

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

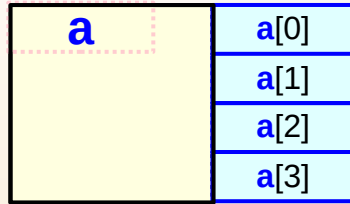
not the size of the array

but of a pointer variable

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

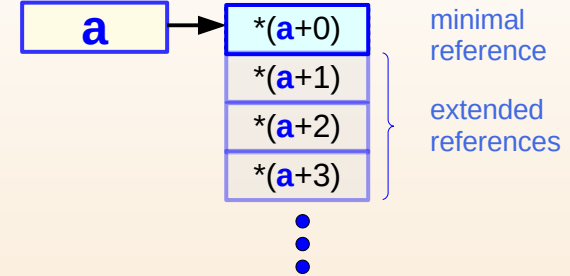
`int a[4];`

an array **a**



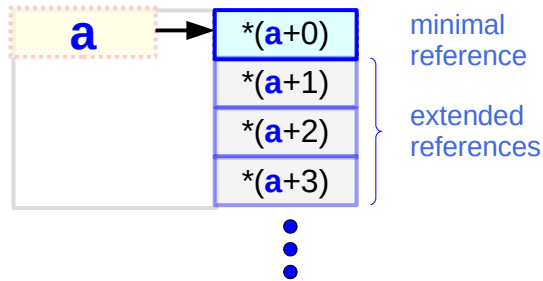
`int (*a);`

a pointer **a**



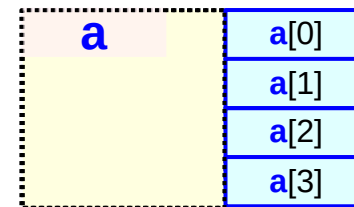
`int (*a)`

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

`int [4][2]` **2-d array**
more constrained type

relaxing the
1st dimension
generalization



a specific instance



`int (*)[2]` **1-d array pointer**
more general type

`int [4]` **1-d array**
more constrained type

relaxing the
1st dimension
generalization



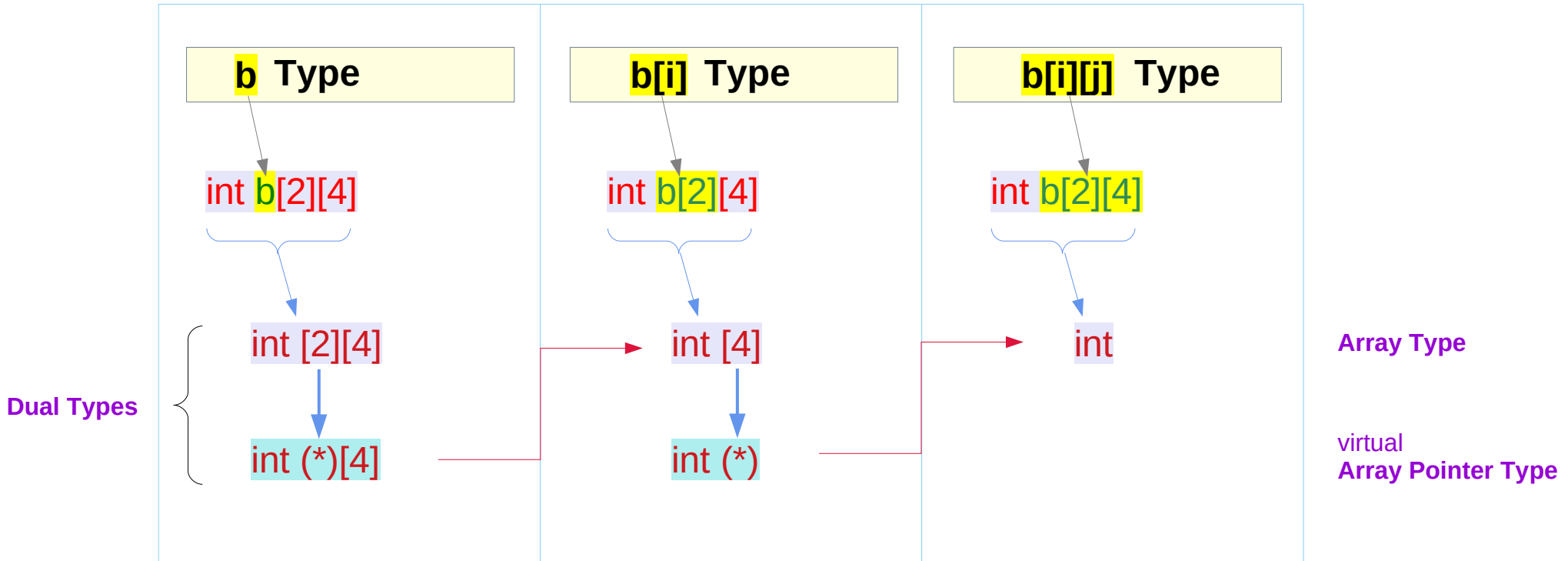
a specific instance



`int (*)` **0-d array pointer**
more general type

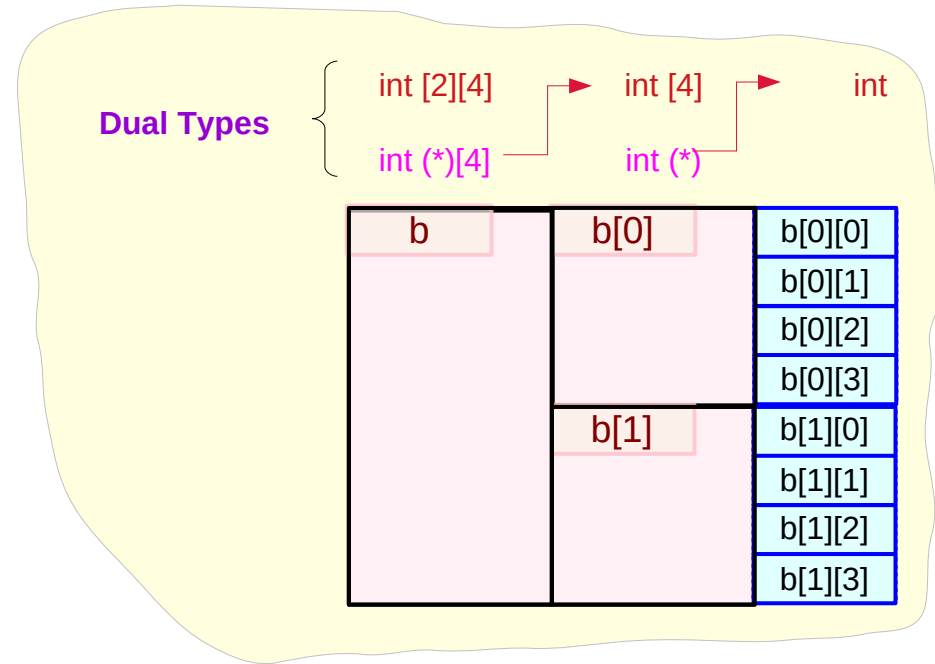
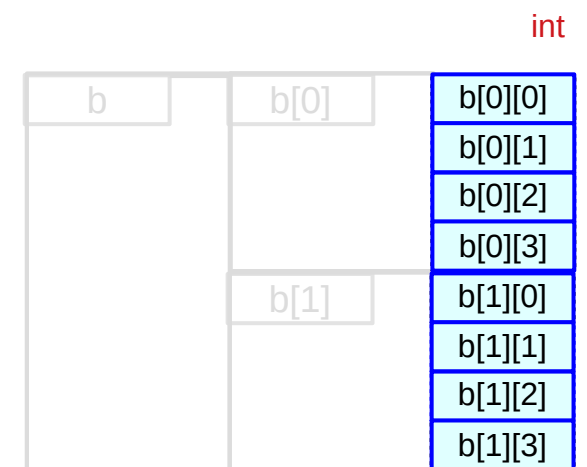
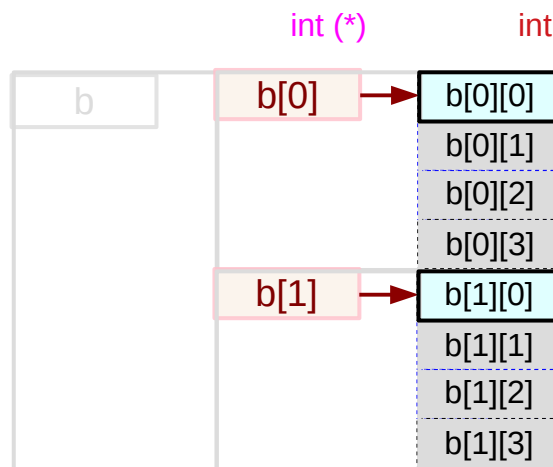
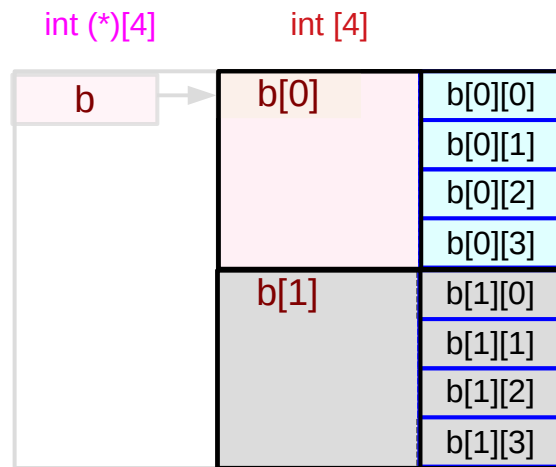
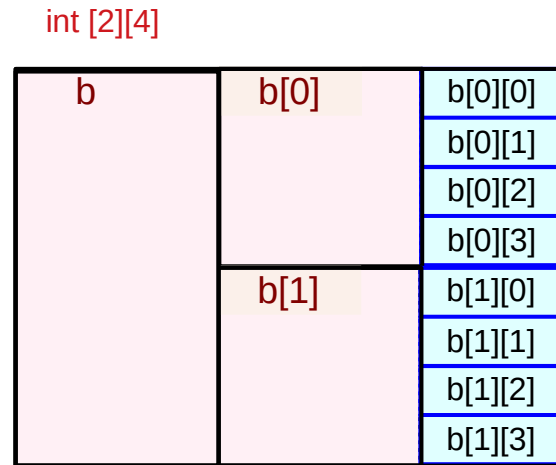
Subarray types in a 2-d array

`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		

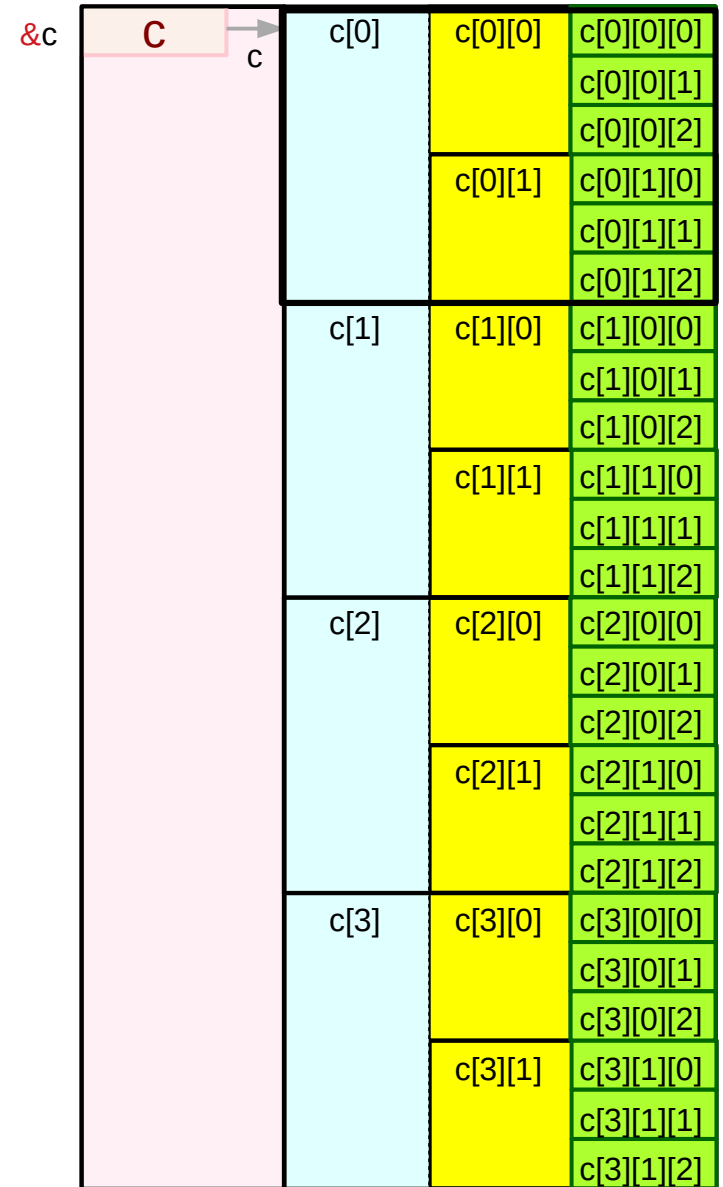
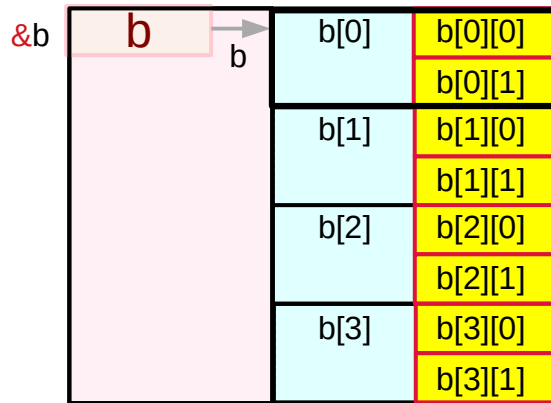
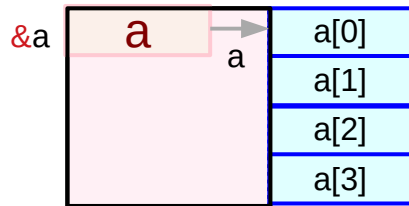
```
int b[2][4];
```

			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



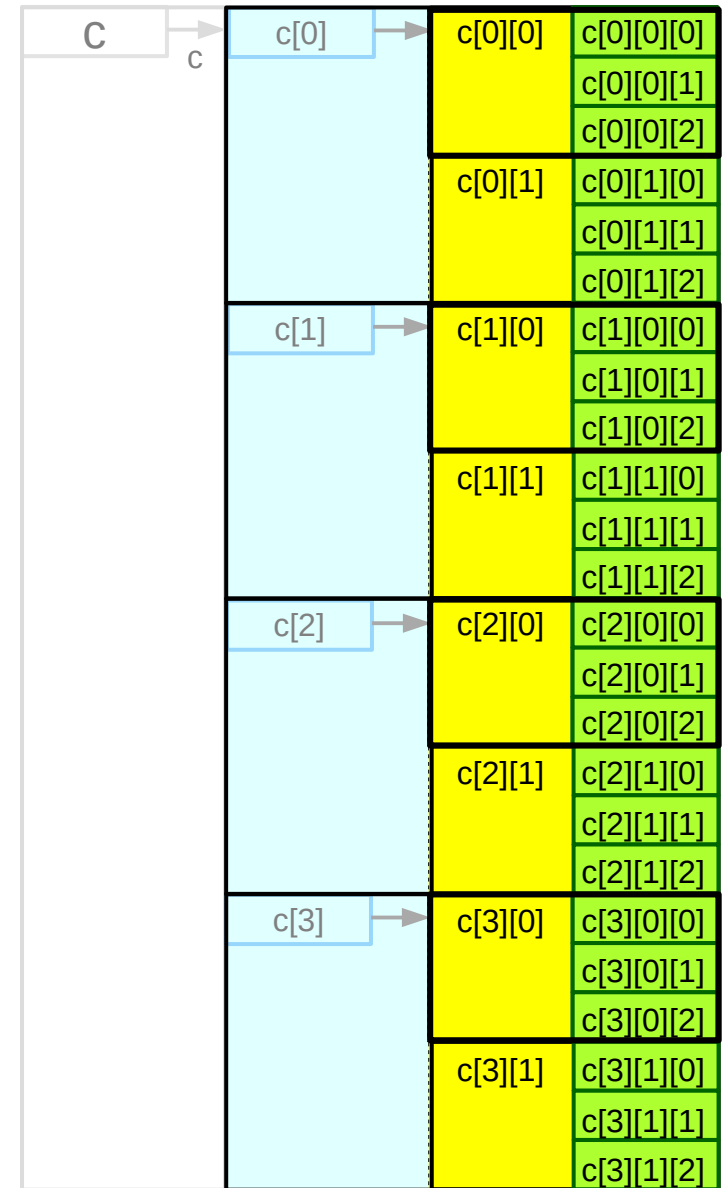
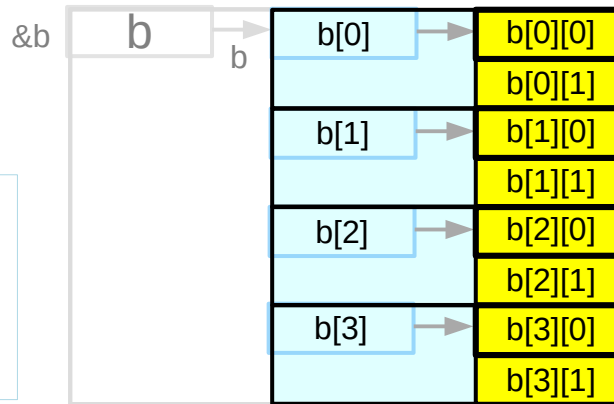
```
int a[4];
int b[2][4];
int c[4][2][3];
```

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];

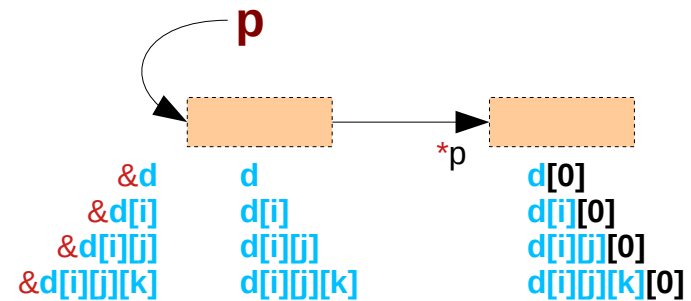
types

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

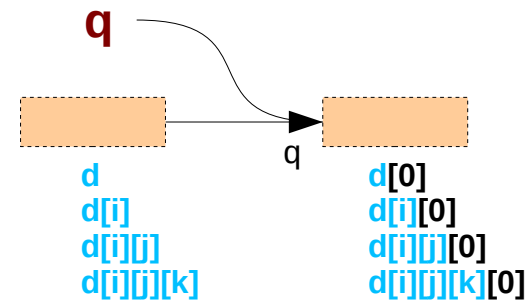
```
int d[4][2][3][4];
```



d	4-d array	<code>d[4][2][3][4]</code>	<code>p = &d</code>	abstract data
p	4-d array pointer	<code>(*p)[4][2][3][4]</code>	<code>int (*p)[4][2][3][4] = &d;</code> <code>(*p)[i][j][k][l] ≡ d[i][j][k][l]</code>	
d[i]	3-d array	<code>d[i][2][3][4]</code>	<code>p = &d[i]</code>	abstract data
p	3-d array pointer	<code>(*p)[2][3][4]</code>	<code>int (*p)[3][4] = &d[i];</code> <code>(*p)[j][k][l] ≡ d[i][j][k][l] given i</code>	
d[i][j]	2-d array	<code>d[i][j][3][4]</code>	<code>p = &d[i][j]</code>	abstract data
p	2-d array pointer	<code>(*p)[3][4]</code>	<code>int (*p)[4] = &d[i][j];</code> <code>(*p)[k][l] ≡ d[i][j][k][l] given i, j</code>	
d[i][j][k]	1-d array	<code>d[i][j][k][4]</code>	<code>p = &d[i][j][k]</code>	abstract data
p	1-d array pointer	<code>(*p)[4]</code>	<code>int (*p) = &d[i][j][k];</code> <code>(*p)[l] ≡ d[i][j][k][l] given i, j, k</code>	

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

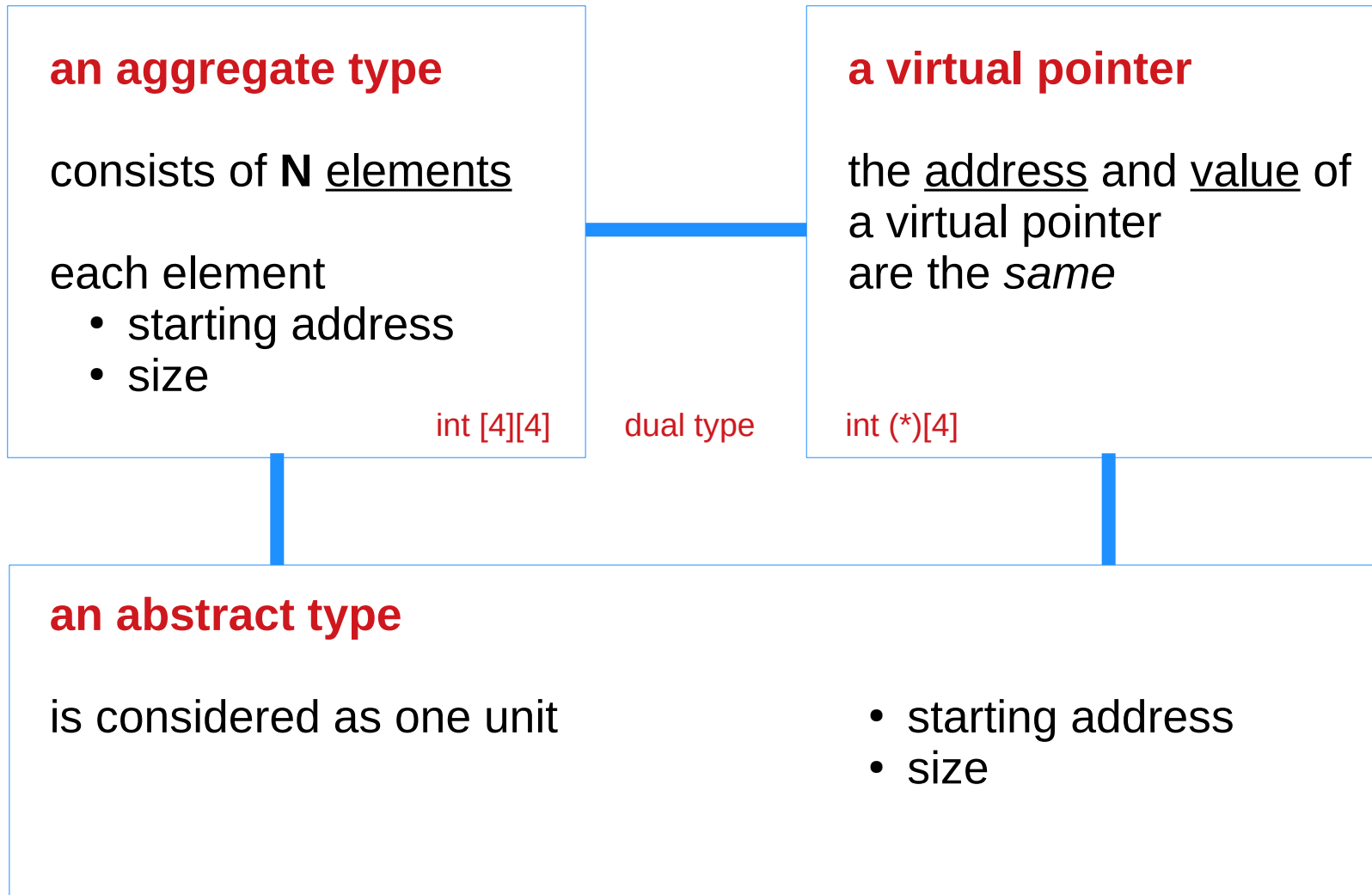
```
int d[4][2][3][4];
```



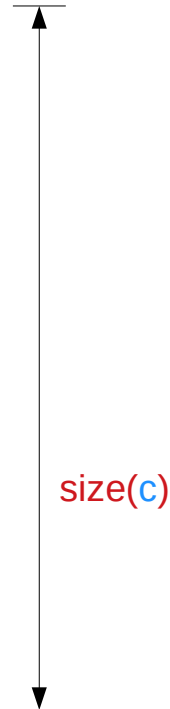
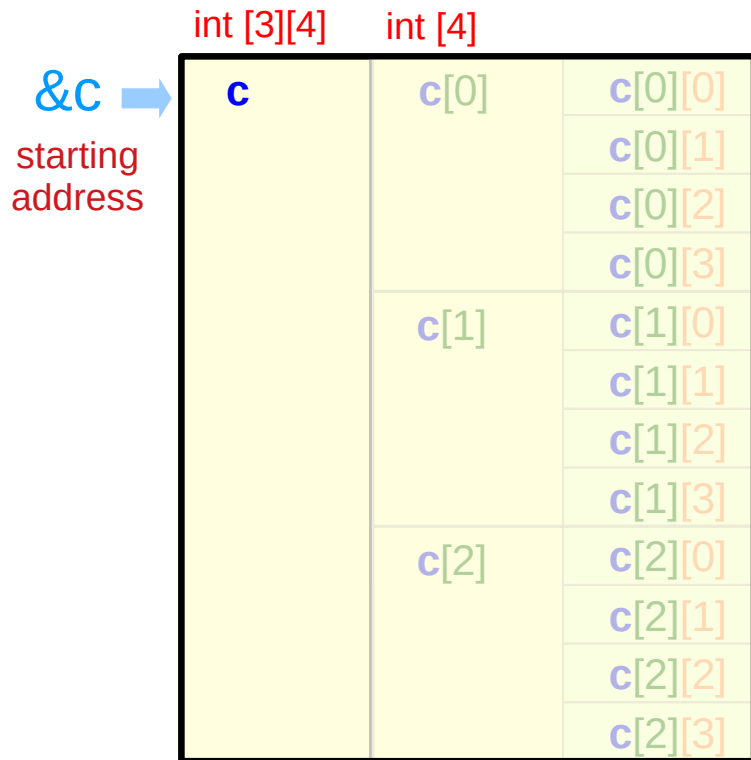
d	4-d array	<code>d[4][2][3][4]</code>	<code>q = d</code>	virtual pointer
q	3-d array pointer	<code>(*q)[2][3][4]</code>	<code>int (*q)[2][3][4] = d;</code> <code>q[i][j][k][l] ≡ d[i][j][k][l]</code>	
d[i]	3-d array	<code>d[i][2][3][4]</code>	<code>q = d[i]</code>	virtual pointer
q	2-d array pointer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



