Applications of Array Pointers (1A)

Young Won Lim
7/3/20

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

Multi-dimensional Array Pointers

(*n-1*)-d array pointer to a *n*-d array

```
int a[4] ;
int (*p) ;
```

1-d array
0-d array pointer (p = a)

```
int b[4]  [2];
int (*q)  [2];
```

2-d array
1-d array pointer (q = b)

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

3-d array
2-d array pointer (r = c)

```
int d[4]  [2][3][4];
int (*s)  [2][3][4];
```

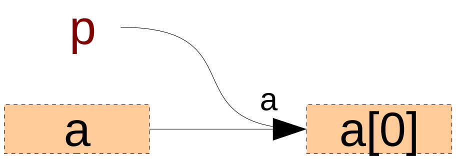
4-d array
3-d array pointer (s = d)

↑ the 1st dimension can be accessed by incrementing (n-1)-d array pointer

***n*-d array name and (*n-1*)-d array pointer**

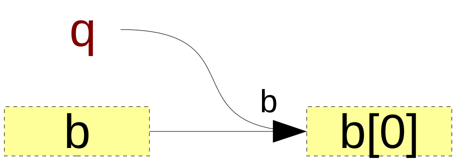
```
int a[4] ;
int (*p) ;
```

```
p = &a[0];
p = a;
```



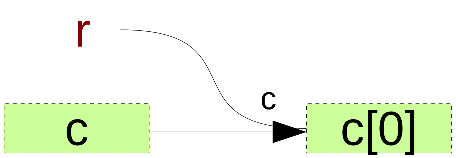

```
int b[4] [2];
int (*q) [2];
```

```
q = &b[0];
q = b;
```



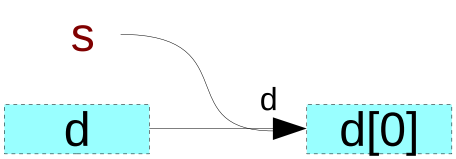

```
int c[4] [2][3];
int (*r) [2][3];
```

```
r = &c[0];
r = c;
```




```
int d[4] [2][3][4];
int (*s) [2][3][4];
```

```
s = &d[0];
s = d;
```



the 1st dimension can be accessed by incrementing (n-1)-d array pointer

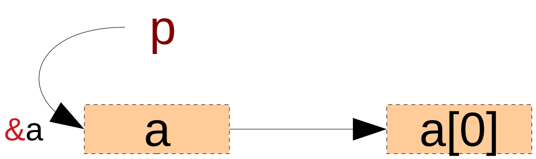
***n*-d** array pointer to a ***n*-d** array

int a [4] ;	**1-d** array	
int (*p) [4];	**1-d** array pointer	(p = &a)
int b [4][2];	**2-d** array	
int (*q) [4][2];	**2-d** array pointer	(q = &b)
int c [4][2][3];	**3-d** array	
int (*r) [4][2][3];	**3-d** array pointer	(r = &c)
int d [4][2][3][4];	**4-d** array	
int (*s) [4][2][3][4];	**4-d** array pointer	(s = &d)

***n*-d** array name and ***n*-d** array pointer

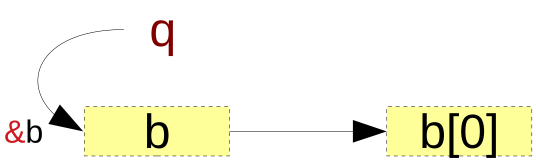
```
int  a   [4] ;
int (*p) [4];
```

p = &a;



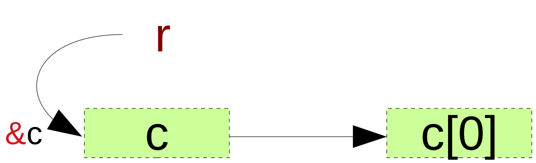

```
int  b   [4][2];
int (*q) [4][2];
```

q = &b;



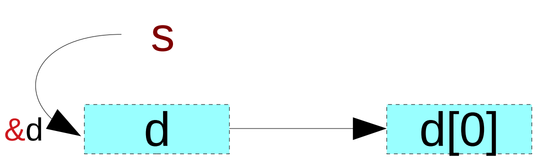

```
int  c   [4][2][3];
int (*r) [4][2][3];
```

r = &c;

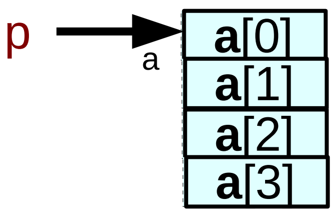


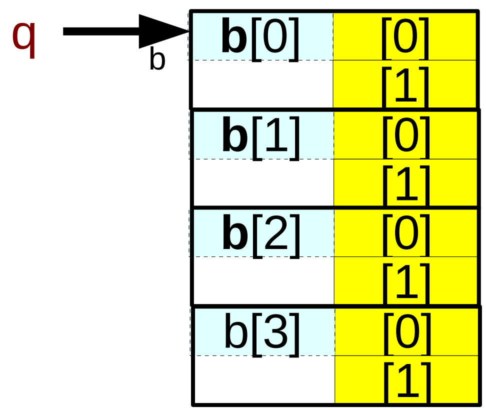

```
int  d   [4][2][3][4];
int (*s) [4][2][3][4];
```

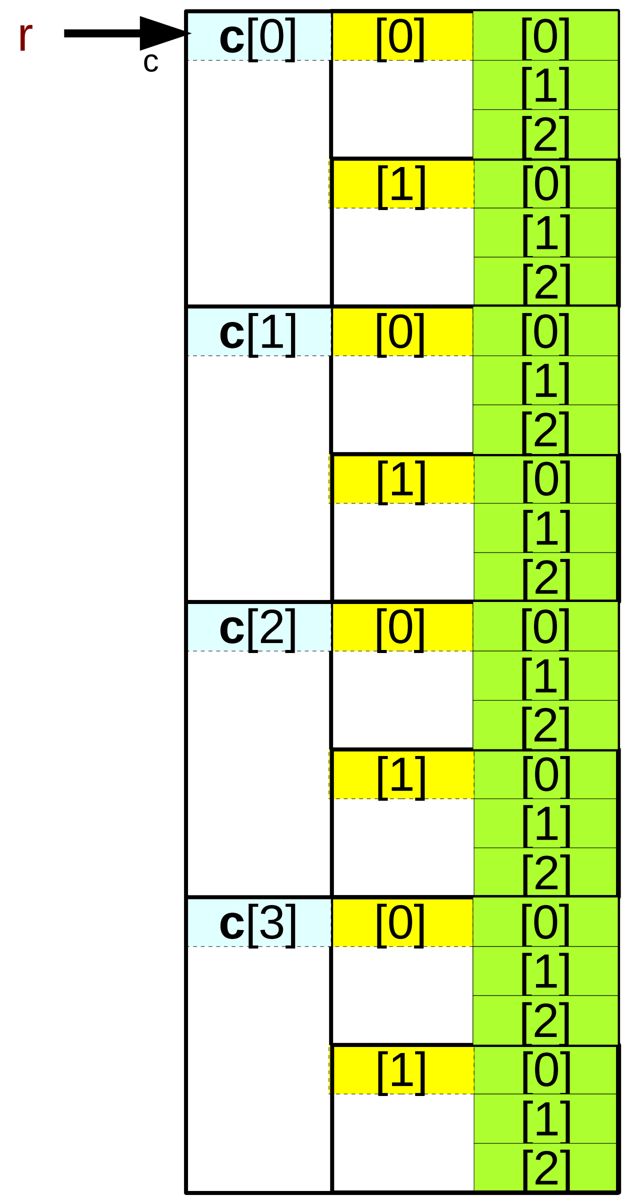
s = &d;



multi-dimensional array pointers





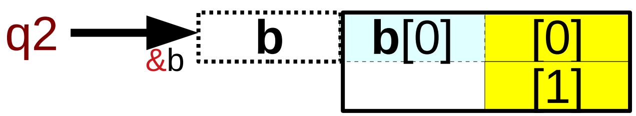


int a[4] ;		**1-d** array
int (*p) ;		**0-d** array pointer
int b[4]	[2];	**2-d** array
int (*q)	[2];	**1-d** array pointer
int c[4]	[2][3];	**3-d** array
int (*r)	[2][3];	**2-d** array pointer
int d[4]	[2][3][4];	**4-d** array
int (*s)	[2][3][4];	**3-d** array pointer

Initializing ***n-d*** array pointers

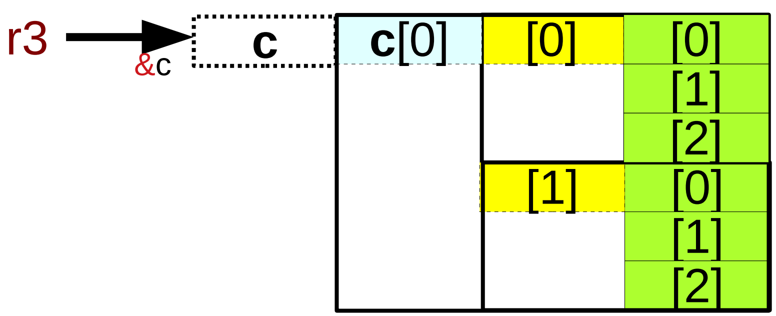
int a[4] ;

int (*p1)[4] = &a ;



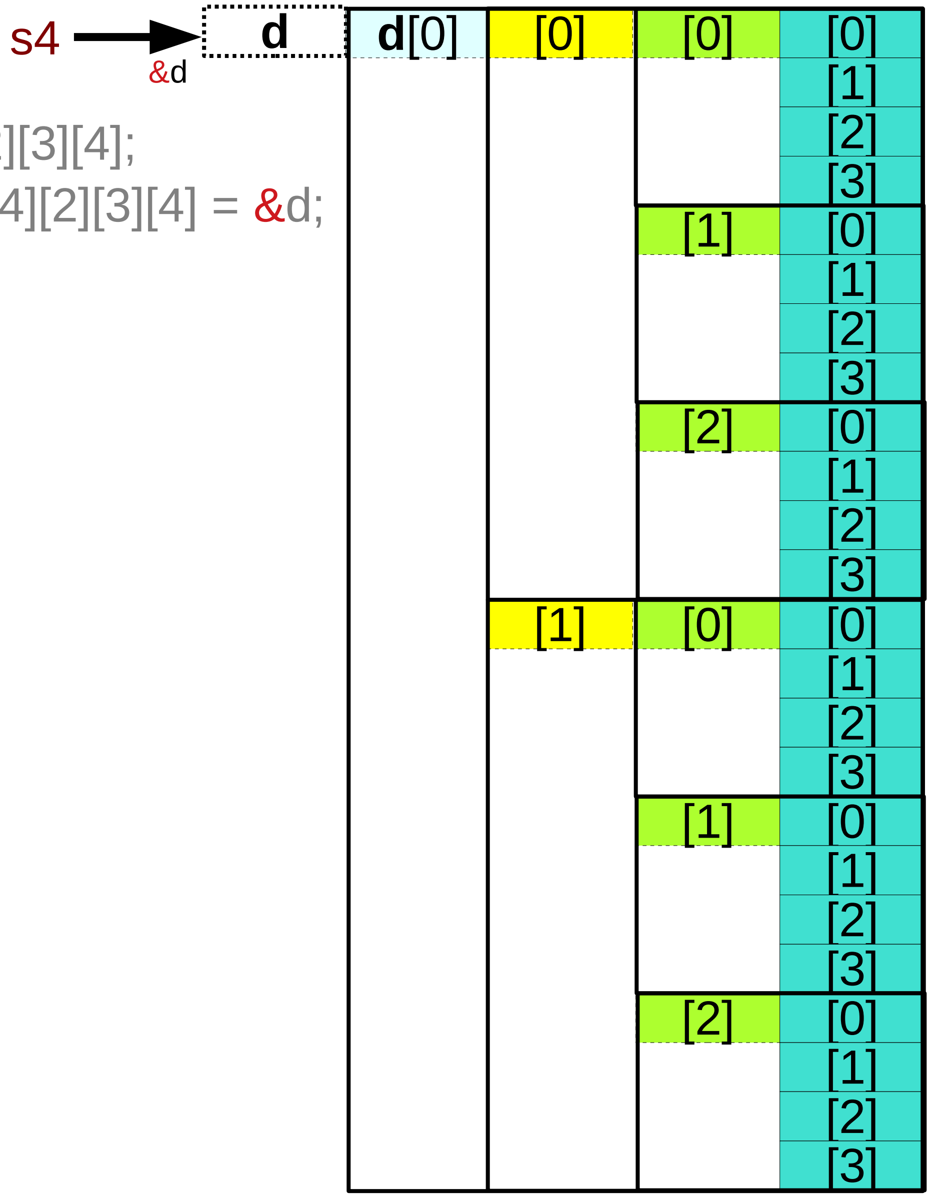
int b[4][2];

int (*q2)[4][2] = &b;



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

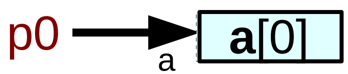
int (*r3)[4][2][3] = &c;



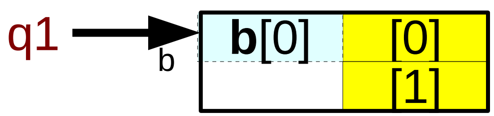
int d[4][2][3][4];

int (*s4)[4][2][3][4] = &d;

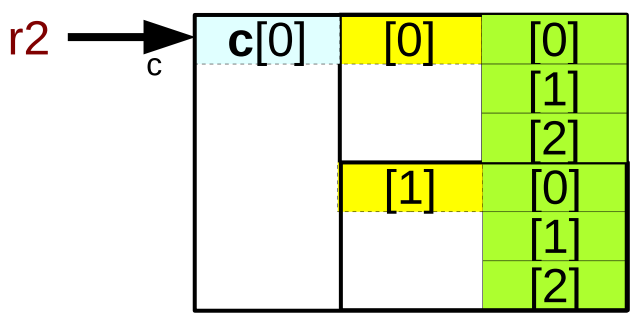
Initializing *(n-1)-d* array pointers




```
int a[4] ;
int (*p0) = a ;
```

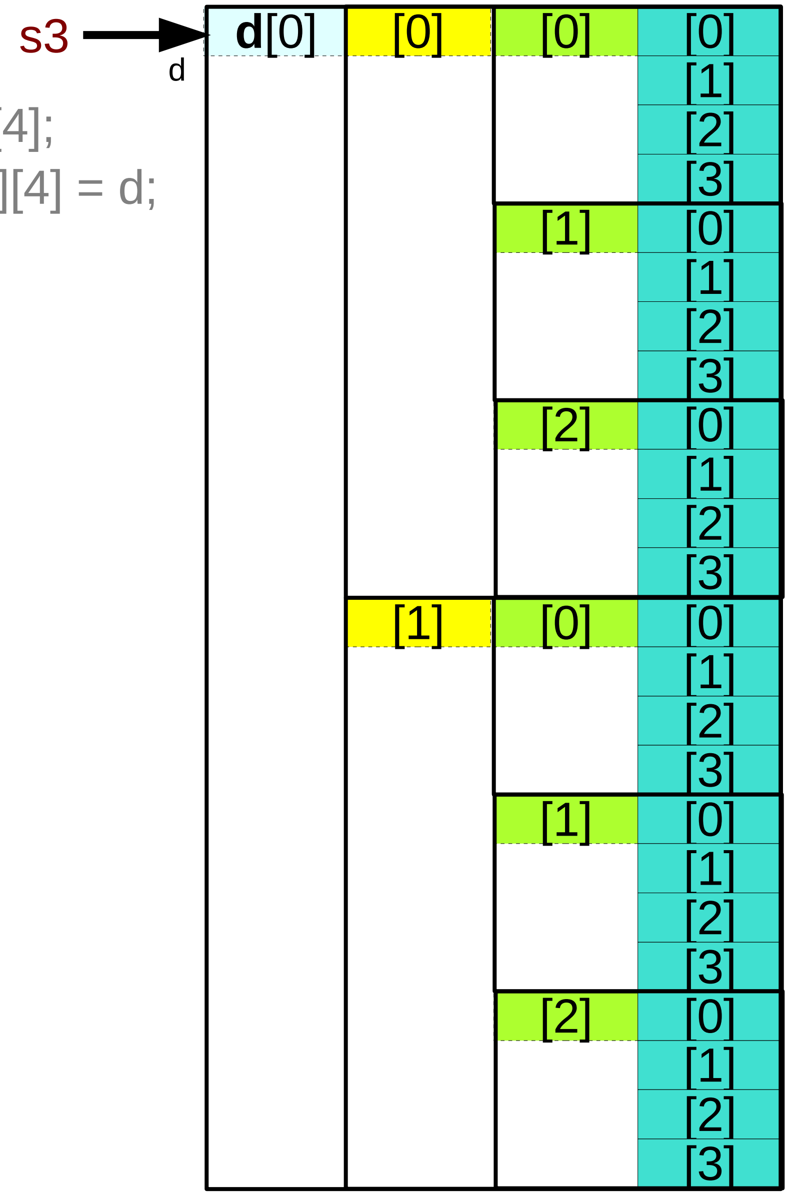



```
int b[4][2];
int (*q1)[2] = b;
```




```
int c[4][2][3];
int (*r2)[2][3] = c;
```

```
int d[4][2][3][4];
int (*s3)[2][3][4] = d;
```



