Applications of Pointers (1A)

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

Background

Operator Precedence of * and [ ]

*x[m]	⟺	*(x[m])	[ ] has a higher priority 1 than *
x[m][n]	⟺	(x[m])[n]	Left-to-right associativity
**x	⟺	*(*x)	Right-to-left associativity

(*x)[m][n]	⟺	((*x)[m])[n]
(*x[m])[n]	⟺	(*(x[m]))[n]

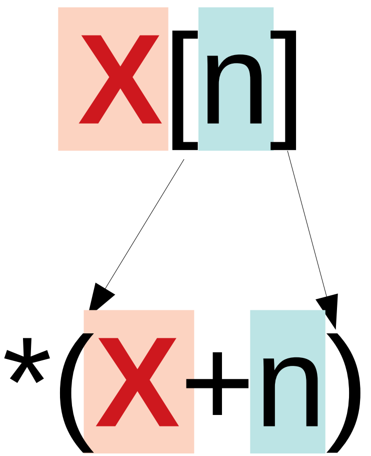
( ) must be used

( ) can be removed

Left to right associative [ ] – recursive indirection

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

*(X+n) ≡ X[n]



Right to left associative *– recursive indirection

*p	⟺	*(p)	p[0]
**p	⟺	*(*(p))	p[0][0]
***p	⟺	*(*(*(p)))	p[0][0][0]

*(X+n) ≡ X[n]

*(X+0) → X[0]

Equivalences

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

&p[i]	⟺	(p+i)
&p[i][j]	⟺	(*(p+i)+j)
&p[i][j][k]	⟺	(*(*(p+i)+j)+k)

p[0]	⟺	*p
p[i][0]	⟺	*p[i]
p[i][j][0]	⟺	*p[i][j]

&p[0]	⟺	p
&p[i][0]	⟺	p[i]
&p[i][j][0]	⟺	p[i][j]

Operator Precedence

Precedence	Operator	Description	Associativity
1	++ -- () [] . -> (type){list}	Suffix/postfix increment and decrement Function call Array subscripting Structure and union member access member access through pointer Compound literal(C99)	Left-to-right ((([m])[n])[p])
2	++ -- + - ! ~ (type) * & sizeof _Alignof	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT Type cast Indirection (dereference) Address-of Size-of Alignment requirement(C11)	Right-to-left *(*(*(*x)))

https://en.cppreference.com/w/c/language/operator_precedence

Pointer Arrays for recursive indirections

1-d array of (int **) pointers	int** c [2];
1-d array of (int *) pointers	int* b [2*3];
1-d array of (int)	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];
```

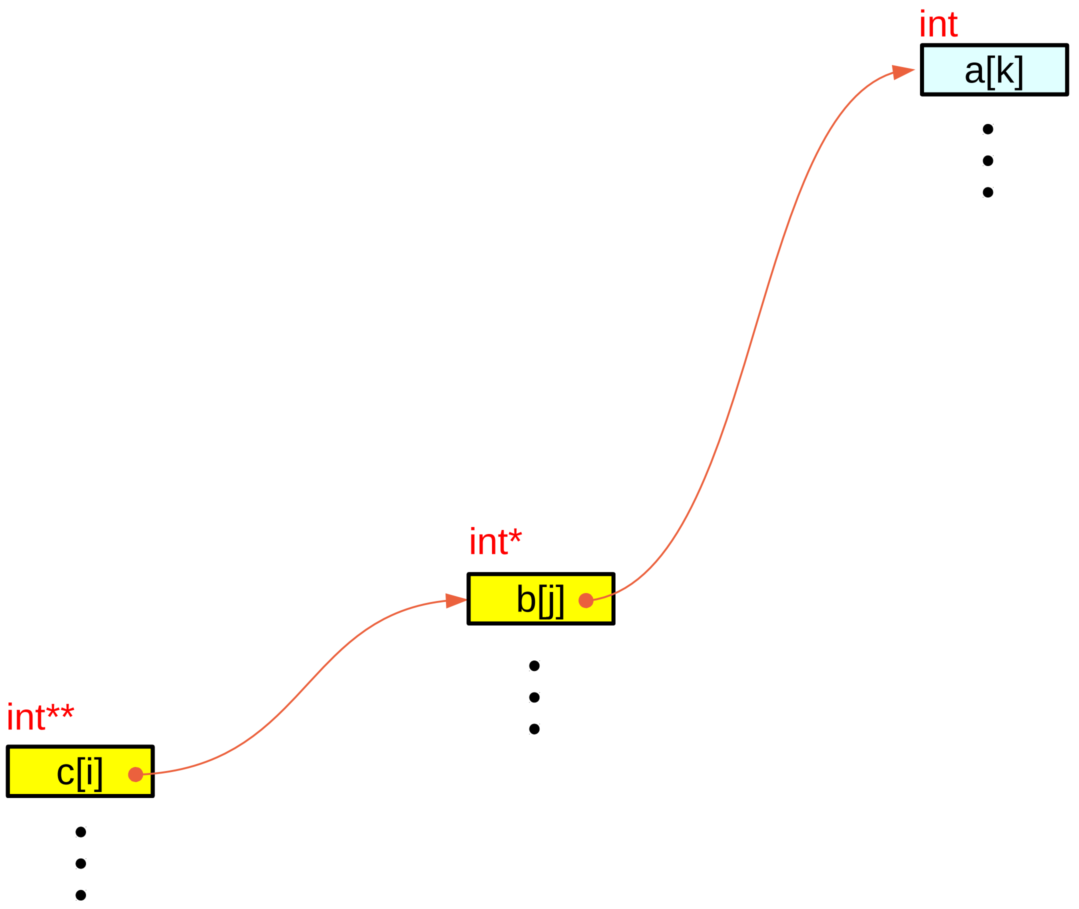
c[i][j][k]	:: int
c[i][j]	:: int *
c[i]	:: int **

c[i] ← &b[i*3]
b[j] ← &a[j*4]

Using pointer arrays

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

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



conditions

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

Array of Pointers

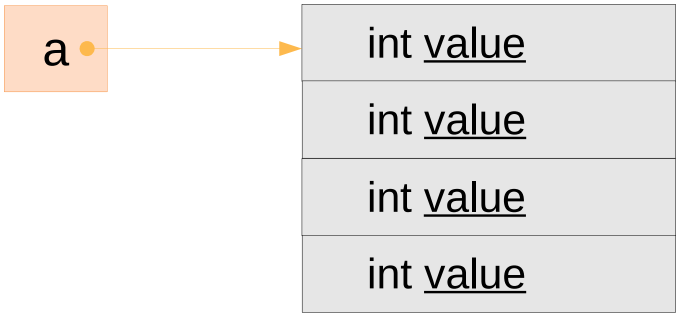
```
int      a [4];

