Applications of Pointers (1A)

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]

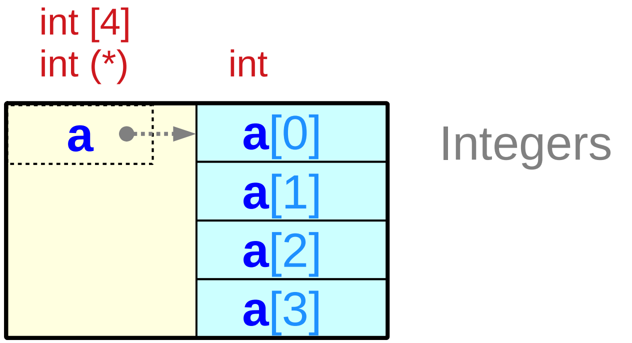
there are 4 elements

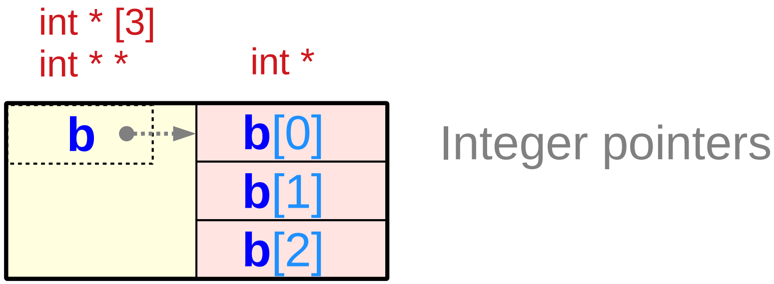
the type of each element: an integer

int * | b | [3]

there are 3 elements

the type of each element: a pointer an integer



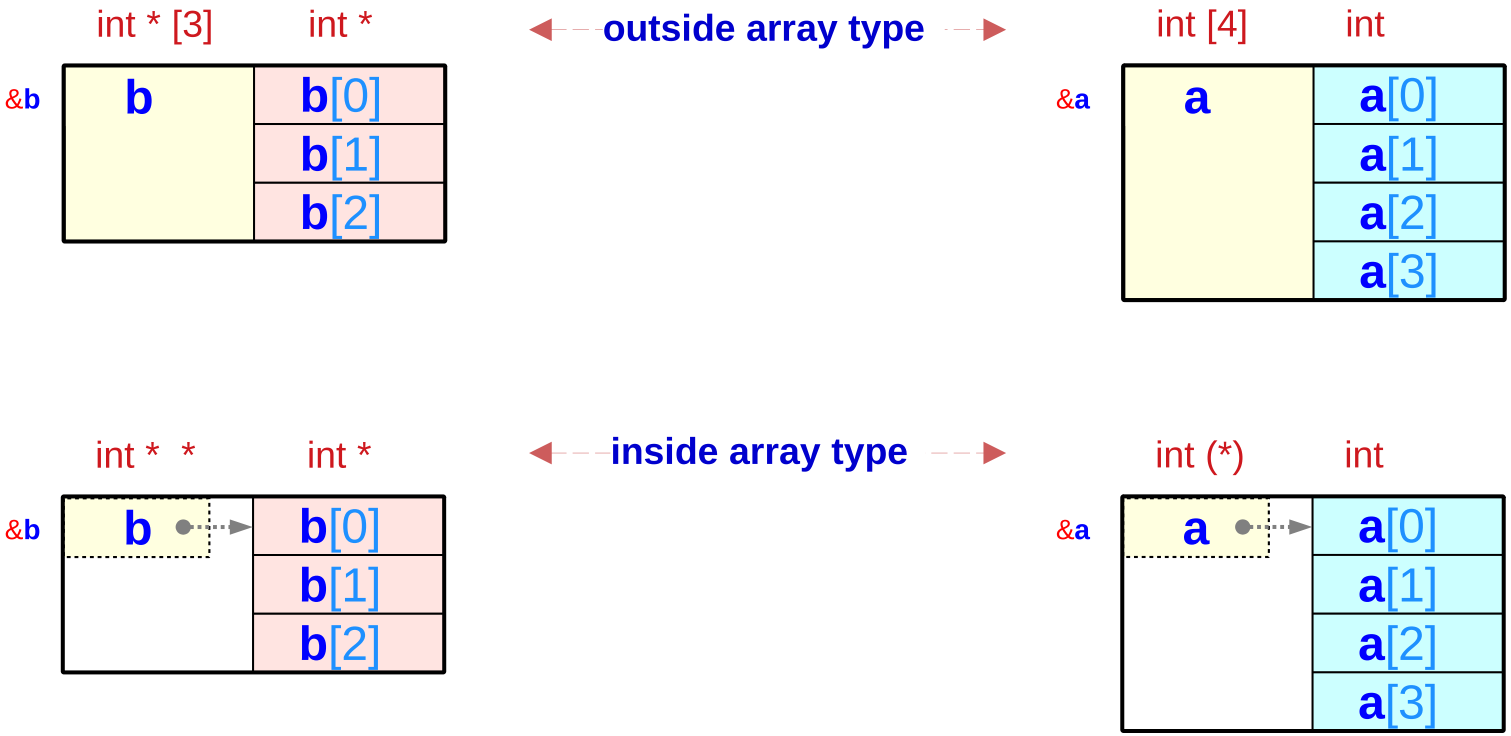


Array of Pointers – a type view

`int *    b [3] ;`

Pointer Array

`int    a [4] ;`



int * [3]   int *

&b b b[0] b[1] b[2]

outside array type

int [4]   int

&a a a[0] a[1] a[2] a[3]

int * *   int *

&b b b[0] b[1] b[2]

inside array type

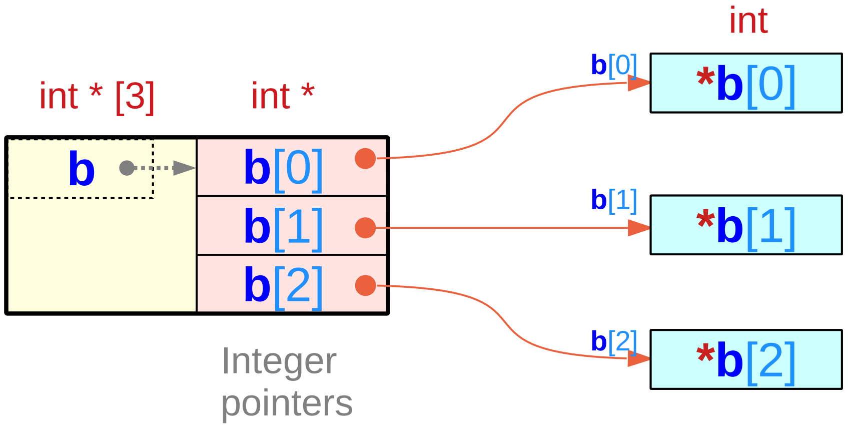
int (*)   int

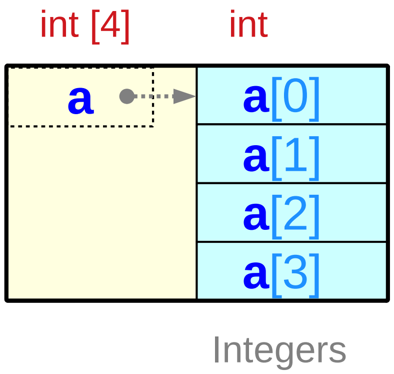
&a a a[0] a[1] a[2] a[3]

Array of Pointers – a variable view

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

int a [4] ;





Assigning a **1-d** array name

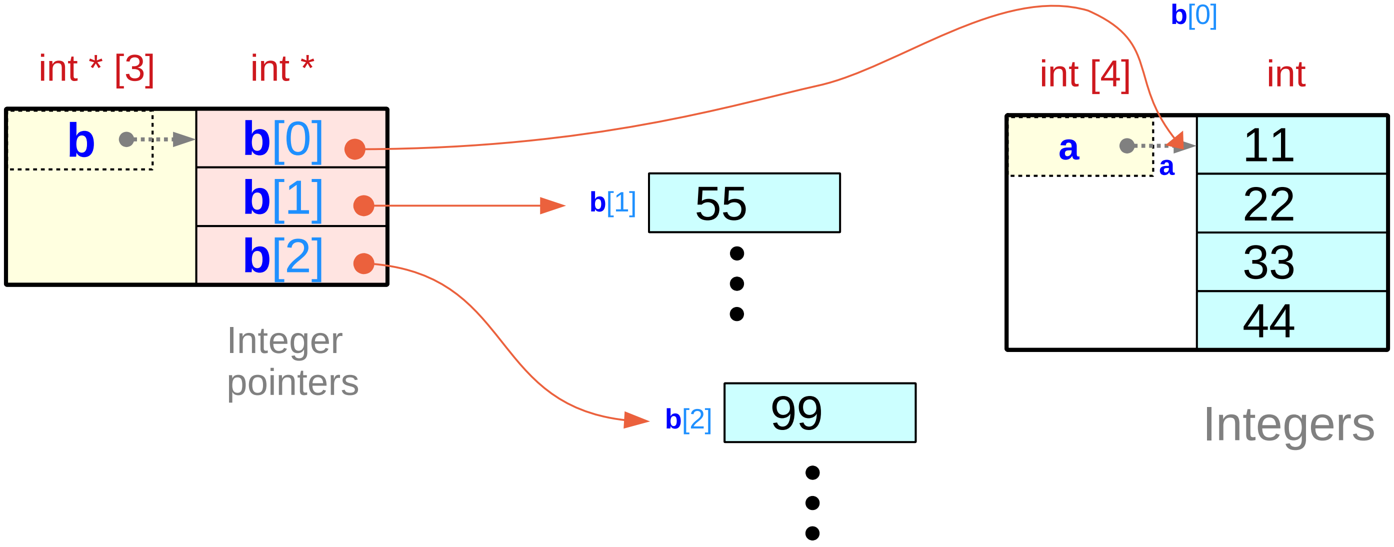
int * **b** [3] ;

Pointer Array

int **a** [4] ;

assignment

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



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

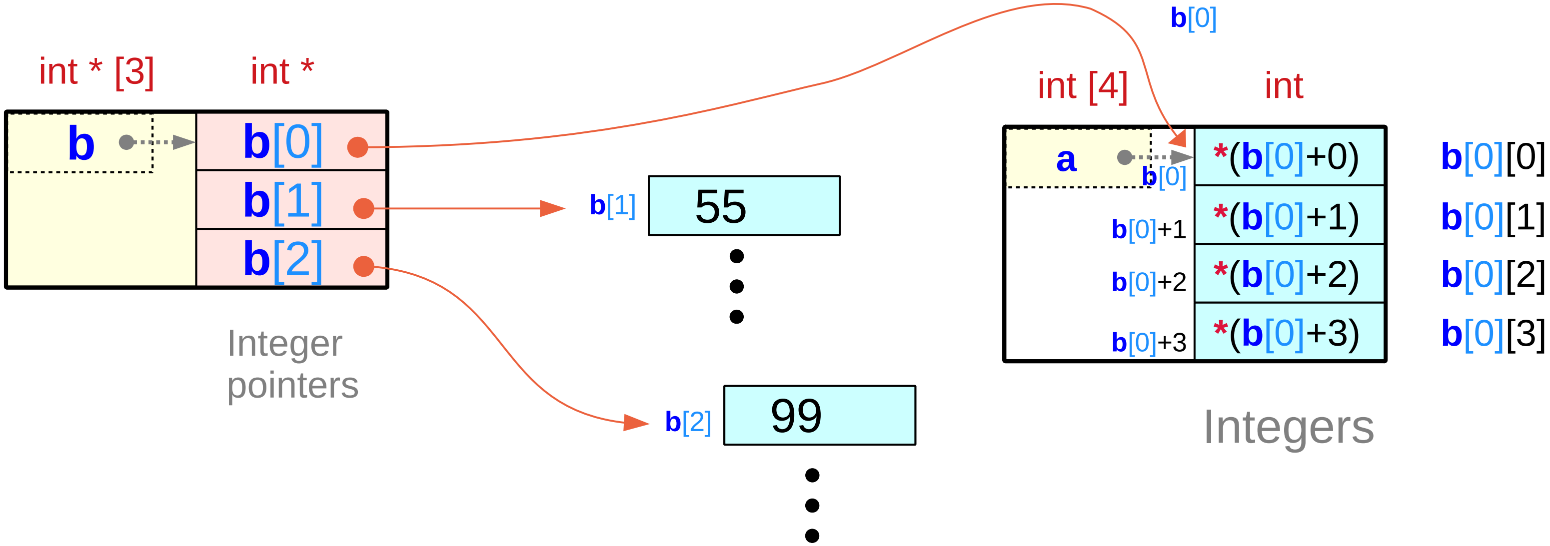
int * **b** [3] ;

Pointer Array

int **a** [4] ;

assignment

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



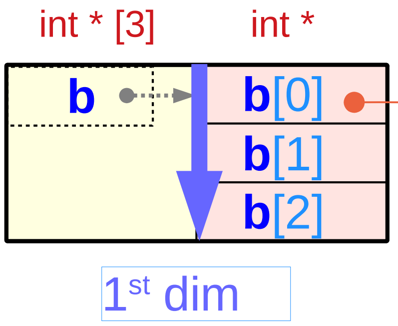
Array of Pointers – extended dimension

int * b [3] ;

Pointer Array

int a [4] ;

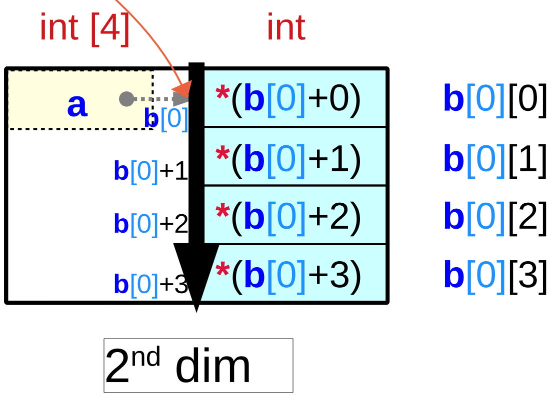
array name b



assignment

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

array name b[0]



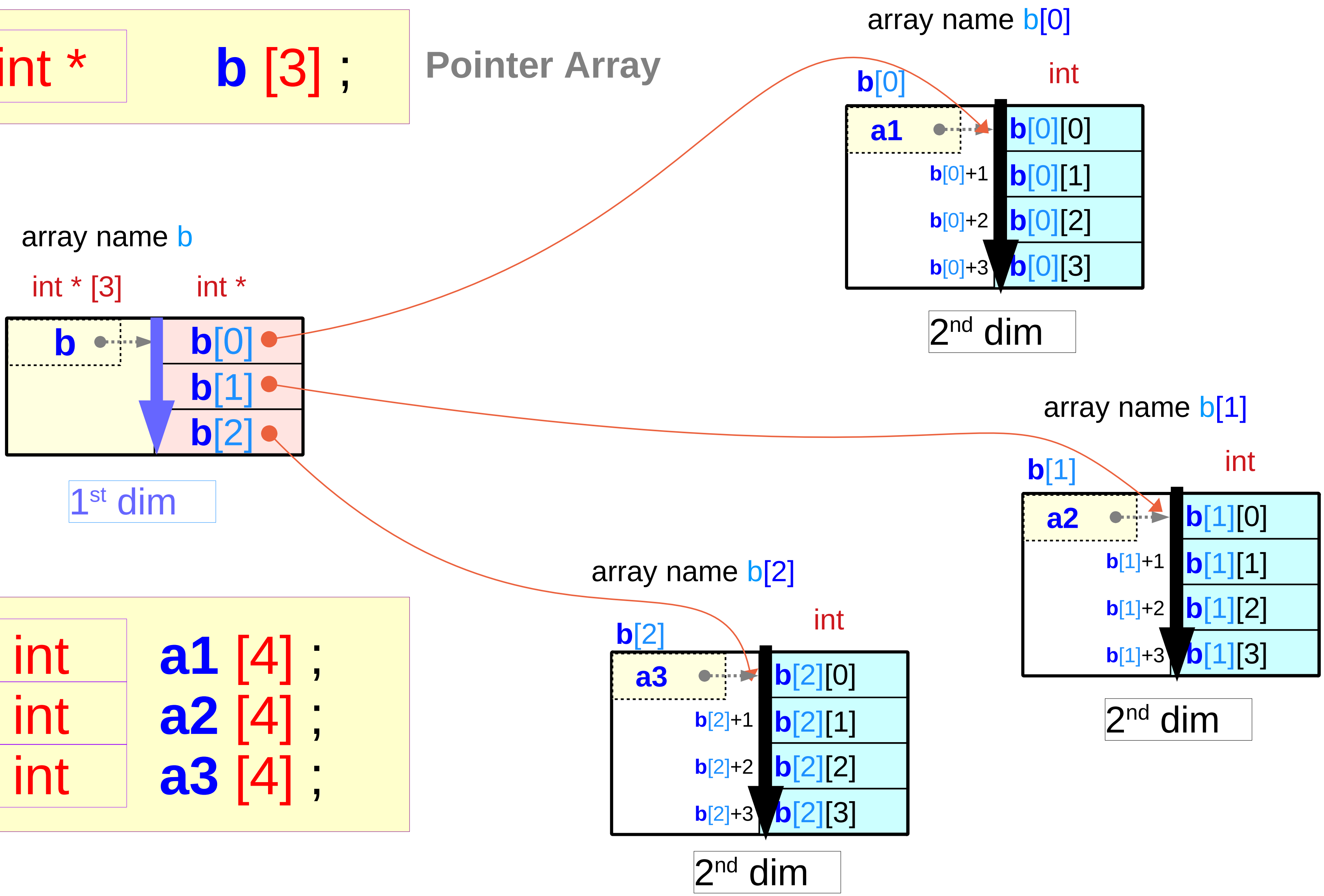
a[0] ≡ b[0][0] ≡ *(*(b+0)+0)
a[1] ≡ b[0][1] ≡ *(*(b+0)+1)
a[2] ≡ b[0][2] ≡ *(*(b+0)+2)
a[3] ≡ b[0][3] ≡ *(*(b+0)+3)

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

int * b [3] ;

Pointer Array

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

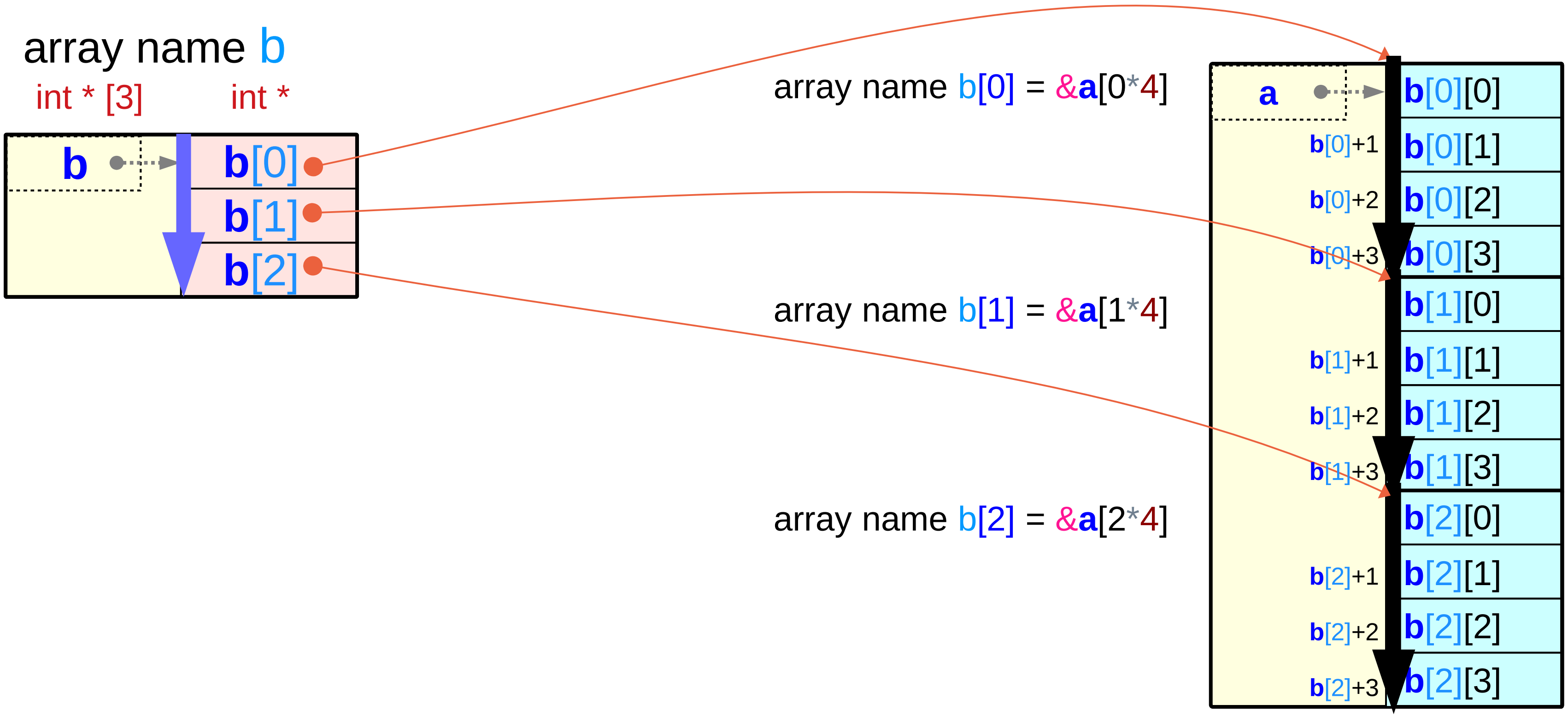


2-d access of a **1-d** array

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)

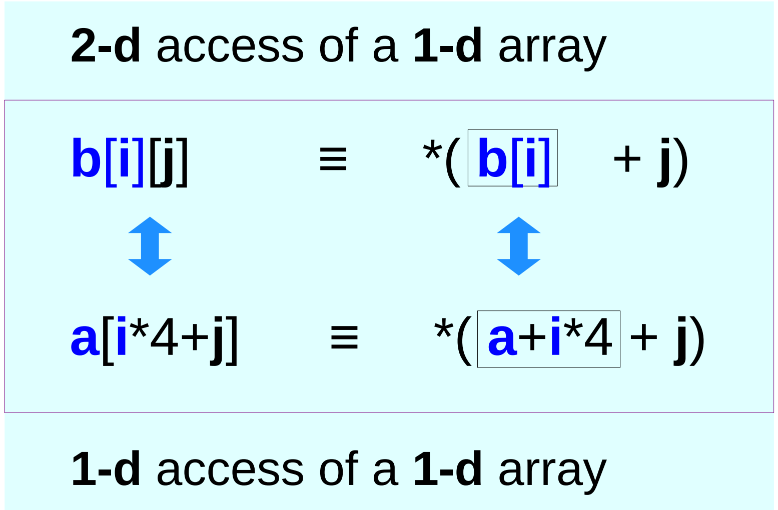
b[j] + k = a+j*4 + k
*(b[j] + k) = *(a+j*4 + k)

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

$$\begin{tikzcd} b[i][j] \arrow[d, leftrightarrow] & \equiv & *(\boxed{b[i]} + j) \arrow[d, leftrightarrow] \\ a[i*4+j] & \equiv & *(\boxed{a+i*4} + j) \end{tikzcd}$$

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 *    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]

