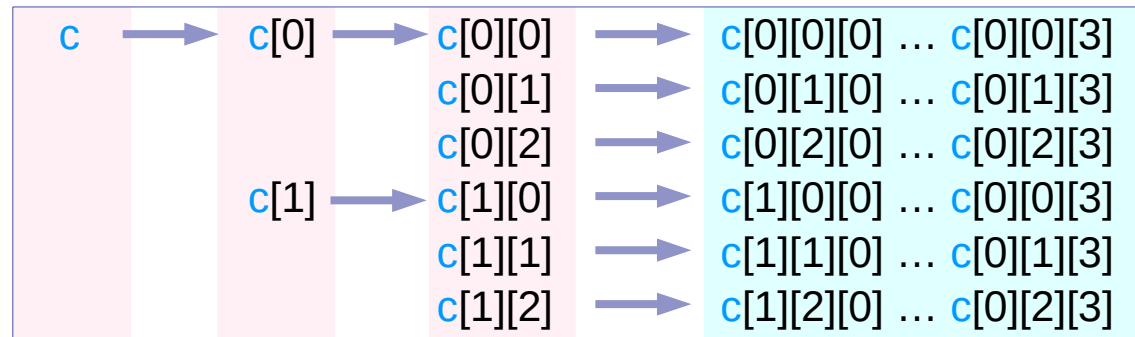
`&c[i][j][0] = c[i][j]`
`&c[i][0] = c[i]`
`&c[0] = c`

`&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 $c[i][j]$

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

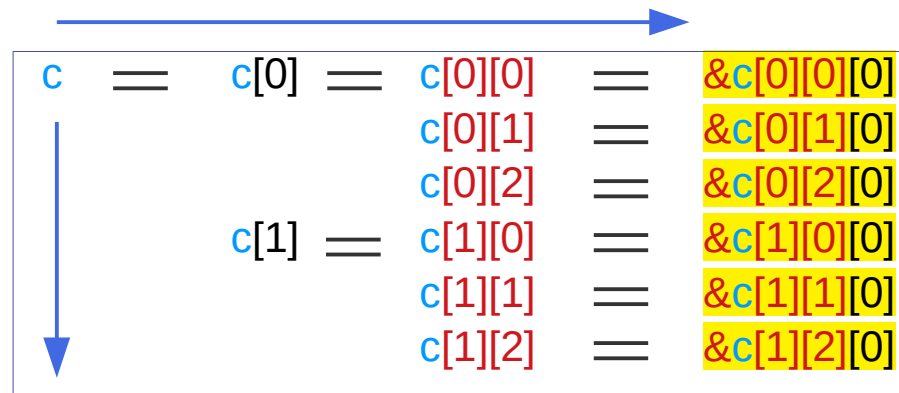
$\&c[i][j][0] = c[i][j]$
 $\&c[i][0] = c[i]$
 $\&c[0] = c$

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

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

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

<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[0][1][0]</code>	<u><u>-[0]</u></u>	<code>c[0][1]</code>				
<code>&c[0][2][0]</code>	<u><u>-[0]</u></u>	<code>c[0][2]</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>		
<code>&c[1][1][0]</code>	<u><u>-[0]</u></u>	<code>c[1][1]</code>				
<code>&c[1][2][0]</code>	<u><u>-[0]</u></u>	<code>c[1][2]</code>				

int

int [4]

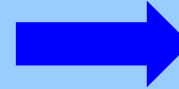
int (*) [4]

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
```

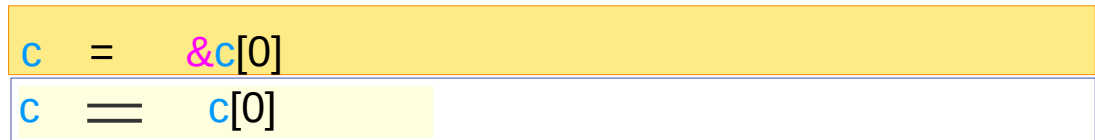
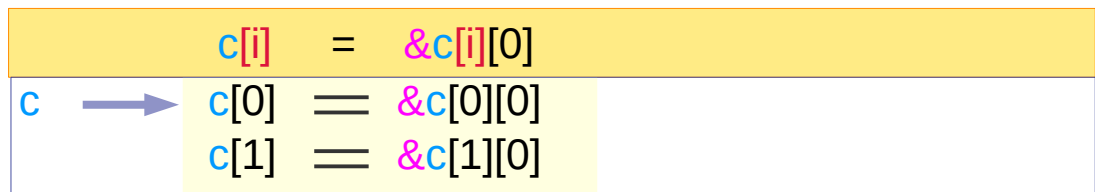
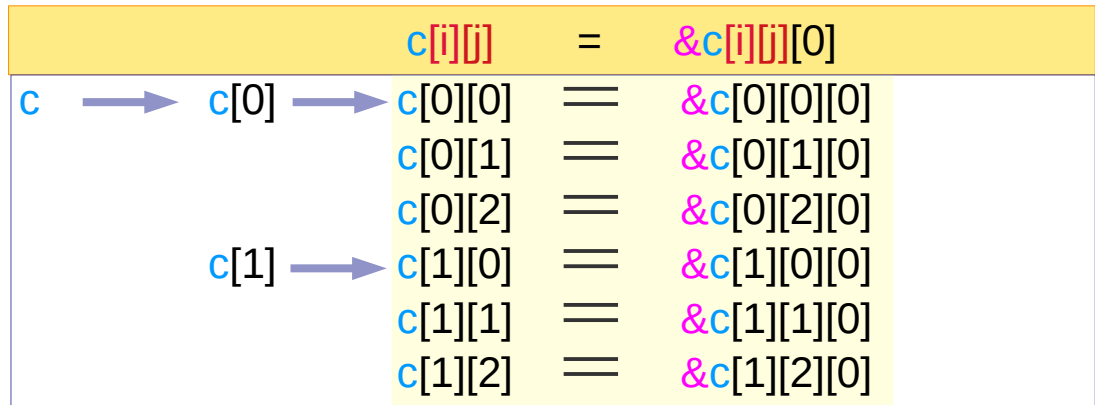
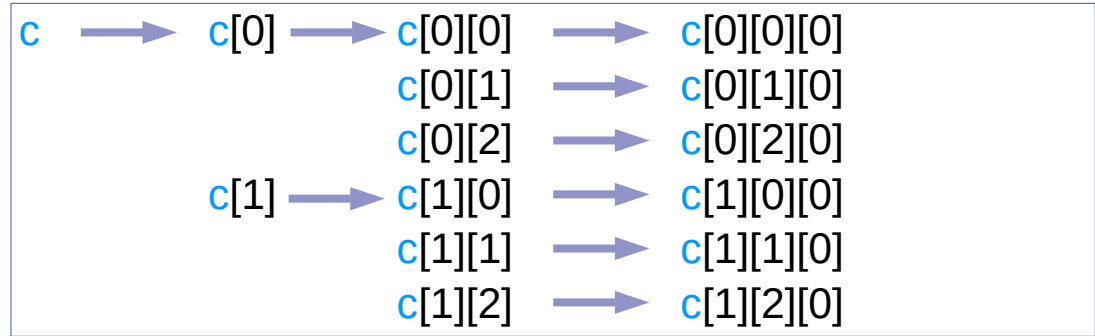
Pointer reference and dereference relationship

`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]`

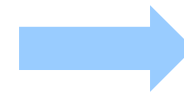
`c[0][0]` is the address of `c[0][0][0]`
`c[0][1]` is the address of `c[0][1][0]`
`c[0][2]` is the address of `c[0][2][0]`
`c[1][0]` is the address of `c[1][0][0]`
`c[1][1]` is the address of `c[1][1][0]`
`c[1][2]` is the address of `c[1][2][0]`

`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]`

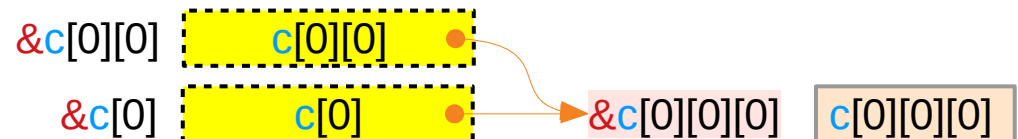
`value(c[0]) = &c[0][0][0]`

`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 $c[0][0]$

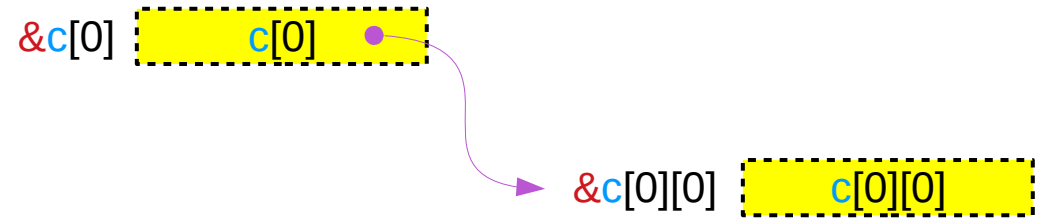
$c[i][j][k];$

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

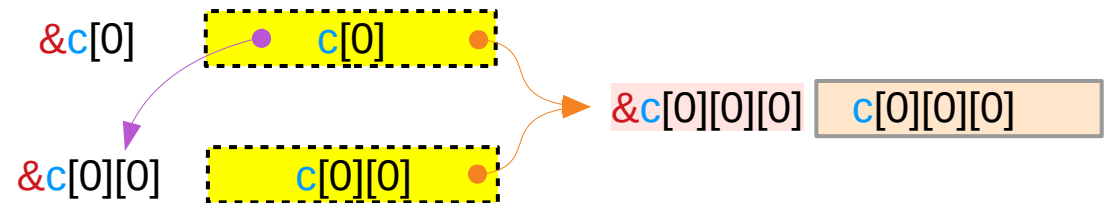
$\&c[i][j][k] = c[i][j] + k$
 $\&c[i][j] = c[i] + j$
 $\&c[i] = c + i$

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

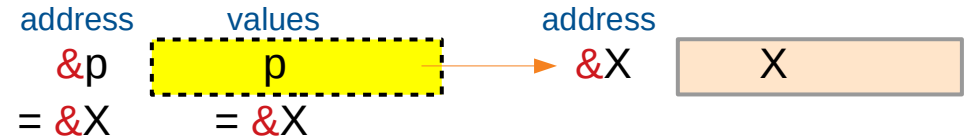
$c \rightarrow c[0] \rightarrow c[0][0] = \&c[0][0][0]$



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



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
```

