Applications of Pointers (1A)

Young Won Lim
11/22/24

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

n-d access of a **1-d** array

- **2-d** array access of a **1-d** array
- **3-d** array access of a **1-d** array
- Accessing a **contiguous 1-d** array
- Accessing a **non-contiguous 1-d** arrays

- Accessing **static** allocated arrays
- Accessing **dynamically** allocated arrays

2-d array access of a **1-d** array

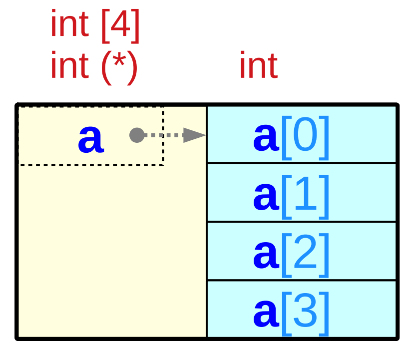
Array of Pointers

```
int      a [4] ;

int *    b [3] ;
```

int a [4]

there are 4 elements

the type of each element:
an integer

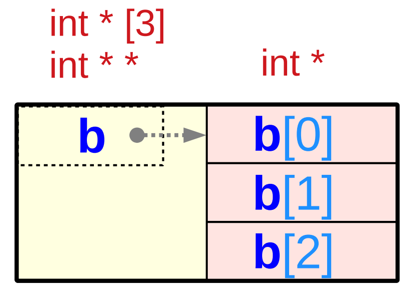


Integers

int * b [3]

there are 3 elements

the type of each element:
a pointer an integer



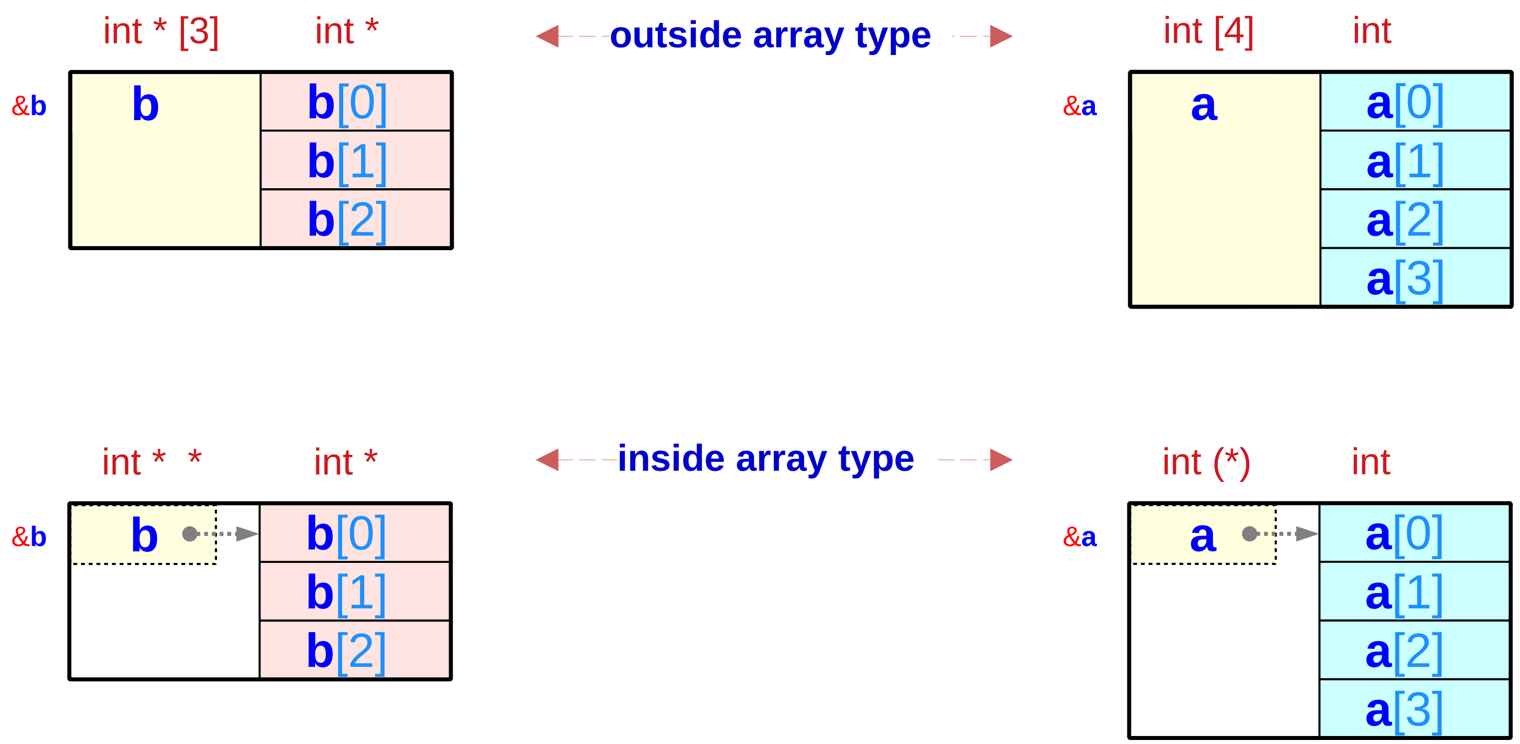
Integer pointers

Array of Pointers – a type view

int * b [3] ;

Pointer Array

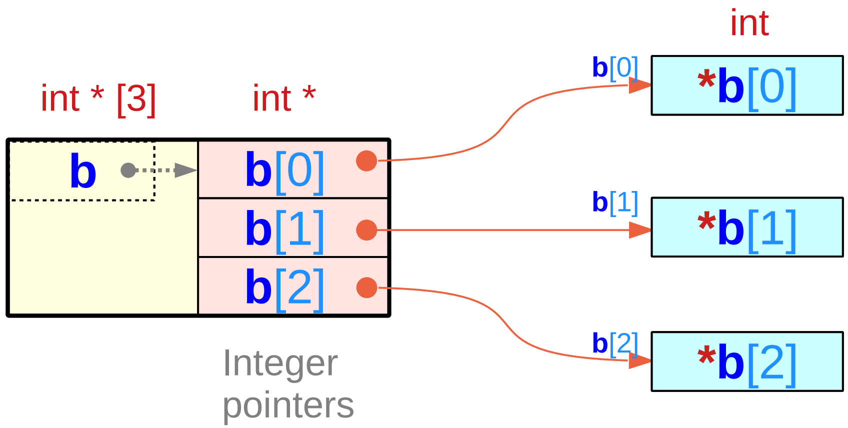
int a [4] ;

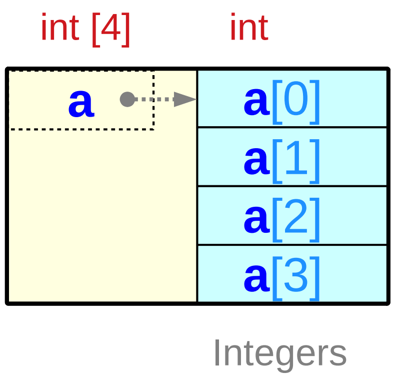


Array of Pointers – a variable view

`int * b [3] ;` **Pointer Array**

`int a [4] ;`



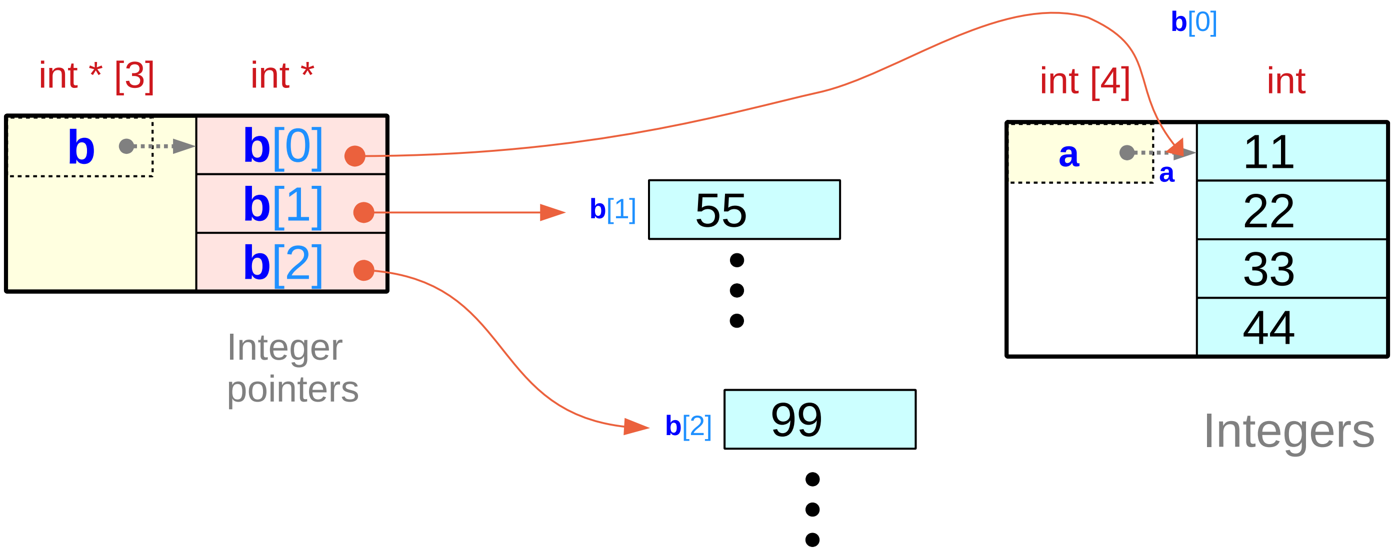


Assigning a **1-d** array name

int * **b** [3] ; **Pointer Array** int **a** [4] ;

assignment

b[0] = **a** (= &**a**[0])



Assigning a **1-d** array name – equivalence

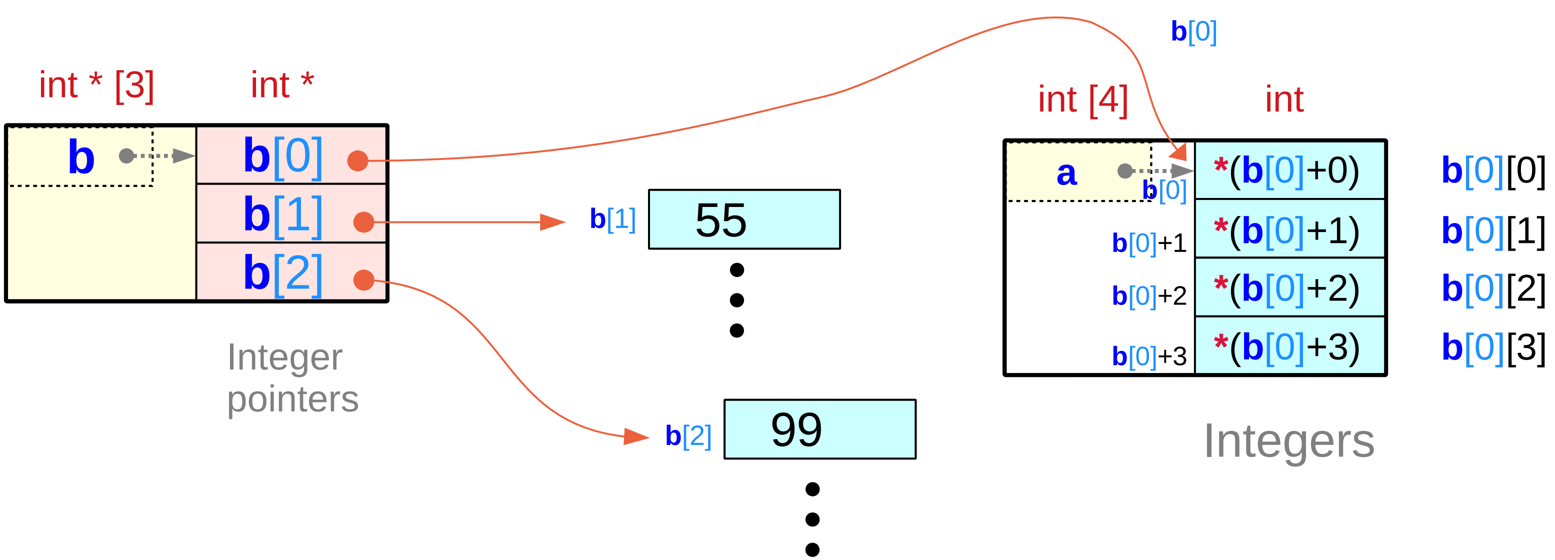
int * b [3] ;

Pointer Array

int a [4] ;

assignment

b[0] = a (= &a[0])



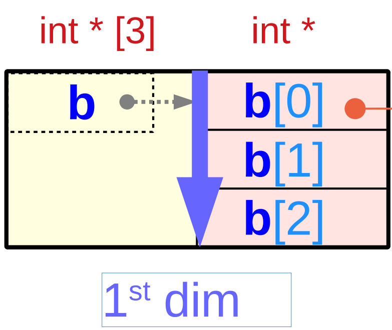
Array of Pointers – extended dimension

int *	b [3] ;

Pointer Array

int	a [4] ;

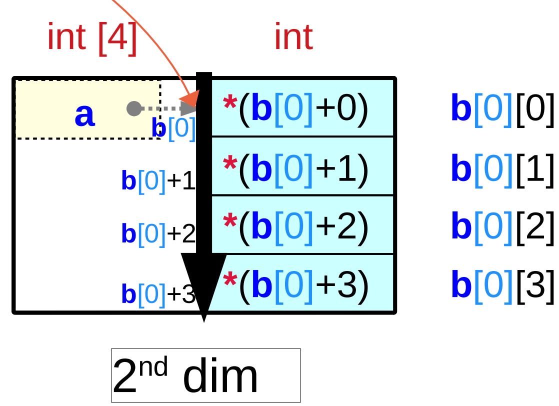
array name b



assignment

b[0] = a (= &a[0])

array name b[0]



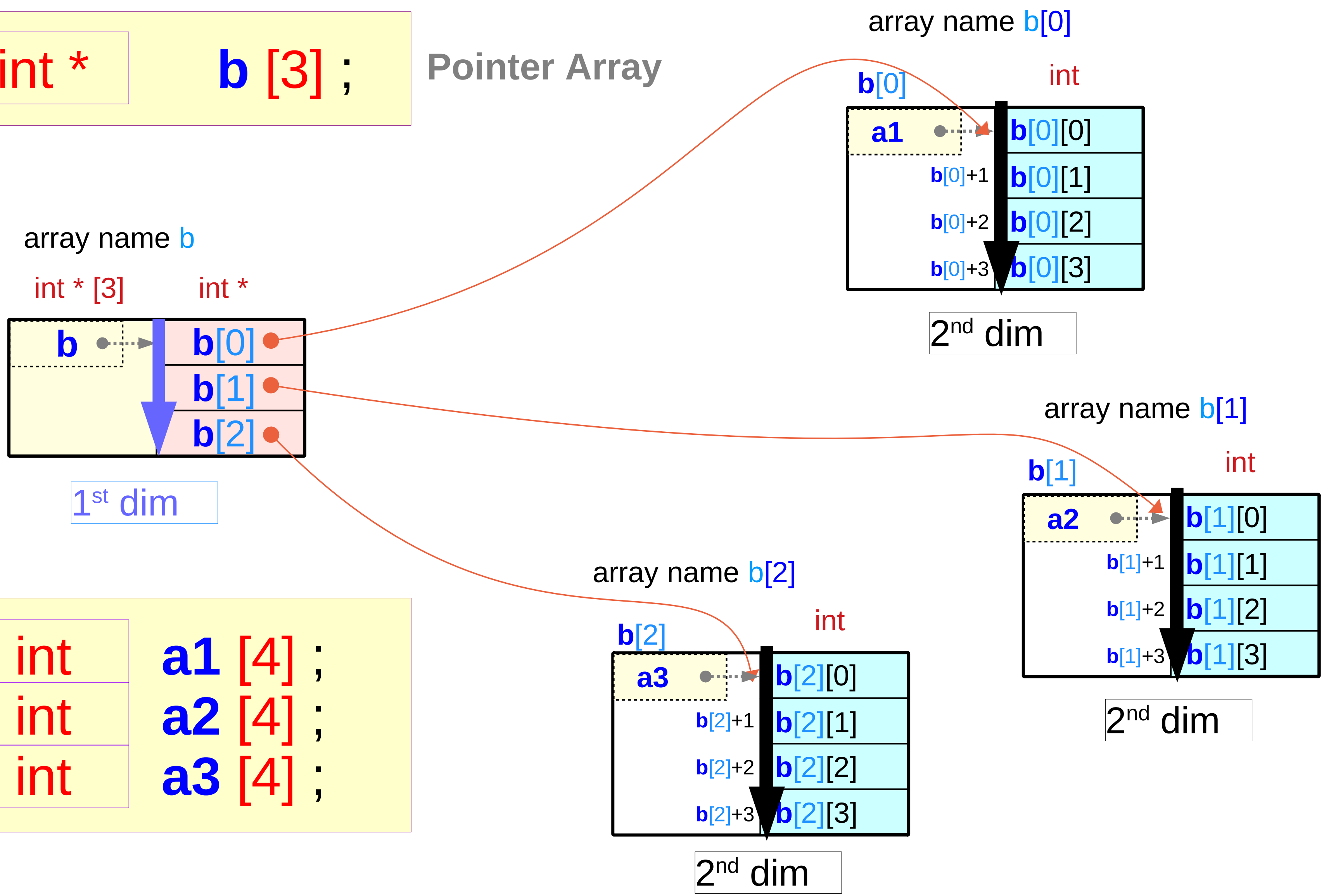
a[0] ≡ b[0][0] ≡ *(*(b+0)+0)
a[1] ≡ b[0][1] ≡ *(*(b+0)+1)
a[2] ≡ b[0][2] ≡ *(*(b+0)+2)
a[3] ≡ b[0][3] ≡ *(*(b+0)+3)

2-d access of **1-d** arrays

```
int *    b [3] ;
```

Pointer Array

```
int    a1 [4] ;
int    a2 [4] ;
int    a3 [4] ;
```

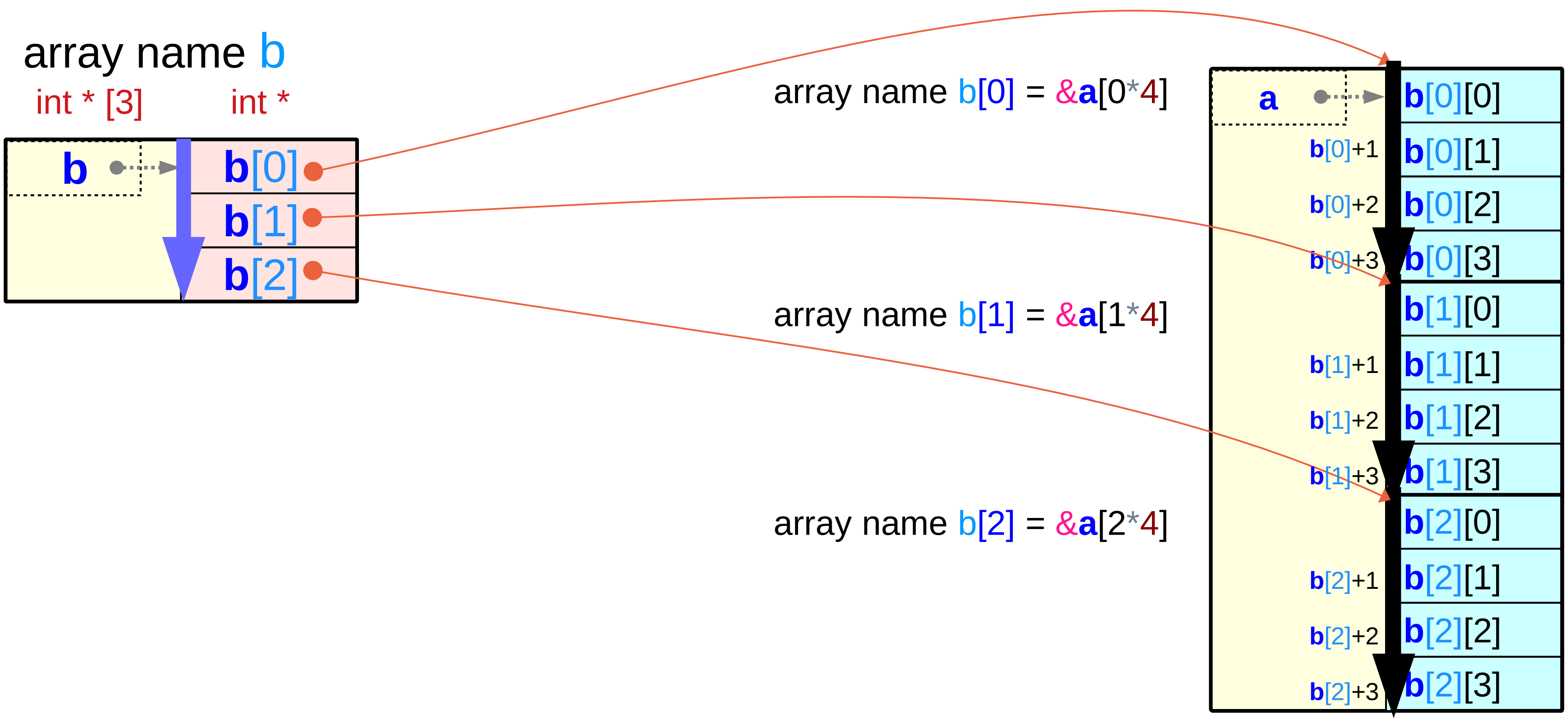


2-d access of a 1-d array

int * b [3] ;

Pointer Array

int * a [3*4] ;