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

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

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

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]
```

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

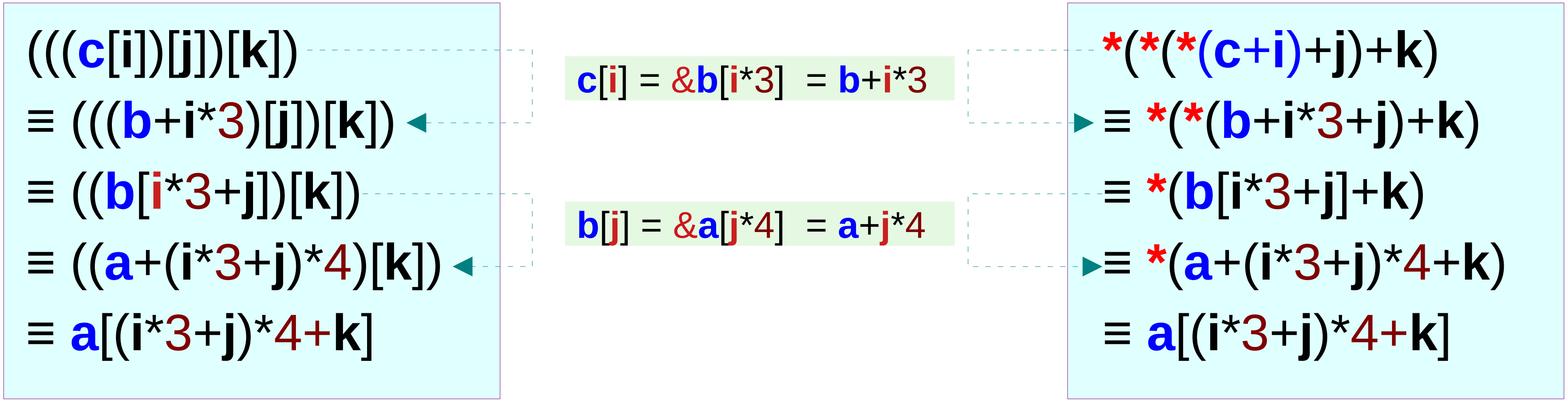
```
c[i][j][k]          ≡  *( c[i][j] +k)
b[i*3+j][k]         ≡  *( b[i*3+j] +k)
a[(i*3+j)*4 +k]     ≡  *(a+(i*3+j)*4 +k)
```

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]

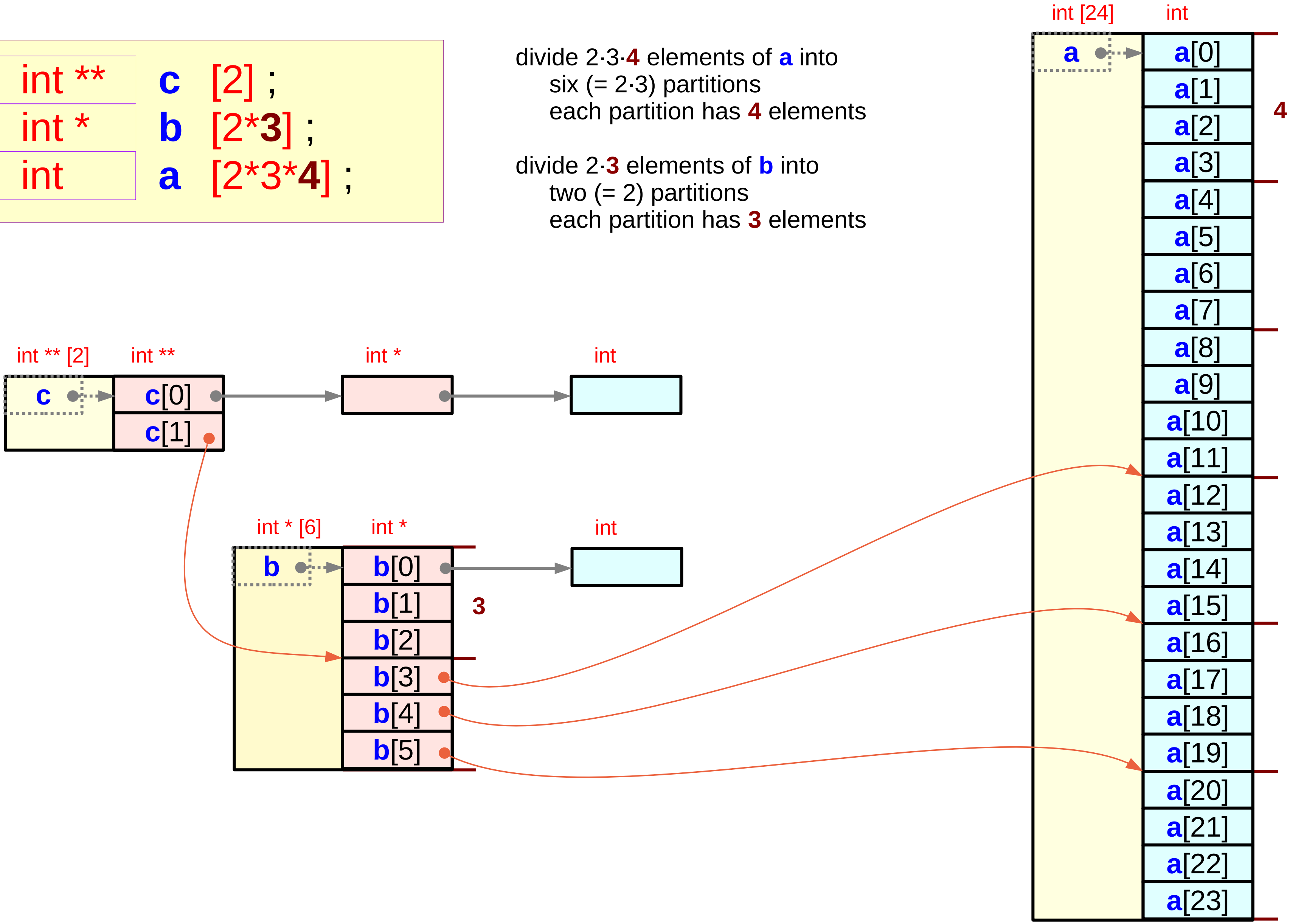
c[i][j][k] = b[i*3+j][k] = a[(i*3+j)*4+k]

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

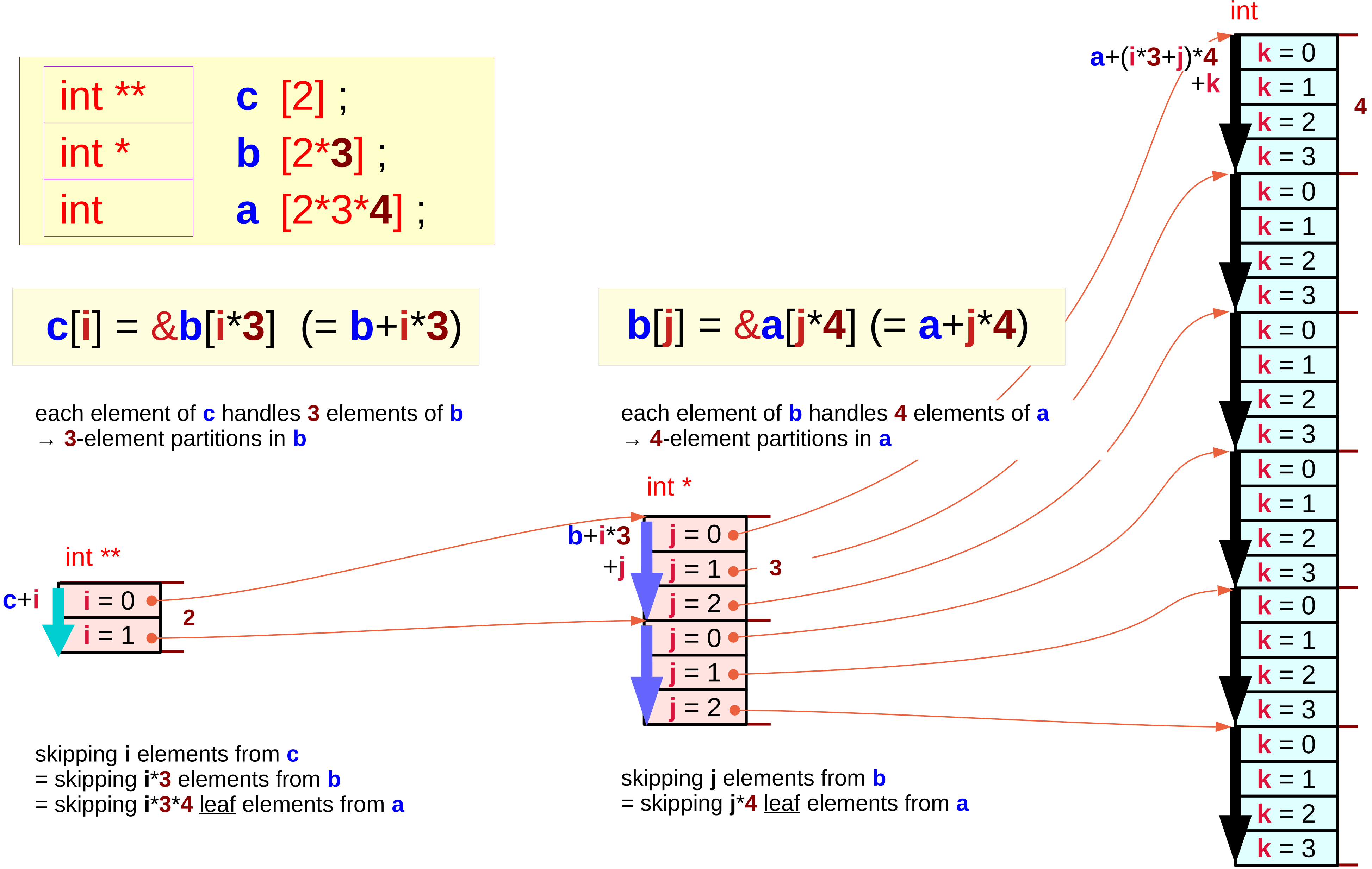
```
int **   c [2] ;
int *    b [2*3] ;
int      a [2*3*4] ;
```

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 leaf elements from **a**

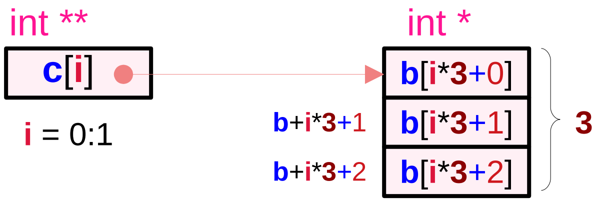
skipping **j** elements from **b**
= skipping **j***4 leaf elements from **a**

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

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

Assigning pointer array

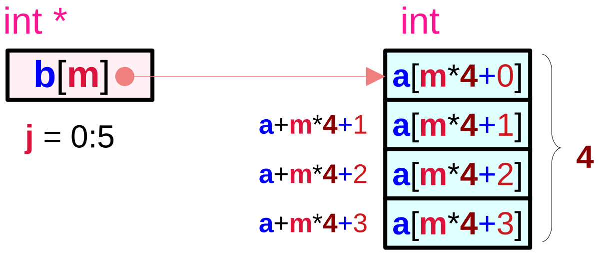
b[j] = &a[j*4] (= a+j*4)
c[i] = &b[i*3] (= b+i*3)



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

c[i]   contiguous c over i = 0:1
b[i*3+j]   contiguous b over j = 0:2

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



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

b[m]   contiguous b over m = 0:5
a[m*4+k]   contiguous a over k = 0:3

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)

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** <u>leaf</u> elements in **a**

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

skipping 1 element in **c**
= skipping **3** elements in **b**
= skipping **3*****4** <u>leaf</u> 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** <u>leaf</u> elements from **a**

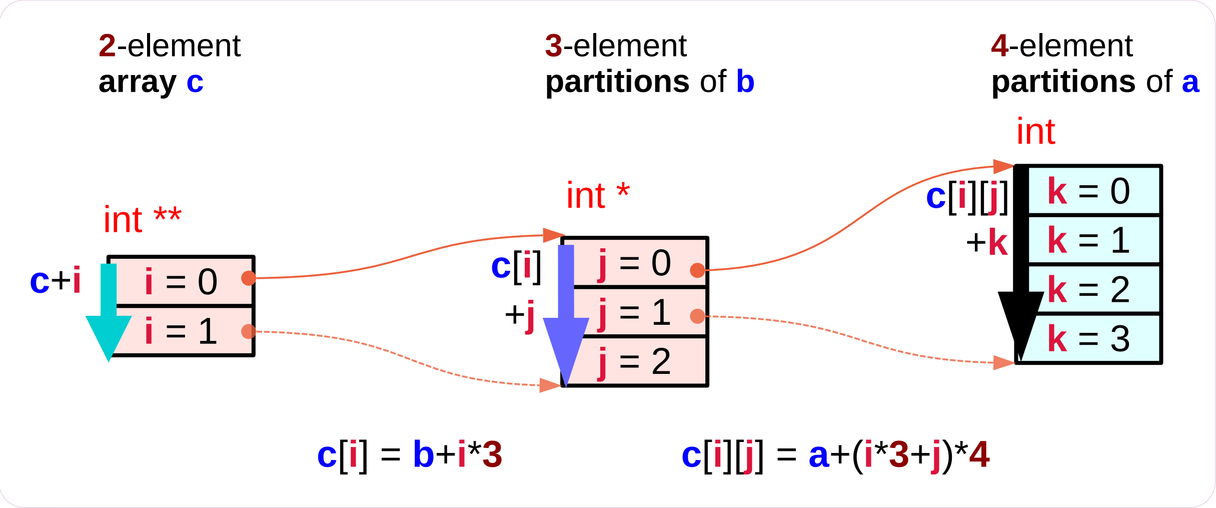
= skipping (**i*****3**+**j**)***4**+**k** <u>leaf</u> elements from **a**

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]



c[i] = b+i*3

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

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)

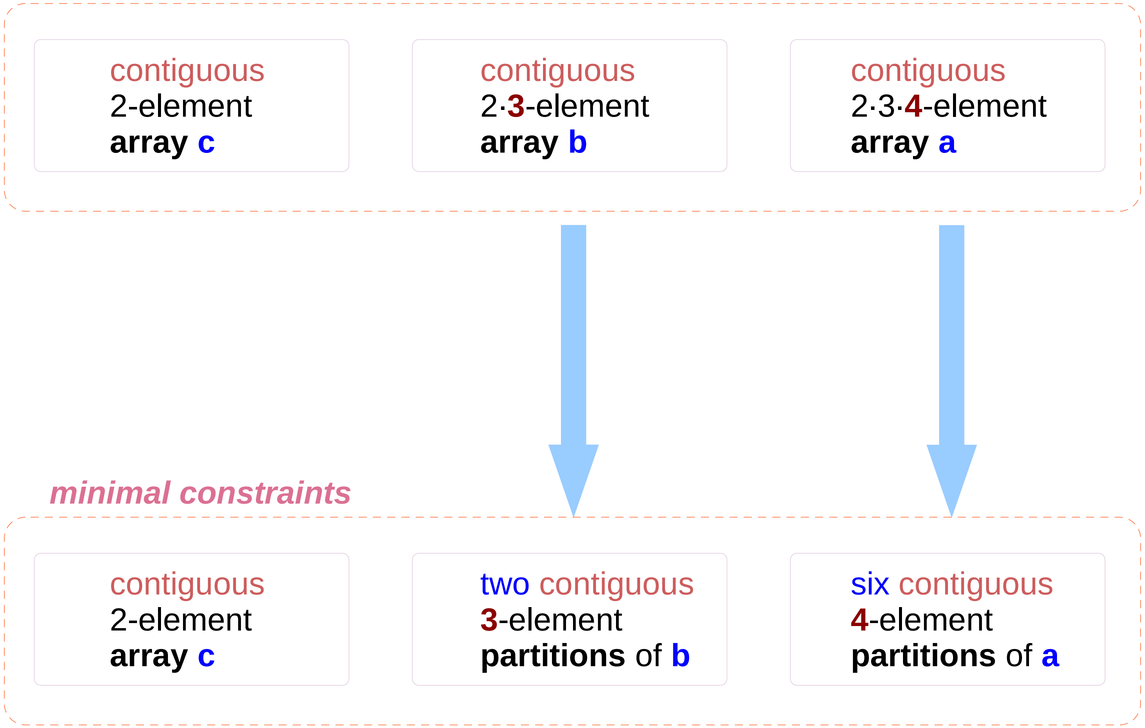
Minimal constraints and implementations

```
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];
```

```
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];
```



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**

two contiguous
3-element
arrays bi

six contiguous
4-element
arrays ai

Accessing a **contiguous 1-d** array

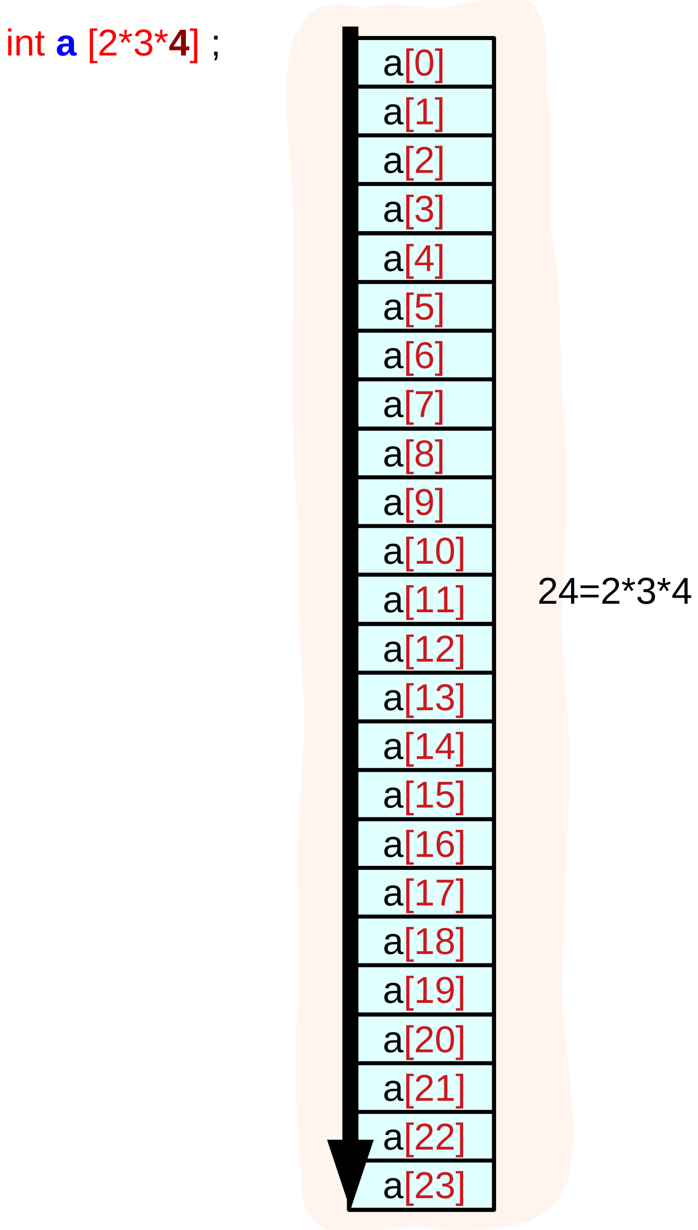
- **1-d** array access
- **2-d** array access
- **3-d** array access

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] ;
```

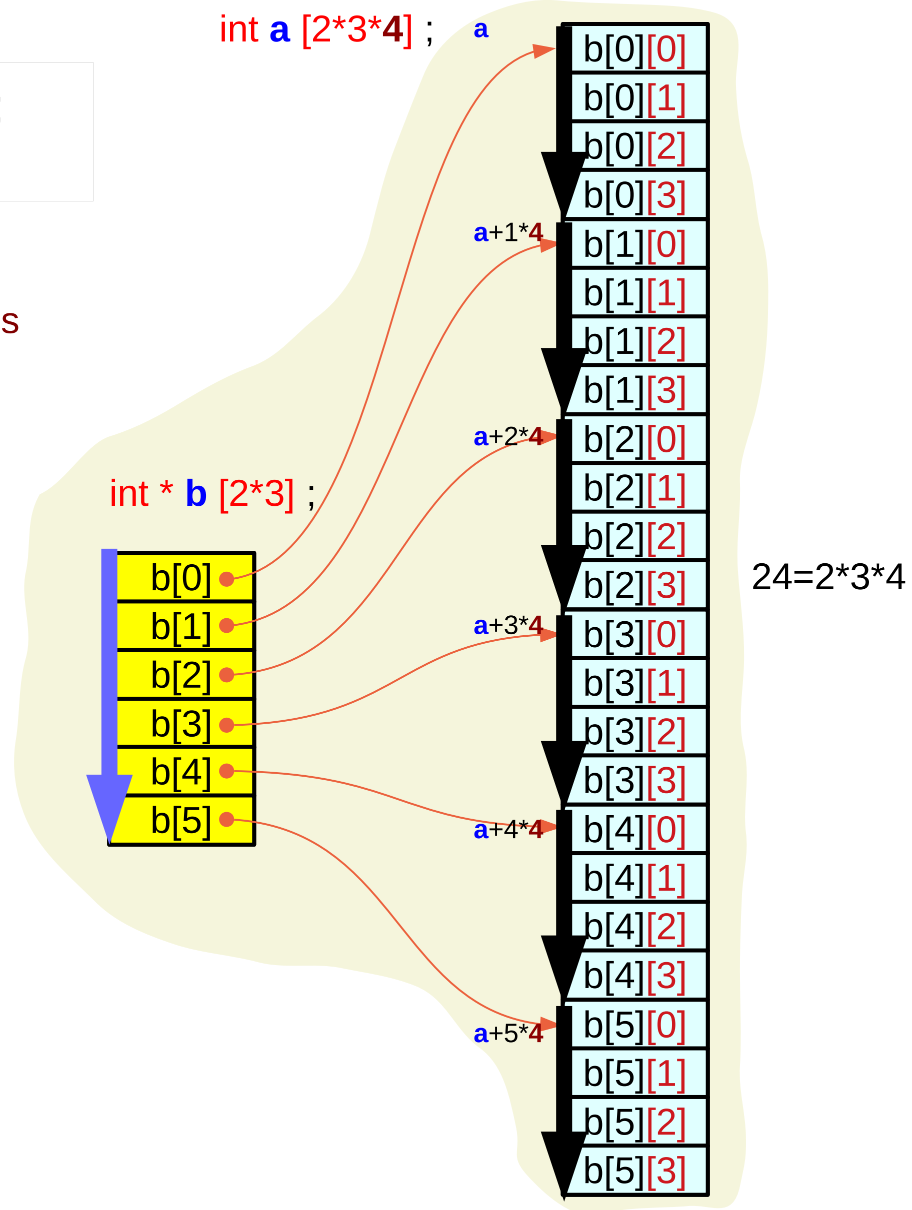
b [j][k] ≡ a [j*4 +k]

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

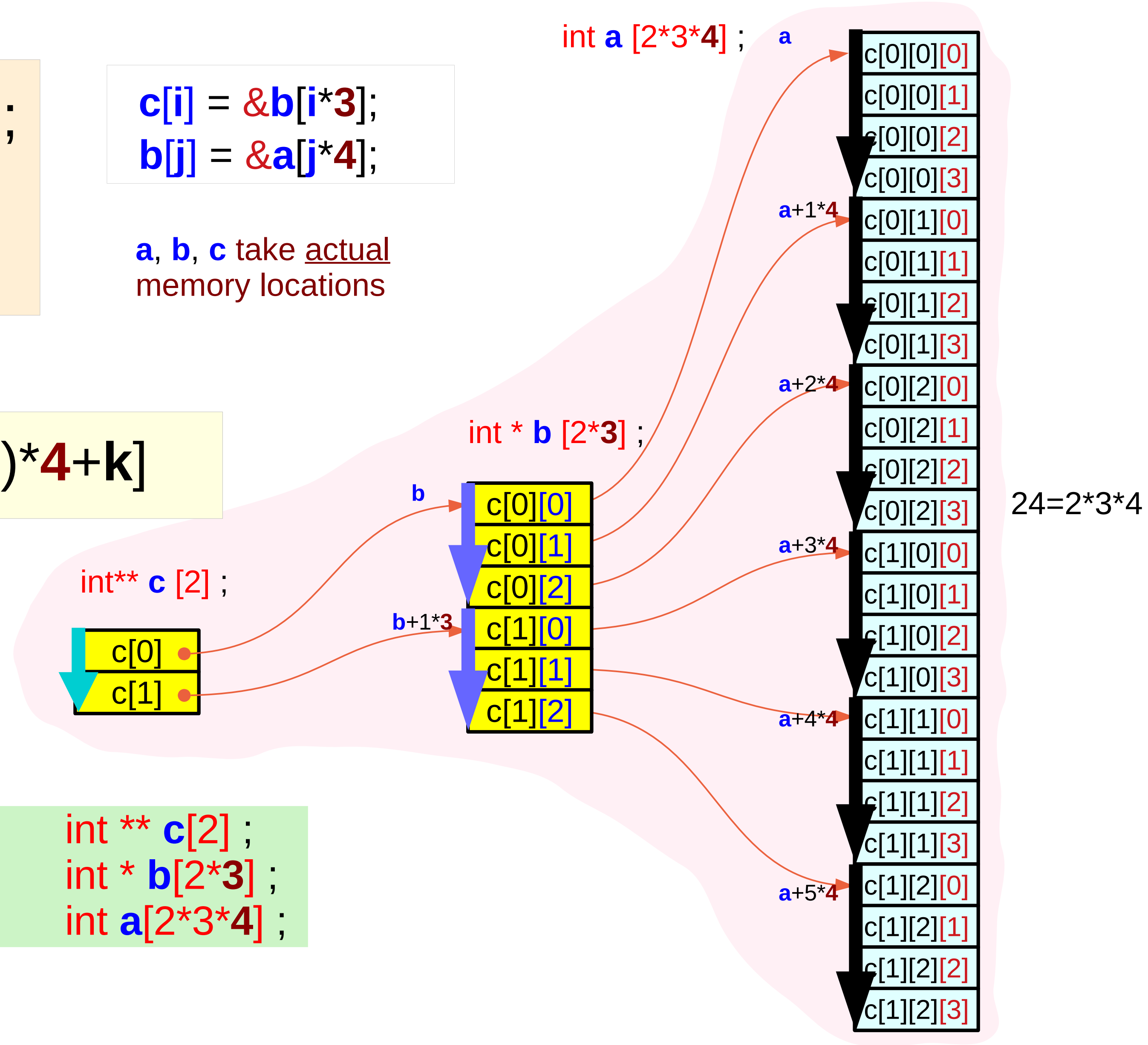
```
int      a [2*3*4] ;
int *    b [2*3] ;
int **   c [2] ;
```

```
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]
```