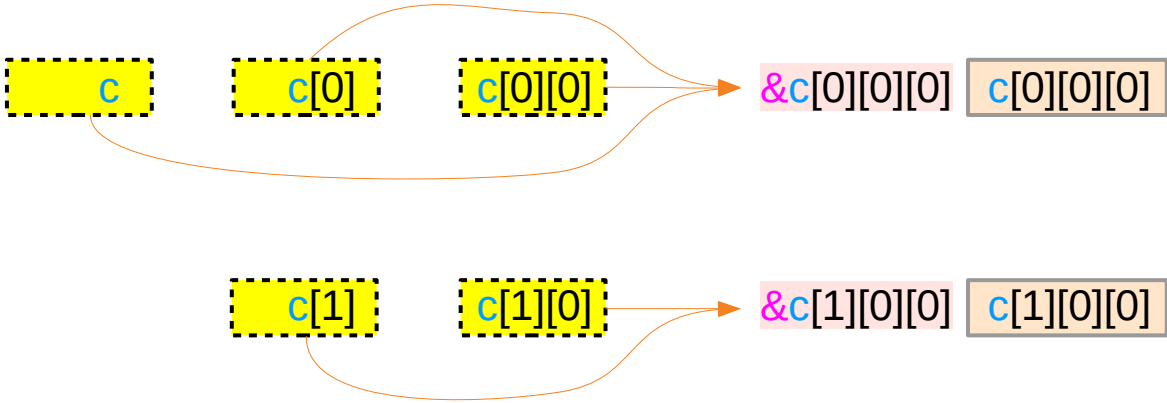
```
&c[i][j][k] = c[i][j]+k
&c[i][j]    = c[i]+j
&c[i]       = c+i
```

```
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

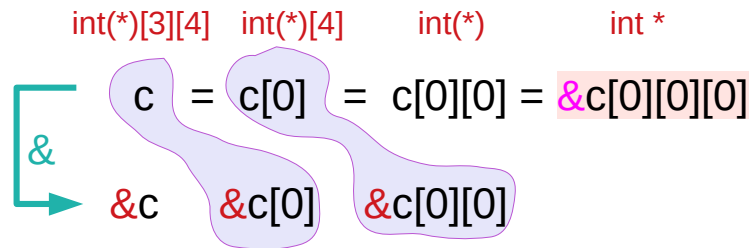
`c [i][j][k];`

`&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`

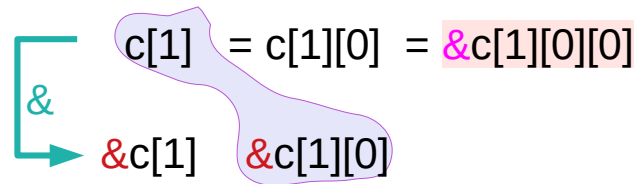
`int c [2][3][4];`

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



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]`

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

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

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

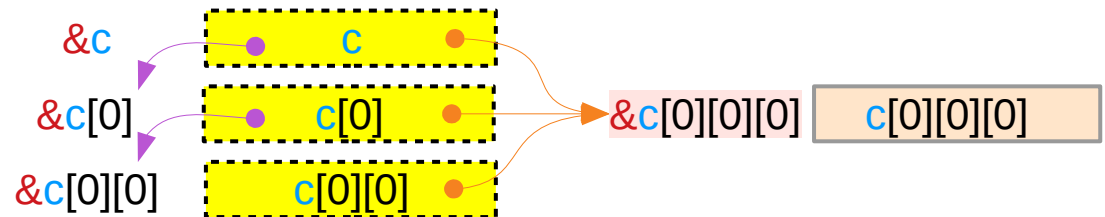
$\&c[i][j][k] = c[i][j] + k$
 $\&c[i][j] = c[i] + j$
 $\&c[i] = c + i$

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

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

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

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



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

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

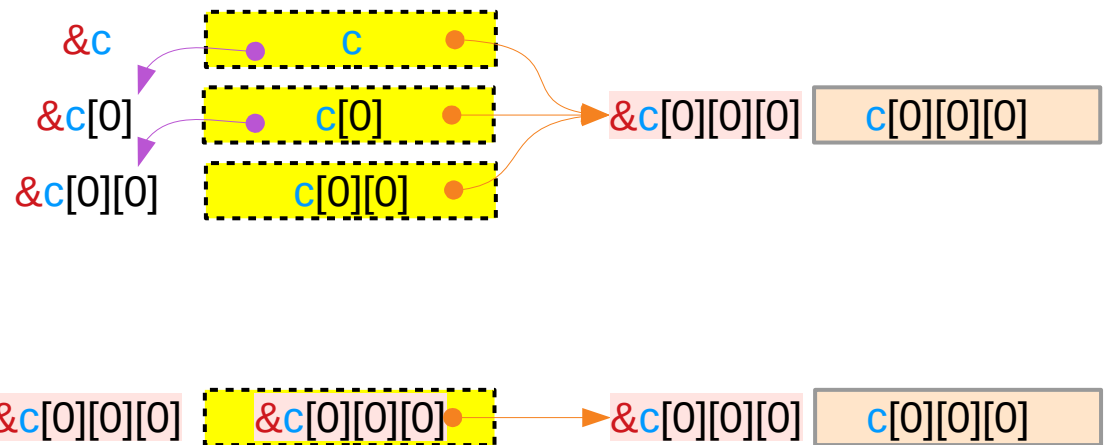
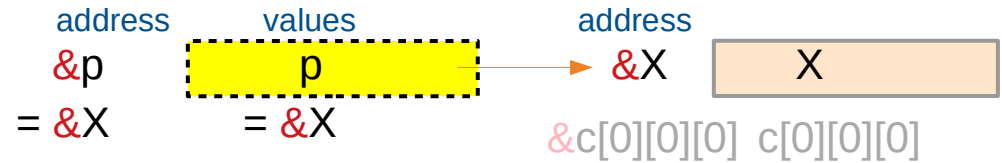
$\&c[i][j][0] = c[i][j]$
 $\&c[i][0] = c[i]$
 $\&c[0] = c$

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

`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