2-d access of a **1-d** array – pointer array assignment

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

b[j] = &**a**[j*4] (= **a**+j*4)

b[j] + **k** = **a**+j*4 + **k**

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

b[j][**k**] ≡ **a**[j*4 +**k**]

j = [0:5] k = [0:3]

j*4+k = [0:23]

constraint : contiguous b[i][j] over j

2-d access of a **1-d** array

b[i][j] ≡ *(**b**[i] + **j**)

↕ ↕

a[i*4+**j**] ≡ *(**a**+i*4 + **j**)

1-d access of a **1-d** array

constraint : contiguous a[i*4+j] over j

3-d array access of a **1-d** array

3-d access of a 1-d array (1)

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

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

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

c[i] = &b[i*3] (= b+i*3)

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

b[j] = &a[j*4] (= a+j*4)

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

c[i][j] = b[i*3 +j]

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

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

b[j][k] ≡ a[j*4 +k]

j = [0:5]   k = [0:3]
j*4+k = [0:23]

c[i][j][k] ≡ a[(i*3+j)*4+k]

i = [0:1]   j = [0:2]   k = [0:3]
(i*3+j)*4+k = [0:23]

3-d access of a 1-d array (2)

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

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

constraint : contiguous a[i], b[i], c[i]

Assignments

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

Initializing pointer arrays **b** and **c**

3-d access of a **1-d** array

c[i][j][k] ≡ *(c[i][j] +k)

b[i*3+j][k] ≡ *(b[i*3+j] +k)

a[(i*3+j)*4 +k] ≡ *(a+(i*3+j)*4 +k)

1-d access of a **1-d** array

3-d access of a 1-d array (3)

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

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

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

c[i] = &b[i*3] = b+i*3

b[j] = &a[j*4] = a+j*4

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

Equivalence relations in pointer array assignments

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

substitute c[i] in *(c[i]+j)

substitute b[m] in *(b[m]+k)

m = i*3+j

	substitute c[i]
c[i][j]	= *(c[i]+j)
	= *(b+i*3+j) = b[i*3+j]
	substitute b[m]
b[m][k]	= *(b[m]+k)
	= *(a+m*4+k) = a[m*4+k]
c[i][j][k]	= b[i*3+j][k] = a[(i*3+j)*4+k]

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

substitute c[i] in (c[i])[j]

substitute b[m] in (b[m])[k]

m = i*3+j

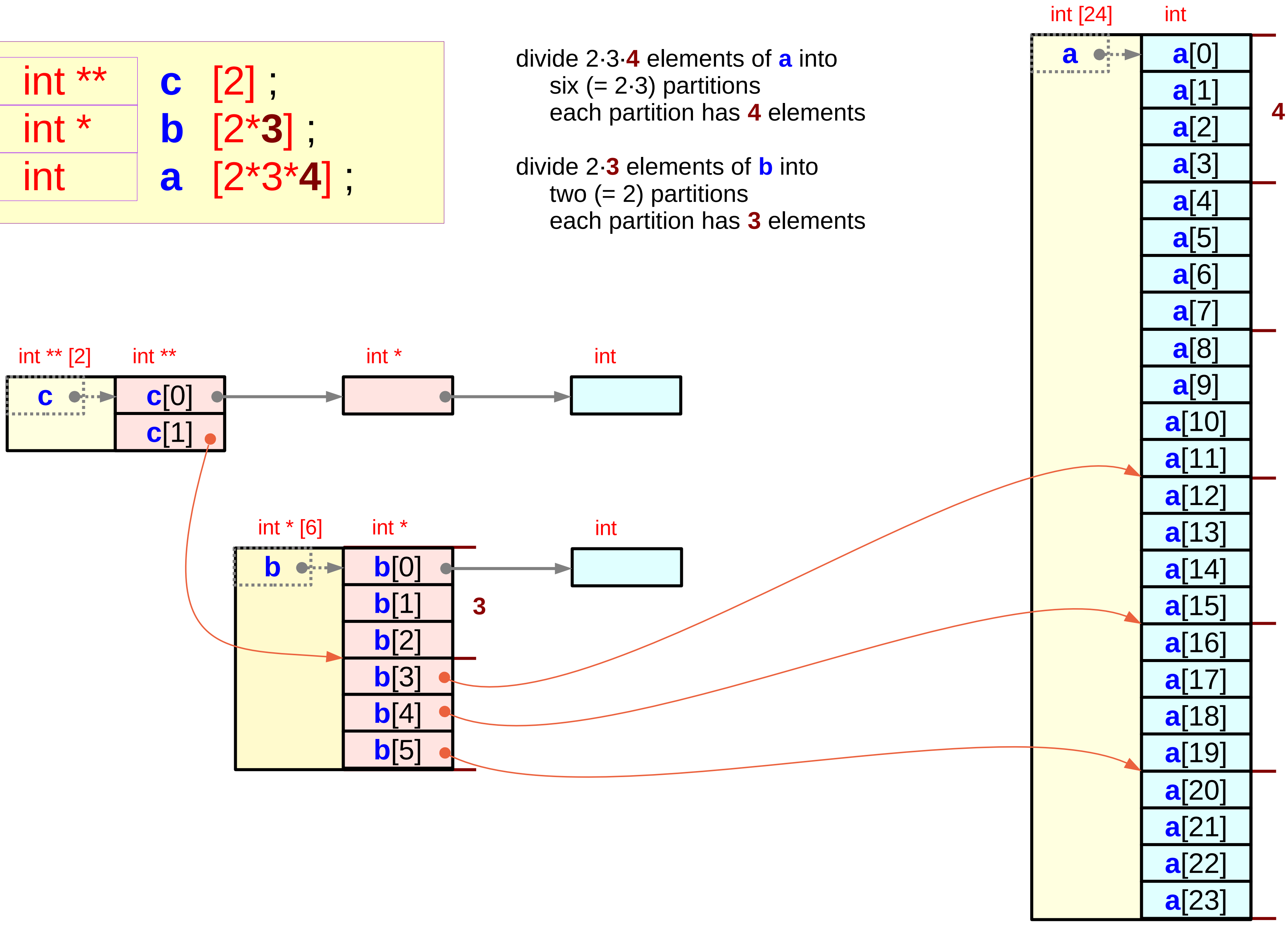
substitute c[i]	
c[i][j]	= (b+i*3)[j]
	= *(b+i*3+j) = b[i*3+j]
substitute b[m]	
b[m][k]	= (a+m*4)[k]
	= *(a+m*4+k) = a[m*4+k]
c[i][j][k]	= b[i*3+j][k] = a[(i*3+j)*4+k]

Integer array **a** and pointer arrays **b**, **c**

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

divide 2·3·**4** elements of **a** into
six (= 2·3) partitions
each partition has **4** elements

divide 2·**3** elements of **b** into
two (= 2) partitions
each partition has **3** elements



Pointer array initializations

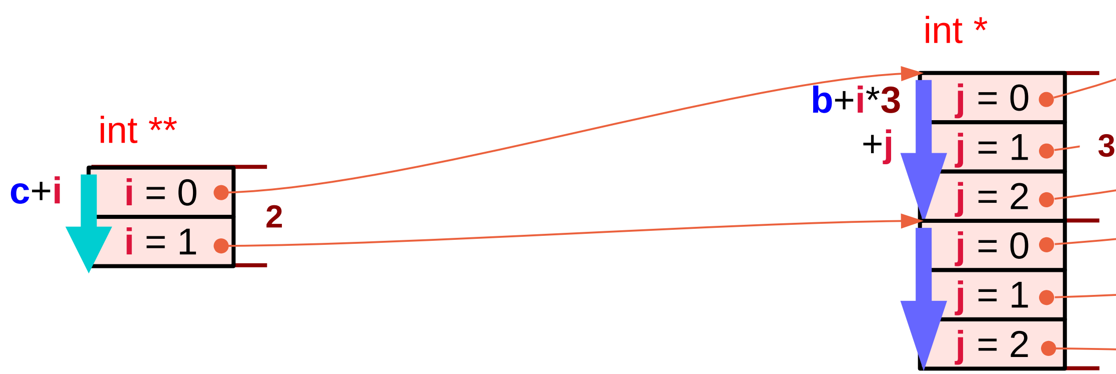
```
int **   c [2] ;
int *    b [2*3] ;
int      a [2*3*4] ;
```

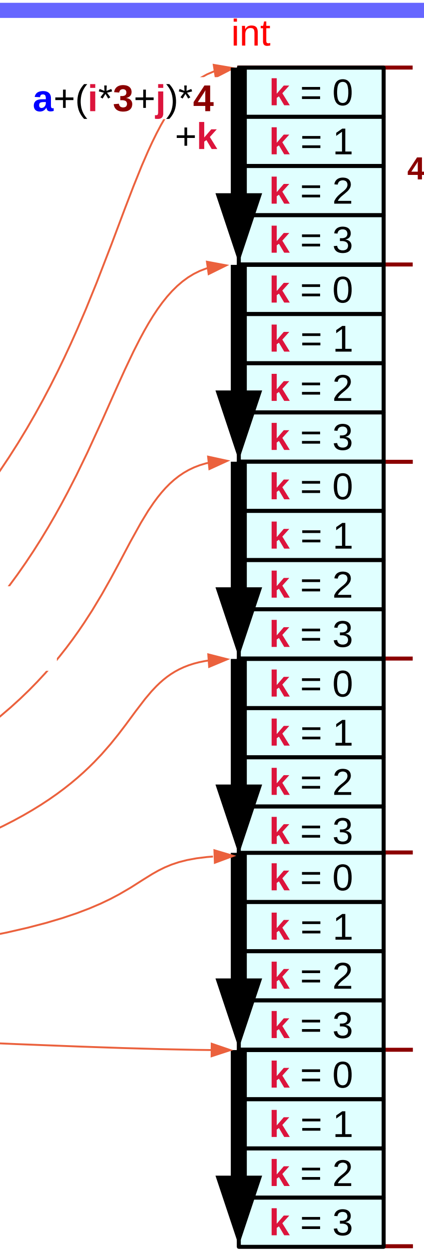
c[i] = &b[i*3] (= b+i*3)

b[j] = &a[j*4] (= a+j*4)

each element of c handles 3 elements of b
→ 3-element partitions in b

each element of b handles 4 elements of a
→ 4-element partitions in a





skipping **i** elements from **c**
= skipping **i***3 elements from **b**
= skipping **i***3*4 <u>leaf</u> elements from **a**

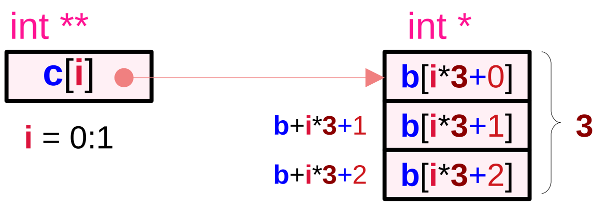
skipping **j** elements from **b**
= skipping **j***4 <u>leaf</u> elements from **a**

Partitioning arrays **a** and **b**

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

Assigning pointer array

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

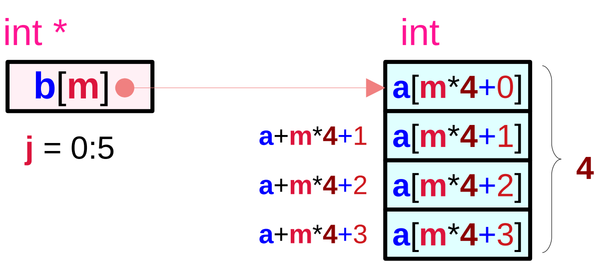


divide 2·3 elements of b
into two 3-element partitions
partition size = 3

c[0] = &b[0*3]; (= b + 0*3)
c[1] = &b[1*3]; (= b + 1*3)

c[i] contiguous c over i = 0:1

b[i*3+j] contiguous b over j = 0:2



divide 2·3·4 elements of a
into six 4-element partitions
partition size = 4

b[0] = &a[0*4]; (= a + 0*4)
b[1] = &a[1*4]; (= a + 1*4)
b[2] = &a[2*4]; (= a + 2*4)
b[3] = &a[3*4]; (= a + 3*4)
b[4] = &a[4*4]; (= a + 4*4)
b[5] = &a[5*4]; (= a + 5*4)

b[m] contiguous b over m = 0:5

a[m*4+k] contiguous a over k = 0:3

Skipping leaf elements

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

b[j] = &a[j*4] (= a+j*4)

skipping 1 element in **b**
= skipping **4** leaf elements in **a**

c[i] = &b[i*3] (= b+i*3)

skipping 1 element in **c**
= skipping **3** elements in **b**
= skipping **3*4** leaf elements in **a**

c[i][j] ≡ b[i*3 +j]

skipping **i** elements from **c**
+ skipping **j** elements from **b**

= skipping **i*3+j** elements from **b**

c[i][j][k] ≡ a[(i*3 +j)*4+k]

skipping **i*3+j** elements from **b**
+ skipping **k** leaf elements from **a**

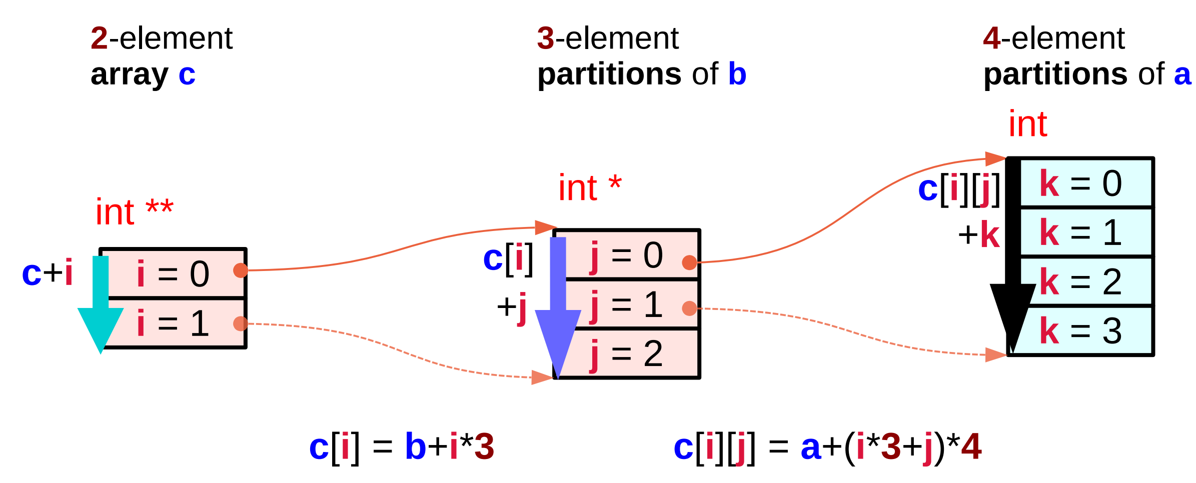
= skipping (**i*3+j**)*4+**k** leaf elements from **a**

Contiguous constraints for c[i][j][k]

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

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

c[i][j][k] ≡
a[(i*3 +j)*4+k]



contiguous 2-elements of array c   i = 0,1   ⟺   c[i] ≡ *(c+i)

contiguous 3-element partitions of array b   j = 0,1,2   ⟺   c[i][j] ≡ *(c[i]+j)

c[i][j] ≡ *(*(c+i)+j)

contiguous 4-element partitions of array a   k = 0, 1,2,3   ⟺   c[i][j][k] ≡ *(c[i][j]+k)

c[i][j][k] ≡ *(*(*(c+i)+j)+k)

Minimal constraints and implementations

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

    c[0] = &b[0*3];   b[0] = &a[0*4];
    c[1] = &b[1*3];   b[1] = &a[1*4];
                      b[2] = &a[2*4];
                      b[3] = &a[3*4];
                      b[4] = &a[4*4];
                      b[5] = &a[5*4];
```

contiguous
2-element
array c

contiguous
2·**3**-element
array b

contiguous
2·3·**4**-element
array a

minimal constraints

contiguous
2-element
array c

two contiguous
3-element
partitions of **b**

six contiguous
4-element
partitions of **a**

```
int c [2];      int b1 [3];      int a1 [4];
                int b2 [3];      int a2 [4];
                                 int a3 [4];
                                 int a4 [4];
                                 int a5 [4];
                                 int a6 [4];

    c[0] = &b1[0];    b1[0] = &a1[0];
    c[1] = &b2[0];    b1[1] = &a2[0];
                      b1[2] = &a3[0];
                      b2[0] = &a4[0];
                      b2[1] = &a5[0];
                      b2[2] = &a6[0];
```

two contiguous
3-element
arrays bi

six contiguous
4-element
arrays ai

Accessing a **contiguous 1-d** array

- **1-d** array access
- **2-d** array access
- **3-d** array access

Accessing an int array **a** as a **1-d** array

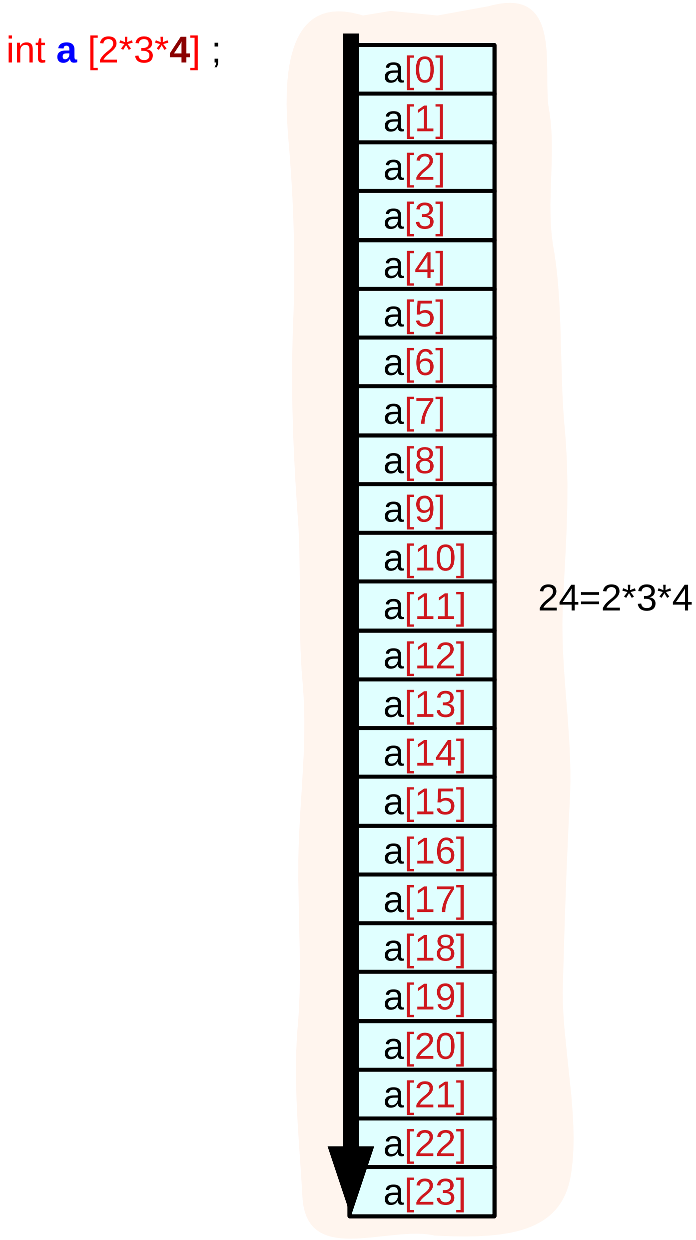
```
int    a [2*3*4] ;
```

↓

```
a [k]
```

k = 0,1, …,23

c[i][j][k]	≡ *(*(*(c+i)+j)+k)	int ** c[2] ;
b[j][k]	≡ *(*(b+j)+k)	int * b[2*3] ;
a[k]	**≡ *(a+k)**	**int a[2*3*4] ;**



Accessing an int array **a** as a **2-d** array using **b**

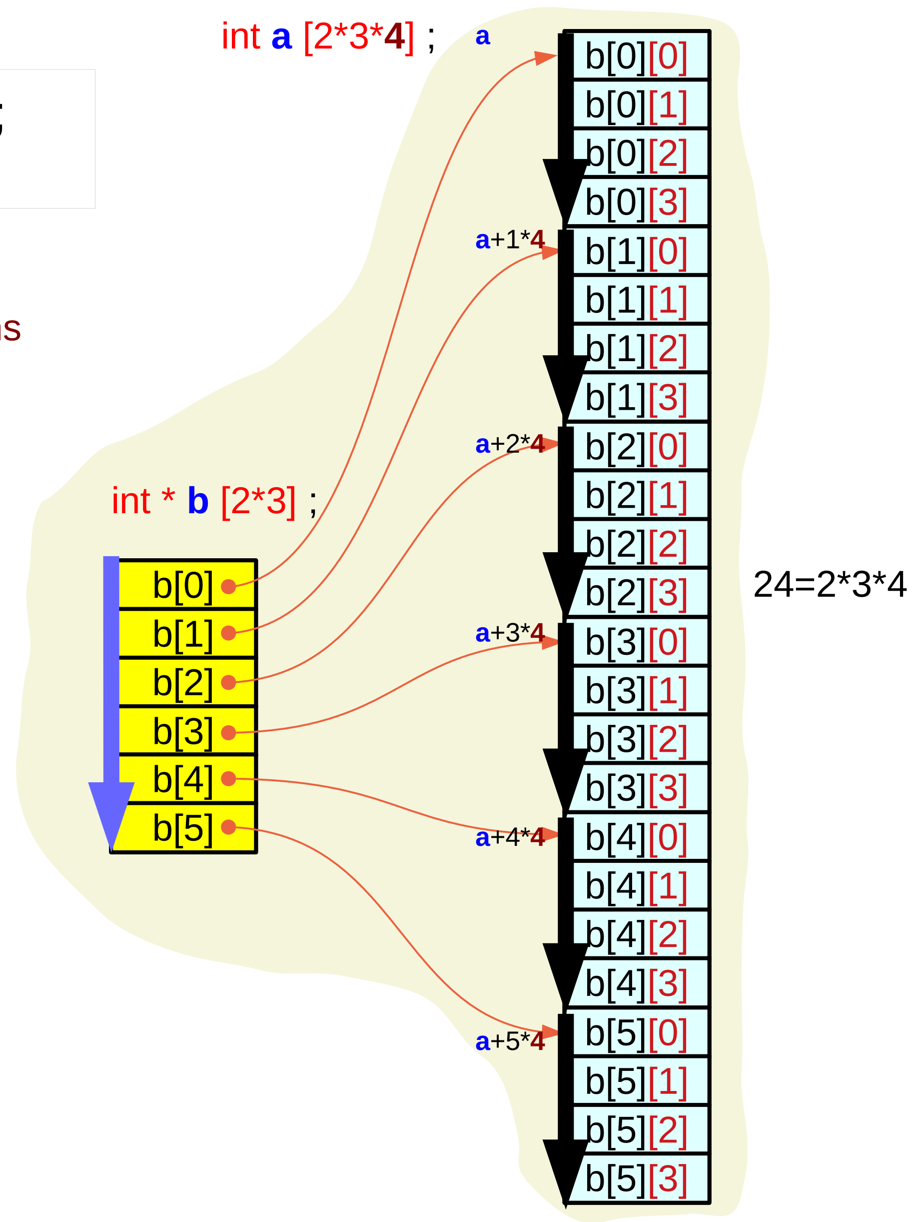
```
int     a [2*3*4] ;
int *   b [2*3] ;
```

b [j][k] ≡ a [j*4 +k]

j = 0,1,2,3,4
k = 0,1,2,3

