Applications of Array Pointers (1A)

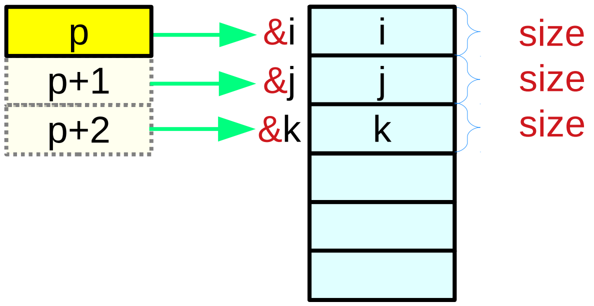
Young Won Lim
4/15/21

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

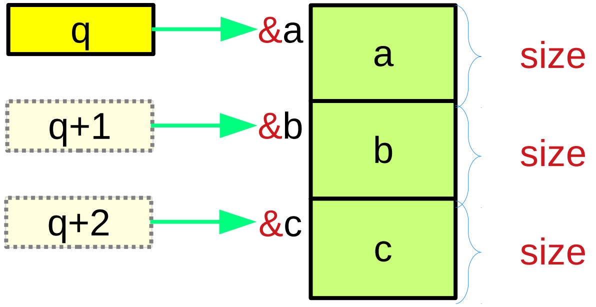
Virtual array pointers in a multi-dimensional array

Pointers to various data types

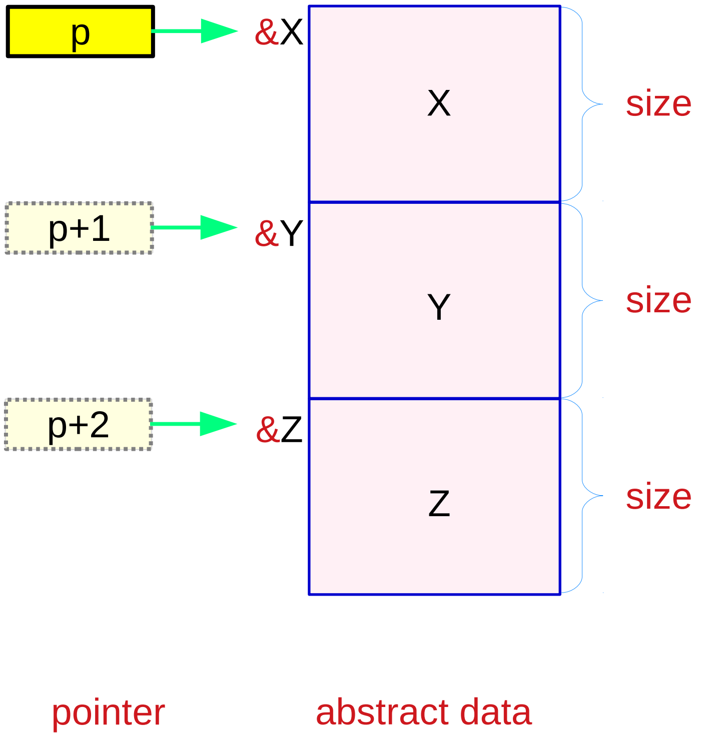
`int *p;        int i, j, k;`



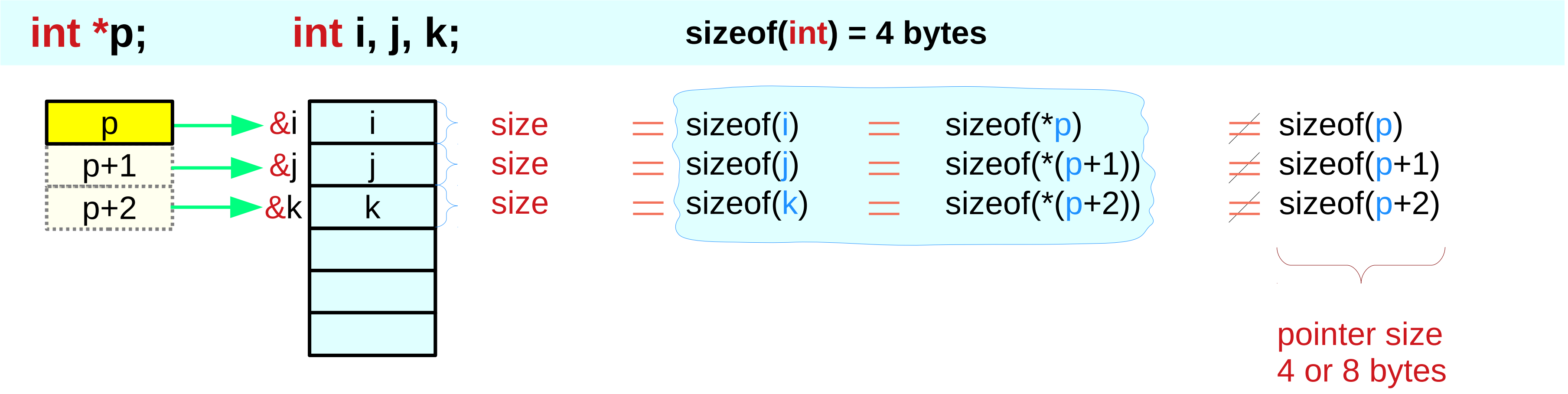
`double *q;     double a, b, c;`



`T *p;          T  X, Y, Z;`



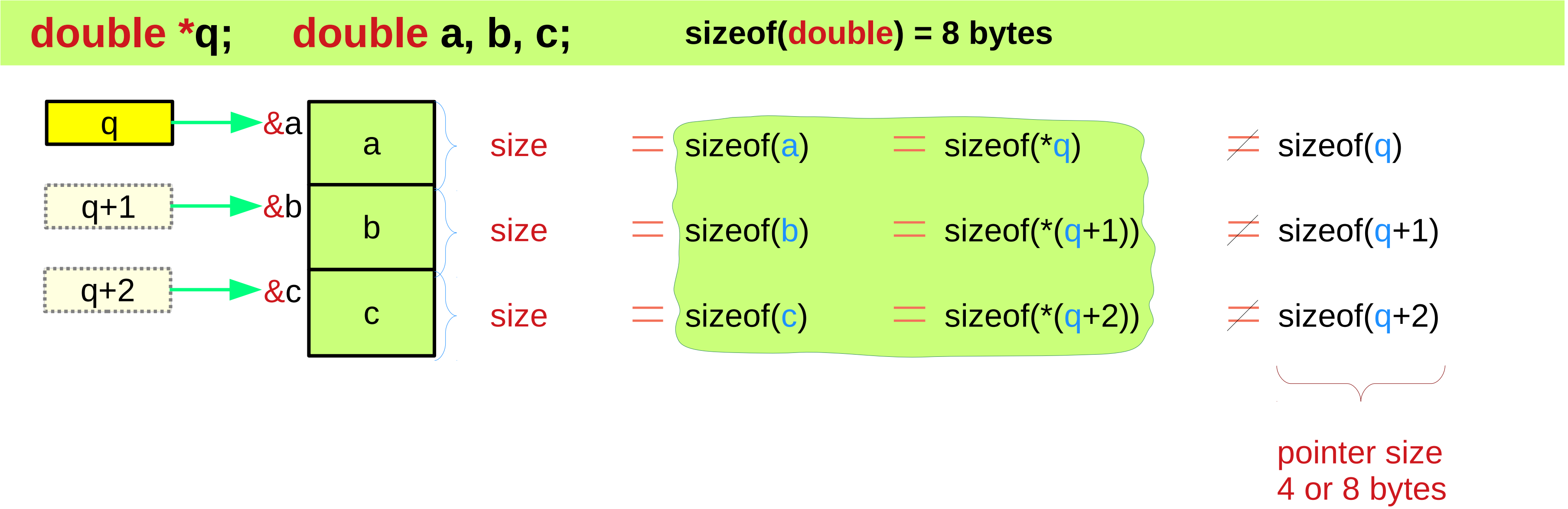
Pointers to primitive data



int *p; **int i, j, k;** **sizeof(int) = 4 bytes**

size = sizeof(i) = sizeof(*p) ≠ sizeof(p)
size = sizeof(j) = sizeof(*(p+1)) ≠ sizeof(p+1)
size = sizeof(k) = sizeof(*(p+2)) ≠ sizeof(p+2)

pointer size
4 or 8 bytes



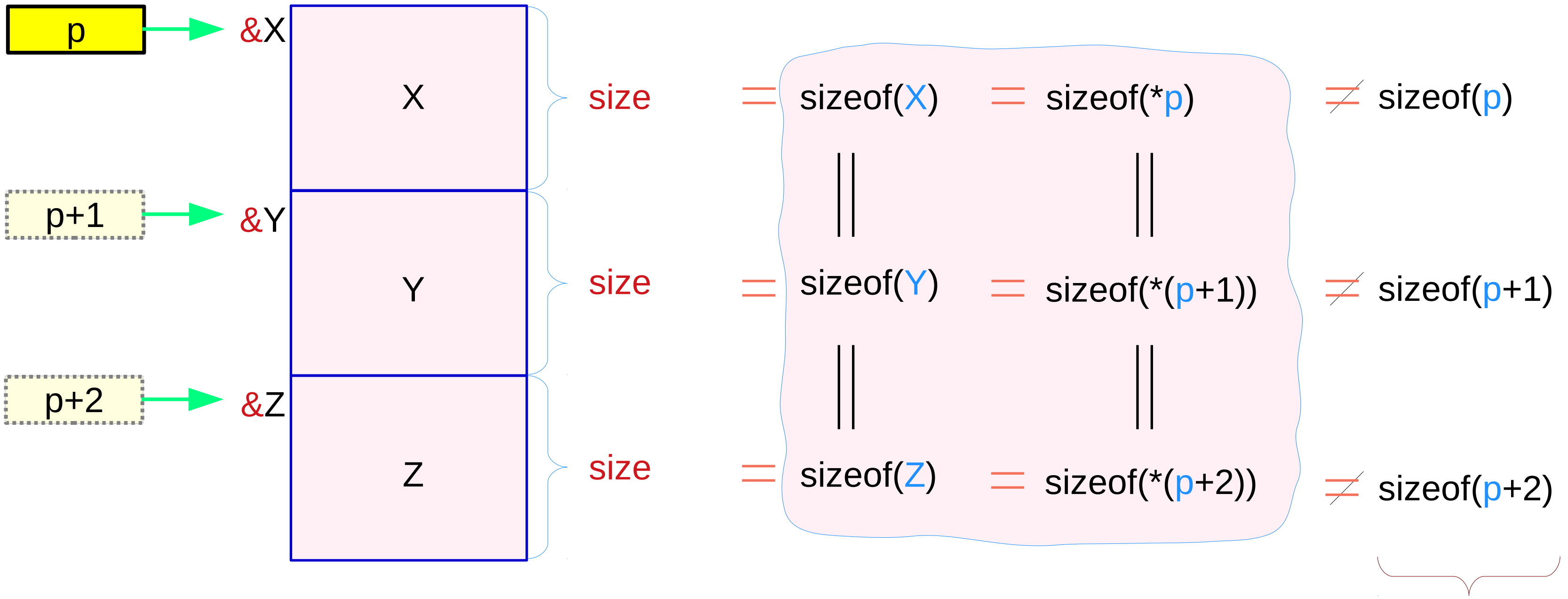
double *q; **double a, b, c;** **sizeof(double) = 8 bytes**

size = sizeof(a) = sizeof(*q) ≠ sizeof(q)
size = sizeof(b) = sizeof(*(q+1)) ≠ sizeof(q+1)
size = sizeof(c) = sizeof(*(q+2)) ≠ sizeof(q+2)

pointer size
4 or 8 bytes

Pointers to abstract data

T *p; **T X, Y, Z;** **sizeof(T)**



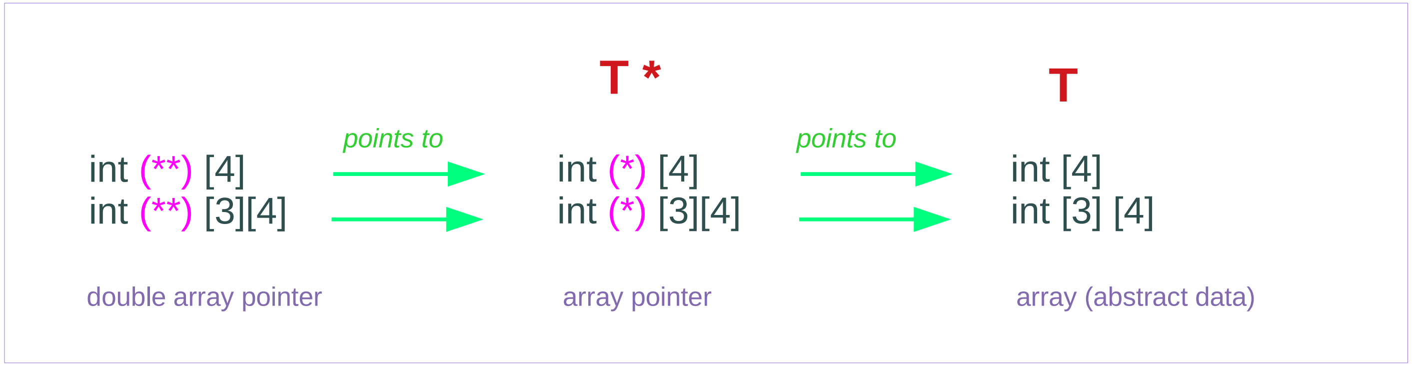
pointer

abstract data

pointer size
4 or 8 bytes

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

Array pointer types v.s. array types



T *

T

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

points to

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

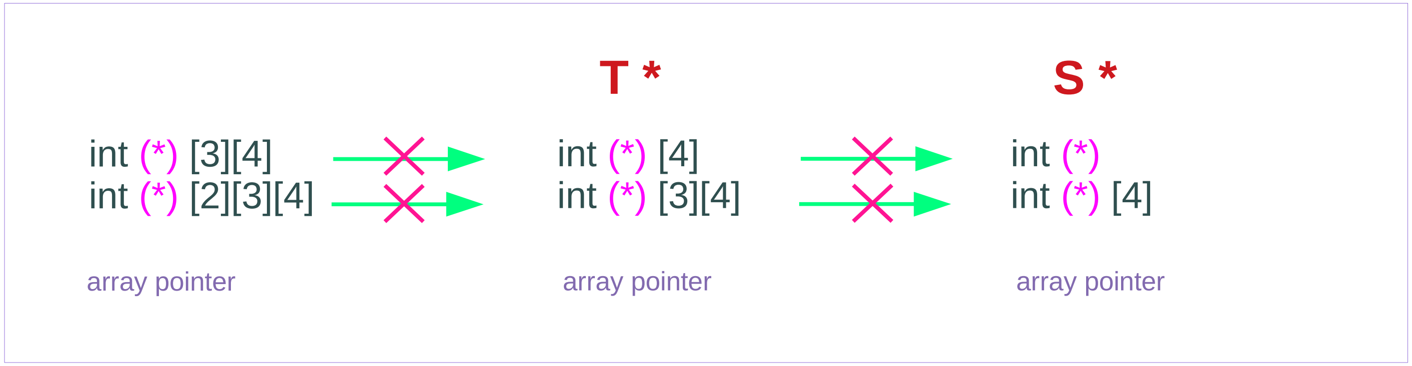
points to

int [4]
int [3] [4]

double array pointer

array pointer

array (abstract data)



T *

S *

int (*) [3][4]
int (*) [2][3][4]

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

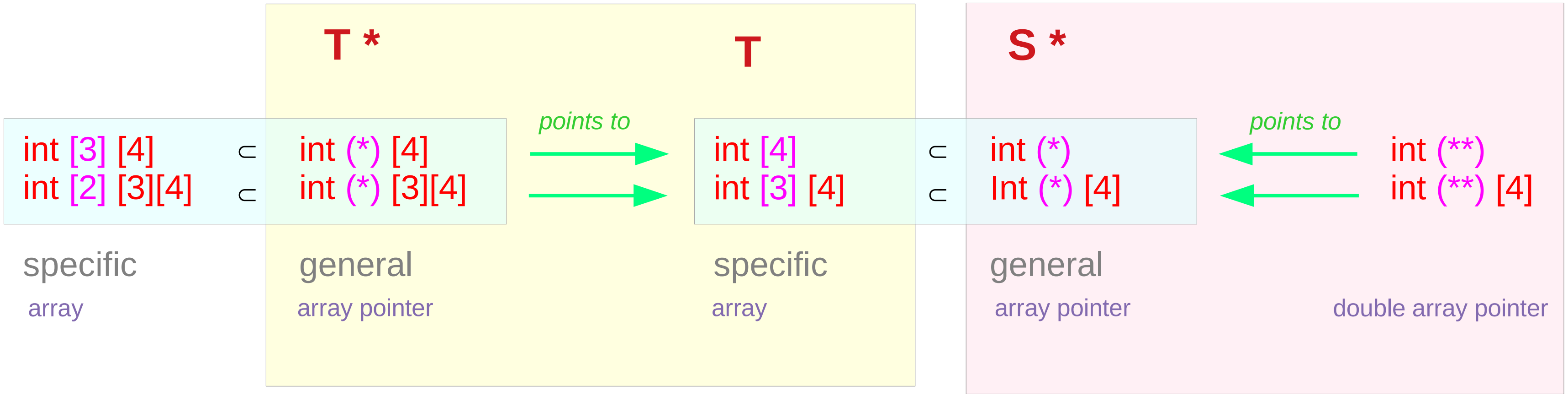
int (*)
int (*) [4]

array pointer

array pointer

array pointer

General array pointer types v.s. specific array types



T *

T

S *

int [3] [4] ⊂ int (*) [4] *points to* → int [4] ⊂ int (*) ← *points to* int (**)

int [2] [3][4] ⊂ int (*) [3][4] → int [3] [4] ⊂ Int (*) [4] ← int (**) [4]

specific
array

general
array pointer

specific
array

general
array pointer

double array pointer

Array pointers have augmented dimensions

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

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

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

int [4]
int [3] [4]
specific

T1 a;
T2 b;

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

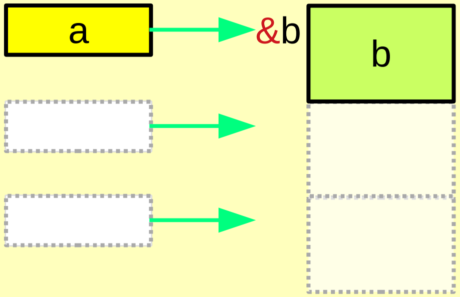
a = &**b**;
*__a__ = **b**;

a references **b**

b is the dereference of **a**

(**a+1**) = ?
* (**a+1**) = ?

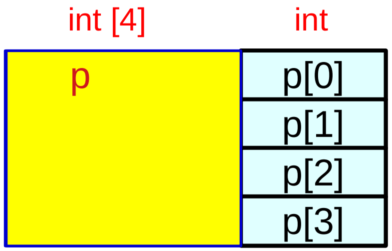
(**a+2**) = ?
* (**a+2**) = ?



Dual types in an array of integers

int p[3];

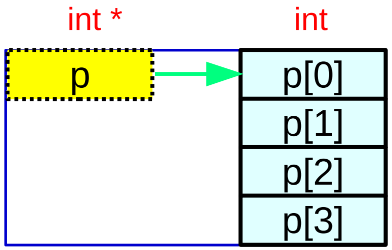
p is an abstract data (array)



p is the name of an array
p has the size of the whole array

p has an array type (abstract data)

p can also be viewed as a pointer

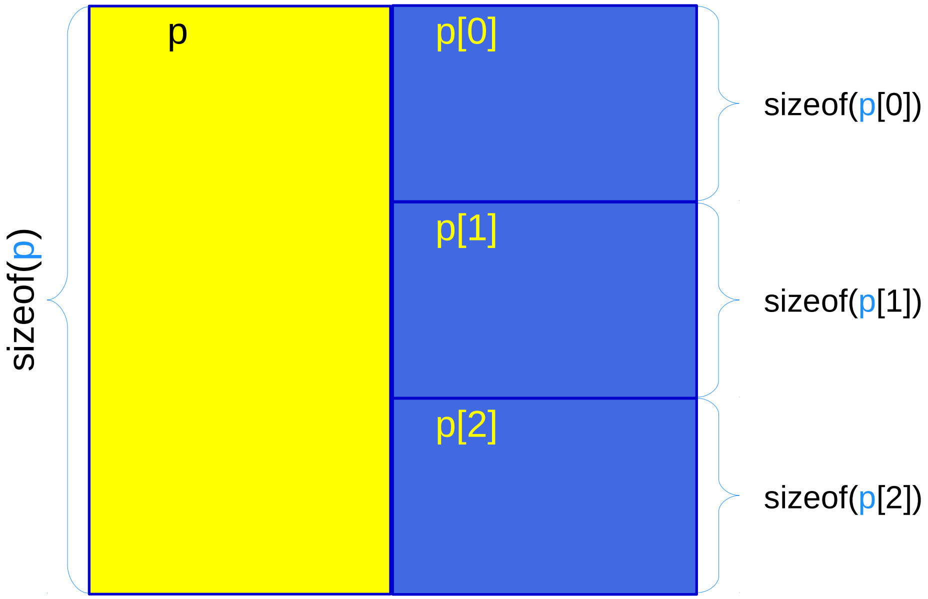


p also has pointer characteristics
p has the value of the starting address

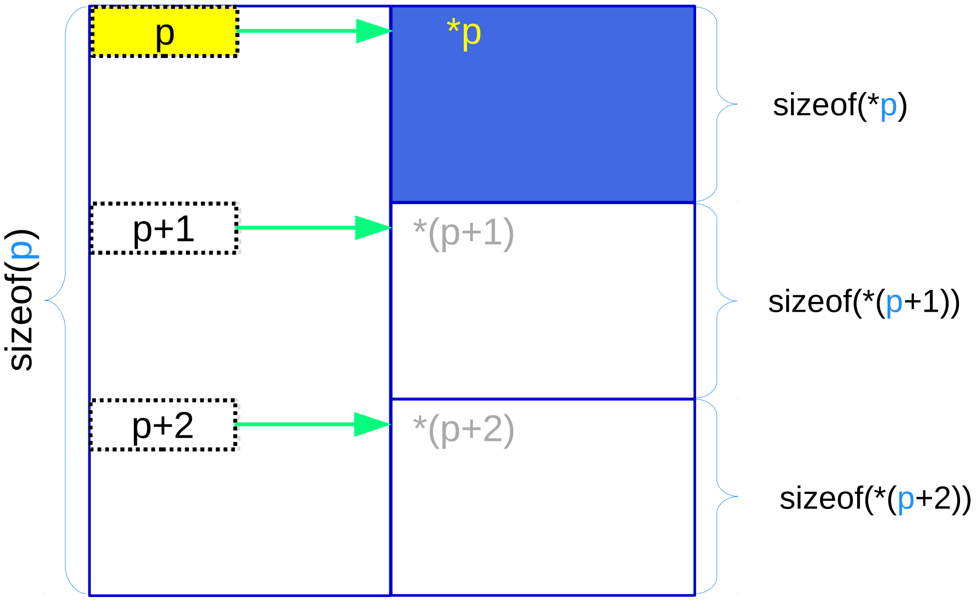
p is a virtual pointer

Dual types in an array of abstract data

Abstract data array p



Virtual pointer p

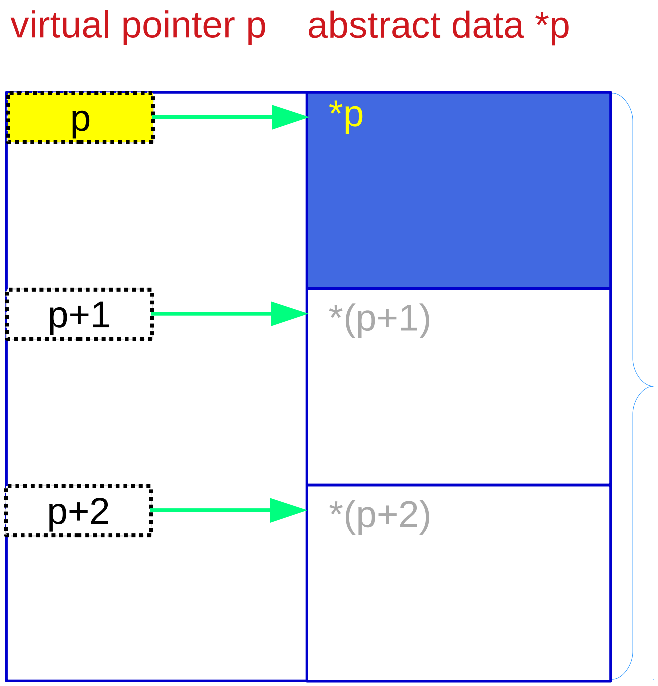


p has an array type (abstract data element)
p is the name of an array
p has the size of the whole array

p also has a pointer type
p has the value of the starting address

p is a virtual array pointer

Virtual pointer to abstract data



virtual pointer p

abstract data *p

p

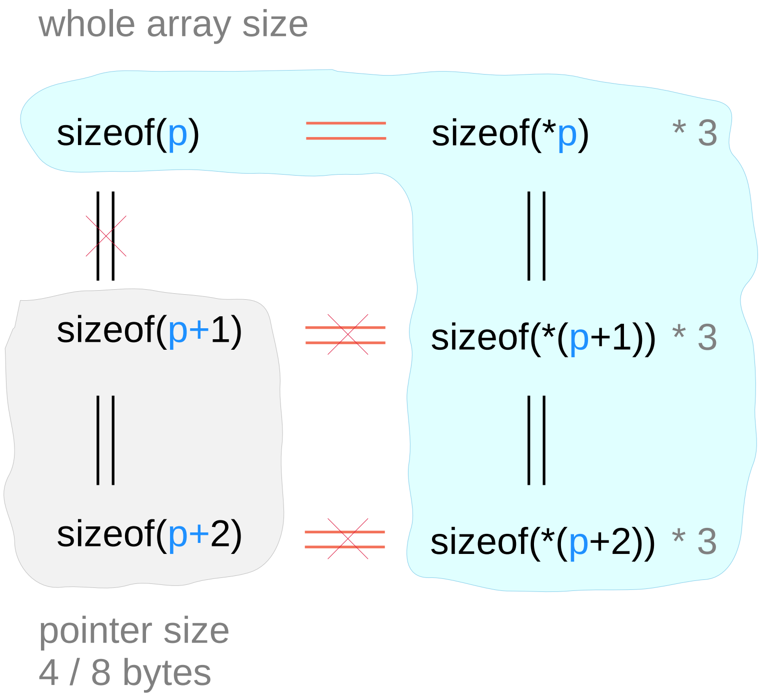
p+1

p+2

*p

*(p+1)