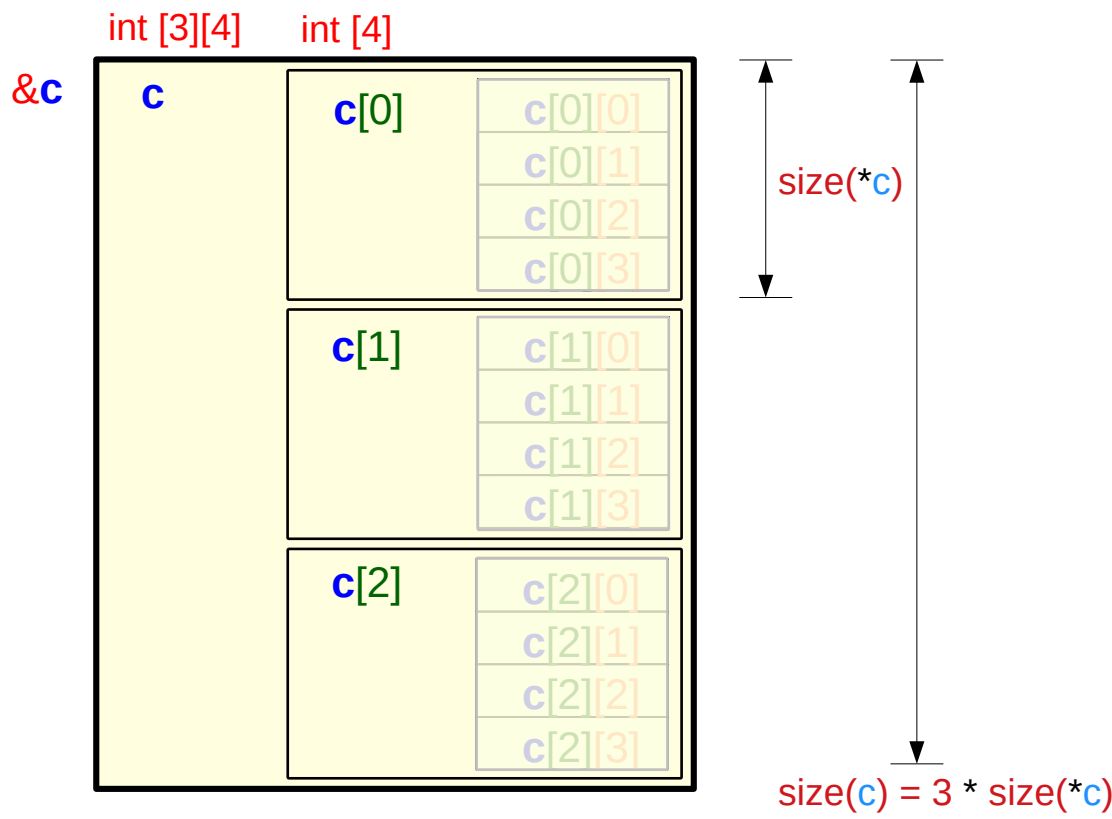
Abstract data **c**



an abstract data
- start address
- size

c
&c
sizeof(c)

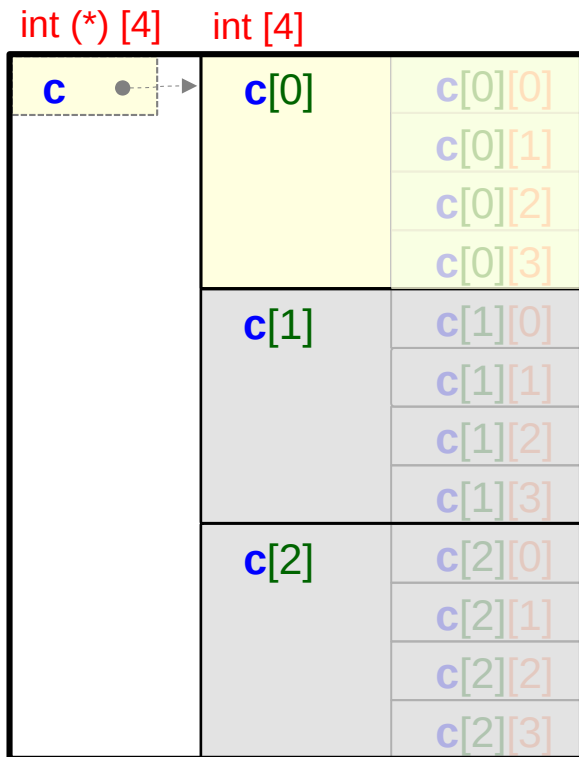
Aggregate data **c**



- 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
 $c = \&c$

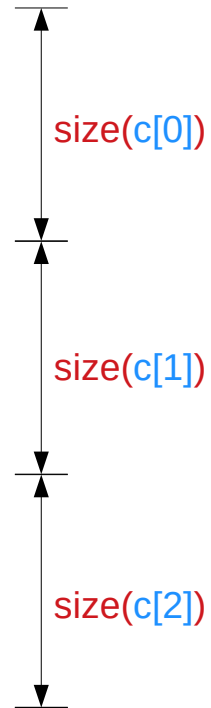
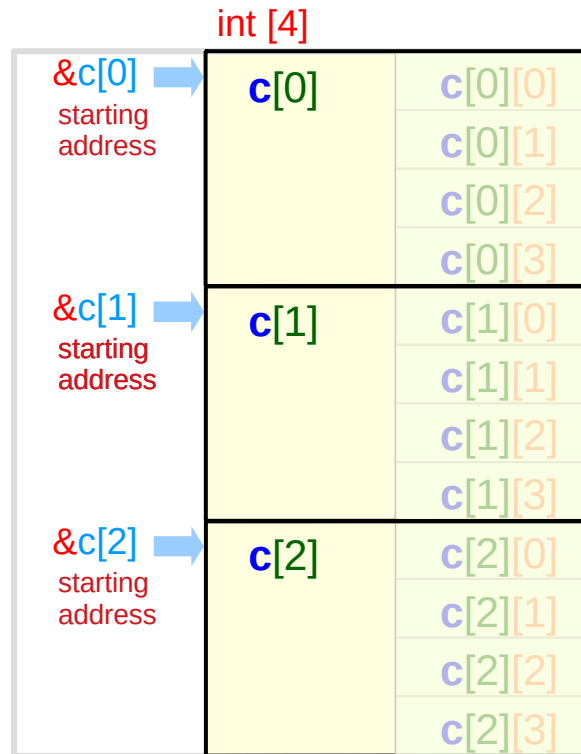
an abstract data $c[0] = *c$
- start address $\&c[0] = c$
- size $\text{sizeof}(c[0])$

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

virtual pointers

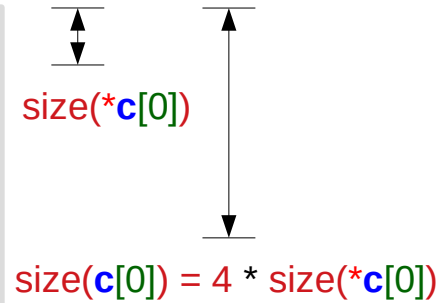
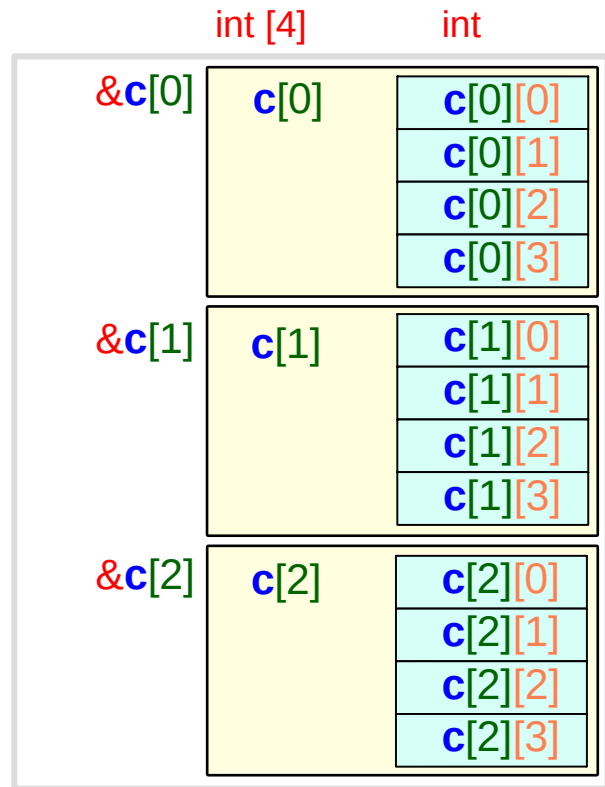
- no physical memory locations are allocated
- address and data have the same value

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 $\&c[2]$
 - $sizeof(c[2])$

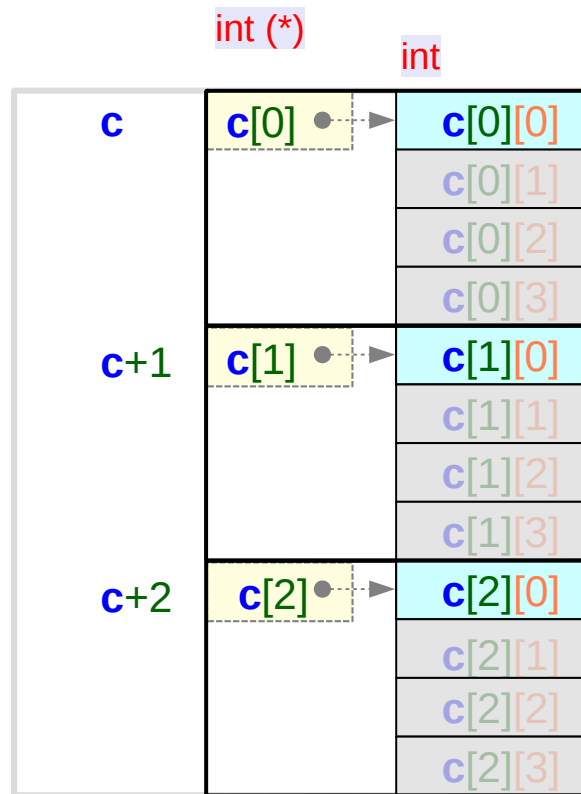
Aggregate data $c[i]$



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 $\text{sizeof}(c[i][0])$

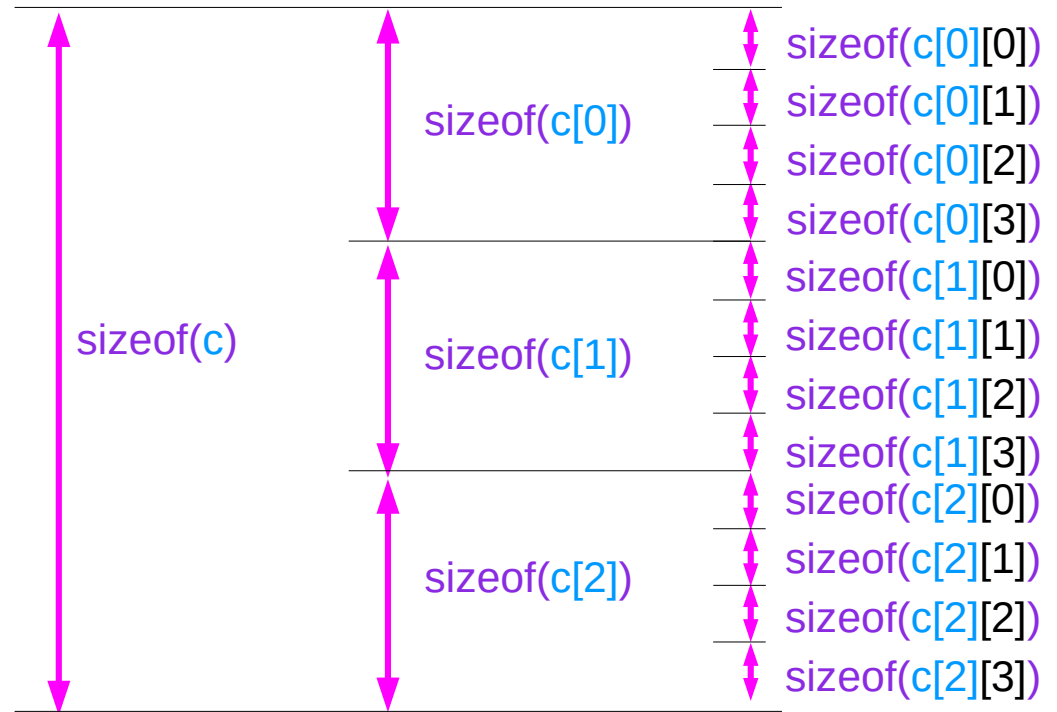
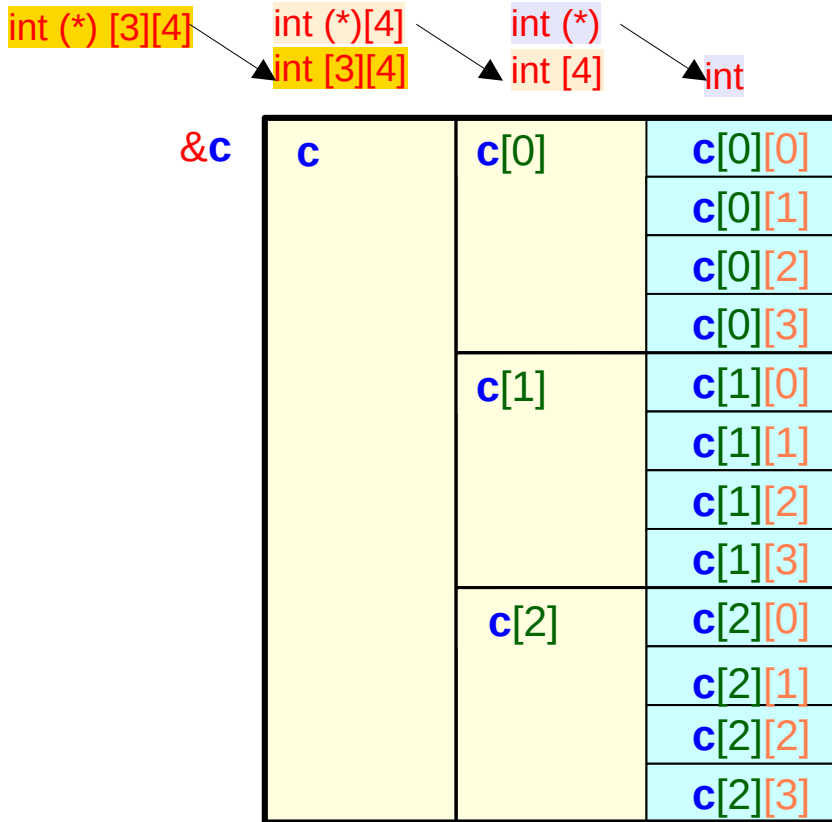
virtual pointer $c[i]$
points to primitive data $c[i][0]$

