Applications of Arrays (1A)

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

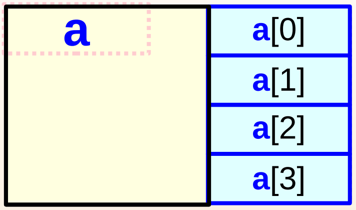
`int a[N]` — view **a** as an array

N is not fixed

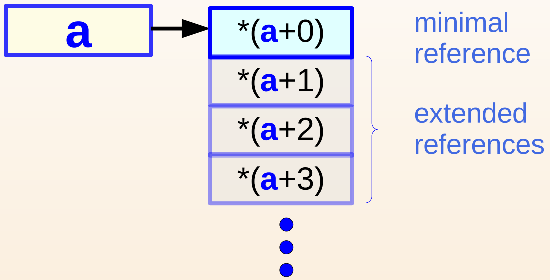
sizeof(**a**) is
not the size of the array
but of a pointer variable

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

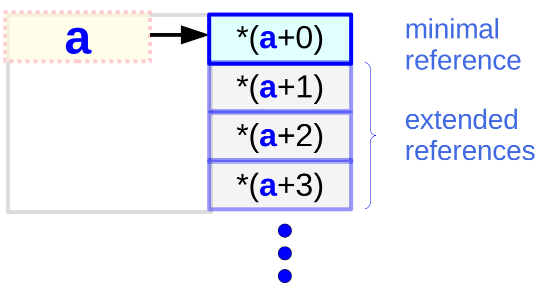
int **a**[4] ;   an array **a**



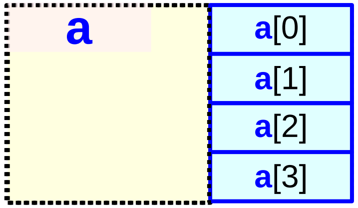
int (*a) ;   a pointer **a**



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**) ≠ value(**a**)
- variable number of elements

int **a**[N] — **a** as an array

a is not a real array

- sizeof(**a**) ≠ an array size
  = a pointer size
- value(&**a**) ≠ 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 (*b)[2] ; — declare a **1-d** array pointer b

a specific instance

int b[N][2] — b as a **2-d** array

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

generalization

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

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**) ≠ value(**b**)
- variable number of elements

int [N][2]

b as a **2-d** array

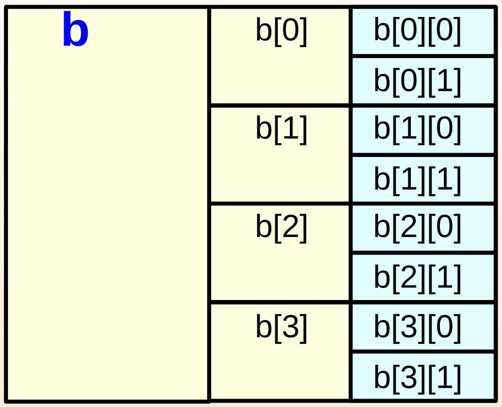
b is not a real array

- sizeof(**b**) ≠ an array size
= a pointer size
- value(&**b**) ≠ value(**b**)
= assigned address

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

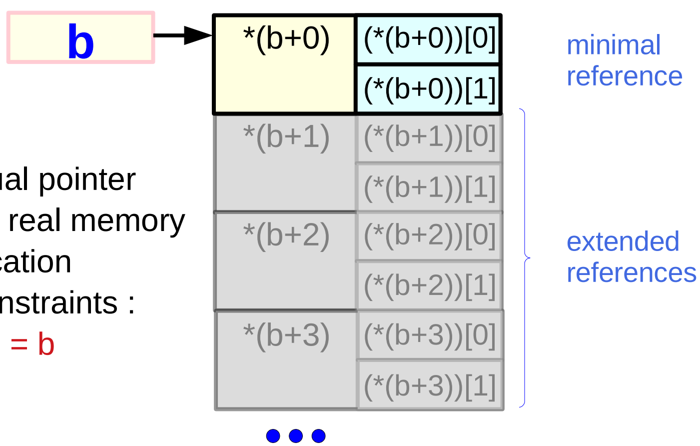
int **b**[4][2] ;

2-d array **b**



int (*) [2]

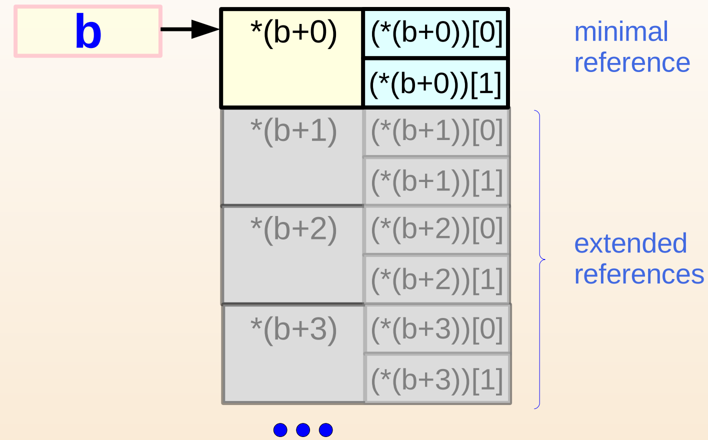
b as a **1-d** array pointer



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

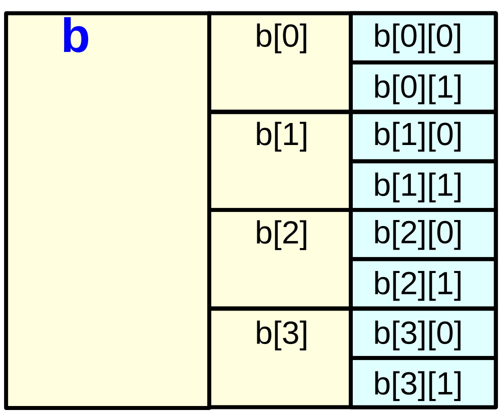
int (***b**) [2] ;

1-d array pointer **b**



int [N][2]

b as a **2-d** array



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

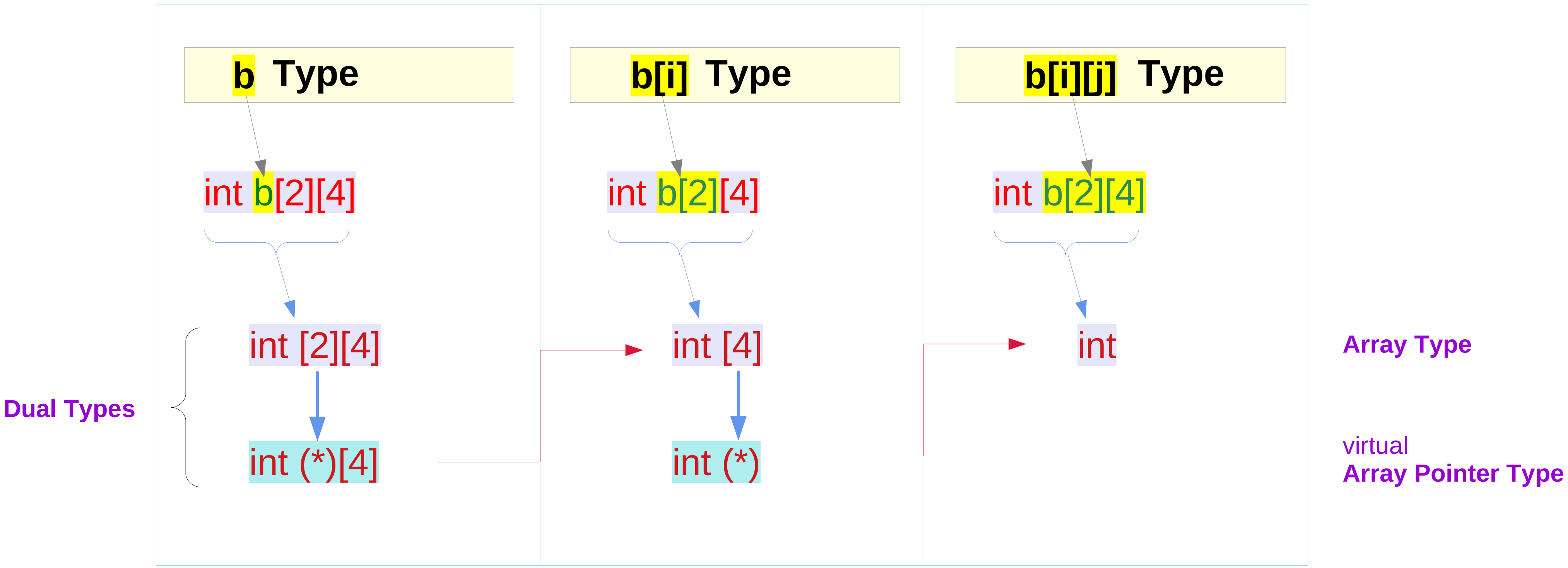
a specific instance

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

Subarray types in a **2-d** array

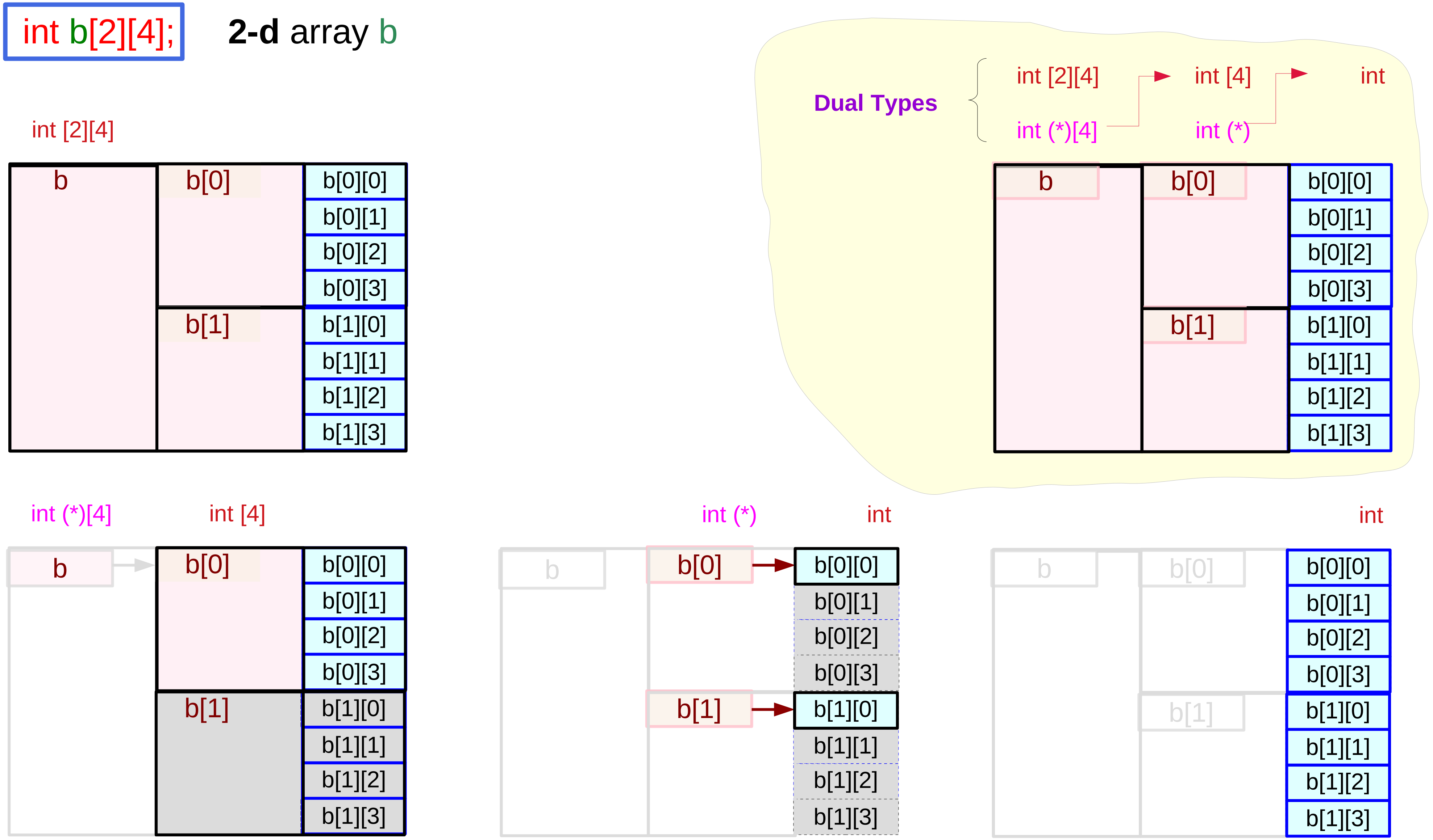
int b[2][4];

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		

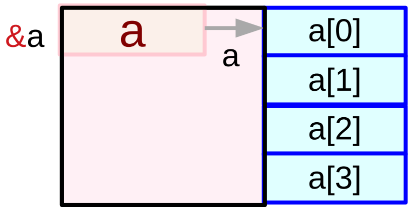
```
int b[2][4];
```

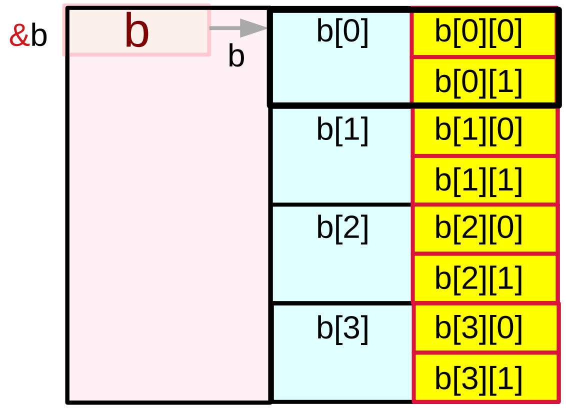
			relaxed type	virtual
b	int [2][4]	**2-d** array type	int (*)[4]	**1-d** array pointer type
b[i]	int [4]	**1-d** array type	int (*)	**0-d** array pointer type
b[i][j]	int	**0-d** array type		

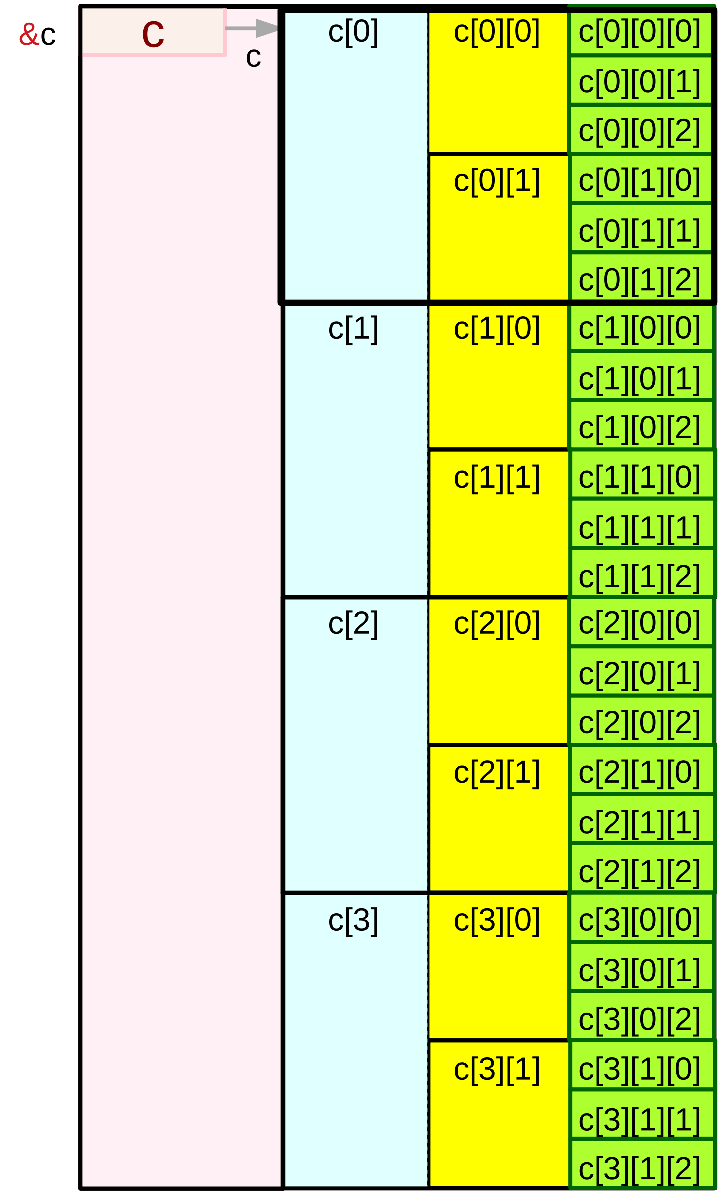
```
int c[4][2][3];
```

			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





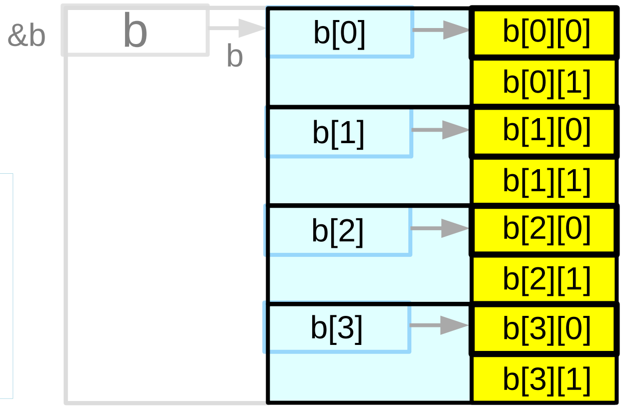



```
int a[4];
int b[2][4];
int c[4][2][3];
```

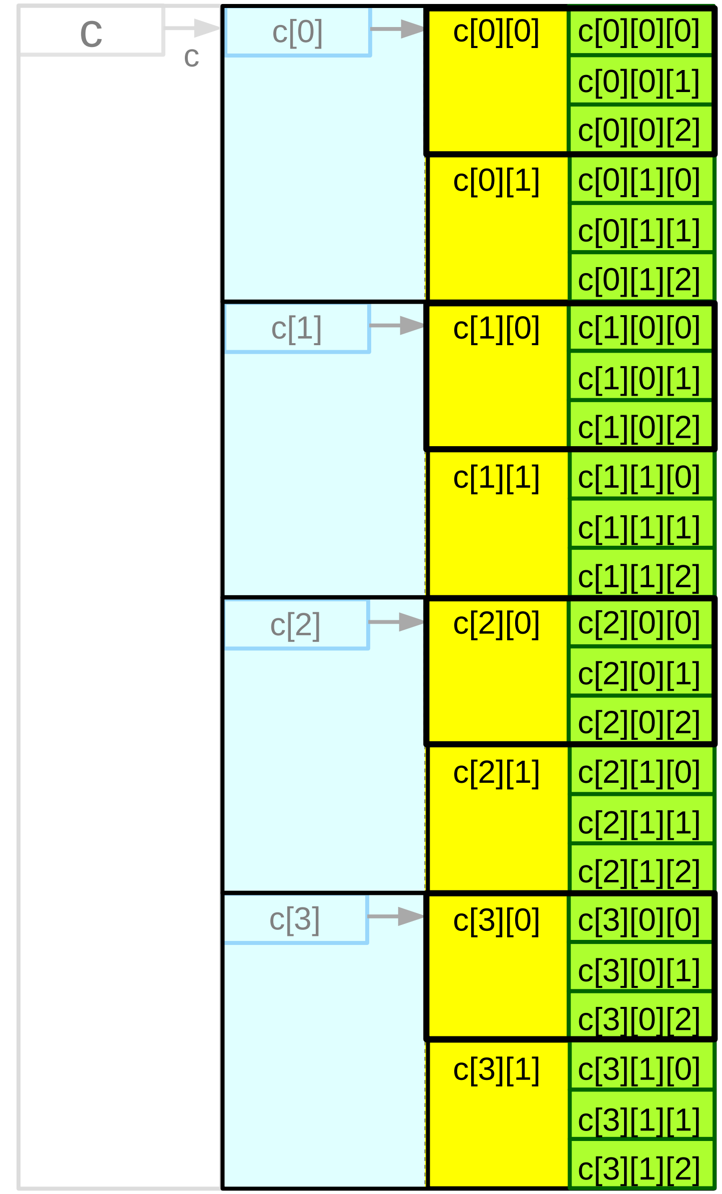
dual types

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

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



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

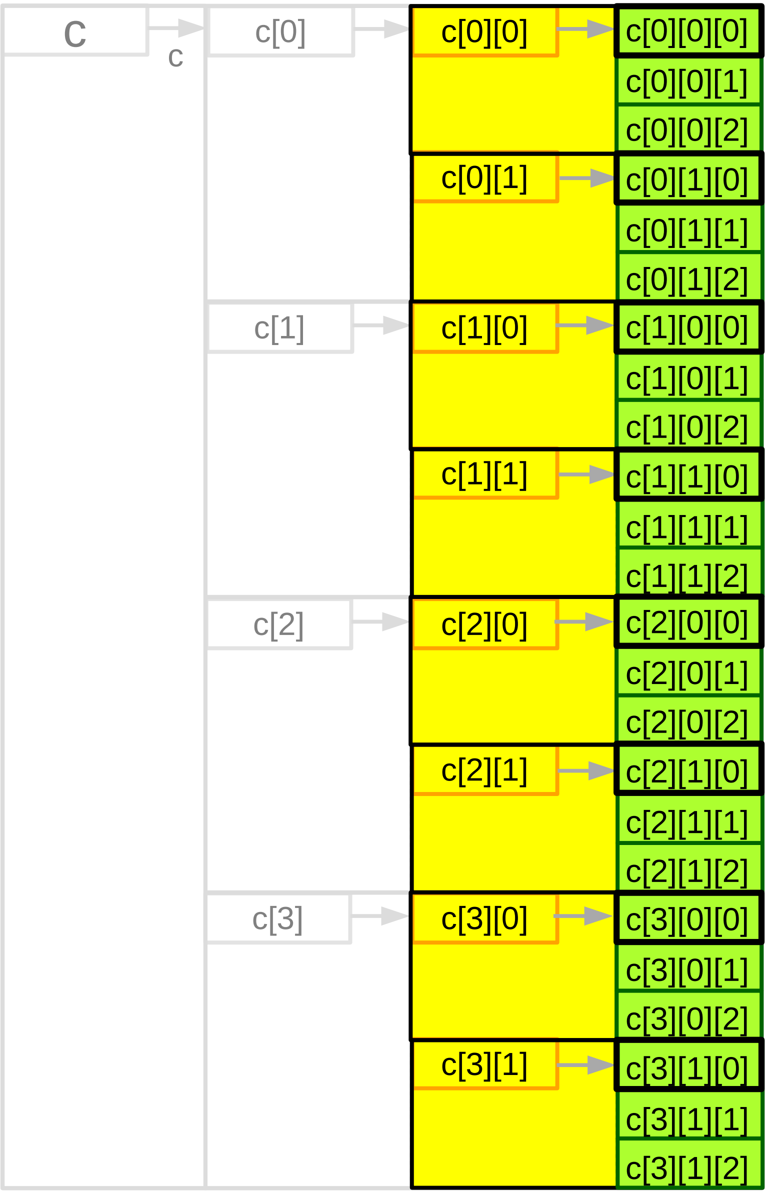


dual types

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

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

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



dual types

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

Types of a **4-d** array and its subarrays

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

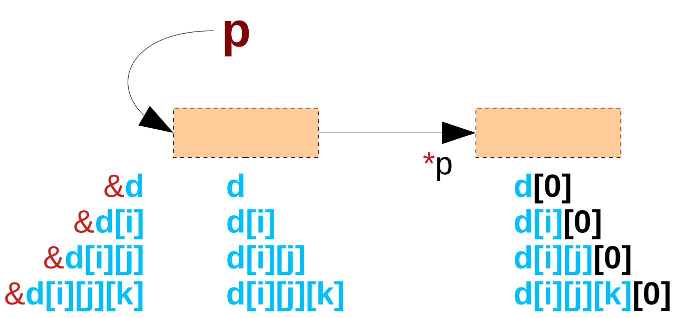
types

d	consider d[4][2][3][4]	int [4][2][3][4]	**4-d** array	
	relax the 1st 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 1st 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 1st 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 1st 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

