

# Applications of Pointers (1A)

---

Copyright (c) 2010 - 2018 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Please send corrections (or suggestions) to [youngwlim@hotmail.com](mailto:youngwlim@hotmail.com).

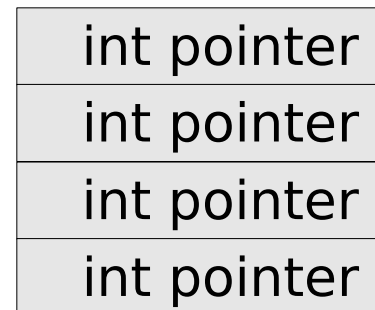
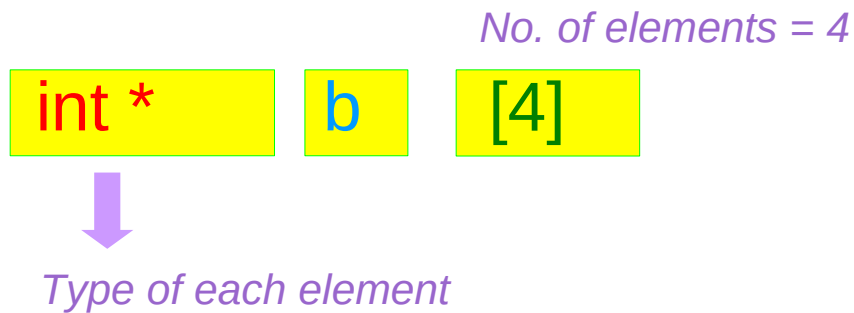
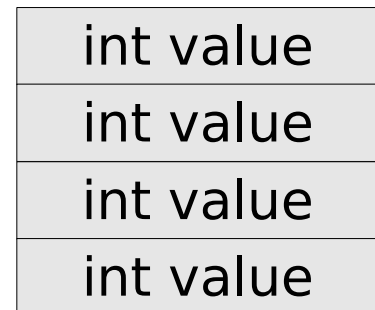
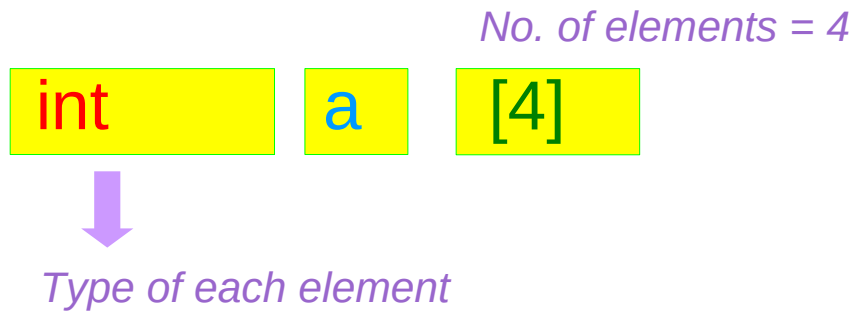
This document was produced by using LibreOffice.

---

# Array of Pointers

# Array of Pointers

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

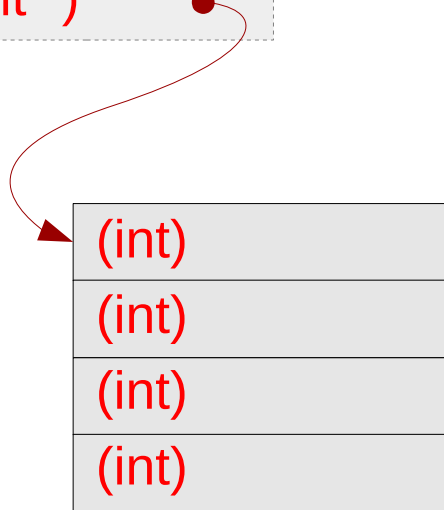


# Array of Pointers – a type view

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

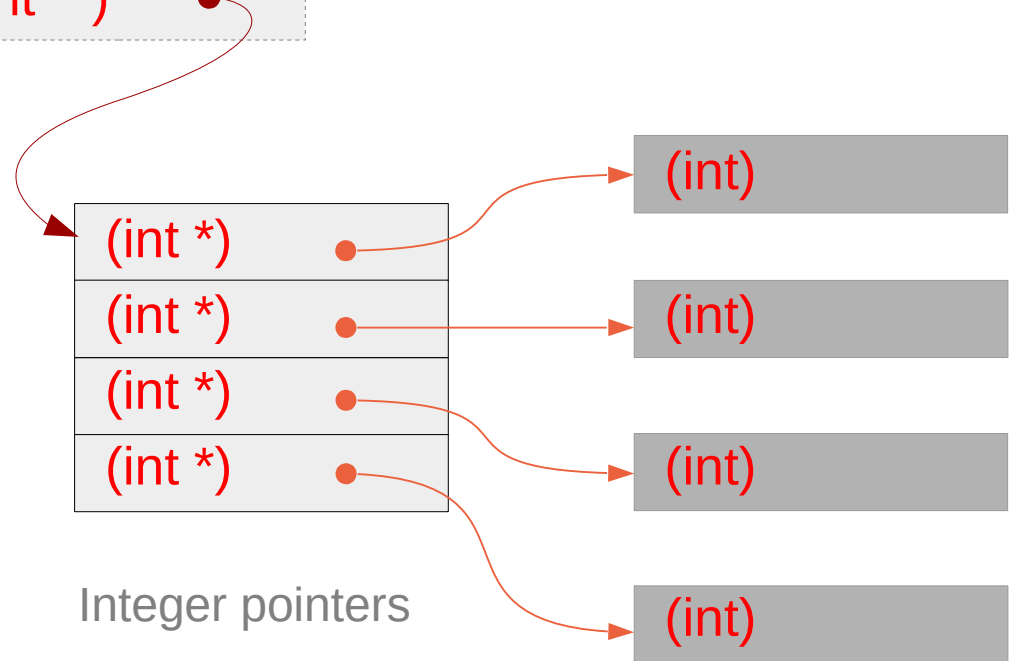
```
int * b [4];
```

(int \*)



Integers

(int \*\*)