int *    b [4];
```

No. of elements = 4

int a [4]

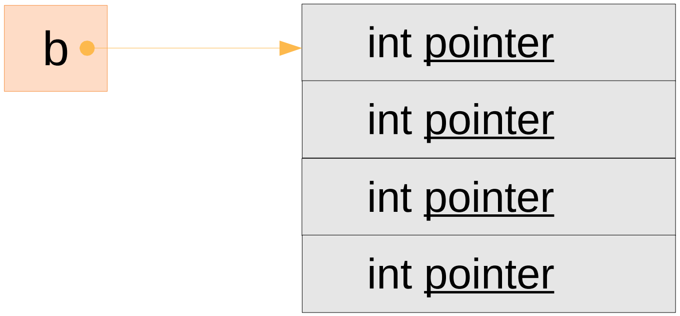
Type of each element



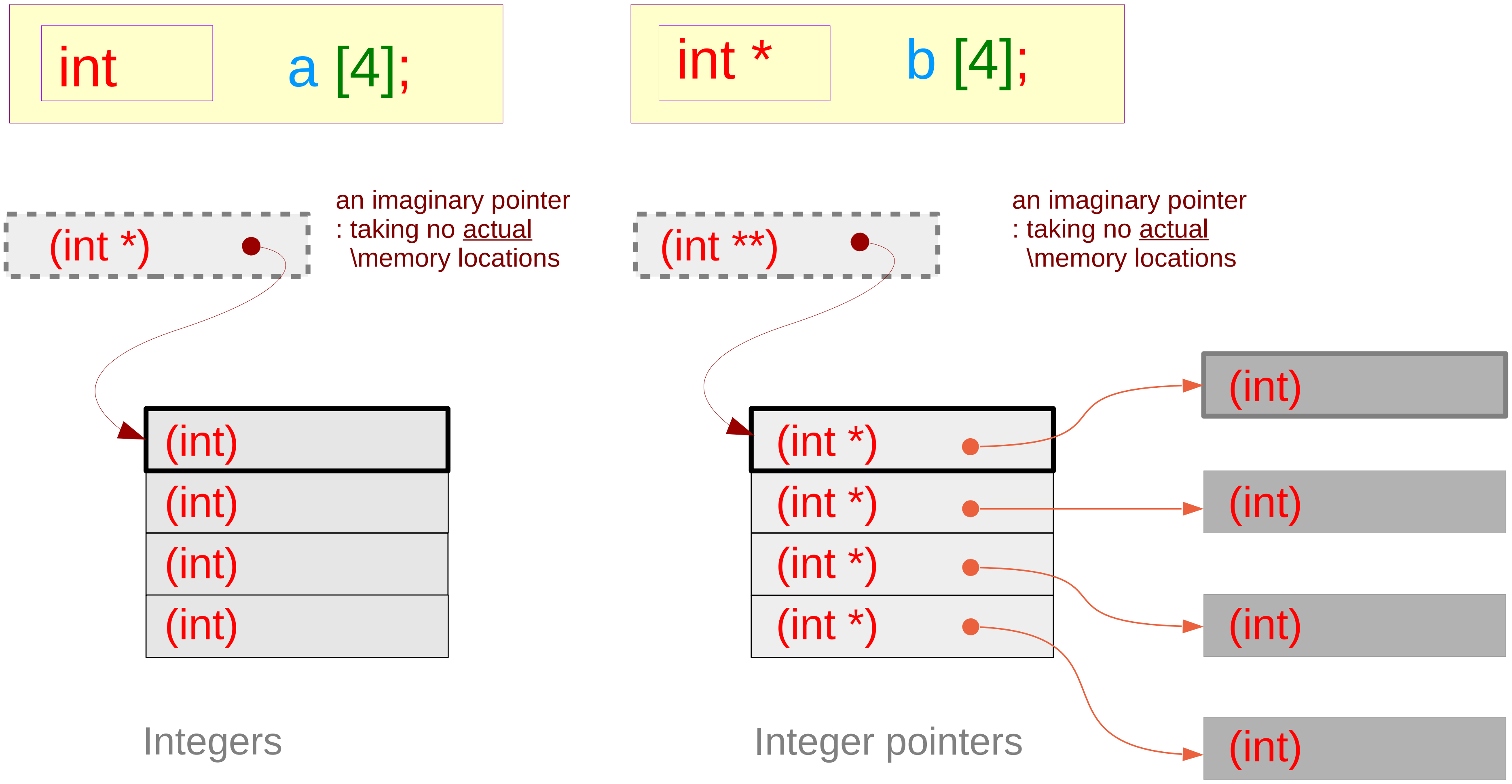
No. of elements = 4

int * b [4]

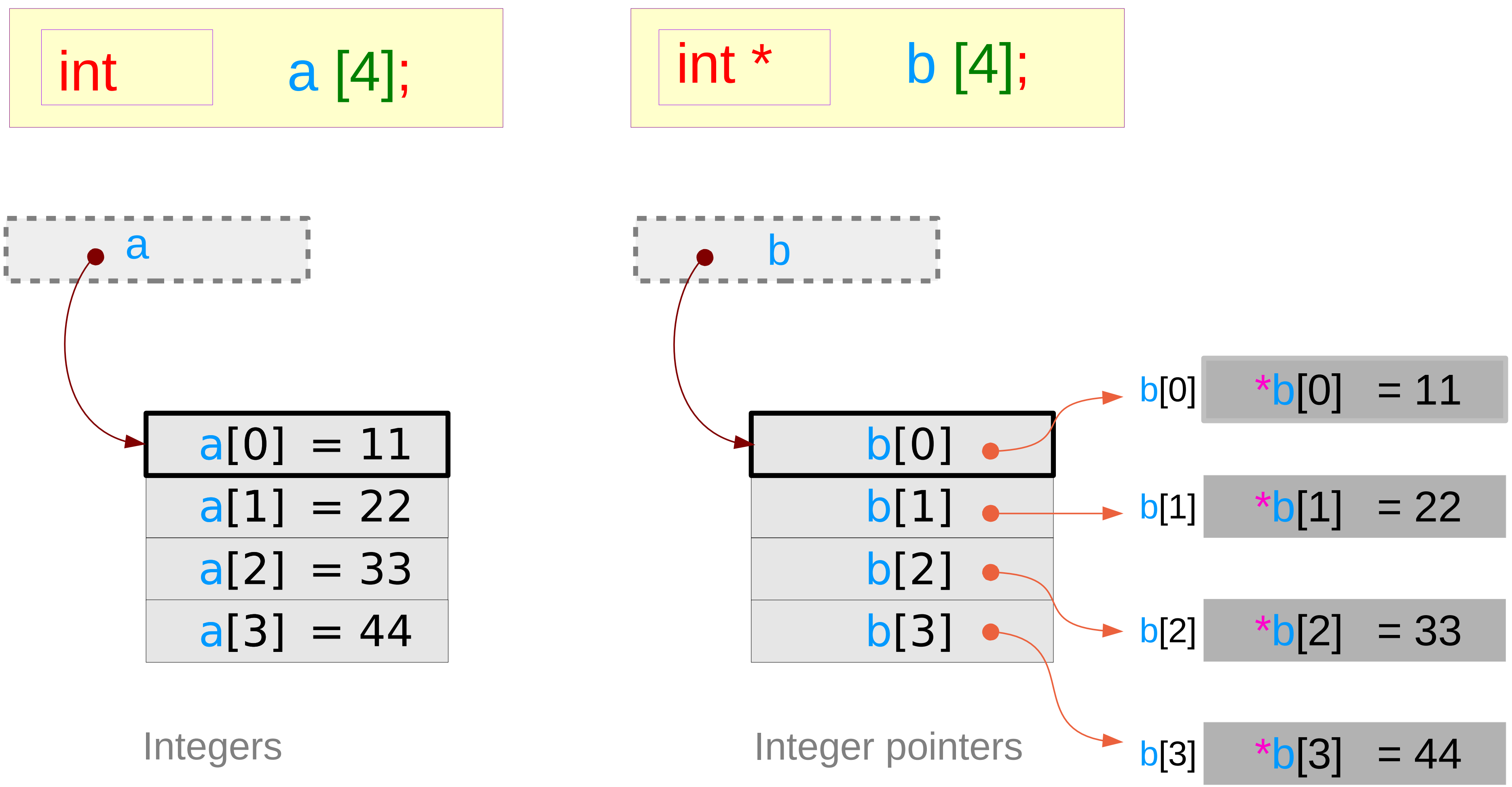
Type of each element



Array of Pointers – a type view



Array of Pointers – a variable view

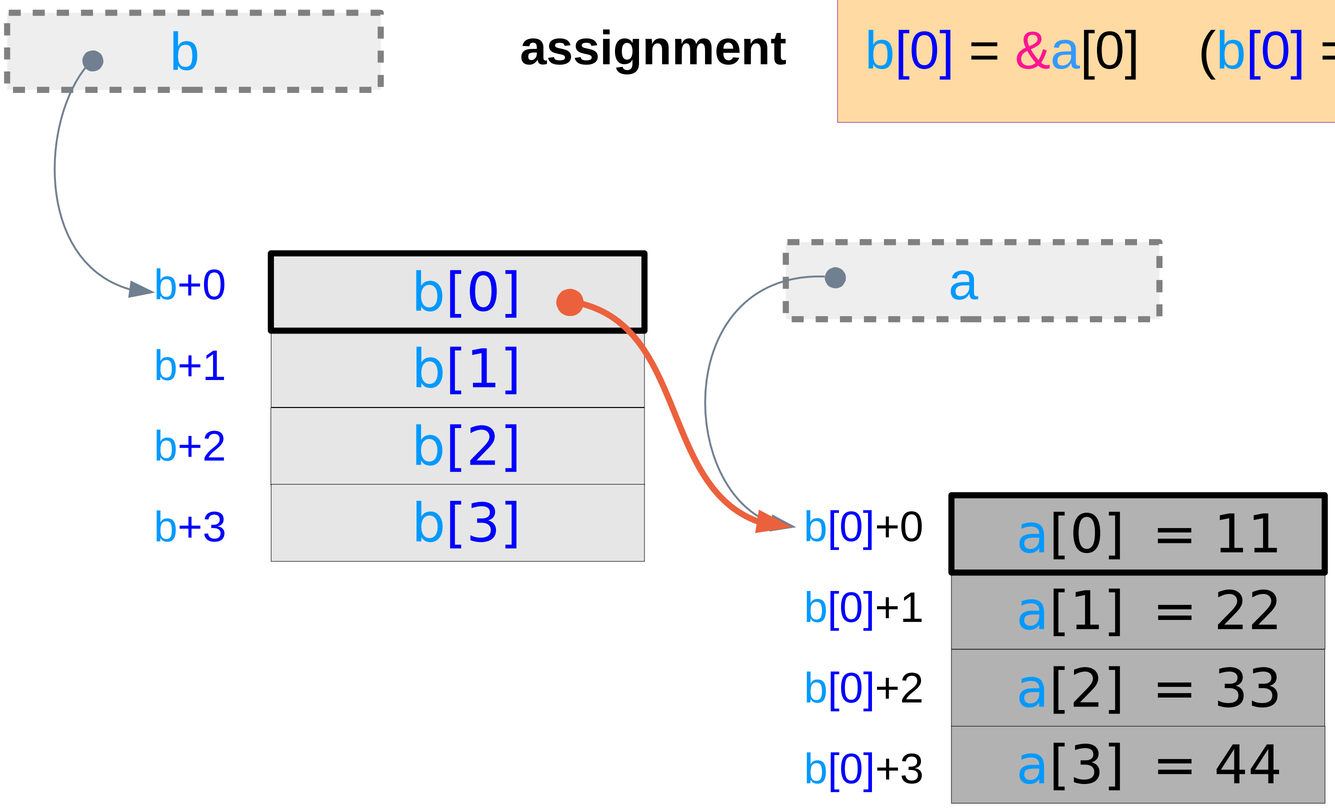


Array of Pointers – assigning a **1-d** array name

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

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

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



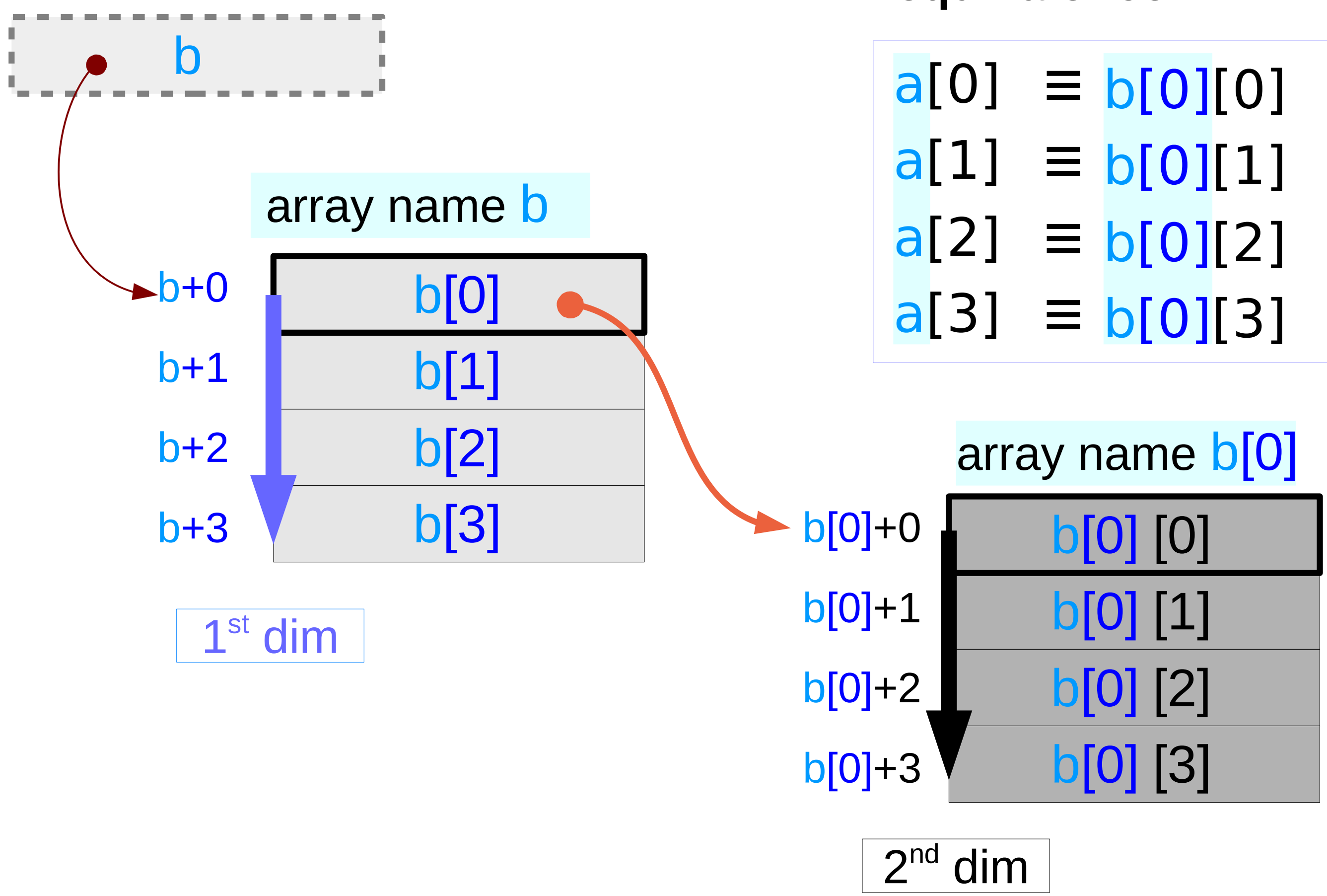
Array of Pointers – extending a dimension

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

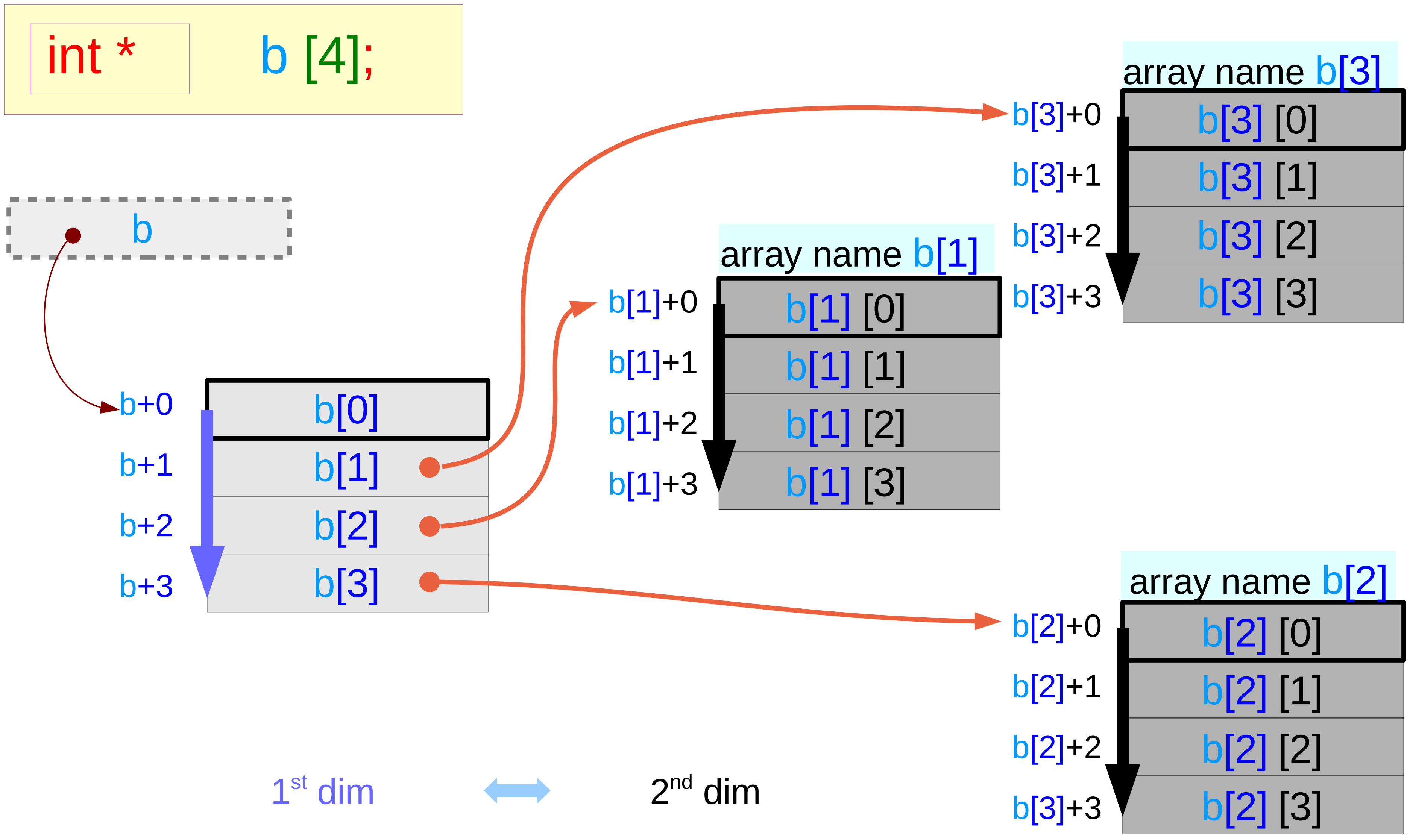
assignment b[0] = a

equivalence

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 a 1-d array – assigning 1-d array names



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

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

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

constraint : contiguous b[i]

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

Assignments