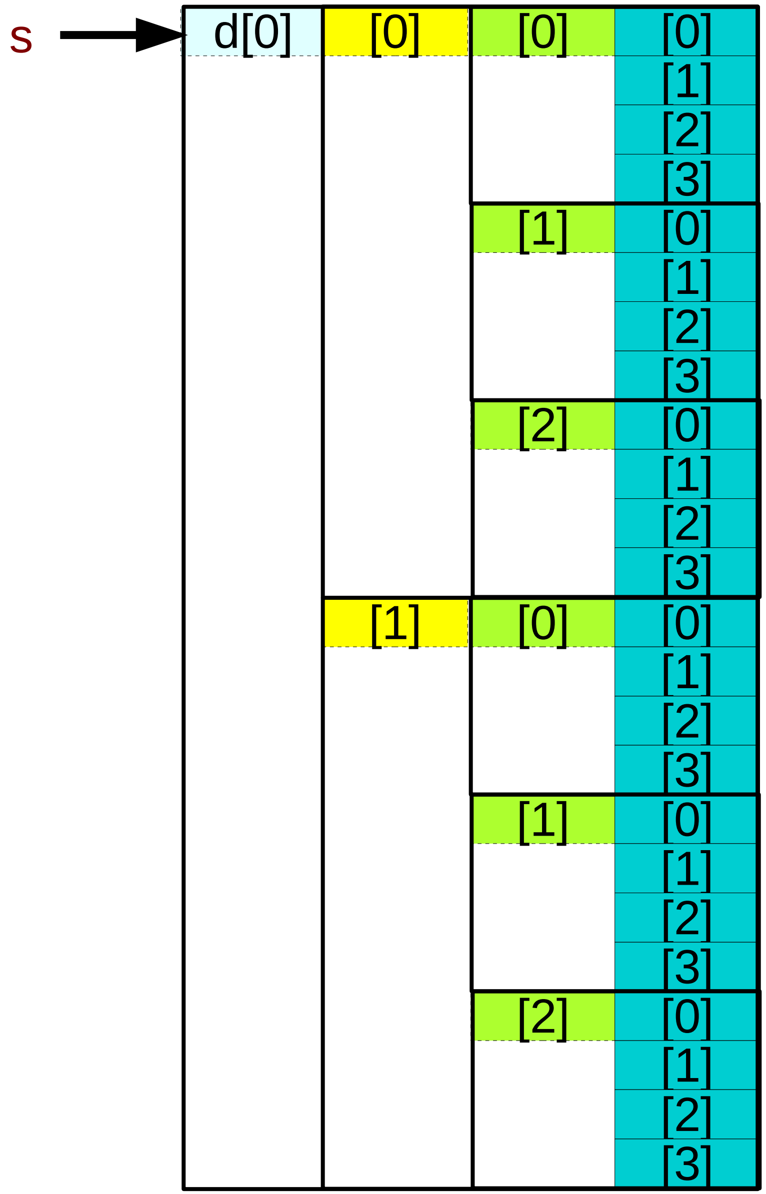
array pointers to multi-dimensional subarrays

int d[4] [2][3][4];
int (*s) [2][3][4];

d	4-d array name	d[4][2][3][4]
	3-d array pointer	(*p)[2][3][4]
d[i]	3-d array name	d[i][2][3][4]
	2-d array pointer	(*q)[3][4]
d[i][j]	2-d array name	d[i][j][3][4]
	1-d array pointer	(*r)[4]
d[i][j][k]	1-d array name	d[i][j][k][4]
	0-d array pointer	(*s)

i,j,k are specific index values

i =[0..3], j = [0..1], k= [0..2]



Initializing array pointers to multi-dimensional subarrays

```
int d[4] [2][3][4];
int (*s) [2][3][4];
```

d	4-d array name	d[4][2][3][4]	p[i][j][k][l]
	3-d array pointer	(*p)[2][3][4]	int (*p)[2][3][4] = d;
d[i]	3-d array name	d[i][2][3][4]	q[j][k][l]
	2-d array pointer	(*q)[3][4]	int (*q)[3][4] = d[i];
d[i][j]	2-d array name	d[i][j][3][4]	r[k][l]
	1-d array pointer	(*r)[4]	int (*r)[4] = d[i][j];
d[i][j][k]	1-d array name	d[i][j][k][4]	s[l]
	0-d array pointer	(*s)	int (*s) = d[i][j][k];

i =[0..3], j = [0..1], k= [0..2]

Passing multidimensional array names

```
int a[4] ;
int (*p) ;
```

call
```
funa(a, …);
```
prototype
```
void funa(int (*p), …);
```

```
int b[4] [2];
int (*q) [2];
```

call
```
funb(b, …);
```
prototype
```
void funb(int (*q)[2], …);
```

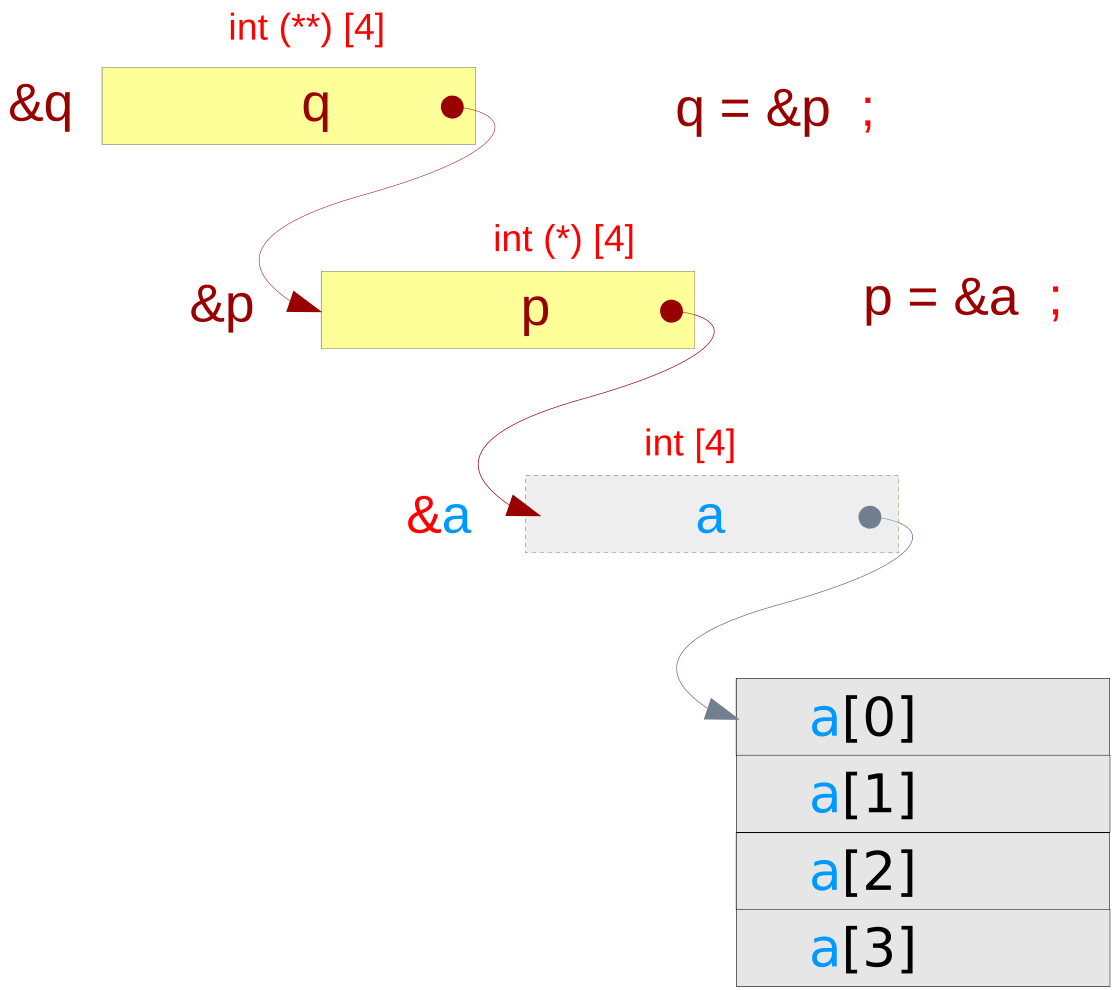
```
int c[4] [2][3];
int (*r) [2][3];
```

call
```
func(c, …);
```
prototype
```
void func(int (*r)[2][3], …);
```

```
int d[4] [2][3][4];
int (*s) [2][3][4];
```

call
```
fund(d, …);
```
prototype
```
void fund(int (*s)[2][3][4], …);
```

Double pointer to a **1-d** array – a variable view (p, q)

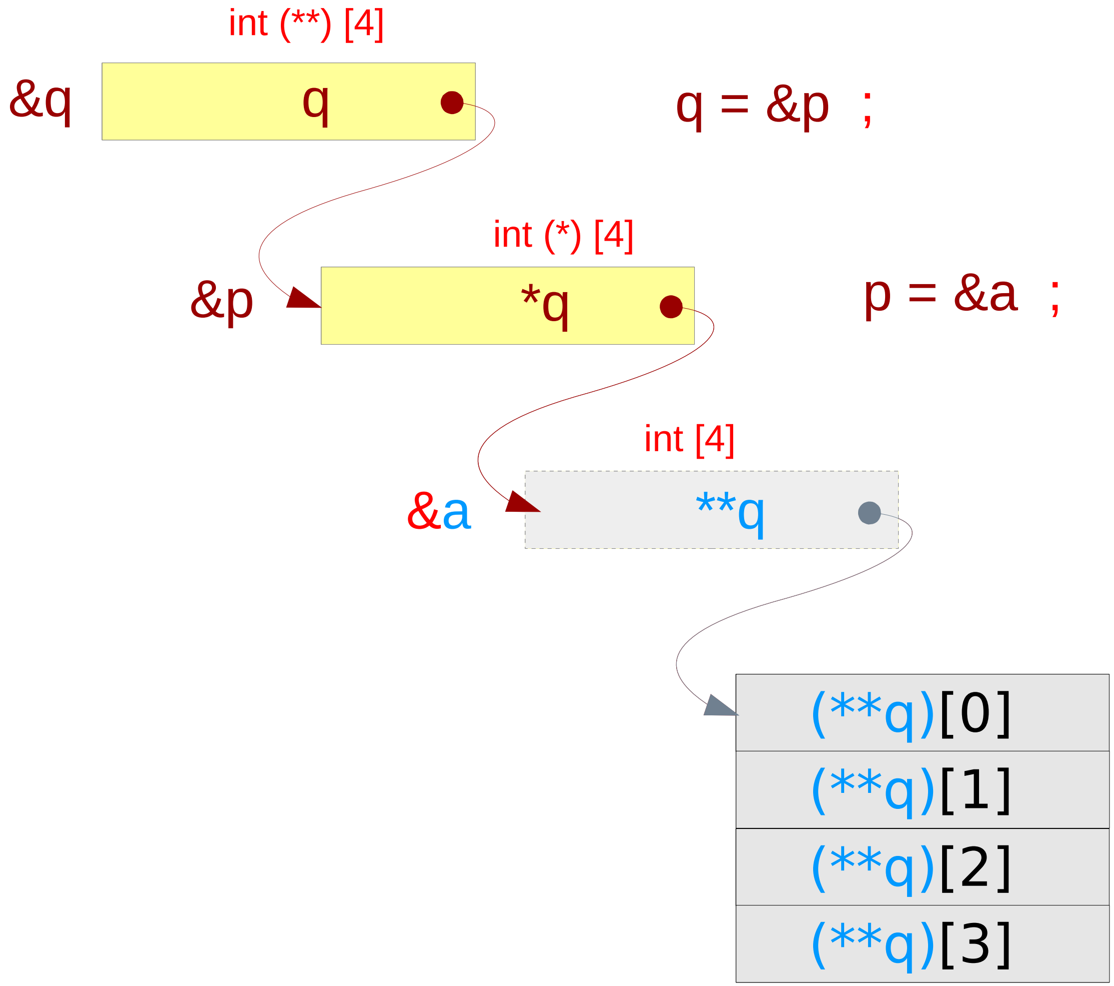



```
int a[4] ;
int (*p) [4] = &a ;
int (**q) [4] = &p ;
```

➡ p = &a ;

➡ q = &p ;

Double pointer to a **1-d** array – a variable view (q)



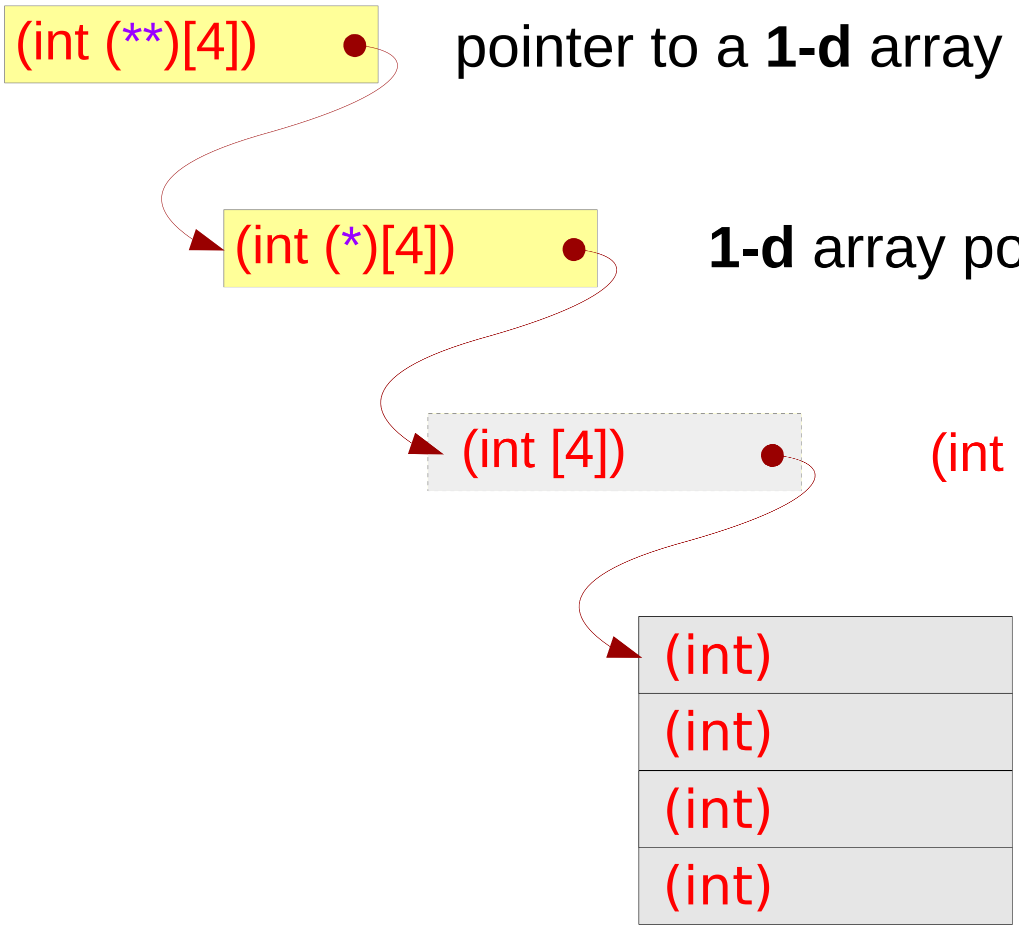
q = &p ;

p = &a ;

```
int  a[4]  ;
int  (*p) [4] = &a ;
int  (**q) [4] = &p ;
```

```
➡  p = &a ;
➡  q = &p ;
```

Double pointer to a **1-d** array – a type view



pointer to a **1-d** array pointer

1-d array pointer

(int *) a pointer to an int

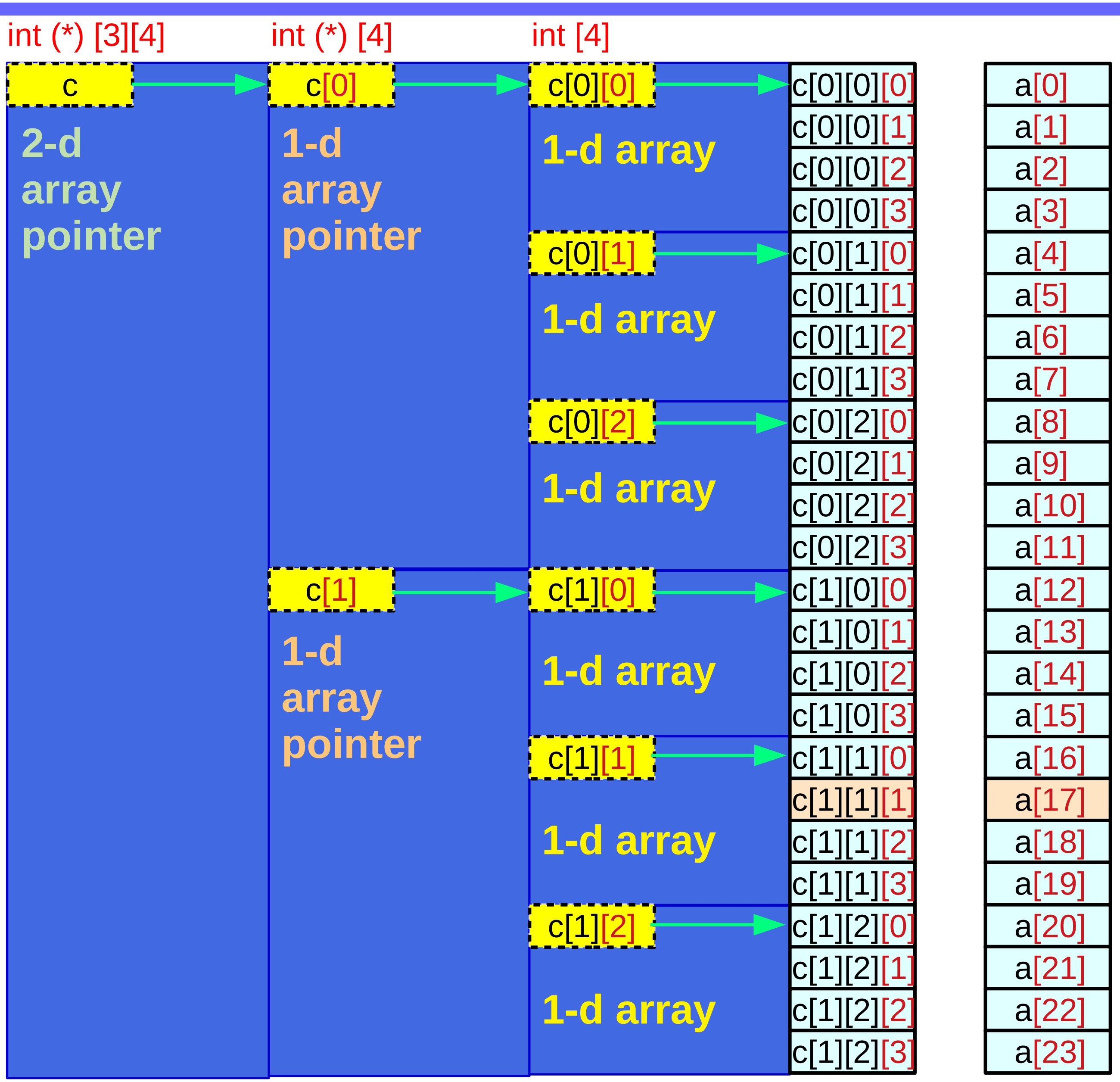
```
int  a[4]  ;
int  (*p) [4] = &a ;
int  (**q) [4] = &p ;
```

```
➡  p = &a ;
➡  q = &p ;
```