```
&c[0][0][0] -[0] c[0][0] -[0] c[0] -[0] c
&c[1][0][0] -[0] c[1][0] -[0] c[1]
```

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] ≡ c[i][j]`

`&c[i][j] ≡ c[i] + j`

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

delete [0] from the right

<code>&c[0][0][0]</code>	<u><u><u>-[0]</u></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[0][1][0]</code>	<u><u>-[0]</u></u>	<code>c[0][1]</code>				
<code>&c[0][2][0]</code>	<u><u>-[0]</u></u>	<code>c[0][2]</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>		
<code>&c[1][1][0]</code>	<u><u>-[0]</u></u>	<code>c[1][1]</code>				
<code>&c[1][2][0]</code>	<u><u>-[0]</u></u>	<code>c[1][2]</code>				

Contiguity Constraints

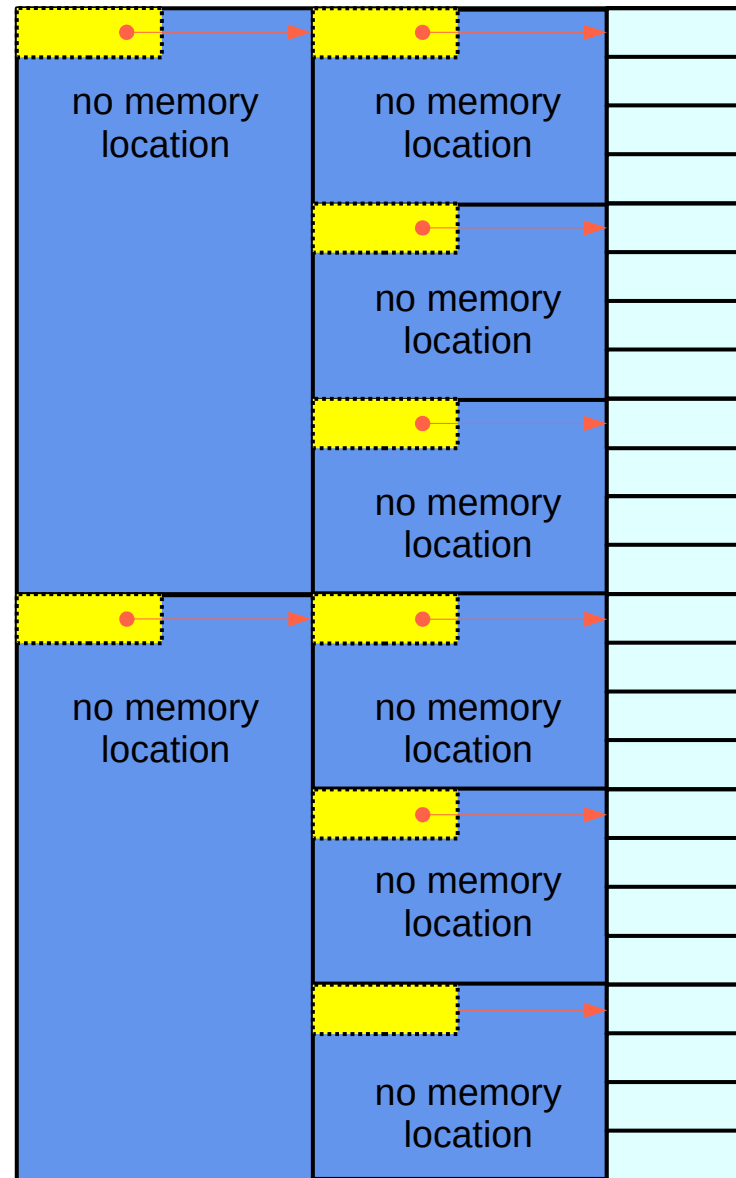
c [i][j][k];

Virtual Array Pointers and Contiguity

Using array pointers

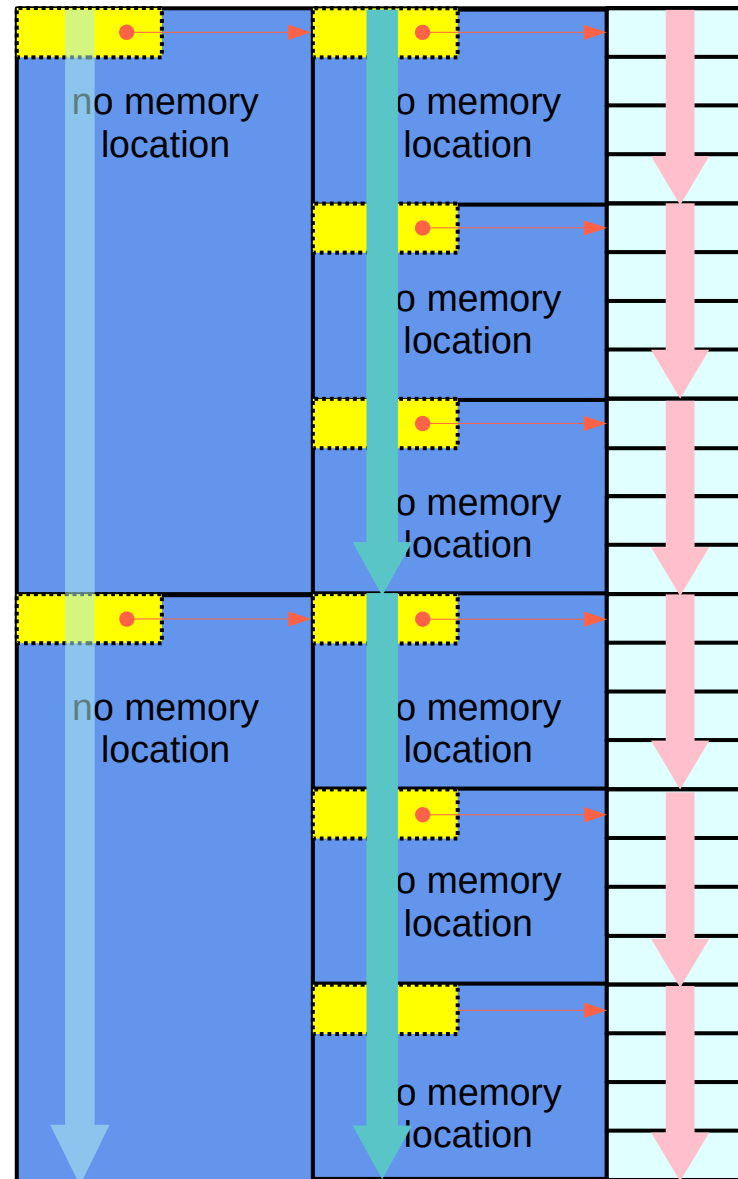
```
int (*) [N], int (*) [M][N], int (*) [L][M][N], ...
```

Array pointer approach for 3-d access patterns



Array Pointer Approach
(pointer to arrays)

Array pointer approach – contiguity constraints



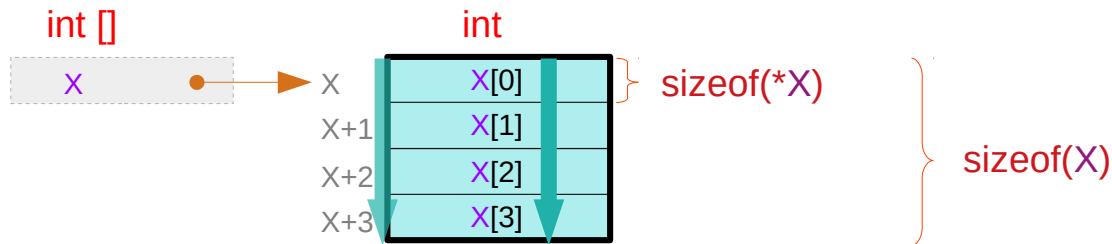
Array Pointer Approach
(pointer to arrays)

Equivalence and contiguity (1)

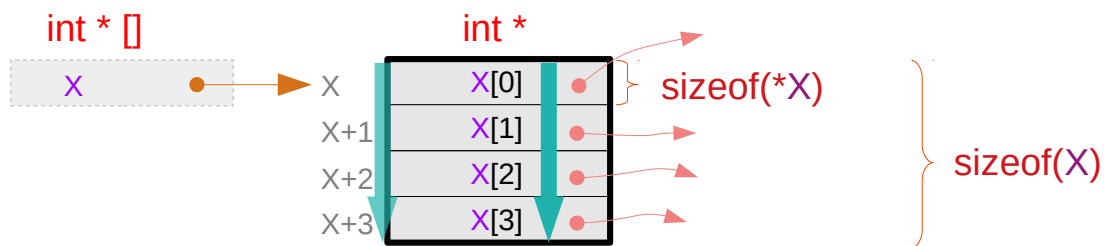
consecutive address consecutive data

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

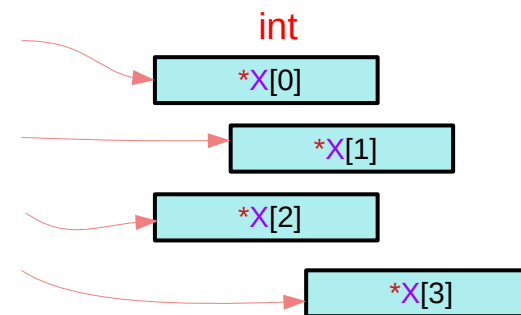
contiguous index : n



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



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



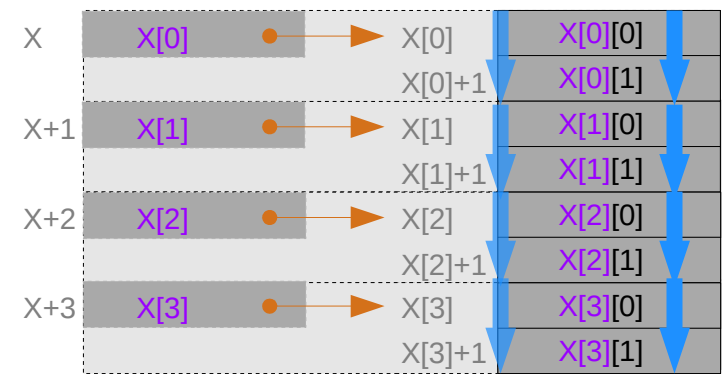
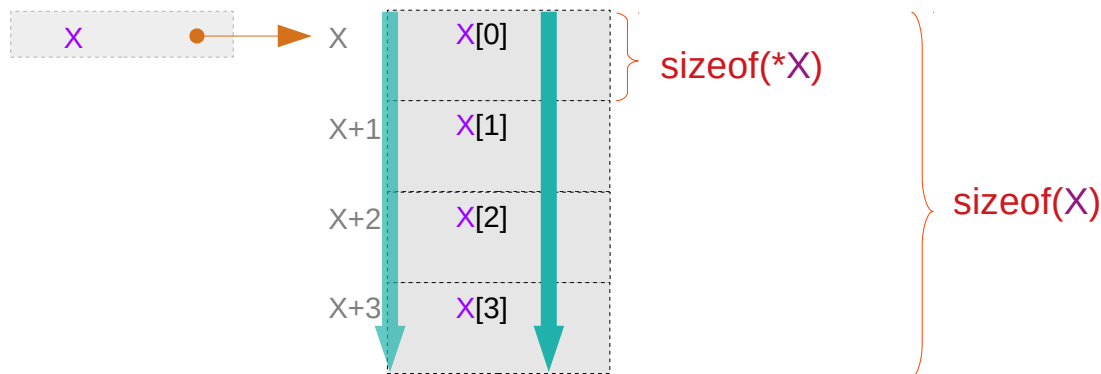
Equivalence and contiguity (2)

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

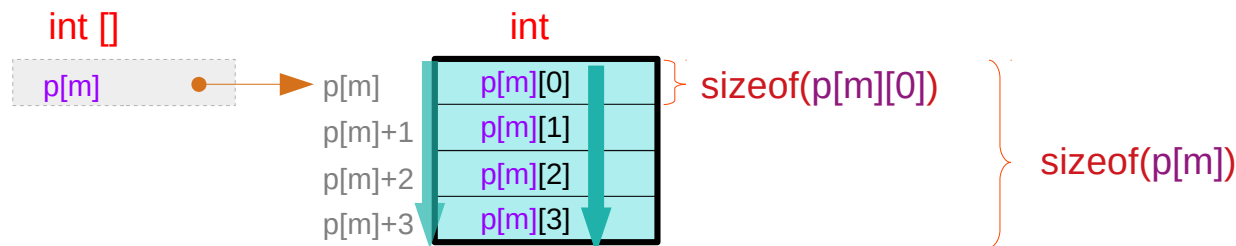
Equivalence for a given p[m] (1)