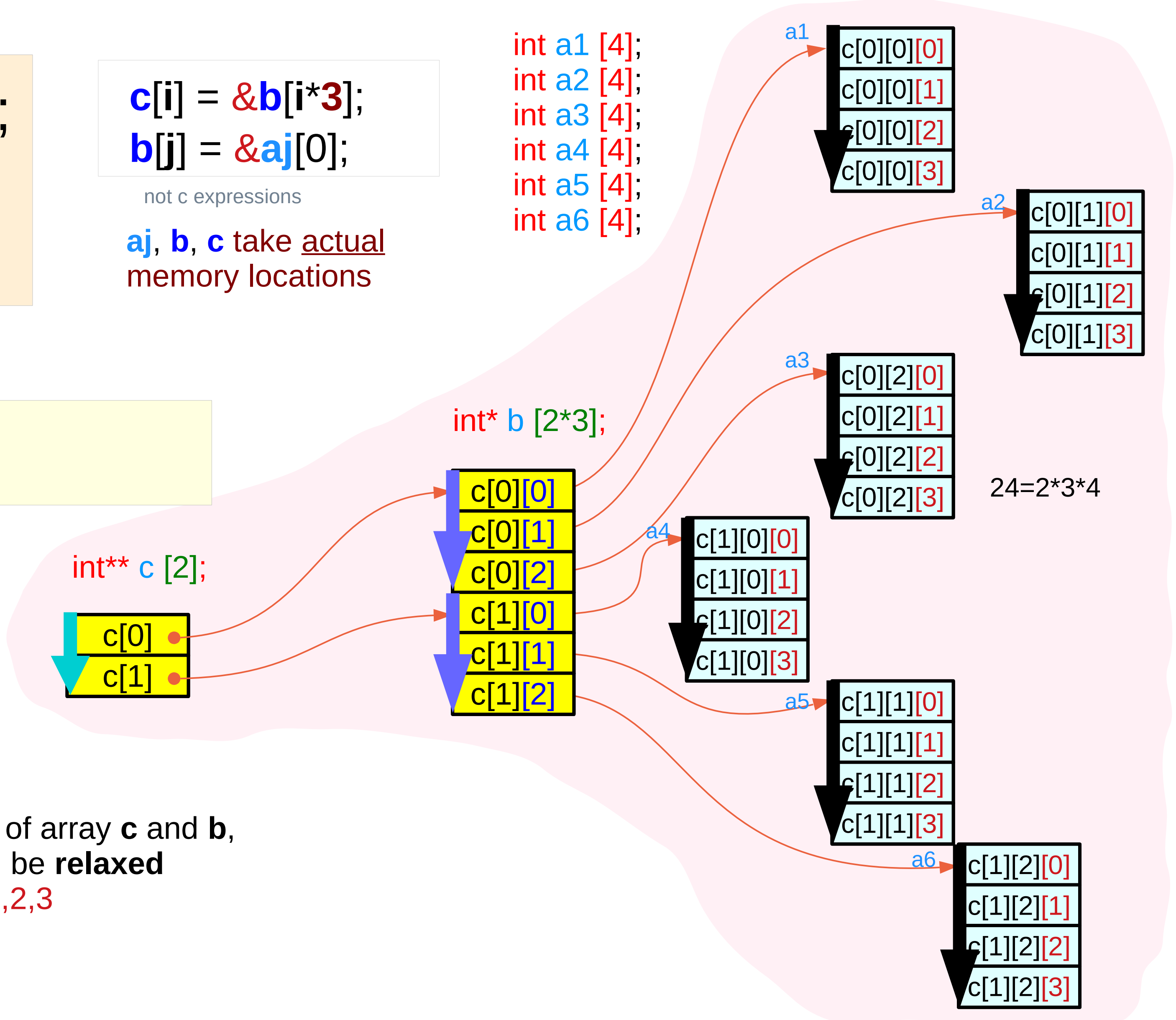
i = 0, 1
j = 0, 1, 2
k = 0, 1, 2, 3

```
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

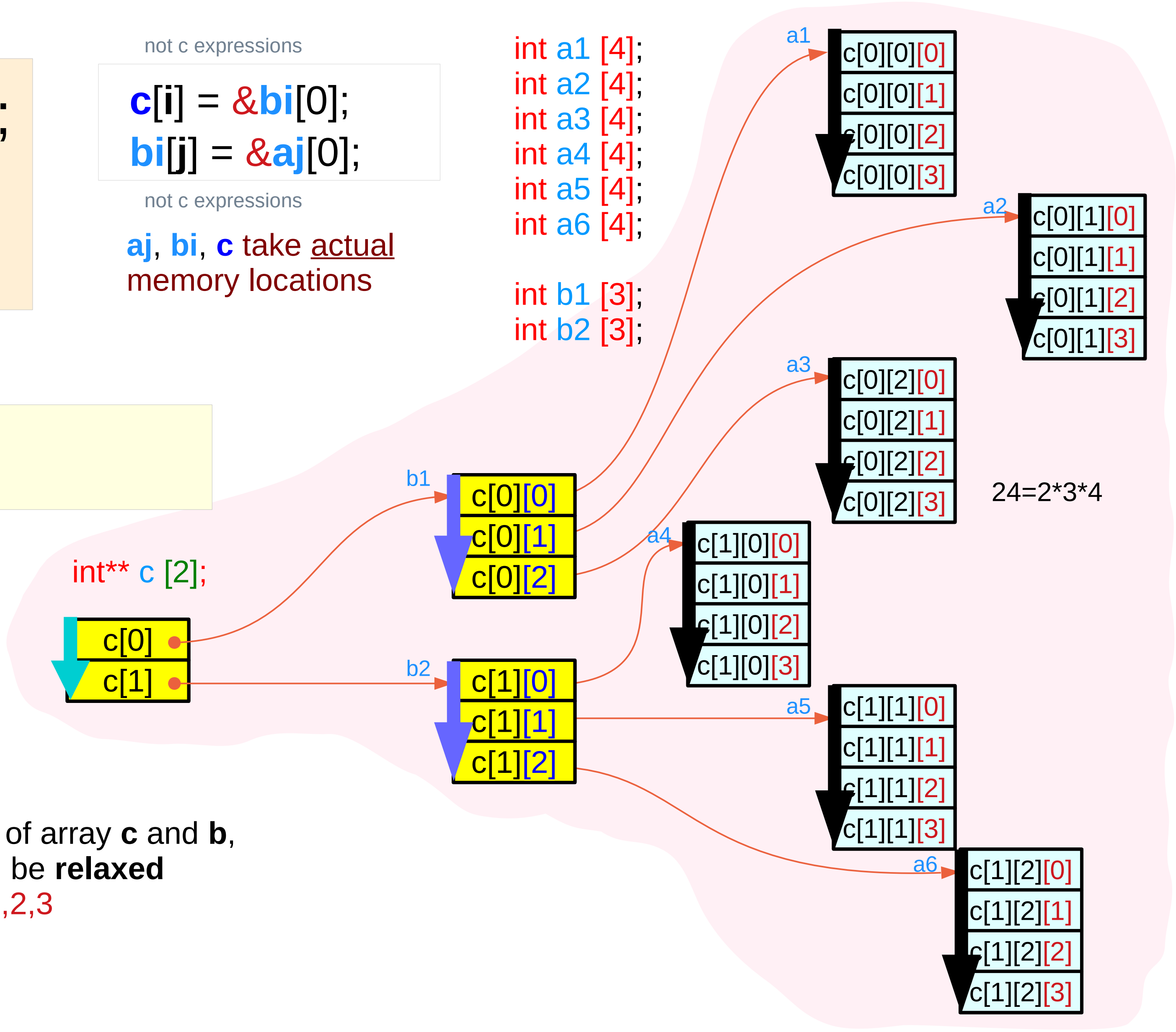
c[**i**] = &**bi**[0];
bi[**j**] = &**aj**[0];

not c expressions

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

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

int b1 [3];
int b2 [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

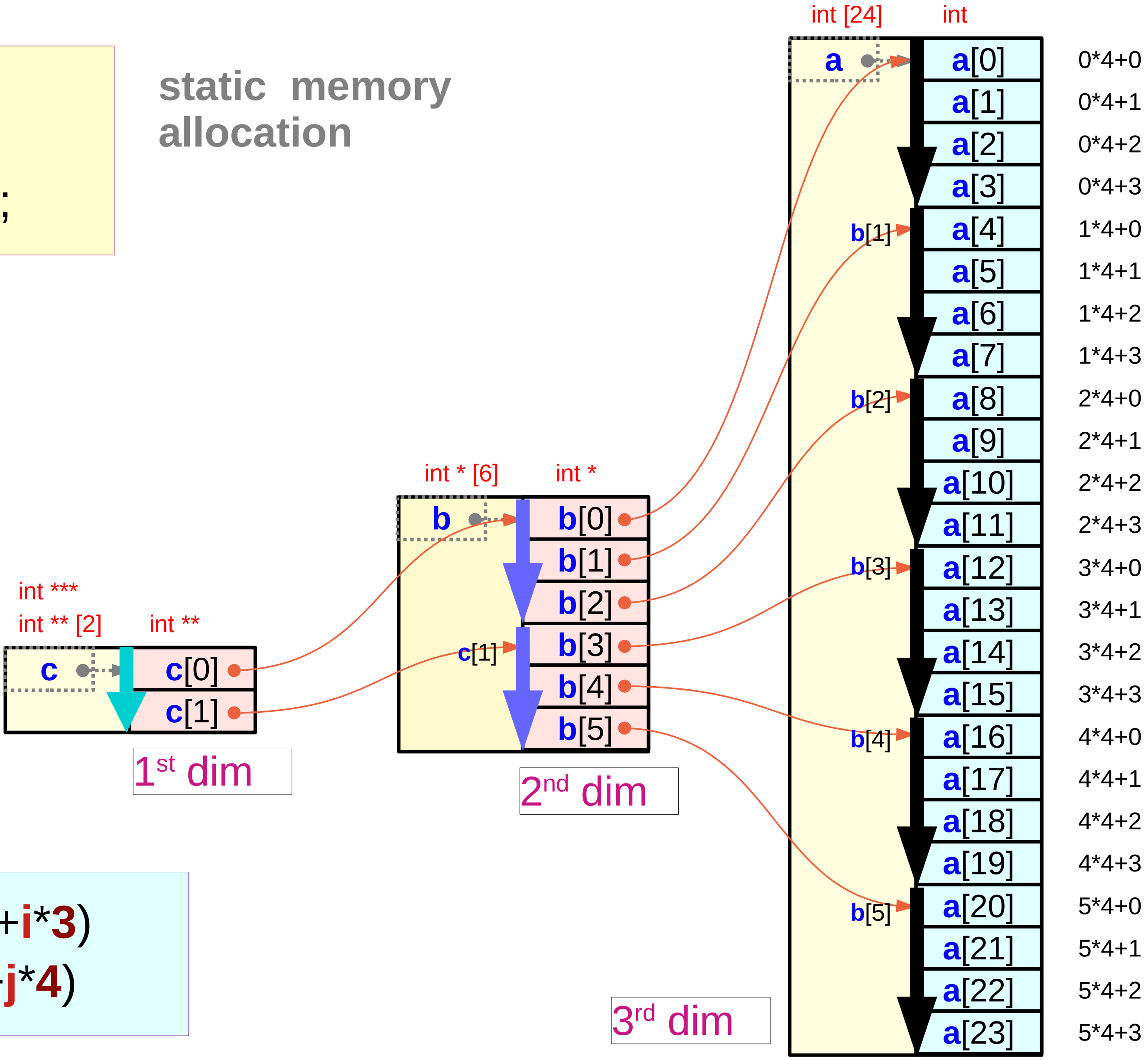
Accessing **statically** allocated arrays

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

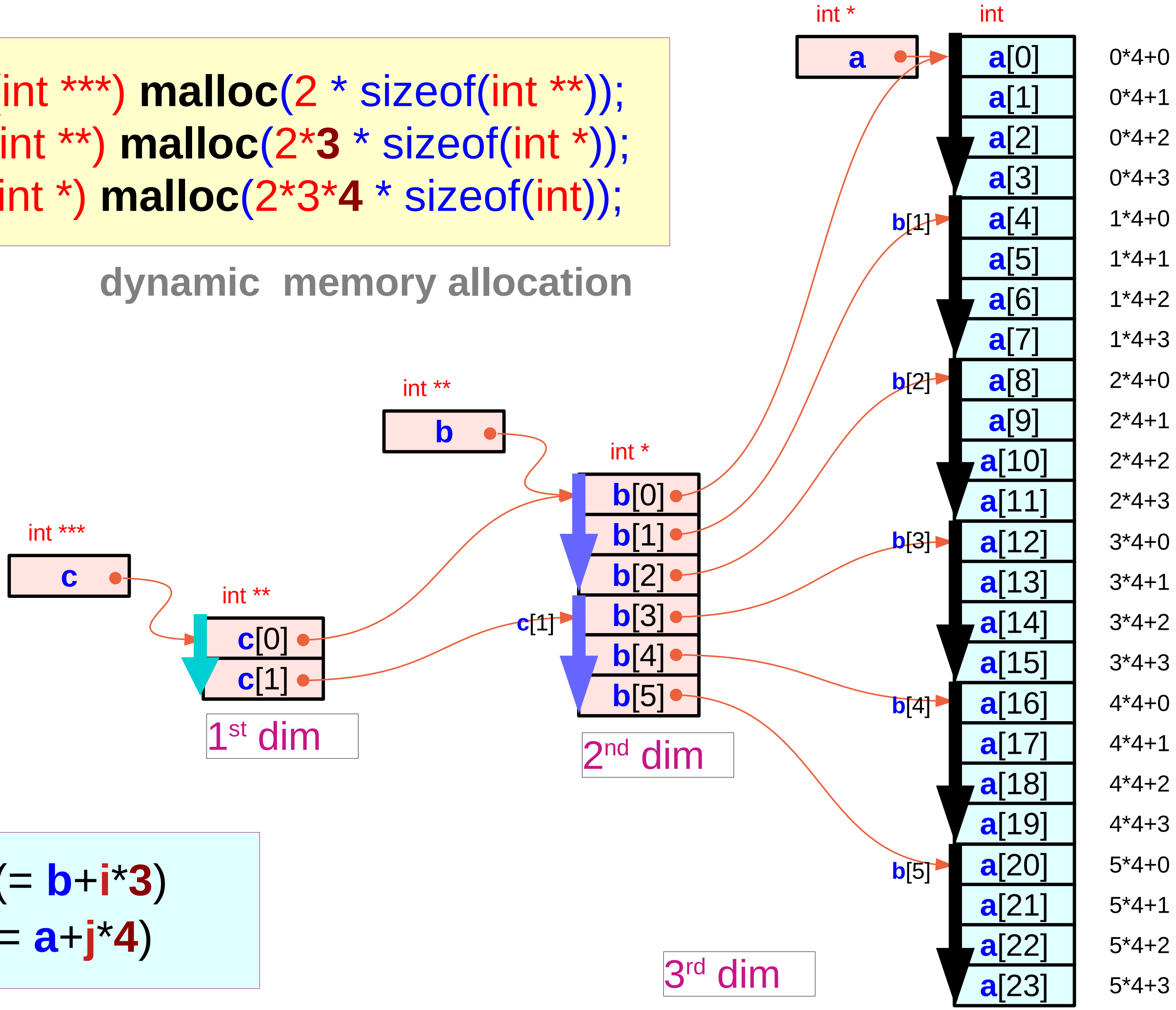


c[i] = &**b**[i***3**] (= **b**+i***3**)
b[j] = &**a**[j***4**] (= **a**+j***4**)

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

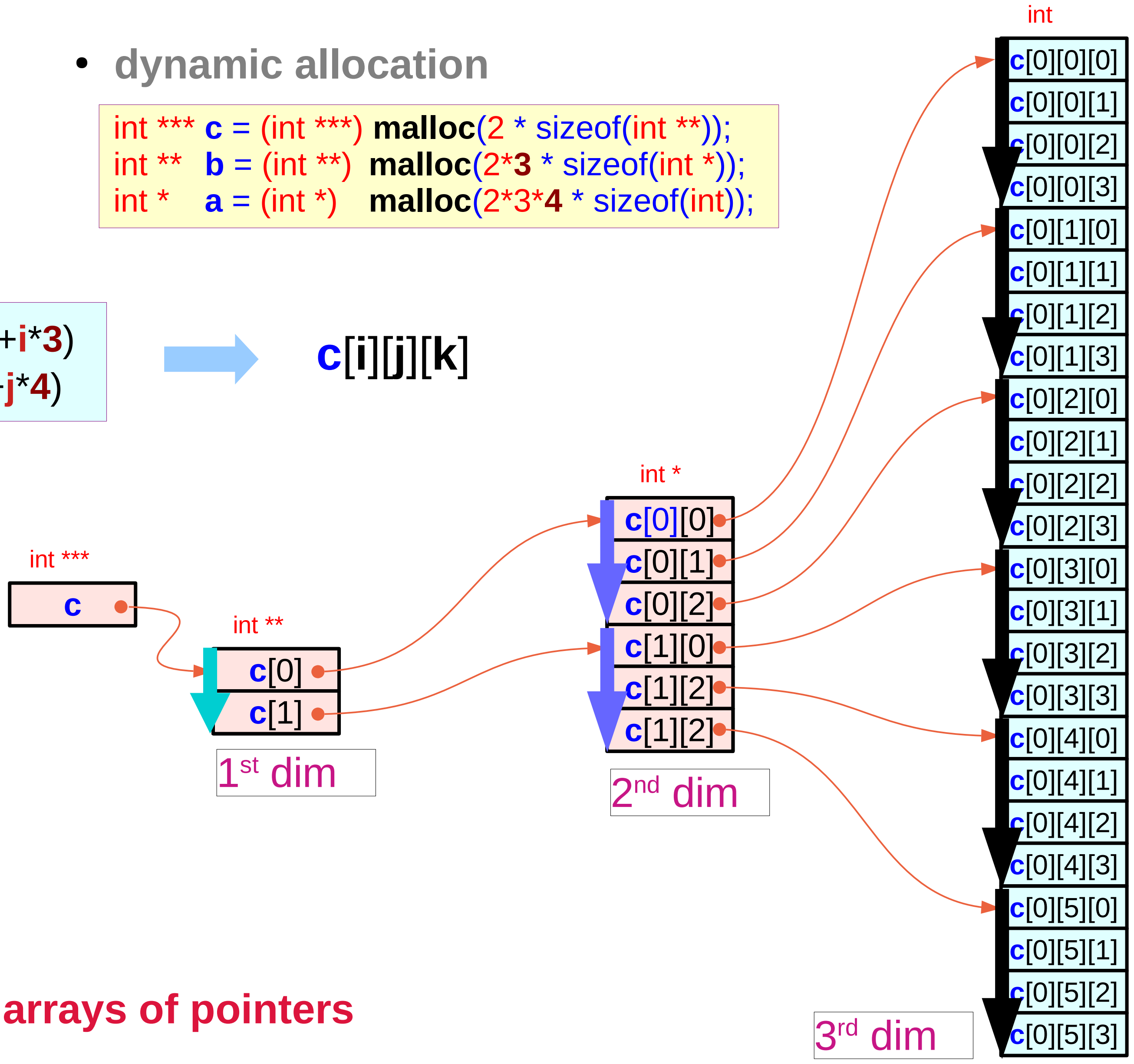
```
int **    c  [2] ;
int *     b  [2*3] ;
int       a  [2*3*4] ;
```

- 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)

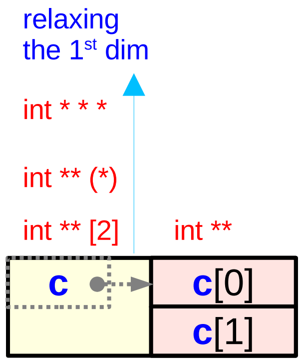
→ c[i][j][k]

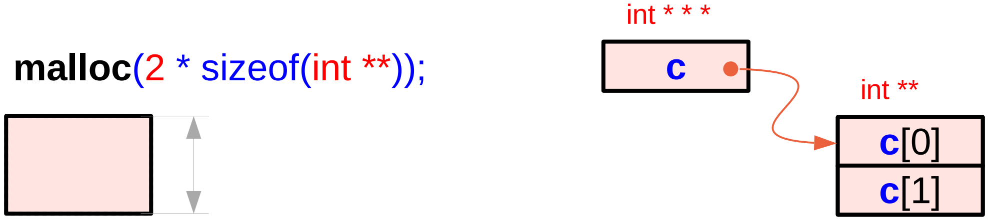


arrays of pointers

Static v.s. dynamic allocation (3)

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





int ** c [2] ;

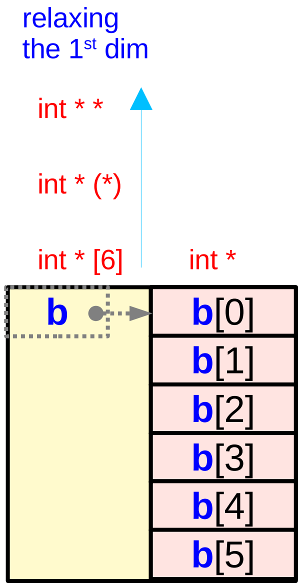
int *** c = (int ***) malloc(2 * sizeof(int **));

static memory allocation

dynamic memory allocation

Static v.s. dynamic allocation (4)

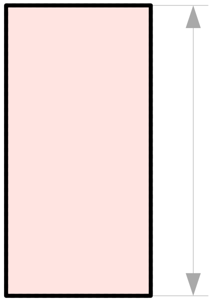
`int ** b = (int ***) malloc(6 * sizeof(int *));`

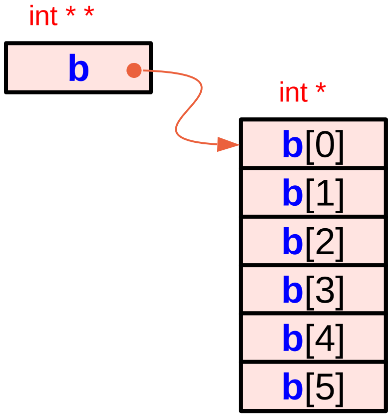


`int * b [6] ;`

static memory allocation

malloc(6 * sizeof(int *));



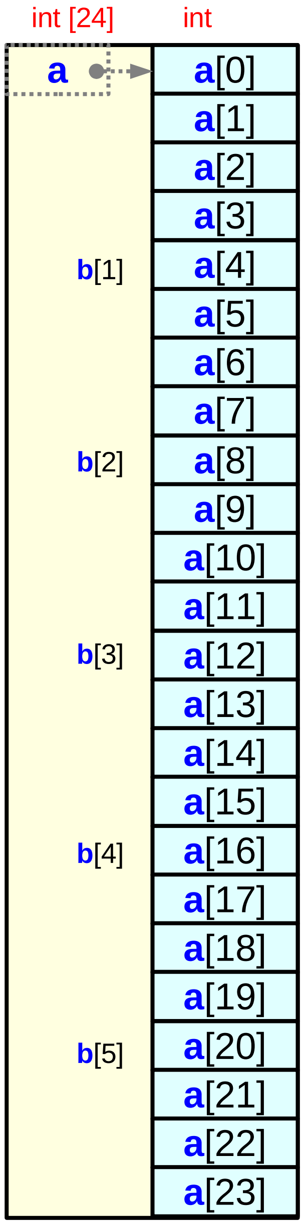


`int ** b = (int **) malloc(6 * sizeof(int *));`

dynamic memory allocation

Static v.s. dynamic allocation (5)

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



relaxing
the 1st dim

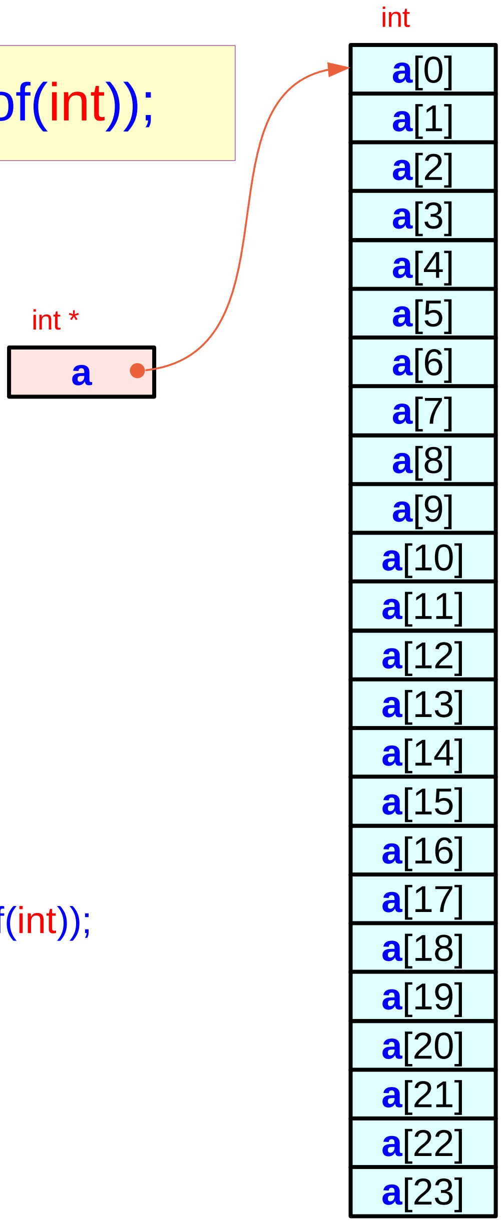
int *

int (*)

int [24]

`int * a   [24] ;`

static memory allocation



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

dynamic memory allocation

& address-of operator

***** dereference operator

Address-of operator and dereferencing operator

the address of a variable :
address-of operator ***&***

the content at an address :
dereferencing operator *******

& variable *:*
returns the address of a variable

variable *has memory locations*
whose value can be changed
by an assignment

(***variable*** *must be an lvalue*)

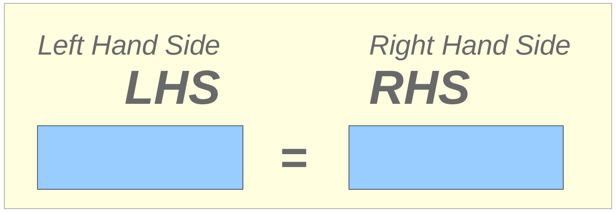
*** address** *:*
returns the value at the address

*** address** *has memory locations*
whose value can be changed
by an assignment

(*** address** *is an lvalue*)

lvalue and **r**value in assignments

an assignment statement



```
int   a,  b = 10 ;
int * p, q = &a ;
```

***l**value* = ***l**value*

***l**value* = ***r**value*

~~***r**value*~~ = ***l**value*

~~***r**value*~~ = ***r**value*

p = **q** ;

p = **&a** ;

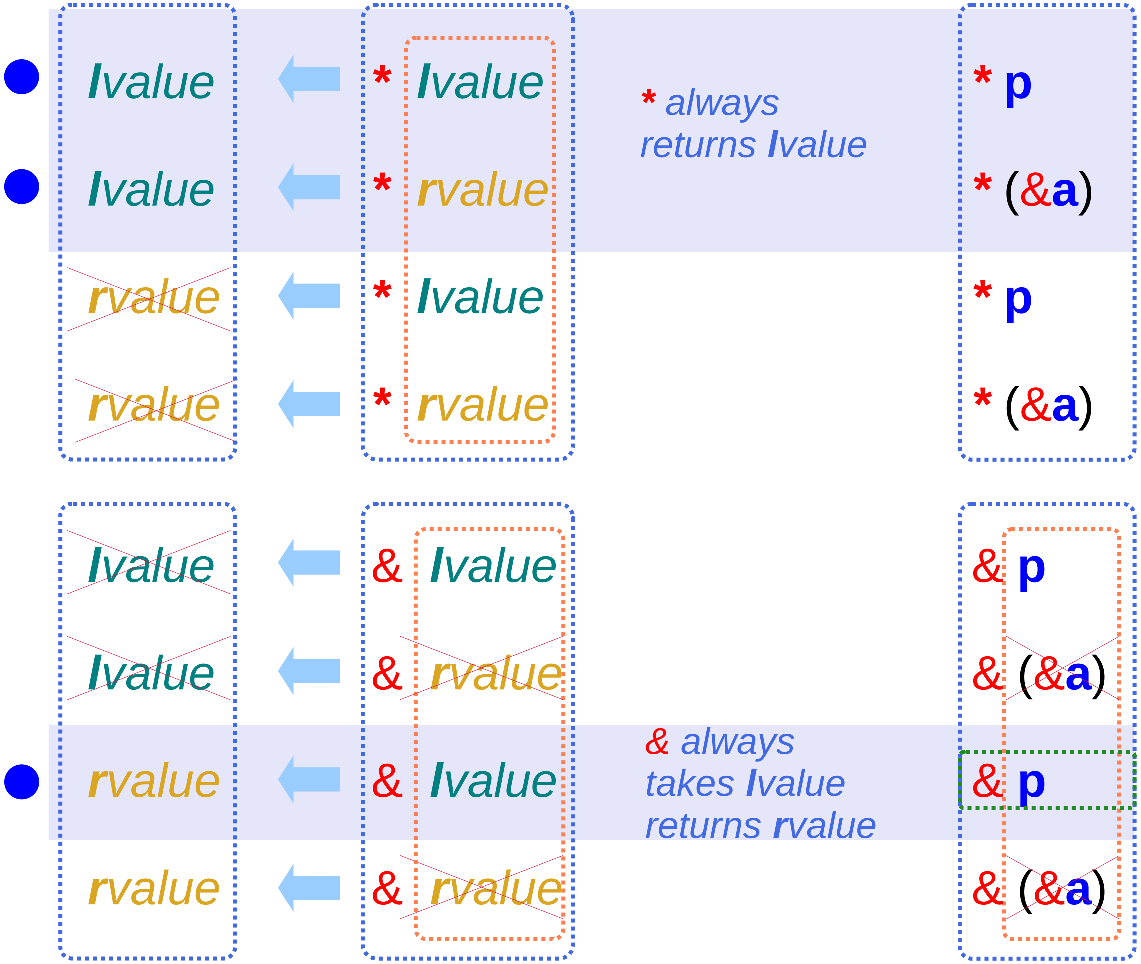
~~**&a**~~ = **p** ;

~~**&a**~~ = **&b** ;

*in the **LHS,** only **l**value can exist*
r**value can exist only in the **RHS

a, **b**, **p**, **q** : **l**values … variables … RW
***p**, ***q** : **l**values … variables … RW
&a, **&b** : **r**values … constants … RO

lvalue and rvalue with * and & operators



● lvalue ← * lvalue — * p

● lvalue ← * rvalue — * (&a)

~~rvalue~~ ← * lvalue — * p

~~rvalue~~ ← * rvalue — * (&a)

* always returns lvalue

~~lvalue~~ ← & lvalue — & p

~~lvalue~~ ← & ~~rvalue~~ — & ~~(&a)~~

● rvalue ← & lvalue — & p

rvalue ← & ~~rvalue~~ — & ~~(&a)~~

& always takes lvalue returns rvalue

```
int     a = 10 ;
int  *  p = &a ;
```

* can be applied
to either an **lvalue** variable
or a **rvalue** address

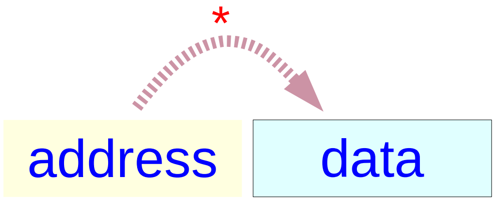
* **operand** becomes
an **lvalue** variable
thus can be applied
successively.

& can be applied
to only an **lvalue** variable and
returns only an **rvalue** address

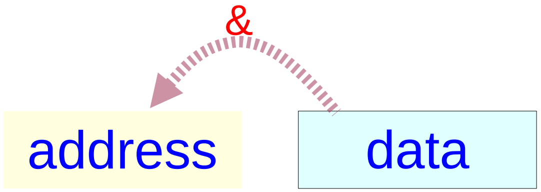
a, **p** : lvalues … variables … RW
*__p__ : lvalues … variables … RW
&__a__ : rvalues … constants … RO

Address-of **&** and dereference ***** C operators (1)

Primitive Data Type

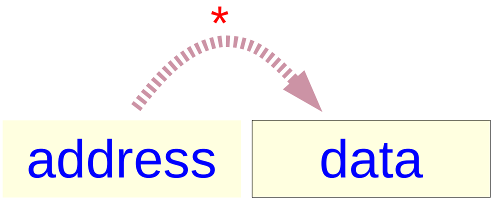


rvalue **&a**	$\xrightarrow{*}$	*lvalue* **a**
constant		*variable*
lvalue **p**	$\xrightarrow{*}$	*lvalue* ***p**
pointer		*variable*

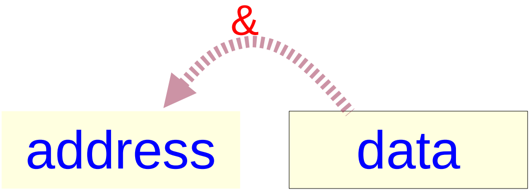


rvalue **&a**	$\xleftarrow{\&}$	*lvalue* **a**
constant		*variable*
~~*lvalue*~~ **p**	$\xleftarrow{\&}$	*lvalue* ***p**
pointer		*variable*

Pointer Data Type

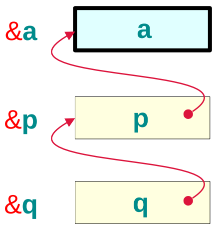


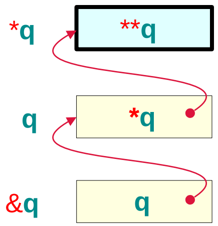
rvalue **&q**	$\xrightarrow{*}$	*lvalue* **q**
constant		*double pointer*
lvalue **q**	$\xrightarrow{*}$	*lvalue* ***q**
double pointer		*pointer*



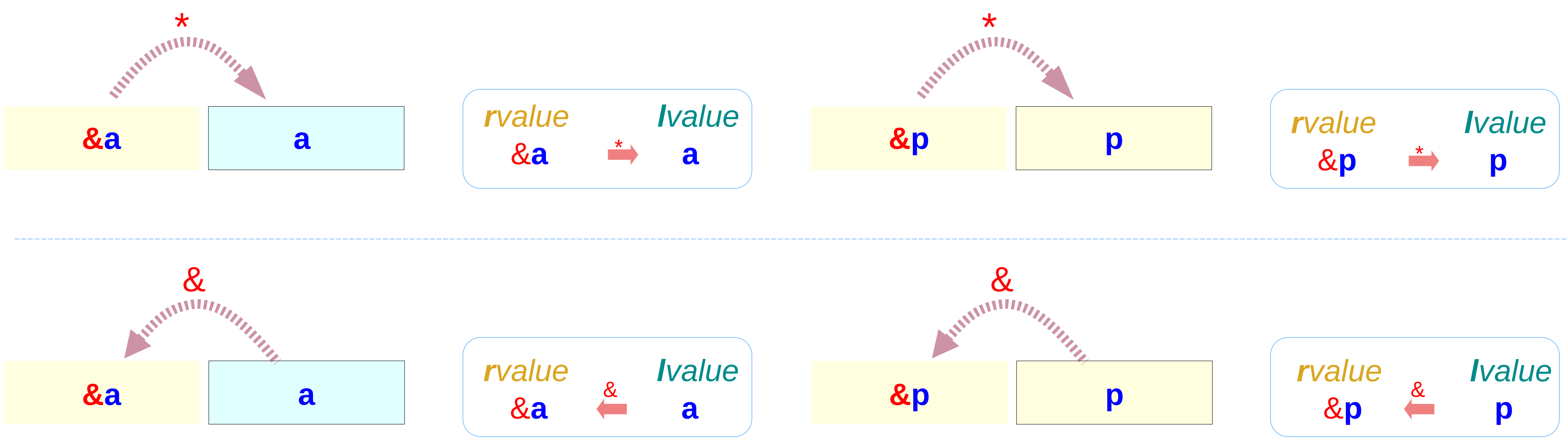
rvalue **&q**	$\xleftarrow{\&}$	*lvalue* **q**
constant		*double pointer*
~~*lvalue*~~ **q**	$\xleftarrow{\&}$	*lvalue* ***q**
double pointer		*pointer*