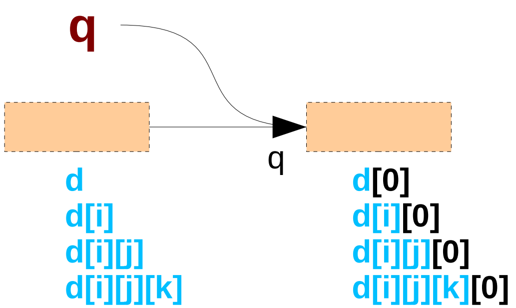
int d[4][2][3][4];



d	**4-d** array	d[4][2][3][4]	p = &d	abstract data
p	**4-d** array pointer	(*p)[4][2][3][4]	int (*p)[4][2][3][4] = &d;	
			(*p)[i][j][k][l] ≡ 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]	int (*p)[3][4] = &d[i];	
			(*p)[j][k][l] ≡ 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]	int (*p)[4] = &d[i][j];	
			(*p)[k][l] ≡ 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]	int (*p) = &d[i][j][k];	
			(*p)[l] ≡ 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	d[4][2][3][4]	**q = d**	virtual pointer
q	**3-d** array pointer	(***q**)[2][3][4]	int (***q**)[2][3][4] = **d**;	
			q[i][j][k][l] ≡ **d**[i][j][k][l]	
d[i]	**3-d** array	d[i][2][3][4]	**q = d[i]**	virtual pointer
q	**2-d** array pointer	(***q**)[3][4]	int (***q**)[3][4] = **d[i]**;	
			q[j][k][l] ≡ **d[i]**[j][k][l]	given i
d[i][j]	**2-d** array	d[i][j][3][4]	**q = d[i][j]**	virtual pointer
q	**1-d** array pointer	(***q**)[4]	int (***q**)[4] = **d[i][j]**;	
			q[k][l] ≡ **d[i][j]**[k][l]	given i, j
d[i][j][k]	**1-d** array	d[i][j][k][4]	**q = d[i][j][k]**	virtual pointer
q	**0-d** array pointer	(***q**)	int (***q**) = **d[i][j][k]**;	
			q[l] ≡ **d[i][j][k]**[l]	given i, j, k

Aggregate Data Types
Abstract Data Types
Virtual Array Pointers

Aggregate data type

an aggregate type

consists of **N** elements

each element
- starting address
- size

int [4][4]

dual type

a virtual pointer

the address and value of
a virtual pointer
are the *same*

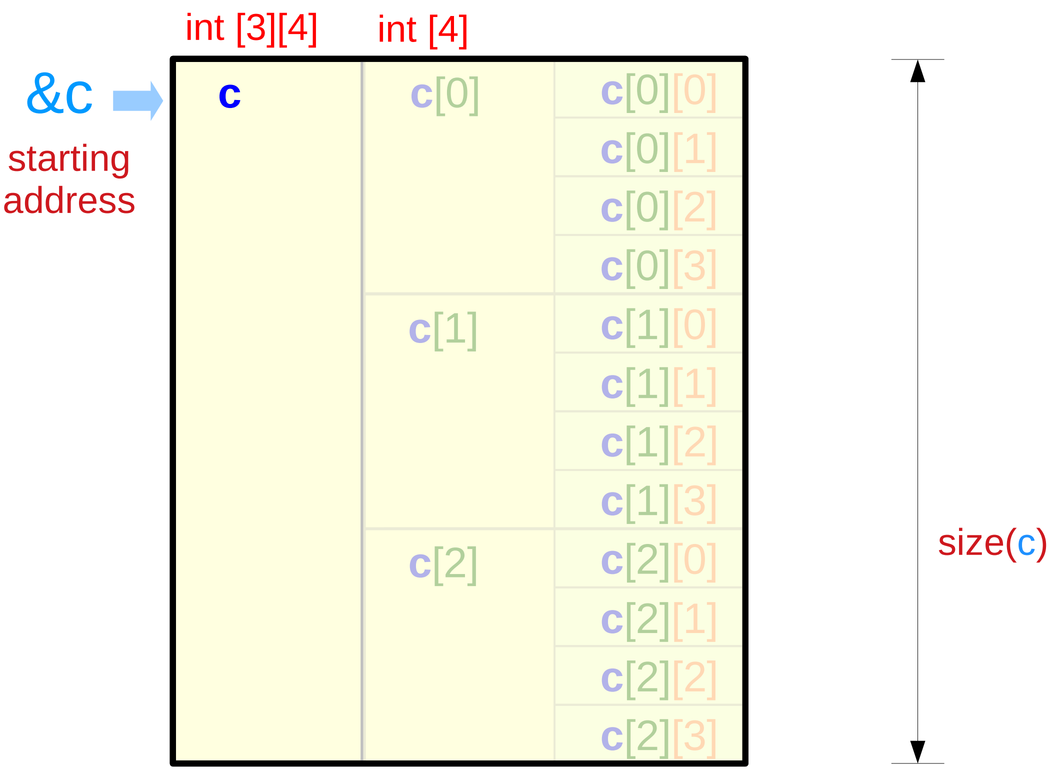
int (*)[4]

an abstract type

is considered as one unit

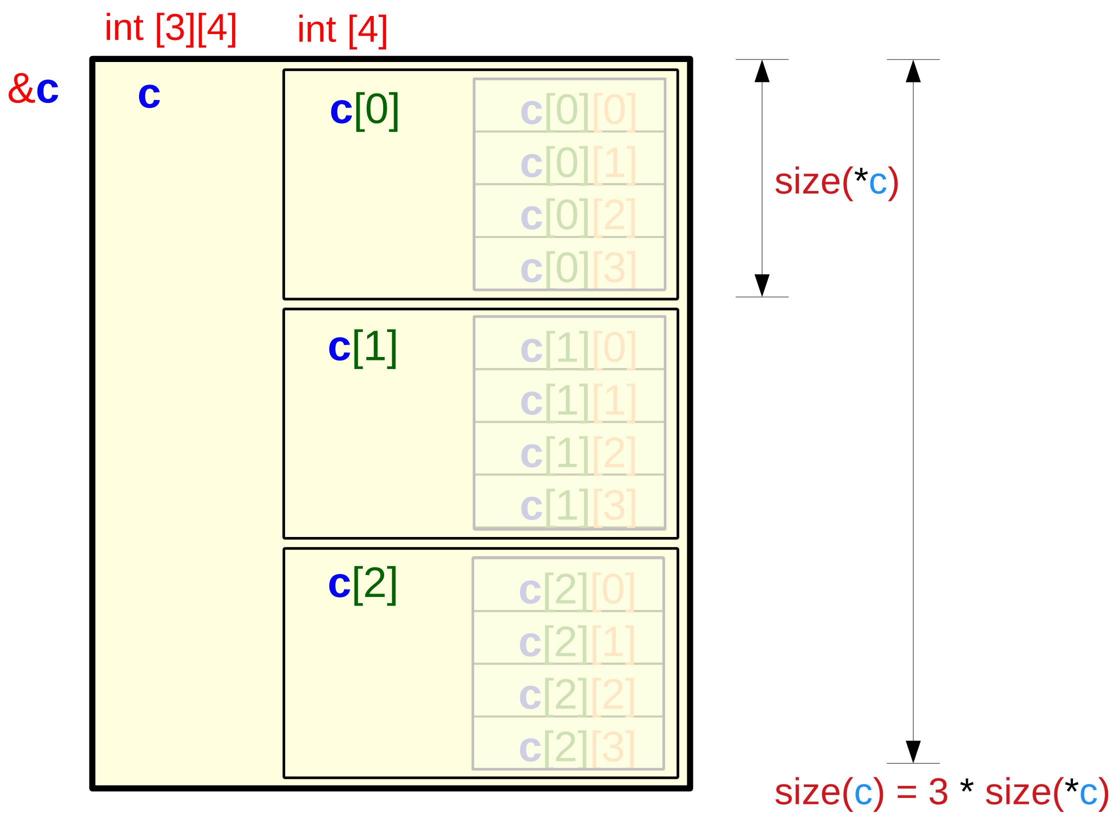
- starting address
- size

Abstract data **c**



an abstract data c
- start address &c
- size sizeof(c)

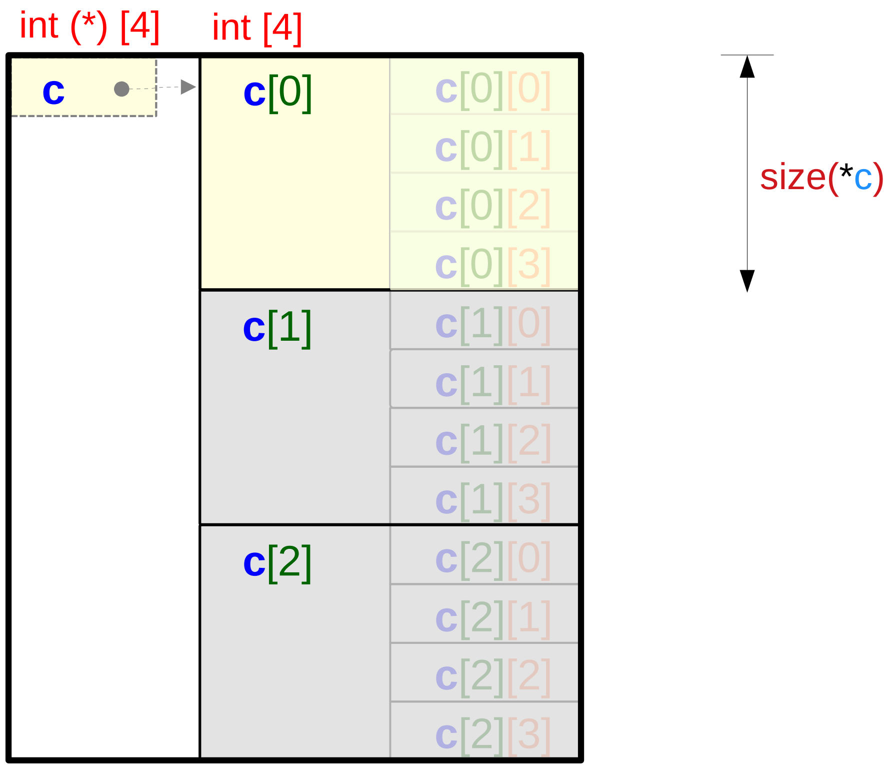
Aggregate data **c**



an aggregate type data	**c**
- 1st element	**c**[0]
- 2nd element	**c**[1]
- 3rd element	**c**[2]

Virtual pointer **c**

&c = c = &c[0]



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

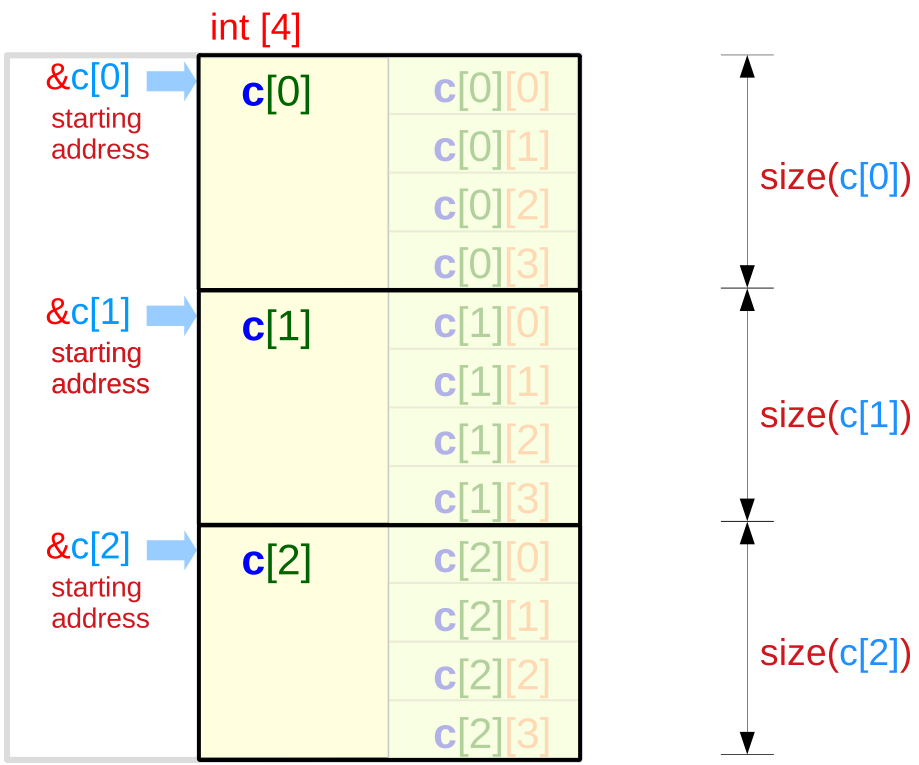
with the constraint
c = &**c**

an abstract data **c**[0] = ***c**
- start address &**c**[0] = **c**
- size sizeof(**c**[0])

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

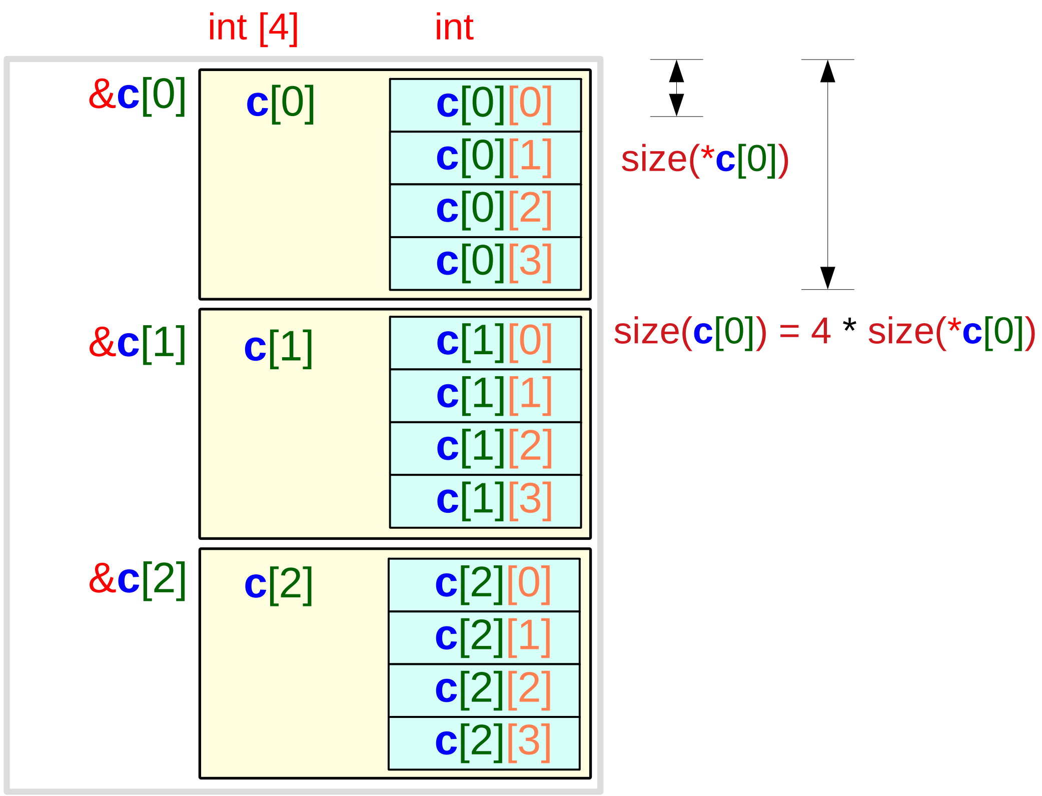
virtual pointers
- no physical memory locations are allocated
- address and data have the same value

Abstract data **c[i]**



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

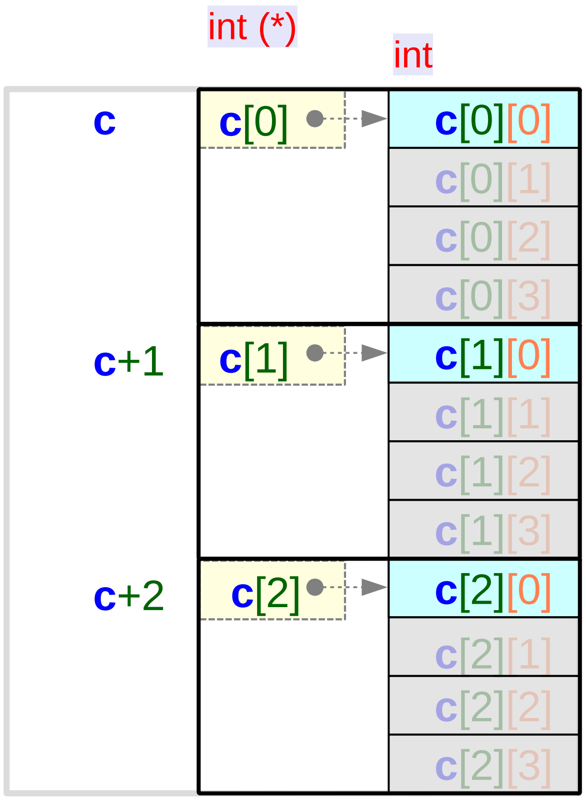
Aggregate data **c[i]**



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]

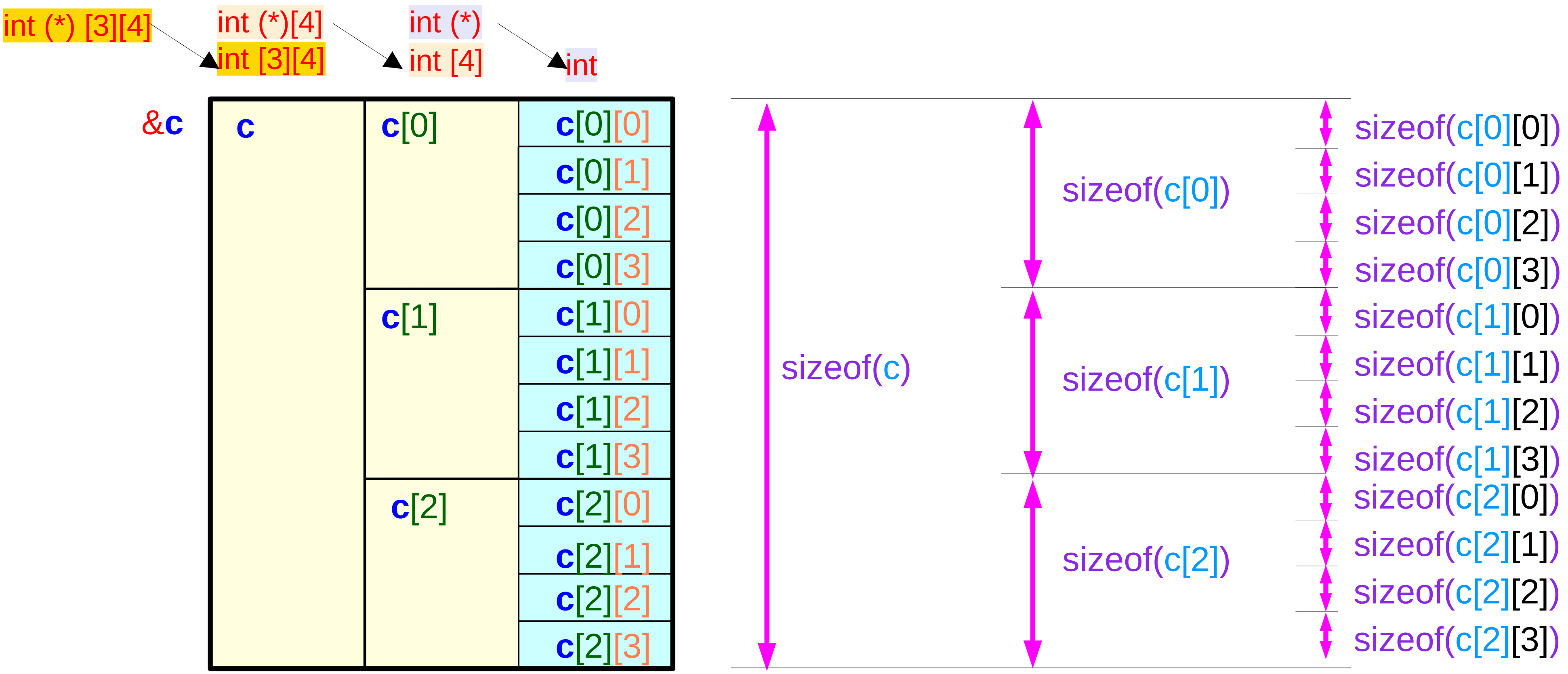
an primitive data c[i][0] = *c[i]
- start address &c[i][0] = c[i]
- size sizeof(c[i][0])