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

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

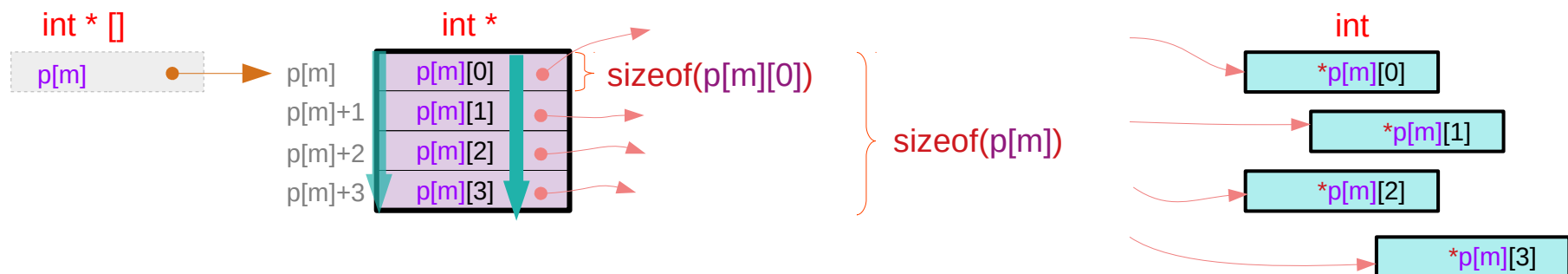
int p[M][4]; contiguous p[m][n] for a given p[m] : **primitive types**

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



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

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

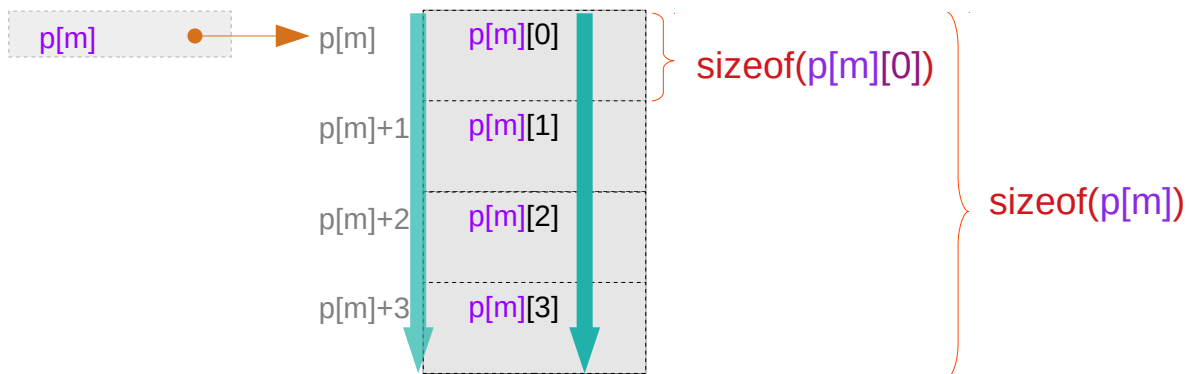


Equivalence for a given p[m] (2)

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

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

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



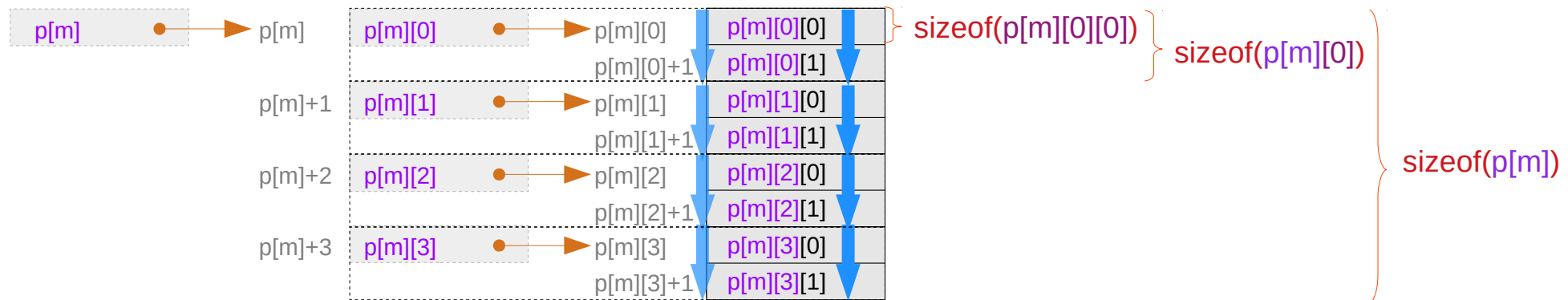
can be recursively applied

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

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

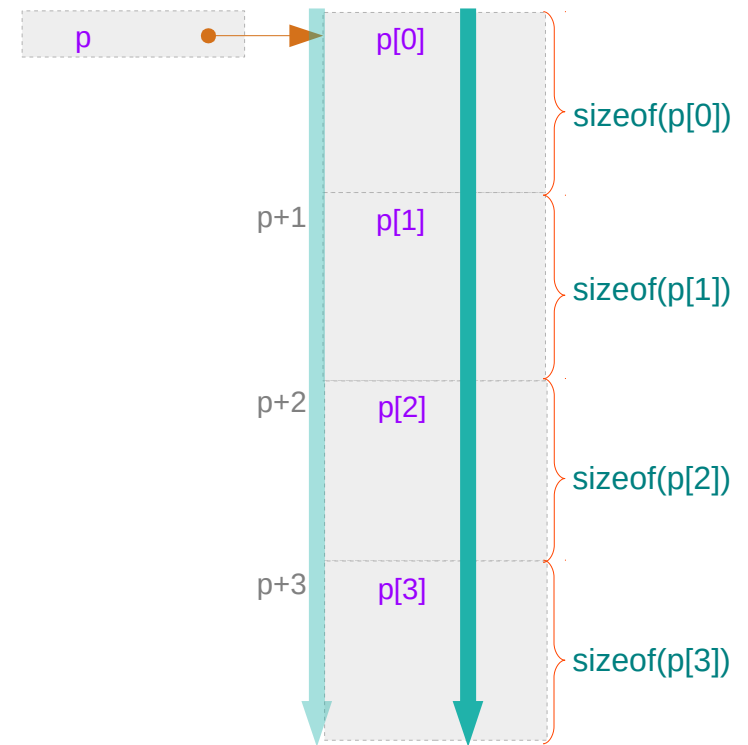
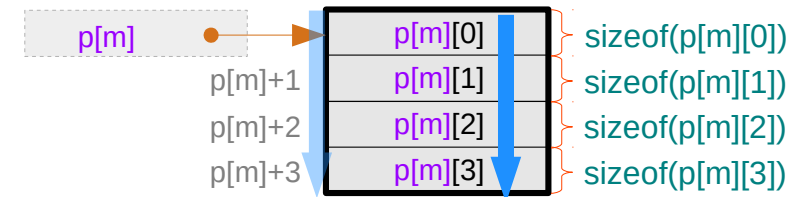
for a given $p[m]$, thus for a given p and m ,
 $p[m][n]$'s must be contiguous for all n .
 $p[m][0], p[m][1], \dots, p[m][N-1]$

contiguous index : n

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

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

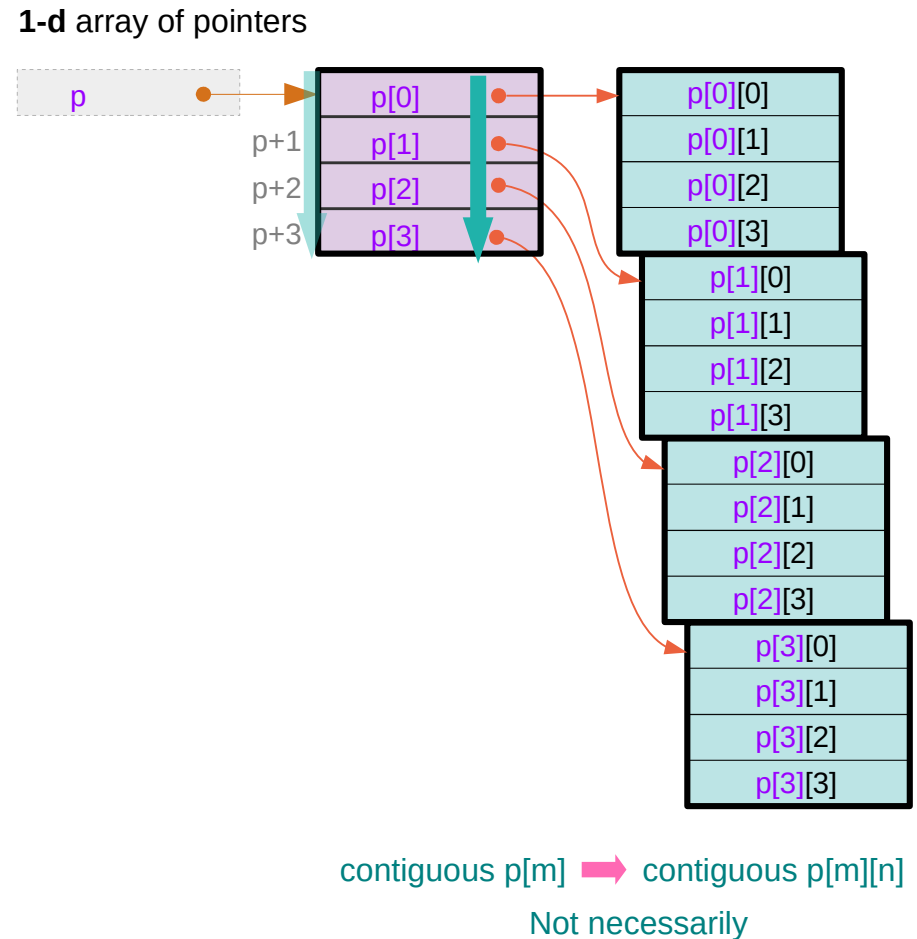
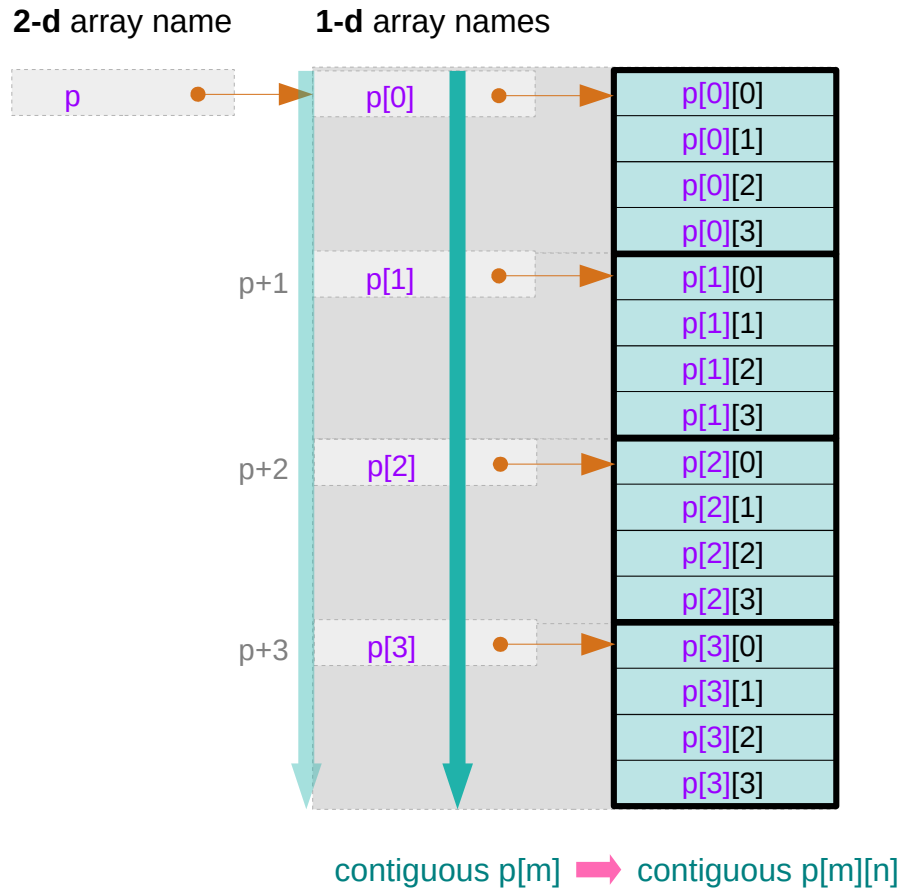
contiguous index : m



Contiguity constraints for p

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

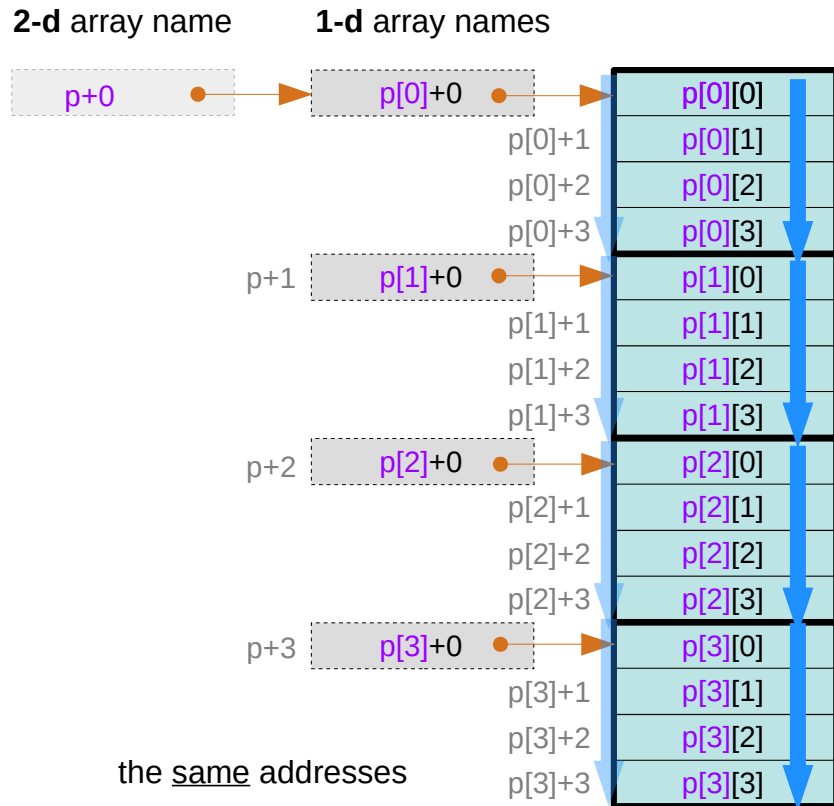
for a given p contiguous index : m



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

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

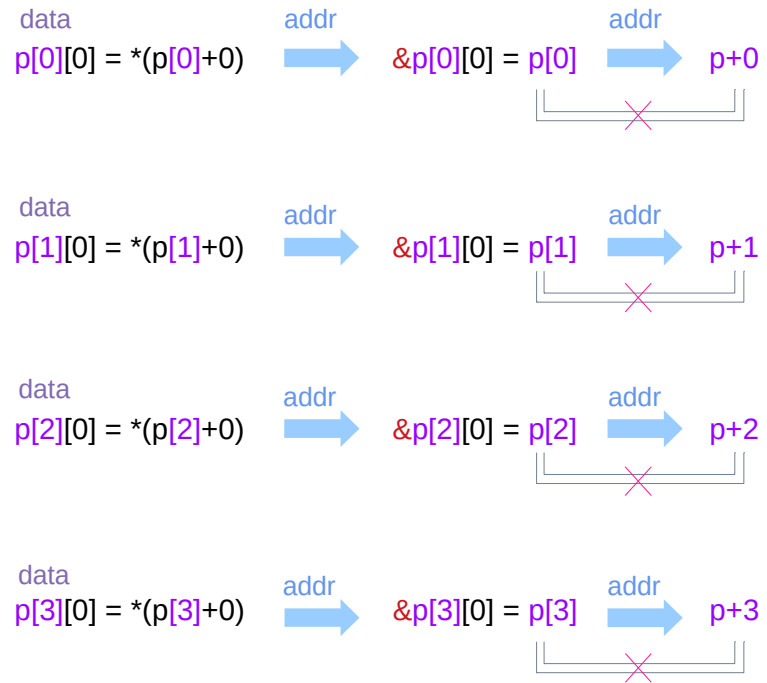
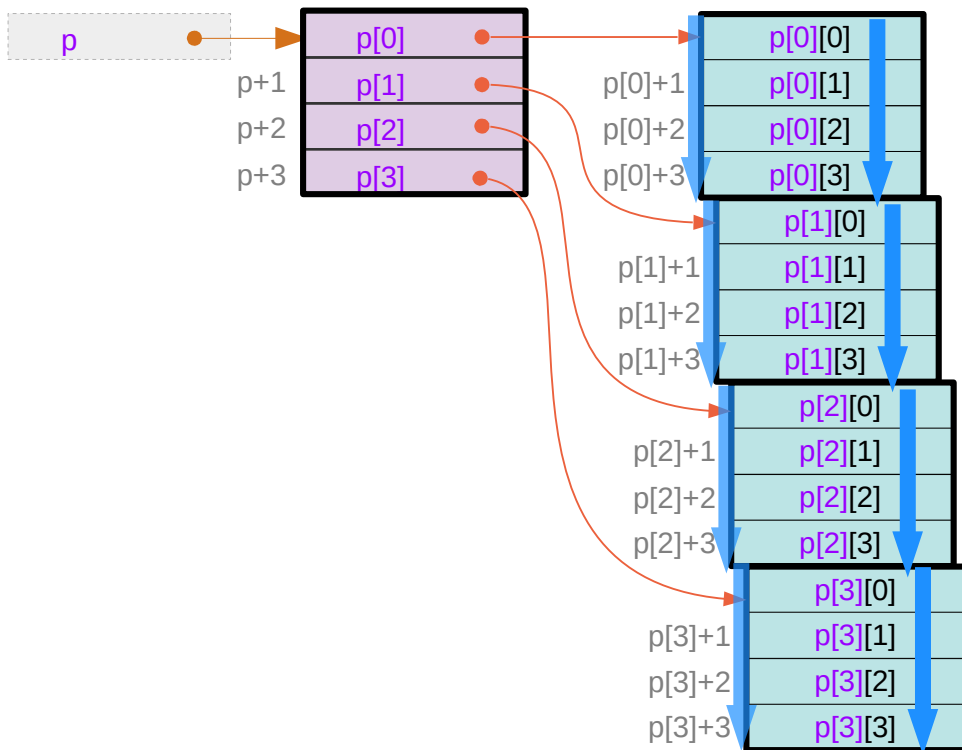
virtual array pointer \iff no real memory locations

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] ;
```