```
int      a ;
int *    p ;
int **   q ;
```





Address-of **&** and dereference ***** C operators (2)



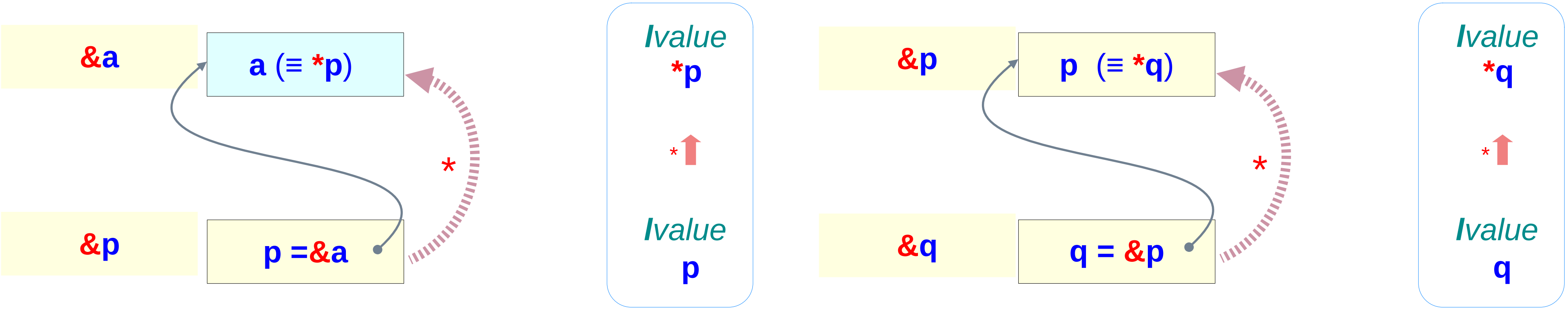
*&a = a

&*a = a

*&p = p

&*p = p

Address-of **&** and dereference ***** C operators (2)



&a

a (≡ *p)

&p

p =&a

*

lvalue
*p

*

lvalue
p

&p

p (≡ *q)

&q

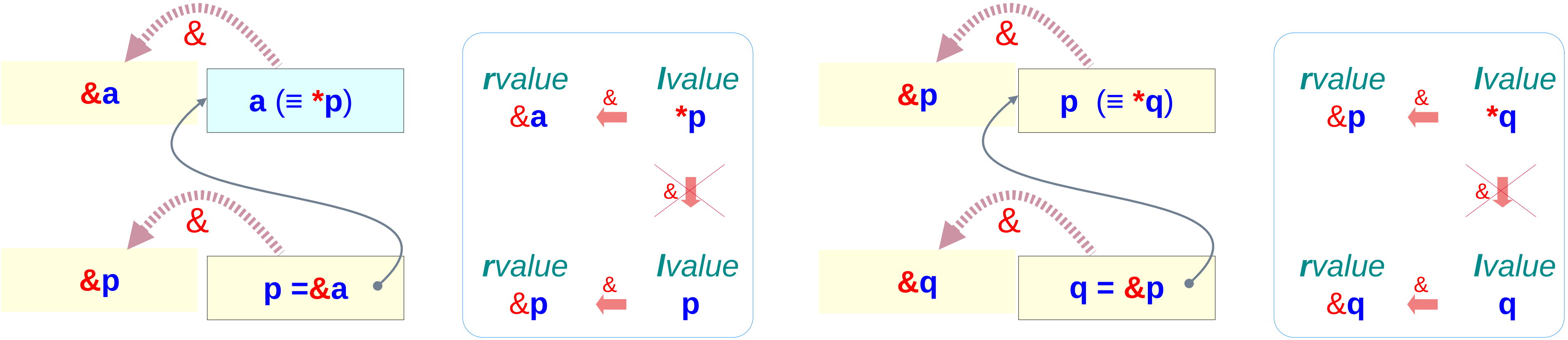
q = &p

*

lvalue
*q

*

lvalue
q



&

&a

a (≡ *p)

&

&p

p =&a

rvalue
&a

&

lvalue
*p

&

rvalue
&p

&

lvalue
p

&

&p

p (≡ *q)

&

&q

q = &p

rvalue
&p

&

lvalue
*q

&

rvalue
&q

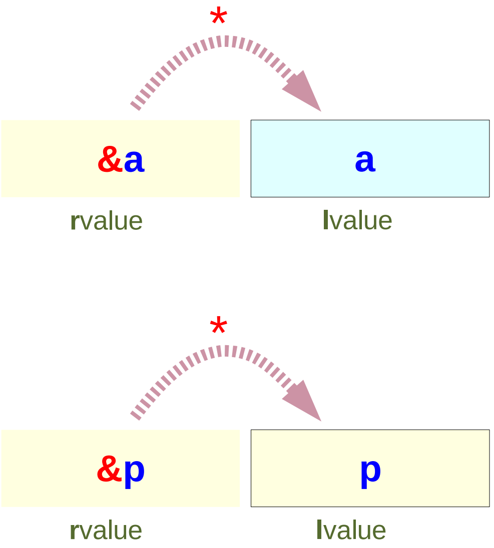
&

lvalue
q

*&p = p
~~&*p = p~~ value(p)

*&q = q
~~&*q = q~~ value(q)

Type, Size, and Value attributes of an **lvalue**



lvalue ← **rvalue*

a **&a*

lvalue ← **rvalue*

p **&p*

lvalue is associated with a memory location

lvalue has the following attributes

- Type
- Size
- Value

rvalue has the only attribute

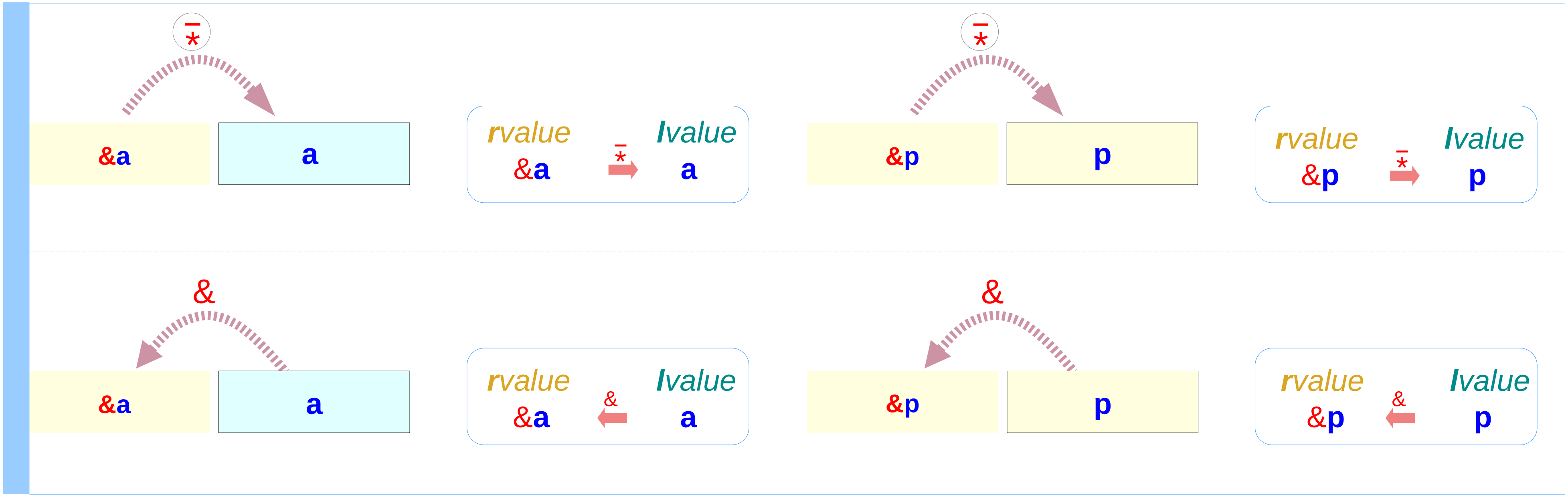
- Value

assume the function value()

value(**lvalue**) returns
the Value attribute of **lvalue**

value(**rvalue**) returns
the Value attribute of **rvalue**

Inverse operators $\bar{*}$ and &



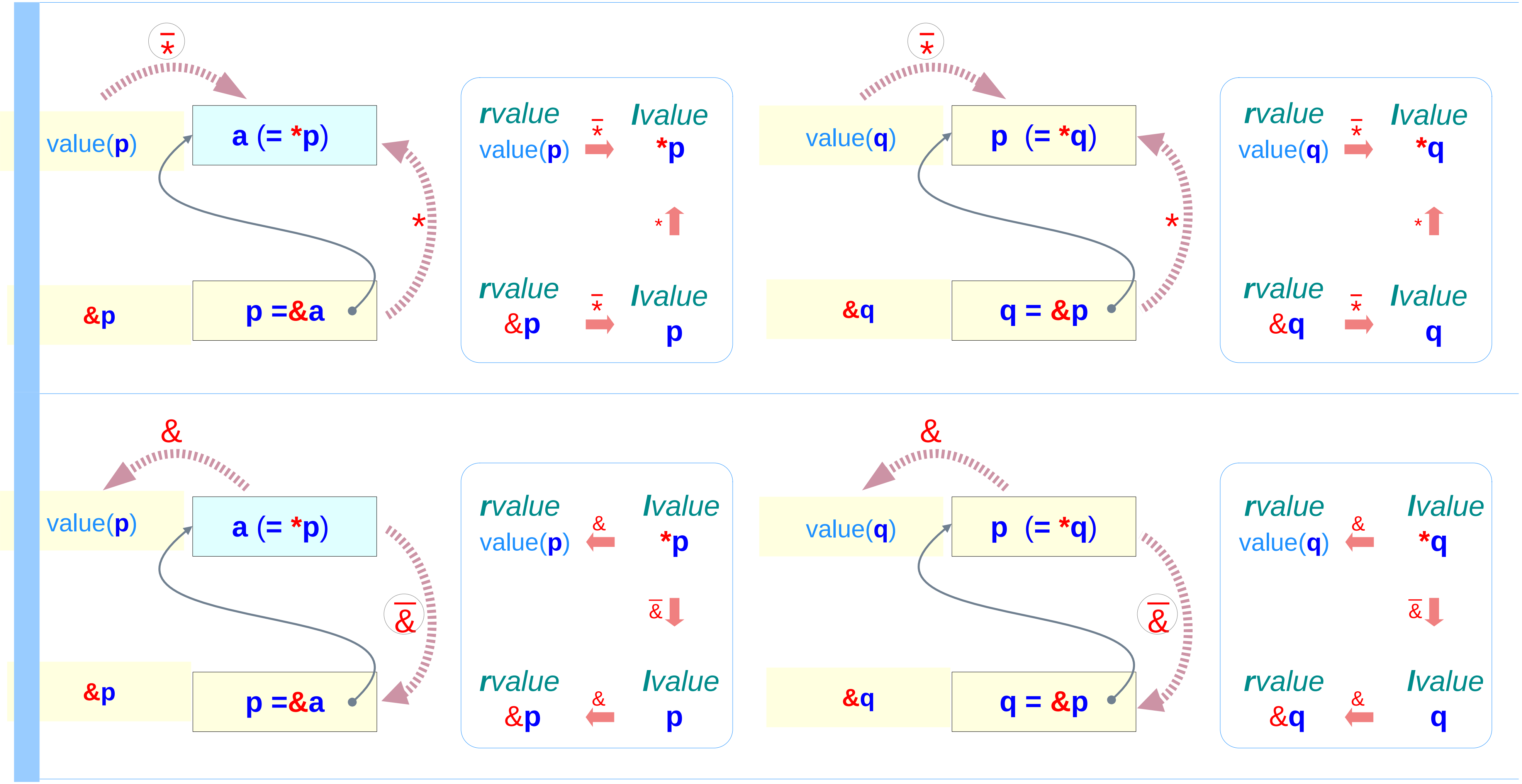
$\bar{*}\&a = a$

$\&\bar{*}a = a$

$\bar{*}\&p = p$

$\&\bar{*}p = p$

Inverse operators $\overline{\&}$ and *



$\overline{*}$&p = p
$\overline{\&}$*p = p

$\overline{*}$&q = q
$\overline{\&}$*q = q

Inverse operators $\overline{*}$ and **&**