```
b[0] = &a[0*4]    (b[0] = a+  0)
b[1] = &a[1*4]    (b[1] = a+  4)
b[2] = &a[2*4]    (b[2] = a+  8)
b[3] = &a[3*4]    (b[3] = a+12)
```

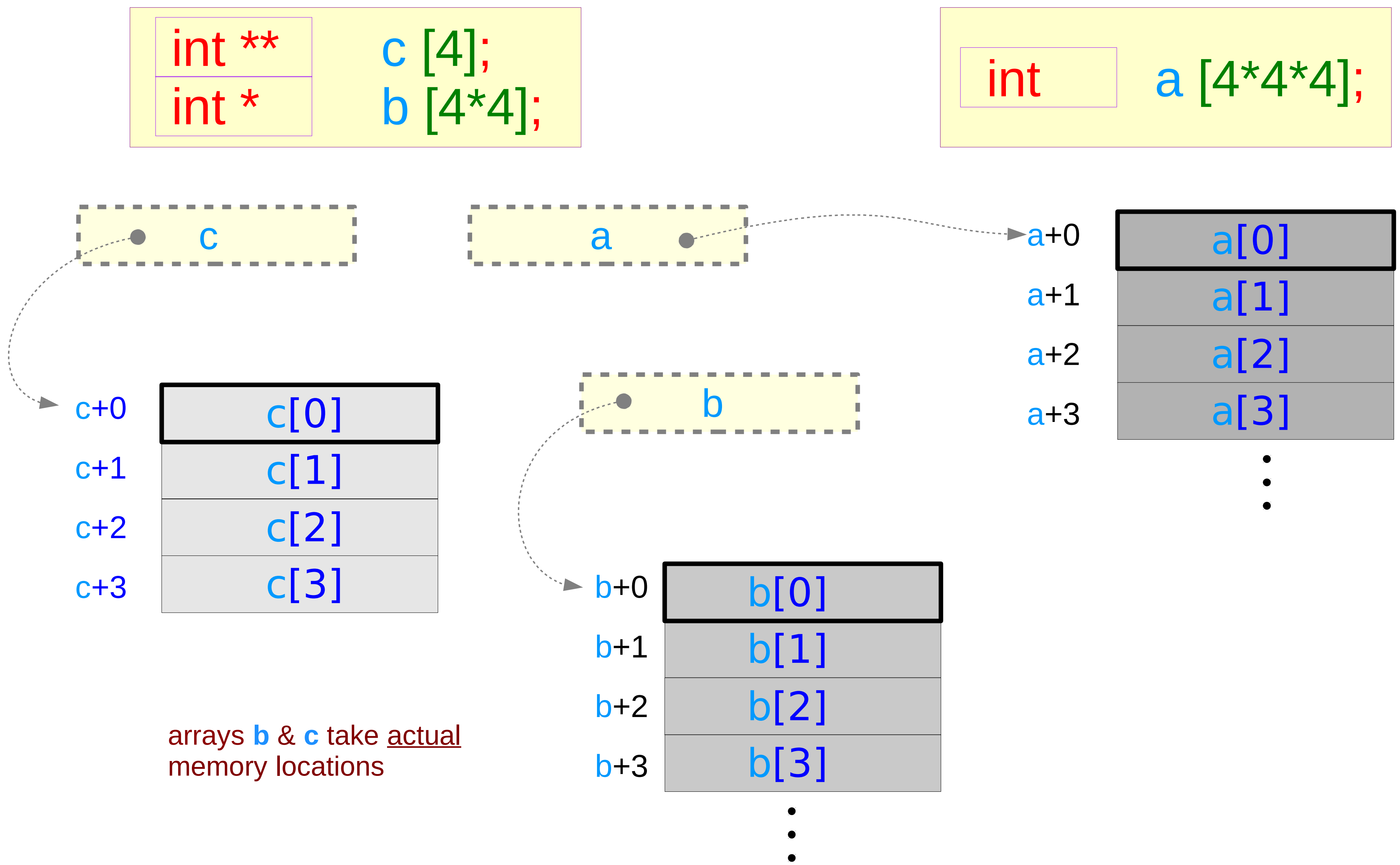
```
b[i][j]      ≡ *(*(b+i) +j)
   ⇕               ⇕
a[i*4+j]  ≡ *(a+i*4 +j)
```

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

constraint : contiguous a[i]

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

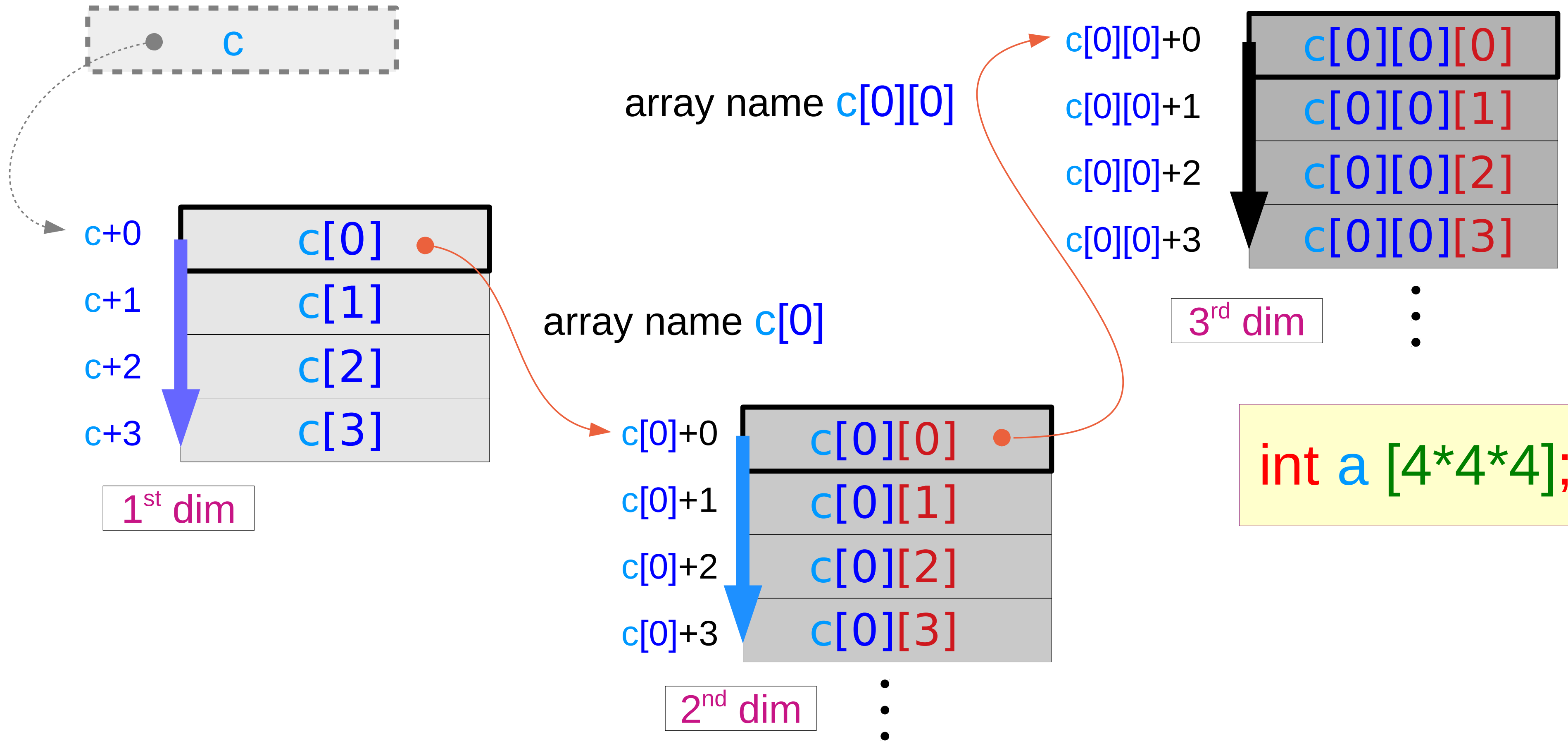
3-d access of a **1-d** array – using pointer arrays **b**, **c**



3-d access of a **1-d** array – assigning **1-d** array names

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