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

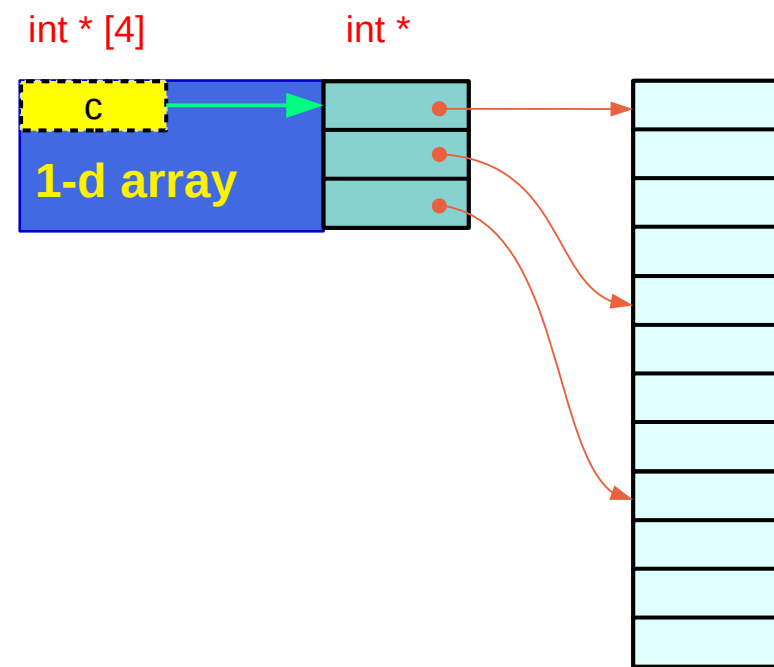
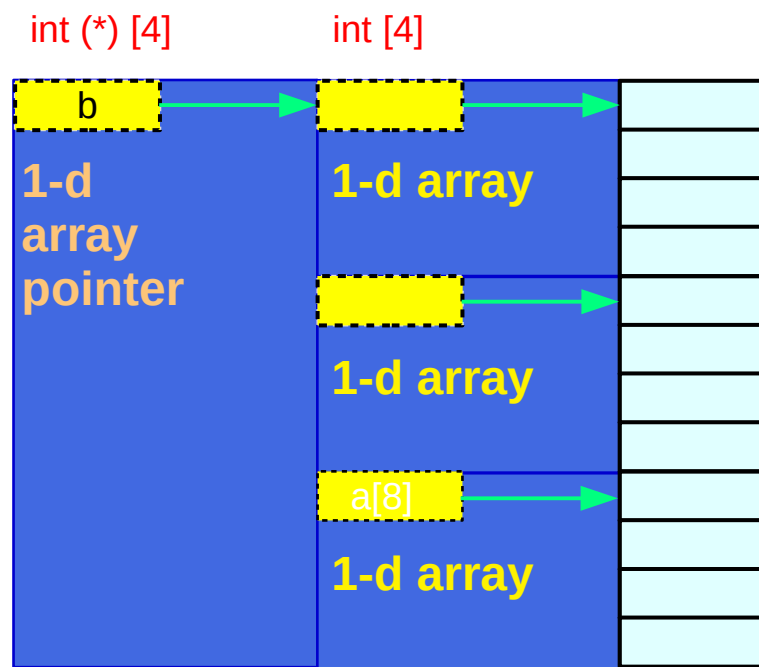
$c[0], c[1], \dots, c[M-1]$
are contiguous