*(p+2)



whole array size

sizeof(p) == sizeof(*p) * 3

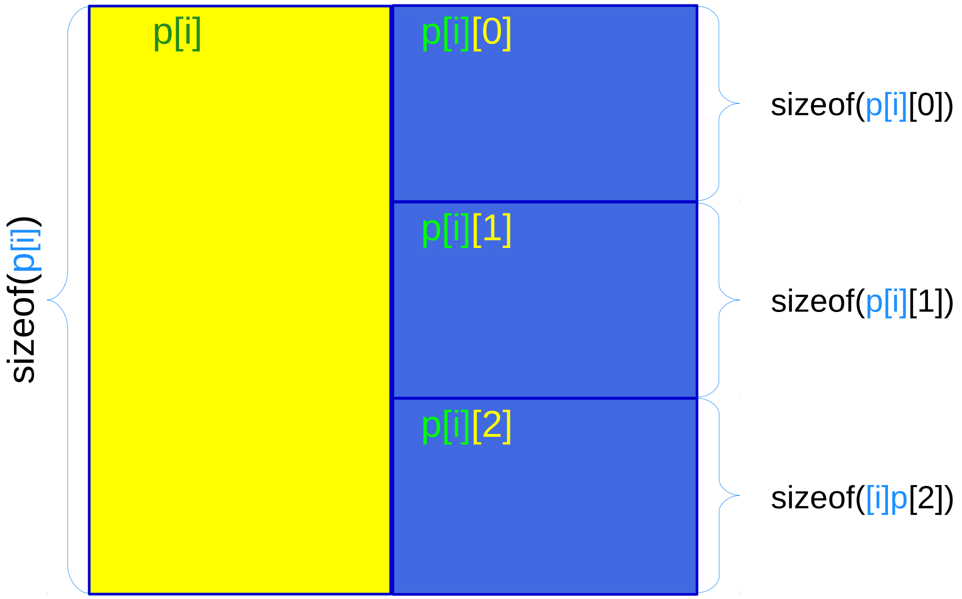
sizeof(p+1) ≠ sizeof(*(p+1)) * 3

sizeof(p+2) ≠ sizeof(*(p+2)) * 3

pointer size
4 / 8 bytes

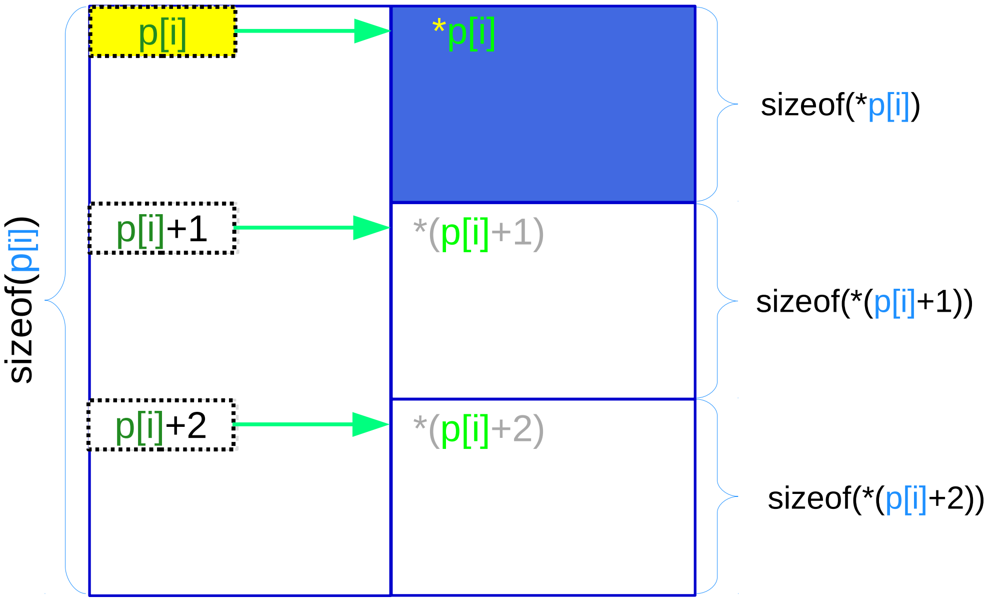
Dual types in a multi-dimensional array

Abstract data (array) p[i]



p[i] has an array type (abstract data)
p[i] is the name of an array
p[i] has the size of the whole array

Virtual array pointer p[i]



p[i] also has an array pointer type
p[i] has the value of the starting address

p[i] is a virtual array pointer

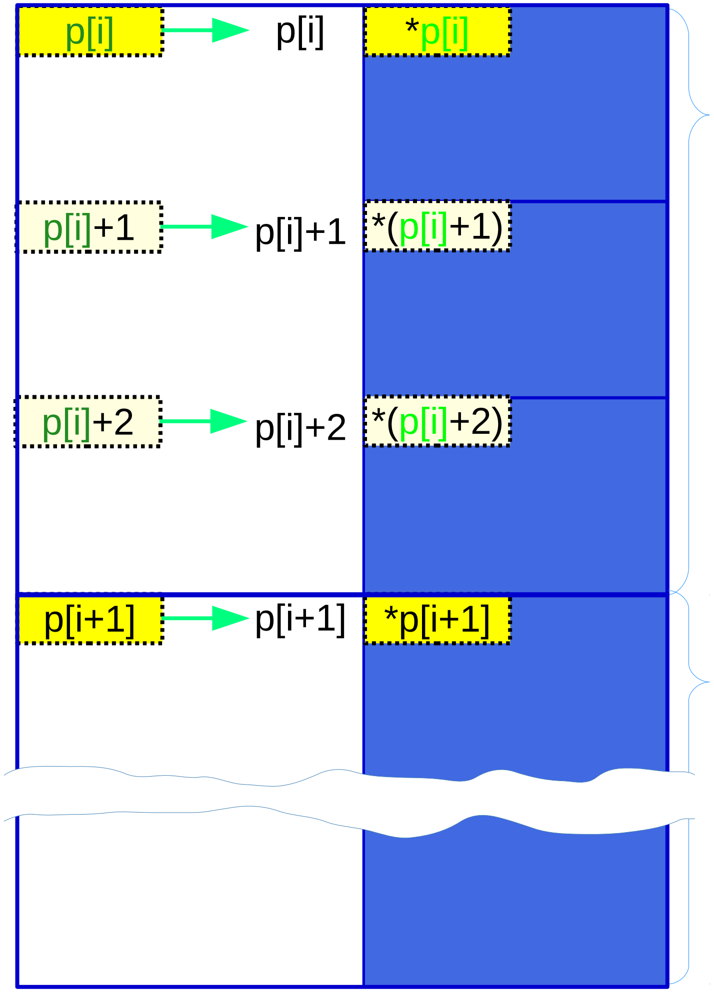
Virtual pointers to sub-arrays

p[i] :: T *

array pointer type

***p[i], *p[i+1] :: T**

array type



sizeof(p[i])

= sizeof(*p[i]) * N
sizeof(p[i][0]) * N

size = sizeof(*p[i]) =
sizeof(p[i][0])

||

size = sizeof(*(p[i]+1)) =
sizeof(p[i][1])

||

size = sizeof(*(p[i]+2)) =
sizeof(p[i][2])

sizeof(p[i+1])

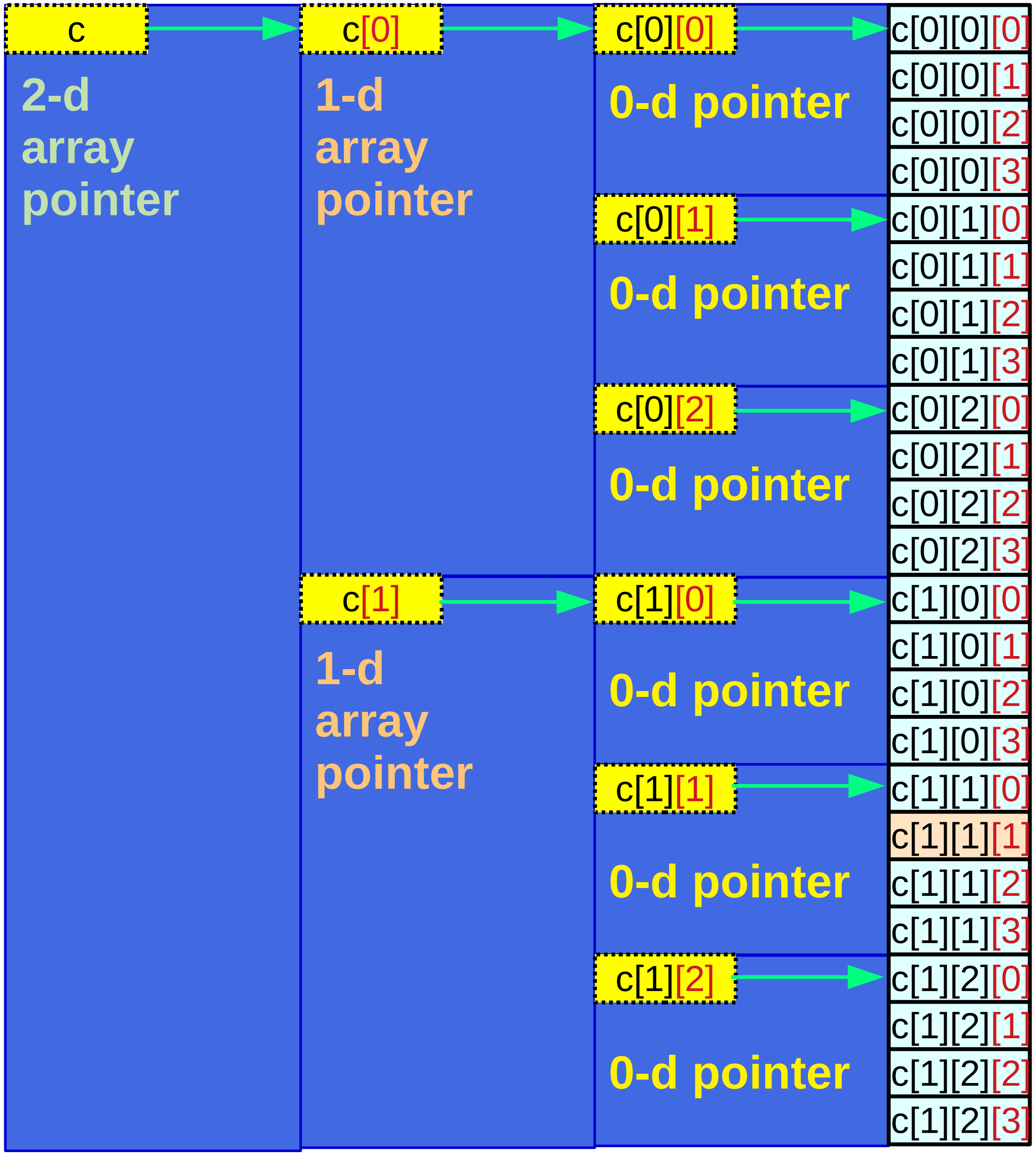
= sizeof(*p[i+1]) * N
sizeof(p[i+1][0]) * N

3-d array structure – virtual pointer representation

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

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

- Hierarchical
- Nested Structure
- Virtual Array Pointers to abstract data (subarrays)
- Contiguous and Linear Data Layout
- Row Major Order



3-d array structure – abstract data representation

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

((c [i])[j])[k]

- Hierarchical
- Nested Structure
- Virtual Array Pointers to abstract data (subarrays)
- Contiguous and Linear Data Layout
- Row Major Order

c
3-d array name

c[0]
2-d array name

c[1]
2-d array name

c[0][0]
1-d array name

c[0][1]
1-d array name

c[0][2]
1-d array name

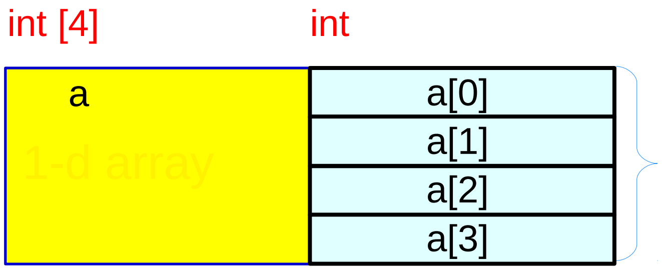
c[1][0]
1-d array name

c[1][1]
1-d array name

c[1][2]
1-d array name

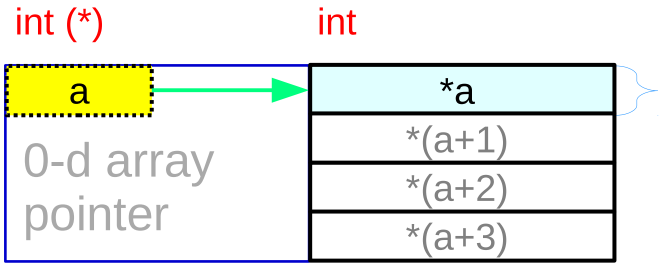
c[0][0][0]
c[0][0][1]
c[0][0][2]
c[0][0][3]
c[0][1][0]
c[0][1][1]
c[0][1][2]
c[0][1][3]
c[0][2][0]
c[0][2][1]
c[0][2][2]
c[0][2][3]
c[1][0][0]
c[1][0][1]
c[1][0][2]
c[1][0][3]
c[1][1][0]
c[1][1][1]
c[1][1][2]
c[1][1][3]
c[1][2][0]
c[1][2][1]
c[1][2][2]
c[1][2][3]

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



1-d array a specific array type

sizeof(a)



pointer a general pointer type

sizeof(a) = sizeof(*a) * 4

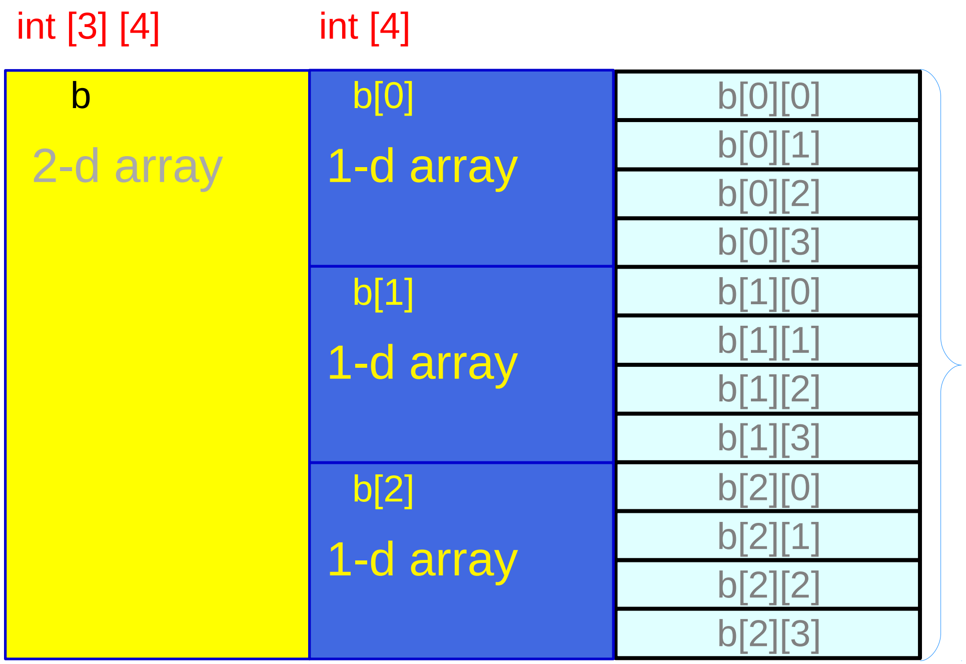
a is the name of a 1-d array
a also has a pointer type
a has the size of the array
a has the value of the starting address

a is a virtual array pointer

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

2-d array b specific array type

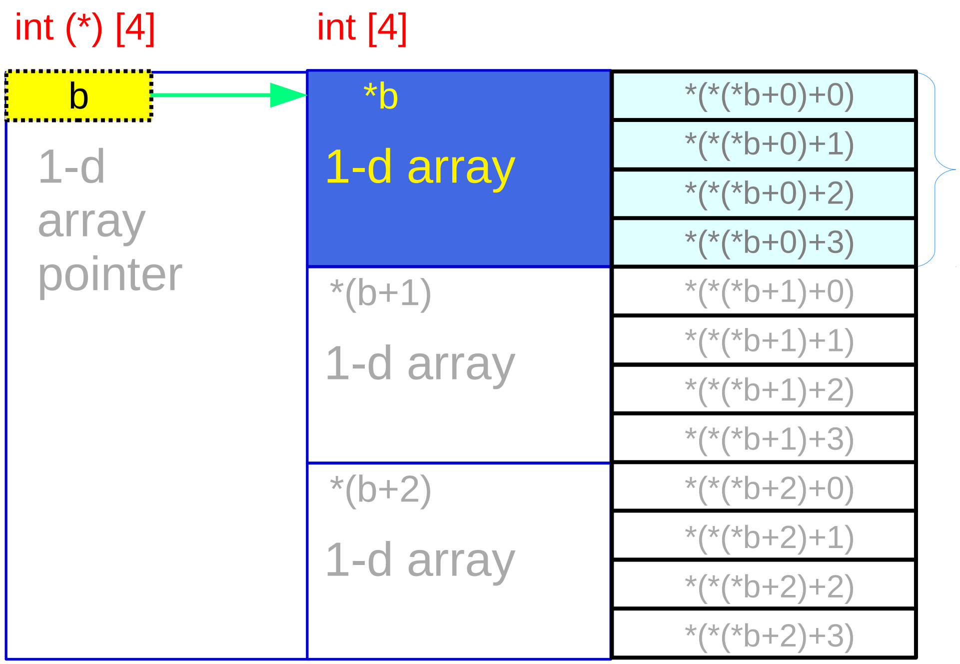
sizeof(b)



b is the name of a 2-d array
b has the size of the array

1-d array pointer b general pointer type

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



b also has a 1-d array pointer type
b has the value of the starting address

b is a virtual array pointer

Array **c**

3-d array c

specific array type

sizeof(c)

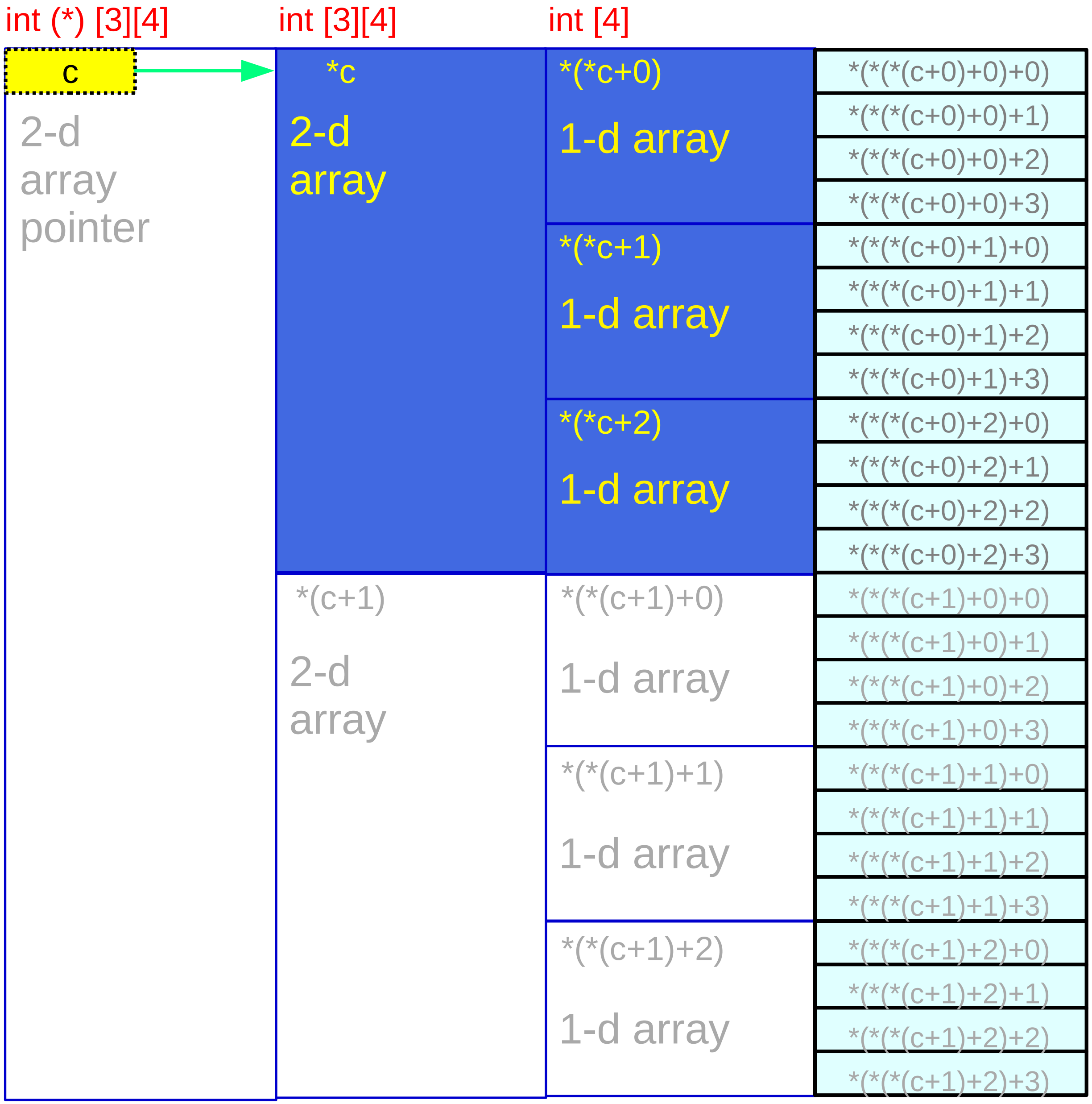
c is the name of a 3-d array
c has the size of the array

Virtual pointer **c**

2-d array pointer c

general pointer type

sizeof(c) = sizeof(*c) * 2



int (*) [3][4]

int [3][4]

int [4]

c

2-d array pointer

*c

2-d array

*(*c+0)

1-d array

*(*c+1)

1-d array

*(*c+2)

1-d array

*(c+1)

2-d array

((c+1)+0)

1-d array

((c+1)+1)

1-d array

((c+1)+2)

1-d array

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

c also has a 2-d array pointer type
c has the value of the starting address

c is a virtual array pointer

Types of virtual array pointers in a **3-d** array

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

	array type (name)	array pointer type
c[i][j][k]	int	int
c[i][j] [k]	int [4] [k]	int (*) [k]
c[i] [j] [k]	int [3][4] [j] [k]	int (*)[4] [j] [k]
c [i] [j] [k]	int [2][3][4] [i] [j] [k]	int (*)[3][4] [i] [j] [k]

Sizes of virtual array pointers in a **3-d** array

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

sizeof(**c[i][j][k]**) = sizeof(int)

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

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

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

Element Size

Addresses of virtual array pointers in a 3-d array

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

			[i] [3] [4]	[j] [4]	[k]
&c[i][j][k]	=	&c[0][0][0]	+ i*3*4*4	+ j*4*4	+ k*4
&c[i][j] [4]	=	&c[0][0][0]	+ i*3*4*4	+ j*4*4	sizeof(*c[i][j]) = sizeof(c[i][j][0]) = sizeof(int)
&c[i] [3] [4]	=	&c[0][0][0]	+ i*3*4*4	sizeof(*c[i]) = sizeof(c[i][0]) = sizeof(int) * 4 [k]	
&c [2] [3] [4]	=	&c[0][0][0]	sizeof(*c) = sizeof(c[0]) = sizeof(int) * 3 * 4 [j] [k]		

Base Address

Values of virtual array pointers in a **3-d** array

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

c[i][j][k] +0 = &c[i][j][k] = the **value** of a virtual array pointer is its **address**

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

sizeof(*c[i][j]) = sizeof(int)

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

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

[k]

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

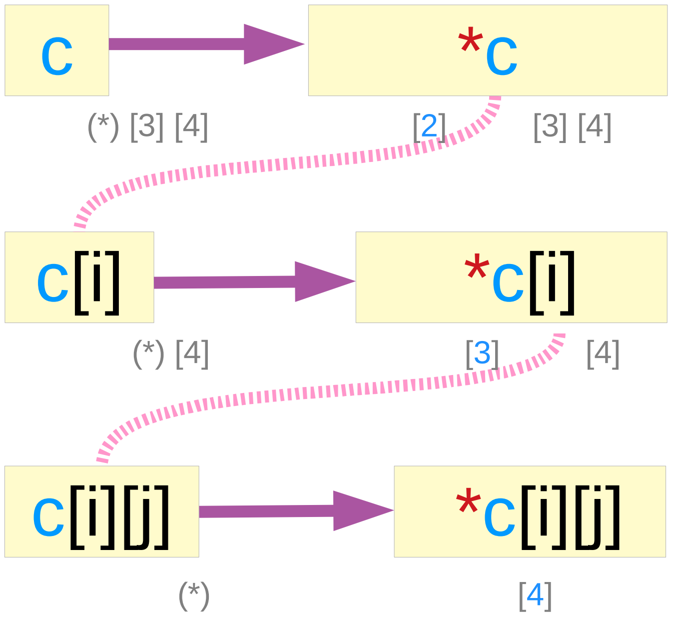
sizeof(*c) = sizeof(int) * 3 * 4

[j] [k]

Start Address

Virtual array pointers and abstract data in a **3-d** array

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



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

(*) [3] [4]   [2] [3] [4]

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

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

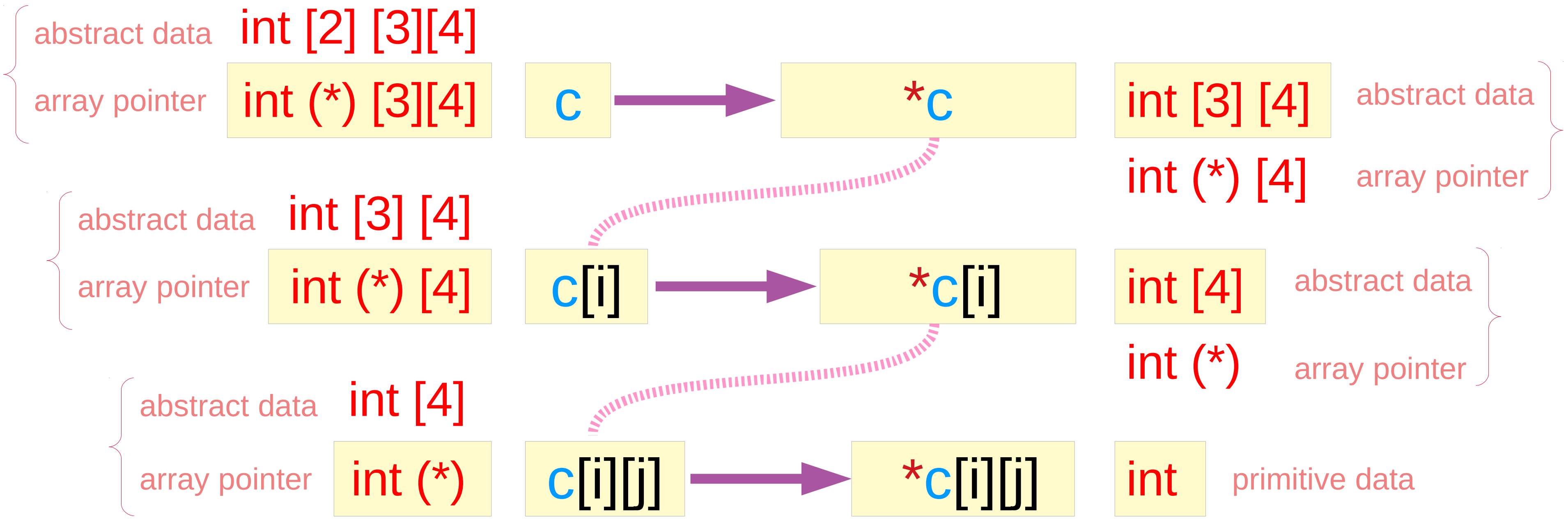
0-d array pointer   **int (*)**   c[i][j] → *c[i][j]   **int**   **0-d** array

(*)   [4]

Virtual Array Pointers   **Abstract Data (Array)**

Dual types in a **3-d** array

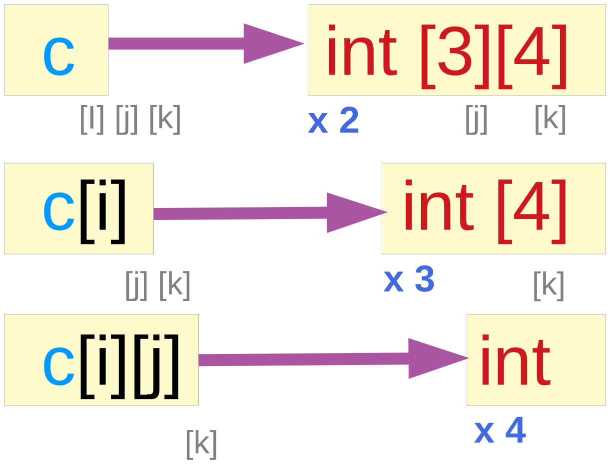
```
int     c [2][3][4];
```



Pointed array sizes in a **3-d** array

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

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



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

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

sizeof(*c[i][j]) = sizeof(int)

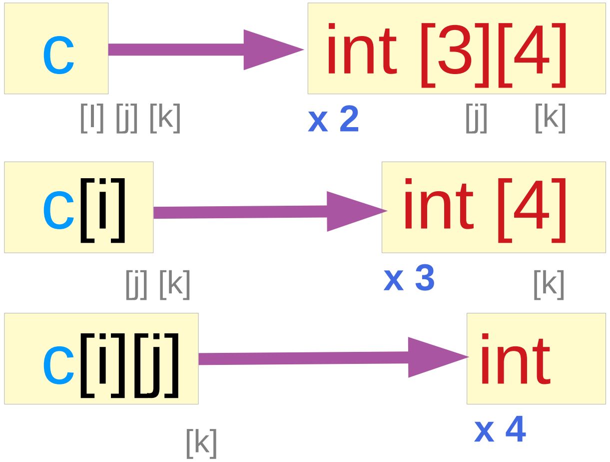
all are sizes of arrays

c, c[i], c[i][j] are virtual array pointers
and they are also abstract data (arrays)
[k]

when sizes are considered,
view them as abstract data (arrays)