```
b[j] = &a[j*4];
```

a, **b** take actual memory locations



c[i][j][k]	≡ *(*(*(c+i)+j)+k)	int ** c[2] ;
b[j][k]	≡ *(*(b+j)+k)	int * b[2*3] ;
a[k]	≡ *(a+k)	int a[2*3*4] ;

Accessing an int array **a** as a **3-d** array

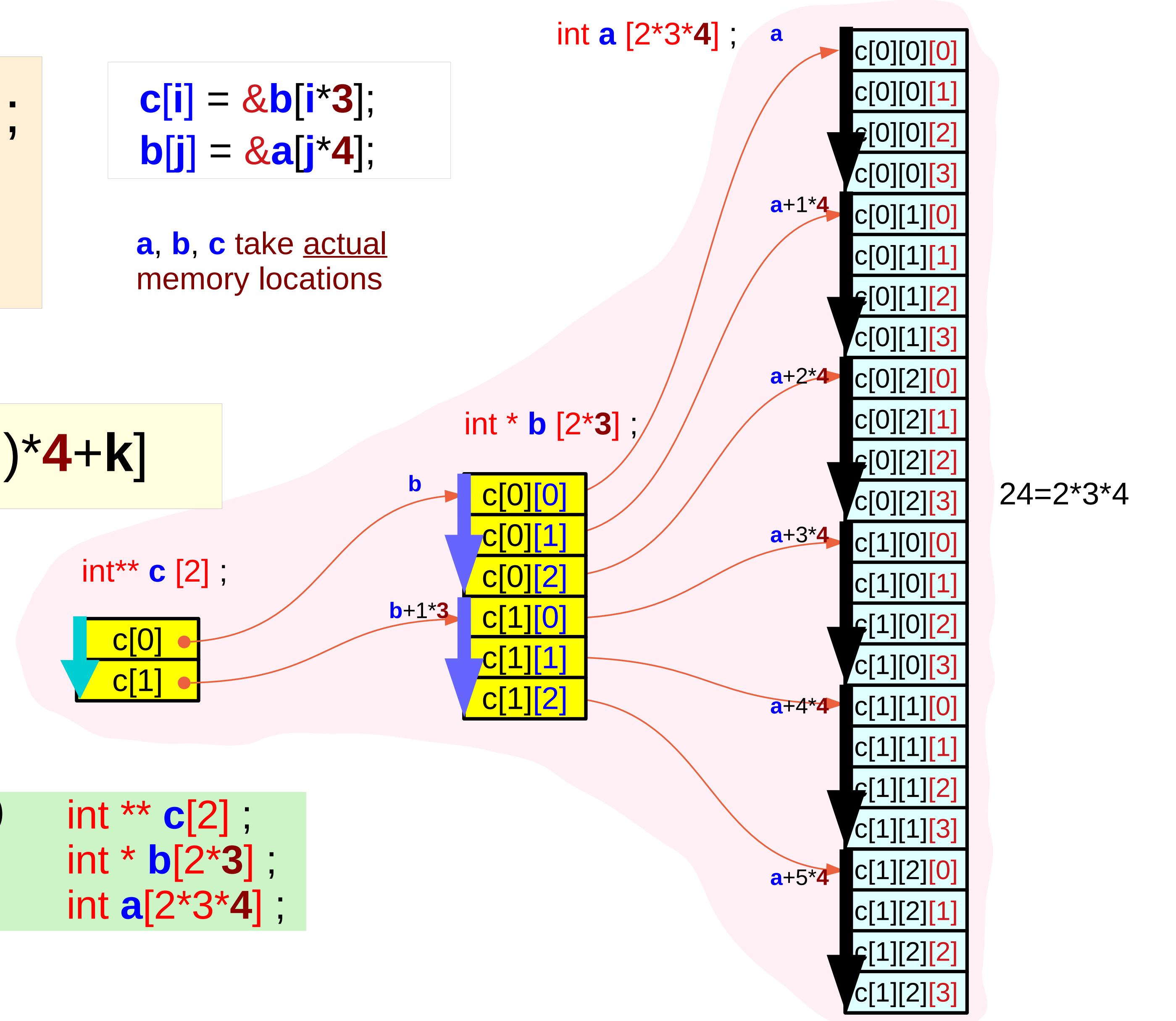
```
int      a [2*3*4] ;
int *    b [2*3] ;
int **   c [2] ;
```

c[i] = &b[i*3];
b[j] = &a[j*4];

a, **b**, **c** take actual memory locations

c [i][j][k] ≡ a [(i*3+j)*4+k]

i = 0, 1
j = 0, 1, 2
k = 0, 1, 2, 3



c[i][j][k]	≡ *(*(*(c+i)+j)+k)	int ** c[2] ;
b[j][k]	≡ *(*(b+j)+k)	int * b[2*3] ;
a[k]	≡ *(a+k)	int a[2*3*4] ;

Accessing a **non-contiguous 1-d** arrays

- **3-d** array access

Accessing **non-contiguous 1-d** arrays as a **3-d** array (1)

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

c [i][j][k]

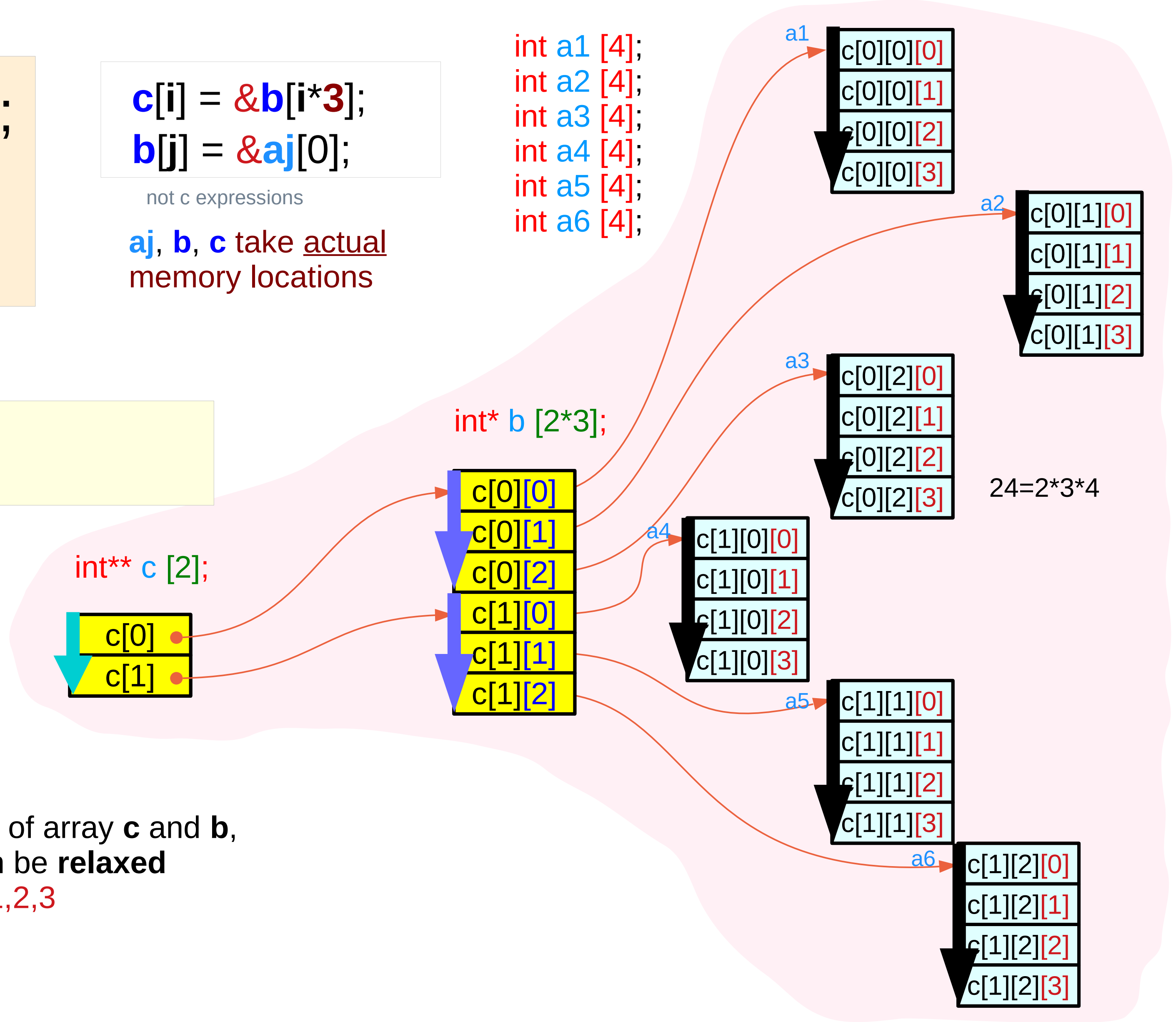
i = 0, 1
j = 0, 1, 2
k = 0, 1, 2, 3

c[**i**] = &**b**[**i***3];
b[**j**] = &**aj**[0];

not c expressions

aj, **b**, **c** take actual memory locations

int a1 [4];
int a2 [4];
int a3 [4];
int a4 [4];
int a5 [4];
int a6 [4];



Because the physical **allocation** of array **c** and **b**, the **contiguous constraints** can be **relaxed**
contiguous c[i][j][k] only for k=0,1,2,3

Accessing **non-contiguous 1-d** arrays as a **3-d** array (2)

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

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

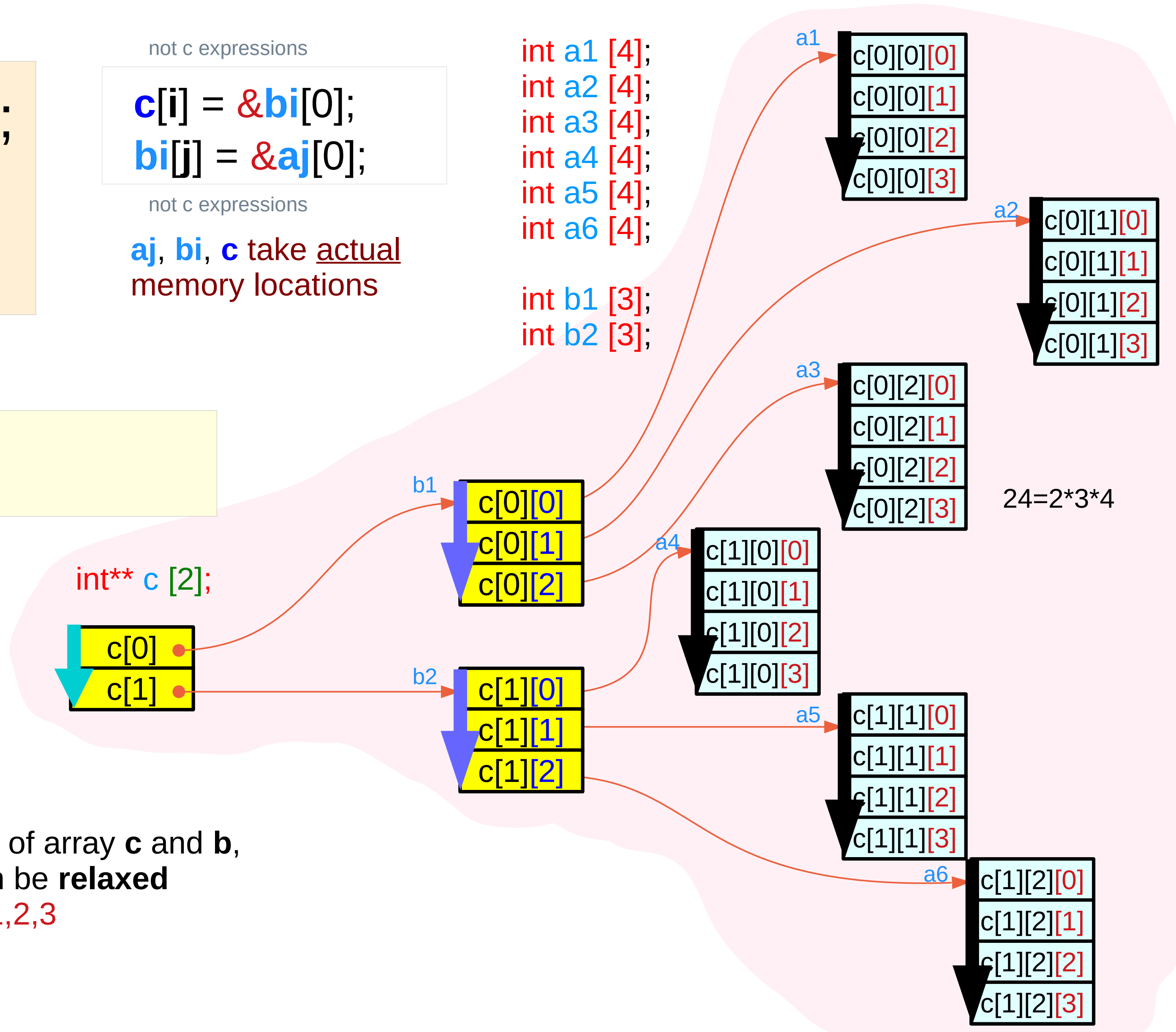
i = 0, 1
j = 0, 1, 2
k = 0, 1, 2, 3

not c expressions

c[**i**] = &**bi**[0];
bi[**j**] = &**aj**[0];

not c expressions

aj, **bi**, **c** take actual memory locations



Because the physical **allocation** of array **c** and **b**,
the **contiguous constraints** can be **relaxed**
contiguous c[i][j][k] only for k=0,1,2,3

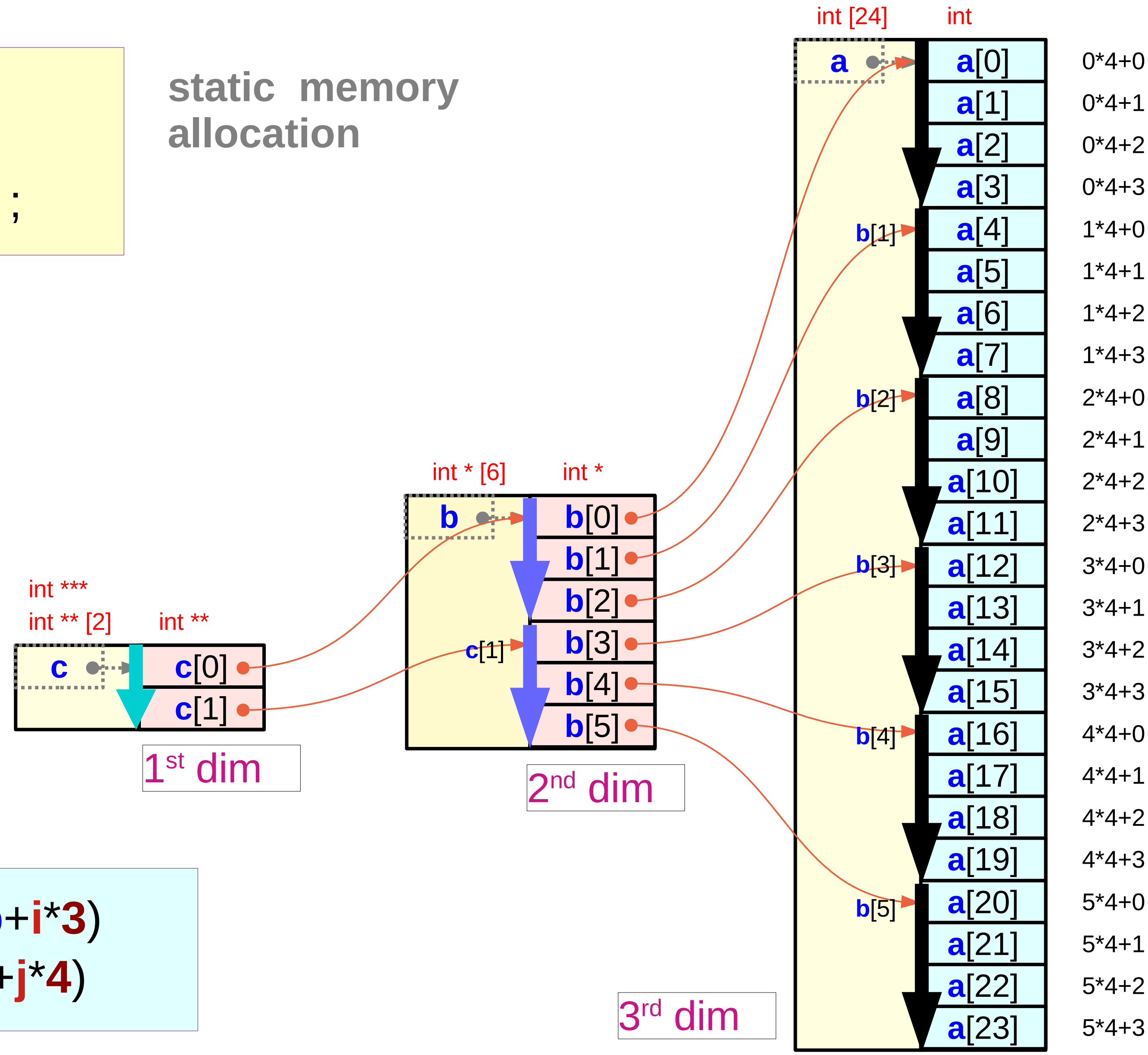
Accessing **statically** allocated arrays

Accessing **dynamically** allocated arrays

Using arrays **a**, **b**, **c** – statically allocated

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

static memory allocation



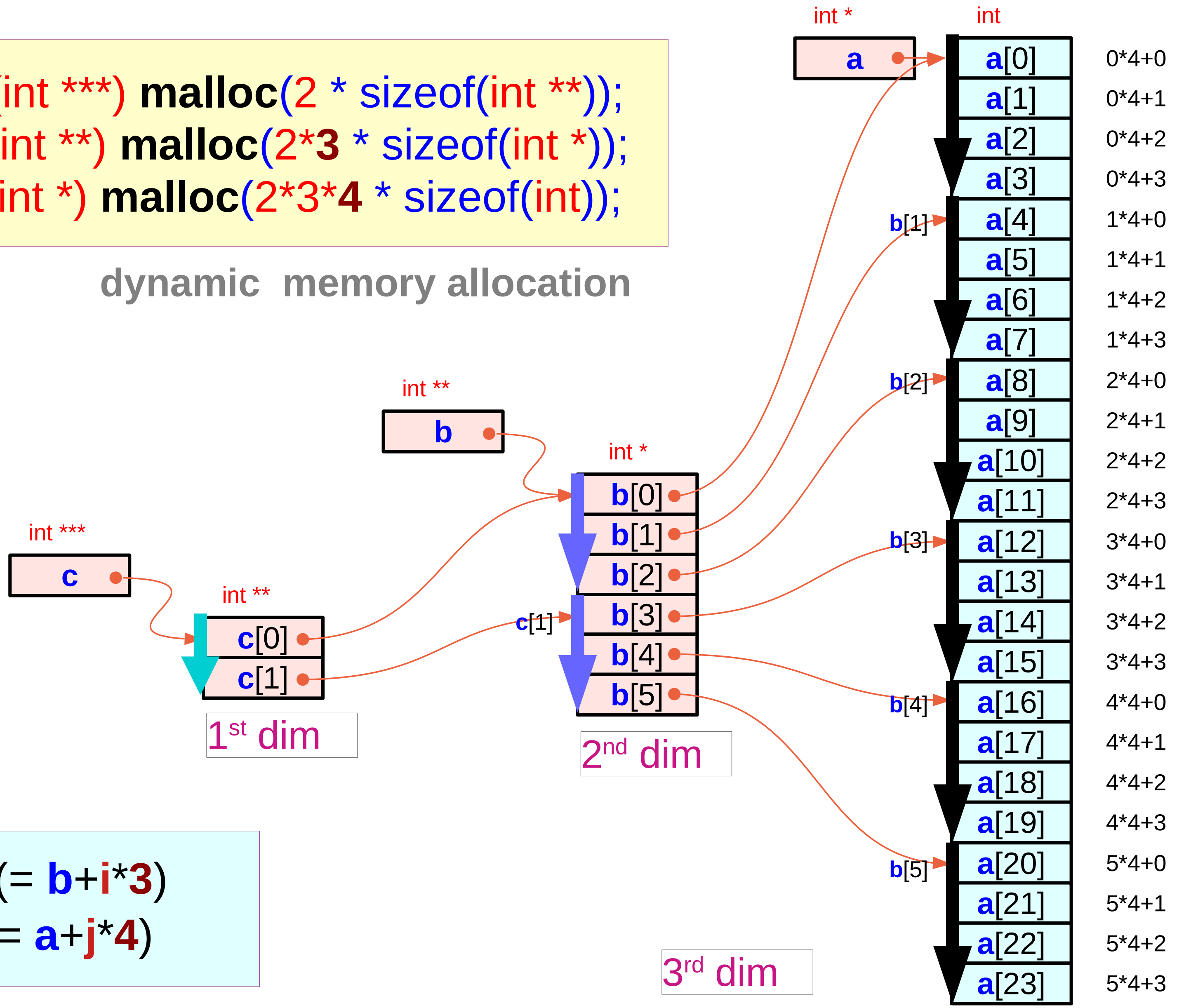
c[i] = &b[i*3] (= b+i*3)

b[j] = &a[j*4] (= a+j*4)

Using pointer a, b, c – dynamically allocated

int ***	c	= (int ***) malloc(2 * sizeof(int **));
int **	b	= (int **) malloc(2*3 * sizeof(int *));
int *	a	= (int *) malloc(2*3*4 * sizeof(int));

dynamic memory allocation



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

Static v.s. dynamic allocation (1)

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