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

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

a set of assignments of pointers
are necessary for this contiguity

Pointer Arrays vs Array Pointers



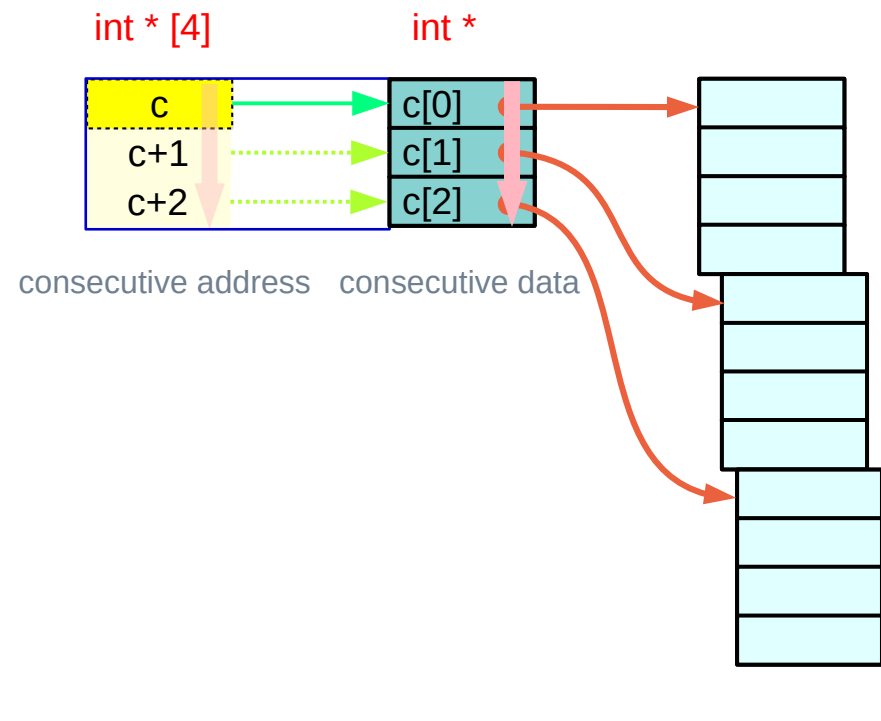
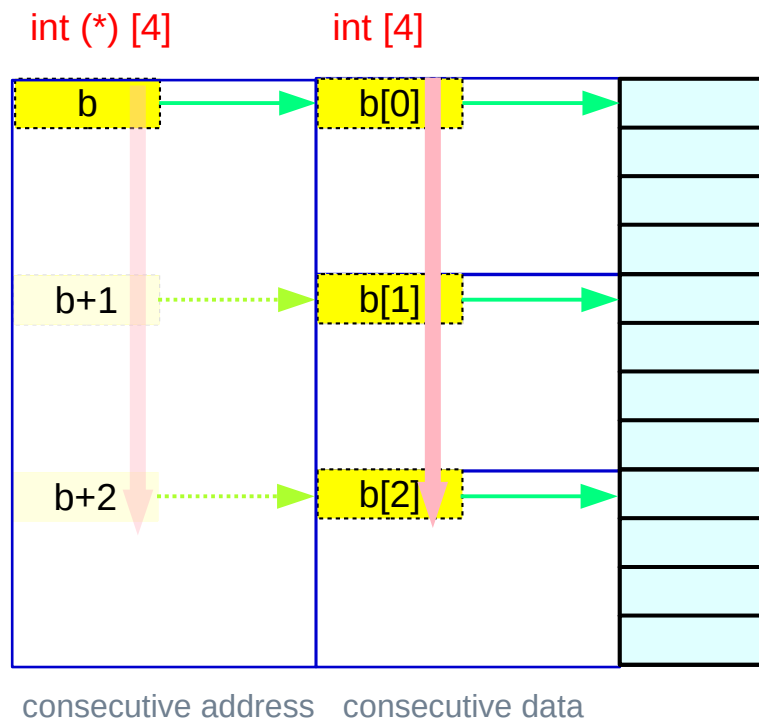
`int (*b)[N] ;`

`*(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] ;`

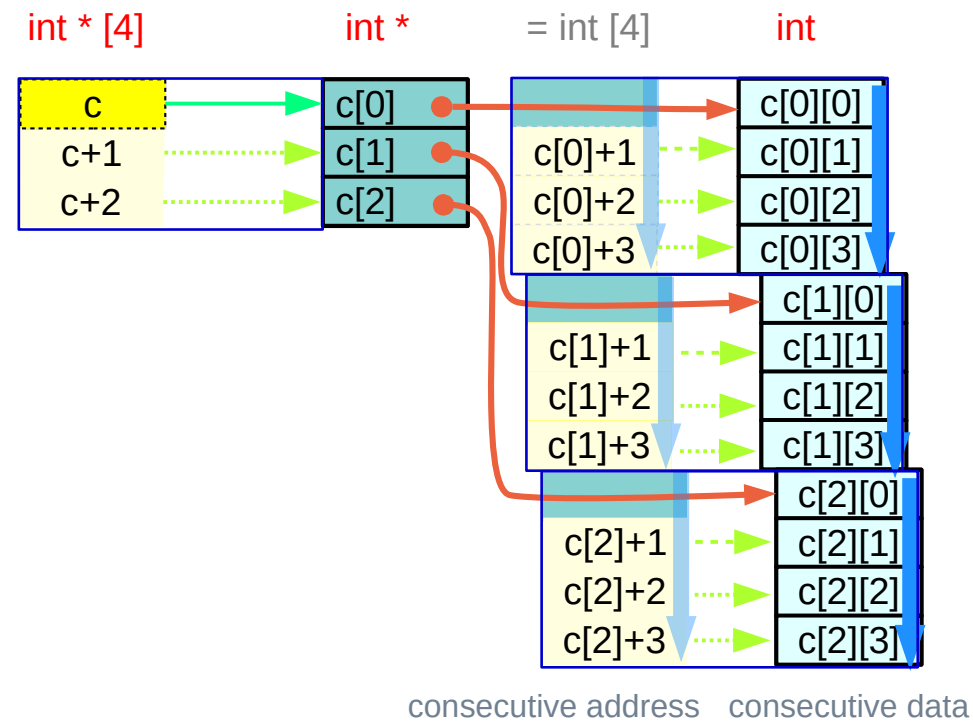
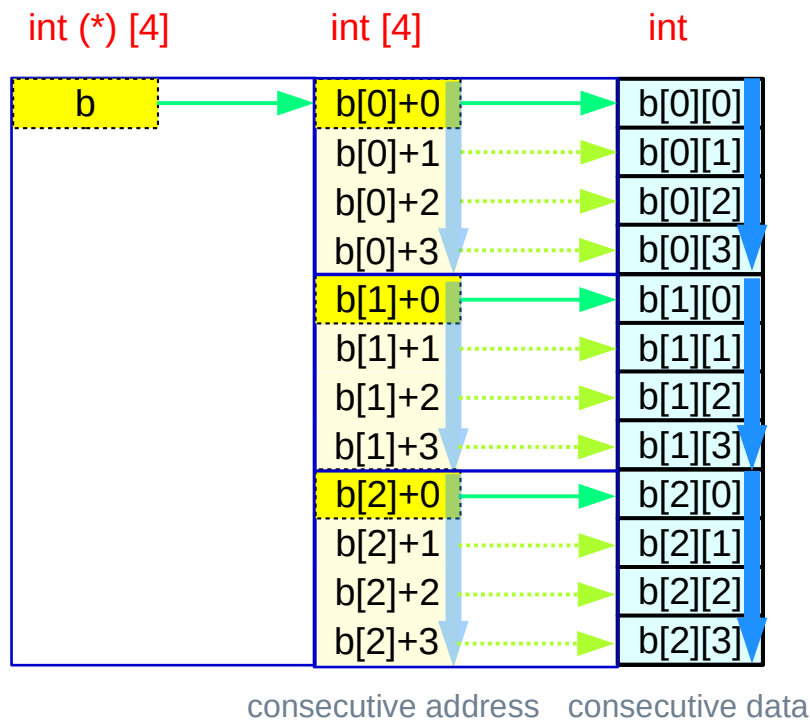
`*(b+m)` \longleftrightarrow `b[m]`
`*(b[m]+n)` \longleftrightarrow `b[m][n]`

`int * c[M] ;`

with proper assignments

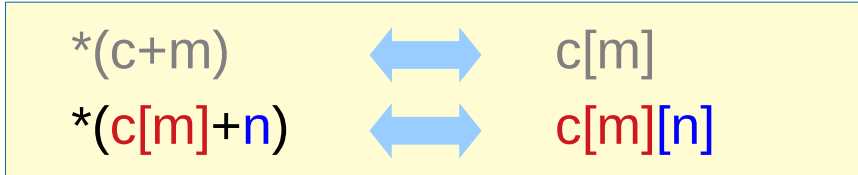
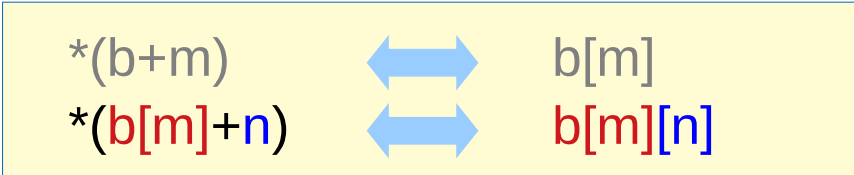
`*(c+m)` \longleftrightarrow `c[m]` or
`*(c[m]+n)` \longleftrightarrow `c[m][n]`

Pointer Arrays vs Array Pointers



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

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



Three contiguity constraints

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

$c[i][j][k] \equiv *(c[i][j] + k)$ contiguous 0-d arrays

```

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)