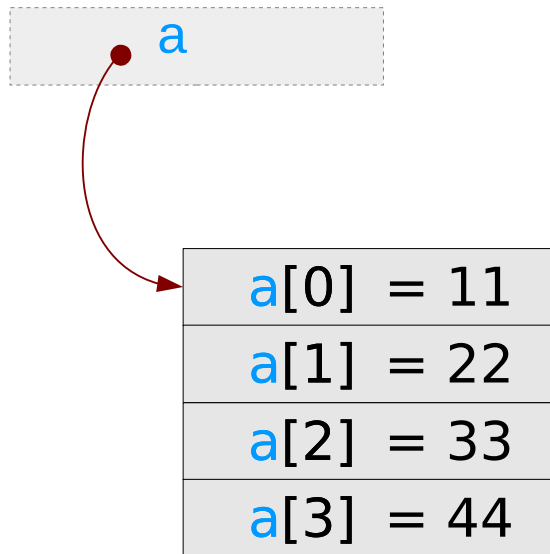
Integer pointers

taking actual  
memory locations

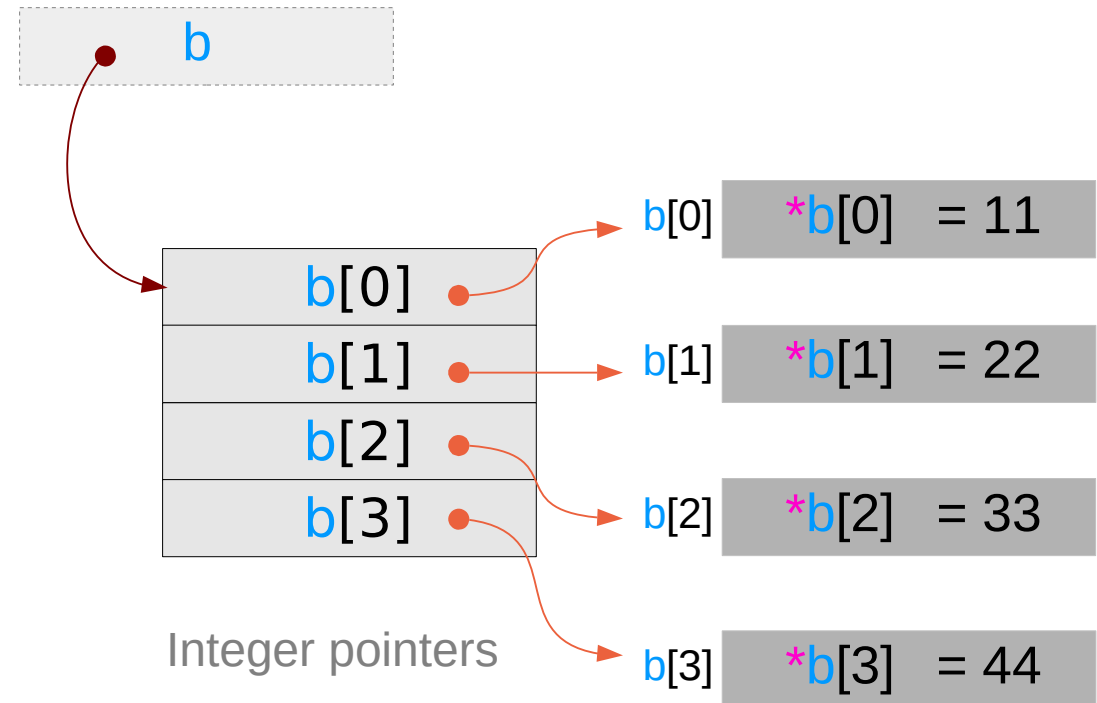
# Array of Pointers – a variable view

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

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

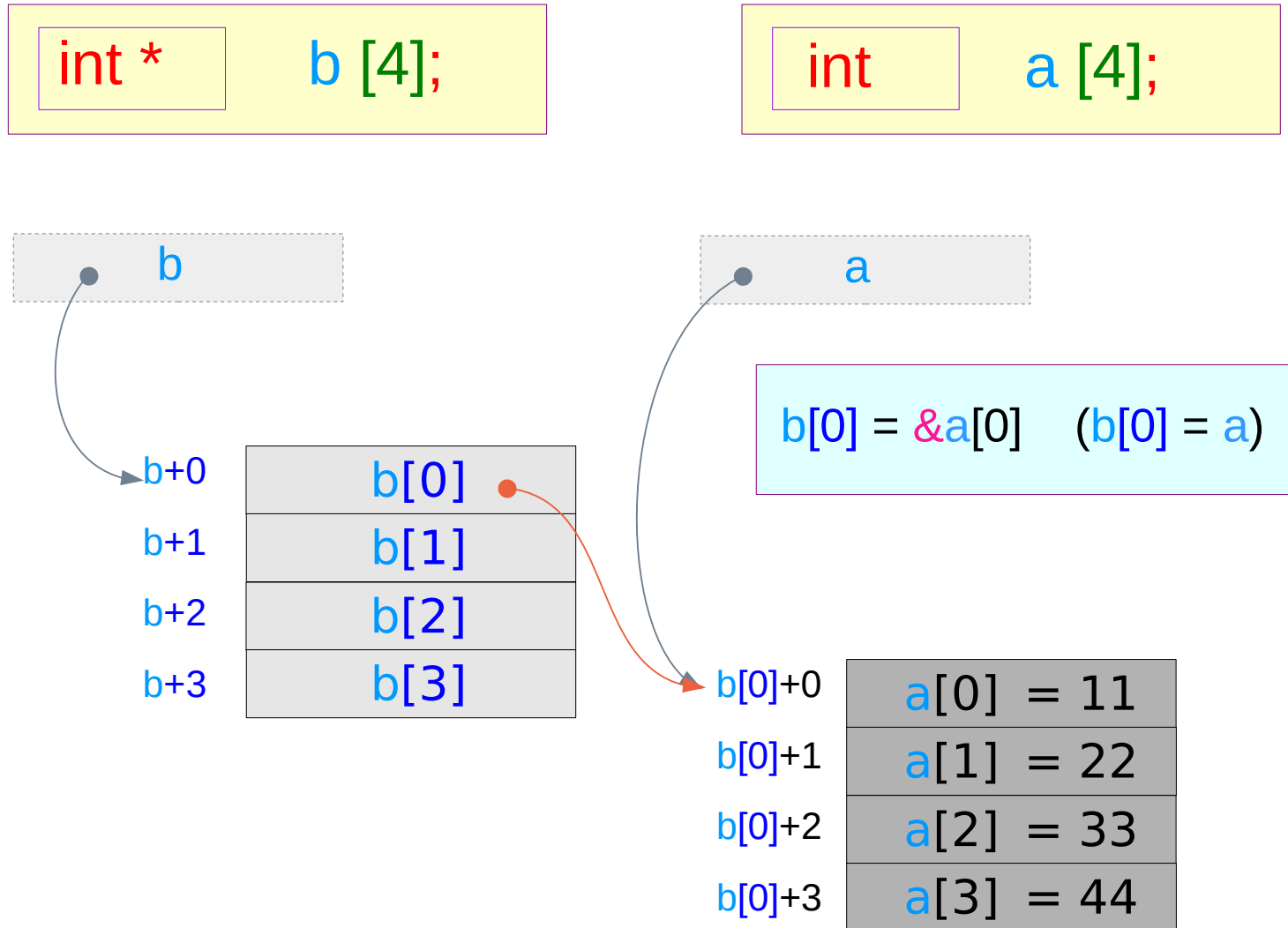


Integers



Integer pointers

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



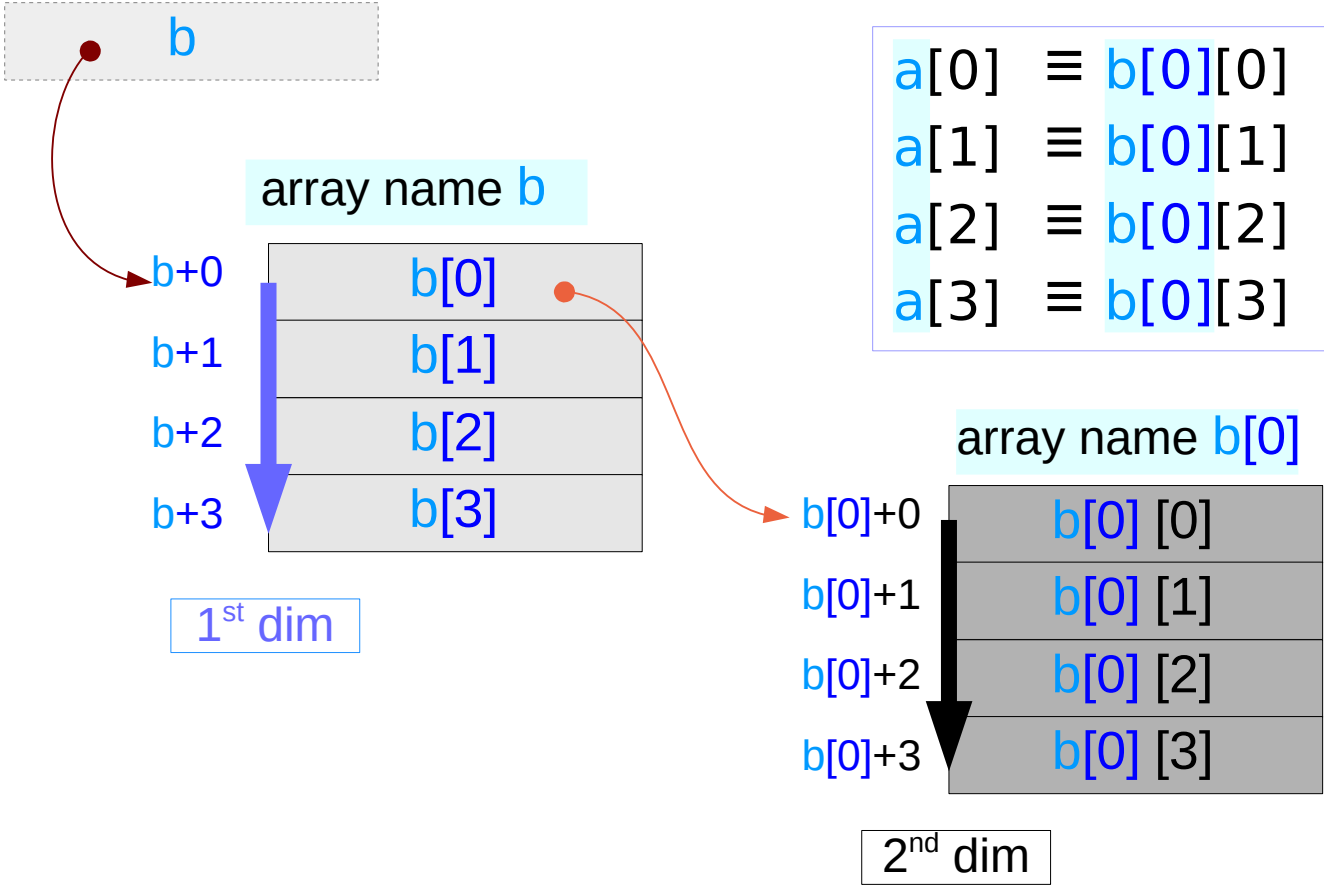
# Array of Pointers – an extended dimension

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

```
assignment b[0] = a
```

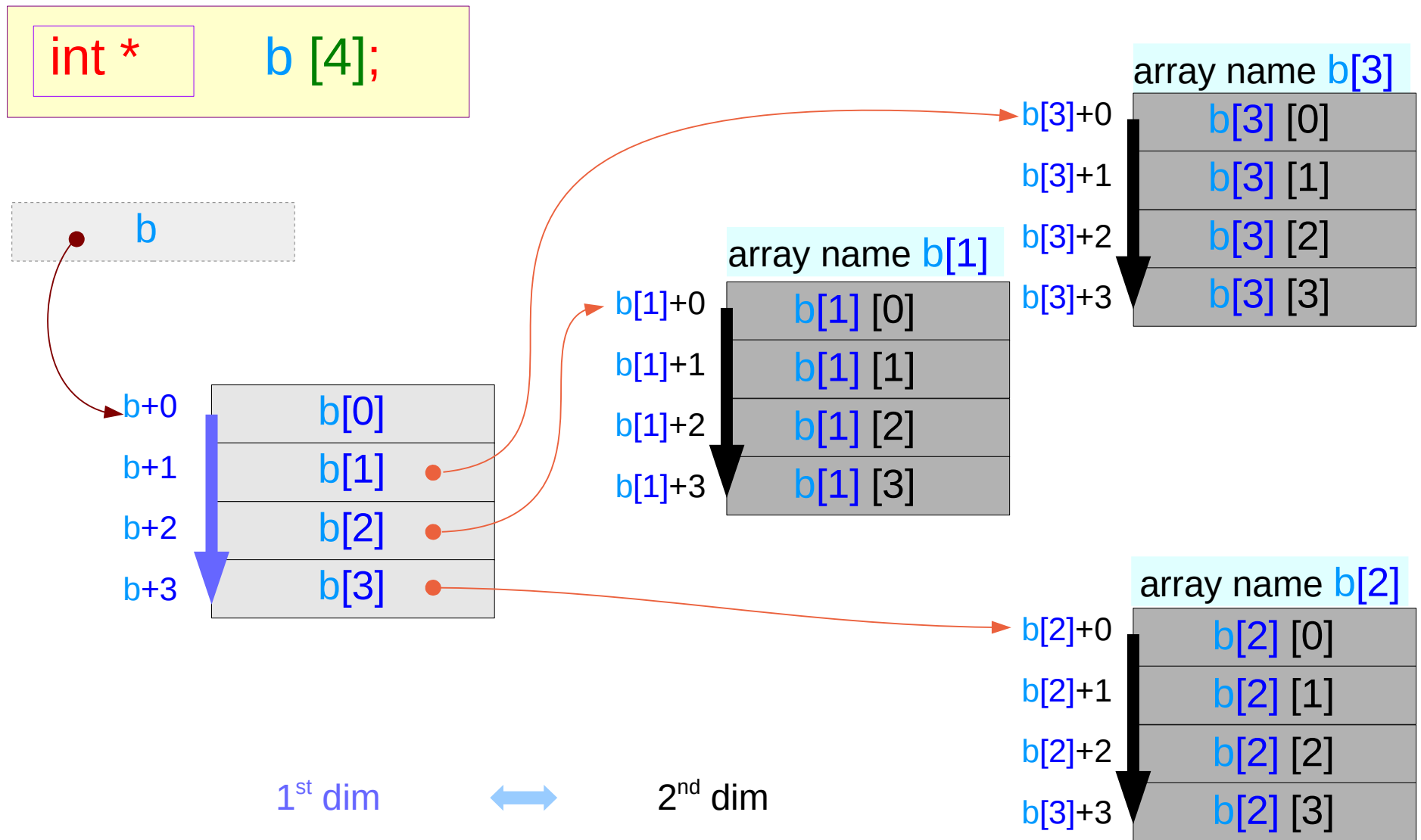
equivalence

$$\begin{aligned} a[0] &\equiv b[0][0] \equiv *((*(b+0)+0)) \\ a[1] &\equiv b[0][1] \equiv *((*(b+0)+1)) \\ a[2] &\equiv b[0][2] \equiv *((*(b+0)+2)) \\ a[3] &\equiv b[0][3] \equiv *((*(b+0)+3)) \end{aligned}$$





# Array of Pointers – assigning other 1-d array names



## 2-d access of a 1-d array – using a pointer array

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

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

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



2-d access of a 1-d array

$$b[i][j] \equiv *((*(b+i)+j))$$
$$a[i*4+j] \equiv *(a + i*4+j)$$

1-d access of a 1-d array

## 3-d access of a 1-d array – using pointer arrays

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



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

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

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

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

```
for (i=0; i<4; ++i)  
    c[i] = &b[i*4];  
  
for (i=0; i<4*4; ++i)  
    b[i] = &a[i*4]
```



