Applications of Pointers (1A)

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

Young Won Lim
9/23/23

2-d array access

Array of Pointers

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

```
int     a     [4]
```

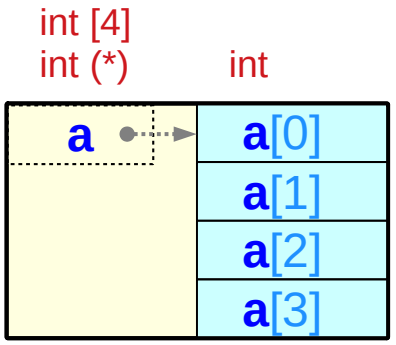
there are 4 elements

the type of each element:
an integer

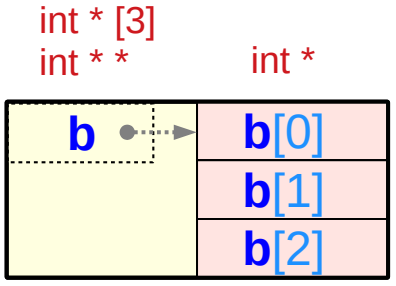
```
int *   b     [3]
```

there are 3 elements

the type of each element:
a pointer an integer



Integers

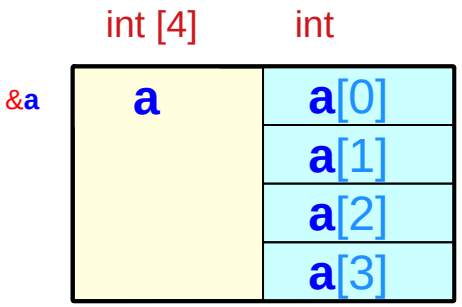


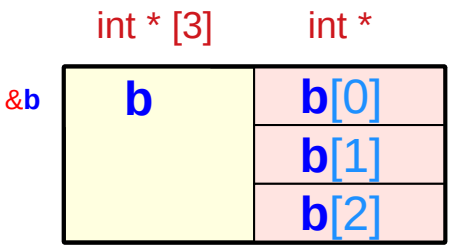
Integer pointers

Array of Pointers – a type view

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

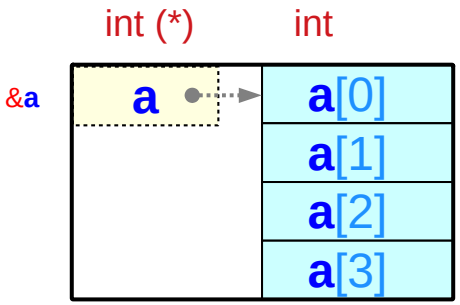
```
int *   b [3] ;
```

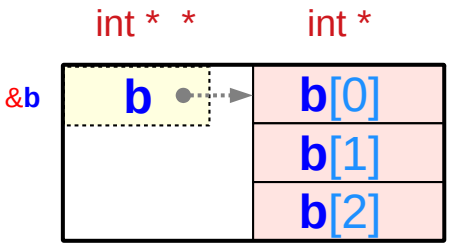




Integers

Integer
pointers

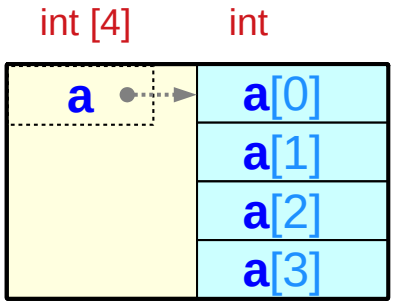




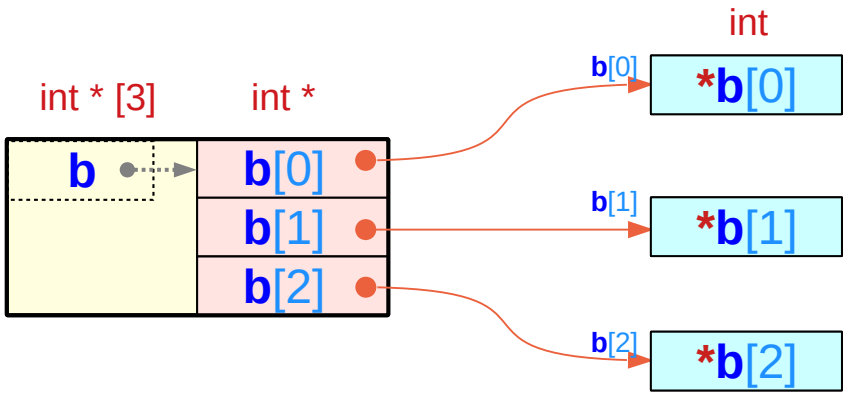
Array of Pointers – a variable view

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

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



Integers



Integer
pointers

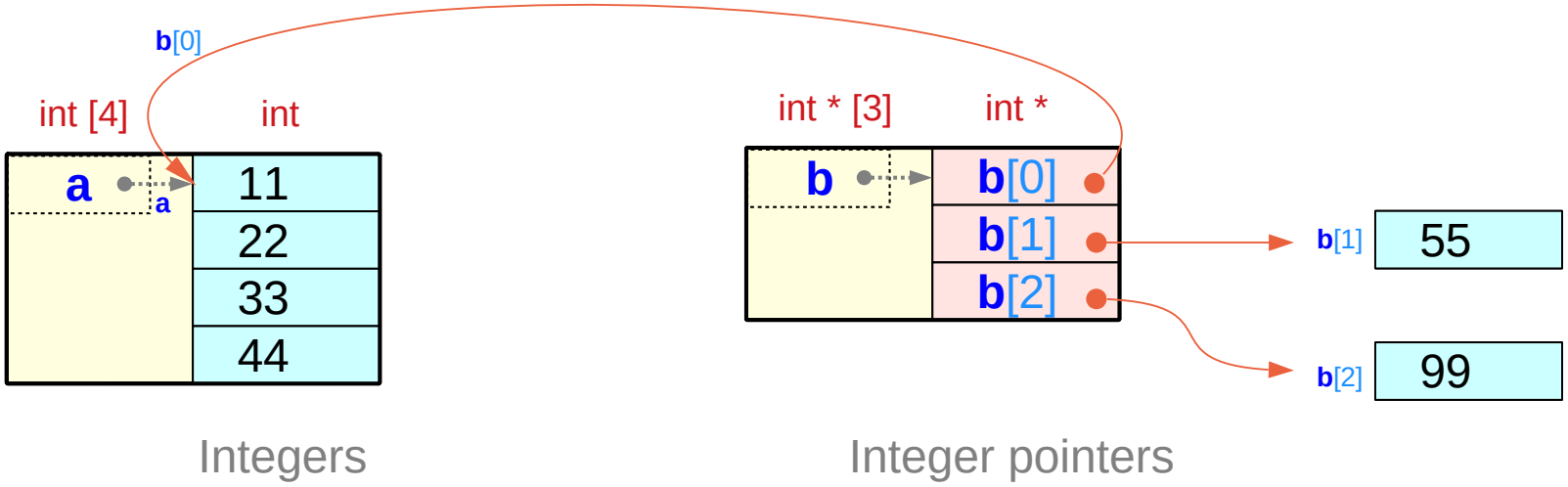
Assigning a **1-d** array name

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

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

assignment

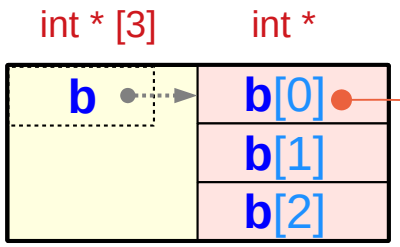
b[0] = &**a**[0] (= **a**)



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

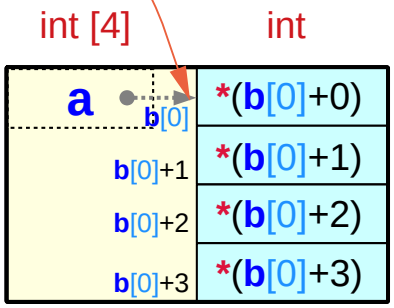
```
int *   b [3] ;
```

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



assignment

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



Array of Pointers – extended dimension

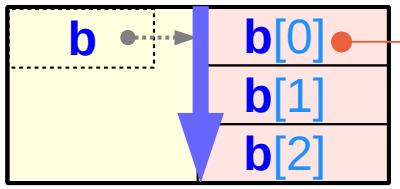
```
int *   b [3] ;
```

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

```
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)
```

array name b

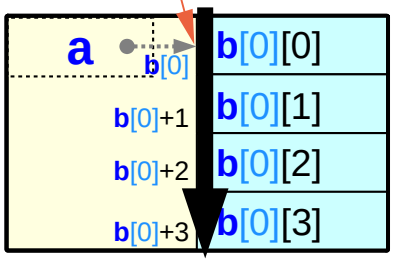
int * [3] int *



1st dim

array name b[0]

int [4] int

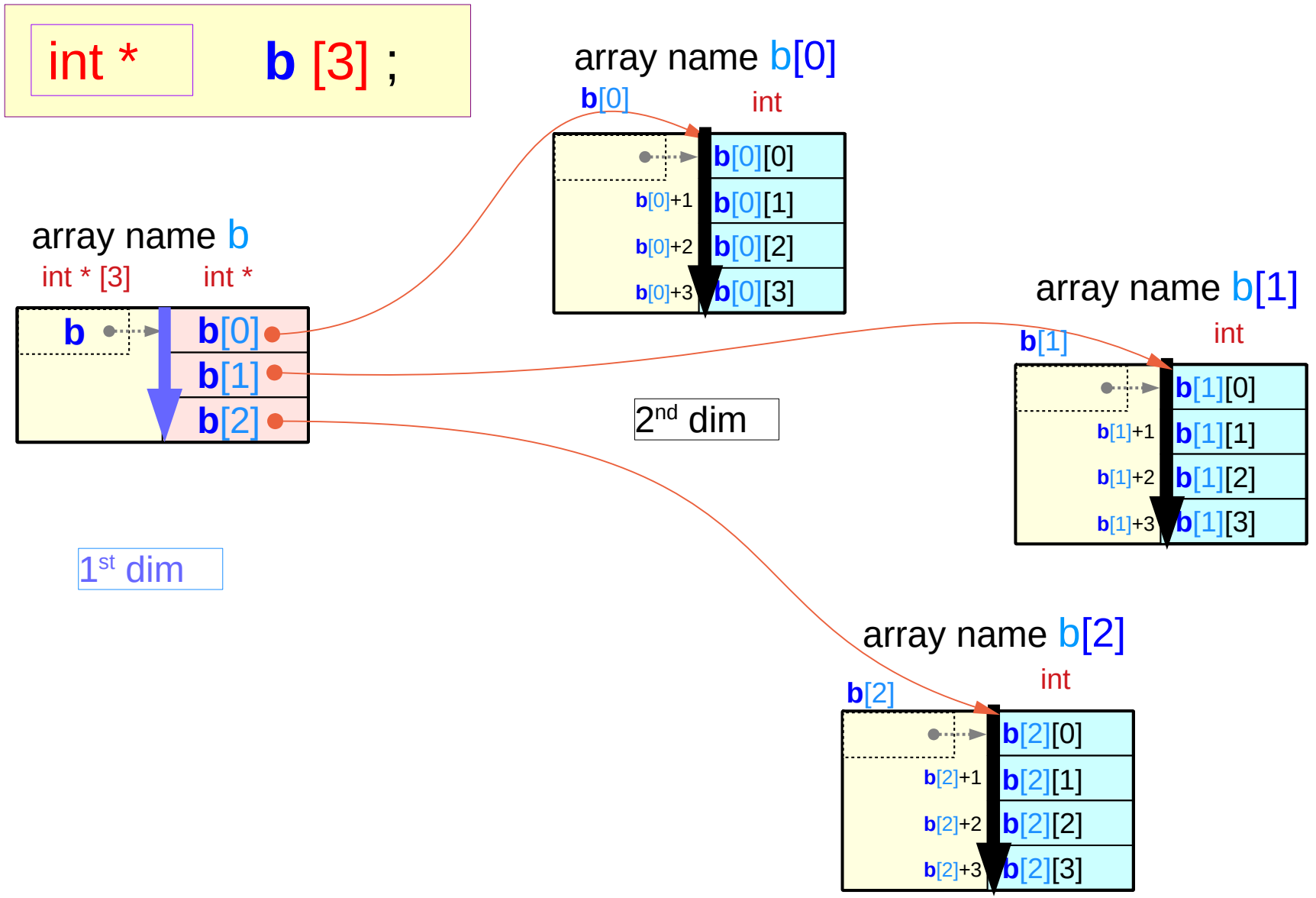


2nd dim

assignment

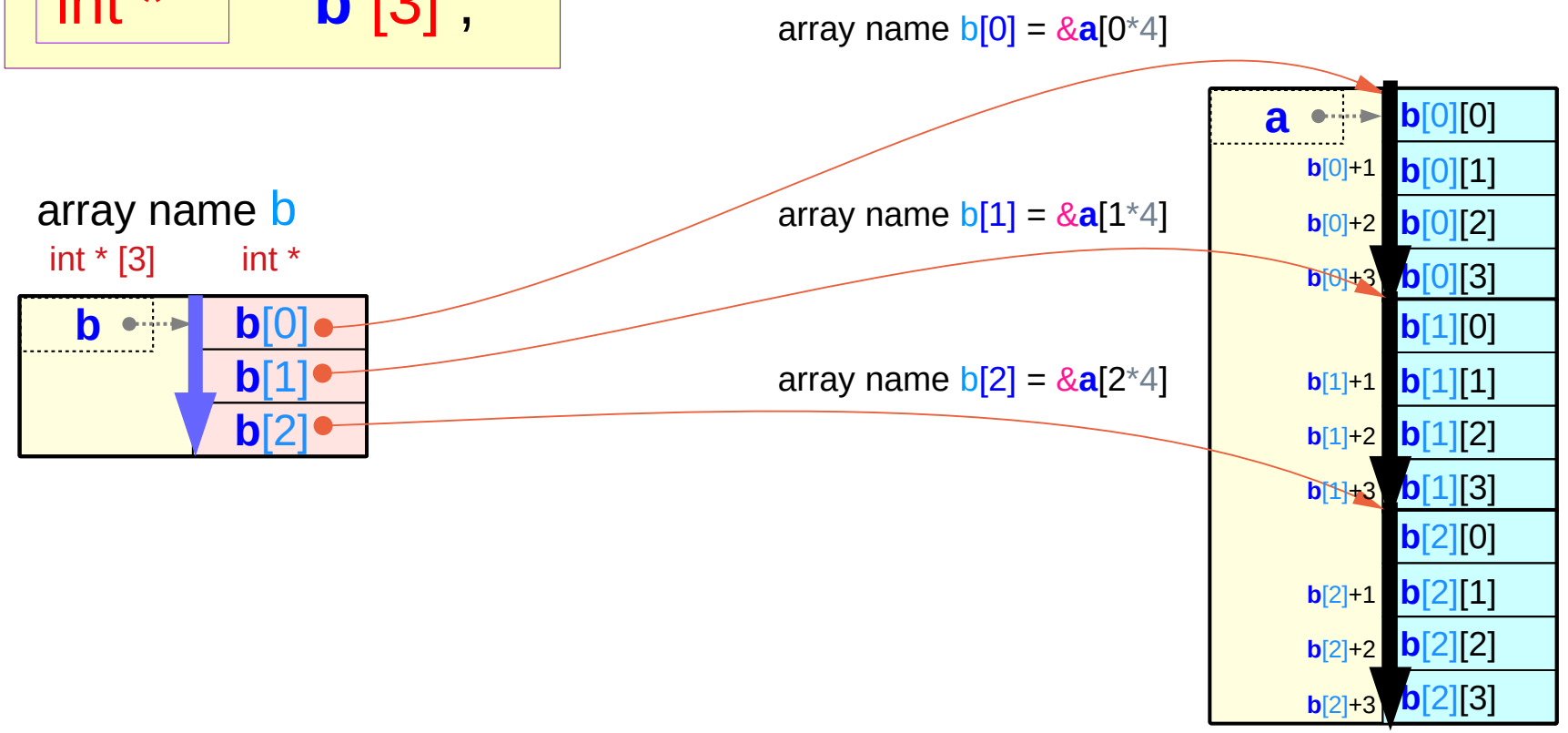
b[0] = &**a**[0] (= **a**)

2-d access of 1-d arrays



2-d access of a 1-d array

int * b [3] ;



int * a [3*4] ;

2-d access of a 1-d array – pointer array assignments

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

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

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

Assignments

```
b[0] = &a[0*4]  (= a +0*4)
b[1] = &a[1*4]  (= a +1*4)
b[2] = &a[2*4]  (= a +2*4)
```