Virtual array pointer sizes in a **3-d** array

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



size of a virtual array pointer = size of the pointed abstract data type * the number of such data

sizeof(c) = **2** * sizeof(*c)

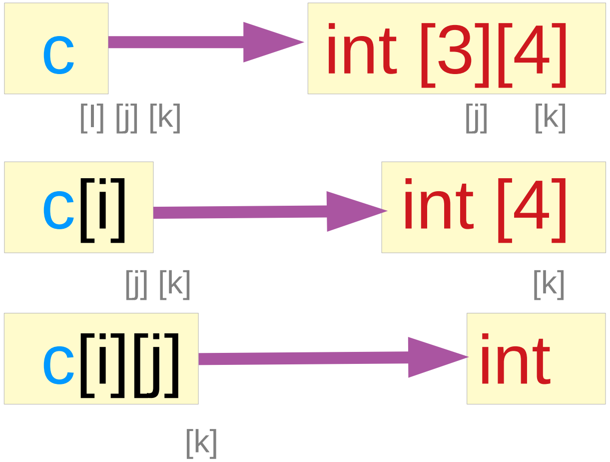
sizeof(c[i]) = **3** * sizeof(*c[i])

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

sizeof(Virtual Array Pointer) =
sizeof(Array of the dual type)

Virtual pointer sizes are subarray sizes

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



sizeof(Virtual Array Pointer) =
sizeof(Array of the dual type)

sizeof(int [2] [3][4]) = sizeof(c) = 2*3*4 * 4
sizeof(int (*) [3][4]) = pointer size = 4 or 8

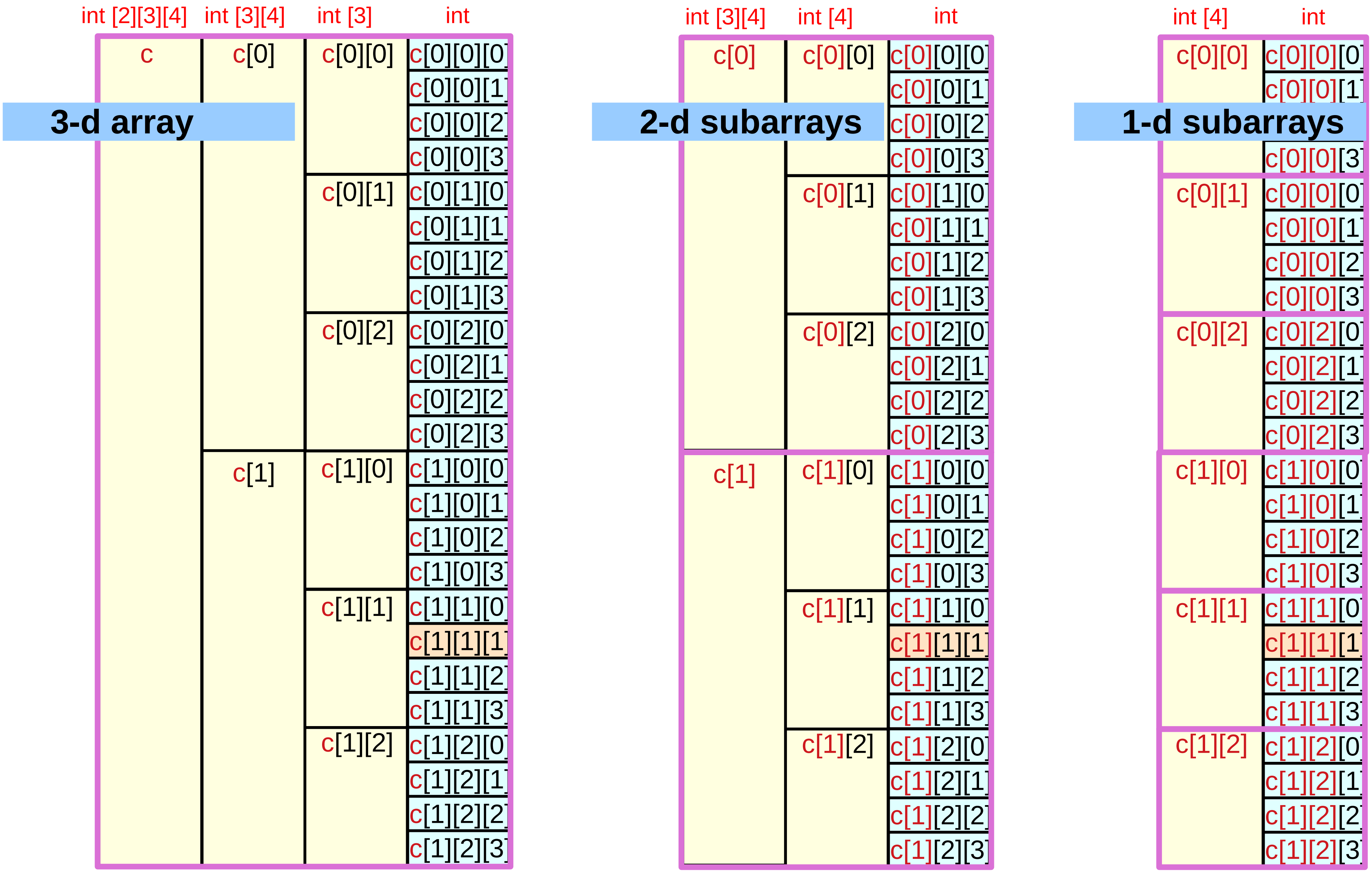
sizeof(int [3] [4]) = sizeof(c[i]) = 3*4 * 4
sizeof(int (*) [4]) = pointer size = 4 or 8

sizeof(int [4]) = sizeof(c[i][j]) = 4 * 4
sizeof(int [4]) = pointer size = 4 or 8

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

not real array pointers
virtual array pointers

Subarrays **c**, **c[i]**, **c[i][j]** in a **3-d** array



3-d array

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

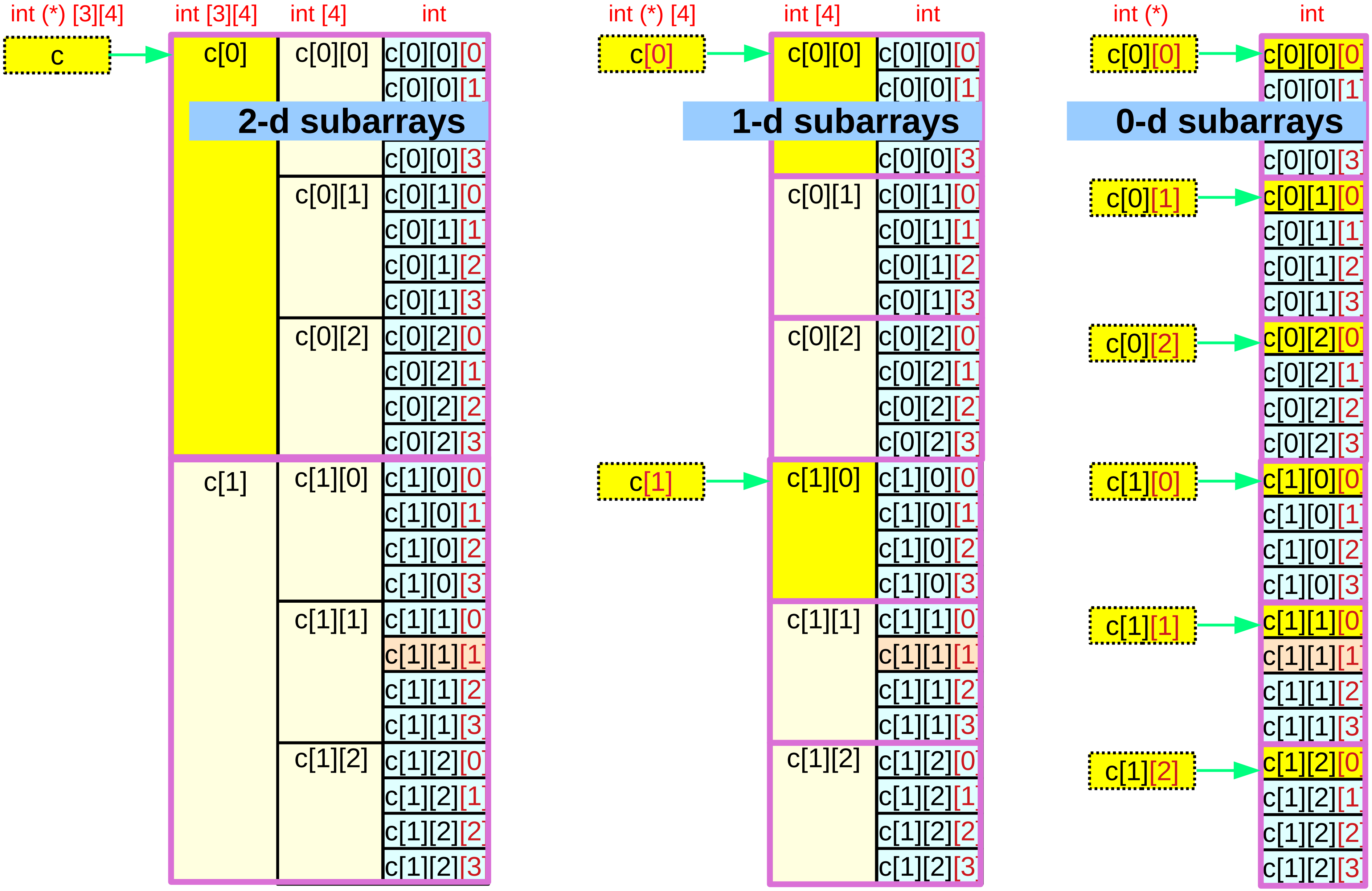
2-d subarrays

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

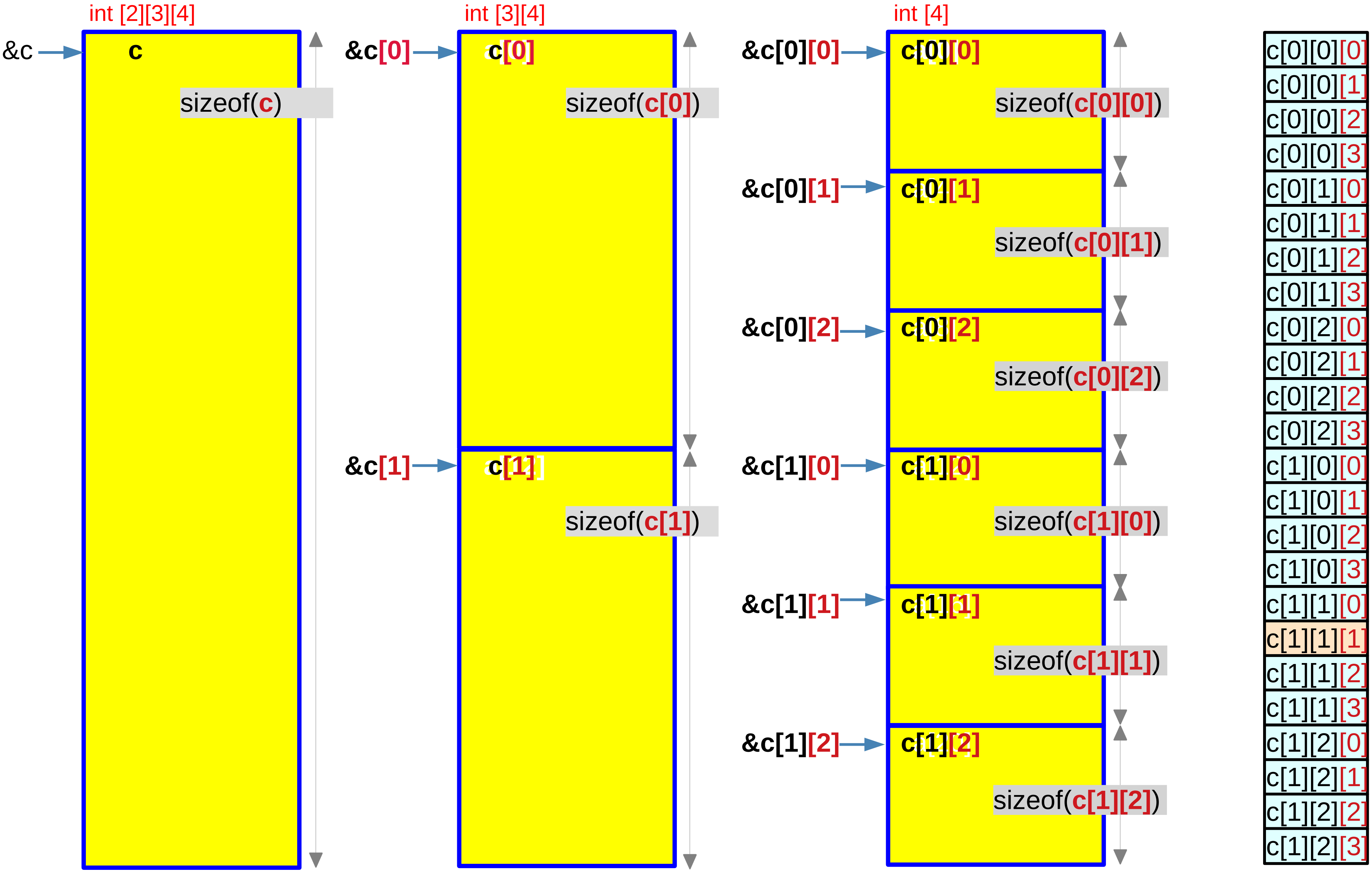
1-d subarrays

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

Virtual array pointers **c**, **c[i]**, **c[i][j]** in a **3-d** array



Abstract data **c**, **c[i]**, **c[i][j]** – start addresses and sizes



int [2][3][4]

&c → **c**

sizeof(c)

int [3][4]

&c[0] → **c[0]**

sizeof(c[0])

&c[1] → **c[1]**

sizeof(c[1])

int [4]

&c[0][0] → **c[0][0]**

sizeof(c[0][0])

&c[0][1] → **c[0][1]**

sizeof(c[0][1])

&c[0][2] → **c[0][2]**

sizeof(c[0][2])

&c[1][0] → **c[1][0]**

sizeof(c[1][0])

&c[1][1] → **c[1][1]**

sizeof(c[1][1])

&c[1][2] → **c[1][2]**

sizeof(c[1][2])

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

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

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

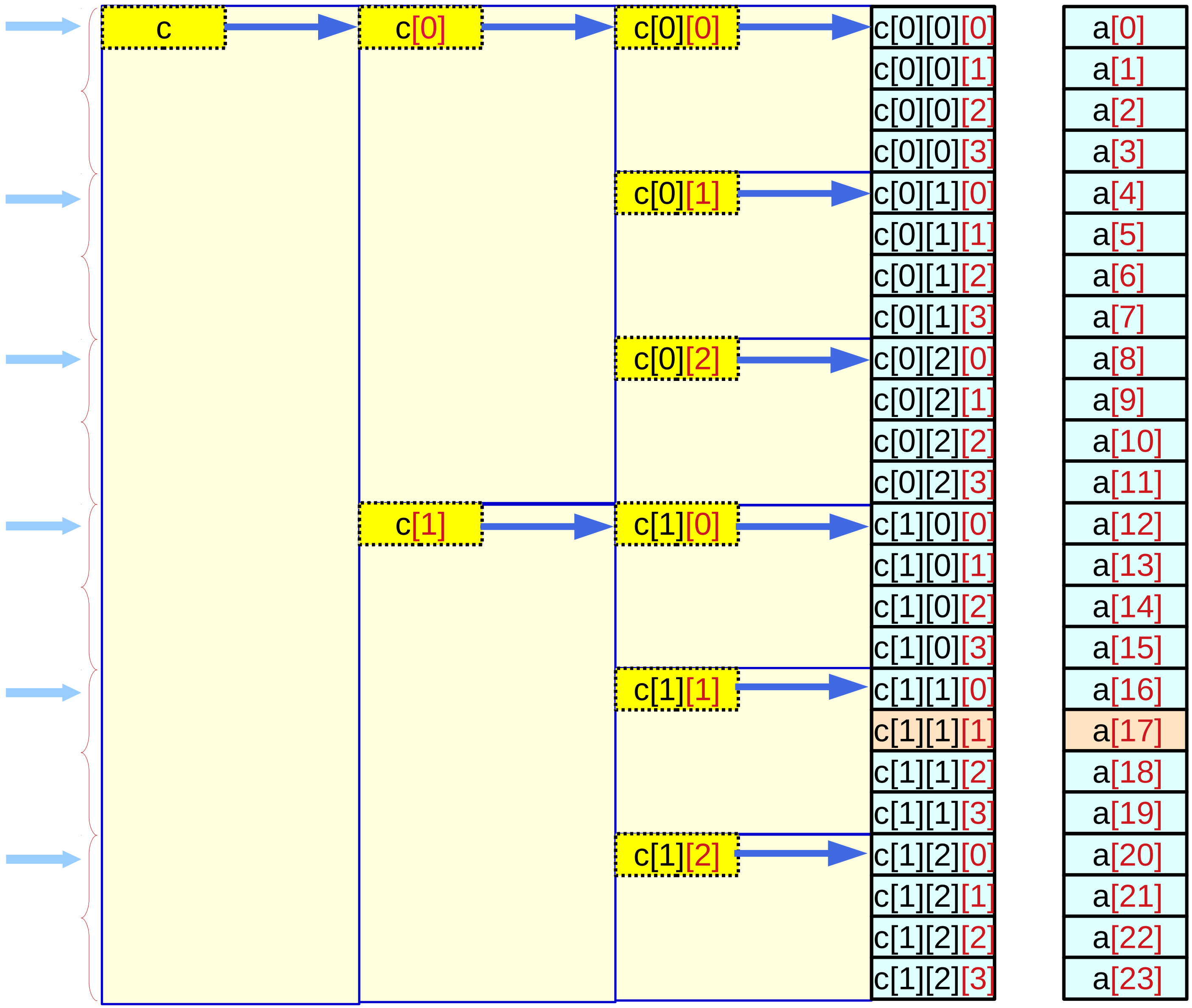
c[0][1] = &c[0][1][0]

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

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

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

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



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

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

means

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

type(c) ≠ type(c[0]) ≠ type(c[0][0]) = type(&c[0][0][0])

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

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

means

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

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

int *   int *

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

means

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

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

int *   int *

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

means

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

type(c[1]) ≠ type(c[1][0]) = type(&c[1][0][0])

int (*) [4]   int *   int *

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

means

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

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

int *   int *

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

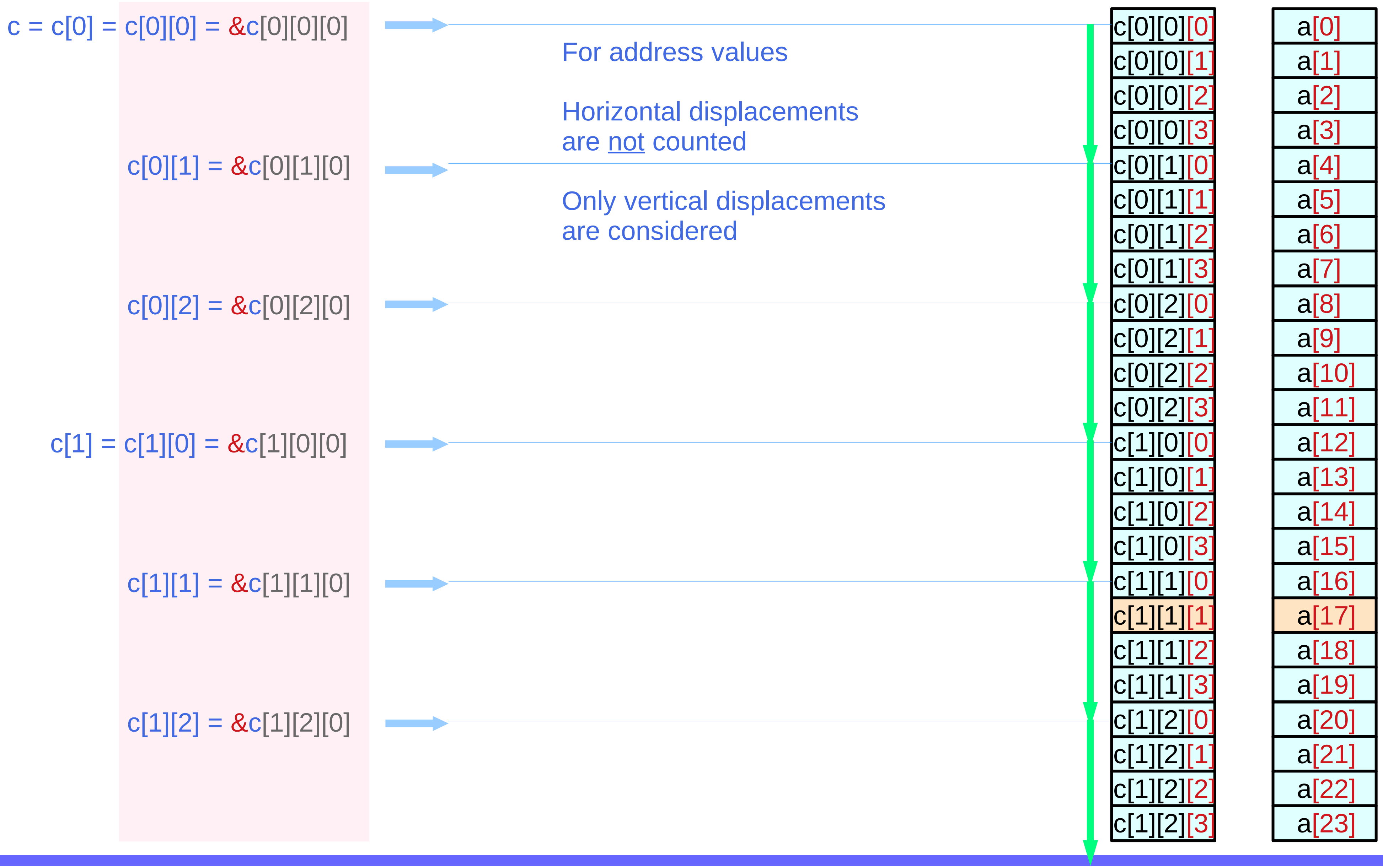
means

value(c[1][2]) = value(&c[1][2][0])

type(c[1][2]) = type(&c[1][2][0])

int *   int *

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



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

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

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

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

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

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

For address values

Horizontal displacements
are not counted

Only vertical displacements
are considered

c[0][0][0]	a[0]
c[0][0][1]	a[1]
c[0][0][2]	a[2]
c[0][0][3]	a[3]
c[0][1][0]	a[4]
c[0][1][1]	a[5]
c[0][1][2]	a[6]
c[0][1][3]	a[7]
c[0][2][0]	a[8]
c[0][2][1]	a[9]
c[0][2][2]	a[10]
c[0][2][3]	a[11]
c[1][0][0]	a[12]
c[1][0][1]	a[13]
c[1][0][2]	a[14]
c[1][0][3]	a[15]
c[1][1][0]	a[16]
c[1][1][1]	a[17]
c[1][1][2]	a[18]
c[1][1][3]	a[19]
c[1][2][0]	a[20]
c[1][2][1]	a[21]
c[1][2][2]	a[22]
c[1][2][3]	a[23]

Virtual array pointers – types, sizes, and values

int c[2][3][4];	**c[i][j]**	**c[i][j][0]**	
type	int [4] int (*)	int int	• abstract data type • array pointer type
size	sizeof(c[i][j]) =	sizeof(c[i][j][0]) * **4**	= sizeof(int) * **4**
value (address)	c[i][j] =	&c[i][j][0]	

int c[2][3][4];	**c[i]**	**c[i][0]**	
type	int [3][4] int (*)[4]	int [4] int (*)	• abstract data type • array pointer type
size	sizeof(c[i]) =	sizeof(c[i][0]) * **3**	= sizeof(int) * **4** * **3**
value (address)	c[i] =	&c[i][0][0]	

int c[2][3][4];	**c**	**c[0]**	
type	int [2][3][4] int (*)[3][4]	int [3][4] int (*)[4]	• abstract data type • array pointer type
size	sizeof(c) =	sizeof(c[0]) * **2**	= sizeof(int) * **4** * **3** * **2**
value (address)	c =	&c[0][0][0]	

Summary of virtual array pointers in a **3-d** array

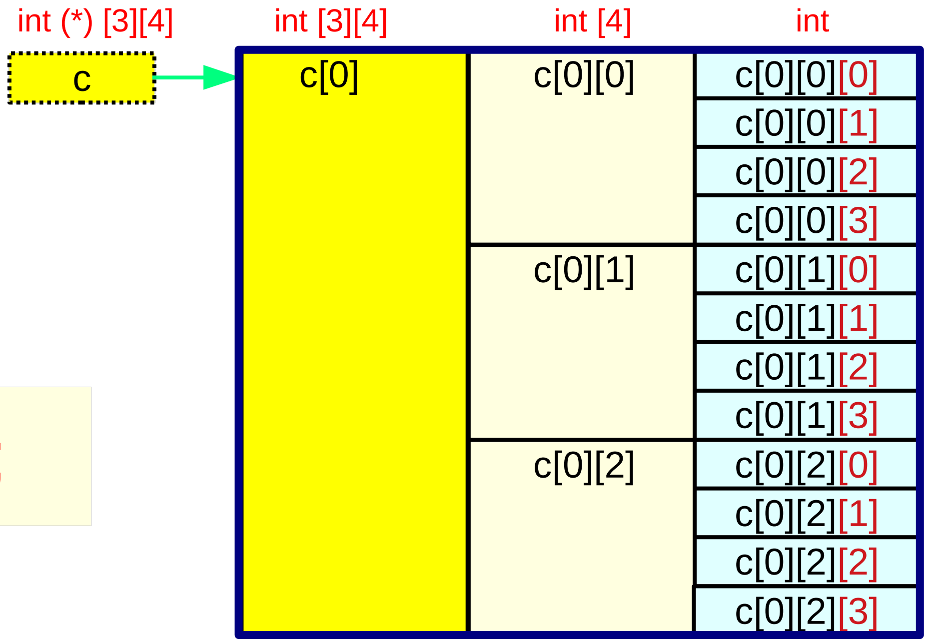
c[i] ≡ *(c +i)	c[i][j] ≡ *(c[i] +j)	c[i][j][k]≡ *(c[i][j] +k)
int (*) [3][4] 2-d array pointer c int [2] [3][4] 3-d array name c	int (*) [4] 1-d array pointers c[i] Int [3] [4] 2-d array names c[i]	int (*) 0-d array pointers c[i][j] int [4] 1-d array names c[i][j]
address value c + i &c[0][0][0] + i * sizeof(*c) &c[0][0][0] + i * sizeof(c[0]) &c[0][0][0] + i * 4 * 3 * 4	**address value** c[i] + j &c[i][0][0] + j * sizeof(*c[i]) &c[i][0][0] + j * sizeof(c[i][0]) &c[i][0][0] + j * 4 * 4	**address value** c[i][j] + k &c[i][j][0] + k * sizeof(*c[i][j]) &c[i][j][0] + k * sizeof(c[i][j][0]) &c[i][j][0] + k * 4
leading elements c[0][0][0]	**leading elements** c[0][0][0] c[1][0][0]	**leading elements** c[0][0][0] c[0][1][0] c[0][2][0] c[1][0][0] c[1][1][0] c[1][2][0]

Dual type constraints in a multi-dimensional array

Virtual array pointers to subarrays in a 3-d array

virtual 2-d array pointer

sizeof(**c**) =
sizeof(**c[0]**) * **2**



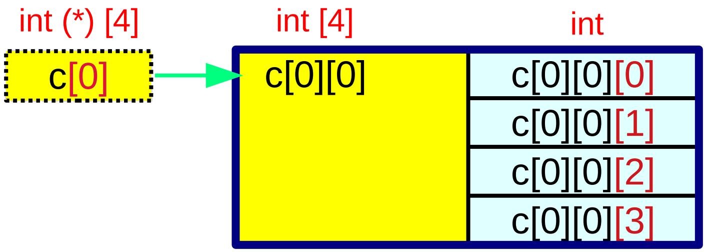
the first 2-d subarray

sizeof(**c[0]**) =
sizeof(**int [3][4]**)