```
c[0] = b;      (c[0] = &b[0];)
b[0] = a;      (b[0] = &a[0];)
```




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

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

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

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

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

Assignments

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

Initialization of
pointer arrays b and c

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

c[i][j][k] ≡
a[i*M*N+j*N+k] ≡
a[(i*M+j)*N +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 – using pointer arrays

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

$\Rightarrow$

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

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

*(c+i) = b+*g*(i)   c[i] =b+4*i

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

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

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

c[i][j] ⟺ b[*g*(i)+j]

b[j][k] ⟺ a[*f*(j)+k]

c[i][j][k] ⟺ a[*f*(*g*(i)+j)+k]

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

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

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

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

on a 32-bit machine | on a 64-bit machine

sizeof(int) =
4 bytes

on a 32-bit machine | on a 64-bit machine

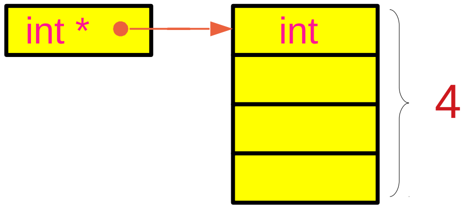
sizeof(c) =
4*sizeof(int **)
sizeof(b) =
4*4*sizeof(int *)

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

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

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

b[i], j=0,1, … , 5

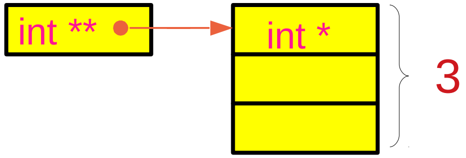



```
b[0] = &a[0*4];
b[1] = &a[1*4];
b[2] = &a[2*4];
b[3] = &a[3*4];
b[4] = &a[4*4];
b[5] = &a[5*4];
```

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

b[j] get the address of the every 4th element of a

c[i], i=0,1




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

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

c[i] get the address of the every 3rd element of b

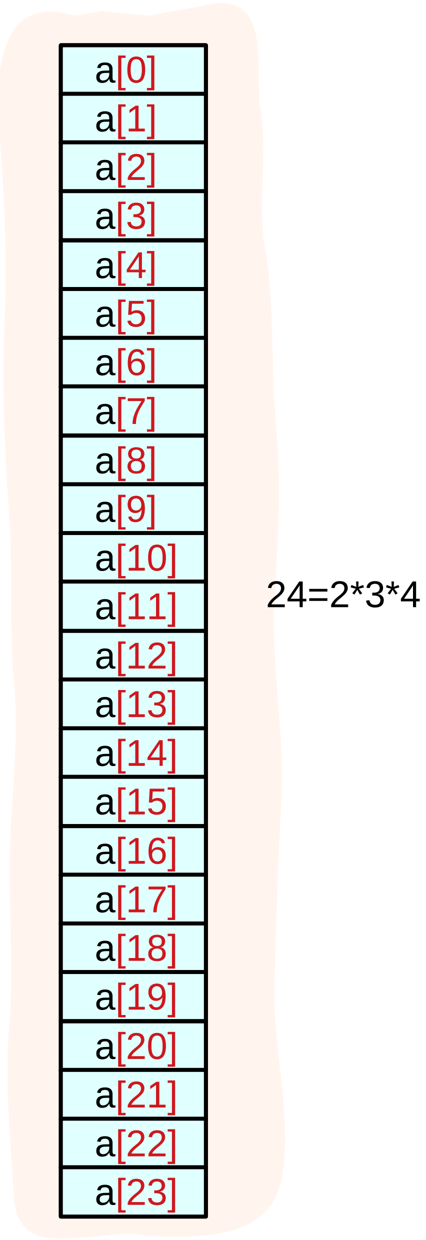
Accessing an int array **a** as a **1-d** array