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

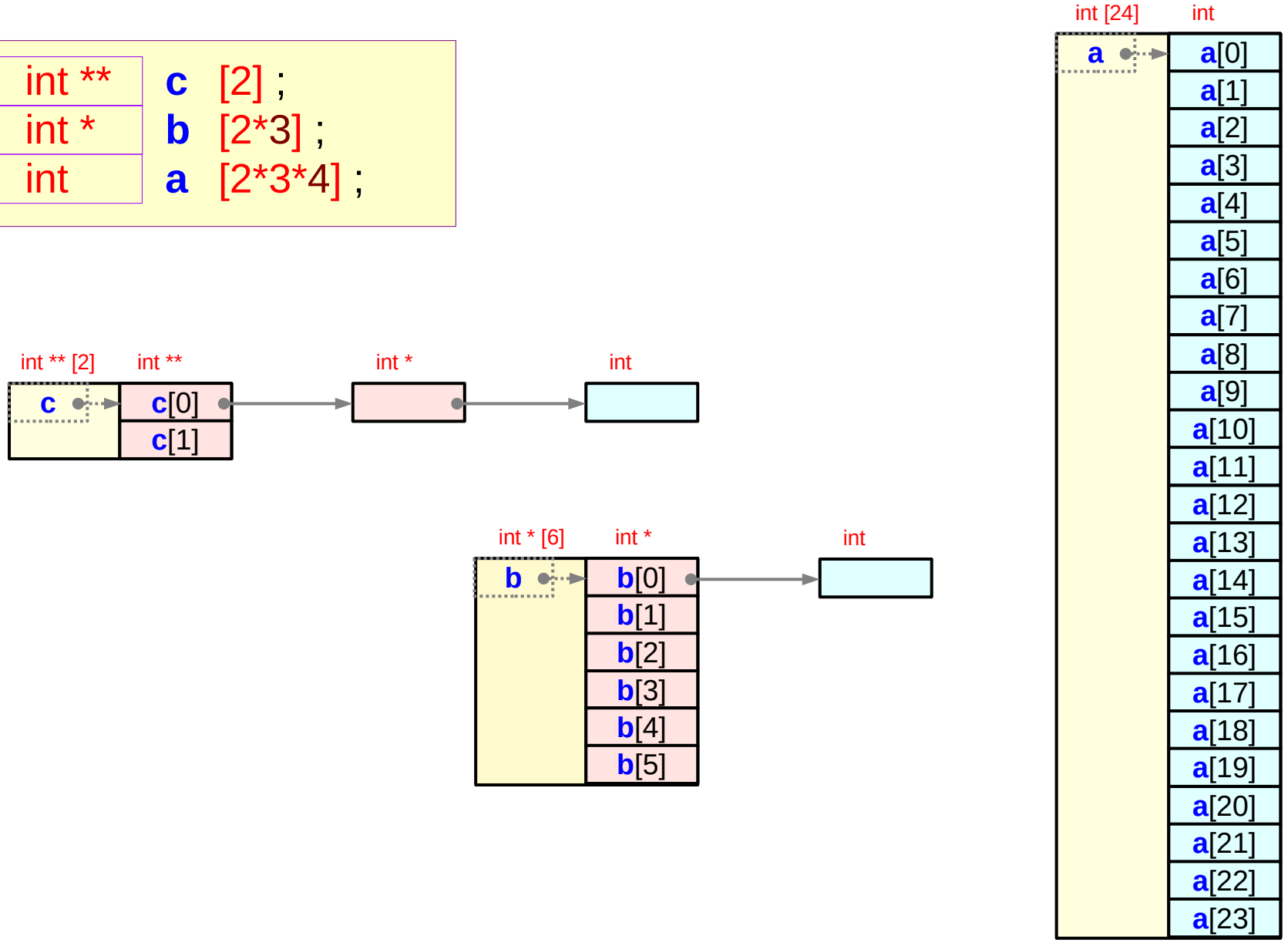
constraint : contiguous a[i*4+j] over j

*(b+i) = a+*f*(*i*)

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

Using pointer arrays **b**, **c**

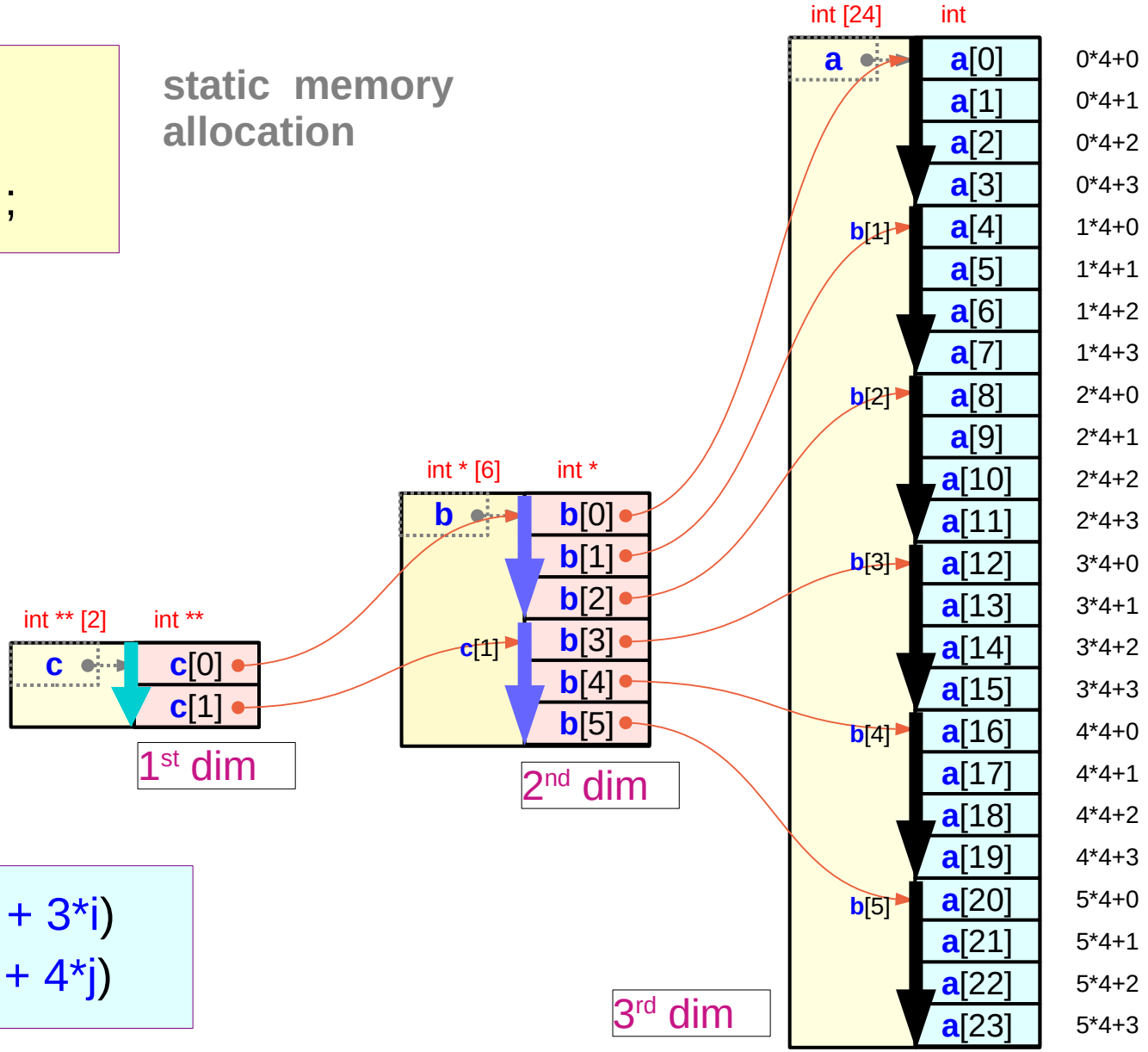
int **	**c** [2] ;
int *	**b** [2*3] ;
int	**a** [2*3*4] ;



Using static memory allocation

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

static memory allocation



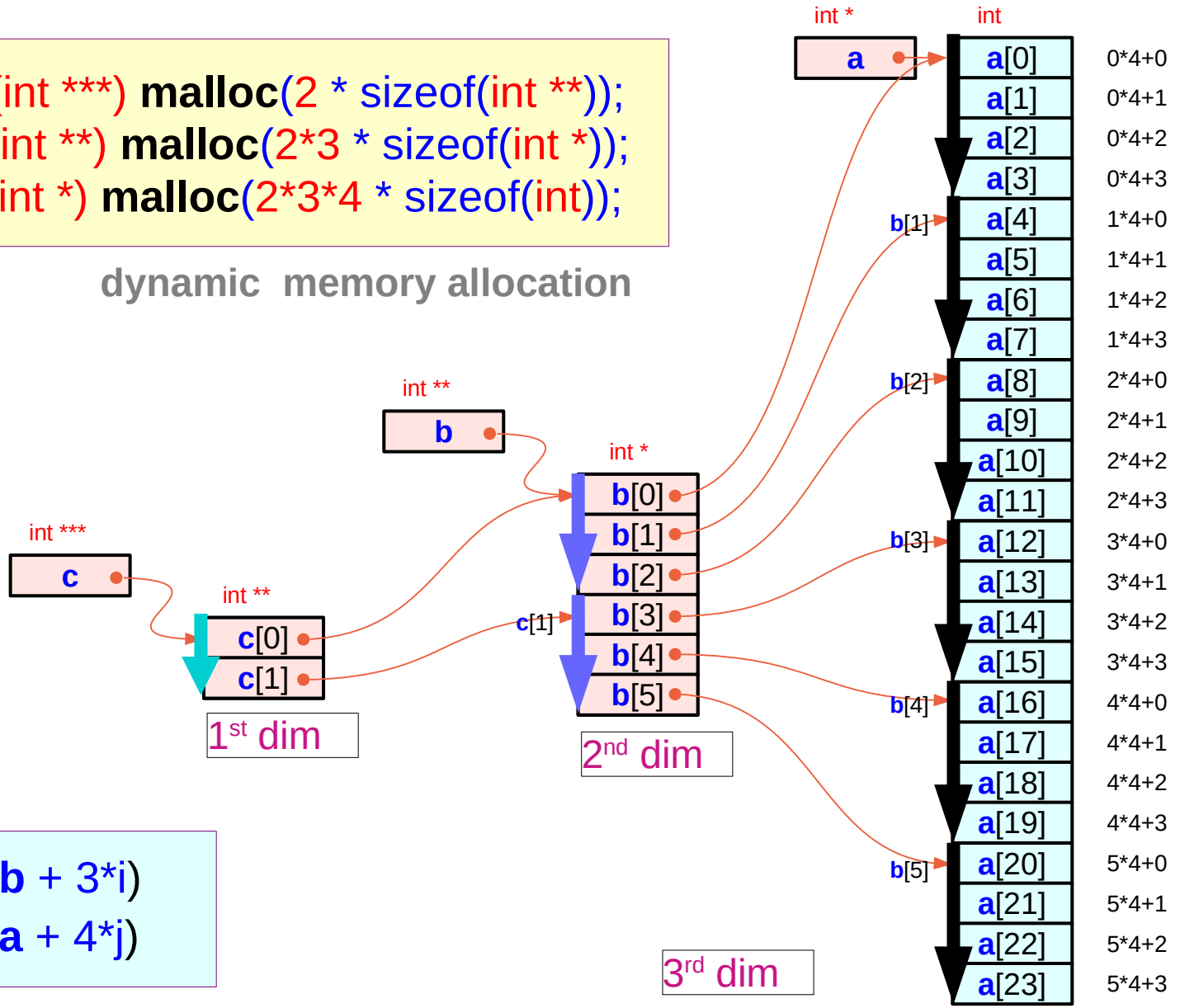
c[i] = &**b**[3*i] (= **b** + 3*i)