```
int ***  c  = (int ***) malloc(2 * sizeof(int **));
int **   b  = (int **)  malloc(2*3 * sizeof(int *));
int *    a  = (int *)   malloc(2*3*4 * sizeof(int));
```

static memory allocations

type(**c**) = int ** [2] → int ***
type(**b**) = int * [2*3] → int **
type(**a**) = int [2*3*4] → int *

sizeof(**c**) = 2 * sizeof(int **)
sizeof(**b**) = 2*3 * sizeof(int *)
sizeof(**a**) = 2*3*4 * sizeof(int)

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

dynamic memory allocations

type(**c**) = int ***
type(**b**) = int **
type(**a**) = int *

sizeof(**c**) = 4 bytes on 32-bit system
sizeof(**b**) = 4 bytes on 32-bit system
sizeof(**a**) = 4 bytes on 32-bit system

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

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

Static v.s. dynamic allocation (2)

- **static allocation**

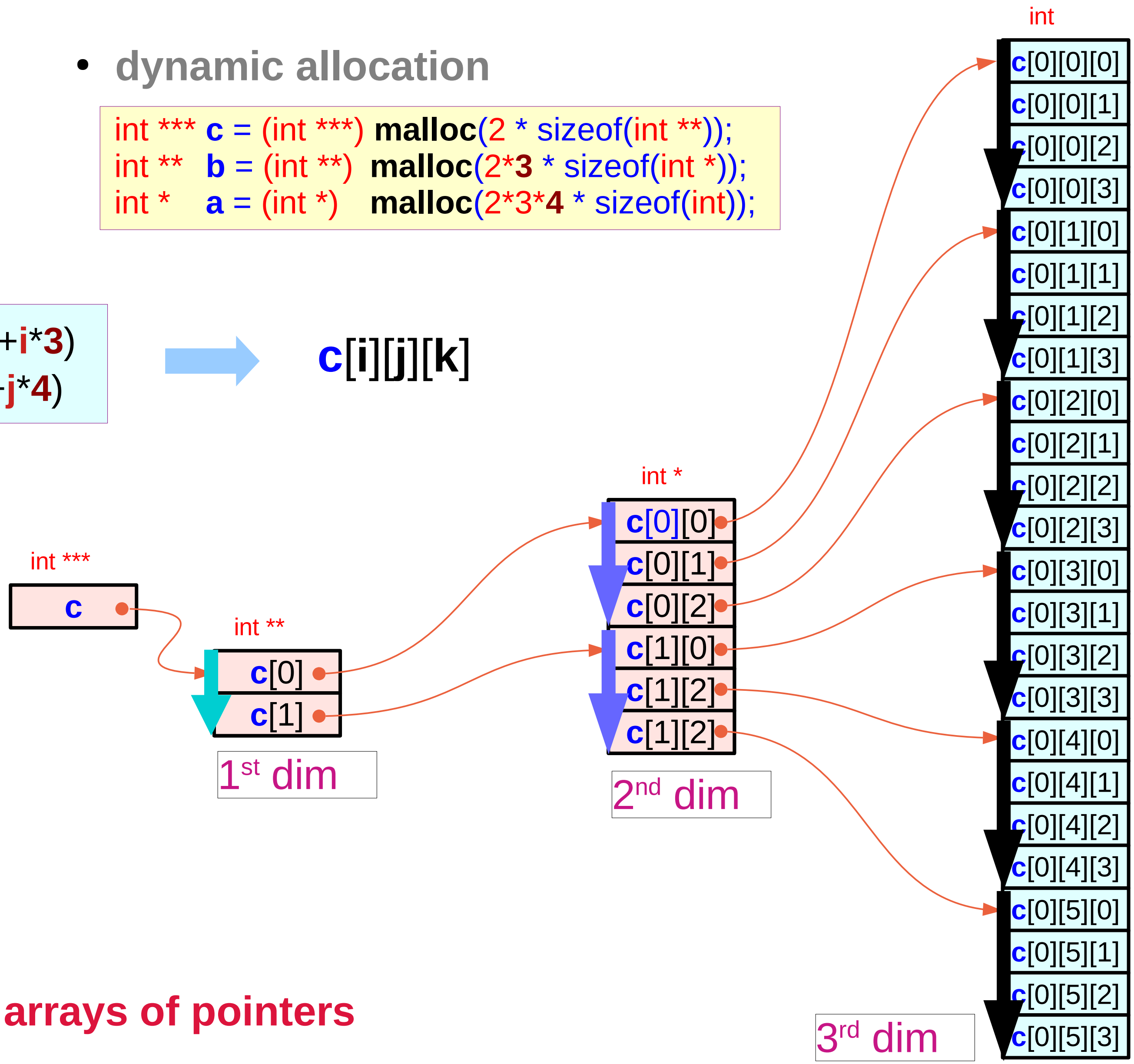
```
int **    c   [2] ;
int *     b   [2*3] ;
int       a   [2*3*4] ;
```

- **dynamic allocation**

```
int *** c = (int ***) malloc(2 * sizeof(int **));
int **  b = (int **)  malloc(2*3 * sizeof(int *));
int *   a = (int *)   malloc(2*3*4 * sizeof(int));
```

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

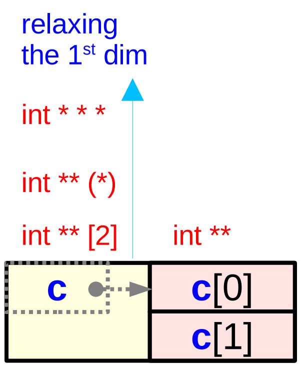
→ **c[i][j][k]**



arrays of pointers

Static v.s. dynamic allocation (3)

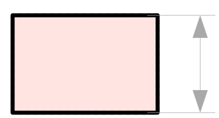
`int ***   c = (int ***)  malloc(2 * sizeof(int **));`

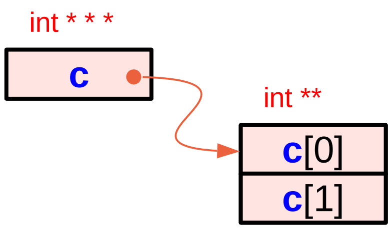


int **   c   [2] ;

static memory allocation

malloc(2 * sizeof(int **));



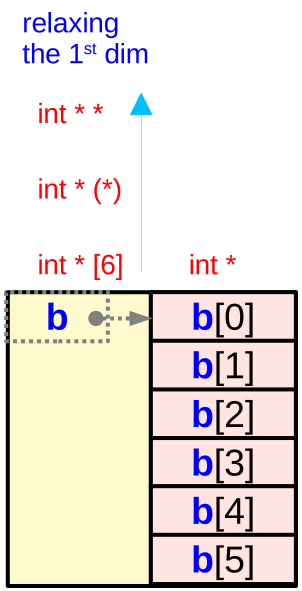


int *** **c** = (int ***) **malloc**(2 * sizeof(int **));

dynamic memory allocation

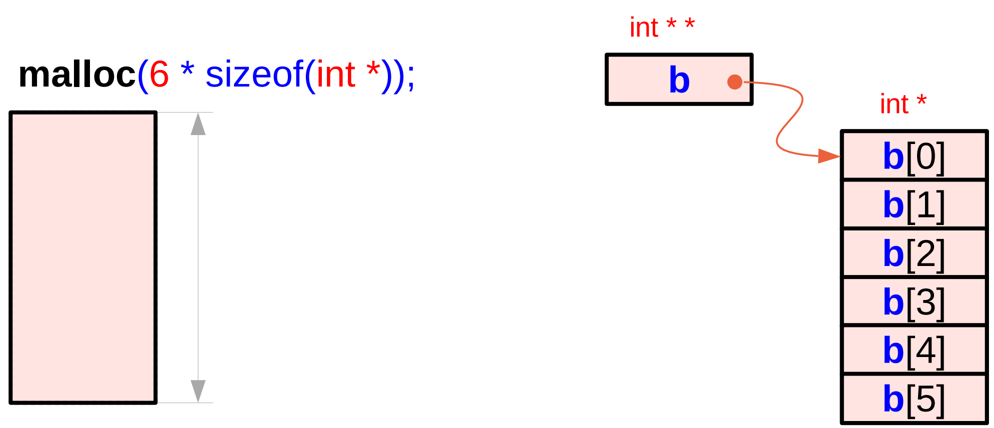
Static v.s. dynamic allocation (4)

```
int **    b = (int ***)  malloc(6 * sizeof(int *));
```




```
int *    b  [6] ;
```

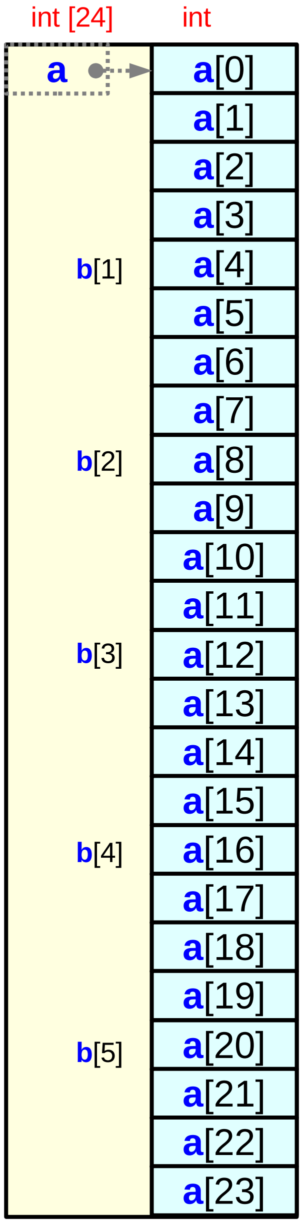
static memory allocation




```
int ** b = (int **) malloc(6 * sizeof(int *));
```

dynamic memory allocation

Static v.s. dynamic allocation (5)



int * a = (int *) **malloc**(24 * sizeof(int));

relaxing
the 1st dim

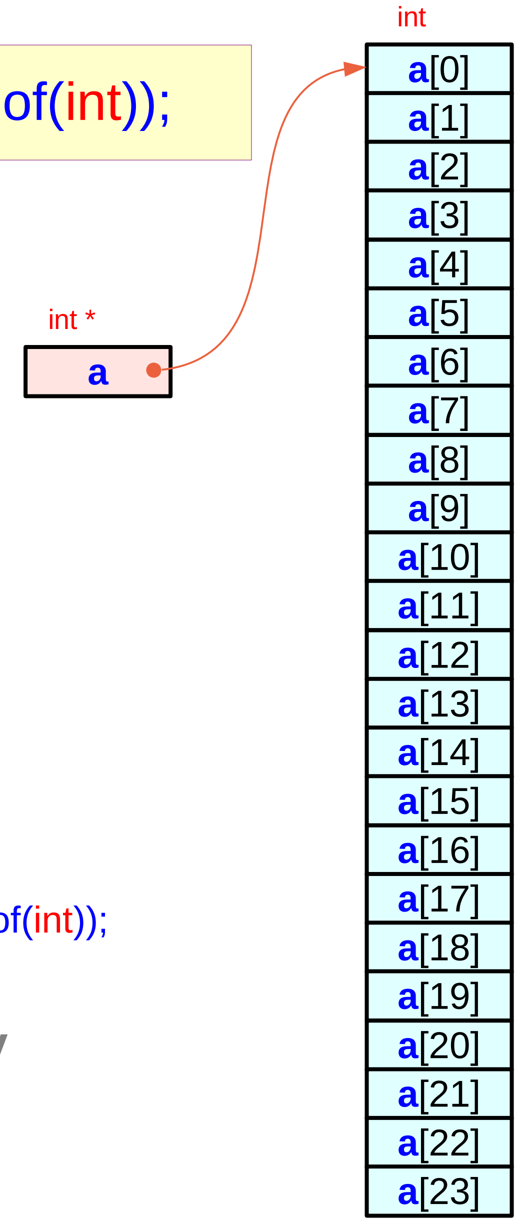
int *

int (*)

int [24]

int * **a** [24] ;

static memory allocation



int * **a** = (int *) **malloc**(24 * sizeof(int));

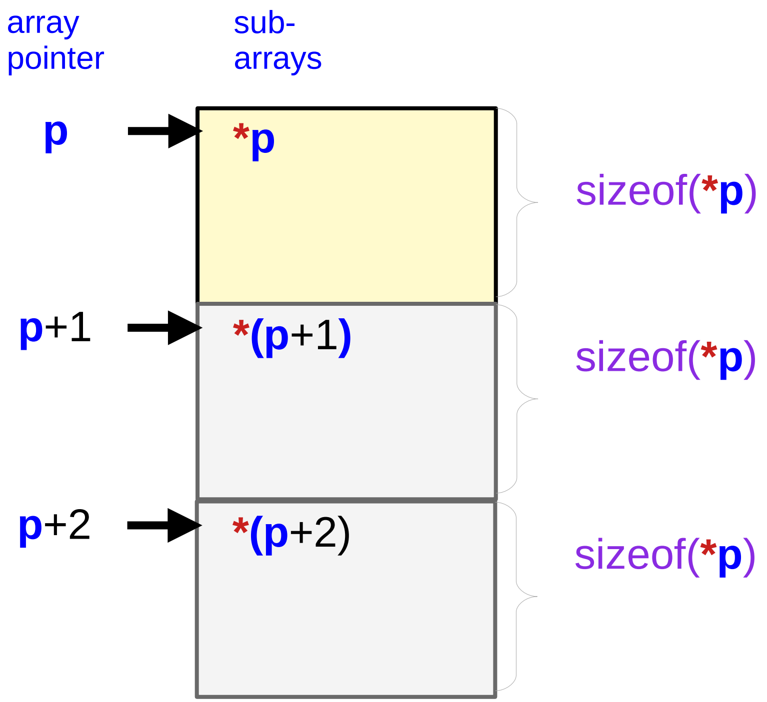
dynamic memory allocation

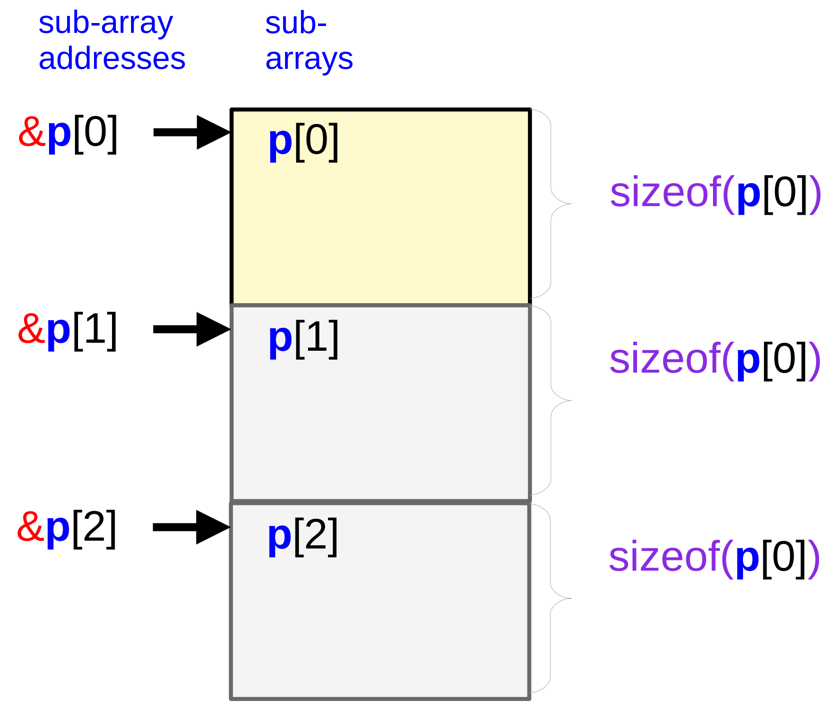
Finding sub-array sizes

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

```
            [2] [3] [4]
sizeof(c[i][j][0])    = sizeof(int)

            [2] [3] [4]
sizeof(c[i][0])       = 4 * sizeof(int)

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

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

Pointer increments and byte addresses





byte address — byte address — byte size

value(p+i) = value(p) + i * sizeof(*p)

math expression with an explicit size information

$(p+i)_{sizeof(*p)}$

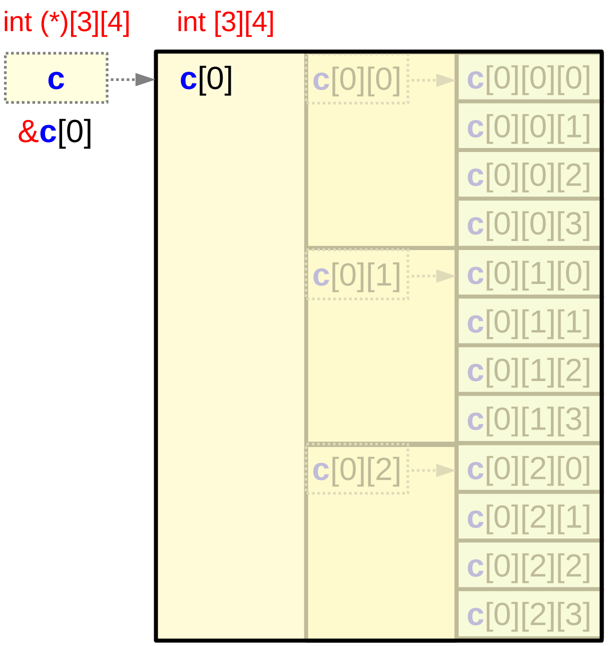
byte address — byte address — byte size

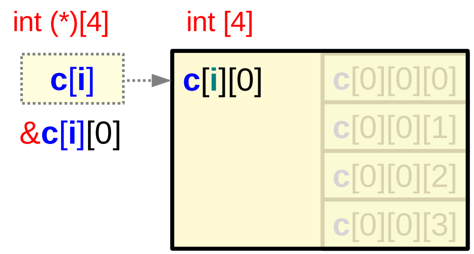
value(&p[i]) = value(p) + i * sizeof(p[0])

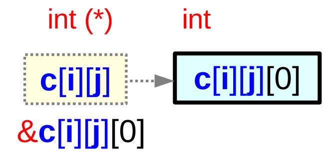
math expression with an explicit size information

$(\&p[i])_{sizeof(*p)}$

Byte addresses of subarrays &c[i], &c[i][j], &c[i][j][k]







i = 0:1
j = 0:2
k = 0:3

value(&c[i]) = value(c+i)
= value(c) + i * sizeof(*c)
= value(c) + i * sizeof(c[0])
= value(c) + i * sizeof(int) * 3 * 4

skip i elements of *c from c

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

value(&c[i][j]) = value(c[i]+j)
= value(c[i]) + j * sizeof(*c[i])
= value(c[i]) + j * sizeof(c[i][0])
= value(c[i]) + j * sizeof(int) * 4

skip j elements of *c[i] from c[i]

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

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

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

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

Address replications and subarray addresses

Address Replication

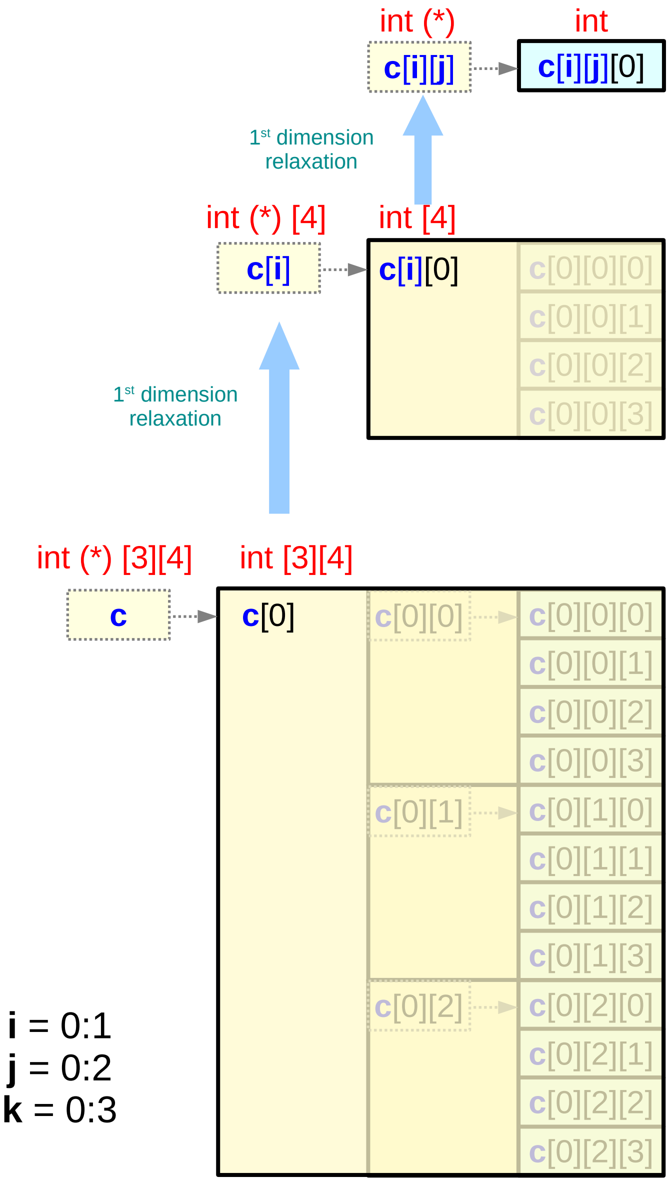
transferring pointing address
to the pointer that references itself