```
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] ;

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

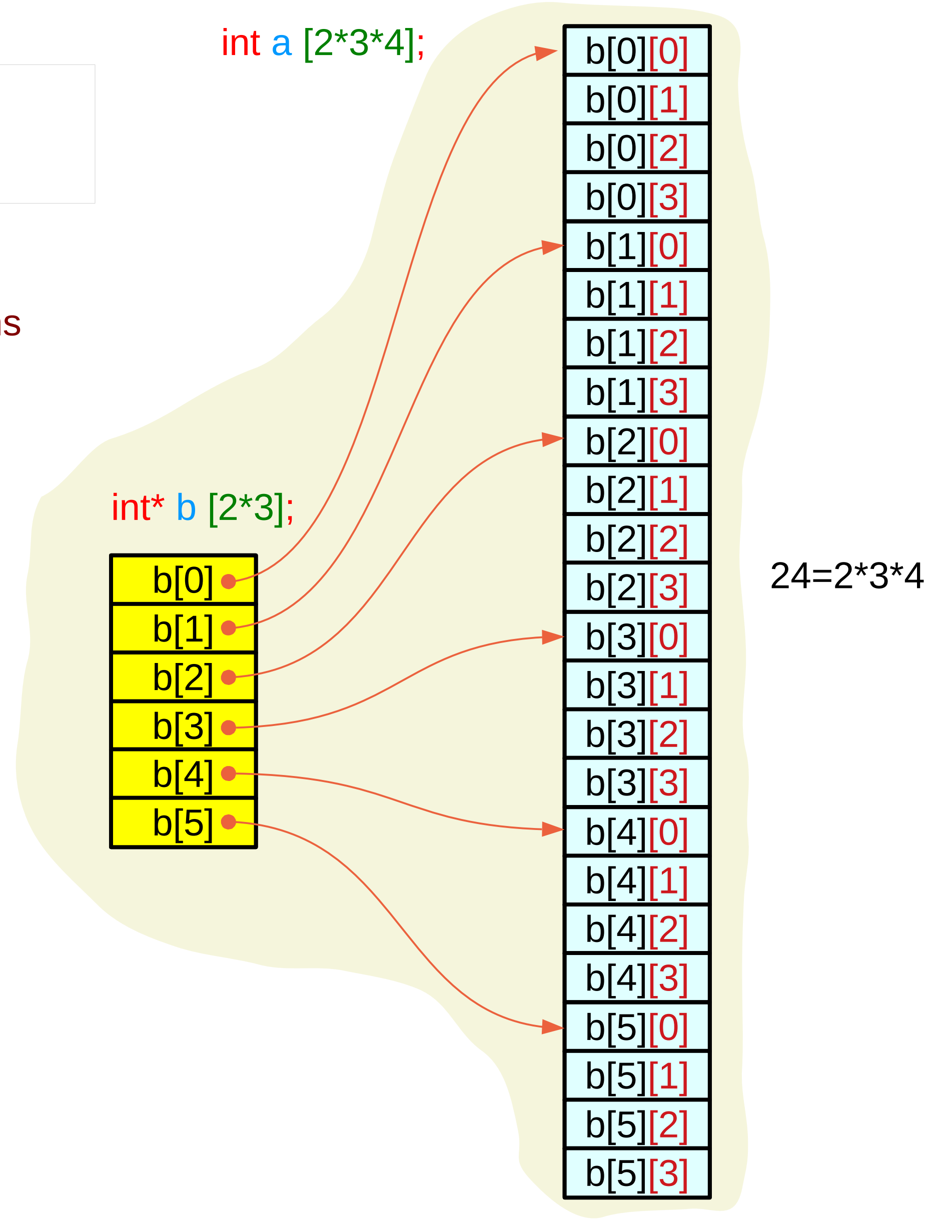
```
int     a [2*3*4];
int*    b [2*3];
```

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

b take actual memory locations

b [j][k]

j = 0,1,2,3,4,5
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 an int array **a** as a **3-d** array

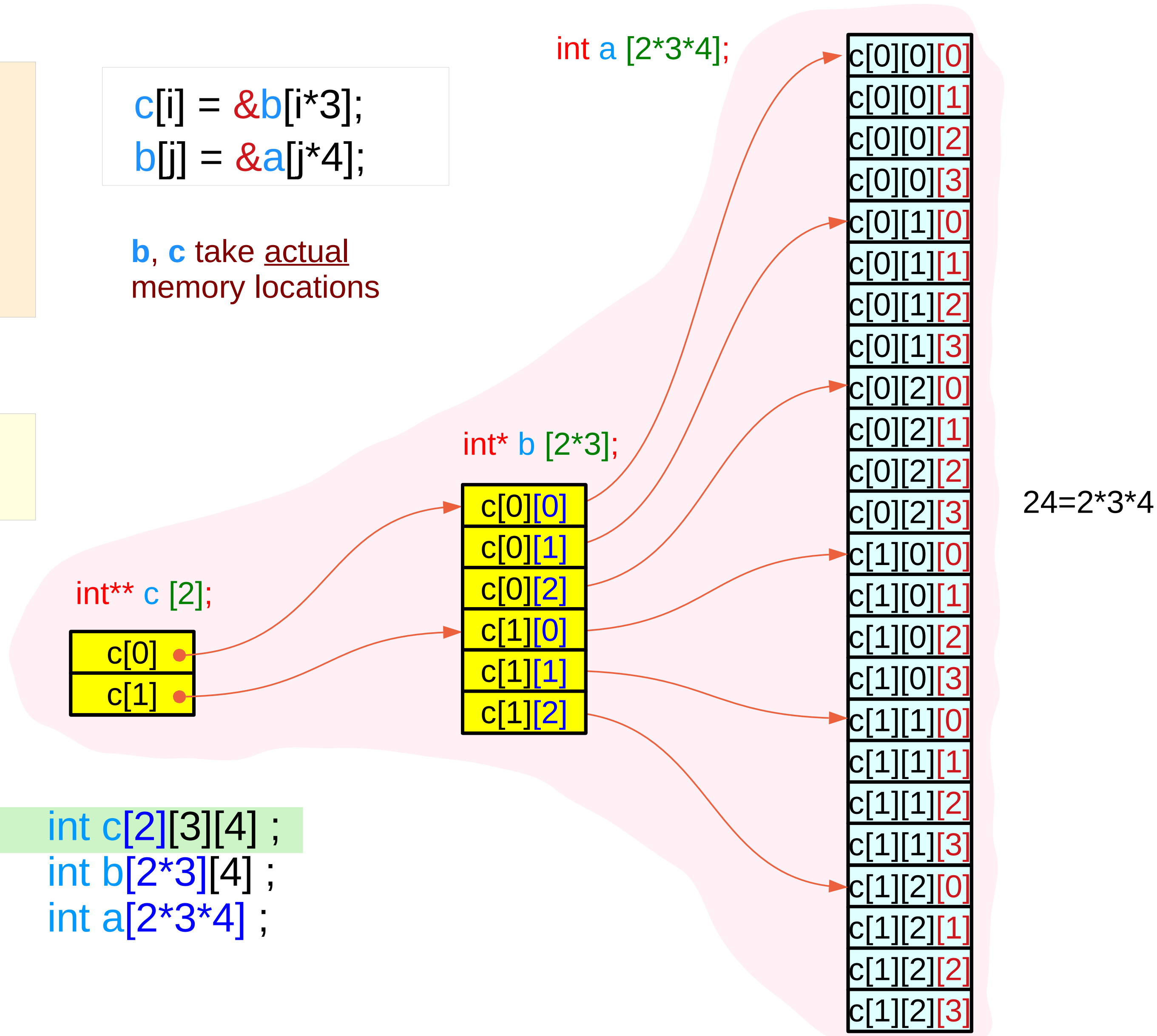
```
int    a [2*3*4];
int*   b [2*3];
int**  c [2];
```

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

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

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

b, **c** take <u>actual</u> 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 **non-contiguous 1-d** arrays as a **3-d** array