3-d access of a 1-d array

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

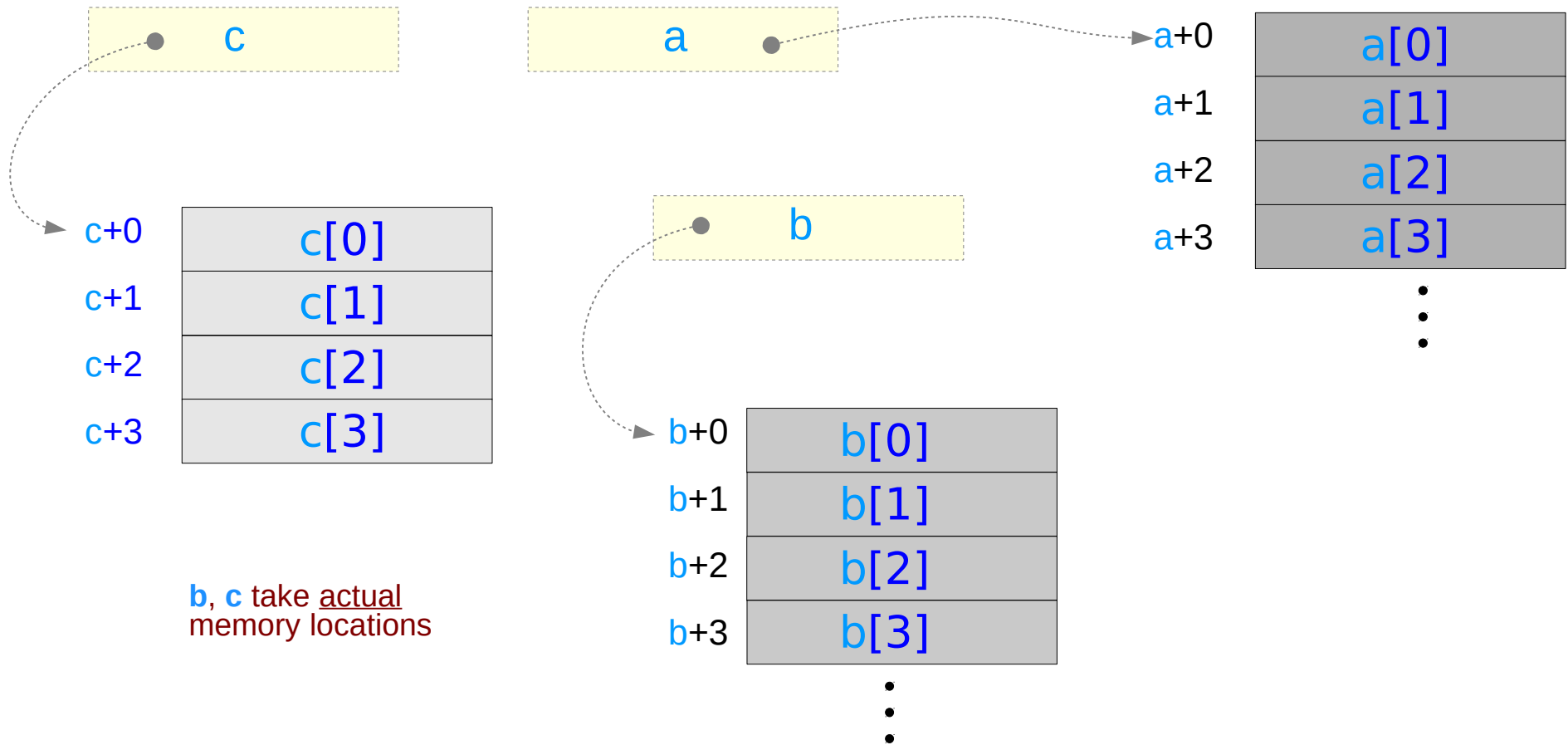
1-d access of a 1-d array

Initialization of pointer arrays **b** and **c**

# 3-d Array – using pointer arrays **b**, **c**

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

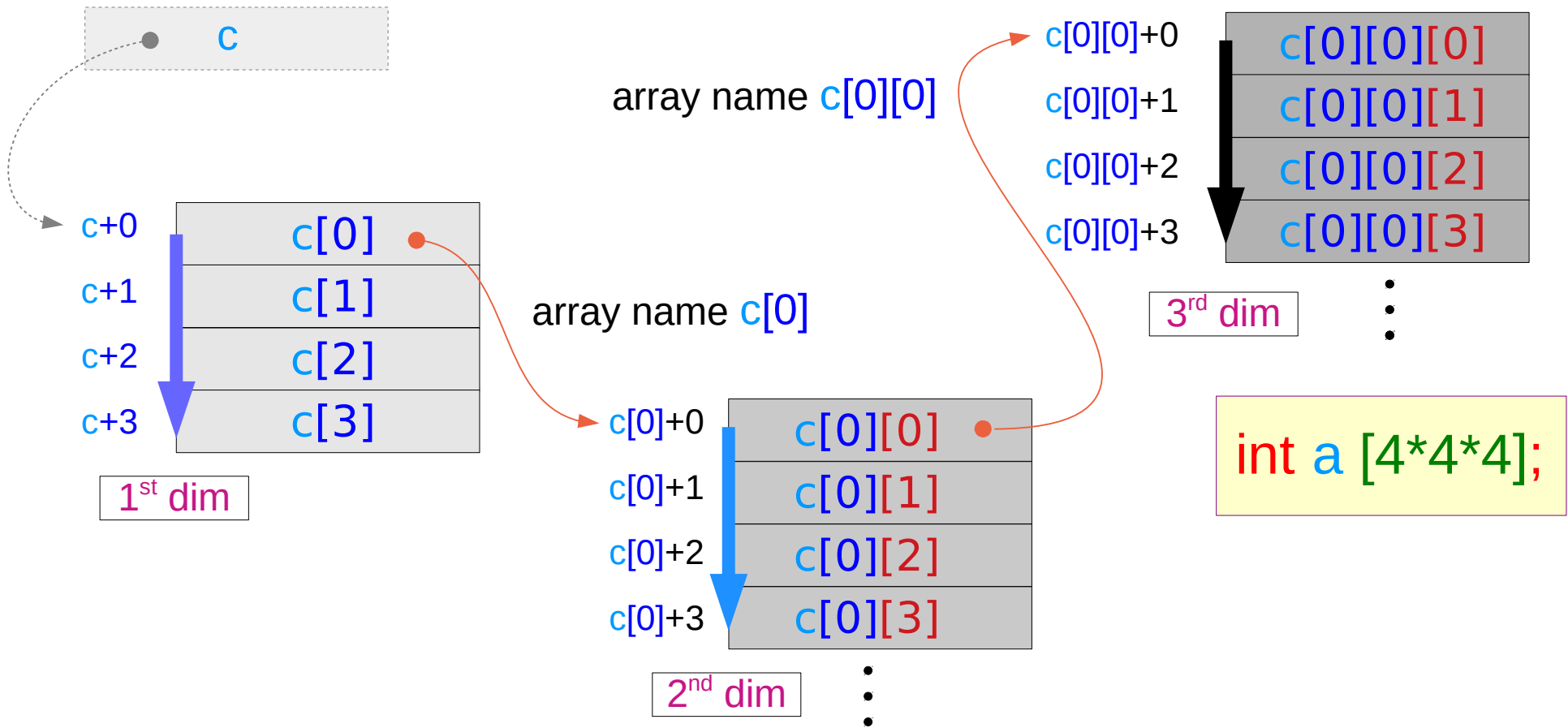
```
int      a [4*4*4];
```



# 3-d Array – pointer arrays extend dimensions

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

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



# Using recursive pointers and brackets

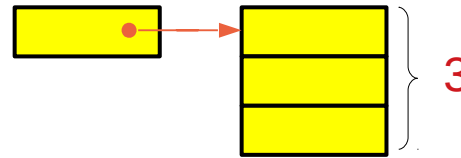
$c[i][j][k]$	$\rightarrow$	$*(c[i][j] + k)$	$X[k] = *(X+k)$
$*(c[i][j] + k)$	$\rightarrow$	$*(*(c[i] + j) + k)$	$Y[j] = *(Y+j)$
$*(*(c[i] + j) + k)$	$\rightarrow$	$*(*(*(c + i) + j) + k)$	$Z[i] = *(Z+i)$

$c[i][j][k]$	$\leftrightarrow$	$*(*(*(c+i)+j)+k)$
--------------	-------------------	--------------------

# Initializing two 1-d pointer arrays **b**, **c**

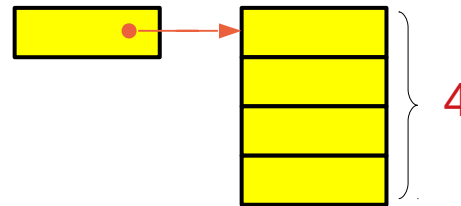
```
int    a [2*3*4];  
int*   b [2*3];  
int**  c [2];
```

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



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

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



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



# Initialization of pointer arrays – a general case

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

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

pointer arrays b, c

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

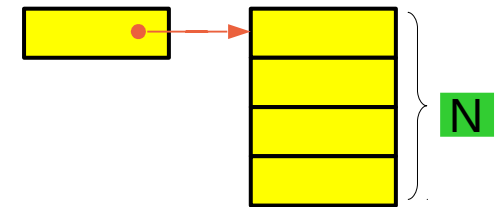
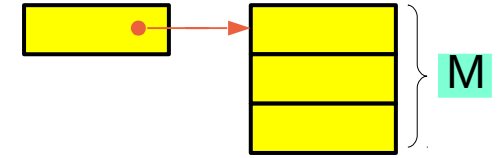
```
for (i=0; i<L; ++i)  
c[i] = &b[i*M];
```

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

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



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



# Accessing the array **a** as a **1-d** array

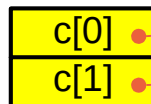
```
int    a [2*3*4];  
int*   b [2*3];  
int**  c [2];
```



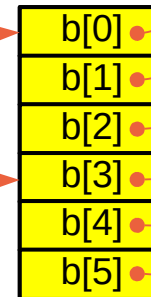
```
int    a [24];
```

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

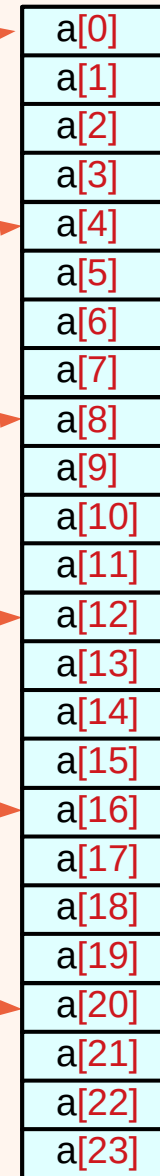
int\*\* c [2];



int\* b [2\*3];