&X = X

byte addresses of abstract data		values of virtual pointers
int (*) int value(&c[i][j][k])	=	int (*) value(c[i][j]) + k * sizeof(int)
int (*) [4] int [4] value(&c[i][j])	=	int (*) [4] value(c[i]) + j * 4 * sizeof(int)
int (*) [3][4] int [3][4] value(&c[i])	=	int (*) [3][4] value(c) + i * 3 * 4 * sizeof(int)

byte addresses
of abstract data

values of
virtual pointers



i = 0:1
j = 0:2
k = 0:3

Address replications in a multi-dimensional array

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

equivalences

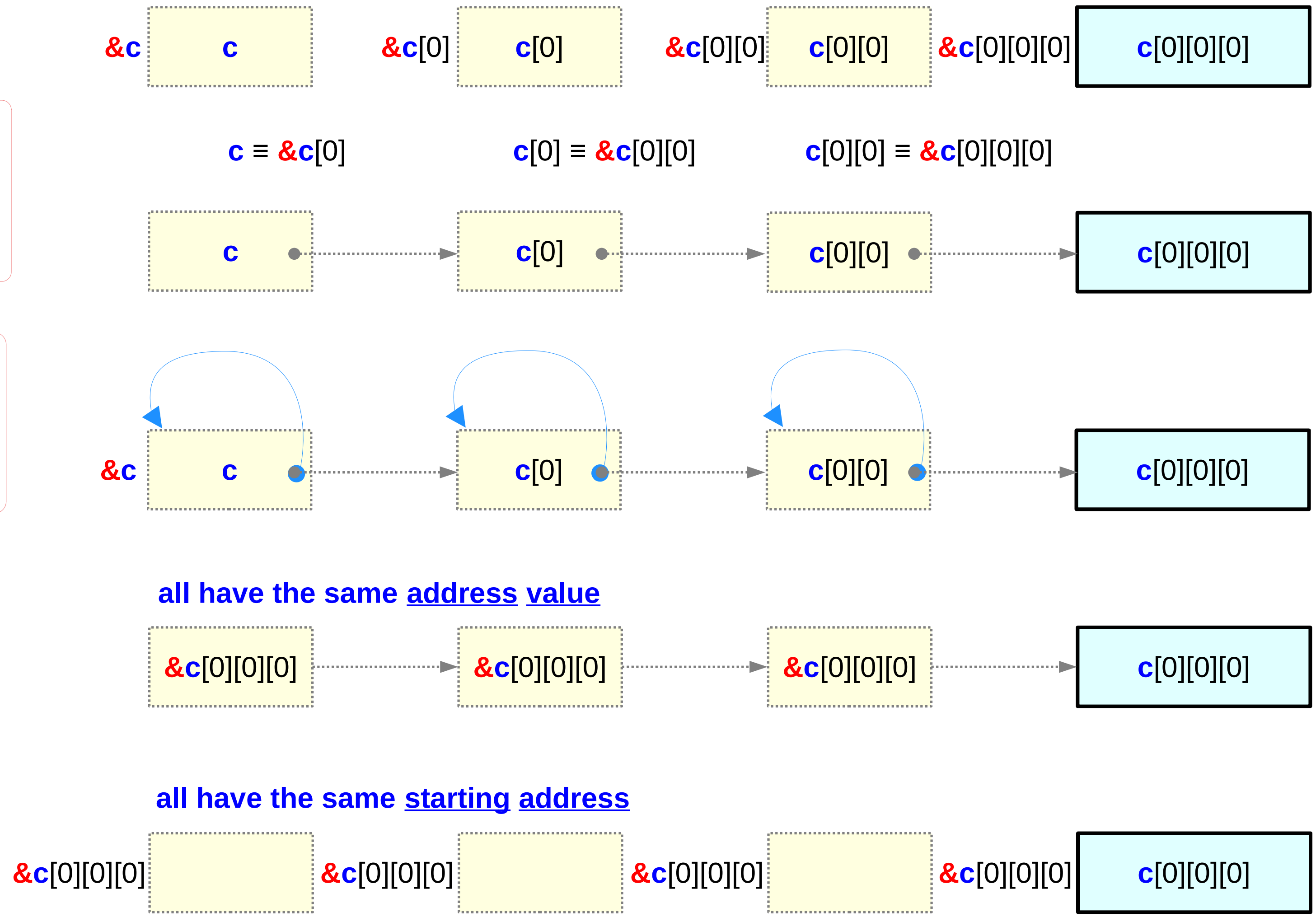
c[i][j] ≡ &c[i][j][0]
c[i] ≡ &c[i][0]
c ≡ &c[0]

address replication

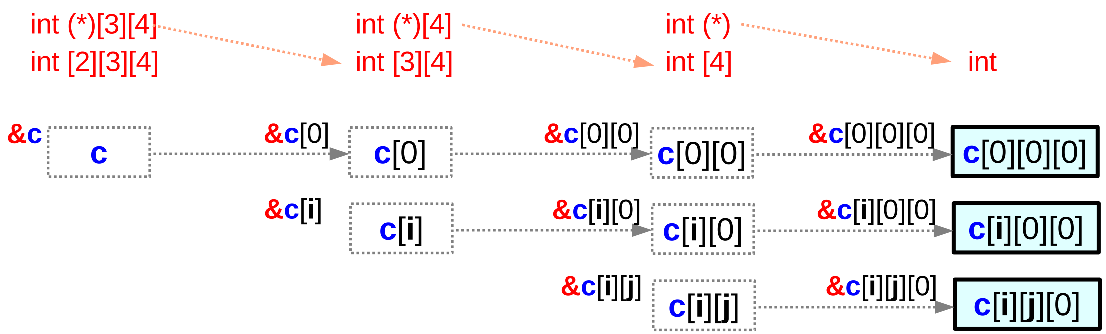
value(c[i][j]) = value(&c[i][j])
value(c[i]) = value(&c[i])
value(c) = value(&c)

c, c[0], c[0][0] :
these virtual pointers have
the same address value

a physical location
has a unique address



Referencing sub-arrays



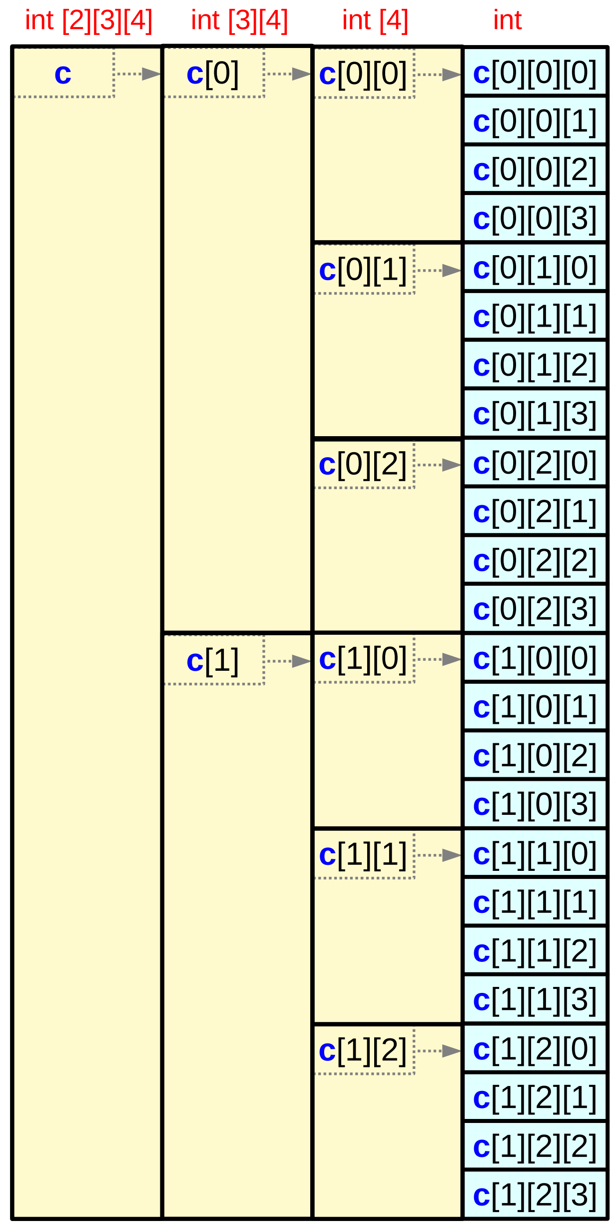
equivalence relations

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

address replication

value(c[i][j]) = value(&c[i][j]) = value(c[i]+j) = *value(c[i]+j)
value(c[i]) = value(&c[i]) = value(c+i) = *value(c+i)

c[i], c[i][0] point to the same data c[i][0][0]
c, c[0], c[0][0] point to the same data c[0][0][0]



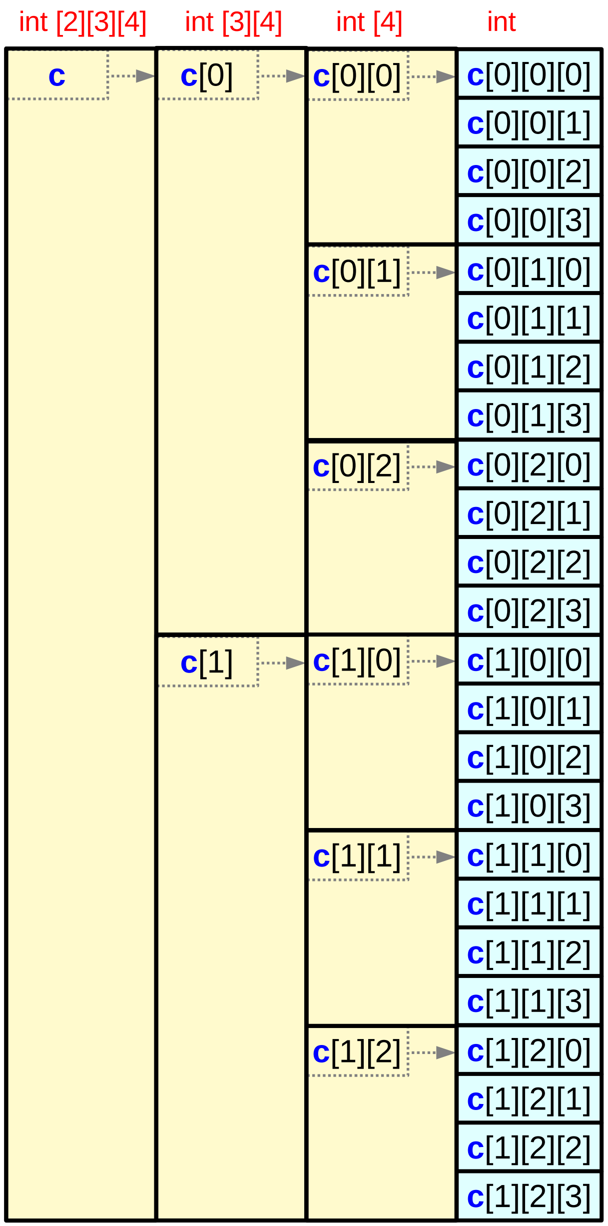
Types, sizes, and values of sub-arrays

int **c** [2][3][4] ;   **static allocation**

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

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

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



pointers to arrays

Using multi-dimensional arrays

Pointer Array Approach

– using explicit pointers

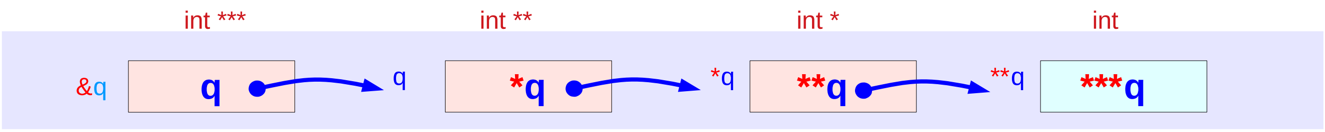
Array Pointer Approach

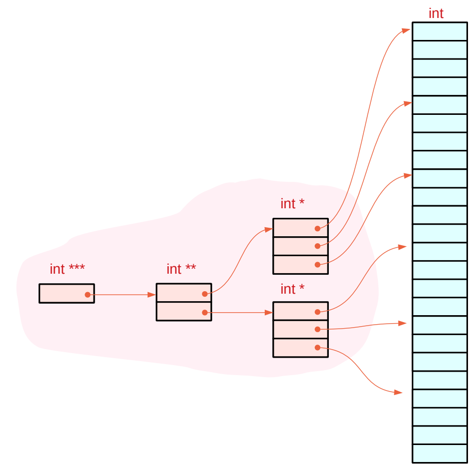
– using implicit pointers

Two types of **3-d** array accesses

Pointer Array Approach (arrays of pointers)

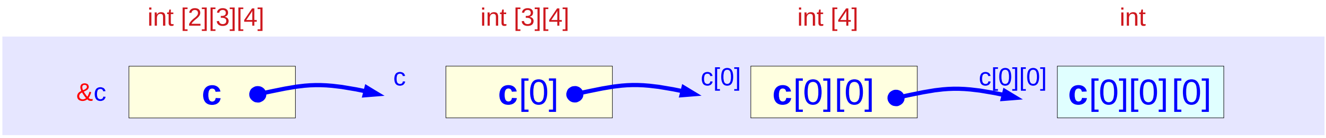
Pointer Chain Type I

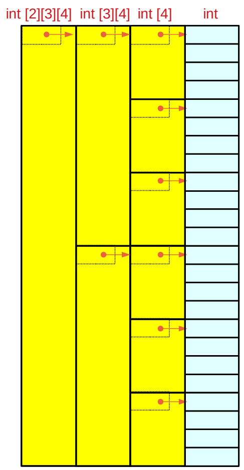




Array Pointer Approach (pointers to arrays)

Pointer Chain Type II





Pointer addition – **math** and **c** expressions

Accessing c[i][j][k] – *unified **c** expressions*

skip **i** elements of **c**[**i**] from **c**

skip **j** elements of **c**[**i**][**j**] from **c**[**i**]

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

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

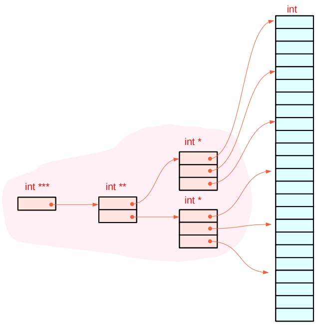
Pointer Array Approach

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

sizeof(***c**) = 1*4

sizeof(***c**[**i**]) = 1*4

sizeof(***c**[**i**][**j**]) = 1*4



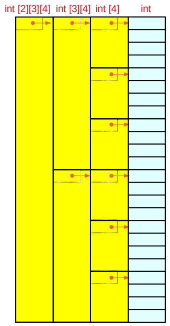
Array Pointer Approach

$(c + i)_{3 \cdot 4 \cdot 4}$ $(c[i] + j)_{4 \cdot 4}$ $(c[i][j] + k)_{1 \cdot 4}$

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

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

sizeof(***c**[**i**][**j**]) = 1*4



Accessing **c[i][j][k]** element

Accessing c[i][j][k]

skip **i** elements
of **c**[**i**] from **c**

(**c** + **i**)

skip **j** elements
of **c**[**i**][**j**] from **c**[**i**]

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

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

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

Pointer Array Approach

skip **i** * sizeof(int **)
(= **i***4) bytes from **c**

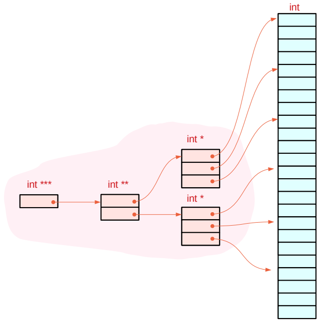
$(\mathbf{c} + \mathbf{i})_{1 \cdot 4}$

skip **j** * sizeof(int *)
(= **j***4) bytes from **c**[**i**]

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

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

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



Array Pointer Approach

skip **i** * sizeof(int [3][4])
(= **i***3*4*4) bytes from **c**

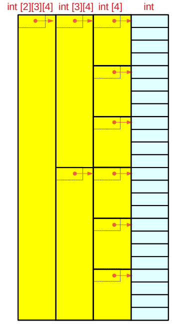
$(\mathbf{c} + \mathbf{i})_{3 \cdot 4 \cdot 4}$

skip **j** * sizeof(int [4])
(= **j***4*4) bytes from **c**[**i**]

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

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

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



Accessing c[i][j][k] – Pointer Array Approach

Pointer Array Approach

skip **i** * sizeof(int **)
= **i** *4 bytes from **c**

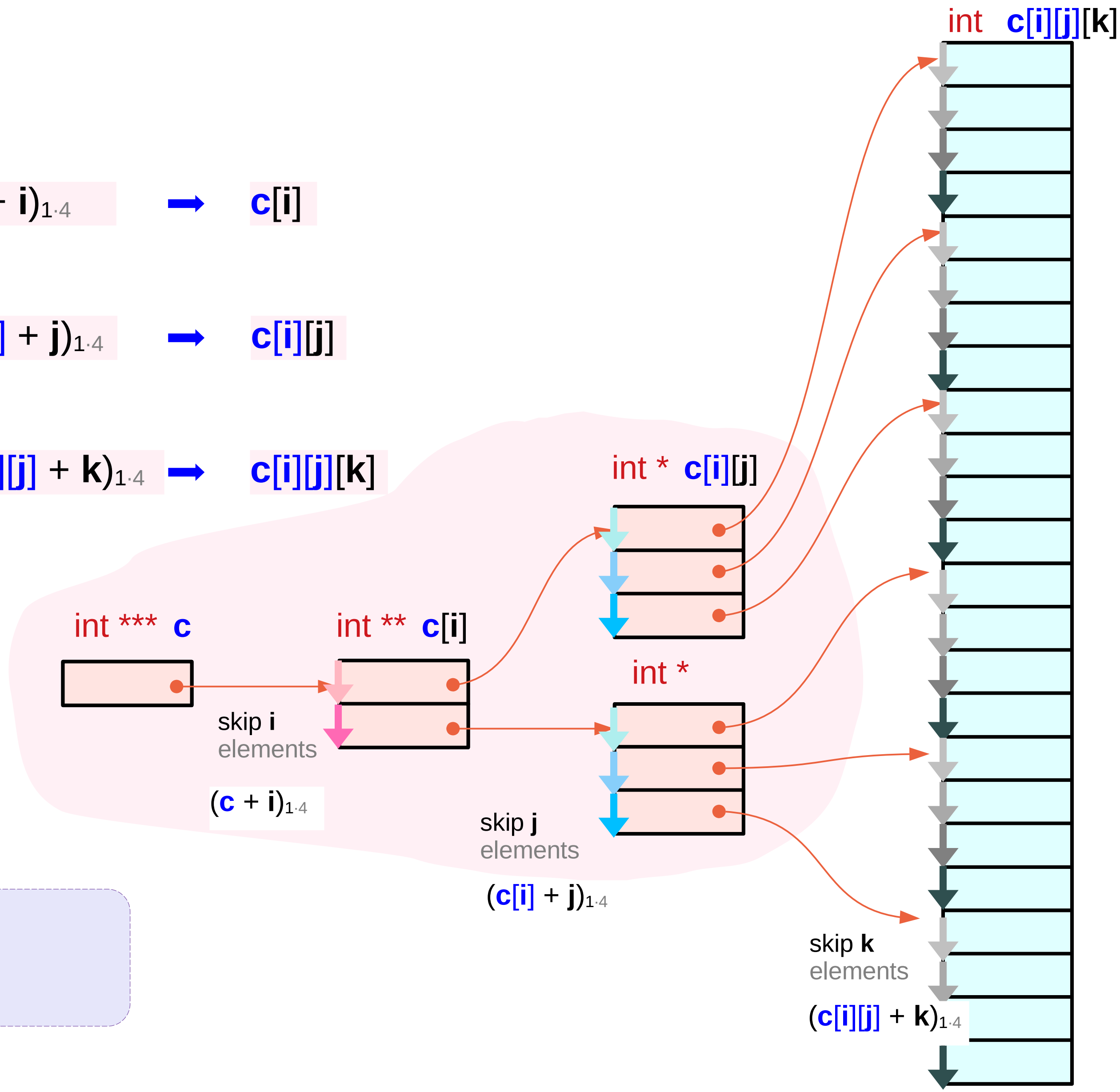
$(c + i)_{1\cdot 4}$ ➡ c[i]

skip **j** * sizeof(int *)
= **j** *4 bytes from **c[i]**

$(c[i] + j)_{1\cdot 4}$ ➡ c[i][j]