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

Using dynamic memory 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));
```

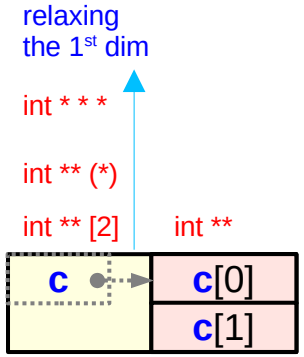
dynamic memory allocation




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

Static v.s. dynamic memory allocation (1)

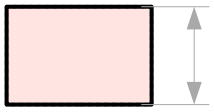
int *** **c** = (int ***) **malloc**(2 * sizeof(int **));

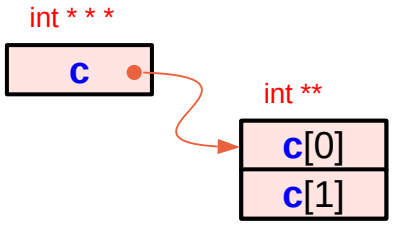


int ** **c** [2] ;

static memory allocation

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



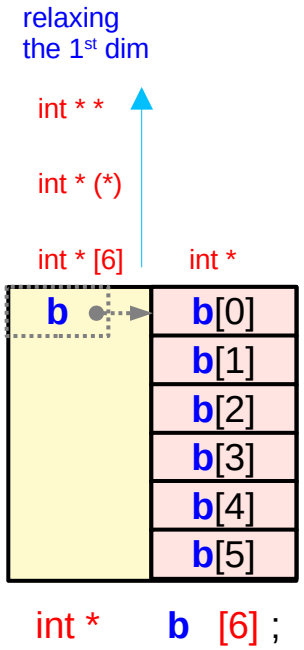


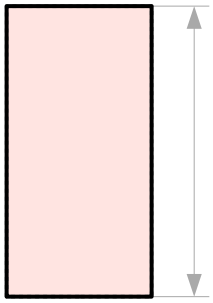
int *** **c** = (int ***) **malloc**(2 * sizeof(int **));

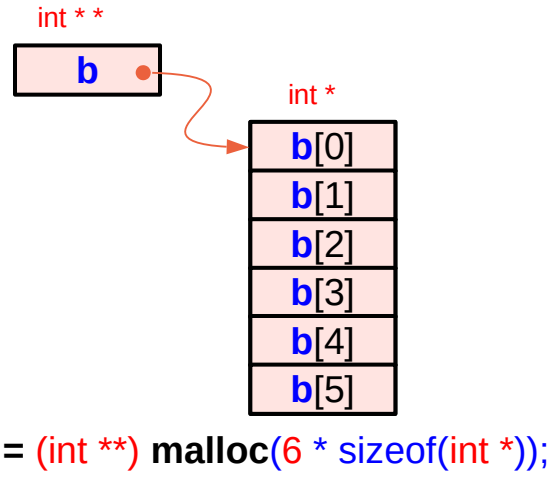
dynamic memory allocation

Static v.s. dynamic memory allocation (2)

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



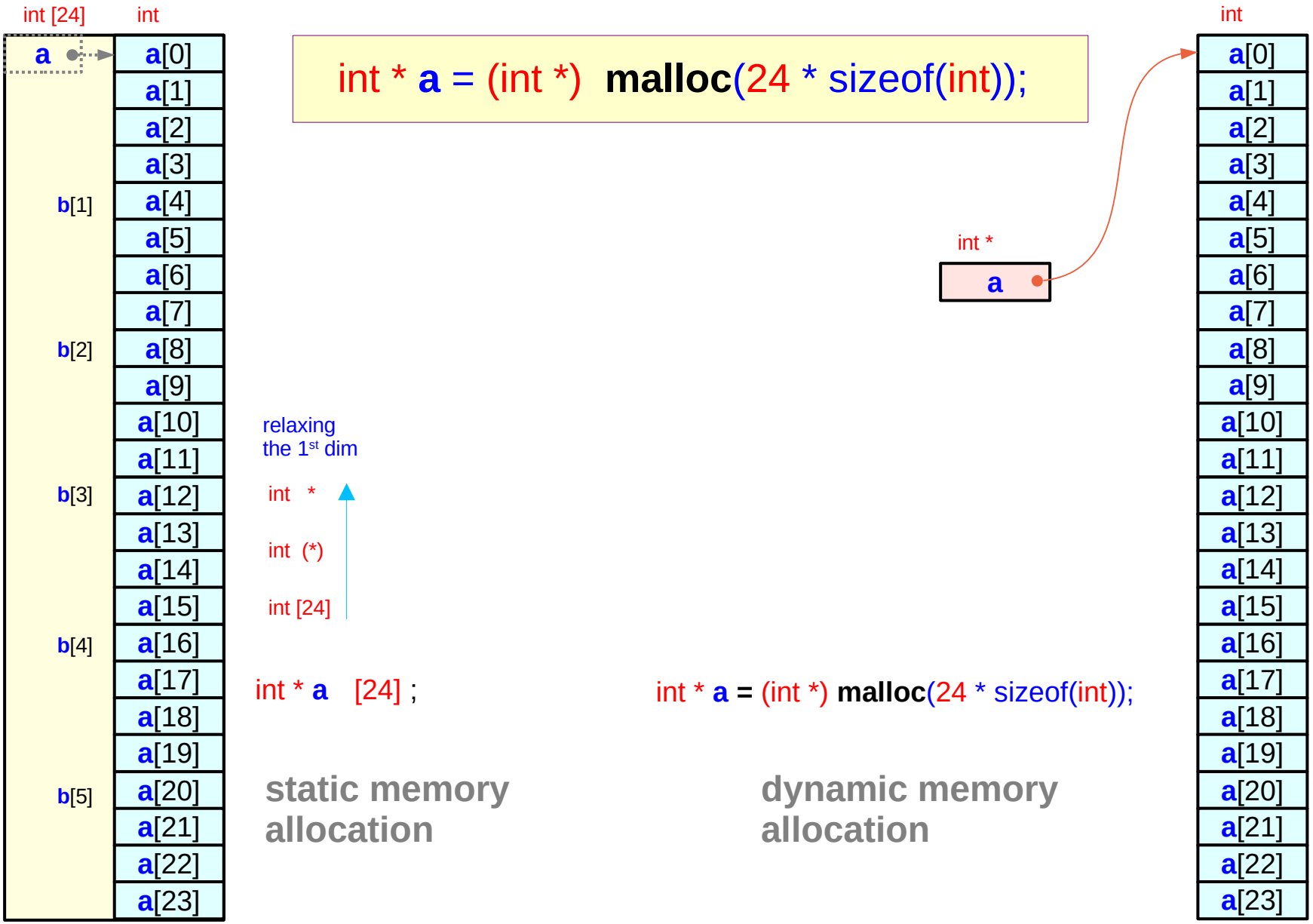




static memory allocation

dynamic memory allocation

Static v.s. dynamic memory allocation (3)



Static v.s. dynamic memory allocation (4)

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

static memory allocation

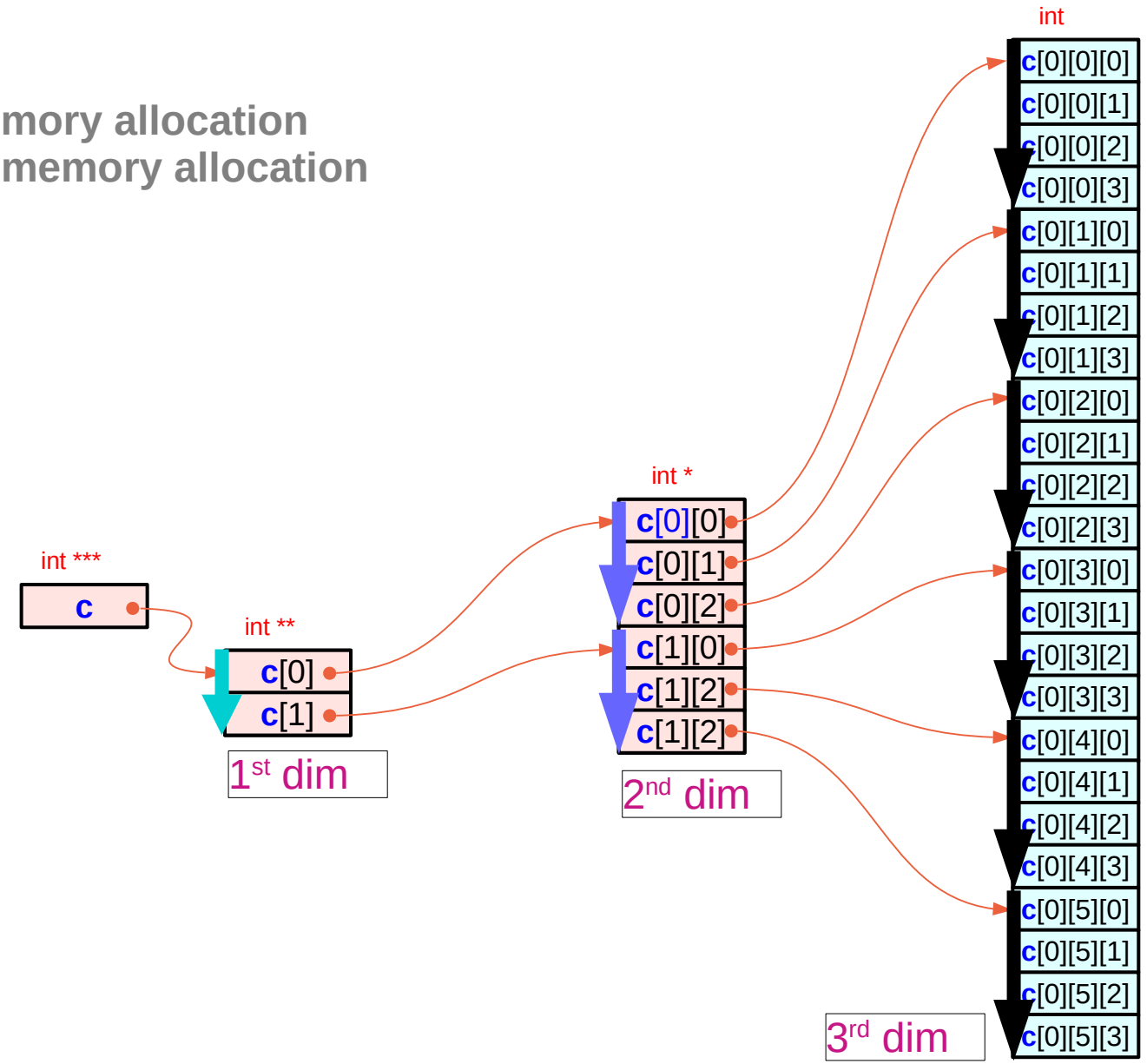
int ***	**c**	= (int ***) **malloc**(2 * sizeof(int **));
int **	**b**	= (int **) **malloc**(2*3 * sizeof(int *));
int *	**a**	= (int *) **malloc**(2*3*4 * sizeof(int));

dynamic memory allocation

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

Static v.s. dynamic memory allocation (5)

- static memory allocation
- dynamic memory allocation



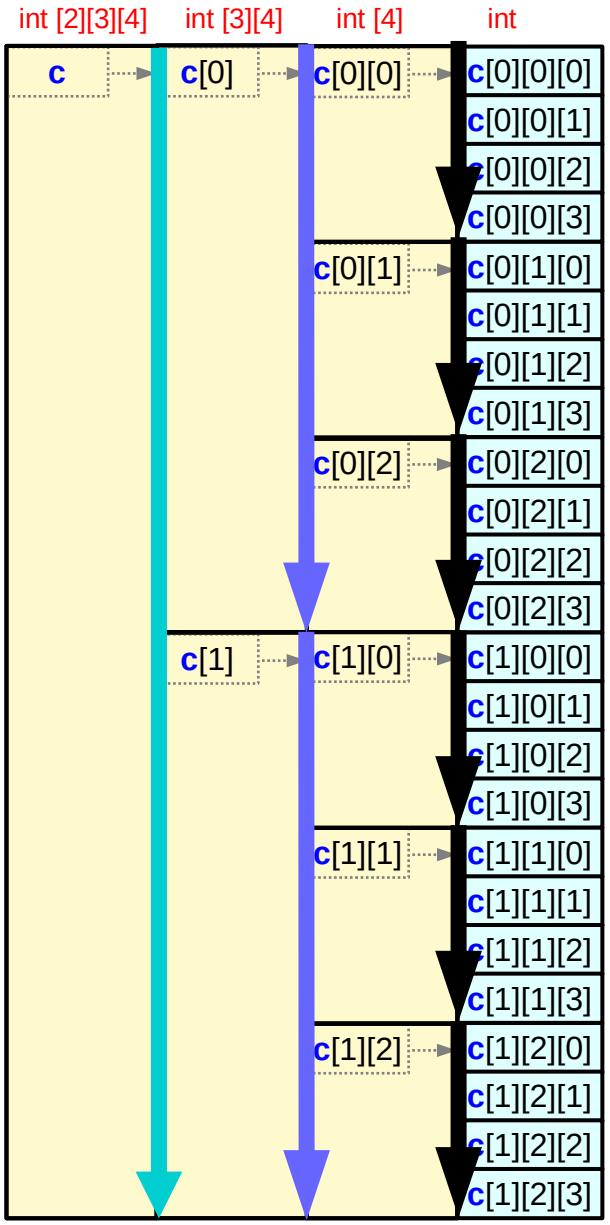
Static memory allocation of an **3-d** array

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

static memory 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][1][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][1][0]

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

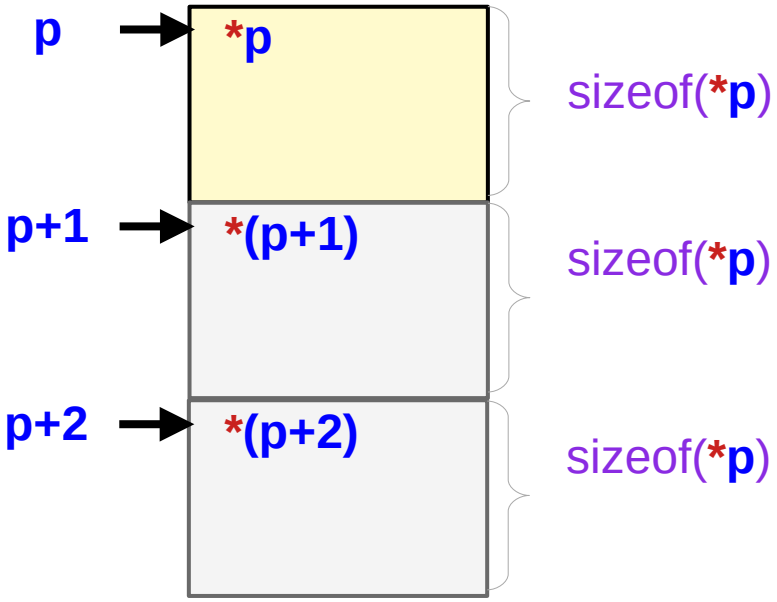


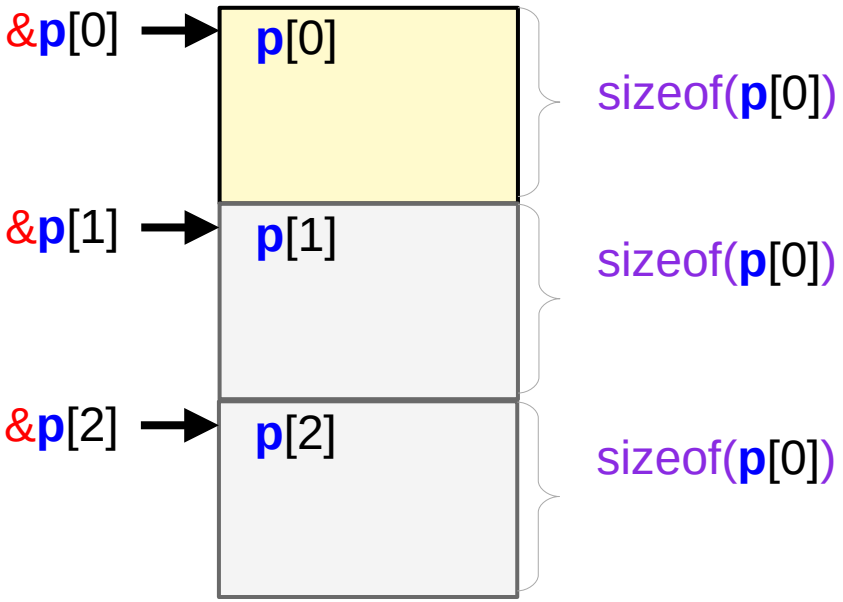
Finding sub-array sizes