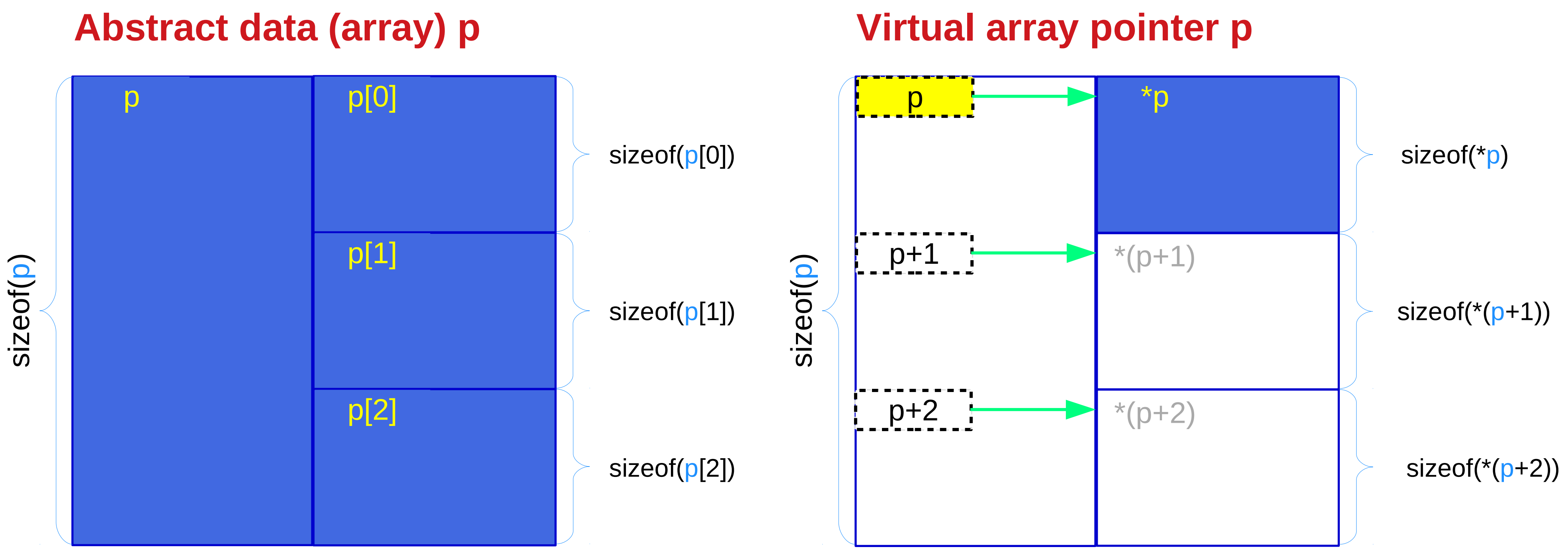
Virtual Array Pointers in Multi-dimensional Arrays

3-d array structure

- Hierarchical
- Nested Structure
- Virtual Array Pointers Over
- Contiguous
- Linear Layout



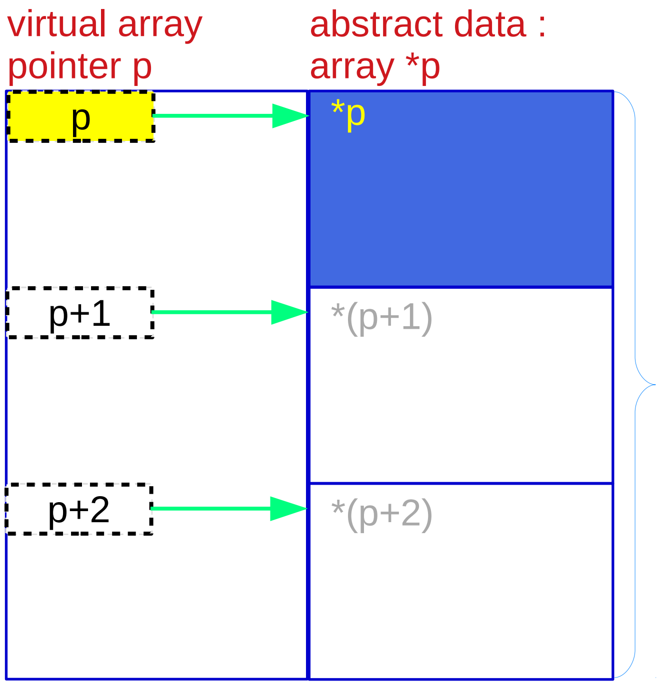
Array **p** and virtual array pointer **p**

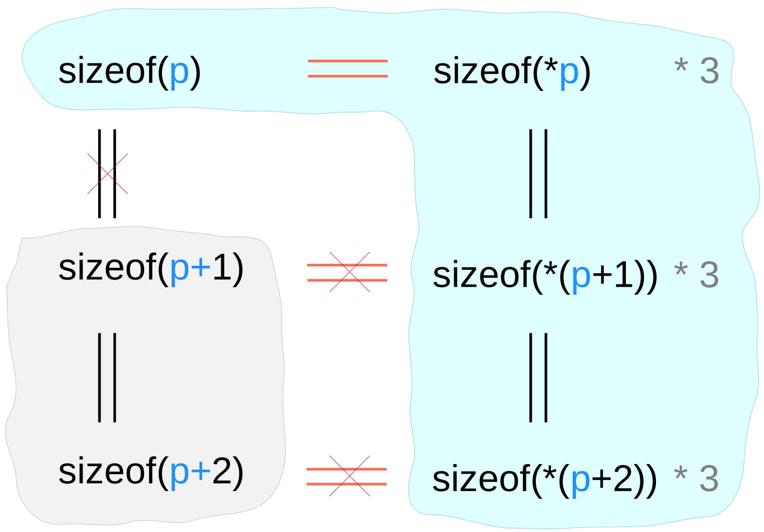


p is the name of an array and
has a array pointer type but
has a size of the array

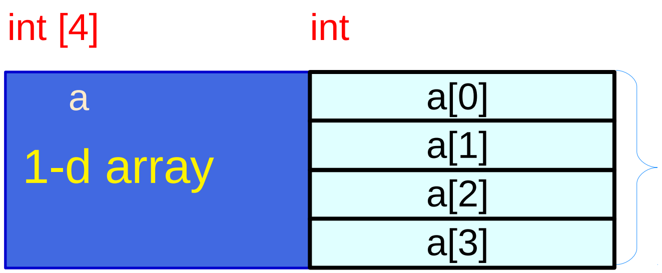
p is a virtual array pointer

Virtual array pointer to abstract data



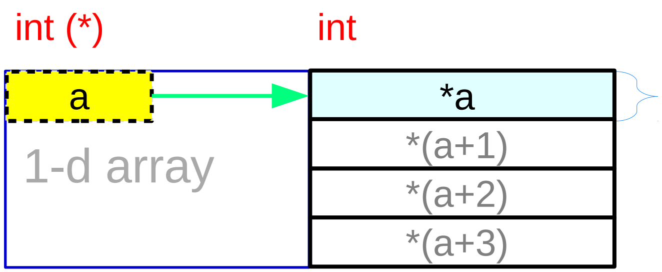


Array **a** and pointer **a**



1-d array a specific array type

sizeof(a)



pointer a general pointer type

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

a is the name of a 1-d array and
has a pointer type but
has a size of the array

a is a virtual array pointer

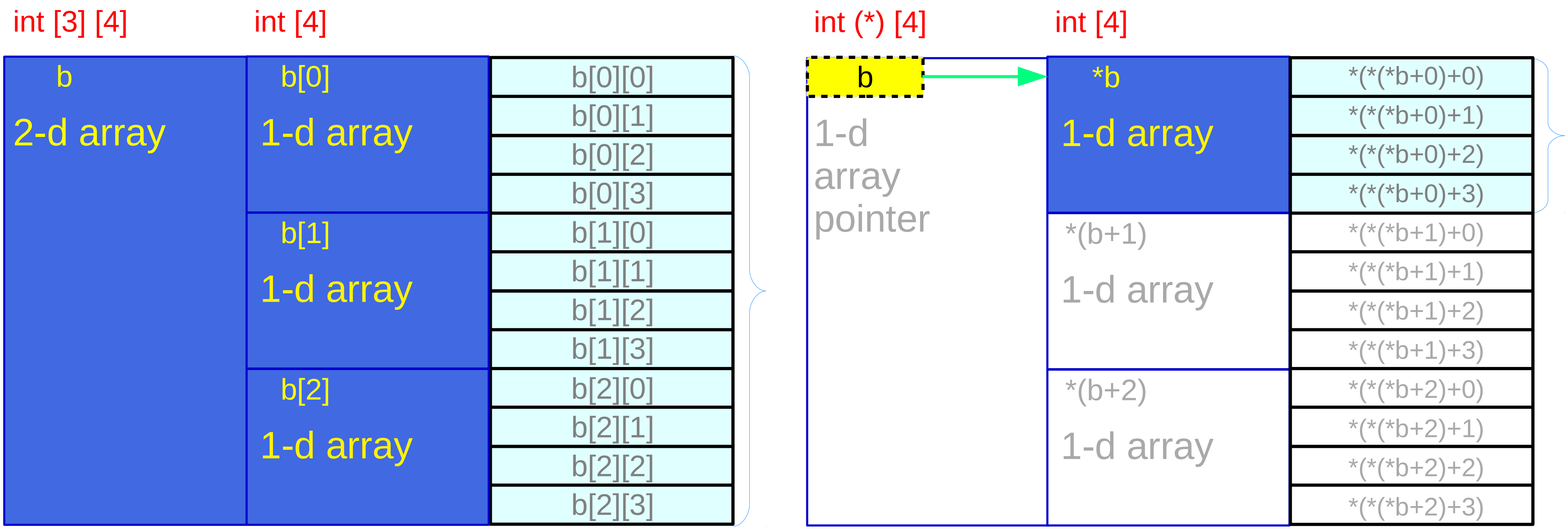
Array b and pointer b

2-d array b specific array type

sizeof(b)

1-d array pointer b general pointer type

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



b is the name of a 2-d array and has a 1-d array pointer type but has a size of the array

b is a <u>virtual</u> array pointer

Array **c**

3-d array c

specific array type

sizeof(c)

c is the name of a 3-d array and has a 2-d array pointer type but has a size of the array

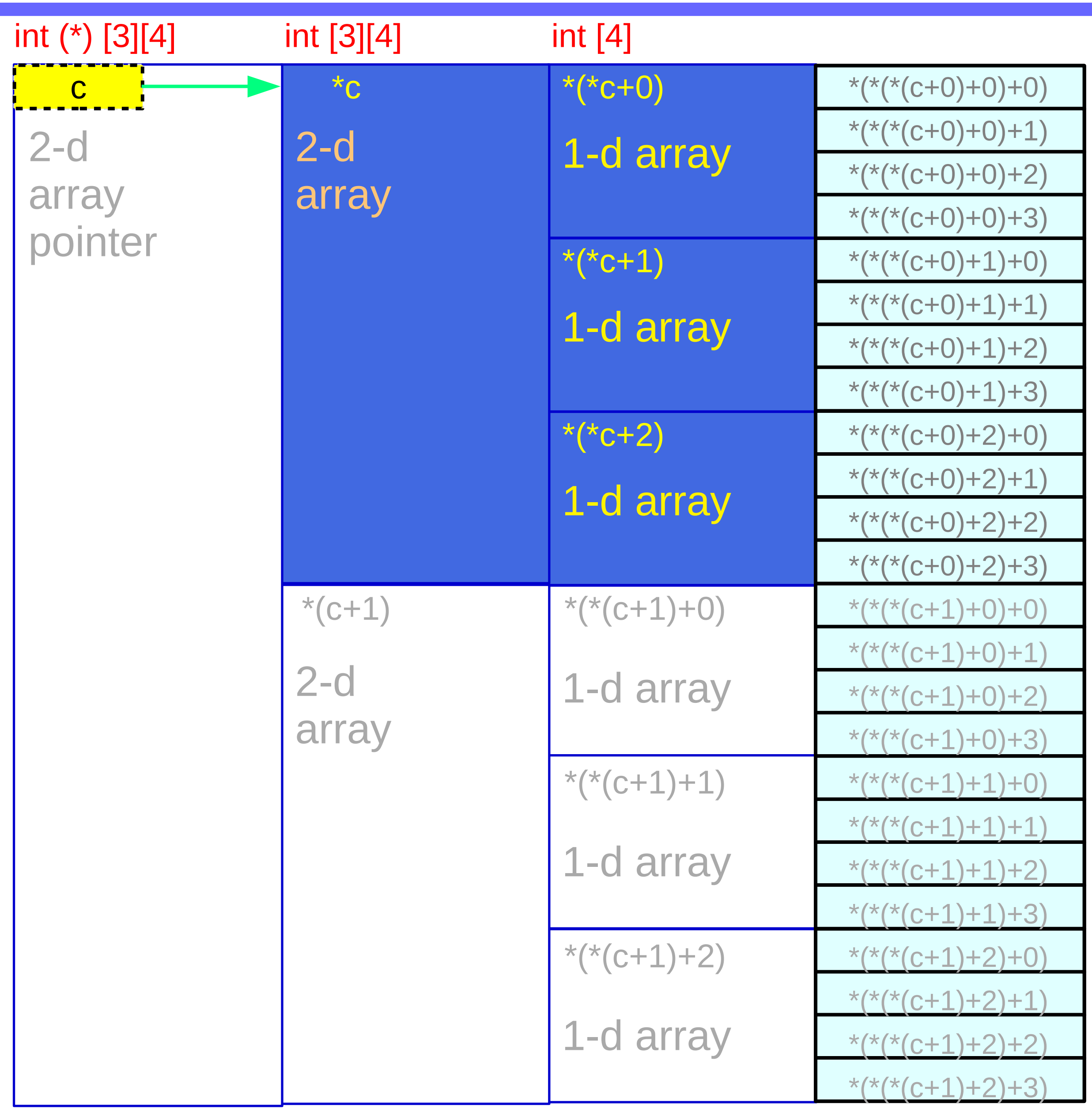
c is a <u>virtual</u> array pointer

Pointer **c**

2-d array pointer c

general pointer type

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

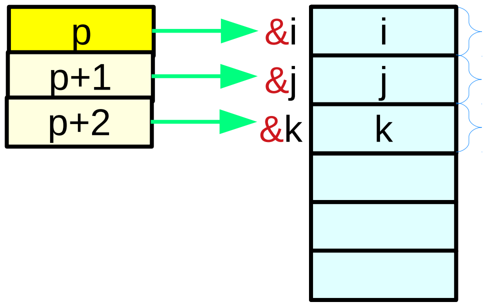


c is the name of a 3-d array and has a 2-d array pointer type but has a size of the array

c is a virtual array pointer

Pointers to primitive data

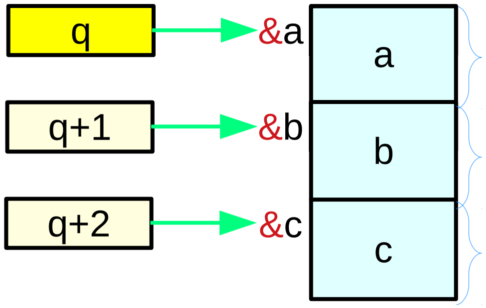
int *p; **int i, j, k;**



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

pointer size
4 or 8 bytes

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



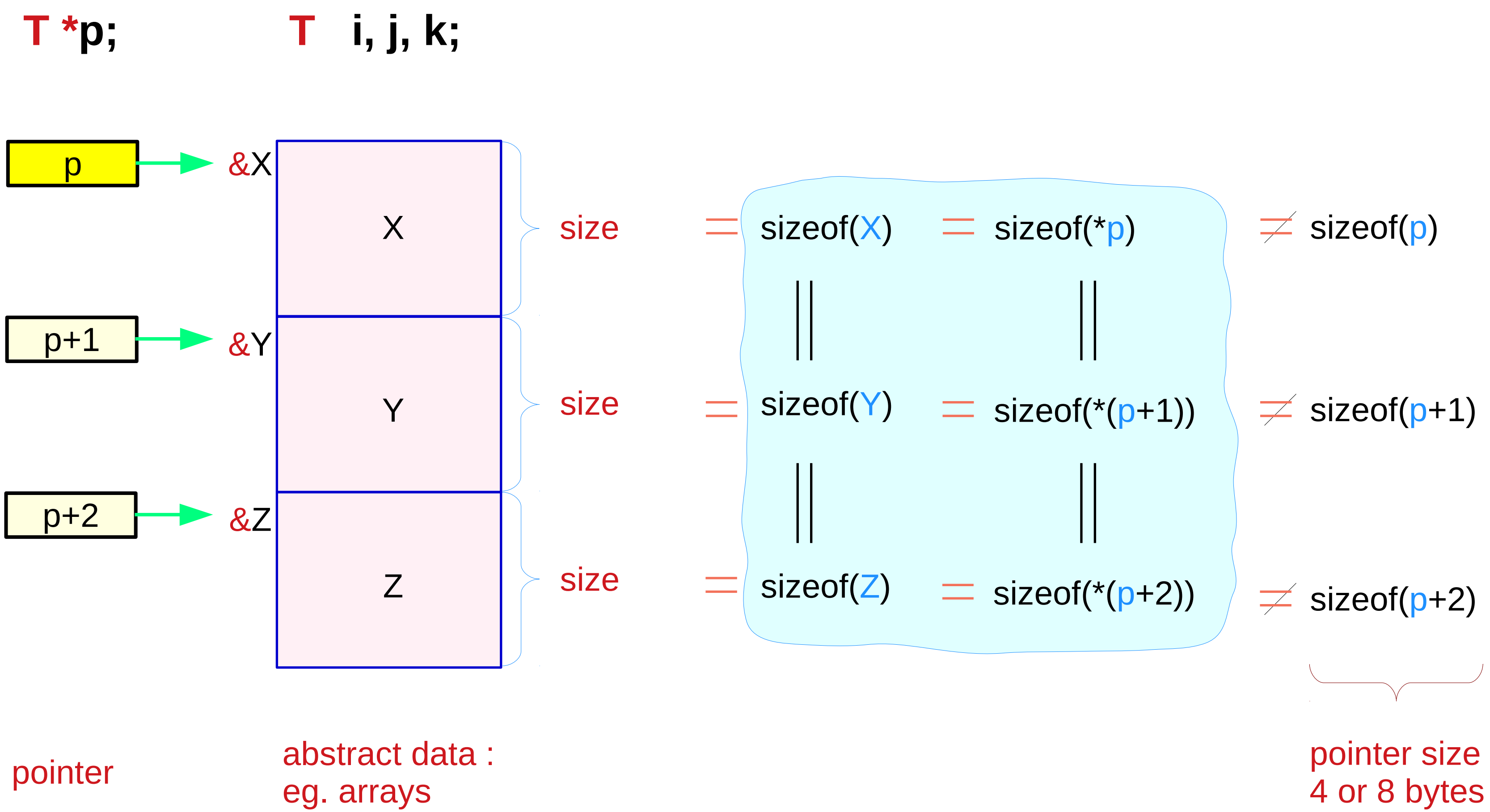
size = sizeof(a) = sizeof(*q) ≠ sizeof(q)

size = sizeof(b) = sizeof(*(q+1)) ≠ sizeof(q+1)

size = sizeof(c) = sizeof(*(q+2)) ≠ sizeof(q+2)