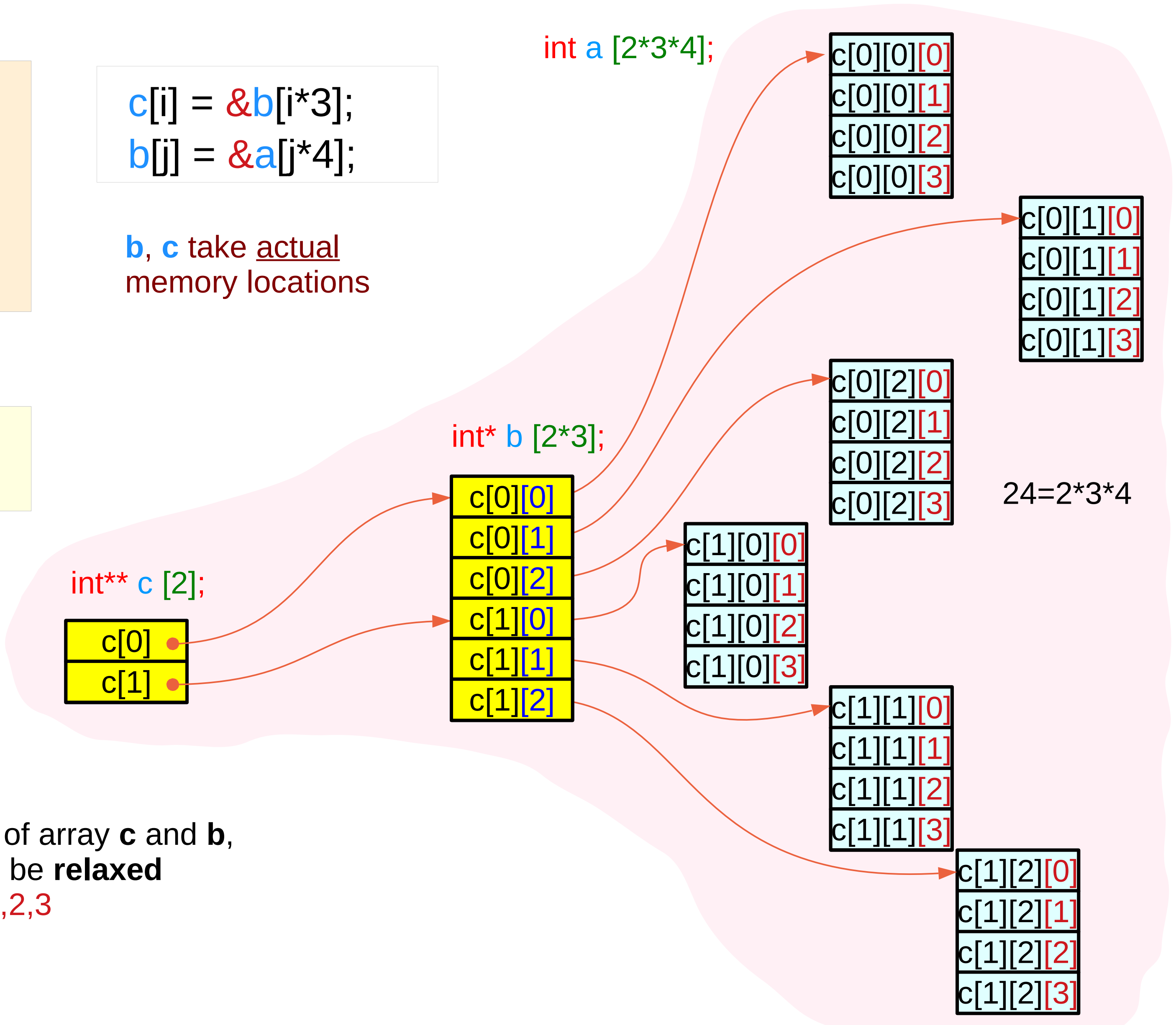
```
int     a [2*3*4];
int*    b [2*3];
int**   c [2];
```

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

b, **c** take actual memory locations

c [i][j][k]

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

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=1	i*3*4 = 0 i*3*4 = 12
M=3	j=0 j=1 j=2	j*4 = 0 j*4 = 4 j*4 = 8
N=4	k=0 k=1 k=2 k=3	k*1= 0 k*1= 1 k*1= 2 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]

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*

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*

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]

General Requirements

Pointer Array Implementation

Assignments and their Equivalent Relations

c[i] ⇐ &b[i*3]
b[j] ⇐ &a[j*4]

[2][3][4]

c[i] ≡ (b+i*3);
b[j] ≡ (a+j*4);

extending a dimension — *(c[i]+j) = *(b+i*3+j);
extending a dimension — *(b[j]+k) = *(a+j*4+k);

c[i][j] ≡ b[i*3+j]
b[j][k] ≡ a[j*4+k]

extending a dimension — *(c[i][j]+k) = *(b[i*3+j]+k)
substitute j ← (i*3+j) — *(b[i*3+j]+k) = *(a+(i*3+j)*4+k)

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

assignments

equivalences

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]

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

≠

a pointer value and its address cannot be the same

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

≠

&c[i] = c+i

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]

Initialization of pointer arrays – a general case

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

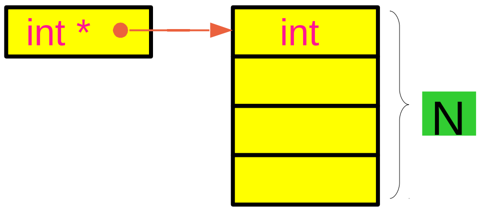
```
int*     b [L*M];
int**    c [L];
```

pointer arrays b, c

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

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

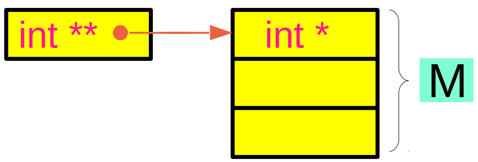
```
b[j] = &a[j*N];
j=0,..., L*M-1
```



b[j] get the address of the every N[th] 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 M[th] 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]

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]       ;
```

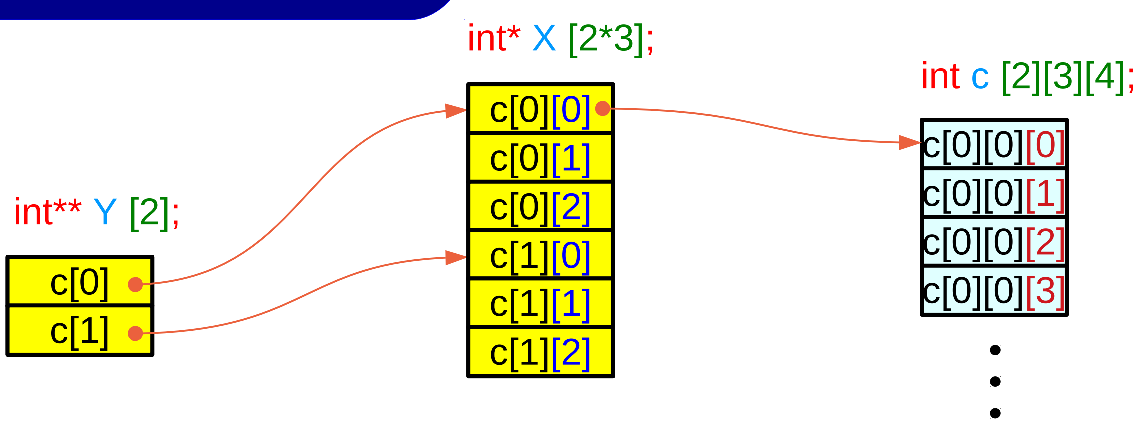
Recursive Indirections – Substitution Analysis

```
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



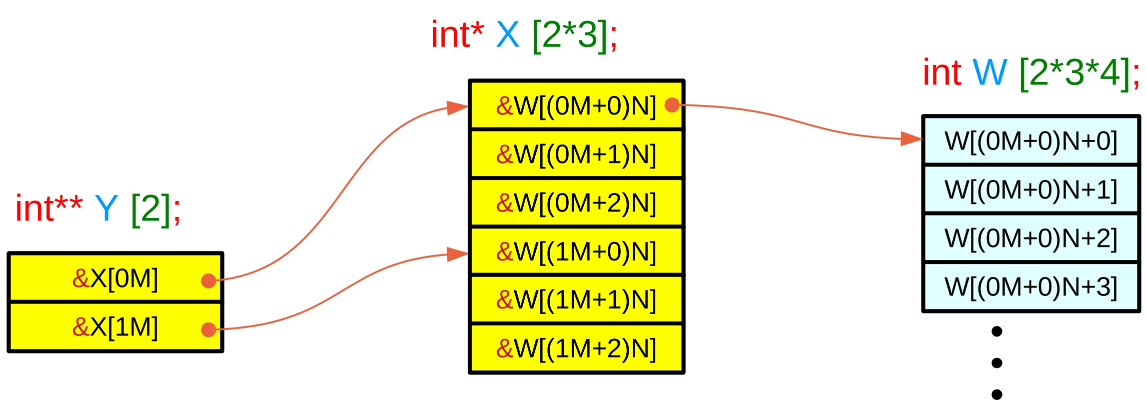
Recursive Indirections – one continuous int array W

