

Applications of Arrays (1A)

Copyright (c) 2024 - 2010 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.

-
- Viewing an **array** as a **pointer**
 - Viewing a **pointer** as an **array**

- Viewing an **array** as a **pointer**

`int a[4];`

an array **a**

generalization



`int (*a)`

view **a** as a pointer

virtual pointer

- no real memory location

- constraints :

`value(&a) = value(a)`

- Viewing a **pointer** as an **array**

`int (*a);`

a pointer **a**

a specific instance



`int a[N]`

view **a** as an array

N is not fixed

`sizeof(a)` is

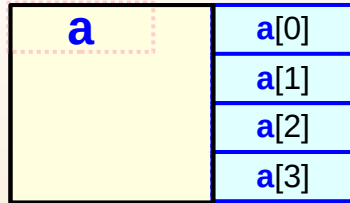
not the size of the array

but of a pointer variable

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

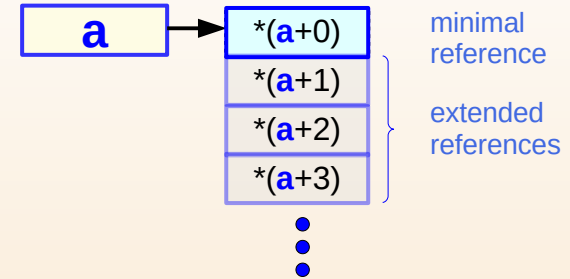
`int a[4];`

an array **a**



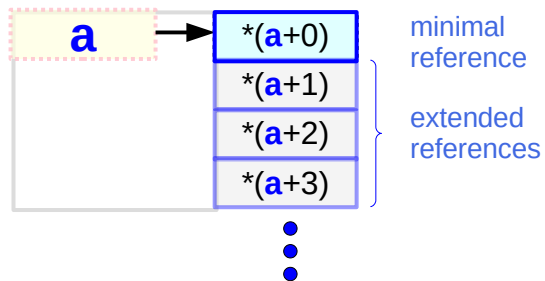
`int (*a);`

a pointer **a**



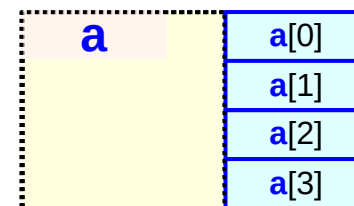
`int (*a)`

a as a pointer



`int a[N]`

a as an array



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

`int a[4];` an array **a**

- `type(a)` = `int [4]`
- `sizeof(a)` = an array size (16 bytes)
- `value(&a)` = `value(a)`
- fixed number of elements

`int (*a)` **a** as a pointer

a is not a real pointer

- `sizeof(a)` = an array size
- `value(&a)` = `value(a)`

`int (*a);` a pointer **a**

- `type(a)` = `int (*)`
- `sizeof(a)` = a pointer size (4 bytes)
- `value(&a)` \neq `value(a)`
- variable number of elements

`int a[N]` **a** as an array

a is not a real array

- `sizeof(a)` \neq an array size
= a pointer size
- `value(&a)` \neq `value(a)`
= assigned address

Relationship between array and array pointer types

`int b[4][2];` declare a **2-d** array **b**

↓ generalization

`int (*b)[2]` **b** as a **1-d** array pointer

`int a[4];` declare a **1-d** array **a**

↓ generalization

`int (*a)` **a** as a **0-d** array pointer

`int (*b)[2];` declare a **1-d** array pointer **b**

↓ a specific instance

`int b[N][2]` **b** as a **2-d** array

`int (*a);` declare a **0-d** array pointer **a**

↓ a specific instance

`int a[N]` **a** as a **1-d** array

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

`int b[4][2];` 2-d array **b**

- `type(b)` = `int [4]`
- `sizeof(b)` = an array size (32 bytes)
- `value(&b)` = `value(b)`
- fixed number of elements

`int (*) [2]` **b** as a 1-d array pointer

b is not a real pointer

- `sizeof(b)` = an array size
- `value(&b)` = `value(b)`

`int (*b) [2];` 1-d array pointer **b**

- `type(b)` = `int (*)`
- `sizeof(b)` = a pointer size (4 bytes)
- `value(&b)` \neq `value(b)`
- variable number of elements

`int [N][2]` **b** as a 2-d array

b is not a real array

- `sizeof(b)` \neq an array size
= a pointer size
- `value(&b)` \neq `value(b)`
= assigned address

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

`int b[4][2];`

2-d array **b**

b	b[0]	b[0][0]
		b[0][1]
	b[1]	b[1][0]
		b[1][1]
	b[2]	b[2][0]
		b[2][1]
	b[3]	b[3][0]
		b[3][1]

`int (*) [2]`

b as a 1-d array pointer

b	*(b+0)	(*(b+0))[0]
		(*(b+0))[1]
	(b+1)	((b+1))[0]
		(*(b+1))[1]
	(b+2)	((b+2))[0]
		(*(b+2))[1]
	(b+3)	((b+3))[0]
		(*(b+3))[1]

minimal reference

extended references

virtual pointer
 - no real memory location
 - constraints :
`&b = b`

`int (*b) [2];`

1-d array pointer **b**

b	*(b+0)	(*(b+0))[0]
		(*(b+0))[1]
	(b+1)	((b+1))[0]
		(*(b+1))[1]
	(b+2)	((b+2))[0]
		(*(b+2))[1]
	(b+3)	((b+3))[0]
		(*(b+3))[1]

minimal reference

extended references

`int [N][2]`

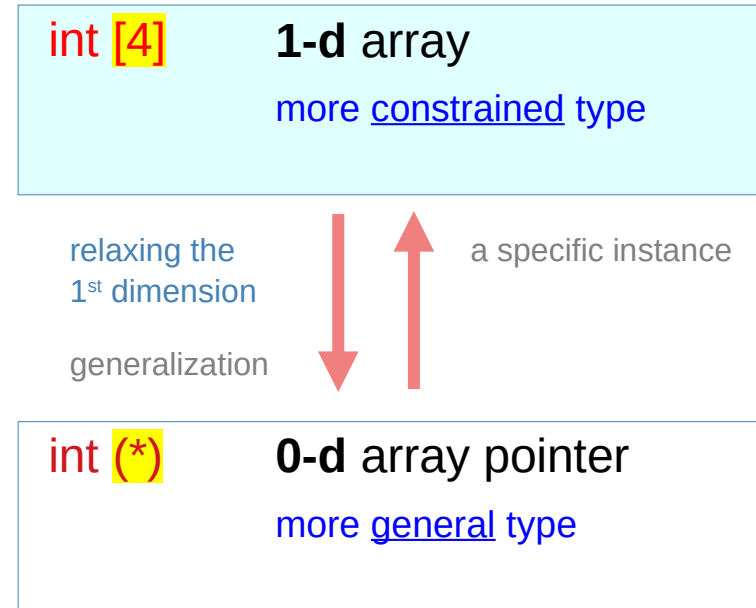
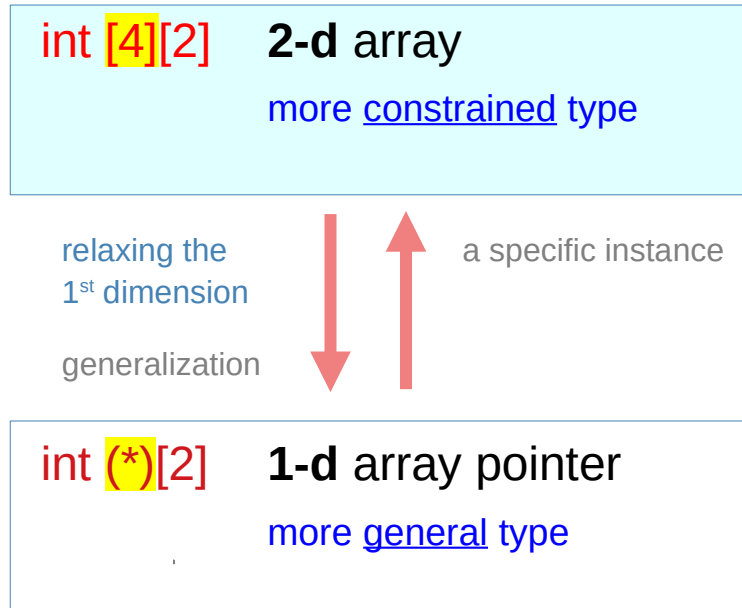
b as a 2-d array

b	b[0]	b[0][0]
		b[0][1]
	b[1]	b[1][0]
		b[1][1]
	b[2]	b[2][0]
		b[2][1]
	b[3]	b[3][0]
		b[3][1]

N is not fixed to 4

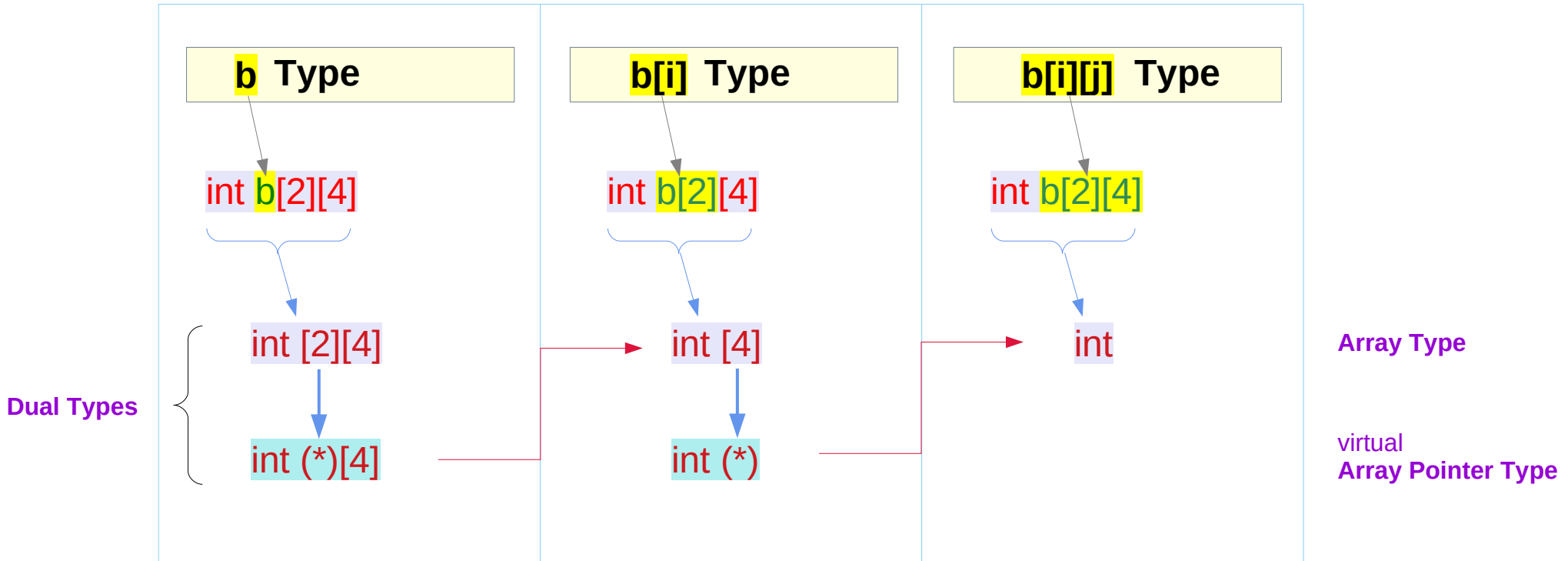
`sizeof(b)` is not the size of the array but the size of a pointer variable

Dual type - relaxing the 1st dimension of an array



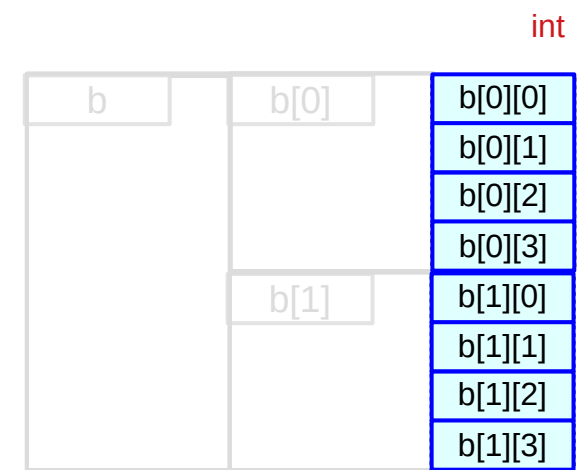
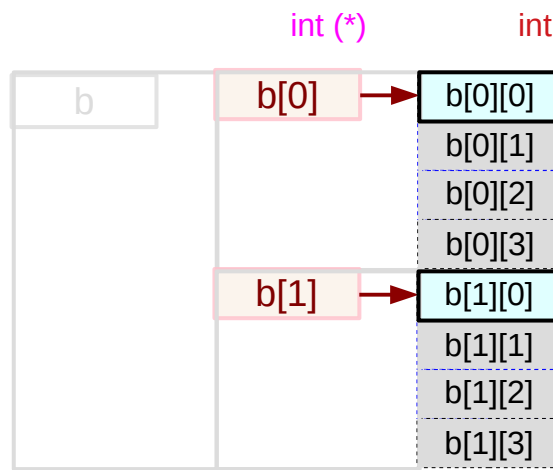
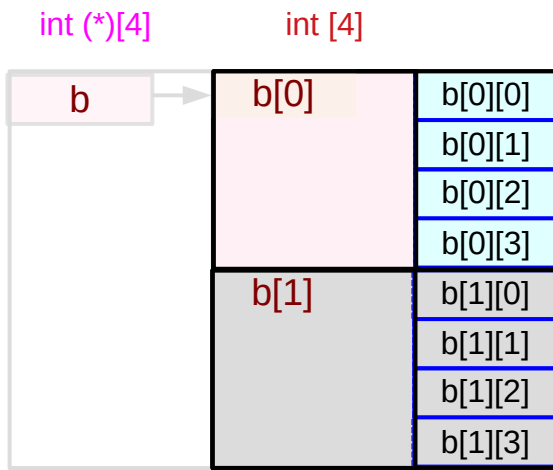
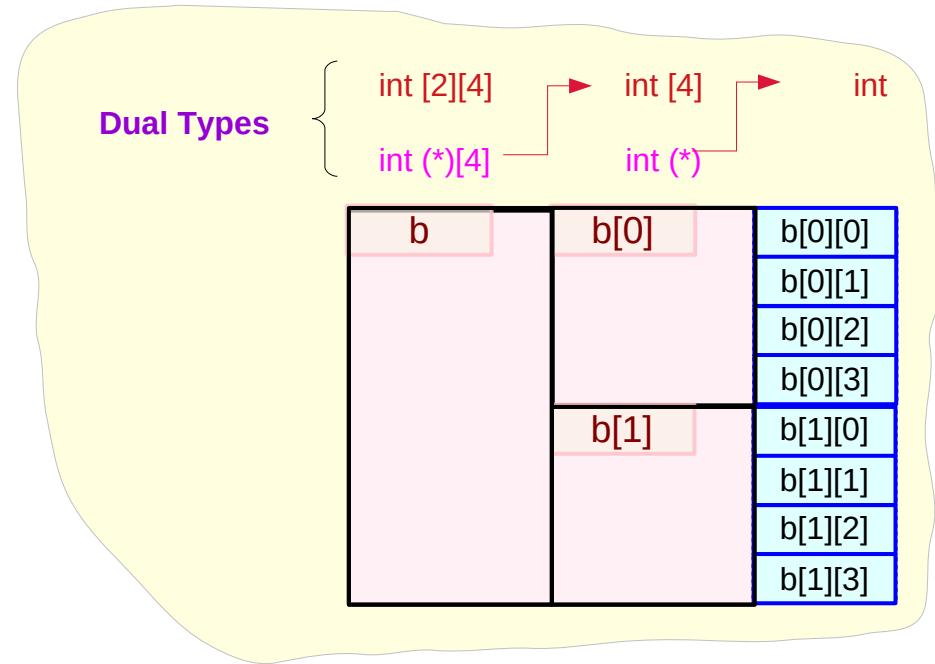
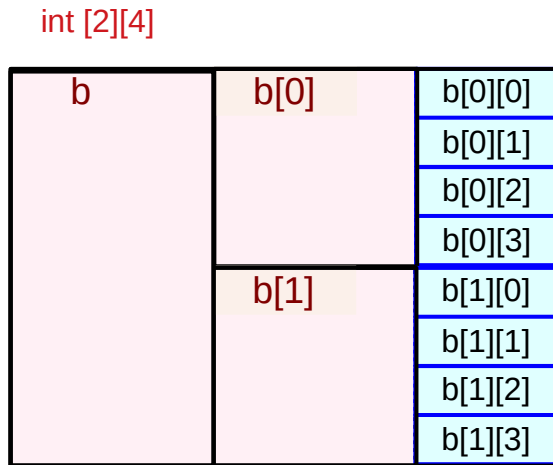
Subarray types in a 2-d array

`int b[2][4];` 2-d array `b`



Dual types in a 2-d array

`int b[2][4];` 2-d array **b**



Subarray type examples

```
int a[4];
```

			relaxed type	virtual
a	int [4]	1-d array type	int (*)	0-d array pointer type
a[i]	int	0-d array type		

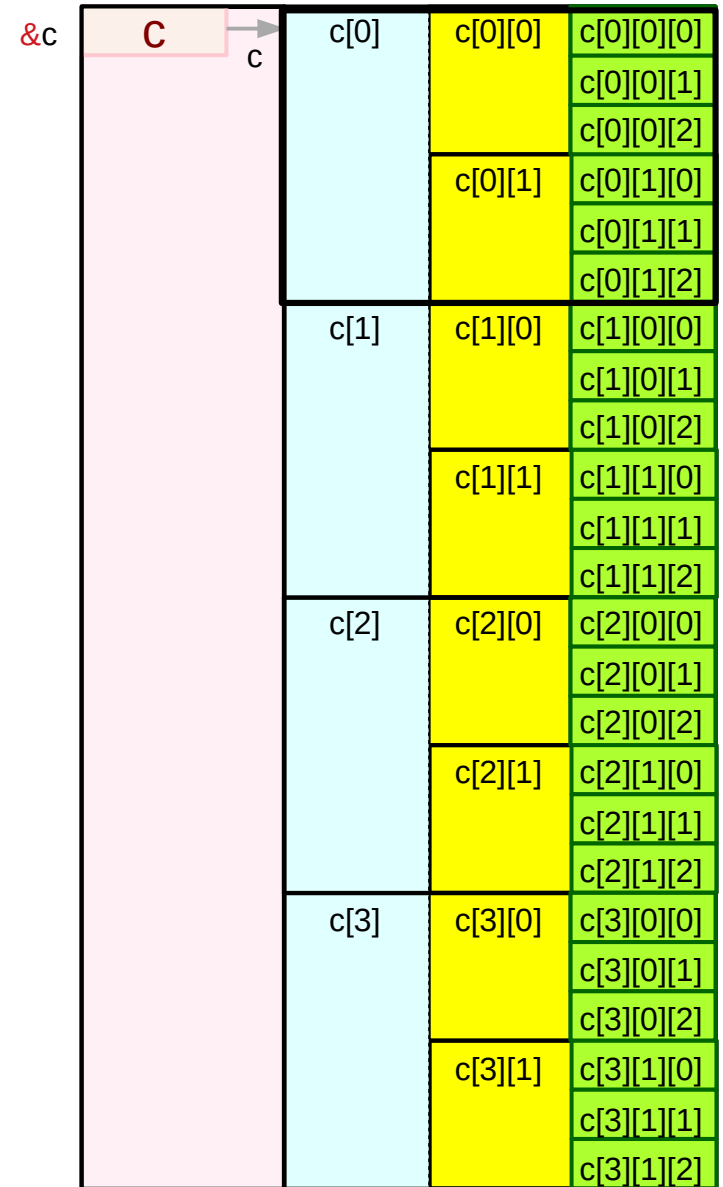
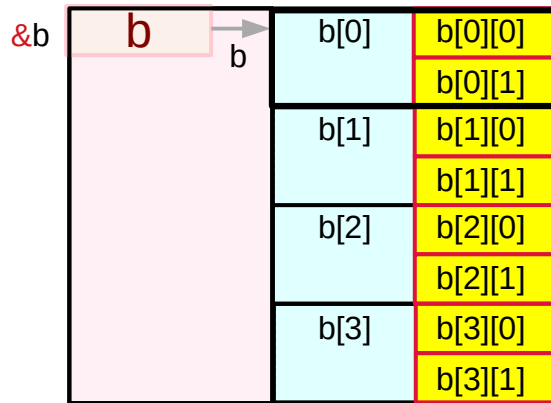
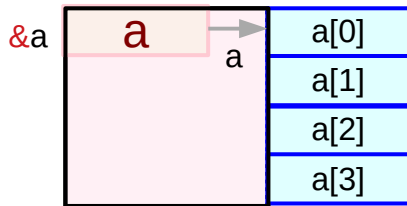
```
int b[2][4];
```

			relaxed type	virtual
b	int [2][4]	2-d array type	int (*)[4]	1-d array pointer type
b[i]	int [4]	1-d array type	int (*)	0-d array pointer type
b[i][j]	int	0-d array type		

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

			relaxed type	virtual
c	int [4][2][3]	3-d array type	int (*)[2][3]	2-d array pointer type
c[i]	int [4][2]	2-d array type	int (*)[2]	1-d array pointer type
c[i][j]	int [4]	1-d array type	int (*)	0-d array pointer type
c[i][j][k]	int	0-d array type		

Types of **a**, **b**, **c** arrays



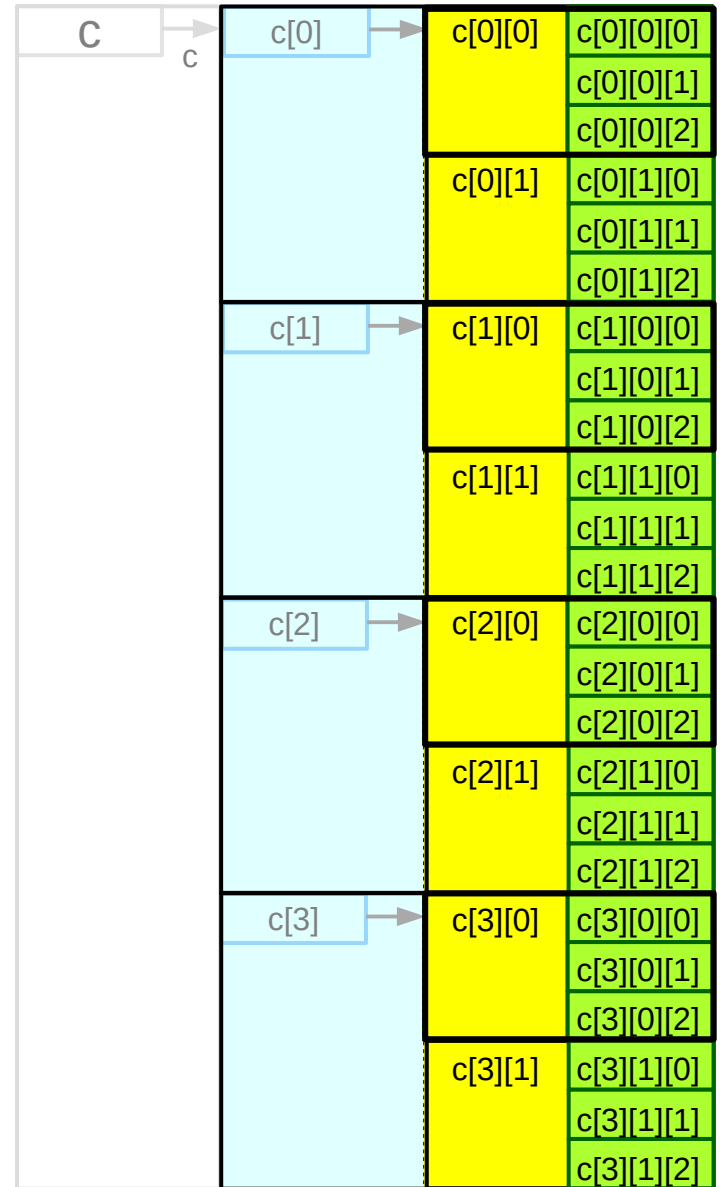
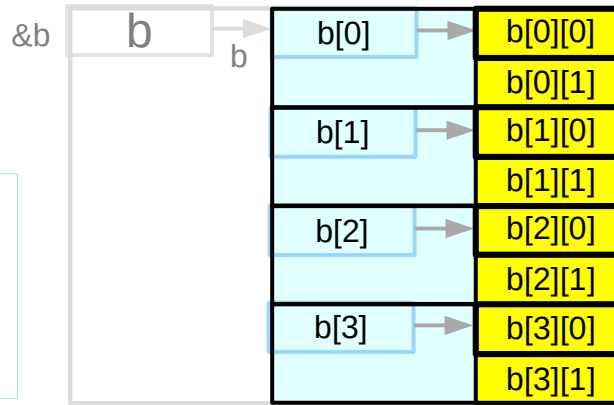
```
int a[4];
int b[2][4];
int c[4][2][3];
```

dual types

int [4]	1-d array a	a[i]
int (*)	0-d array pointer a (virtual)	*(a+i)
int [4][2];	2-d array b	b[i]
int (*)[2];	1-d array pointer b (virtual)	*(b+i)
int [4][2][3];	3-d array c	c[i]
int (*)[2][3];	2-d array pointer c (virtual)	*(c+i)