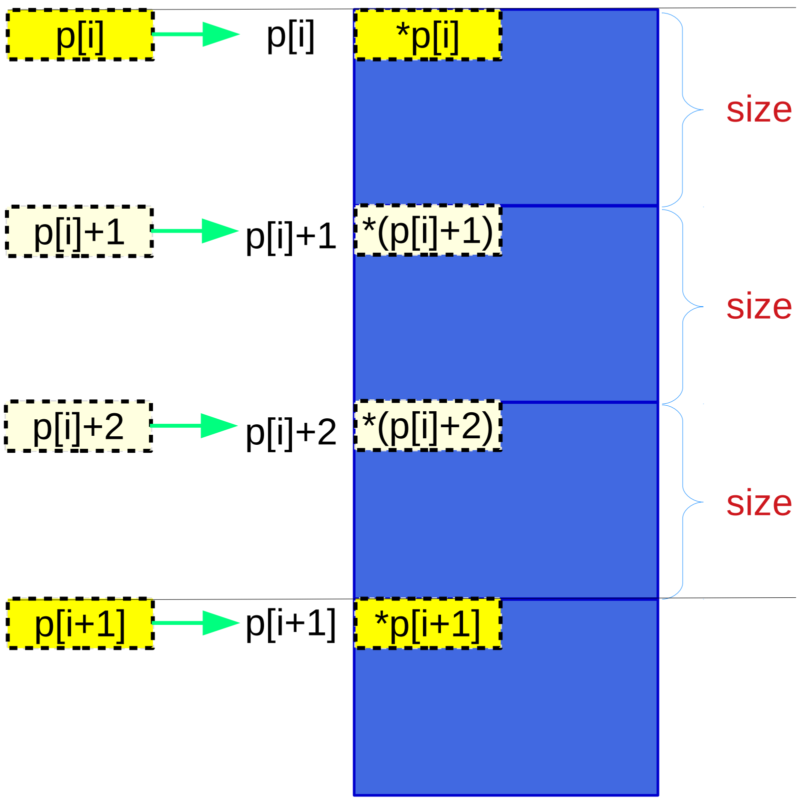
pointer size
4 or 8 bytes

Pointers to abstract data



Virtual pointers in a multi-dimensional array

p[i] :: T1 ***p[i], *p[i+1] :: T2** **T1 p[N];**



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

||

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

||

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

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

p[i] is a virtual pointer
to the abstract data *p[i] = p[i][0]
p[i] is also an abstract data

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

T1		**T2**		
int (*) [4]	→	int [4]	⊂	int (*)
int (*) [3][4]	→	int [3][4]	⊂	int (*)[4]

Virtual pointers in a multi-dimensional array

		T1		**T2**		
int [3] [4]	⊂	int (*) [4]	→	int [4]	⊂	int (*)
int [2] [3][4]	⊂	int (*) [3][4]	→	int [3] [4]	⊂	Int (*) [4]
specific		general		specific		general

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

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

T1 **a**;

T2 **b**;

T1 references T2
T2 is a dereference of T1

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

Virtual array pointer and its increment size

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

c[i][j]

int [4]
int (*)

sizeof(c[i][j]) =

c[i][j][k]

int
int

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

- abstract data type
- array pointer type

= sizeof(int) * **4**

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

c[i]

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

sizeof(c[i]) =

c[i][j]

int [4]
int (*)

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

- abstract data type
- array pointer type

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

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

c

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

sizeof(c) =

c[i]

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

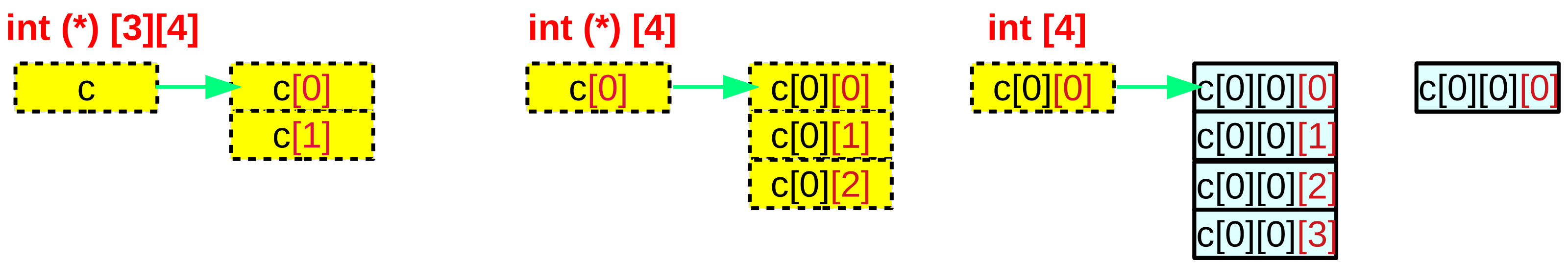
sizeof(c[i]) * **2**

- abstract data type
- array pointer type

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

Virtual array pointer – types and sizes

Types – array pointers



Sizes – abstract data

sizeof(**c**)
sizeof(**c[0]**) * 2
sizeof(**c[0][0]**) * 2 * 3
sizeof(**c[0][0][0]**) * 2 * 3 * 4

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

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

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

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

sizeof(**int [4]**)

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

sizeof(**int**)

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

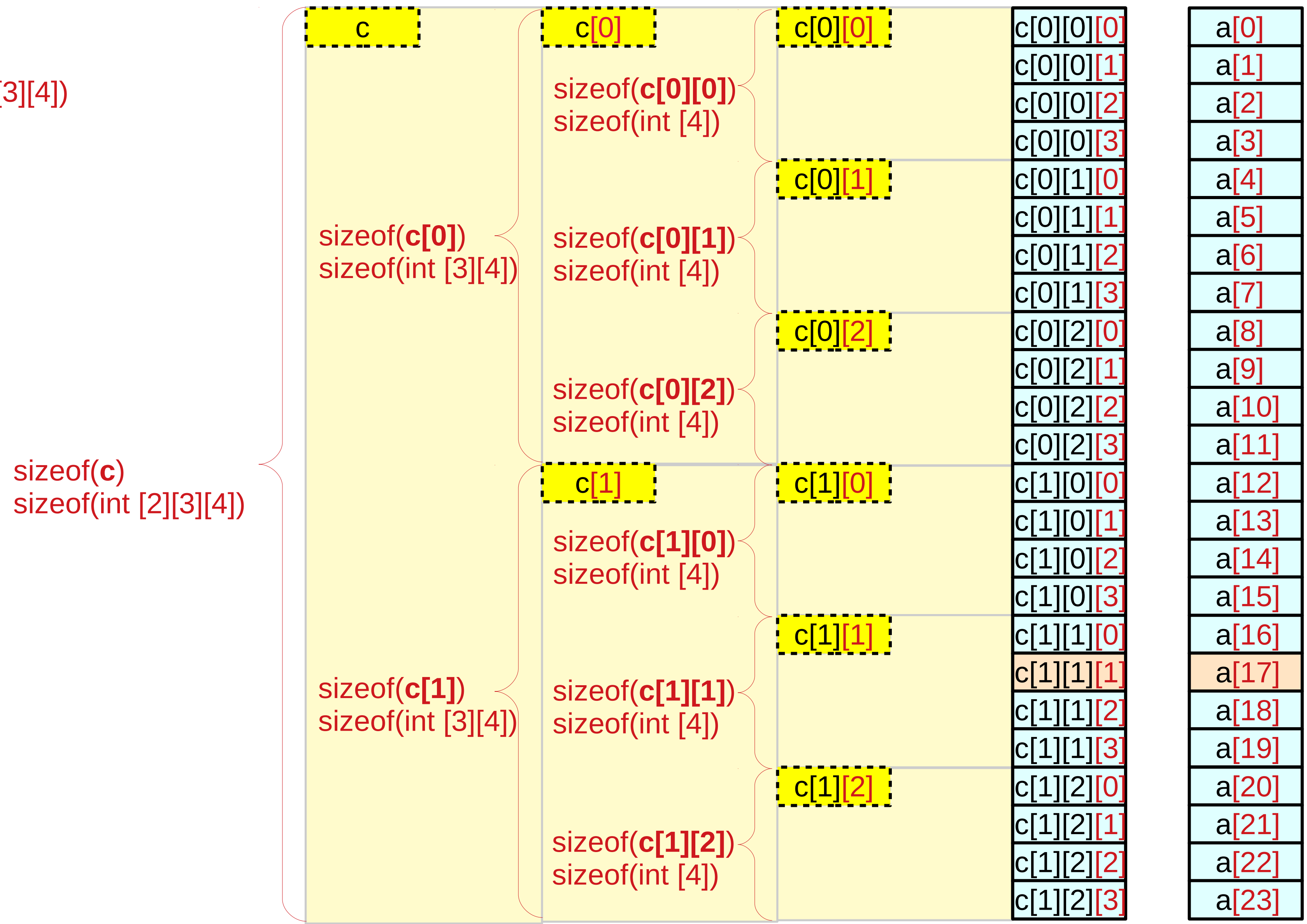
sizeof(**int [3][4]**) = 48
sizeof(**int (*)[4]**) = 4 / 8

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

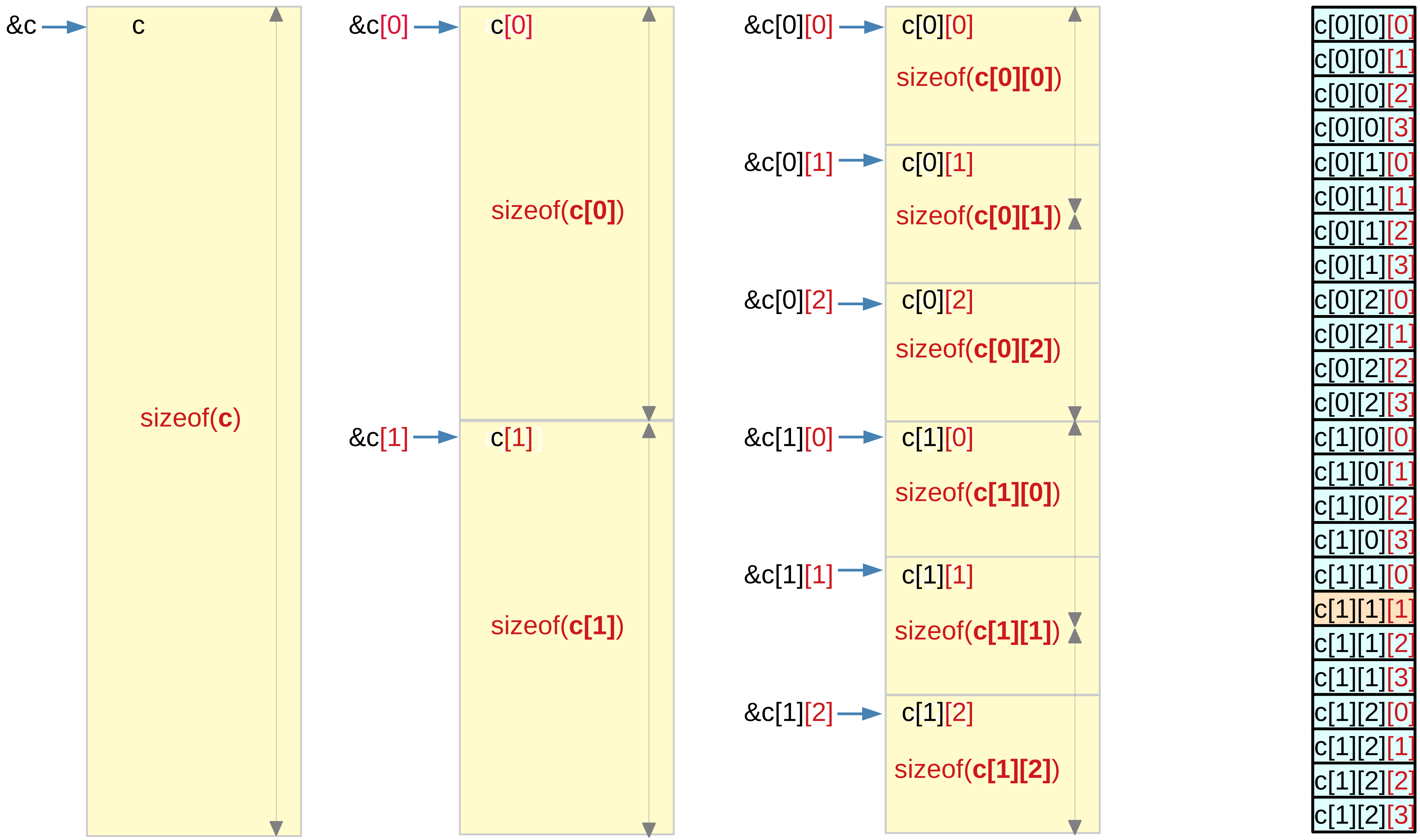
sizeof(**int**) = 4

Virtual array pointers : c, c[i], c[i][j]

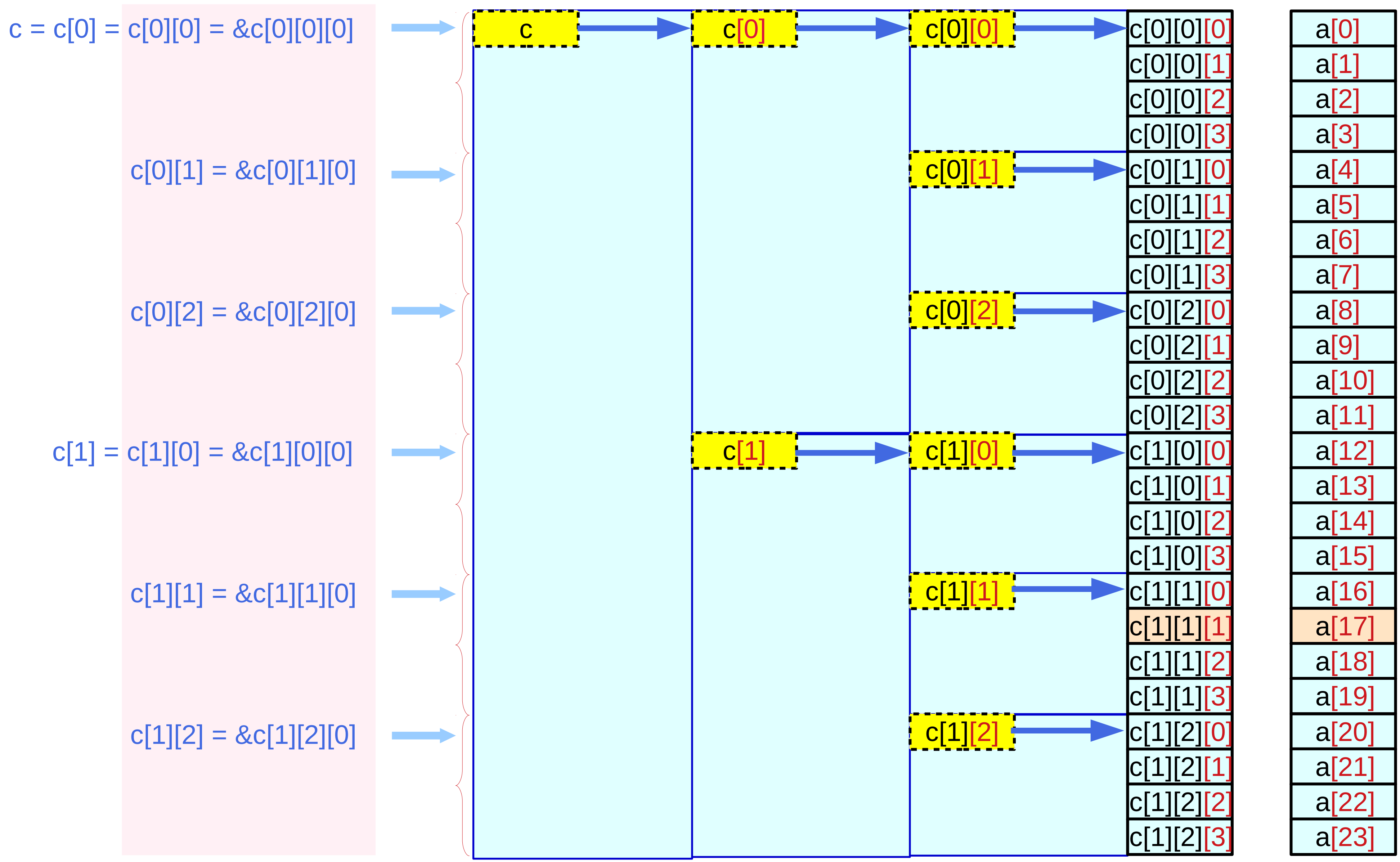
sizeof(**c**)
sizeof(int [2][3][4])



Abstract Data Types: c, c[i], c[i][j]



Virtual array pointer – values (addresses)