virtual pointers

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

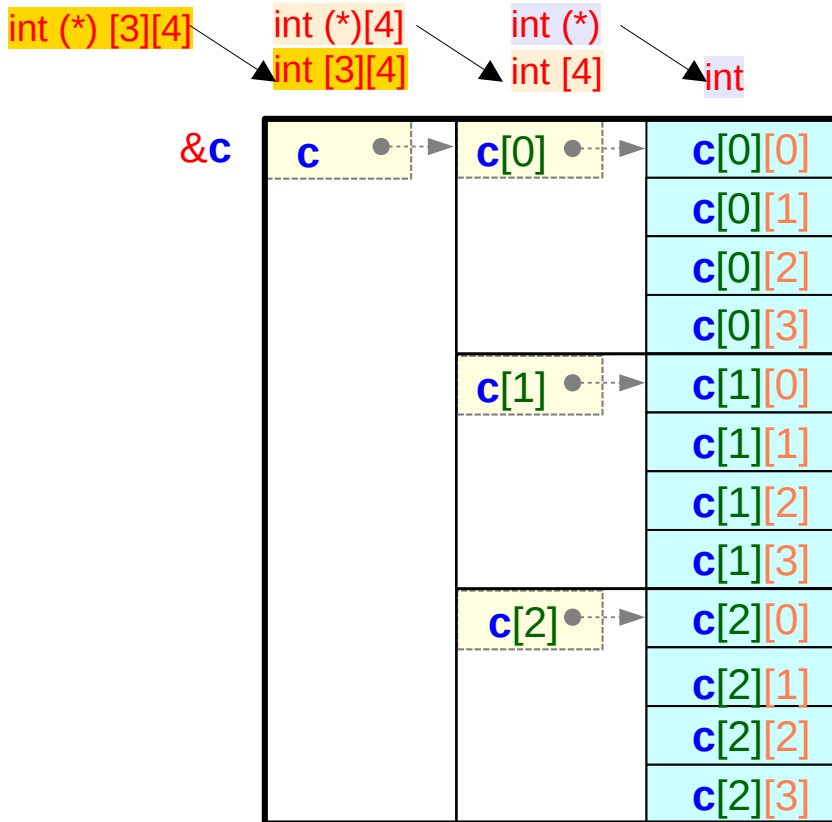
A 2-d array and its 1-d sub-arrays – a size view

```
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])`

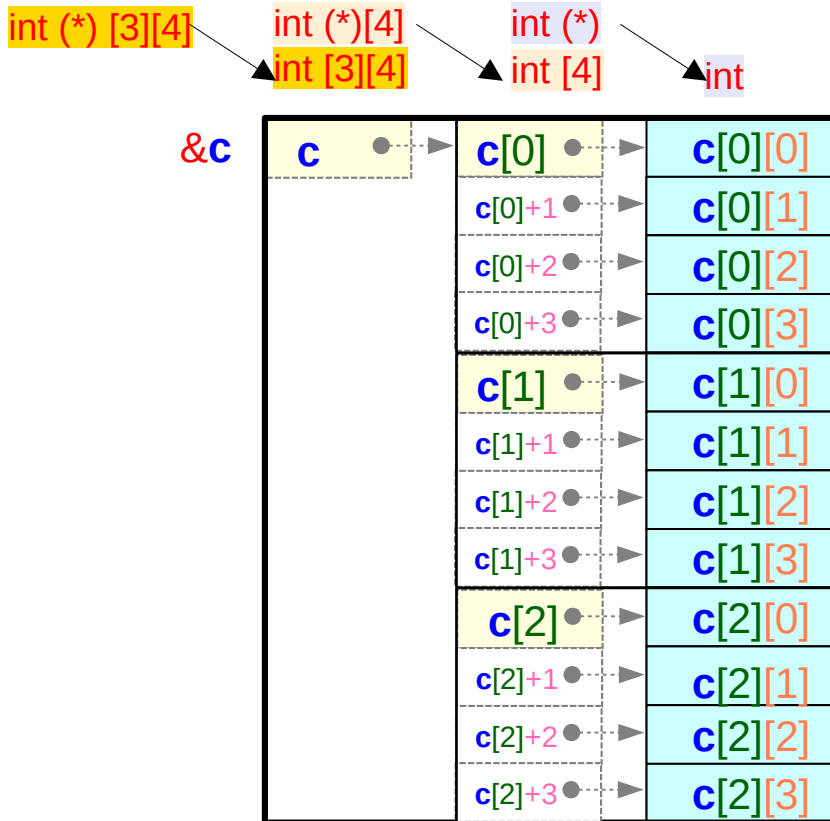
`value(c[1]) = value(&c[1][0])`
`value(&c[1]) = value(&c[1][0])`

`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)

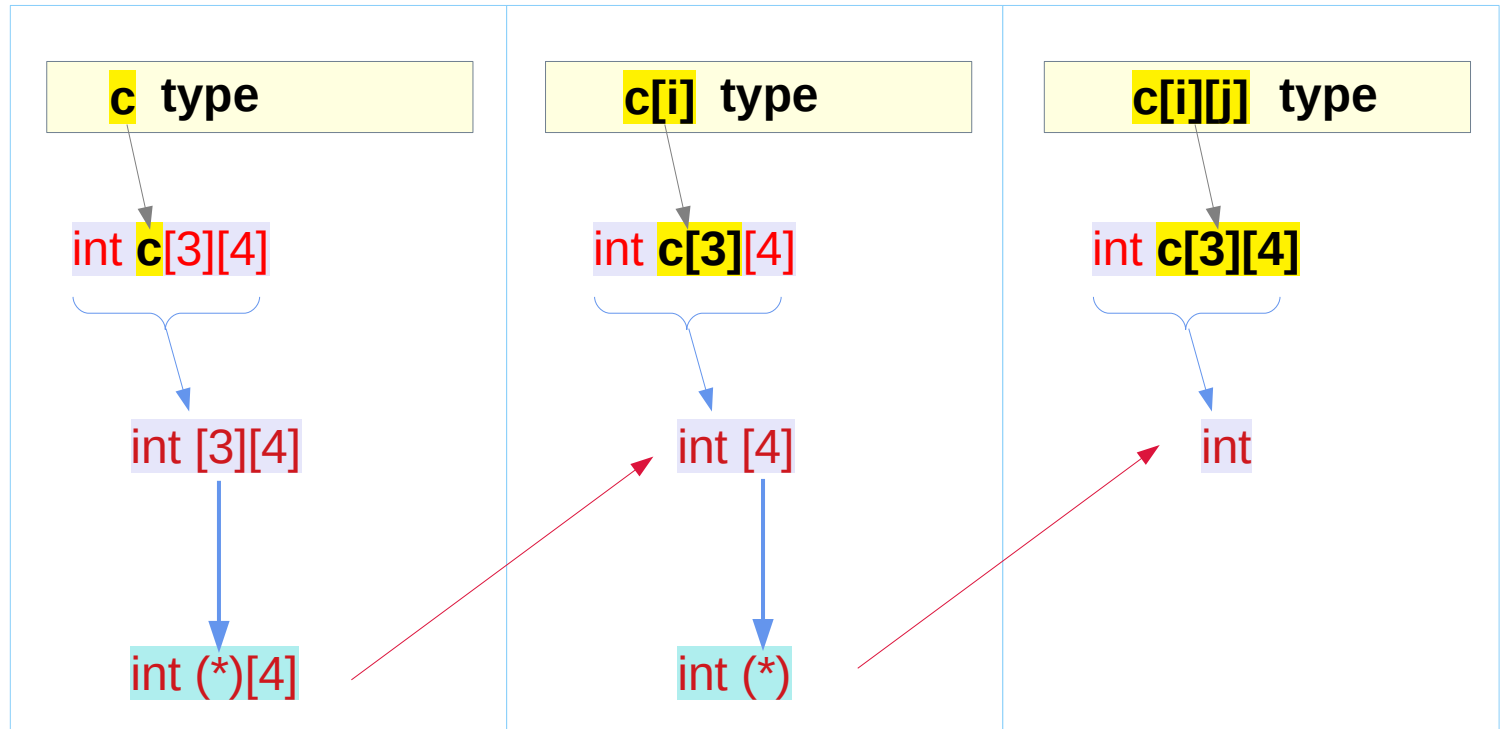
`sizeof(c[0])` = `sizeof(c[0][0])` * 4 ... leading element
`sizeof(c[0]+1)` = pointer size (4/8 bytes)
`sizeof(c[0]+2)` = pointer size (4/8 bytes)
`sizeof(c[0]+3)` = pointer size (4/8 bytes)

`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

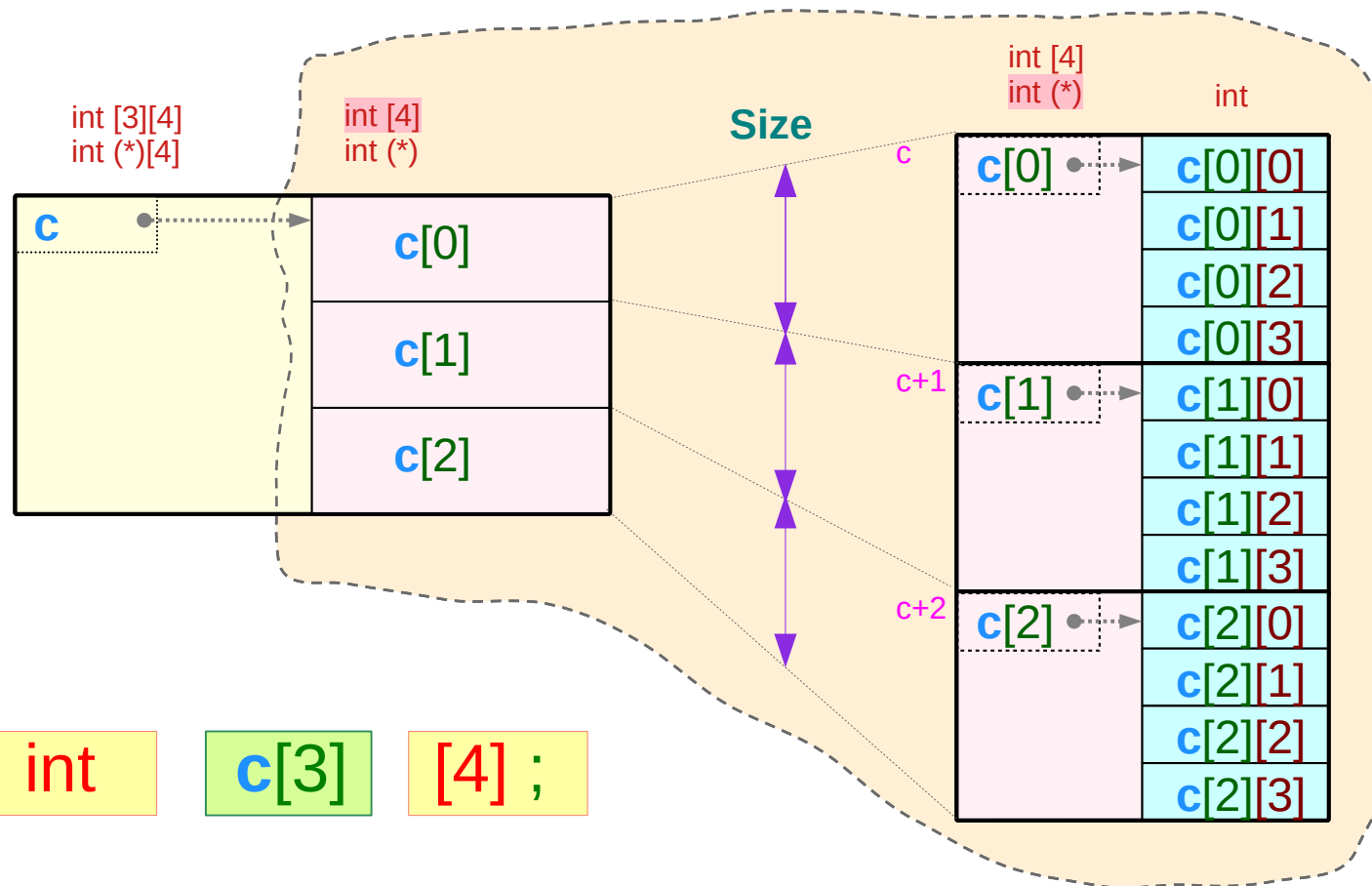
`c[3]`

`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

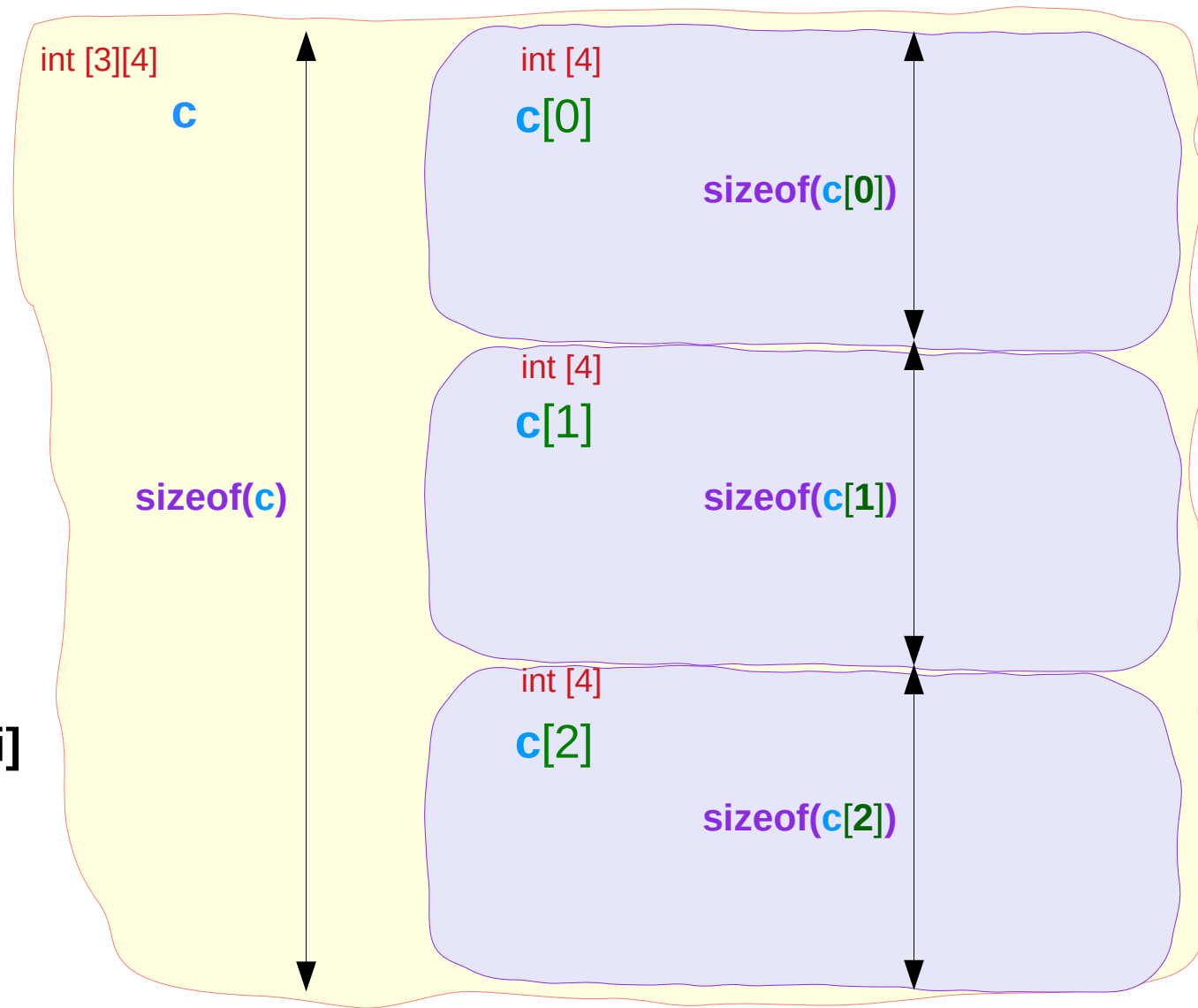
c	2-d array	int [3][4]
c[i]	1-d array	int [4]

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

3-element array c

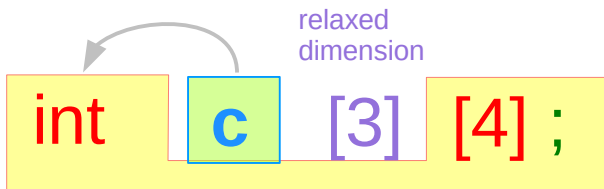
abstract data element **c[i]**

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



c : pointer to a 4-element array

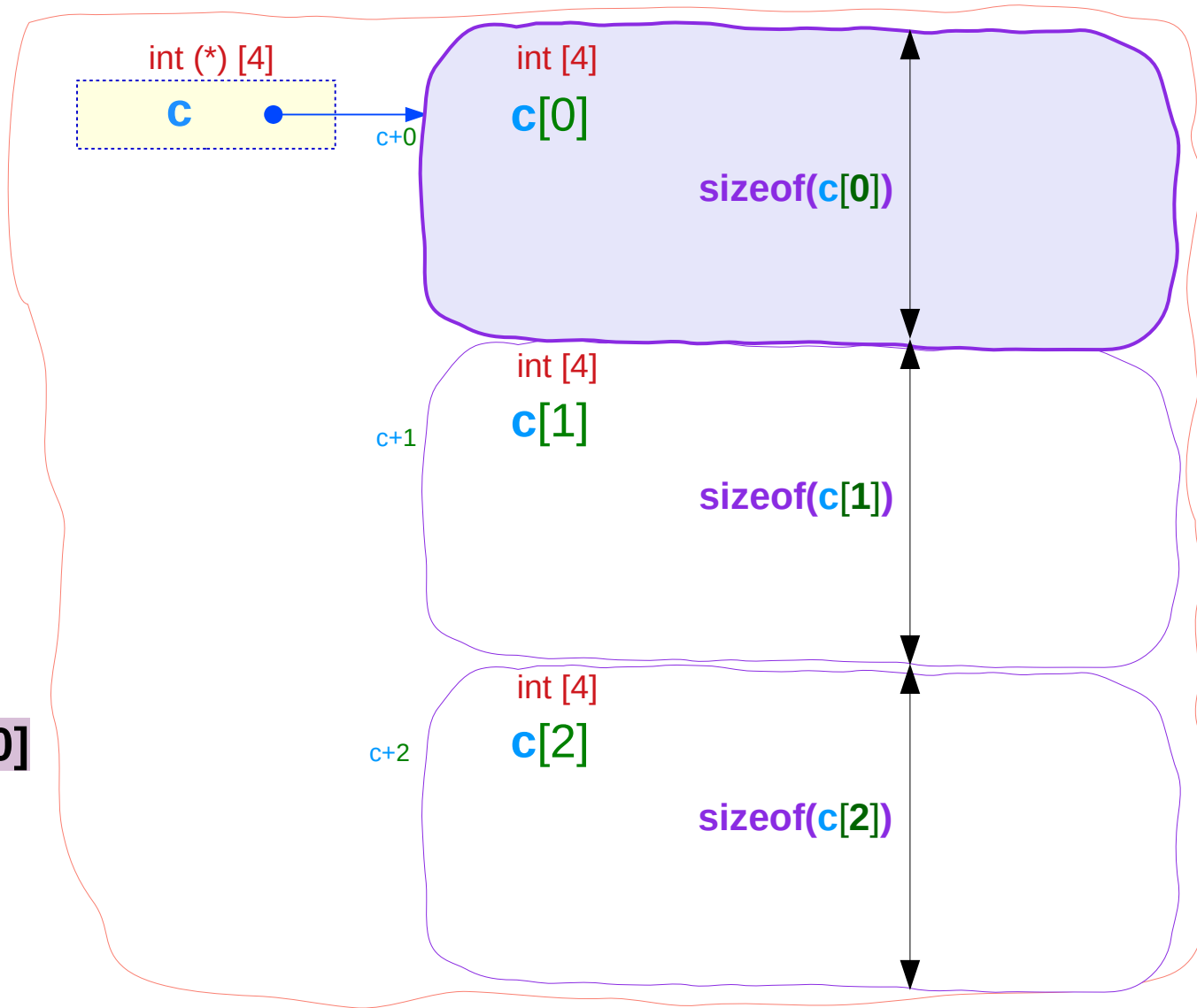
c	1-d array pointer	int (*)[4]
c[i]	1-d array	int [4]



pointer c

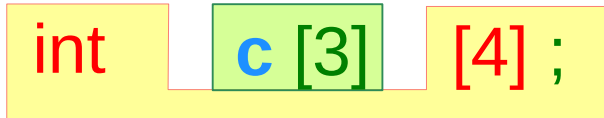
abstract data element **c[0]**

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



c[i] : 4-element array

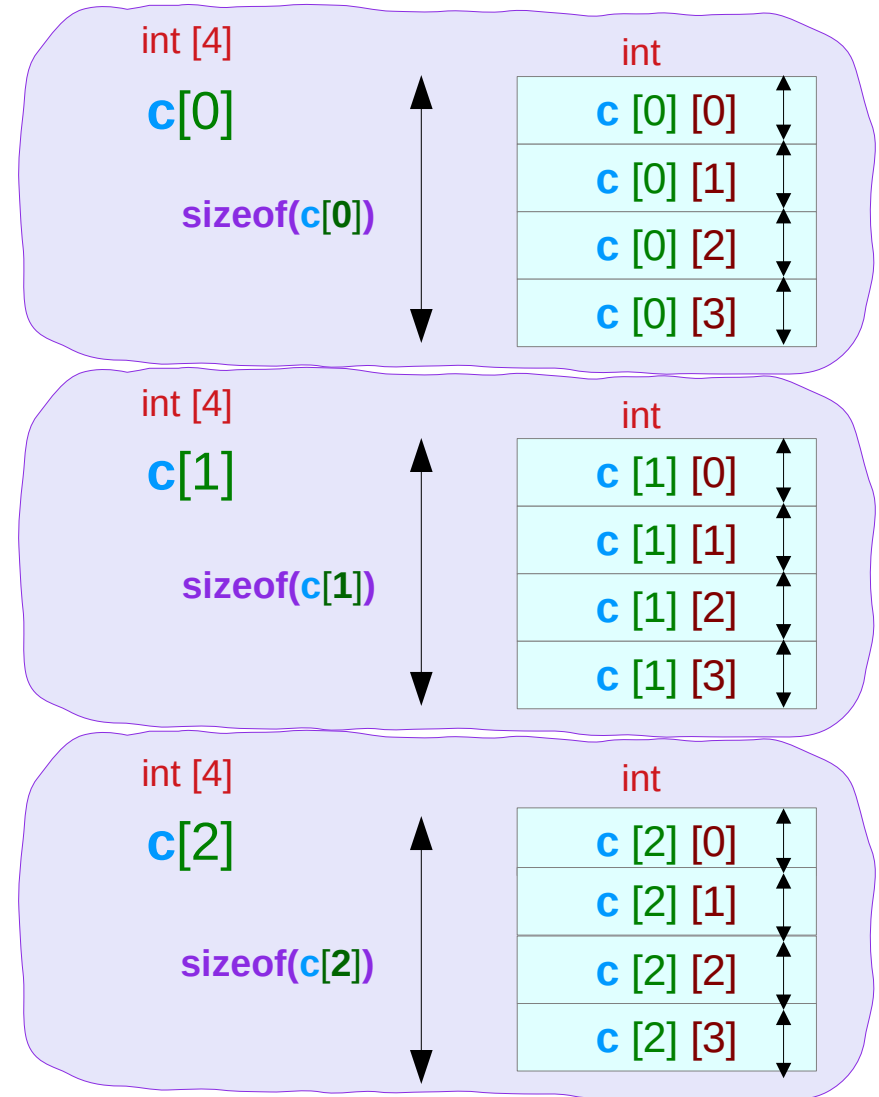
c[i]	1-d array	int [4]
c[i][j]	0-d array	int



4-element array c[i]

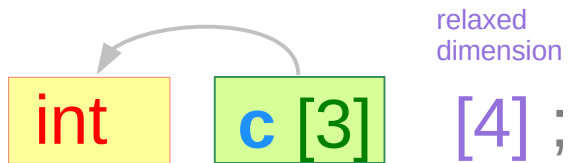
primitive data element c[i][j]

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



c[i] : pointer to a primitive data

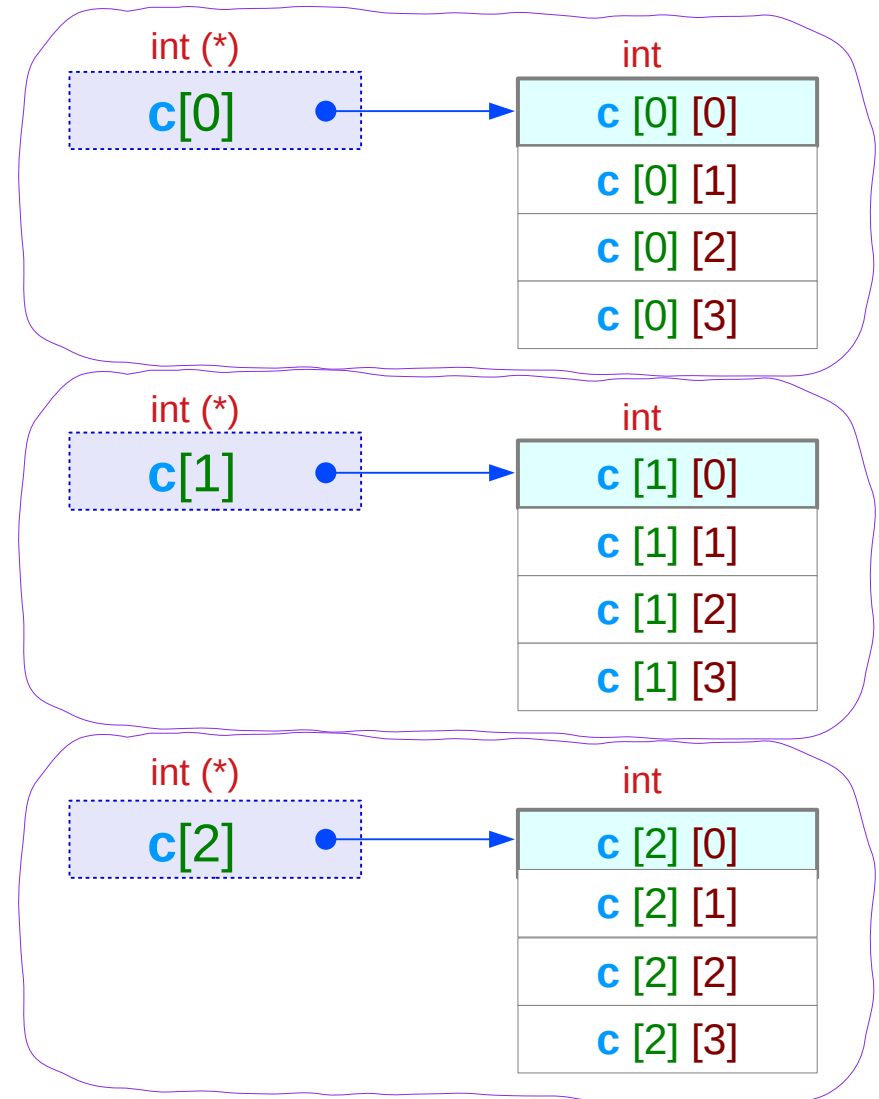
c[i]	0-d array pointer	int (*)
c[i][j]	0-d array	int



pointer c[i]

primitive data element c[i][0]

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



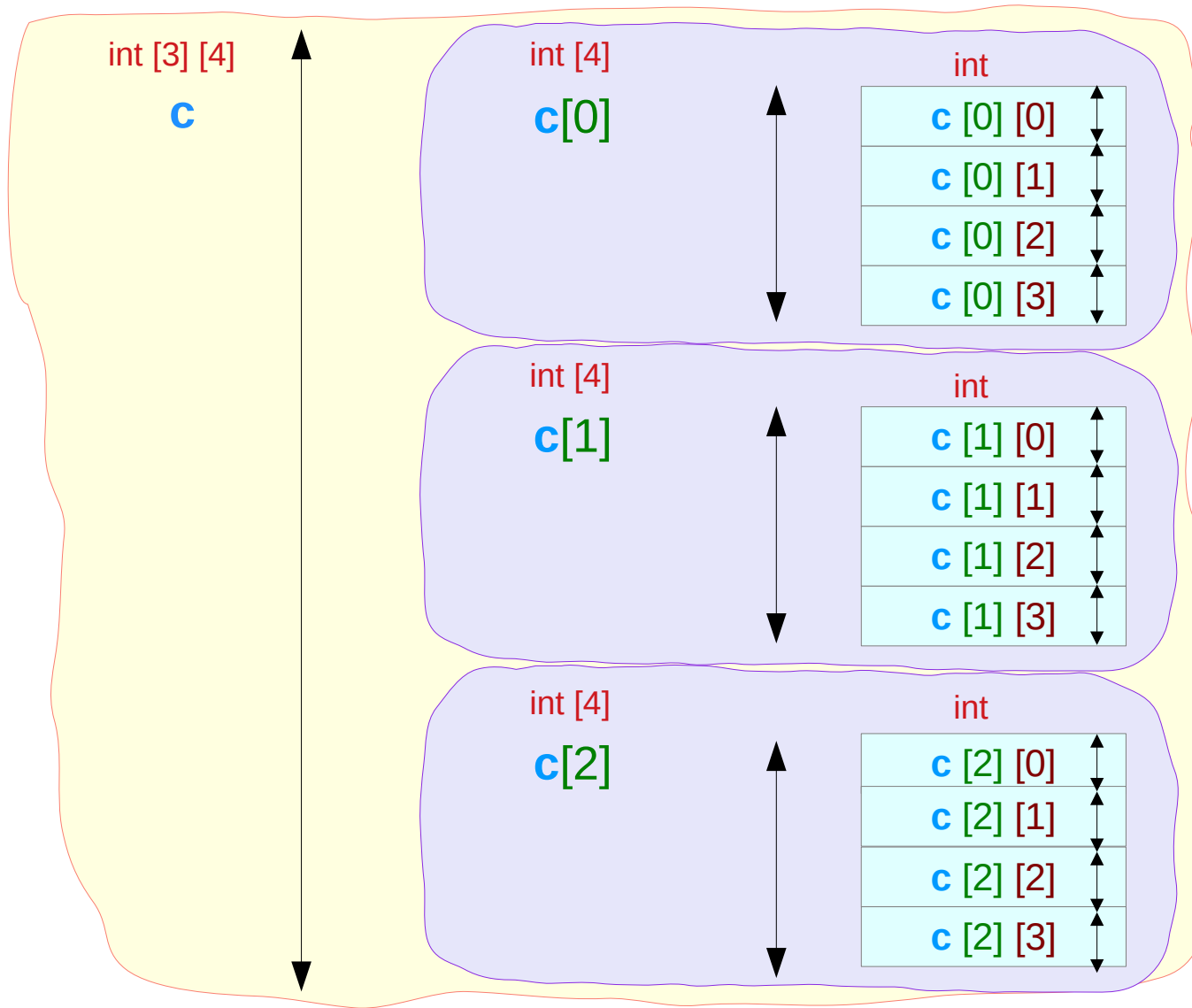
Recursive data view

<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]`



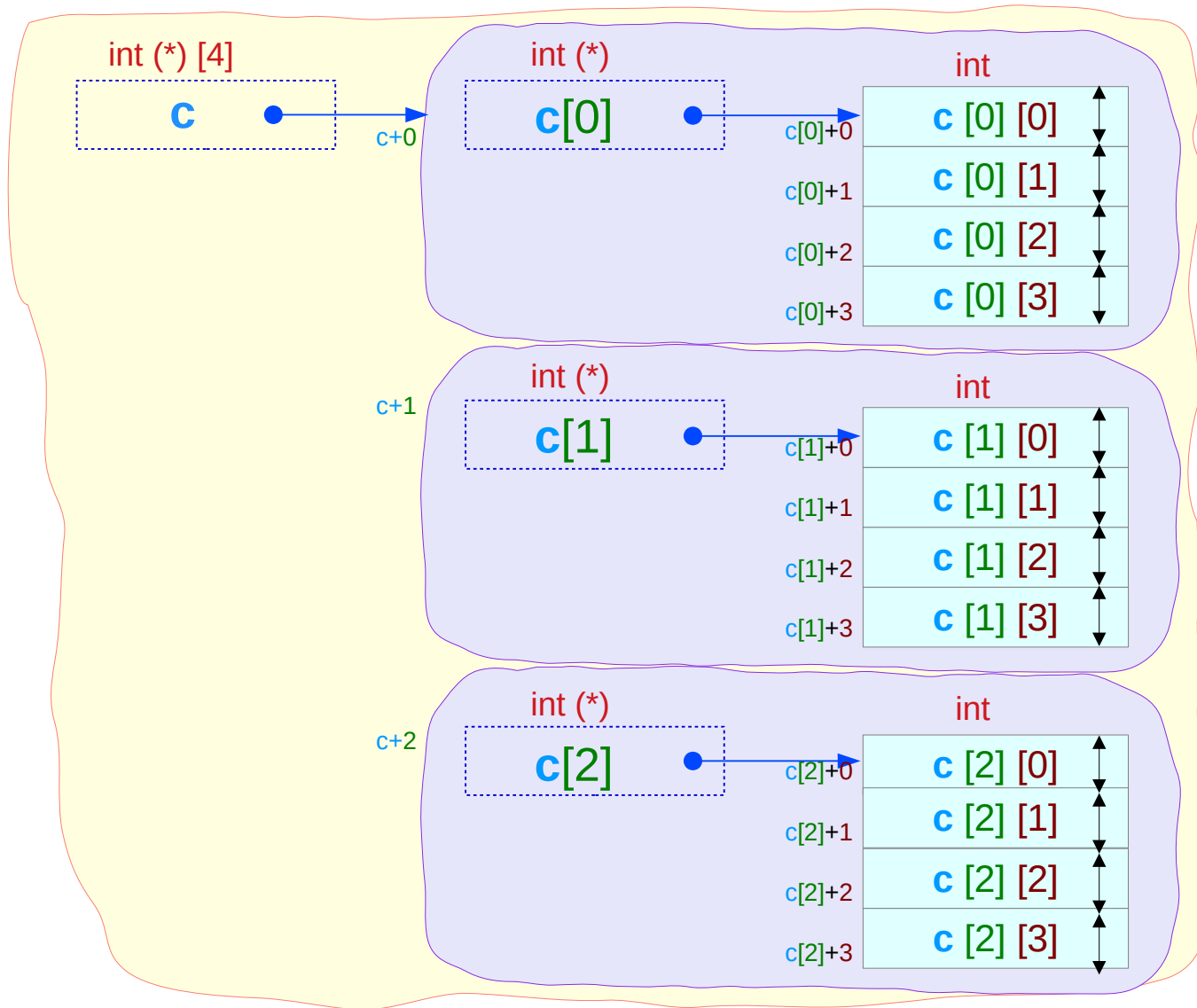
Pointer view

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] ;**

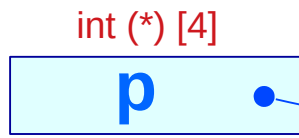
$$\begin{aligned}
 v(\mathbf{c}) &= v(\mathbf{c}[0]) = v(\&\mathbf{c}[0][0]) \\
 v(\mathbf{c}[1]) &= v(\&\mathbf{c}[1][0]) \\
 v(\mathbf{c}[2]) &= v(\&\mathbf{c}[2][0])
 \end{aligned}$$

$v \equiv$ value



1-d array pointer

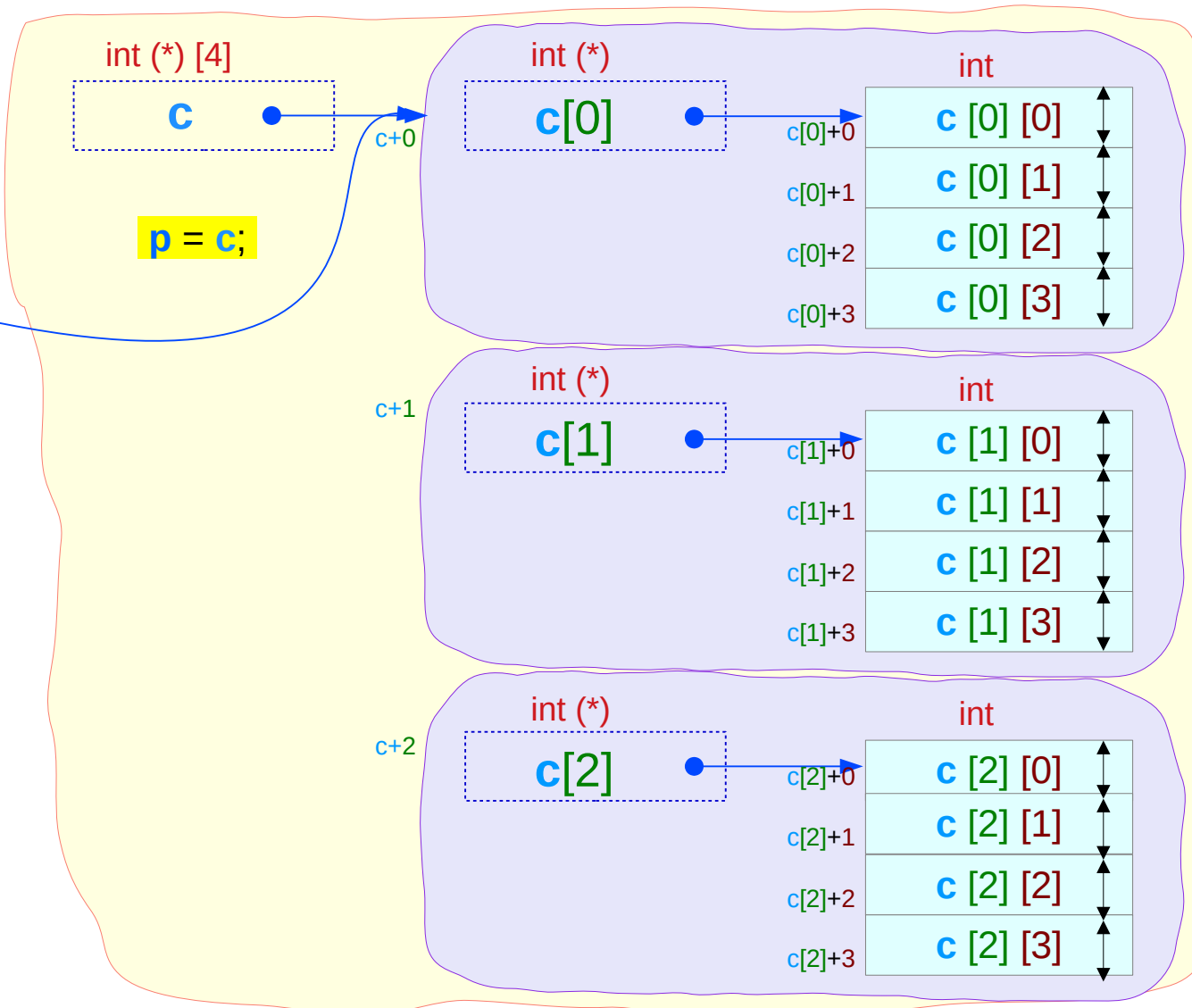
```
int (*p) [4];
```



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

$$\begin{aligned} v(c) &= v(c[0]) = v(\&c[0][0]) \\ v(c[1]) &= v(\&c[1][0]) \\ v(c[2]) &= v(\&c[2][0]) \end{aligned}$$

$v \equiv$ value

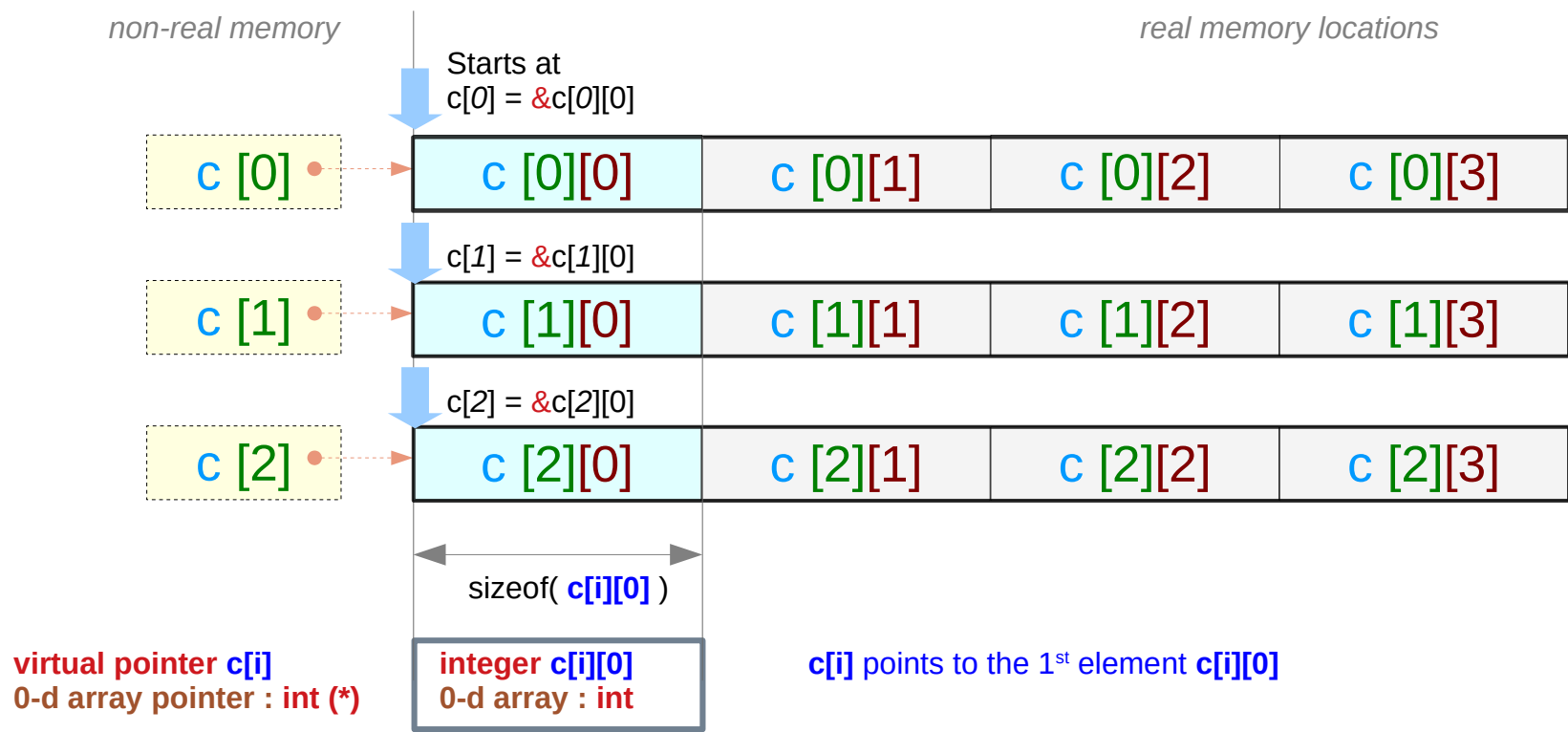


Pointer $c[i]$ and integer $c[i][0]$

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

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

0-d array pointer

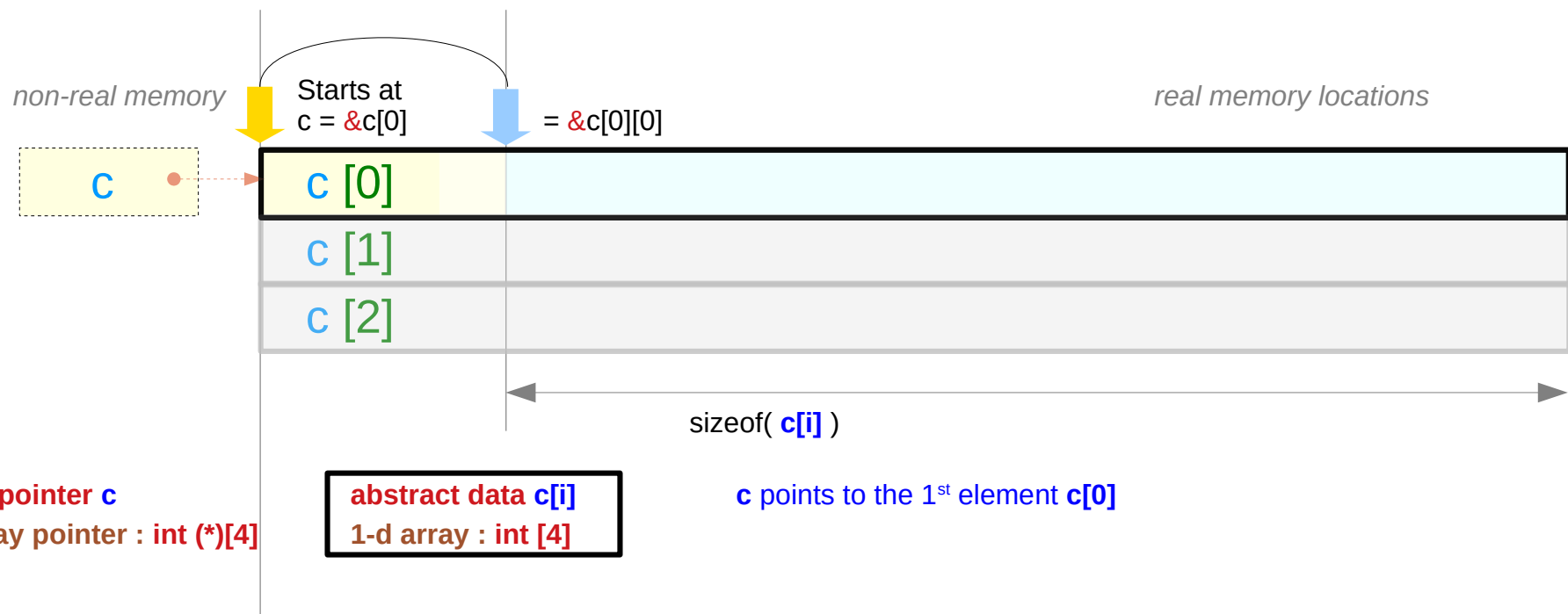


Pointer **c** and abstract data **c[i]**

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

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

1-d array pointer
1-d array

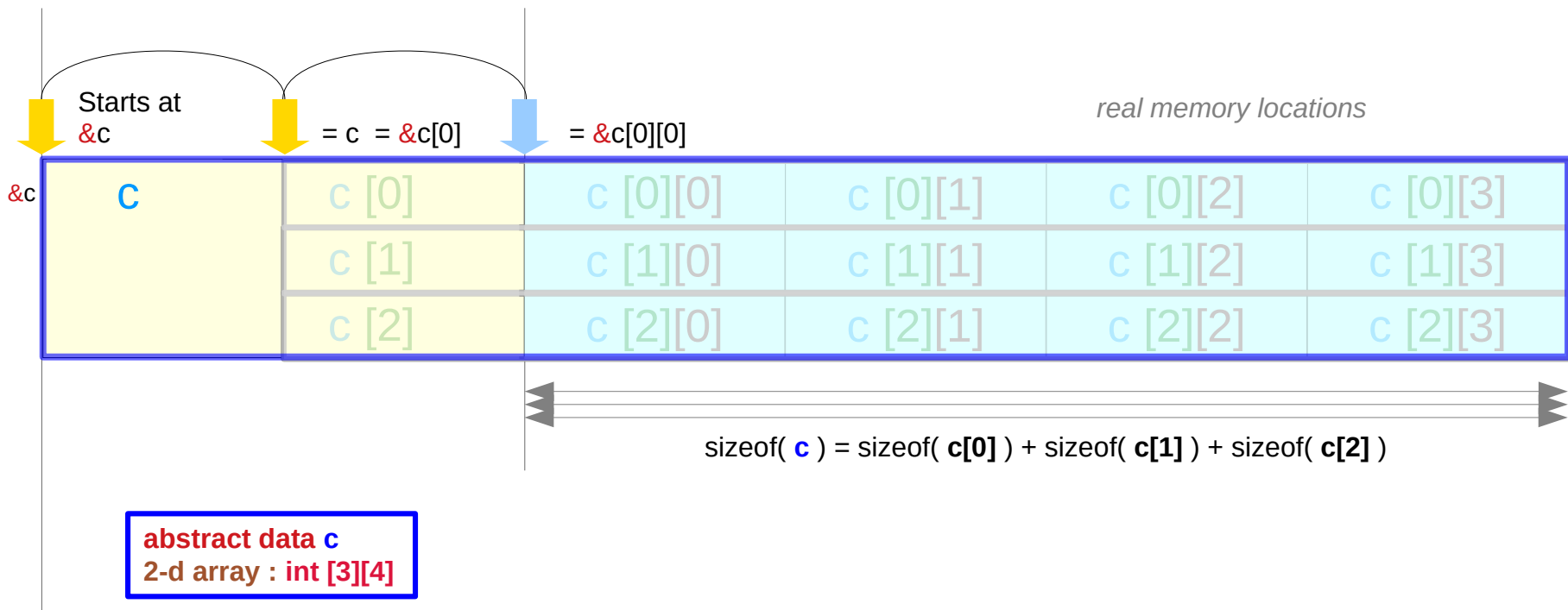


Abstract data **c**

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

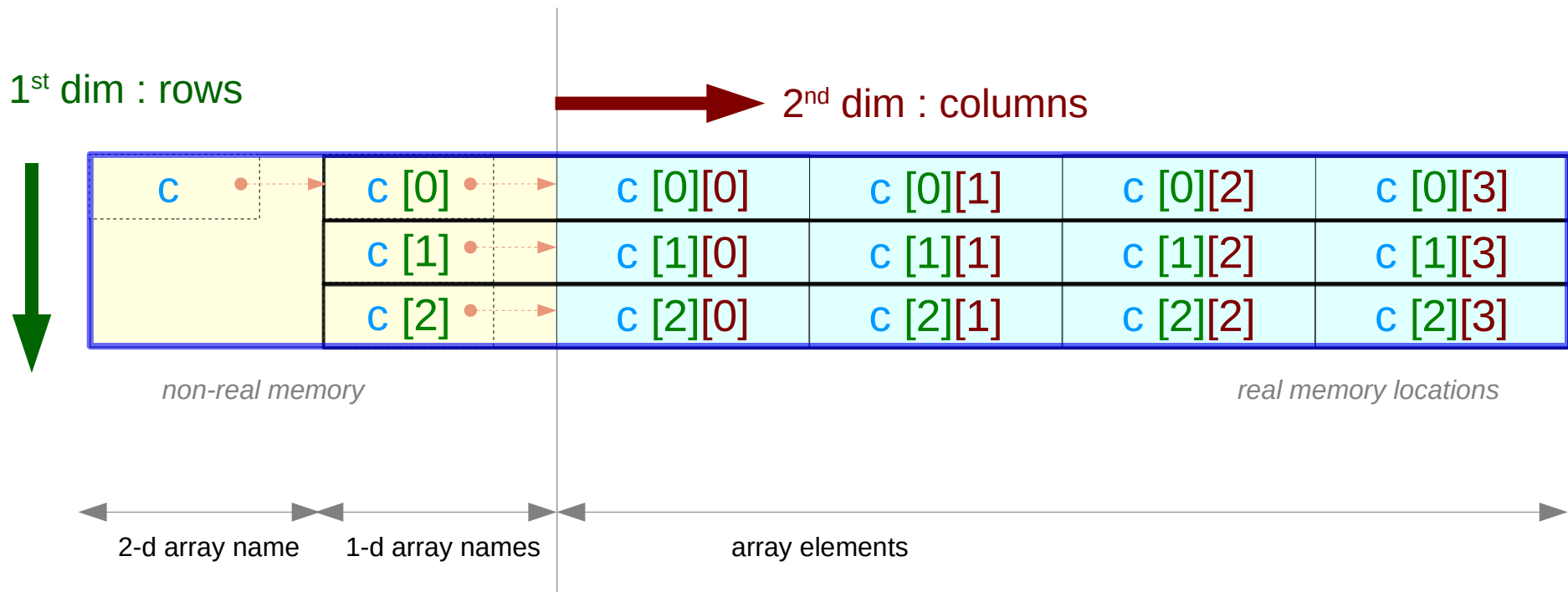
abstract data **c**: $\text{sizeof}(\mathbf{c}) = 3 * \text{sizeof}(\mathbf{c}[\mathbf{i}])$

2-d array



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

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



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];
```