Types of $b[i]$, $c[i]$ subarrays

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



dual types

<code>int [2]</code>	1-d array $b[i]$	<code>$b[i][j]$</code>
<code>int (*)</code>	0-d array pointer $b[i]$ (virtual)	<code>$*(b[i]+j)$</code>
<code>int [2][3];</code>	2-d array $c[i]$	<code>$c[i][j]$</code>
<code>int (*)[3];</code>	1-d array pointer $c[i]$ (virtual)	<code>$*(c[i]+j)$</code>

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

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

dual types

<code>int [3]</code>	1-d array <code>c[i][j]</code>	<code>c[i][j][k]</code>
<code>int (*)</code>	0-d array pointer <code>c[i][j]</code> (virtual)	<code>*(c[i][j]+k)</code>



Types of a 4-d array and its subarrays

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

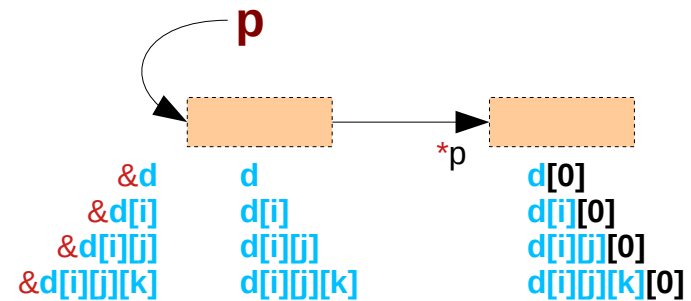
types

d	consider d [4][2][3][4]	→	int [4][2][3][4]	⇒	4-d array
	relax the 1 st dimension	→	int (*)[2][3][4]	⇒	3-d array pointer (virtual)
d [i]	consider d [i][2][3][4]	→	int [2][3][4]	⇒	3-d array
	relax the 1 st dimension	→	int (*)[3][4]	⇒	2-d array pointer (virtual)
d [i][j]	consider d [i][j][3][4]	→	int [3][4]	⇒	2-d array
	relax the 1 st dimension	→	int (*)[4]	⇒	1-d array pointer (virtual)
d [i][j][k]	consider d [i][j][k][4]	→	int [4]	⇒	1-d array
	relax the 1 st dimension	→	int (*)	⇒	0-d array pointer (virtual)

i,j,k are specific index values i=[0..3], j=[0..1], k=[0..2]

Initializing n -d array pointers with n -d subarrays

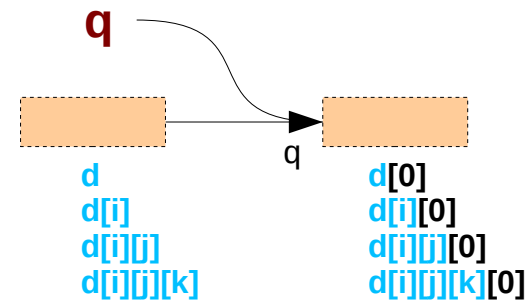
```
int d[4][2][3][4];
```



d	4-d array	$d[4][2][3][4]$	$p = \&d$	abstract data
p	4-d array pointer	$(*p)[4][2][3][4]$	$\text{int } (*p)[4][2][3][4] = \&d;$ $(*p)[i][j][k][l] \equiv d[i][j][k][l]$	
$d[i]$	3-d array	$d[i][2][3][4]$	$p = \&d[i]$	abstract data
p	3-d array pointer	$(*p)[2][3][4]$	$\text{int } (*p)[3][4] = \&d[i];$ $(*p)[j][k][l] \equiv d[i][j][k][l]$ given i	
$d[i][j]$	2-d array	$d[i][j][3][4]$	$p = \&d[i][j]$	abstract data
p	2-d array pointer	$(*p)[3][4]$	$\text{int } (*p)[4] = \&d[i][j];$ $(*p)[k][l] \equiv d[i][j][k][l]$ given i, j	
$d[i][j][k]$	1-d array	$d[i][j][k][4]$	$p = \&d[i][j][k]$	abstract data
p	1-d array pointer	$(*p)[4]$	$\text{int } (*p) = \&d[i][j][k];$ $(*p)[l] \equiv d[i][j][k][l]$ given i, j, k	

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

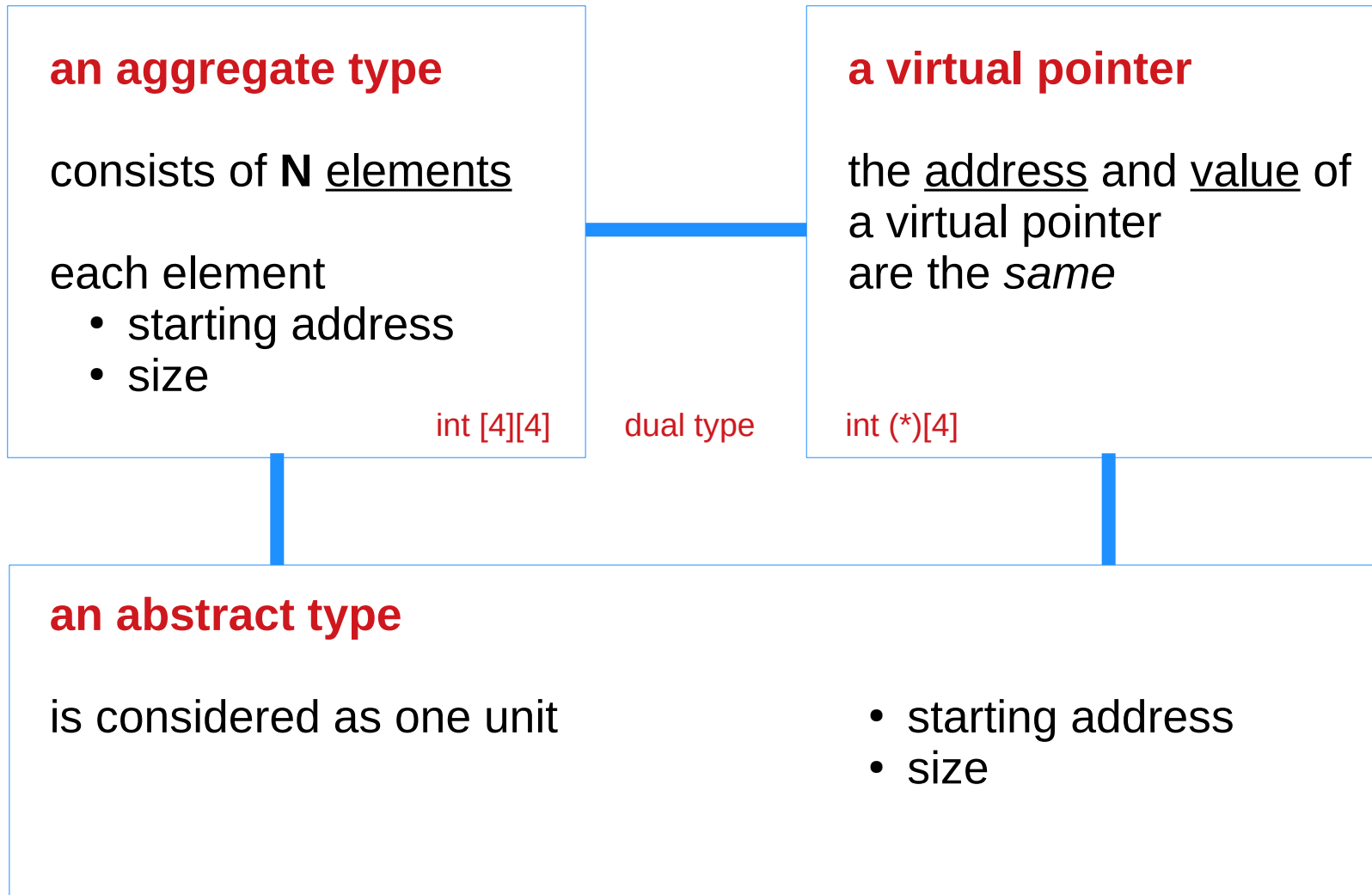
```
int d[4][2][3][4];
```



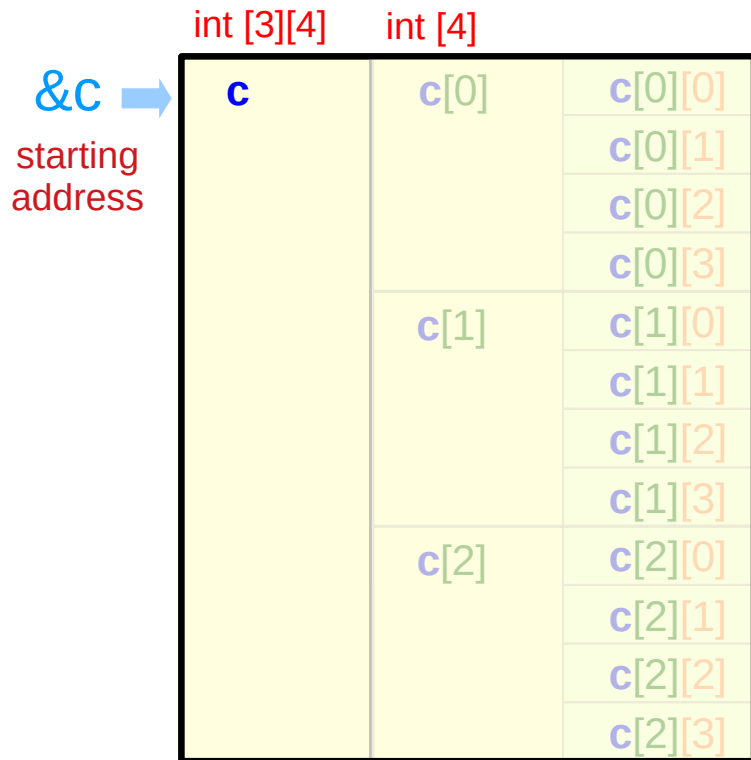
d	4-d array	<code>d[4][2][3][4]</code>	<code>q = d</code>	virtual pointer
q	3-d array pointer	<code>(*q)[2][3][4]</code>	<code>int (*q)[2][3][4] = d;</code> <code>q[i][j][k][l] ≡ d[i][j][k][l]</code>	
d[i]	3-d array	<code>d[i][2][3][4]</code>	<code>q = d[i]</code>	virtual pointer
q	2-d array pointer	<code>(*q)[3][4]</code>	<code>int (*q)[3][4] = d[i];</code> <code>q[j][k][l] ≡ d[i][j][k][l]</code> given i	
d[i][j]	2-d array	<code>d[i][j][3][4]</code>	<code>q = d[i][j]</code>	virtual pointer
q	1-d array pointer	<code>(*q)[4]</code>	<code>int (*q)[4] = d[i][j];</code> <code>q[k][l] ≡ d[i][j][k][l]</code> given i, j	
d[i][j][k]	1-d array	<code>d[i][j][k][4]</code>	<code>q = d[i][j][k]</code>	virtual pointer
q	0-d array pointer	<code>(*q)</code>	<code>int (*q) = d[i][j][k];</code> <code>q[l] ≡ d[i][j][k][l]</code> given i, j, k	

Aggregate Data Types
Abstract Data Types
Virtual Array Pointers

Aggregate data type



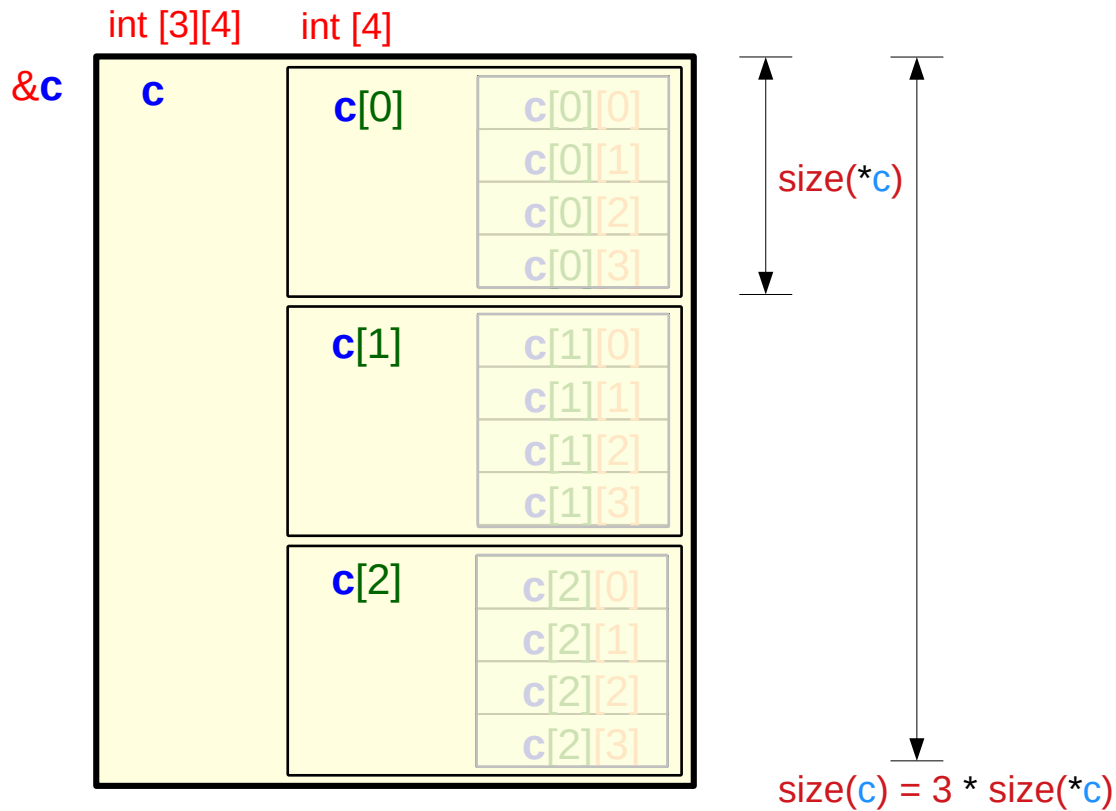
Abstract data **c**



an abstract data
- start address
- size

c
&c
sizeof(c)

Aggregate data **c**

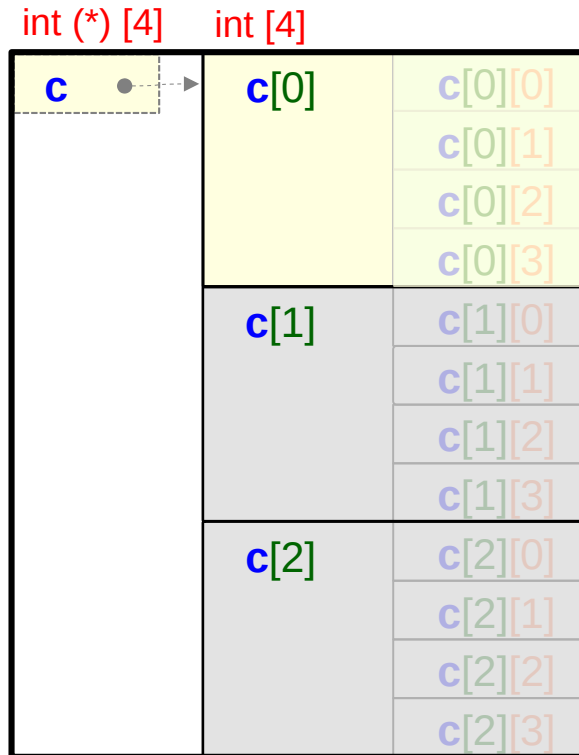


an aggregate type data **c**

- 1st element **c[0]**
- 2nd element **c[1]**
- 3rd element **c[2]**

Virtual pointer **c**

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



$\text{size}(*c)$

a virtual pointer **c**
- pointer address $\&c$
- pointer value $c = \&c[0]$

with the constraint
 $c = \&c$

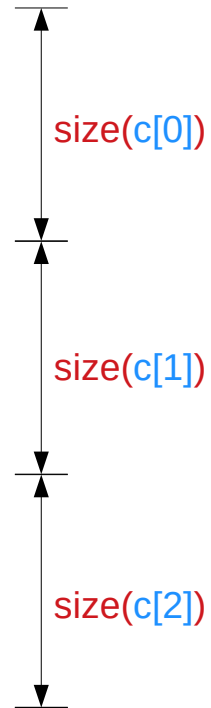
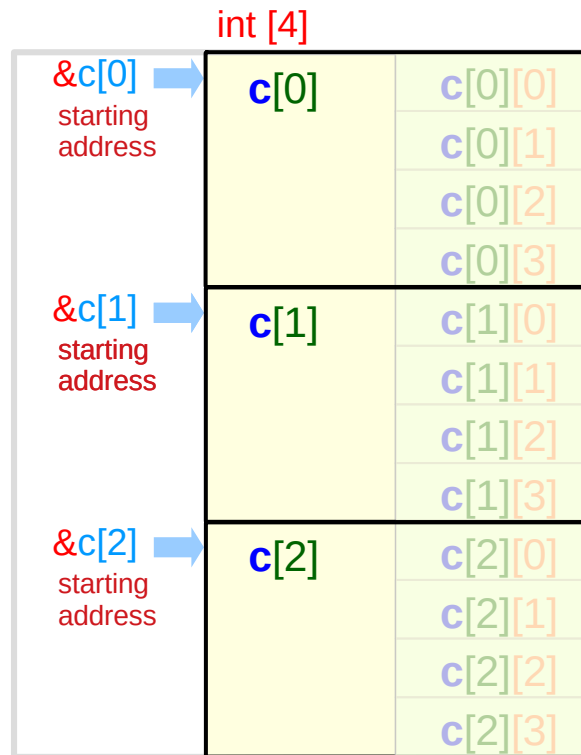
an abstract data $c[0] = *c$
- start address $\&c[0] = c$
- size $\text{sizeof}(c[0])$

virtual pointer **c** points
to abstract data **c[0]**

virtual pointers

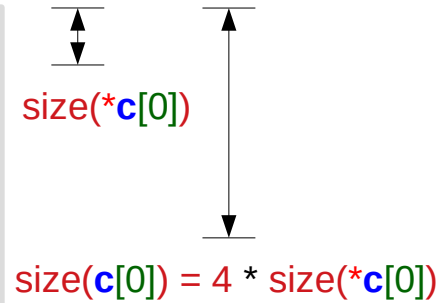
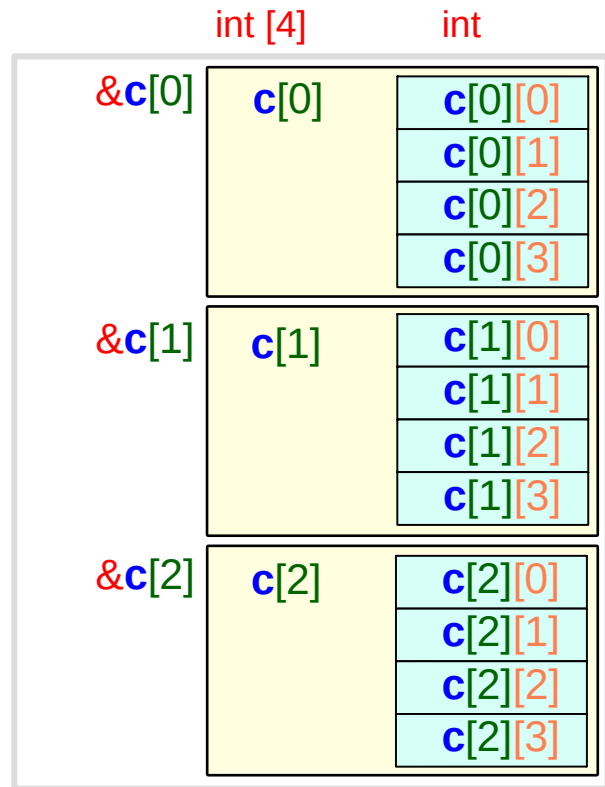
- no physical memory locations are allocated
- address and data have the same value

Abstract data $c[i]$



- an abstract data
 - start address $c[0]$
 - size $\&c[0]$
 - $sizeof(c[0])$
- an abstract data
 - start address $c[1]$
 - size $\&c[1]$
 - $sizeof(c[1])$
- an abstract data
 - start address $c[2]$
 - size $\&c[2]$
 - $sizeof(c[2])$

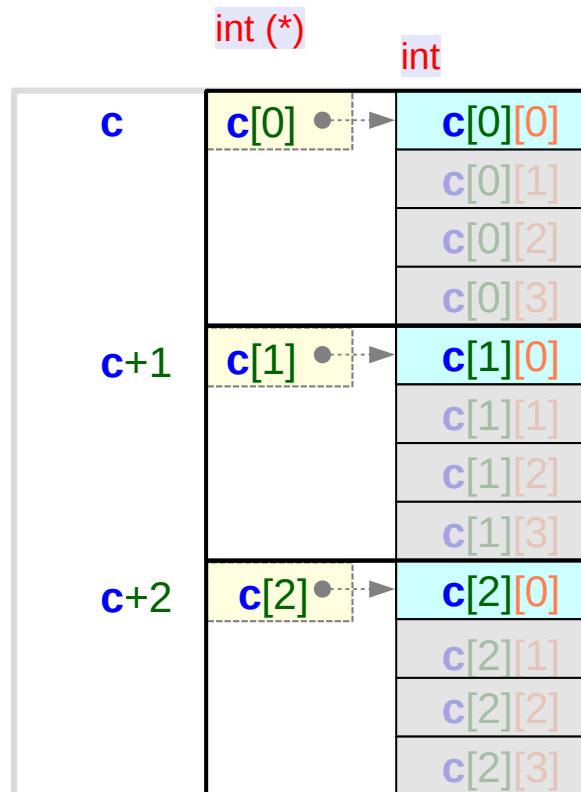
Aggregate data $c[i]$



an aggregate type data $c[i]$

- 1st element $c[i][0]$
- 2nd element $c[i][1]$
- 3rd element $c[i][2]$
- 4th element $c[i][3]$

Virtual pointer $c[i]$



a virtual pointer $c[i]$

- pointer address $\&c[i]$
- pointer value $c+i = \&c[i]$

with the constraint

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

an primitive data $c[i][0] = *c[i]$

- start address $\&c[i][0] = c[i]$
- size $\text{sizeof}(c[i][0])$

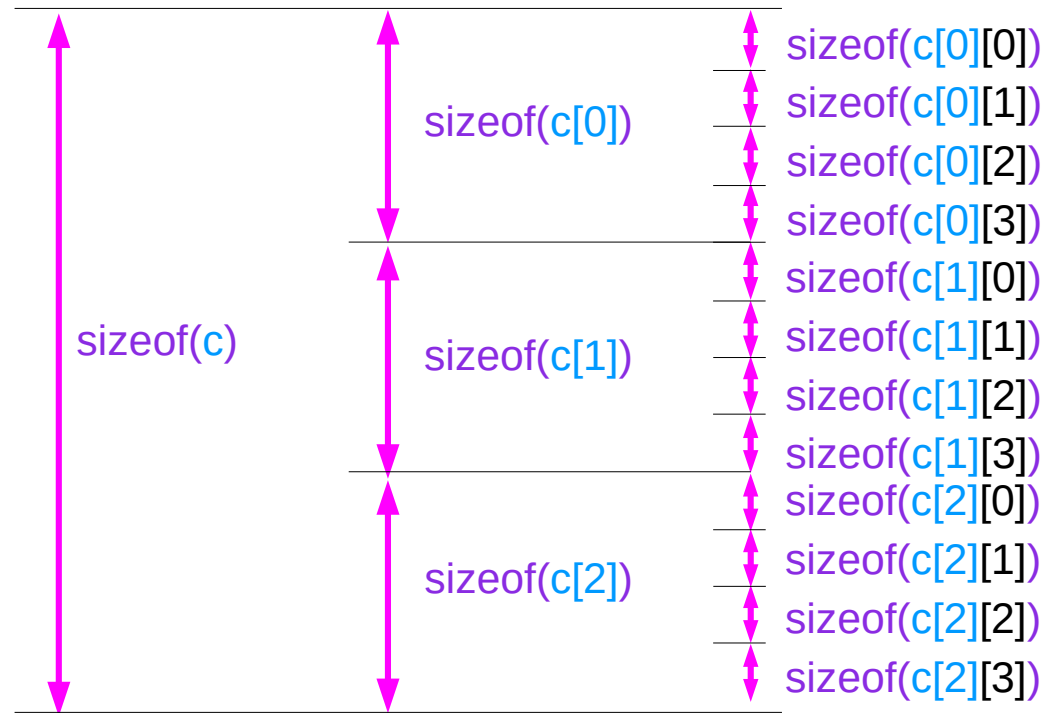
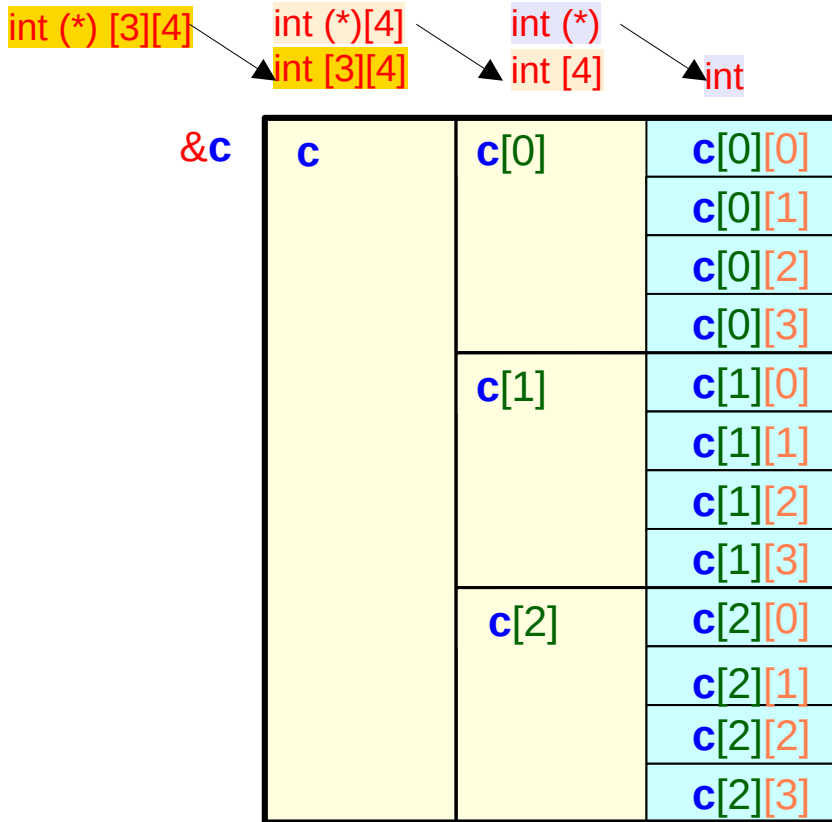
virtual pointer $c[i]$
points to primitive data $c[i][0]$

virtual pointers

- no physical memory locations are allocated
- address and data have the same value