```
int    a ;
int *  p ;
int ** q ;
```

$\overline{*}\&a = a$

$\&\overline{*}a = a$

$\overline{*}\&p = p$

$\&\overline{*}p = p$

$\overline{*}\&p = p$

$\overline{\&}*p = p$

$*\&p = p$

$\&*p = \text{value}(p)$

$\overline{*}\&q = q$

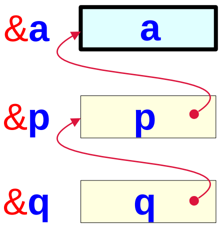
$\&\overline{*}q = q$

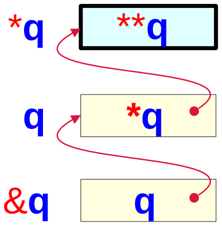
$\overline{*}\&q = q$

$\overline{\&}*q = q$

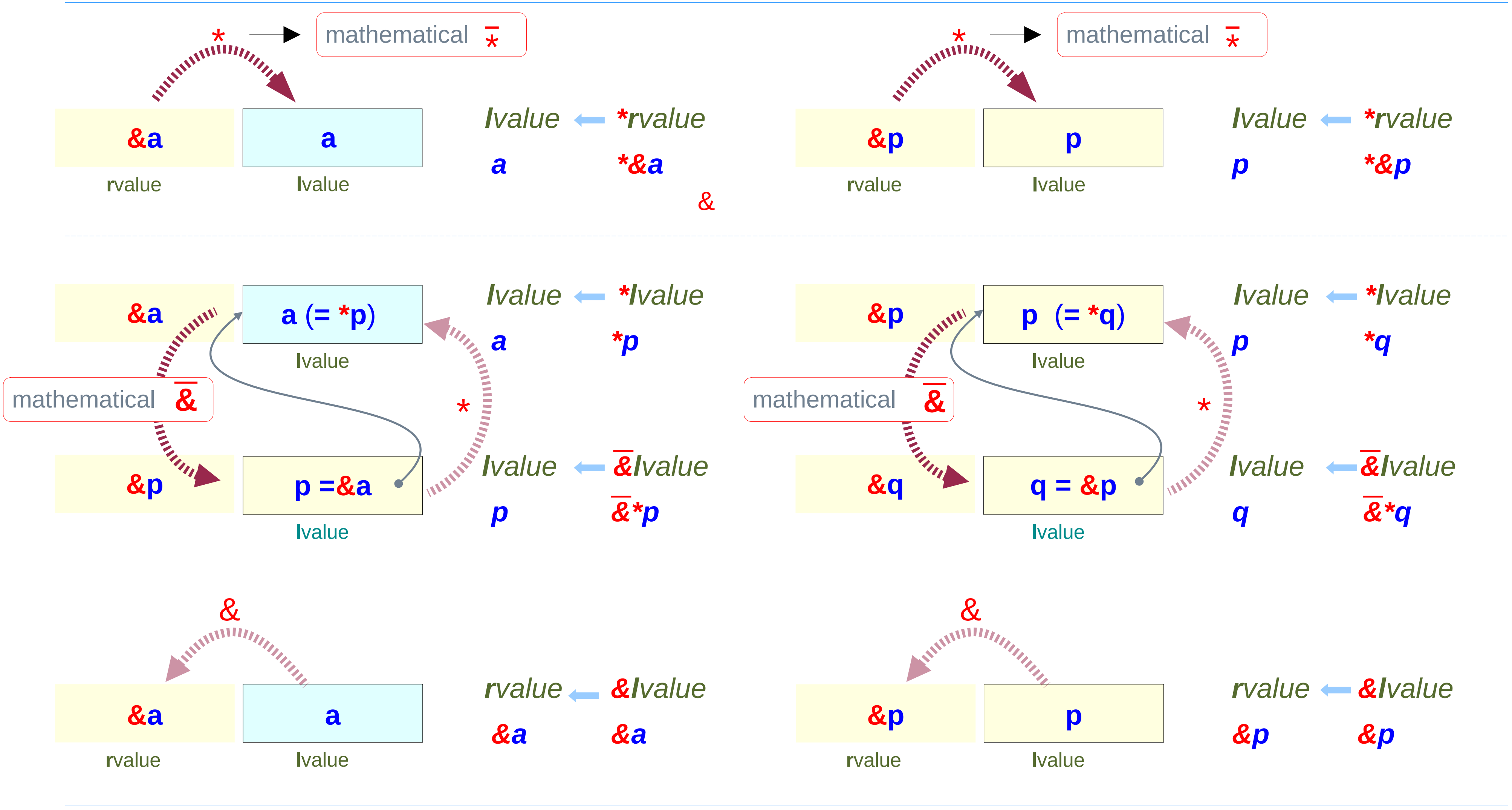
$*\&q = q$

$\&*q = \text{value}(q)$





Introducing mathematical operators : $\overline{\&}$ and $\overline{*}$



* → mathematical $\overline{*}$

&a rvalue | a lvalue

lvalue ← **rvalue*

a **&a*

&

* → mathematical $\overline{*}$

&p rvalue | p lvalue

lvalue ← **rvalue*

p **&p*

&a | a (= *p) lvalue

mathematical $\overline{\&}$

*

&p | p =&a lvalue

lvalue ← **lvalue*

a **p*

lvalue ← $\overline{\&}$*lvalue*

p $\overline{\&}$**p*

&p | p (= *q) lvalue

mathematical $\overline{\&}$

*

&q | q = &p lvalue

lvalue ← **lvalue*

p **q*

lvalue ← $\overline{\&}$*lvalue*

q $\overline{\&}$**q*

&

&a rvalue | a lvalue

rvalue ← *&lvalue*

&a *&a*

&

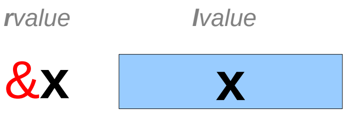
&p rvalue | p lvalue

rvalue ← *&lvalue*

&p *&p*

& and * operators in two steps (1)

Two step Address-of operation

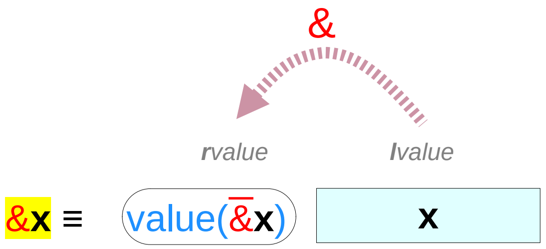


*&**x** evaluates the address value of a variable **x***

devise a new operator :

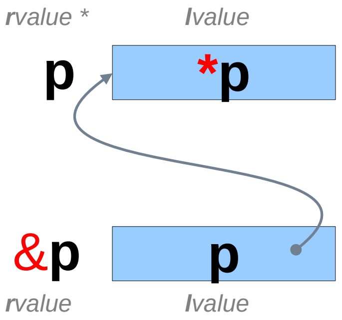
$\overline{\&}$ is a mathematical operator

$\overline{\&}$ takes **l**value and returns **l**value



& in two steps:
first $\overline{\&}$, then value()

Two step De-reference operation



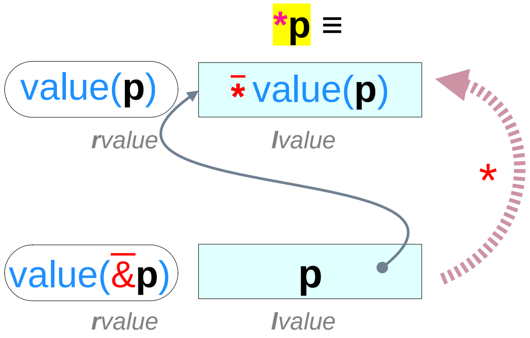
p** must be evaluated to an address before de-referencing ***p

devise a new operator :

$\overline{*}$ is a mathematical operator

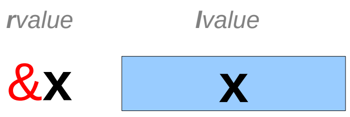
$\overline{*}$ takes **r**value and returns **l**value

***** *in two steps:*
first value(), then $\overline{*}$



& and * operators in two steps (2)

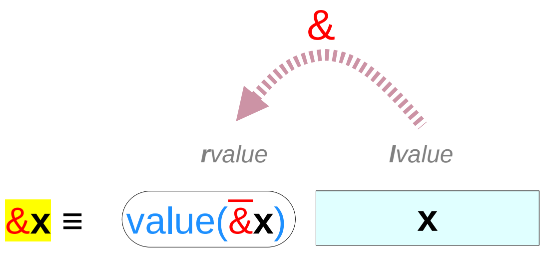
Two step Address-of operation



$\&x = \text{value}(\overline{\&}x)$

C Expressions

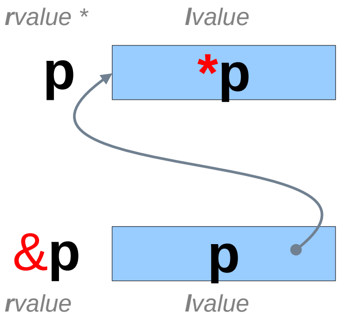
Mixed Expressions



& in two steps:
first $\overline{\&}$, then value()

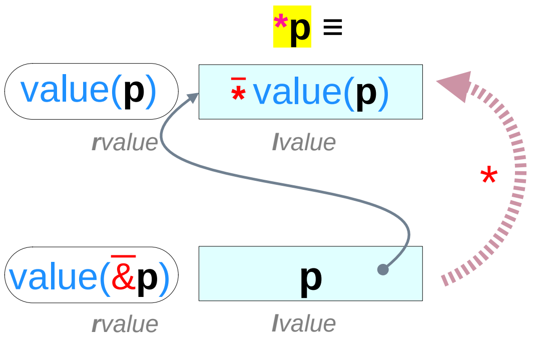
Two step De-reference operation

* in two steps:
first value(), then $\overline{*}$



$*p = \overline{*}\text{value}(p)$

Mixed Expressions



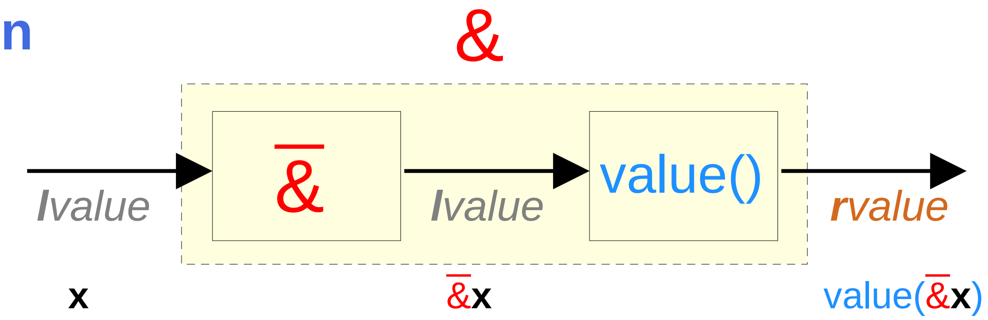
& and * operators in two steps (3)

Two step Address-of operation

&x = value($\overline{\&}$**x**)

C Expressions

Mixed Expressions

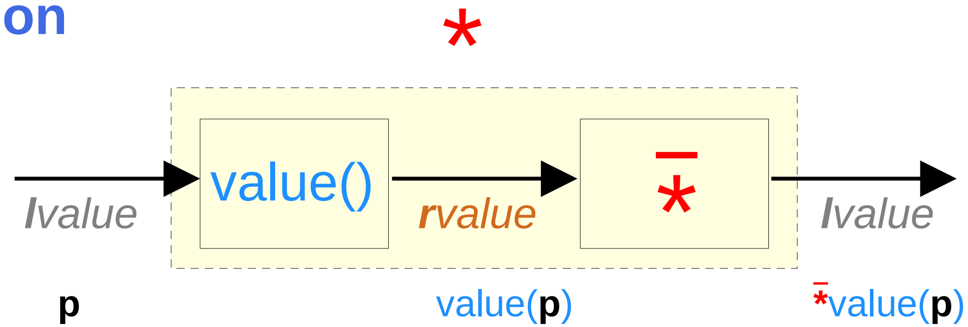


Two step De-reference operation

***p** = $\overline{*}$value(**p**)

C Expressions

Mixed Expressions



Inverse operators of & and *

Two step Address-of operation

$\&x$ = $\text{value}(\overline{\&}x)$

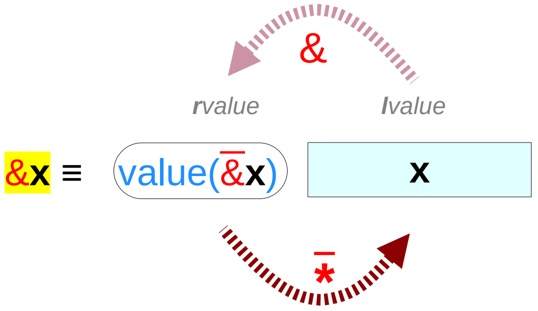
C Expressions

Mixed Expressions

$\&x \equiv \text{value}(\overline{\&}x)$

$\overline{*}\ \&x = x$

$\overline{*}\ \text{value}(\overline{\&}x) = x$



Two step De-reference operation

$*p$ = $\overline{*}\text{value}(p)$

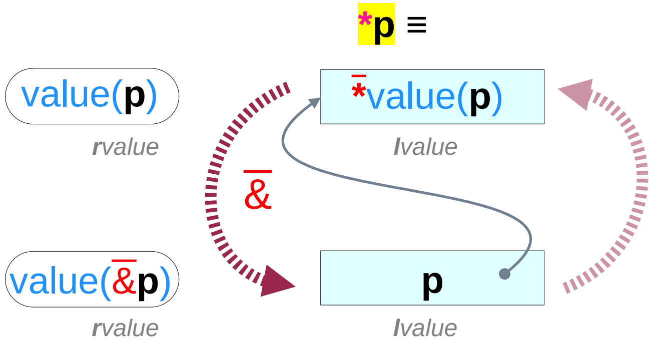
C Expressions

Mixed Expressions

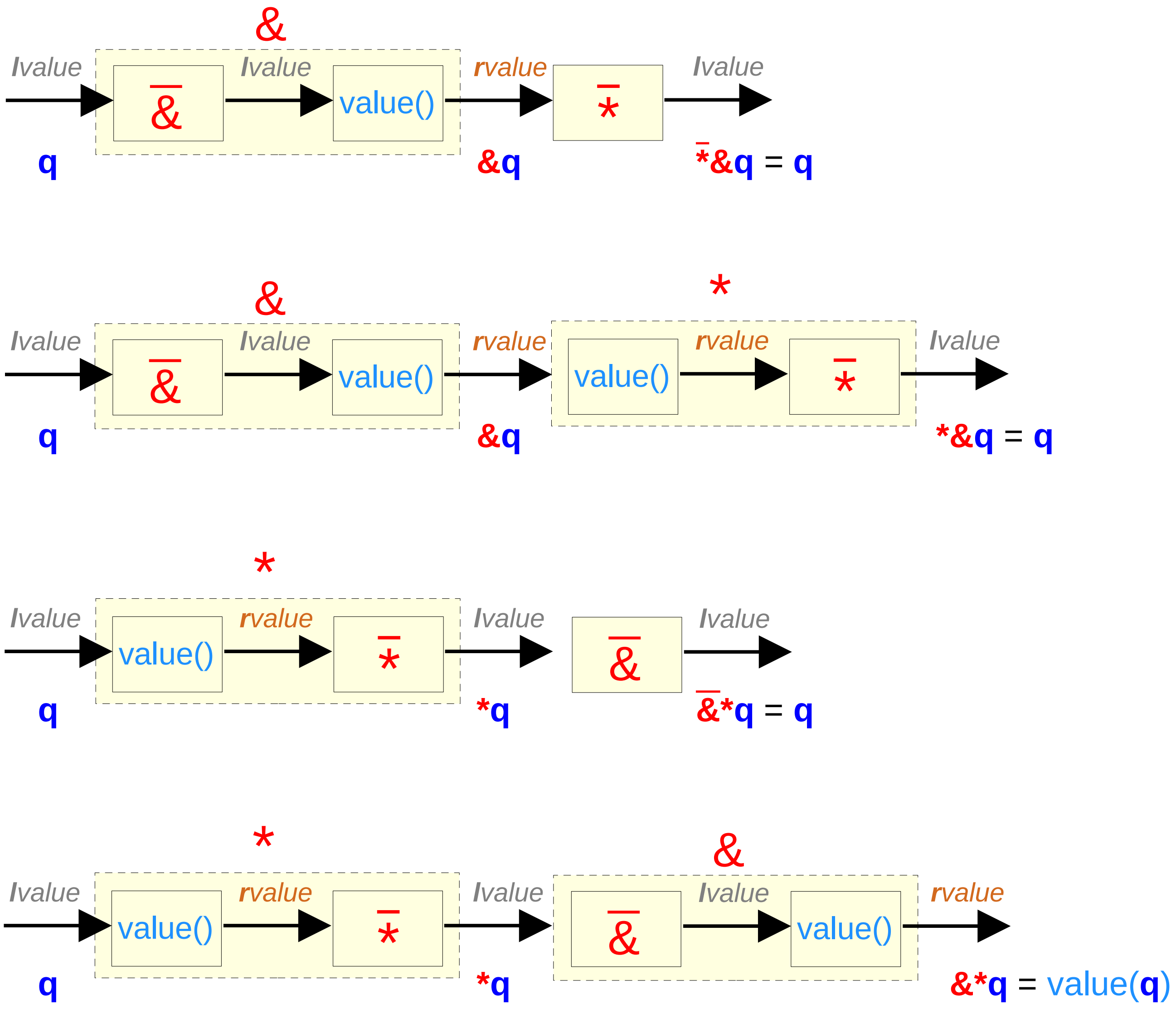
$*p \equiv \overline{*}\text{value}(p)$

$\overline{\&}\ *p = p$

$\overline{\&}\ \overline{*}\text{value}(p) = p$



Inverse operators of & and *



$\overline{*}\&q = q$

value(value(rvalue)) = rvalue

$*\&q = q$

$\overline{\&}*q = q$

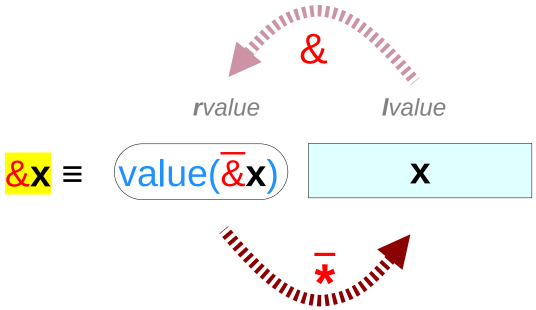
&*q = value(q)

Inverse operators of & and *

&x ≡ value($\overline{\&}$x)

$\cancel{\overline{*}}\ \cancel{\&}x = x$

$\overline{*}$ value($\overline{\&}$x) = x

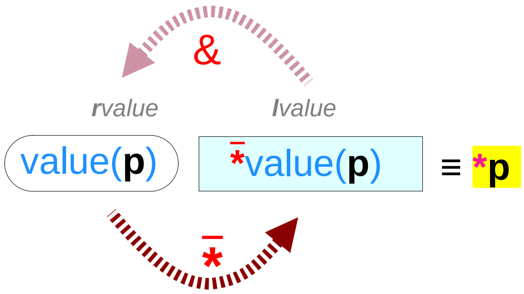


*p ≡ $\overline{*}$value(p)

$\cancel{\&}\ \cancel{*}p$ = value(p)

&*p = value($\overline{\&}$*p)

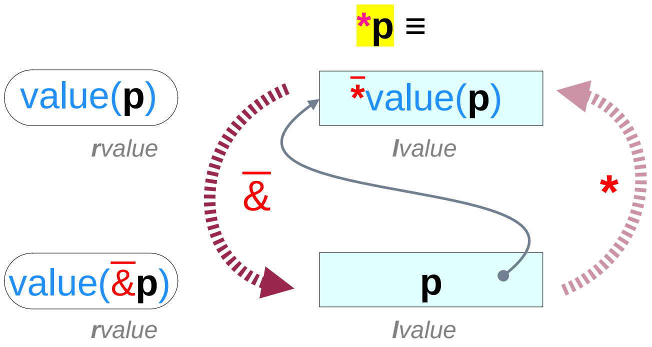
&*p = value(p)



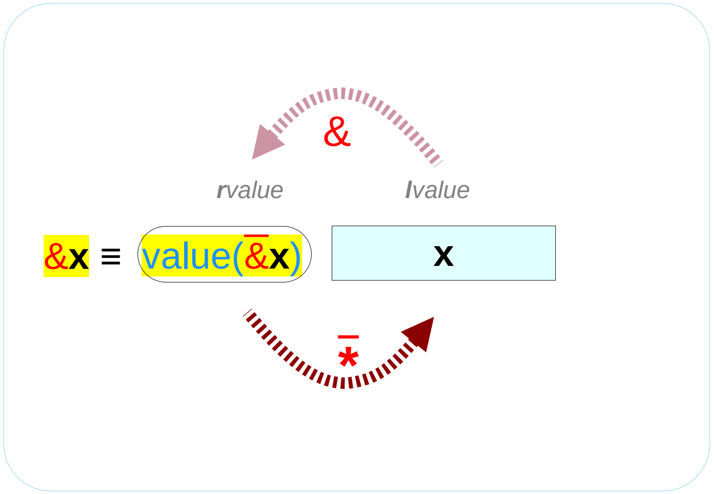
*p ≡ $\overline{*}$value(p)

$\cancel{\overline{\&}}\ \cancel{*}p = p$

$\overline{\&}$ $\overline{*}$value(p) = p



Inverse operators of **&** and *****



&

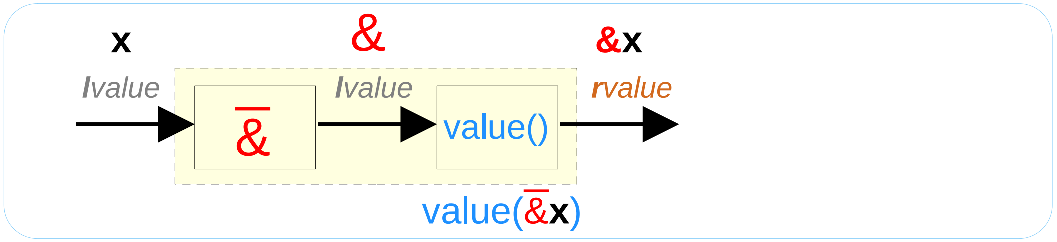
rvalue

lvalue

&x ≡ value($\overline{\&}$**x**)

x

$\overline{*}$



x

lvalue

&

$\overline{\&}$

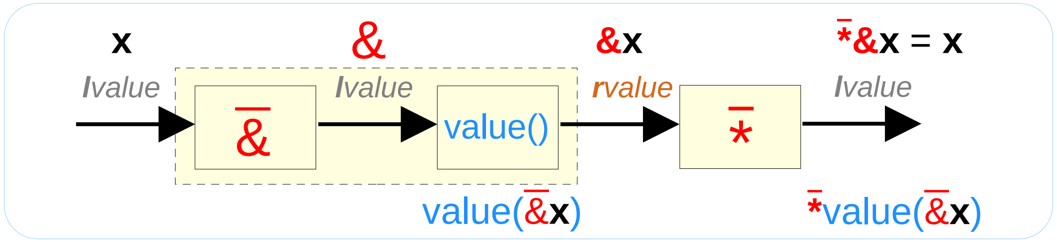
lvalue

value()

&x

rvalue

value($\overline{\&}$**x**)



x

lvalue

&

$\overline{\&}$

lvalue

value()

&x

rvalue

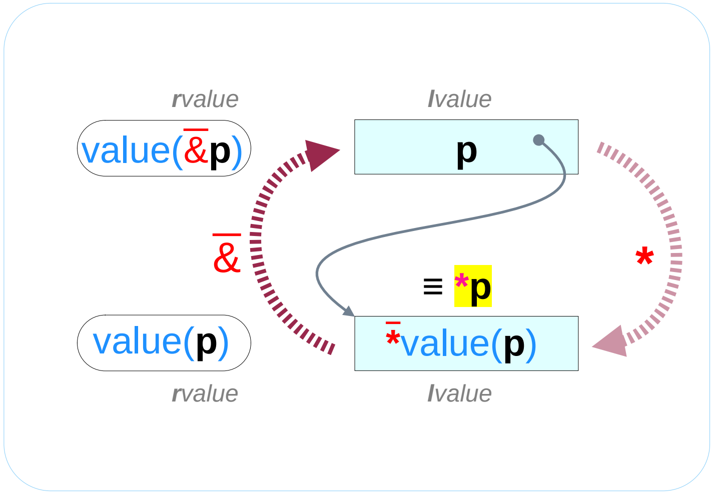
$\overline{*}$

$\overline{*}$**&x = x**

lvalue

value($\overline{\&}$**x**)

$\overline{*}$value($\overline{\&}$**x**)



rvalue

lvalue

value($\overline{\&}$**p**)

p

$\overline{\&}$

*

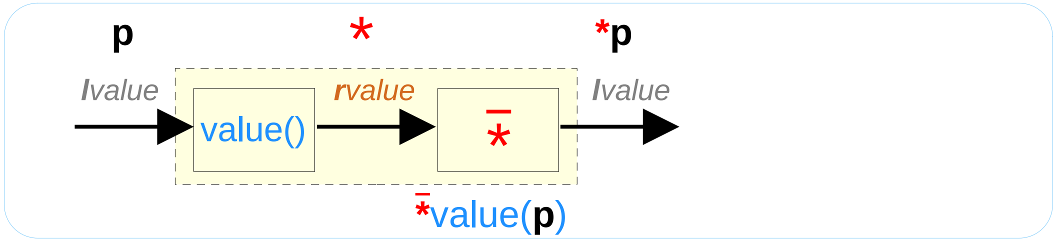
≡ ***p**

value(**p**)

$\overline{*}$value(**p**)

rvalue

lvalue



p

lvalue

*

value()

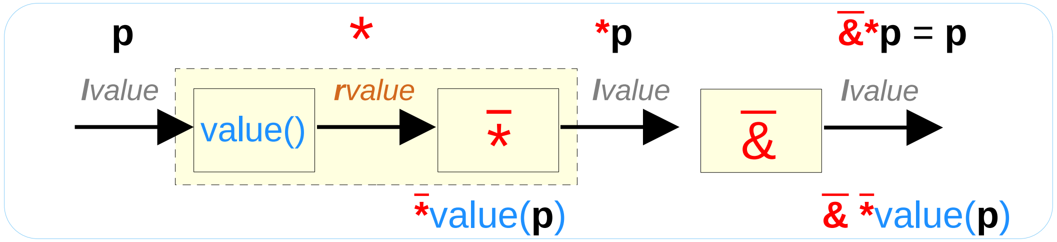
rvalue

$\overline{*}$

***p**

lvalue

$\overline{*}$value(**p**)



p

lvalue

*

value()

rvalue

$\overline{*}$

***p**

lvalue

$\overline{\&}$

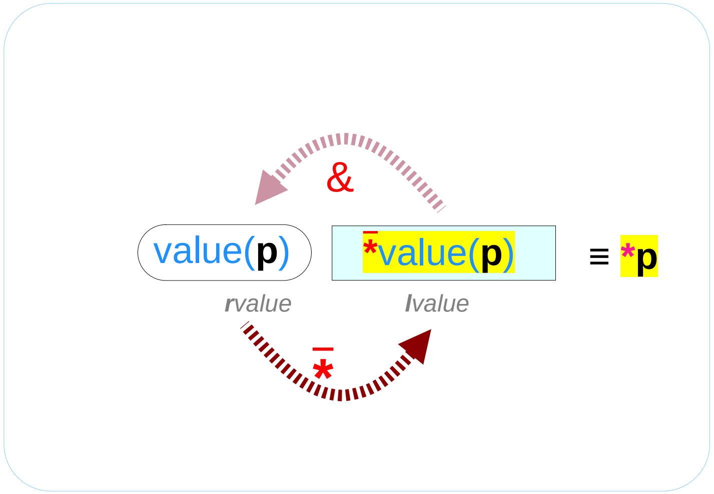
$\overline{\&}$***p = p**

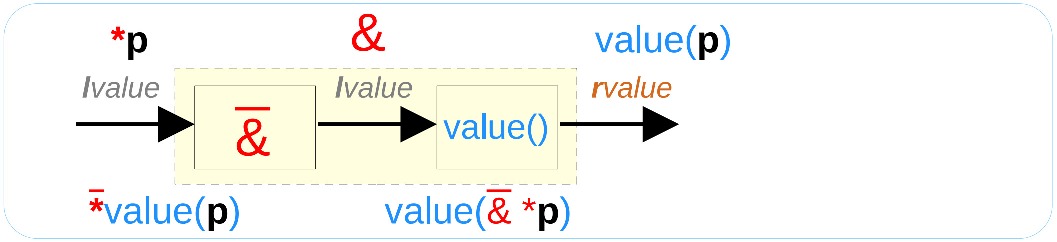
lvalue

$\overline{*}$value(**p**)

$\overline{\&}$ $\overline{*}$value(**p**)

Inverse operators of & and *





$\overline{*}$ value($\overline{\&}$x) = x

$\overline{\&}$ $\overline{*}$value(p) = p

&*p = value(p)

value($\overline{\&}$ $\overline{*}$value(p))

value($\overline{\&}$ *p) ⇒ value(p)

value($\overline{\&}$ $\overline{*}$value(p)) ⇒ value(p)

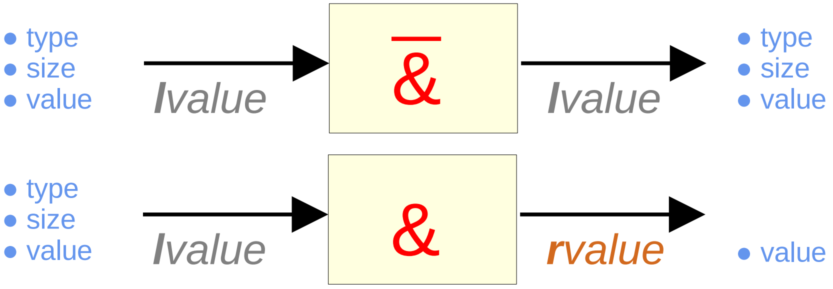
C operators and mathematical operators

Address-of operation

$\&x \quad = \quad \text{value}(\overline{\&}x)$

C Expressions

Mixed Expressions

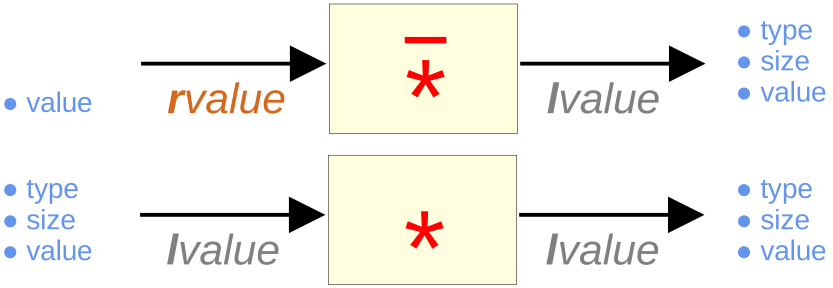


De-reference operation

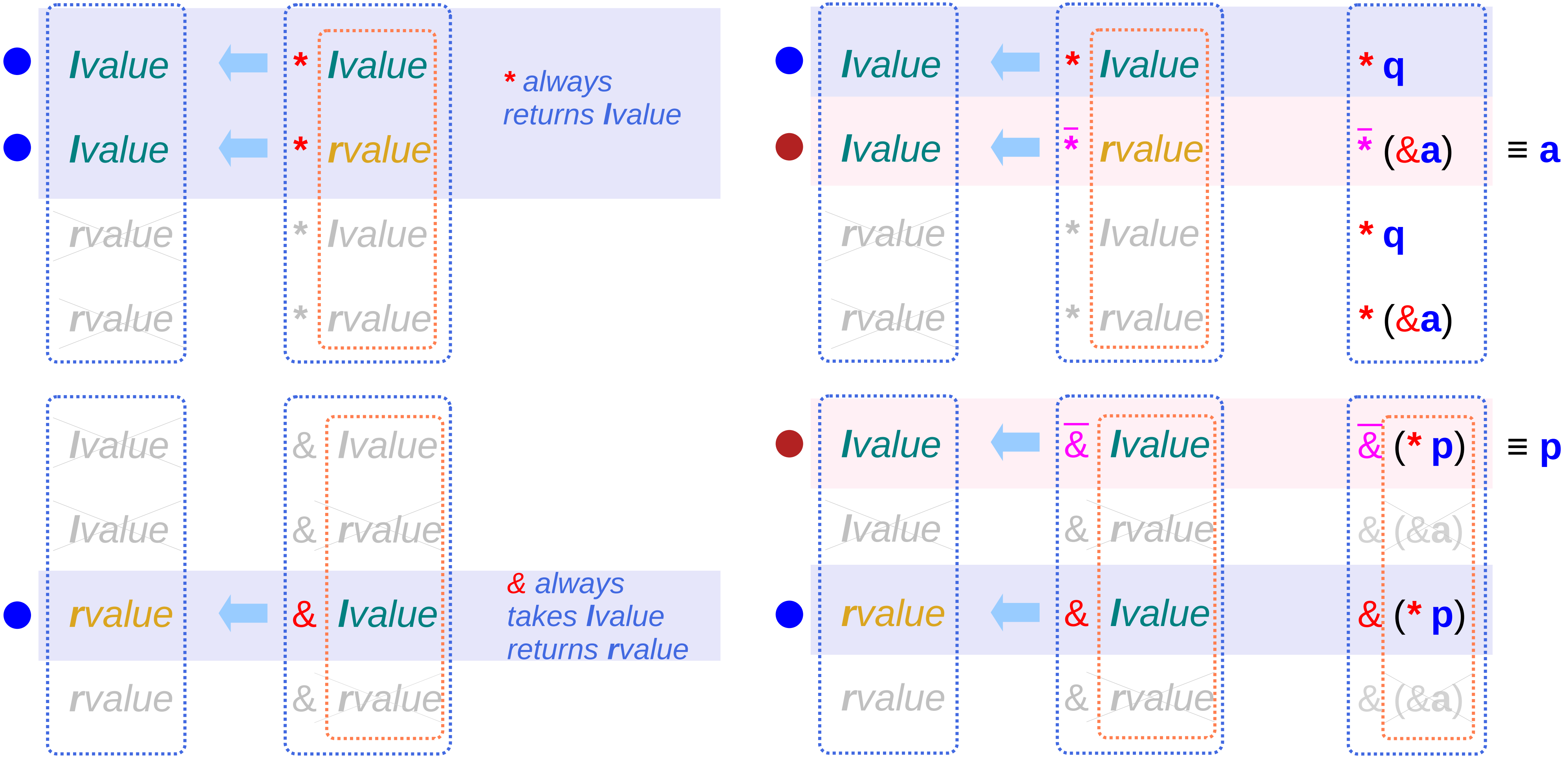
$*p \quad = \quad \overline{*}\text{value}(p)$

C Expressions

Mixed Expressions



lvalue and rvalue with * and & operators

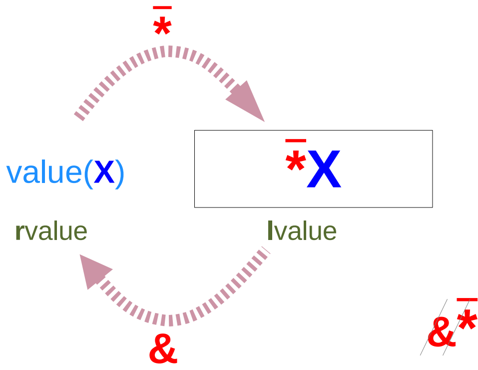


* always returns lvalue

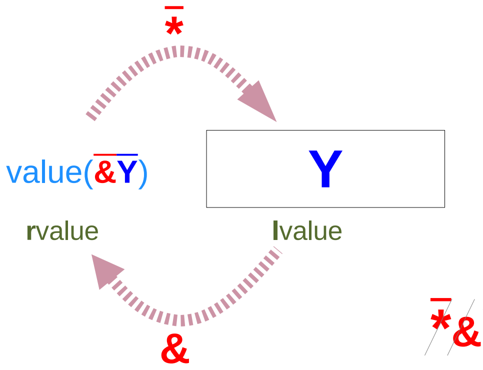
& always takes lvalue returns rvalue

a, **p**, **q** : lvalues … variables … RW
*__p__, *__q__, **__q__ : lvalues … variables … RW
&__a__, &__p__, &__q__ : rvalues … constants … RO

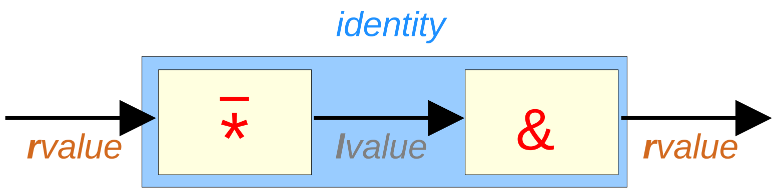
& and mathematical $\overline{*}$



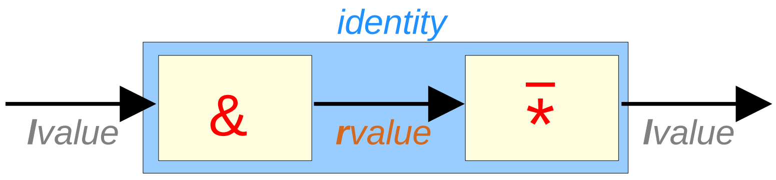
$\cancel{\&}\cancel{\overline{*}}X = X$



$\cancel{\overline{*}}\cancel{\&}Y= Y$

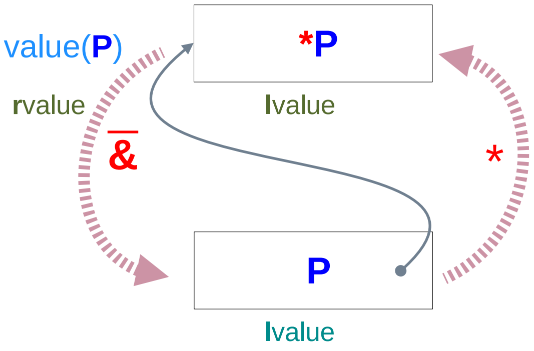


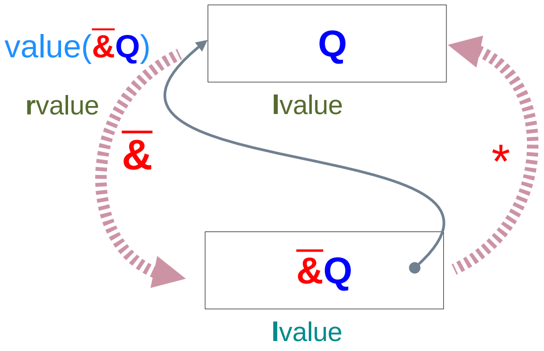
X $\qquad \overline{*}$X $\qquad \&\overline{*}X = X$



Y $\qquad$ &Y $\qquad \overline{*}\&Y = Y$

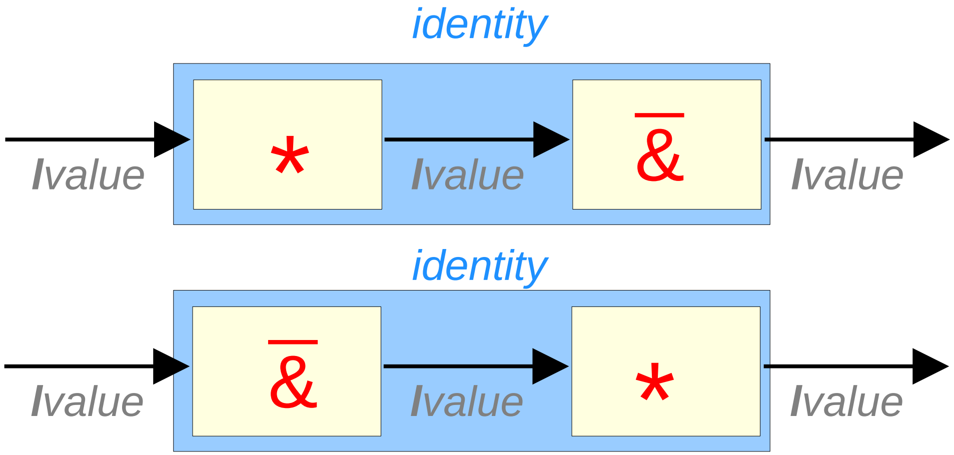
* and mathematical $\overline{\&}$





$\cancel{\overline{\&}}\cancel{*}P = P$

$\cancel{*}\cancel{\overline{\&}}Q= Q$



Recursive application of the address-of operator

$$\xcancel{\&(\&(\&(c[i])[j])[k])}$$

$$\overline{\&}(\overline{\&}(\overline{\&}(c[i])[j])[k])$$

& C operator

can be applied to only **lvalue** variable

returns address value

thus, the above expression is not possible

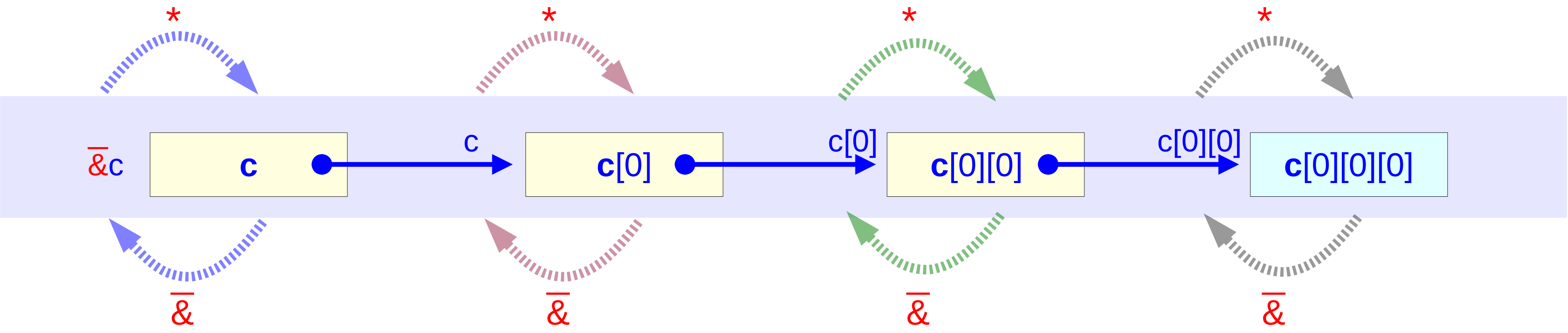
successive application of & is not possible

In constrast, ***p** becomes a lvalue variable

* operator can be applied successively.

$\overline{\&}$ mathematical operator

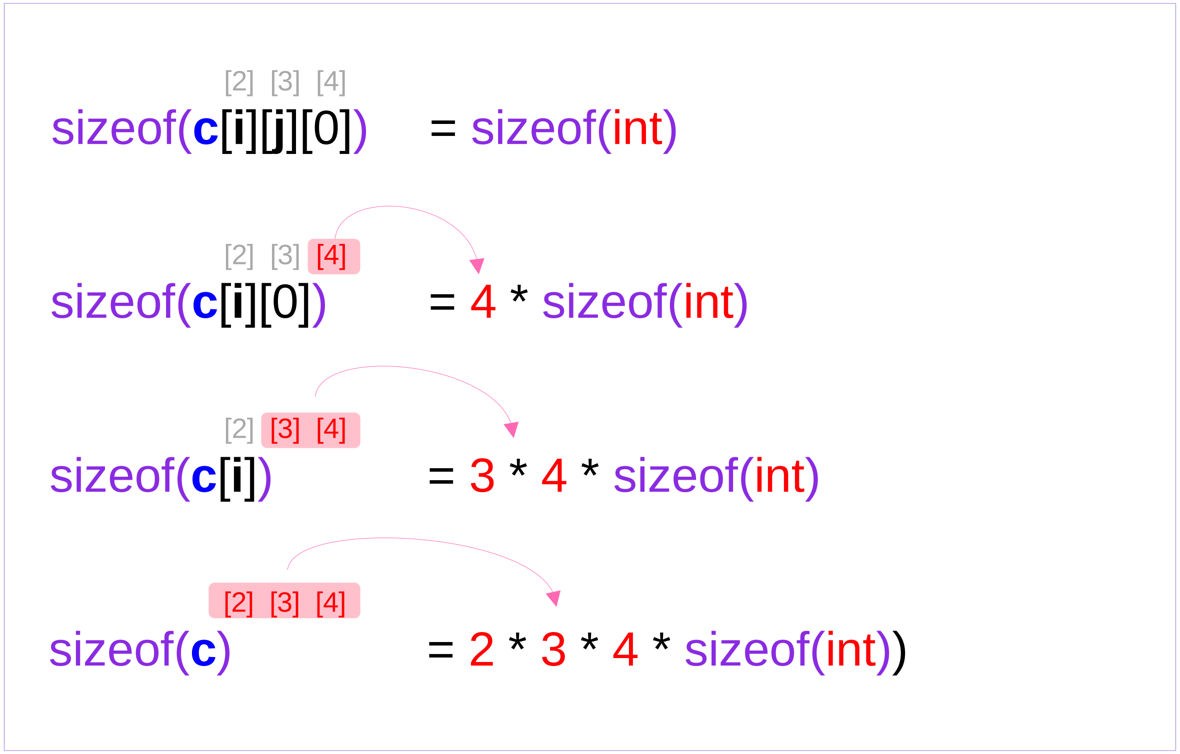
Two step deferencing in type **II** (1) – without skipping



$\overline{\&}$: **mathematical & operator**

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)

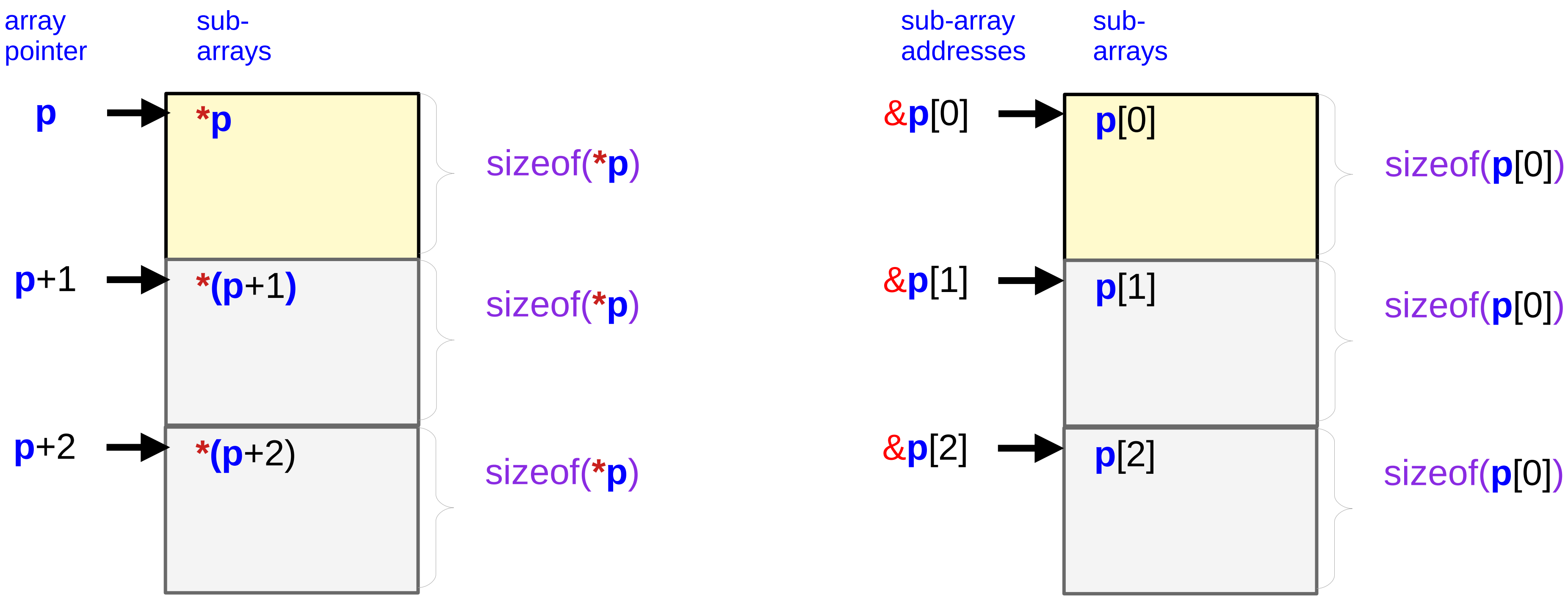
[2] [3] [4]

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

[2] [3] [4]

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

Pointer increments and byte addresses



array pointer

sub-arrays

p

*p

sizeof(*p)

p+1

*(p+1)

sizeof(*p)

p+2

*(p+2)

sizeof(*p)

sub-array addresses

sub-arrays

&p[0]

p[0]

sizeof(p[0])

&p[1]

p[1]

sizeof(p[0])

&p[2]

p[2]

sizeof(p[0])

byte address — byte address — byte size

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