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

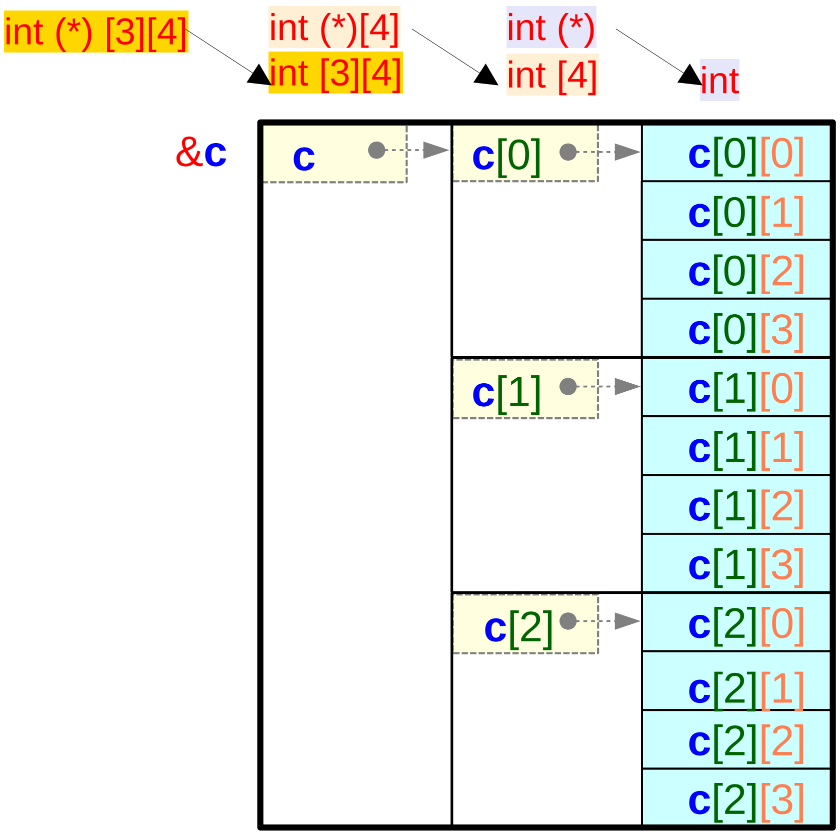
A **2-d** array and its **1-d** sub-arrays – a size view

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])

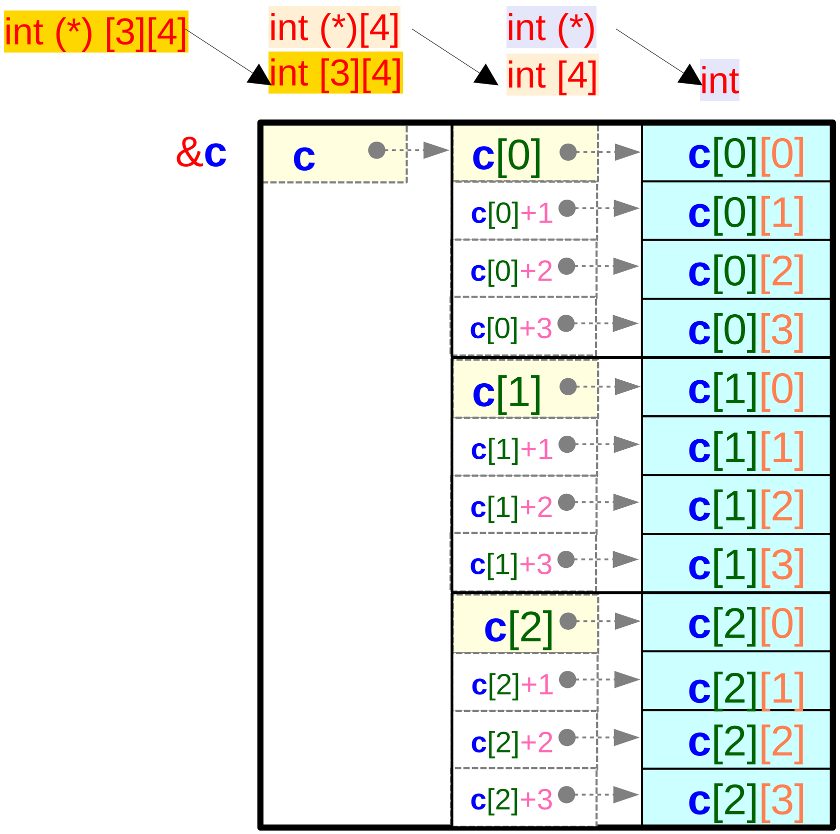
value(c[1]) = value(&c[1][0])
value(&c[1]) = value(&c[1][0])

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

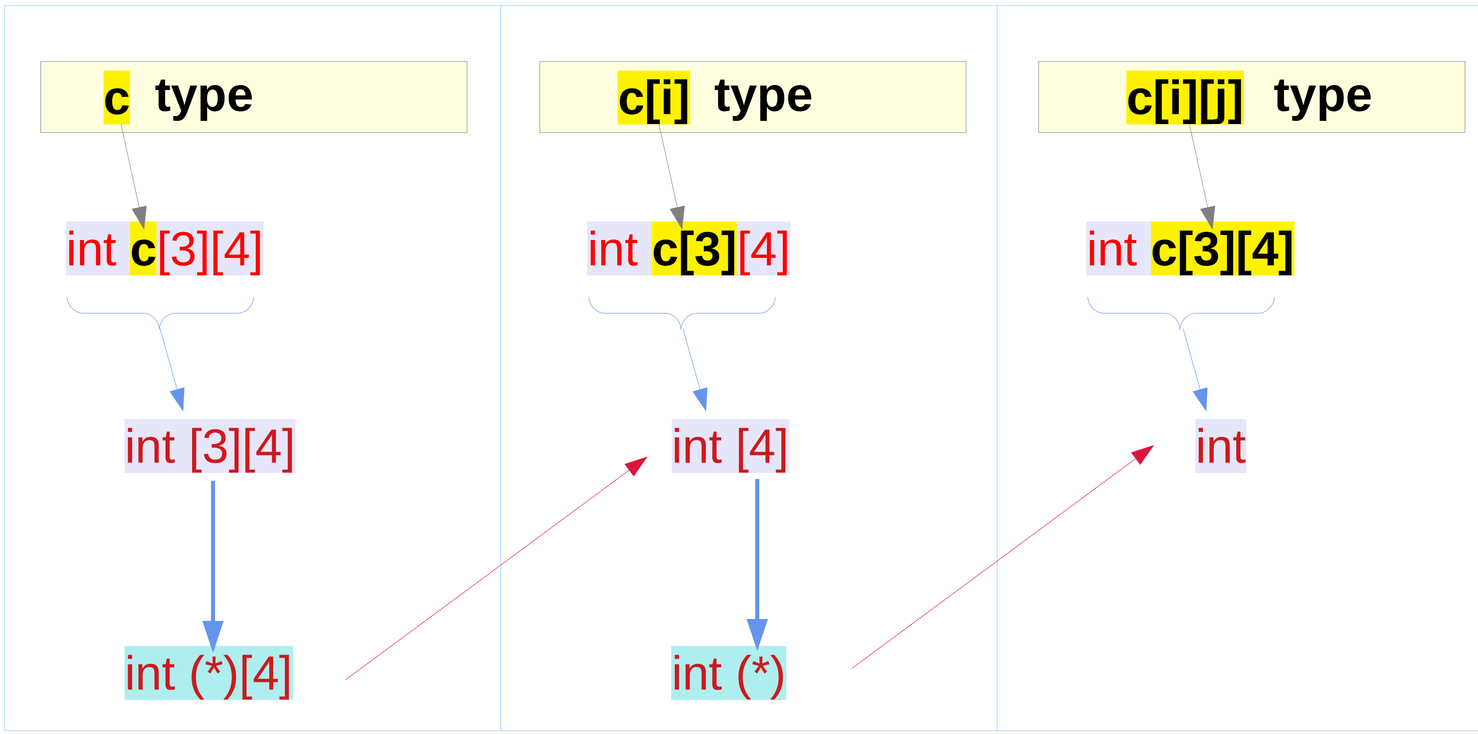
int c[3][4];

2-d array c

Array Type

relaxing the
1^{st} 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] ;**

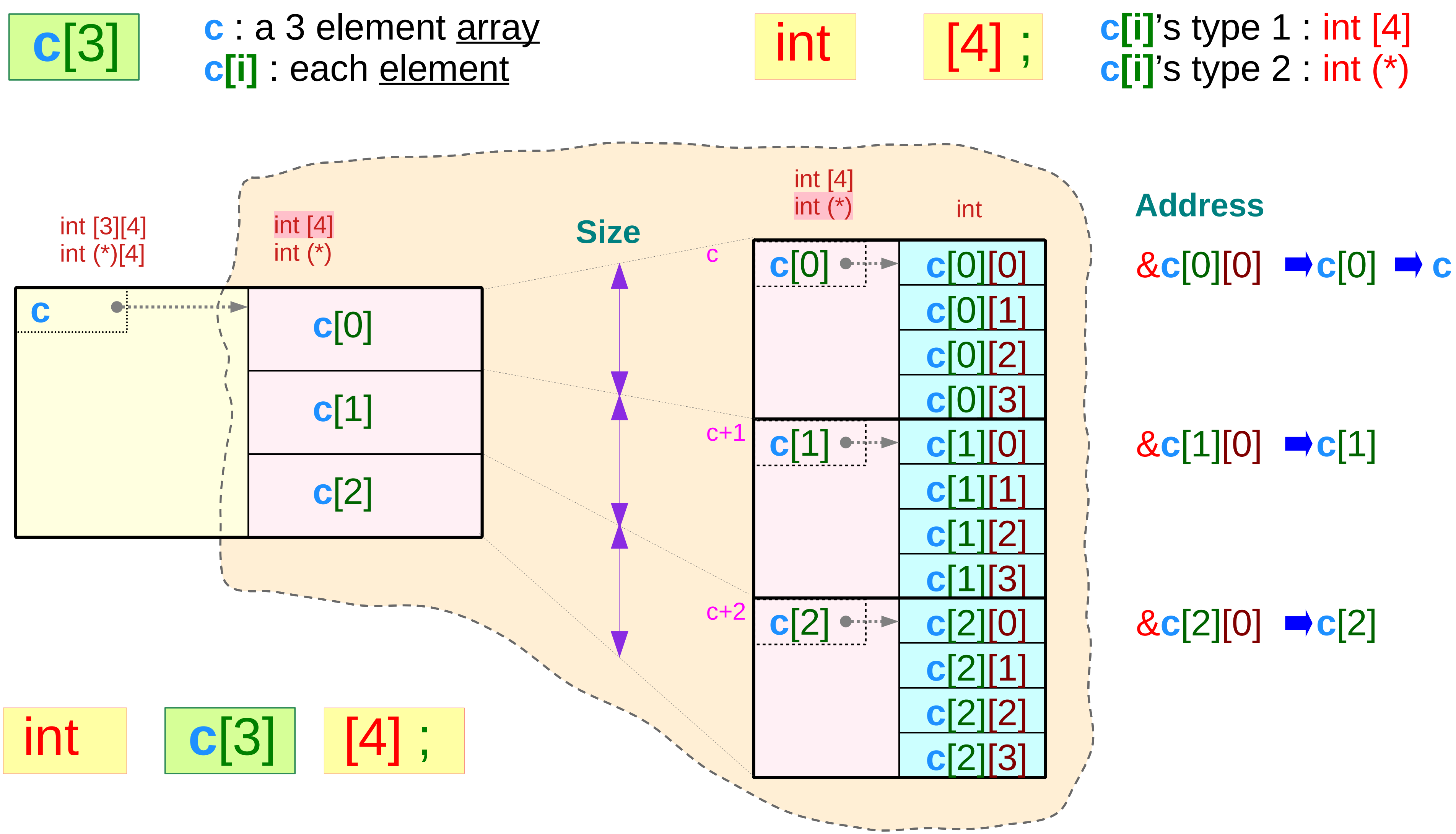
c[i]'s type 1 : int [4]
an array of 4 integers

relaxed dimension

int c[3] [4] ;

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

Nested arrays



c : 3-element array

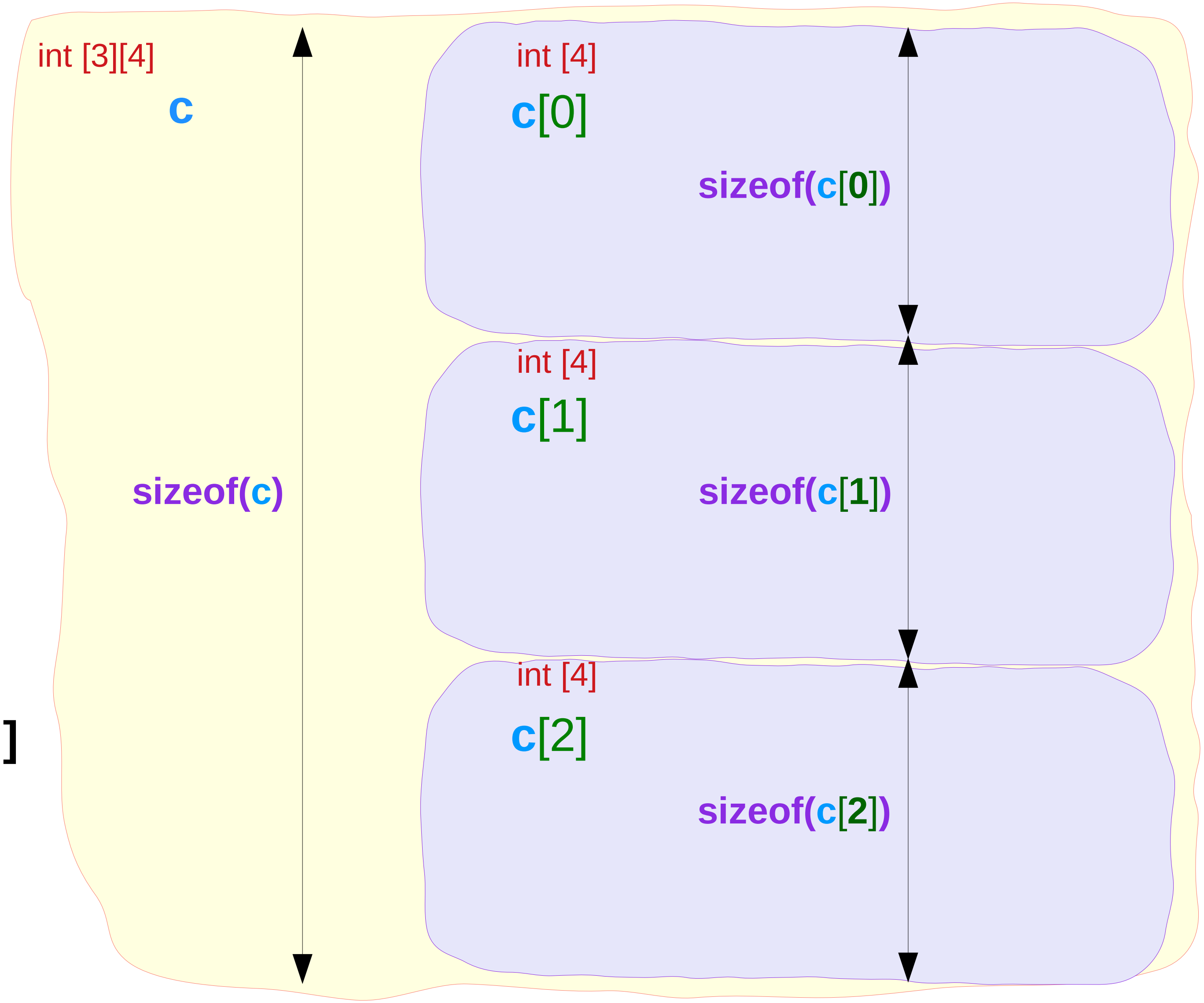
c	**2-d** array	int [3][4]
c[i]	**1-d** array	int [4]

int c [3] [4] ;

3-element array c

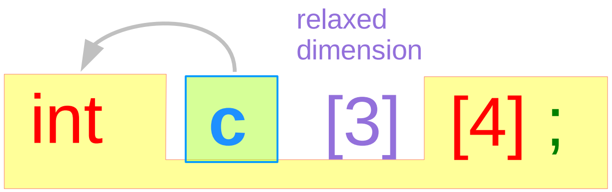
abstract data element **c[i]**

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



c : pointer to a 4-element array

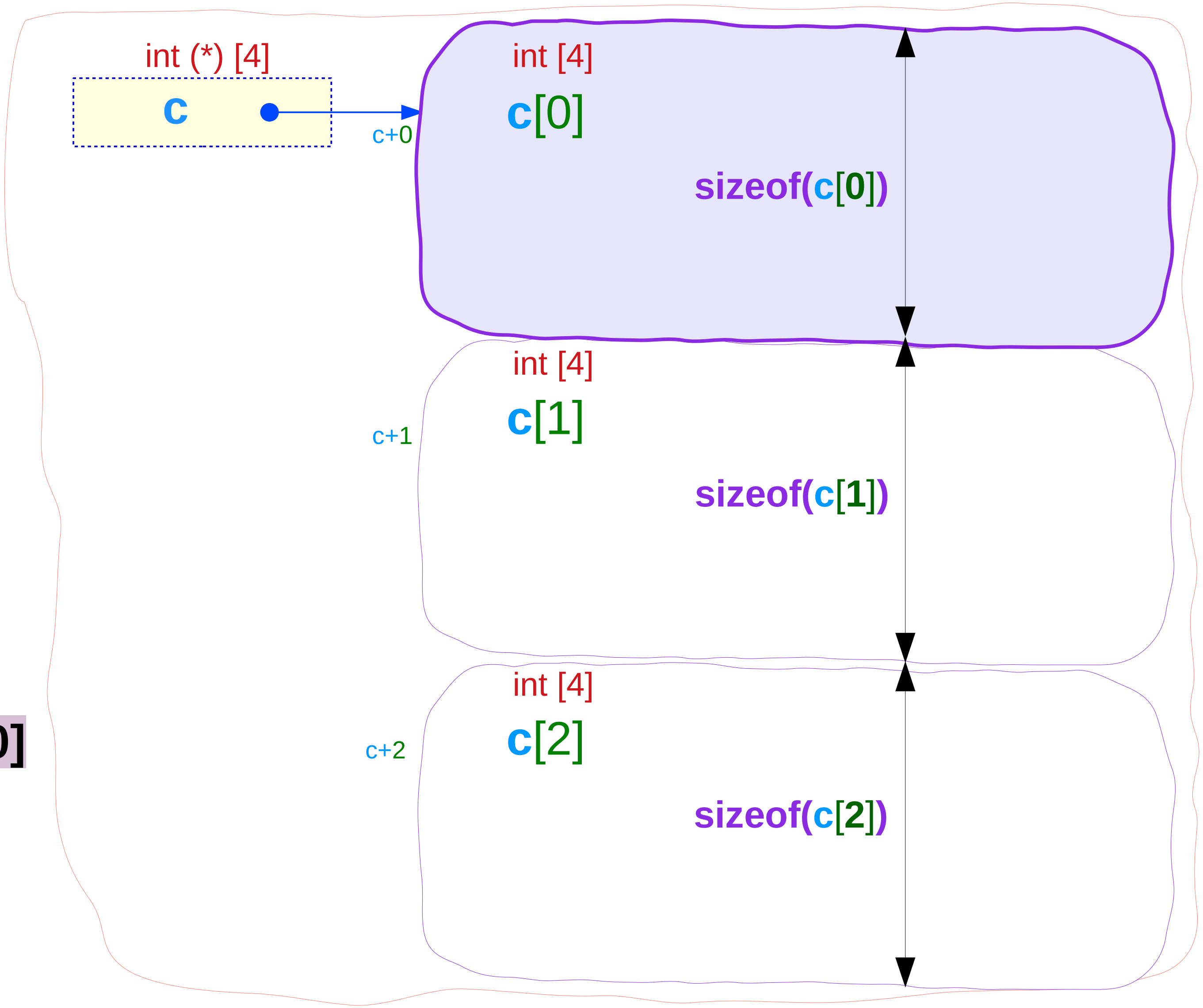
c	**1-d** array pointer	int (*)[4]
c[i]	**1-d** array	int [4]



pointer c

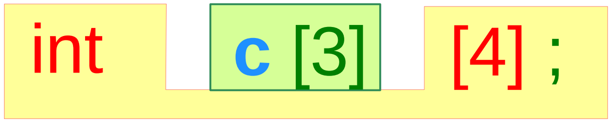
abstract data element **c[0]**

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



c[i] : 4-element array

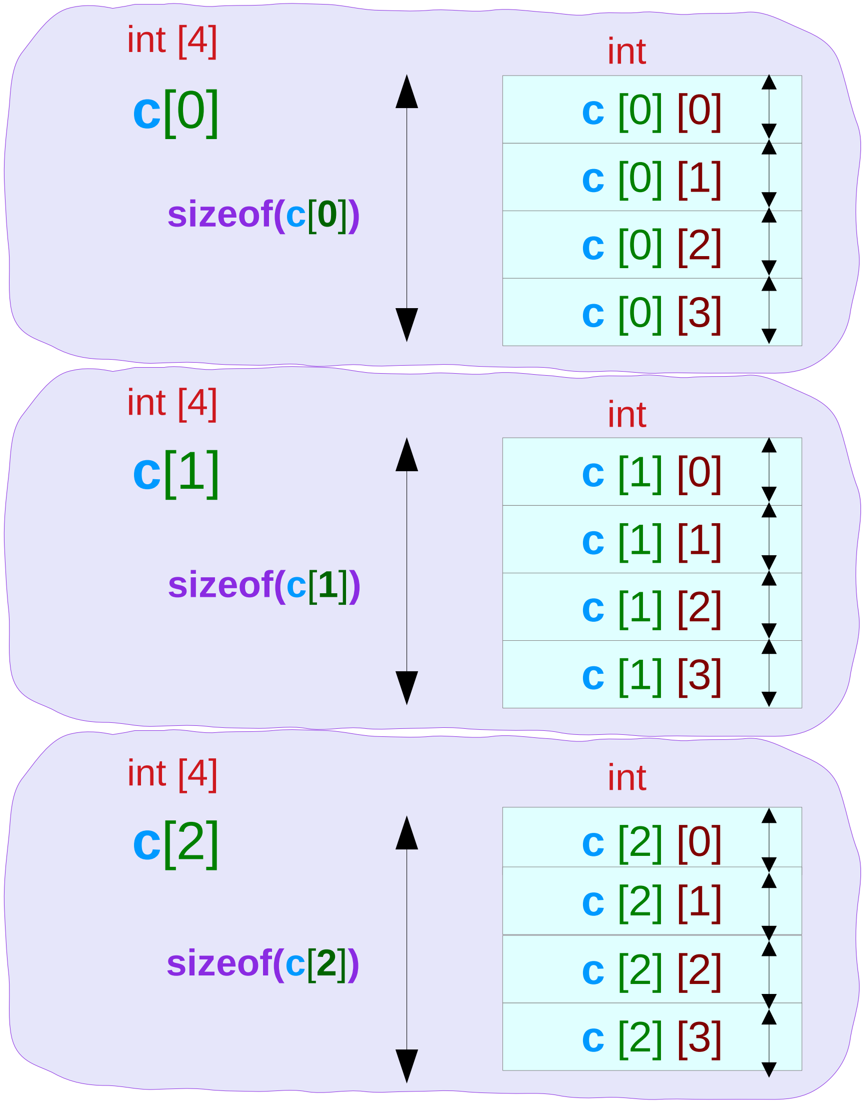
c[i]	**1-d** array	int [4]
c[i][j]	**0-d** array	int