```
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))
```

Byte addresses in an array





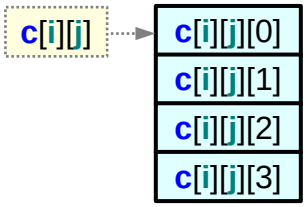
value(**p**+1) = value(**p**) + 1 * sizeof(***p**)
value(**p**+2) = value(**p**) + 2 * sizeof(***p**)

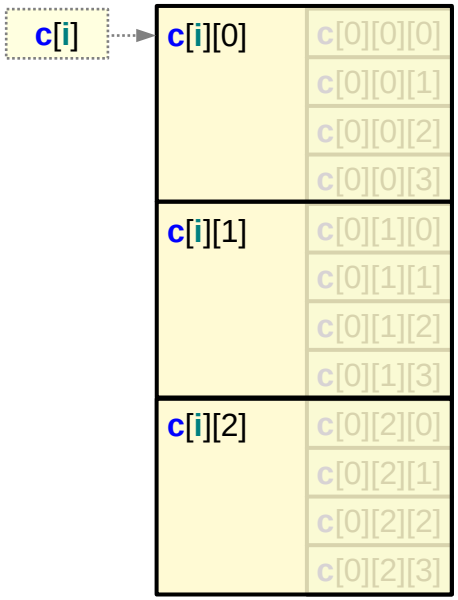
byte address byte address byte size

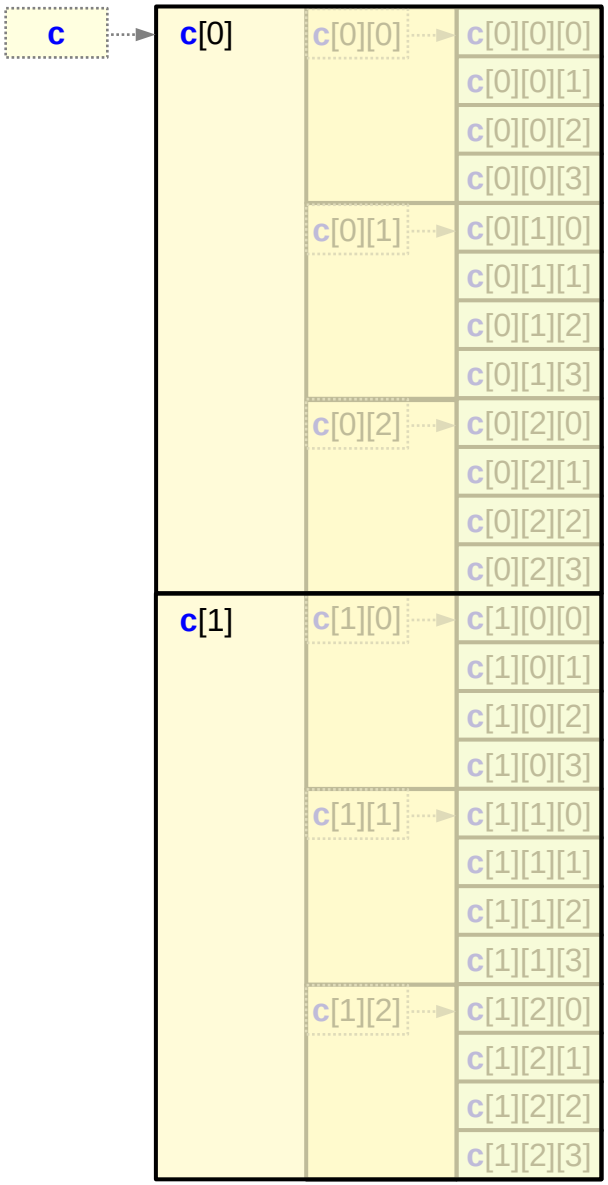
value(&**p**[1]) = value(**p**) + 1 * sizeof(**p**[0])
value(&**p**[2]) = value(**p**) + 2 * sizeof(**p**[0])

byte address byte address byte size

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







value(&c[i][j][k]) **k** = 0:3
= 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)

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

value(&c[i]) **i** = 0 : 1
= value(c) + **i** * sizeof(*c)
= value(c) + **i** * sizeof(c[0])
= value(c) + **i** *sizeof(int) * 3 * 4

Abstract and byte addresses of sub-arrays

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

abstract address *type independent*

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

byte address *type dependent*

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

Values of &c[i][j], c[i][j], and &c[i], c[i]

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

Byte address

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

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

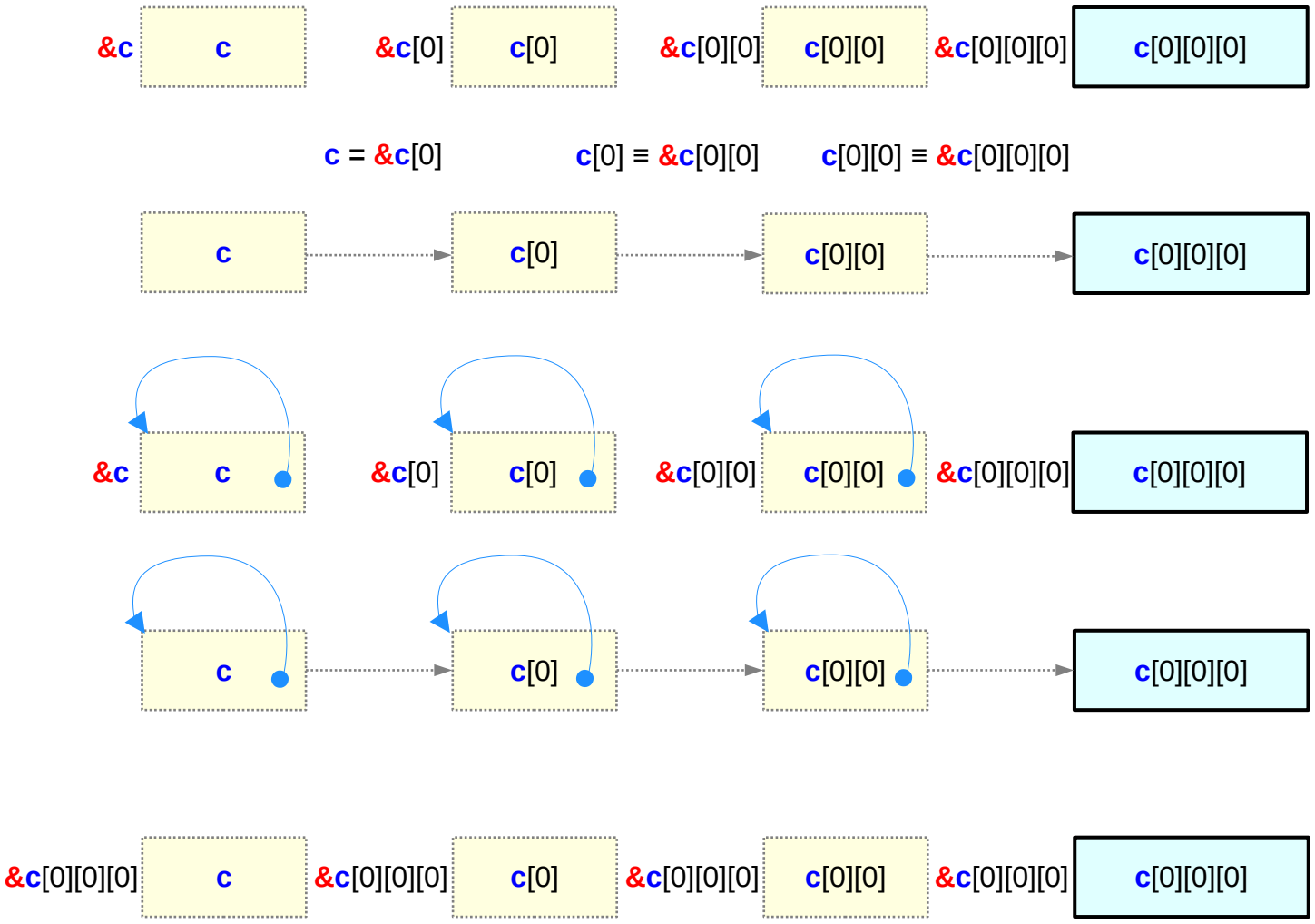
value(&c[i]) = value(c) + i * 3 * 4 * sizeof(int)

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

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

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

Virtual pointers – subarray names **c**, **c**[0], **c**[0][0]



Byte addresses of sub-arrays in an array

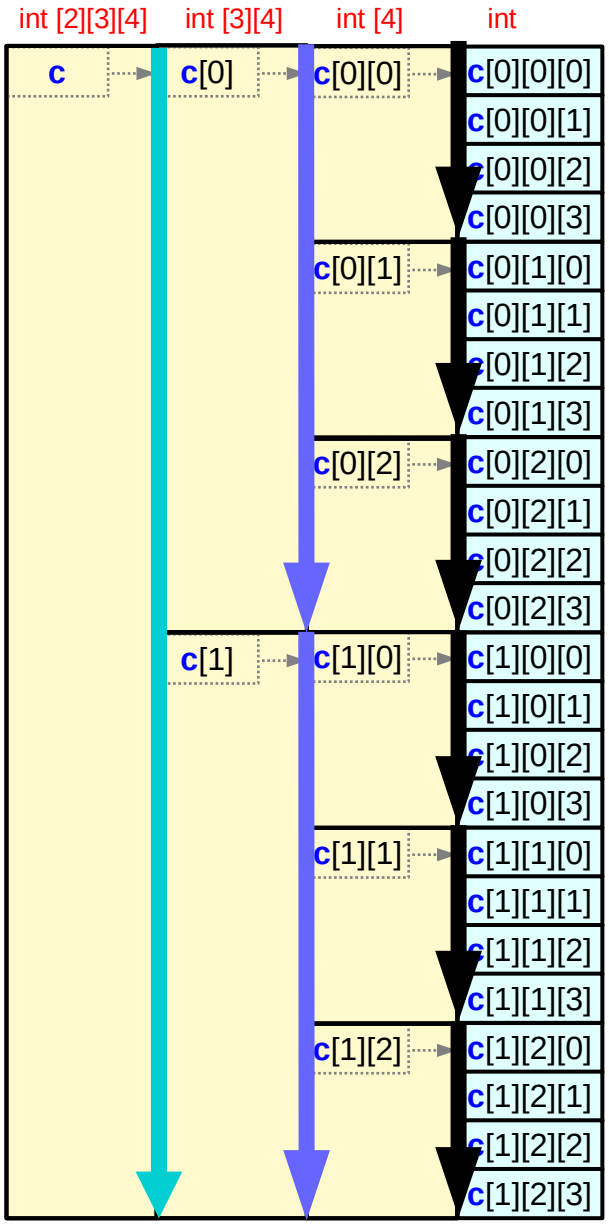
value(**c**) = value(**c**[0]) = value(**c**[0][0]) = value(&**c**[0][0][0])
value(**c**[0][1]) = value(&**c**[0][1][0])
value(**c**[0][2]) = value(&**c**[0][1][0])
value(**c**[1]) = value(**c**[1][0]) = value(&**c**[1][0][0])
value(**c**[1][1]) = value(&**c**[1][1][0])
value(**c**[1][2]) = value(&**c**[1][1][0])

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

new conditions
value(**c**[**i**][**j**]) = value(&**c**[**i**][**j**])
value(**c**[**i**]) = value(&**c**[**i**])
value(**c**) = value(&**c**)

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

virtual pointers



Address values of &c[i][j][k] (1)

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

Byte address

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

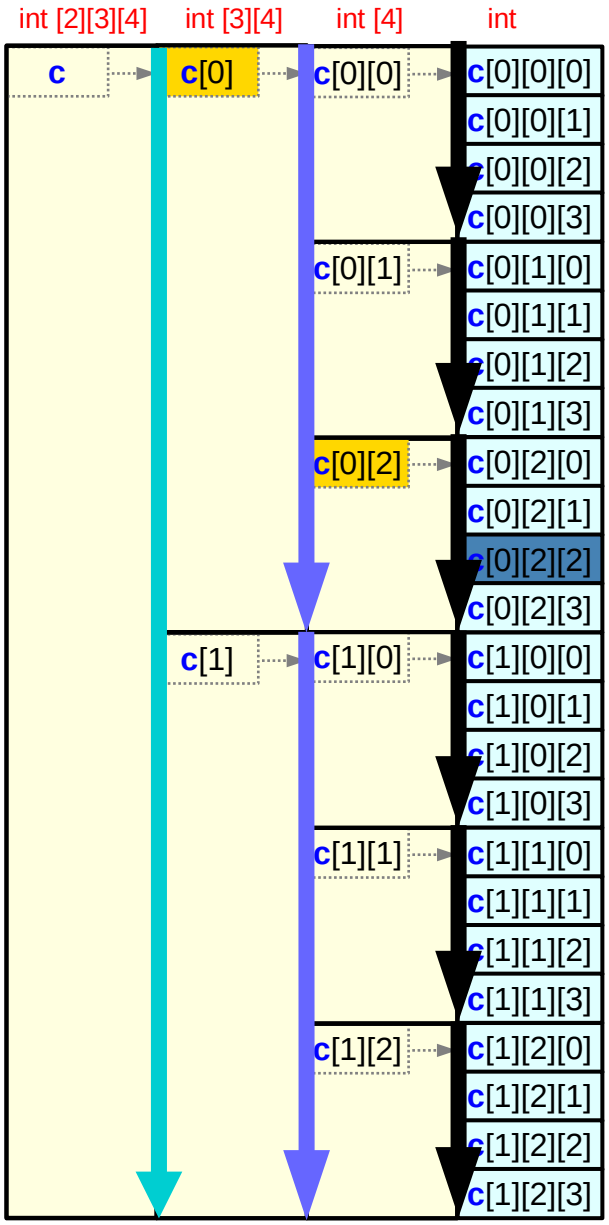
```
value(c[i][j])  = value(&c[i][j])
value(c[i])     = value(&c[i])
```

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

Byte addresses of sub-arrays in an array

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

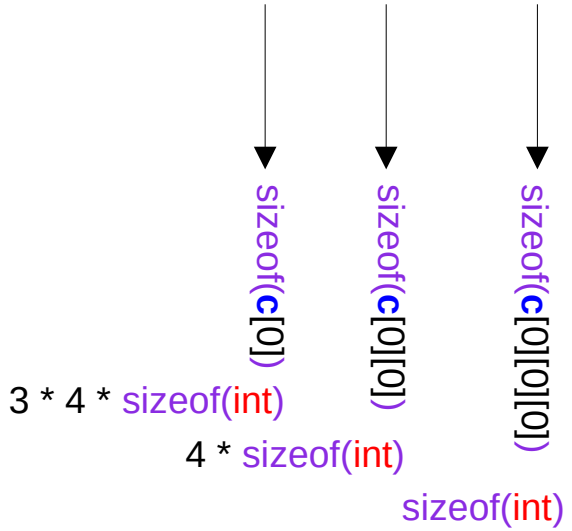
```
c[i][j][k]  = *(c[i][j]+k)
c[i][j]     = *(c[i]+j)
c[i]        = *(c+i)

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



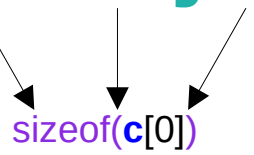
Byte addresses of sub-arrays in an array

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

Though they are equivalent mathematically,
in the respect of pointer arithmetic,
they are very different and
parentheses shall be used to distinguish them
As another way, value() expression is used,
Which returns the address value.



$$\neq c + i + j + k$$



Byte addresses of sub-arrays in an array

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

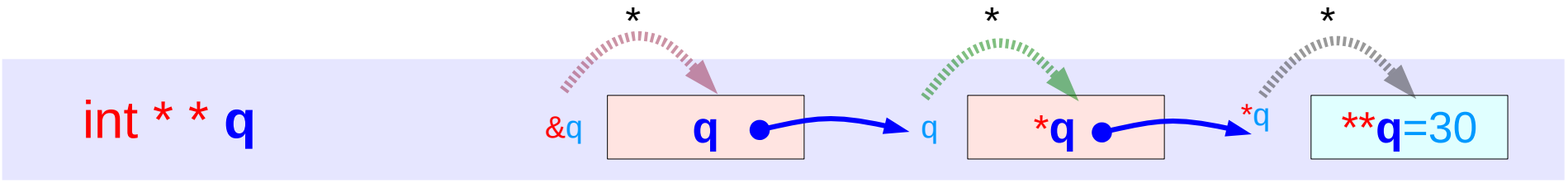
Ideal & operator

C & operator

can be applied to only **lvalue** variable
returns address value
thus, the above expression is not possible
Successive application of & is not possible

In constrast, ***p** becomes a lvalue variable
* operator can be applied successively.

Address-of and dereference operators



*(&q) = q

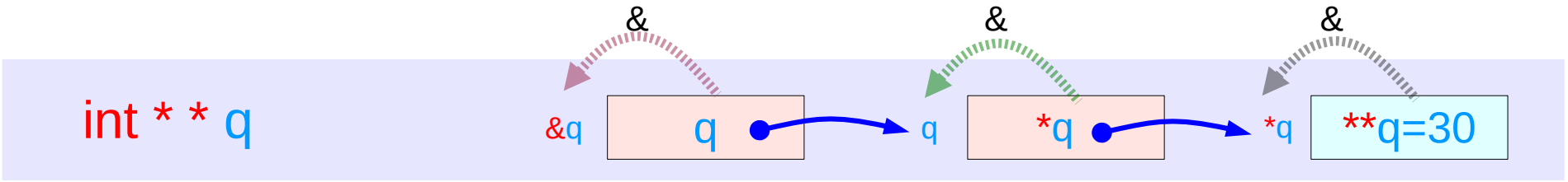
C expression returns data value(**q**) which is an address

*(q) = *q

C expression returns data value(***q**) which is an address

*(*q) = **q

C expression returns data value(****q**) which is an integer



&q

C expression returns address value(**&q**) which is an address

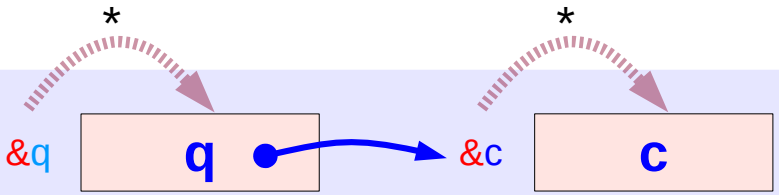
&(*q) = q

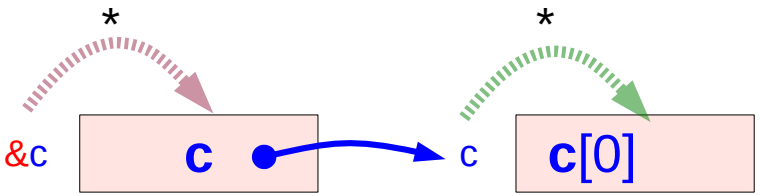
C expression returns address value(**q**) which is an address

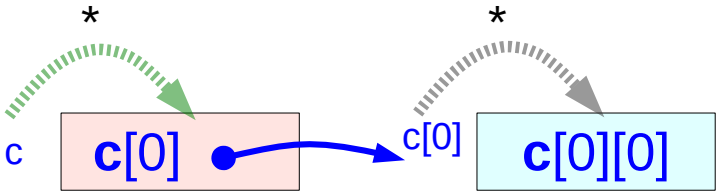
&(**q) = *q

C expression returns address value(***q**) which is an address

Address-of and dereference operators

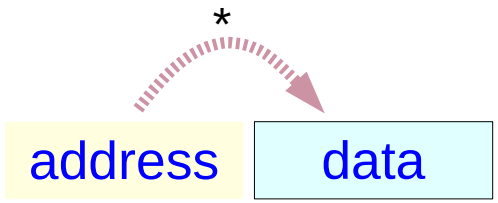




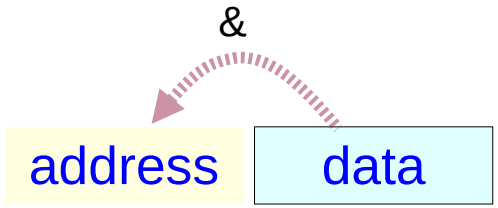


Address-of and dereference operators

Primitive Data Type

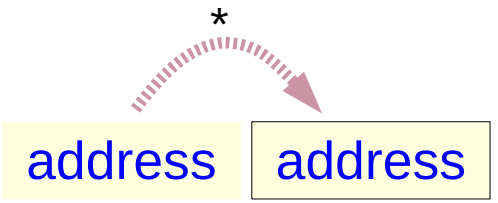


pointer	***l**value*	***l**value*
constant	***r**value*	

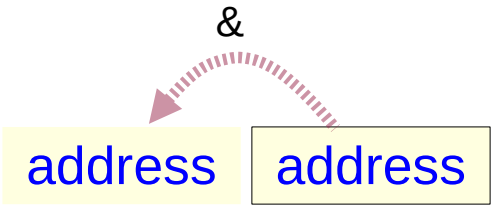


constant	***r**value*	***l**value*

Pointer Data Type

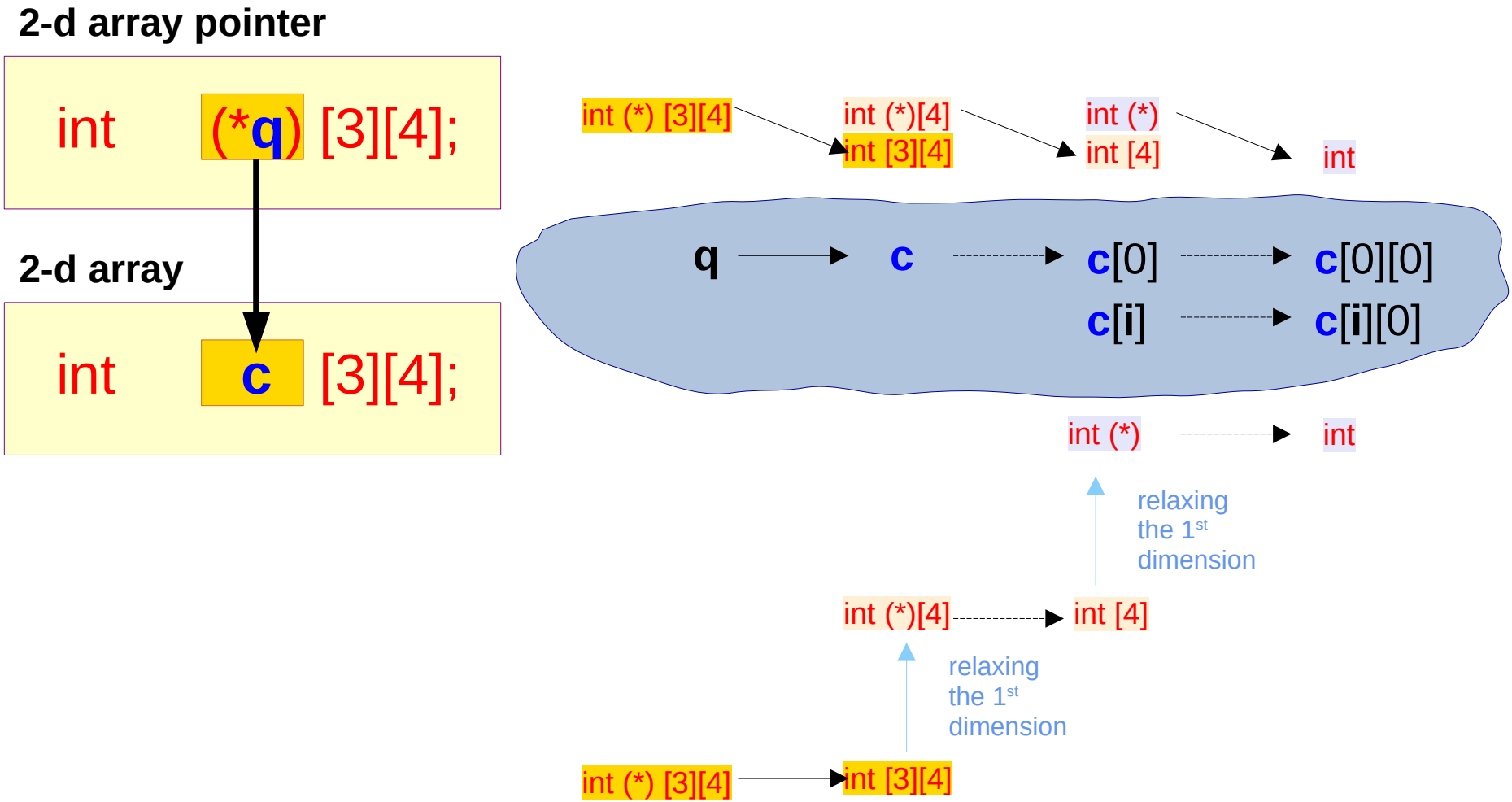


pointer	***l**value*	***l**value*
constant	***r**value*	



constant	***r**value*	***l**value*

2-d array pointer q – (2) types in a pointer chain



Address values of &c[i][j][k] (1)