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

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



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

Accessing **c[i][j][k]** – Array Pointer Approach

Array **Pointer** Approach

skip **i** * sizeof(int [3][4])
= **i** *3*4*4 bytes from **c**

$(c + i)_{3\cdot 4\cdot 4}$ ➡ **c**[i]

skip **j** * sizeof(int [4])
= **j** *4*4 bytes from **c[i]**

$(c[i] + j)_{4\cdot 4}$ ➡ **c[i][j]**

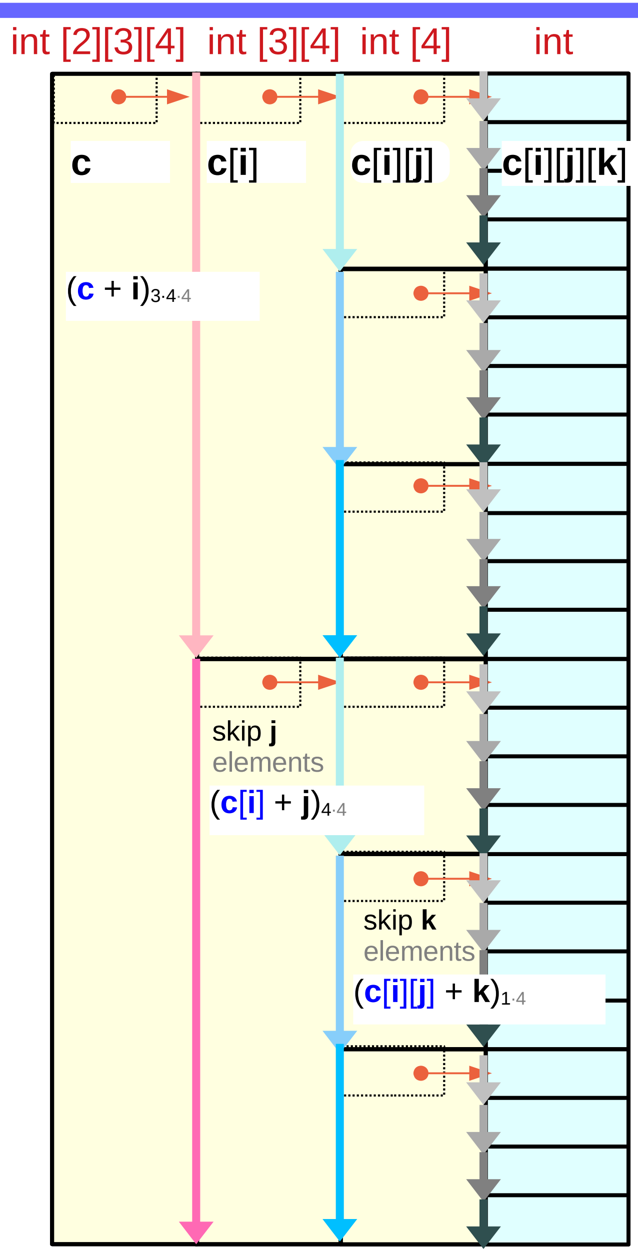
skip **k** * sizeof(int)
= **k** *4 bytes from **c[i][j]**

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

- ***subarray partitioning***
- ***address replication***

size information

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



Array element address – Pointer Array Approach

equivalence relations – c expressions

&c[i][j][k] = (c[i][j] + k)
&c[i][j] = (c[i] + j)
&c[i] = (c + i)

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

size information

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

address fetch – math expressions

value(c[i][j]) ⇒ *value((c[i] + j)$_{1\cdot4}$) = *value(c[i] + j * 4) ⇐ sizeof(*c[i])
value(c[i]) ⇒ *value((c + i)$_{1\cdot4}$) = *value(c + i * 4) ⇐ sizeof(*c)

address of c[i][j][k] – math expressions

&c[i][j][k] = value((c[i][j] + k)$_{1\cdot4}$)
= value(*value((c[i] + j)$_{1\cdot4}$) + k * 4)
= value(*value(*value((c + i)$_{1\cdot4}$) + j * 4) + k * 4)
= value(*value(*value(c + i * 4) + j * 4) + k * 4)

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

Array element address – Array Pointer Approach

equivalence relations – c expressions

&c[i][j][k] = (c[i][j] + k)
&c[i][j] = (c[i] + j)
&c[i] = (c + i)

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

size information

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

address replication – math expressions

value(c[i][j]) = value(&c[i][j]) ⇒ value($(c[i] + j)_{4\cdot 4}$) = value(c[i]) + j * 4*4 ⇐ sizeof(*c[i])
value(c[i]) = value(&c[i]) ⇒ value($(c + i)_{3\cdot 4\cdot 4}$) = value(c) + i * 3*4*4 ⇐ sizeof(*c)

address of c[i][j][k] – math expressions

&c[i][j][k] = value($(c[i][j] + k)_{1\cdot 4}$)
= value($(c[i] + j)_{4\cdot 4}$) + k * 4
= value($(c + i)_{3\cdot 4\cdot 4}$) + j * 4*4 + k * 4
= value(c) + i * 3*4*4 + j * 4*4 + k * 4

⇐ &c[i][j][k] ≡ c[i][j] + k
⇐ &c[i][j] ≡ c[i] + j ⇐ c[i][j] address replication
⇐ &c[i] ≡ c + i ⇐ c[i] address replication

- ***address replication***
- ***combining size and address information***

Accessing c[i][j][k] via byte addresses

Pointer Array Approach

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

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

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

three memory accesses for **c[i][j][k]**

Array Pointer Approach

&c[i][j][k] = value((c[i][j] + k)$_{1\cdot 4}$) = value(c[i][j] + k * 4)
&c[i][j] = value((c[i] + j)$_{4\cdot 4}$) = value(c[i] + j * 4*4)
&c[i] = value((c + i)$_{3\cdot 4\cdot 4}$) = value(c + i * 3*4*4)

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

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

address replication
single memory access for **c[i][j][k]**

Equivalence relations in c[i][j][k]

Pointer Array Approach

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

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

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

different semantics !
be careful in mixed c and math expressions

Array Pointer Approach

&c[i][j][k] = value($(c[i][j] + k)_{1 \cdot 4}$) = value(c[i][j]) + k * 4
&c[i][j] = value($(c[i] + j)_{4 \cdot 4}$) = value(c[i]) + j * 4*4
&c[i] = value($(c + i)_{3 \cdot 4 \cdot 4}$) = value(c) + i * 3*4*4

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

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

different semantics !
be careful in mixed c and math expressions

Accessing c[i][j][k]

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

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

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

equivalence relations			multiple indirections		address replications
c[i]	≡ *(c+i)	≡	*(c+i)	≡	(c+i)
c[i][j]	≡ *(c[i]+j)	≡	*(*(c+i)+j)	≡	((c+i)+j)
c[i][j][k]	≡ *(c[i][j]+k)	≡	*(*(*(c+i)+j)+k)	≡	(((c+i)+j)+k)
				➡	*(c +i +j +k)

Pointer Array Approach

Array Pointer Approach

Conditions for c[i][j][k]

Equivalence relations in c[i][j][k]

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

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

Pointer Array Approach

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

⟺

contiguous 4 c[i][j][k]'s	4 * (int	4 bytes)
contiguous 3 c[i][j]'s	3 * (int *	4 or 8 bytes)
contiguous 2 c[i]'s	2 * (int **	4 or 8 bytes)

Array Pointer Approach

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

⟺

contiguous 4 c[i][j][k]'s	4 * (int	4 bytes)
contiguous 3 c[i][j]'s	3 * (int [4]	4*4 bytes)
contiguous 2 c[i]'s	2 * (int [3][4]	3*4*4 bytes)

Skipping leaf elements

Continuity Constraints

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

contiguous c[i][j][k]	over k=0:3
contiguous c[i][j]	over j=0:2
contiguous c[i]	over i=0:1

Pointer Array Approach

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

k leaf elements	k * 4 bytes
j * 4 leaf elements	j * 4 * 4 bytes
i * 3 * 4 leaf elements	i * 3 * 4 * 4 bytes

Array Pointer Approach

$(c[i][j] + k)_{1 \cdot 4}$	skip k * 4 bytes from c[i][j]
$(c[i] + j)_{4 \cdot 4}$	skip j * 4*4 bytes from c[i]
$(c + i)_{3 \cdot 4 \cdot 4}$	skip i * 3*4*4 bytes from c

k leaf elements	k * 4 bytes
j * 4 leaf elements	j * 4 * 4 bytes
i * 3 * 4 leaf elements	i * 3 * 4 * 4 bytes

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

Accessing c[i][j][k] – Conditions

General requirements

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

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

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

Pointer array approach

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

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

c[i] ⇐ &b[i*3]
b[j] ⇐ &a[j*4]

Hierarchical Pointer Arrays

Array pointer approach

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

c[i][j][k]	:: int
c[i][j]	:: int [4]
c[i]	:: int [3][4]

c ⇐ &c[0][0][0]
c[i] ⇐ &c[i][0][0]
c[i][j] ⇐ &c[i][j][0]

Virtual Array Pointers

Accessing c[i][j][k] – Pointer Array Approach (1)

c[i] ⇐ &b[i*3]
b[j] ⇐ &a[j*4]

⇒

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

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

[2][3][4]

b[j] ≡ (a+j*4)

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

b[j][k] ≡ a[j*4+k]

c[i] ≡ (b+i*3)

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

c[i][j] ≡ b[i*3+j] ⇒ c[i][j] ≡ (a+(i*3+j)*4)

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

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

c[i][j][k] ≡ a[(i*3+j)*4+k]

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

Accessing c[i][j][k] – Pointer Array Approach (2)

c[i] ⇐ &b[i*3]
b[j] ⇐ &a[j*4]

[2][3][4]

$(c + i)_{1\cdot 4}$

skip i * sizeof(int **)
i * 4 bytes from c

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

skip j * sizeof(int *)
j * 4 bytes from c[i]

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

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

b[j] ≡ (a+j*4)

skip j elements of b — skip j*4 elements of a

c[i] ≡ (b+i*3)

skip i elements of c — skip i*3 elements of b

b[j][k] ≡ a[j*4+k]

skip j elements of b +
skip k elements of a

c[i][j] ≡ b[i*3+j]

skip i elements of c +
skip j elements of b

skip i*3 elements of b +
skip j elements of b

c[i][j] ≡ (a+(i*3+j)*4)

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

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

c[i][j][j] ≡ a[(i*3+j)*4+k]

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

Accessing c[i][j][k] – Array Pointer Approach (1)

c	⇐	&c[0][0][0]
c[i]	⇐	&c[i][0][0]
c[i][j]	⇐	&c[i][j][0]

→

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

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

[2][3][4]

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

→

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

→

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

→

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

→

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

Accessing **c[i][j][k]** – Array Pointer Approach (2)

c	⟸	&**c**[0][0][0]
c[**i**]	⟸	&**c**[**i**][0][0]
c[**i**][**j**]	⟸	&**c**[**i**][**j**][0]

[2][3][4]

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

skip
i * sizeof(int [3][4])
i * 3*4*4 bytes
from **c**

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

skip
j * sizeof(int [4])
j *4*4 bytes
from **c**[**i**]

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

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

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

skip **i** elements of **c**[**i**]	**c**[**i**]	skip **i*3*4** elements of **c**[**i**][0][0]

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

skip **i** elements of **c**[**i**]	**c**[**i**]	skip **i*3*4** elements of **c**[**i**][0][0]
+		+
skip **j** elements of **c**[**i**][**j**]	**c**[**i**][**j**]	skip **j*4** elements of **c**[**i**][**j**][0]

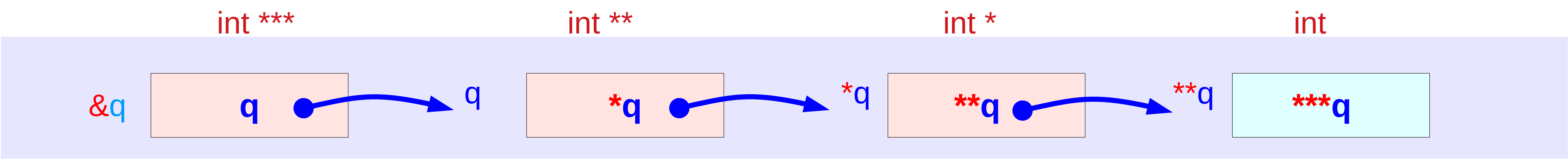
value(**c**[**i**][**j**][**k**]) = &**c**[**i**][**j**][**k**]

skip **i** elements of **c**[**i**]	**c**[**i**]	skip **i*3*4** elements of **c**[**i**][0][0]
+		+
skip **j** elements of **c**[**i**][**j**]	**c**[**i**][**j**]	skip **j*4** elements of **c**[**i**][**j**][0]
+		+
skip **k** elements of **c**[**i**][**j**][**k**]	**c**[**i**][**j**][**k**]	skip **k** elements of **c**[**i**][**j**][**k**]

Pointer Chain Types

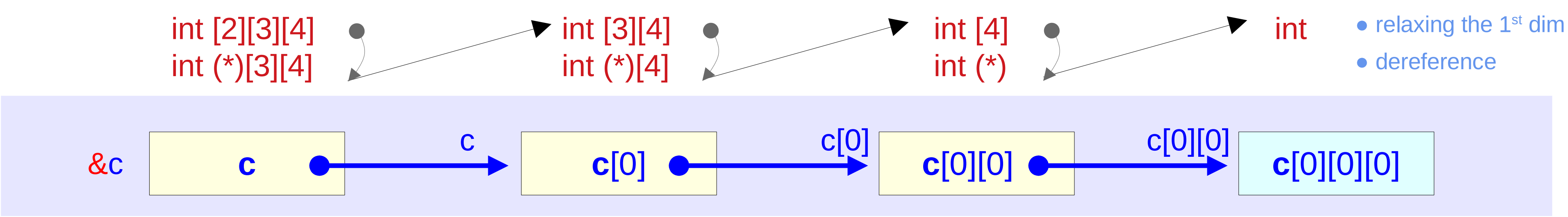
Pointer Chain Types

Pointer Chain Type I



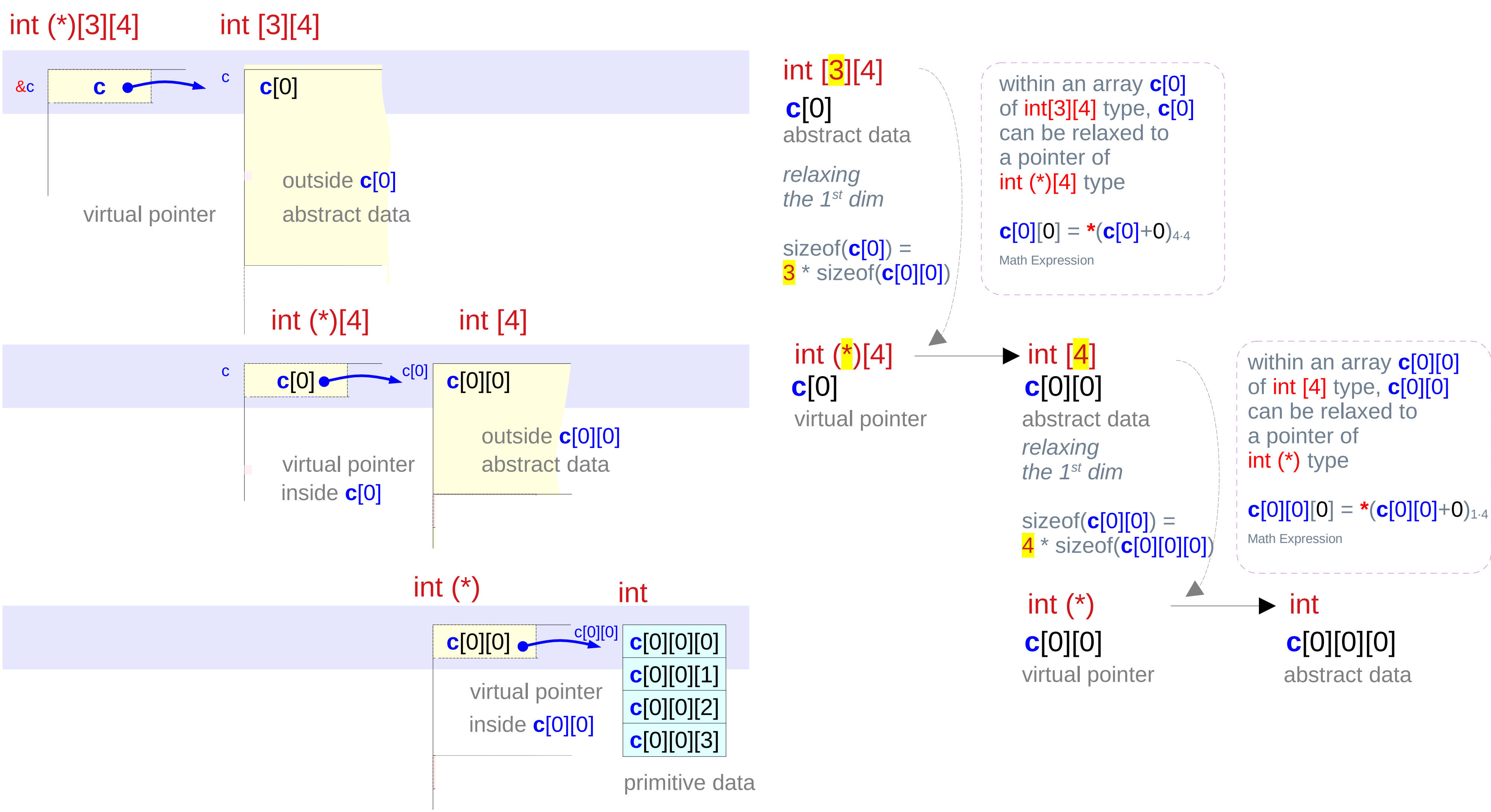
- **Pointer Array Approach** (index operations are handled by a user)

Pointer Chain Type II



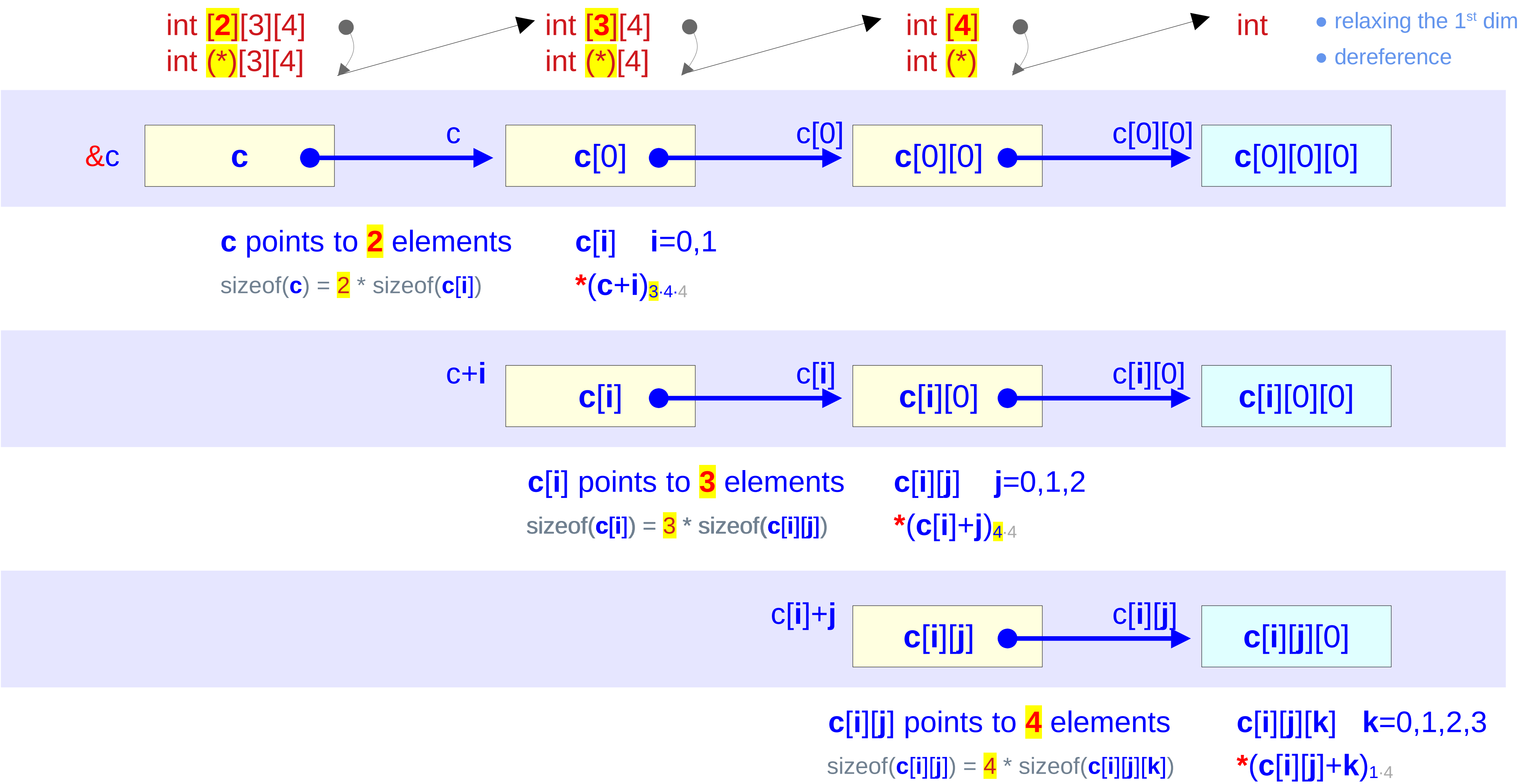
- **Array Pointer Approach** (index operations are handled by a compiler)

Examples of two step dereferencing in type **II**



Pointer Chains in Type **II** (1)

Pointer Chain Type II



Pointer Chains in Type **II** (2)