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

virtual 1-d array pointer

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

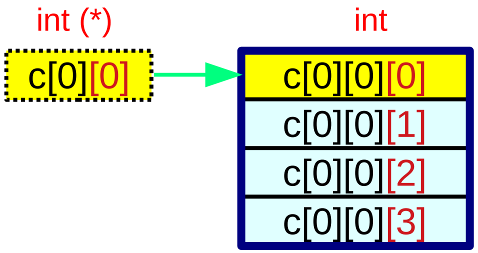


the first 1-d subarray

sizeof(**c[0][0]**) =
sizeof(**int [4]**)

virtual 0-d array pointer

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



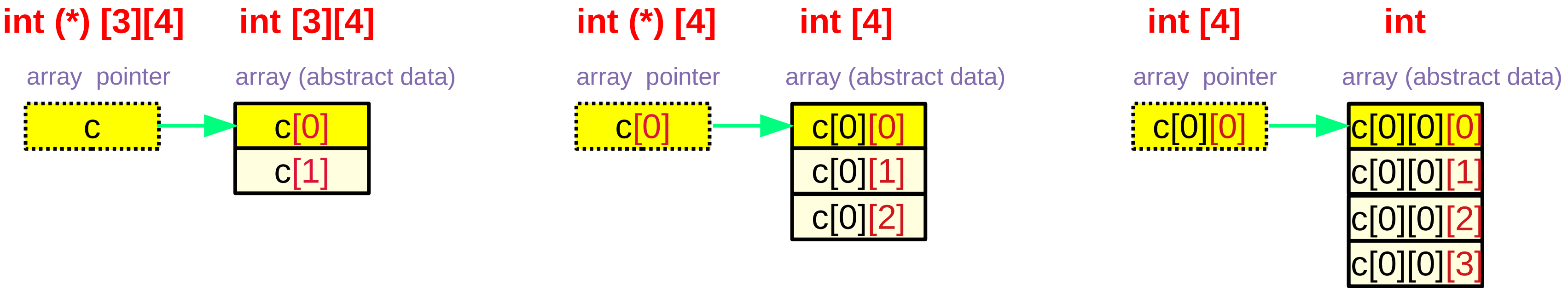
the first 0-d subarray

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

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

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

Types – array pointers



Sizes – abstract data

sizeof(**c**)
sizeof(**int [2][3][4]**)
sizeof(**int**) * **2** * **3** * **4**

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

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

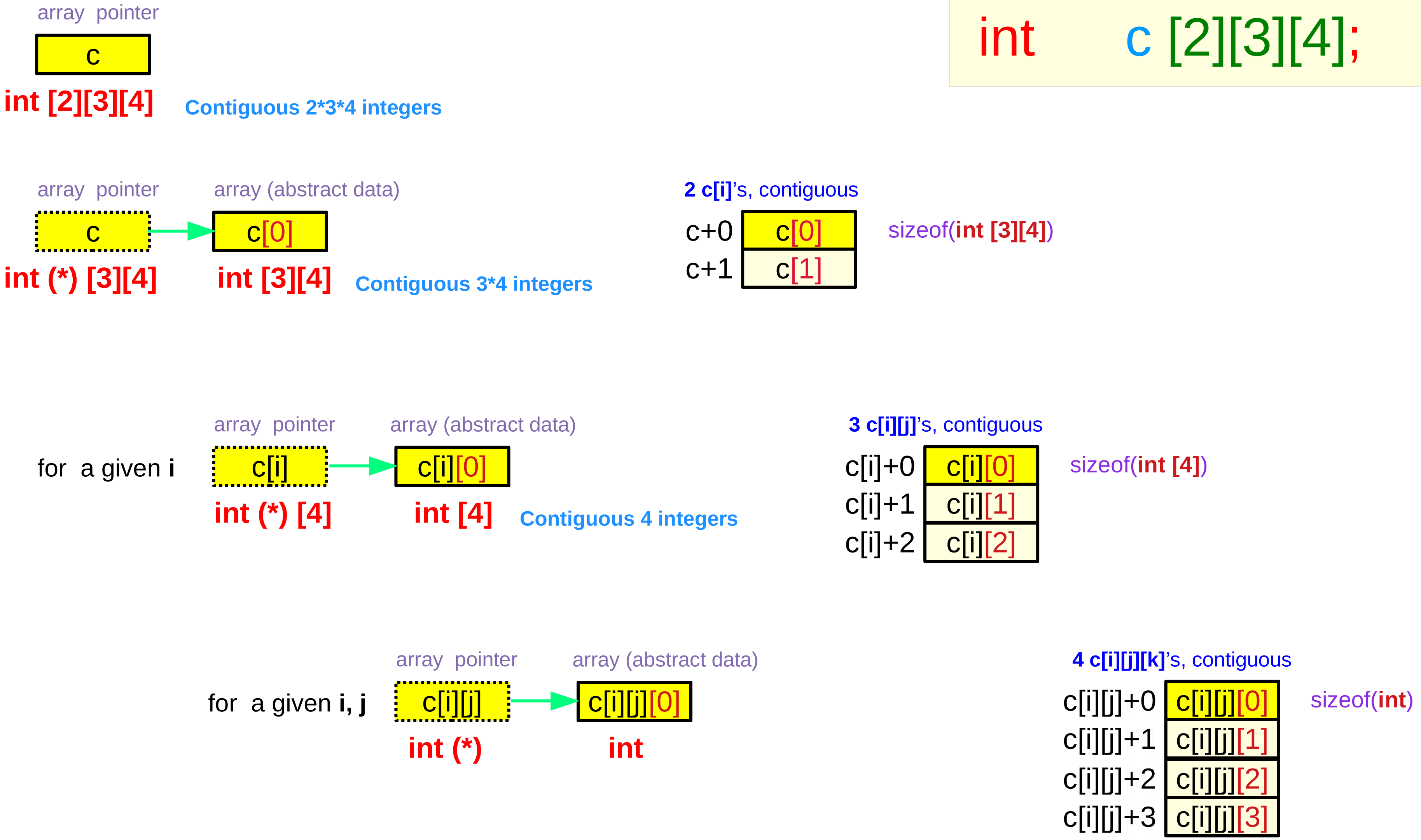
sizeof(**int [2][3][4]**) = 96
sizeof(**int (*)[3][4]**) = 4 / 8

sizeof(**int [3][4]**) = 48
sizeof(**int (*)[4]**) = 4 / 8

sizeof(**int [4]**) = 16
sizeof(**int (*)**) = 4 / 8

Contiguous subarrays **c[i]**, **c[i][j]**, **c[i][j][k]**

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



array pointer

c

int [2][3][4]

Contiguous 2*3*4 integers

array pointer

array (abstract data)

c → c[0]

int (*) [3][4]

int [3][4]

Contiguous 3*4 integers

2 c[i]’s, contiguous

c+0 c[0]

c+1 c[1]

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

for a given **i**

array pointer

array (abstract data)

c[i] → c[i][0]

int (*) [4]

int [4]

Contiguous 4 integers

3 c[i][j]’s, contiguous

c[i]+0 c[i][0]

c[i]+1 c[i][1]

c[i]+2 c[i][2]

sizeof(**int [4]**)

for a given **i, j**

array pointer

array (abstract data)

c[i][j] → c[i][j][0]

int (*)

int

4 c[i][j][k]’s, contiguous

c[i][j]+0 c[i][j][0]

c[i][j]+1 c[i][j][1]

c[i][j]+2 c[i][j][2]

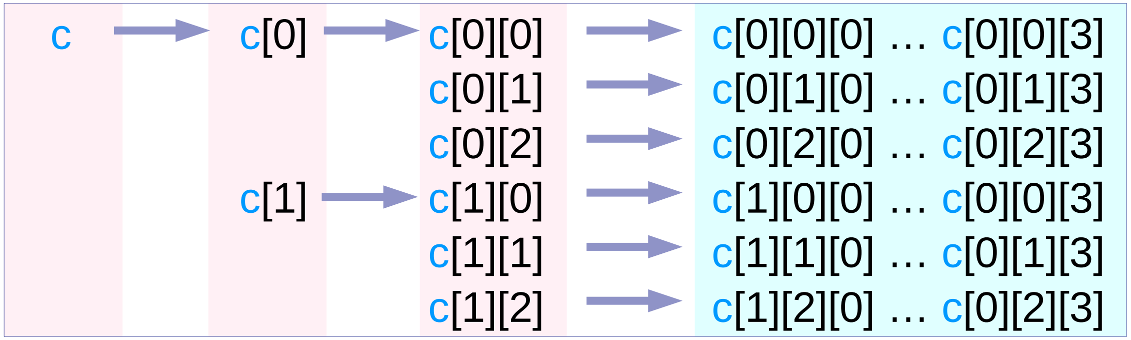
c[i][j]+3 c[i][j][3]

sizeof(**int**)

Dual types of **c**, **c[i]**, **c[i][j]**

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

c :: int (*) [3][4]	2-d array pointers
c[i] :: int [3] [4]	2-d arrays
c[i] :: int (*) [4]	1-d array pointers
c[i][j] :: int [4]	1-d arrays
c[i][j] :: int (*)	0-d array pointers
c[i][j][k] :: int	0-d arrays (integers)

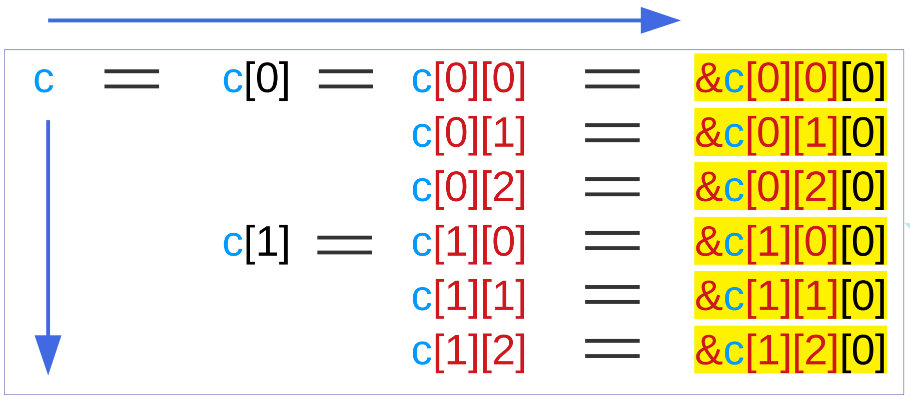


int [2] [3][4]	int [3] [4]	int [4]	int	…	int
int (*) [3][4]	int (*) [4]	int (*)	int	…	int

Values of virtual array pointers **c**, **c[i]**, **c[i][j]**

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

virtual array pointers have address values
in each row in the following figure
have the <u>same address value</u>



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

Horizontal displacements are <u>not</u> counted
only **vertical displacements** are considered
for address values

virtual assignments

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

Finding values of virtual array pointers **c**, **c[i]**, **c[i][j]**

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

append [0] to the right

c	$\overset{+[0]}{\equiv}$	c[0]	$\overset{+[0]}{\equiv}$	c[0][0]	$\overset{+[0]}{\equiv}$	&c[0][0][0]
				c[0][1]	$\overset{+[0]}{\equiv}$	&c[0][1][0]
				c[0][2]	$\overset{+[0]}{\equiv}$	&c[0][2][0]
		c[1]	$\overset{+[0]}{\equiv}$	c[1][0]	$\overset{+[0]}{\equiv}$	&c[1][0][0]
				c[1][1]	$\overset{+[0]}{\equiv}$	&c[1][1][0]
				c[1][2]	$\overset{+[0]}{\equiv}$	&c[1][2][0]
int (*) [3][4]		int (*) [4]		int [4]		int

c[0][0][0] **:** leading elements of **c**	**c[i]**[0][0] **:** leading elements of **c[i]**	**c[i][j]**[0] **:** leading elements of **c[i][j]**
&c[0][0][0]	&c[0][0][0] &c[1][0][0]	&c[0][0][0] &c[0][1][0] &c[0][2][0] &c[1][0][0] &c[1][1][0] &c[1][2][0]

Finding sub-arrays with the address &c[i][j][0]

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

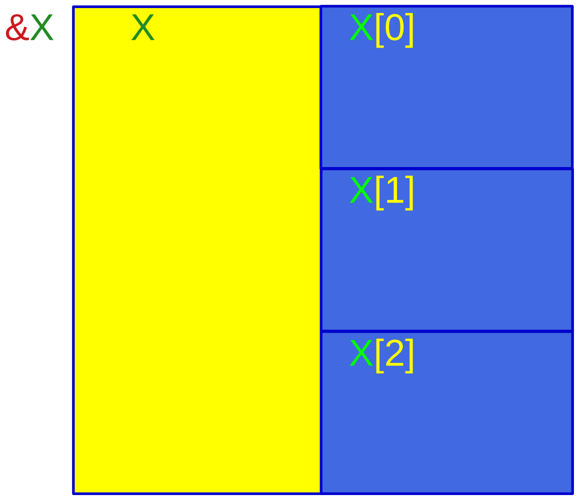
delete [0] from the right

&c[0][0][0]	$\overset{-[0]}{\equiv}$	c[0][0]	$\overset{-[0]}{\equiv}$	c[0]	$\overset{-[0]}{\equiv}$	c
&c[0][1][0]	$\overset{-[0]}{\equiv}$	c[0][1]				
&c[0][2][0]	$\overset{-[0]}{\equiv}$	c[0][2]				
&c[1][0][0]	$\overset{-[0]}{\equiv}$	c[1][0]	$\overset{-[0]}{\equiv}$	c[1]		
&c[1][1][0]	$\overset{-[0]}{\equiv}$	c[1][1]				
&c[1][2][0]	$\overset{-[0]}{\equiv}$	c[1][2]				
int		int [4]		int (*) [4]		int (*) [3][4]

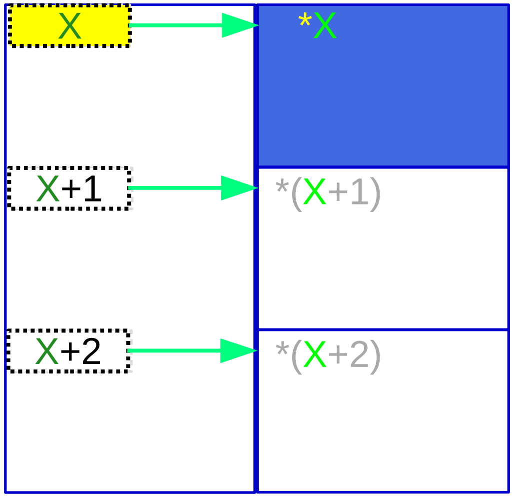
c[0][0][0] is the leading element of c[0][0], c[0], c
c[0][1][0] is the leading element of c[0][1]
c[0][2][0] is the leading element of c[0][2]
c[1][0][0] is the leading element of c[1][0], c[1]
c[1][1][0] is the leading element of c[1][1]
c[1][2][0] is the leading element of c[1][2]

Dual types in a **3-d** array

Abstract data (array) X



Virtual array pointer X



array (abstract data) array (abstract data)

c[i][j] starts from &c[i][j][0]
c[i] starts from &c[i][0]
c starts from &c[0]

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

array pointer array (abstract data)

c[i][j] points to c[i][j][0]
c[i] points to c[i][0]
c points to c[0]

address value

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

Dual type constraints

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

+

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

c[i][j] = &c[i][j]
c[i] = &c[i]
c = &c

Virtual array pointer

array pointer		array (abstract data)	
c[i][j]	points to	c[i][j][0]	X
c[i]	points to	c[i][0]	
c	points to	c[0]	

address value

Abstract data (array)

array (abstract data)		array (abstract data)	
c[i][j]	starts from	&c[i][j][0]	&X X
c[i]	starts from	&c[i][0]	
c	starts from	&c[0]	

array (abstract data)				Address of an array pointer
c[i][j]	pointer value	=	pointer address	&c[i][j]
c[i]	pointer value	=	pointer address	&c[i]
c	pointer value	=	pointer address	&c
X				&X

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

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

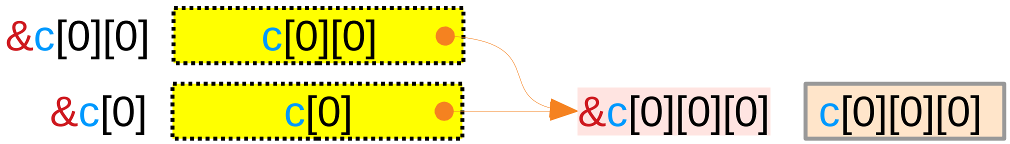
c = c[0] = c[0][0] = &c[0][0][0]

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

type(c[0]**)** = int (*)[4]
type(c[0][0]**)** = int [4]

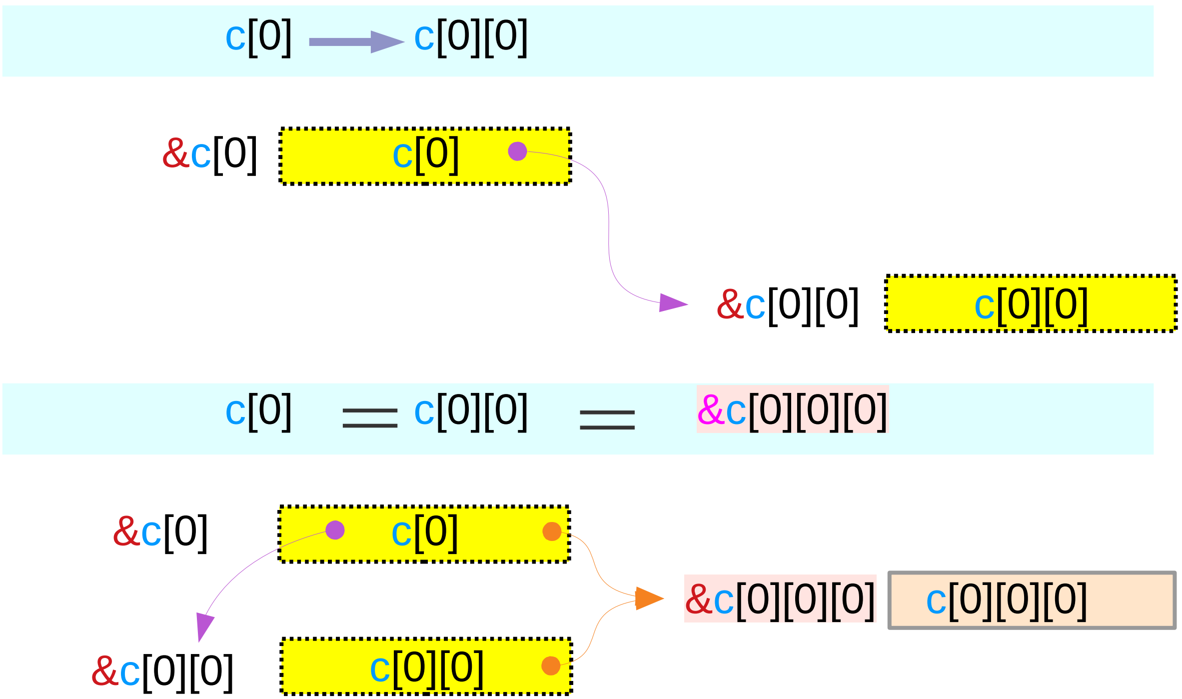
c[0] = c[0][0] *means*
value(c[0]**)** = **value(**c[0][0]**)**

c[0] = c[0][0] *does **not** mean*
type(c[0]**)** = **type(**c[0][0]**)**



Addresses and Values of **c[0]** and **c[0][0]**

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



c[0] → c[0][0]

&c[0] c[0]

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

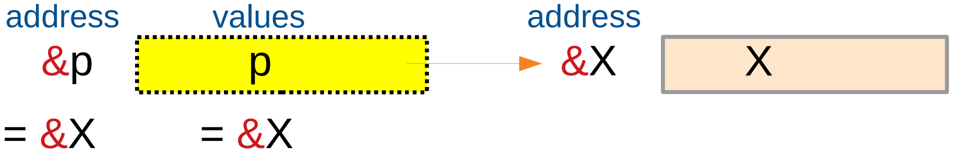
c[0] = c[0][0] = &c[0][0][0]

&c[0] c[0]

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

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

A virtual pointer's <u>address</u> and <u>value</u> are the same



address values address

&p p &X X

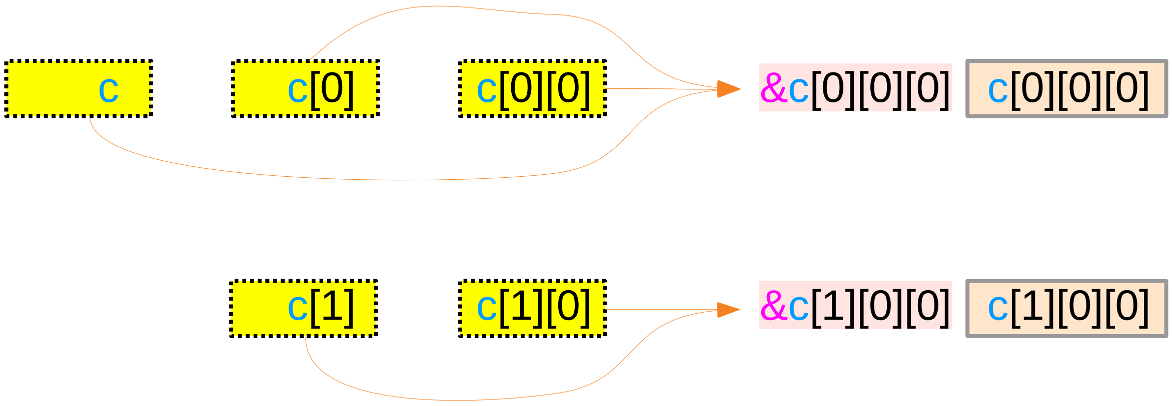
= &X = &X

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

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

c =	c[0] =	c[0][0] =	&c[0][0][0]	⇐ **value**
int(*)[3][4]	int(*)[4]	int(*)	int	⇐ **type**

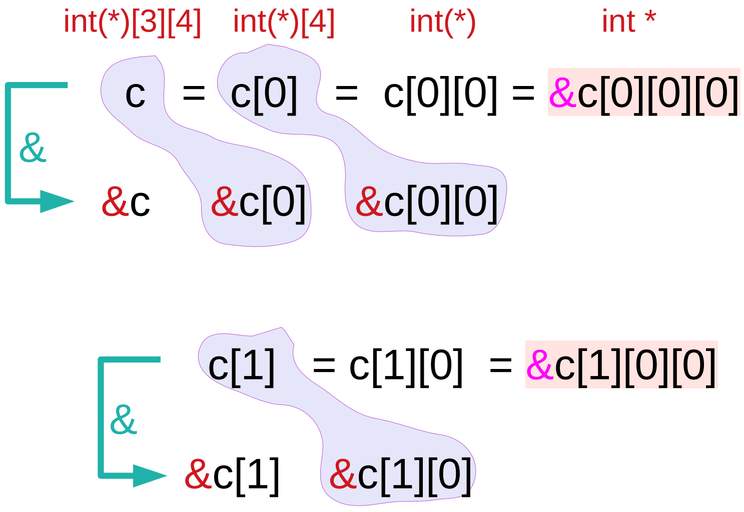
c[1] =	c[1][0] =	&c[1][0][0]	⇐ **value**
int(*)[4]	int(*)	int	⇐ **type**



These virtual pointers have different types
but the same value (address)

&c[i][0] and **&c[i][0][0]** – equivalence relations

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



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

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

&

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

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

&

&c[1] &c[1][0]

equivalences

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

equivalences

c[1] ≡ &c[1][0]
c[1][0] ≡ &c[1][0][0]

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

equivalences

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

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

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

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

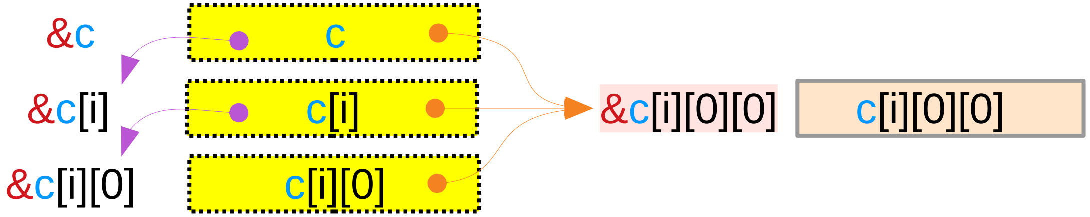
|| || ||

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

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

|| ||

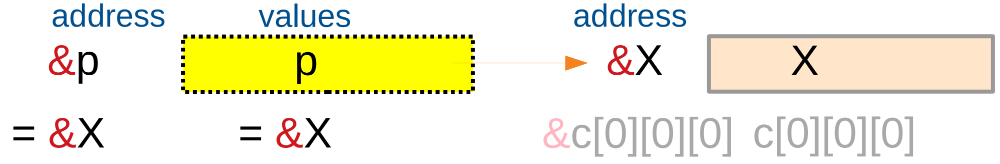
&c[1] = &c[1][0]

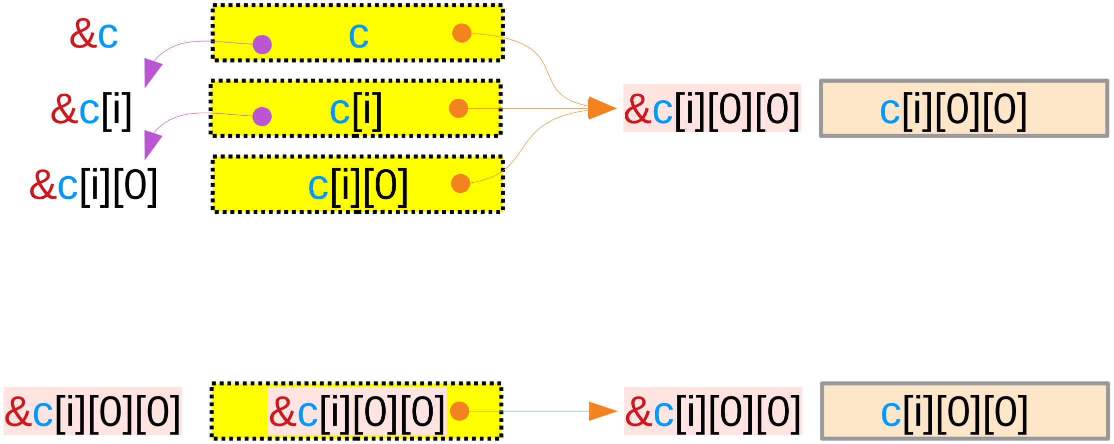


c[i] = &c[i] and c[i][0] = &c[i][0]

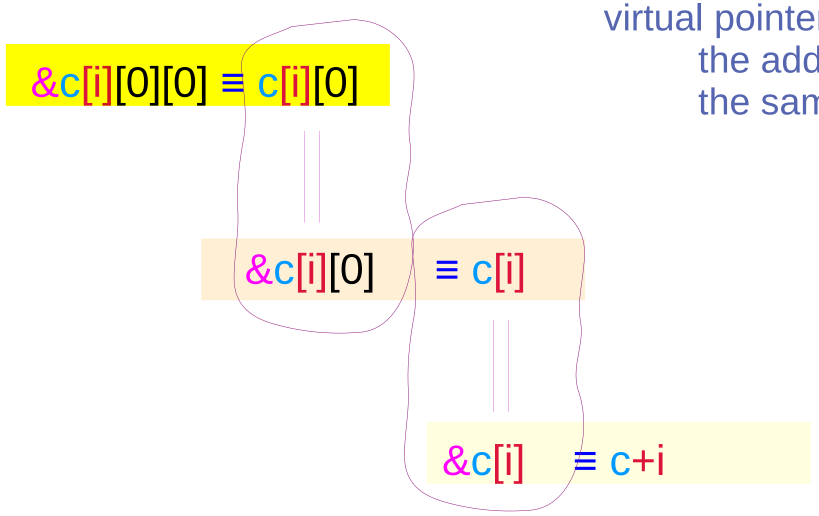
```
int     c [2][3][4];
```

A virtual pointer's <u>address</u> and <u>value</u> are the same





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



&c[i][0][0] ≡ c[i][0]

&c[i][0] ≡ c[i]

&c[i] ≡ c+i

= c + i*sizeof(*c)