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

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

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

Address values of &c[i][j][k] (2)

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

```
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)

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

                    = value(c) + i * sizeof(*c) + j * sizeof(*c[i]) + k * sizeof(*c[i][j])
                    = value(c) + i * sizeof(c[0]) + j * sizeof(c[i][0]) + k * sizeof(c[i][j][0])
                    = value(c) + (i * M * N  + j * N  + k) * sizeof(int)
                    = value(c) + ((i * M + j ) * N  + k) * sizeof(int)
```

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

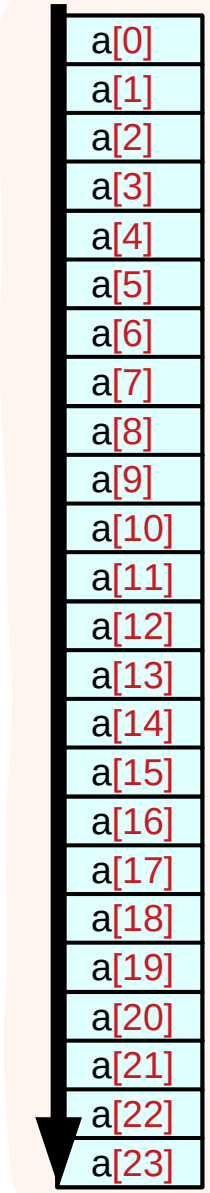
Accessing an int array **a** as a **1-d** array

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

```
a [k]
```

k = 0,1, …,23

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



24=2*3*4

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

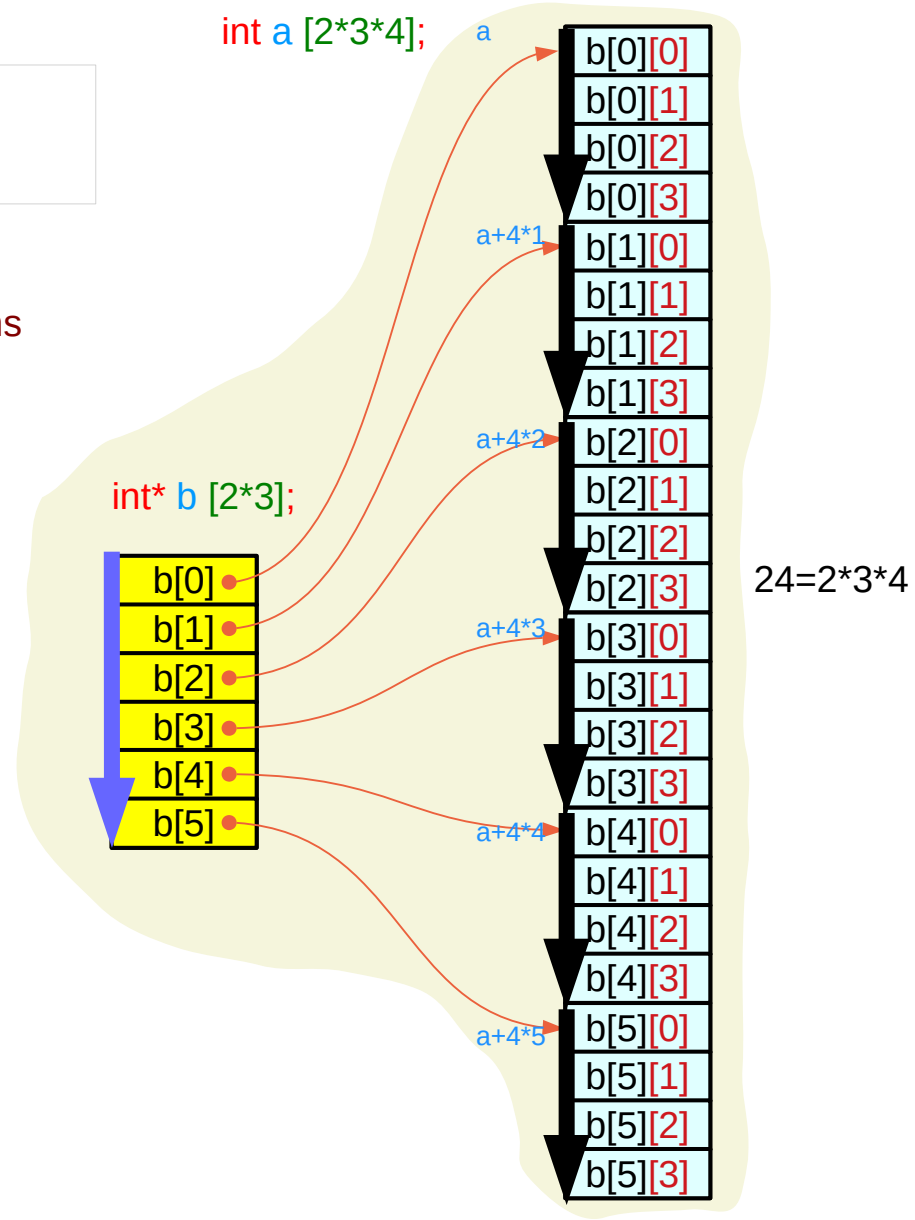
```
int     a [2*3*4] ;
int *   b [2*3] ;
```

b [j][k] ≡ **a** [j*4 +k]

j = 0:5
k = 0:4

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

b take actual memory locations



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

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

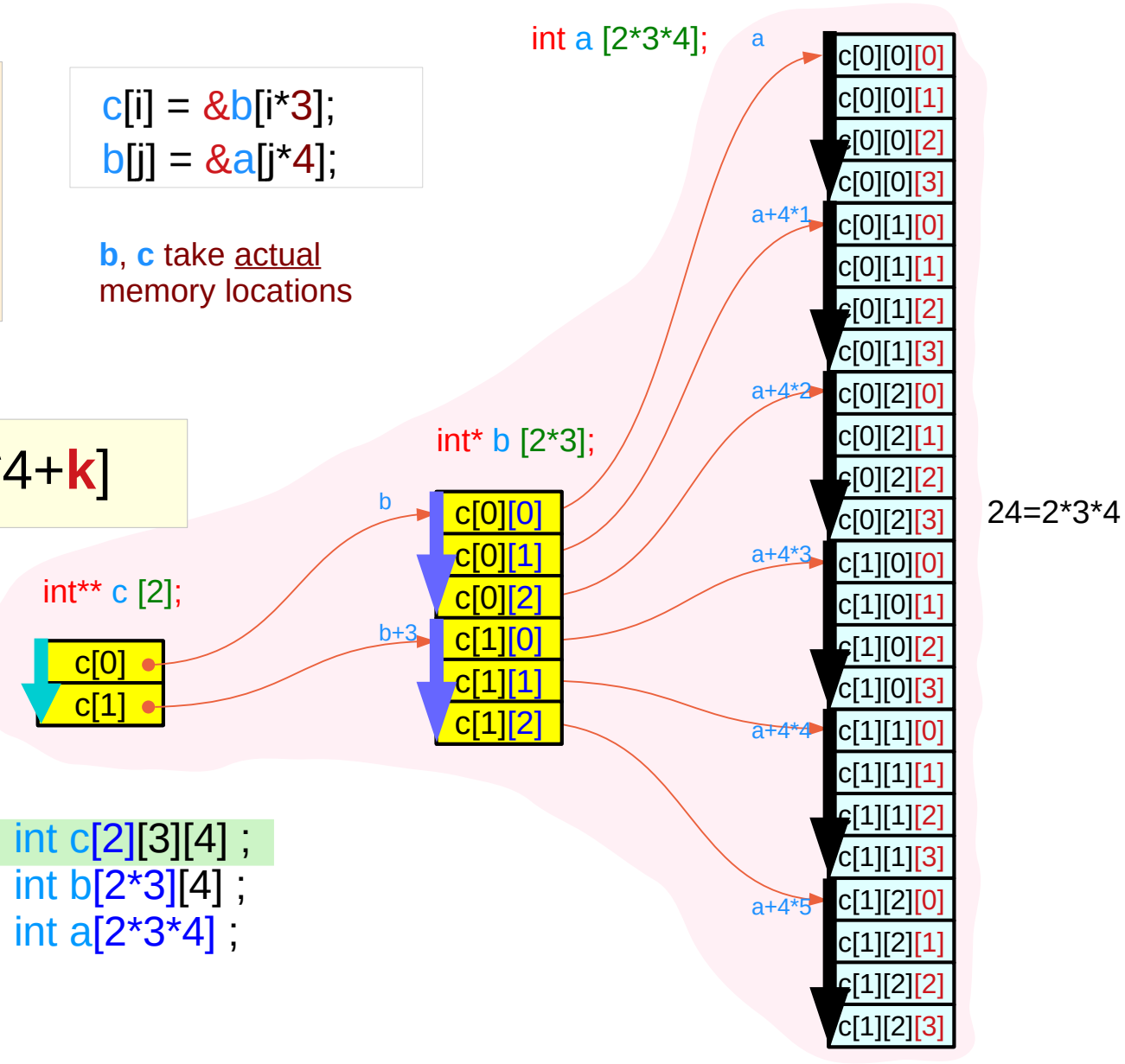
```
int      a [2*3*4] ;
int *    b [2*3] ;
int **   c [2] ;
```

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

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

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

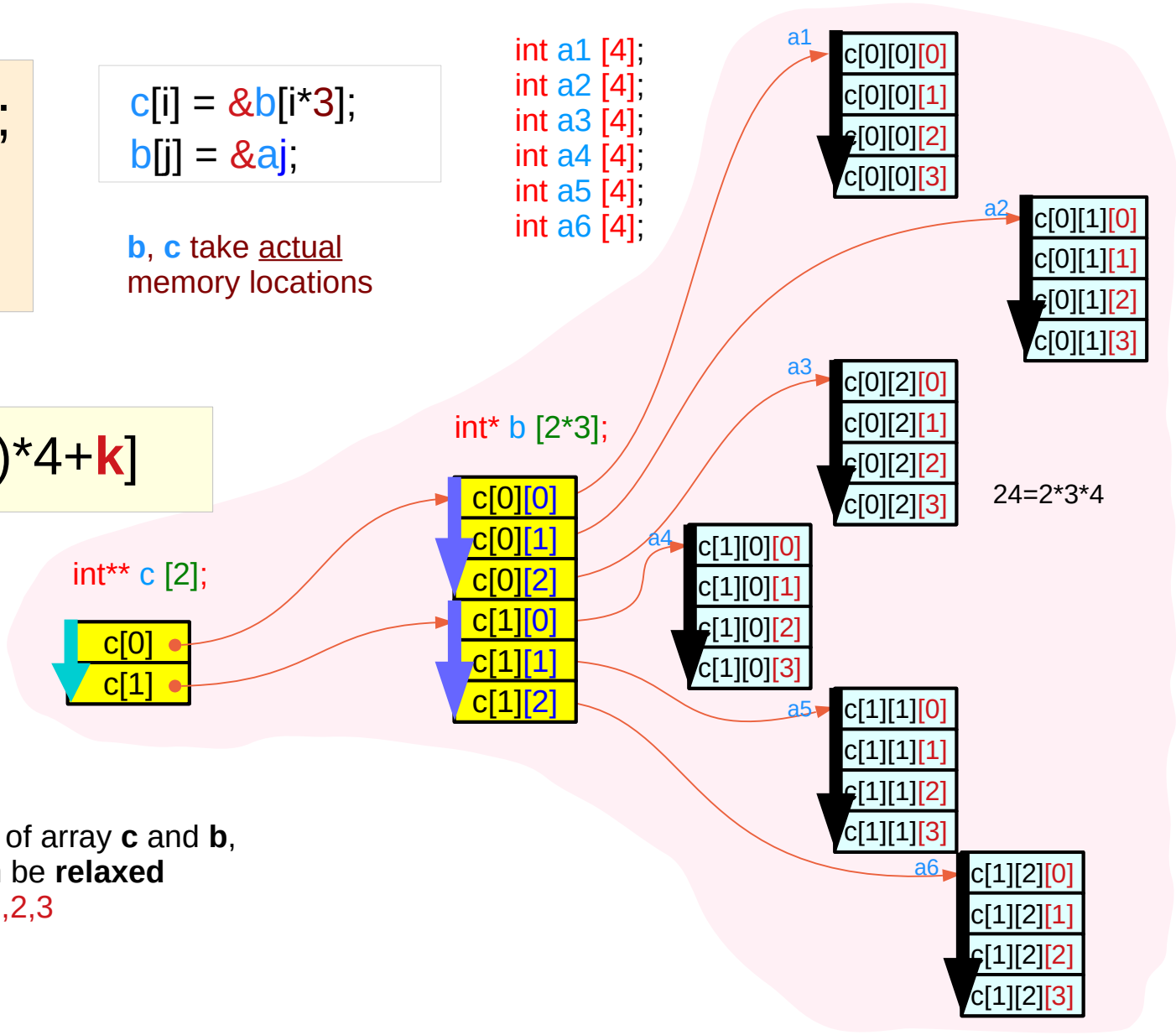
```
int      a [2*3*4] ;
int *    b [2*3] ;
int **   c [2] ;
```

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

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



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] ;
```

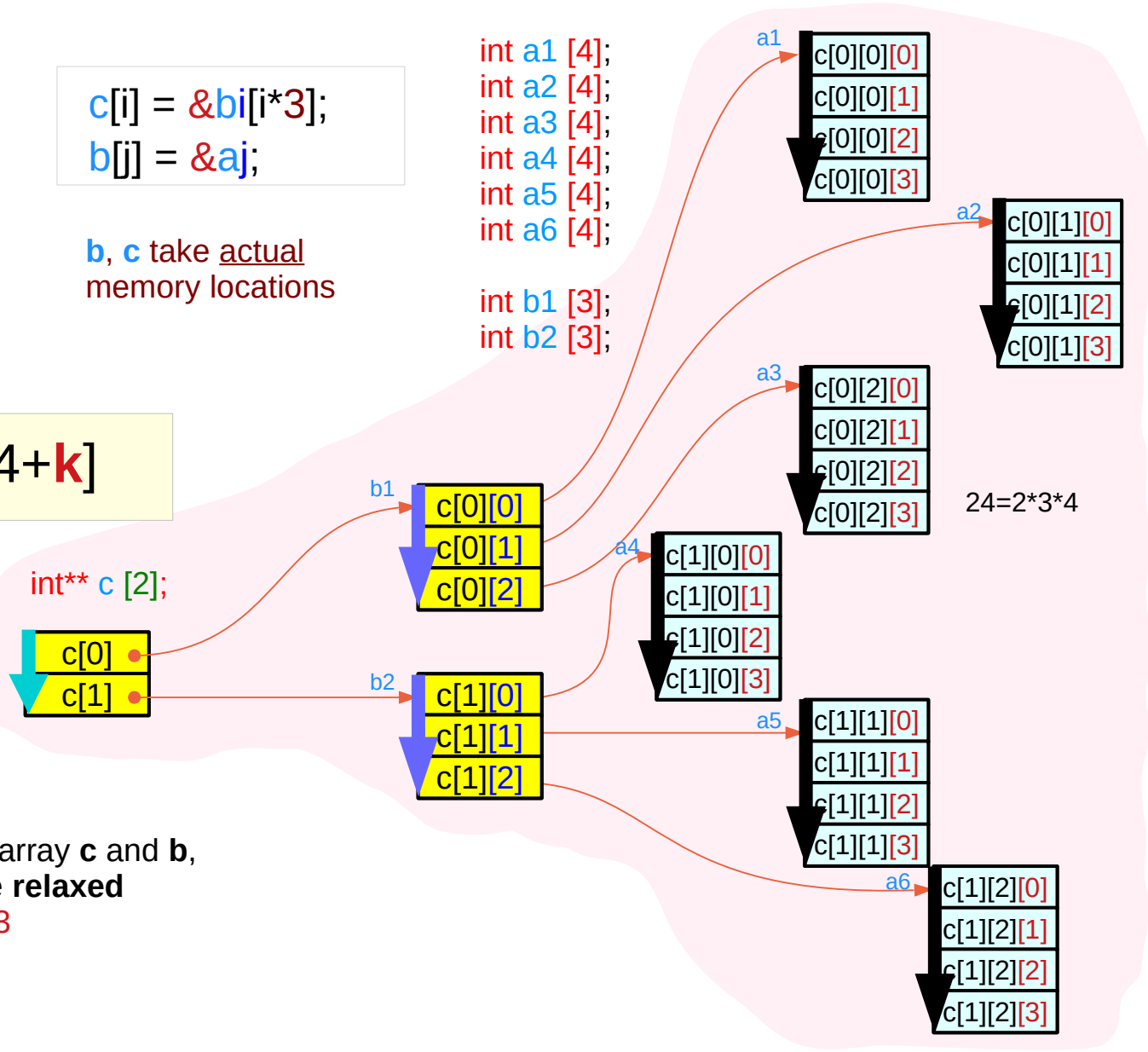
```
c[i] = &bi[i*3];
b[j] = &aj;
```

b, **c** take actual memory locations

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

int b1 [3];
int b2 [3];
```

c [i][j][k] ≡ a [(i*3+j)*4+k]