math expression with an explicit size information

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

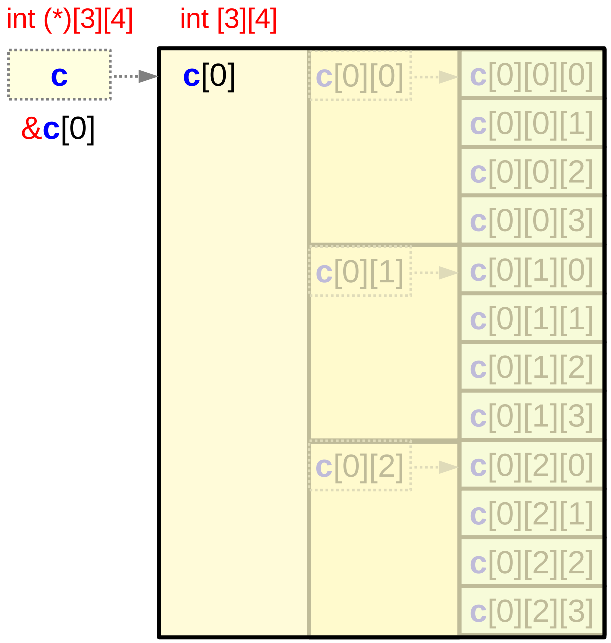
byte address — byte address — byte size

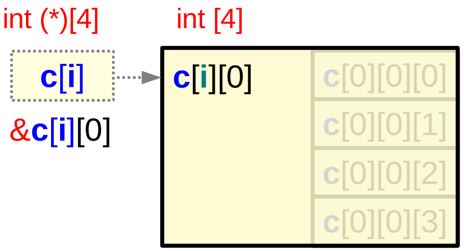
value(&p[i]) = value(p) + i * sizeof(p[0])

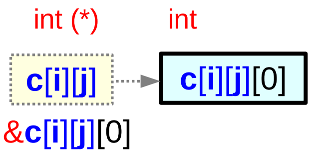
math expression with an explicit size information

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

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







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

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

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]

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

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]

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

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

Address replications and subarray addresses

Address Replication

transferring pointing address
to the pointer that references itself

&X = X

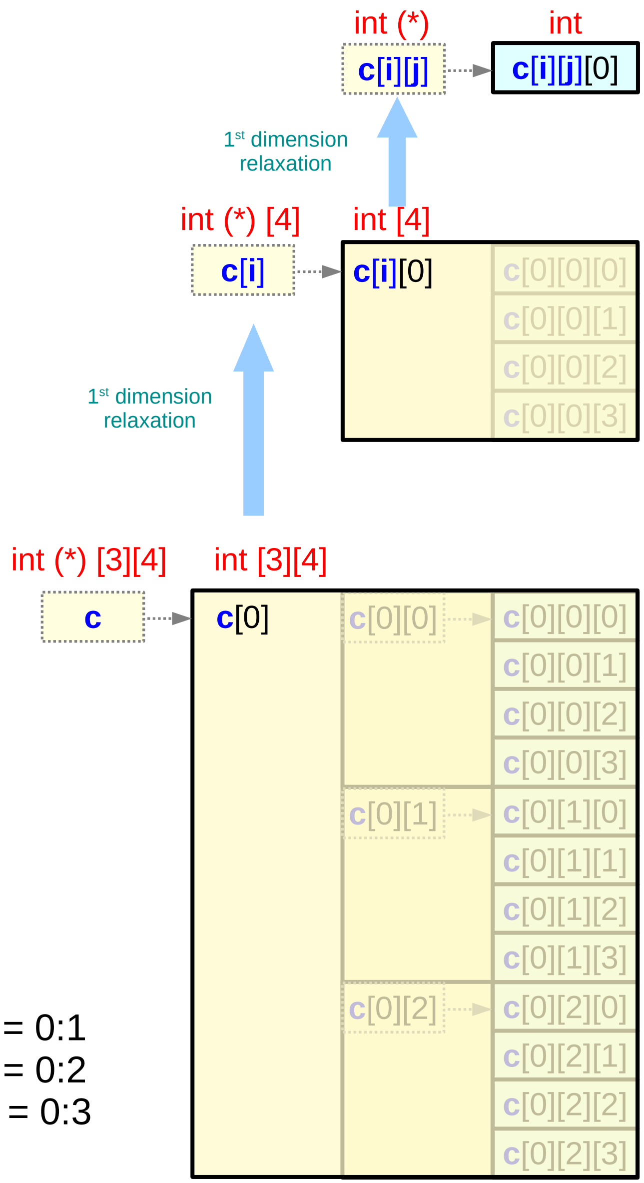
int (*) int
value(&c[i][j][k]) = value(c[i][j]) + k * sizeof(int)
int (*)

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

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

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

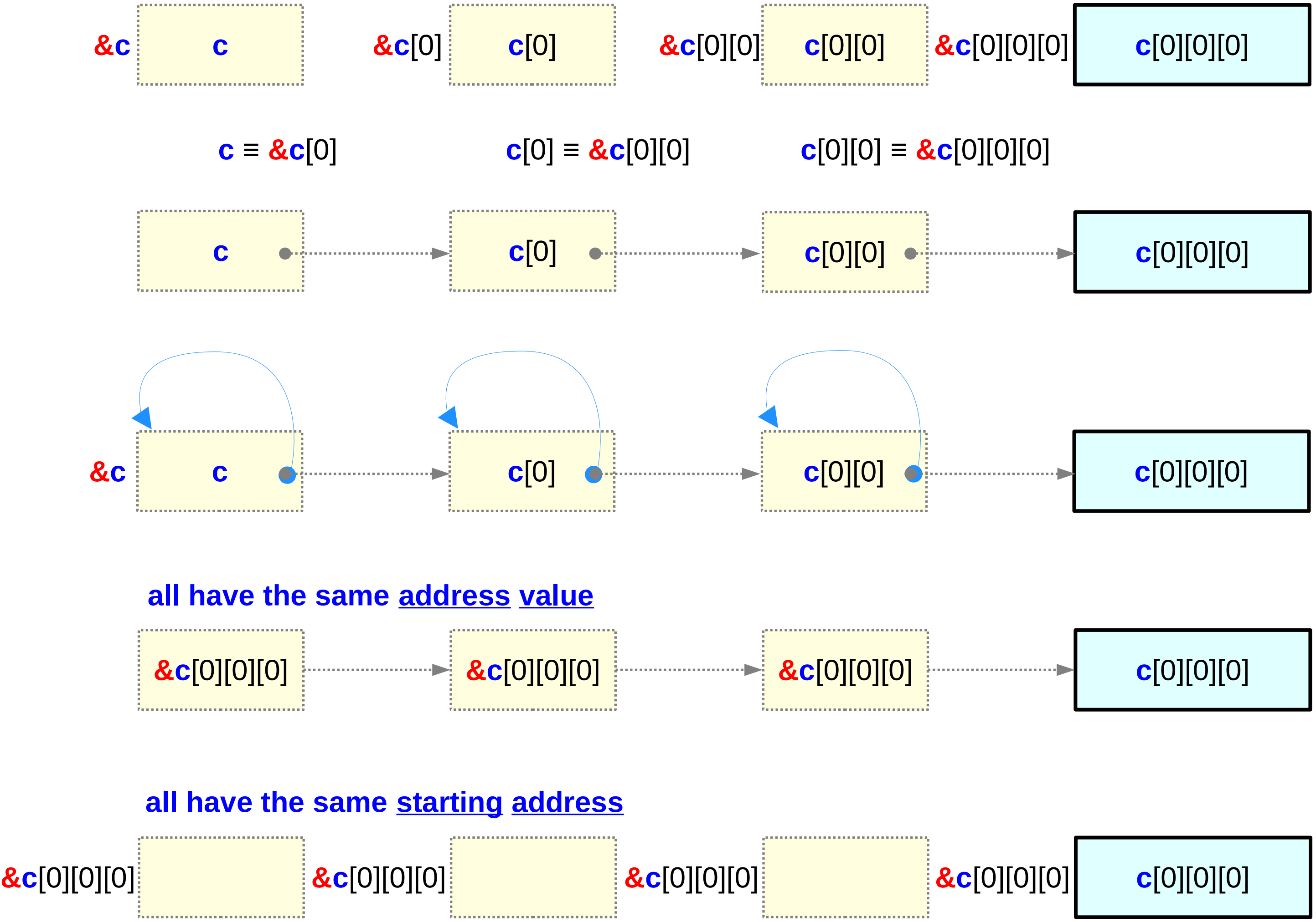
c[**i**][**j**] ≡ &**c**[**i**][**j**][0]
c[**i**] ≡ &**c**[**i**][0]
c ≡ &**c**[0]

address replication

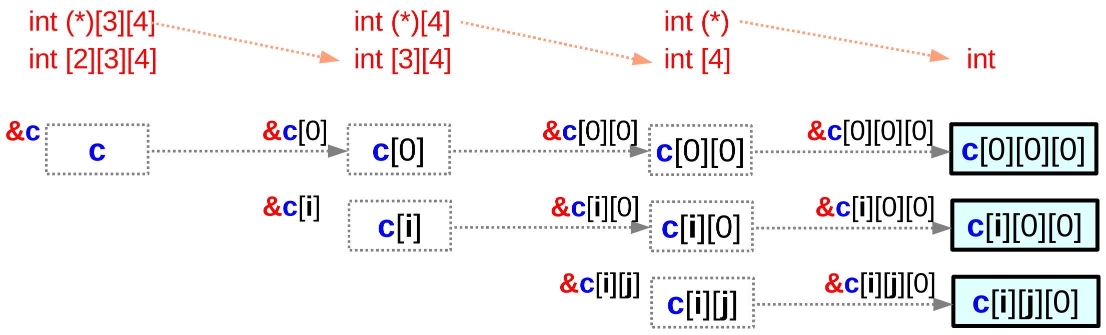
value(**c**[**i**][**j**]) = value(&**c**[**i**][**j**])
value(**c**[**i**]) = value(&**c**[**i**])
value(**c**) = value(&**c**)

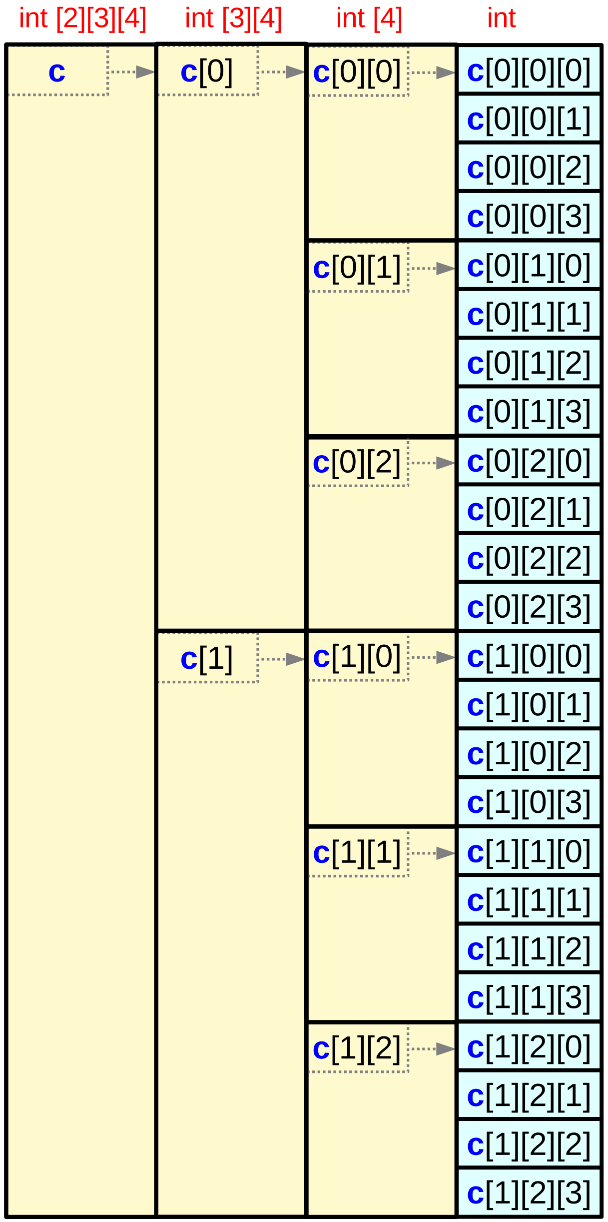
c, **c**[0], **c**[0][0] :
these virtual pointers have
the same address value

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]

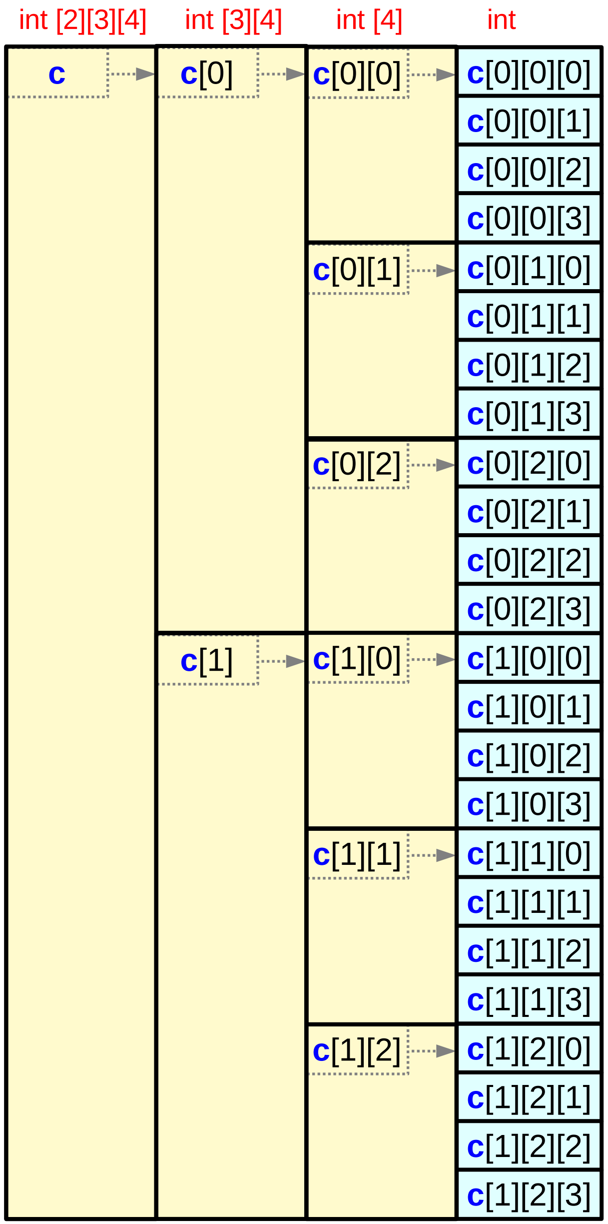
Types, sizes, and values of sub-arrays

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 (*)



int [2][3][4]   int [3][4]   int [4]   int

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

pointers to arrays

Using multi-dimensional arrays

Pointer Array Approach

– using <u>explicit</u> pointers

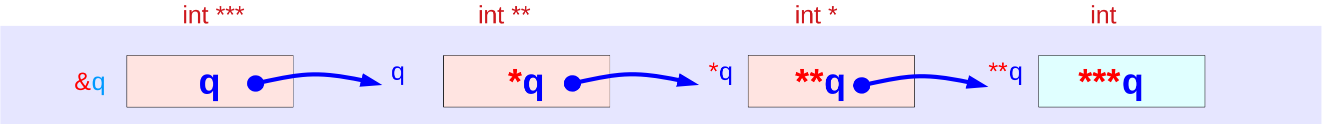
Array Pointer Approach

– using <u>implicit</u> pointers

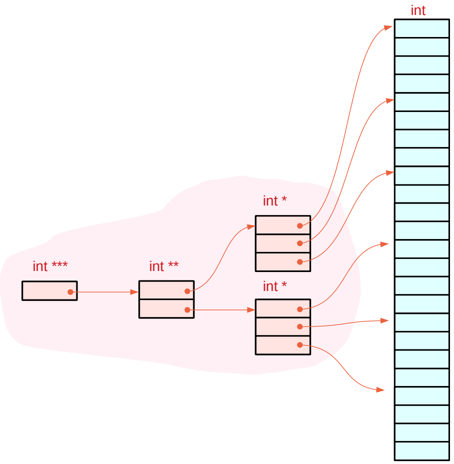
Two types of **3-d** array accesses

Pointer Array Approach (arrays of pointers)

Pointer Chain Type I

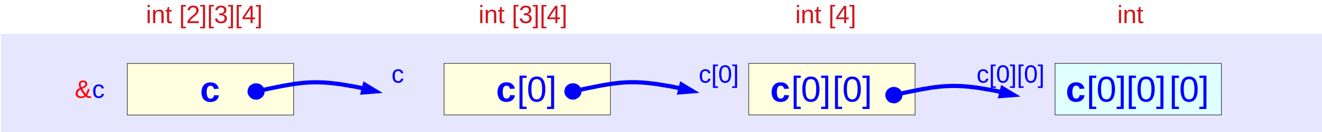

int *** int ** int * int

&q **q** q ***q** *q ****q** **q *****q**

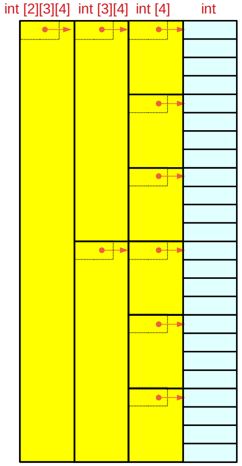


Array Pointer Approach (pointers to arrays)

Pointer Chain Type II


int [2][3][4] int [3][4] int [4] int

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



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**]

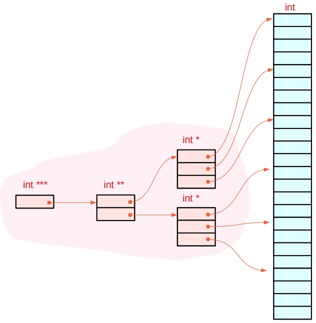
skip **k** elements
of **c**[**i**][**j**][**k**] from **c**[**i**][**j**]

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

Pointer Array Approach

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

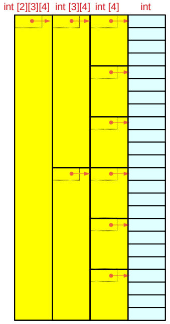
sizeof(***c**) = 1*4 sizeof(***c**[**i**]) = 1*4 sizeof(***c**[**i**][**j**]) = 1*4



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**)

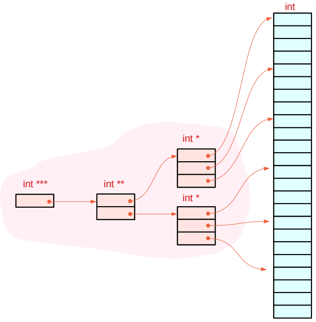
skip **j** elements
of **c**[**i**][**j**] from **c**[**i**]

(**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**

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

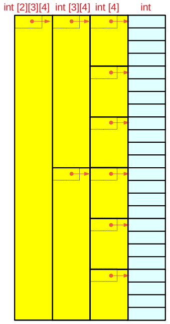
skip **j** * sizeof(int *)
(= **j***4) bytes from **c**[**i**]

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

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

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

Array Pointer Approach



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

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

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

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

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

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

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

Pointer Array Approach

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

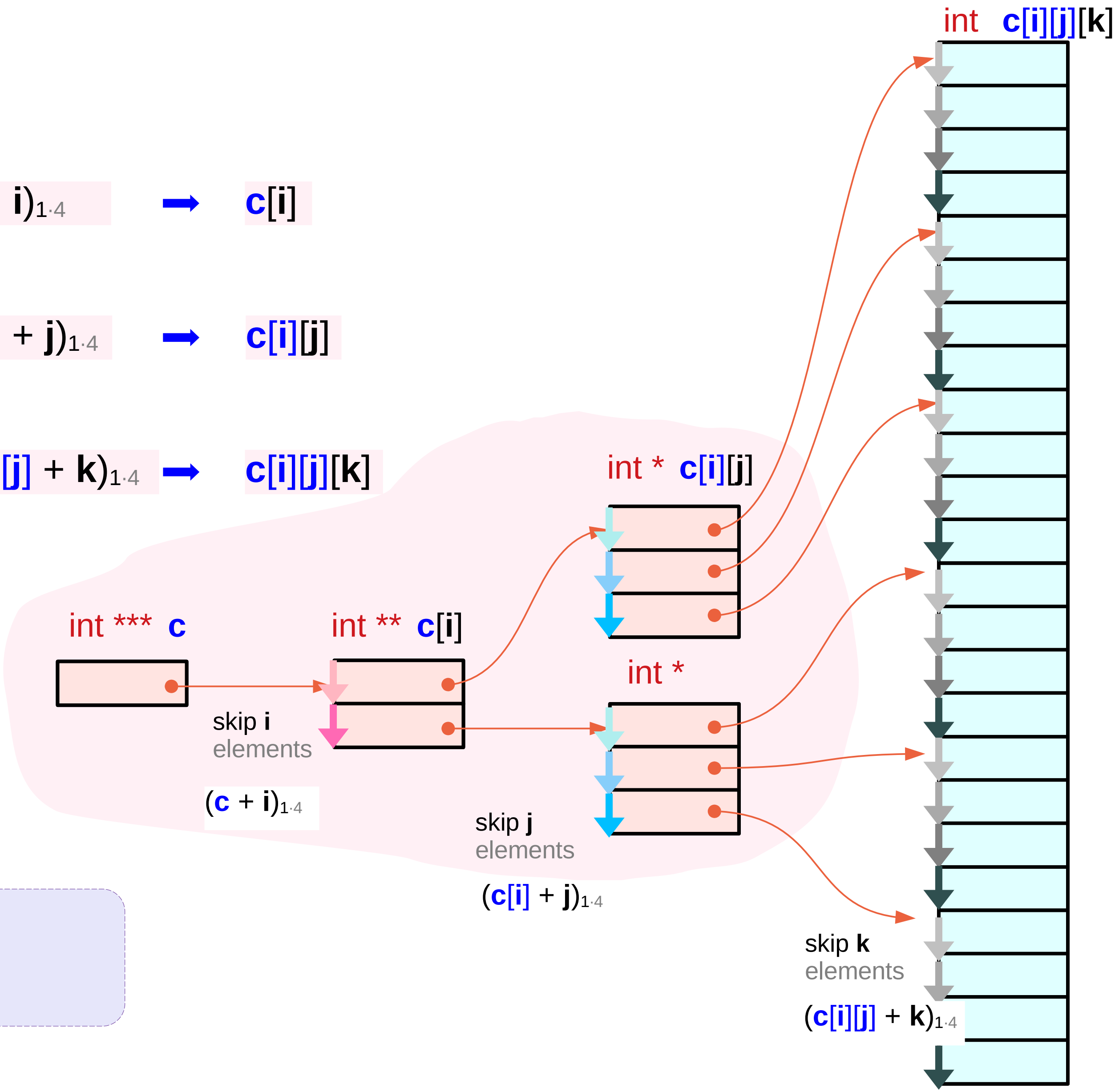
skip **j** * sizeof(int *)
= **j** *4 bytes from **c[i]**

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

$(c + i)_{1\cdot 4}$ ➡ **c[i]**

$(c[i] + j)_{1\cdot 4}$ ➡ **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**

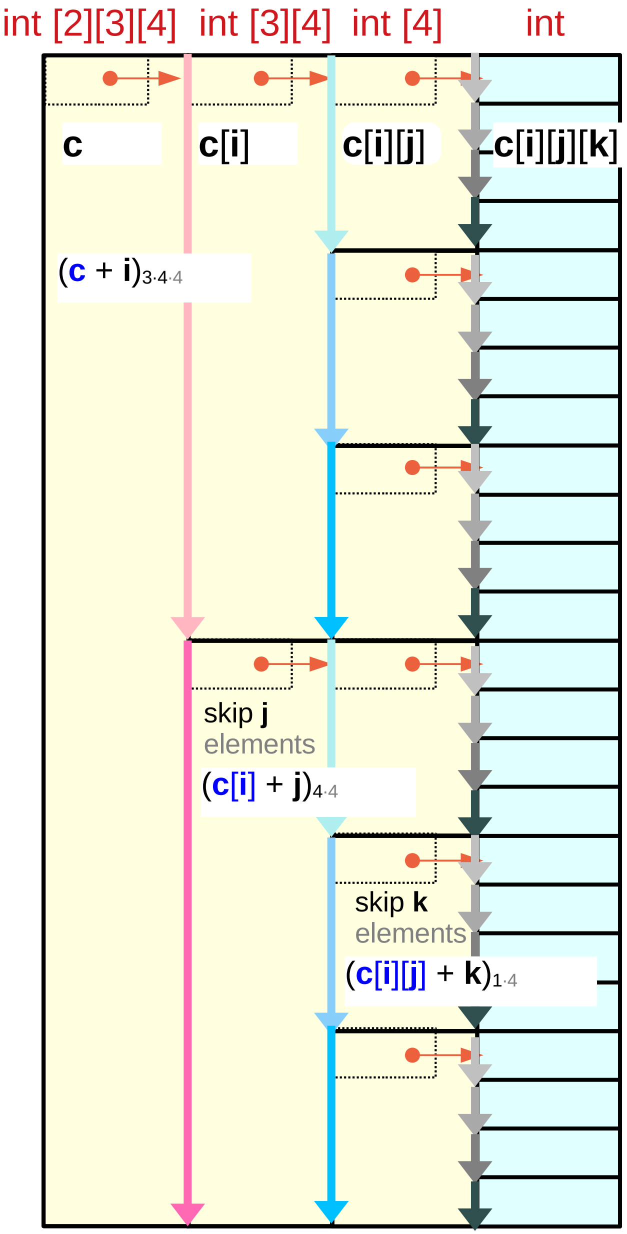
$(\mathbf{c} + \mathbf{i})_{3\cdot4\cdot4}$ ➡ **c[i]**

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

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

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

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

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

&c[i][j][k] = value($(c[i][j] + k)_{1\cdot 4}$)
= value(*value($(c[i] + j)_{1\cdot 4}$) + k * 4)
= value(*value(*value($(c + i)_{1\cdot 4}$) + 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 <u>size</u> and <u>address</u> information***

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

Pointer Array Approach

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

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

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

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

&c[i] = value((c + i)$_{3\cdot4\cdot4}$) = 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

different semantics !
be careful in mixed c and math expressions

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

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

different semantics !
be careful in mixed c and math expressions

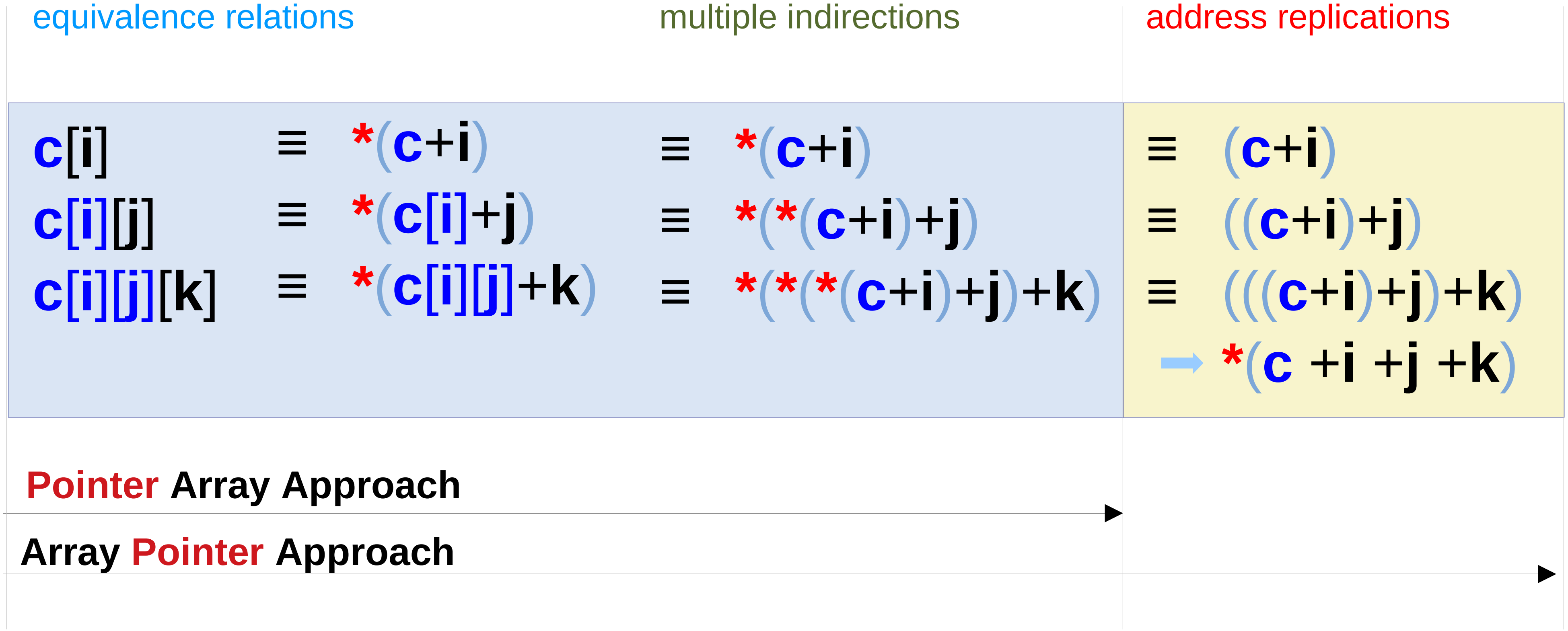
c[i][j][k] = *value(c[i][j] + k * 4)
c[i][j] = *value(c[i] + j * 4*4)
c[i] = *value(c + i * 3*4*4)

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**)$_{1\cdot4}$	skip **k** * 4 bytes from **c[i][j]**
(**c[i]** + **j**)$_{1\cdot4}$	skip **j** * 4 bytes from **c[i]**
(**c** + **i**)$_{1\cdot4}$	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**)$_{1\cdot4}$	skip **k** * 4 bytes from **c[i][j]**
(**c[i]** + **j**)$_{4\cdot4}$	skip **j** * 4*4 bytes from **c[i]**
(**c** + **i**)$_{3\cdot4\cdot4}$	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]

[2][3][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)

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**)$_{1\cdot4}$

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

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

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

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

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]

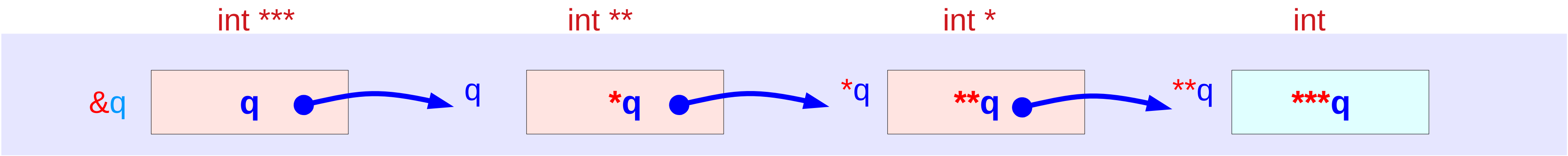
value(**c[i][j][k]**) = &**c[i][j][k]**

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

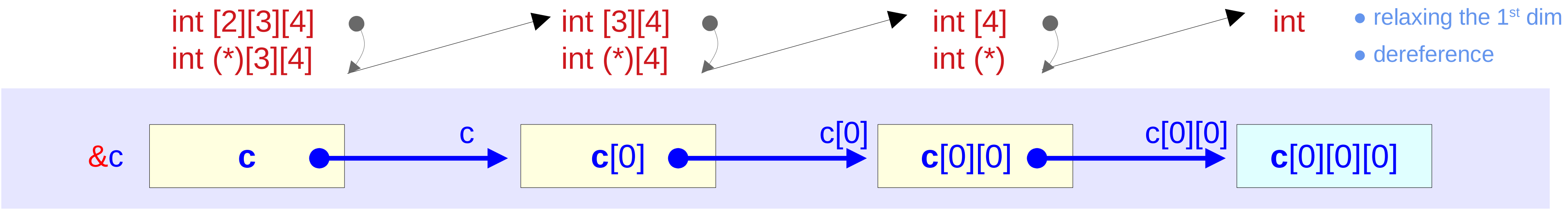
Pointer Chain Types

Pointer Chain Type I



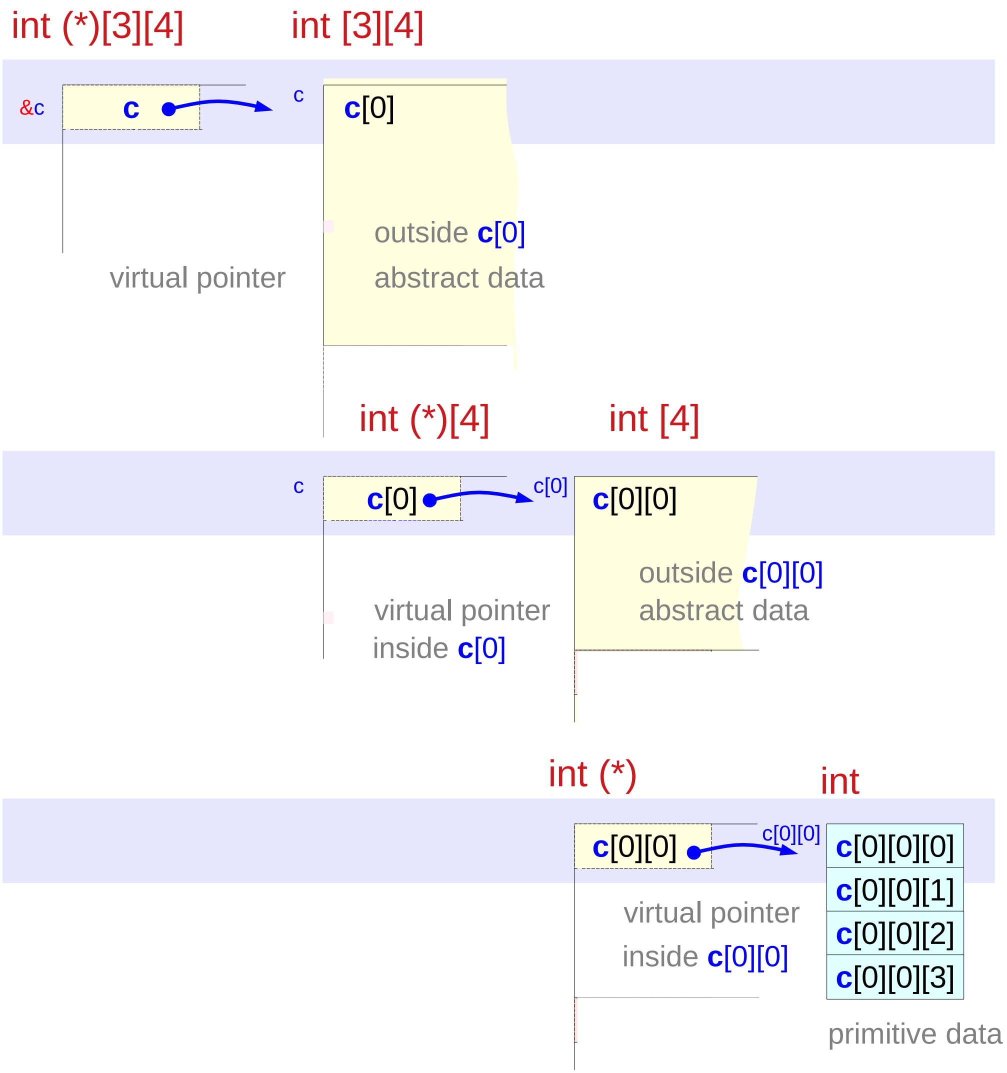
- **Pointer Array Approach** (index operations are handled by a user)

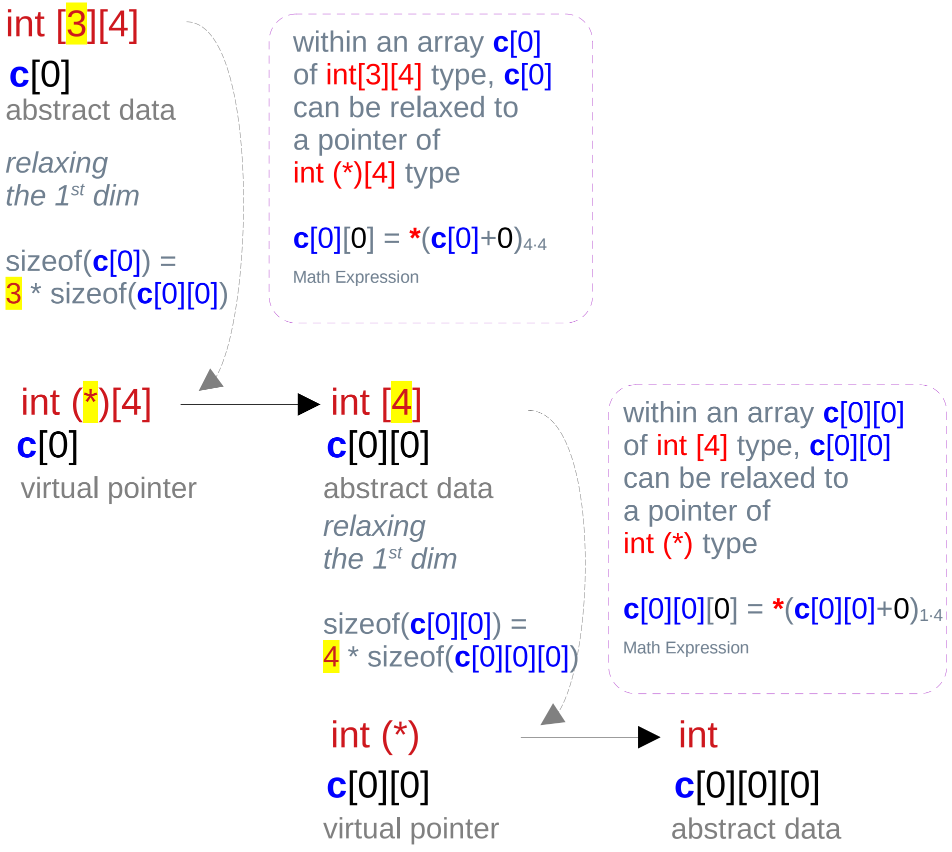
Pointer Chain Type II



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

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





relaxing the 1st dim

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

within an array **c**[0] of int[3][4] type, **c**[0] can be relaxed to a pointer of int (*)[4] type

$c[0][0] = *(c[0]+0)_{4\cdot 4}$

Math Expression

relaxing the 1st dim

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

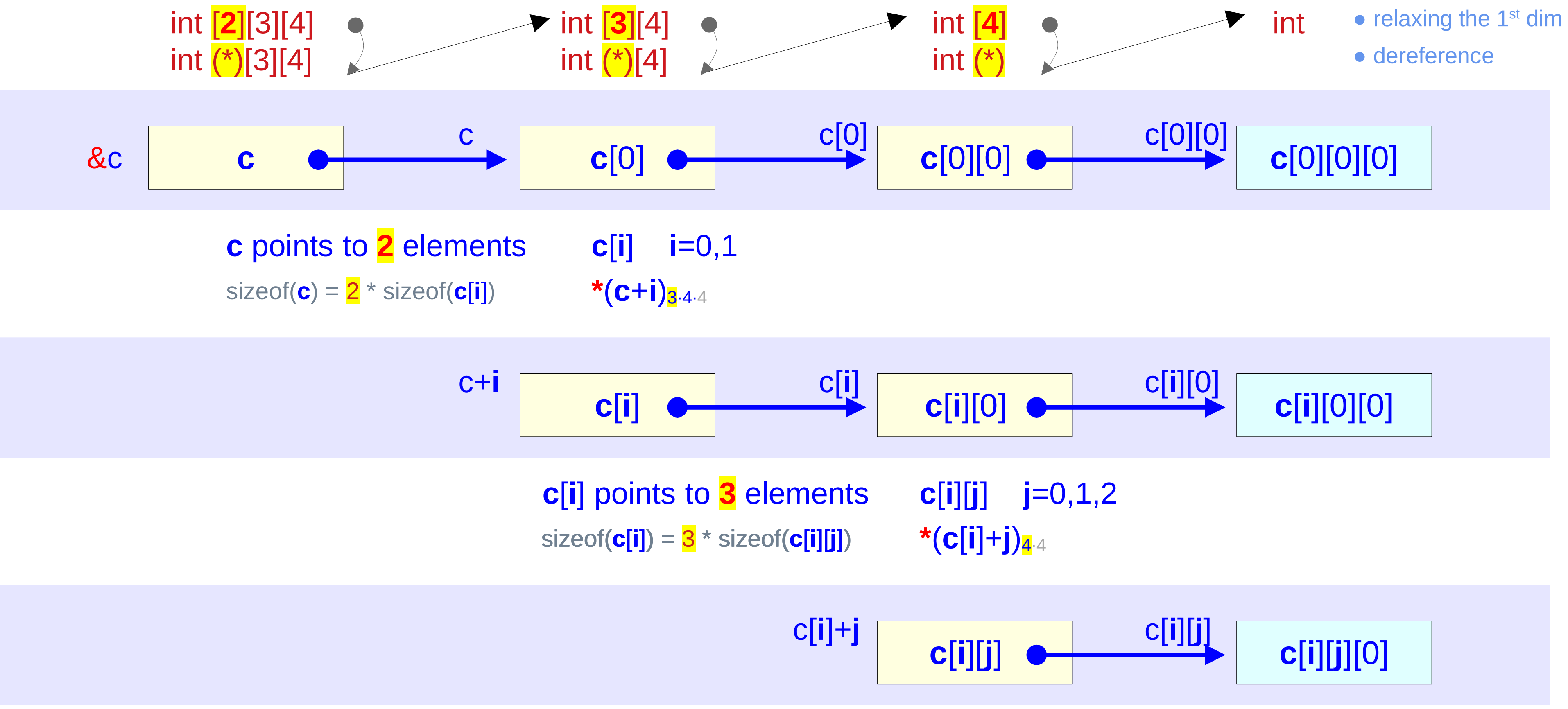
within an array **c**[0][0] of int [4] type, **c**[0][0] can be relaxed to a pointer of int (*) type

$c[0][0][0] = *(c[0][0]+0)_{1\cdot 4}$

Math Expression

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

Pointer Chain Type II