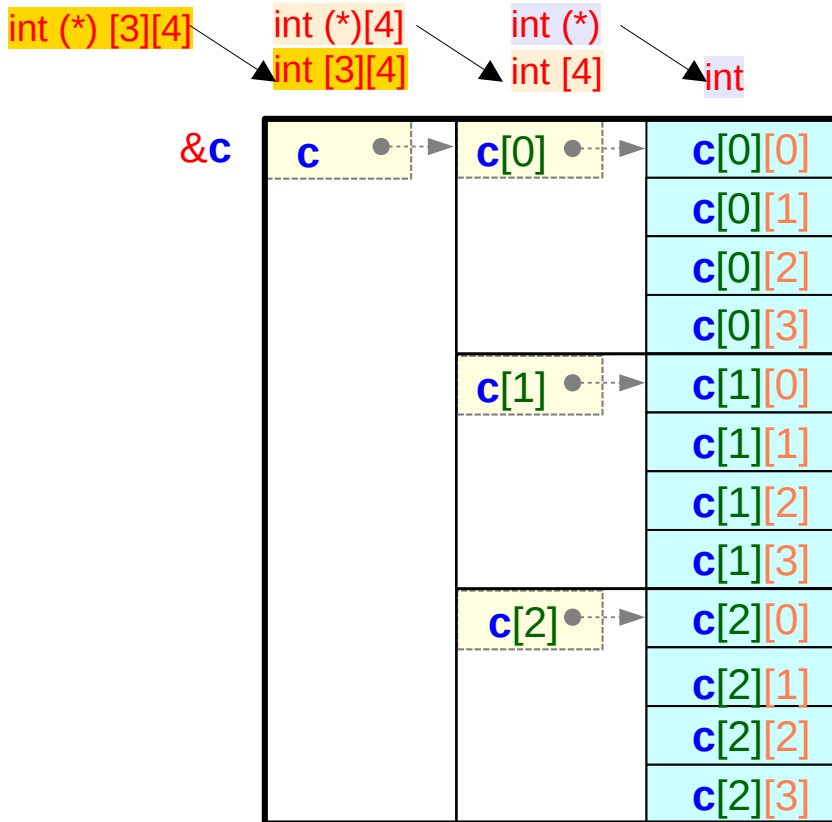
A 2-d array and its 1-d sub-arrays – a size view

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



A 2-d array and its 1-d sub-arrays – a virtual pointer view

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



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

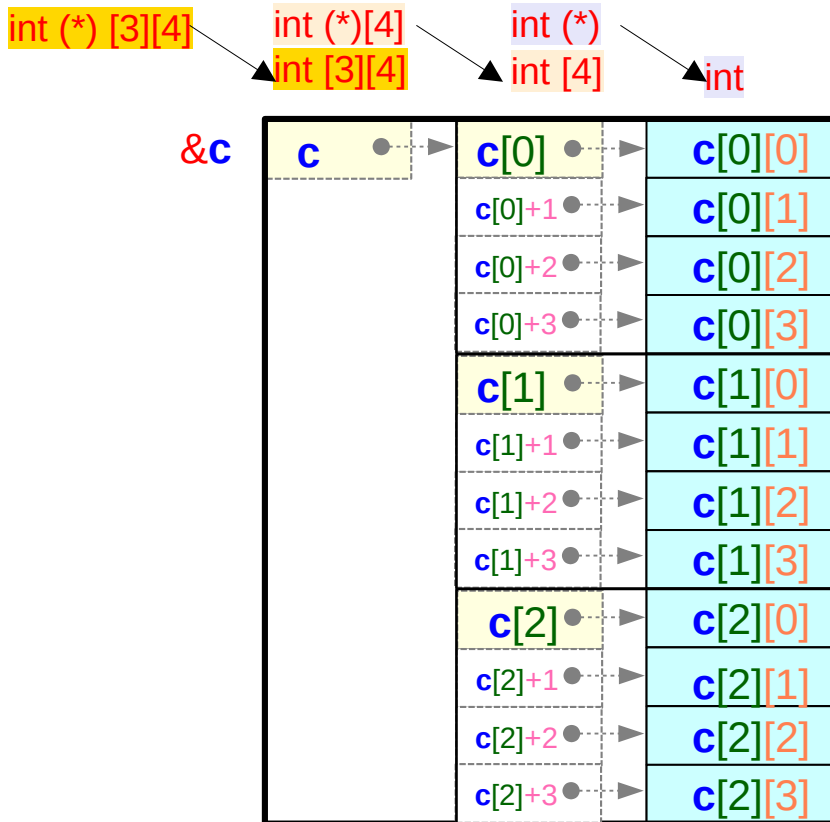
`value(c[1]) = value(&c[1][0])`
`value(&c[1]) = value(&c[1][0])`

`value(c[2]) = value(&c[2][0])`
`value(&c[2]) = value(&c[2][0])`

`address(c) = address(c[0]) = address(c[0][0])`
`address(c[1]) = address(c[1][0])`
`address(c[2]) = address(c[2][0])`

A 2-d array and its 1-d sub-arrays – size relation

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



`sizeof(c)` = `sizeof(c[0])` * 3 ... leading element
`sizeof(c+1)` = pointer size (4/8 bytes)
`sizeof(c+2)` = pointer size (4/8 bytes)

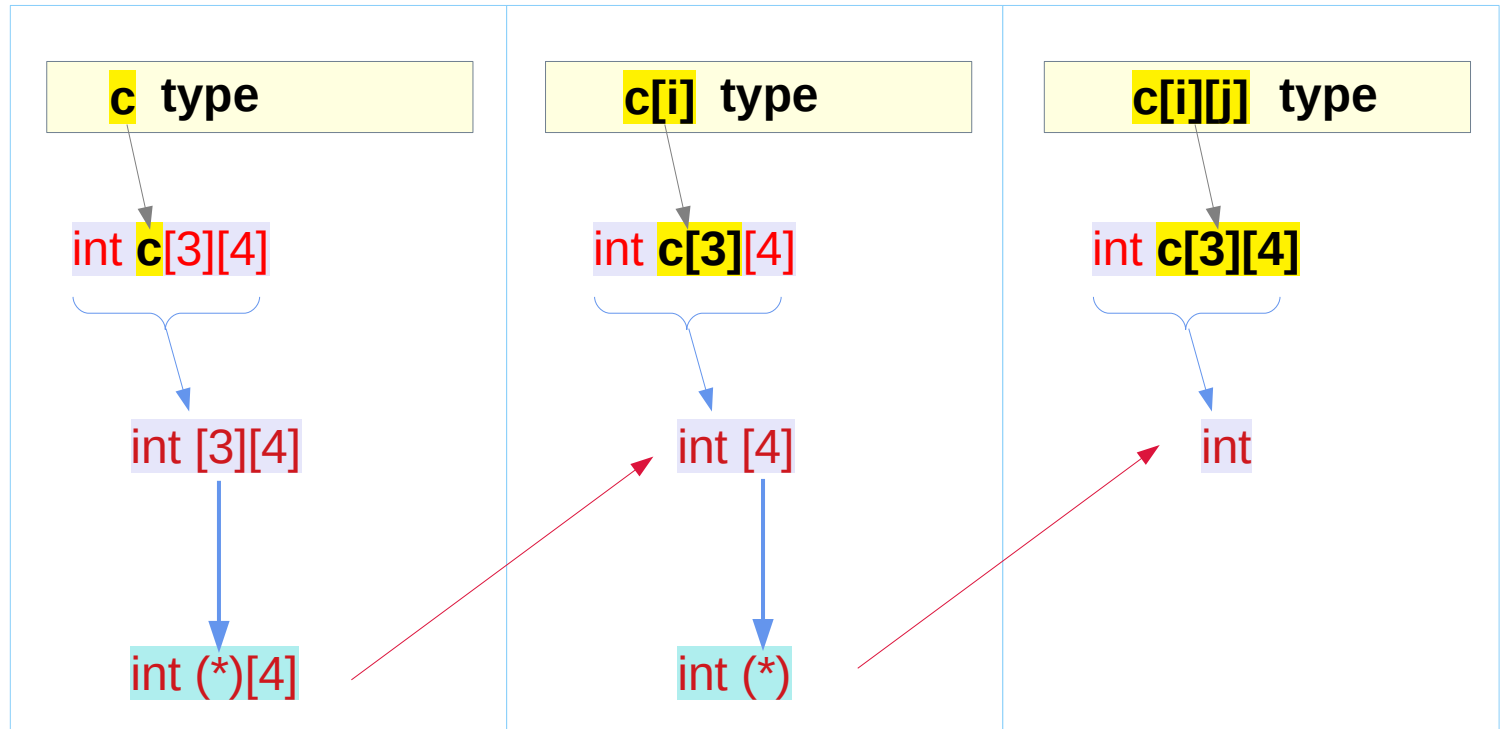
`sizeof(c[0])` = `sizeof(c[0][0])` * 4 ... leading element
`sizeof(c[0]+1)` = pointer size (4/8 bytes)
`sizeof(c[0]+2)` = pointer size (4/8 bytes)
`sizeof(c[0]+3)` = pointer size (4/8 bytes)

`sizeof(c[1])` = `sizeof(c[1][0])` * 4 ... leading element
`sizeof(c[1]+1)` = pointer size (4/8 bytes)
`sizeof(c[1]+2)` = pointer size (4/8 bytes)
`sizeof(c[1]+3)` = pointer size (4/8 bytes)

`sizeof(c[2])` = `sizeof(c[2][0])` * 4 ... leading element
`sizeof(c[2]+1)` = pointer size (4/8 bytes)
`sizeof(c[2]+2)` = pointer size (4/8 bytes)
`sizeof(c[2]+3)` = pointer size (4/8 bytes)

Sub-array types in a 2-d array

`int c[3][4];` 2-d array `c`



Array Type

relaxing the
1st dimension

Array Pointer Type

Dual Types

-
- **Identifying nested arrays
in a 2-d array declaration**

Nested arrays in a 2-d array declaration

int c[3] [4] ;

int c[3] [4] ;

c : a 3 element array
c[i] : each element

int c[3] [4] ;

c[i]'s type 1 : int [4]
an array of 4 integers

int c[3] [4] ;

relaxed dimension



c[i]'s type 2: int (*)
a pointer to an integer

Nested arrays

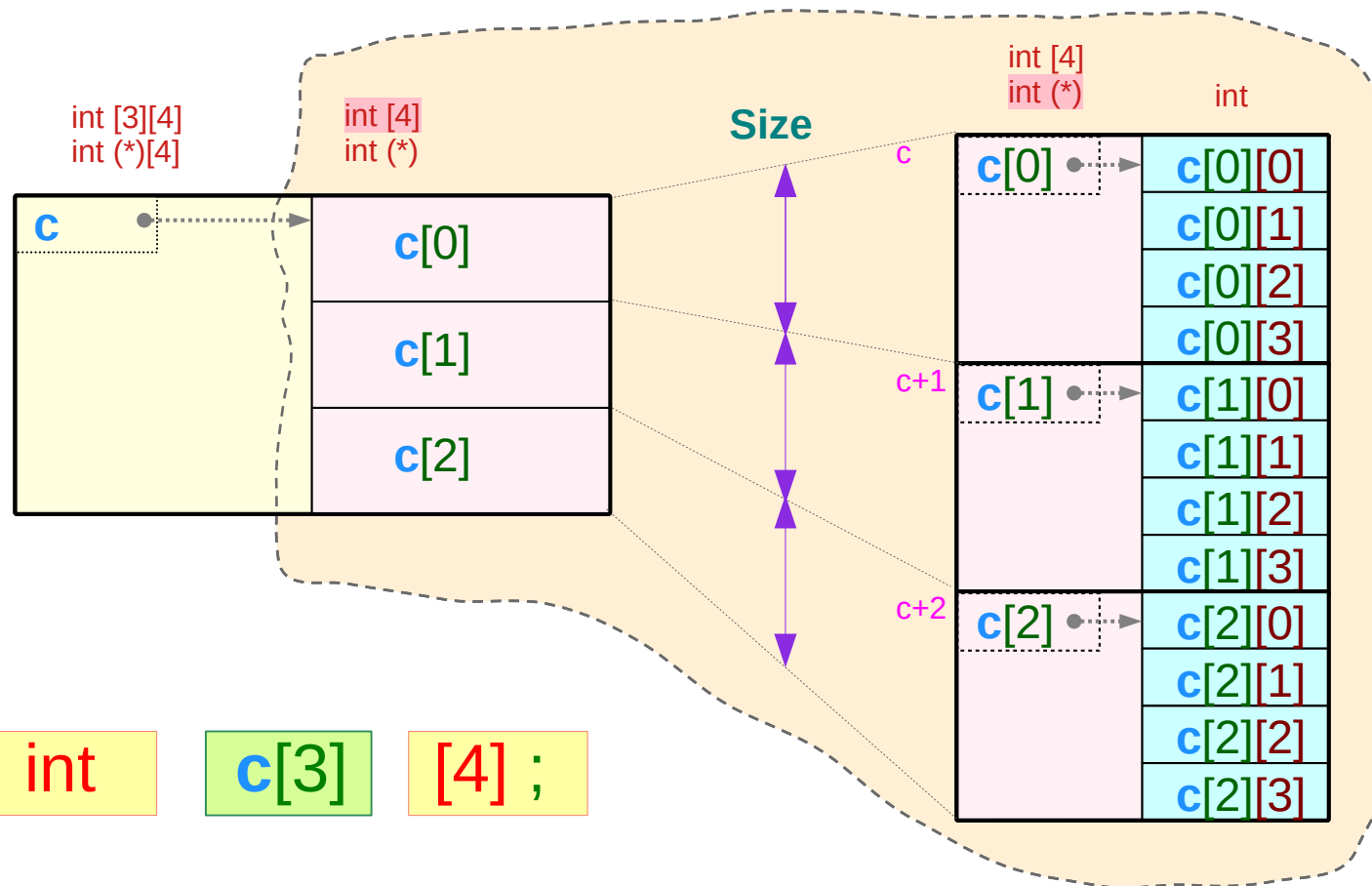
`c[3]`

`c` : a 3 element array
`c[i]` : each element

`int`

`[4];`

`c[i]`'s type 1 : `int [4]`
`c[i]`'s type 2 : `int (*)`



Address

`&c[0][0]` → `c[0]` → `c`

`&c[1][0]` → `c[1]`

`&c[2][0]` → `c[2]`

`int`

`c[3]`

`[4];`

c : 3-element array

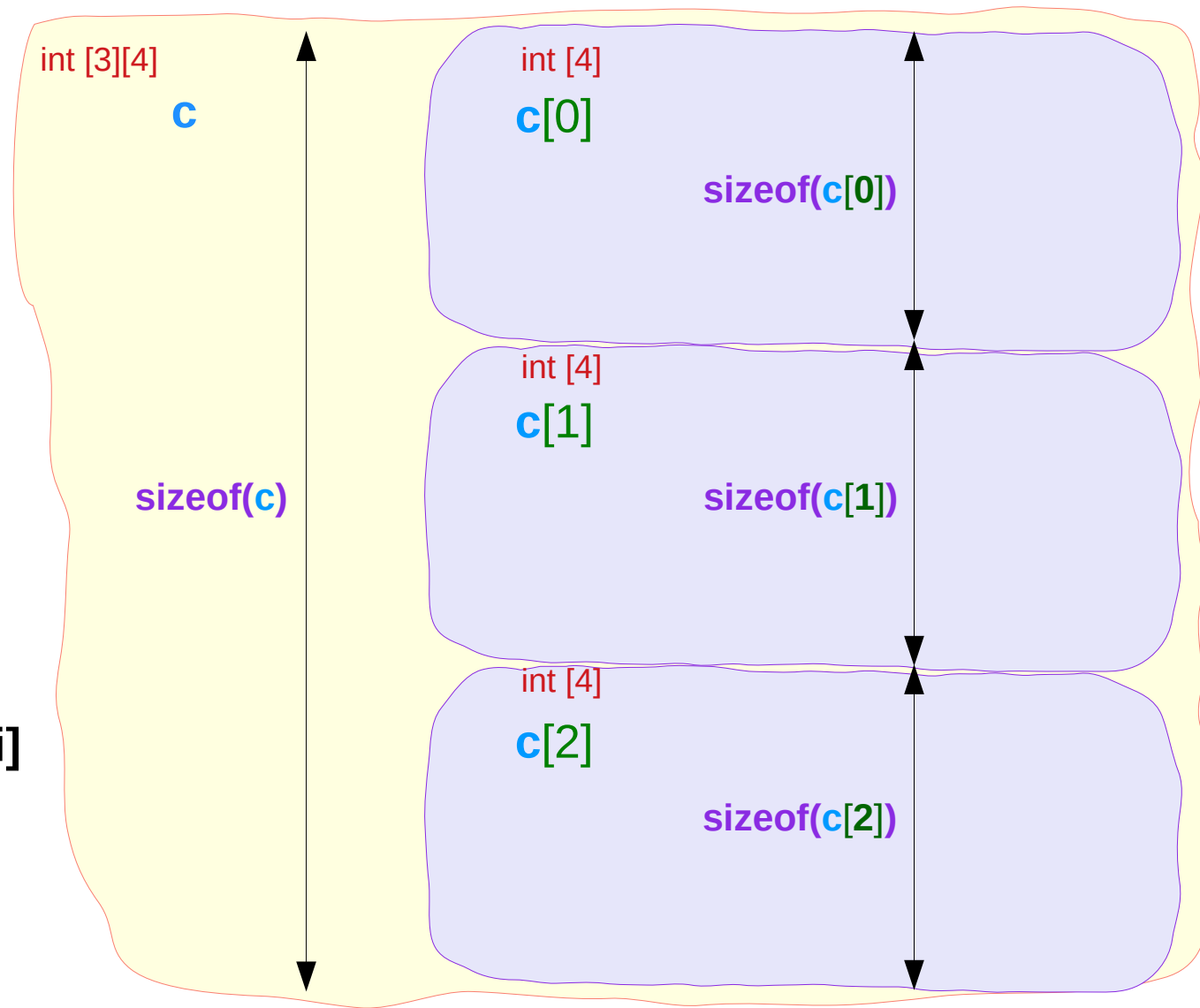
c	2-d array	int [3][4]
c[i]	1-d array	int [4]

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

3-element array c

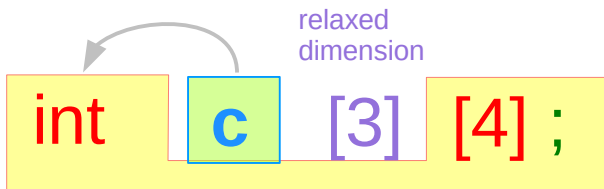
abstract data element **c[i]**

each element **c[i]** has the 1-d array type **int [4]**



c : pointer to a 4-element array

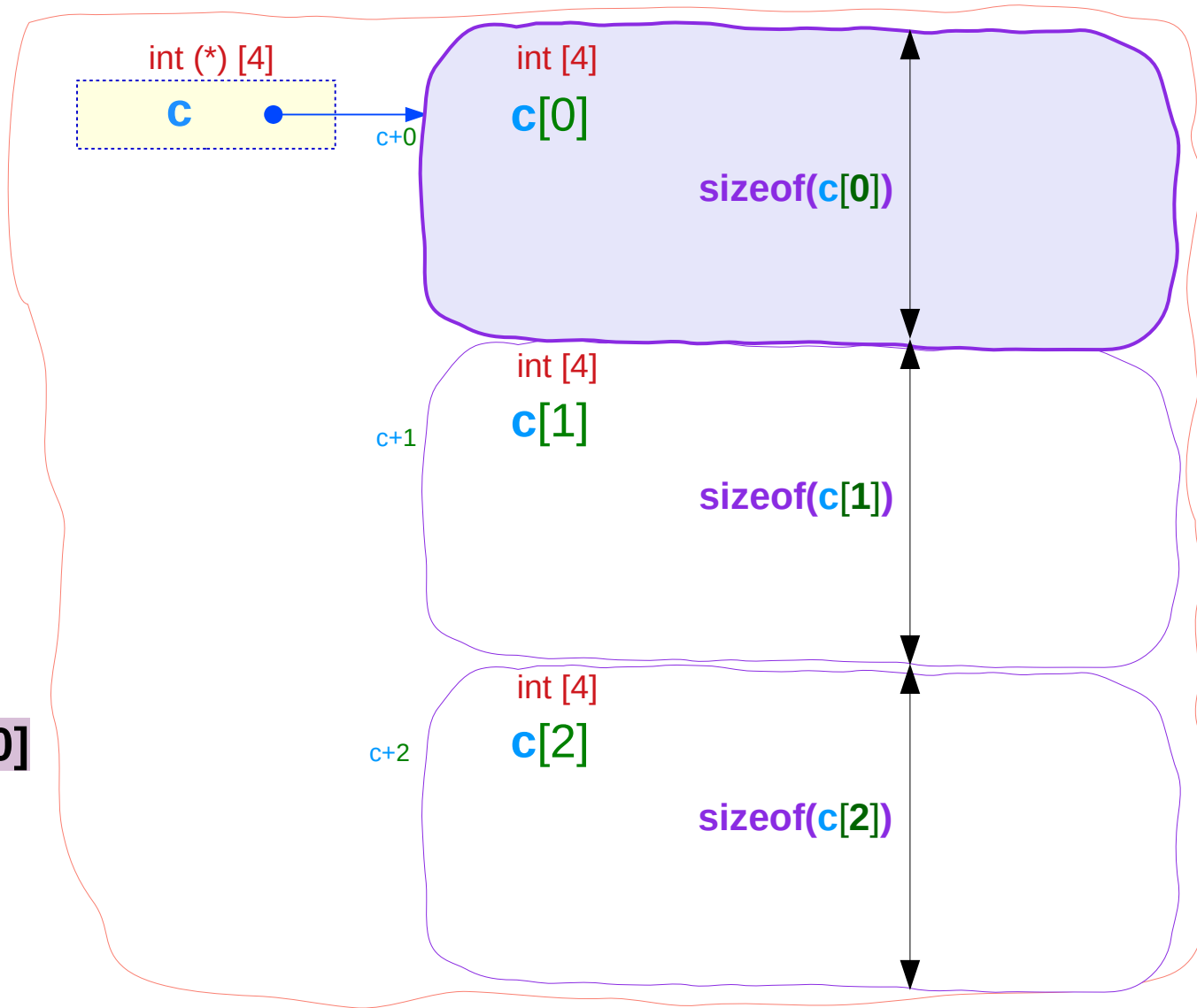
c	1-d array pointer	int (*)[4]
c[i]	1-d array	int [4]



pointer c

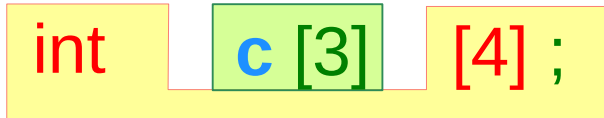
abstract data element **c[0]**

each element **c[i]** has the 1-d array type **int [4]**



c[i] : 4-element array

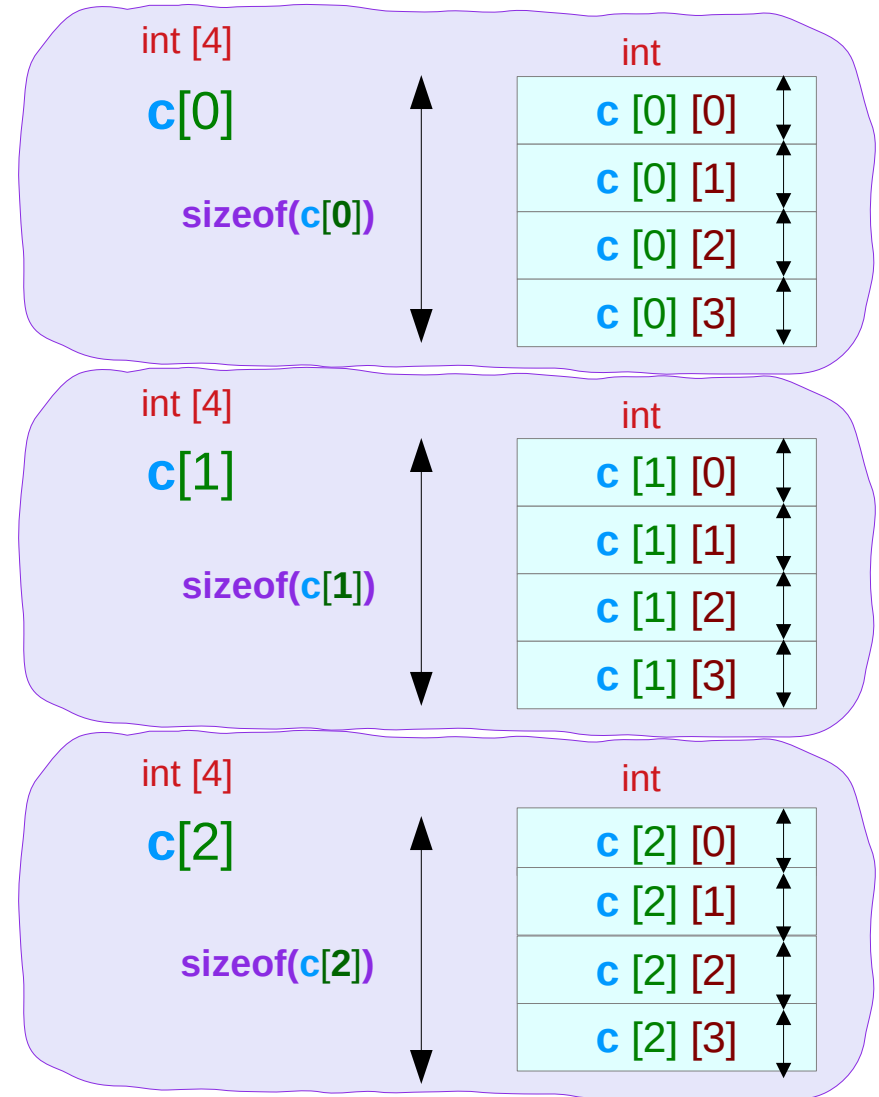
c[i]	1-d array	int [4]
c[i][j]	0-d array	int