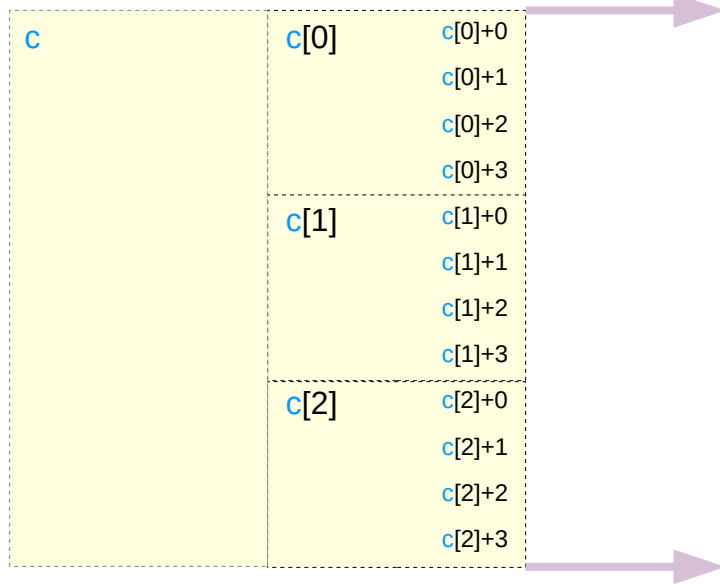
c	c[0]	c[0]+0	c[0][0]
		c[0]+1	c[0][1]
		c[0]+2	c[0][2]
		c[0]+3	c[0][3]
	c[1]	c[1]+0	c[1][0]
		c[1]+1	c[1][1]
		c[1]+2	c[1][2]
		c[1]+3	c[1][3]
	c[2]	c[2]+0	c[2][0]
		c[2]+1	c[2][1]
		c[2]+2	c[2][2]
		c[2]+3	c[2][3]