- relaxing the 1st dim
- dereference

i = 0,1

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

c points to **2** elements

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

c[**i**]   **i**=0,1

*(**c**+**i**)$_{3\cdot4\cdot4}$

j = 0,1,2

int [**3**][4] → int [4]
int (*)[4]

c[**i**] points to **3** elements

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

c[**i**][**j**]   **j**=0,1,2

*(**c**[**i**]+**j**)$_{4\cdot4}$

k = 0,1,2,3

int [**4**] → int
int (*)

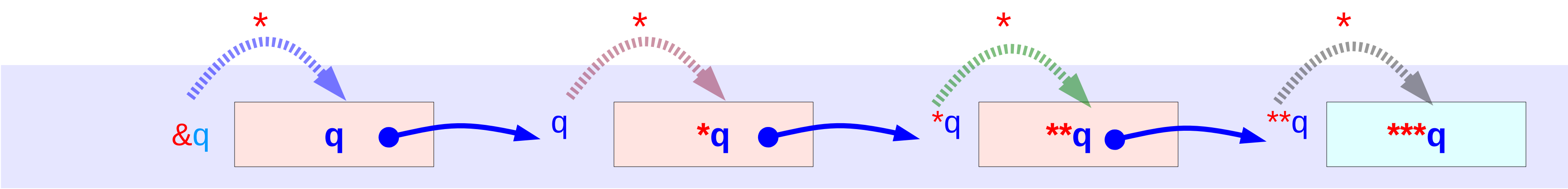
c[**i**][**j**] points to **4** elements

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

c[**i**][**j**][**k**]   **k**=0,1,2,3

*(**c**[**i**][**j**]+**k**)$_{1\cdot4}$

Pointer Chain Type I – * and & operators



*(&q) ≡ q

*(q) ≡ *q

*(*q) ≡ **q

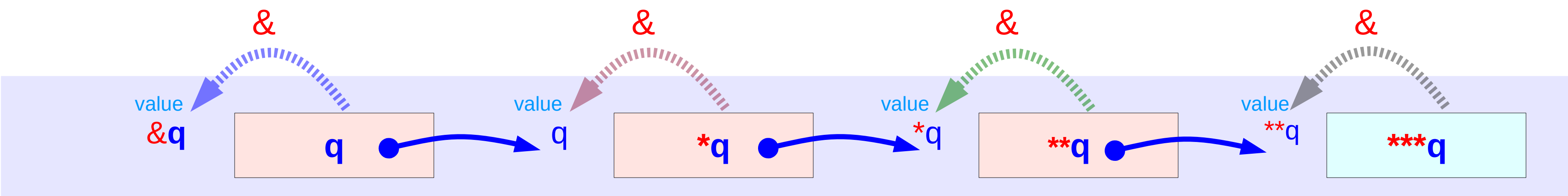
*(**q) ≡ ***q

C expression *(&q)
equals to the variable q

C expression *(q)
equals to the variable q

C expression *(*q)
equals to the variable **q

C expression *(**q)
equals to the variable ***q



&q ≡ value(&q)

&(*q) ≡ value(q)

&(**q) ≡ value(*q)

&(***q) ≡ value(**q)

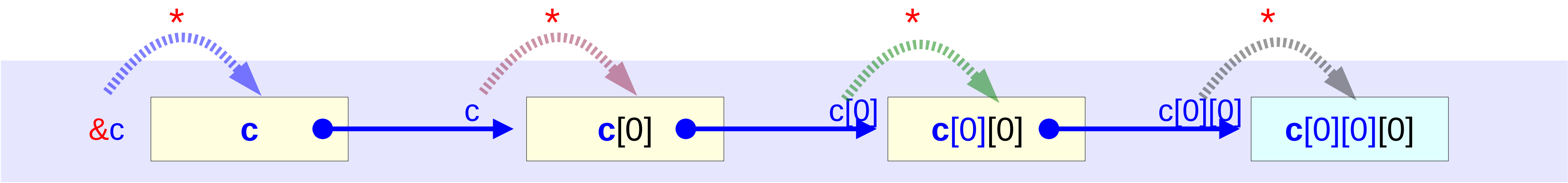
C expression &q
equals to value(&q)
which is the address
value of a variable q

C expression &(*q)
equals to value(q)
which is the address
value of a variable *q

C expression &(**q)
equals to value(*q)
which is the address
value of a variable **q

C expression &(***q)
equals to value(**q)
which is the address
value of a variable ***q

Pointer Chain Type II – * and & operators



*(&c) ≡ c

*(c) ≡ c[0]

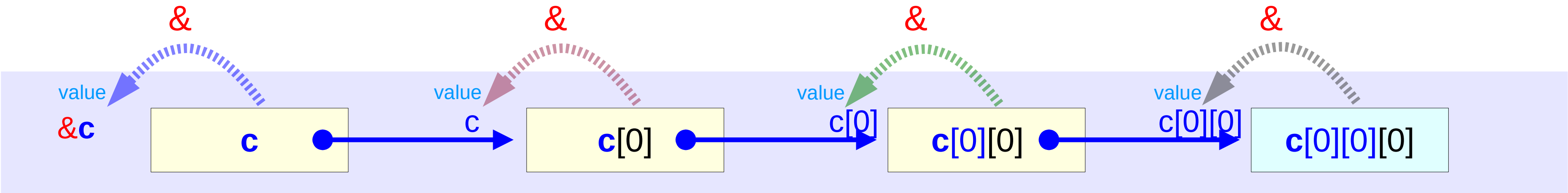
*(c[0]) ≡ c[0][0]

*(c[0][0]) ≡ c[0][0][0]

(int (*)[3][4]) c can be viewed as a pointer to (int [3][4]) c[0]

(int (*)[4]) c[0] can be viewed as a pointer to (int [4]) c[0][0]

(int (*)) c[0][0] can be viewed as a pointer to (int) c[0][0][0]



&c ≡ value(&c)

&(c[0]) ≡ value(c)

&(c[0][0]) ≡ value(c[0])

&(c[0][0][0]) ≡ value(c[0][0])

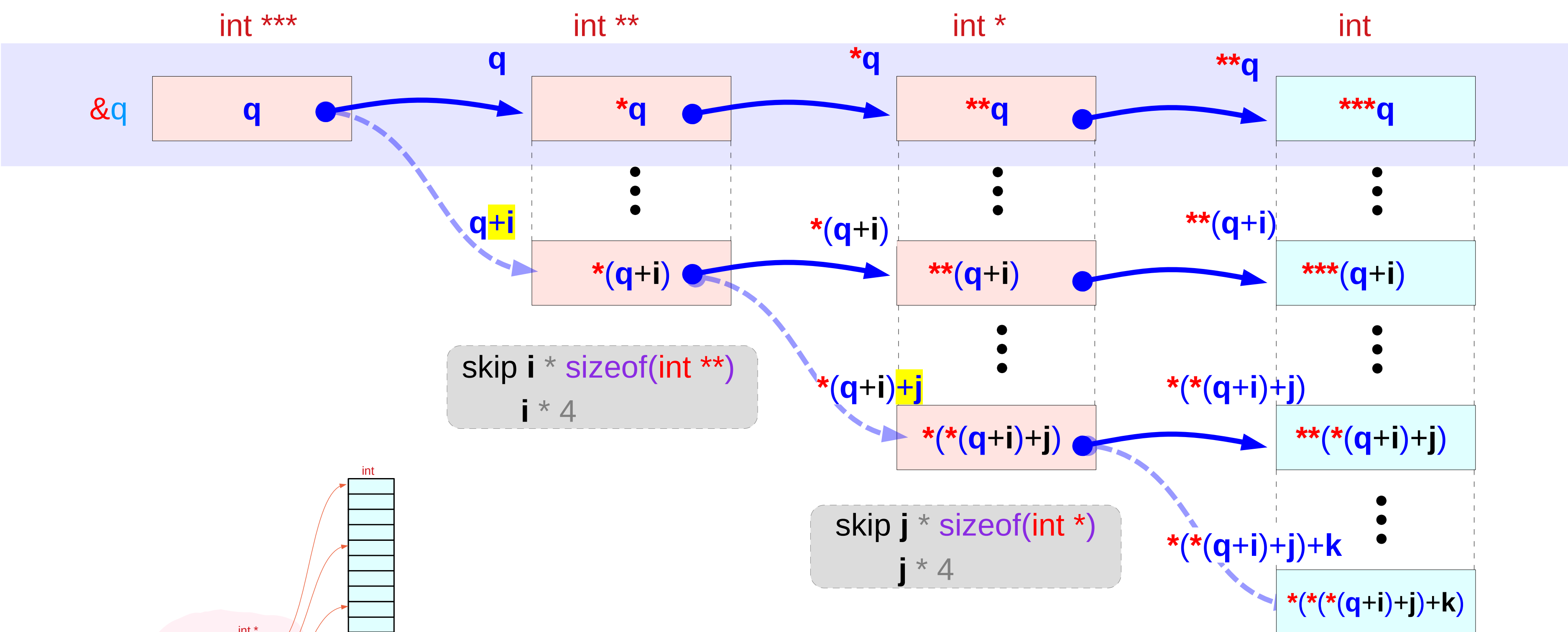
(int (*)[3][4]) c has the address value of (int [3][4]) c[0]

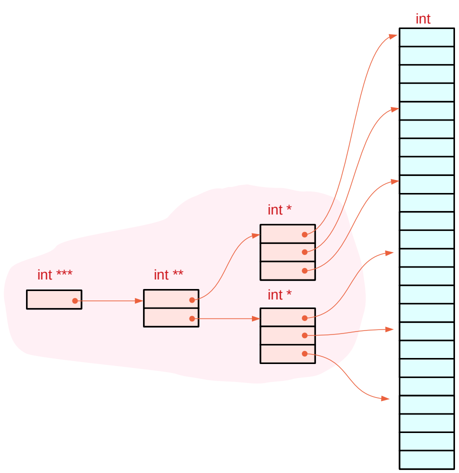
(int (*)[4]) c[0] has the address value of (int [4]) c[0][0]

(int (*)) c[0][0] has the address value of (int) c[0][0][0]

Pointer Chain Type I – skipping elements

Pointer Chain Type I *multiple indirection* *size: pointer size*

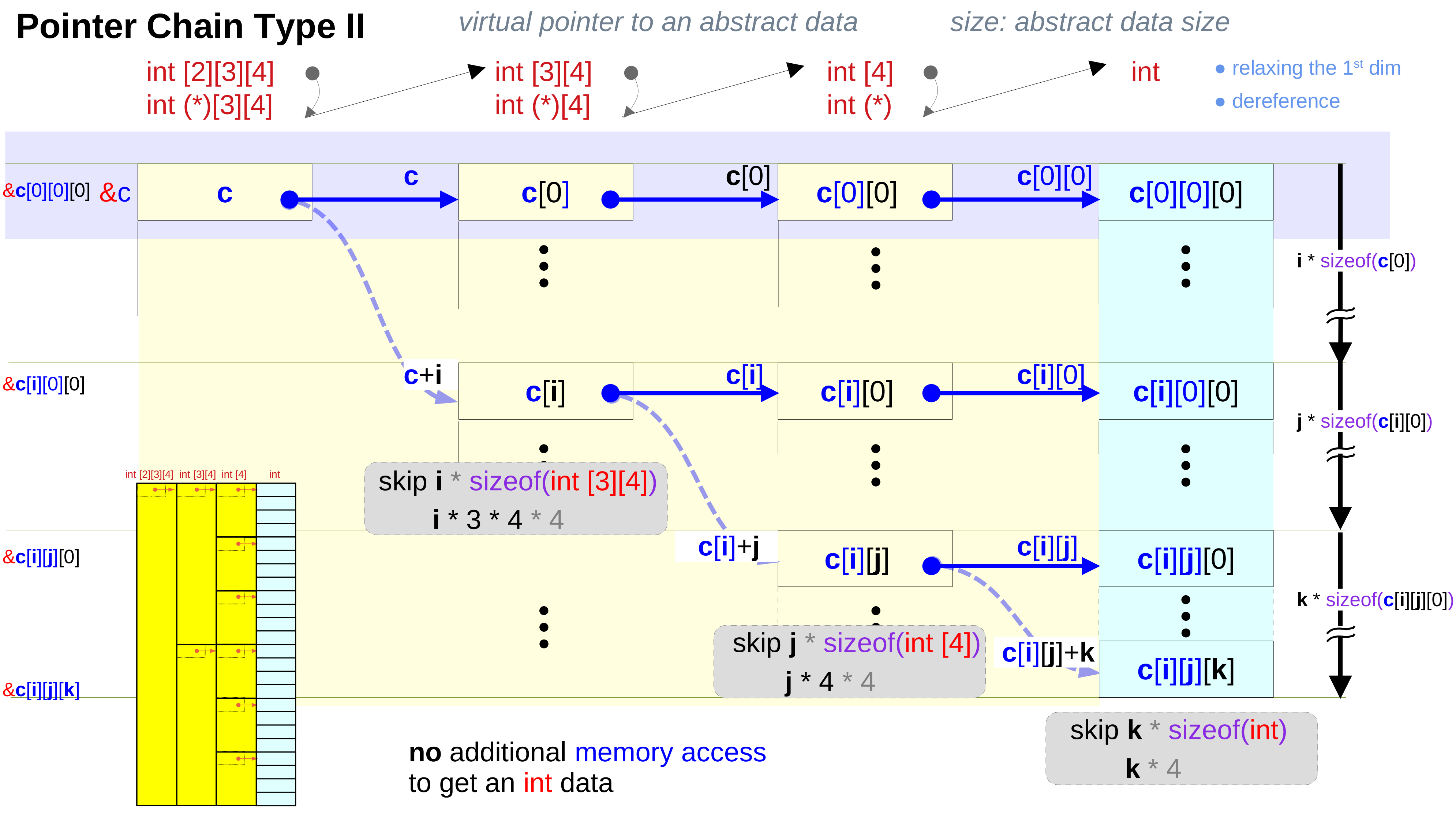




3 additional memory access
to get an int data

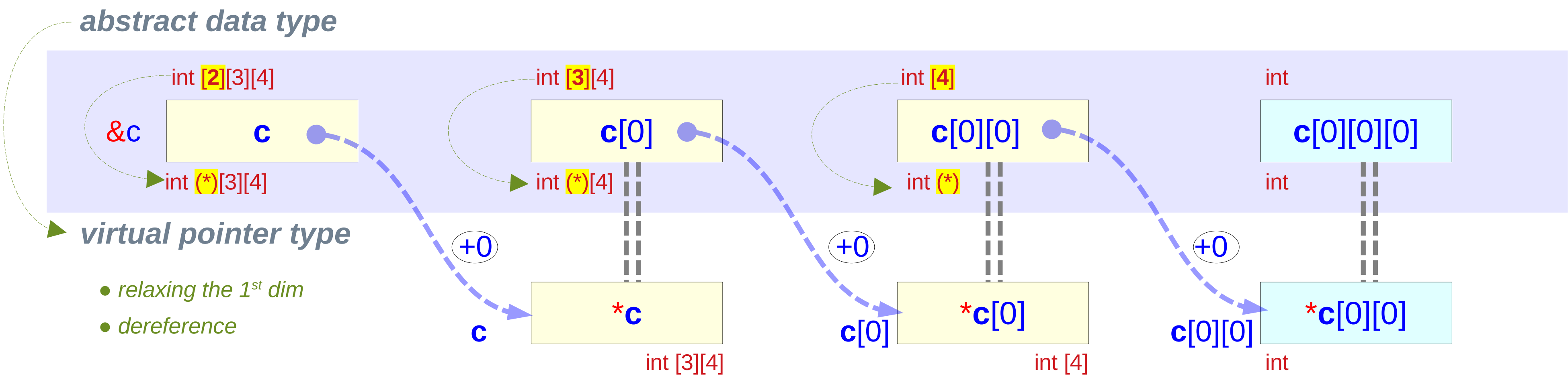
skip **k** * sizeof(int)
k * 4

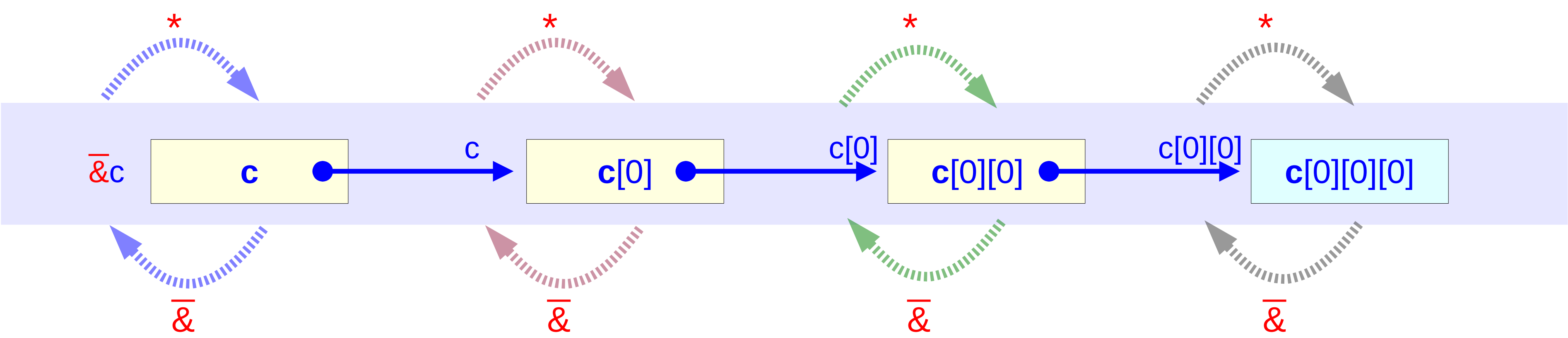
Pointer Chain Type II – skipping elements



no additional memory access
to get an int data

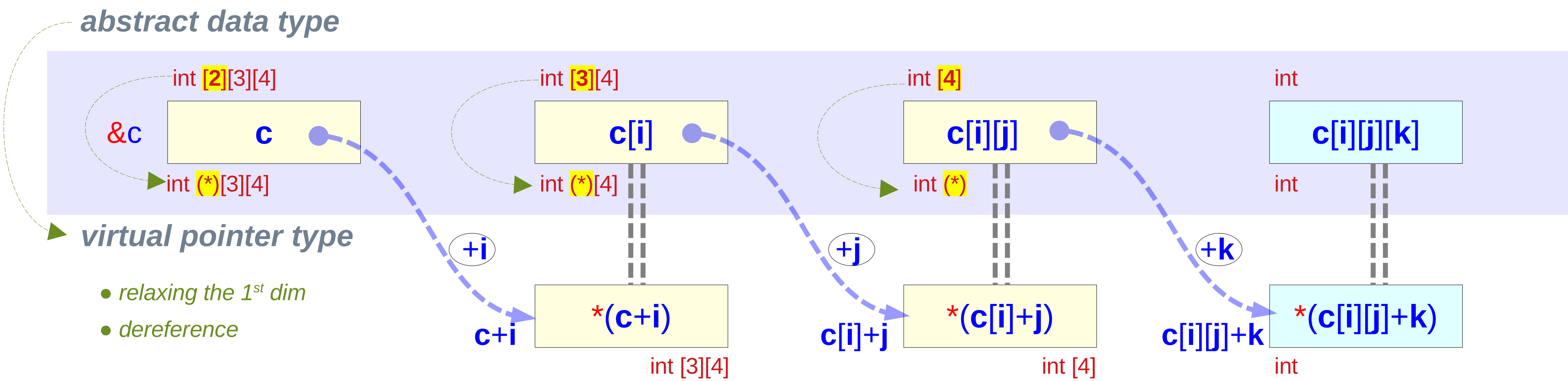
Two step deferencing in type II (1) – without skipping