= &c[0][0][0] + i*3*4

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

delete [0] from the right

$$\&c[0][0][0] \overset{-[0]}{=} c[0][0] \overset{-[0]}{=} c[0] \overset{-[0]}{=} c$$

$$\&c[1][0][0] \overset{-[0]}{=} c[1][0] \overset{-[0]}{=} c[1]$$

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

&c[i][j][0] ≡ c[i][j]

||

&c[i][j] ≡ c[i] + j

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

= c + i*sizeof(*c) + j*4

= &c[0][0][0] + i*3*4 + j*4

delete [0] from the right

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

Access expressions and dual type constrains

c **[i][j][k]** 3-d access

Hierarchical sub-arrays in a 3-d array

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

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

left-to-right associativity

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

Array Names and Types

Pointers to hierarchical sub-arrays

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

Associativity and Equivalence Relations

left-to-right associativity

left-to-right associativity

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

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

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

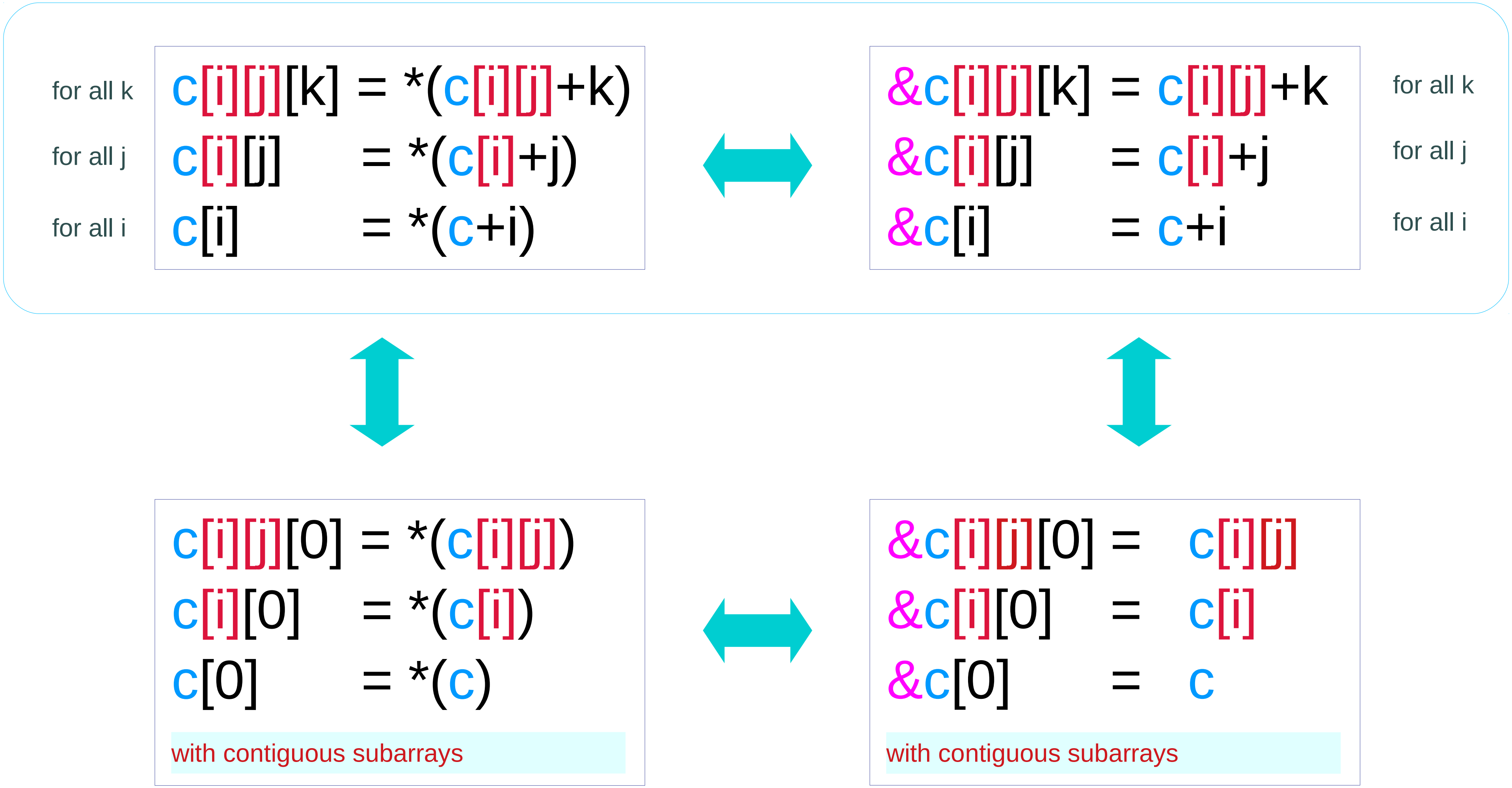
Requirements for the expression **c[i][j][k]**

for a given c[i][j],	for all k	c[i][j][k] = *(c[i][j]+k)
for a given c[i],	for all j	c[i][j] = *(c[i]+j)
for a given c,	for all i	c[i] = *(c+i)

3 contiguity requirements

for a given c[i][j],	contiguous c[i][j][k]’s
for a given c[i],	contiguous c[i][j]’s
for a given c,	contiguous c[i]’s
for a given subarray pointer	contiguous subarrays

Equivalent requirements for the expression **c[i][j][k]**



for all k c[i][j][k] = *(c[i][j]+k)

for all j c[i][j] = *(c[i]+j)

for all i c[i] = *(c+i)

&c[i][j][k] = c[i][j]+k for all k

&c[i][j] = c[i]+j for all j

&c[i] = c+i for all i

c[i][j][0] = *(c[i][j])

c[i][0] = *(c[i])

c[0] = *(c)

with contiguous subarrays

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

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

&c[0] = c

with contiguous subarrays

3 contiguity requirements for the expression **c[i][j][k]**

&c[i][j][k] = c[i][j]+k   for all k   = c[i][j] + k *sizeof(c[i][j][0])

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

&c[i] = c+i   for all i   = c + I *sizeof(c[0])

consecutive c[i][j][k]   for all k   = &c[i][j][0] + k * 4

consecutive c[i][j]   for all j   = &c[i][0][0] + j * 4*4

consecutive c[i]   for all i   = &c[0][0][0]+ i * 3*4*4

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

General requirements

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

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

Pointer array approach

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

```
c[i][j][k]     :: int
c[i][j]        :: int *
c[i]           :: int **
c              :: int ***
```

```
c[i]    ←   &b[i*3]
b[j]    ←   &a[j*4]
```

with contiguous a, b, c

Explicit
Arrays of pointers with
Multiple Indirection

N-dim Array approach

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

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

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

with contiguous c

Implicit
Nested
Virtual Array Pointers

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

General requirements

c [i][j][k]

&c[i][j][k] = c[i][j]+k for all k
&c[i][j] = c[i]+j for all j
&c[i] = c+i for all i

N-dim Array approach

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

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

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

with contiguous c

Implicit
Nested
Virtual Array Pointers

Virtual assignments

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

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

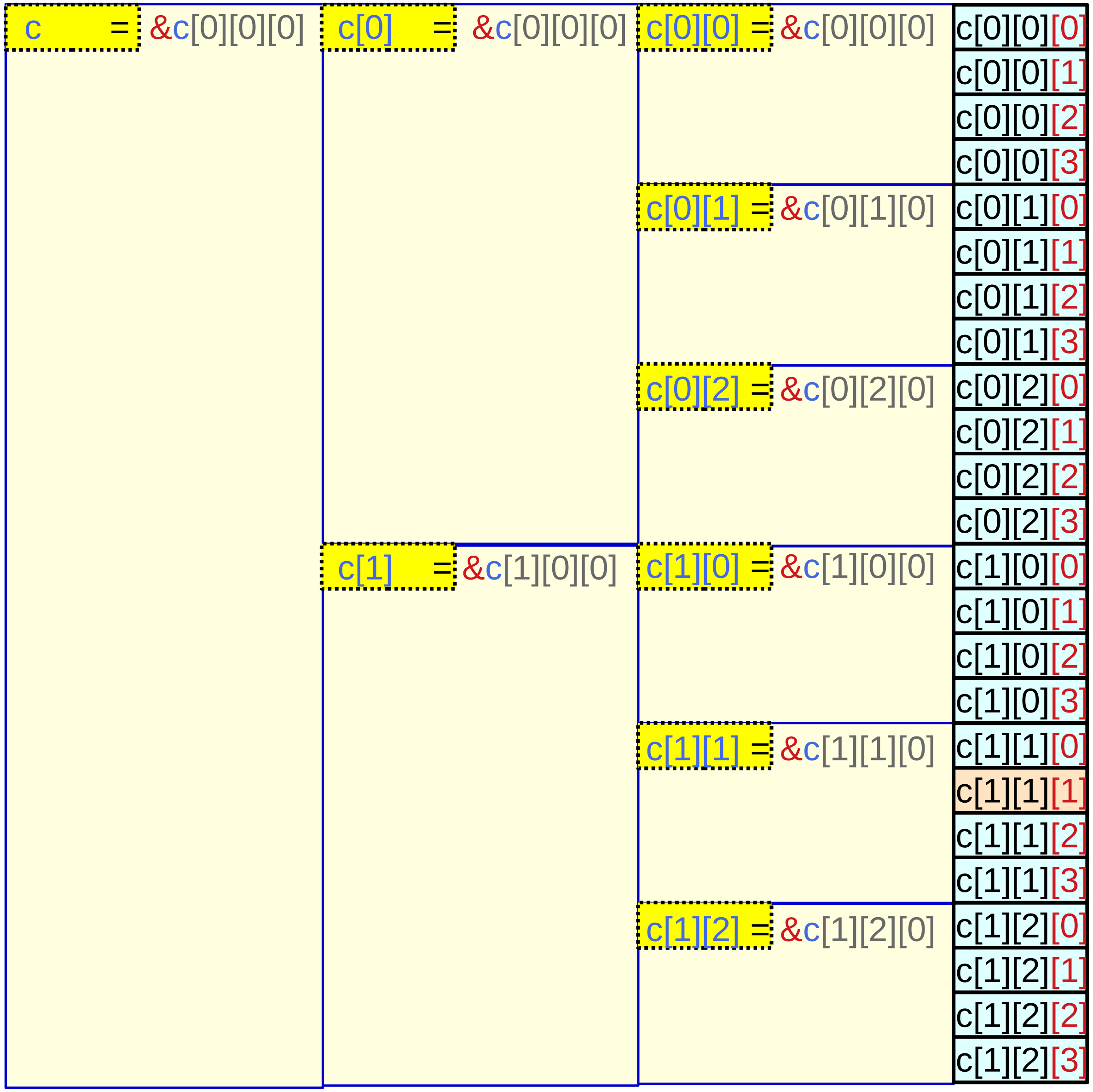
virtual assignments

c ← &c[0][0][0]

c[i] ← &c[i][0][0]

c[i][j] ← &c[i][j][0]

row major ordering
contiguous linear layout



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

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

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

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

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

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

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

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

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

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

Virtual assignments

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

c [i][j][k];

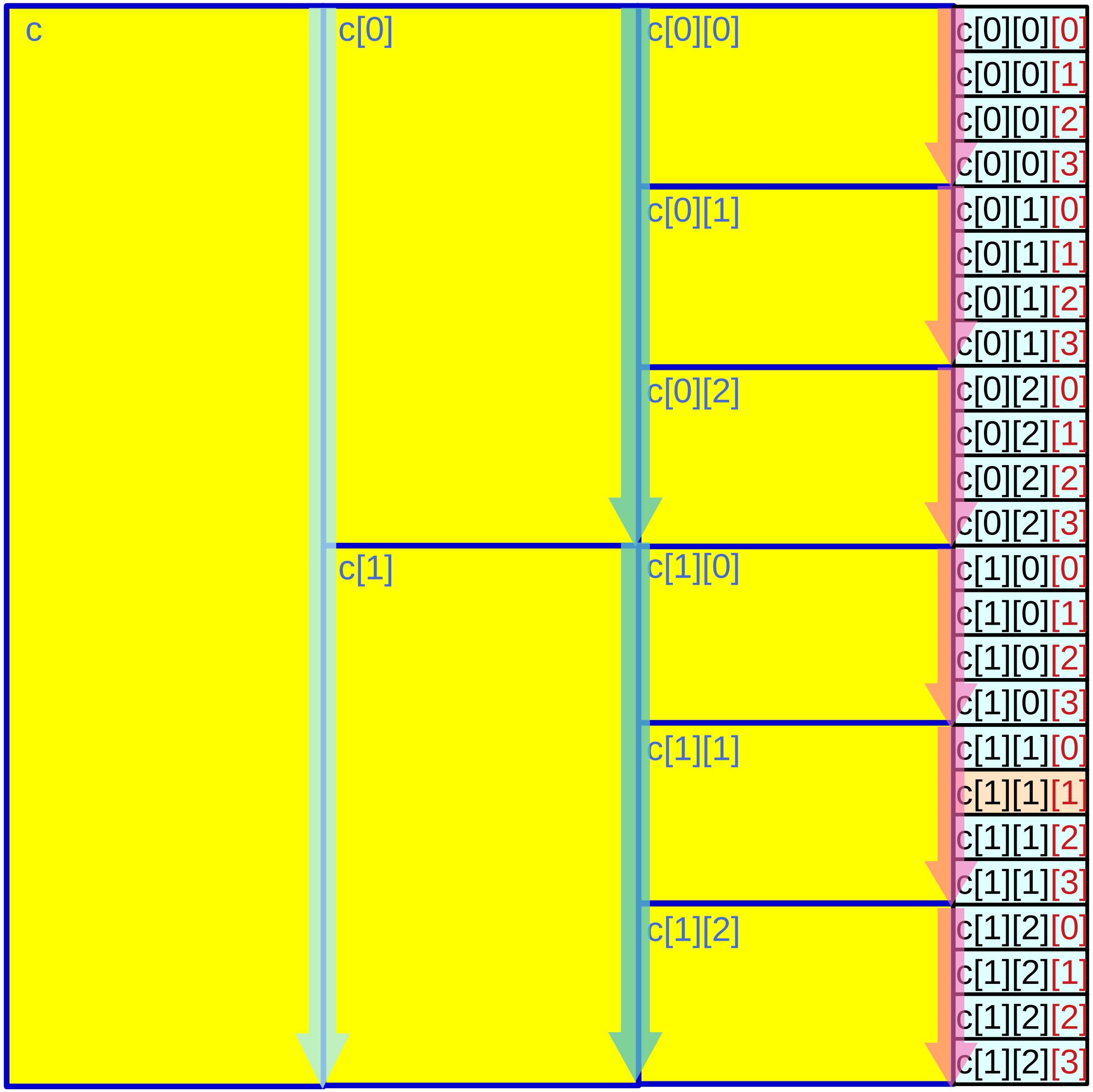
Three contiguity requirements

&c[i][j][k] = c[i][j]+k for all k

&c[i][j] = c[i]+j for all j

&c[i] = c+i for all i

for a given c[i][j],	contiguous c[i][j][k]'s
for a given c[i],	contiguous c[i][j]'s
for a given c,	contiguous c[i]'s



Virtual assignments and type casts

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

↓

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

virtual assignments

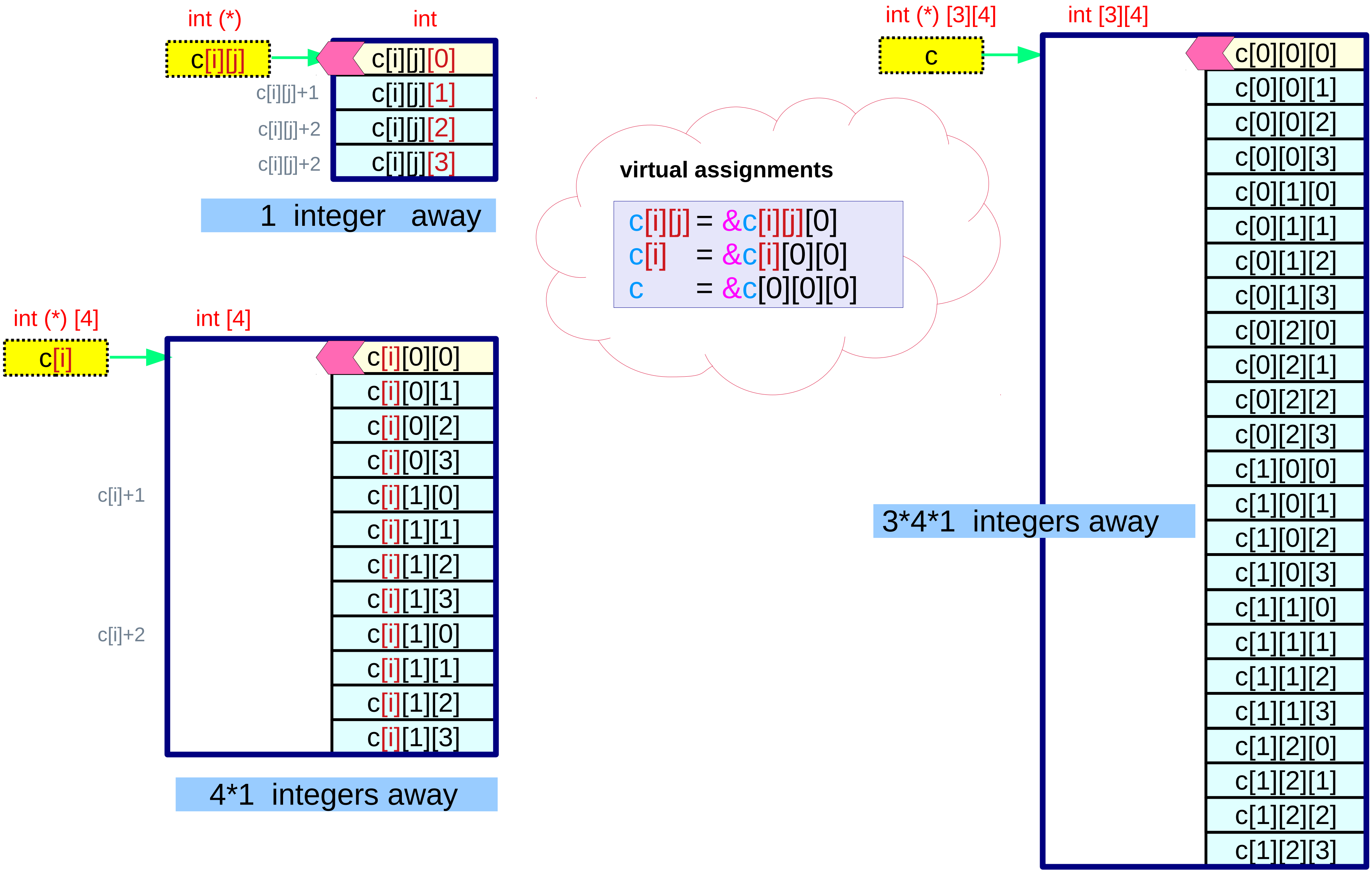
```
c       ← &c[0][0][0]
c[i]    ← &c[i][0][0]
c[i][j] ← &c[i][j][0]
```

row major ordering
contiguous linear layout

		type casts	**address values**
c	←	(int (*)[3][4])	&c[0][0][0]
c[i]	←	(int (*)[4])	&c[i][0][0]
c[i][j]	←	(int (*))	&c[i][j][0]

if c, c[i], c[i][j] were real pointer variables,
type casts would be needed

Assigning **c[i][j]**, **c[i]**, **c**



int (*)

c[i][j]

int

c[i][j][0]

c[i][j]+1 c[i][j][1]

c[i][j]+2 c[i][j][2]

c[i][j]+2 c[i][j][3]

1 integer away

int (*) [4]

c[i]

int [4]

c[i][0][0]

c[i][0][1]

c[i][0][2]

c[i][0][3]

c[i]+1 c[i][1][0]

c[i][1][1]

c[i][1][2]

c[i][1][3]

c[i]+2 c[i][1][0]

c[i][1][1]

c[i][1][2]

c[i][1][3]

4*1 integers away

virtual assignments

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

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

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

int (*) [3][4]

c

int [3][4]

c[0][0][0]

c[0][0][1]

c[0][0][2]

c[0][0][3]

c[0][1][0]

c[0][1][1]

c[0][1][2]

c[0][1][3]

c[0][2][0]

c[0][2][1]

c[0][2][2]

c[0][2][3]

c[1][0][0]

c[1][0][1]

c[1][0][2]

c[1][0][3]

c[1][1][0]

c[1][1][1]

c[1][1][2]

c[1][1][3]

c[1][2][0]

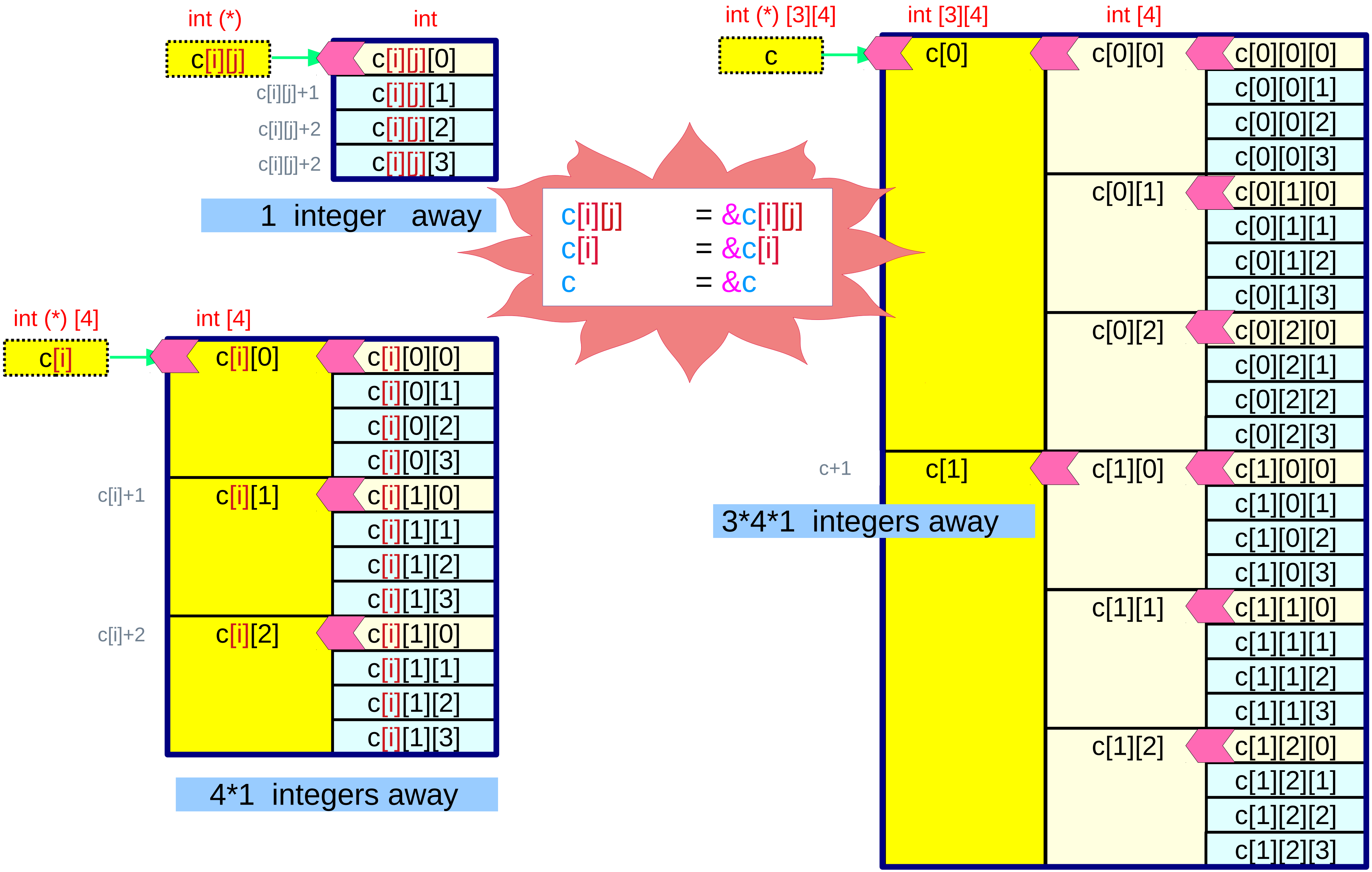
c[1][2][1]

c[1][2][2]

c[1][2][3]

3*4*1 integers away

The addresses **c[i][j]**, **c[i]**, **c** with dual type constraints



int (*) int

c[i][j]

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

1 integer away

c[i][j] = &c[i][j]
c[i] = &c[i]
c = &c

int (*) [4] int [4]

c[i]

c[i][0]: c[i][0][0], c[i][0][1], c[i][0][2], c[i][0][3]
c[i]+1 c[i][1]: c[i][1][0], c[i][1][1], c[i][1][2], c[i][1][3]
c[i]+2 c[i][2]: c[i][1][0], c[i][1][1], c[i][1][2], c[i][1][3]

4*1 integers away

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

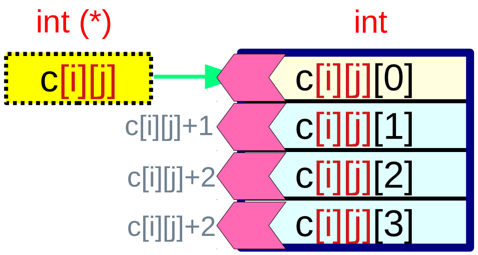
c

c[0]
c[0][0]: c[0][0][0], c[0][0][1], c[0][0][2], c[0][0][3]
c[0][1]: c[0][1][0], c[0][1][1], c[0][1][2], c[0][1][3]
c[0][2]: c[0][2][0], c[0][2][1], c[0][2][2], c[0][2][3]

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

3*4*1 integers away

The addresses **c[i][j]**+k, **c[i]**+j, **c**+i

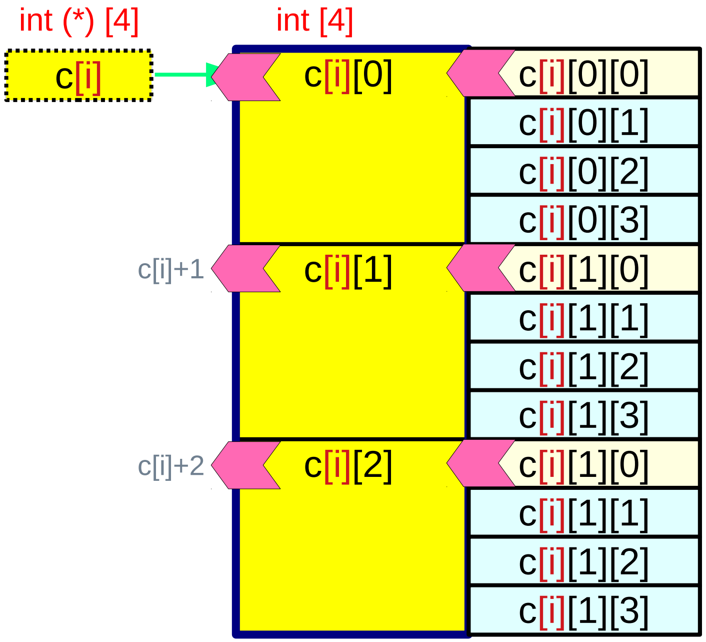


1 integer away

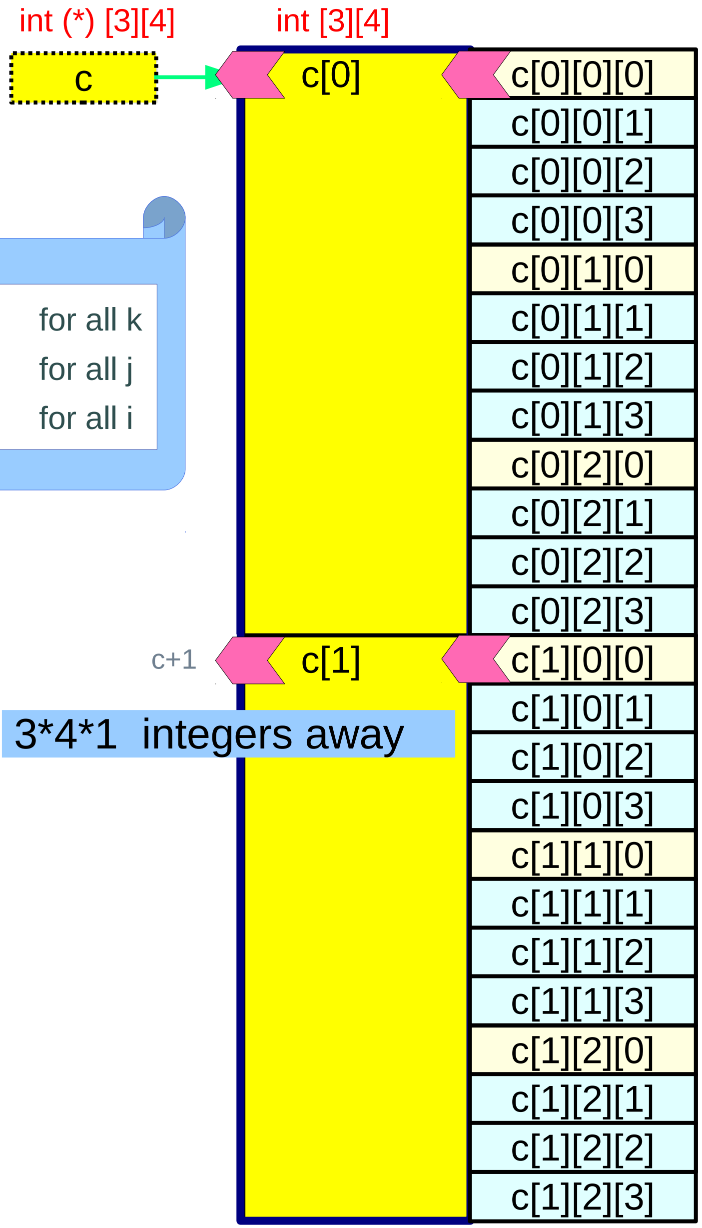
&c[i][j][k] = c[i][j]+k for all k

&c[i][j] = c[i]+j for all j

&c[i] = c+i for all i



4*1 integers away

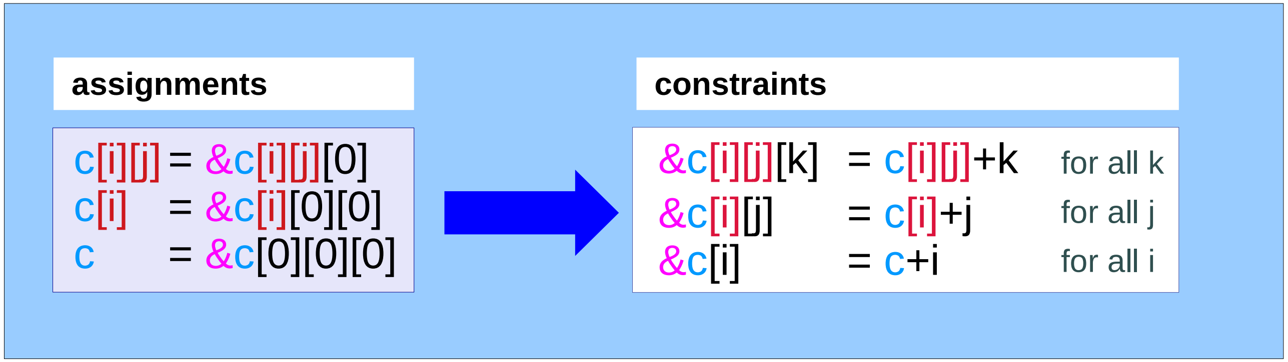


3*4*1 integers away

Assignment → 3 contiguity requirements

multi-dimensional arrays

c [i][j][k]