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



24=2\*3\*4

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

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

# Accessing the array **a** as a 2-d array using **b**

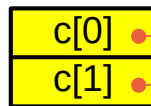
```
int    a [2*3*4];
int*   b [2*3];
int**  c [2];
```



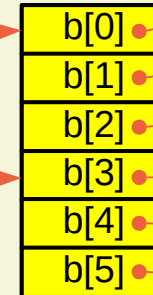
```
int    b [6][4];
```

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

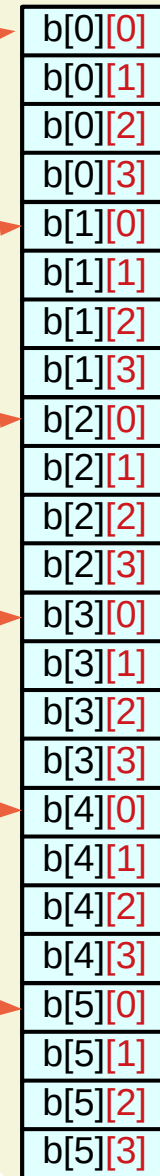
int\*\* c [2];



int\* b [2\*3];



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



24=2\*3\*4

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

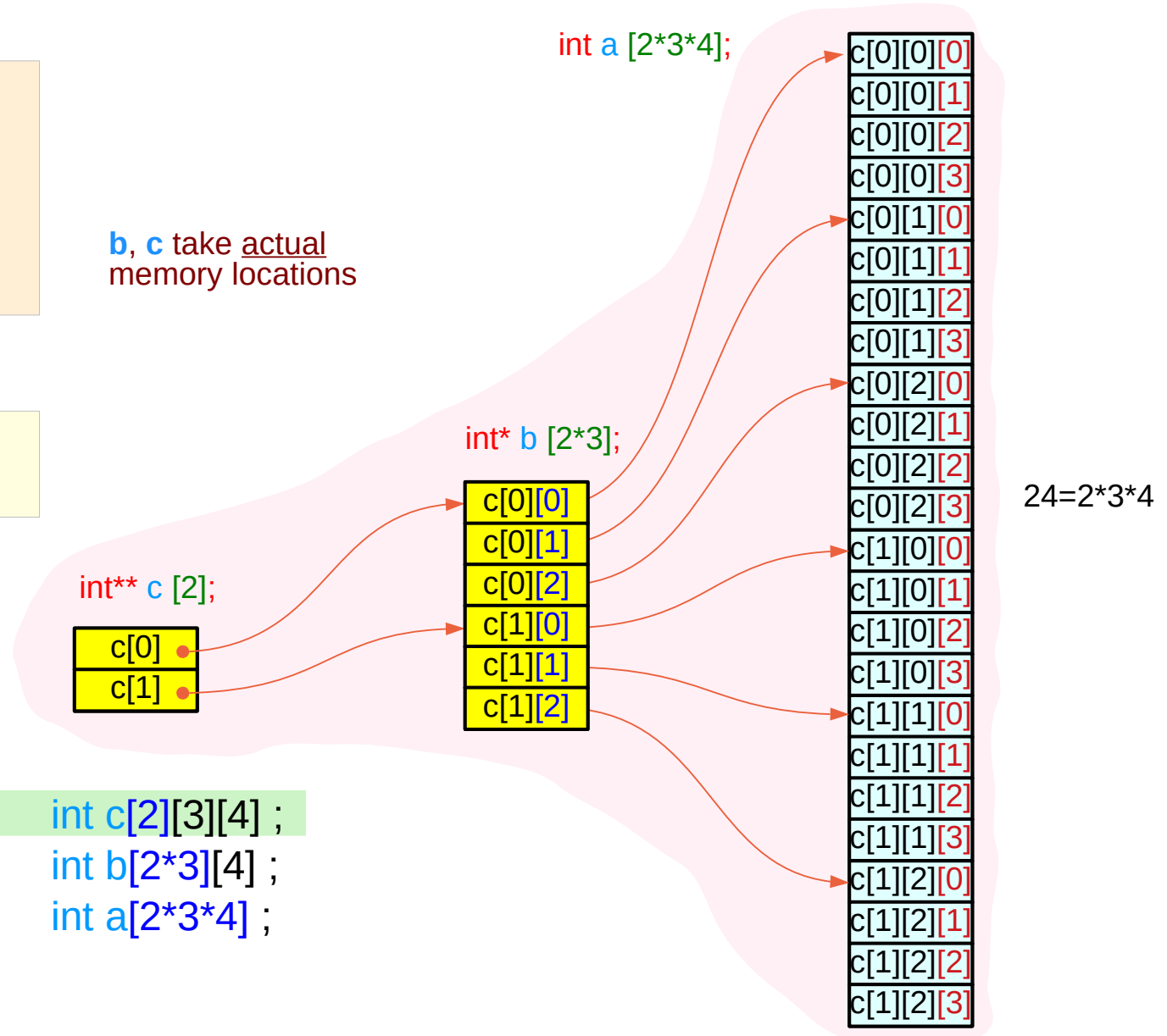
# Accessing the array **a** as a **3-d** array using **c**

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



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

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



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

# Array names of 2-d and 1-d sub-arrays

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



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

`c[0]` array name of a 2-d array [M][N]  
`c[1]` array name of a 2-d array [M][N]

`c[0][0][0] = a[0*M*N]`  
`c[1][0][0] = a[1*M*N]`

starting elements

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

`c[0][0]` array name of a 1-d array [N]  
`c[0][1]` array name of a 1-d array [N]  
`c[0][2]` array name of a 1-d array [N]  
`c[1][0]` array name of a 1-d array [N]  
`c[1][1]` array name of a 1-d array [N]  
`c[1][2]` array name of a 1-d array [N]

`c[0][0][0] = a[(0*M+0)*N]`  
`c[0][1][0] = a[(0*M+1)*N]`  
`c[0][2][0] = a[(0*M+2)*N]`  
`c[1][0][0] = a[(1*M+0)*N]`  
`c[1][1][0] = a[(1*M+1)*N]`  
`c[1][2][0] = a[(1*M+2)*N]`

starting elements

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

# Starting element Index

```

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

```

↓

```

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

```

- L=2** {  
   i=0     $i*3*4 = 0$   
   i=1     $i*3*4 = 12$
- M=3** {  
   j=0     $j*4 = 0$   
   j=1     $j*4 = 4$   
   j=2     $j*4 = 8$
- N=4** {  
   k=0     $k*1 = 0$   
   k=1     $k*1 = 1$   
   k=2     $k*1 = 2$   
   k=3     $k*1 = 3$

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

c[0][0][0]	a[0]
c[0][0][1]	a[1]
c[0][0][2]	a[2]
c[0][0][3]	a[3]
c[0][1][0]	a[4]
c[0][1][1]	a[5]
c[0][1][2]	a[6]
c[0][1][3]	a[7]
c[0][2][0]	a[8]
c[0][2][1]	a[9]
c[0][2][2]	a[10]
c[0][2][3]	a[11]
c[1][0][0]	a[12]
c[1][0][1]	a[13]
c[1][0][2]	a[14]
c[1][0][3]	a[15]
c[1][1][0]	a[16]
c[1][1][1]	a[17]
c[1][1][2]	a[18]
c[1][1][3]	a[19]
c[1][2][0]	a[20]
c[1][2][1]	a[21]
c[1][2][2]	a[22]
c[1][2][3]	a[23]

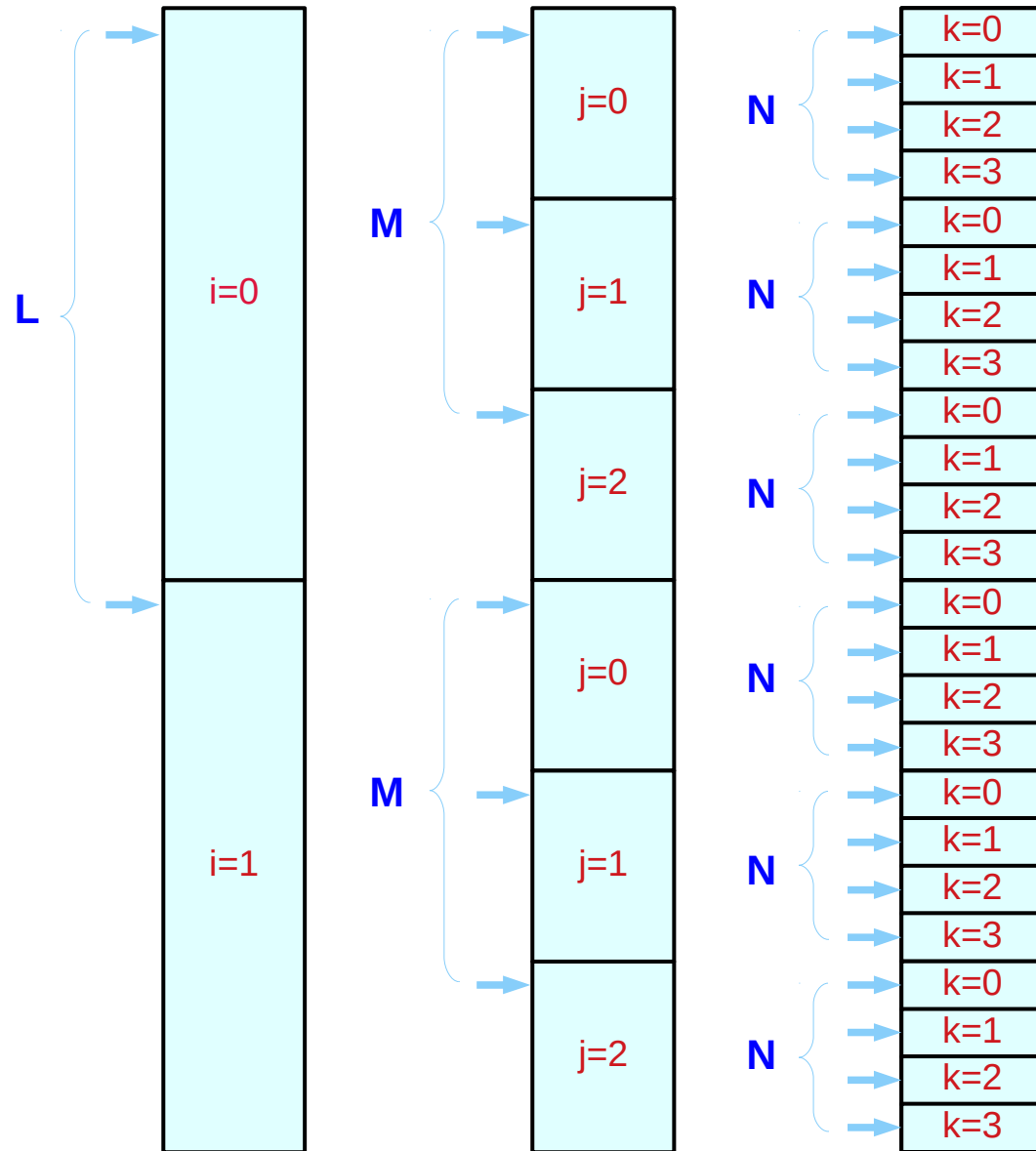
# L, M, N – the number of index values

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



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

L	M	N
<b>i</b> [0..L-1]	<b>j</b> [0..M-1]	<b>k</b> [0..N-1]