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



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

3-d access of a 1-d array – pointer array assignment

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

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

b [j][k] ≡ a [j*4 +k]

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

j*4+k = [0:23]

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

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]

Assignments

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

Initialization of
pointer arrays b and c

3-d access of a 1-d array – pointer array assignment

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

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)

Initialization of
pointer arrays b and c

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

c[i][j][k] ≡

a[(i*3+j)*4 +k] ≡ **a**[i*3*4+j*4+k]

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

*(c+i) = b+*g*(i)
*(b+j) = a+*f*(j)

3-d access of a 1-d array – pointer array assignment

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

a[k] ≡ *(a+k)

contiguous over k = 0:23

b[j][k] ≡ *(*(b+j)+k)

➡ *(b[j]+k) = *(a+j*4+k) = a[j*4+k]

⇦ b[j] = &a[j*4] (= a+j*4)

contiguous over j = 0:5 & k = 0:3

partition 24 into 6 * 4

c[i][j][k] ≡ *(*(*(c+i)+j)+k)

partition 6 into 2 * 3

➡ *(*(c[i]+j)+k) = *(*(b+i*3+j)+k)

⇦ c[i] = &b[i*3] (= b+i*3)

➡ *(b[i*3+j]+k) = *(a+(i*3+j)*4+k)

⇦ b[j] = &a[j*4] (= a+j*4)

➡ a[(i*3+j)*4+k]

contiguous over i =0:1 & j= 0:2 & k = 0:3

partition 24 into 2 * 3 * 4

3-d access of a 1-d array – pointer array assignment

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

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

partition 24 into 6 * 4
partition size = 4

contiguous **a** over k = 0:3
contiguous **b** over j = 0:5

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

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

(1) partition 24 into 6 * 4
1st partition size = 4

(2) partition 6 into 2 * 3
2nd partition size = 3

contiguous **a** over k = 0:3
contiguous **b** over j = 0:2
contiguous **c** over i = 0:1

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

3-d access of a 1-d array – pointer array assignment

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

(1) partition 24 into six 4's (6 * 4)
1st partition size = 4

(2) partition 6 into two 3's (2 * 3)
2nd partition size = 3

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

b [j][k] ≡ a [j*4 +k]

contiguous a over k = 0:3 (=4-1)
contiguous b over j = 0:5 (=6-1)

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 a over k = 0:3 (=4-1)

c [i][j][k] ≡ a [(i*3+j)*4+k]

contiguous b over j = 0:2 (=3-1)
contiguous c over i = 0:1 (=2-1)

3-d access of a 1-d array – pointer array sizes

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

int a [2*3*4] ;

sizeof(int **) = 4 or 8 bytes
sizeof(int *) = 4 or 8 bytes

on a 32-bit machine

on a 64-bit machine

sizeof(int) = 4 bytes

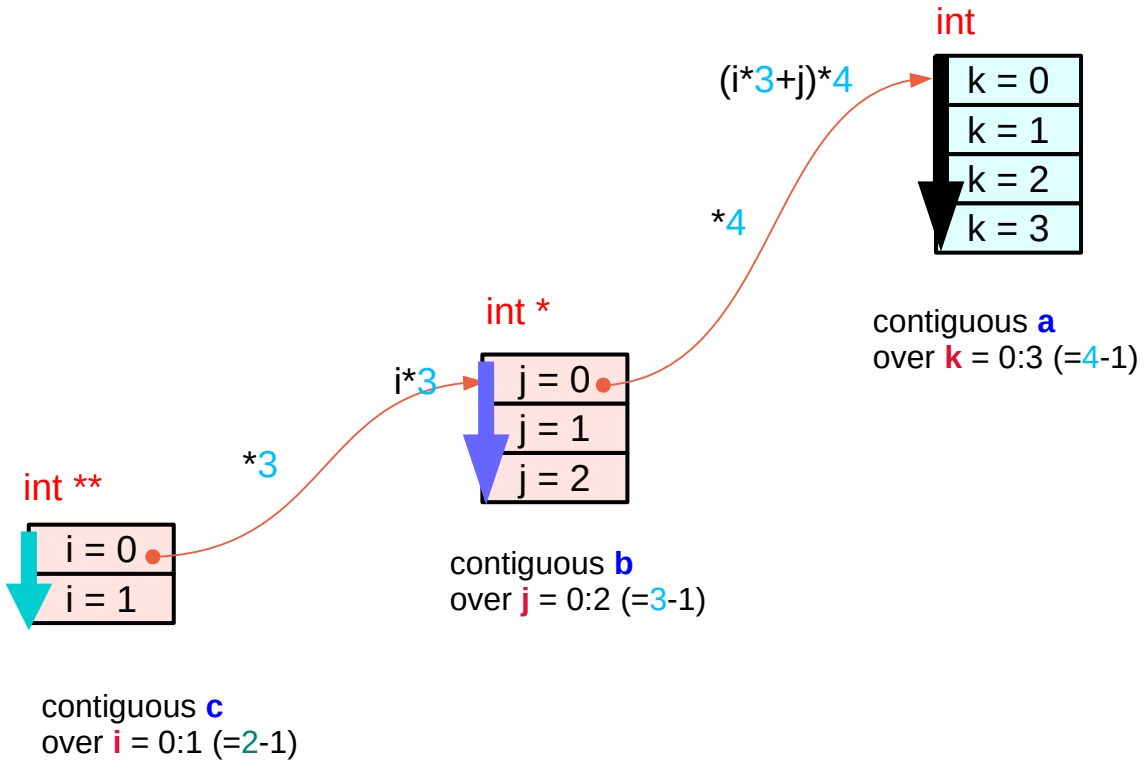
sizeof(c) = 2*sizeof(int **)
sizeof(b) = 2*3*sizeof(int *)

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

Using pointer arrays

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

c [i][j][k]



conditions

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

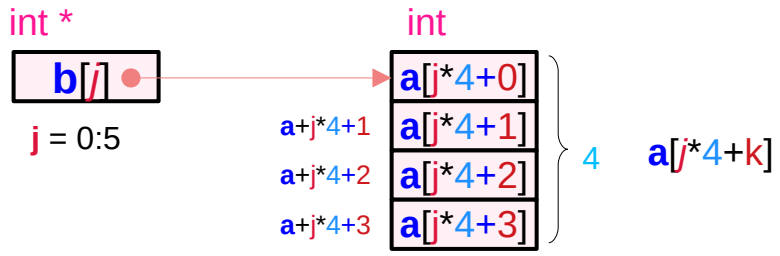
c [**i**][**j**][**k**] ≡ **a** [(**i***3+**j**)*4+**k**]

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

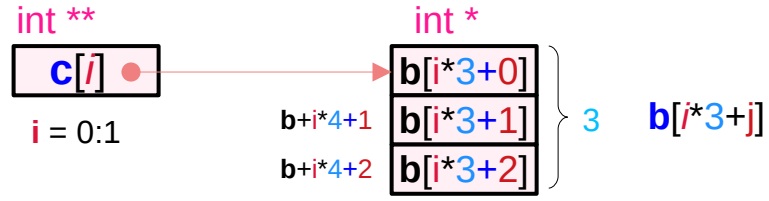
int	**a** [2*3*4] ;
int *	**b** [2*3] ;
int **	**c** [2] ;

(1) partition 24 into six 4's (6 * 4)
1st partition size = 4

(2) partition 6 into two 3's (2 * 3)
2nd partition size = 3



contiguous **a** over **k** = 0:3 (=4-1)



contiguous **b** over **j** = 0:2 (=3-1)
contiguous **c** over **i** = 0:1 (=2-1)

j = 0:5

```
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)
```

i = 0:1

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

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

Static memory allocation of an **3-d** array

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

```
int * p = (int *) c ;
```

```
int *              int *
&c[i][j][k] = (p+(i*3+j)*4+k)
```

```
int                int *
c[i][j][k] = *(c[i][j]+k)
&c[i][j][k] = (c[i][j]+k)
```

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

```
int *
c[i][j]      = (p+(i*3+i)*4)
```

```
int *        int **
c[i][j] = *(c[i]+j)
&c[i][j] = (c[i]+j)
```

```
int *        int **
c[i][0] = *c[i]
&c[i][0] = c[i]
```

```
int **
c[i]         = (int **) (p+i*3)
```

```
int **       int ***
c[i] = *(c+i)
&c[i] = (c+i)
```

```
int **       int ***
c[0] = *c
&c[0] = c
```

```
c
```

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

Static memory allocation of an **3-d** array

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

```
                                  int *          int *
&c[i][j][k] = (p+(i*3+j)*4+k)
```

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

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

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

int **    c
```

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

Static memory allocation of an **3-d** array

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

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][1][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][1][0]

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

c[**i**][**j**] ⟶ **c**[**i**][**j**][0] if **c**[**i**][**j**] = &**c**[**i**][**j**][0] **c**[**i**][**j**] = (**c**+**i***3+**j***4)

c[**i**] ⟶ **c**[**i**][0] if **c**[**i**] = &**c**[**i**][0]
= &**c**[**i**][0][0] **c**[**i**] = (**c**+**i***3)

c ⟶ **c**[0] if **c** = &**c**[0]
= &**c**[**i**][0]
= &**c**[**i**][0][0] **c**

Static memory allocation of an **3-d** array

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

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][1][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][1][0]

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

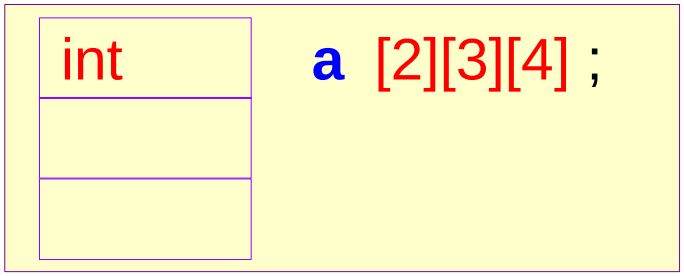
c[**i**][**j**] = (**c**+**i***3+**j***4) if **c**[**i**][**j**] = &**c**[**i**][**j**][0]

c[**i**] = (**c**+**i***3) if **c**[**i**] = &**c**[**i**][0][0]

c[**i**][0] = (**c**+**i***3) if **c**[**i**][0] = &**c**[**i**][0][0]
&**c**[**i**][0] = (**c**+**i***3) if

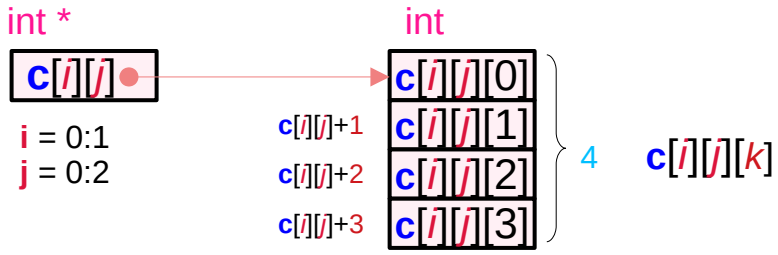
c[**i**] = (**c**+**i***3)

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

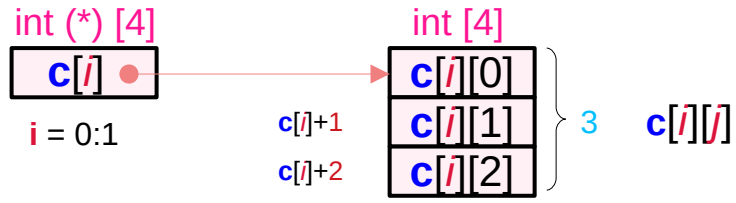


(1) partition 24 into six 4's (6 * 4)
1st partition size = 4

(2) partition 6 into two 3's (2 * 3)
2nd partition size = 3



contiguous **c**[i][j] over **k** = 0:3 (=4-1)



contiguous **c**[i] over **j** = 0:2 (=3-1)
contiguous **c** over **i** = 0:1 (=2-1)

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][1][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][1][0]

Leading elements : **c[i][0][0], c[i][j][0]**

```
int     a [L*M*N];
int*    b [L*M];
int**   c [L];
```

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

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

L=2

i=0	i*3*4 = 0
i=1	i*3*4 = 12

M=3

j=0	j*4 = 0
j=1	j*4 = 4
j=2	j*4 = 8

N=4

k=0	k*1= 0
k=1	k*1= 1
k=2	k*1= 2
k=3	k*1= 3

c[0][0][0] = a[0]	0
c[1][0][0] = a[12]	12
c[0][0][0] = a[0]	0+0
c[0][1][0] = a[4]	0+4
c[0][2][0] = a[8]	0+8
c[1][0][0] = a[12]	12+0
c[1][1][0] = a[16]	12+4
c[1][2][0] = a[20]	12+8

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]

Initialization of pointer arrays – a general case

```
int      a [L*M*N];
```

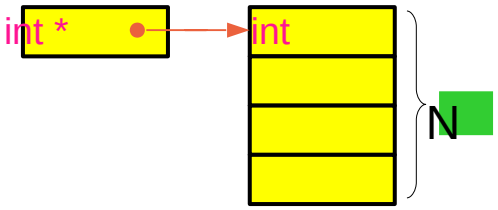
```
int*     b [L*M];
int**    c [L];
```

pointer arrays b, c

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

```
int *    b[L*M];
int      a[L*M*N];
```

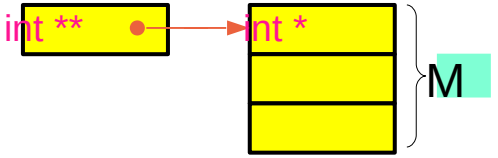
```
b[j] = &a[j*N];
j=0,…, L*M-1
```



b[j] get the address of the every Nth element of a

```
int **   c[L];
int *    b[L*M];
```

```
c[i] = &b[i*M];
i=0, …. L-1
```



c[i] get the address of the every Mth element of b

3-d and 1-d accesses (recursive pointers vs. brackets)

conditions

```
c[i] = &b[i*M];
b[j] = &a[j*N];
```

c[i][j][k] ≡ a[i*M*N + j*N + k]

≡ a[(i*M + j)*N + k]

```
int **      c[L];
int *       b[L*M];
```

```
for (i=0; i<L; ++i)
    c[i] = &b[i*M];
```

```
int *       b[L*M];
int         a[L*M*N];
```

```
for (j=0; j<L*M; ++j)
    b[j] = &a[j*N];
```

c[i][j][k]

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

= *(*(c[i]+j)+k) ⬅ c[i] =&b[i*M]

= *(*(&b[i*M]+j)+k) ➡ *(*(b+i*M+j)+k)

= *(b[i*M+j]+k) ⬅ b[m] = &a[m*N]

= *(&a[(i*M+j)*N]+k) ➡ *(a+(i*M+j)*N+k)

= a[(i*M+j)*N+k]

Pointer Arrays for recursive indirections

1-d array of (int **) pointers
1-d array of (int *) pointers
1-d array of (int)

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

3-d access

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

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

equivalence relations

multiple indirections

c [i]	≡	*(c+i)	≡	*(c+i)
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
&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
```

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

General requirements

```
&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 Array Constraints

Abstract Data Type

Array pointer approach

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

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

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

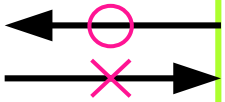
Virtual Array Pointer Constraints

Abstract Data Type

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

General requirements

```
&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]
```

Types and values of **c[i]** and **c[i][j]** for **int c[2][3][4];**

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

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

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

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

c[i] virtual array pointer of the type int (*) [4] … a narrow sense
can also be viewed as the int** type … a wide sense

```
&c[0][0][0] = c[0][0]
&c[1][0][0] = c[1][0]
```
int*

```
&c[0][0] = c[0]
&c[1][0] = c[1]
```
int**

c[i][j] virtual int pointer of the type int (*) … a narrow sense
can also be viewed as the int* type … a wide sense

```
&c[0][0][0] = c[0][0]
&c[0][1][0] = c[0][1]
&c[0][2][0] = c[0][2]
&c[1][0][0] = c[1][0]
&c[1][1][0] = c[1][1]
&c[1][2][0] = c[1][2]
```
int*

Using **int**** and **int*** pointer arrays for **3-d** accesses

```
int c[2][3][4];
&c[i][0]    = c[i]

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

            int**
```

```
int c[2][3][4];
&c[i][j][0]    = c[i][j]