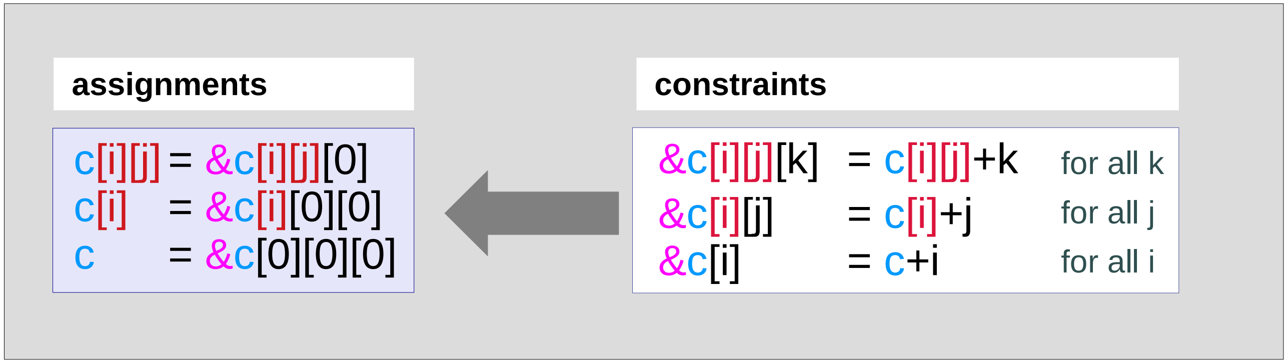


assignments

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

constraints

&c[i][j][k] = c[i][j]+k for all k
&c[i][j] = c[i]+j for all j
&c[i] = c+i for all i



assignments

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

constraints

&c[i][j][k] = c[i][j]+k for all k
&c[i][j] = c[i]+j for all j
&c[i] = c+i for all i

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

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

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

with contiguous subarrays

&c[i][j][k] = c[i][j]+k	for all k
&c[i][j] = c[i]+j	for all j
&c[i] = c+i	for all i

virtual assignments

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

Virtual assignments

int (*)	c[i][j]	=	(int (*))	&c[i][j][0]
int (*) [4]	c[i]	=	(int (*) [4])	&c[i][0][0]
int (*) [3][4]	c	=	(int (*) [3][4])	&c[0][0][0]

Pointer Types

Sizes of abstract data types

int [4]	c[i][j]	size =4*4
int [3][4]	c[i]	size =3*4*4
int [2][3][4]	c	size =2*3*4*4

Abstract Data Types

Strides of array elements

c[i][j][0]	stride =4*4
c[i][0][0]	stride =3*4*4
c[0][0][0]	stride =2*3*4*4

contiguous

c[i][j]	contains	4 integers	k=[0:3]
c[i]	contains	3*4 integers	j=[0:2], k=[0:3]
c	contains	2*3*4 integers	i=[0:1], j=[0:2], k=[0:3]

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

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

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

with contiguous subarrays

&c[i][j][k] = c[i][j]+k for all k
&c[i][j] = c[i]+j for all j
&c[i] = c+i for all i

virtual assignments

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

c[i][j]	has an address of	c[i][j][0]	as its value
c[i]	has an address of	c[i][0][0]	as its value
c	has an address of	c[0][0][0]	as its value
Pointer Types			

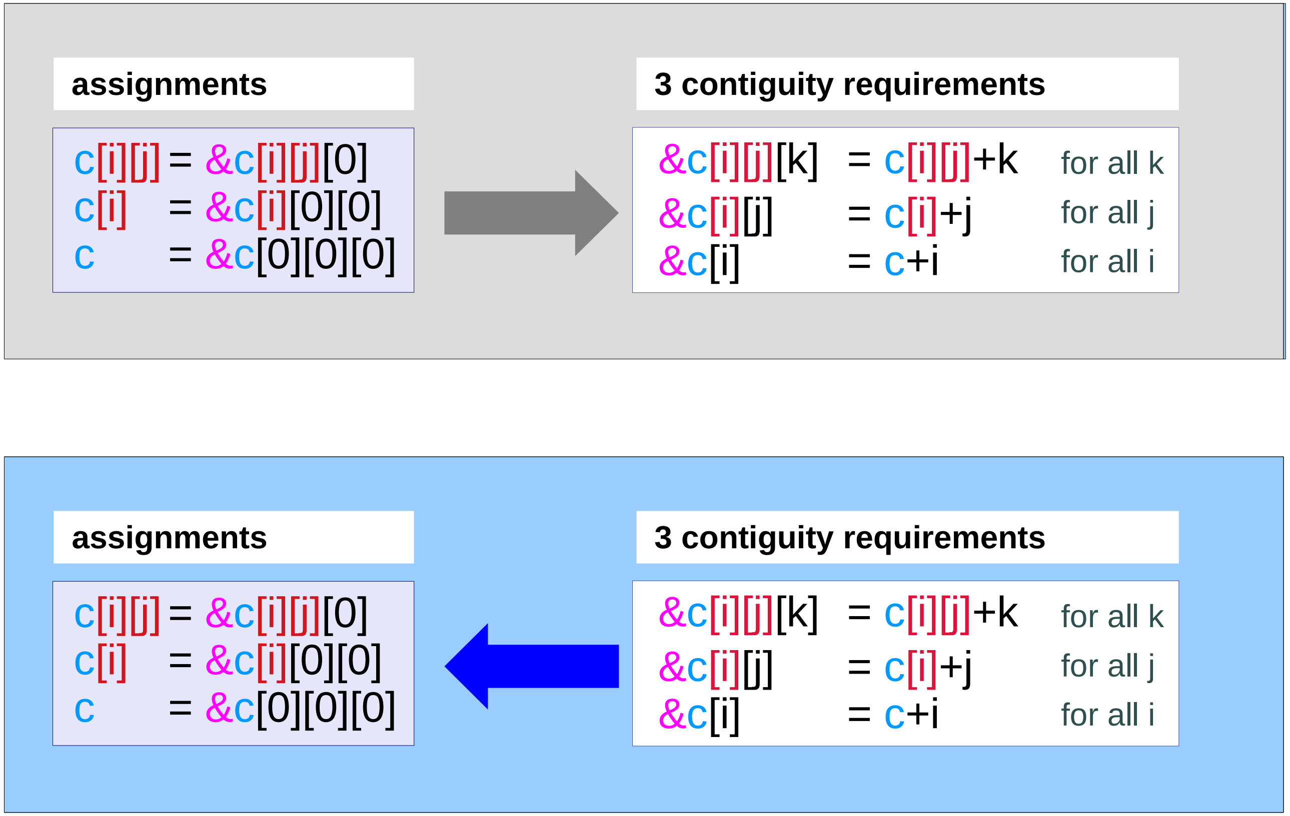
c[i][j]+1	has an address of	c[i][j][1]	1 integer away
c[i]+1	has an address of	c[i][1][0]	4*1 integers away
c+1	has an address of	c[1][0][0]	3*4*1 integers away
Pointer Types			

c[i][j]+k	has an address of	c[i][j][k]	1*k integers away
c[i]+j	has an address of	c[i][j][0]	4*1*j integers away
c+i	has an address of	c[i][0][0]	3*4*1*i integers away
Pointer Types			

Assignment ← 3 contiguity requirements

multi-dimensional arrays

c [i][j][k]



assignments

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

3 contiguity requirements

&c[i][j][k] = c[i][j]+k for all k
&c[i][j] = c[i]+j for all j
&c[i] = c+i for all i

assignments

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

3 contiguity requirements

&c[i][j][k] = c[i][j]+k for all k
&c[i][j] = c[i]+j for all j
&c[i] = c+i for all i

Subarray starting addresses

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

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

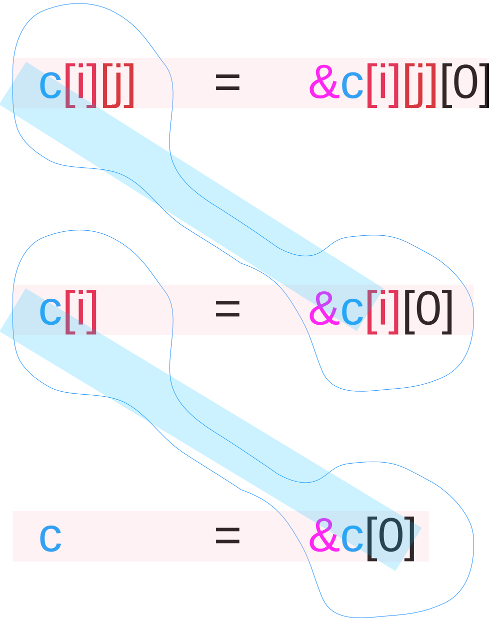
with contiguous subarrays

&c[i][j][k] = c[i][j]+k   for all k
&c[i][j] = c[i]+j   for all j
&c[i] = c+i   for all i

virtual assignments

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

Array pointer relationships



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

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

c = &c[0]

Dual type constraints

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

&c[0] = c[0]

Subarray starting addresses

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

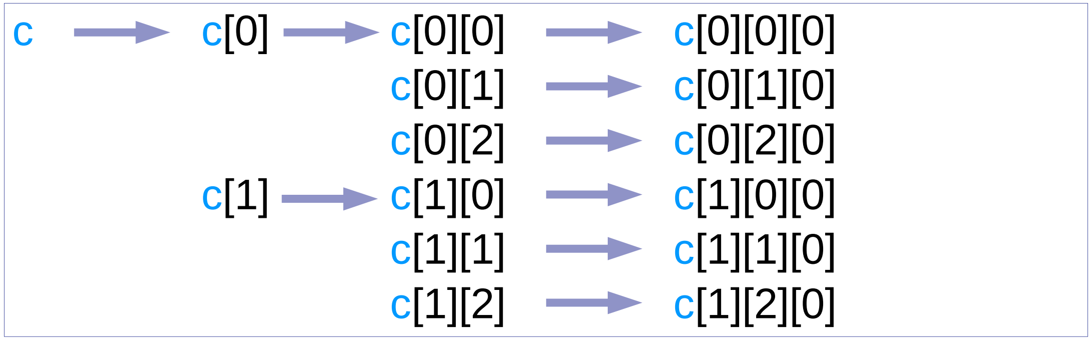
&c[i][j][0] = c[i][j]
&c[i][0] = c[i]
&c[0] = c

with contiguous subarrays

&c[i][j][k] = c[i][j]+k   for all k
&c[i][j] = c[i]+j   for all j
&c[i] = c+i   for all i

virtual assignments

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



c → c[0] → c[0][0] → c[0][0][0]
c[0][1] → c[0][1][0]
c[0][2] → c[0][2][0]
c[1] → c[1][0] → c[1][0][0]
c[1][1] → c[1][1][0]
c[1][2] → c[1][2][0]

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

contiguity

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

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

contiguity

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

dual type constraints

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

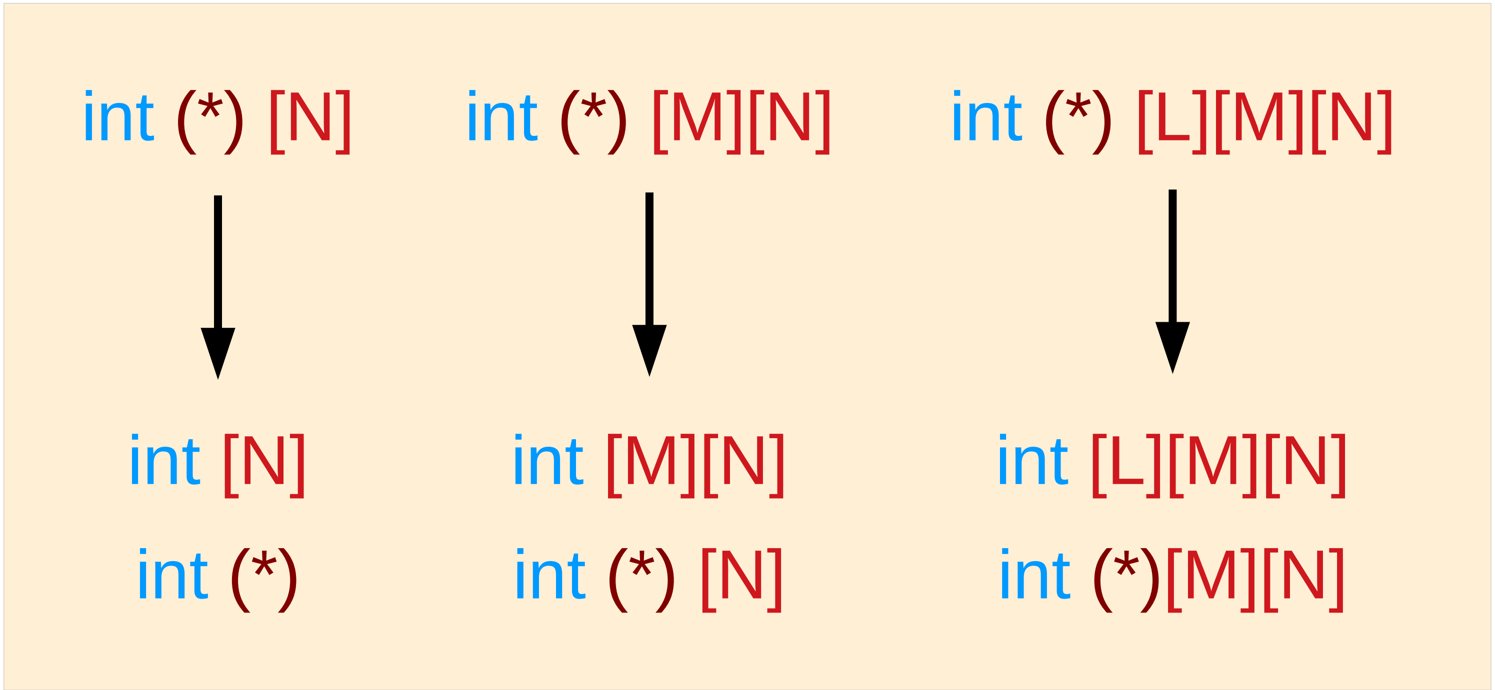
contiguity

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

dual type constraints

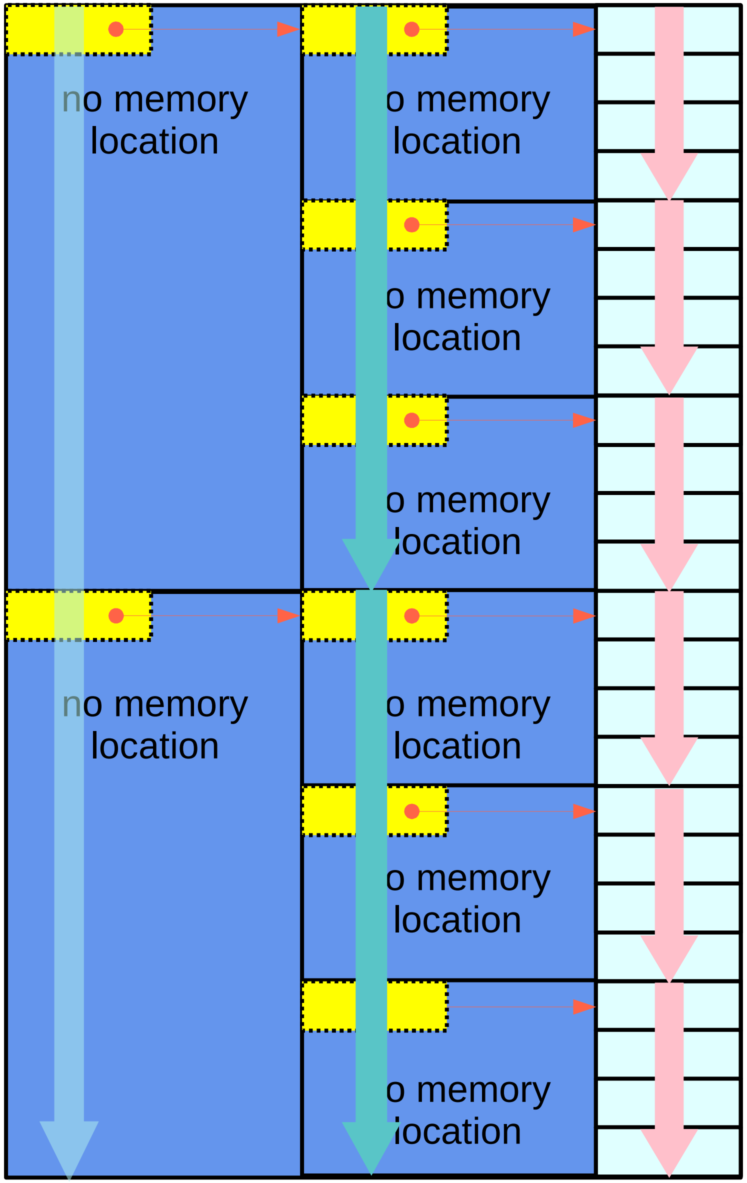
Contiguity constraints in a multi-dimensional array

Array pointers and dual types



int (*) [N]	int (*) [M][N]	int (*) [L][M][N]	Array Pointer
int [N]	int [M][N]	int [L][M][N]	Array Name
int (*)	int (*) [N]	int (*)[M][N]	Array Pointer

Array pointer approach – contiguity constraints



Array Pointer Approach (pointer to arrays)

Equivalence and contiguity (1)

consecutive address consecutive data

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

contiguous index : n

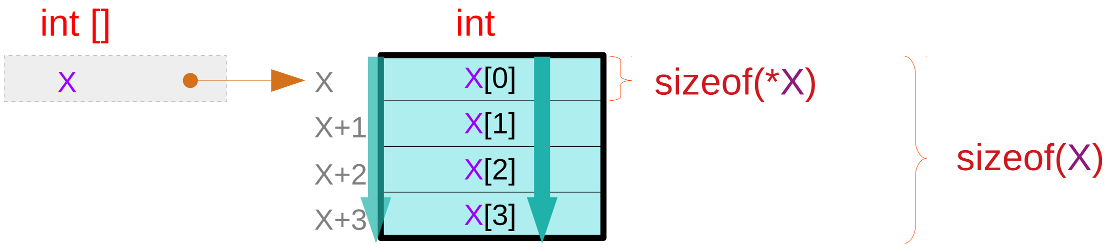
int	**X[4];**	contiguous X[i] for a given X :	**primitive types**
int *	**X[4];**	contiguous X[i] for a given X :	**pointer types**
atype	**X[4];**	contiguous X[i] for a given X :	**abstract data types**

Equivalence and contiguity (2)

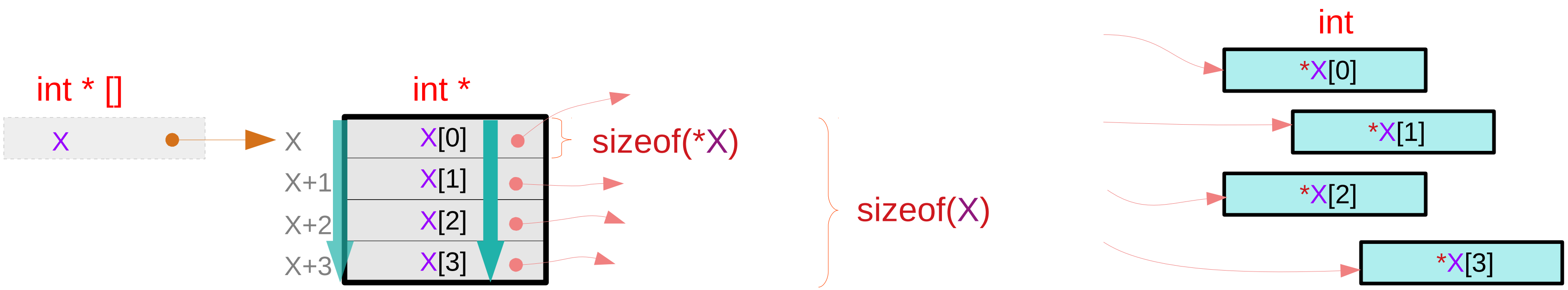
consecutive address   consecutive data

*(X+n) ≡ X[n]

contiguous index : n



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



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

Equivalence and contiguity (3)

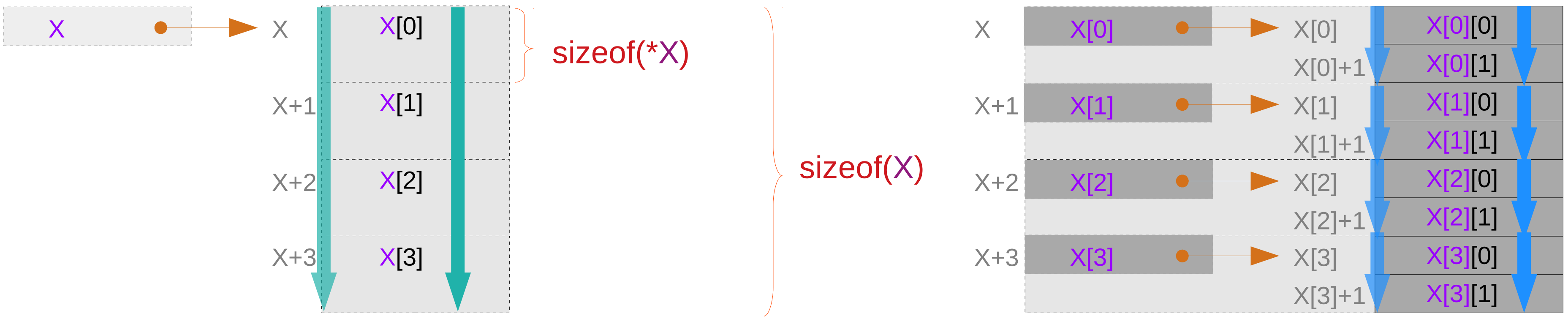
consecutive address

consecutive data

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

contiguous index : n

can be recursively applied



atype X[4]; contiguous X[i] for a given X : **abstract data types**

Recursive applications of equivalences

By definition, contiguous memory locations are assumed

consecutive address

consecutive data

*(X+n) ≡ X[n]

contiguous index : n

*(p[m]+n) ⟷ p[m][n]

X = p[m]

contiguous index : **n**

(*(p+m))[n]; ⟷ p[m][n];

X = p

contiguous index : **m**

Equivalence and contiguity (1)

consecutive address consecutive data

*(X+n) ≡ X[n]

contiguous index : n

*(p[m]+n) ⟺ p[m][n]

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

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

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

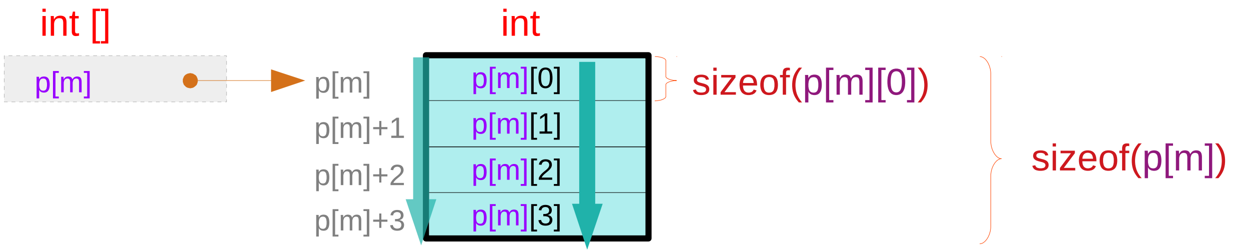
atype **p[M][4];** contiguous p[m][i] for a given p[m] : **abstract data types**

Equivalence for a given p[m] (2)

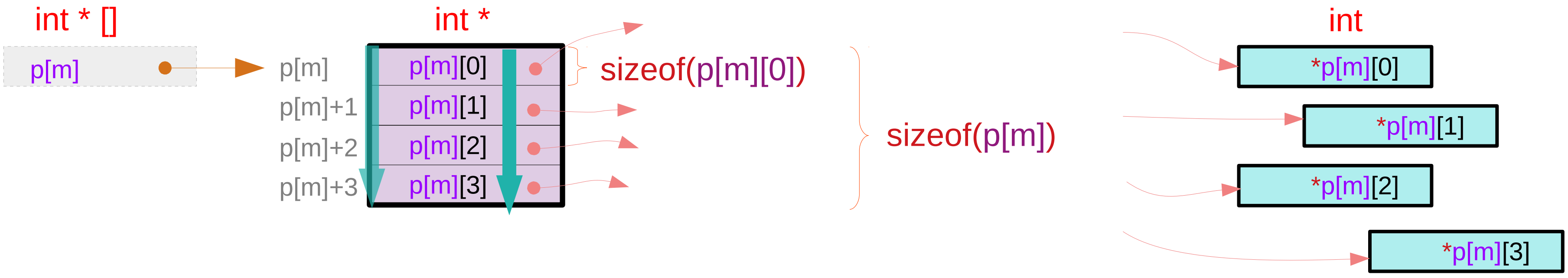
*(p[m]+n) ⟷ p[m][n]

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

int p[M][4]; contiguous p[m][n] for a given p[m] : **primitive types**   m = 0, 1, …, M-1



int * p[M][4]; contiguous p[m][n] for a given p[m] : **pointer types**   m = 0, 1, …, M-1