4-element array c[i]

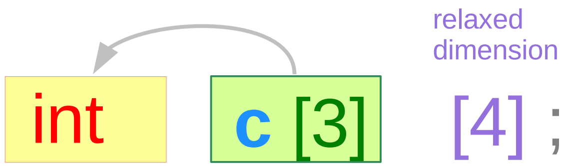
primitive data element **c[i][j]**

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



c[i] : pointer to a primitive data

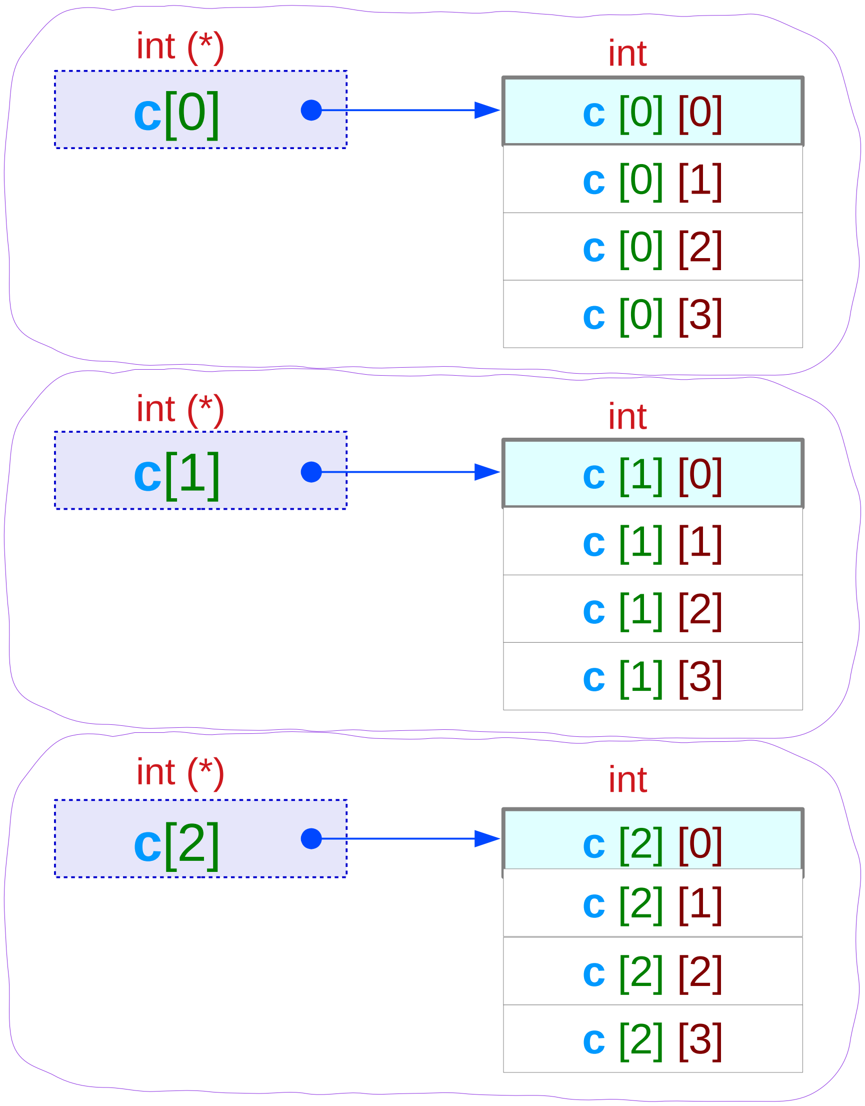
c[i]	**0-d** array pointer	int (*)
c[i][j]	**0-d** array	int



pointer c[i]

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

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

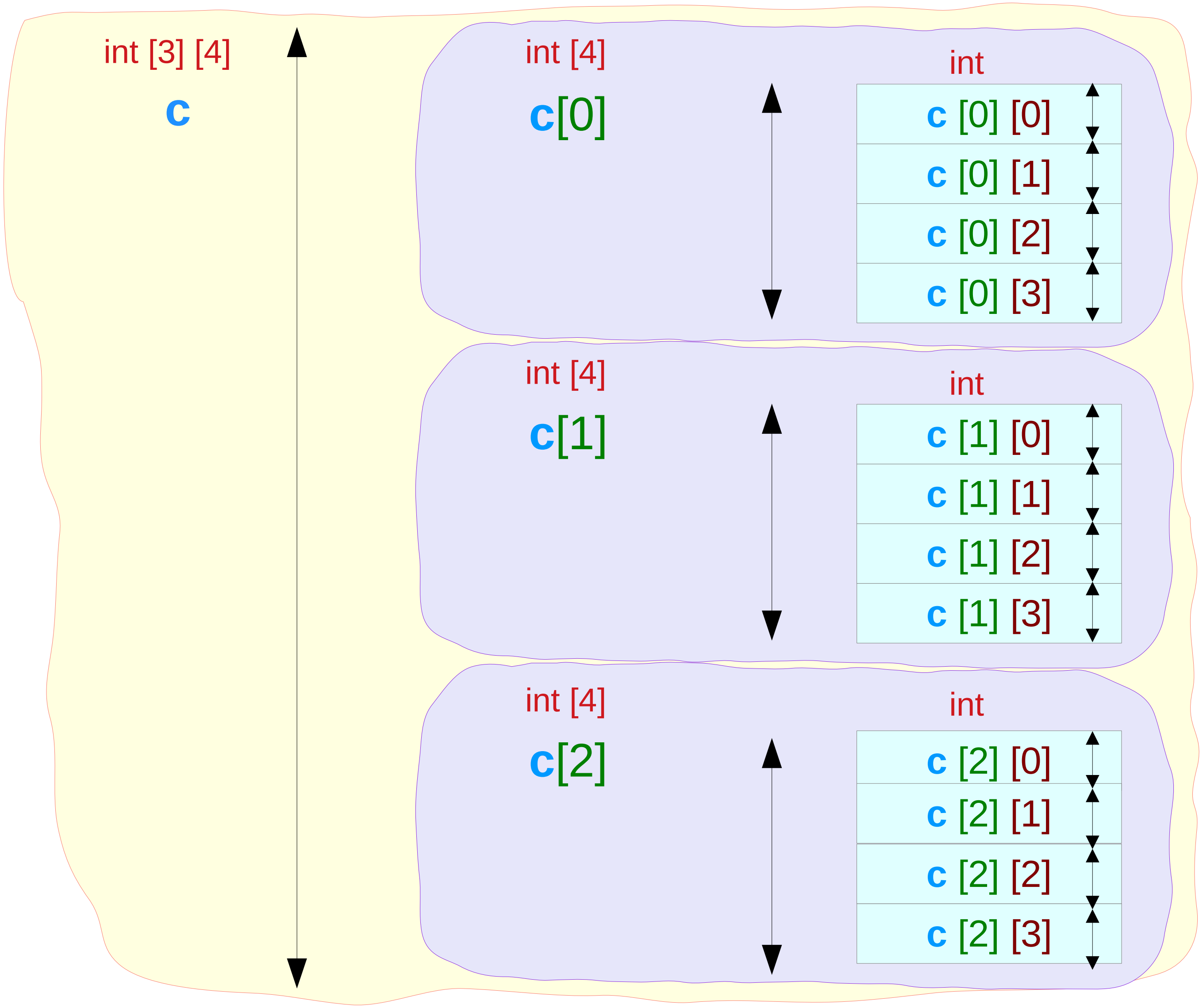


Recursive data view

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] ;

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

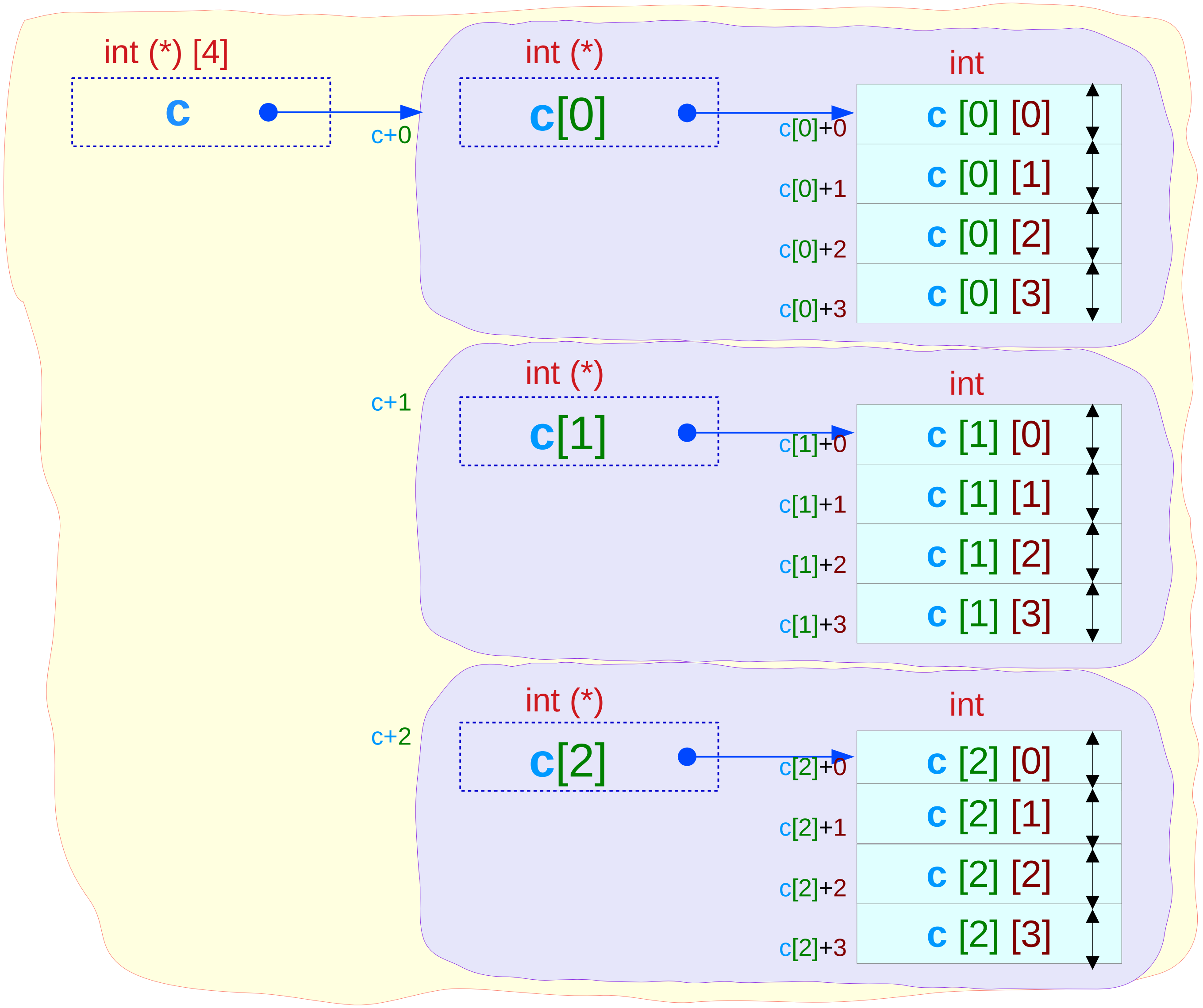
int c[3] [4] ;

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

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

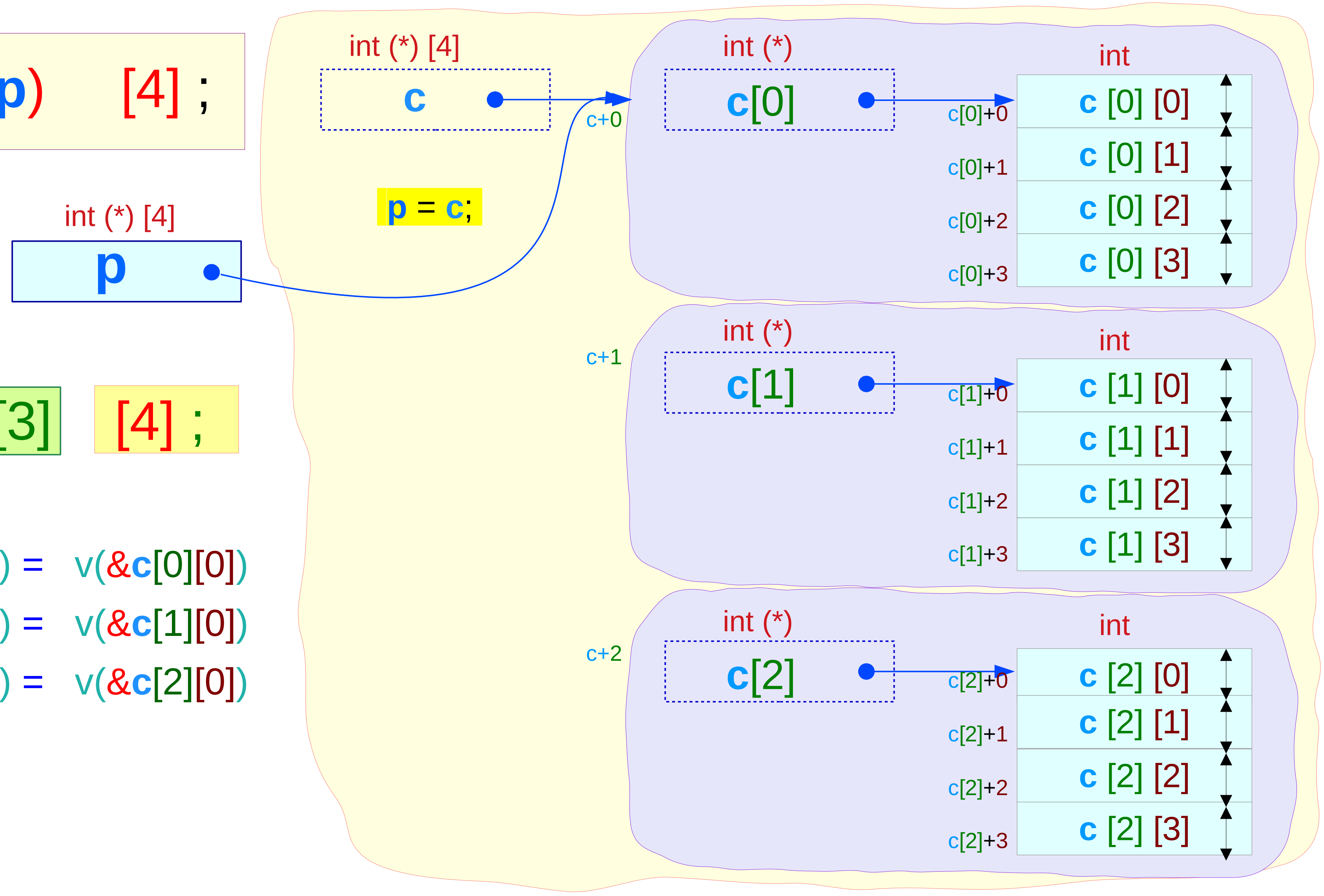
v(c[2]) = v(&c[2][0])

v ≡ 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])

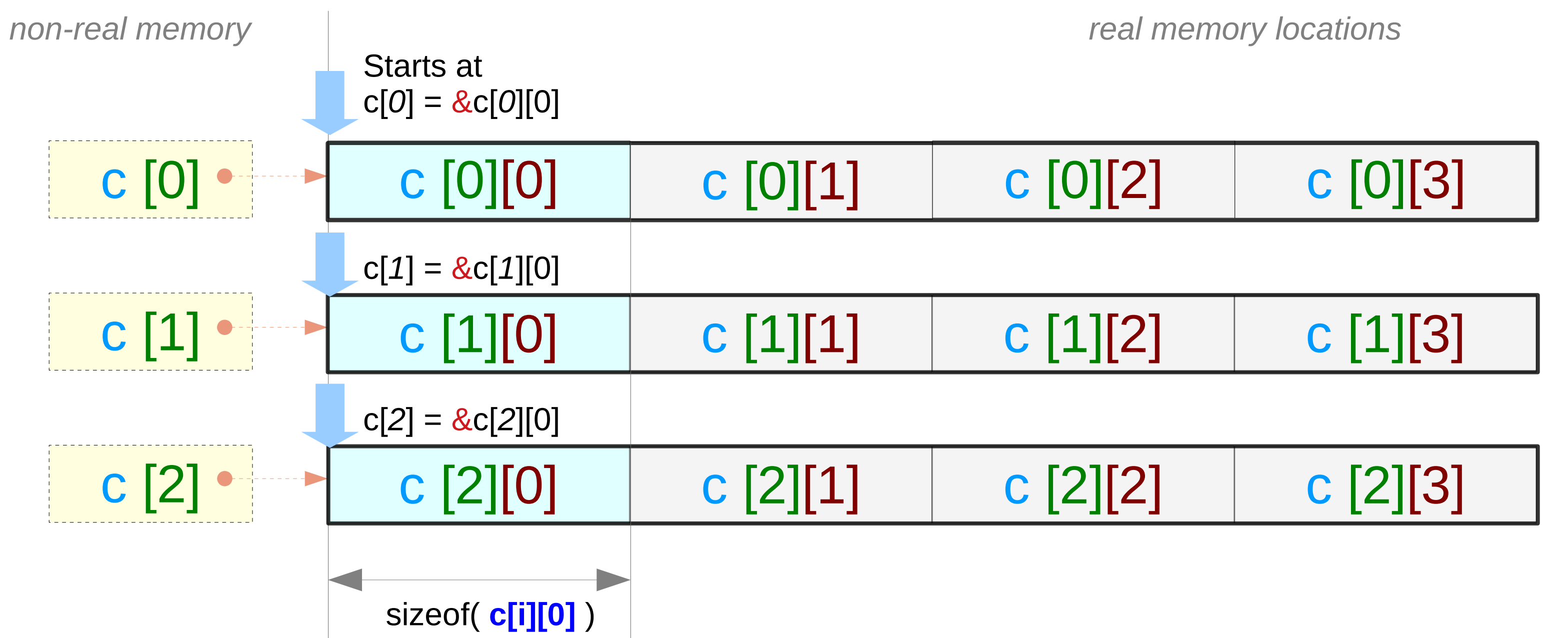
v ≡ value

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

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

non-real pointer **c[i]** : value(**c[i]**)= &**c[i][0]**

0-d array pointer



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

integer c[i][0]
0-d array : int

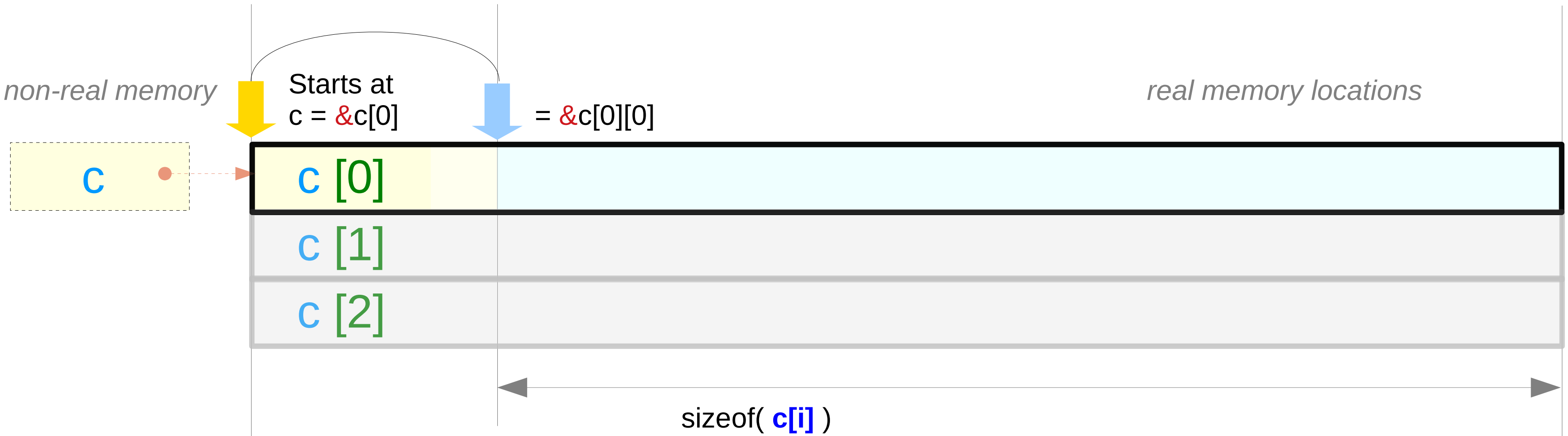
c[i] points to the 1st element **c[i][0]**