```

• •
• •
• •

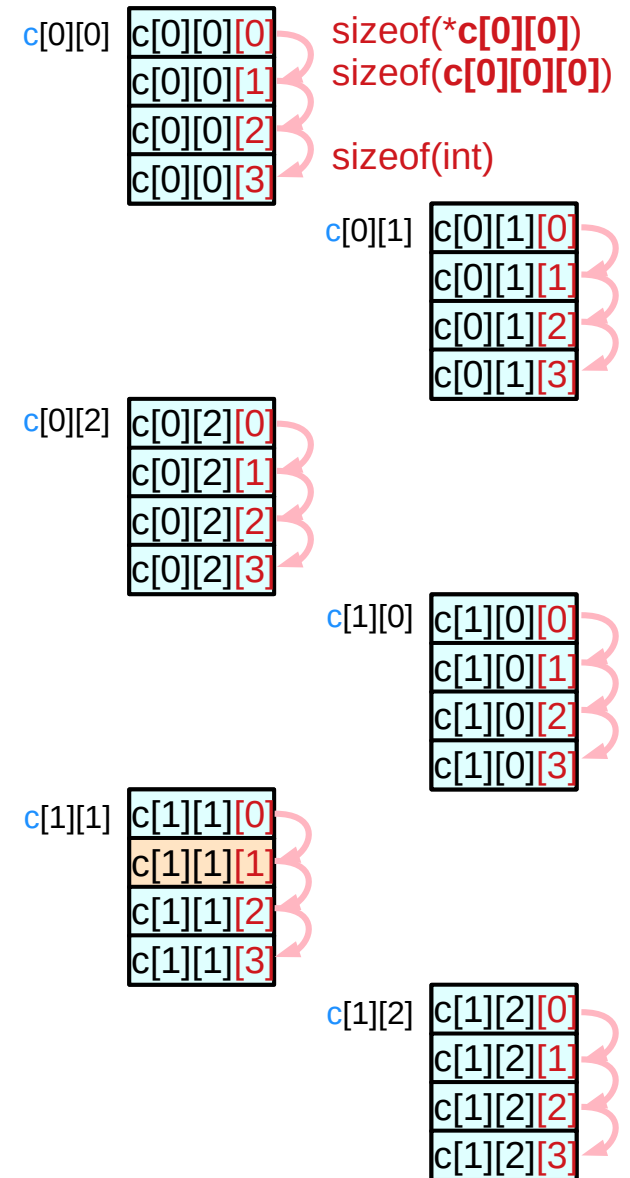
contiguous 1-d
array elements

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

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



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

contiguous 1-d arrays

```

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

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

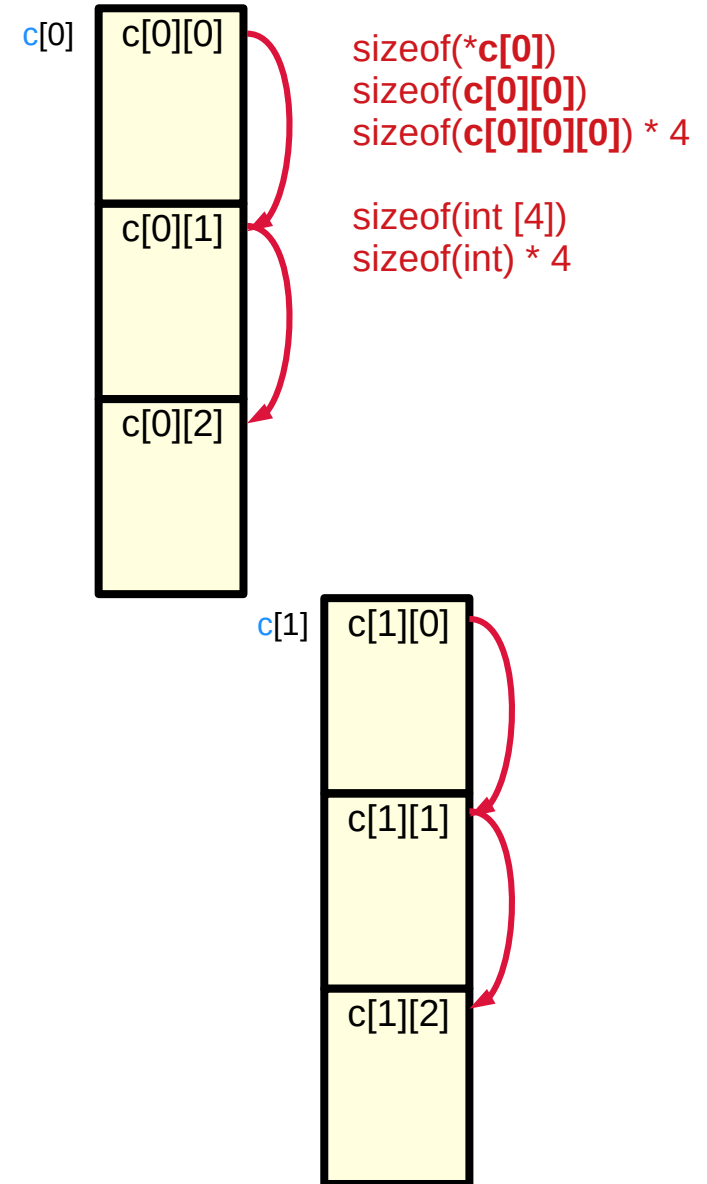
Address Value

c[i] + j

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

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

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



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

contiguous 2-d arrays

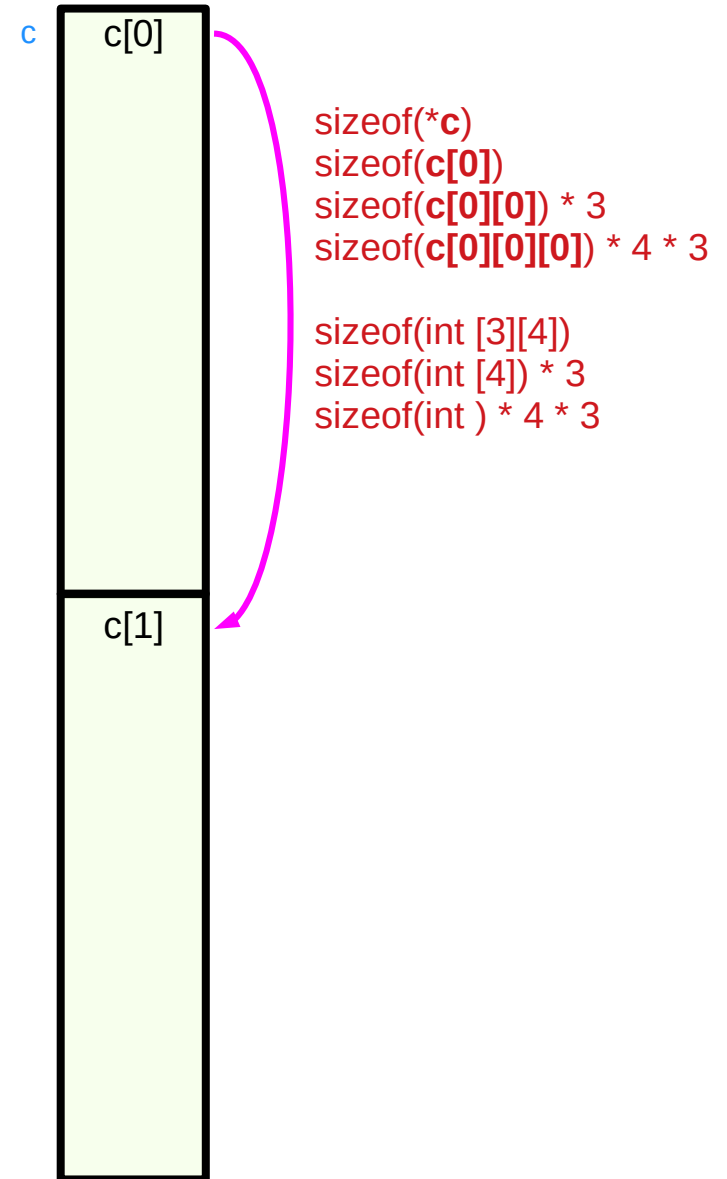
```
c[0] = *(c + 0)
c[1] = *(c + 1)
```

c
int [2][3][4] 2 contiguous 2-d arrays
int (*) [3][4] points to the 1st 2-d array
int [3][4] 2-d array

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



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