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

            int*
```

General Requirements

```
int** c[2];
c[i]   = &b[i*3]

c[0] = &b[0*3]
c[1] = &b[1*3]

            int**
```

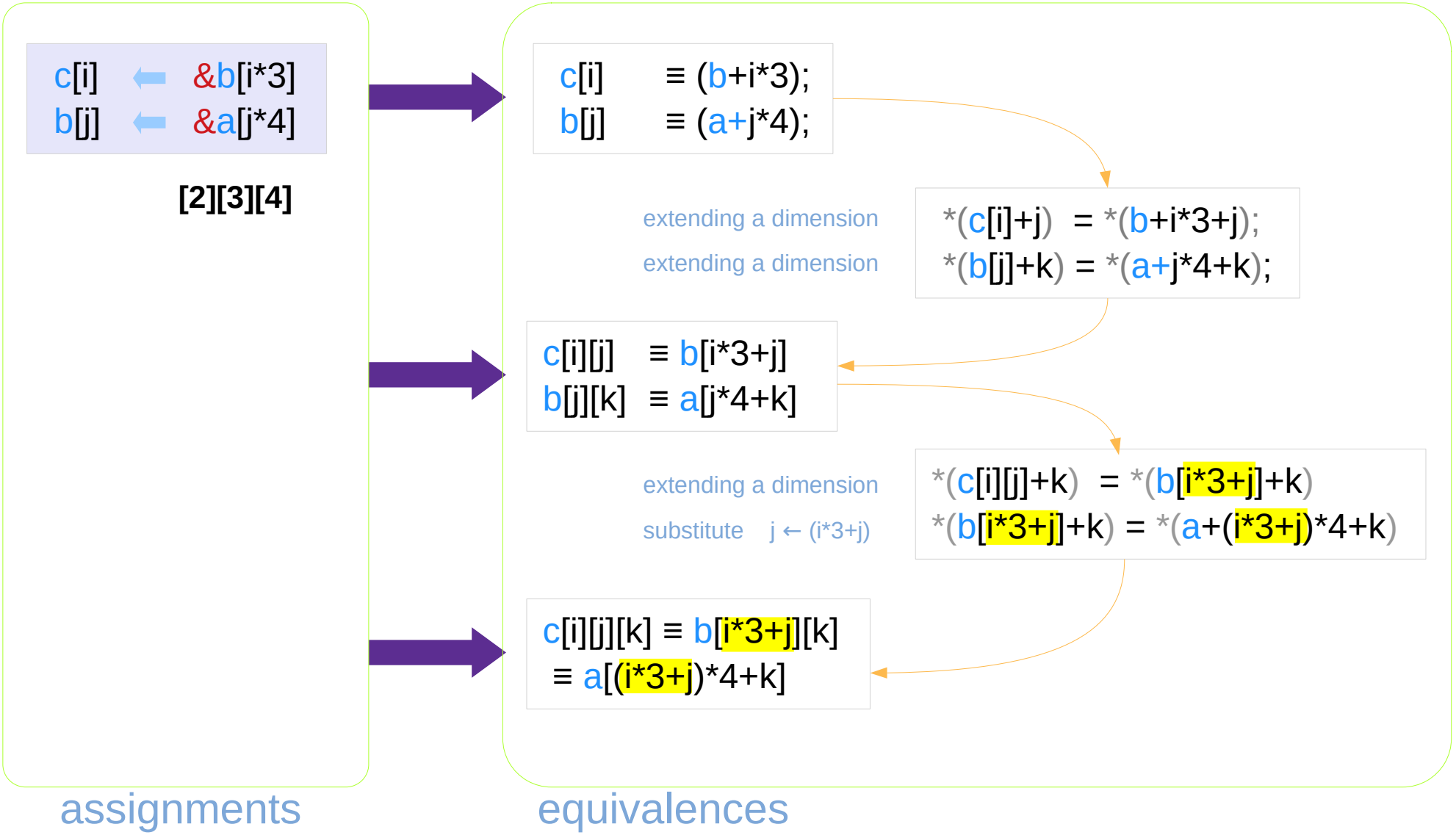
```
int* b[2*3];
b[j]   = &a[j*4]

b[0] = &a[0*4]
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]

            int*
```

Pointer Array Implementation

instead of using **int c[2][3][4]**,
use these 1-d arrays of pointers
int c[2]** and **int* b[2*3]**
with proper initializations:
c[i] = &b[i*3] and **b[j] = &a[j*4]**

then **c[i][j][k]** can be used
to access the 1-d array
int a[2*3*4]

Assignments and their Equivalent Relations



The leading elements of pointer arrays

c[i] ⬅ &b[i*3]
b[j] ⬅ &a[j*4]

assignments

c[i] ≡ (b+i*3);
b[j] ≡ (a+j*4);

equivalence

c[i][j] ≡ b[i*3+j]
b[j][k] ≡ a[j*4+k]

equivalence

c[i][0] ≡ b[i*3];
b[j][0] ≡ a[j*4];

The 1st elements of **c[i][j]**, **b[i][j]**

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

equivalence

c[i][j][0] ≡ b[i*3+j];
c[i][0][0] ≡ a[(i*3)*4];

The 1st elements of **c[i][j][k]**

c[i], c[i][j], c[i][j][k] in terms of array a and b

c[i] ⬅ &b[i*3]
b[j] ⬅ &a[j*4]

assignments

c[i] ≡ (b+i*3);
b[j] ≡ (a+j*4);

equivalence

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

&& is not allowed

c[i][j] ≡ b[i*3+j]
b[j][k] ≡ a[j*4+k]

equivalence

c[i][j] ≡ b[i*3+j]
≡ &a[(i*3+j)*4]

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

equivalence

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

Pointer Arrays – **c[i]** reaches **c[i][0][0]** via **c[i][0]**

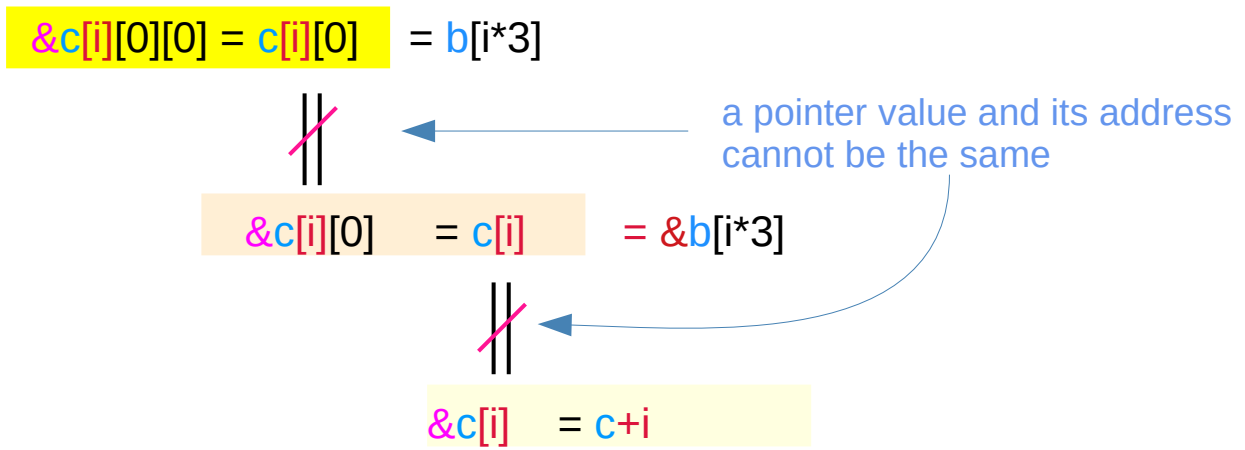
```
c [i][j][k];
```

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

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

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

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



Pointer Arrays – **c[i][j]** reaches **c[i][j][0]**

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

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

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

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

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

&c[i][j][0] = c[i][j] = b[i*3+j] = &a[(i*3+j)*4]

Recursive Indirections – thinking pointer substitutions

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

```
X = c[i][j]     int *
Y = c[i]        int **
Z = c           int ***
```

for a given **i**, **j**, **k**

```
X[k]            ≡   *(X+k)
Y[j][k]         ≡   *(*(Y+j)+k)
Z[i][j][k]      ≡   *(*(*(Z+i)+j)+k)
```

Recursive Indirections – general cases of 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)
```

$X_{i,j}$ = c[i][j] int *

Y_i = c[i] int **

Z = c = Y int ***

for general cases of indices i, j, k,
X and **Y** need to be arrays of pointers

```
X[i,j][k]           ≡   *(X[i,j]+k)
Y[i][j][k]          ≡   *(*(Y[i]+j)+k)
Z[i][j][k]          ≡   *(*(*(Z+i)+j)+k)
```

Recursive Indirections – Pointer array initialization

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

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

$X_{i,j}$ = c[i][j] int *

Y_i = c[i] int **

Z = c = Y int ***

```
X[i*M+j]   = c[i][j];
Y[i]       = c[i];
```

$X_{i,j}$[k] ≡ *($X_{i,j}$+k)

Y_i[j][k] ≡ *(*(Y_i+j)+k)

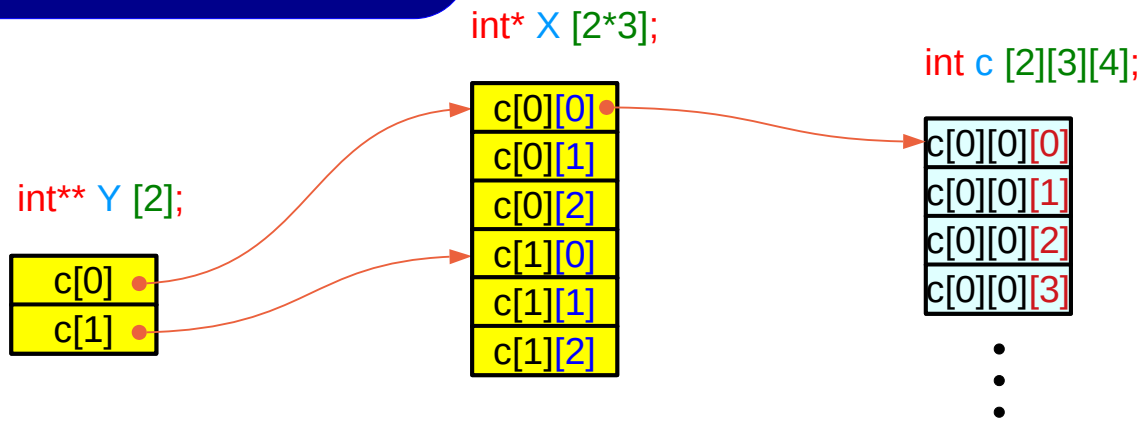
Y[i][j][k] ≡ *(*(*(Y+i)+j)+k)

```
int     W  [L*M*N]   ;
int *   X  [L*M]     ;
int **  Y  [L]       ;
```

Recursive Indirections – Substitution Analysis

```
X[i*M+j]= c[i][j];
Y[i]    = c[i];

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

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

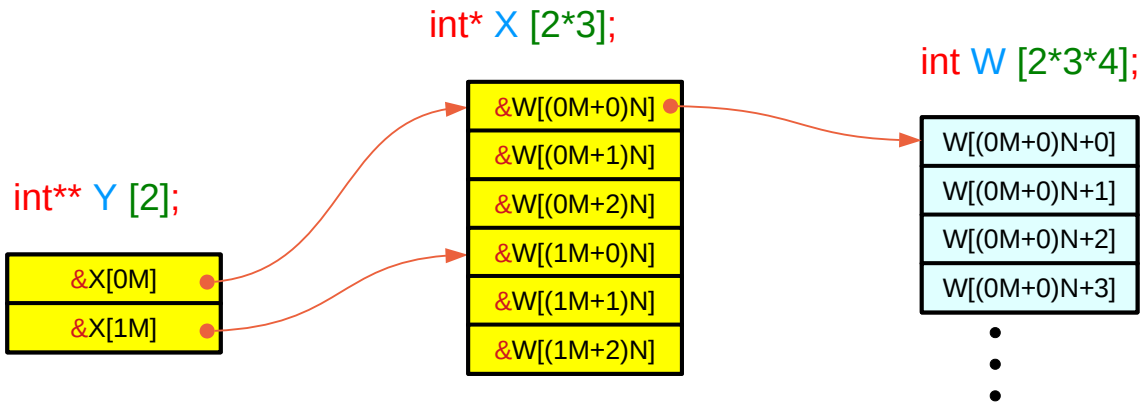
```
&Y[i][j][0] = &c[i][j][0] = c[i][j] = Y[i][j]
&Y[i][0]    = &c[i][0]    = c[i]    = Y[i]
&Y[i]                                = Y+i
```



Recursive Indirections – one continuous int array W

&Y[i][j][0]	&W[(i*M+j)*N+0]	= Y[i][j]
=	X[(i*M+j)]	
&Y[i][0]	&X[i*M+0]	= Y[i]
&Y[i]		= Y+i

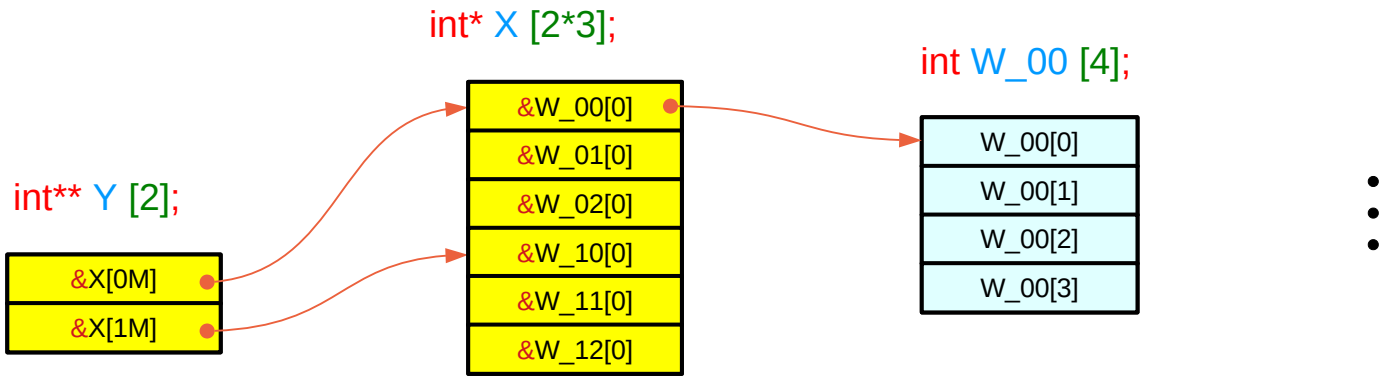
```
X[i*M+j]    = &W[(i*M+j)*N];
Y[i]        = &X[i*M];

Y[i][j]     = *(Y[i]+j)
            = *(X+i*M+j)
            = X[i*M+j]

Y[i][j][k]  = *(Y[i][j]+k)
            = *(W+(i*M+j)*N+k)
            = W[(i*M+j)*N+k]
```



Recursive Indirections – non-contiguous 1-d arrays W_ij

&Y[i][j][0]	&W_ij	= Y[i][j]
=	X[(i*M+j)]	
&Y[i][0]	&X[i*M+0]	= Y[i]
&Y[i]		= Y+i

X[i*M+j]	= &W_ij[0];
Y[i]	= &X[i*M];
Y[i][j]	= *(Y[i]+j) = *(X+i*M+j) = X[i*M+j]
Y[i][j][k]	= *(Y[i][j]+k) = *(W+(i*M+j)*N+k) = W[(i*M+j)*N+k]



Recursive Indirections – contiguous v.s. non-contiguous