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

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

non-real pointer **c** : value(**c**) = &**c[0]** = &**c[0][0]** — **1-d** array pointer

abstract data **c[i]** : sizeof(**c[i]**) = 4 * sizeof(**int**) — **1-d** array



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

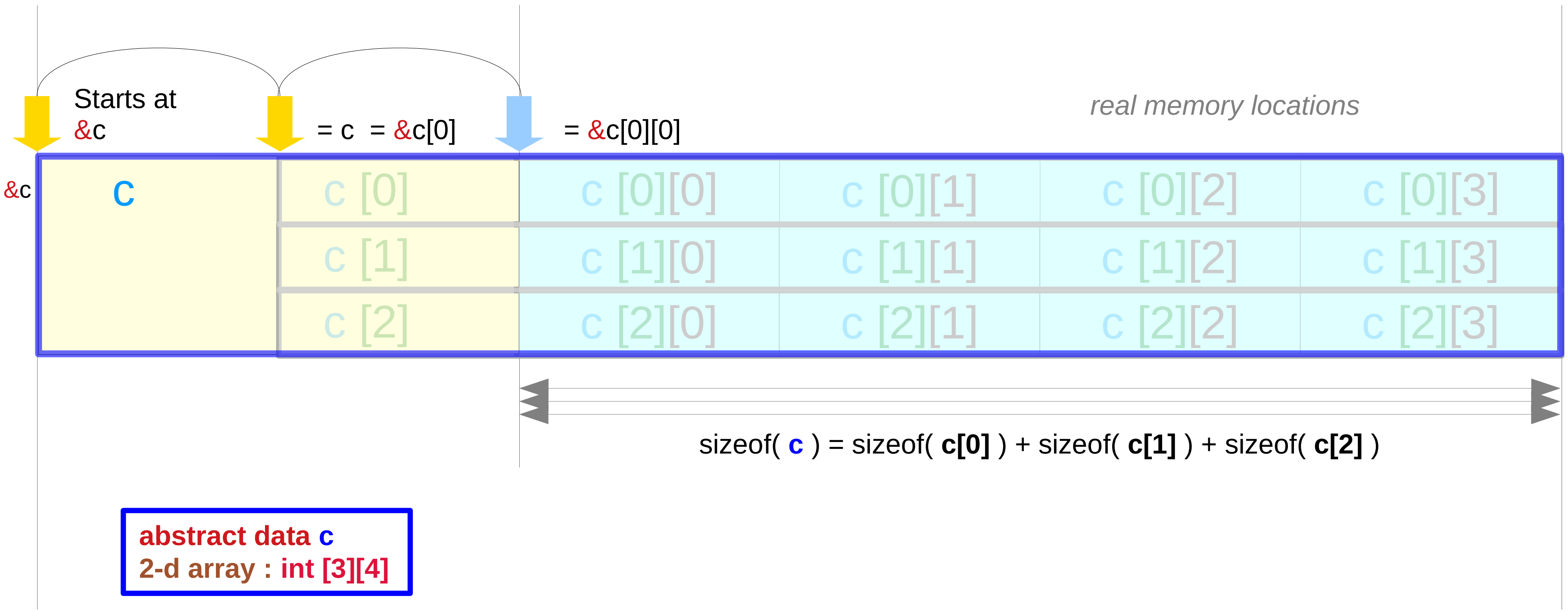
abstract data c[i]
1-d array : int [4]

c points to the 1st element **c[0]**

Abstract data **c**

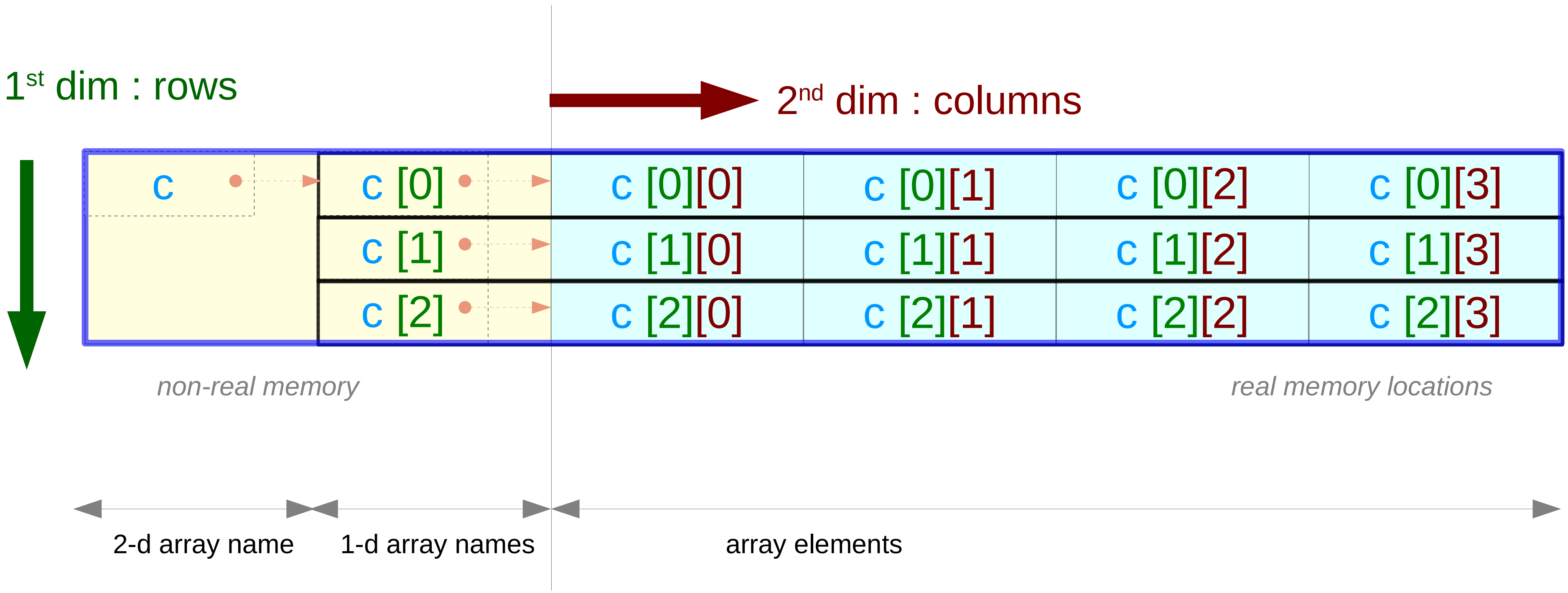
```
int c [3] [4];
```

abstract data **c**: sizeof(**c**) = 3 * sizeof(**c[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]**

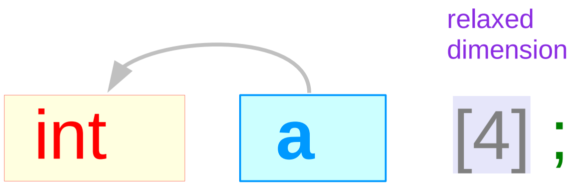
sizeof(**c[i]**) = 4 * 4

int c [3] [4]

c is the name of the **2-d** array **int [3][4]**

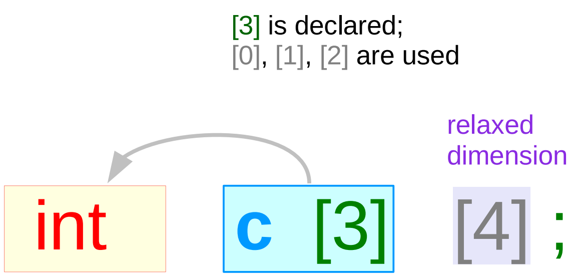
sizeof(**c**) = 3 * 4 * 4

Values of array names



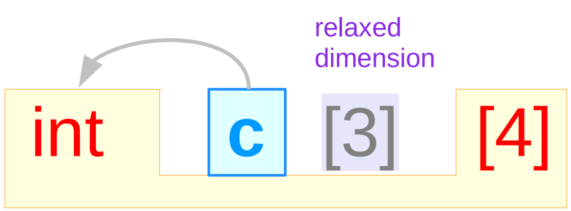
the value of **a** is the starting address **int (*)**
of an array with 4 elements of **int** type

a: pointer to the first element **a = &a[0]**



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

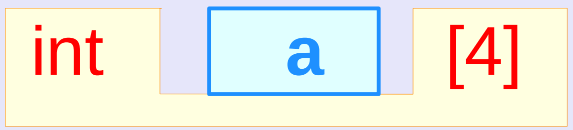
c[i]: pointer to the first element **c[i] = &c[i][0]**



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

c: pointer to the first element **c = &c[0]**

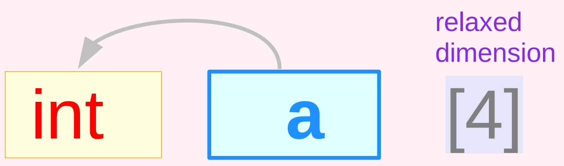
Array and pointer types in a **1-d** array



a **1-d array**

type : int [4]

size : 4 * 4

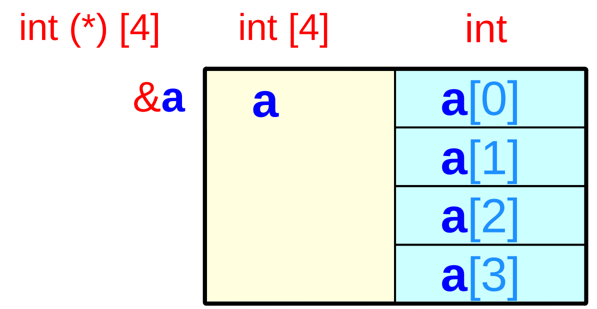


a **0-d array pointer**

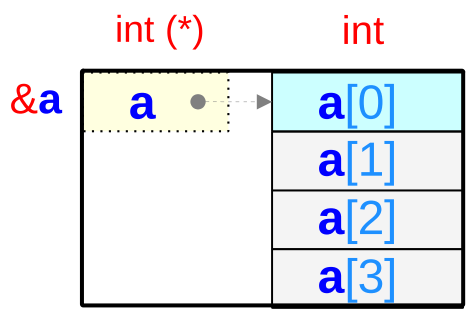
type : int (*)

value : &a[0]

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

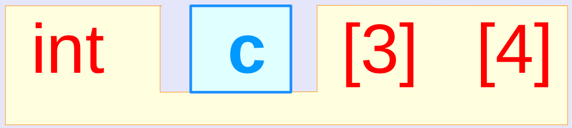


outside of an array a



inside of an array a

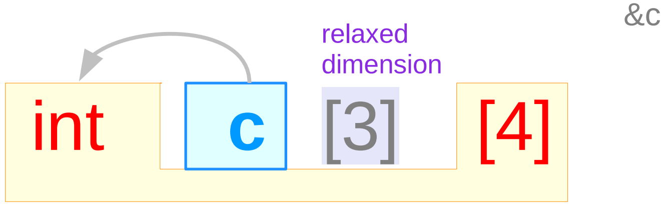
2-d array type



c **2-d array**

type : int [3][4]

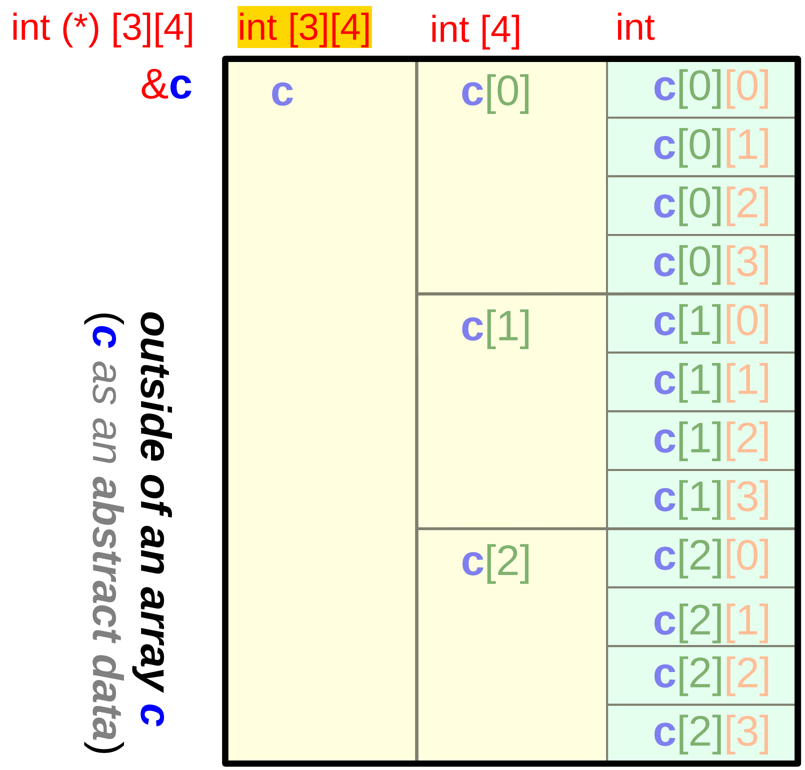
size : 3 * 4 * 4



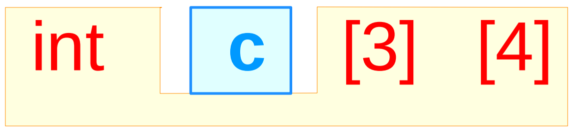
c **1-d array pointer**

type : int (*) [4]

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



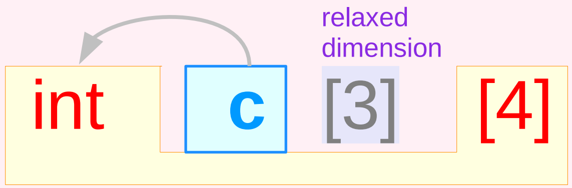
1-d array pointer type



c **2-d array**

type : int [3][4]

size : 3 * 4 * 4

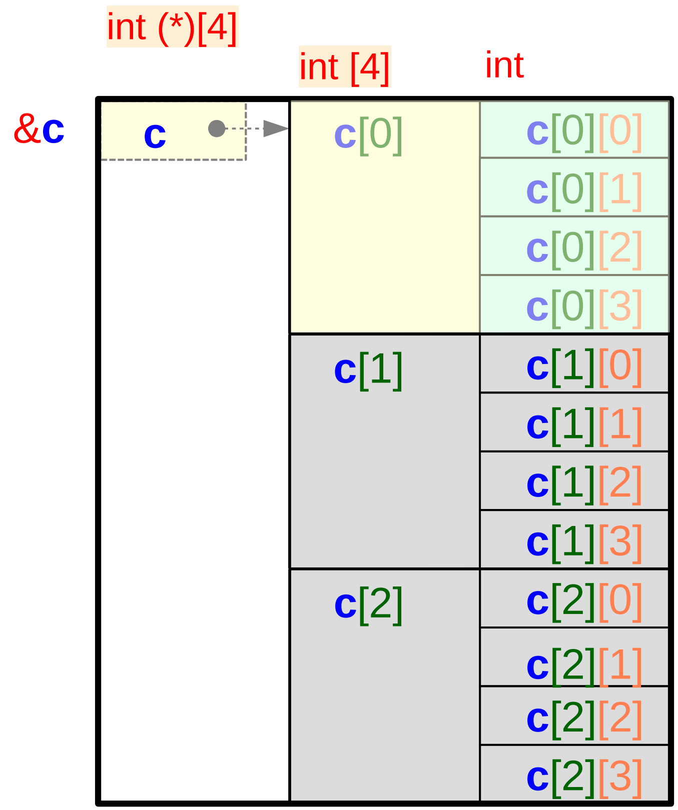


c **1-d array pointer**

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

int c [3] [4]

c[i] 1-d array

type : int [4]

size : 4 * 4

relaxed dimension

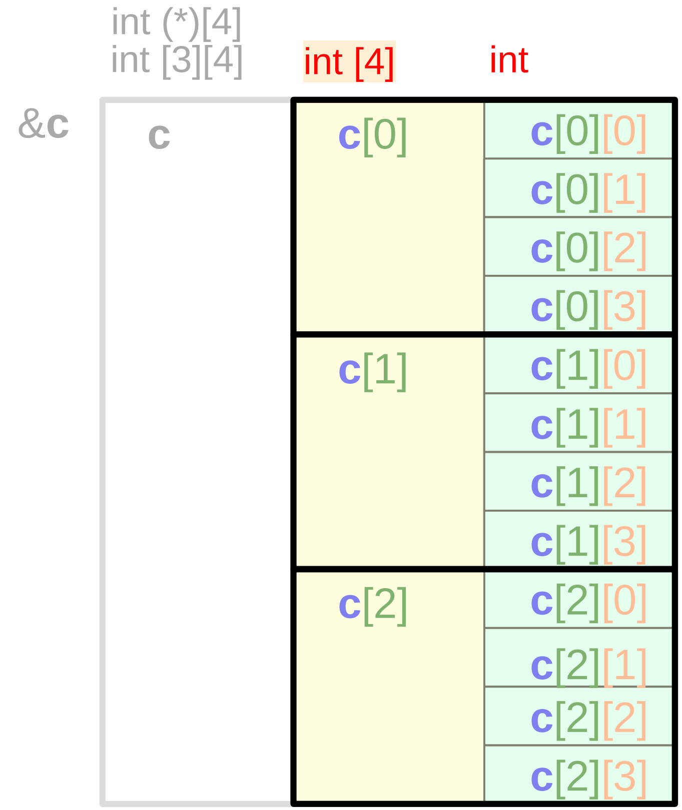
int c [3] [4]

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

int c [3] [4]

c[i] 1-d array

type : int [4]

size : 4 * 4

int c [3] [4]

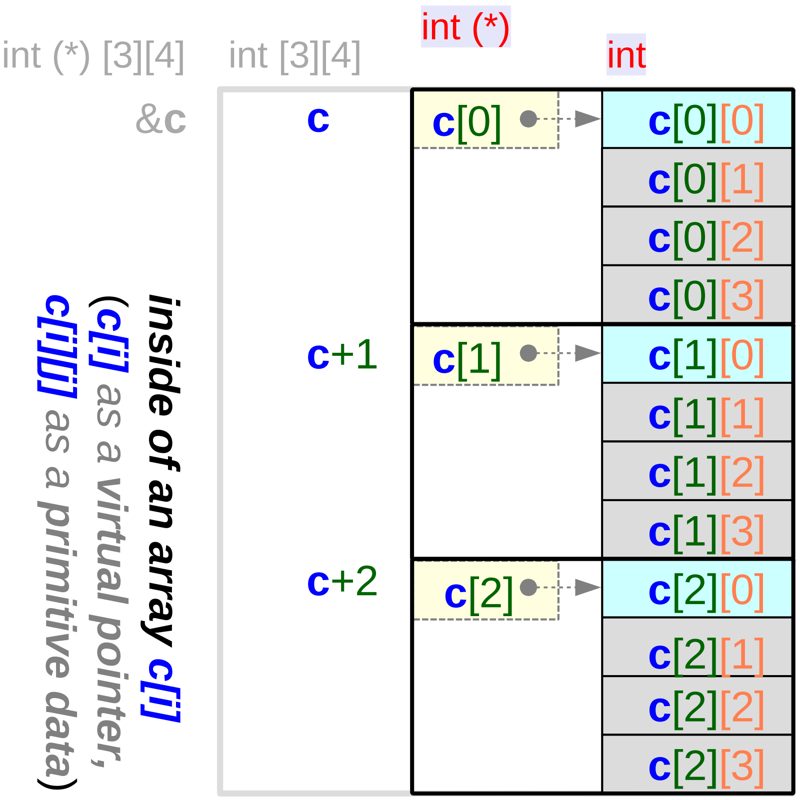
relaxed dimension

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 →

relaxed 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 →

relaxed 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)