4-element array c[i]

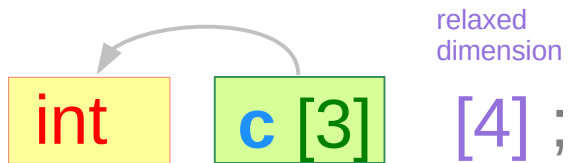
primitive data element `c[i][j]`

each element `c[i][j]` has the primitive type `int`



c[i] : pointer to a primitive data

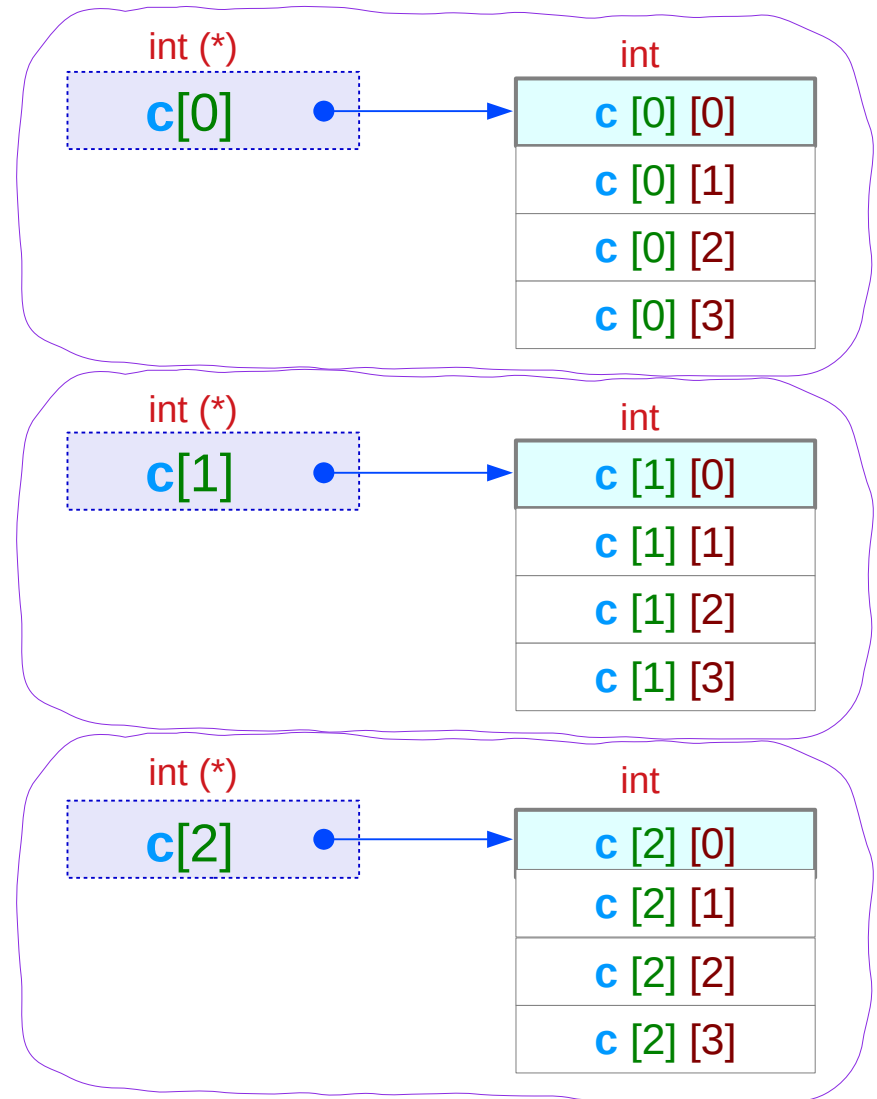
c[i]	0-d array pointer	int (*)
c[i][j]	0-d array	int



pointer c[i]

primitive data element **c[i][0]**

each element **c[i][j]** has
the primitive type **int**

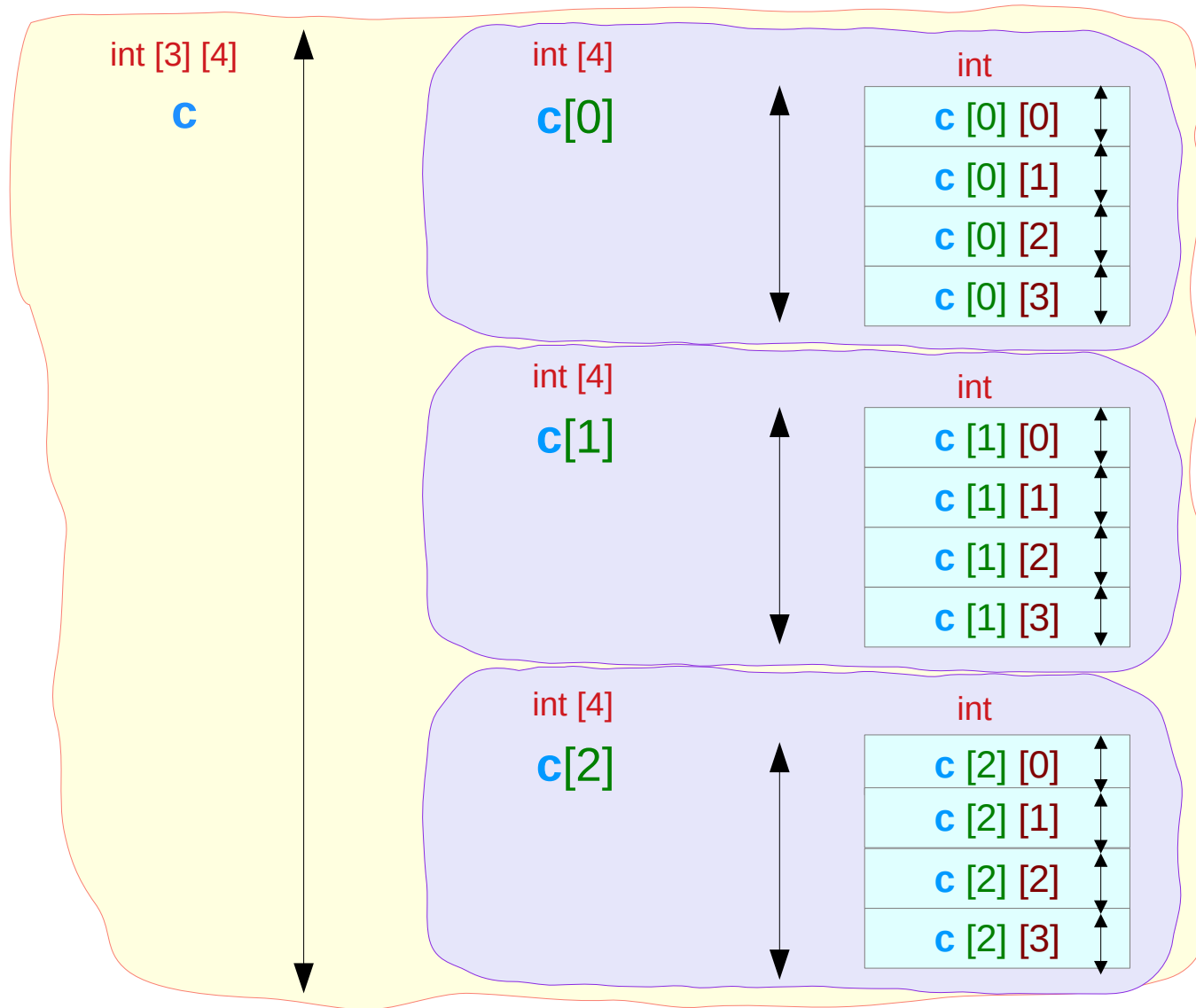


Recursive data view

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

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

3-element array `c`
4-element array `c[i]`



Pointer view

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

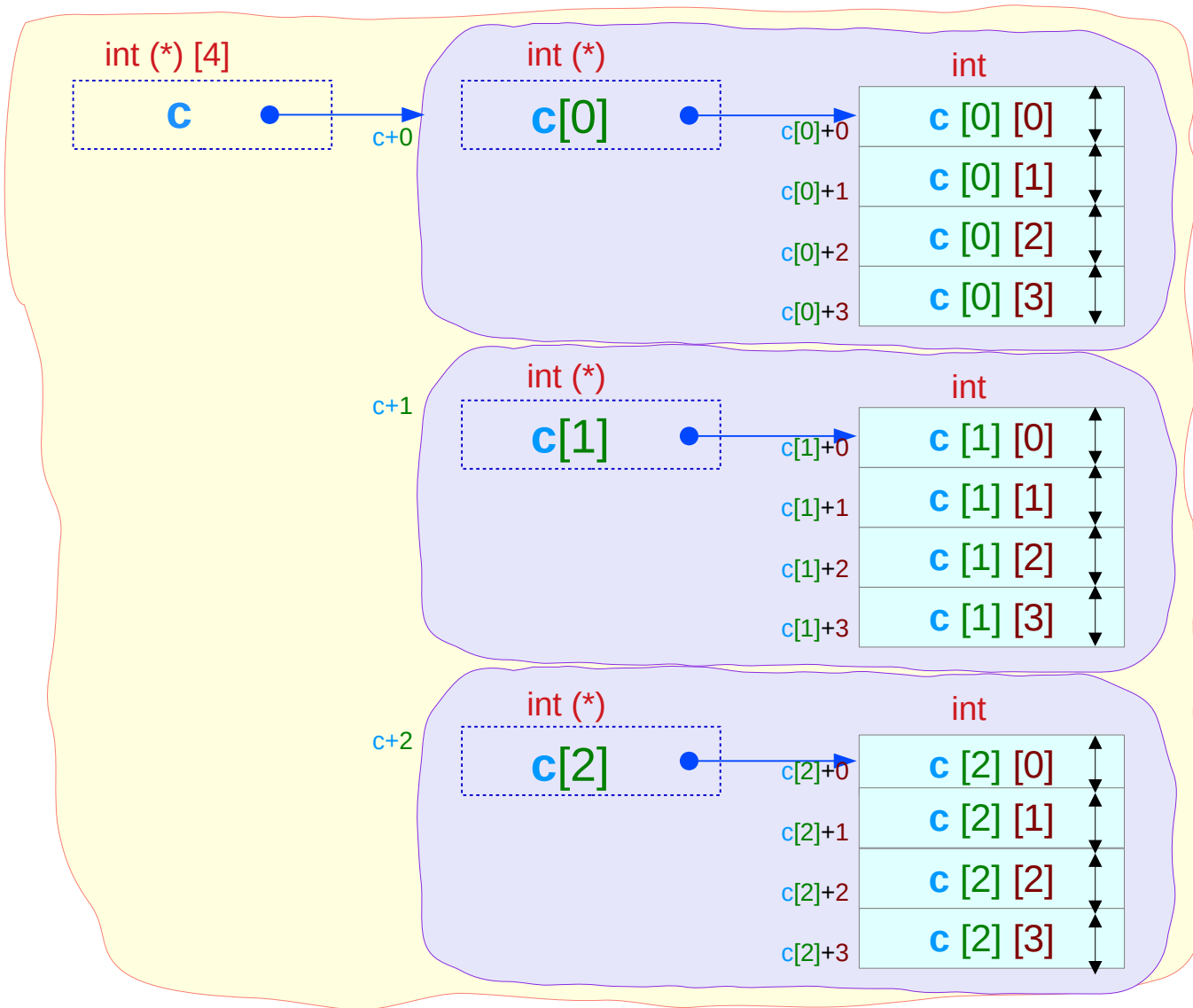
int **c[3]** **[4] ;**

$$v(c) = v(c[0]) = v(\&c[0][0])$$

$$v(c[1]) = v(\&c[1][0])$$

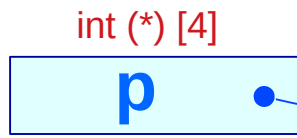
$$v(c[2]) = v(\&c[2][0])$$

$v \equiv$ value



1-d array pointer

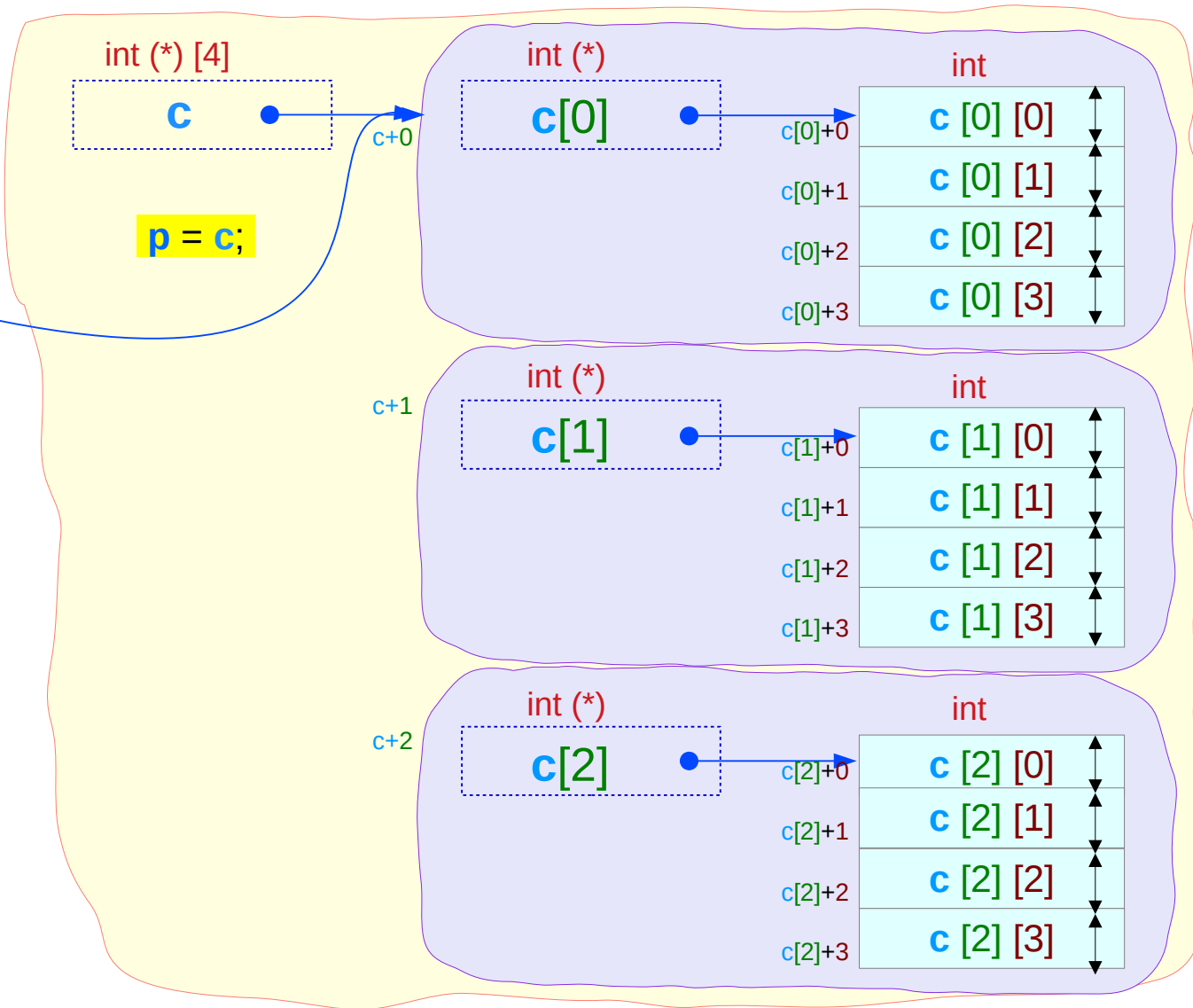
```
int (*p) [4];
```



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

$v(c) = v(c[0]) = v(\&c[0][0])$
 $v(c[1]) = v(\&c[1][0])$
 $v(c[2]) = v(\&c[2][0])$

$v \equiv \text{value}$

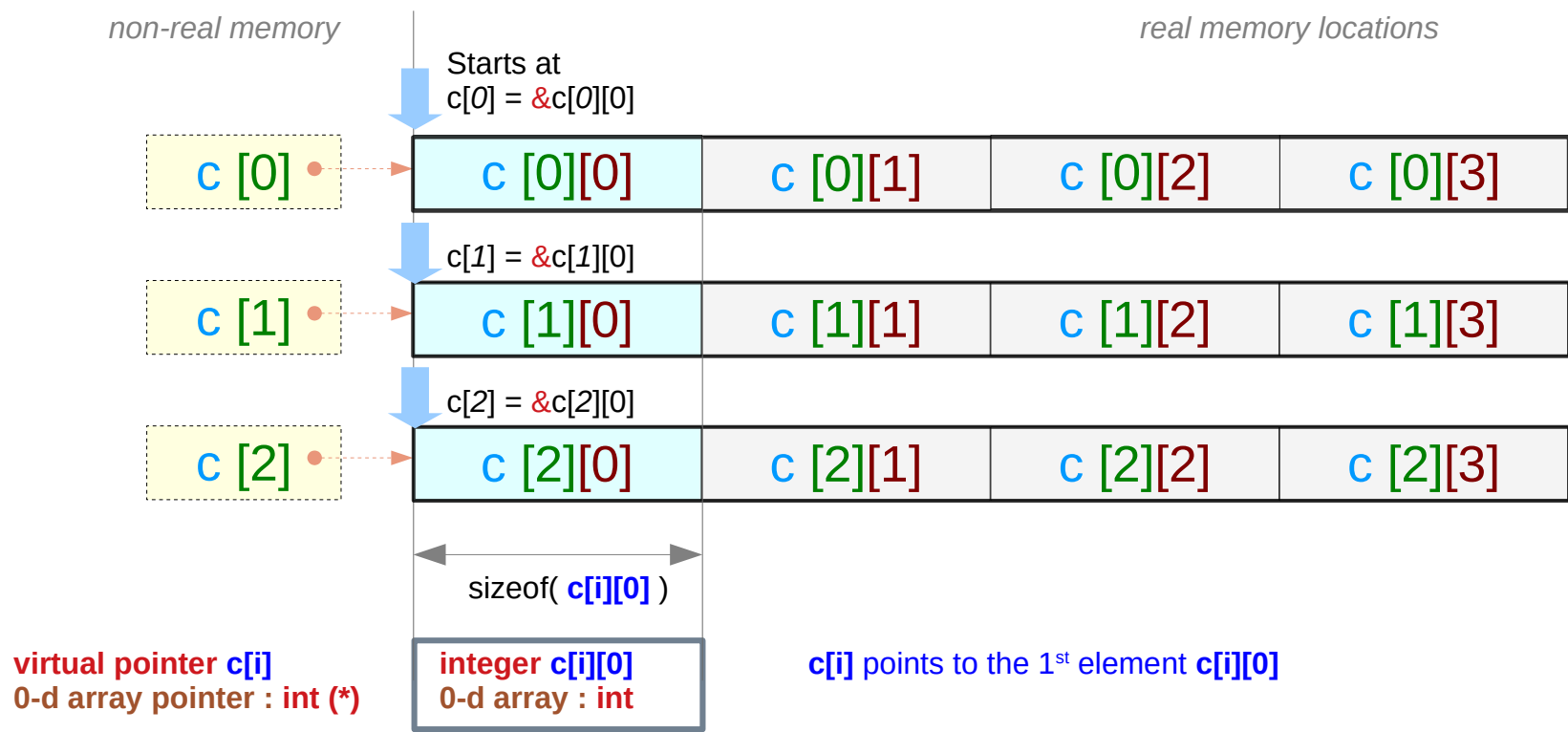


Pointer $c[i]$ and integer $c[i][0]$

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

non-real pointer $c[i]$: $\text{value}(c[i]) = \&c[i][0]$

0-d array pointer

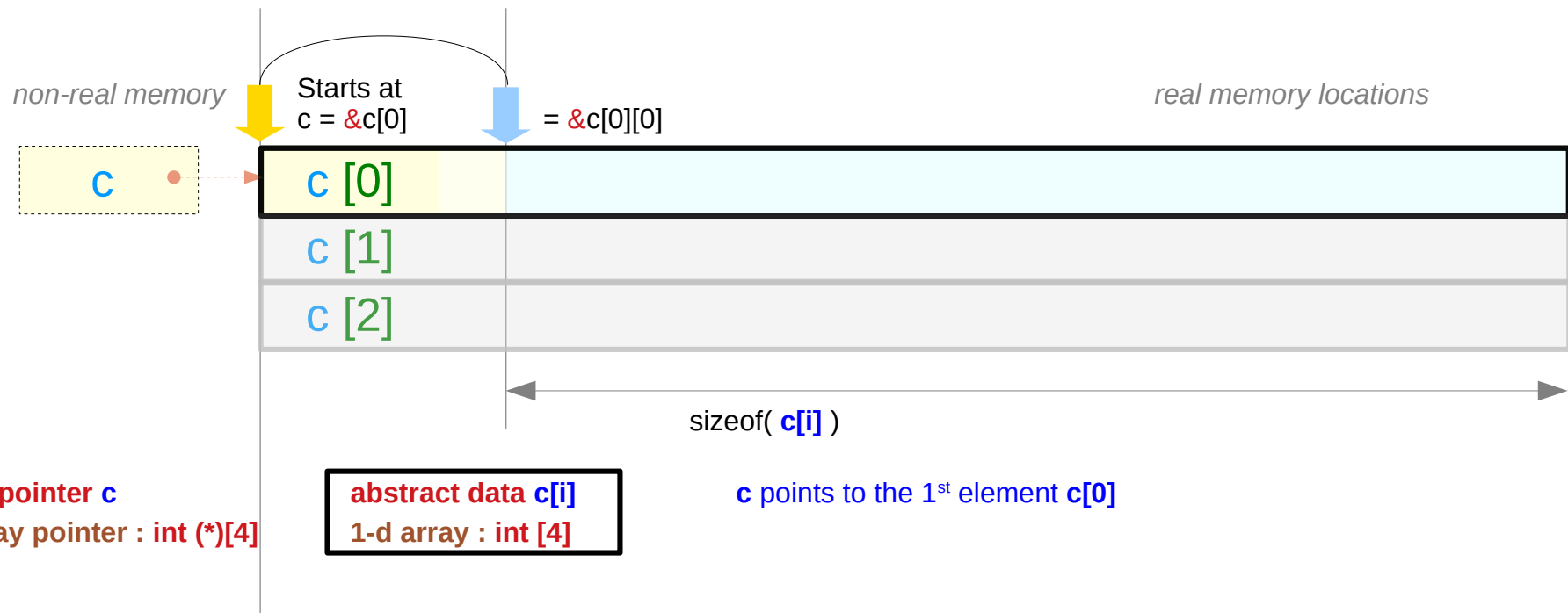


Pointer **c** and abstract data **c[i]**

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

non-real pointer **c** : $\text{value}(c) = \&c[0] = \&c[0][0]$
abstract data **c[i]** : $\text{sizeof}(c[i]) = 4 * \text{sizeof}(\text{int})$