```
int     W  [L*M*N]  ;
int *   X  [L*M]    ;
int **  Y  [L]      ;
```

```
int     W_00 [N]    ;
int     W_01 [N]    ;
        :
int *   X    [L*M]  ;
int **  Y    [L]    ;
```

W[(i*M+j)*N+k];

one contiguous 1-d array
with the size of L*M*N

W_ij[k];

L*M non-contiguous 1-d arrays
with the size of N

Contiguity Constraints

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

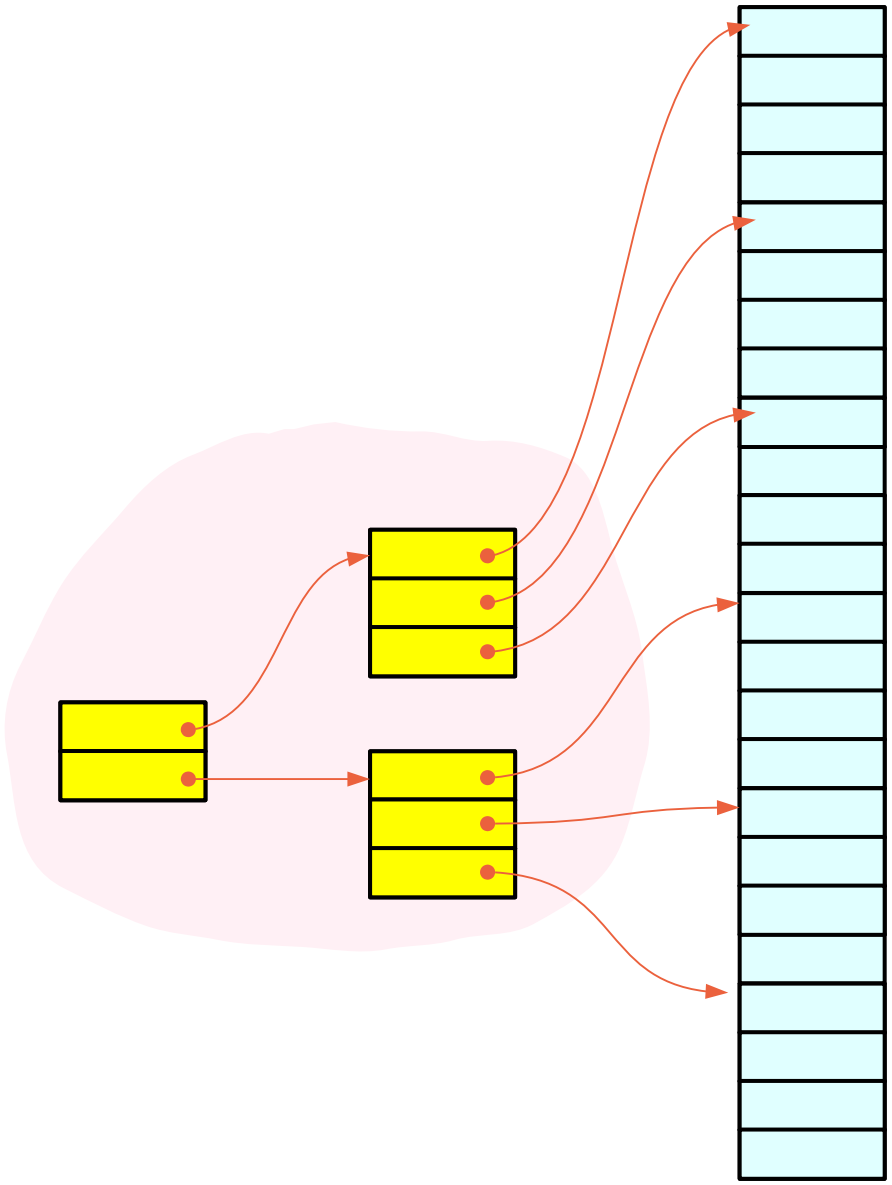
Pointer Arrays and Contiguity

Using pointer arrays

int * [N], int ** [M], int *** [L], …

Pointer array approach for 3-d access patterns

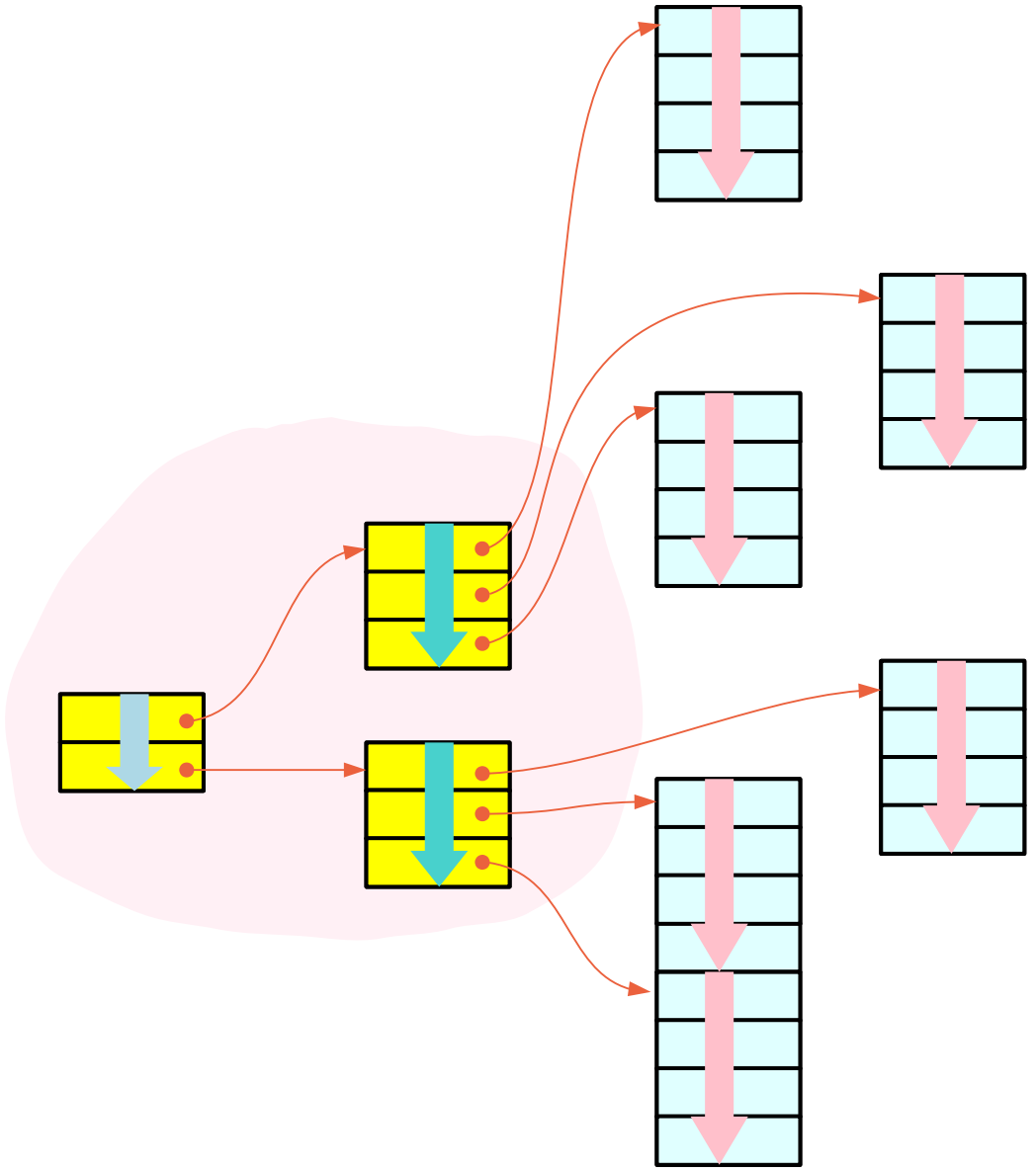
A programmer manually allocates
memory locations for pointer arrays



Pointer Array Approach
(array of pointers)

Pointer array approach – contiguity constraints

contiguity constraints
can be relaxed



Pointer Array Approach
(array of pointers)

Three contiguity constraints

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)

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

c[2][3][4]

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)

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

c[2][3][4]

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

c[2][3][4]

Three contiguity constraints

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 **1-d** array elements — int
contiguous **int** pointers — int *
contiguous **int** double pointers — int **

The contiguity constraints are satisfied by the allocated 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)

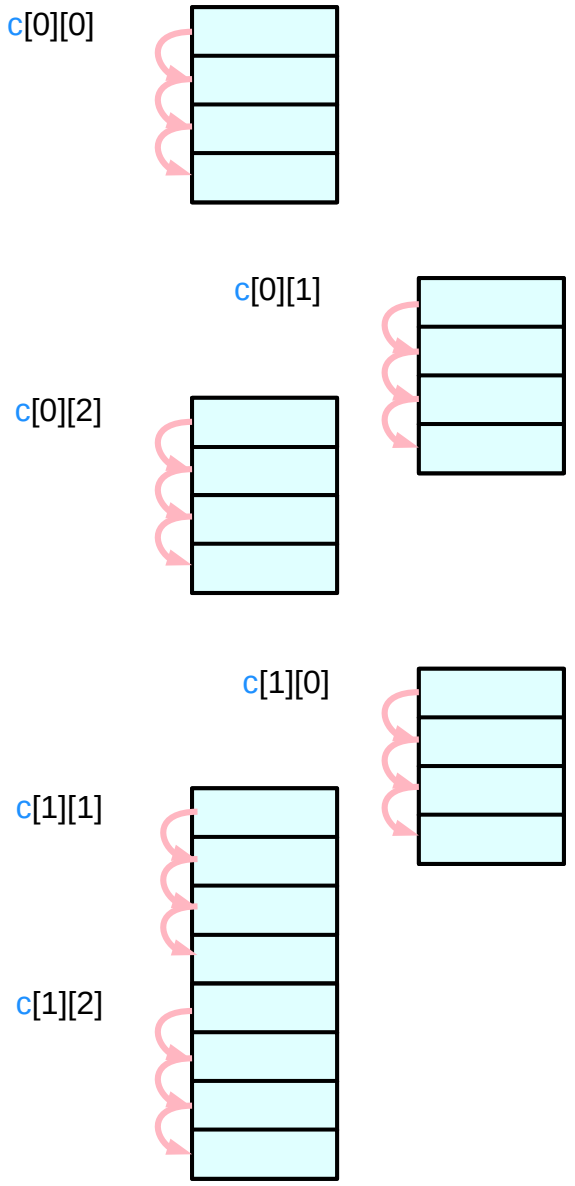
contiguous **1-d** array elements — int
contiguous **1-d** arrays — int [4]
contiguous **1-d** array pointers — int (*) [4]

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

c[i][j][k] ≡ *(c[i][j] +k)

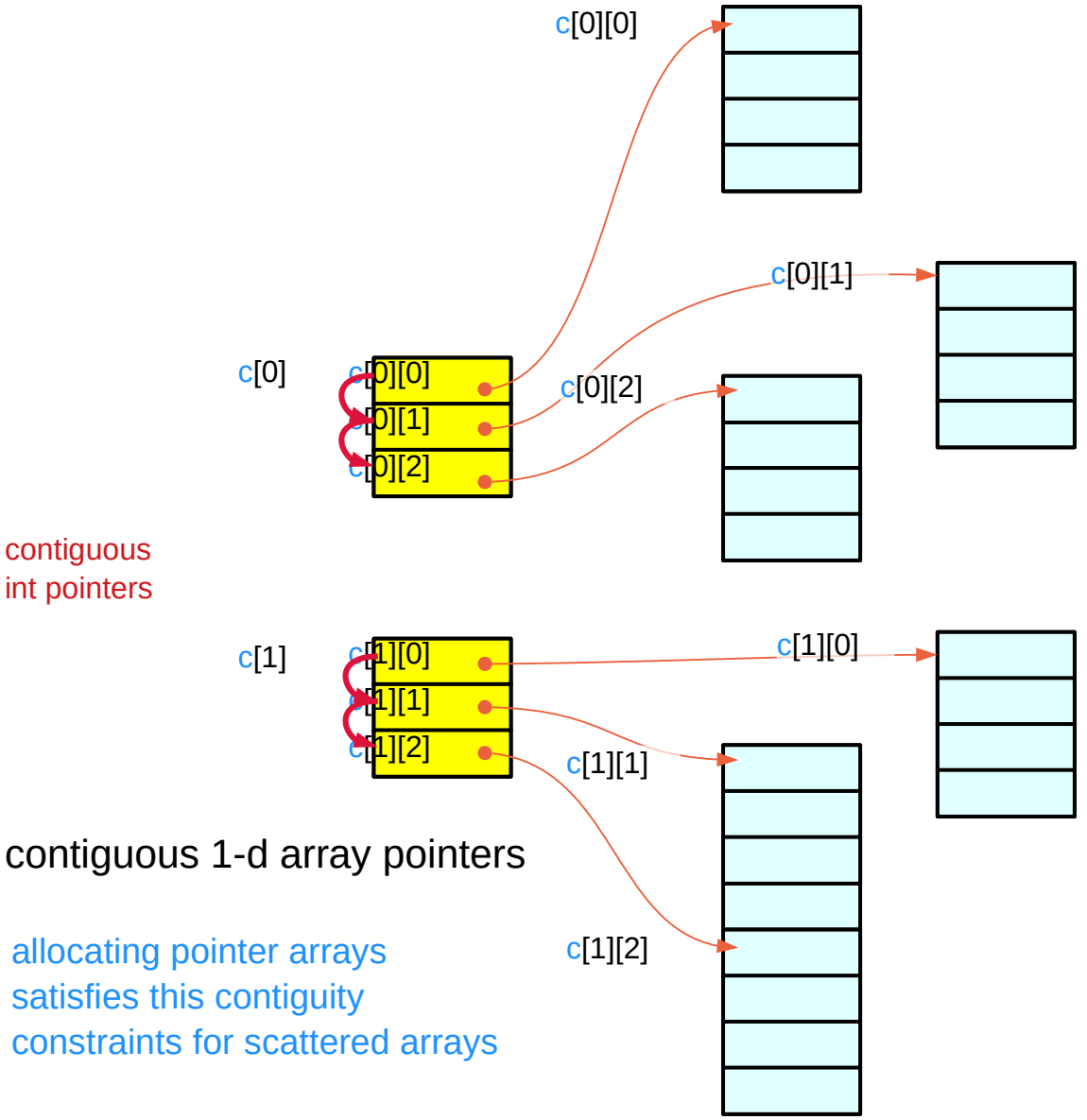
```
c[0][0][0] =  *(c[0][0] + 0)
c[0][0][1] =  *(c[0][0] + 1)
c[0][0][2] =  *(c[0][0] + 2)
c[0][0][3] =  *(c[0][0] + 3)
c[0][1][0] =  *(c[0][1] + 0)
c[0][1][1] =  *(c[0][1] + 1)
c[0][1][2] =  *(c[0][1] + 2)
c[0][1][3] =  *(c[0][1] + 3)
```

contiguous 1-d
array elements



c[i][j] ≡ *(c[i] +j)

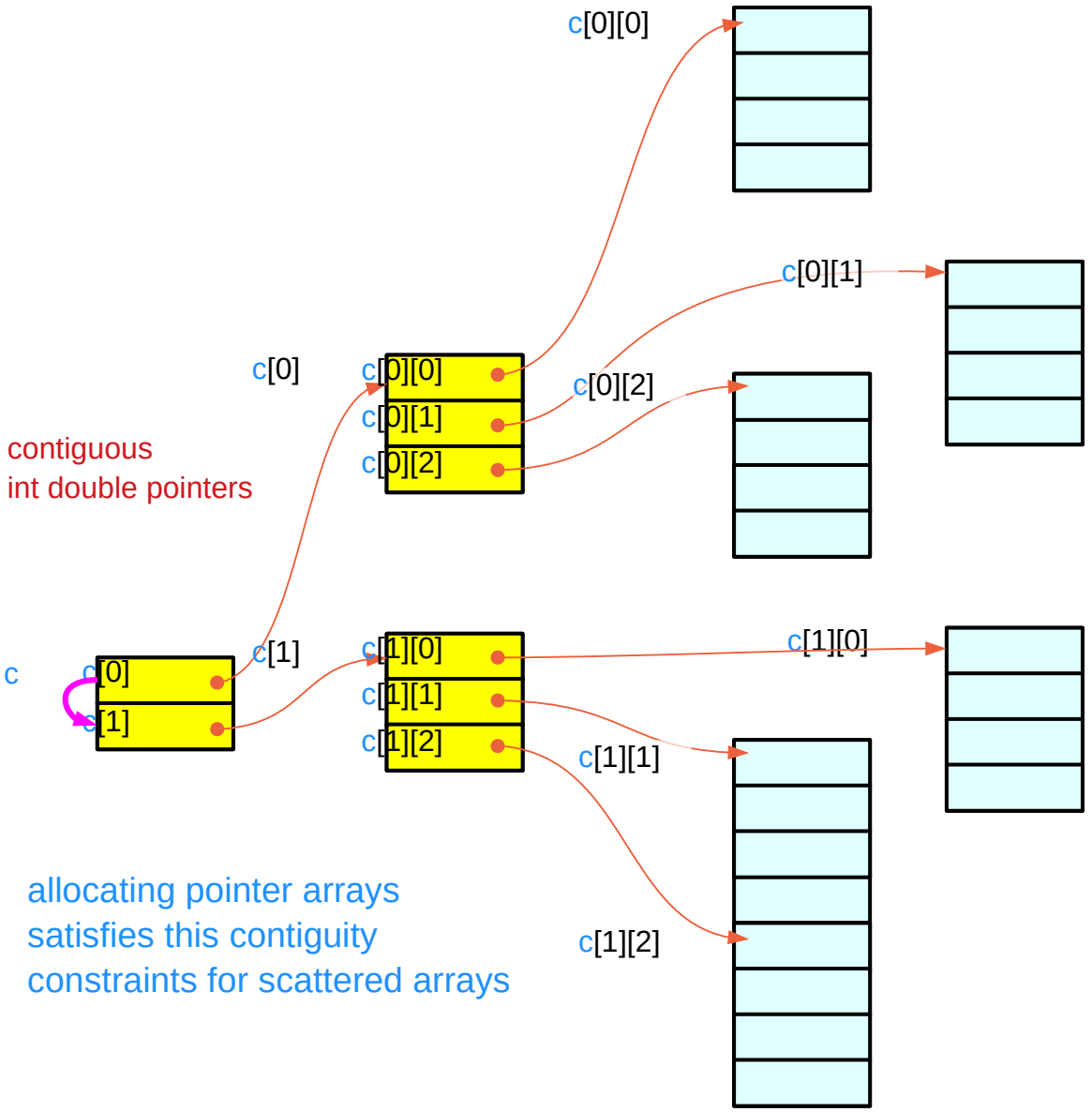
```
c[0][0] =  *(c[0] + 0)
c[0][1] =  *(c[0] + 1)
c[0][2] =  *(c[0] + 2)
c[1][0] =  *(c[1] + 0)
c[1][1] =  *(c[2] + 1)
c[1][2] =  *(c[3] + 2)
```



c[i] ≡ *(c +i)

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

contiguous 1-d array pointers



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