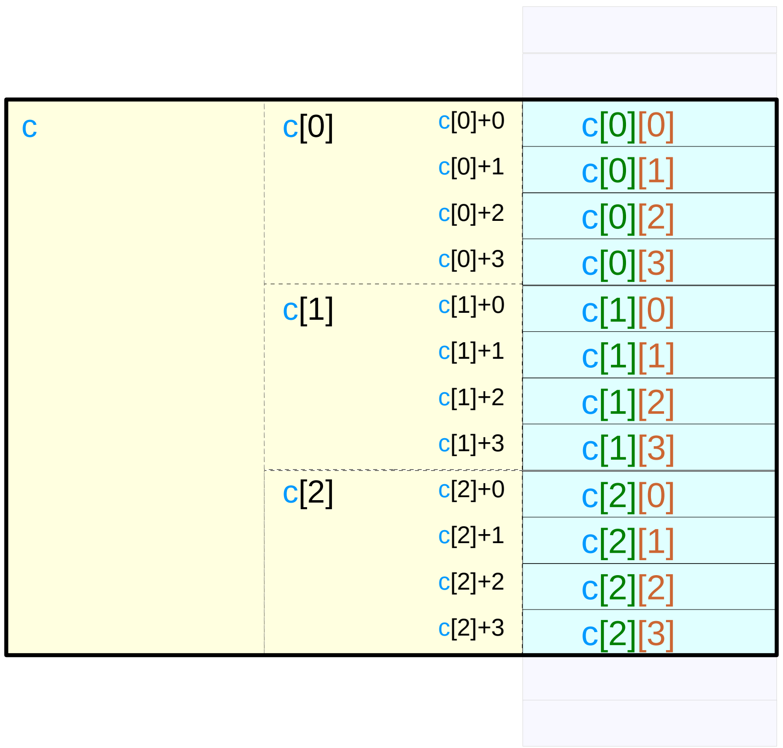
2. a double pointer

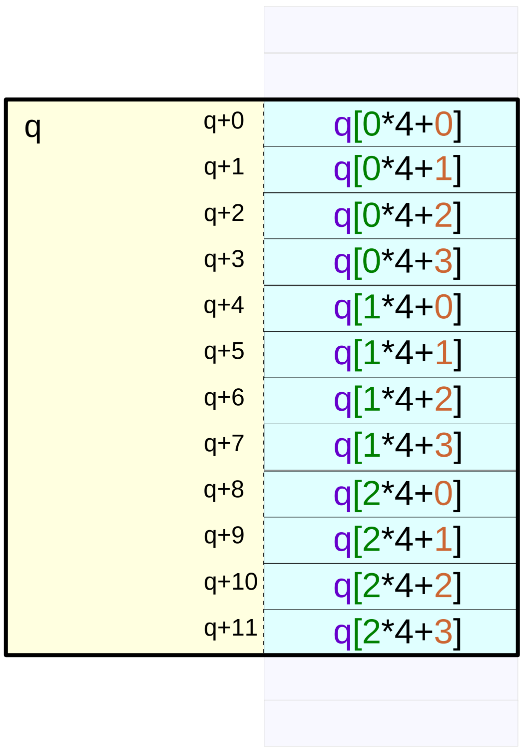
3. a pointer to an array

2-d array **c** and **1-d** array **q**

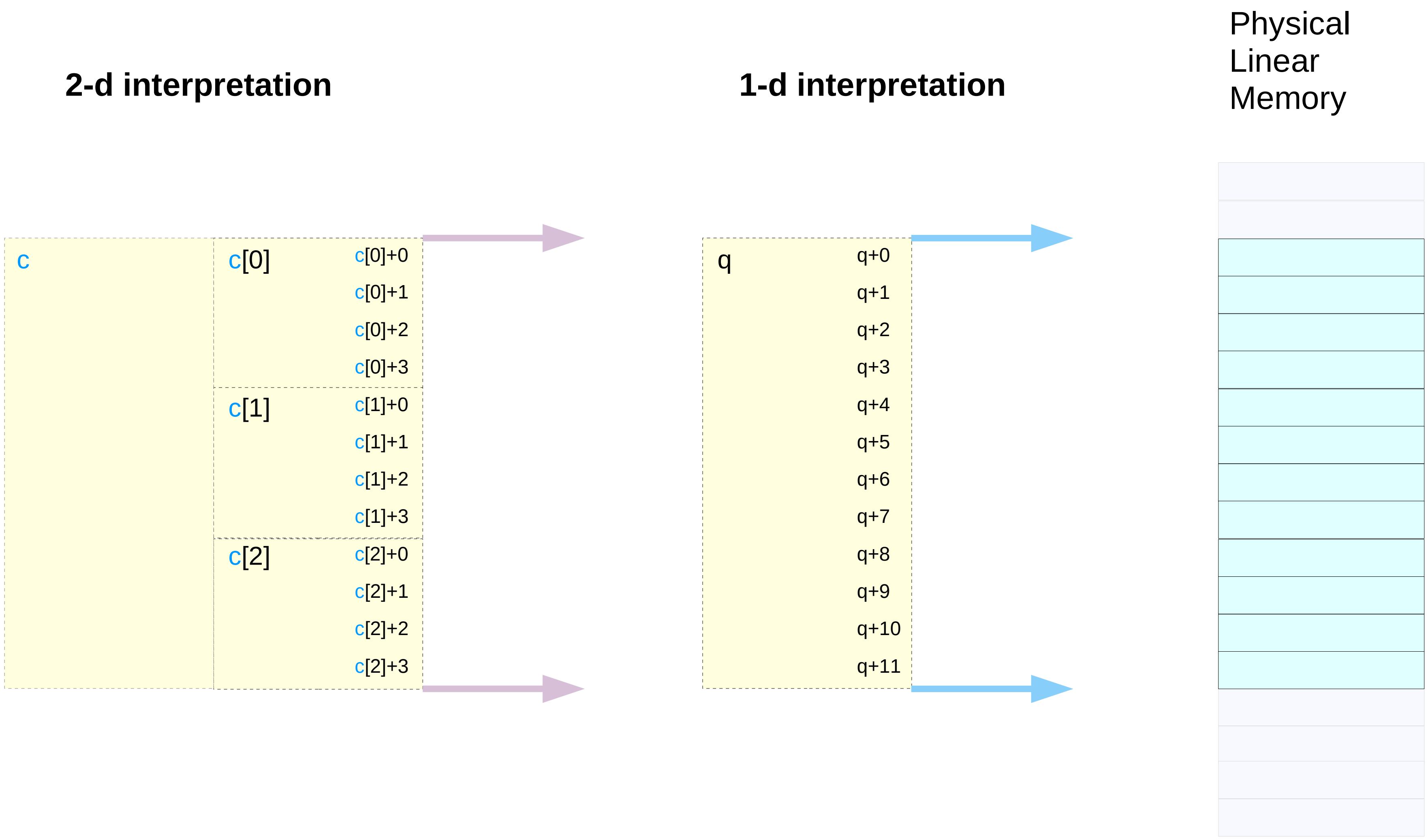
```
int c [3] [4];
```

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





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]

index values

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

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)
```

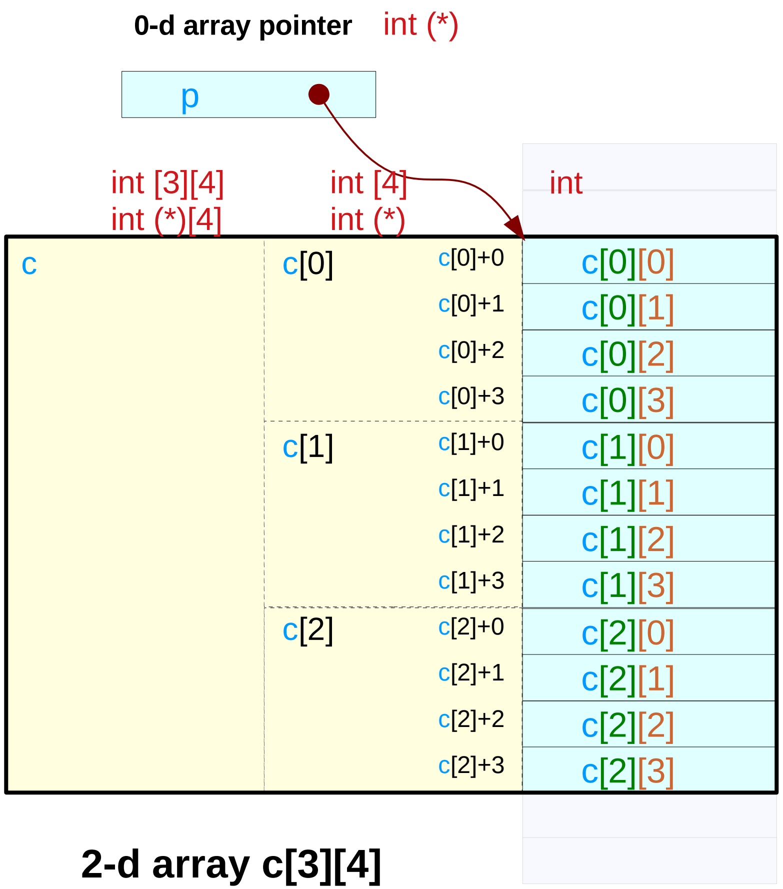
```
*(p +k)        i = k / 4;
               j = k % 4;
```

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

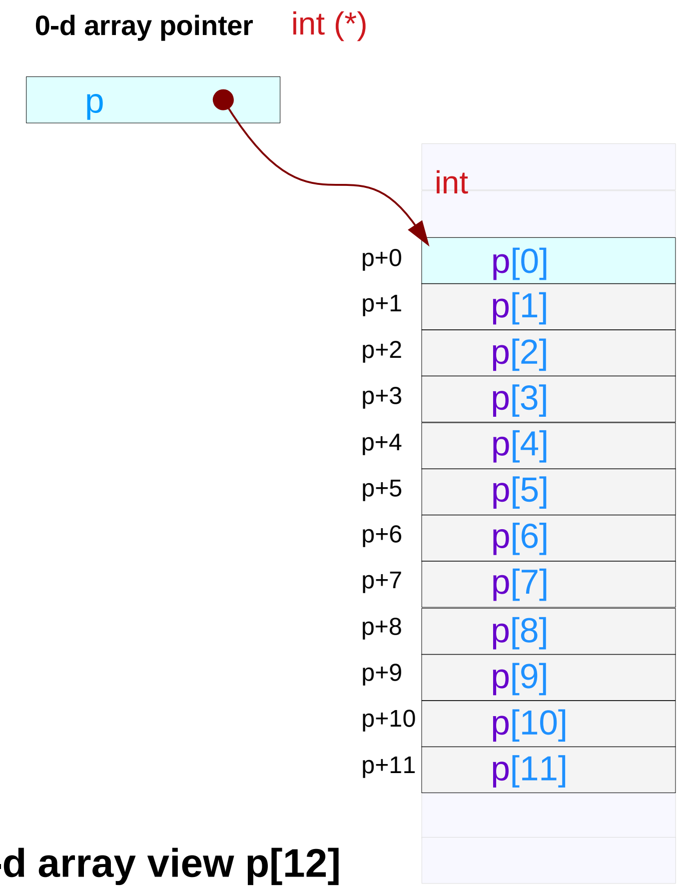
int c [3][4];

int *p = c[0];

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

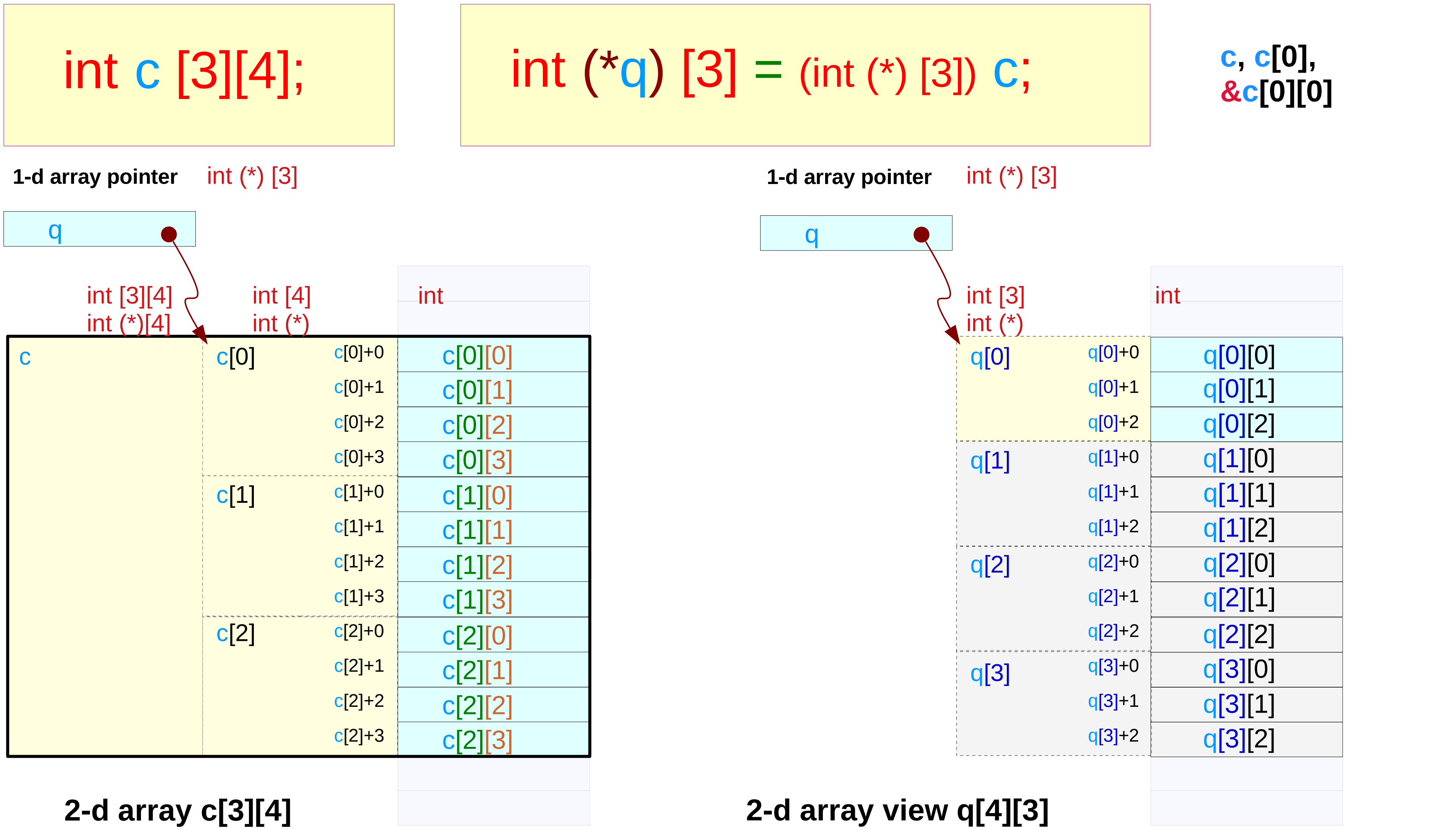


2-d array c[3][4]



1-d array view p[12]

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



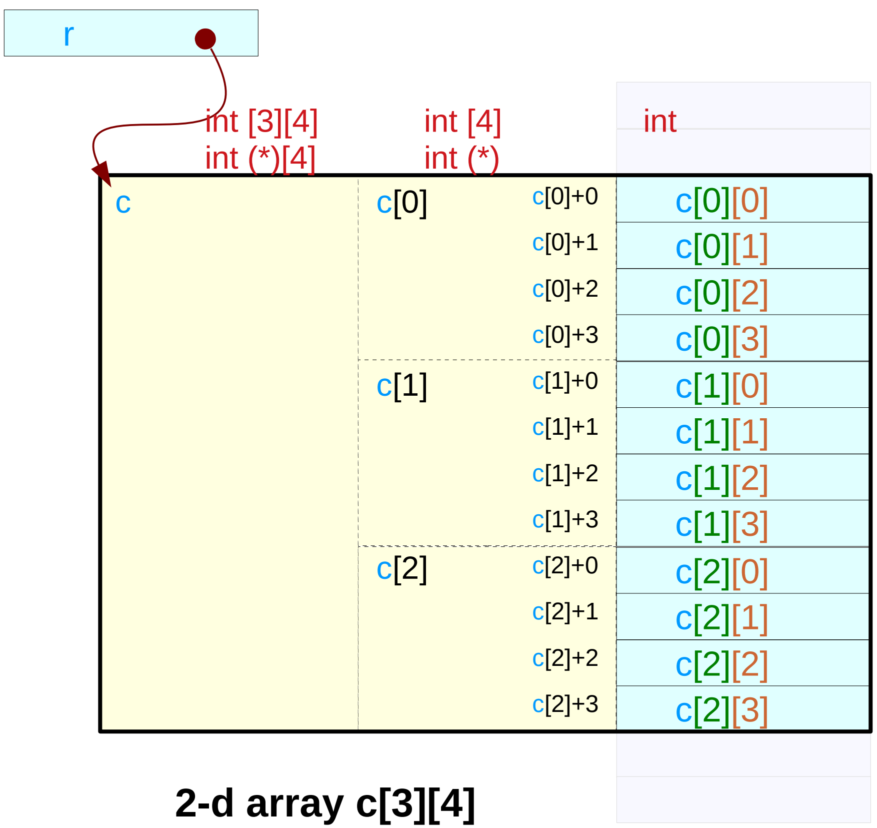
A **2-d** array stored as a **1-d** array (row major order)

`int c [3] [4];`

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

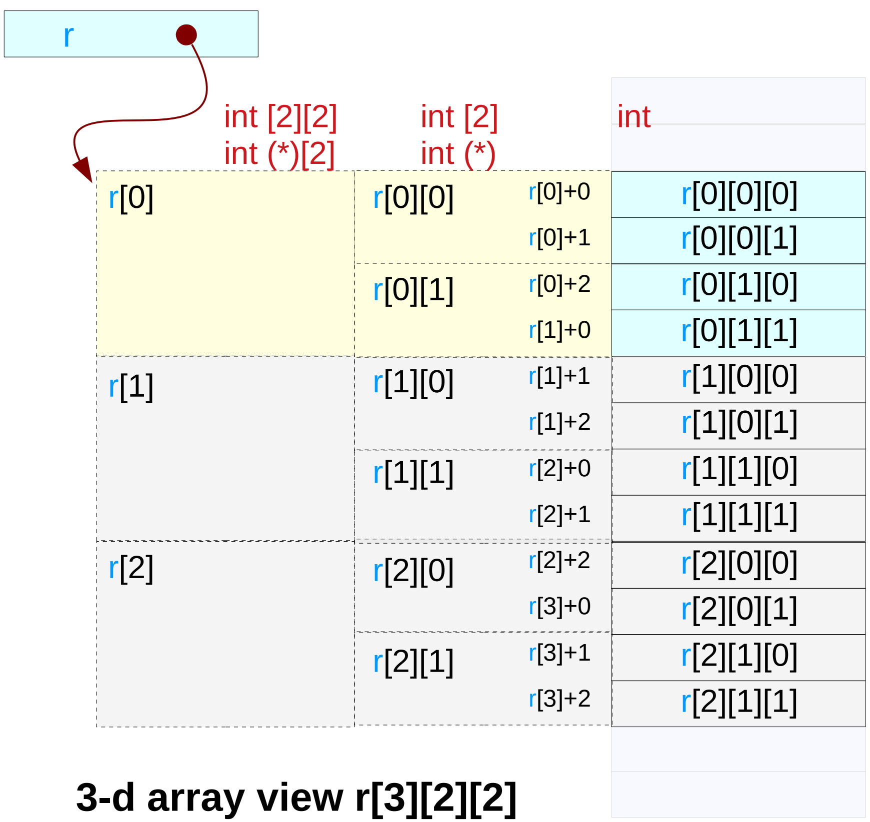
c, c[0], &c[0][0]

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



2-d array c[3][4]

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



3-d array view r[3][2][2]

2-d array access via pointers

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

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

1. recursive pointers

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

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

⇒ `int (*p)[4];`

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

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

⇒ `int **q;`

2. linear array pointers

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

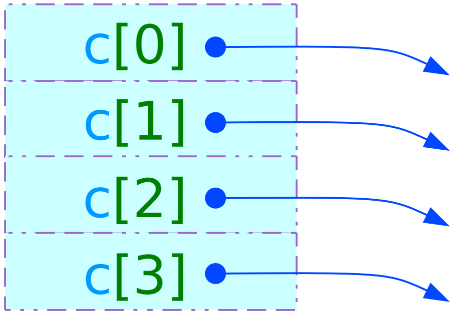
```
*(p+ i*4 + j )
```

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[ X_A+4(3i +j) ]
```

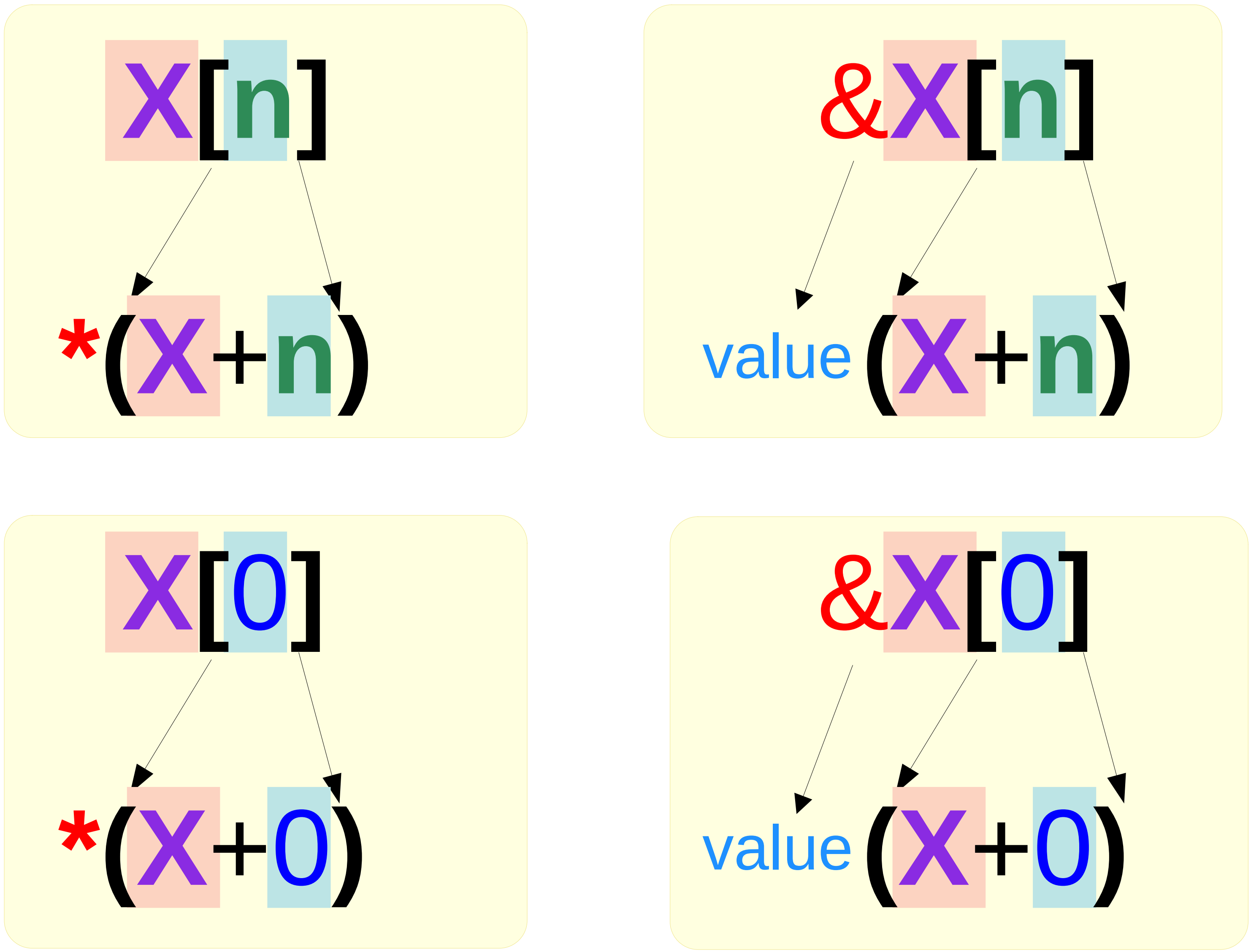


The pointer array :
not allocated
in the memory

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

Equivalences between *****, **&**, and **[ ]** operators



Operator Precedence of * and [ ]