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

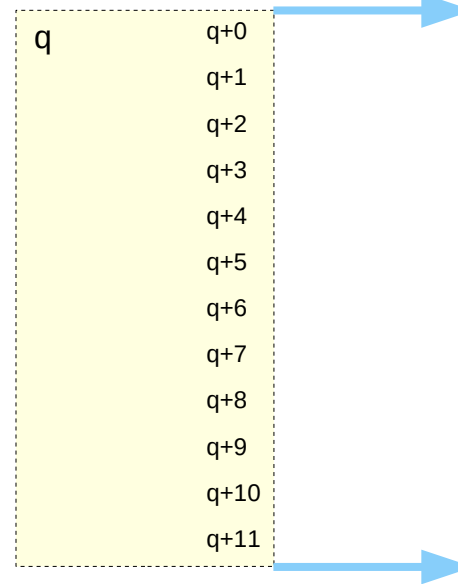
q	q+0	q[0*4+0]
	q+1	q[0*4+1]
	q+2	q[0*4+2]
	q+3	q[0*4+3]
	q+4	q[1*4+0]
	q+5	q[1*4+1]
	q+6	q[1*4+2]
	q+7	q[1*4+3]
	q+8	q[2*4+0]
	q+9	q[2*4+1]
	q+10	q[2*4+2]
	q+11	q[2*4+3]

2-d and 1-d interpretations of linear memories

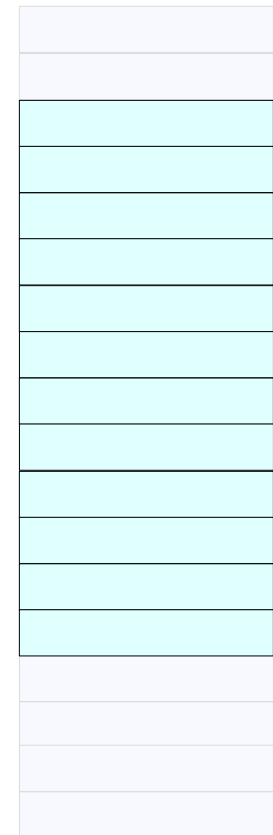
2-d interpretation



1-d interpretation



Physical
Linear
Memory



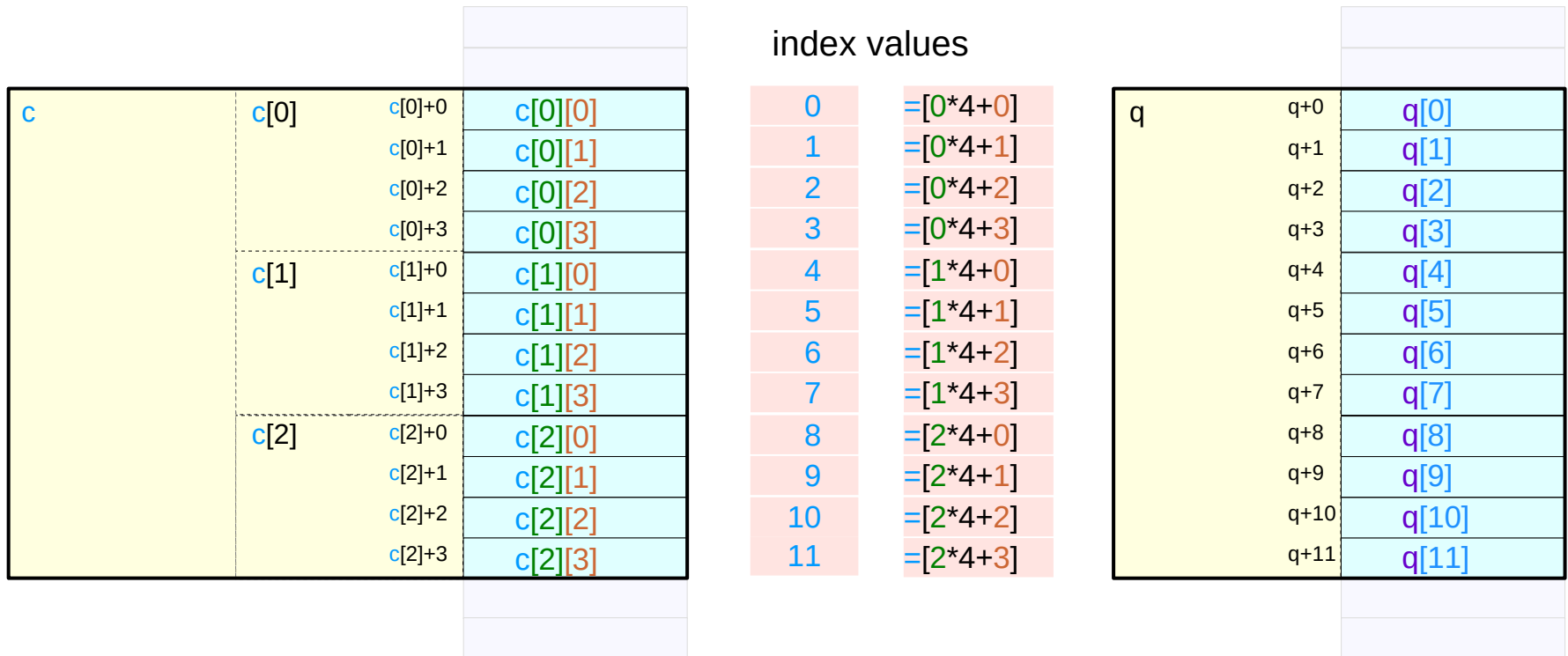
A 2-d array stored as a 1-d array (row major order)

```
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

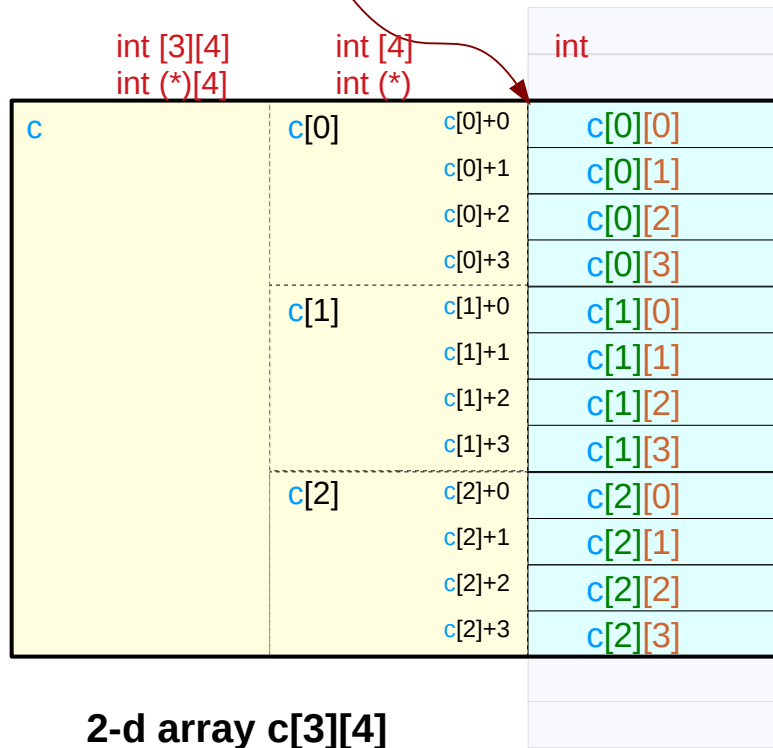
```
int c [3][4];
```

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

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

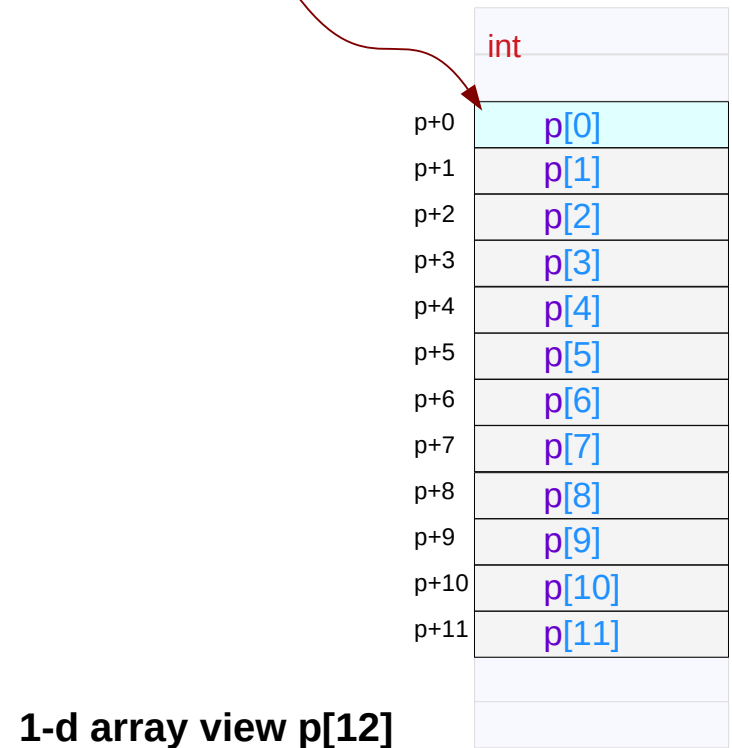
0-d array pointer int (*)

p



0-d array pointer int (*)

p



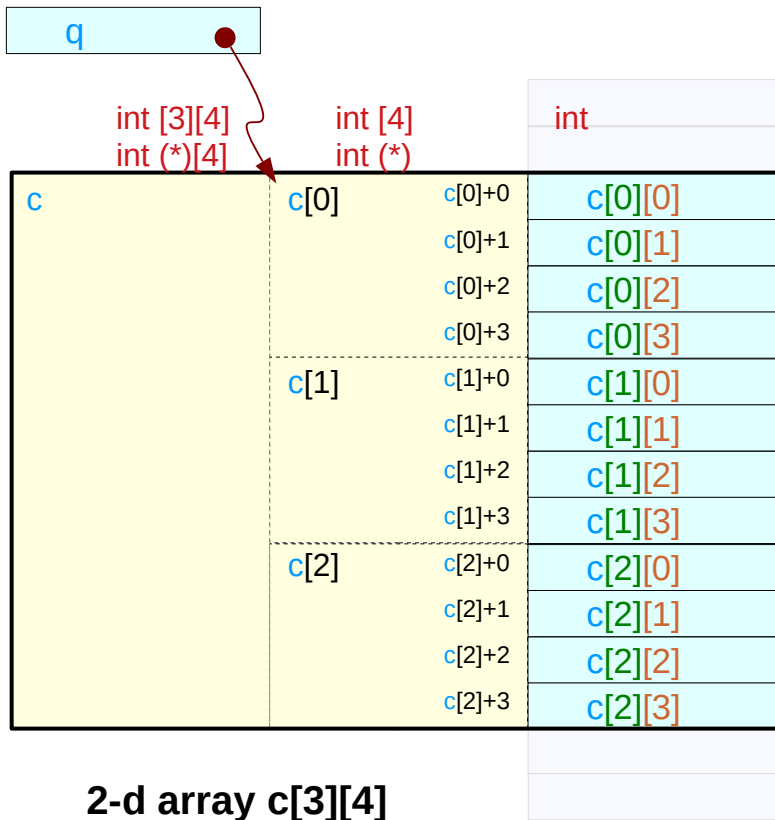
View a 2-d array as another 2-d array

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

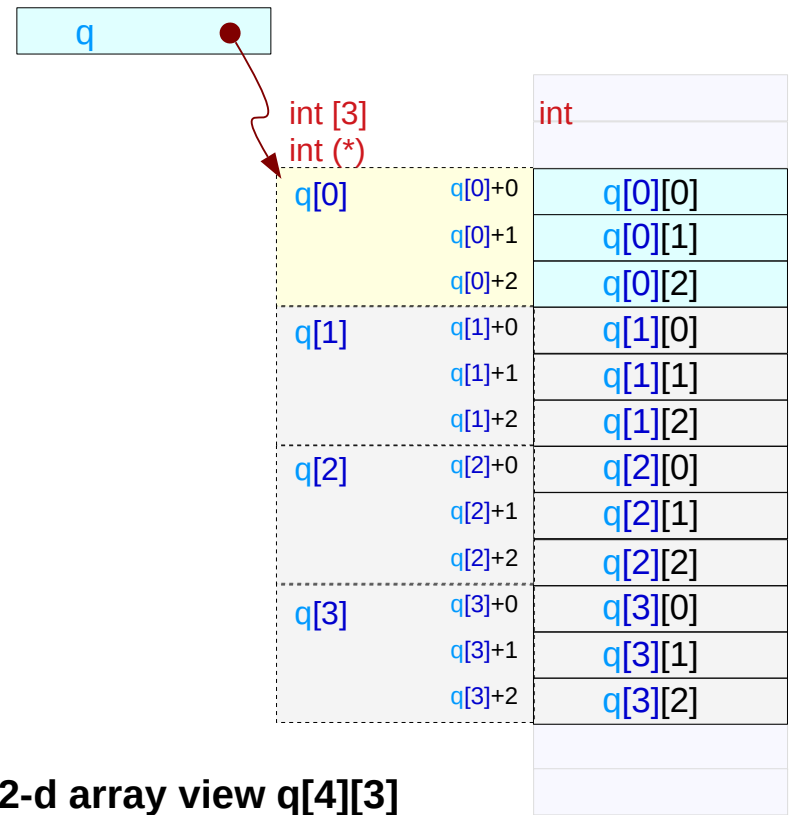
```
int (*q) [3] = (int (*) [3]) c;
```

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

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



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



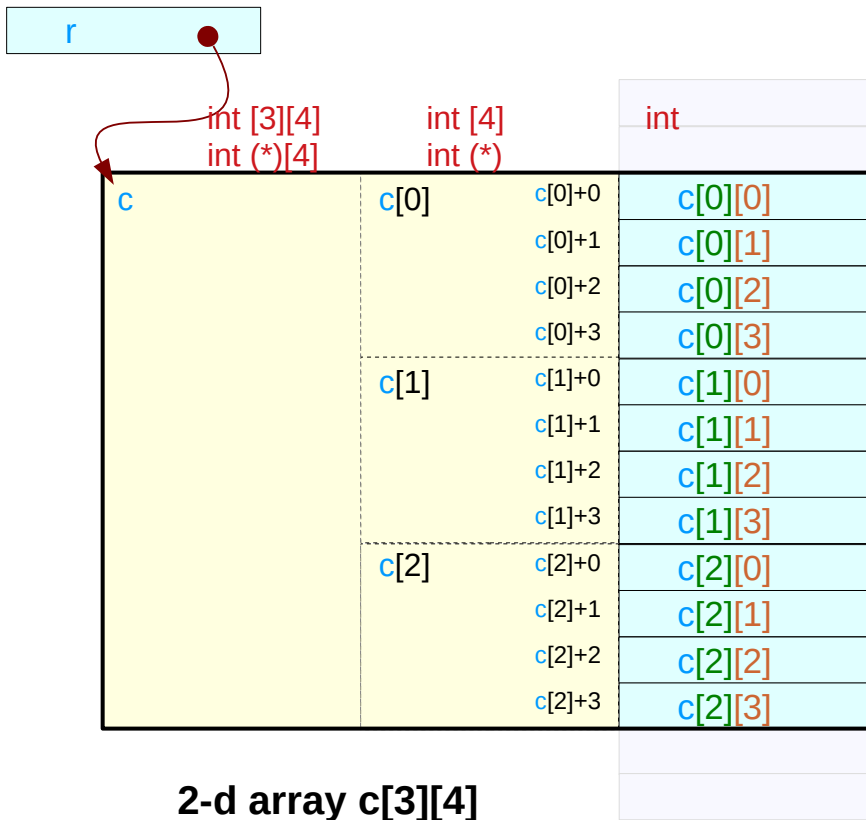
A 2-d array stored as a 1-d array (row major order)

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

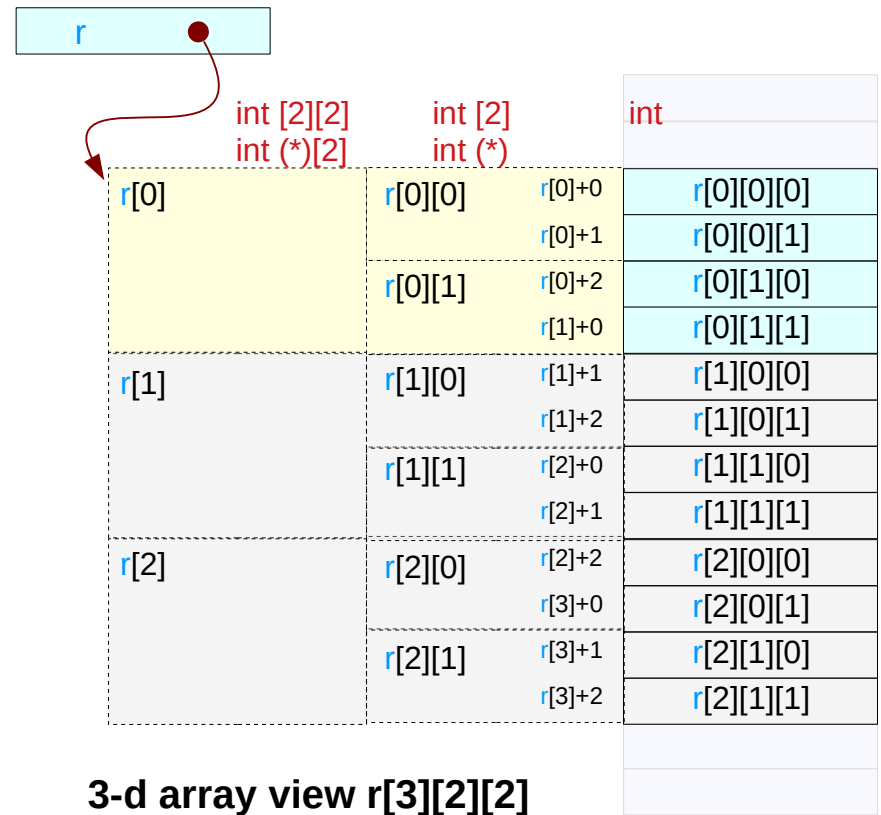
```
int (*r) [2][2] = (int (*) [2][2]) c;
```

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 ]
```

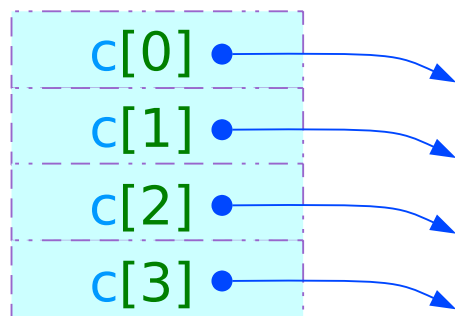
```
*(p+ i*4 + j )
```

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

Address-of & and dereference * operators

Address-of operation

&X

C Expressions

=

value(&X)

Mixed Expressions

rvalue

rvalue

&X

lvalue



&X evaluates the address value of a variable X

& is a mathematical operator (the inverse operator of ***)

$\text{value}(\&X) = \text{value}(\text{value}(\&X)) = \text{value}(\&X) = \&X$

Dereference operation

***X**

C Expressions

=

***value(X)**

Mixed Expressions

lvalue

rvalue



lvalue must be evaluated to rvalue

lvalue



X must be evaluated to an address before de-referencing

Equivalences in address replications

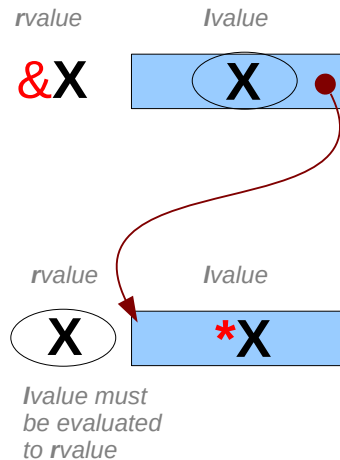
Equivalences in Address Replications

a pointer variable X

$\text{value}(\&X) \equiv \text{value}(X)$

at the pointed address X

$\text{value}(X) \equiv * \text{value}(X)$

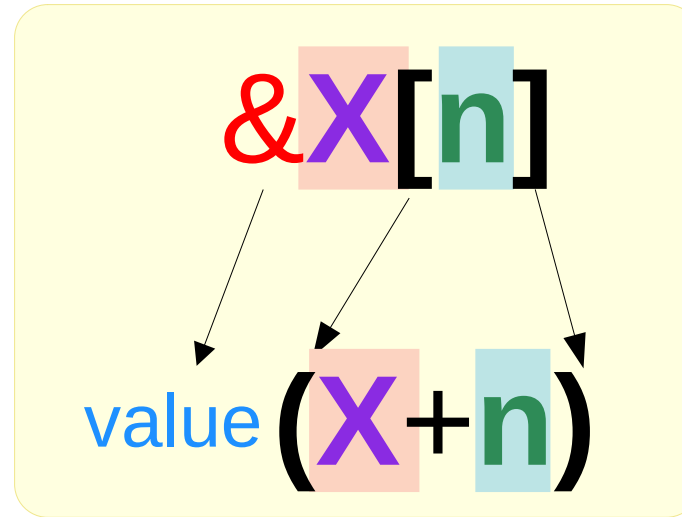
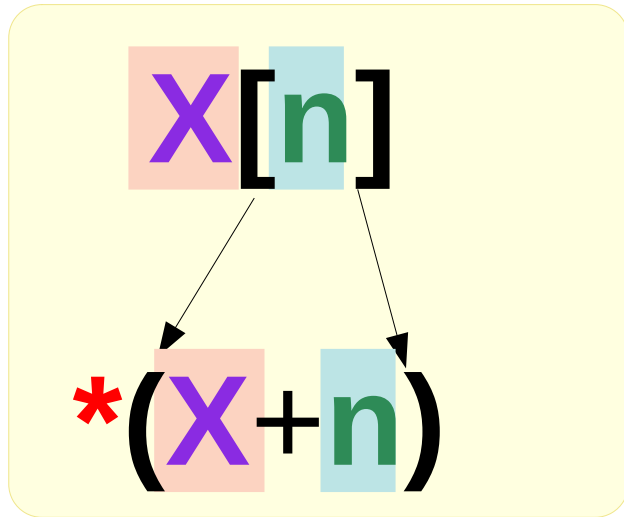


$\&X$ and X have different types
but have the same value

X and $*X$ have different types
but have the same value

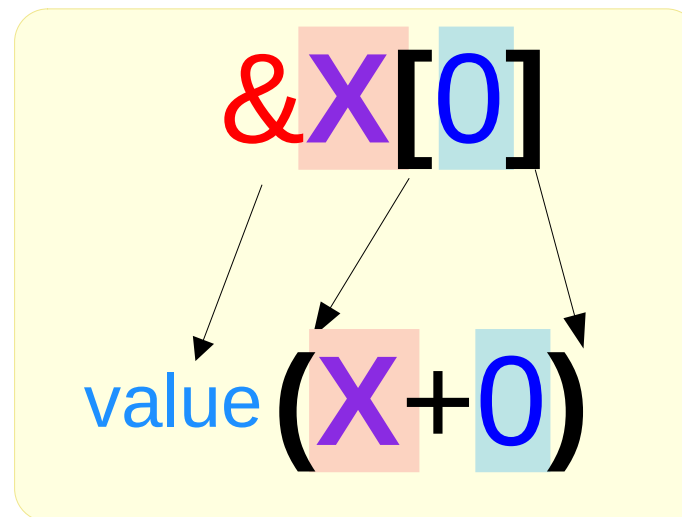
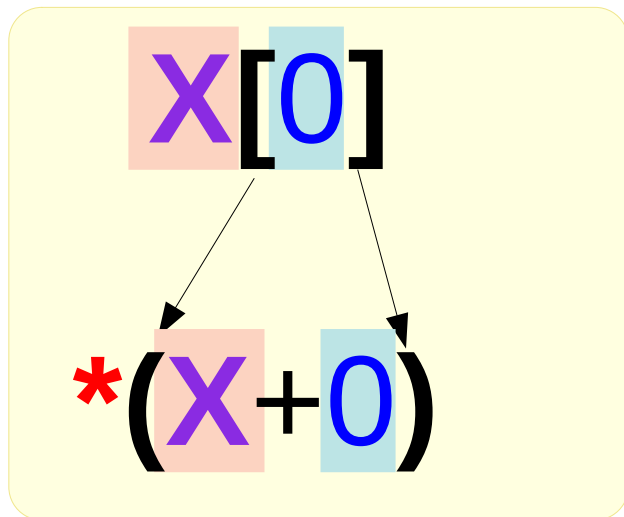
$\text{value}(\&X) \equiv \text{value}(X) \equiv * \text{value}(X)$

Equivalences in array notations



← C operator $\&$
 \neq inverse of $*$

math operator $\&$
= inverse of $*$
↓
 $\text{value}(\&X[n])$
= $\text{value}(\&*(X+n))$
= $\text{value}(X+n)$



← C operator $\&$
 \neq inverse of $*$

math operator $\&$
= inverse of $*$
↓
 $\text{value}(\&X[0])$
= $\text{value}(\&*(X+0))$
= $\text{value}(X+0)$

Pointer Arithmetic

- increment / decrement $++X, --X, X++, X--$
- addition of an integer $X + i$
- subtraction of an integer $X - i$
- subtracting two pointers of the same type $X - Y$
- comparison of pointers $==, !=, >, >=, <, <=$
- adding two pointers are not allowed ~~$X + Y$~~

pointer variables: X, Y

integer compatible variables : i

(**int, short, char, ...**)

Pointer Addition / Subtraction

pointer variables: **X, Y**

primitive variables : **A, B**

`value(X)` is used to avoid confusion
between **pointer additions**
and **arithmetic additions**

X + A the variable **A** must have
X - A **integer** compatible types,
 otherwise **error**

X + Y **error!**

X - Y o.k.