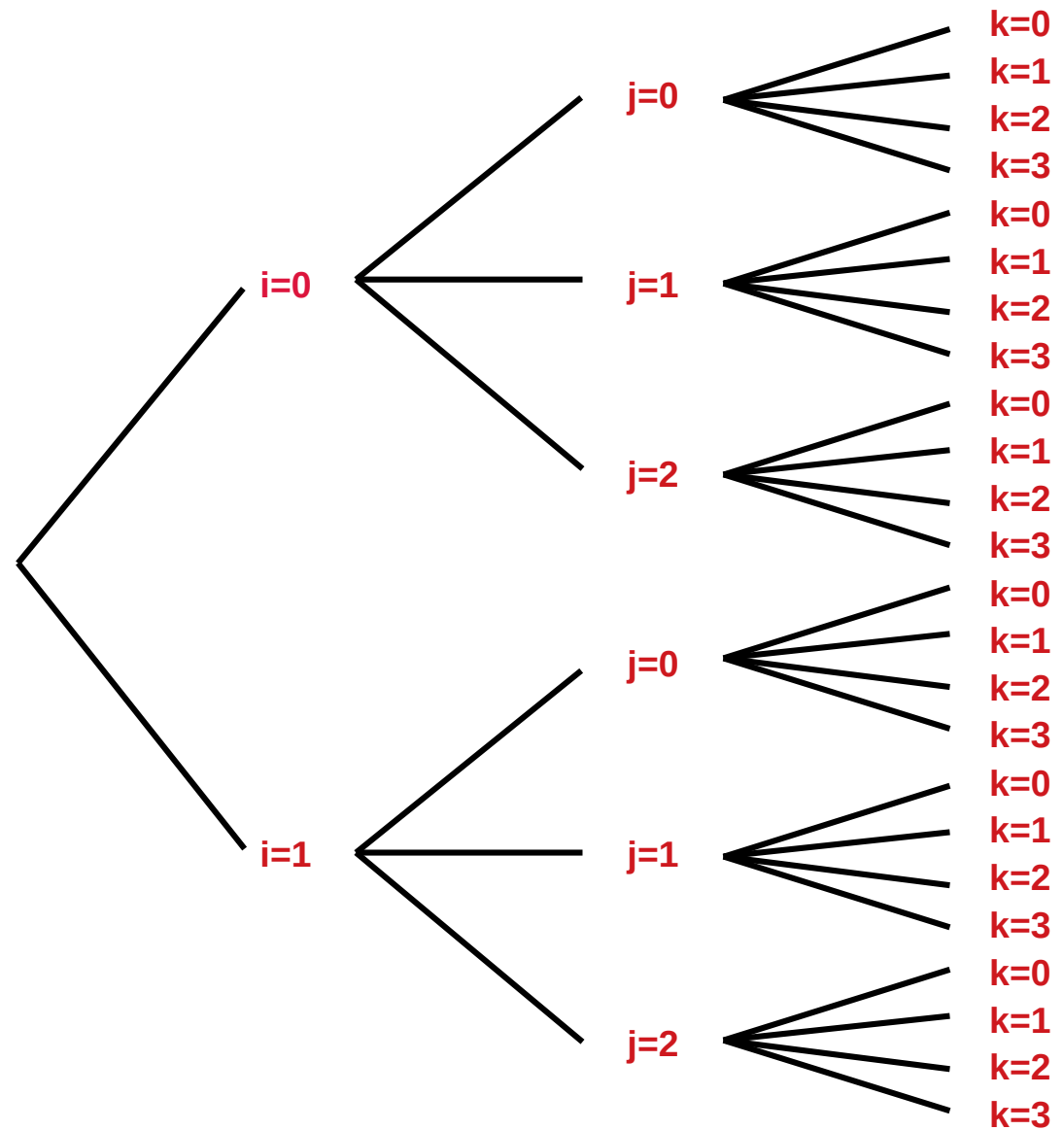
# L, M, N – index value tree

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



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

L	M	N
<b>i</b> [0..L-1]	<b>j</b> [0..M-1]	<b>k</b> [0..N-1]





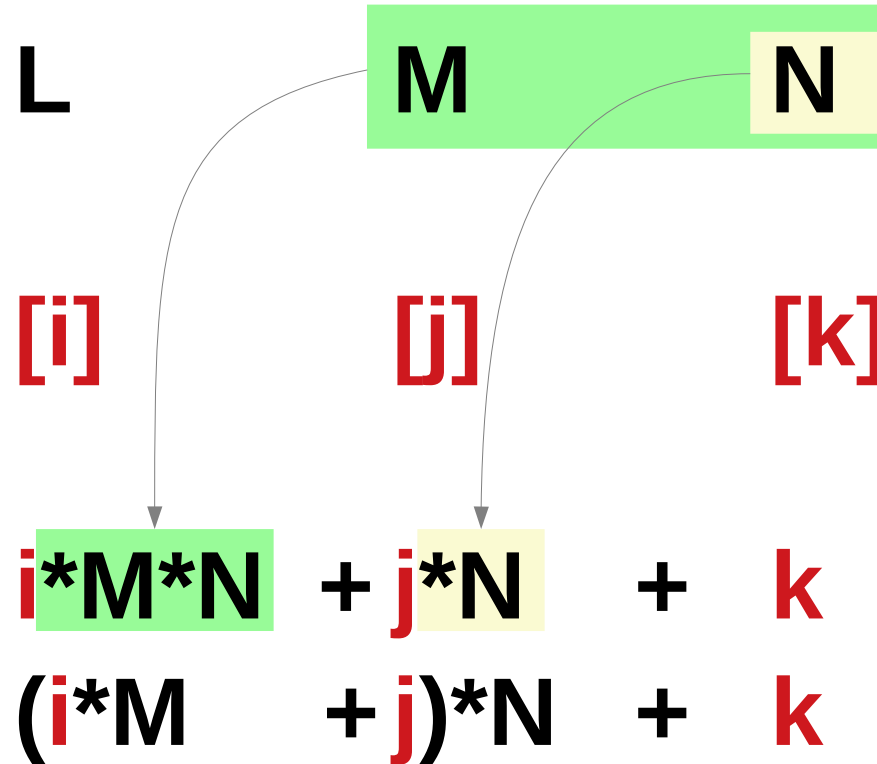
# Converting a 3-d index into a 1-d index

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



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

L	M	N
<i>i</i>	<i>j</i>	<i>k</i>
[0..L-1]	[0..M-1]	[0..N-1]
$i*M*N$	$j*N$	<i>k</i>



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

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



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

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

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

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

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

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

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

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

```
= *((&b[i*M]+j)+k)
```

```
= *(b[i*M+j]+k)
```

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

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

```
← c[i] = &b[i*M]
```

```
← *((b+i*M+j)+k)
```

```
← b[m] = &a[m*N]
```

```
← *(a+(i*M+j)*N+k)
```

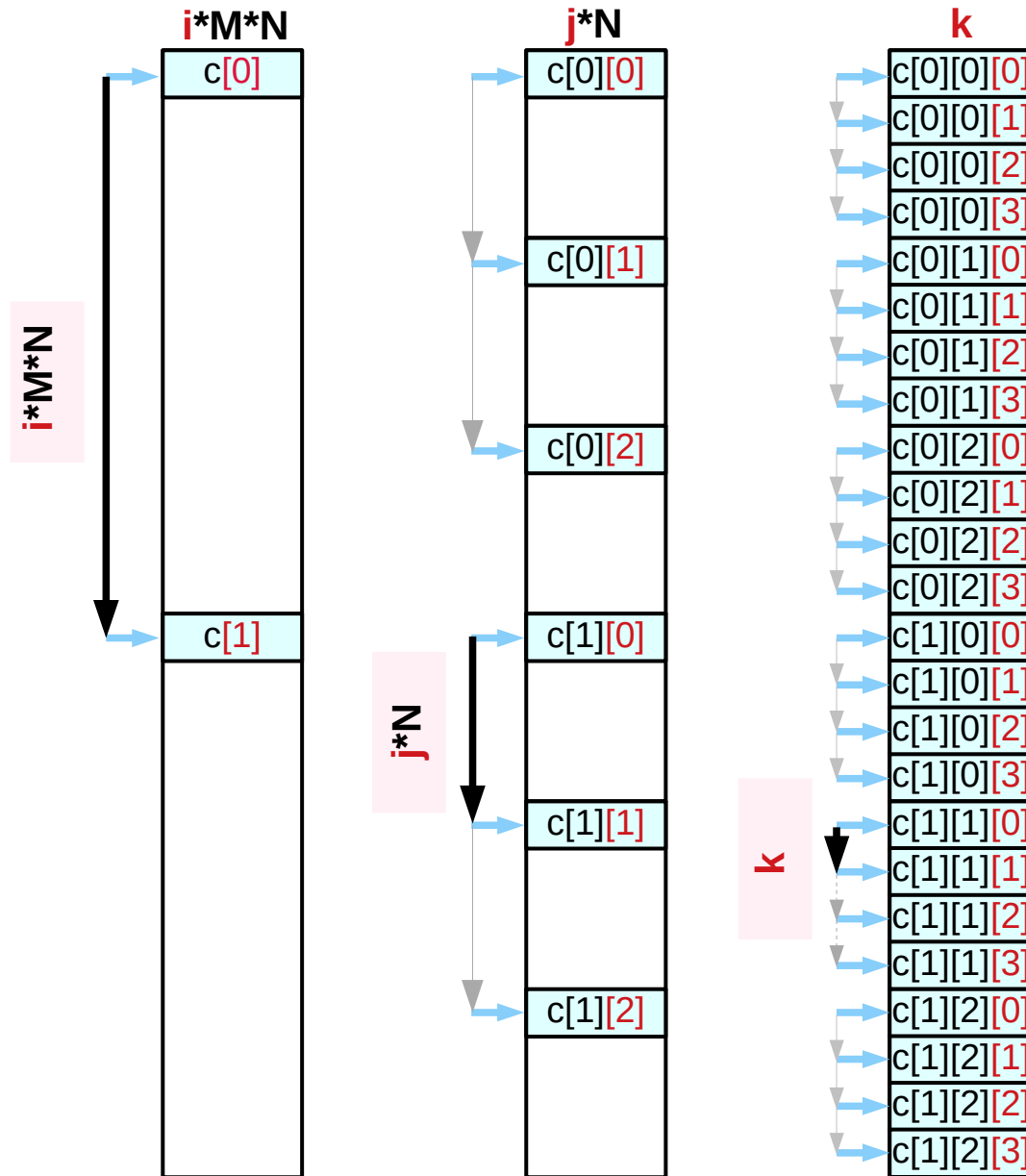
# $i * M * N$ , $j * N$ , $k$ – index offset values