*x[m]	≡	*(x[m])	[ ] has a higher priority than *
x[m][n]	≡	(x[m])[n]	[ ] has left-to-right associativity
**x	≡	*(*x)	* has right-to-left associativity

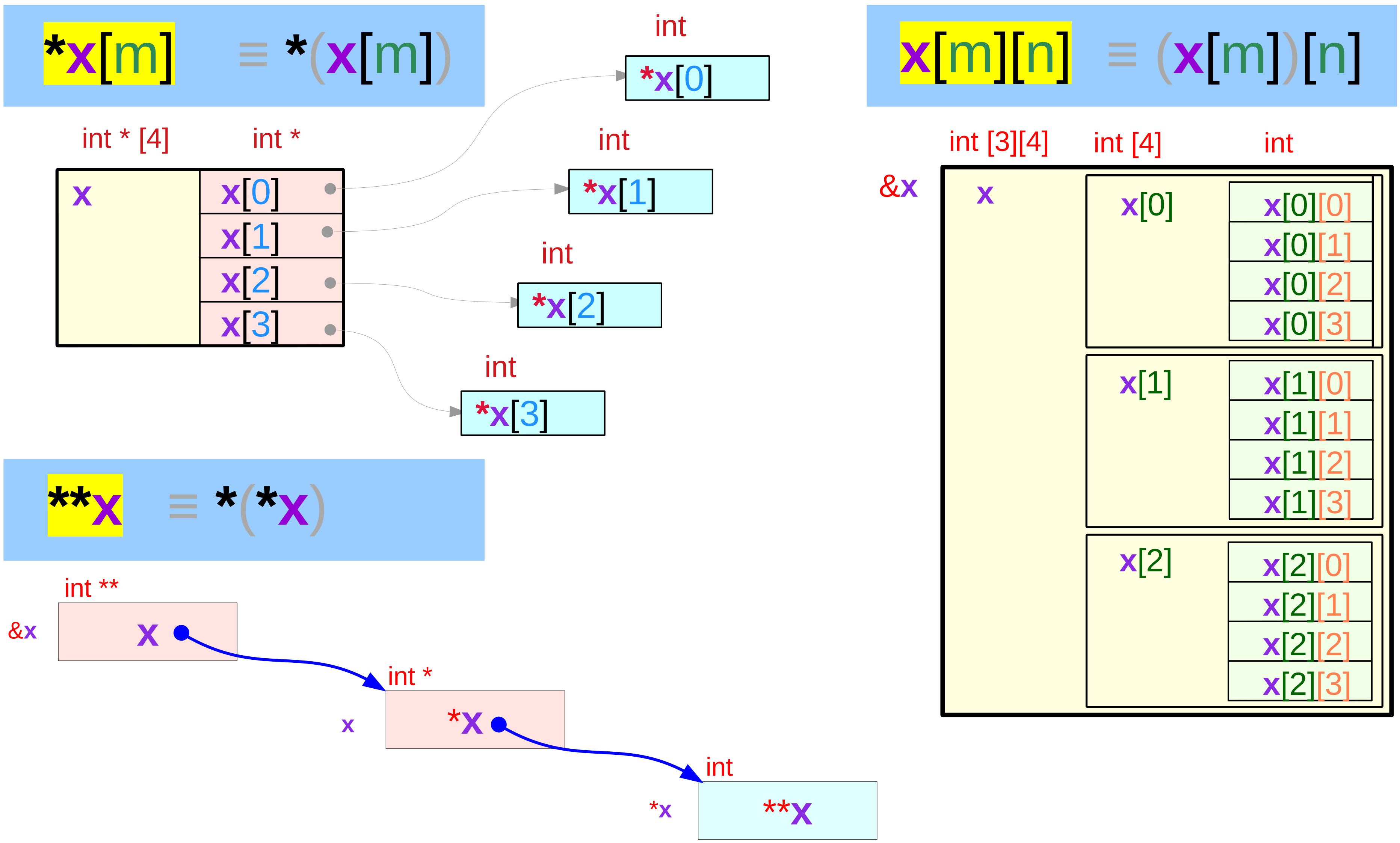
(*x)[m][n] ⟺ ((*x)[m])[n]

(*x[m])[n] ⟺ (*(x[m]))[n]

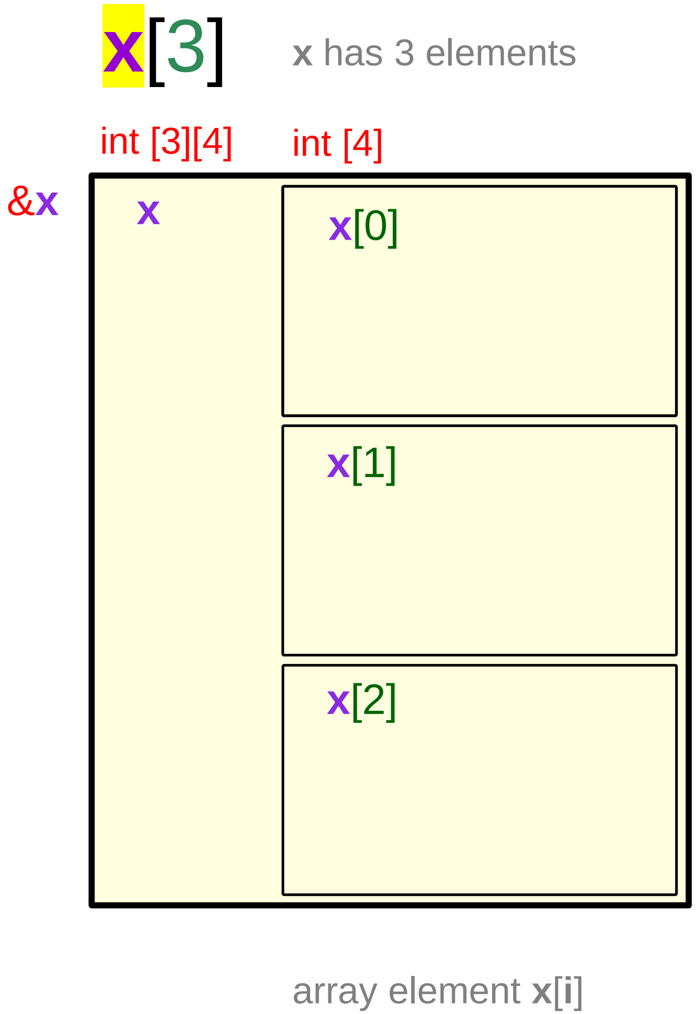
red parentheses ( ) must not be removed

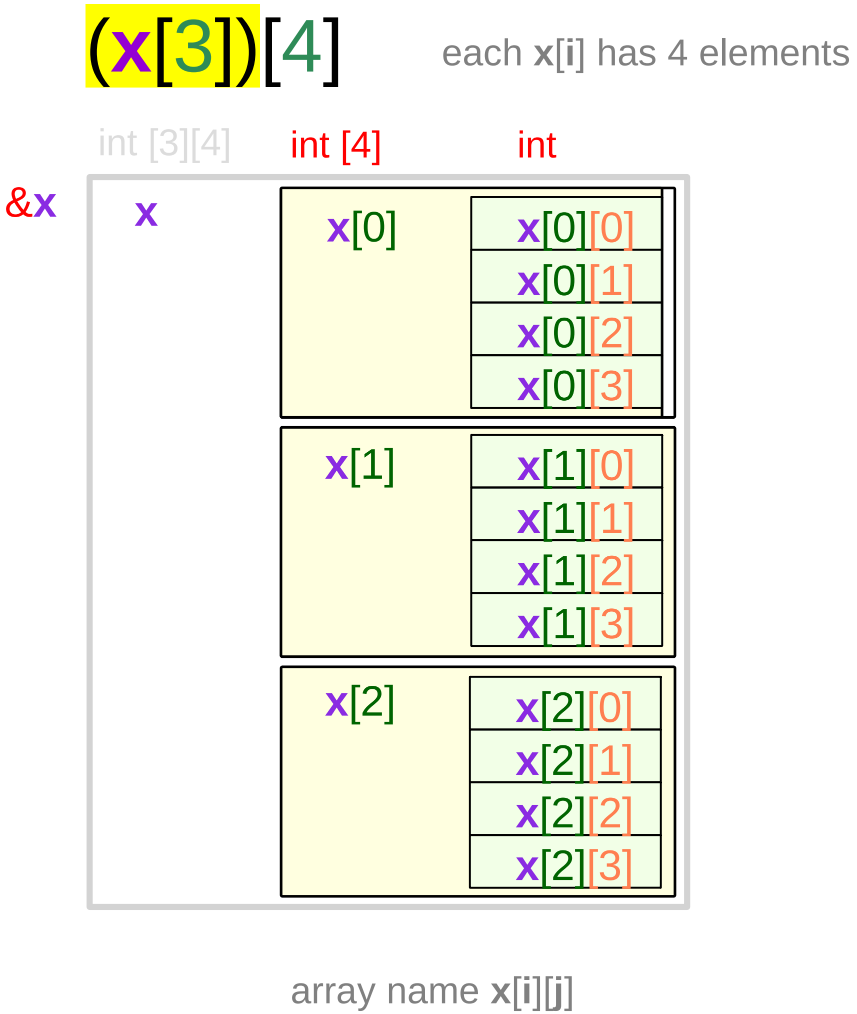
gray parentheses ( ) can be removed

Operator Precedence of * and [ ]

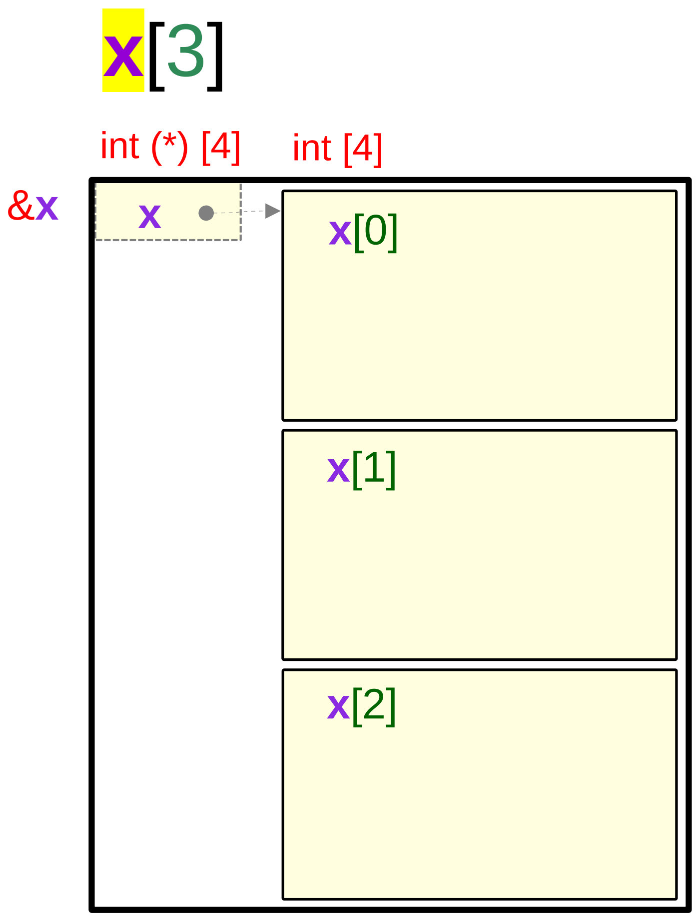


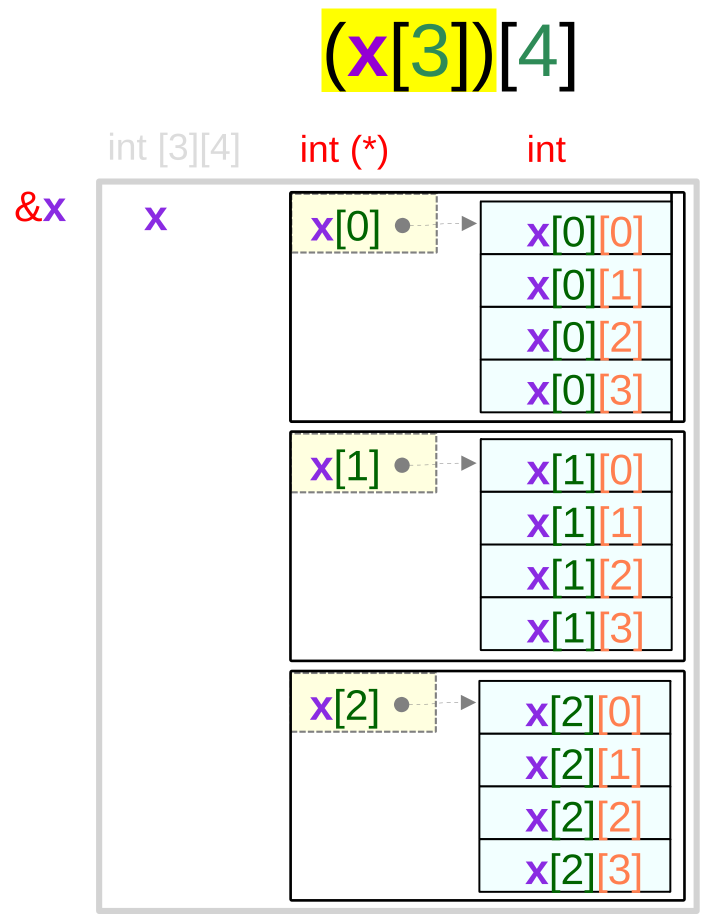
Abstract Data **x** and **x[i]**





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

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

p[i]	≡	p[i]	⇒	*(p+i)
p[i][j]	≡	(p[i])[j]	⇒	*(*(p+i)+j)
p[i][j][k]	≡	((p[i])[j])[k]	⇒	*(*(*(p+i)+j)+k)

*p	≡	*(p)	⇒	p[0]
**p	≡	*(*(p))	⇒	(p[0])[0]
***p	≡	*(*(*(p)))	⇒	((p[0])[0])[0]

Equivalence on relaxing outer most dimension

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

&p[i]	≡	value(p+i)
&p[i][j]	≡	value(p[i]+j)
&p[i][j][k]	≡	value(p[i][j]+k)

p[0]	≡	*p
p[i][0]	≡	*p[i]
p[i][j][0]	≡	*p[i][j]

&p[0]	≡	value(p)
&p[i][0]	≡	value(p[i])
&p[i][j][0]	≡	value(p[i][j])

valid for proper **i**, **j**, **k** values

Relaxing all the dimensions

p[i]	⇄	*(p+i)	➡ *(p+i)
p[i][j]	⇄	*(p[i]+j)	➡ *(*(p+i)+j)
p[i][j][k]	⇄	*(p[i][j]+k)	➡ *(*(*(p+i)+j)+k)

*(X+n) ≡ X[n]

*(X+n)

X[n]

valid for proper **i**, **j**, **k** values

Equivalences on relaxing all the dimensions

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

&p[i]	≡	value(p+i)
&p[i][j]	≡	value(*(p+i)+j)
&p[i][j][k]	≡	value(*(*(p+i)+j)+k)

p[0]	≡	*p
p[0][0]	≡	**p
p[0][0][0]	≡	***p

&p[0]	≡	value(p)
&p[0][0]	≡	value(*p)
&p[0][0][0]	≡	value(**p)

valid for proper **i**, **j**, **k** values

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)