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

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

int [4]
int (*)

int

- relaxing the 1st dim
- dereference

&c → **c** → c → **c**[0] → c[0] → **c**[0][0] → c[0][0] → **c**[0][0][0]

c points to **2** elements **c**[**i**] **i**=0,1

sizeof(**c**) = 2 * sizeof(**c[i]**) ***(c+i)**$_{3\cdot 4\cdot 4}$

c+**i** → **c**[**i**] → c[**i**] → **c**[**i**][0] → c[**i**][0] → **c**[**i**][0][0]

c[**i**] points to **3** elements **c**[**i**][**j**] **j**=0,1,2

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

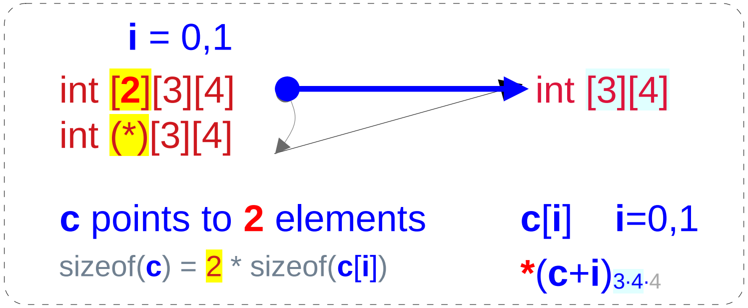
c[**i**]+**j** → **c**[**i**][**j**] → c[**i**][**j**] → **c**[**i**][**j**][0]

c[**i**][**j**] points to **4** elements **c**[**i**][**j**][**k**] **k**=0,1,2,3

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

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

- relaxing the 1st dim
- dereference



i = 0,1

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

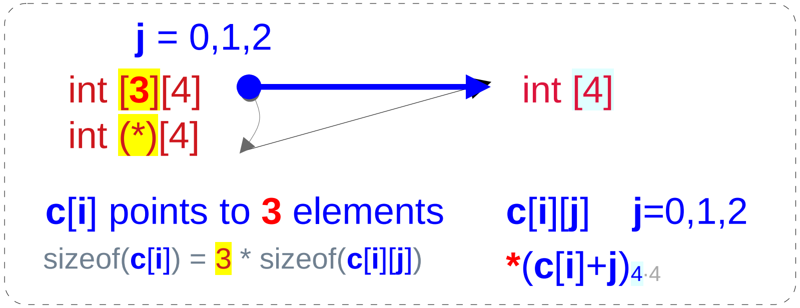
int [3][4]

c points to **2** elements

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

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

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



j = 0,1,2

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

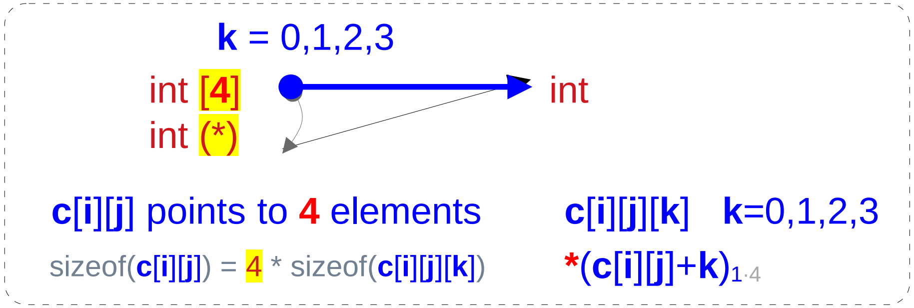
int [4]

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

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

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

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



k = 0,1,2,3

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

int

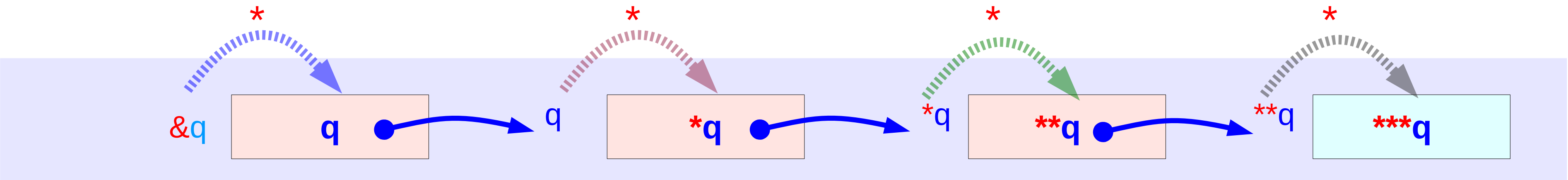
c[**i**][**j**] points to **4** elements

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

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

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

Pointer Chain Type I – * and & operators



*(&q) ≡ q

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

*(q) ≡ *q

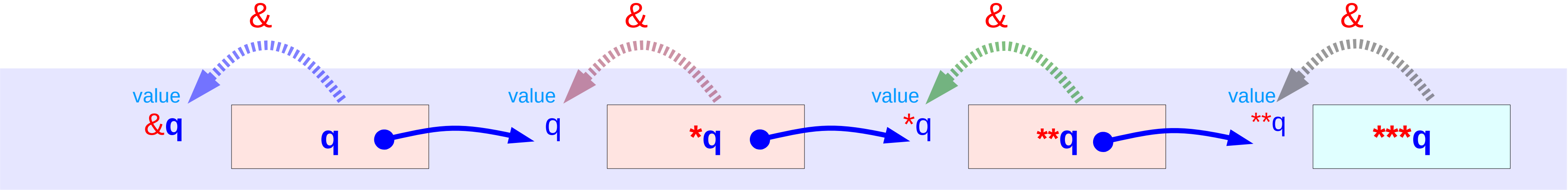
C expression *(q) equals to the variable q

*(*q) ≡ **q

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

*(**q) ≡ ***q

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



&q ≡ value(&q)

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

&(*q) ≡ value(q)

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

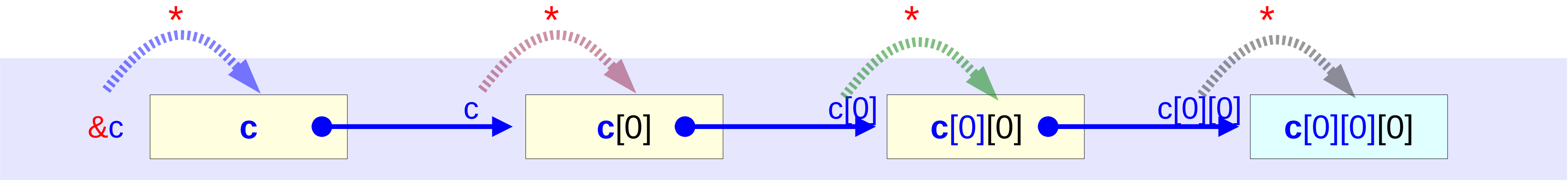
&(**q) ≡ value(*q)

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

&(***q) ≡ value(**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]

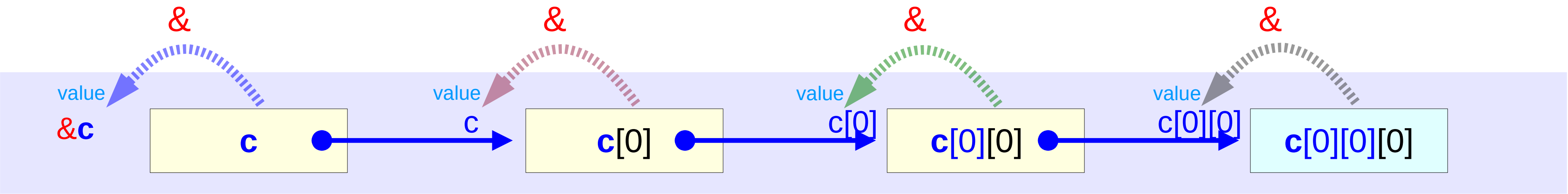
*(c[0]) ≡ c[0][0]

*(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]

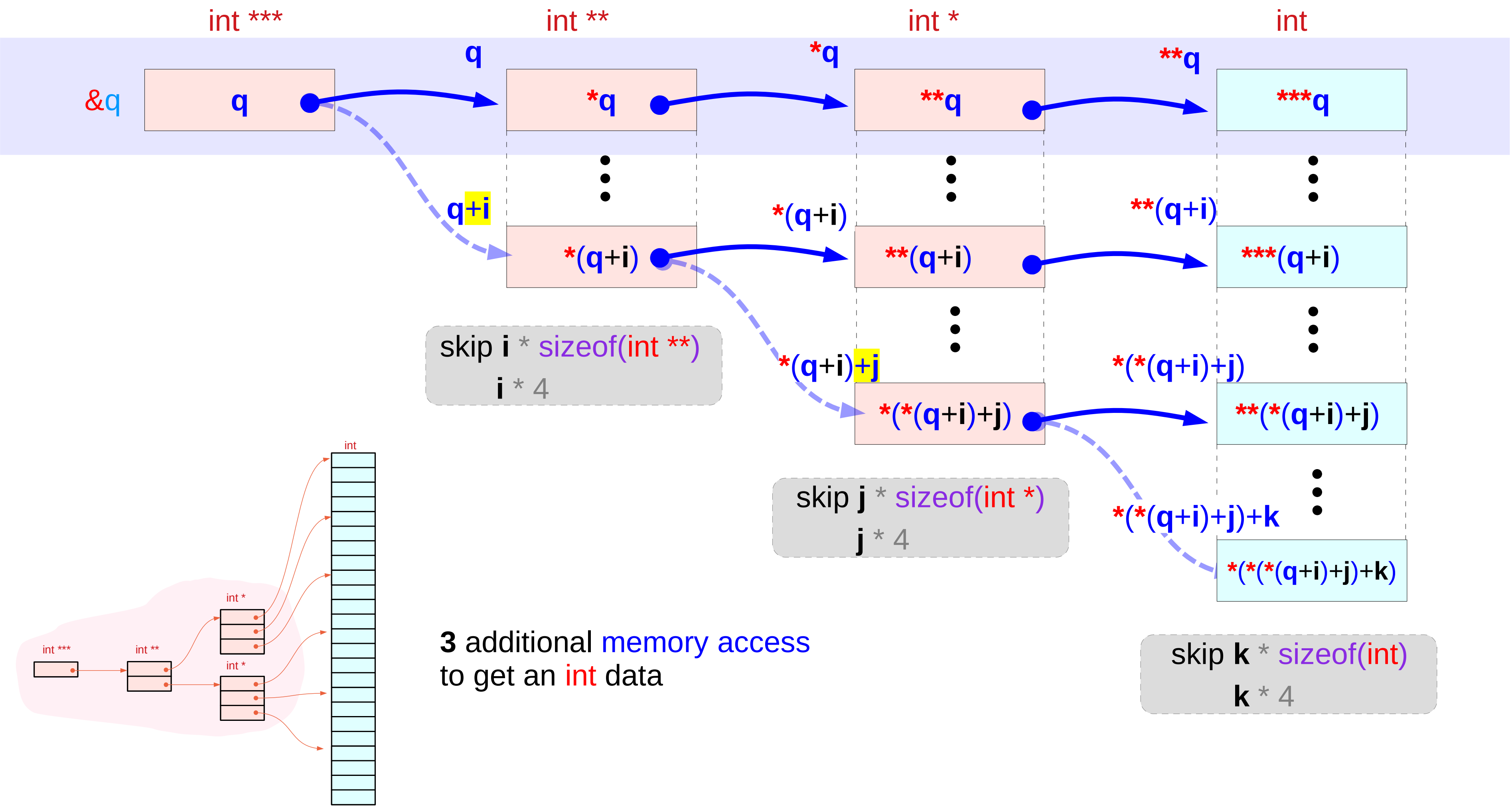
(int (*)) **c**[0][0] has the address value of (int) **c**[0][0][0]

Pointer Chain Type I – skipping elements

Pointer Chain Type I

multiple indirection

size: pointer size



skip **i** * sizeof(int **)
i * 4

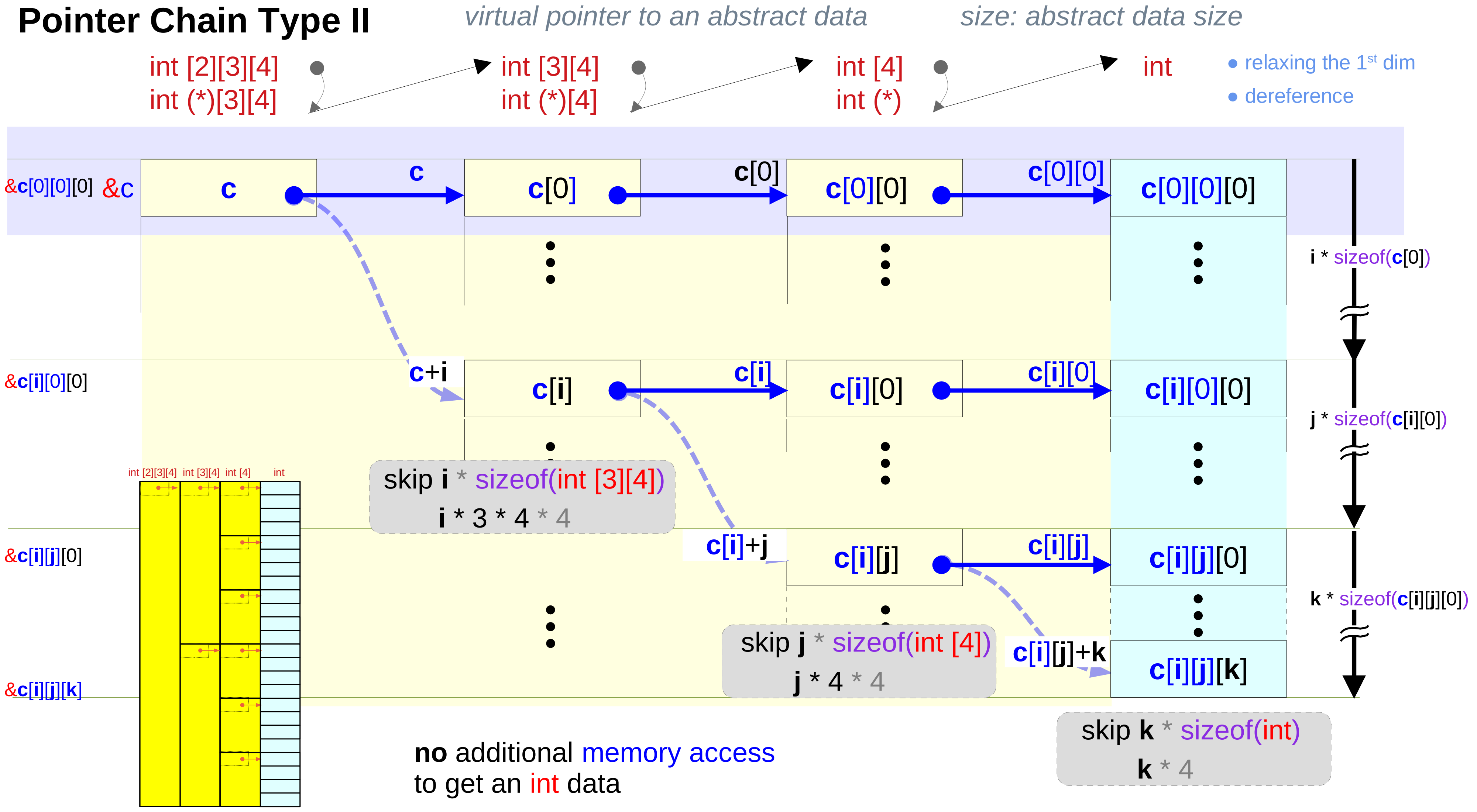
skip **j** * sizeof(int *)
j * 4

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

3 additional memory access
to get an int data

Pointer Chain Type II – skipping elements

Pointer Chain Type II



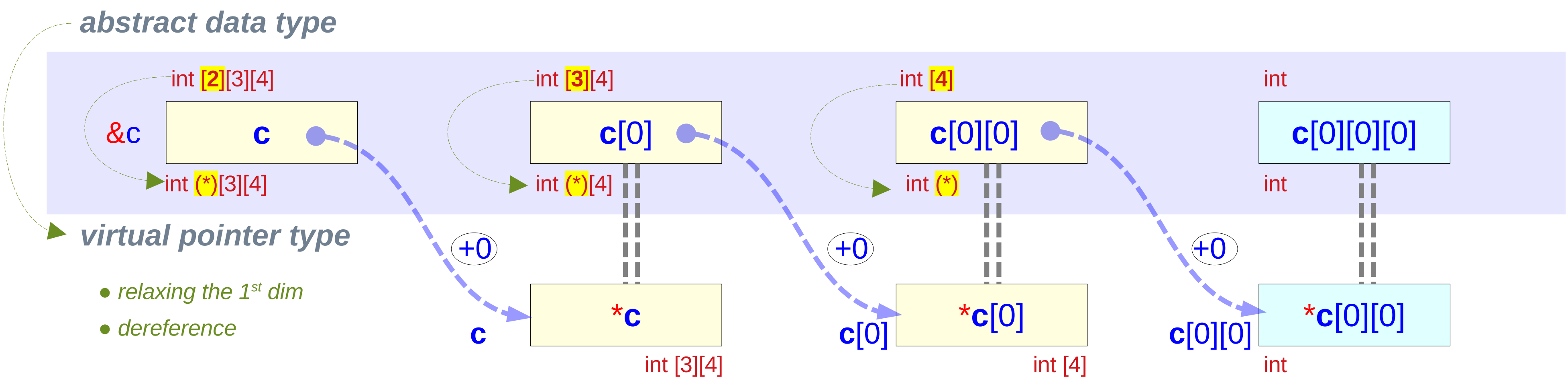
skip **i** * sizeof(int [3][4])
i * 3 * 4 * 4

skip **j** * sizeof(int [4])
j * 4 * 4

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

no additional memory access
to get an int data

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



abstract data type

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

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

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

int
c[0][0][0]
int

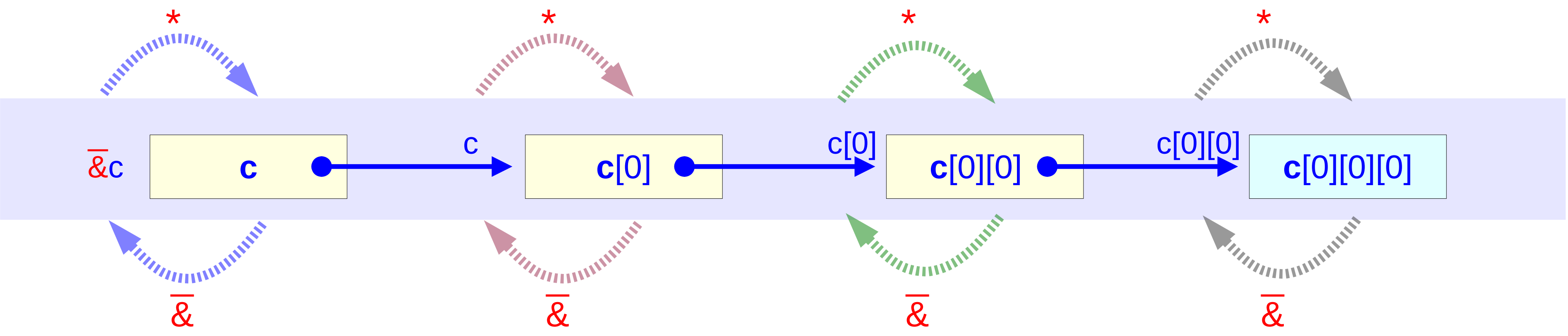
virtual pointer type

- *relaxing the 1st dim*
- *dereference*

+0
c
*$\mathbf{c}$
int [3][4]

+0
c[0]
*$\mathbf{c}$[0]
int [4]

+0
c[0][0]
*$\mathbf{c}$[0][0]
int



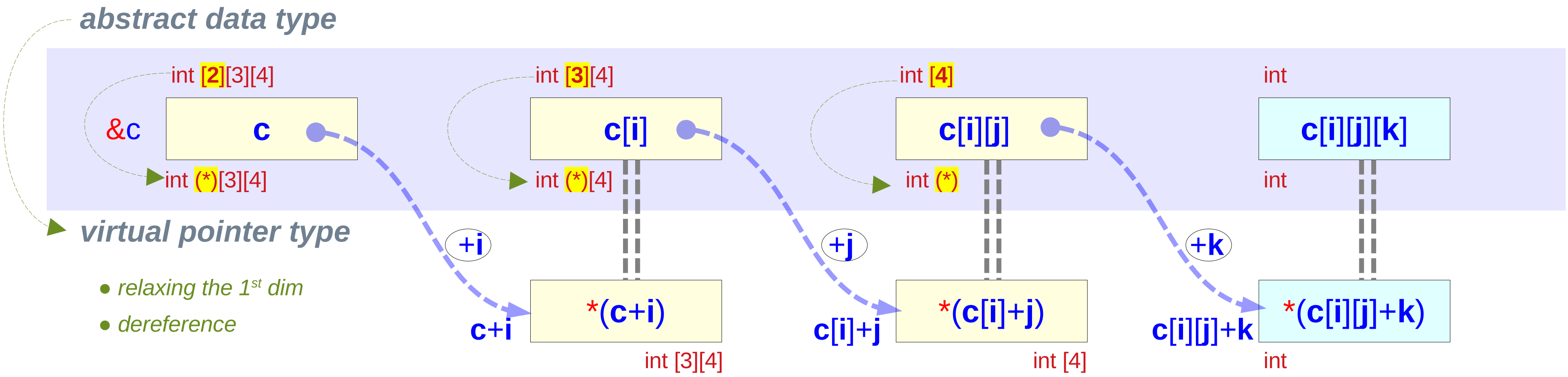
*　*　*　*

$\overline{\&}$c
c
c
c[0]
c[0]
c[0][0]
c[0][0]
c[0][0][0]

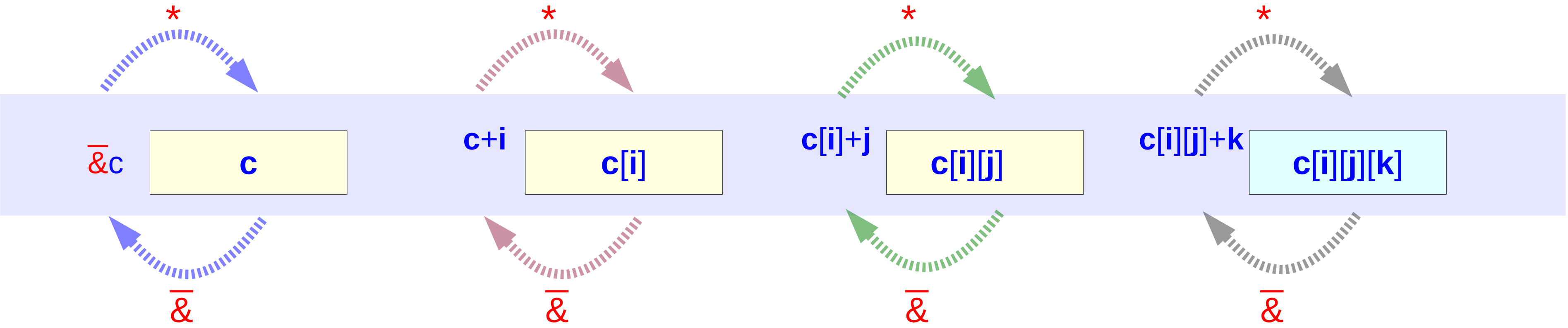
$\overline{\&}$　$\overline{\&}$　$\overline{\&}$　$\overline{\&}$

$\overline{\&}$: **mathematical & operator**

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

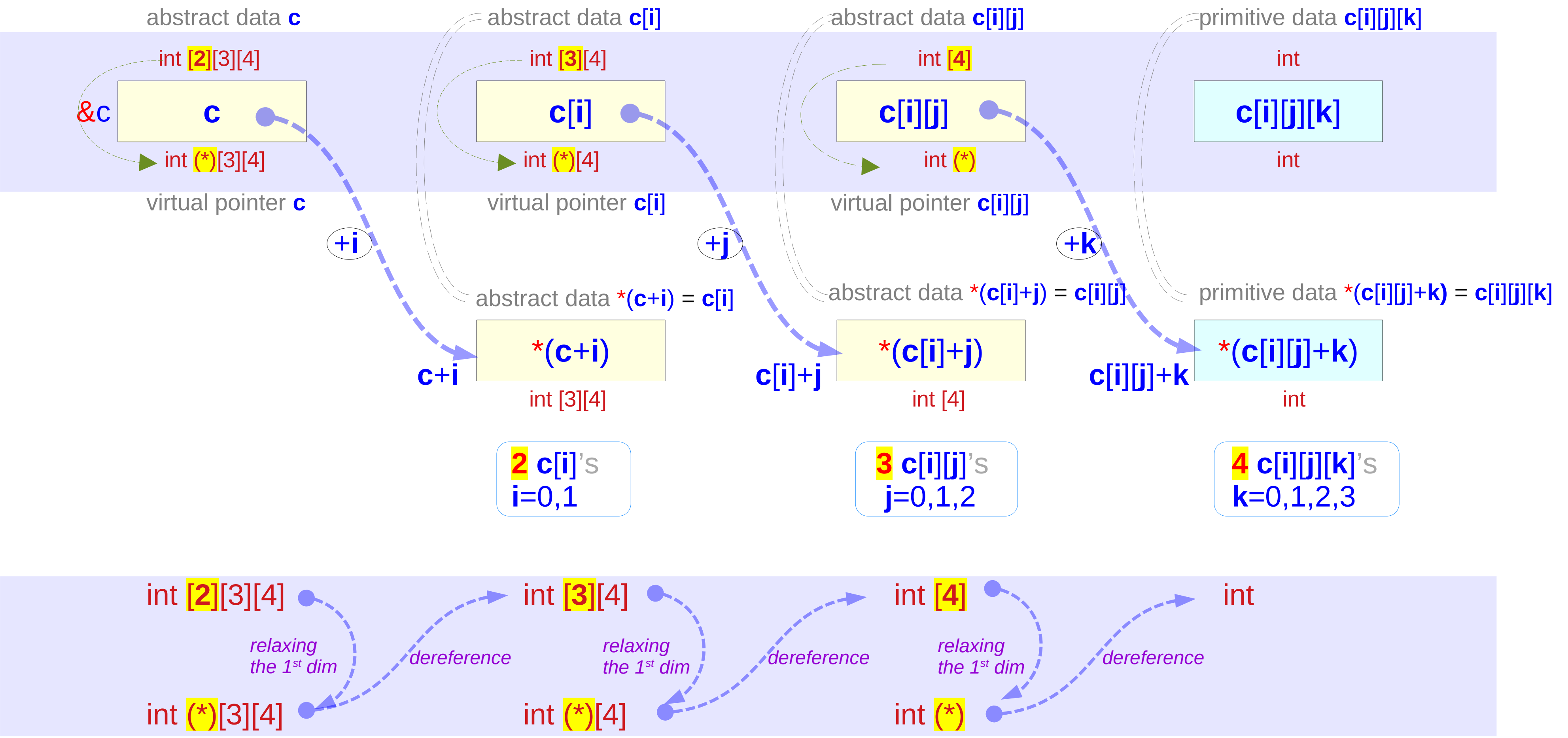


- *relaxing the 1st dim*
- *dereference*



$\overline{\&}$: **mathematical & operator**

Two step deferencing in type **II** (3)



abstract data **c**
int [2][3][4]
&c **c**
int (*)[3][4]
virtual pointer **c**

abstract data **c[i]**
int [3][4]
c[i]
int (*)[4]
virtual pointer **c[i]**

abstract data **c[i][j]**
int [4]
c[i][j]
int (*)
virtual pointer **c[i][j]**

primitive data **c[i][j][k]**
int
c[i][j][k]
int

+i
+j
+k

abstract data *(**c**+**i**) = **c**[**i**]
c+**i** *(**c**+**i**)
int [3][4]

abstract data *(**c[i]**+**j**) = **c[i][j]**
c[i]+**j** *(**c[i]**+**j**)
int [4]

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

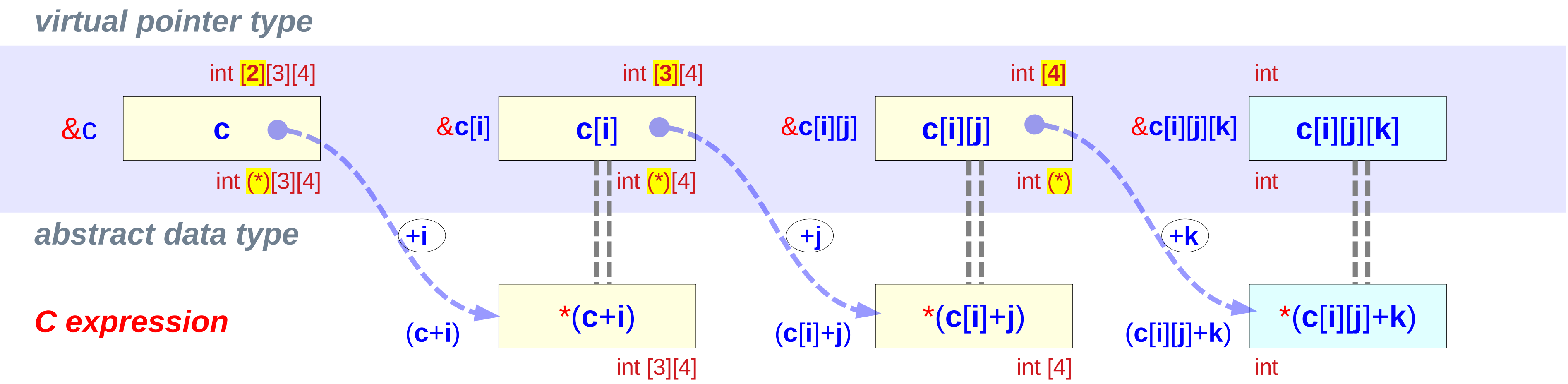
2 **c[i]**'s
i=0,1

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

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

int [2][3][4]
relaxing the 1st dim
int (*)[3][4]
dereference
int [3][4]
relaxing the 1st dim
int (*)[4]
dereference
int [4]
relaxing the 1st dim
int (*)
dereference
int

Skipping elements



virtual pointer type

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

int [3][4]
&c[i]
c[i]
int (*)[4]

int [4]
&c[i][j]
c[i][j]
int (*)

int
&c[i][j][k]
c[i][j][k]
int

abstract data type

+i
+j
+k

C expression

(c+i)
*(c+i)
int [3][4]

(c[i]+j)
*(c[i]+j)
int [4]

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

Math expression

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

skip $i\cdot3\cdot4$ integer elements from **c**

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

skip $j\cdot4$ integer elements from **c[i]**

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

skip $k\cdot1$ integer elements from **c[i][j]**

sizeof(*c) = 3*4*4

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

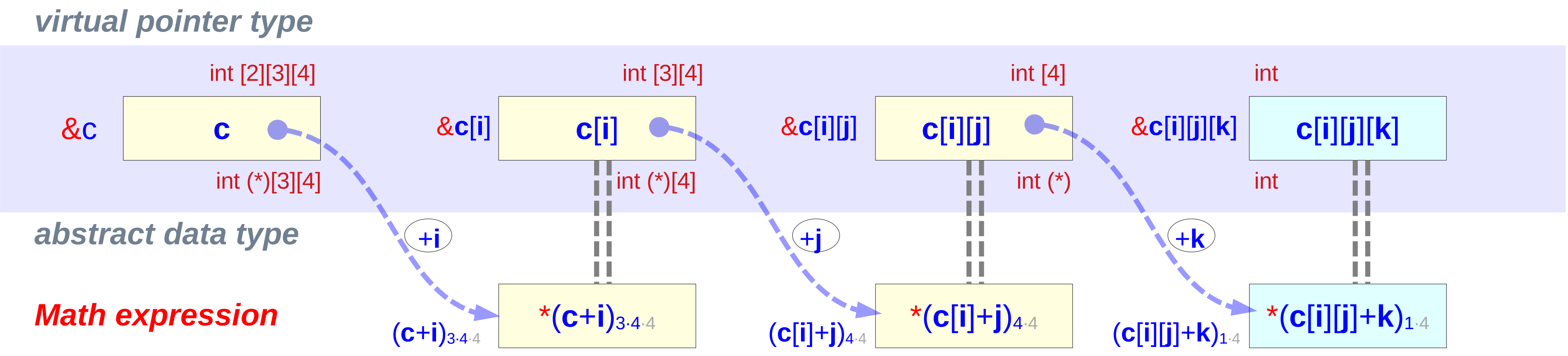
sizeof(*c[i]) = 4*4

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



virtual pointer type

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

int [3][4]
&c[i]
c[i]
int (*)[4]

int [4]
&c[i][j]
c[i][j]
int (*)

int
&c[i][j][k]
c[i][j][k]
int

abstract data type

+i
+j
+k

Math expression

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

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

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

equivalence relations – c expressions

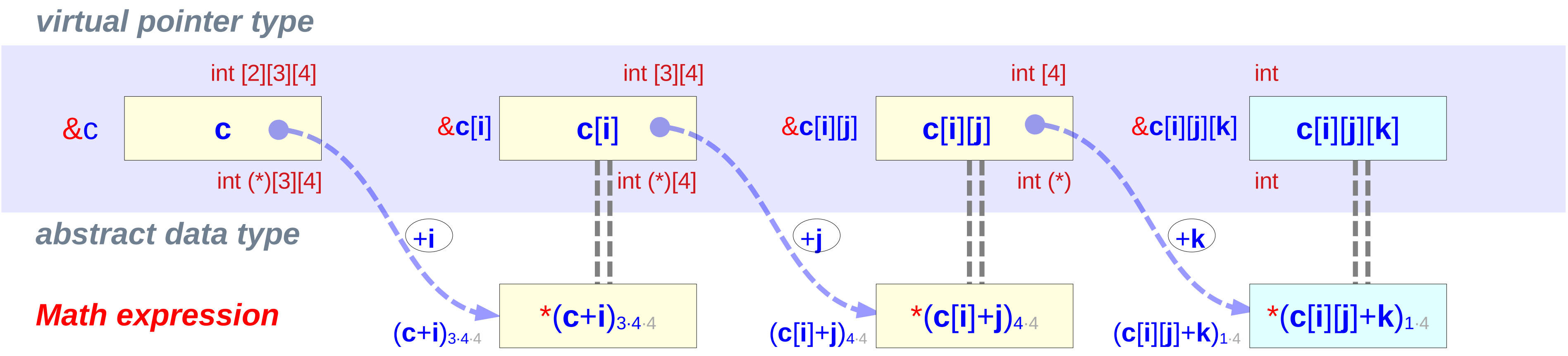
c[i][j][k] = *(c[i][j] + k)
c[i][j] = *(c[i] + j)
c[i] = *(c + i)

&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($(c[i] + j)_{4\cdot4}$)
value(c[i]) = value($(c + i)_{3\cdot4\cdot4}$)

Applying address replication



virtual pointer type

abstract data type

Math expression

int [2][3][4]	int [3][4]	int [4]	int
&c **c**	&c[i] **c[i]**	&c[i][j] **c[i][j]**	&c[i][j][k] **c[i][j][k]**
int (*)[3][4]	int (*)[4]	int (*)	int
+i	+j	+k	
$(c+i)_{3\cdot 4\cdot 4}$	$*(c+i)_{3\cdot 4\cdot 4}$	$*(c[i]+j)_{4\cdot 4}$	$*(c[i][j]+k)_{1\cdot 4}$
	$(c[i]+j)_{4\cdot 4}$	$(c[i][j]+k)_{1\cdot 4}$	

address replication   *address replication*

element size
= sizeof(int)
= 4 bytes

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

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

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

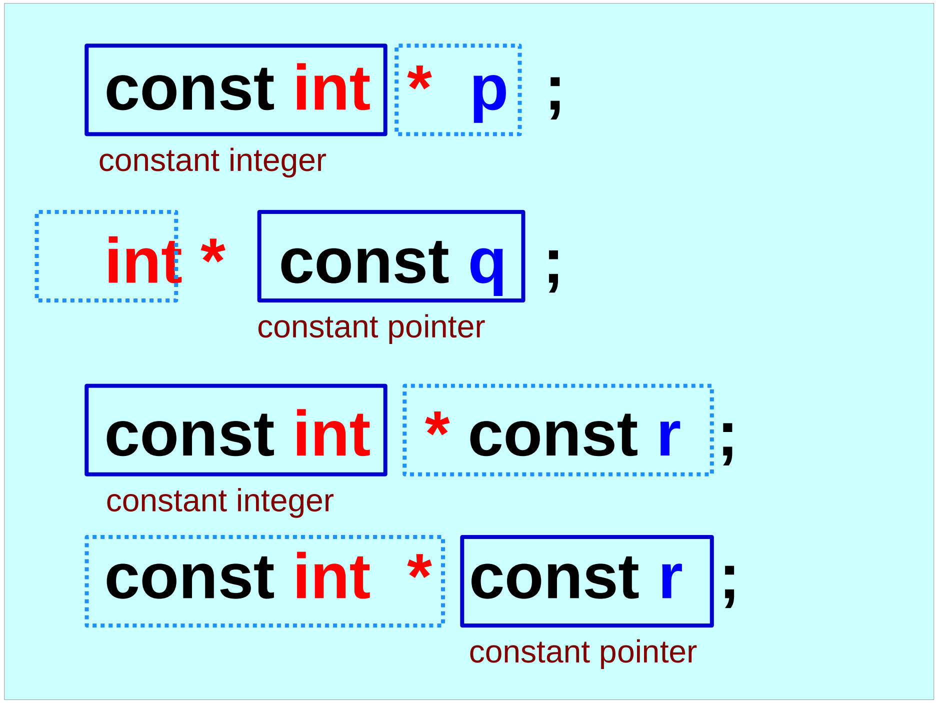
const pointers

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 ;`

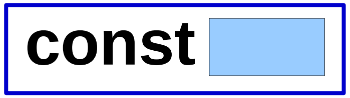
constant pointer

***p** : constant integer value

q : constant (int *) pointer

***r** : constant integer value

r : constant (int *) pointer



const

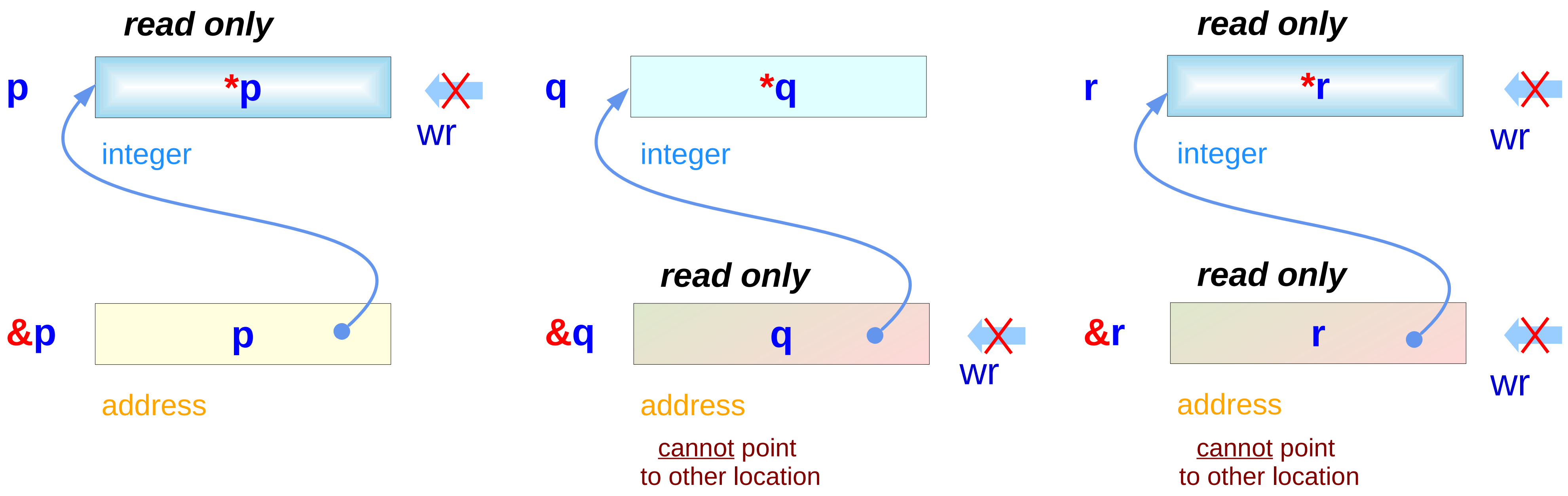
group with the following

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;
```