```

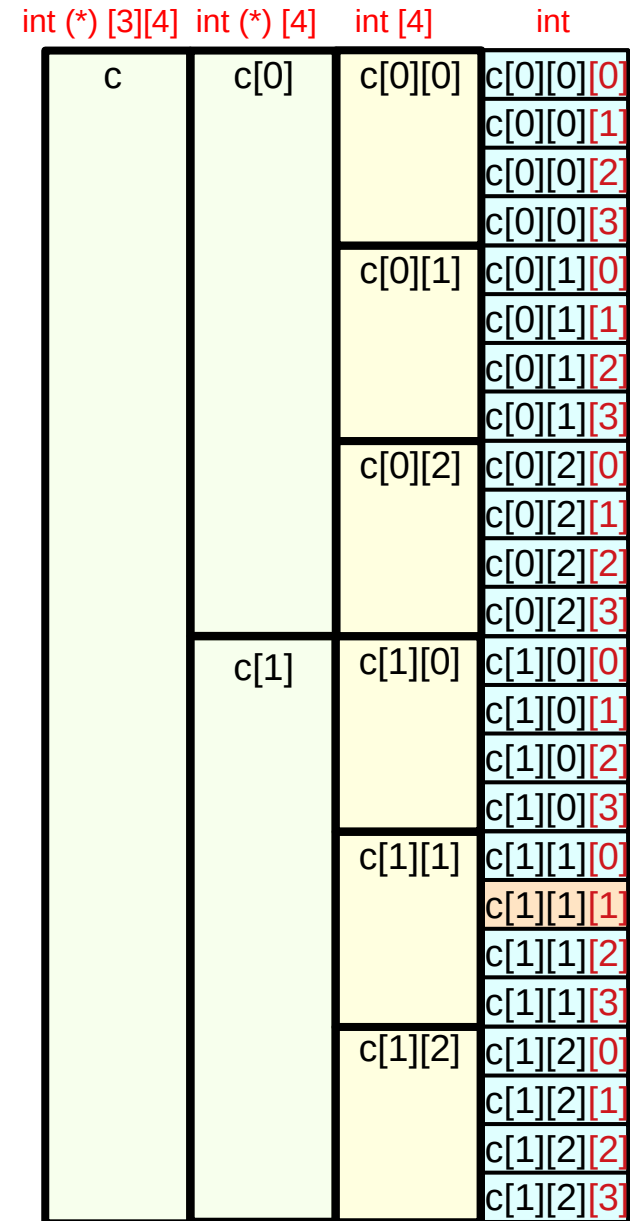
c[i][j][k]      → *(c[i][j] + k)
*(c[i][j] + k) → *(*c[i] + j) + k)
 *(*c[i] + j) + k → *(*(*c + i) + j) + k)

```

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

- contiguous 1-d array elements int
- contiguous 1-d array names int (*)
- contiguous 1-d array pointers int (*) [4]

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



Contiguity Constraints

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

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

address value $c + i$

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

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

$c[0][0][0]$

$c[0][1][0]$

$c[0][2][0]$

$c[1][0][0]$

$c[1][1][0]$

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

Contiguous linear layout

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

L	M	N
<i>i</i>	<i>j</i>	<i>k</i>
$i * M * N$	$j * N$	<i>k</i>

Base Index = 0

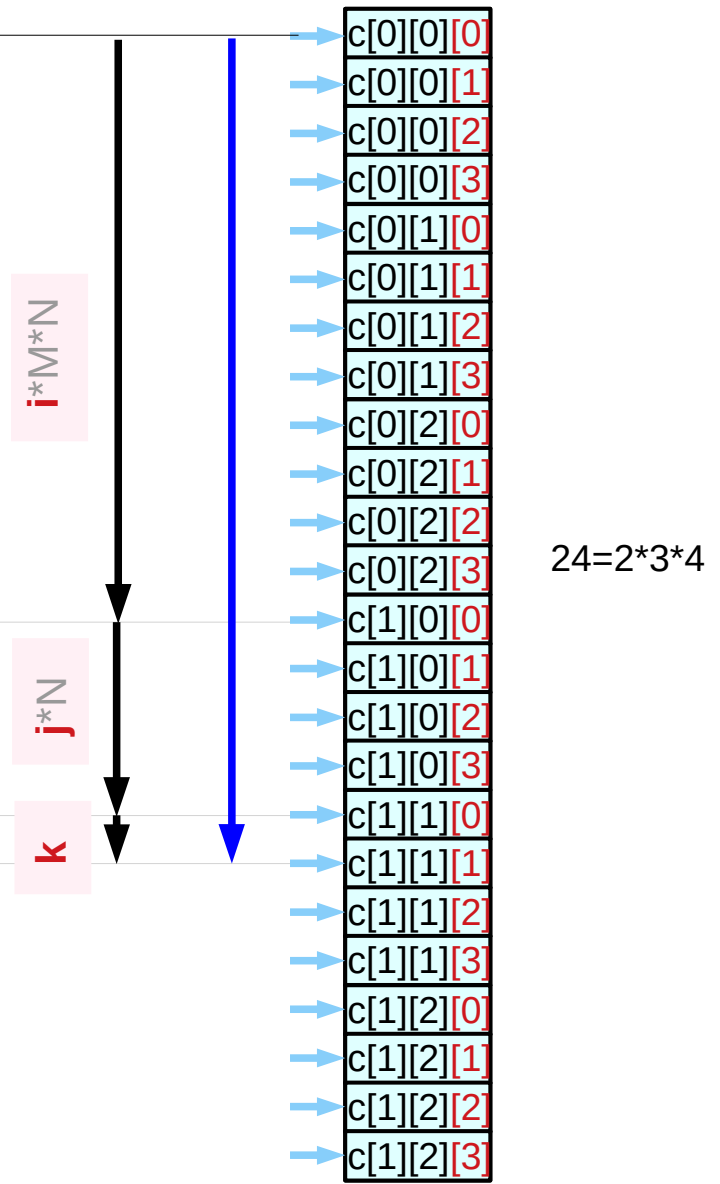
Offset Index 1 (*i*=1)

Offset Index 2 (*j*=1)

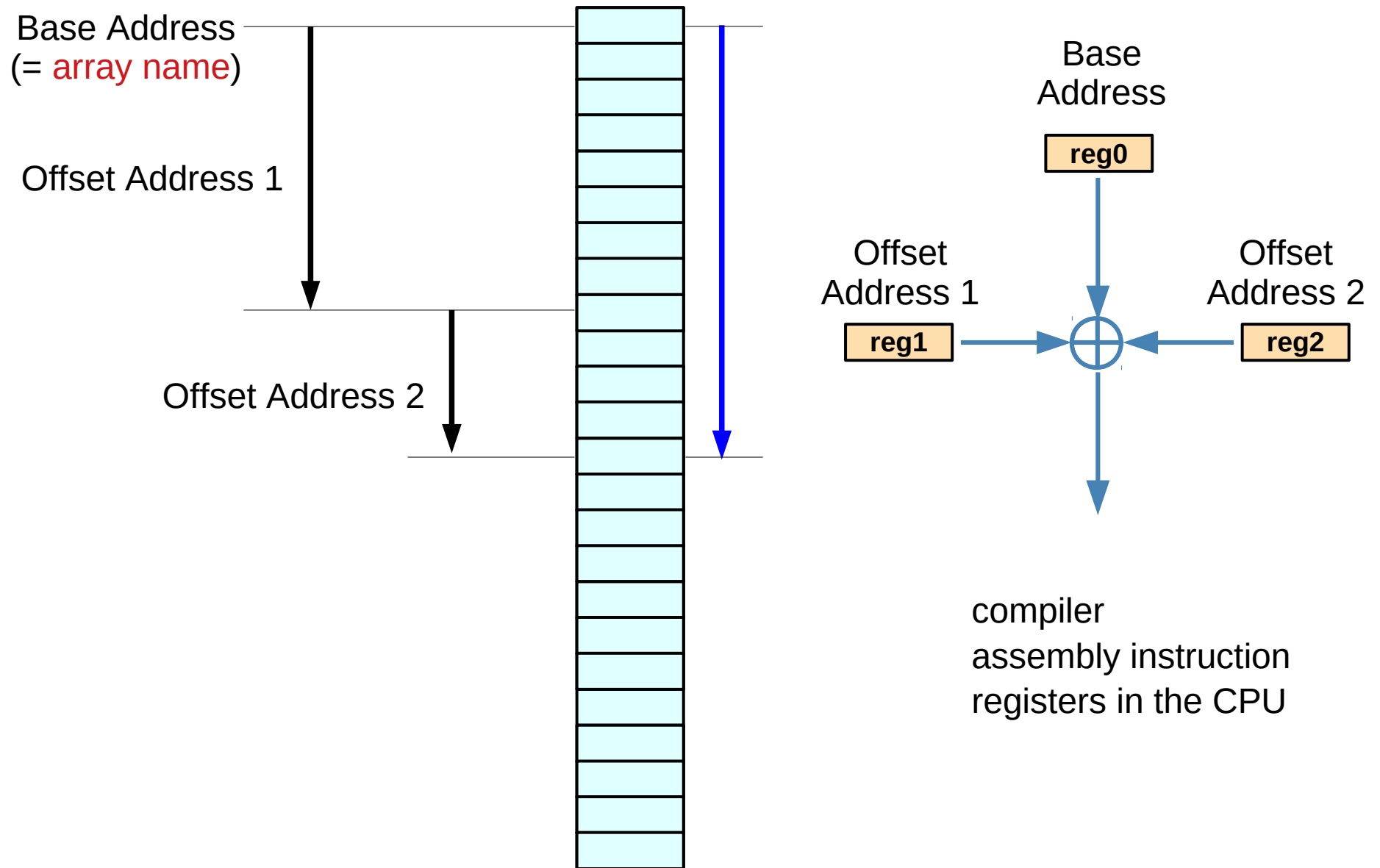
Offset Index 3 (*k*=1)

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

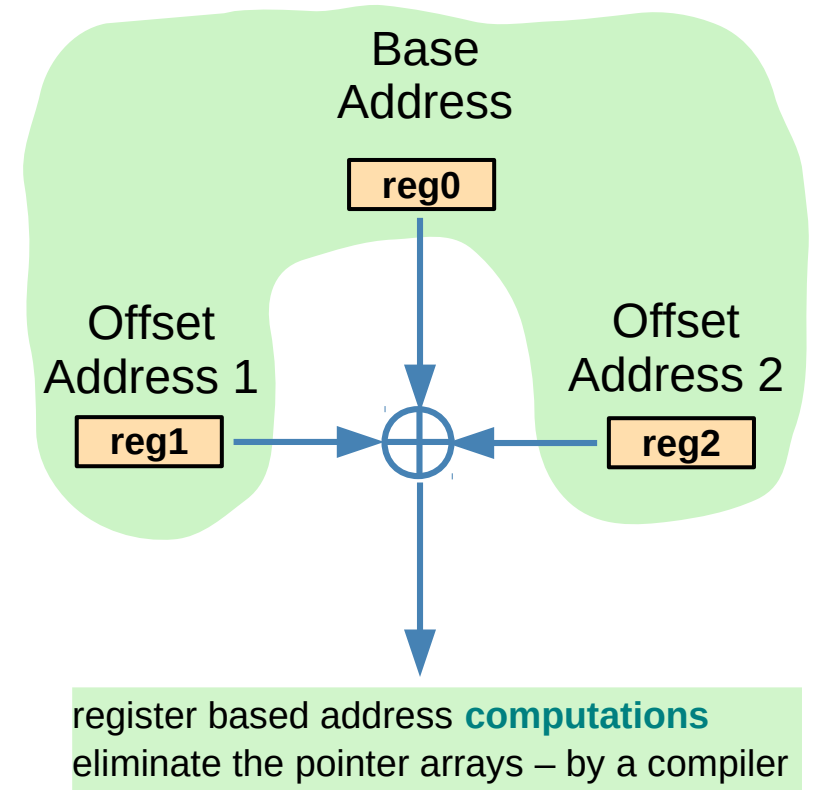
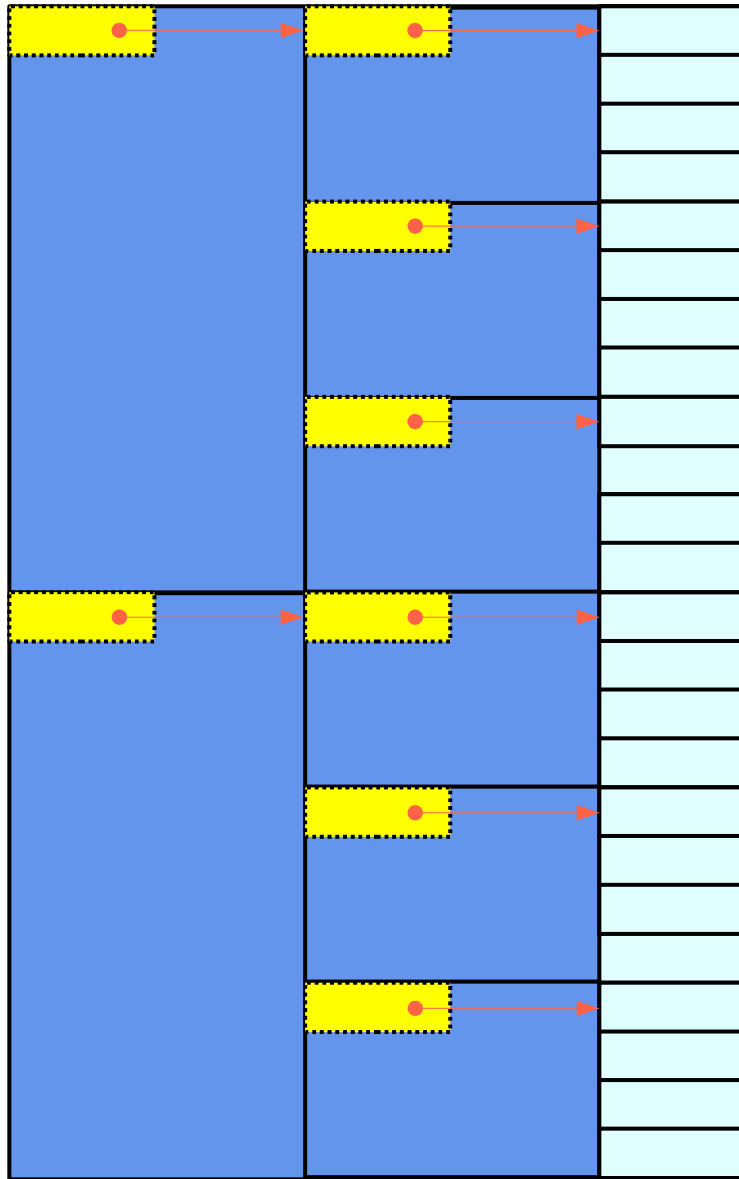
$$((i * M + j) * N + k)$$



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