Virtual array pointer – vertical displacement

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

For address values

Horizontal displacements
are <u>not</u> counted

Only vertical displacements
are considered

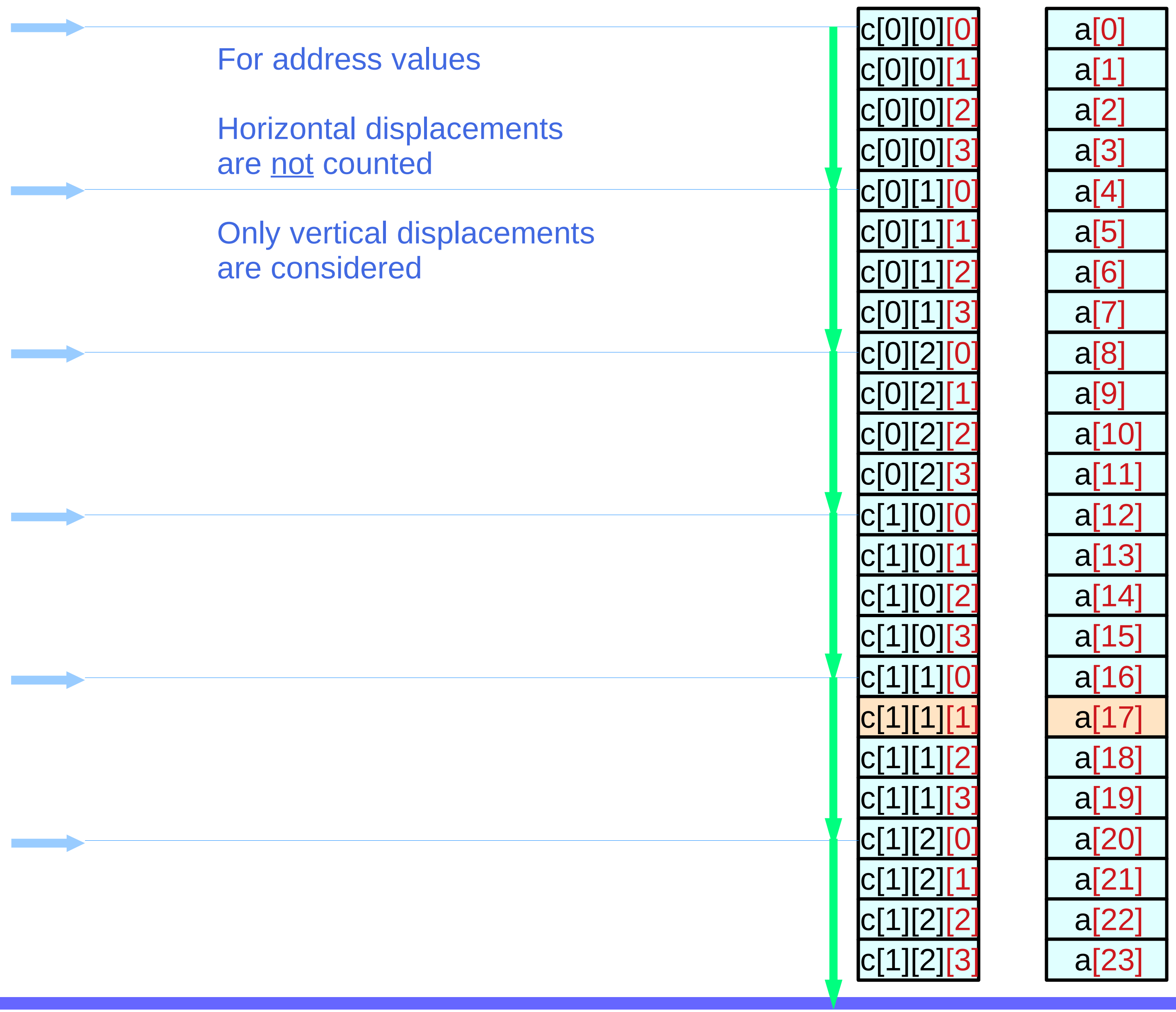
c[0][1] = &c[0][1][0]

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

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

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

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



Virtual array pointer – values and types

c = c[0] = c[0][0] = &c[0][0][0]	means	value(c) = value(c[0]) = value(c[0][0]) = value(&c[0][0][0]) type(c) ≠ type(c[0]) ≠ type(c[0][0]) = type(&c[0][0][0]) int (*) [3][4] int (*) [4] int * int *
c[0][1] = &c[0][1][0]	means	value(c[0][1]) = value(&c[0][1][0]) type(c[0][1]) = type(&c[0][1][0]) int * int *
c[0][2] = &c[0][2][0]	means	value(c[0][2]) = value(&c[0][2][0]) type(c[0][2]) = type(&c[0][2][0]) int * int *
c[1] = c[1][0] = &c[1][0][0]	means	value(c[1]) = value(c[1][0]) = value(&c[1][0][0]) type(c[1]) ≠ type(c[1][0]) = type(&c[1][0][0]) int (*) [4] int * int *
c[1][1] = &c[1][1][0]	means	value(c[1][1]) = value(&c[1][1][0]) type(c[1][1]) = type(&c[1][1][0]) int * int *
c[1][2] = &c[1][2][0]	means	value(c[1][2]) = value(&c[1][2][0]) type(c[1][2]) = type(&c[1][2][0]) int * int *

Array pointers in a multi-dimensional array

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

abstract data int [2] [3][4]
array pointer int (*) [3][4]

c → *c (x 2)

int [3] [4] abstract data
int (*) [4] array pointer

abstract data int [3] [4]
array pointer int (*) [4]

c[i] → *c[i] (x 3)

int [4] abstract data
int (*) array pointer

abstract data int [4]
array pointer int (*)

c[i][j] → *c[i][j] (x 4)

int primitive data

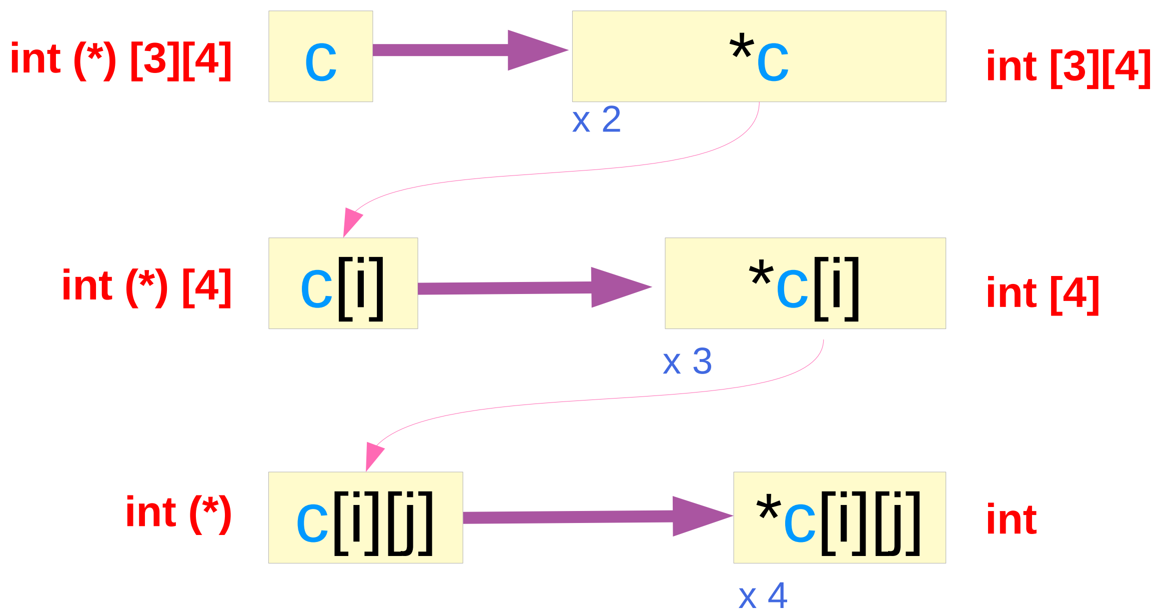
Virtual array pointers

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

2-d array pointer

1-d array pointer

0-d array pointer



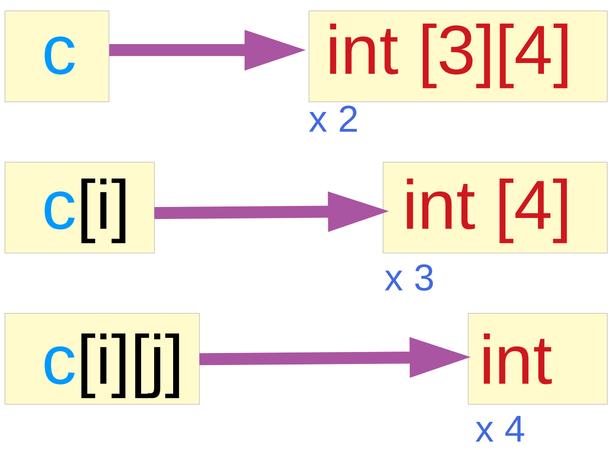
all these pointers are virtual, and take no actual memory locations