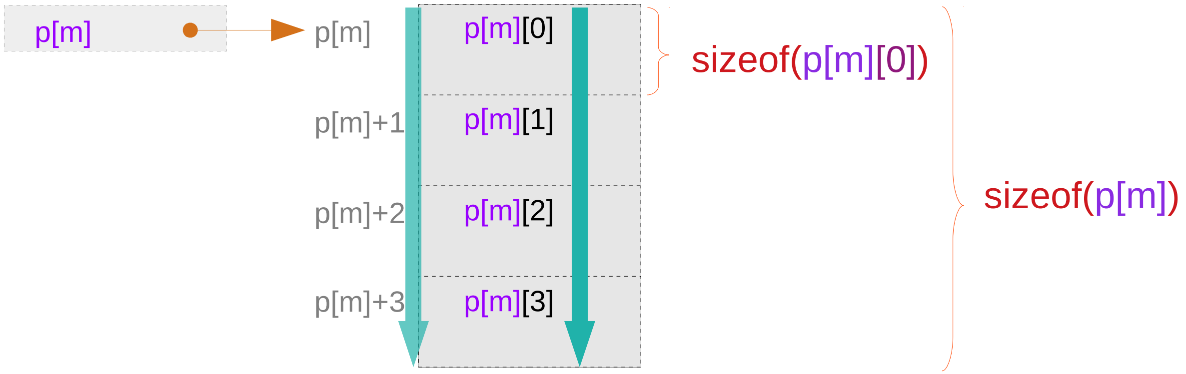


Equivalence for a given p[m] (3)

*(p[m]+n) ⟺ p[m][n]

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

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



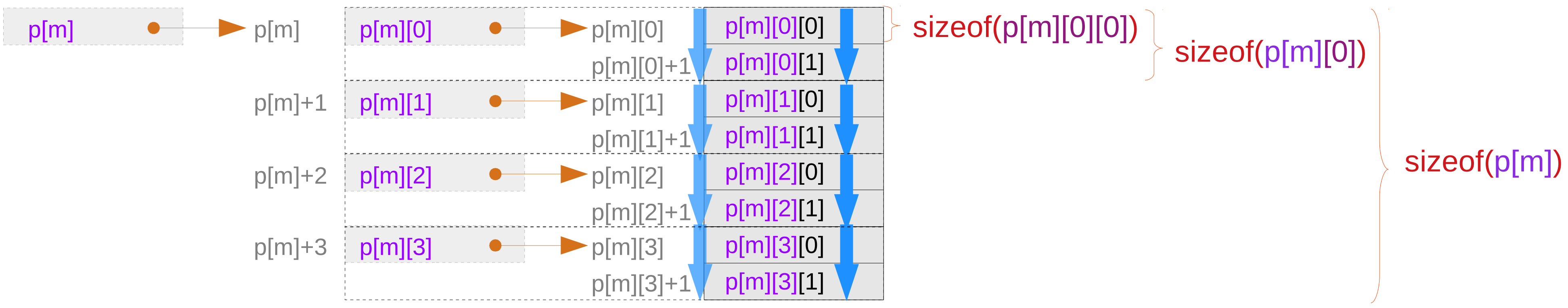
can be recursively applied

Equivalence for a given p[m][n]

*(p[m][n]+k) ⟺ p[m][n][k]

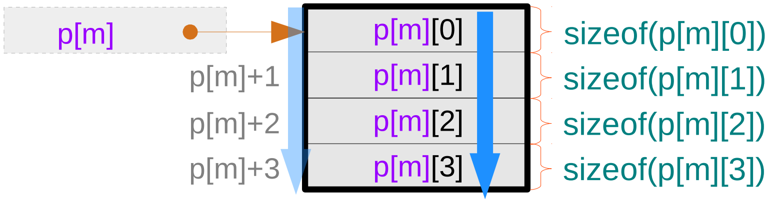
for a given p[m][n] contiguous index : **k**

atype * p[M][4][2]; contiguous p[m][n][k] for a given p[m][n] : **abstract data types** m = 0, 1, …, M-1



Contiguity constraints in multi-dimensional arrays

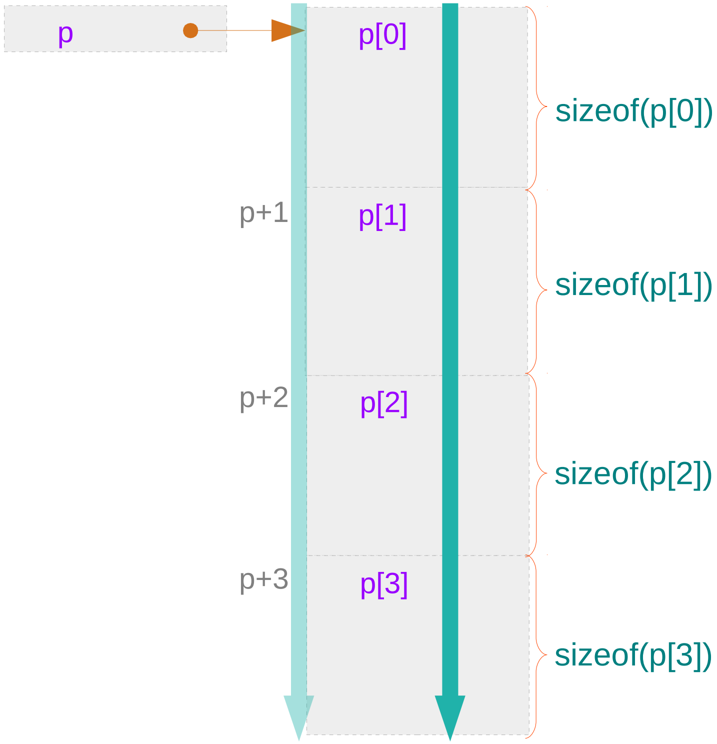
*(p[m]+n) ⟺ p[m][n]



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

contiguous index : n

*(p+m) ⟺ p[m]



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

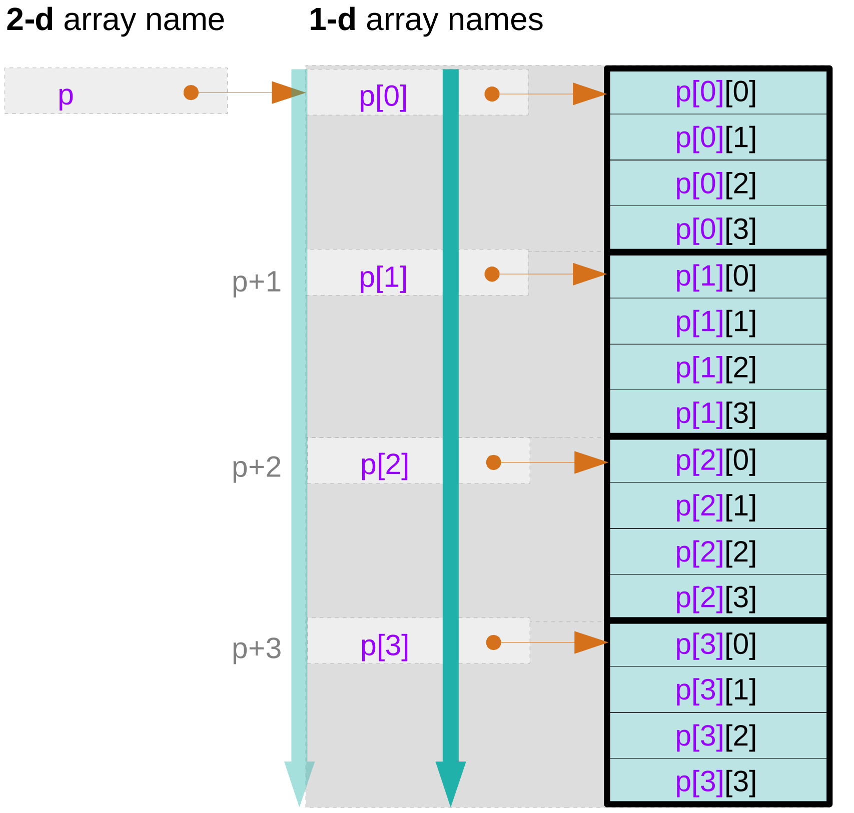
contiguous index : m

Contiguity constraints for p

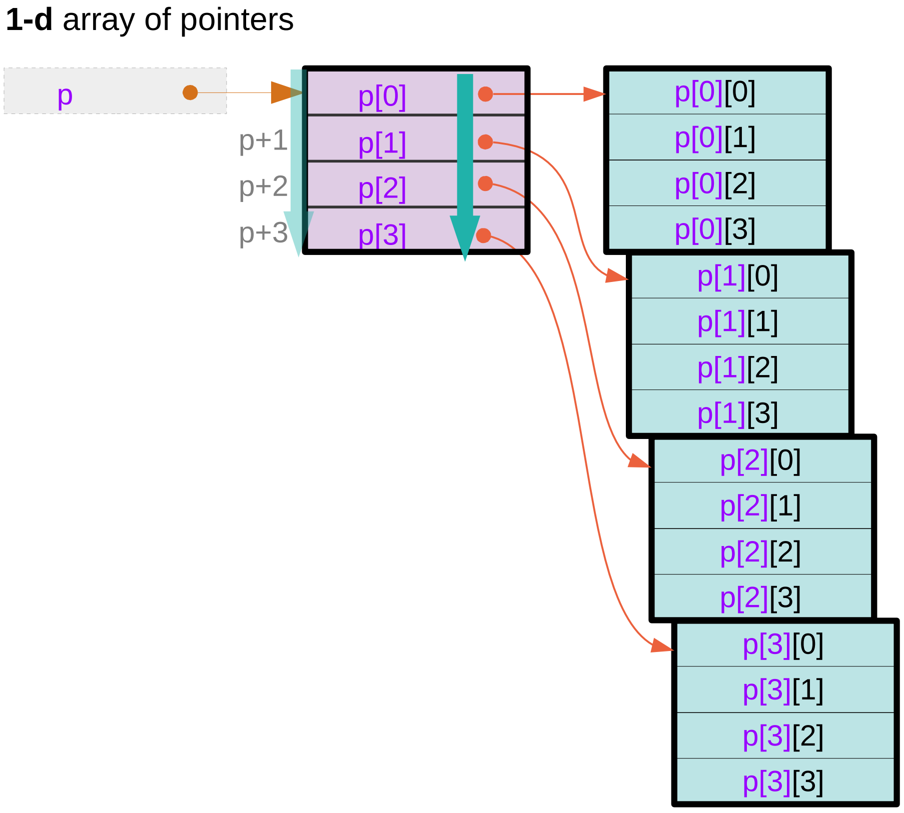
*(p+m) ⟺ p[m]

for a given p

contiguous index : **m**



contiguous p[m] ➡ contiguous p[m][n]



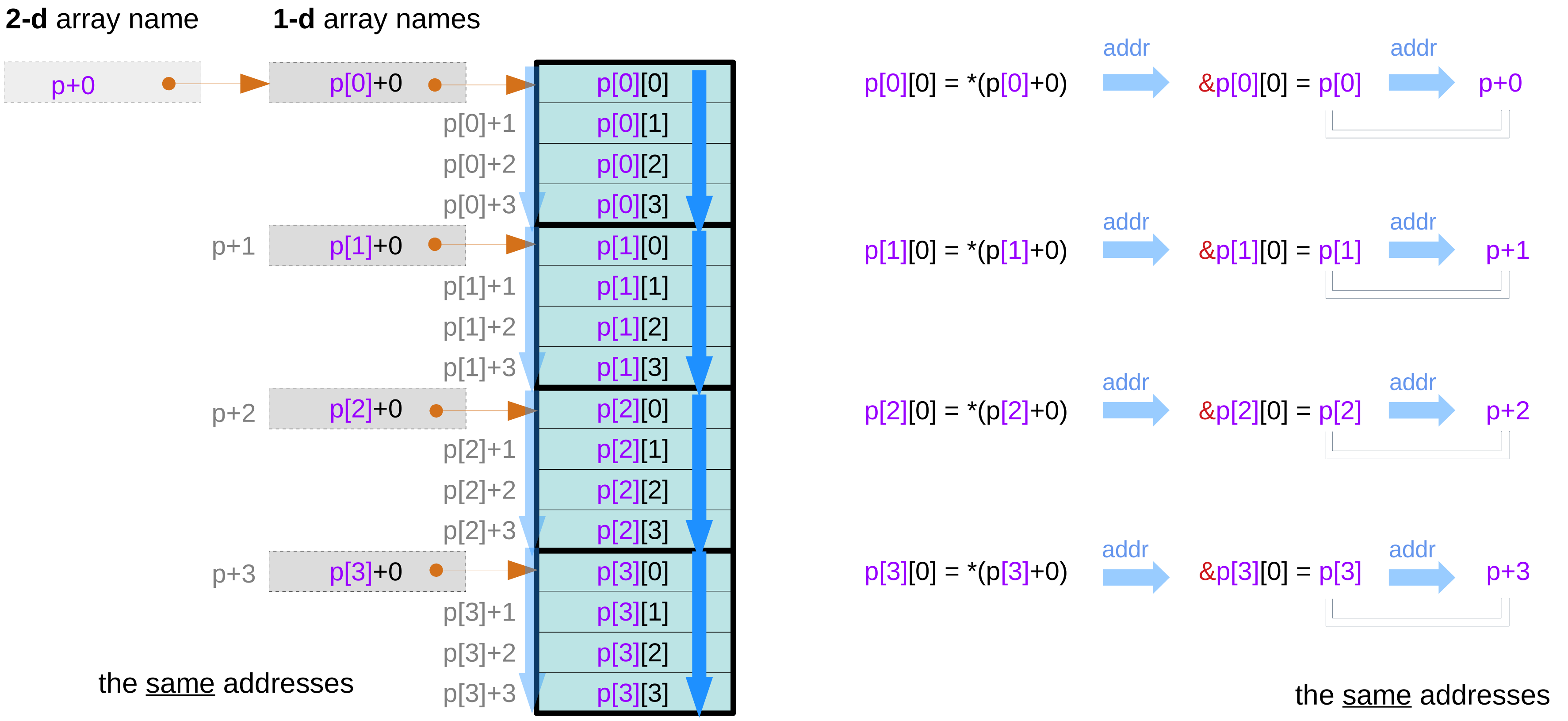
contiguous p[m] ➡ contiguous p[m][n]

Not necessarily

Contiguity constraints for p[m] – using array pointers

*(p[m]+n) ⟷ p[m][n]

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



2-d array name	**1-d** array names	
p+0	p[0]+0	p[0][0]
	p[0]+1	p[0][1]
	p[0]+2	p[0][2]
	p[0]+3	p[0][3]
p+1	p[1]+0	p[1][0]
	p[1]+1	p[1][1]
	p[1]+2	p[1][2]
	p[1]+3	p[1][3]
p+2	p[2]+0	p[2][0]
	p[2]+1	p[2][1]
	p[2]+2	p[2][2]
	p[2]+3	p[2][3]
p+3	p[3]+0	p[3][0]
	p[3]+1	p[3][1]
	p[3]+2	p[3][2]
	p[3]+3	p[3][3]

the same addresses

p[0][0] = *(p[0]+0) $\xrightarrow{\text{addr}}$ &p[0][0] = p[0] $\xrightarrow{\text{addr}}$ p+0

p[1][0] = *(p[1]+0) $\xrightarrow{\text{addr}}$ &p[1][0] = p[1] $\xrightarrow{\text{addr}}$ p+1

p[2][0] = *(p[2]+0) $\xrightarrow{\text{addr}}$ &p[2][0] = p[2] $\xrightarrow{\text{addr}}$ p+2

p[3][0] = *(p[3]+0) $\xrightarrow{\text{addr}}$ &p[3][0] = p[3] $\xrightarrow{\text{addr}}$ p+3

the same addresses

contiguous p[m] → contiguous p[m][n]

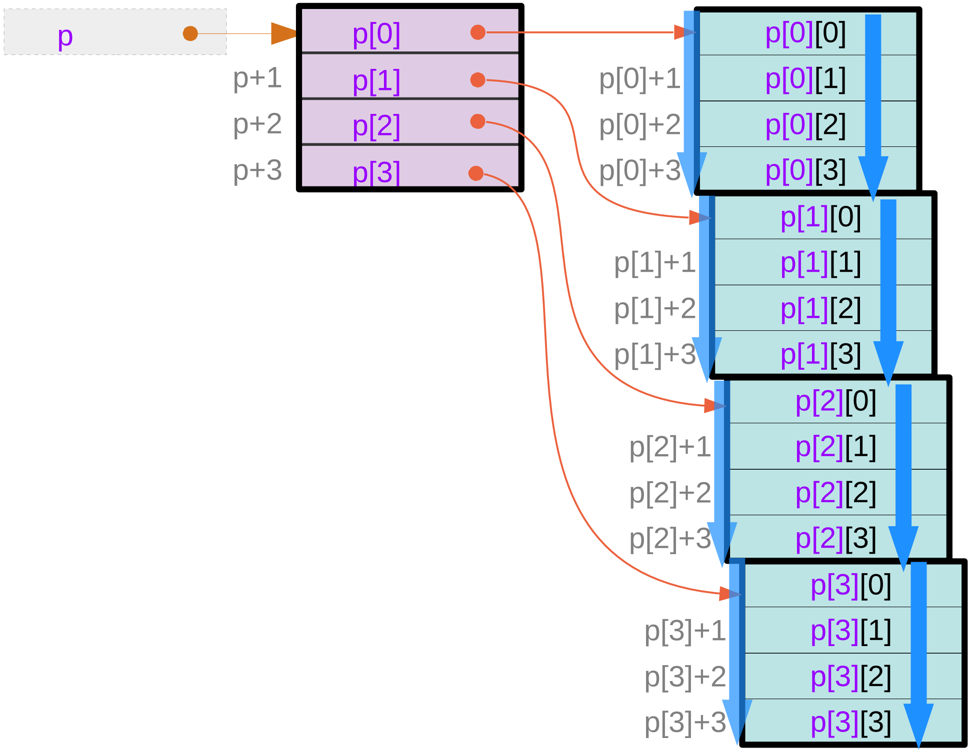
virtual array pointer ⟷ no real memory locations

Contiguity constraints for p[m] – using pointer arrays

*(p[m]+n) ⟷ p[m][n]

for a given p[m] contiguous index : n

1-d array of pointers



data
p[0][0] = *(p[0]+0) $\xrightarrow{\text{addr}}$ &p[0][0] = p[0] $\xrightarrow{\text{addr}}$ p+0

data
p[1][0] = *(p[1]+0) $\xrightarrow{\text{addr}}$ &p[1][0] = p[1] $\xrightarrow{\text{addr}}$ p+1

data
p[2][0] = *(p[2]+0) $\xrightarrow{\text{addr}}$ &p[2][0] = p[2] $\xrightarrow{\text{addr}}$ p+2

data
p[3][0] = *(p[3]+0) $\xrightarrow{\text{addr}}$ &p[3][0] = p[3] $\xrightarrow{\text{addr}}$ p+3

the different addresses

contiguous p[m] → contiguous p[m][n]

Not necessarily

Contiguity constraints for 2-d arrays

int a[M][N] ;

*(a+m) ⟺ a[m]

a[0], a[1], … , a[M-1]
are contiguous

*(a[m]+n) ⟺ a[m][n]

a[m][0], a[m][1], … , a[m][N-1]
are contiguous

int (*b)[N] ;

*(b+m) ⟺ b[m]

b[0], b[1], … , b[M-1]
are contiguous

*(b[m]+n) ⟺ b[m][n]

b[m][0], b[m][1], … , b[m][N-1]
are contiguous

int * c[M] ;

*(c+m) ⟺ c[m]

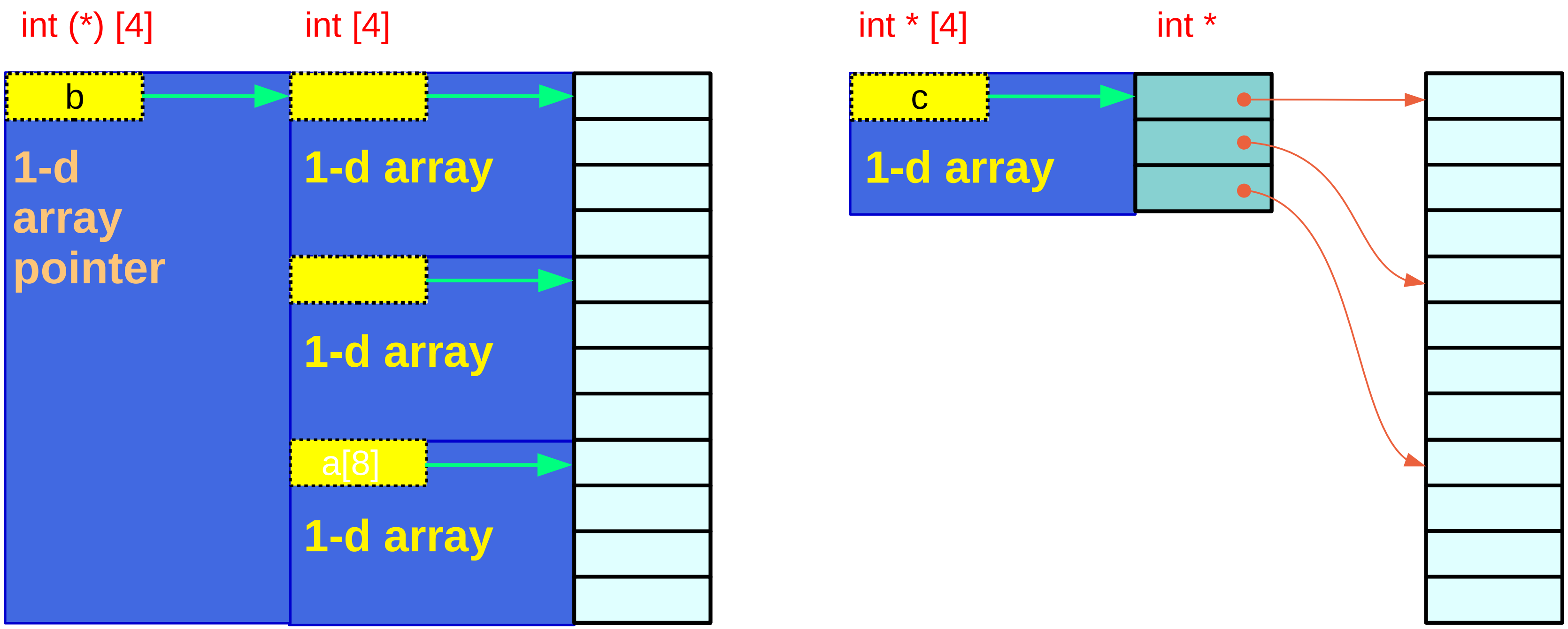
c[0], c[1], … , c[M-1]
are contiguous

*(c[m]+n) ⟺ c[m][n]

c[m][0], c[m][1], … , c[m][N-1]
are contiguous

a set of assignments of pointers
are necessary for this contiguity

Pointer Arrays vs Array Pointers



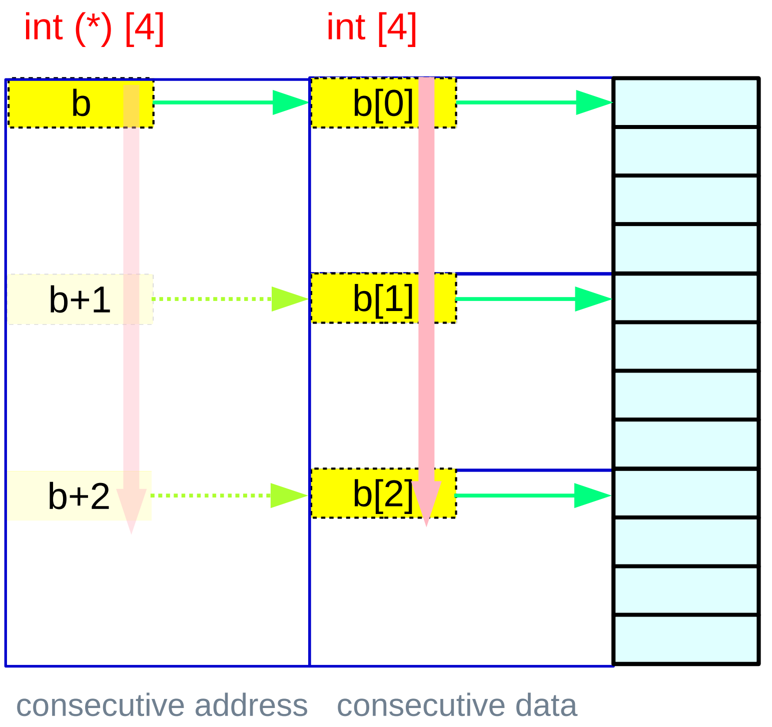
int (*b)[N] ;

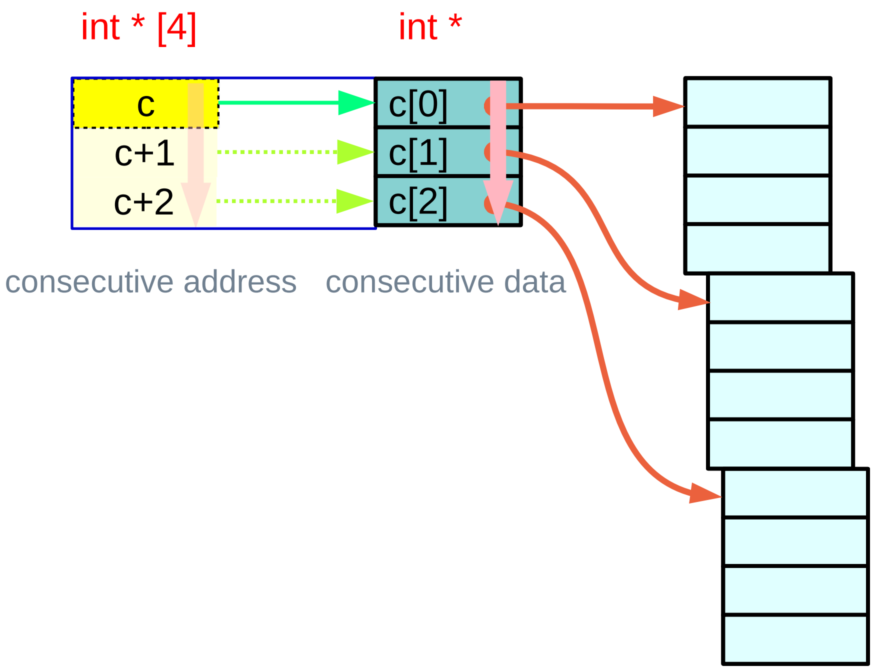
int * c[M] ; with proper assignments

*(b+m) ⟺ b[m]
*(b[m]+n) ⟺ b[m][n]

*(c+m) ⟺ c[m] or
*(c[m]+n) ⟺ c[m][n]

Pointer Arrays vs Array Pointers





int (*b)[N] ;

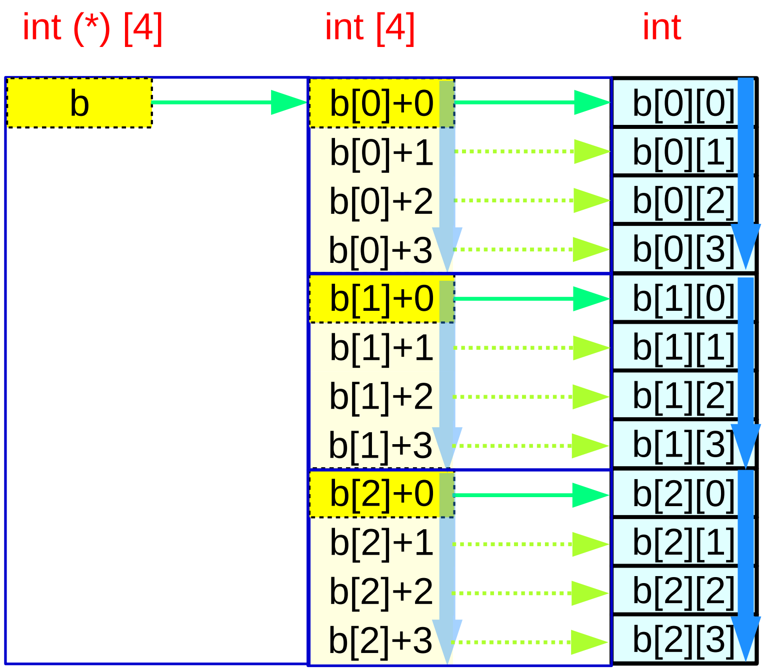
int * c[M] ;

with proper assignments

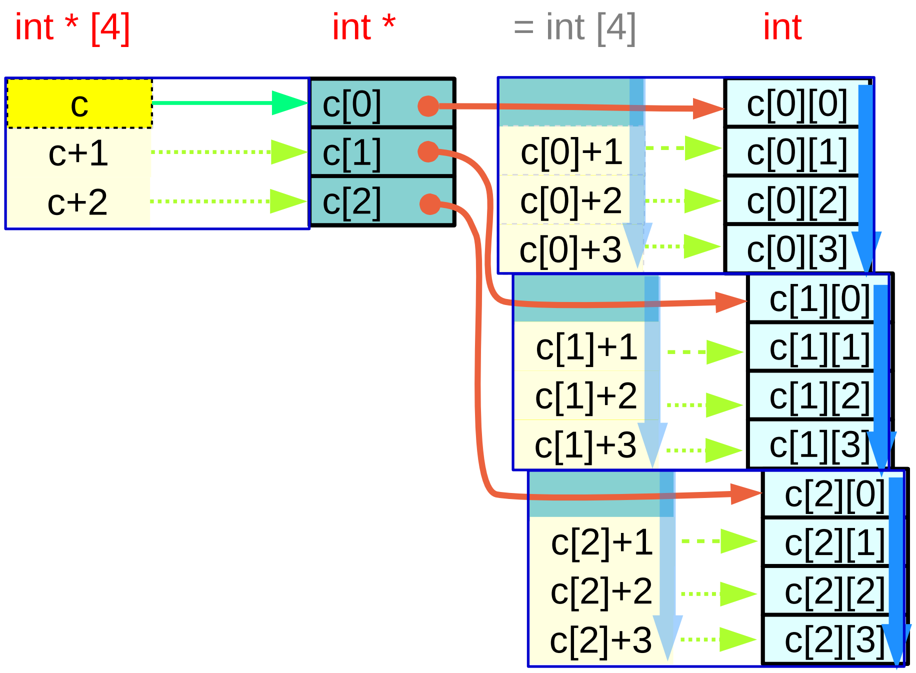
*(b+m)	⟺	b[m]
*(b[m]+n)	⟺	b[m][n]