1-d array pointer
1-d array

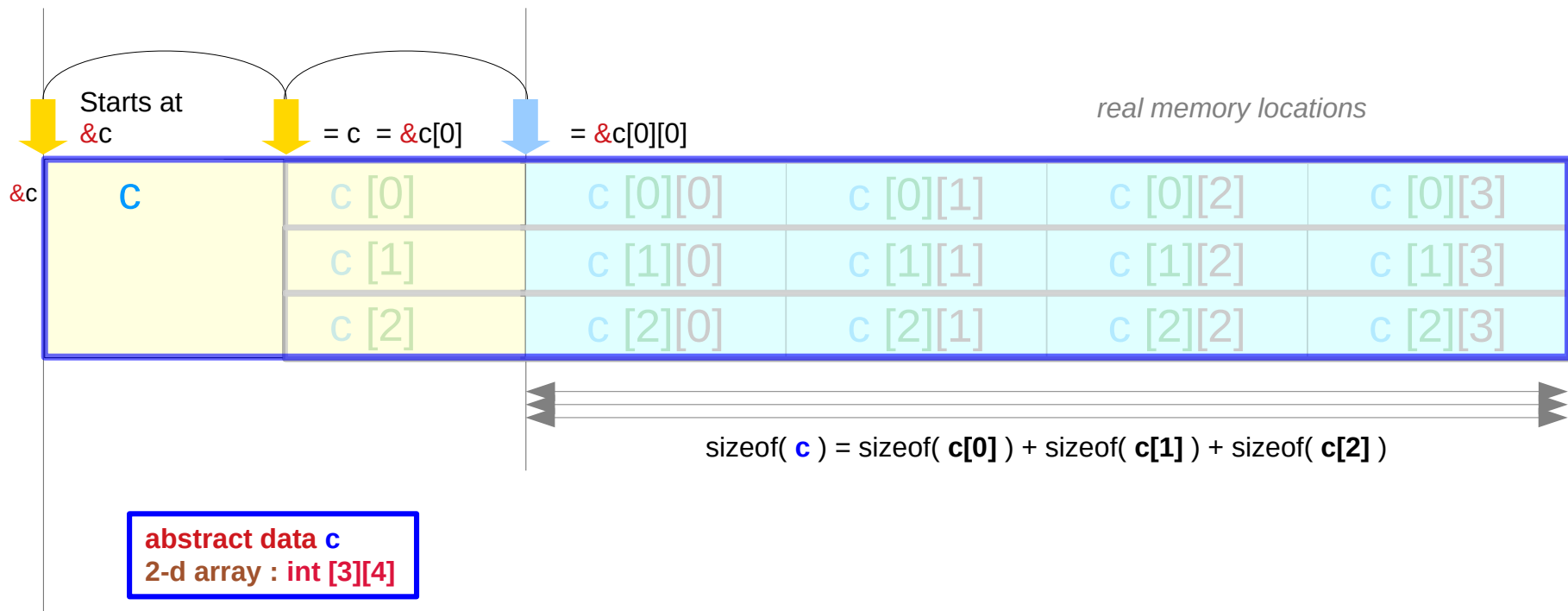


Abstract data **c**

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

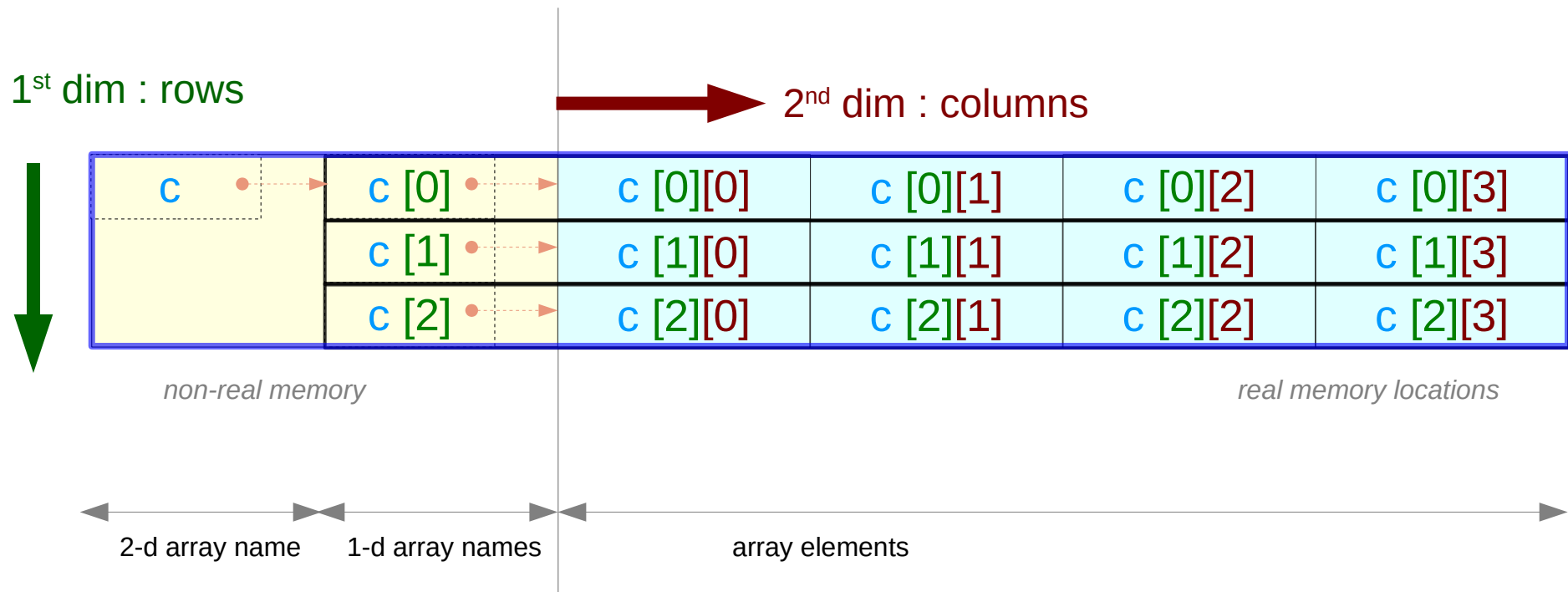
abstract data **c**: $\text{sizeof}(\mathbf{c}) = 3 * \text{sizeof}(\mathbf{c}[\mathbf{i}])$

2-d array



Rows and columns of a 2-d array **c**

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



Determining types of sub-arrays

from the declaration of an array

Types of array names

```
int a[4];
```

a is the name of the 1-d array

int [4]

`sizeof(a)` = 4 * 4

[3] is declared;
[0], [1], [2] are used

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

c[i] is the name of the 1-d subarray

int [4]

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

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

c is the name of the 2-d array

int [3][4]

`sizeof(c)` = 3 * 4 * 4

Values of array names



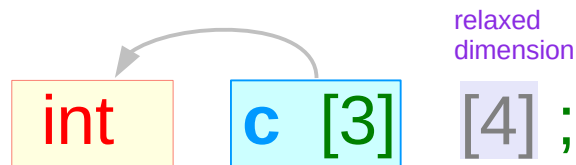
the value of **a** is the starting address of an array with 4 elements of **int** type

int (*)

a: pointer to the first element

a = &a[0]

[3] is declared;
[0], [1], [2] are used

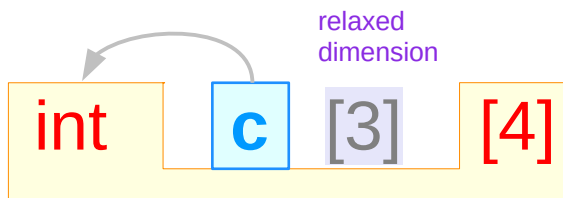


each value of **c[i]** is the starting address of an array with 4 elements of **int** type

int (*)

c[i]: pointer to the first element

c[i] = &c[i][0]



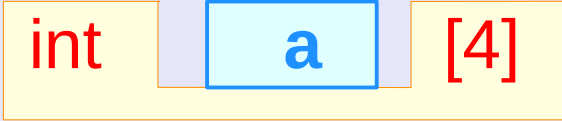
the value of **c** is the starting address of an array with 3 elements of **int [4]** type

int (*) [4]

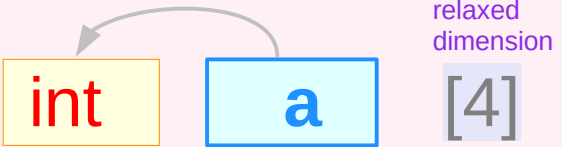
c: pointer to the first element

c = &c[0]

Array and pointer types in a 1-d array

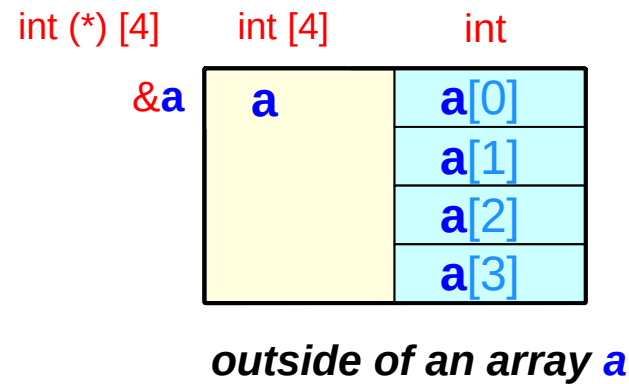


a 1-d array
type : `int [4]`
size : `4 * 4`



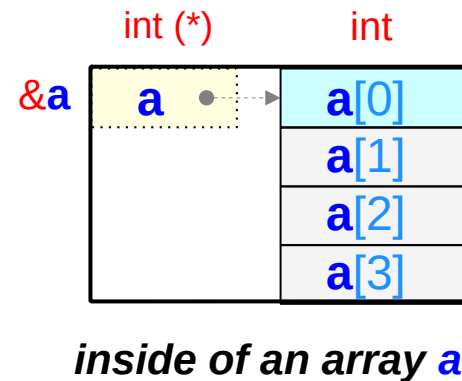
a 0-d array pointer
type : `int (*)`
value : `&a[0]`

a points to the 1st `int` element
there are 4 `int` elements



`int (*) [4]` `int [4]` `int`
&a a a[0]
a[1]
a[2]
a[3]

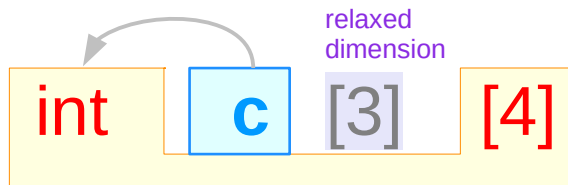
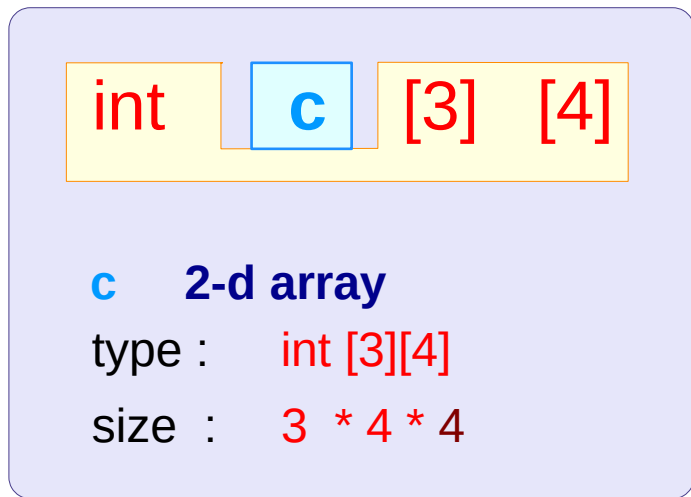
outside of an array a



`int (*)` `int`
&a a a[0]
a[1]
a[2]
a[3]

inside of an array a

2-d array type

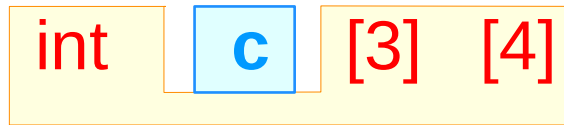


int (*) [3][4]
 &c

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

outside of an array c
 (c as an abstract data)

1-d array pointer type



c 2-d array

type : `int [3][4]`

size : `3 * 4 * 4`

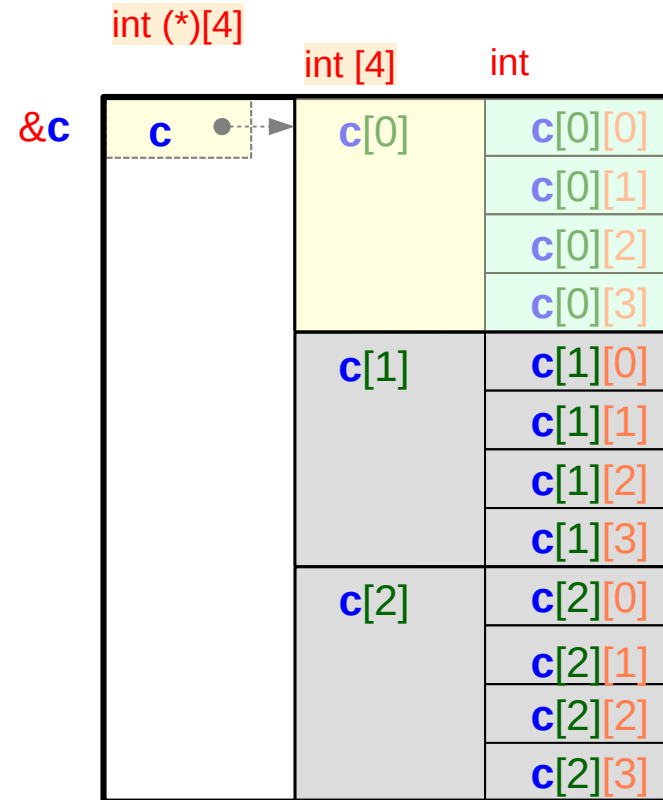
A diagram showing the declaration of a 1-d array pointer. It consists of three boxes: a yellow box containing the text 'int', a blue box containing the text 'c', and a yellow box containing the text '[3] [4]'. An arrow points from the 'c' box to the first '[3]' box. A label 'relaxed dimension' is placed above the arrow.

c 1-d array pointer

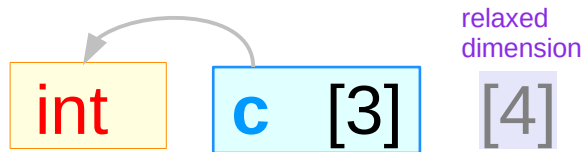
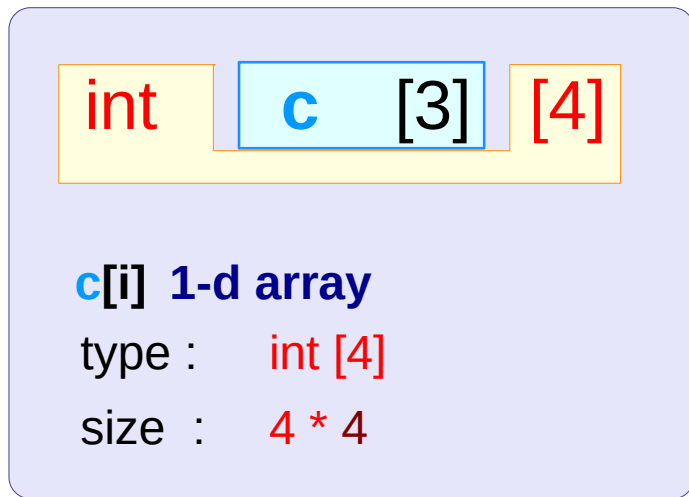
type : `int (*) [4]`

value : `c = &c[0][0]`

c points to the 1st `int [4]` element
There are 3 `int [4]` elements



1-d array type

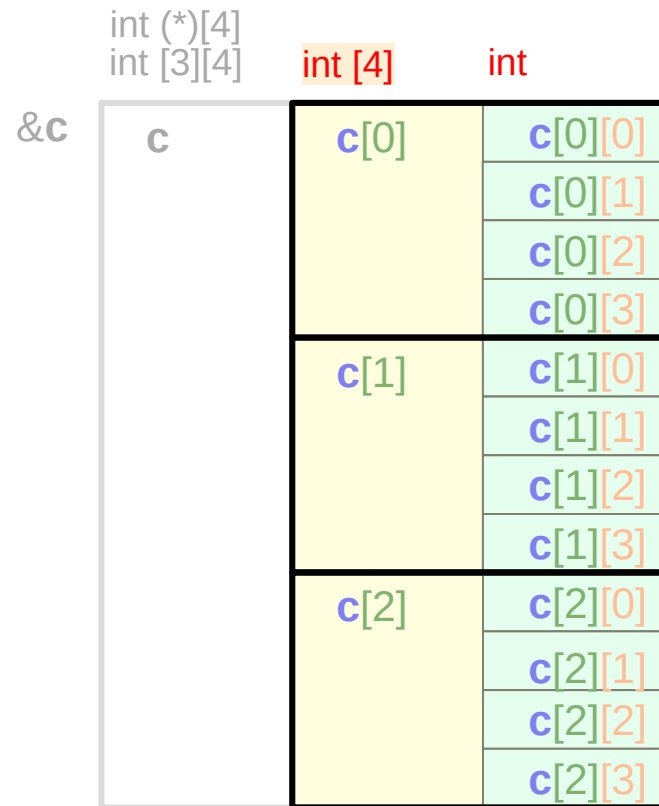


c[i] 0-d array pointer

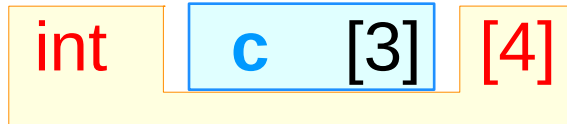
type : **int (*)**

value : **c[i] = &c[i][0]**

c[i] points to the 1st **int** element
There are 4 **int** elements



0-d array pointer type



c[i] 1-d array

type : int [4]

size : 4 * 4

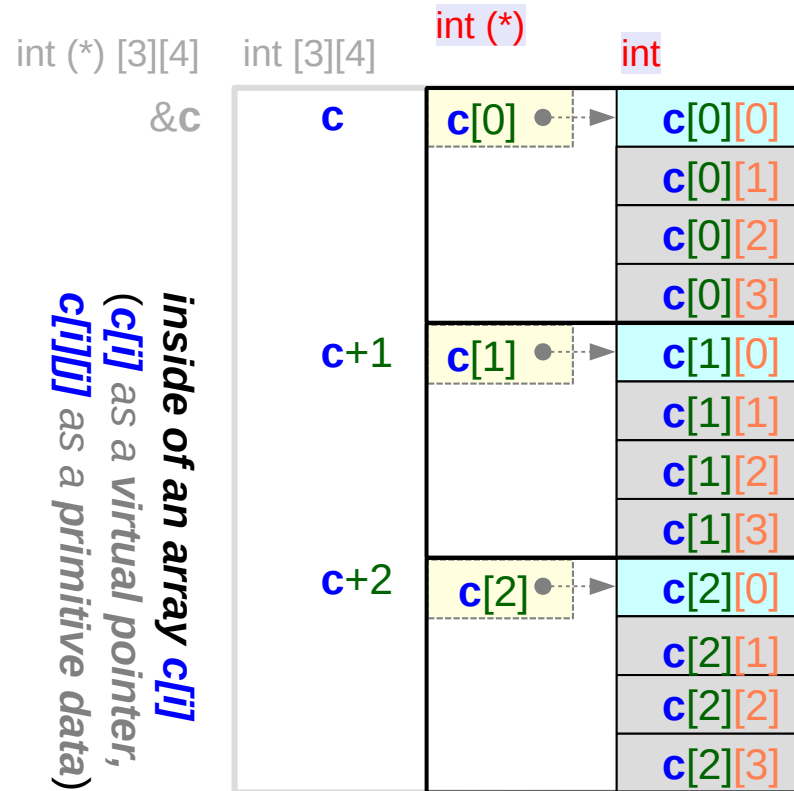
A diagram showing a yellow box labeled 'int' pointing to a light blue box labeled 'c' with '[3]' and '[4]' next to it. The '[4]' is labeled 'relaxed dimension'.

c[i] 0-d array pointer

type : int (*)

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

c[i] points to the 1st int element
There are 4 int elements



Types in a 2-d array

int c [3] [4]

c 2-d array

type : int [3][4]

size : 3 * 4 * 4

relaxing the 1st dimension

int c [3] [4]

c 1-d array pointer (virtual)

type : int (*) [4]

value : &c[0][0]

int c [3] [4]

c[i] 1-d array

type : int [4]

size : 4 * 4

relaxing the 1st dimension

int c [3] [4]

c[i] 0-d array pointer (virtual)

type : int (*)

value : &c[i][0]

The name of a 2-d array

```
int    a [4];
```

```
int    c [4] [4];
```

1. the name of the nested array (recursive definition)
2. a double pointer
3. a pointer to an array

2-d array c and 1-d array q

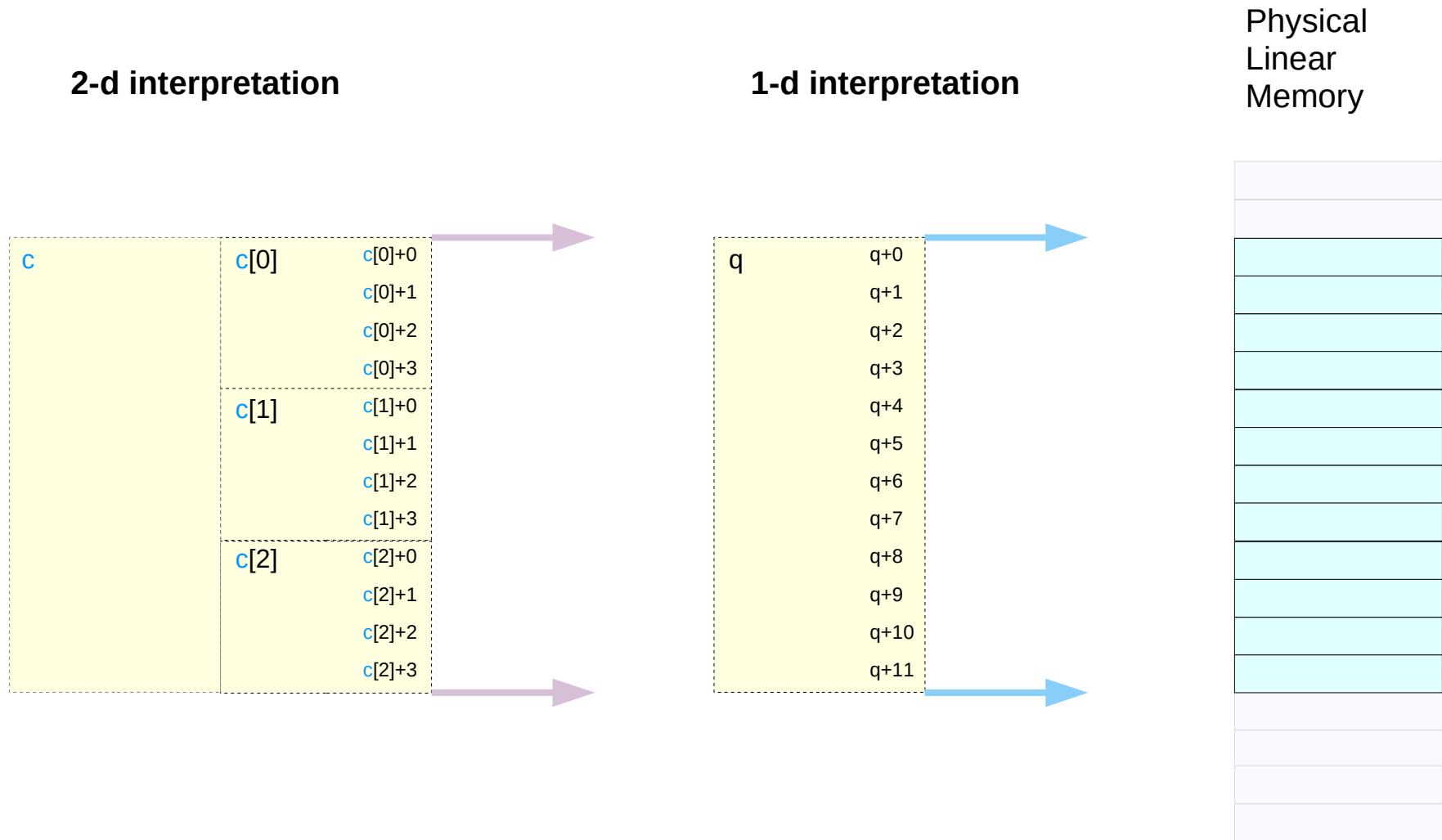
```
int c [3] [4];
```

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

```
int q [3*4];
```

q	q+0	q[0*4+0]
	q+1	q[0*4+1]
	q+2	q[0*4+2]
	q+3	q[0*4+3]
	q+4	q[1*4+0]
	q+5	q[1*4+1]
	q+6	q[1*4+2]
	q+7	q[1*4+3]
	q+8	q[2*4+0]
	q+9	q[2*4+1]
	q+10	q[2*4+2]
	q+11	q[2*4+3]