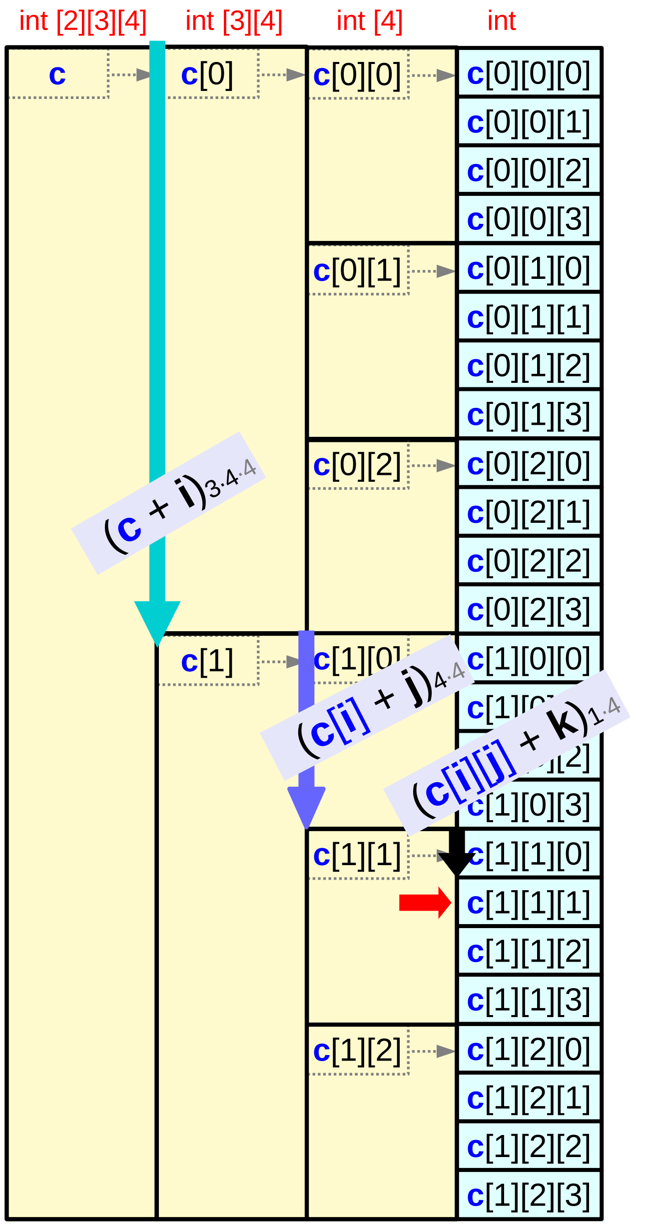
address replication

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] ;
```

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]

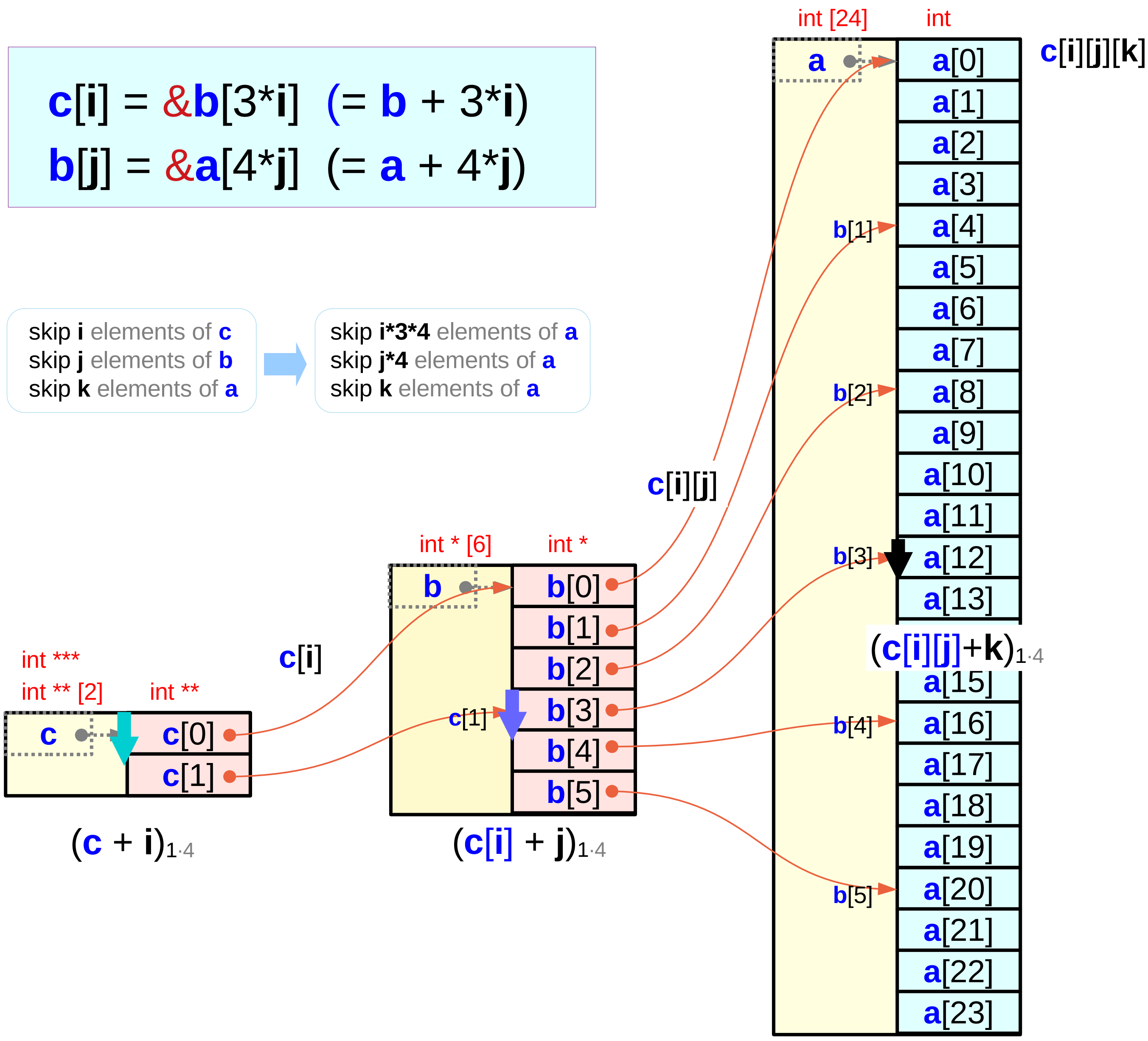
c[i][j] ≡ (a+(i*3+j)*4)

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

c[i][j][j] ≡ a[(i*3+j)*4+k]

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)

value(c + i) = value(c) + i * 3 * 4 * 4
value(c[i] + j) = value(c[i]) + j * 4 * 4
value(c[i][j] + k) = value(c[i][j]) + k * 4

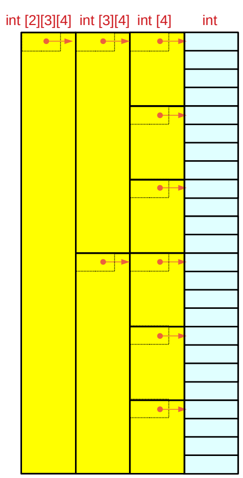
value(c + i) = value(c) + i * sizeof(*c)
value(c[i] + j) = value(c[i]) + j * sizeof(*c[i])
value(c[i][j] + k) = value(c[i][j]) + k * sizeof(*c[i][j])

Array Pointer Approach

$(c + i)_{3 \cdot 4 \cdot 4}$

$(c[i] + j)_{4 \cdot 4}$

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

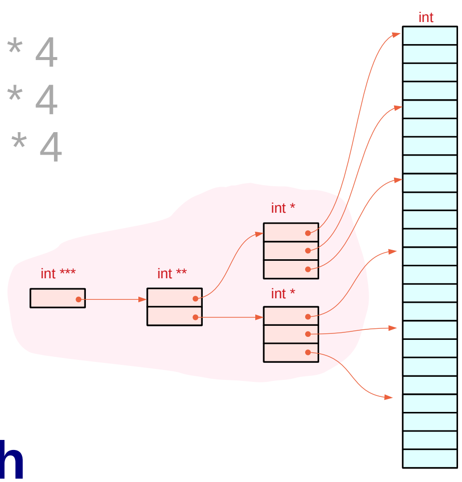


c[i] ≡ *(c + i)
c[i][j] ≡ *(c[i] + j)
c[i][j][k] ≡ *(c[i][j] + k)

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

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

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



Pointer Array Approach

Subscript [ ] and dereference * notations (2a)

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

from **p**, skip
i·M·N integers

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

from **p[i]**, skip
j·N integers

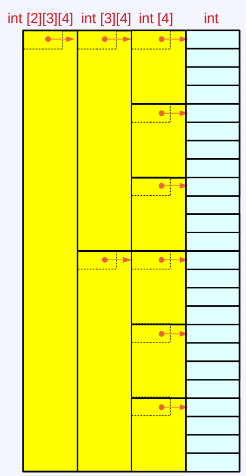
$\&\mathbf{p[i][j]} = \text{value}((\mathbf{p[i]} + \mathbf{j})_{N \cdot 4})$
$= \text{value}(\mathbf{p[i]}) + \mathbf{j} * N \cdot 4$

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

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

int p[L][M][N]

Array Pointer Approach



address replication

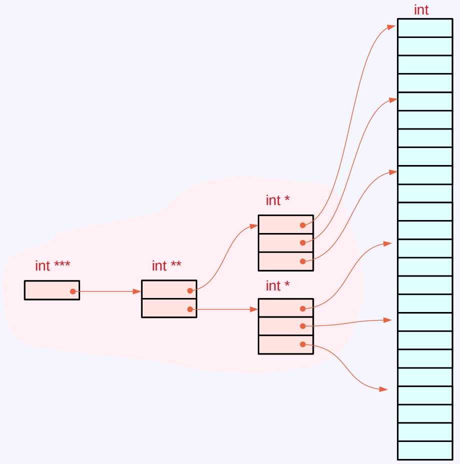
value(**p[i]**) = value(**p** + **i**)

value(**p[i][j]**) = value(**p[i]** + **j**)

&**p[i][j][k]** = value(**p** + **i** *M·N·4 + **j** *N·4 + **k** * 4)

Subscript [ ] and dereference * notations (2b)

p[i]	⇄	*(p+i)	skip **i** pointers from **p**	&**p**[**i**] = value(**p** + **i**)$_{1\cdot 4}$ = value(**p**) + **i** *4
p[i][j]	⇄	*(*(p+i)+j)	skip **j** pointers from **p**[**i**]	&**p**[**i**][**j**] = value(**p**[**i**] + **j**)$_{1\cdot 4}$ = value(**p**[**i**]) + **j** *4
p[i][j][k]	⇄	*(*(*(p+i)+j)+k)	skip **k** integers from **p**[**i**][**j**]	&**p**[**i**][**j**][**k**] = value(**p**[**i**][**j**] + **k**)$_{1\cdot 4}$ = value(**p**[**i**][**j**]) + **k** *4



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

Pointer Array Approach

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

Equivalences (3)

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

C Expressions

&p[i]	≡	value(p+i)
&p[i][j]	≡	value(*(p+i)+j)
&p[i][j][k]	≡	value(*(*(p+i)+j)+k)

C Expressions

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

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

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

Math Expressions

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

Math Expressions

int ** p[L], * q[L·M], r[L·M·N] ;

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

Math Expressions

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

Math Expressions

Equivalences (4)

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

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

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

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

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

&p[i][j][k] = value(p + i * M·N·4 + j * N·4 + k * 4)

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

Math Expressions

int ** p[L], * q[L·M], r[L·M·N] ;

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

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

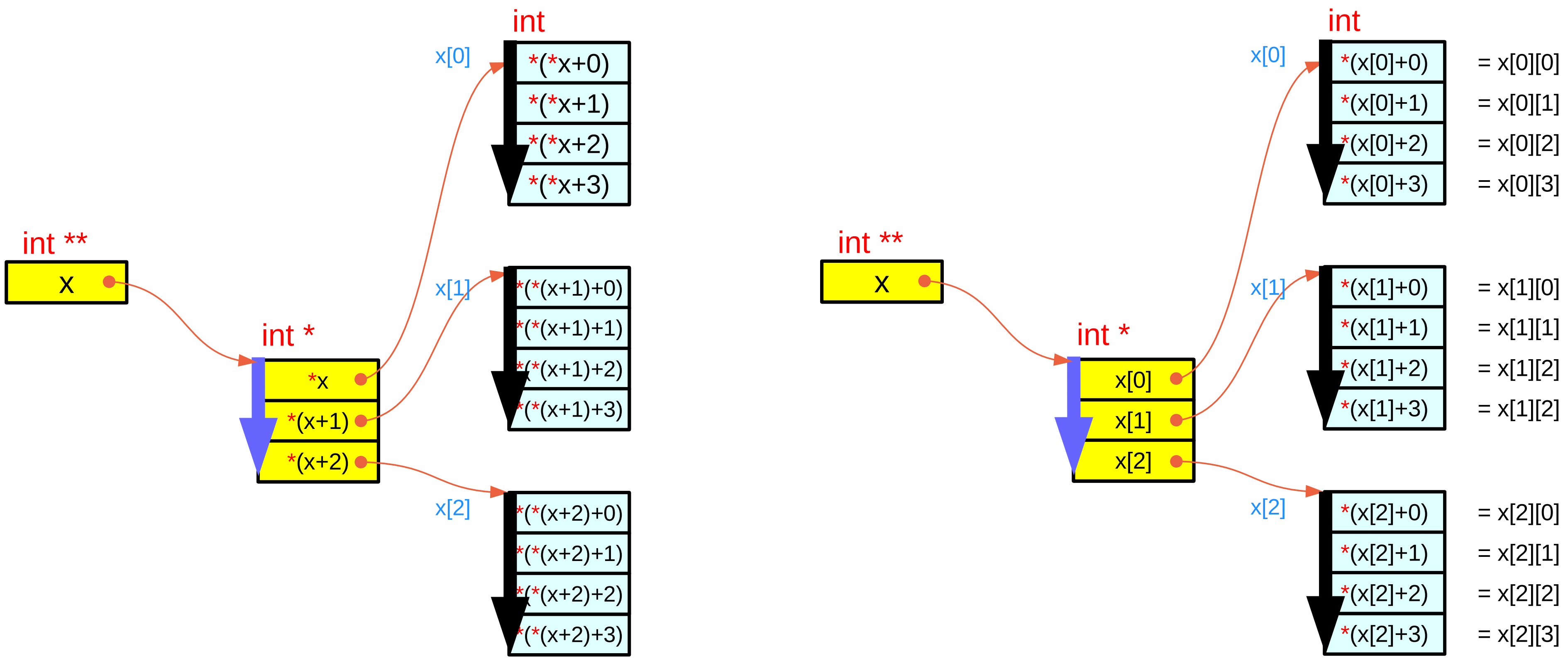
&p[i][j][k] = $\text{value}(p[i][j] + k)_{1 \cdot 4}$ = value(p[i][j] + k * 1·4)

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

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

Math Expressions

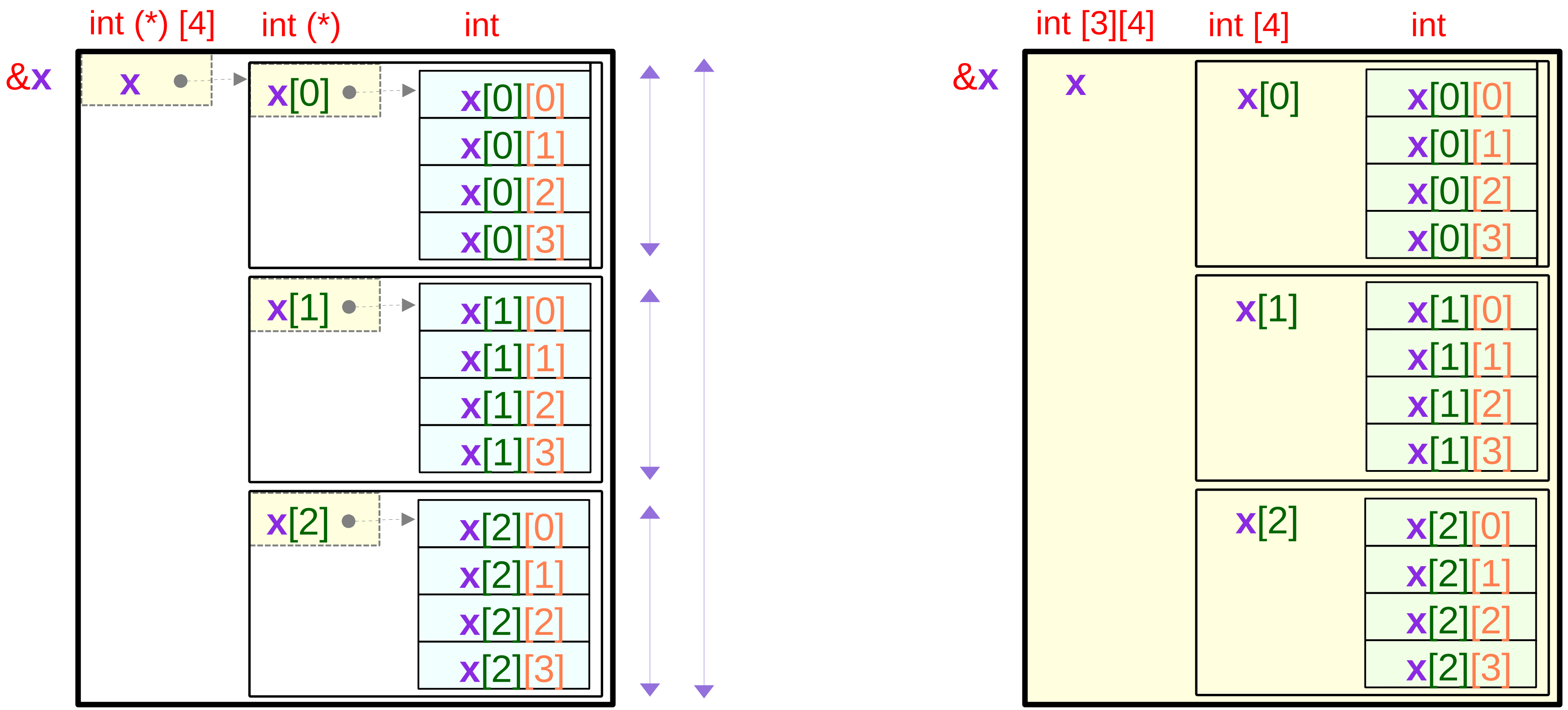
* and [ ] notations (1)



C expression　　*(*(x+i)+j) → x[i][j]

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

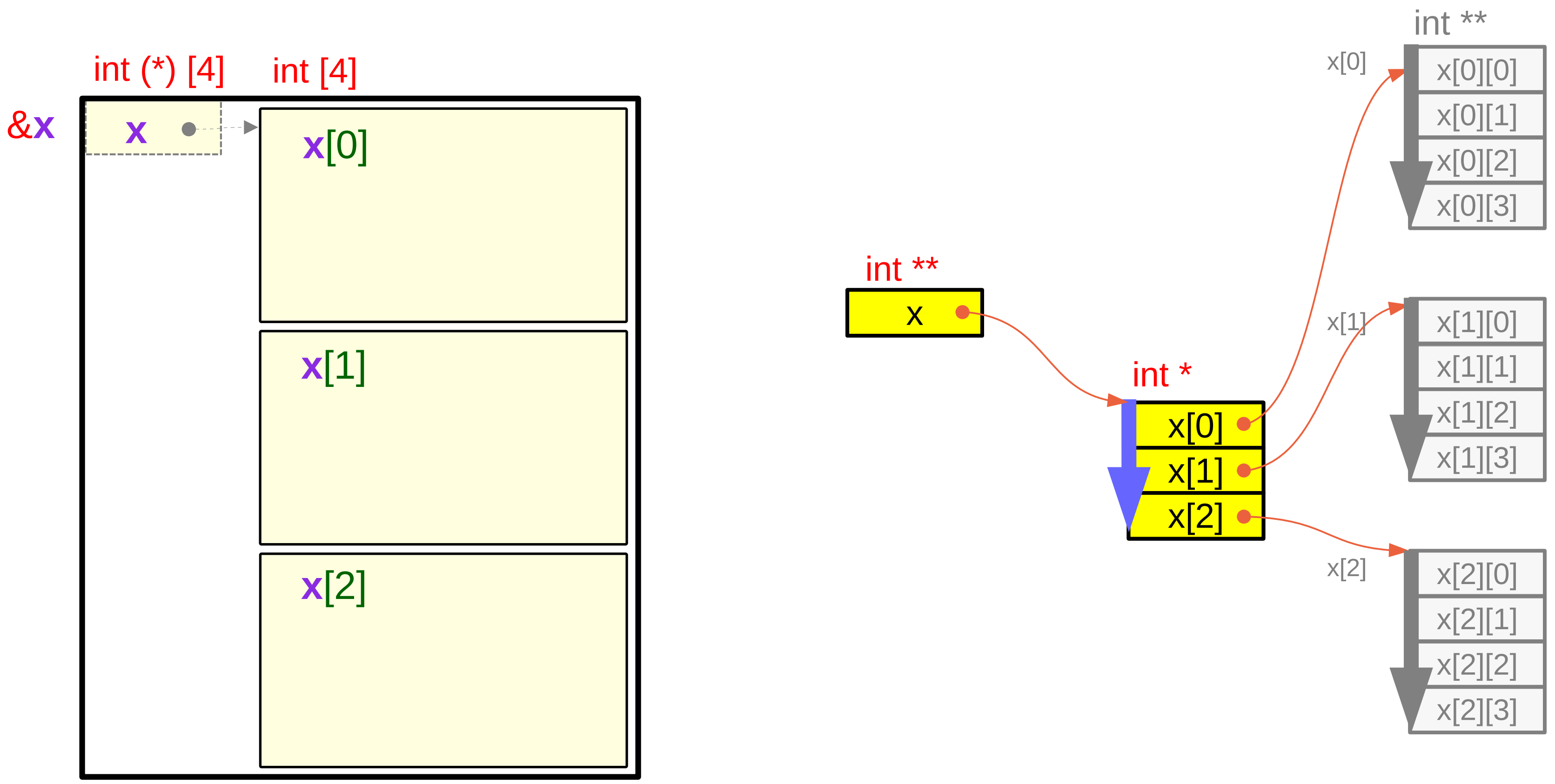
* and [ ] notations (2)



C expression $*(*(x+i)+j) \leftarrow x[i][j]$

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

Virtual pointers vs. real pointers (1)



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

x[**i**] = *(**x** + **i**)$_{4 \cdot 4}$

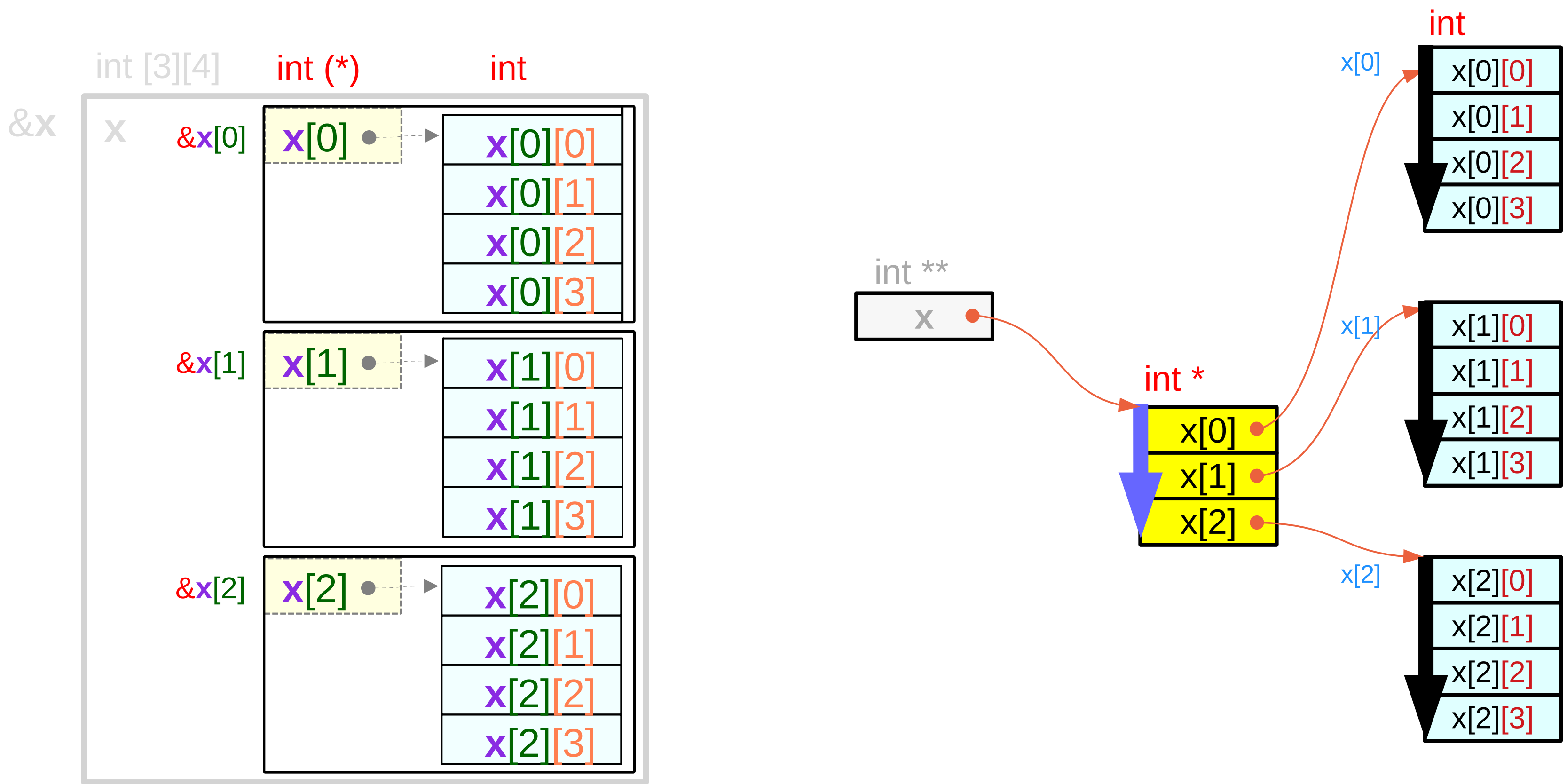
value(**x**+**i**) = value(**x**) + **i** * sizeof(***x**)
= value(**x**) + **i** * 4 *4

&x ≠ **x** sizeof(**x**) = sizeof(***x**) = 4

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

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

Virtual pointers vs. real pointers (2)



&x[i] = x[i] sizeof(x[i]) = 4 * sizeof(*x[i])

x[i] = *(x + i)$_{1\cdot4}$

value(x[i]+j) = value(x[i]) + j * sizeof(*x[i])
= value(x[i]) + j * 4

&x[i] = x[i] sizeof(x[i]) = sizeof(*x[i]) = 4

x[i][j] = *(x[i] + j)$_{1\cdot4}$

value(x[i]+j) = value(x[i]) + j * sizeof(*x[i])
= value(x[i]) + j * 4

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