`value(A) = A` primitive variable **A**

`value(X) ≠ X` pointer variable **X**

`value(X + A) = value(X) +a A * sizeof(*X)`

`value(X - A) = value(X) -a A * sizeof(*X)`

`value(X + Y)` **error!**

`value(X - Y) = value(X) -a value(Y)`

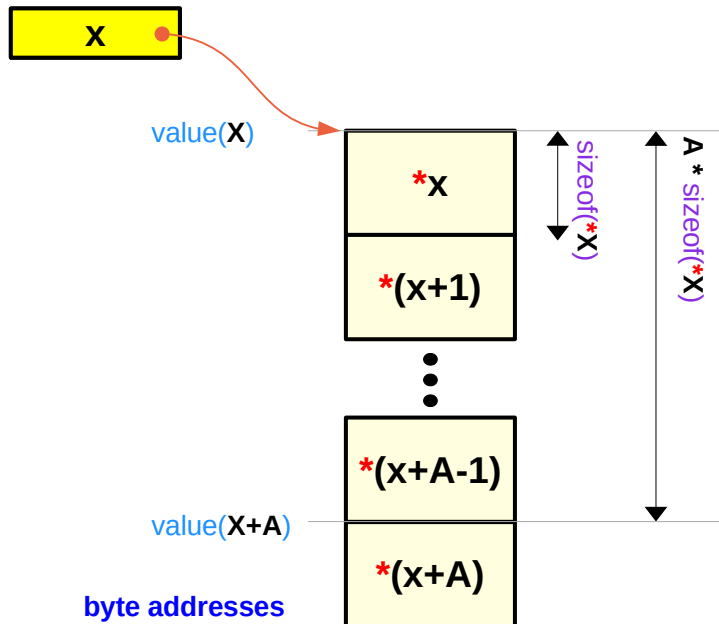
`+a` is used to denote **arithmetic additions**

`-a` is used to denote **arithmetic subtractions**

Pointer Addition / Subtraction

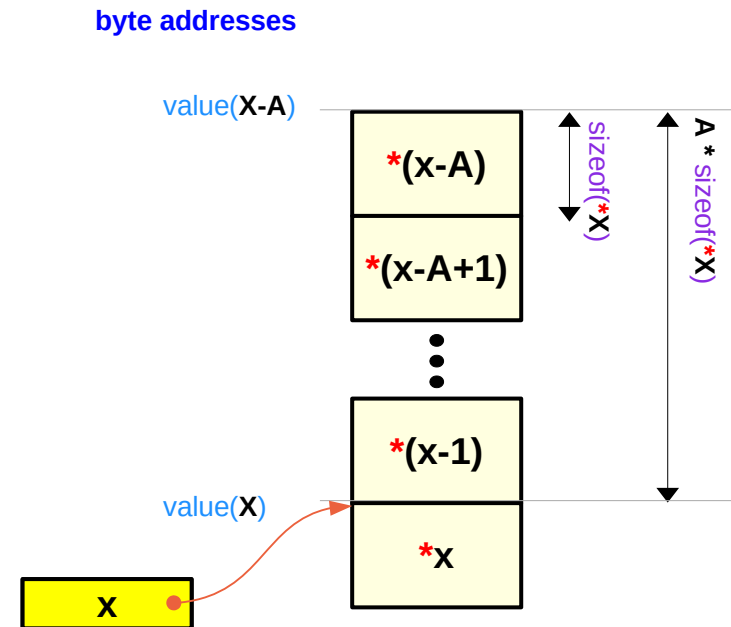
X + A : pointer addition

$$\text{value}(X + A) = \text{value}(X) +_a A * \text{sizeof}(*X)$$



X - A : pointer subtraction

$$\text{value}(X - A) = \text{value}(X) -_a A * \text{sizeof}(*X)$$



Pointer variables vs. primitive variables

pointer variables: **X**

$\text{value}(\mathbf{X}) \neq \mathbf{X}$

integer compatible variables : **A**

$\text{value}(\mathbf{A}) = \mathbf{A}$

X	C expression	<p>+ \rightarrow pointer additions</p> <p>- \rightarrow pointer subtractions</p>	<p>$\text{value}(\mathbf{X} + \mathbf{A}) = \text{value}(\mathbf{X}) +_a \mathbf{A} * \text{sizeof}(*\mathbf{X})$</p> <p>$\text{value}(\mathbf{X} - \mathbf{A}) = \text{value}(\mathbf{X}) -_a \mathbf{A} * \text{sizeof}(*\mathbf{X})$</p>
A	C expression	<p>+ \rightarrow pointer additions</p> <p>- \rightarrow pointer subtractions</p> <p><i>for a <u>pointer</u> type operand</i></p> <p>+ \rightarrow arithmetic additions $+_a$</p> <p>- \rightarrow arithmetic subtractions $-_a$</p> <p><i>for a <u>non pointer</u> type operand</i></p>	<p>$\text{value}(\mathbf{X} + \mathbf{A}) = \text{value}(\mathbf{X}) +_a \mathbf{A} * \text{sizeof}(*\mathbf{X})$</p> <p>$\text{value}(\mathbf{X} - \mathbf{A}) = \text{value}(\mathbf{X}) -_a \mathbf{A} * \text{sizeof}(*\mathbf{X})$</p>
$\text{value}(\mathbf{X})$	Math expression	<p>+ \rightarrow arithmetic additions $+_a$</p> <p>- \rightarrow arithmetic subtractions $-_a$</p>	
$\text{value}(\mathbf{A})$	Math expression	<p>+ \rightarrow arithmetic additions $+_a$</p> <p>- \rightarrow arithmetic subtractions $-_a$</p>	

Recursive application of `value()`

`value(value(A)) = value(A) = A`

`value(value(X)) = value(X) \neq X`

`value(value(X + i))`

`= value(value(X) +a i * sizeof(*X))`

`= value(value(X) +a value(i * sizeof(*X)))`

`= value(X) +a i * sizeof(*X)`

`= value(X) + i * sizeof(*X) in math expression`

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

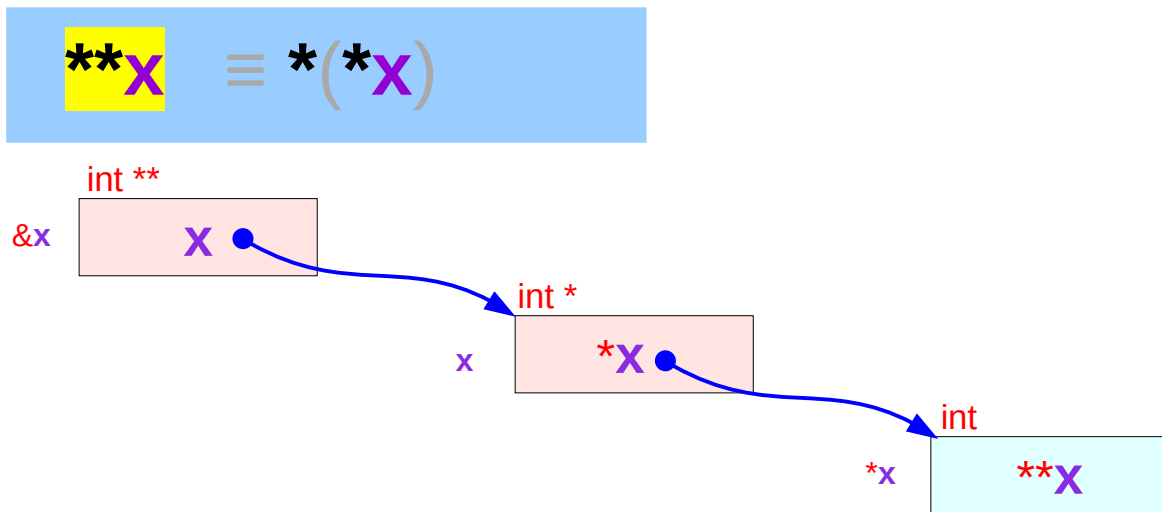
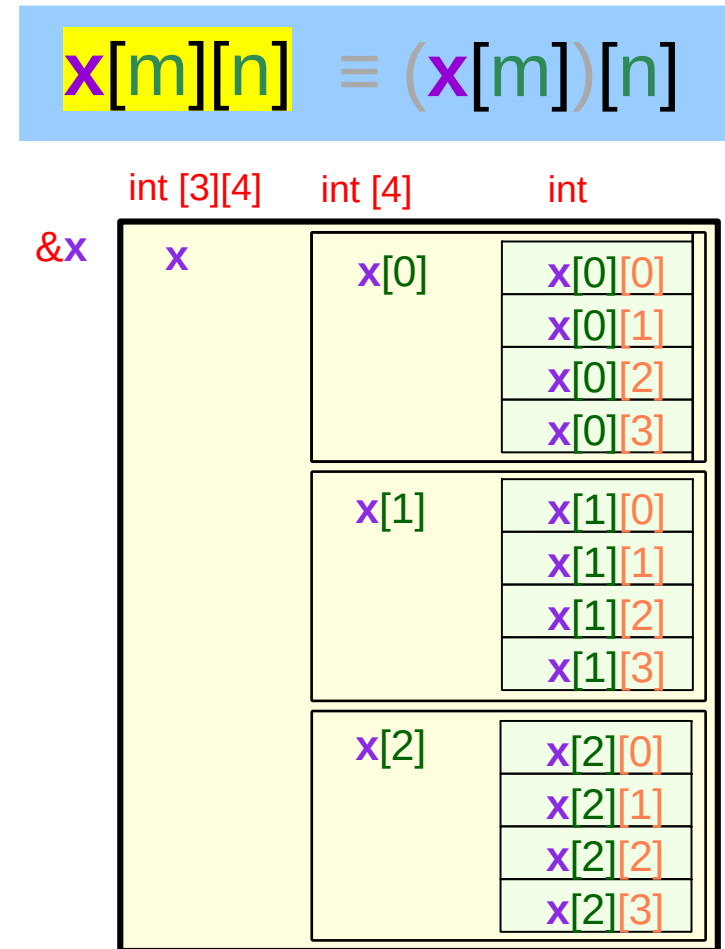
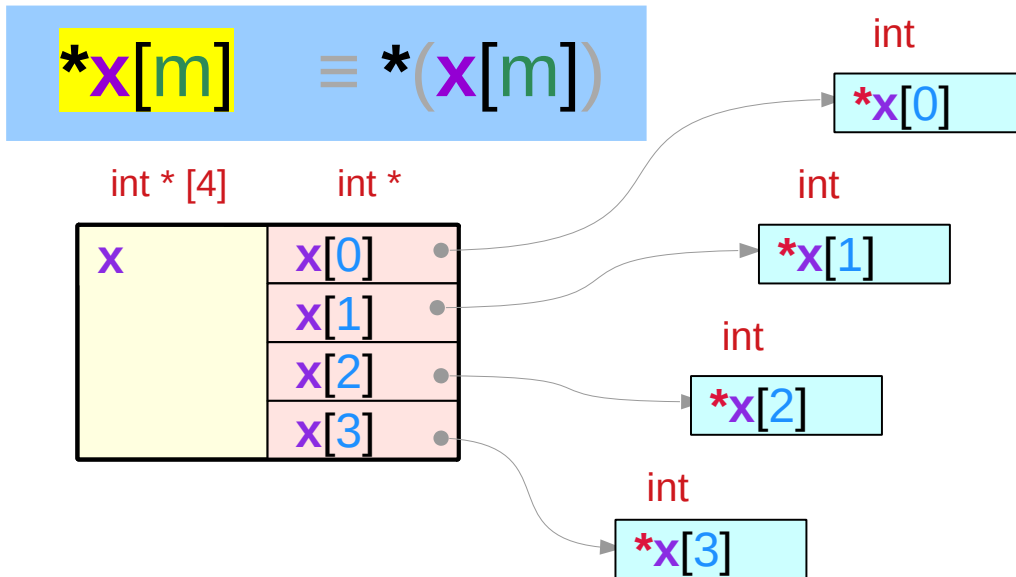
* has **right-to-left** associativity

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

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

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

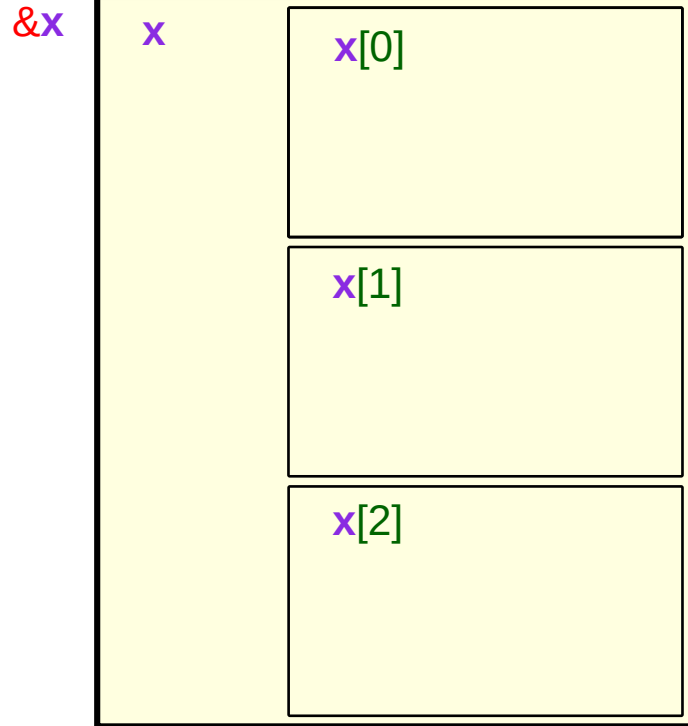
Operator Precedence of * and []



Abstract Data x and $x[i]$

$x[3]$ x has 3 elements

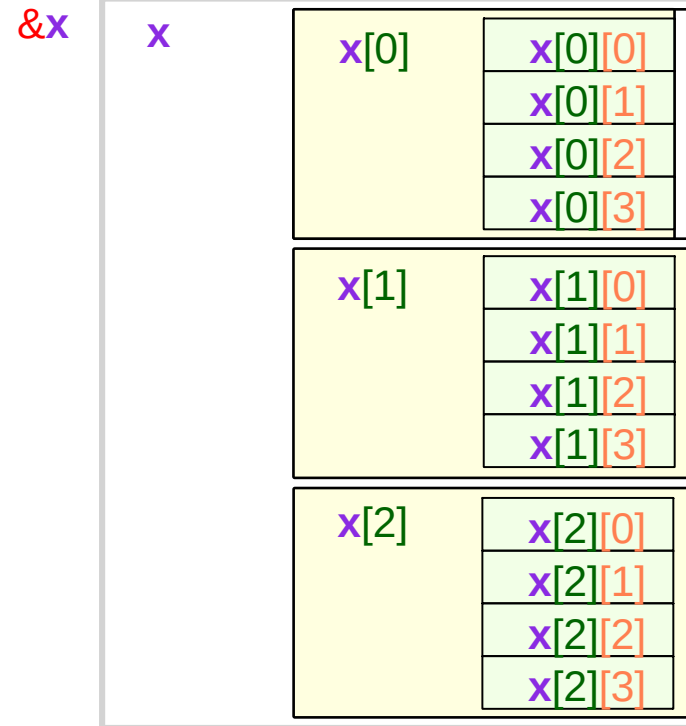
$\text{int } [3][4]$ $\text{int } [4]$



array element $x[i]$

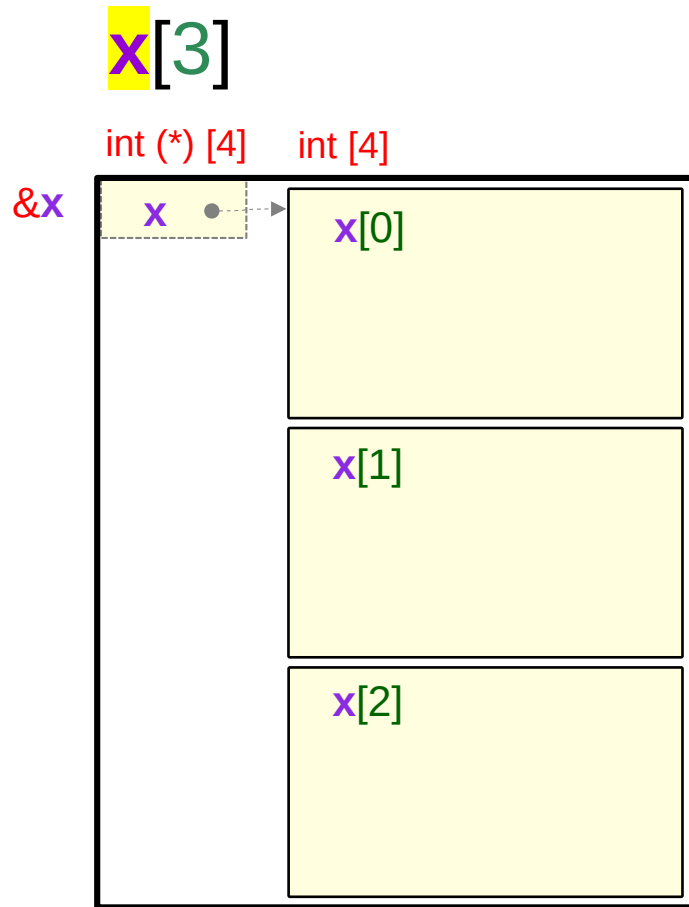
$(x[3])[4]$ each $x[i]$ has 4 elements

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

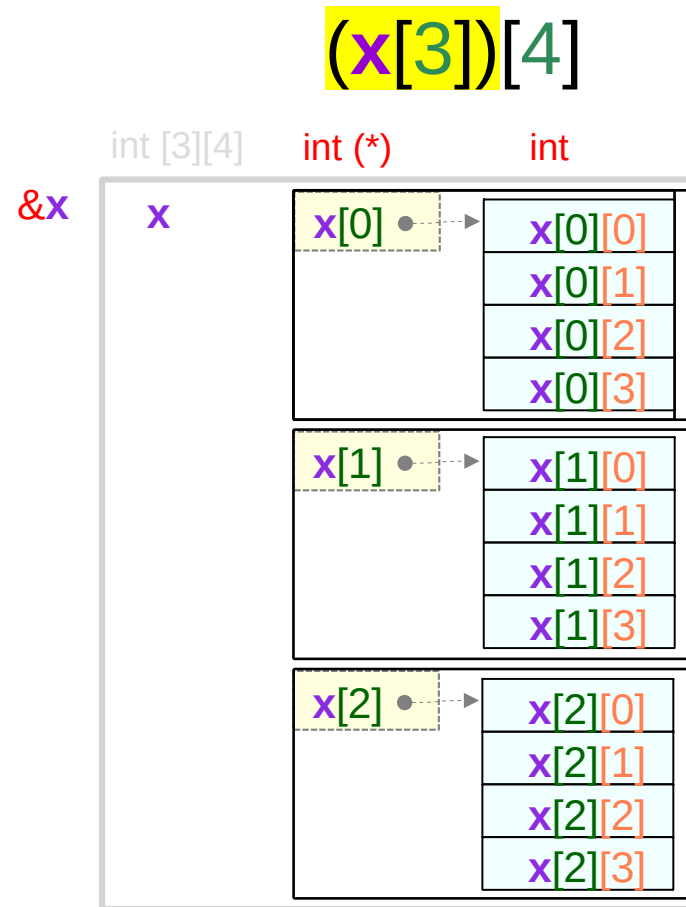


array name $x[i][j]$

Virtual Pointers x and $x[i]$

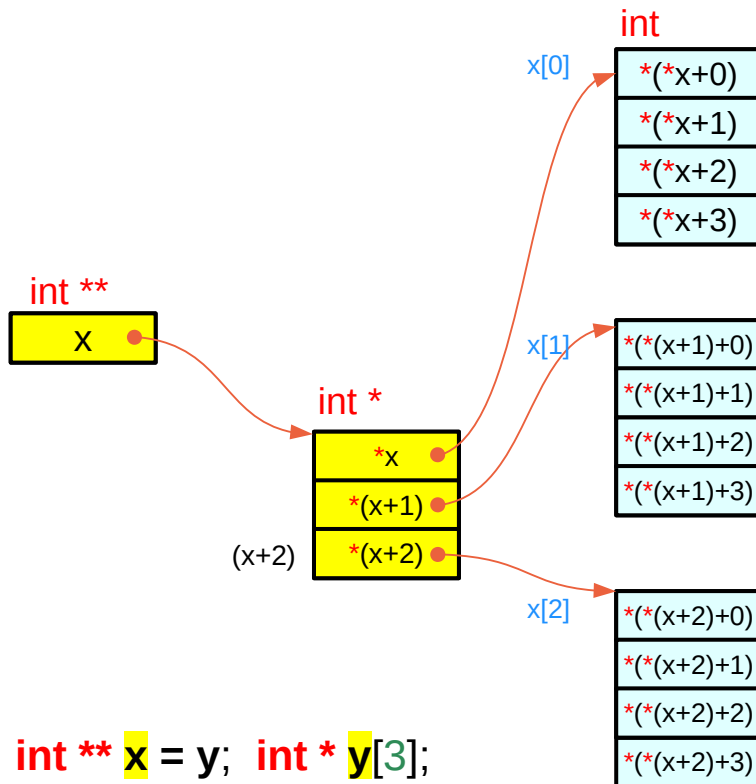


array name x virtual pointer
array element $x[i]$ abstract data



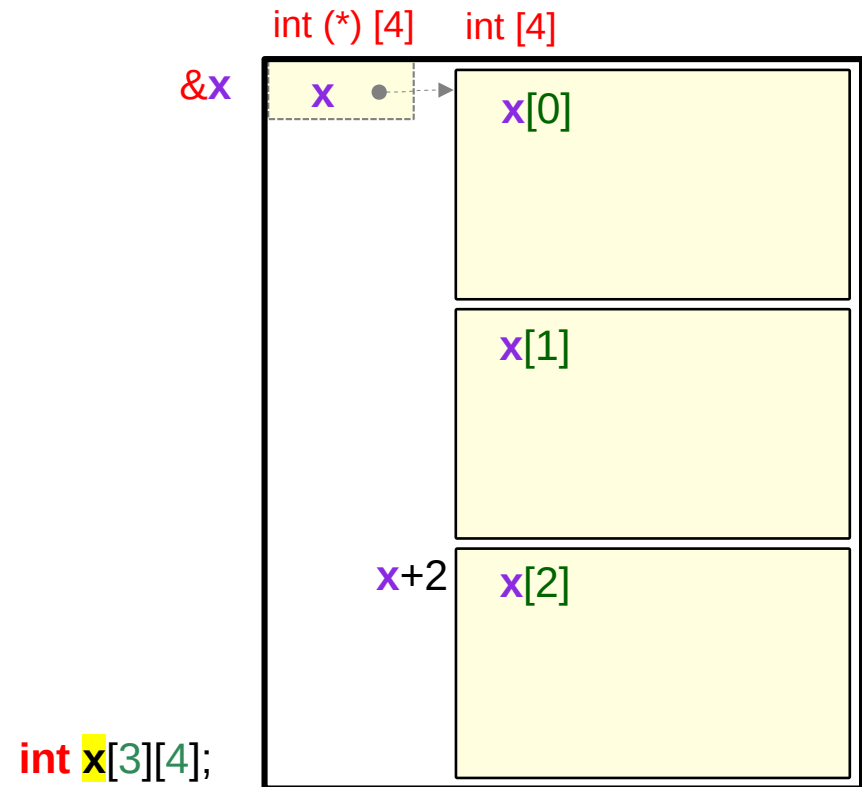
array name $x[i]$ virtual pointer
array element $x[i][j]$ primitive data