2-d and 1-d interpretations of linear memories



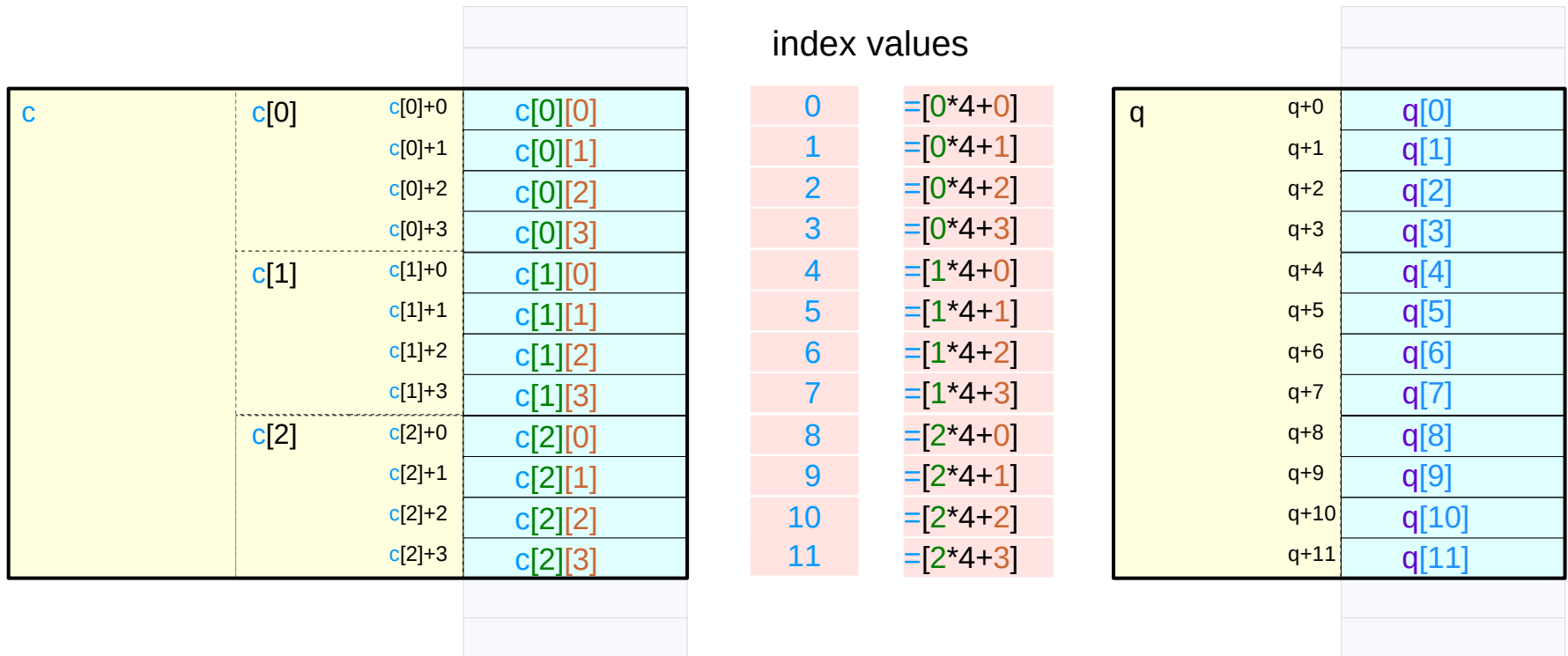
A 2-d array stored as a 1-d array (row major order)

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

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

```
[i*4+j]
```

```
[k]
```



2-d array access via a single pointer

```
int *p = c[0];
```



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

```
p[ i*4 + j ]
```



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

```
*(p + i*4 + j)
```



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

```
*(p + k)    i = k / 4;  
            j = k % 4;
```

View a 2-d array as a 1-d array

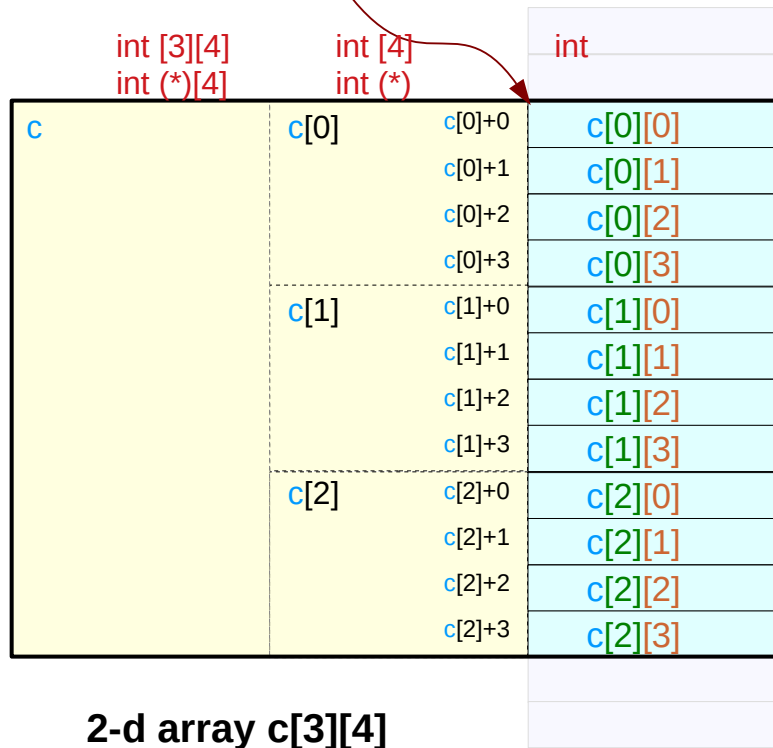
```
int c [3][4];
```

```
int *p = c[0];
```

c, c[0],
&c[0][0]

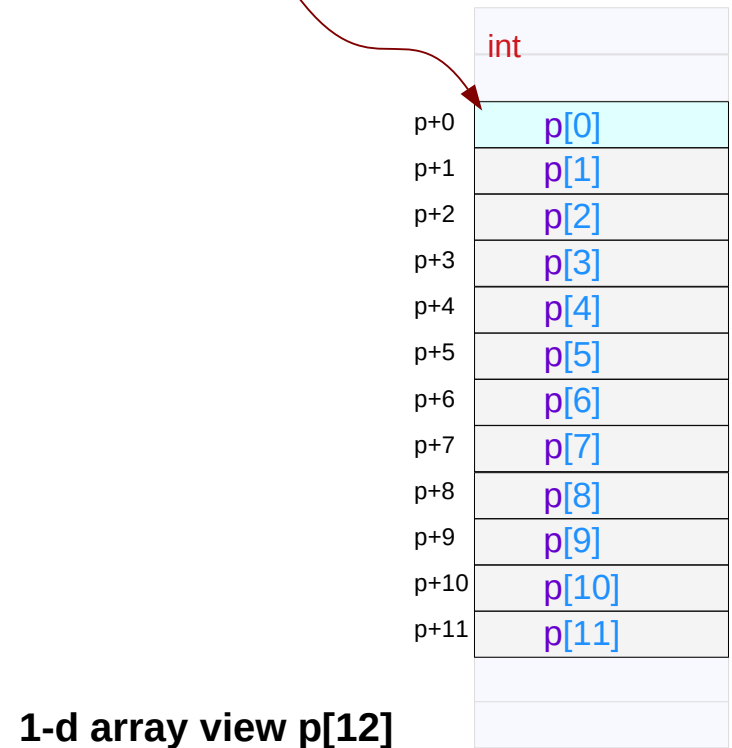
0-d array pointer int (*)

p



0-d array pointer int (*)

p



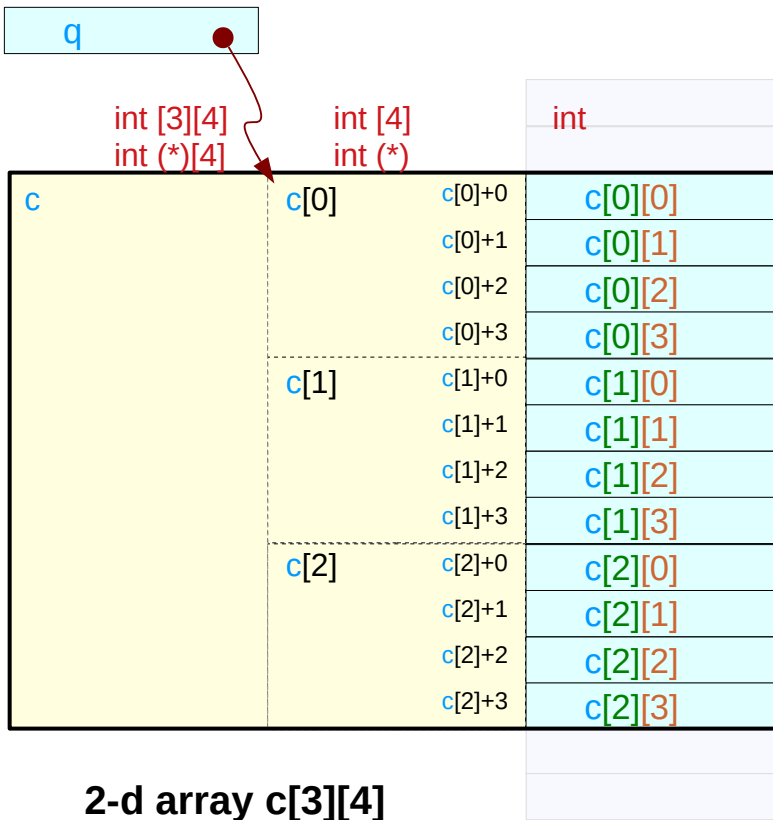
View a 2-d array as another 2-d array

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

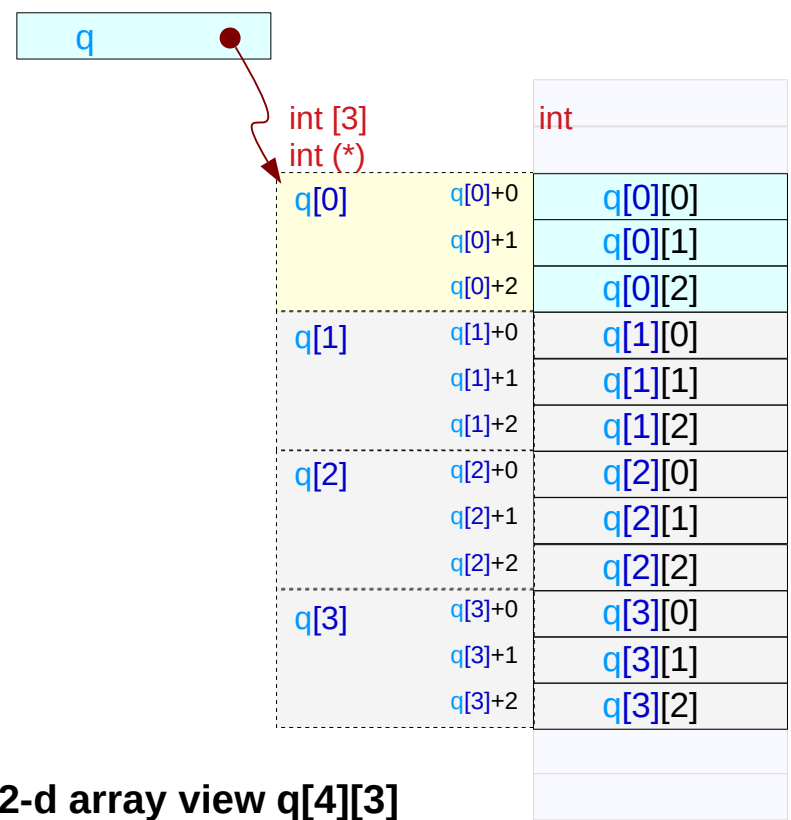
```
int (*q) [3] = (int (*) [3]) c;
```

c, c[0],
&c[0][0]

1-d array pointer int (*) [3]



1-d array pointer int (*) [3]



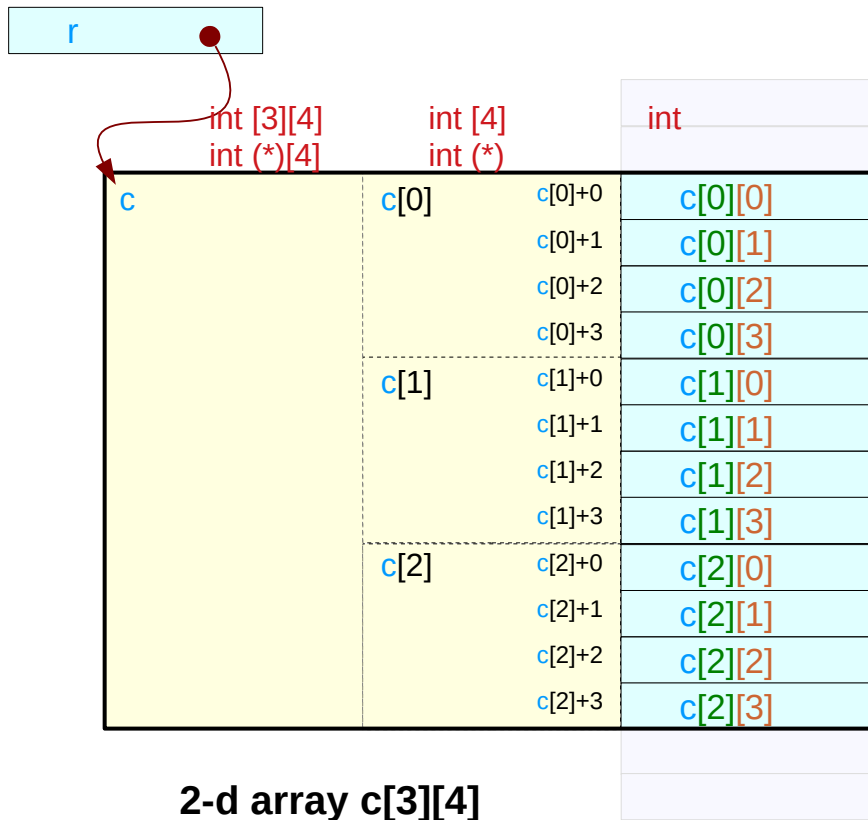
A 2-d array stored as a 1-d array (row major order)

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

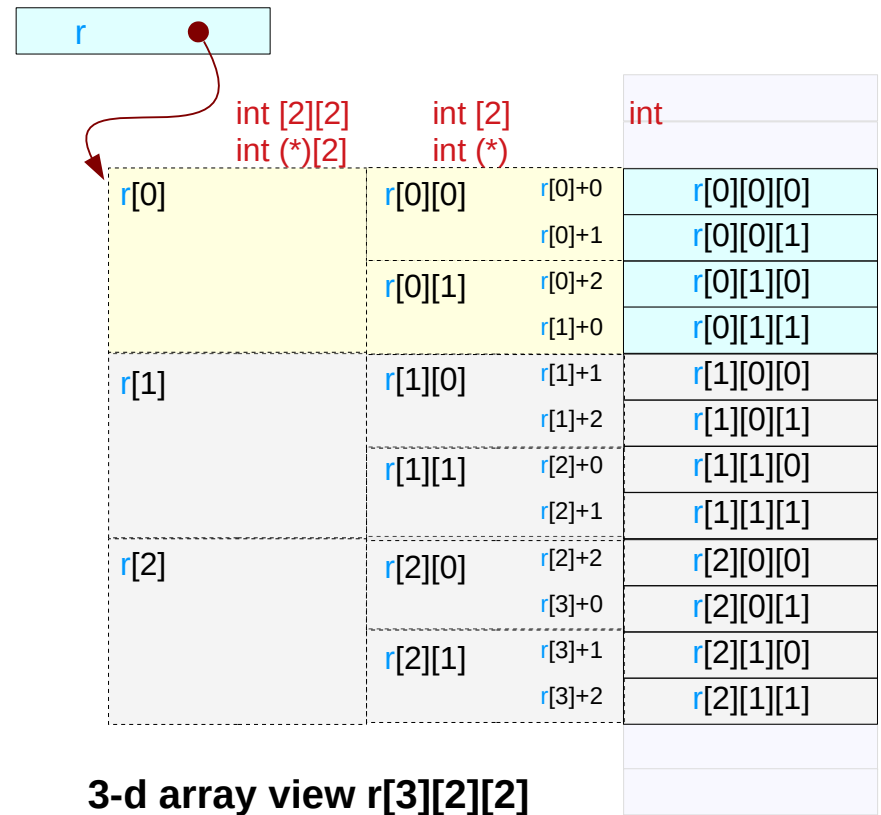
```
int (*r) [2][2] = (int (*) [2][2]) c;
```

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

2-d array pointer `int (*) [2][2]`



2-d array pointer `int (*) [2][2]`



2-d array access via pointers

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

1. recursive pointers

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

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

→ int (*p)[4];

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

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

→ int **q;

```
int *p = c[0] ;
```

2. linear array pointers

```
p[ i*4 + j ]
```

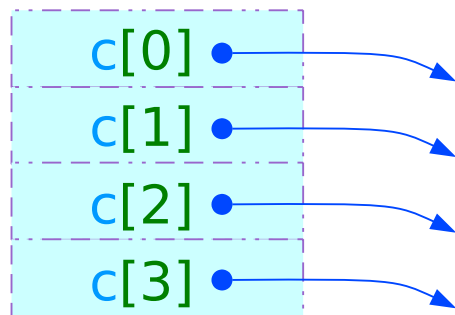
```
*(p+ i*4 + j)
```

Static Allocation of a 2-d Array

```
int A [3][4];
```

A in %eax,
i in %edx,
j in %ecx

```
sall    $2, %ecx           ;; j * 4  
leal   (%edx, %edx, 2), %edx  ;; i * 3  
leal   (%ecx, %edx, 4), %edx  ;; j * 4 + i * 12  
movl   (%eax, %edx), %eax    ;; read M[ XA+4(3i +j) ]
```



The pointer array :
not allocated
in the memory

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

Pointers, arrays, and operator precedence

Operator Precedence of * and []

$$*x[m] \equiv *(x[m])$$

$$x[m][n] \equiv (x[m])[n]$$

$$**x \equiv *(*x)$$

[] has a **higher** priority than *

[] has **left-to-right** associativity

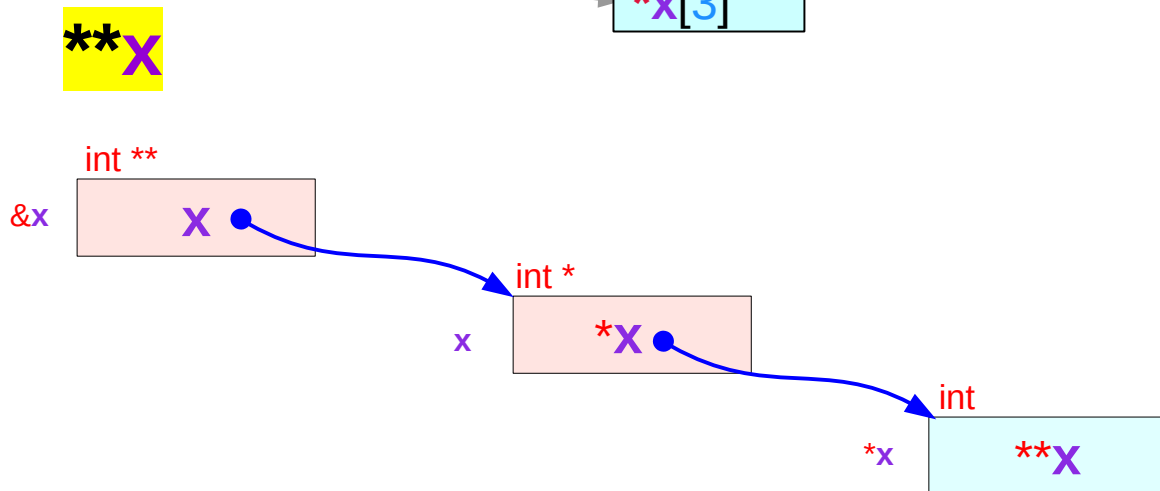
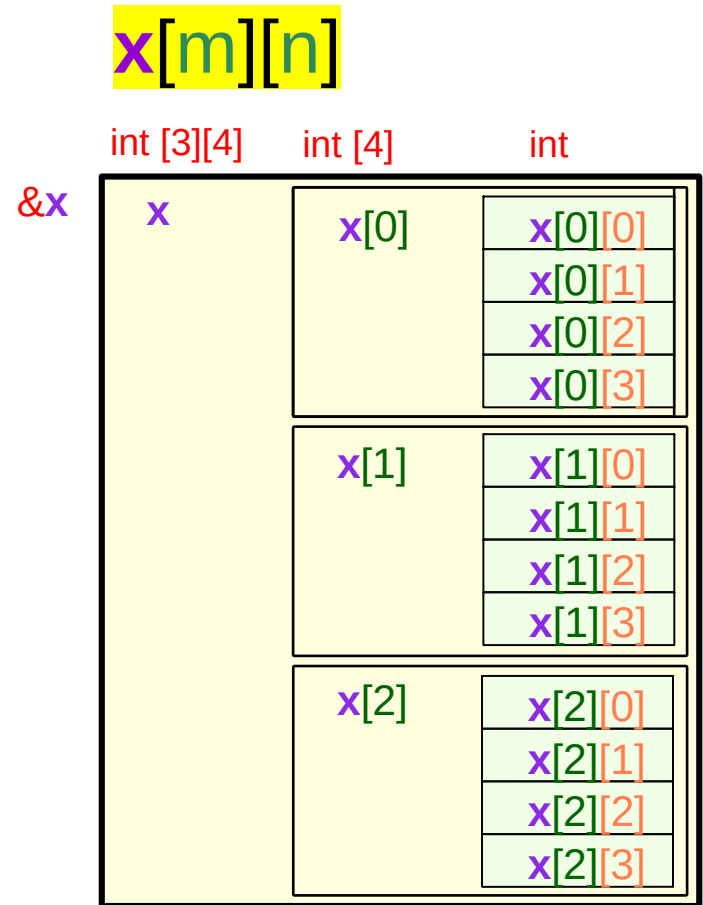
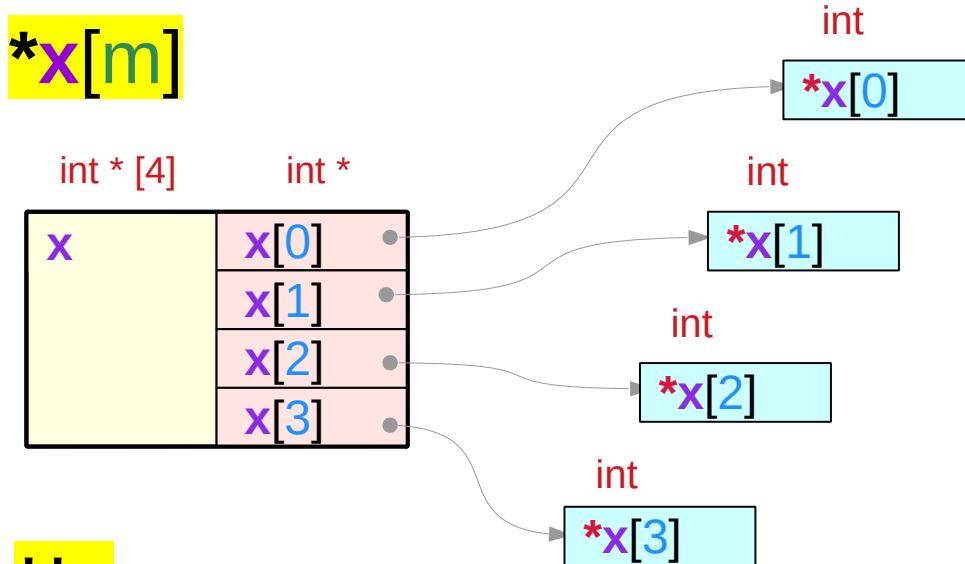
* has **right-to-left** associativity

$$(*x)[m][n] \leftrightarrow ((*x)[m])[n]$$

red parentheses () must not be removed
gray parentheses () can be removed

$$(*x[m])[n] \leftrightarrow (*(x[m]))[n]$$

Operator Precedence of * and []