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



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

L	M	N
$i$	$j$	$k$
$[0..L-1]$	$[0..M-1]$	$[0..N-1]$
$i * M * N$	$j * N$	$k$



24=2\*3\*4

# Accessing `a` by base and offset indices

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



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

L	M	N
<i>i</i>	<i>j</i>	<i>k</i>
[0..L-1]	[0..M-1]	[0..N-1]
$i*M*N$	$j*N$	<i>k</i>

Base Index = 0

Offset Index 1

Offset Index 2

Offset Index 3

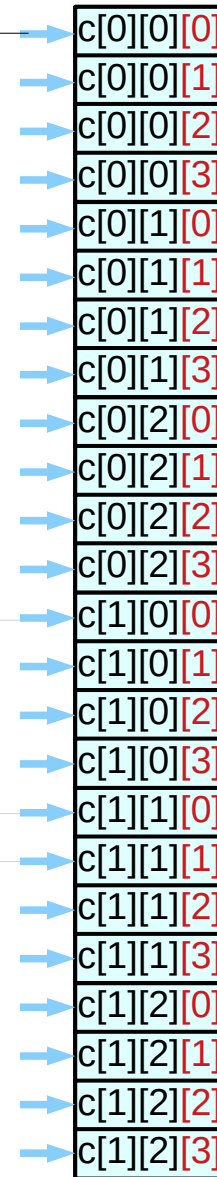
$i*M*N$

$j*N$

*k*

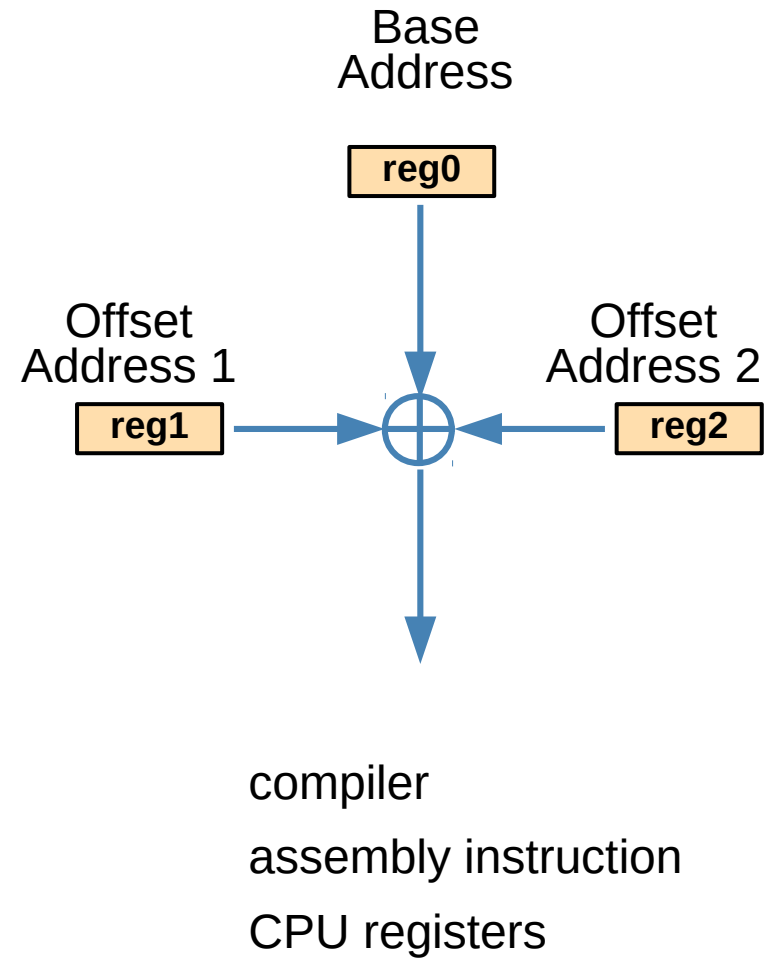
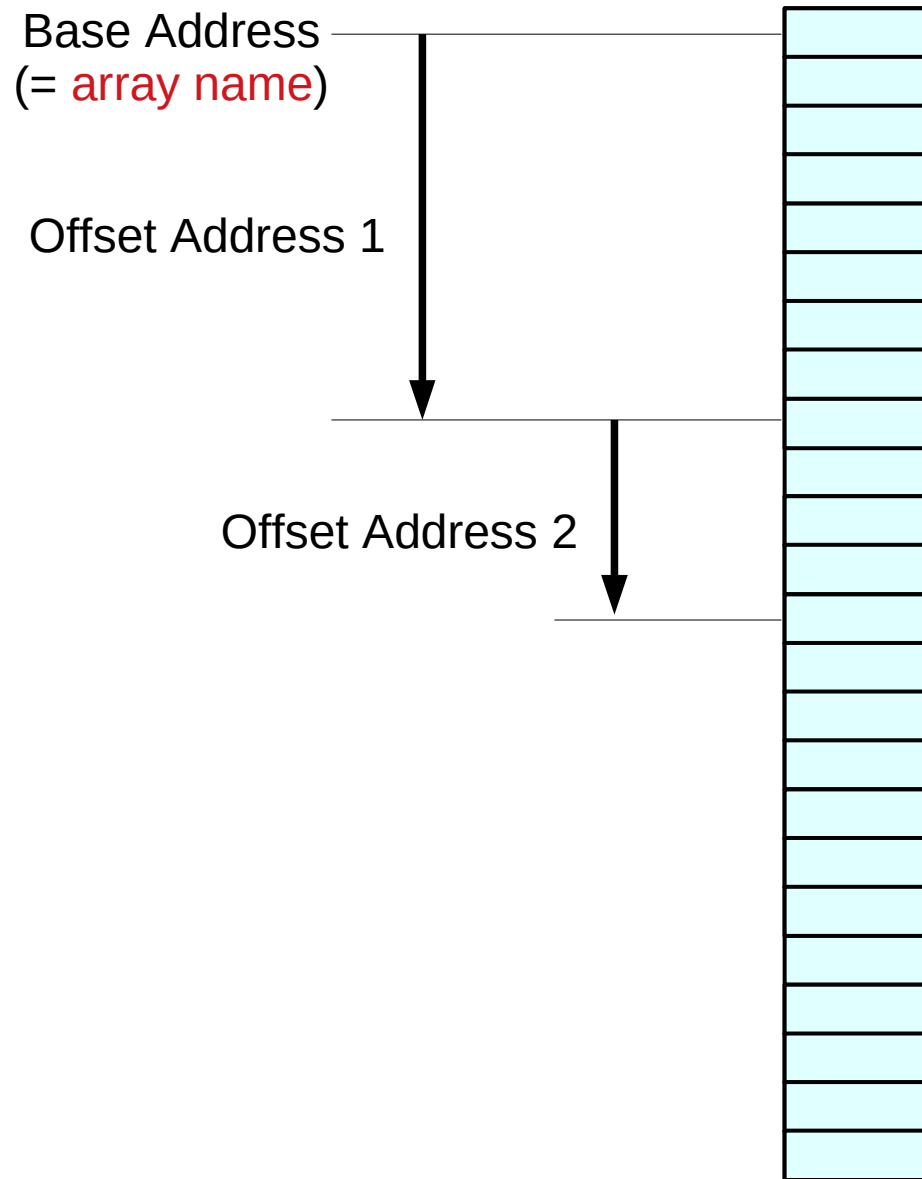
$$(i*M*N + j*N + k)$$