Virtual Pointers x and $x[i]$



Pointer Array Approach

$$\text{value}(x + i) = \text{value}(x) +_a i * \text{sizeof}(*x)$$



Array Pointer Approach

$$\text{value}(x + i) = \text{value}(x) +_a i * \text{sizeof}(x[0])$$

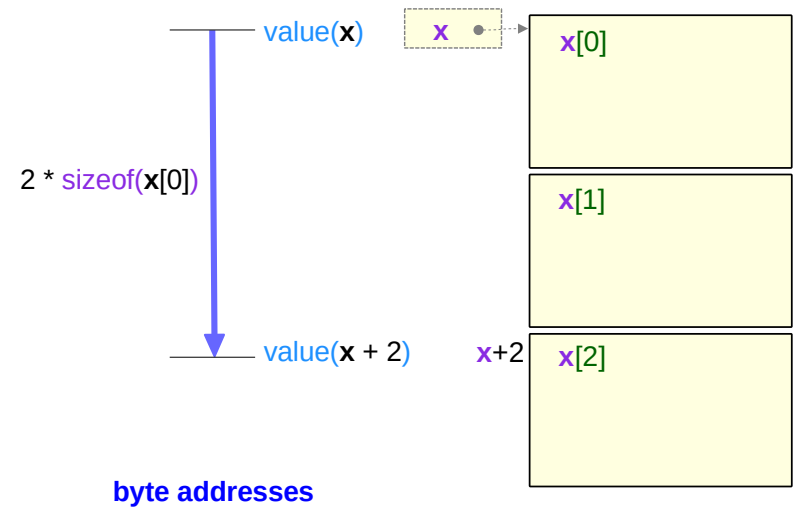
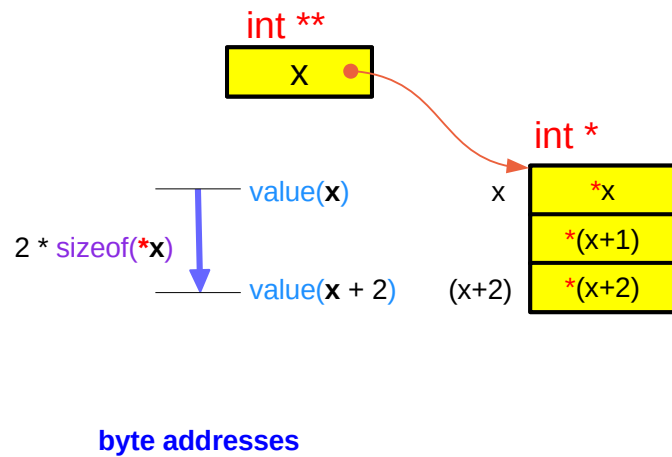
Base and offset in byte addresses

pointer variable **X**

$\text{value}(\mathbf{x} + \mathbf{i})$ pointer addition +
= $\text{value}(\mathbf{x}) +_{\mathbf{a}} \mathbf{i} * \text{sizeof}(*\mathbf{x})$ arithmetic addition +_a

array variable **X**

$\text{value}(\mathbf{x} + \mathbf{i})$ pointer addition +
= $\text{value}(\mathbf{x}) +_{\mathbf{a}} \mathbf{i} * \text{sizeof}(\mathbf{x}[0])$ arithmetic addition +_a



`int ** x = y; int * y[3];`

`int x[3][4];`

Base and offset in byte address (1)

pointer variable \mathbf{X}

$\text{value}(\mathbf{x} + \mathbf{i})$ pointer addition +
= $\text{value}(\mathbf{x}) +_{\mathbf{a}} \mathbf{i} * \text{sizeof}(*\mathbf{x})$ arithmetic addition $+_{\mathbf{a}}$
= $\text{value}(\mathbf{x}) + \mathbf{i} * \text{sizeof}(*\mathbf{x})$ in math expression

$\text{value}(\mathbf{X})$ may be involved
in only arithmetic additions / subtractions,
therefore the arithmetic addition $+_{\mathbf{a}}$
can be replaced with the ordinary $+$

= $\text{value}(\mathbf{x}) +_{\mathbf{a}} \mathbf{i} * \text{sz}$, $\text{sz} = \text{sizeof}(*\mathbf{x})$
= $\text{value}(\mathbf{x} + \mathbf{i})_{\text{sz}}$

array variable \mathbf{X}

$\text{value}(\&\mathbf{x}[\mathbf{i}])$ pointer addition +
= $\text{value}(\mathbf{x}) +_{\mathbf{a}} \mathbf{i} * \text{sizeof}(\mathbf{x}[0])$ arithmetic addition $+_{\mathbf{a}}$
= $\text{value}(\mathbf{x}) + \mathbf{i} * \text{sizeof}(\mathbf{x}[0])$ in math expression

= $\text{value}(\mathbf{x}) +_{\mathbf{a}} \mathbf{i} * \text{sz}$, $\text{sz} = \text{sizeof}(\mathbf{x}[0])$
= $\text{value}(\mathbf{x} + \mathbf{i})_{\text{sz}}$

$\text{int} ** \mathbf{x} = \mathbf{y}; \text{int} * \mathbf{y}[3];$

$\text{int} \mathbf{x}[3][4];$

Base and offset in byte address (2)

pointer variable \mathbf{x}

$\text{value}(*(\mathbf{x} + \mathbf{i}) + \mathbf{j})$ value(value(x)) = value(x)
= $\text{value}(*(\mathbf{x} + \mathbf{i})) + \mathbf{j} * \text{sizeof}(**\mathbf{x})$
= $*\text{value}(\mathbf{x} + \mathbf{i}) + \mathbf{j} * \text{sizeof}(**\mathbf{x})$
= $*(\text{value}(\mathbf{x}) + \mathbf{i} * \text{sizeof}(*\mathbf{x})) + \mathbf{j} * \text{sizeof}(**\mathbf{x})$

= $*(\text{value}(\mathbf{x}) + \mathbf{i} * \text{sizeof}(*\mathbf{x})) + \mathbf{j} * \text{sizeof}(**\mathbf{x})$
in math expressions

= $*(\text{value}(\mathbf{x}) + \mathbf{i} * \mathbf{s1}) + \mathbf{j} * \mathbf{s2}$
= $(*(\mathbf{x} + \mathbf{i})_{\mathbf{s1}} + \mathbf{j})_{\mathbf{s2}}$
 $\mathbf{s1} = \text{sizeof}(*\mathbf{x}), \mathbf{s2} = \text{sizeof}(**\mathbf{x})$

$\text{int } **\mathbf{x} = \mathbf{y}; \text{ int } * \mathbf{y}[3];$

array variable \mathbf{x}

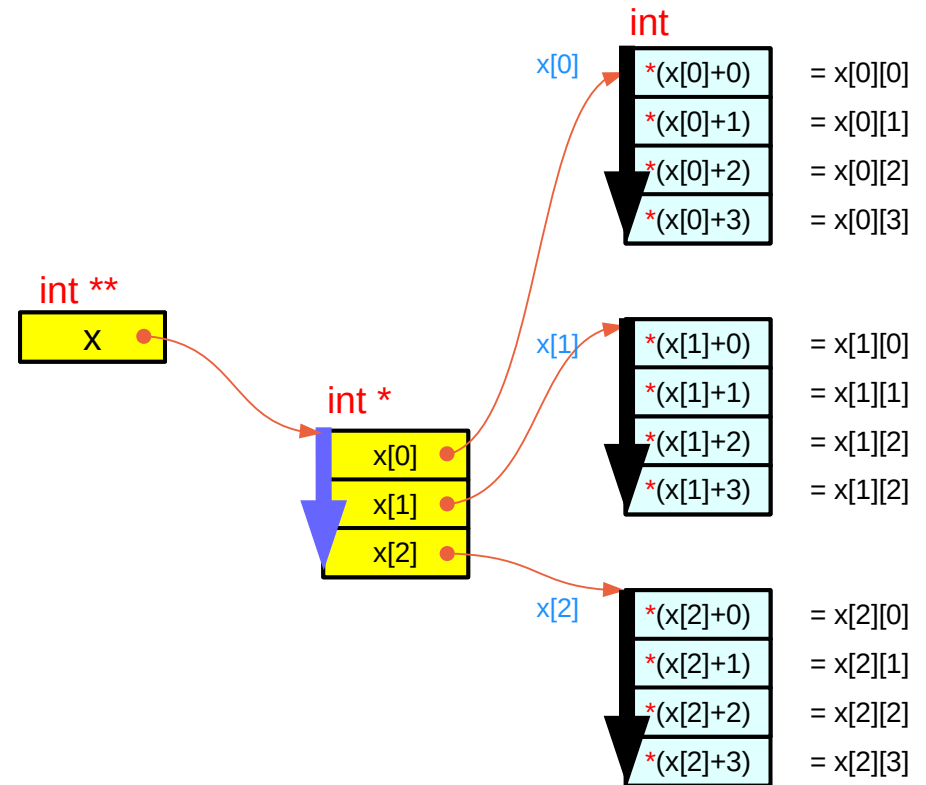
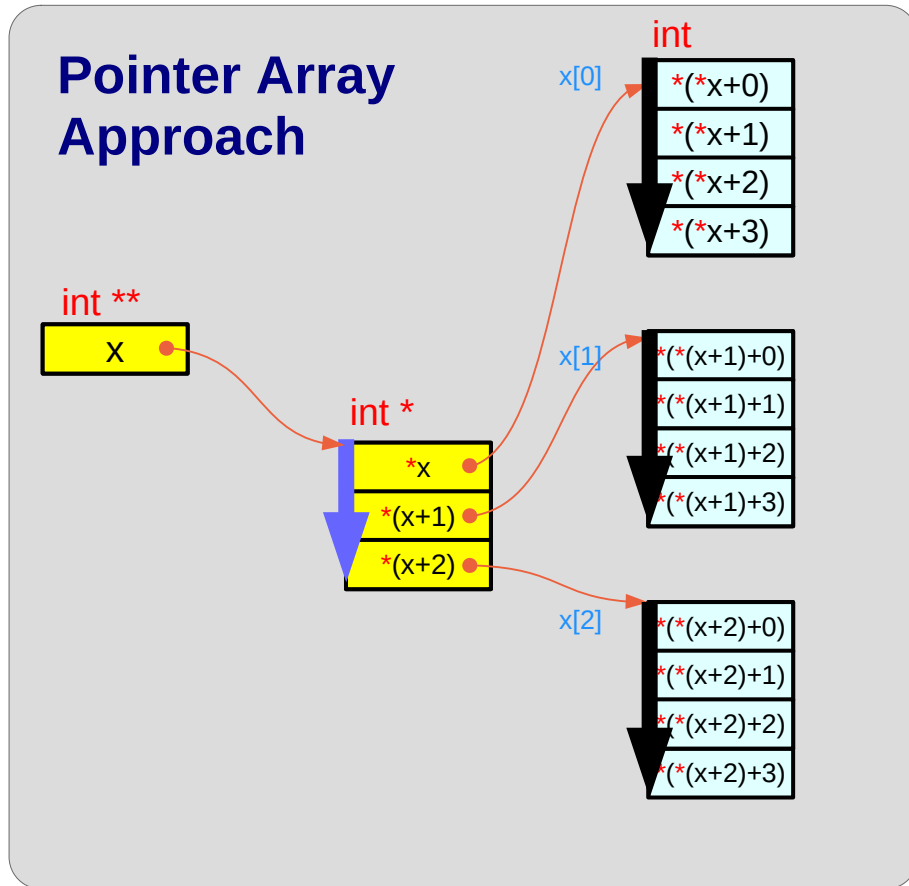
$\text{value}(\mathbf{x}[\mathbf{i}] + \mathbf{j})$ address replications
= $\text{value}(\mathbf{x}[\mathbf{i}]) + \mathbf{j} * \text{sizeof}(\mathbf{x}[\mathbf{i}][\mathbf{j}])$
= $*(\text{value}(\mathbf{x}) + \mathbf{i} * \text{sizeof}(\mathbf{x}[0])) + \mathbf{j} * \text{sizeof}(\mathbf{x}[0][0])$
= $(\text{value}(\mathbf{x}) + \mathbf{i} * \text{sizeof}(\mathbf{x}[0])) + \mathbf{j} * \text{sizeof}(\mathbf{x}[0][0])$

= $\text{value}(\mathbf{x}) + \mathbf{i} * \text{sizeof}(\mathbf{x}[0]) + \mathbf{j} * \text{sizeof}(\mathbf{x}[0][0])$
in math expressions

= $\text{value}(\mathbf{x}) + \mathbf{i} * \mathbf{s1} + \mathbf{j} * \mathbf{s2}$
= $(*(\mathbf{x} + \mathbf{i})_{\mathbf{s1}} + \mathbf{j})_{\mathbf{s2}}$
 $\mathbf{s1} = \text{sizeof}(\mathbf{x}[0]), \mathbf{s2} = \text{sizeof}(\mathbf{x}[0][0])$

$\text{int } \mathbf{x}[3][4];$

* into [] notations – Pointer Array Approach



C expression

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

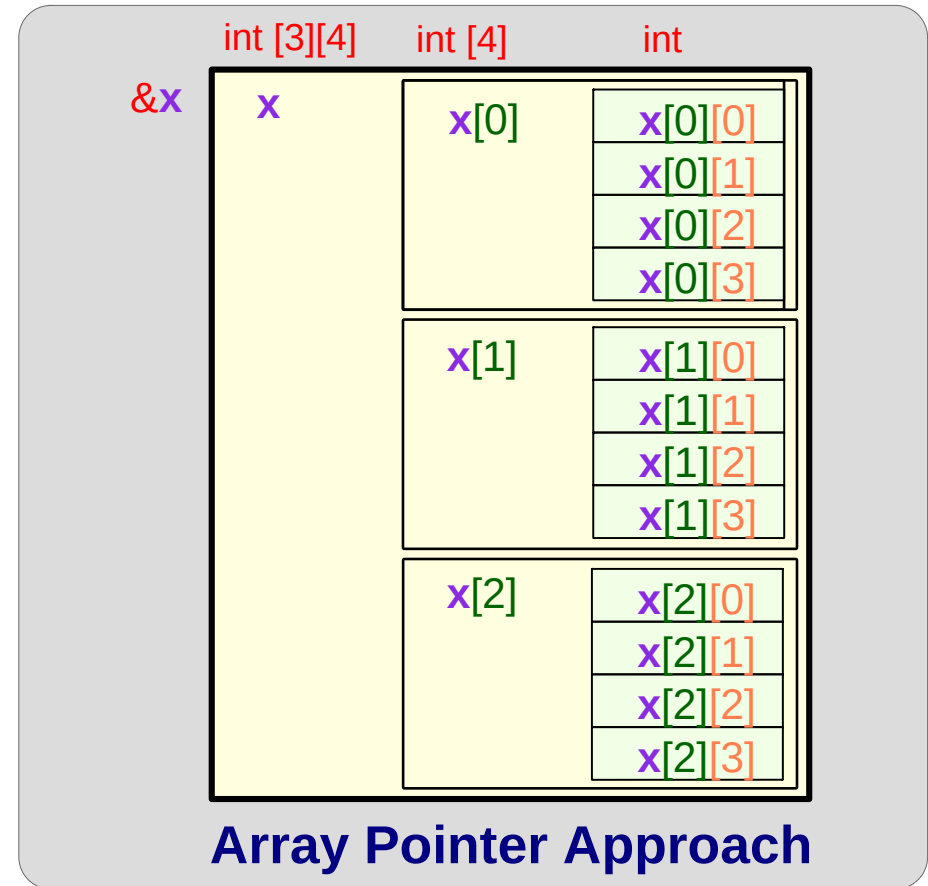
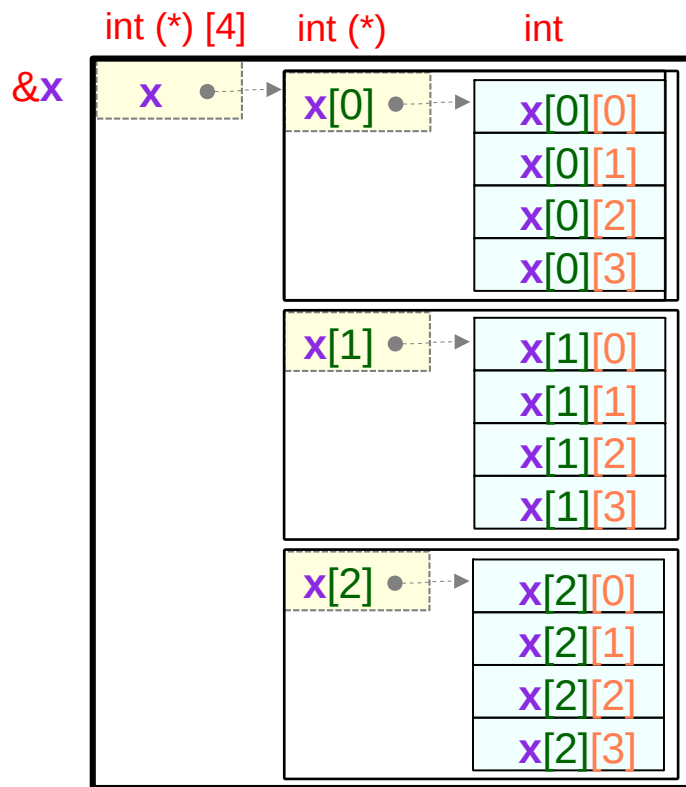


$\mathbf{x}[\mathbf{i}][\mathbf{j}]$

Math expression

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

* and [] notations – Array Pointer Approach



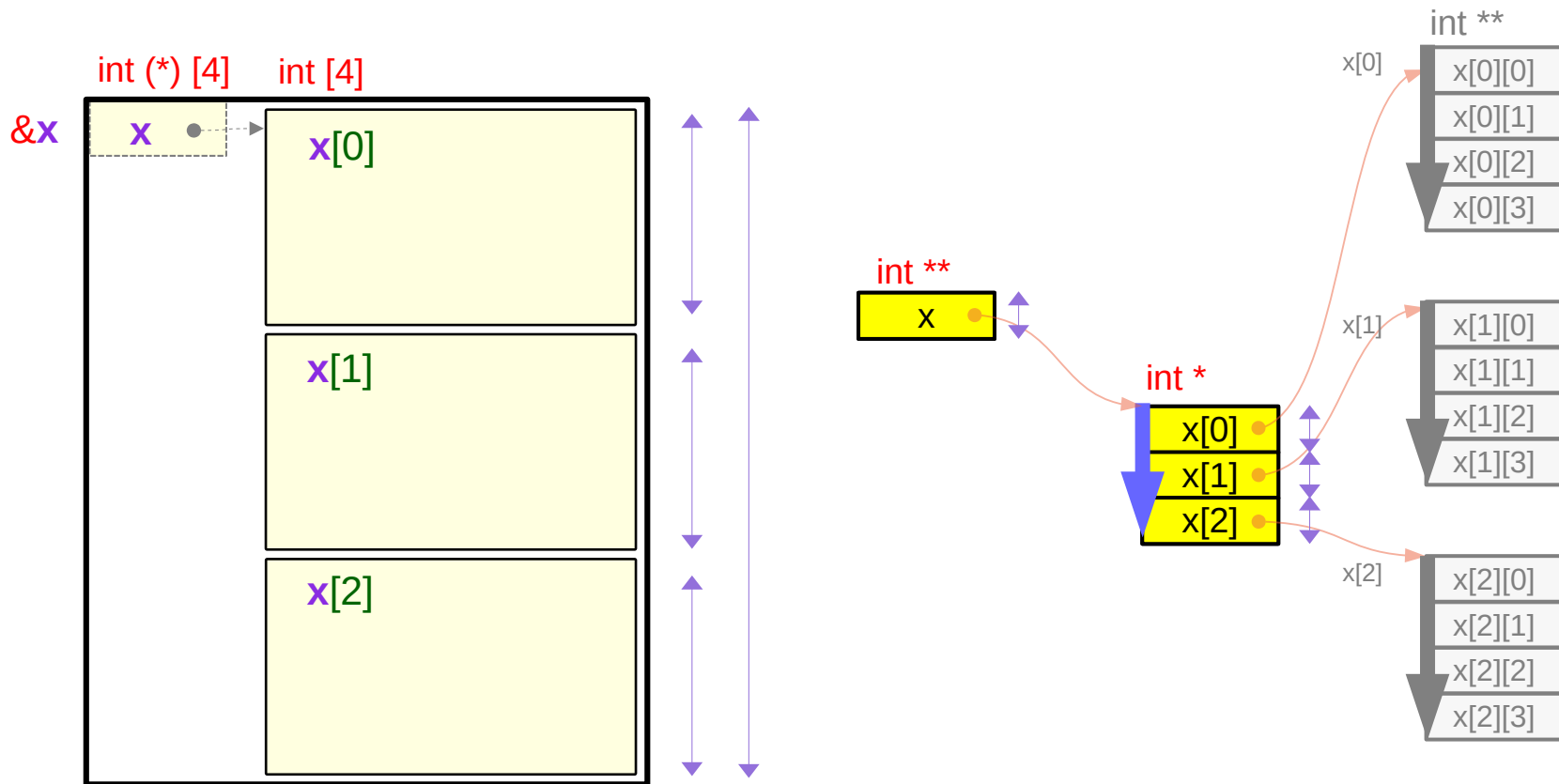
C expression $*(*(\mathbf{x}+\mathbf{i})+\mathbf{j})$



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

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

Virtual pointers vs. real pointers (1)