Left-to-right associative subscript [] operators

$$p[i] \equiv p[i]$$

$$\rightarrow *(p+i)$$

$$p[i][j] \equiv (p[i])[j]$$

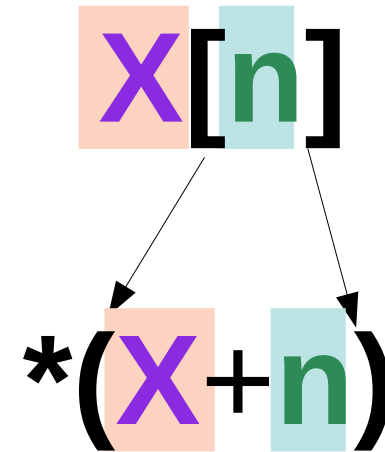
$$\rightarrow *(p[i]+j)$$

$$p[i][j][k] \equiv ((p[i])[j])[k]$$

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

$$*(X+n) \equiv X[n]$$

valid for the last dimension



Right-to-left associative dereference * operator

***p** \equiv ***(p)**

\rightarrow **p**[0]

****p** \equiv *****(p)**

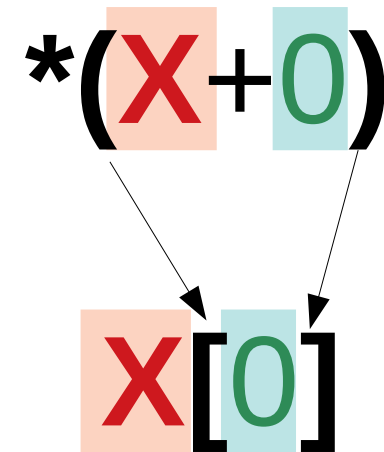
\rightarrow (**p**[0])[0]

*****p** \equiv ******(p)**

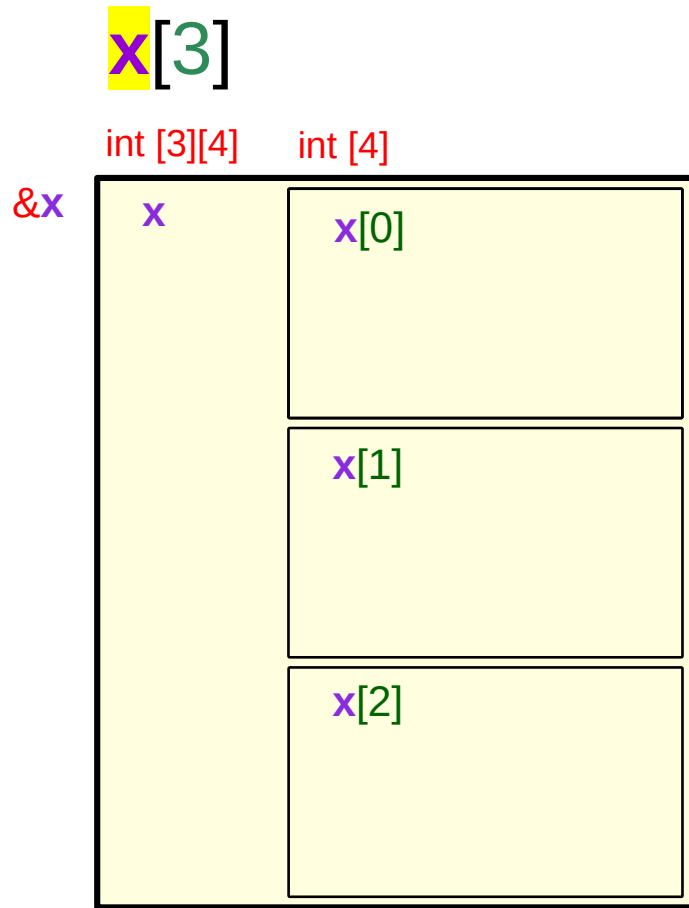
\rightarrow (((**p**[0])[0])[0])

***(X+0)** \equiv **X**[0]

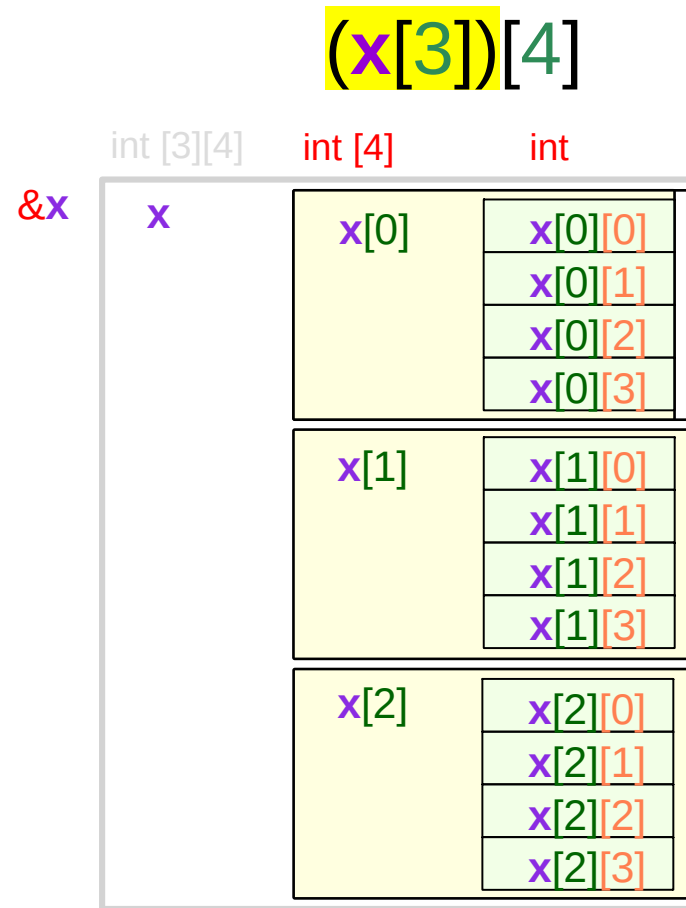
valid for any dimension



Abstract Data x and $x[i]$

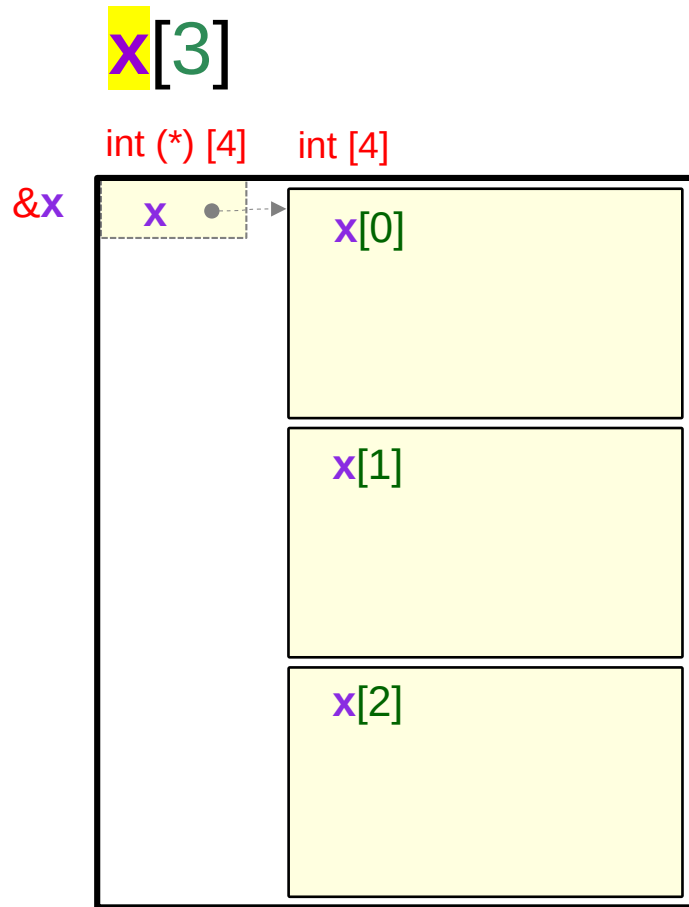


array element $x[i]$

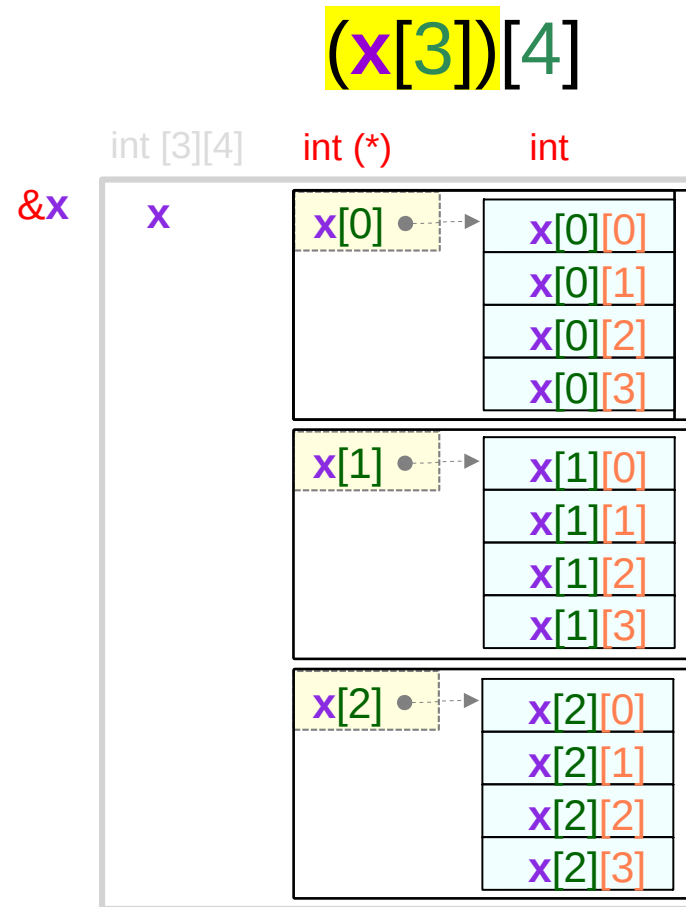


array name $x[i]$

Virtual Pointer x and $x[i]$



array name x virtual pointer
array element $x[i]$ abstract data



array name $x[i]$ virtual pointer
array element $x[i][j]$ primitive data

Subscript [] and dereference * operators (1a)

$p[i]$ \longleftrightarrow $*(p+i)$

skip
 $i \cdot M \cdot N$ integers

$\&p[i] = \text{value}((p + i)_{M \cdot N \cdot 4})$
 $= \text{value}(p) + i * M \cdot N \cdot 4$

$p[i][j]$ \longleftrightarrow $((*(p+i)+j)$

skip
 $j \cdot N$ integers

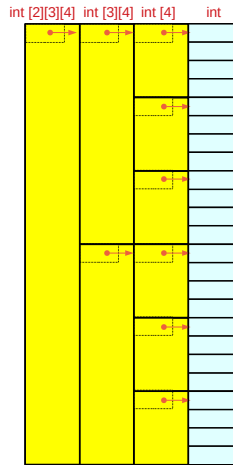
$\&p[i][j] = \text{value}((p[i] + j)_{N \cdot 4})$
 $= \text{value}(p[i]) + j * N \cdot 4$

$p[i][j][k]$ \longleftrightarrow $(((*(p+i)+j)+k)$

skip
 k integers

$\&p[i][j][k] = \text{value}((p[i][j] + k)_{1 \cdot 4})$
 $= \text{value}(p[i][j]) + k * 1 \cdot 4$

$\text{int } p[L][M][N]$



address replication

$\text{value}(p[i]) = \text{value}(p + i)$

$\text{value}(p[i][j]) = \text{value}(p[i] + j)$

$\&p[i][j][k] = \text{value}(p) + i * M \cdot N \cdot 4 + j * N \cdot 4 + k * 4$

Subscript [] and dereference * operators (1b)

$p[i]$ \longleftrightarrow $*(p+i)$

skip i pointers
in p

$$\begin{aligned} \&p[i] &= \text{value}(p + i)_{1..4} \\ &= \text{value}(p) + i * 4 \end{aligned}$$

$p[i][j]$ \longleftrightarrow $((* (p+i) + j)$

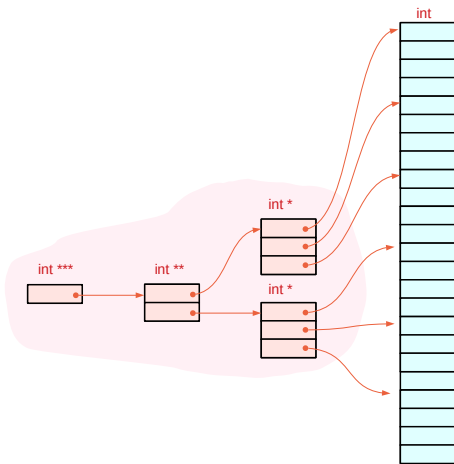
skip j pointers
in q

$$\begin{aligned} \&p[i][j] &= \text{value}(p[i] + j)_{1..4} \\ &= \text{value}(p[i]) + j * 4 \end{aligned}$$

$p[i][j][k]$ \longleftrightarrow $(((* (p+i) + j) + k)$

skip k integers
in r

$$\begin{aligned} \&p[i][j][k] &= \text{value}(p[i][j] + k)_{1..4} \\ &= \text{value}(p[i][j]) + k * 4 \end{aligned}$$

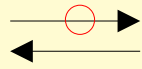


```
int ** p [L];
int * q [L·M];
int r [L·M·N];
```

$$\begin{aligned} \&p[i][j][k] &= *value(*value(*value(p + i * 4) + j * 4) + k * 4) \\ &= *value(*value(*value(p[i]) + j * 4) + k * 4) \end{aligned}$$

Subscript [] and dereference * operators (1a)

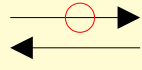
$p[i]$



$*(p+i)$

skip i elements

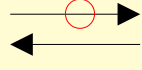
$p[i][j]$



$*(p[i]+j)$

skip j elements

$p[i][j][k]$



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

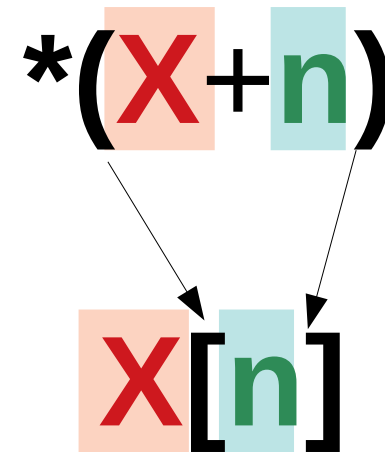
skip k elements

$*(X+n)$

\equiv

$X[n]$

valid for the last dimension



Address Calculation (1)

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

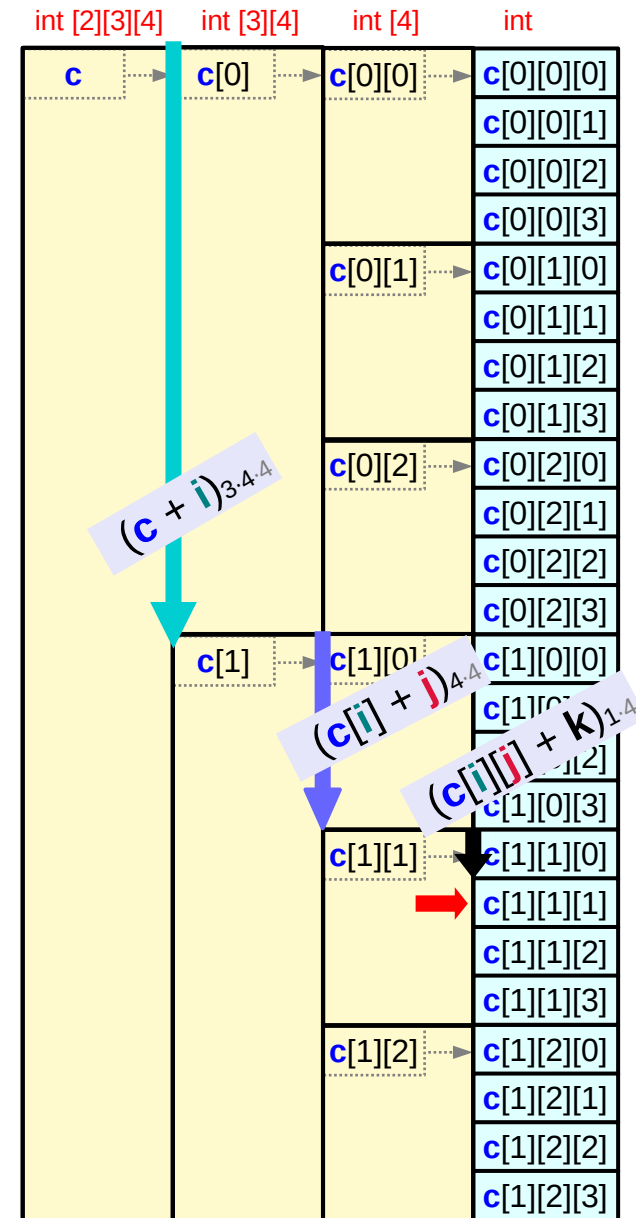
$c[i] \equiv *(c + i)$
 $c[i][j] \equiv *(c[i] + j)$
 $c[i][j][k] \equiv *(c[i][j] + k)$

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

$value(c + i) =$
 $value(c) + i * sizeof(*c)$

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

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



Address Calculation (2)

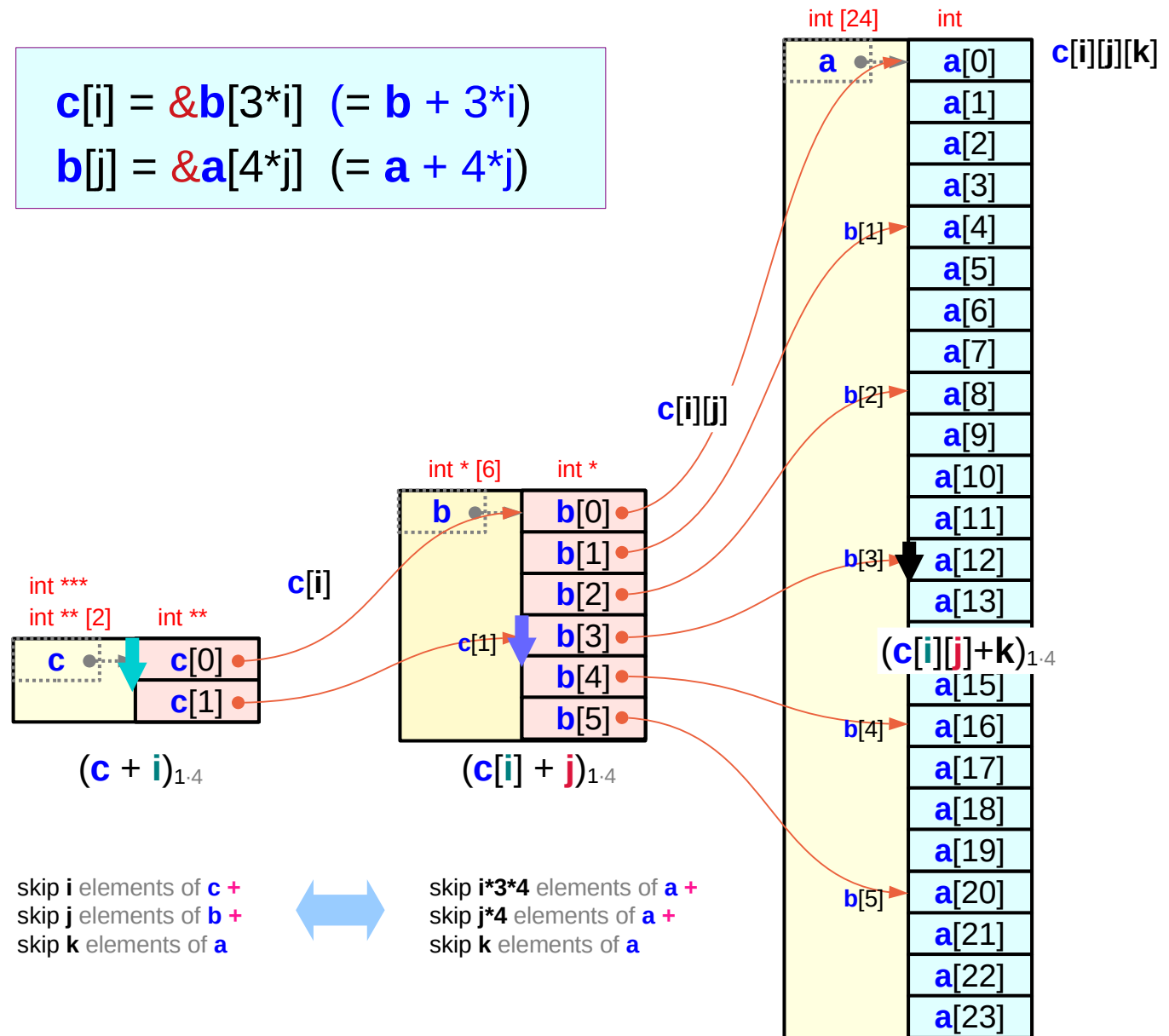
```
int ** c [2];
int *  b [2*3];
int   a [2*3*4];
```

```
c[i] = &b[3*i] (= b + 3*i)
b[j] = &a[4*j] (= a + 4*j)
```

```
b[j]      ≡ (a+j*4)
*(b[j]+k) = *(a+j*4+k);
b[j][k]   ≡ a[j*4+k]
```

```
c[i]      ≡ (b+i*3)
*(c[i]+j) = *(b+i*3+j);
c[i][j]   ≡ b[i*3+j]
```

```
c[i][j]   ≡ (a+(i*3+j)*4)
*(c[i][j]+k) = *(a+(i*3+j)*4+k);
c[i][j][k] ≡ a[(i*3+j)*4+k]
```



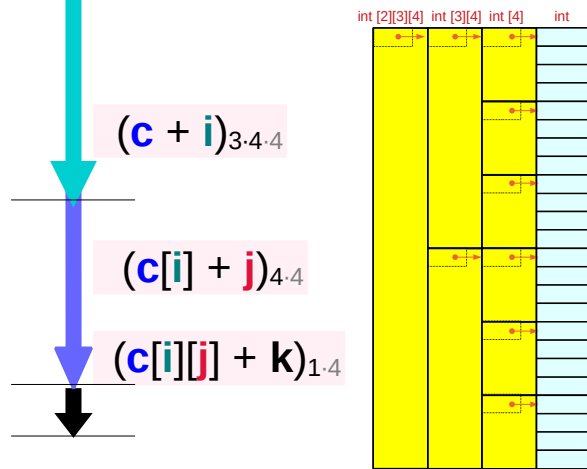
Address Calculation (2)

$$\begin{aligned} \text{value}(c + i) &= \text{value}(c) + i * 3 * 4 * 4 \\ \text{value}(c[i] + j) &= \text{value}(c[i]) + j * 4 * 4 \\ \text{value}(c[i][j] + k) &= \text{value}(c[i][j]) + k * 1 * 4 \end{aligned}$$

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

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

Array Pointer Approach

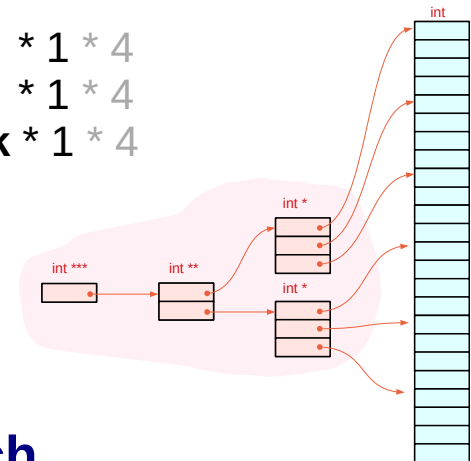


$$\begin{aligned} \text{value}(c + i) &= \text{value}(c) + i * \text{sizeof}(*c) \\ \text{value}(c[i] + j) &= \text{value}(c[i]) + j * \text{sizeof}(*c[i]) \\ \text{value}(c[i][j] + k) &= \text{value}(c[i][j]) + k * \text{sizeof}(*c[i][j]) \end{aligned}$$