&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]
```



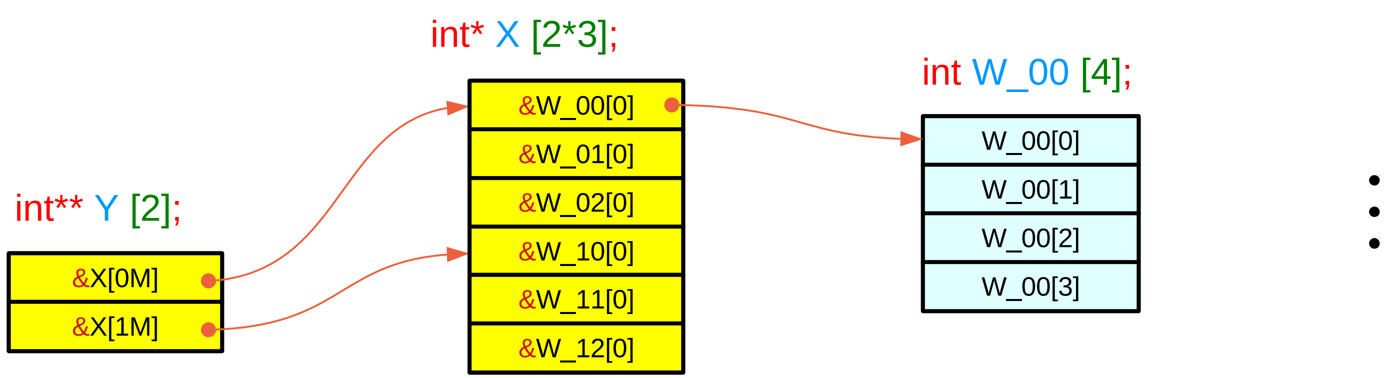
Recursive Indirections – non-contiguous 1-d arrays W_ij

&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];

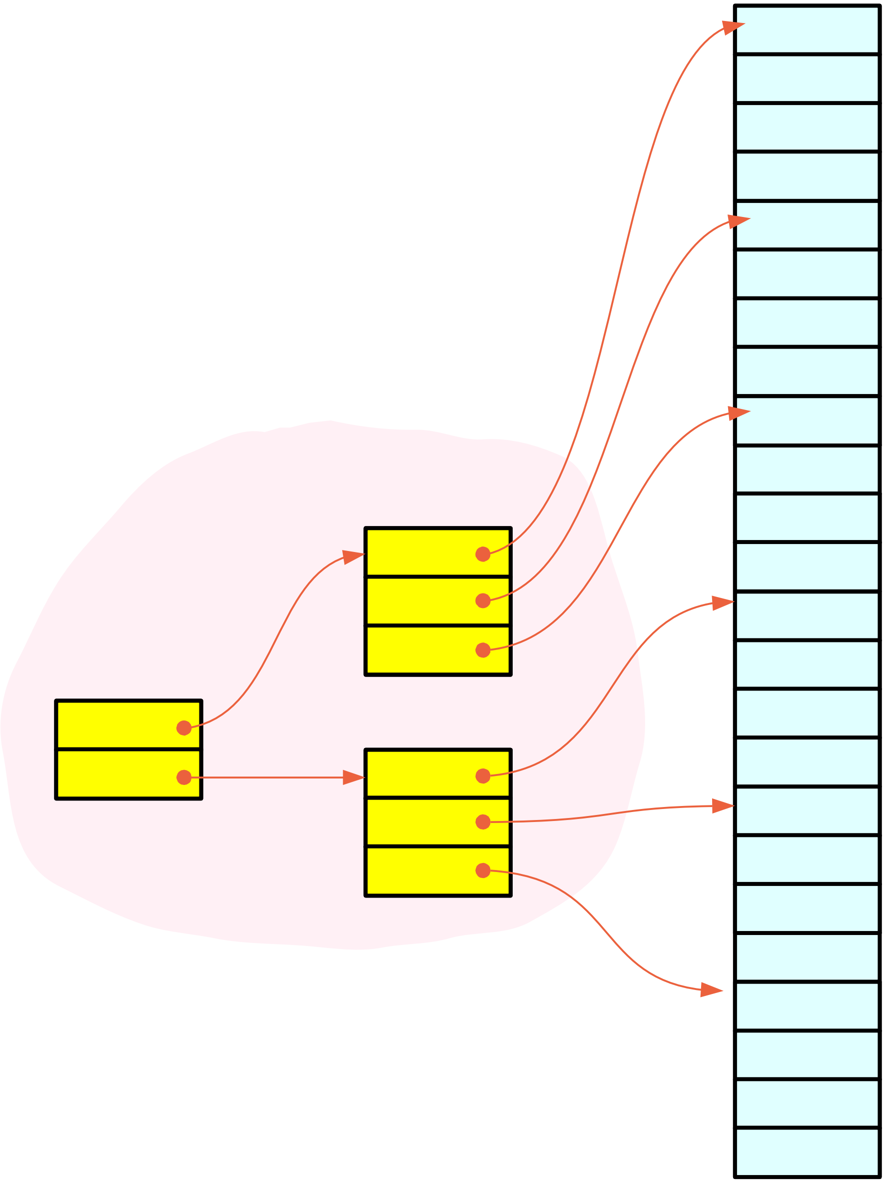
Pointer Arrays and Contiguity

Using pointer arrays

int * [N], int ** [M], int *** [L], …

Pointer array approach for 3-d access patterns

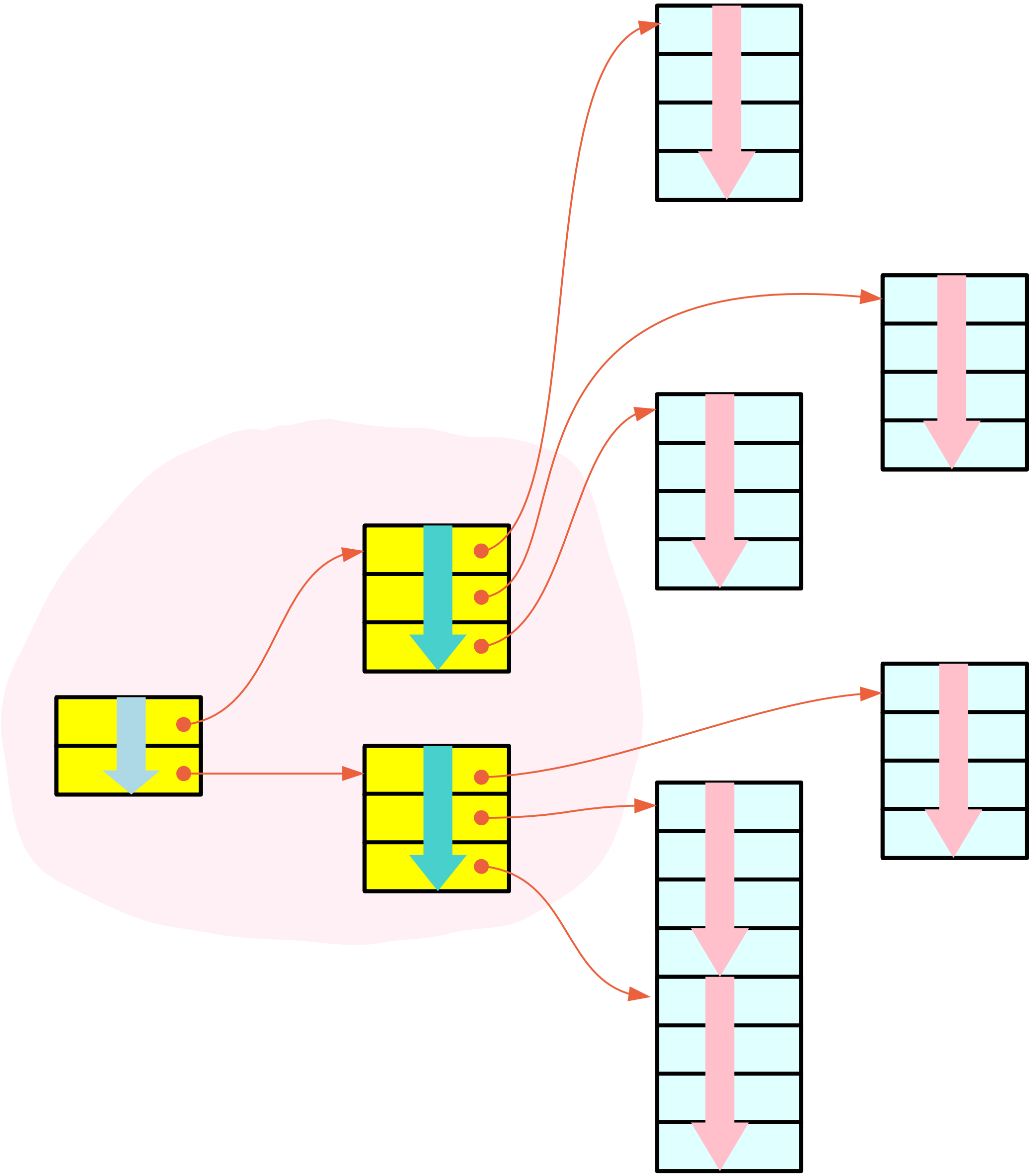
A programmer manually allocates
memory locations for pointer arrays



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)

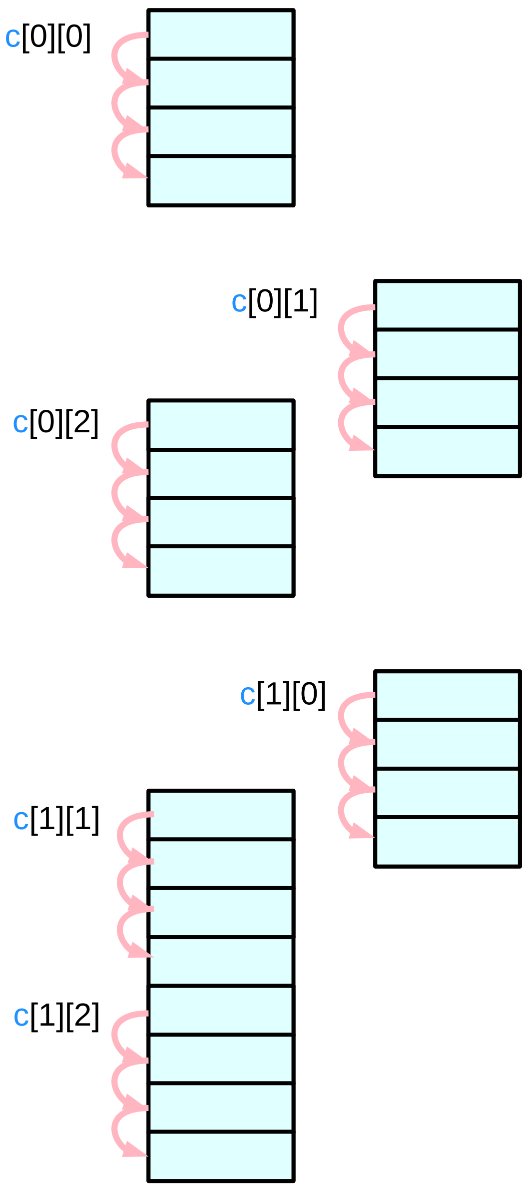
contiguous **1-d** array elements	int
contiguous **1-d** arrays	int [4]
contiguous **1-d** array pointers	int (*) [4]

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

c[i][j][k] ≡ *(c[i][j] +k)

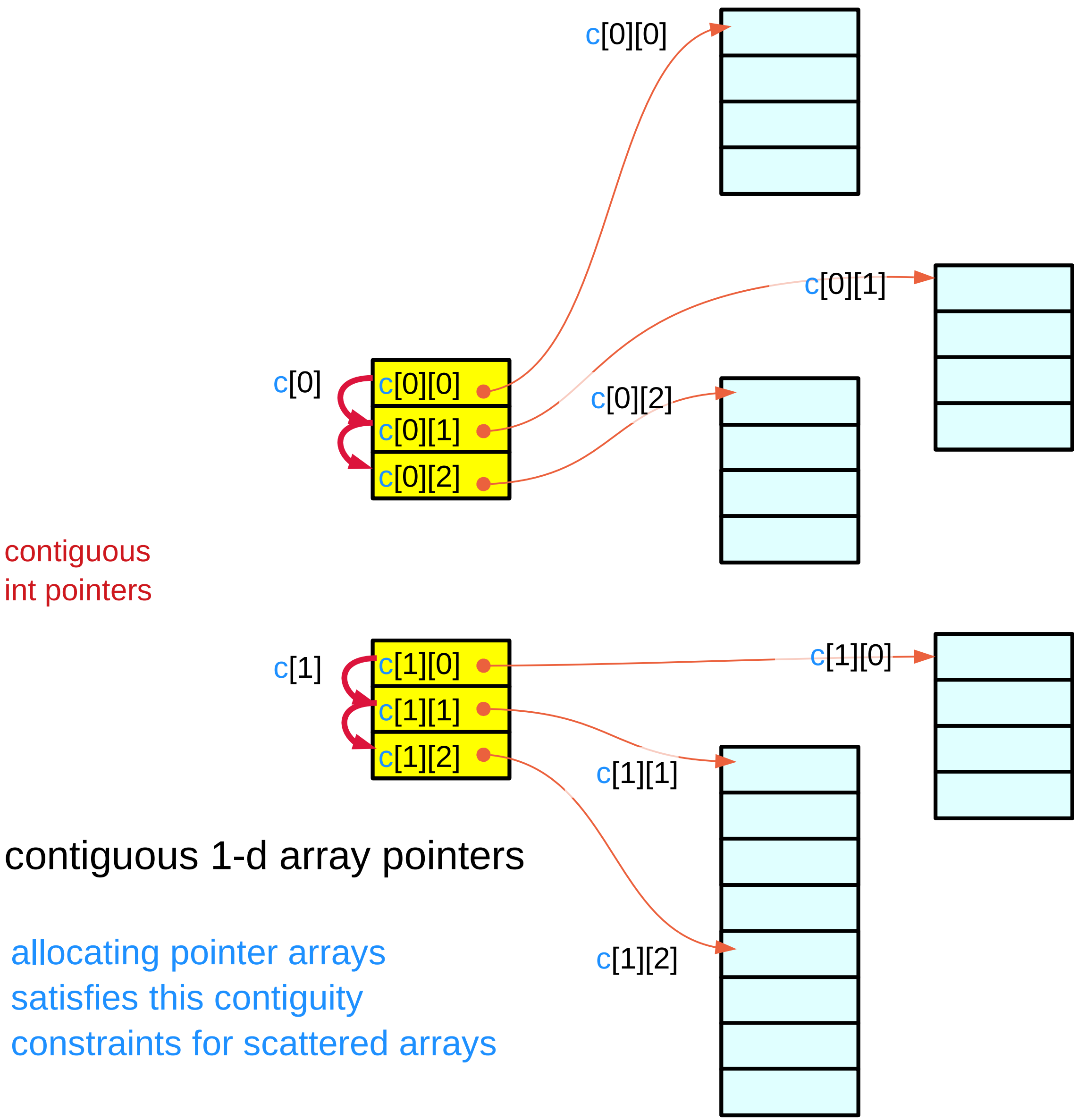