*(c+m)	⟺	c[m] or
*(c[m]+n)	⟺	c[m][n]

Pointer Arrays vs Array Pointers



consecutive address consecutive data



consecutive address consecutive data

int (*b)[N] ;

int * c[M] ; with proper assignments

*(b+m) ⟺ b[m]
*(b[m]+n) ⟺ b[m][n]

*(c+m) ⟺ c[m]
*(c[m]+n) ⟺ c[m][n]

Three contiguity constraints for 3-d arrays

Pointer Array Approach (array of pointers)

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

contiguous **int**	int
contiguous pointers to **int**	int *
contiguous double pointers to **int**	int **

the contiguity constraints are satisfied by allocating arrays of pointers

Array Pointer Approach (pointer to arrays)

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

contiguous **0-d** arrays	int	int
contiguous **1-d** arrays	int [4]	int *
contiguous **2-d** arrays	int [3][4]	int (*) [4]

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

Contiguous array pointers c[i][j][k] ≡ *(c[i][j] +k)

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

⋮ ⋮

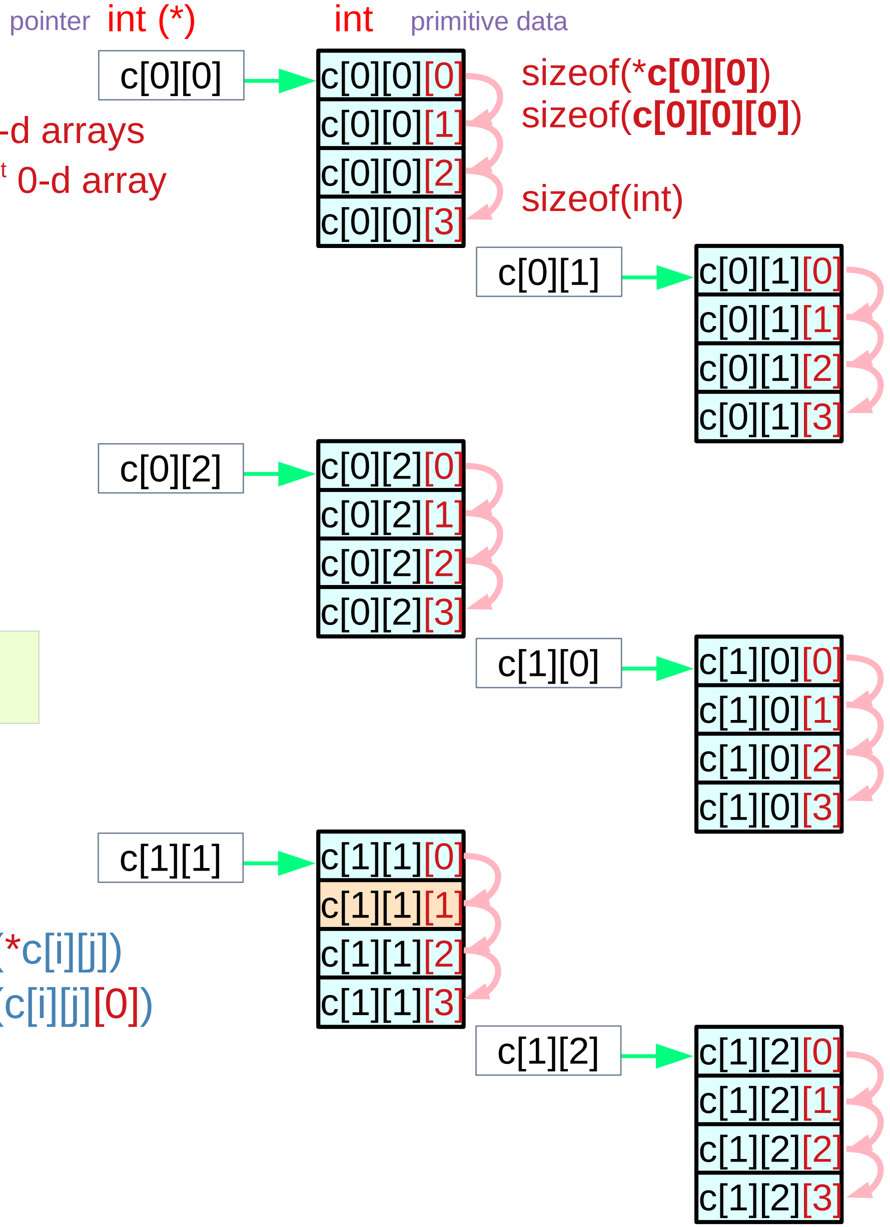
contiguous 1-d
array elements

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

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

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

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



pointer int (*)

int primitive data

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

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

sizeof(int)

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

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

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

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

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

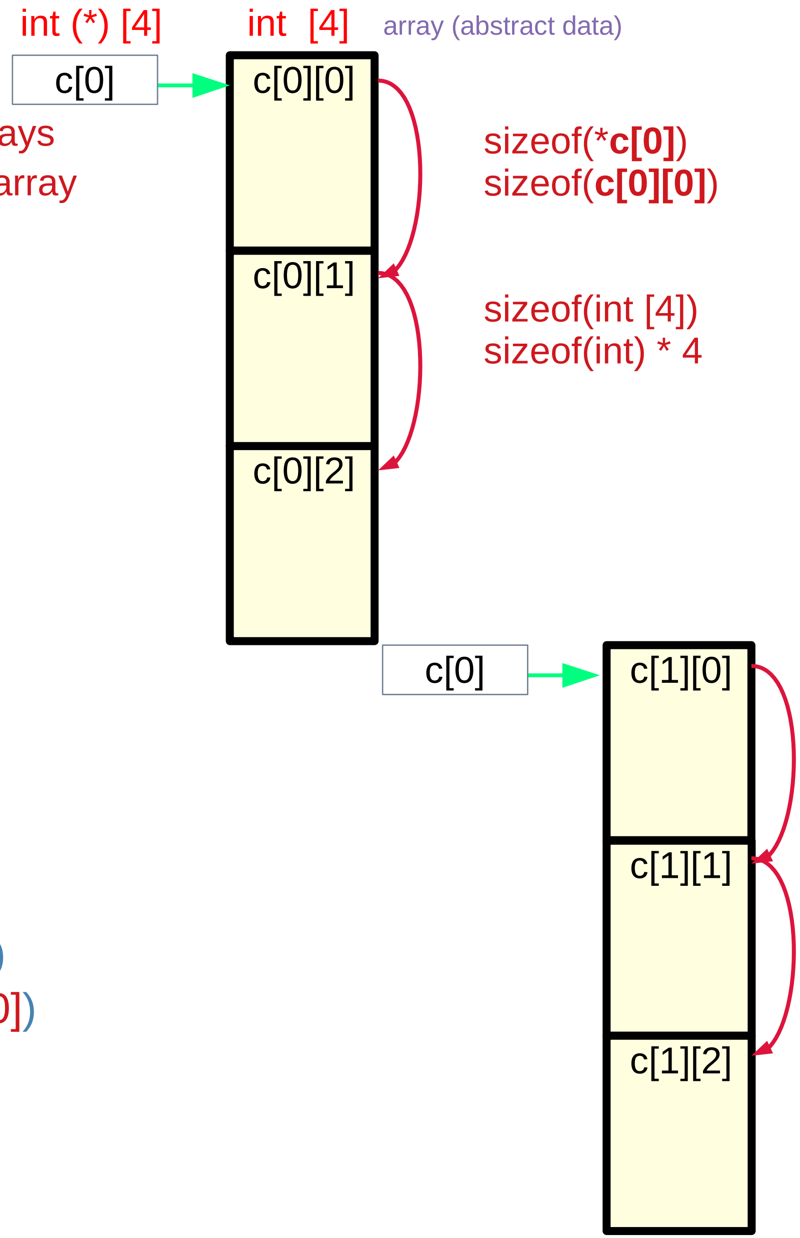
Contiguous array pointers c[i][j] ≡ *(c[i] +j)

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

c[i]

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

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



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

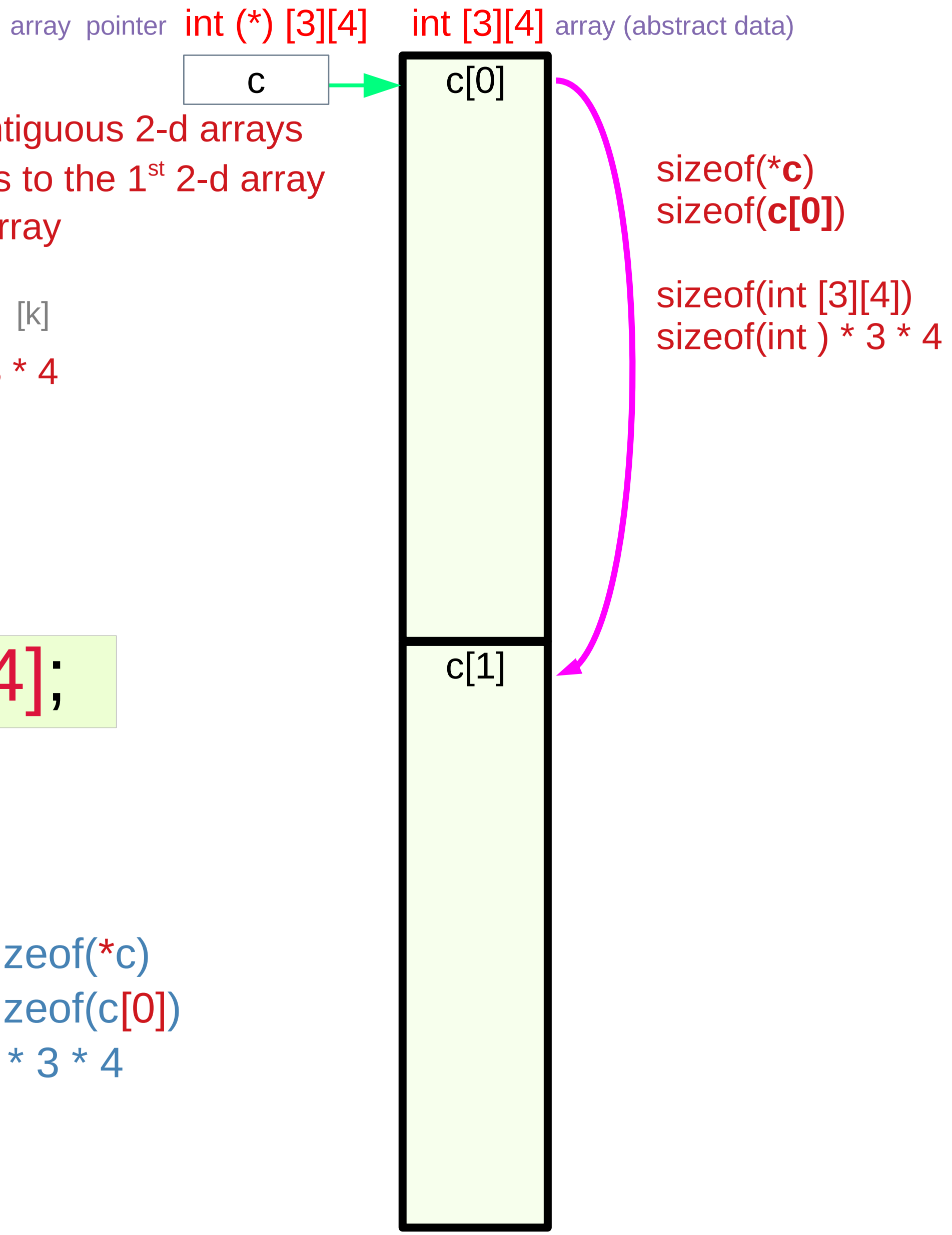
Address Value
c[i] + j
&c[i][0][0] + j * sizeof(*c[i])
&c[i][0][0] + j * sizeof(c[i][0])
&c[i][0][0] + j * 4 * 4

Contiguous array pointers c[i] ≡ *(c +i)

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

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

sizeof(**c**)
$\quad$ [i] [j] [k]
sizeof(**c[i][j][k]**) * 2 * 3 * 4
sizeof(**int**) * 2 * 3 * 4



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

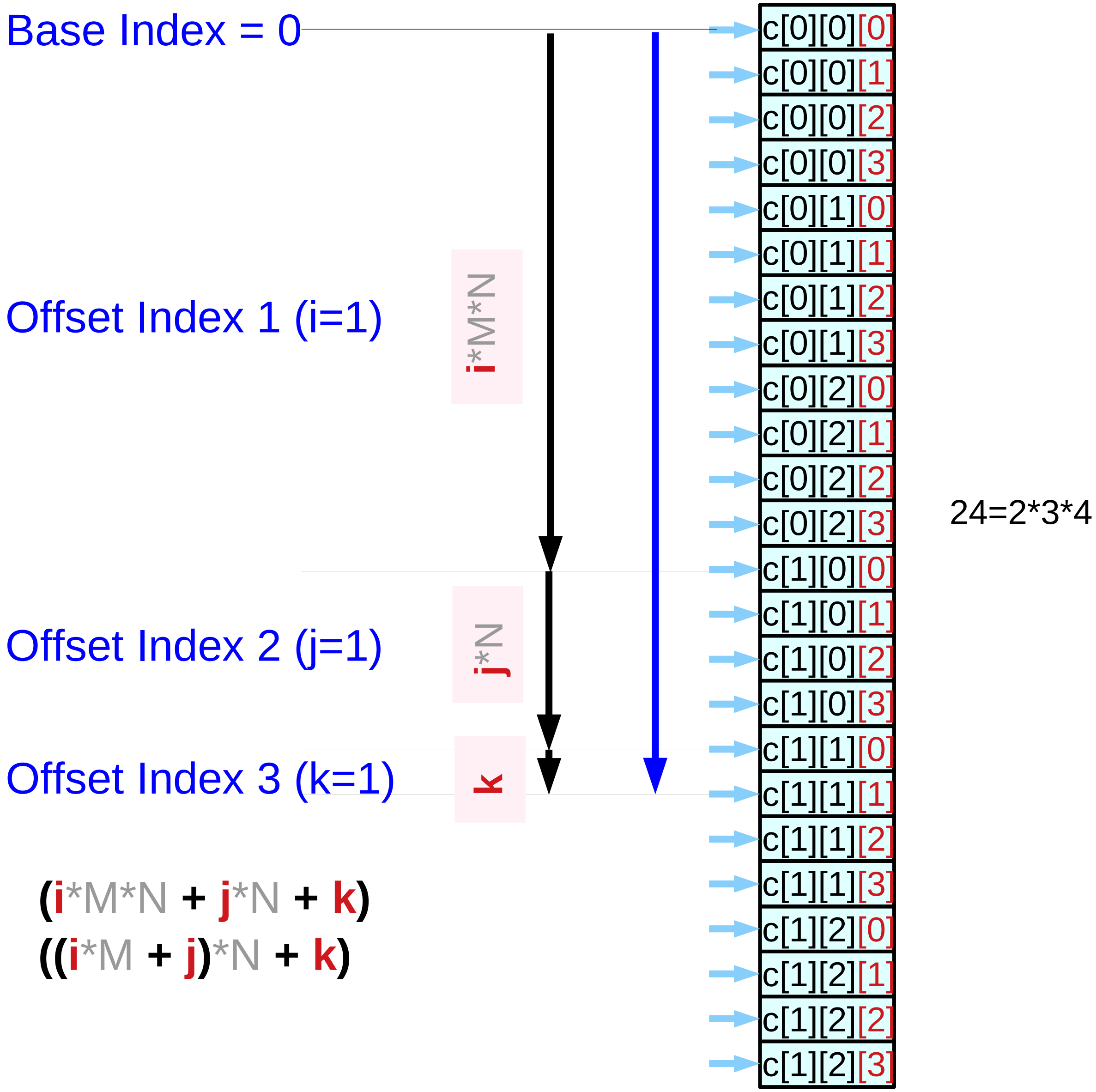
Address Value
c + i
&c[0][0][0] + i * sizeof(*c)
&c[0][0][0] + i * sizeof(c[0])
&c[0][0][0] + i * 4 * 3 * 4

Contiguous linear layout

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

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

L	M	N
i	j	k
i*M*N	j*N	k



Base Index = 0

Offset Index 1 (i=1)

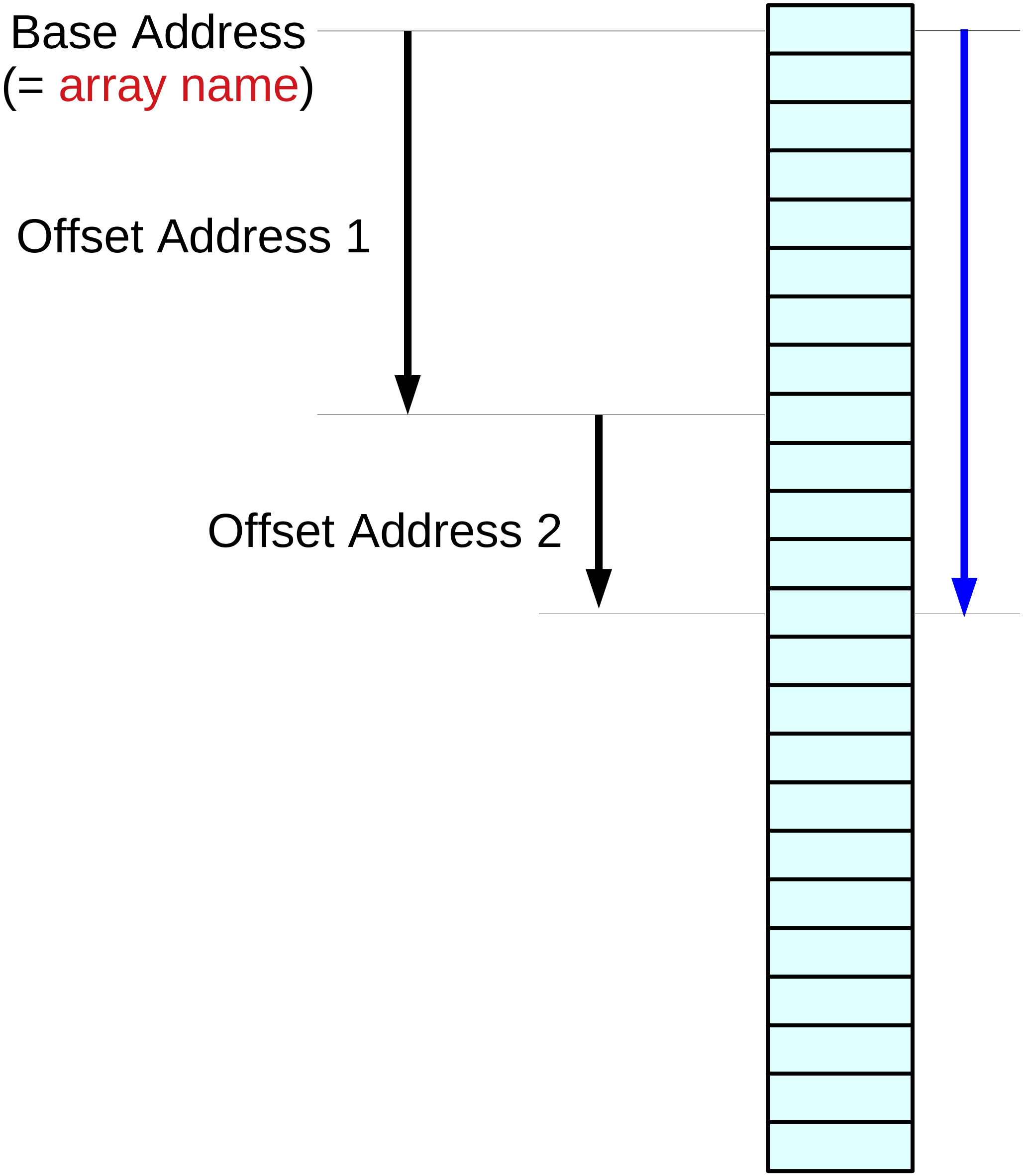
Offset Index 2 (j=1)

Offset Index 3 (k=1)

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

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

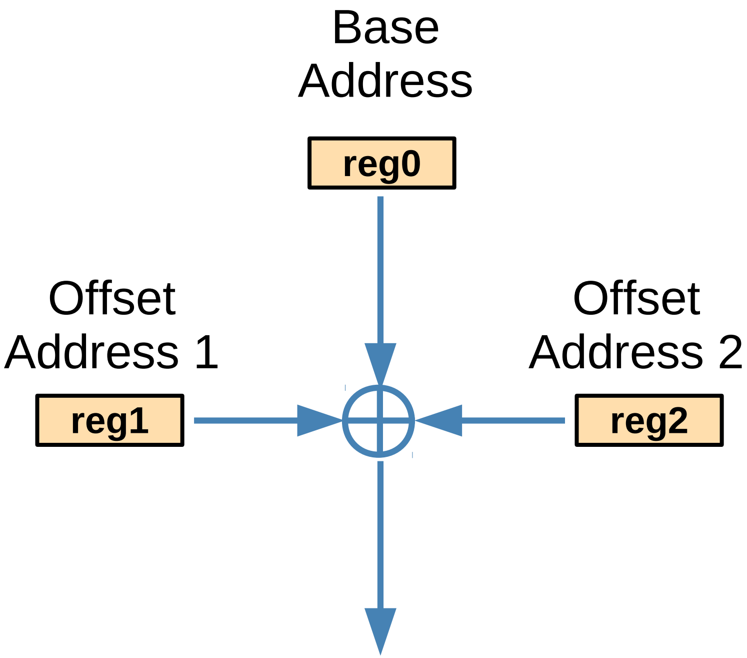
Base and Offset Addressing



Base Address
(= array name)

Offset Address 1

Offset Address 2



Base
Address

reg0

Offset
Address 1

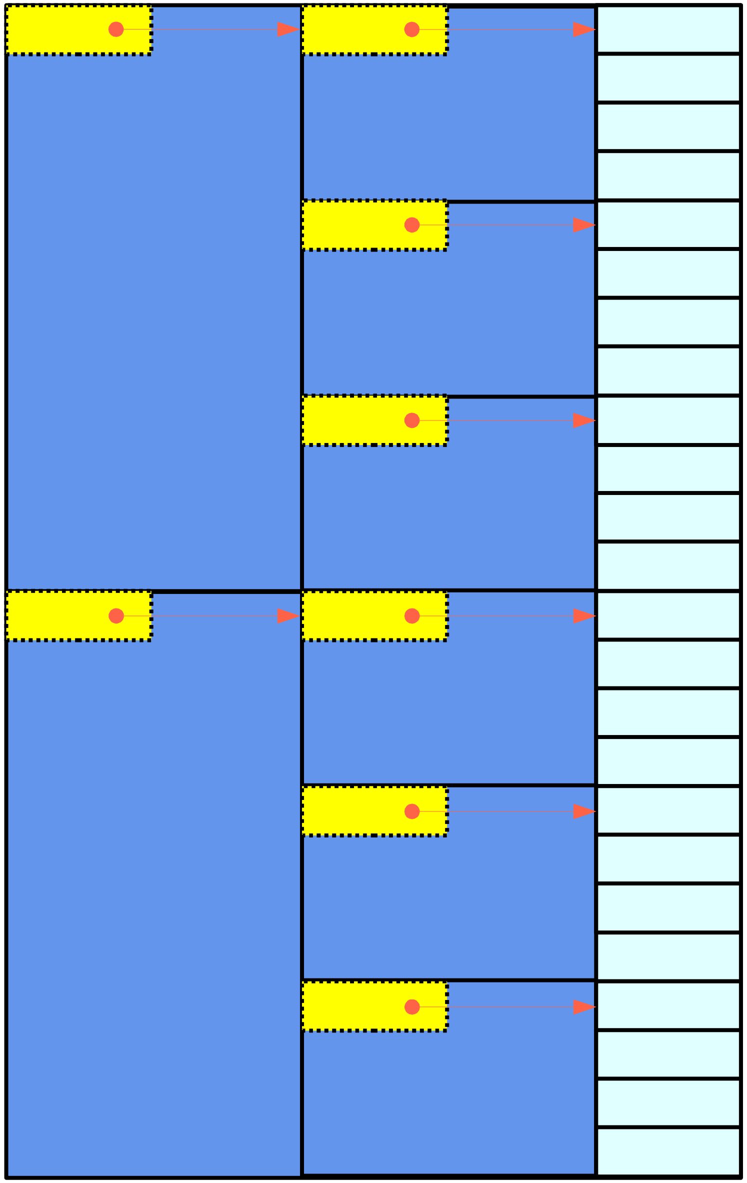
reg1

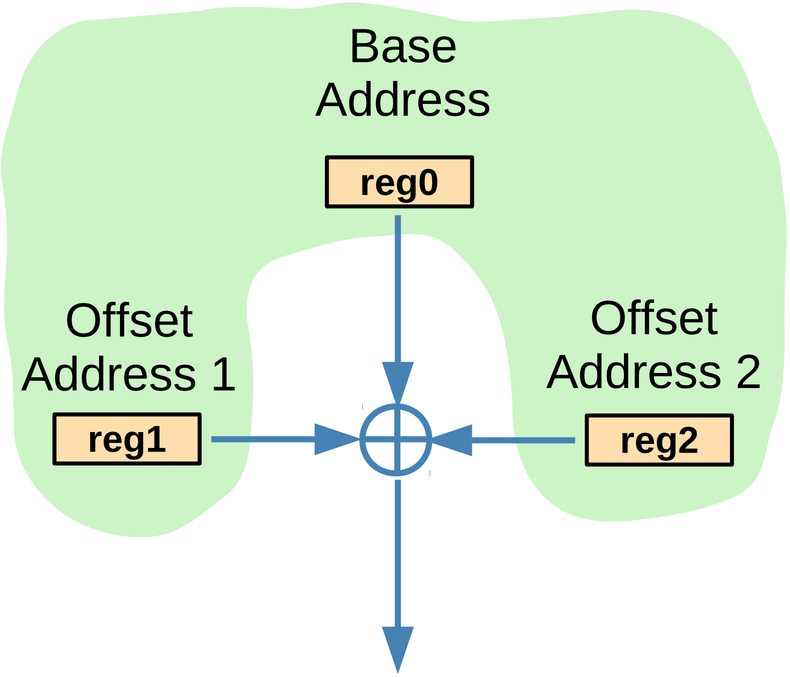
Offset
Address 2

reg2

compiler
assembly instruction
registers in the CPU

Array Pointer Approach





register based address **computations**
eliminate the pointer arrays – by a compiler

Array Pointer Approach
(pointer to arrays)

References

[1] Essential C, Nick Parlante

[2] Efficient C Programming, Mark A. Weiss

[3] C A Reference Manual, Samuel P. Harbison & Guy L. Steele Jr.

[4] C Language Express, I. K. Chun