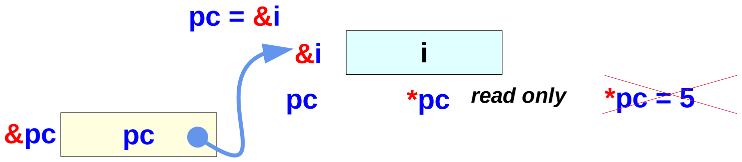
```
 pc = &i;     // (const int *) ← (int *)
*pc  = 5;     // (const int) error
```

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

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

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

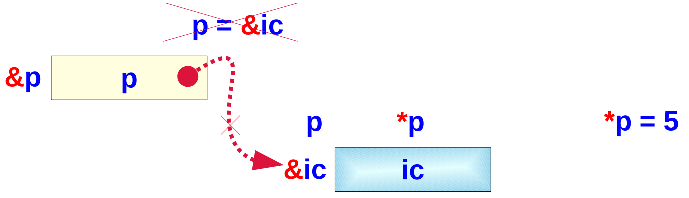
C A Reference Manual, Harbison & Steele Jr.



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



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

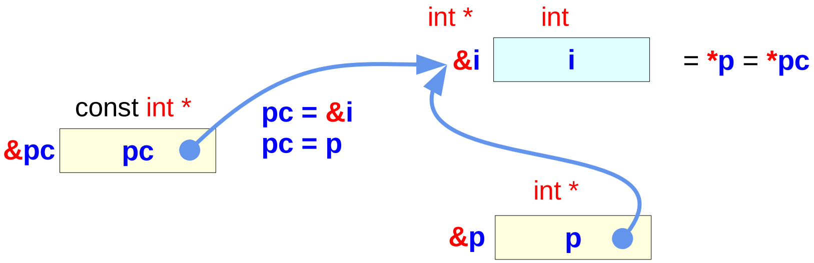
therefore, **p** should not point to **const ic**

const examples (2)

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

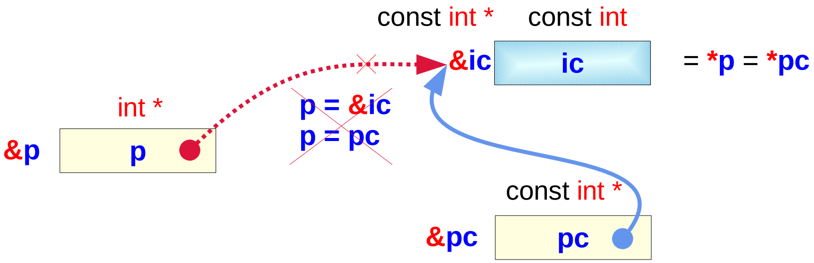
```
p    = &i;
pc   = &ic
```

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



int *
int
&i
i
= *p = *pc
const int *
&pc
pc
pc = &i
pc = p
int *
&p
p

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

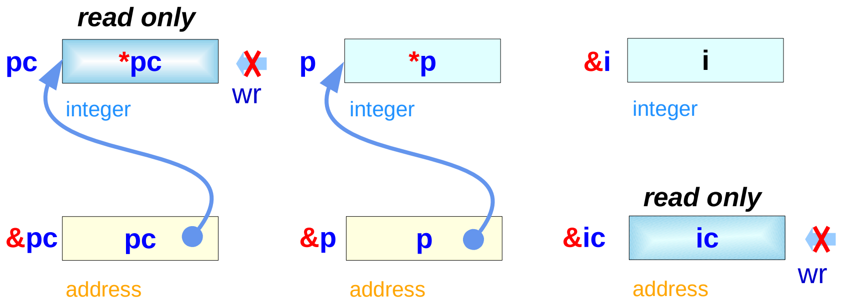


const int *
const int
&ic
ic
= *p = *pc
int *
&p
p
~~p = &ic~~
~~p = pc~~
const int *
&pc
pc

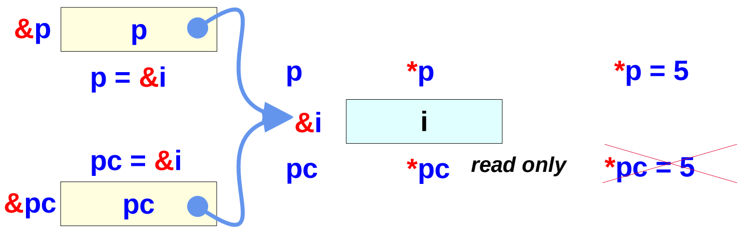
C A Reference Manual, Harbison & Steele Jr.

const examples (3)

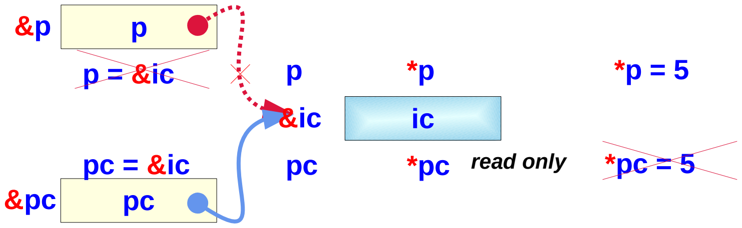
```
const int * pc;
      int * p, i;
const int   ic;
```



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



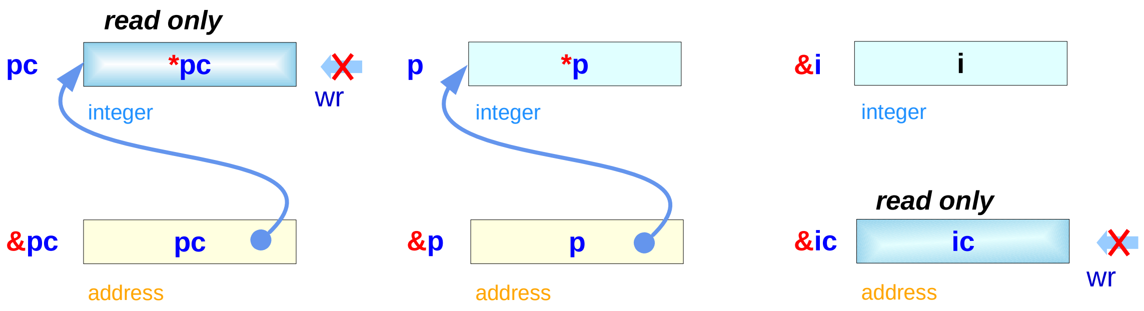
```
 p   = &ic;     // (int *) ← (const int *) warning
*p   = 5;       // (int)
 pc  = &ic;     // (const int *) ← (const int *)
*pc  = 5;       // (const int) error
```



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