exploiting the **contiguity** of allocated memory locations

Virtual array pointers and increment sizes

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

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

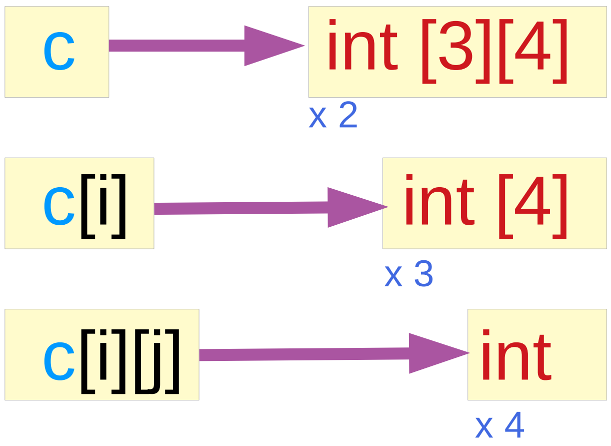



```
sizeof( c)          = sizeof(int [2][3][4])
sizeof(*c)          = sizeof(int [3][4])

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

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

Virtual array pointers and increment sizes

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

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



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

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

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

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

c points to a **2-d** array
increment size: sizeof(int)***2***3*4

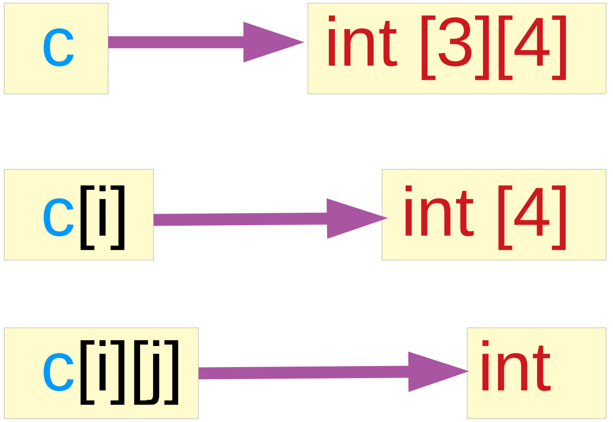
int (*) [3][4]

c[i] points to an **1-d** array
increment size: sizeof(int)***3***4

int (*) [4]

c[i][j] points to an integer
increment size: sizeof(int)***4**

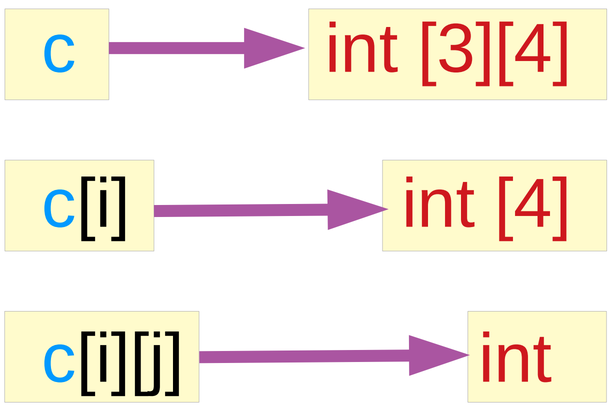
int (*)



Virtual array pointer types and sizes

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

not real array pointers
virtual array pointers



c int (*)[3][4]
sizeof(int (*) [3][4]) = pointer size ≠ sizeof(c)

c[i] int (*) [4]
sizeof(int (*) [4]) = pointer size ≠ sizeof(c[i])

c[i][j] int [4]
sizeof(int [4]) = pointer size ≠ sizeof(c[i][j])

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

Sub-array properties in multi-dimensional arrays

int c [2][3][4]; → 3-d access c [i][j][k]

2-d array pointer	c	int (*) [3][4]
1-d array pointers	c[i]	int (*) [4]
0-d array pointers	c[i][j]	int (*)

Hierarchical Sub-arrays in a 3-d array

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

c [i][j][k]

left-to-right associativity

c[i][j][k]

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

Array Names and Types

Pointers to hierarchical sub-arrays

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

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

Virtual Array Pointer Constraints

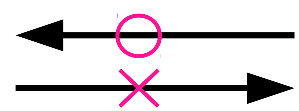
Abstract Data Type

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



Array pointer approach

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

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

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

Virtual Array Pointer Constraints

Abstract Data Type

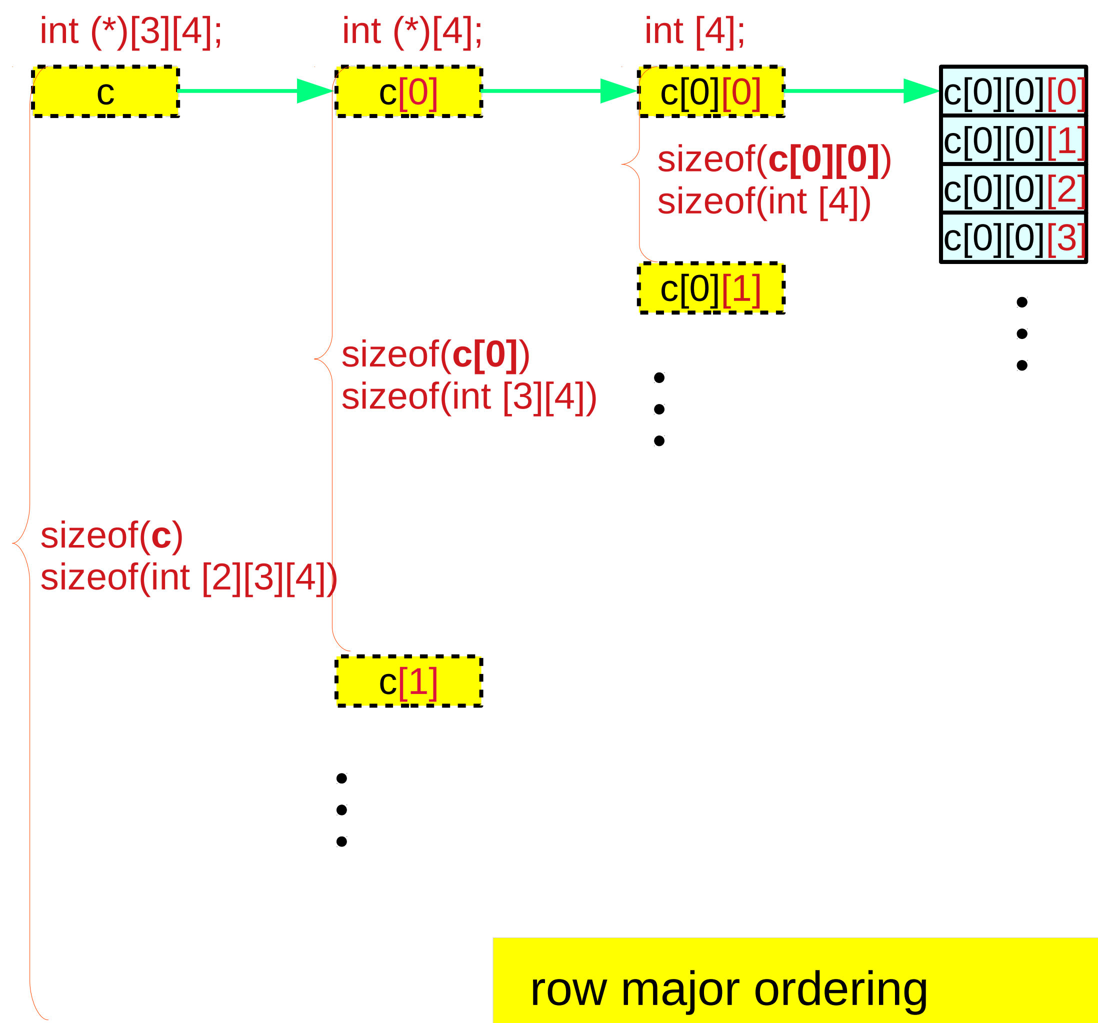
Using virtual array pointers

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

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

constraints

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



row major ordering
contiguous linear layout

Types of **c[i]** and **c[i][j]**

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]

c[i][j] : the name of 1-d array with 4 integers

c[i][j] (virtual array) pointer of the type int (*)

c[i][j][k] : an element of a 4-integer array

c →	c[0] →	c[0][0] →	c[0][0][0] … c[0][0][3]
		c[0][1] →	c[0][1][0] … c[0][1][3]
		c[0][2] →	c[0][2][0] … c[0][2][3]
	c[1] →	c[1][0] →	c[1][0][0] … c[0][0][3]
		c[1][1] →	c[1][1][0] … c[0][1][3]
		c[1][2] →	c[1][2][0] … c[0][2][3]
int (*) [3][4]	int (*) [4]	int [4]	int … int
pointers to a 2-d array	pointers to a 1-d array	1-d array names	leading element of 4-integer array

Values of **c[i]** and **c[i][j]**

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

c →	c[0] →	c[0][0]	≡	&c[0][0][0]
		c[0][1]	≡	&c[0][1][0]
		c[0][2]	≡	&c[0][2][0]
	c[1] →	c[1][0]	≡	&c[1][0][0]
		c[1][1]	≡	&c[1][1][0]
		c[1][2]	≡	&c[1][2][0]

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

c →	c[0]	≡	&c[0][0]
		≡	&c[0][1]
		≡	&c[0][2]
	c[1]	≡	&c[1][0]
		≡	&c[1][1]
		≡	&c[1][2]

c[i] and c[i][j] : virtual array pointers

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]
c[i][j] : a 4-element 1-d array name

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

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

c[0] is the address of c[0][0]

c[1] is the address of c[1][0]

c[i][j] virtual array pointer of the type int (*)
c[i][j][0] : leading element of a 4-integer array

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

c[0][0] is the address of c[0][0][0]
c[0][1] is the address of c[0][1][0]
c[0][2] is the address of c[0][2][0]
c[1][0] is the address of c[1][0][0]
c[1][1] is the address of c[1][1][0]
c[1][2] is the address of c[1][2][0]

Values of **c[i]** and **c[i][j]**

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];

virtual array pointers
in each row in the following figure
have the same value (address value)

c	≡	c[0]	≡	c[0][0]	≡	&c[0][0][0]
				c[0][1]	≡	&c[0][1][0]
				c[0][2]	≡	&c[0][2][0]
		c[1]	≡	c[1][0]	≡	&c[1][0][0]
				c[1][1]	≡	&c[1][1][0]
				c[1][2]	≡	&c[1][2][0]

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

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

Finding address values of **c[i]** and **c[i][j]**

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

add [0] to the right

c	+[0] =	c[0]	+[0] =	c[0][0]	+[0] =	&c[0][0][0]
				c[0][1]	+[0] =	&c[0][1][0]
				c[0][2]	+[0] =	&c[0][2][0]
		c[1]	+[0] =	c[1][0]	+[0] =	&c[1][0][0]
				c[1][1]	+[0] =	&c[1][1][0]
				c[1][2]	+[0] =	&c[1][2][0]
int (*) [3][4]		int (*) [4]		int [4]		int

delete [0] from the right

&c[0][0][0]	-[0] =	c[0][0]	-[0] =	c[0]	-[0] =	c
&c[0][1][0]	-[0] =	c[0][1]				
&c[0][2][0]	-[0] =	c[0][2]				
&c[1][0][0]	-[0] =	c[1][0]	-[0] =	c[1]		
&c[1][1][0]	-[0] =	c[1][1]				
&c[1][2][0]	-[0] =	c[1][2]				
int		int [4]		int (*) [4]		int (*) [3][4]

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

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

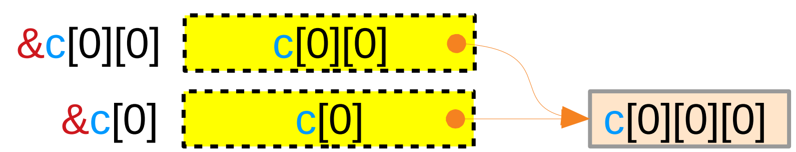
int c [2][3][4];

c ═ c[0] ═ c[0][0] ═ &c[0][0][0]

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

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

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



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

Addresses and Values of **c[i]** and **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][3][4];

c → c[0] → c[0][0] = &c[0][0][0]

&c[0] c[0]

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

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

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

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

address	values	
&X	&X	X
&c[0][0][0]	&c[0][0][0]	c[0][0][0]

c[i] and c[i][0] point to the same c[i][0][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][3][4];