$\text{value}(\&\mathbf{x}) = \text{value}(\mathbf{x})$ address replications

$\text{sizeof}(\mathbf{x}) = 3 * \text{sizeof}(*\mathbf{x})$ abstract data size

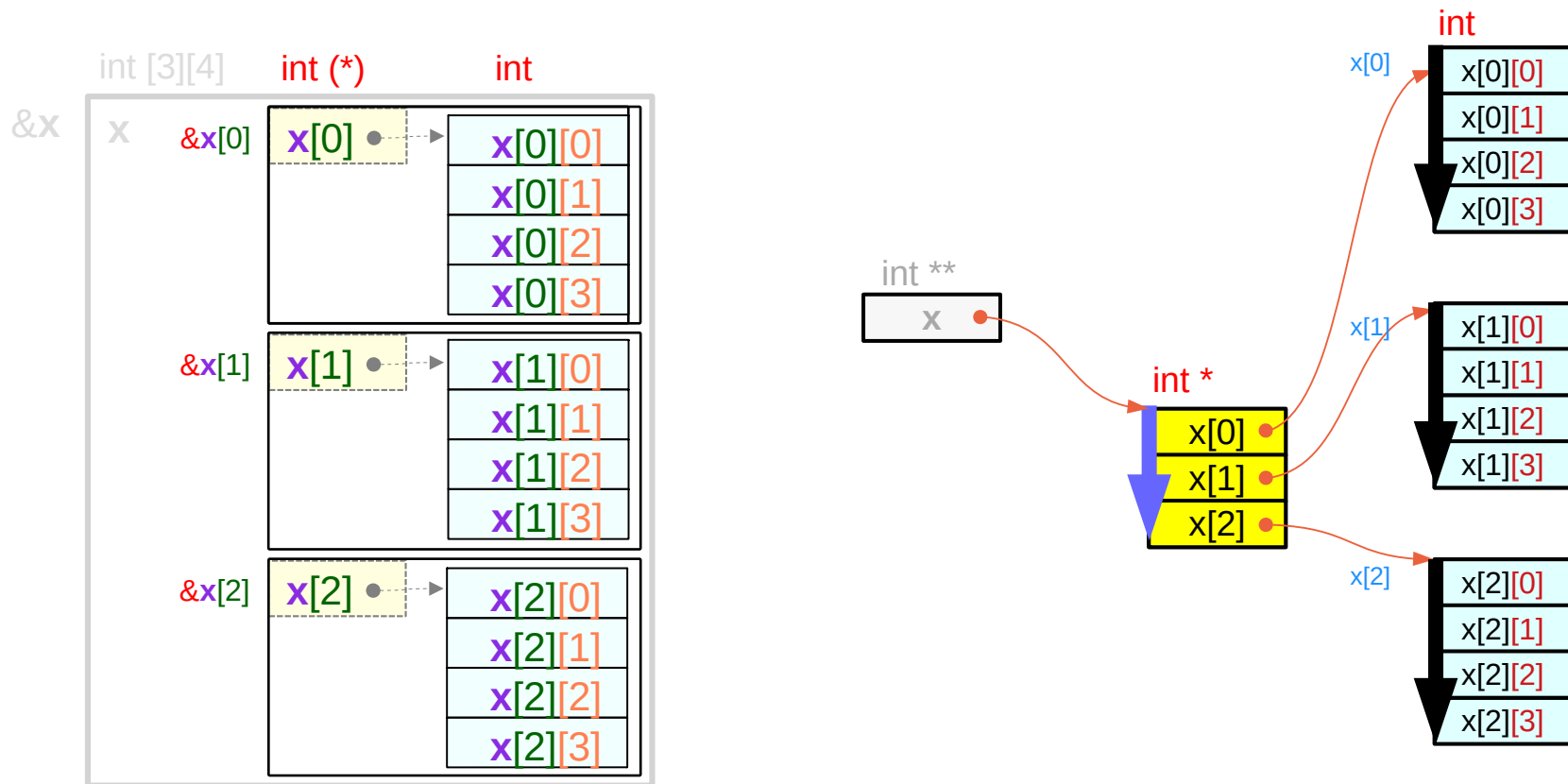
$\text{value}(\mathbf{x}+\mathbf{i}) = \text{value}(\mathbf{x}) + \mathbf{i} * \text{sizeof}(*\mathbf{x})$
 $= \text{value}(\mathbf{x}) + \mathbf{i} * 4 * 4$ pointer arithmetic

$\text{value}(\&\mathbf{x}) \neq \text{value}(\mathbf{x})$ physical memory

$\text{sizeof}(\mathbf{x}) = \text{sizeof}(*\mathbf{x}) = 4$ pointer size

$\text{value}(\mathbf{x}+\mathbf{i}) = \text{value}(\mathbf{x}) + \mathbf{i} * \text{sizeof}(*\mathbf{x})$
 $= \text{value}(\mathbf{x}) + \mathbf{i} * 4$ pointer arithmetic

Virtual pointers vs. real pointers (2)



$\text{value}(\&x[i]) = \text{value}(x[i])$ address replications

$\text{sizeof}(x[i]) = 4 * \text{sizeof}(*x[i])$ abstract data size

$\text{value}(x[i]+j) = \text{value}(x[i]) + j * \text{sizeof}(*x[i])$
 $= \text{value}(x[i]) + j * 4$ pointer arithmetic

$\text{value}(\&x[i]) \neq \text{value}(x[i])$ physical memory

$\text{sizeof}(x[i]) = \text{sizeof}(*x[i]) = 4$ pointer size

$\text{value}(x[i]+j) = \text{value}(x[i]) + j * \text{sizeof}(*x[i])$
 $= \text{value}(x[i]) + j * 4$ pointer arithmetic

Relaxing the outermost dimension

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

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

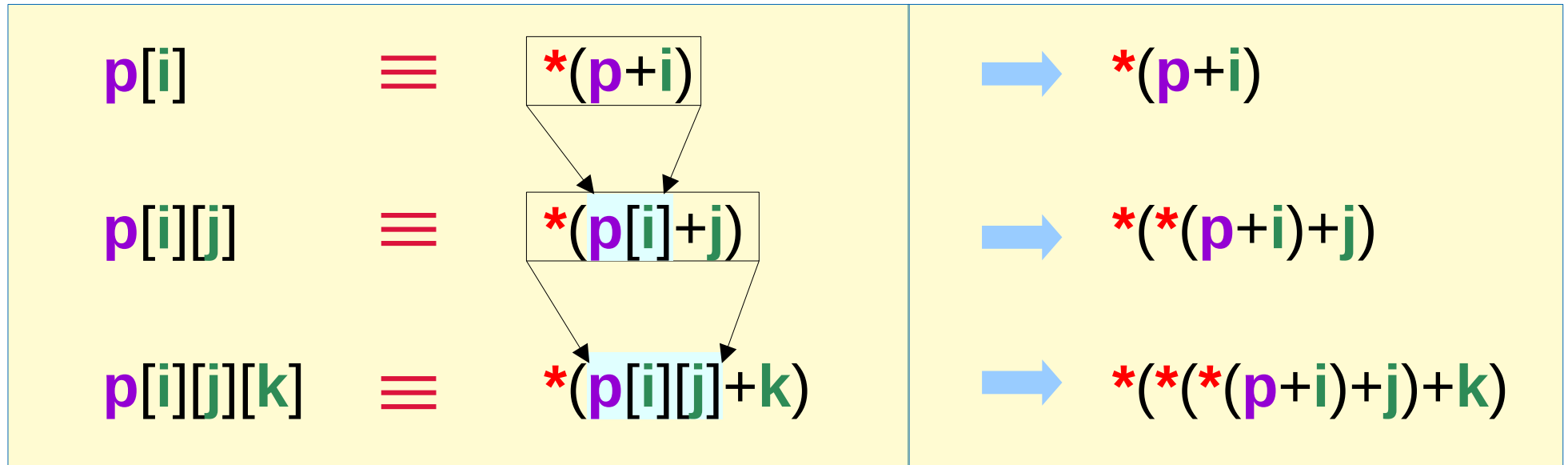
$*\text{value}(X) = *X$

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

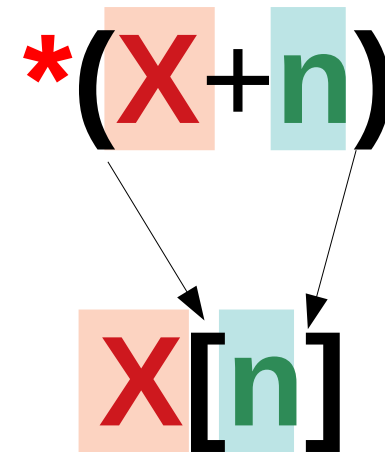
$\&p[0] \equiv \text{value}(p)$
 $\&p[i][0] \equiv \text{value}(p[i])$
 $\&p[i][j][0] \equiv \text{value}(p[i][j])$

valid for proper i, j, k values

Relaxing all the dimensions



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



valid for proper i, j, k values

Left-to-right and right-to-left associative operators

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

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

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

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

Address Calculation (1) Array Pointer Approach

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

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

```
&c[i]     ≡ value(c + i)
&c[i][j]  ≡ value(c[i] + j)
&c[i][j][k] ≡ value(c[i][j] + k)
```

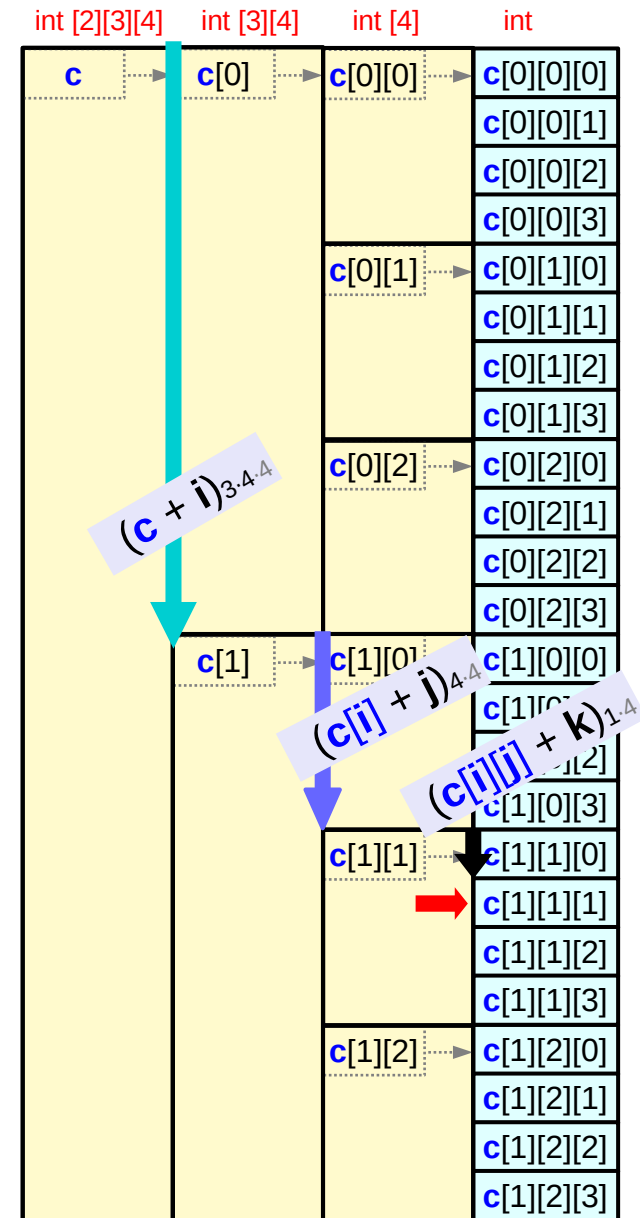
address replication

value(c[i][j][k]) ≠ value(&c[i][j][k]) ← primitive data & address

value(c[i][j])	= value(&c[i][j])	= value(&c[i][j][0])
value(c[i])	= value(&c[i])	= value(&c[i][0][0])
value(c)	= value(&c)	= value(&c[0][0][0])

skip i elements of c
skip j elements of c[i]
skip k elements of c[i][j]

skip i*3*4 primitive elements of c
skip j*4 primitive elements of c
skip k primitive elements of c



Address Calculation (2) Pointer Array Approach

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

```
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]
```

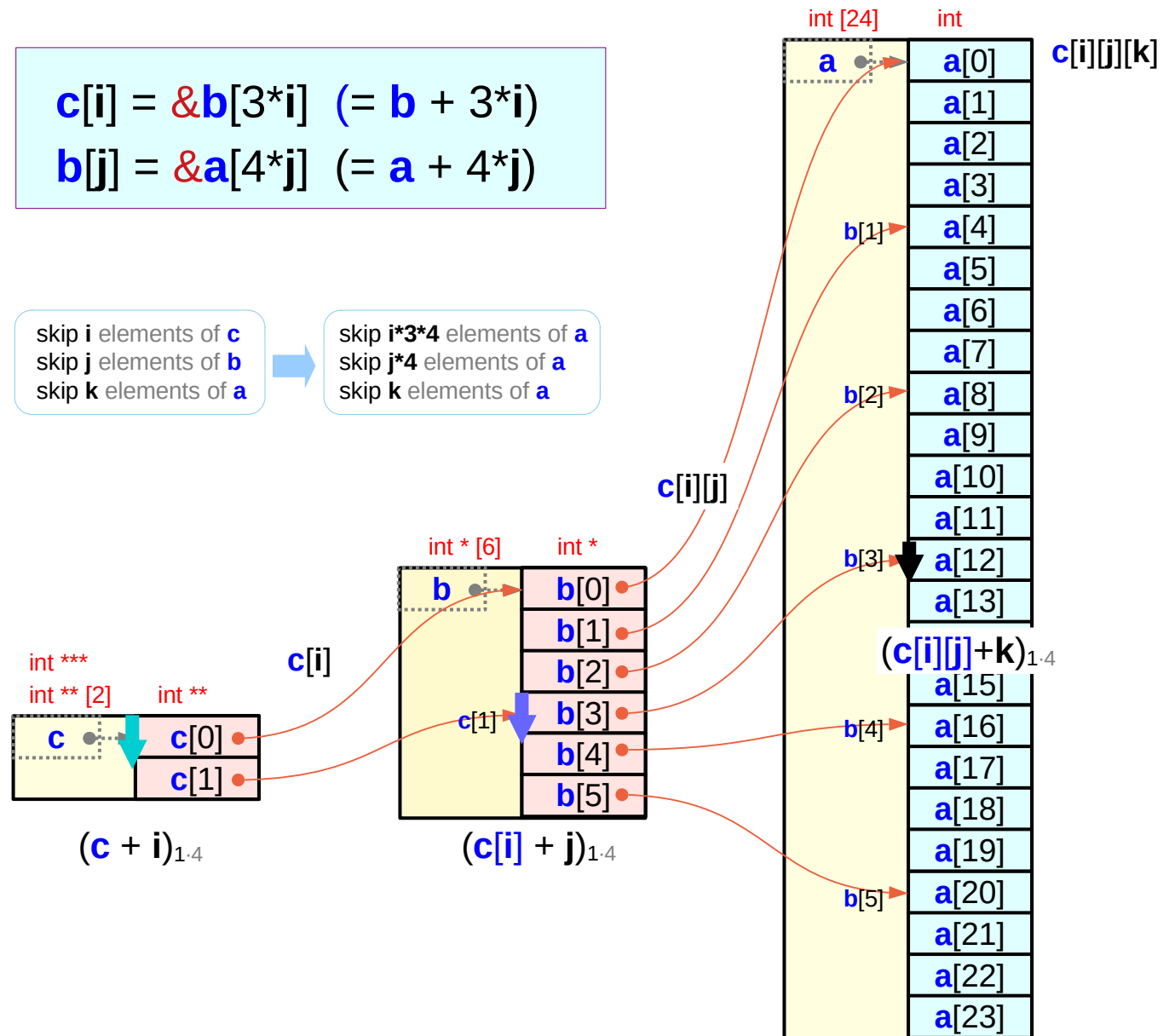
```
c[i][j] ≡ (a+(i*3+j)*4)
*(c[i][j]+k) = *(a+(i*3+j)*4+k);
c[i][j][k] ≡ a[(i*3+j)*4+k]
```

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

skip i elements of c
skip j elements of b
skip k elements of a

→

skip i*3*4 elements of a
skip j*4 elements of a
skip k elements of a

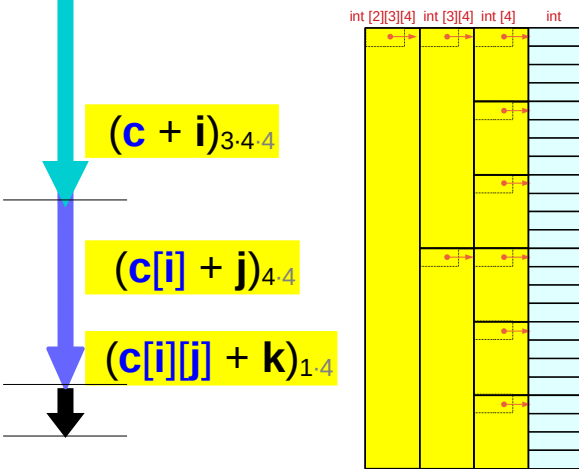


Address Calculation (3)

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

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

Array Pointer Approach



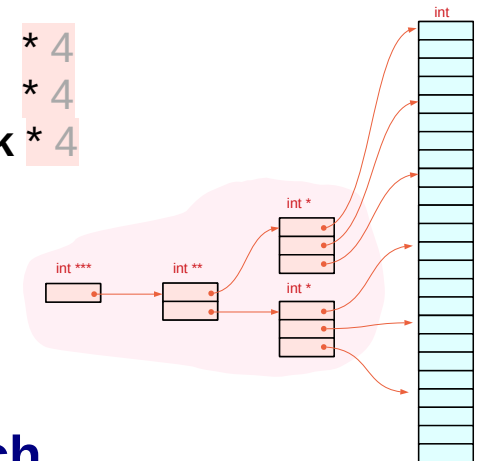
$$\begin{aligned} \mathbf{c}[\mathbf{i}] &\equiv *(\mathbf{c} + \mathbf{i}) \\ \mathbf{c}[\mathbf{i}][\mathbf{j}] &\equiv *(\mathbf{c}[\mathbf{i}] + \mathbf{j}) \\ \mathbf{c}[\mathbf{i}][\mathbf{j}][\mathbf{k}] &\equiv *(\mathbf{c}[\mathbf{i}][\mathbf{j}] + \mathbf{k}) \end{aligned}$$

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

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

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

Pointer Array Approach



Array Pointer Approach (1)

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

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

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

from p , skip
 $i \cdot M \cdot N$ integers

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

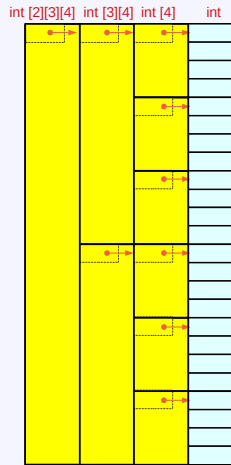
from $p[i]$, skip
 $j \cdot N$ integers

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

from $p[i][j]$, skip
 k integers

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

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



Array Pointer Approach

address replications

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

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

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

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

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

Pointer Array Approach (1)

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

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

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

skip i pointers
from p

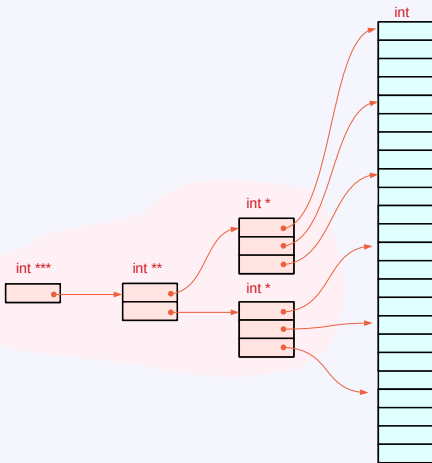
$$\begin{aligned} & \&p[i] = \text{value}(p + i)_{1..4} \\ & = \text{value}(p) + i * 4 \end{aligned}$$

skip j pointers
from $p[i]$

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

skip k integers
from $p[i][j]$

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



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

Pointer Array Approach

address dereferences

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

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

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

$$p[i][j][k] = *value(*value(*value(p) + i * 4) + j * 4) + k * 4$$

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

Subscript [] and dereference * notations (1a)

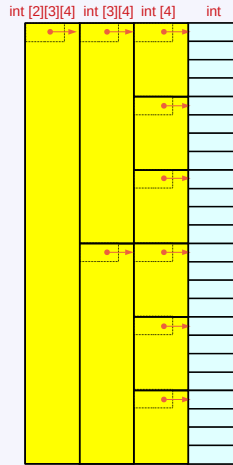
$$\begin{aligned} p[i][j][k] = *value(*value(*value(p + i) + j) + k) &\xrightarrow[\text{address}]{\text{replications}} = value(value(value(p + i)_{3 \cdot 4 \cdot 4} + j)_{4 \cdot 4} + k)_4 \\ &= value(value((value(p) + i * 3 \cdot 4 \cdot 4) + j)_{4 \cdot 4} + k)_4 \\ &= value((value(p) + i * 3 \cdot 4 \cdot 4 + j * 4 \cdot 4) + k)_4 \\ &= (value(p) + i * 3 \cdot 4 \cdot 4 + j * 4 \cdot 4 + k * 4) \end{aligned}$$

$$\begin{aligned} p[i][j][k] = *value(*value(*value(p + i) + j) + k) &\xrightarrow{\quad} = *value(*value(*value(p + i)_4 + j)_4 + k)_4 \\ &= *value(*value(*(value(p) + i * 4) + j)_4 + k)_4 \\ &= *value(*value(*(value(p) + i * 4) + j * 4) + k)_4 \\ &= *value(*value(*(value(p) + i * 4) + j * 4) + k * 4) \end{aligned}$$

Array Pointer Approach (2)

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

Array Pointer Approach



address replications

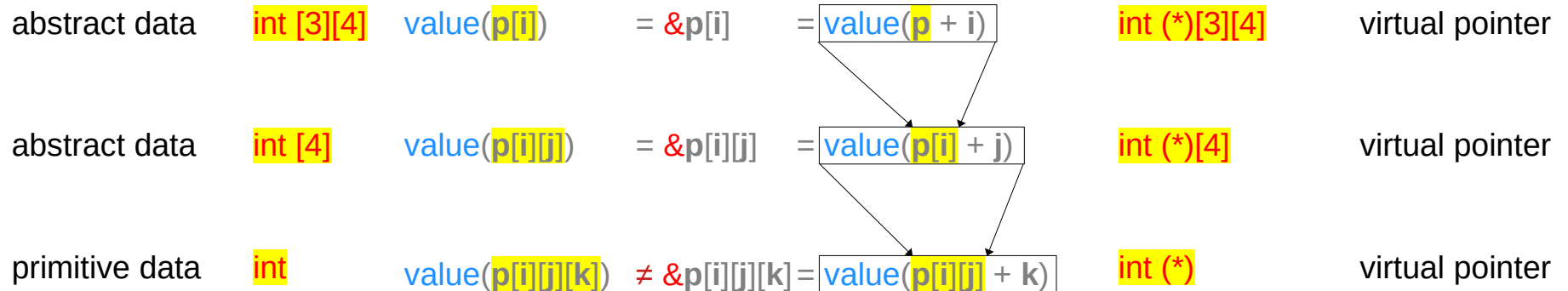
$$\text{value}(p[i]) = \&p[i] = \text{value}(p + i)$$

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

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

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

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

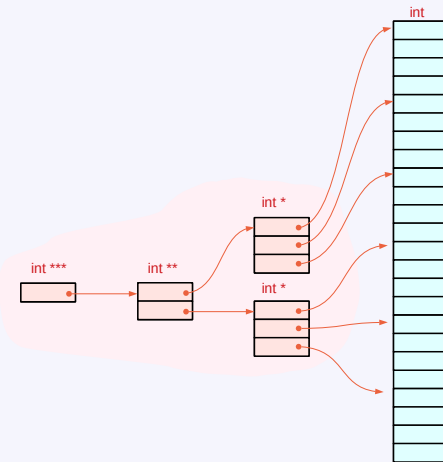


$$p[i][j][k] = *value(*value(*value(p+i)+j)+k) \xrightarrow{\text{address replications}} \begin{aligned} &= \text{value}(\text{value}(\text{value}(p + i)_{3 \cdot 4 \cdot 4} + j)_{4 \cdot 4} + k)_{4} \\ &= \text{value}(p + i * 3 \cdot 4 \cdot 4 + j * 4 \cdot 4 + k * 4) \end{aligned}$$

Pointer Array Approach (2)

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

Pointer Array Approach



address dereferences

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

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

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

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

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

pointer data `int [3][4]` `value(p[i])` = `*(&p[i])` = `*value(p + i)` `int **` double pointer

pointer data `int [4]` `value(p[i][j])` = `*(&p[i][j])` = `*value(p[i] + j)` `int *` int pointer

primitive data `int` `value(p[i][j][k])` \neq `*(&p[i][j][k])` = `*value(p[i][j] + k)` `int` integer

$$p[i][j][k] = \text{value}(\text{value}(\text{value}(p+i)+j)+k) \xrightarrow{\text{address dereferences}} \begin{aligned} &= \text{value}(\text{value}(\text{value}(p + i)_4 + j)_4 + k)_4 \\ &= \text{value}(\text{value}(\text{value}(p) + i * 4) + j * 4) + k * 4 \end{aligned}$$

Subscript [] and dereference * notations (2)

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

C Expressions

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

C Expressions

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

$\text{value}(\&X) = \text{value}(X)$ (address replication)

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

Math Expressions

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

Math Expressions

`int ** p[L], * q[L·M], r[L·M·N] ;`

$*\text{value}(X) = *X$

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

Math Expressions

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

Math Expressions

Subscript [] and dereference * notations (3)

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

$\text{value}(\&X) = \text{value}(X)$ (address replication)

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

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

Math Expressions

`int ** p[L], * q[L·M], r[L·M·N] ;`

$*\text{value}(X) = *X$

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

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

Math Expressions

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>