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



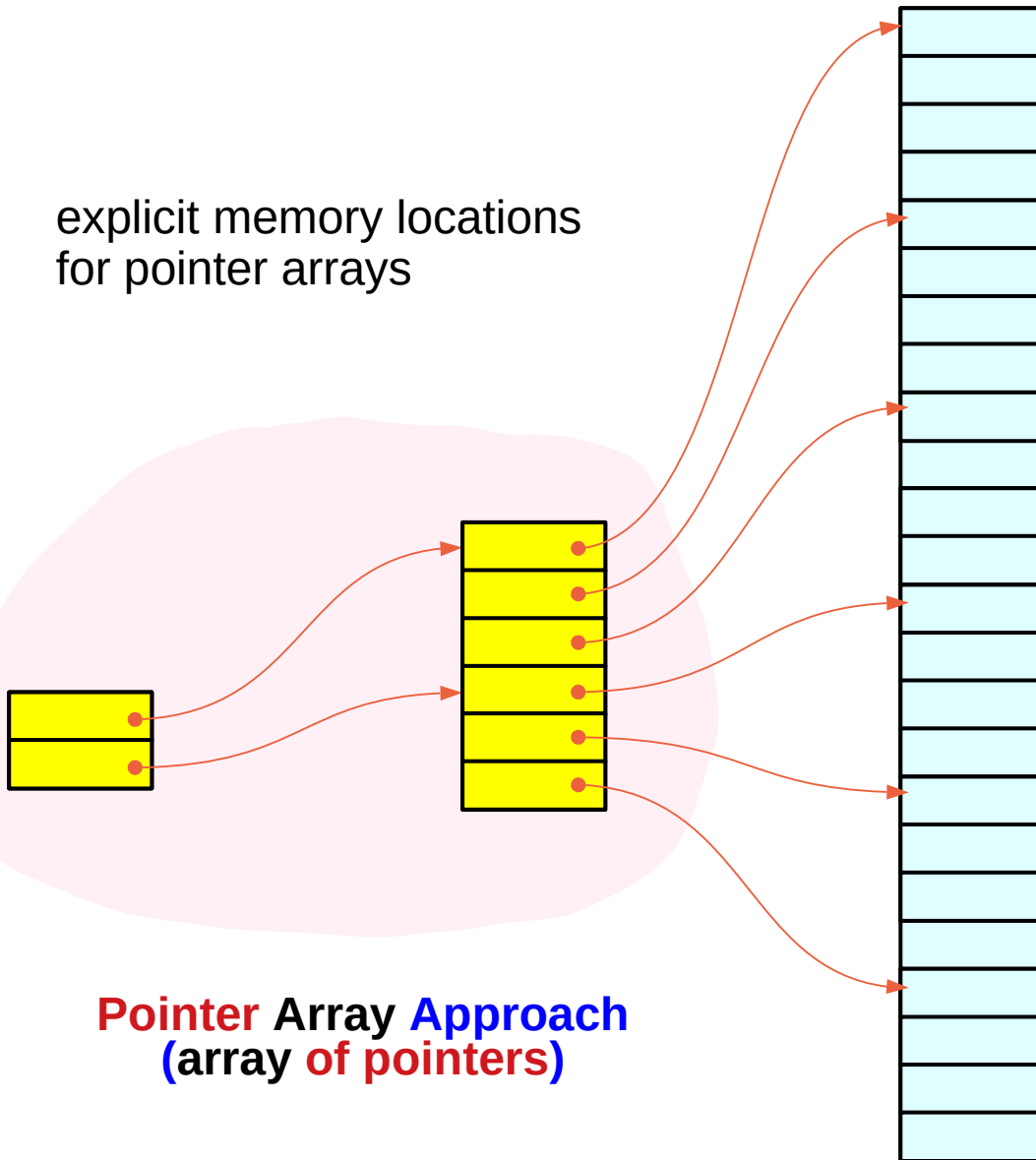
$$24 = 2 * 3 * 4$$

# Base and Offset Addressing

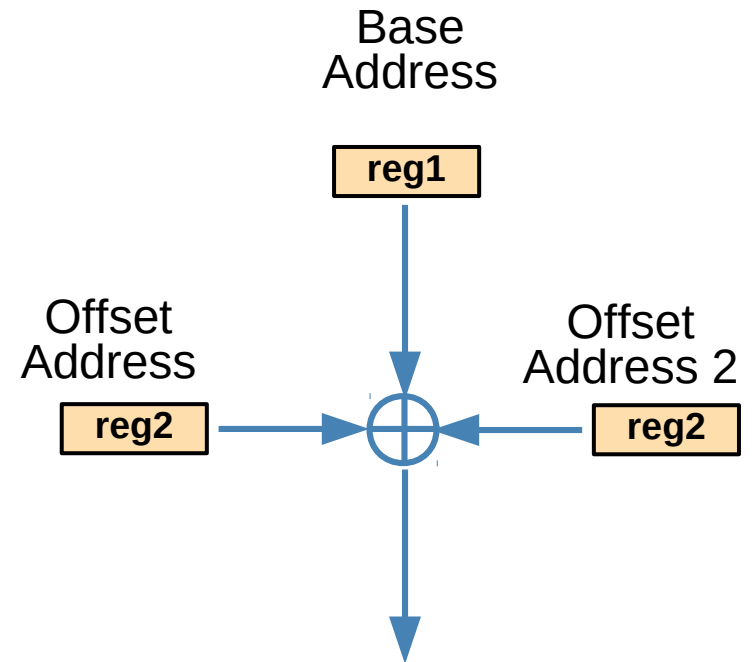


# Pointer Array vs. Array Pointer

explicit memory locations  
for pointer arrays



**Pointer Array Approach**  
(array of pointers)



CPU register based address  
**computations** eliminate the  
pointer arrays

**Array Pointer Approach**  
(pointer to arrays)

# Pointer to an array – variable declarations

```
int m ;
```

```
int *n ;
```

an integer pointer

Array **Pointer Approach**  
(**pointer to arrays**)

```
int a [4]
```

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

an array pointer

```
int func (int a, int b) ;
```

```
int (*fp) (int a, int b) ;
```

a function pointer

# Pointer to an array – a type view

**int** 4 byte data

**int \***

an integer pointer

array pointer:  
a pointer to an array

pointer array:  
an array of pointers

**int [4]** 4\*4 byte data

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

an array pointer

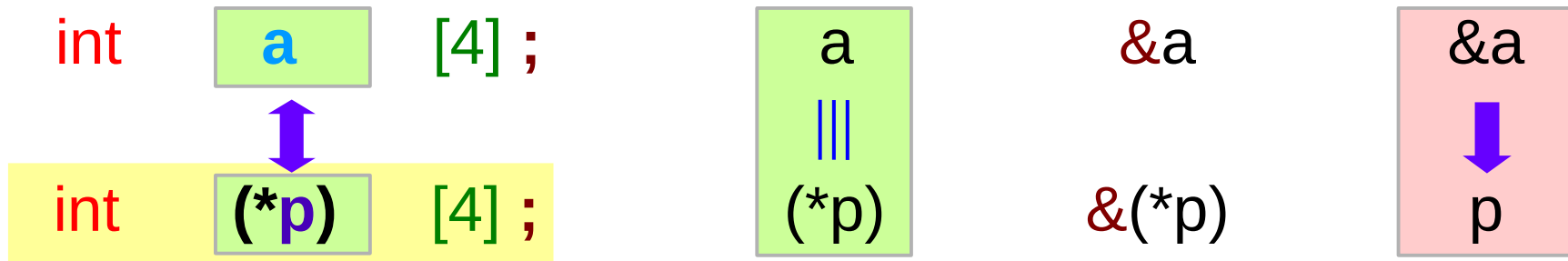
**int (int, int)** instructions

**int (\*) (int, int)**

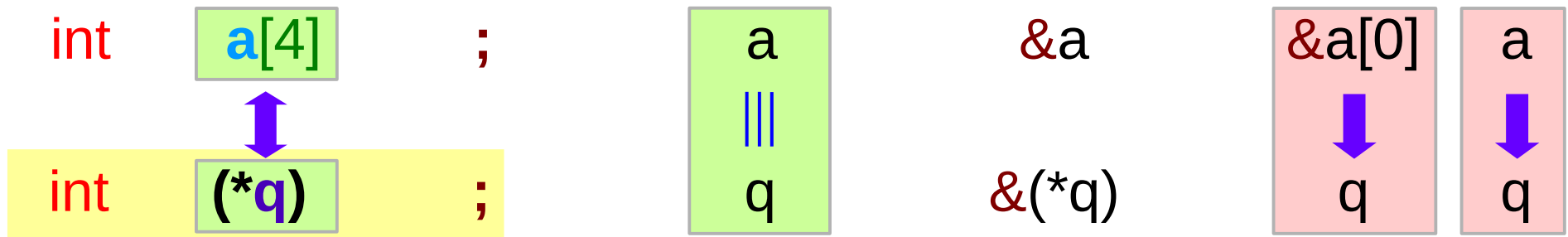
a function pointer



# Pointer to an array : assignment and equivalence

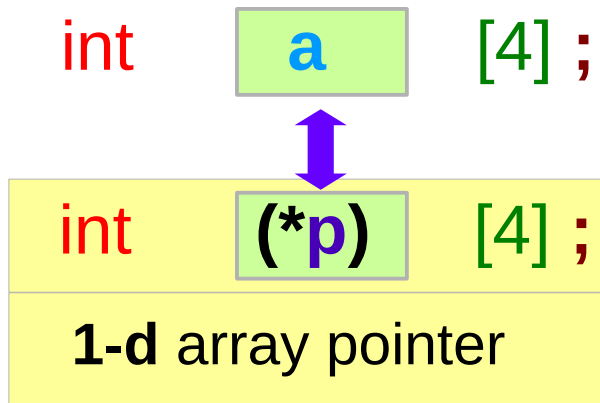


1-d array pointer



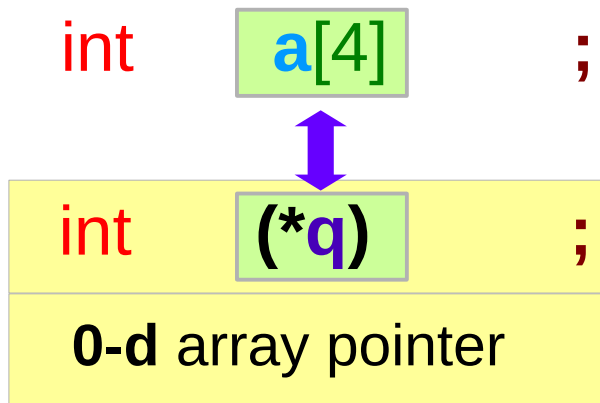
0-d array pointer (= int pointer)

# Pointer to an array : size of array



`p = &a;`

`sizeof(p)` = 8 bytes : the size of a pointer  
`sizeof(*p)` = 4\*4 bytes : the whole size of the pointed 1-d array



`q = a;`

`sizeof(q)` = 8 bytes : the size of a pointer  
`sizeof(*q)` = 4 bytes : the whole size of the pointed 0-d array