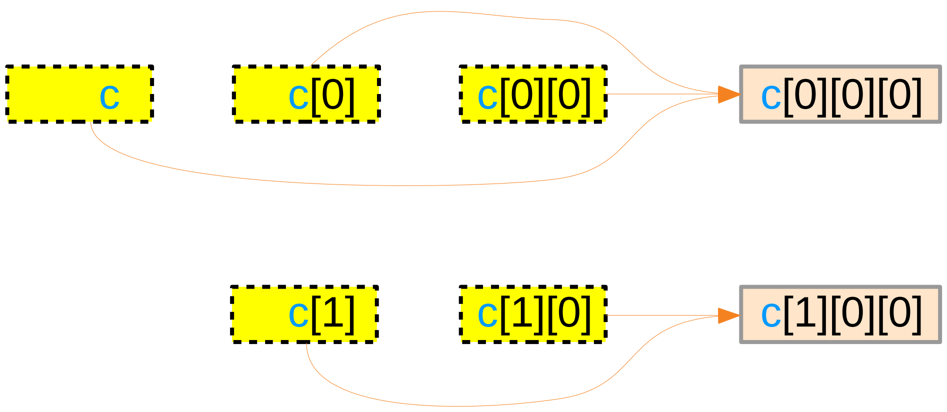
c[i][j] = &c[i][j][0]
c[i] = &c[i][0][0]
c = &c[0][0][0]

c = c[0] = c[0][0] = &c[0][0][0] ⟸ value

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

c[1] = c[1][0] = &c[1][0][0] ⟸ value

int(*)[4] int(*) int ⟸ type



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

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

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

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

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

&

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

equivalences

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

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

&

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

equivalences

c[1] ≡ &c[1][0]
c[1][0] ≡ &c[1][0][0]

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

equivalences

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

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

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

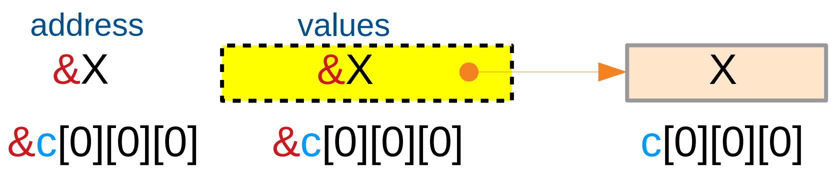
int c [2][3][4];

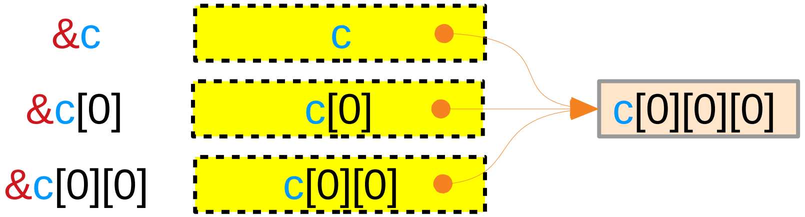
c[i][j] = &c[i][j][0]
c[i] = &c[i][0][0]
c = &c[0][0][0]

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

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

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





Array Pointers to **c[i][0][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][3][4];

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

&c[i][0][0] ≡ c[i][0]

&c[i][0] ≡ c[i]

&c[i] ≡ c+i = &c[0][0][0] + i*3*4

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

delete [0] from the right

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

Array Pointers to **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][3][4];

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

&c[i][j][0] ≡ c[i][j]

||

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

delete [0] from the right

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

Contiguity Constraints

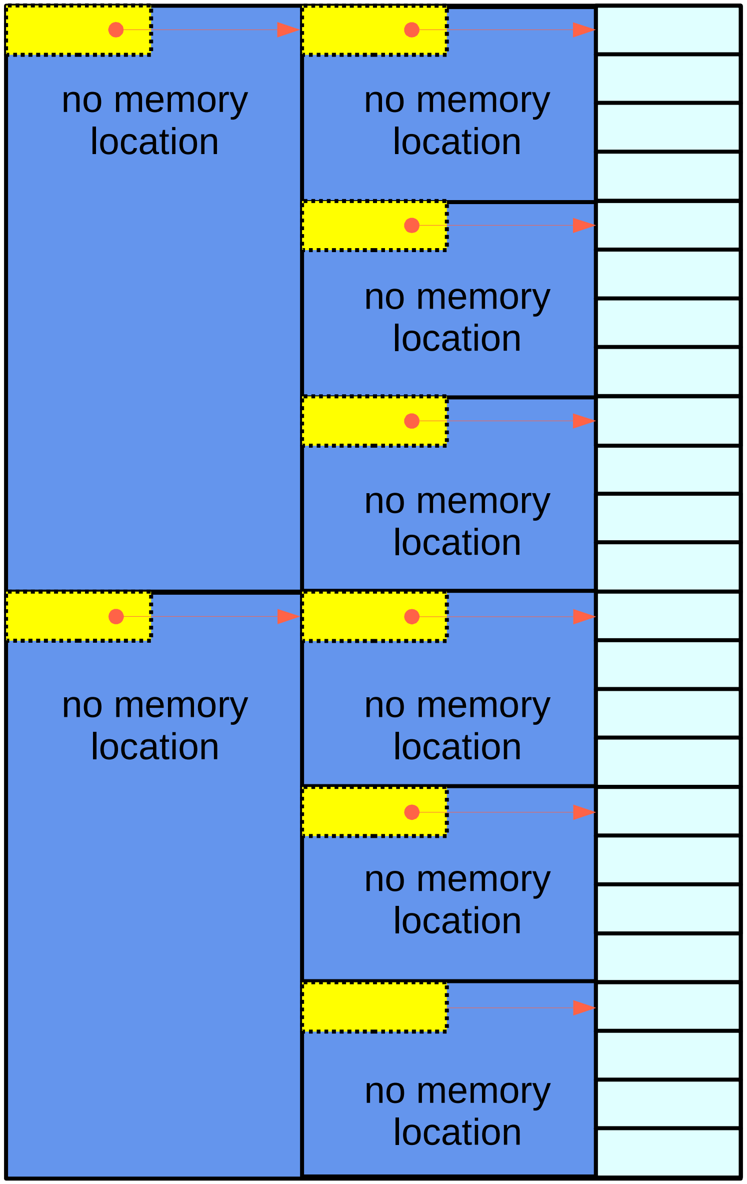
c [i][j][k];

Virtual Array Pointers and Contiguity

Using array pointers

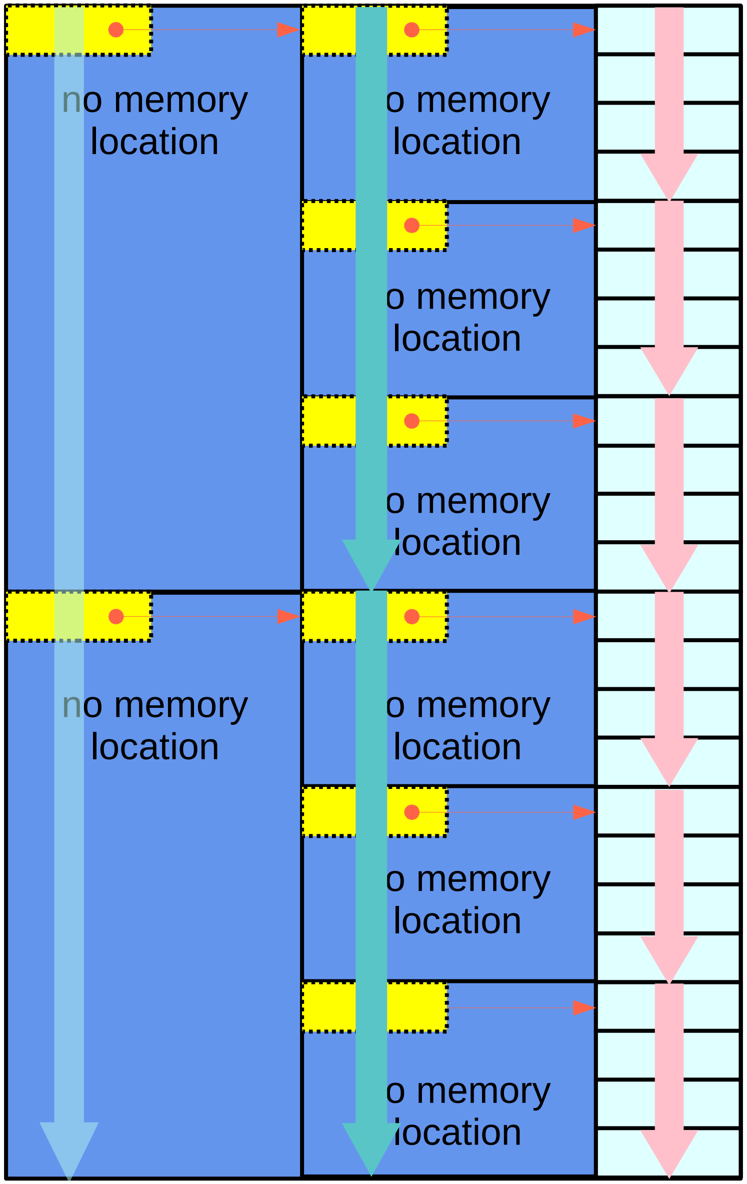
int (*) [N], int (*) [M][N], int (*) [L][M][N], …

Array pointer approach for 3-d access patterns



Array Pointer Approach (pointer to arrays)

Array pointer approach – contiguity constraints



Array Pointer Approach (pointer to arrays)

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)

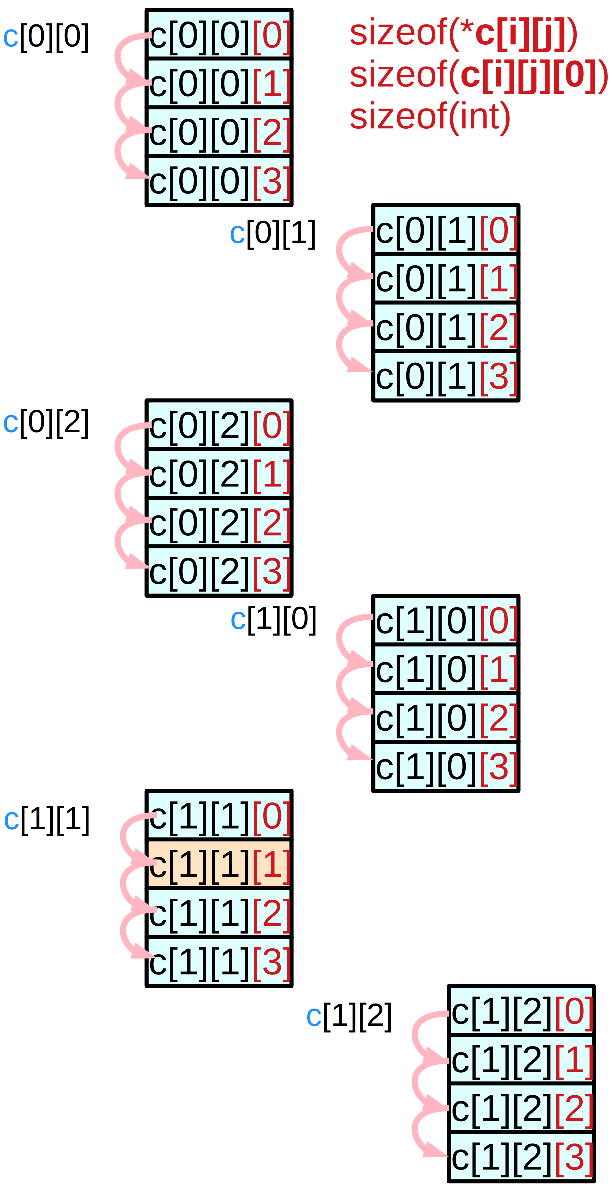
⋮ ⋮

contiguous 1-d
array elements

c[i][j] :: int *
contiguous 1-d
array elements
int … 4 elements
sizeof(**c[i][j]**)
sizeof(int) * 4

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

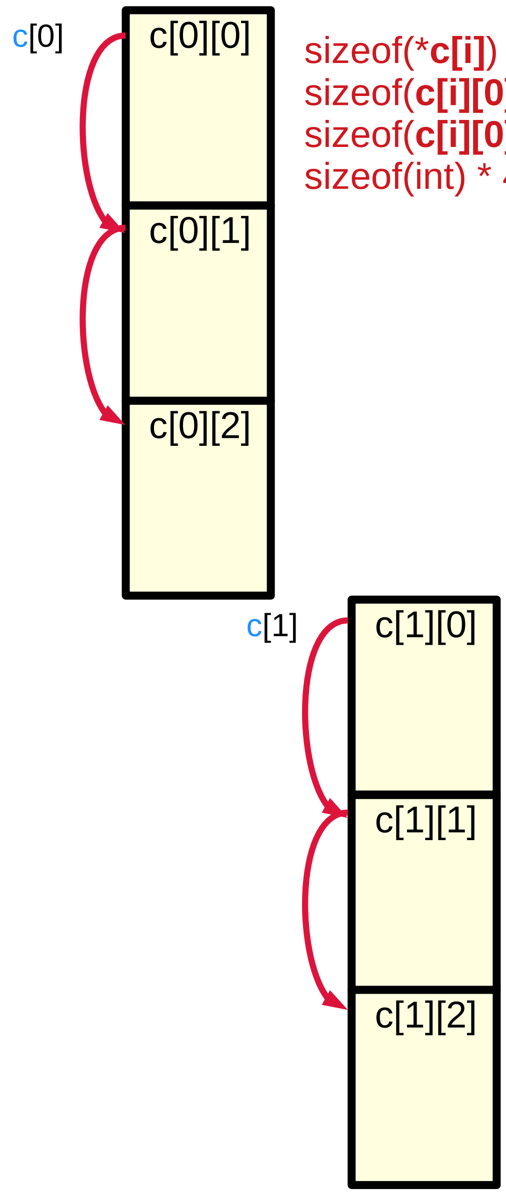
sizeof(***c[i][j]**)
sizeof(**c[i][j][0]**)
sizeof(int)



c[i][j] ≡ *(c[i] +j)

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

c[i] :: int (*) [4]
contiguous 1-d arrays
int[4] = int * … 3 arrays
sizeof(**c[i]**)
sizeof(**c[i][j]**) * 3
sizeof(**c[i][j][k]**) * 3 * 4
sizeof(int) * 3 * 4



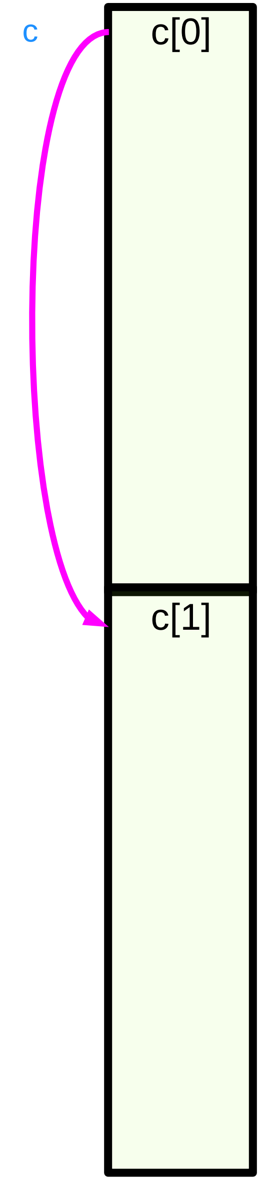
sizeof(***c[i]**)
sizeof(**c[i][0]**)
sizeof(**c[i][0][0]**) * 4
sizeof(int) * 4

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

c[i] ≡ *(c +i)

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

c :: int (*) [3][4]
contiguous
1-d array pointers
Int (*) [4] … 2 array pointers
sizeof(**c**)
sizeof(**c[i]**) * 2
sizeof(**c[i][j]**) * 2 * 3
sizeof(**c[i][j][k]**) * 2 * 3 * 4
sizeof(int) * 2 * 3 * 4



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

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

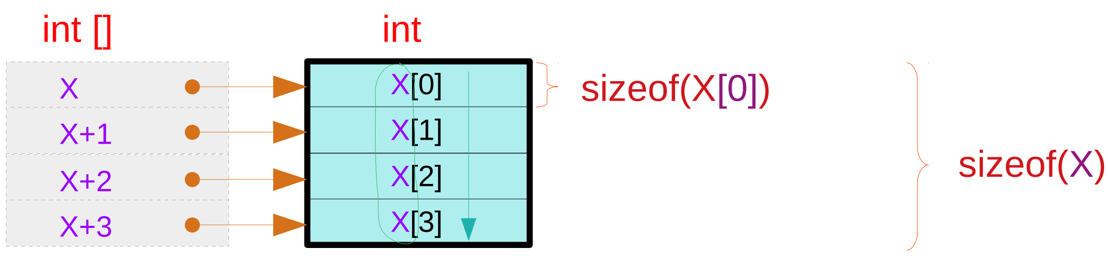
c[i] ≡ *(c +i)

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

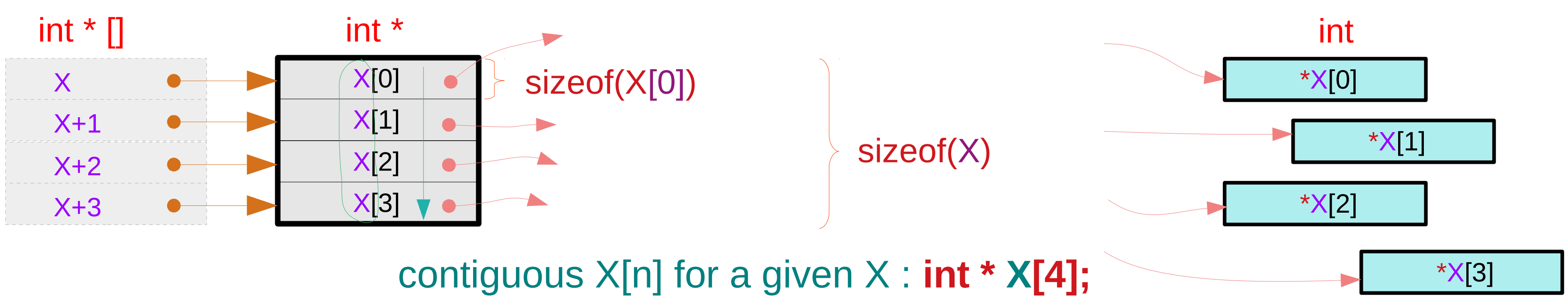
Equivalence and contiguity

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

contiguous index : n



contiguous X[n] for a given X : **int X[4];**



contiguous X[n] for a given X : **int * X[4];**

Equivalence

By definition, contiguous memory locations are assumed

*(X+n)	≡	X[n]

contiguous index : n

*(p[m]+n)	⟷	p[m][n]
(*(p+m))[n];	⟷	p[m][n];

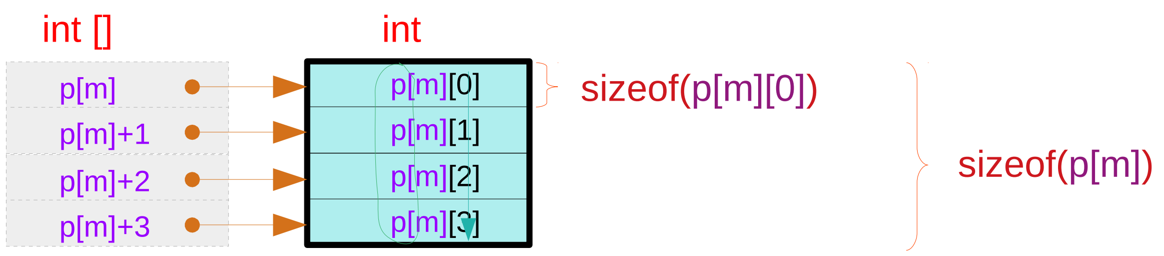
X = p[m] contiguous index : **n**

X = p contiguous index : **m**

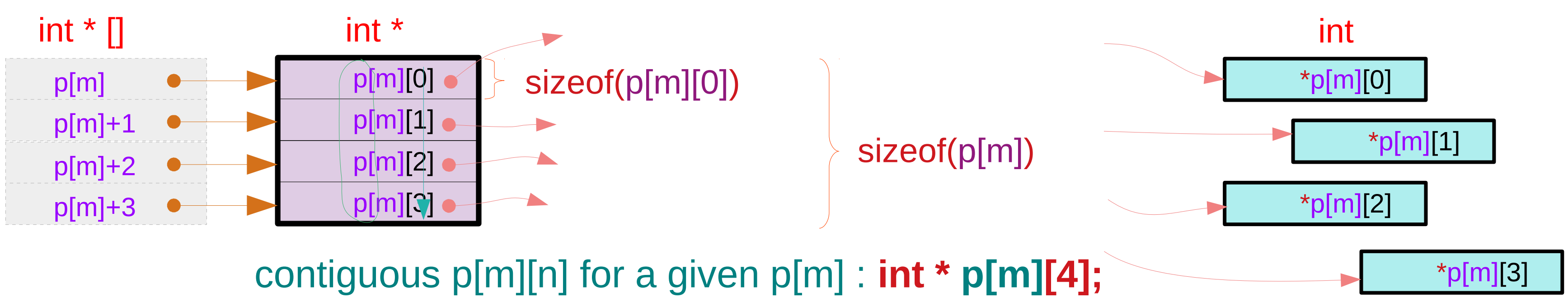
For a given p[m] – int pointer / pointer to int pointer

*(p[m]+n) ⟺ p[m][n]

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



contiguous p[m][n] for a given p[m] : **int p[m][4];**



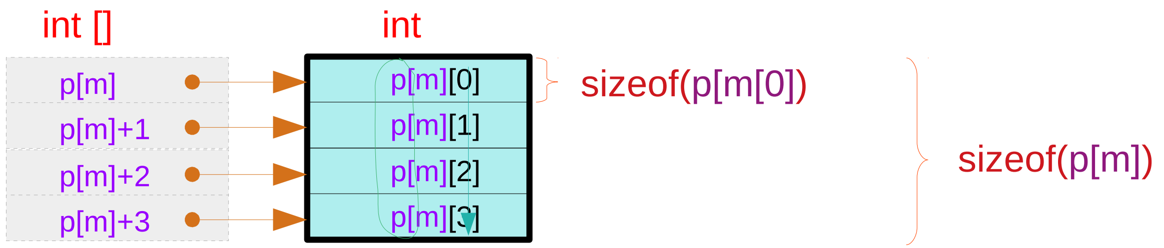
contiguous p[m][n] for a given p[m] : **int * p[m][4];**

For a given p[m] – pointer to an abstract data

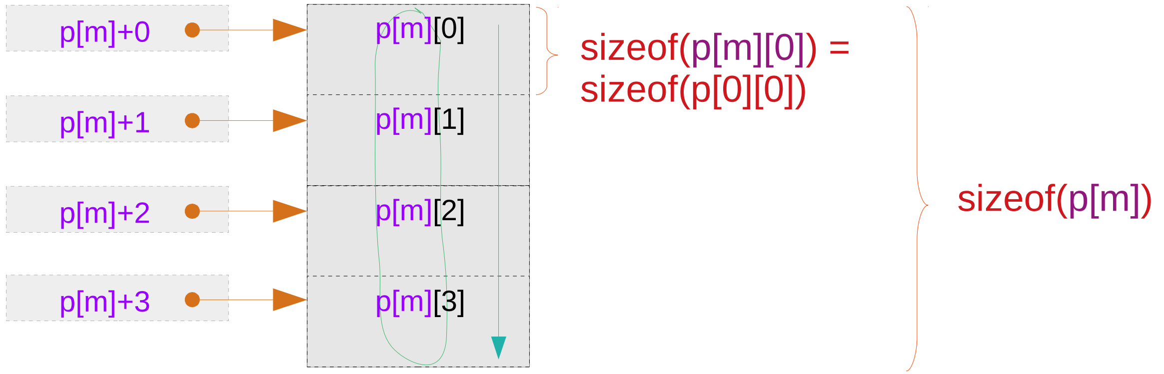
*(p[m]+n) ⟺ p[m][n]

for a given p[m]

contiguous index : **n**



contiguous p[m][n] for a given p[m] : int X[4];

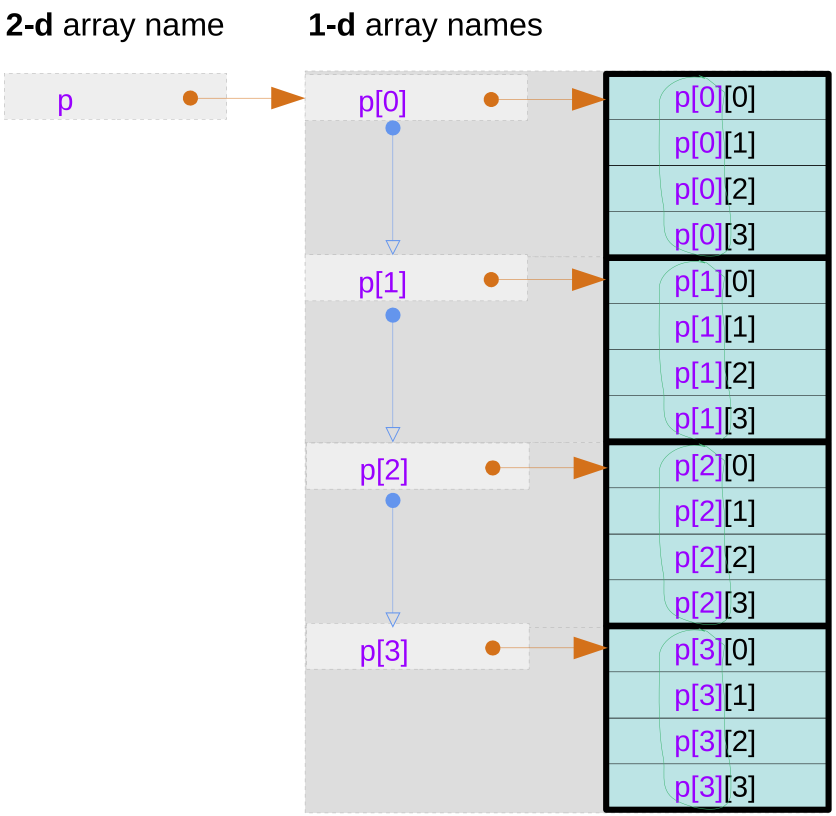


Contiguity constraints

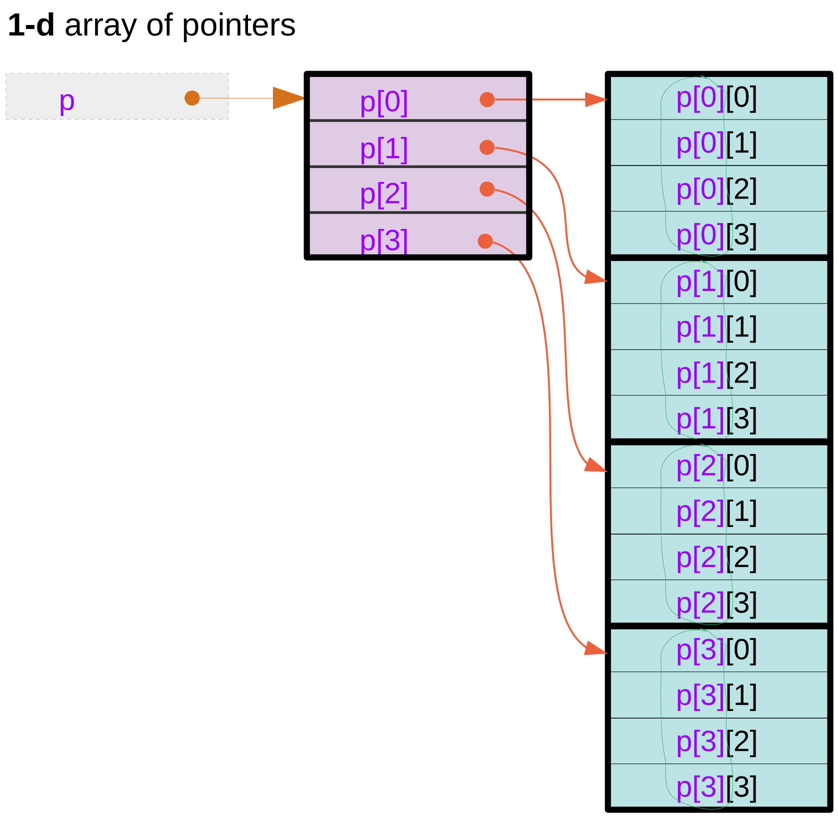
(*(p+m))[n]; ⟺ p[m][n];

for a given p

contiguous index : **m**



contiguous p[m] ➡ contiguous p[m][n]



contiguous p[m] ➡ contiguous p[m][n]

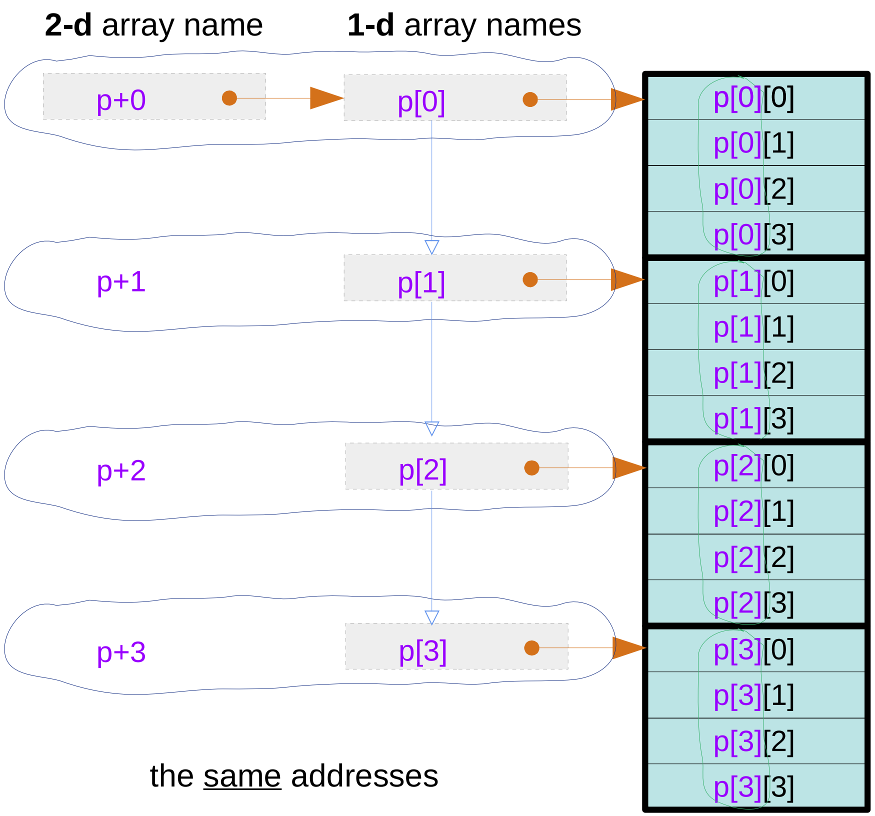
Not necessarily

Contiguity constraints – using array pointers

(*(p+m))[n]; ⟷ p[m][n];

for a given p

contiguous index : **m**



contiguous p[m] ➡ contiguous p[m][n]

virtual array pointer

data	addr		addr	
p[0][0] = *(p[0]+0)	➡	(p[0]+0) = p[0]	➡	p+0
p[1][0] = *(p[1]+0)	➡	(p[1]+0) = p[1]	➡	p+1
p[2][0] = *(p[2]+0)	➡	(p[2]+0) = p[2]	➡	p+2
p[3][0] = *(p[3]+0)	➡	(p[3]+0) = p[3]	➡	p+3

the same addresses

⟷ no real memory locations

Contiguity constraints – using pointer arrays

(*(p+m))[n]; ⟷ p[m][n];

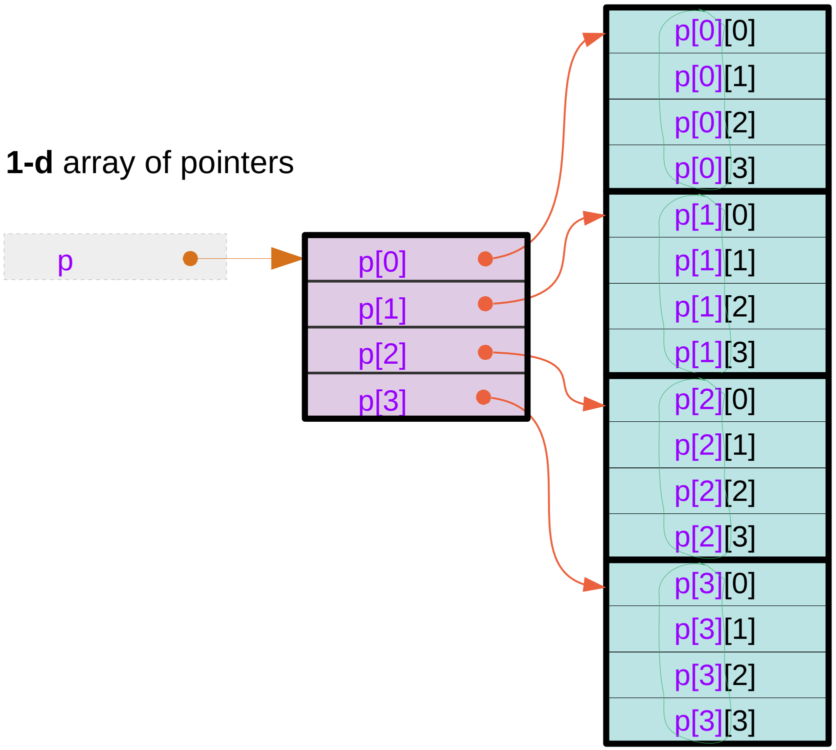
for a given p

contiguous index : **m**

the different addresses

data	addr		addr	
p[0][0] = *(p[0]+0)	⇒	(p[0]+0) = p[0]	⇒	p+0
p[1][0] = *(p[1]+0)	⇒	(p[1]+0) = p[1]	⇒	p+1
p[2][0] = *(p[2]+0)	⇒	(p[2]+0) = p[2]	⇒	p+2
p[3][0] = *(p[3]+0)	⇒	(p[3]+0) = p[3]	⇒	p+3

1-d array of pointers

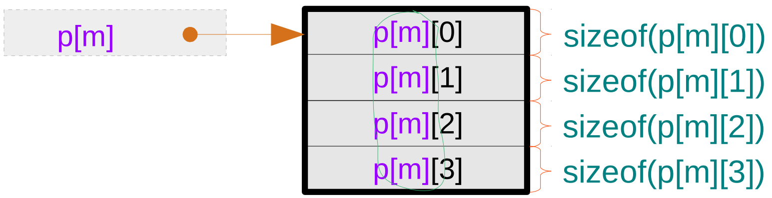


contiguous p[m] ⇒ contiguous p[m][n]

Not necessarily

Contiguity constraints

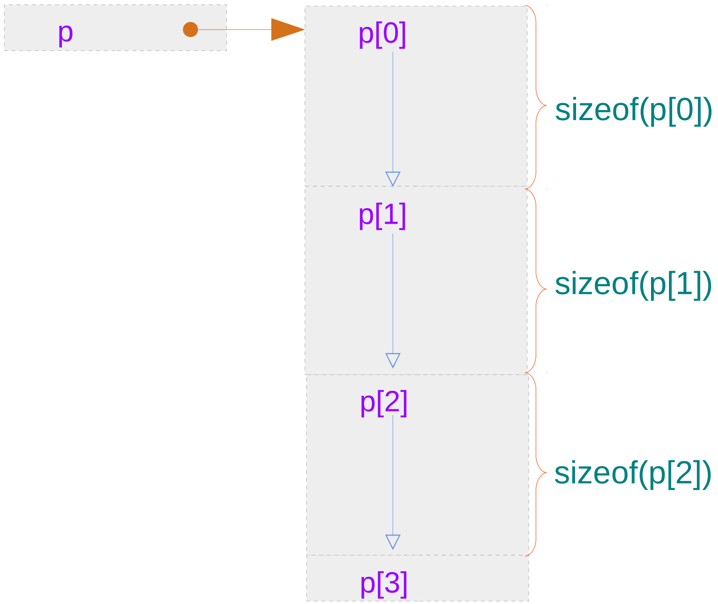
*(p[m]+n) ⟺ p[m][n]



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

contiguous index : n

(*(p+m))[n]; ⟺ p[m][n];



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

contiguous index : m

➡ all p[m][n]'s must be contiguous for all m, n

Contiguity constraints

int a[M][N] ;

(*(a+m))[n] ⟺ a[m][n]
*(a[m]+n) ⟺ a[m][n]

int (*b)[N] ;

(*(b+m))[n] ⟺ b[m][n]
*(b[m]+n) ⟺ b[m][n]

int * c[M] ;

(*(c+m)) ⟺ c[m]
needs assignments

Contiguity constraints

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