c[0][0][0] =	*(c[0][0] + 0)
c[0][0][1] =	*(c[0][0] + 1)
c[0][0][2] =	*(c[0][0] + 2)
c[0][0][3] =	*(c[0][0] + 3)
c[0][1][0] =	*(c[0][1] + 0)
c[0][1][1] =	*(c[0][1] + 1)
c[0][1][2] =	*(c[0][1] + 2)
c[0][1][3] =	*(c[0][1] + 3)
⋮	⋮

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)



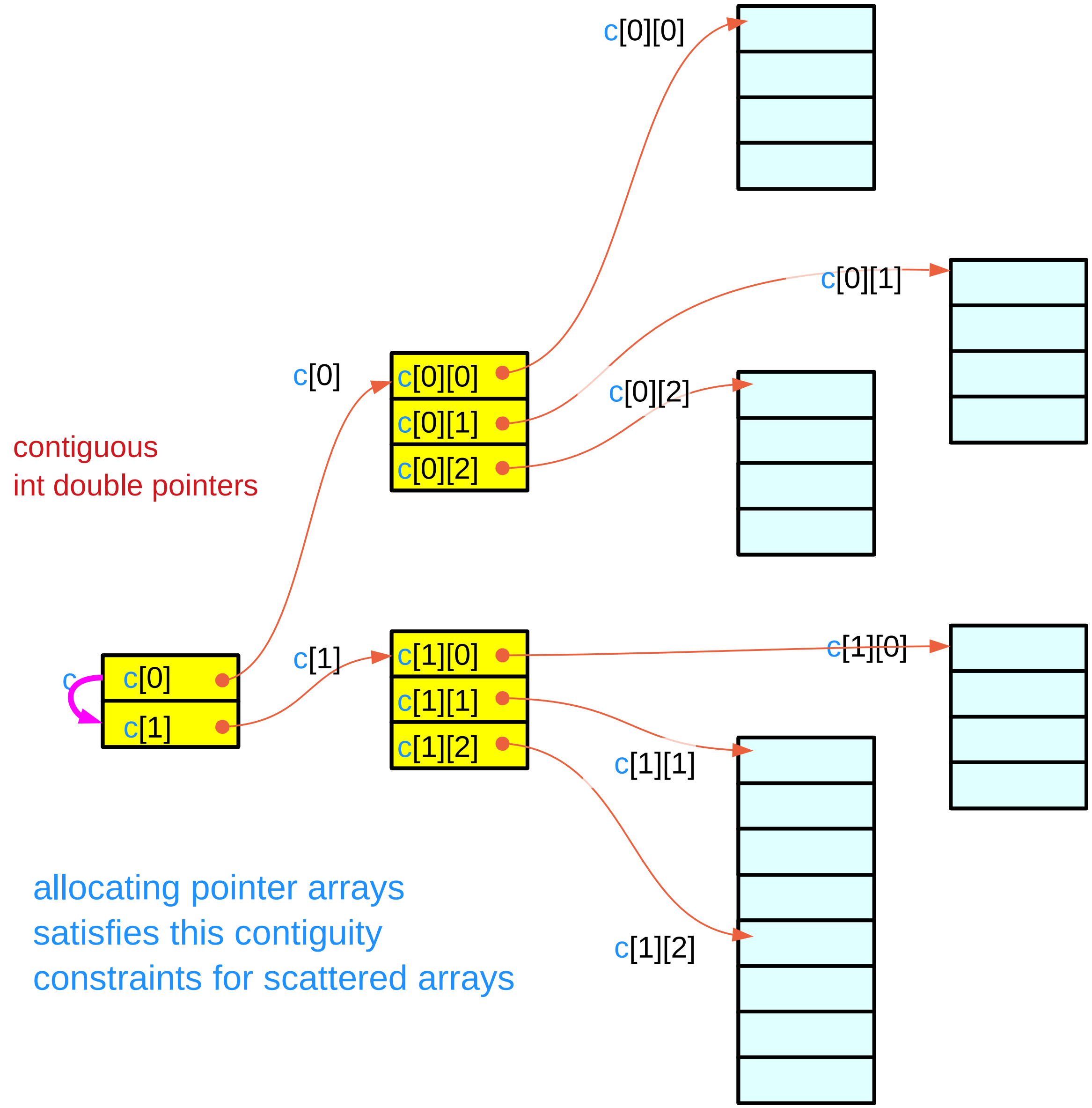
contiguous 1-d array pointers

allocating pointer arrays
satisfies this contiguity
constraints for scattered arrays

c[i] ≡ *(c +i)

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

contiguous 1-d array pointers



allocating pointer arrays
satisfies this contiguity
constraints for scattered arrays

References

[1] Essential C, Nick Parlante
[2] Efficient C Programming, Mark A. Weiss
[3] C A Reference Manual, Samuel P. Harbison & Guy L. Steele Jr.
[4] C Language Express, I. K. Chun