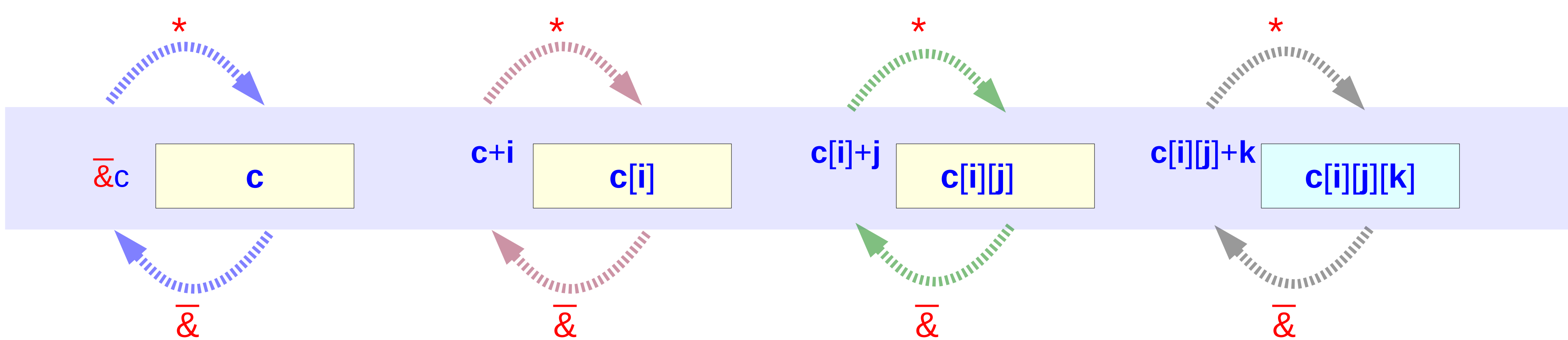




& : **mathematical & operator**

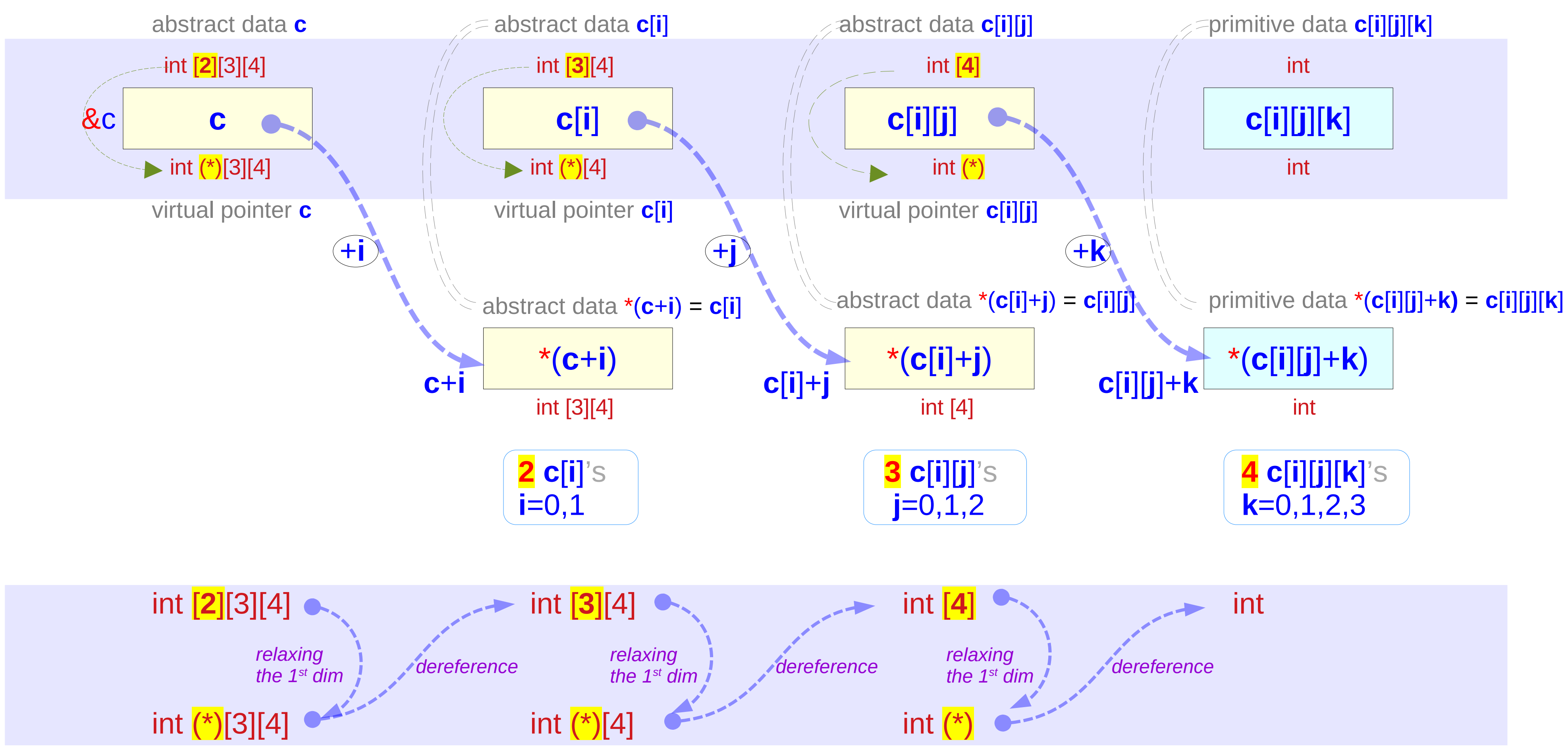
Two step deferencing in type II (2) – with skipping



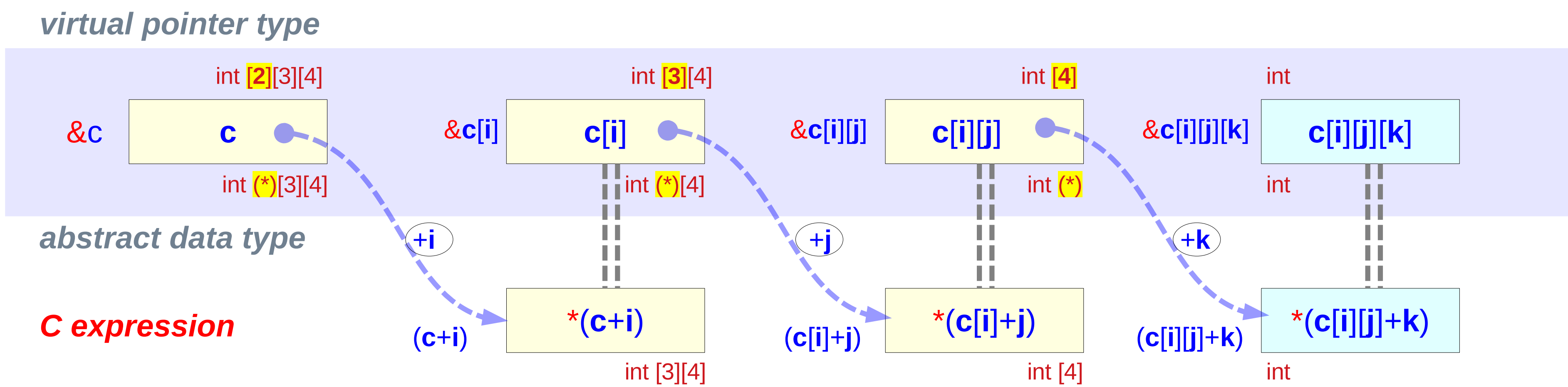


& : mathematical & operator

Two step deferencing in type II (3)



Skipping elements



Math expression

$(c+i)_{3\cdot 4\cdot 4}$ — skip $i\cdot 3\cdot 4$ integer elements from **c**

$(c[i]+j)_{4\cdot 4}$ — skip $j\cdot 4$ integer elements from **c[i]**

$(c[i][j]+k)_{1\cdot 4}$ — skip $k\cdot 1$ integer elements from **c[i][j]**

sizeof(*c) = 3*4*4

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

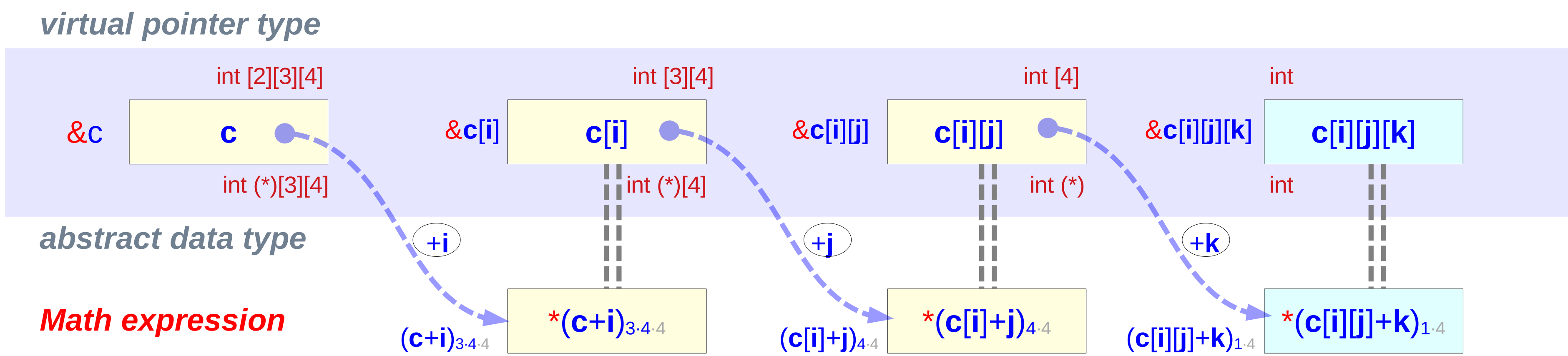
sizeof(*c[i]) = 4*4

value((c[i]+j)$_{4}$)
= value(c[i]) + j *4*4

sizeof(*c[i][j]) = 1*4

value((c[i][j]+k)$_{1}$)
= value(c[i][j]) + k *4

Address replication



equivalence relations – c expressions

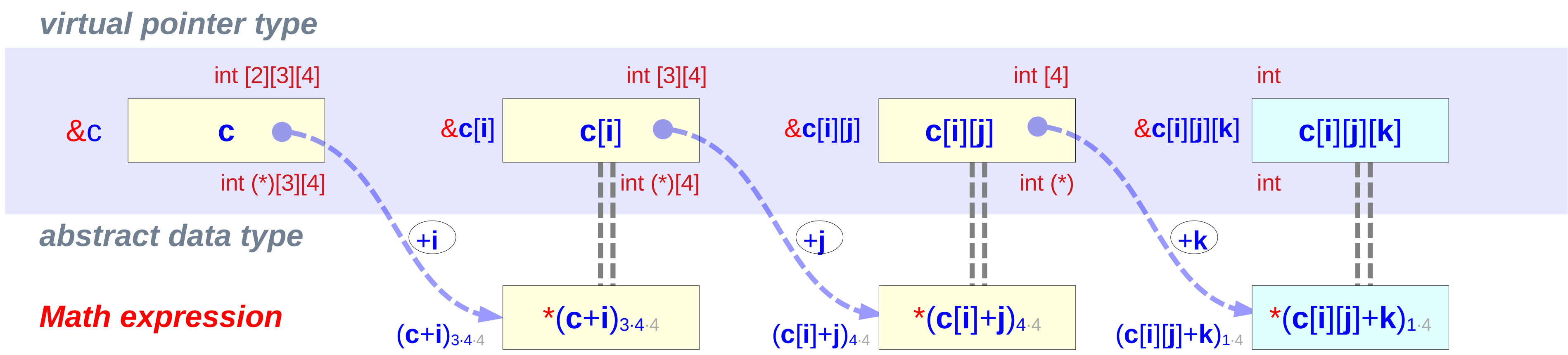
c[i][j][k] = *(c[i][j] + k)
c[i][j] = *(c[i] + j)
c[i] = *(c + i)

&c[i][j][k] = (c[i][j] + k)
&c[i][j] = (c[i] + j)
&c[i] = (c + i)

address replication – math expressions

value(c[i][j]) = value((c[i] + j)$_{4\cdot 4}$)
value(c[i]) = value((c + i)$_{3\cdot 4\cdot 4}$)

Applying address replication



address replication (from value($(c+i)_{3\cdot4\cdot4}$) to value($(c[i]+j)_{4\cdot4}$))

address replication (from value($(c[i]+j)_{4\cdot4}$) to value($(c[i][j]+k)_{1\cdot4}$))

element size
= sizeof(int)
= 4 bytes

value($(c+i)_{3\cdot4\cdot4}$)
= value(c)
+ i *3*4*4

value($(c[i]+j)_{4\cdot4}$)
= value(c[i])
+ j *4*4

value($(c[i][j]+k)_{1\cdot4}$)
= value(c[i][j])
+ k *4

const pointers

const type, const pointer type (1)

const int * p;	⟺	int * p;
int * const q ;	⟺	int * q ;
const int * const r ;	⟺	int * r ;

constant

must not be changed
must not be updated
must not be written
must not be assigned

const type, const pointer type (2)

```
const int * p ;
```
constant integer

```
int * const q ;
```
constant pointer

```
const int * const r ;
```
constant integer

```
const int * const r ;
```
constant pointer

*p : constant integer value

q : constant (int *) pointer

*r : constant integer value

r : constant (int *) pointer

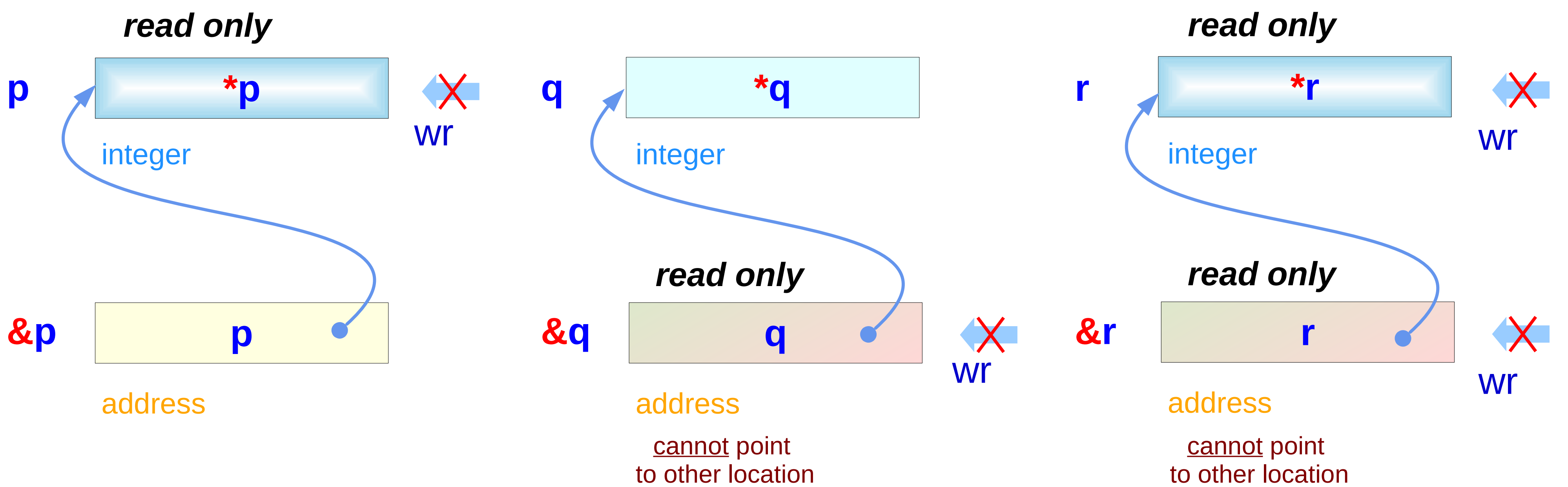
const [ ]
group with the following

const type, const pointer type (3)

const int *p;

int * const q ;

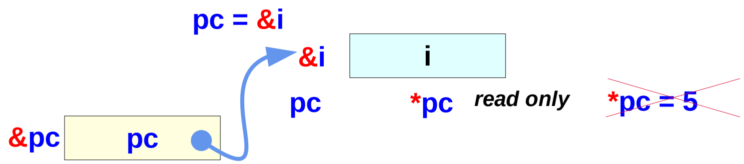
const int * const r ;



const examples (1)

```
const int * pc;
      int * p, i;
const int   ic;
```

```
 pc = &i;     // (const int *) ← (int *)
*pc  = 5;     // (const int) error
```



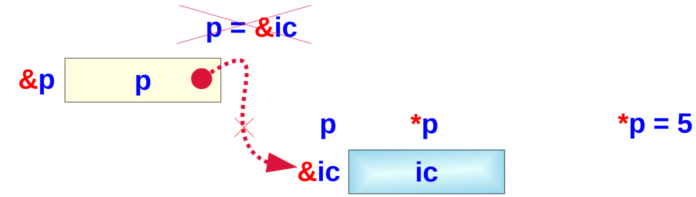
pc can point to **i**
***pc** must be **const**

the same memory location
that can be written via **i**
cannot be written via ***pc**

***pc** should not write
the writable memory location

Writing to the writable memory location (**i**)
is forbidden via **pc** … (no harm, OK)

```
 p  = &ic;    // (int *) ← (const int *) warning
*p  = 5;      // (int)
```



Assume **p** points to **const ic**

the same memory location
that cannot be written via **ic,**
can be written via ***p**

thus ***p** can write
the **const** memory location

therefore, **p** should not point to **const ic**

Writing to the read only memory location (**ic**)
is not forbidden via **p** … (hazardous, not OK)

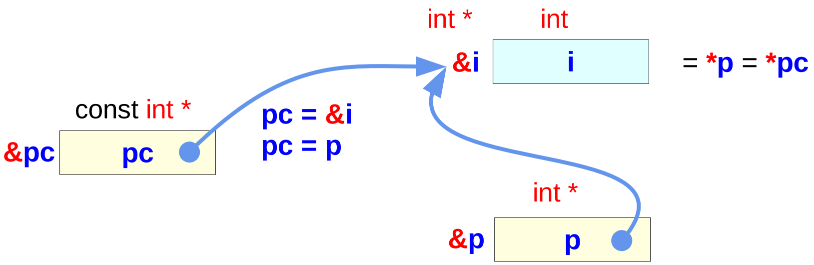
C A Reference Manual, Harbison & Steele Jr.

const examples (2)

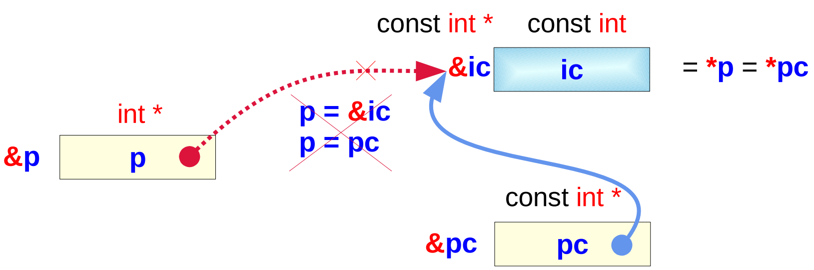
```
const int * pc;
        int * p, i = 5;
const int   ic = 7;
```

```
p   = &i;
pc  = &ic
```

```
// more constrained type ← general type (O)
pc  = &i;    // (const int * ←  int *)
pc  =  p;    // (const int * ←  int *)
```



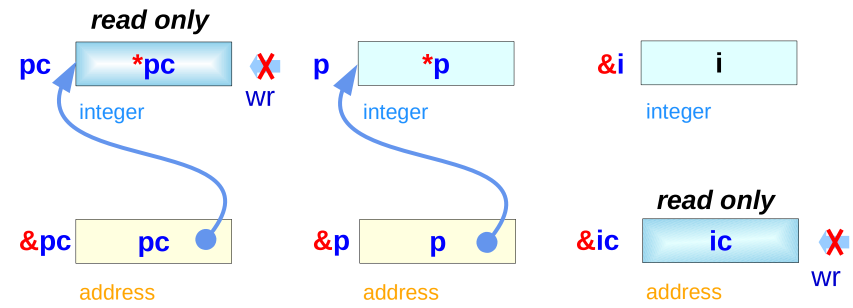

```
// general type ← more constrained type (X)
p   = &ic;   // (int * ← const int *)  warning
p   =  pc;   // (int * ← const int *)  warning
```



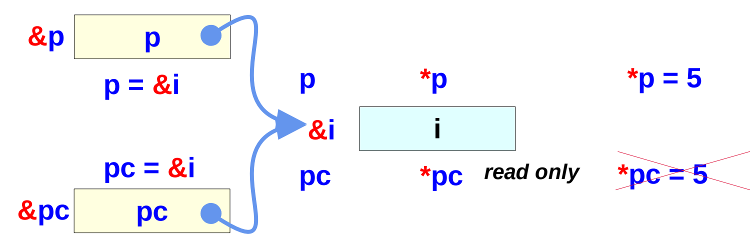
C A Reference Manual, Harbison & Steele Jr.

const examples (3)

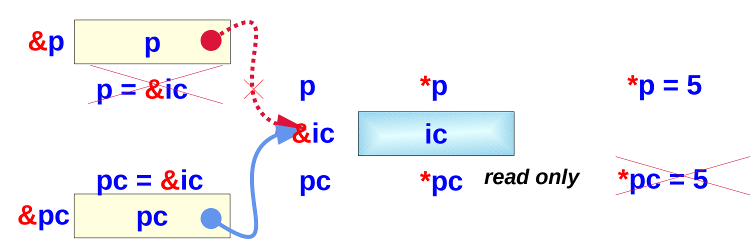
```
const int * pc;
      int * p, i;
const int   ic;
```




```
 p  = &i;    // (int *) ← (int *)
*p  = 5;     // (int)
 pc = &i;    // (const int *) ← (int *)
*pc = 5;     // (const int) error
```



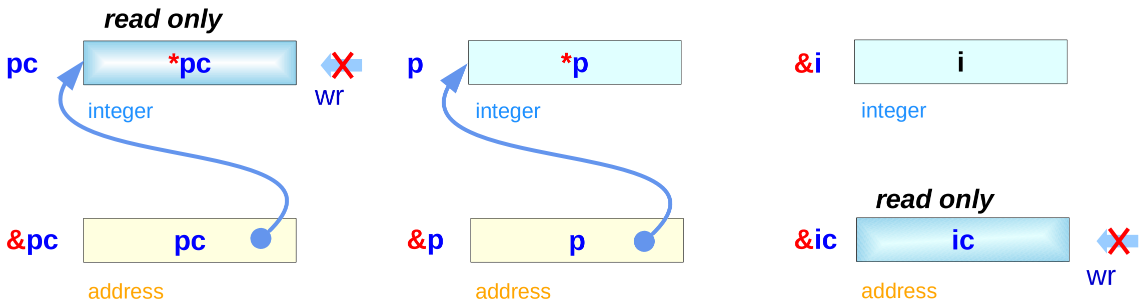

```
 p  = &ic;   // (int *) ← (const int *) warning
*p  = 5;     // (int)
 pc = &ic;   // (const int *) ← (const int *)
*pc = 5;     // (const int) error
```



C A Reference Manual, Harbison & Steele Jr.

const examples (4)

```
const int * pc;
      int * p, i;
const int   ic;
```




```
 pc  =   p = &i;
 pc  = &ic
*p   =   5;
*pc  =   5;        // invalid      *pc :: cons int

pc   = &i;         //              (const int * ←  int *)
pc   =  p;         //              (const int * ←  int *)
p    = &ic;        // invalid      (int * ← const int *)
p    =  pc;        // invalid      (int * ← const int *)
p    = (int *) &ic;  // type cast
p    = (int *)  pc;  // type cast
```

C A Reference Manual, Harbison & Steele Jr.

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