```
int (*b)[N] ;
```

(*(a+m))[n] ⟺ a[m][n]

a[0], a[1], … , a[M-1]
are contiguous

*(a[m]+n) ⟺ a[m][n]

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

(*(b+m))[n] ⟺ b[m][n]

b[0], b[1], … , b[M-1]
are contiguous

*(b[m]+n) ⟺ b[m][n]

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

Contiguity constraints

int a[M][N] ;

(*(a+m))[n] ⟺ a[m][n]

a[0], a[1], … , a[M-1]
are contiguous

*(a[m]+n) ⟺ a[m][n]

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

int * c[M] ;

(*(c+m)) ⟺ c[m]

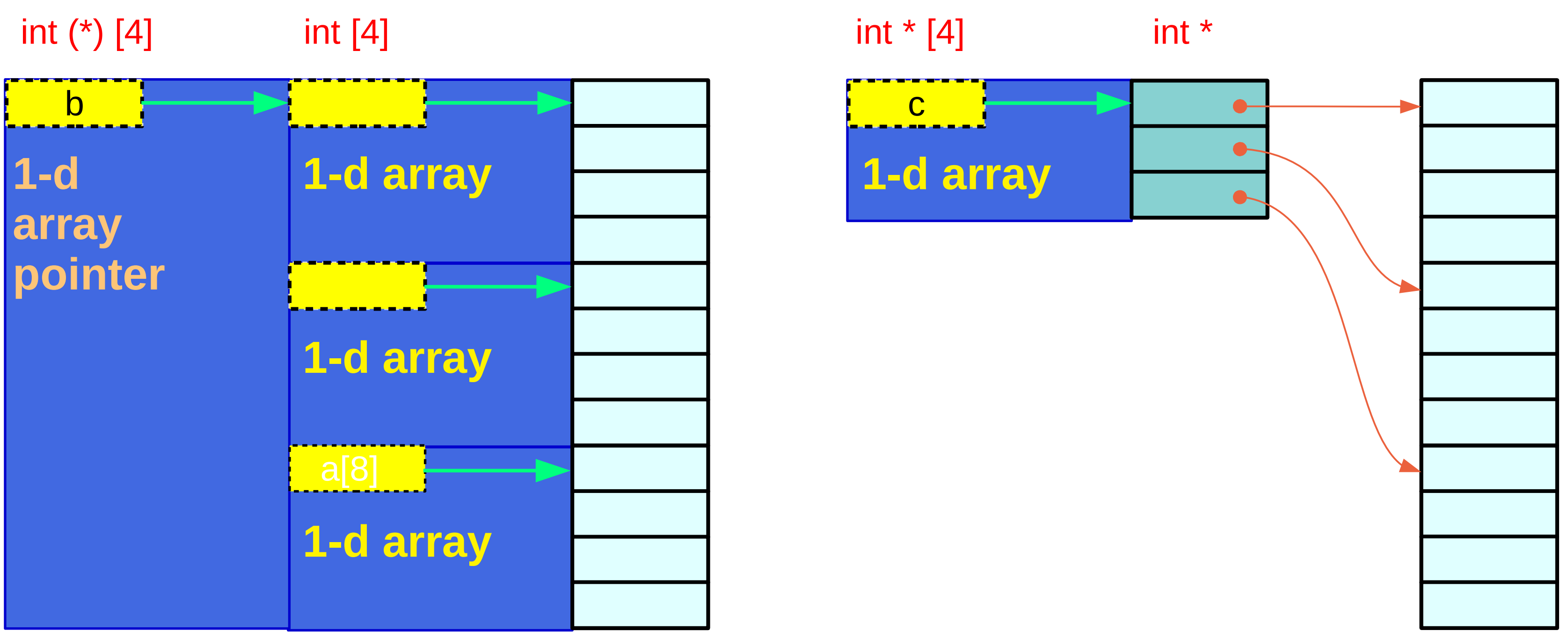
c[0], c[1], … , c[M-1]
are contiguous

(*(c+m))[n] ⟺ c[m][n]

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

a set of assignments of pointers
are necessary for this contiguity

Pointer Arrays vs Array Pointers



int (*b)[N] ;

(*(b+m))[n] ⟺ b[m][n]
*(b[m]+n) ⟺ b[m][n]

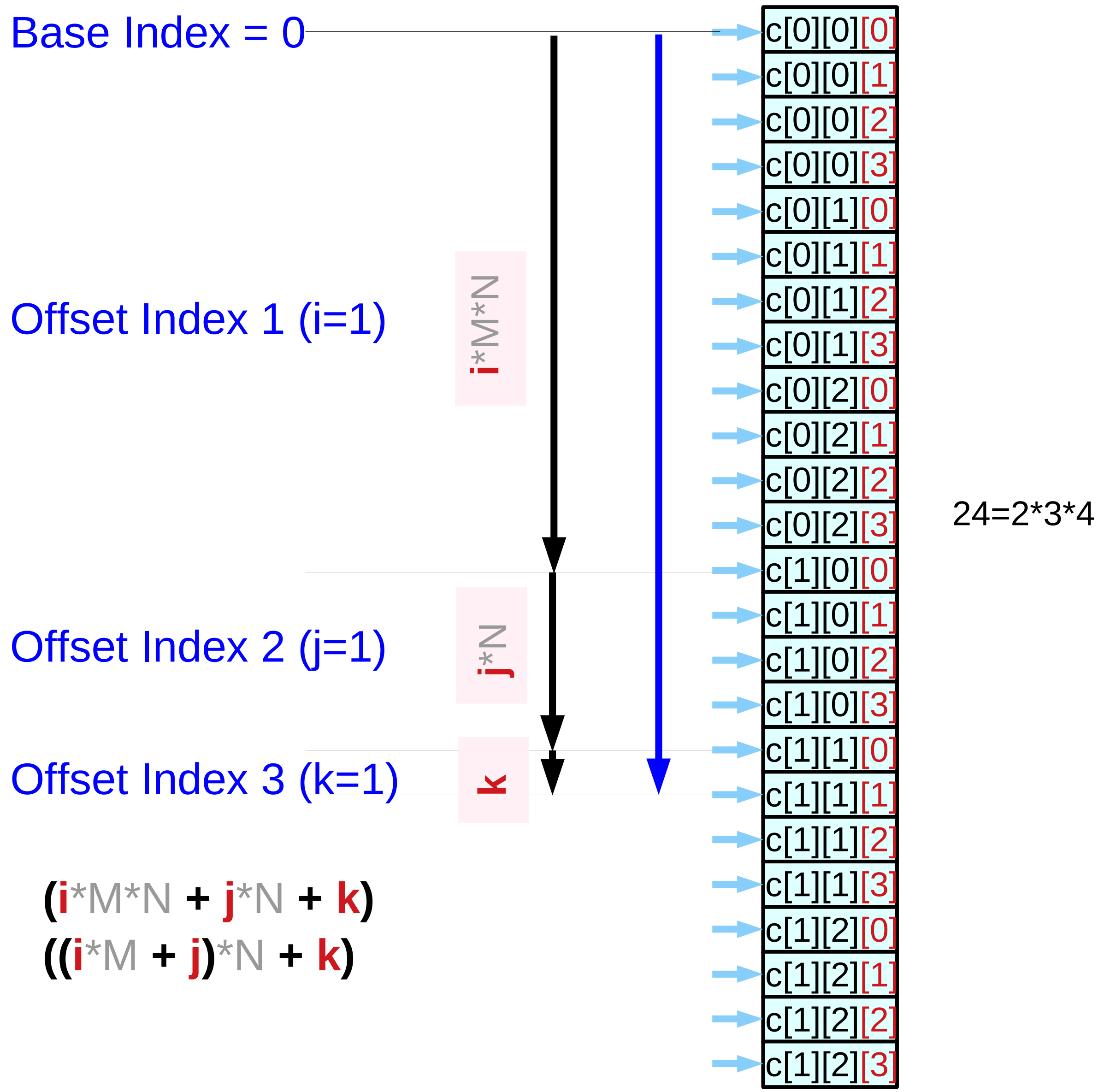
int * c[M] ;

(*(c+m)) ⟺ c[m] or
(*(c+m))[n] ⟺ c[m][n]

Contiguous linear layout

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

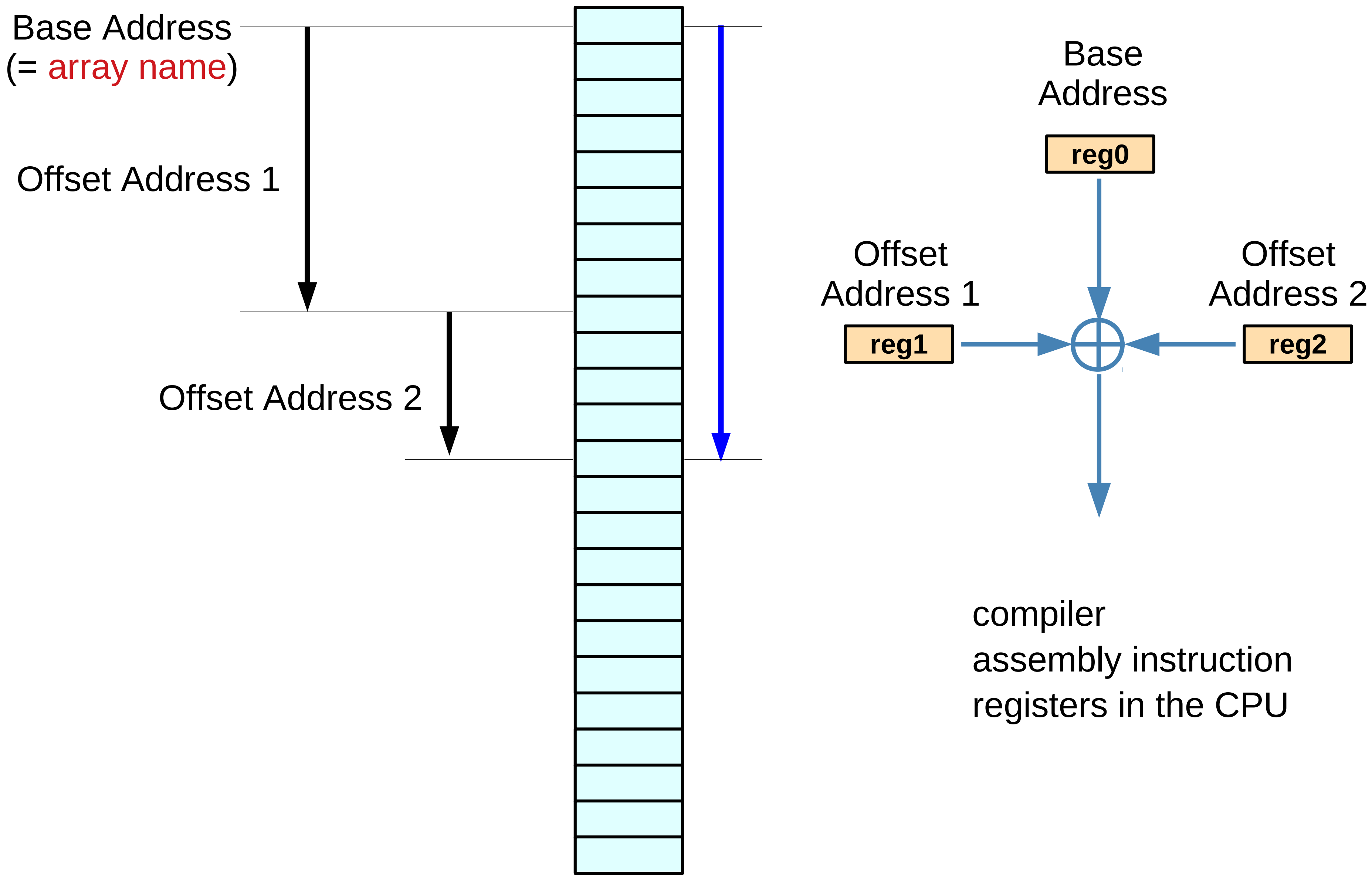
L	M	N
i	j	k
i*M*N	j*N	k



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

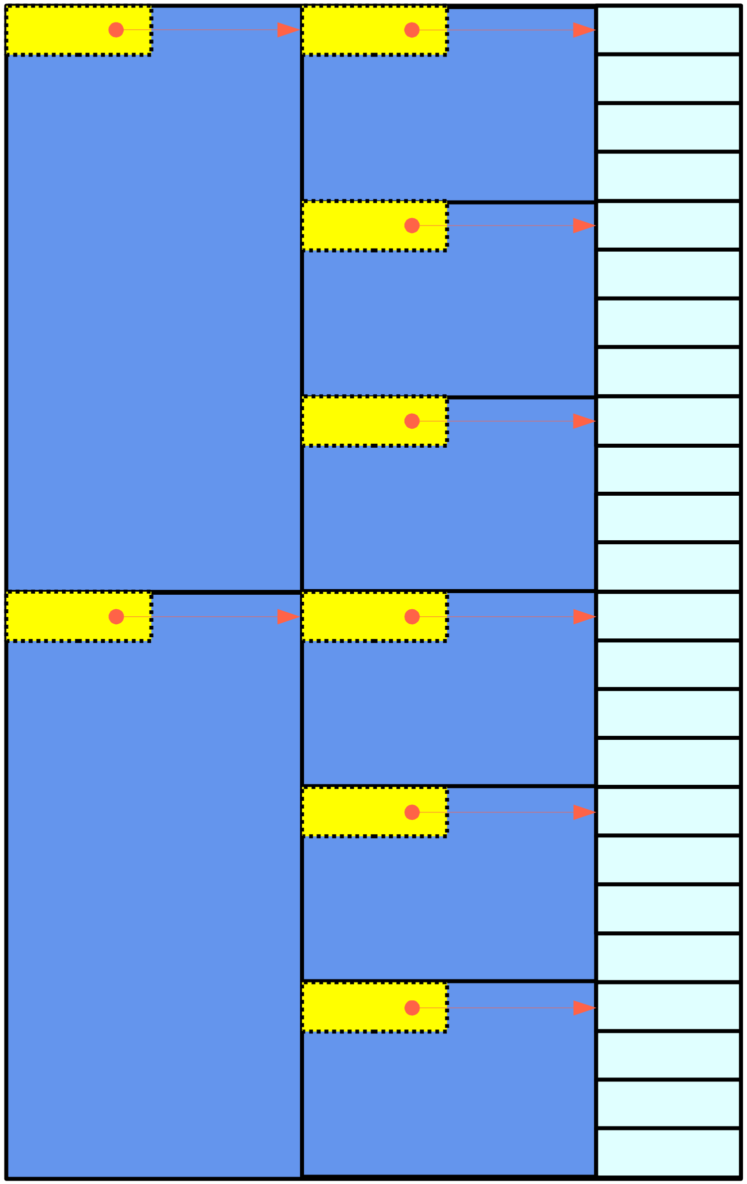
((i*M + j)*N + k)

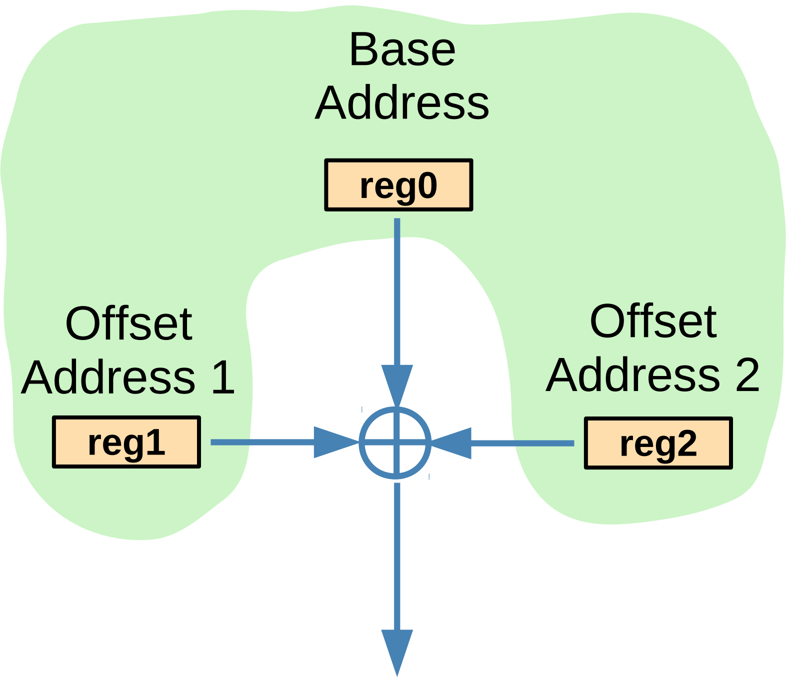
Base and Offset Addressing



compiler
assembly instruction
registers in the CPU

Array Pointer Approach





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

Array Pointer Approach
(pointer to arrays)

References

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