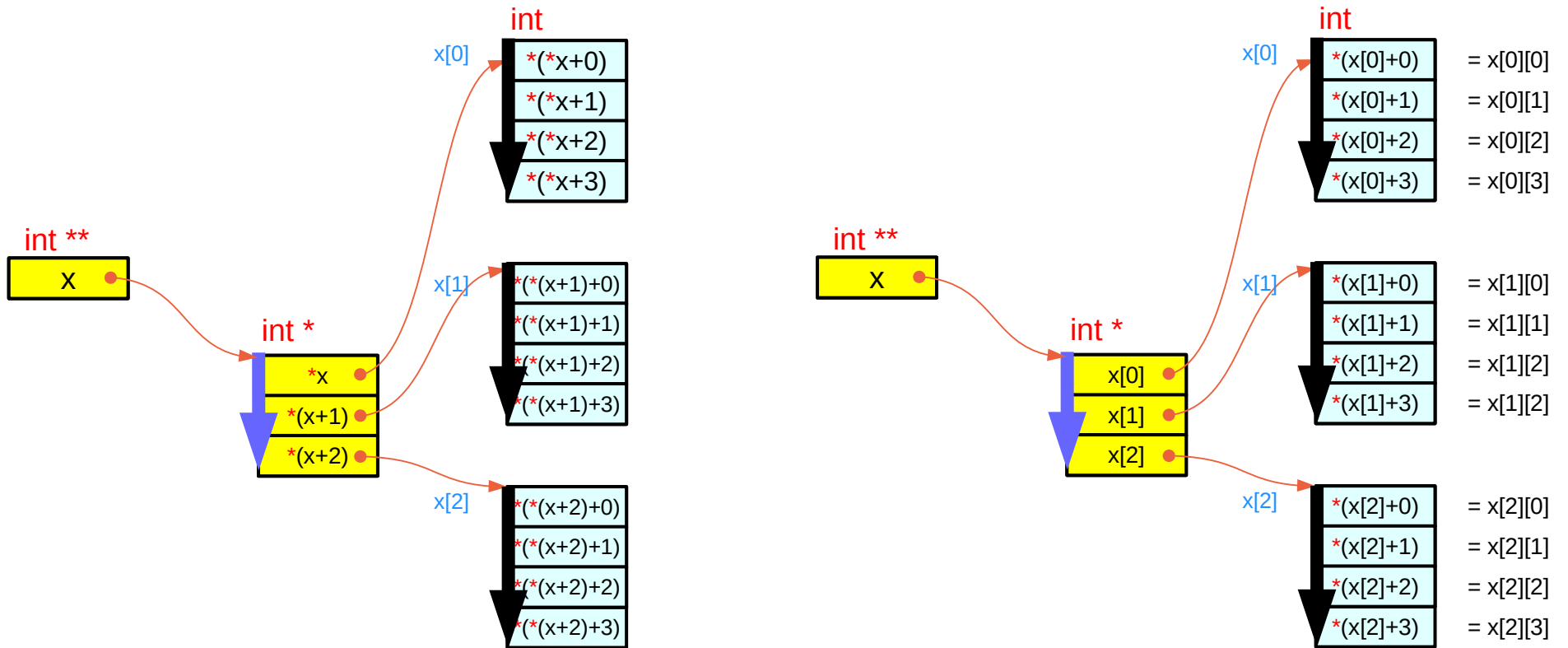
$$\begin{aligned} \text{value}(c + i) &= \text{value}(c) + i * 1 * 4 \\ \text{value}(c[i] + j) &= \text{value}(c[i]) + j * 1 * 4 \\ \text{value}(c[i][j] + k) &= \text{value}(c[i][j]) + k * 1 * 4 \end{aligned}$$

$$(c + i)_{1-4} \quad (c[i] + j)_{1-4} \quad (c[i][j] + k)_{1-4}$$

Pointer Array Approach



* and [] notations (1)



C expression

$*(*(\mathbf{x}+\mathbf{i})+\mathbf{j})$

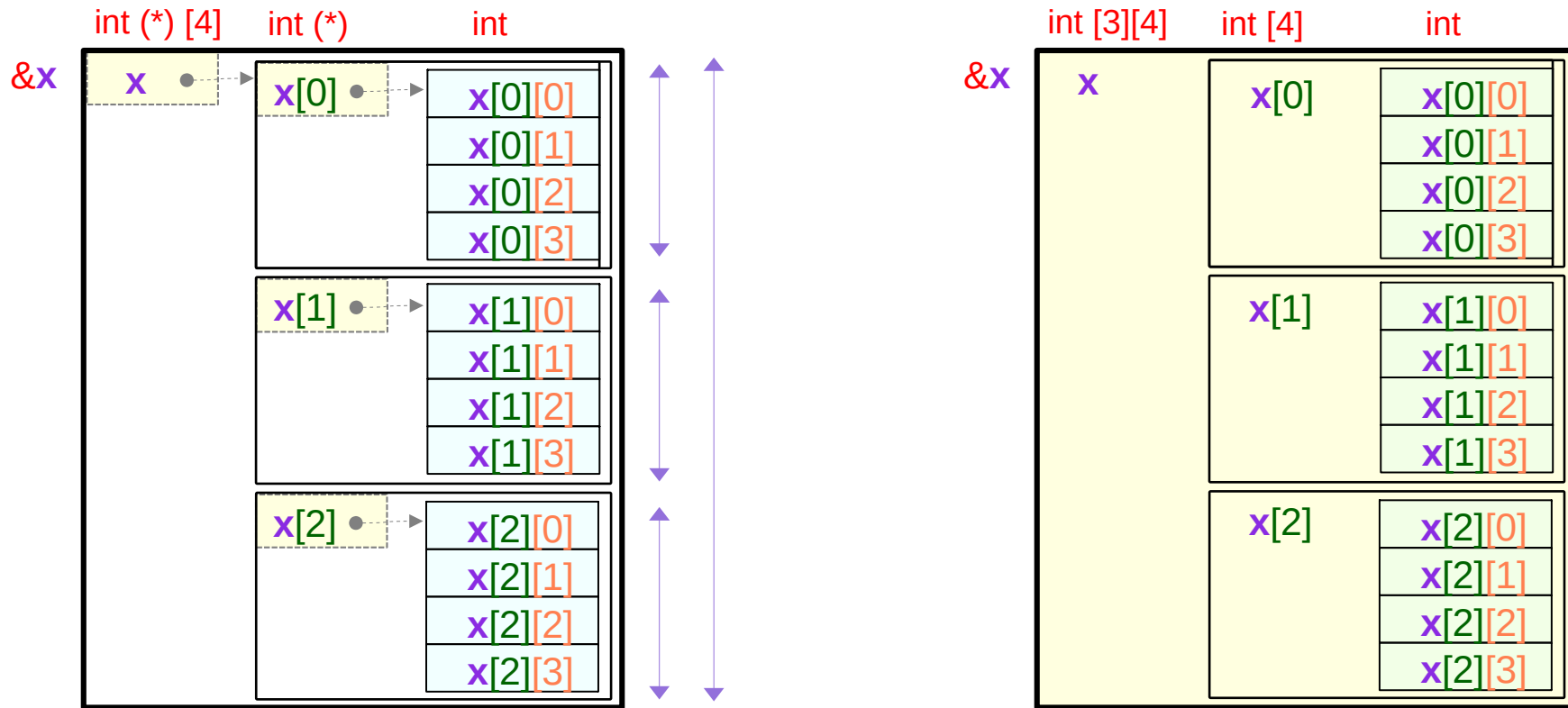


$\mathbf{x}[\mathbf{i}][\mathbf{j}]$

Math expression

$*(*(\mathbf{x}+\mathbf{i})_{1..4}+\mathbf{j})_{1..4}$

* and [] notations (2)



C expression

$$*(*(x+i)+j)$$

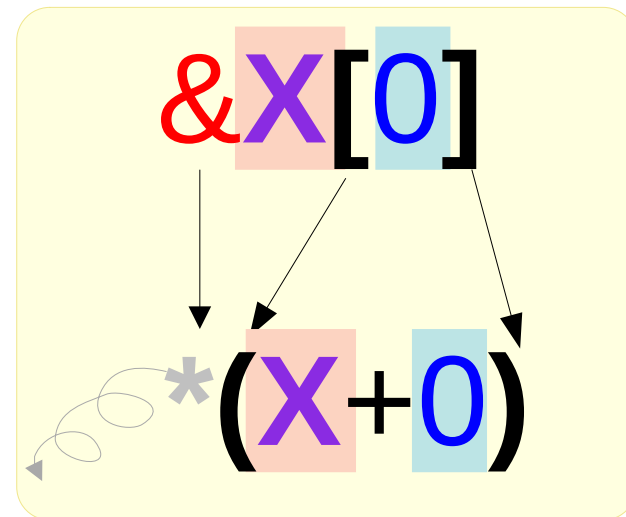
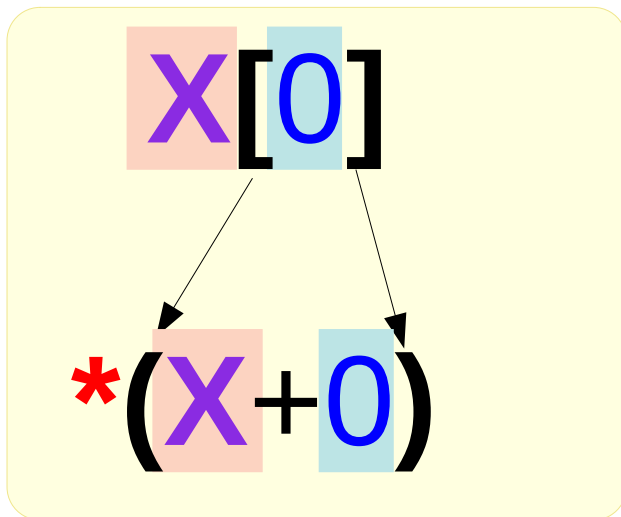
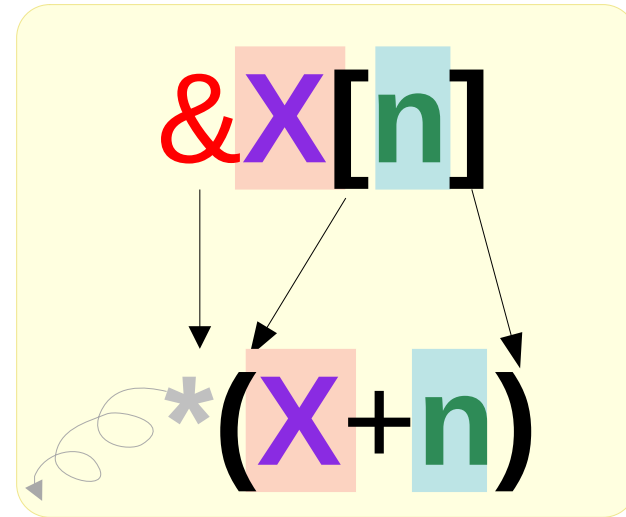
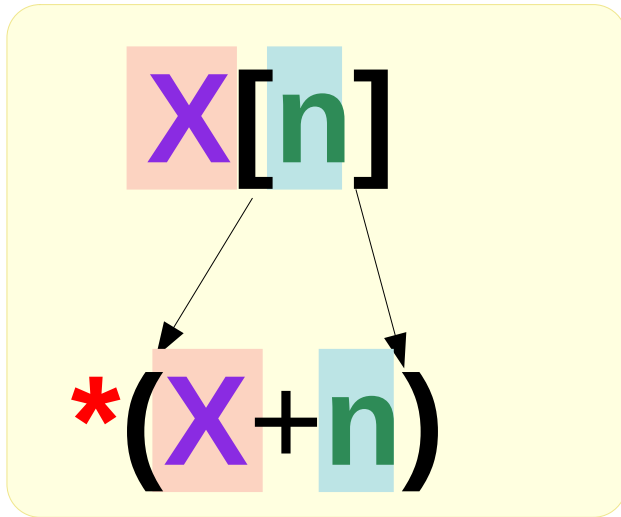


$$x[i][j]$$

Math expression

$$*(*(x+i)_{4 \cdot 4} + j)_{1 \cdot 4}$$

Equivalences (1)



Equivalences (2)

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

$\&p[i] \equiv (p+i)$
 $\&p[i][j] \equiv (p[i]+j)$
 $\&p[i][j][k] \equiv (p[i][j]+k)$

$p[0] \equiv *p$
 $p[i][0] \equiv *p[i]$
 $p[i][j][0] \equiv *p[i][j]$

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

Equivalences (3)

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

C Expressions

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

C Expressions

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

***X = X (address replication)**

$$\begin{aligned} p[i] &\longrightarrow *(p+i)_{M \cdot N \cdot 4} \\ p[i][j] &\longrightarrow *((p+i)_{M \cdot N \cdot 4} + j)_{N \cdot 4} \\ p[i][j][k] &\longrightarrow (((p+i)_{M \cdot N \cdot 4} + j)_{N \cdot 4} + k)_{1 \cdot 4} \end{aligned}$$

Math Expressions

$$\begin{aligned} \&p[i] &\longrightarrow (p+i)_{M \cdot N \cdot 4} \\ \&p[i][j] &\longrightarrow ((p+i)_{M \cdot N \cdot 4} + j)_{N \cdot 4} \\ \&p[i][j][k] &\longrightarrow (((p+i)_{M \cdot N \cdot 4} + j)_{N \cdot 4} + k)_{1 \cdot 4} \end{aligned}$$

Math Expressions

`int ** p[L], * q[L·M], r[L·M·N] ;`

$$\begin{aligned} p[i] &\longrightarrow *(p+i)_{1 \cdot 4} \\ p[i][j] &\longrightarrow *(*p+i)_{1 \cdot 4} + j)_{1 \cdot 4} \\ p[i][j][k] &\longrightarrow *(*(*p+i)_{1 \cdot 4} + j)_{1 \cdot 4} + k)_{1 \cdot 4} \end{aligned}$$

Math Expressions

$$\begin{aligned} \&p[i] &\longrightarrow (p+i)_{1 \cdot 4} \\ \&p[i][j] &\longrightarrow (*(p+i)_{1 \cdot 4} + j)_{1 \cdot 4} \\ \&p[i][j][k] &\longrightarrow *(*p+i)_{1 \cdot 4} + j)_{1 \cdot 4} + k)_{1 \cdot 4} \end{aligned}$$

Math Expressions

Equivalences (4)

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

***X = X** (address replication)

$$\&p[i] = \text{value}((p + i)_{M \cdot N \cdot 4}) = \text{value}(p) + i * M \cdot N \cdot 4$$

$$\&p[i][j] = \text{value}((p[i] + j)_{N \cdot 4}) = \text{value}(p[i]) + j * N \cdot 4$$

$$\&p[i][j][k] = \text{value}((p[i][j] + k)_{1 \cdot 4}) = \text{value}(p[i][j]) + k * 1 \cdot 4$$

$$\&p[i][j][k] = \text{value}(p) + i * M \cdot N \cdot 4 + j * N \cdot 4 + k * 4$$

$$\&p[i] \longrightarrow (p+i)_{M \cdot N \cdot 4}$$

$$\&p[i][j] \longrightarrow ((p+i)_{M \cdot N \cdot 4} + j)_{N \cdot 4}$$

$$\&p[i][j][k] \longrightarrow (((p+i)_{M \cdot N \cdot 4} + j)_{N \cdot 4} + k)_{1 \cdot 4}$$

Math Expressions

```
int ** p[L], * q[L·M], r[L·M·N] ;
```

$$\&p[i] = \text{value}(p + i)_{1 \cdot 4} = \text{value}(p) + i * 1 \cdot 4$$

$$\&p[i][j] = \text{value}(p[i] + j)_{1 \cdot 4} = \text{value}(p[i]) + j * 1 \cdot 4$$

$$\&p[i][j][k] = \text{value}(p[i][j] + k)_{1 \cdot 4} = \text{value}(p[i][j]) + k * 1 \cdot 4$$

$$\&p[i][j][k] = * \text{value}(* \text{value}(* \text{value}(p + i * 4) + j * 4) + k * 4)$$

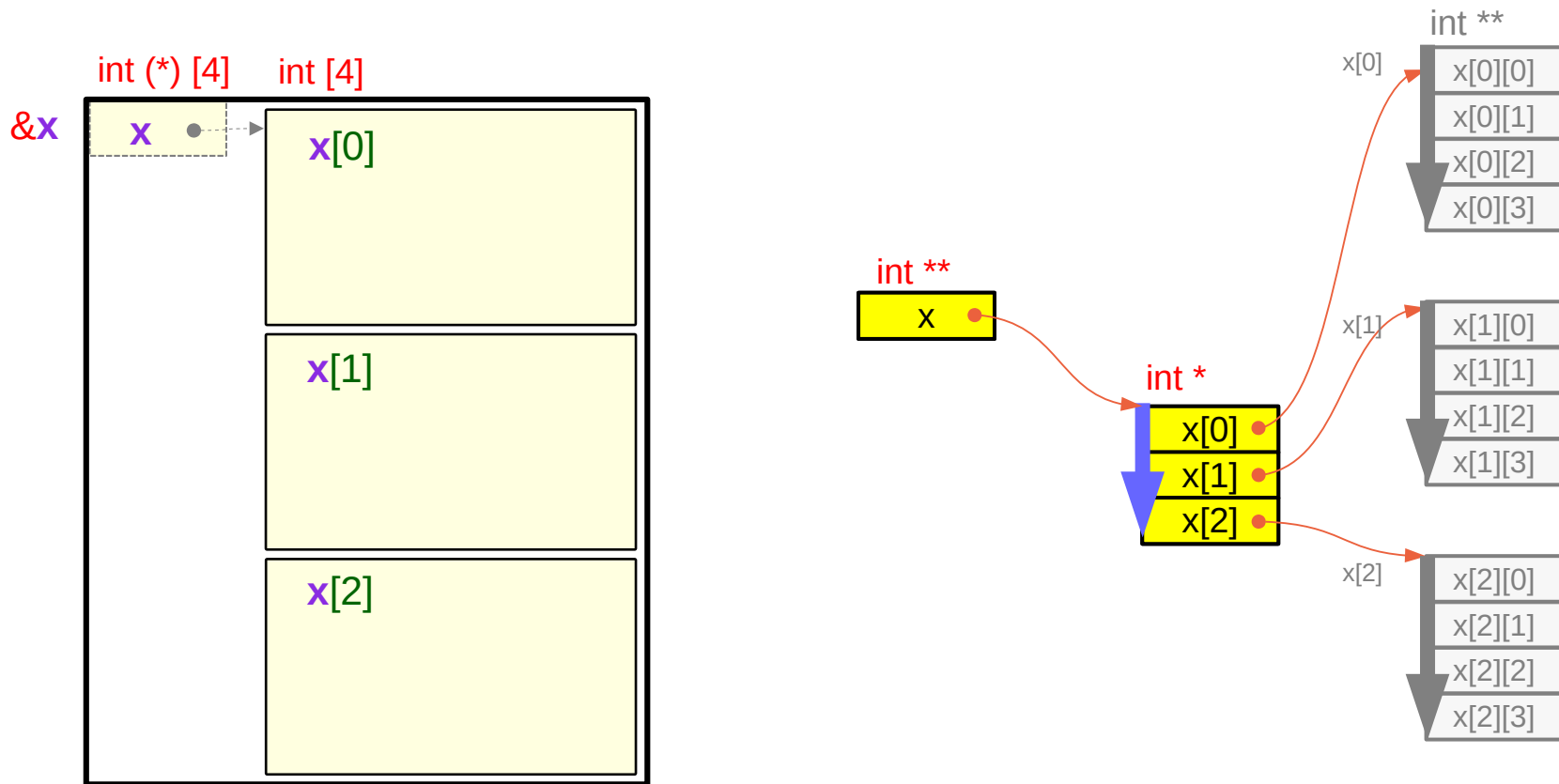
$$\&p[i] \longrightarrow (p+i)_{1 \cdot 4}$$

$$\&p[i][j] \longrightarrow (*(p+i)_{1 \cdot 4} + j)_{1 \cdot 4}$$

$$\&p[i][j][k] \longrightarrow (*(*(p+i)_{1 \cdot 4} + j)_{1 \cdot 4} + k)_{1 \cdot 4}$$

Math Expressions

Virtual pointers vs. real pointers (1)



`&x = x` `sizeof(x) = 3 * sizeof(*x)`

`x[i] = *(x + i)4,4`

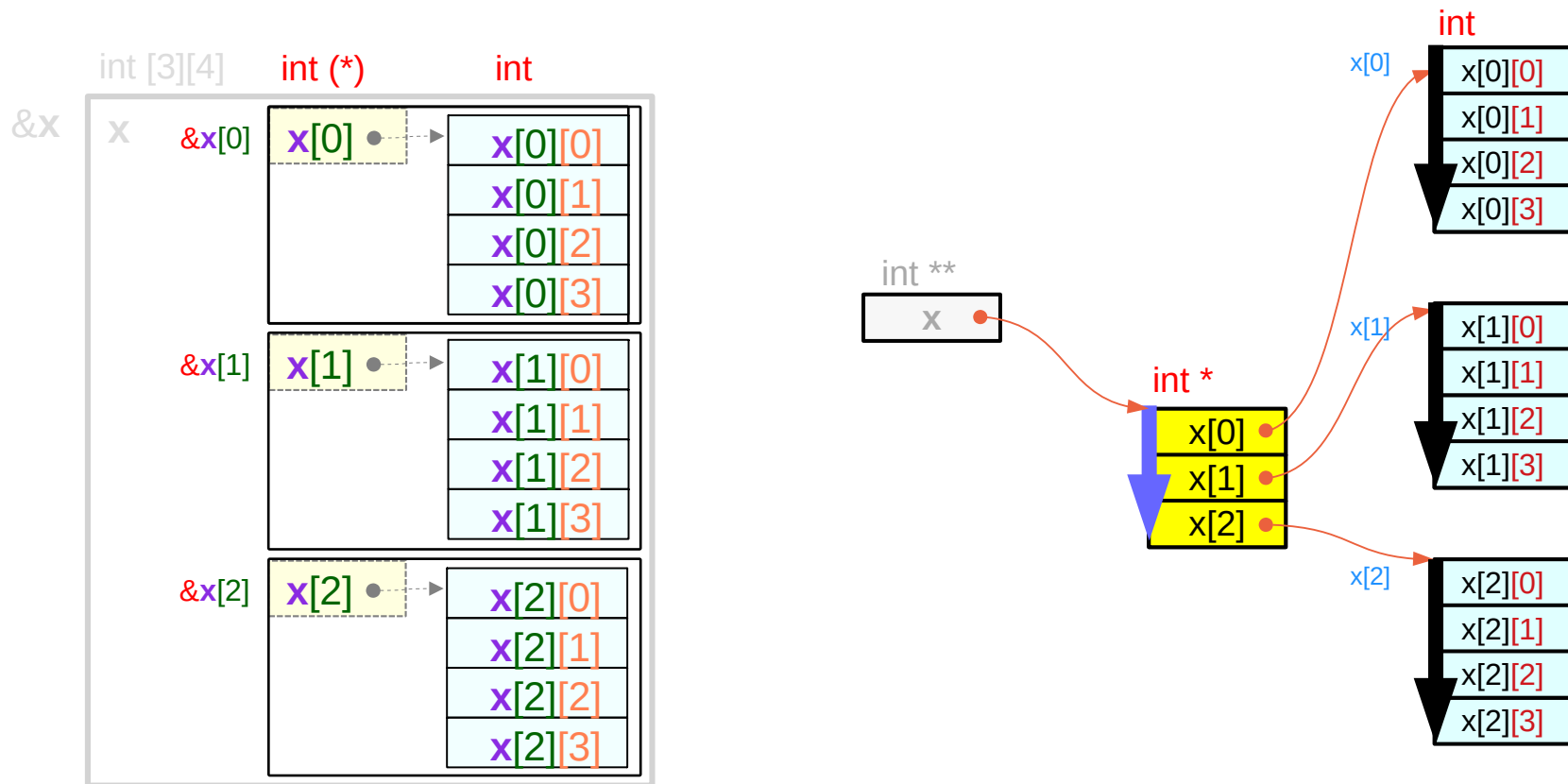
`value(x+i) = value(x) + i * sizeof(*x)`
`= value(x) + i * 4 * 4`

`&x ≠ x` `sizeof(x) = sizeof(*x) = 4`

`x[i] = *(x + i)1,4`

`value(x+i) = value(x) + i * sizeof(*x)`
`= value(x) + i * 4`

Virtual pointers vs. real pointers (2)



$$\&x[i] = x[i] \quad \text{sizeof}(x[i]) = 4 * \text{sizeof}(*x[i])$$

$$x[i] = *(x + i)_{1..4}$$

$$\begin{aligned} \text{value}(x[i]+j) &= \text{value}(x[i]) + j * \text{sizeof}(*x[i]) \\ &= \text{value}(x[i]) + j * 4 \end{aligned}$$

$$\&x[i] = x[i] \quad \text{sizeof}(x[i]) = \text{sizeof}(*x[i]) = 4$$

$$x[i][j] = *(x[i] + j)_{1..4}$$

$$\begin{aligned} \text{value}(x[i]+j) &= \text{value}(x[i]) + j * \text{sizeof}(*x[i]) \\ &= \text{value}(x[i]) + j * 4 \end{aligned}$$

Operator Precedence

Precedence	Operator	Description	Associativity
1	++ -- () [] . -> (type){list}	Suffix/postfix increment and decrement Function call Array subscripting Structure and union member access member access through pointer Compound literal(C99)	Left-to-right (((x[m])[n])[p]) →
2	++ -- + - ! ~ (type) * & sizeof _Alignof	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT Type cast Indirection (dereference) Address-of Size-of Alignment requirement(C11)	Right-to-left *((**(*X))) ←

https://en.cppreference.com/w/c/language/operator_precedence

Limitations

No index Range Checking

Array Size must be a constant expression

Variable Array Size

Arrays cannot be Copied or Compared

Aggregate Initialization and Global Arrays

Precedence Rule

Index Type Must be Integral

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
- [5] <https://pdos.csail.mit.edu/6.828/2008/readings/pointers.pdf>