# Pointer to an array – a variable view (1)

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

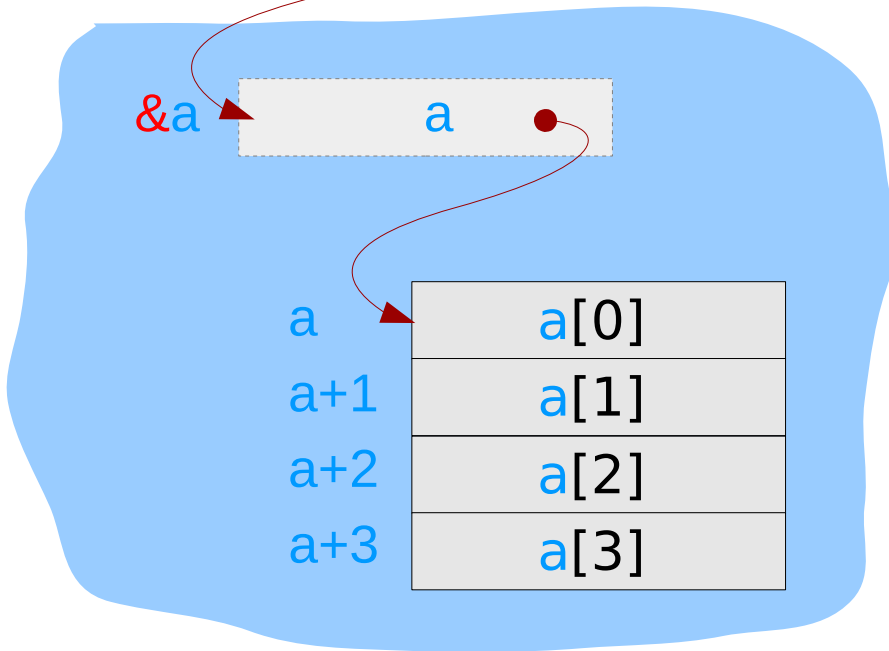
assignment  
 $p = \&a$

equivalence  
 $*p \equiv a$

p

1-d array pointer

points to a 1-d array –  
a aggregated type data



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

$p : \text{int } (*) [4]$  type

# Pointer to an array – a variable view (2)

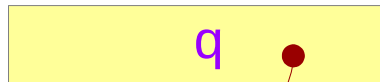
```
int (*q);
```

assignment

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

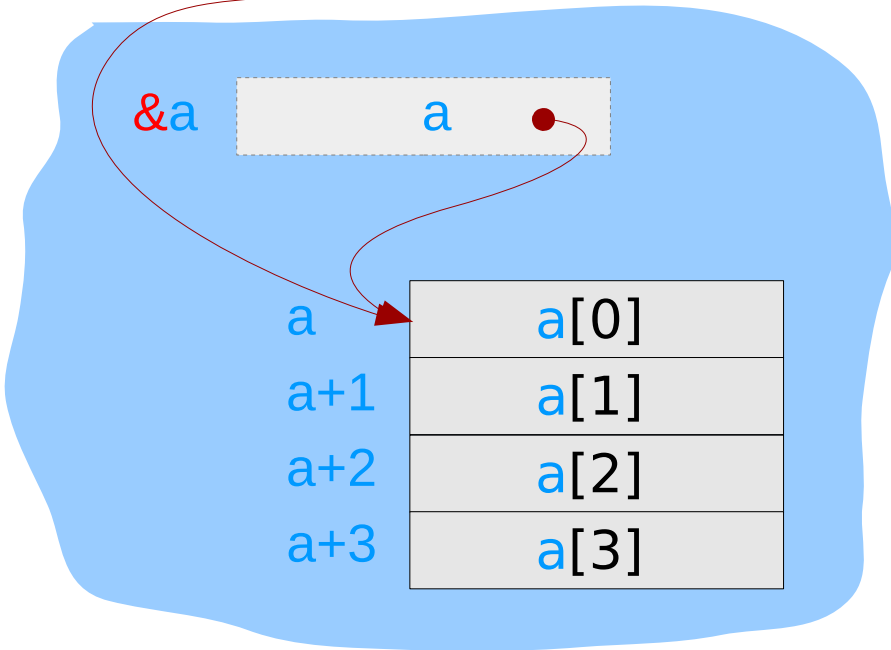
equivalence

```
*q ≡ *a  
q ≡ a
```



0-d array pointer

points to an array element –  
an integer type data



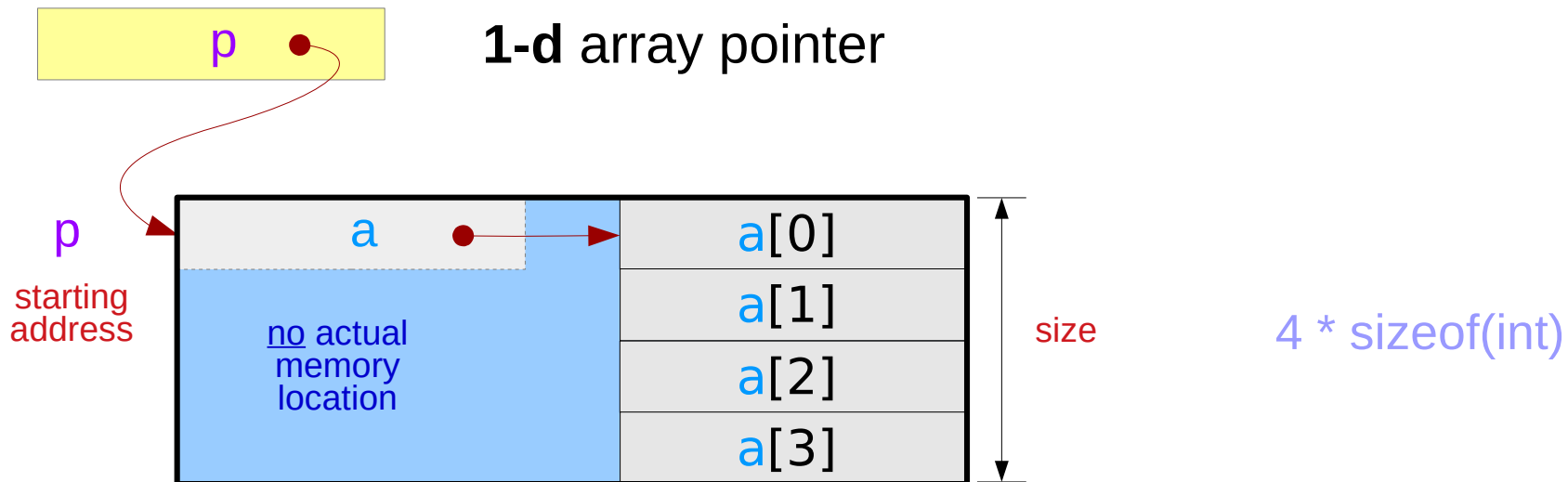
```
int a[4];
```

$q : \text{int} (*) = \text{int} * \text{type}$

# Pointer to a 1-d array – an aggregated type view

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

- An aggregated type
- starting address (&a)
  - size of all the array elements (16 bytes)

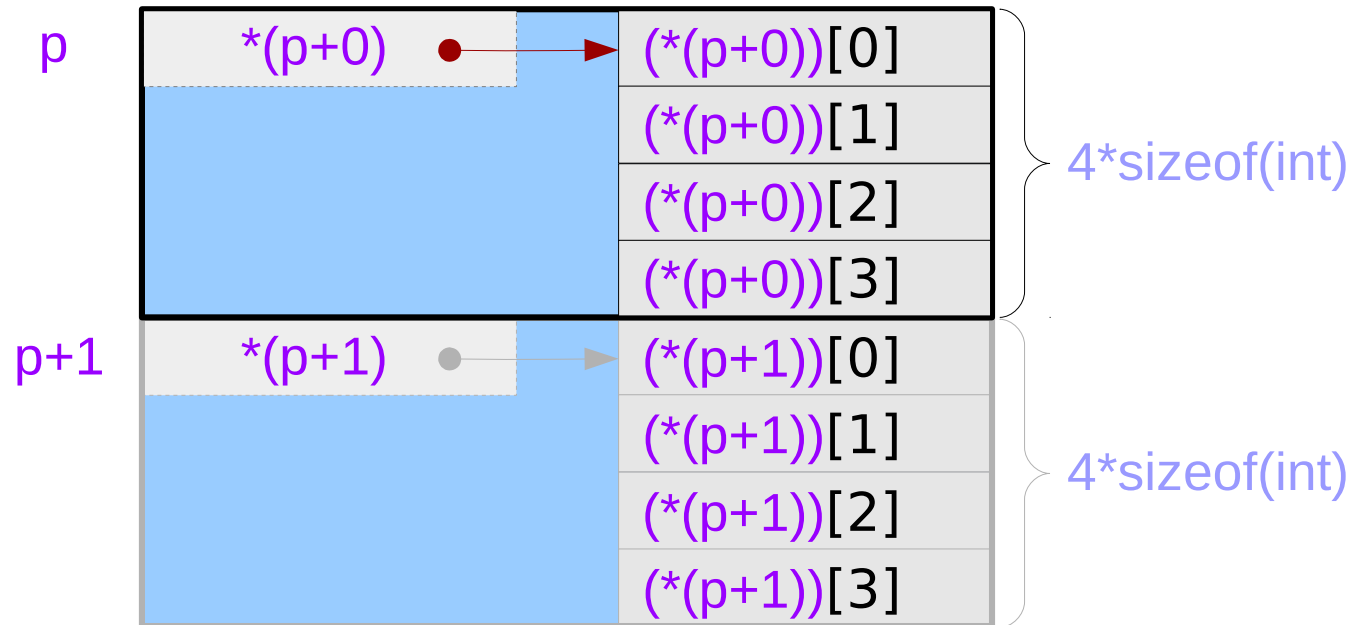
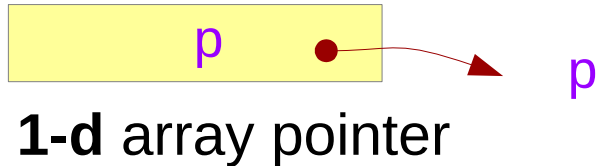


# Incrementing an array pointer

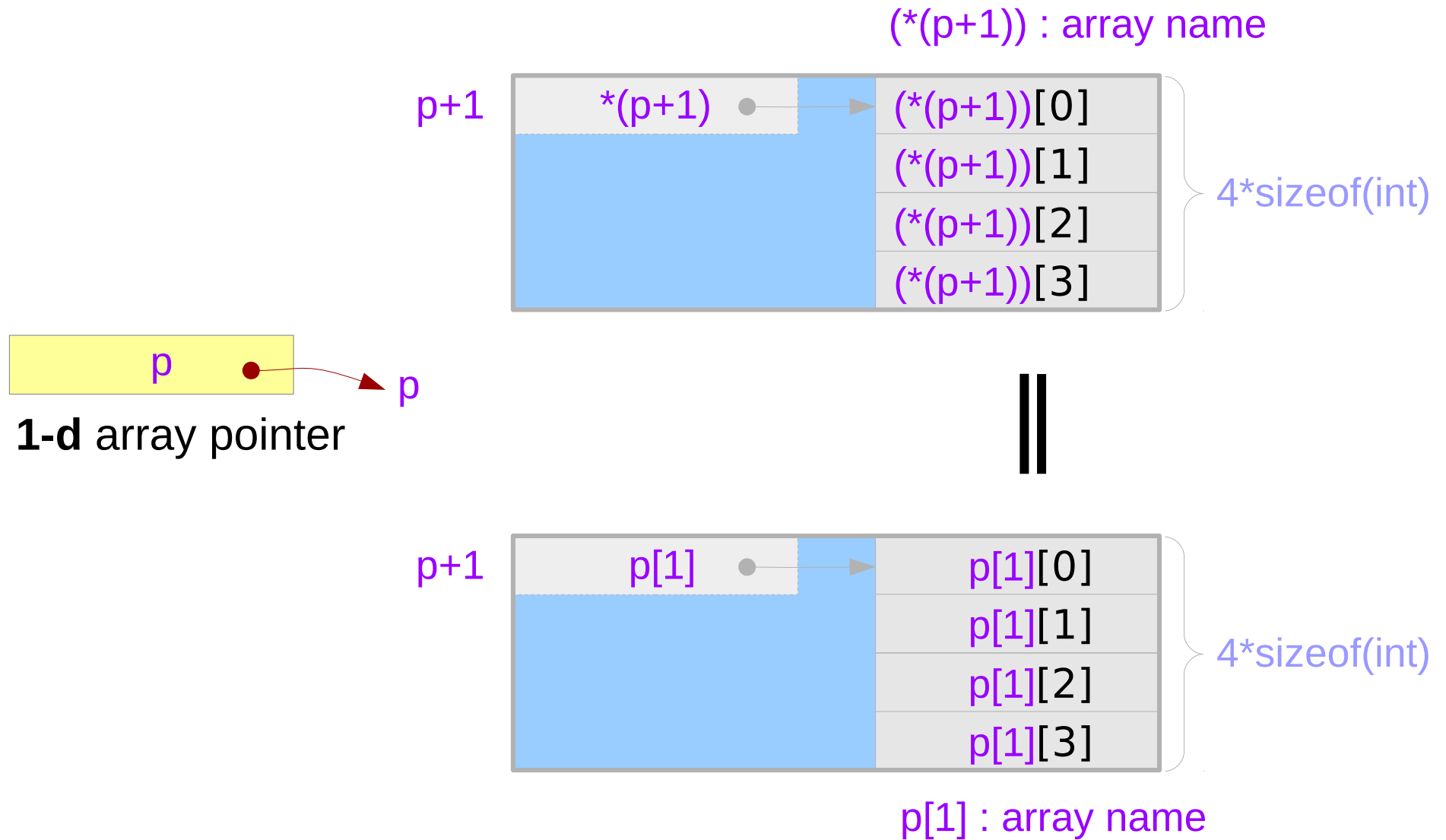
```
int (*p) [4];
```

## Aggregated Type Size

$$\begin{aligned} \text{address } p+1 - \text{address } p \\ = (\text{long}) (p+1) - (\text{long}) (p) &= 4 * \text{sizeof}(\text{int}) \end{aligned}$$



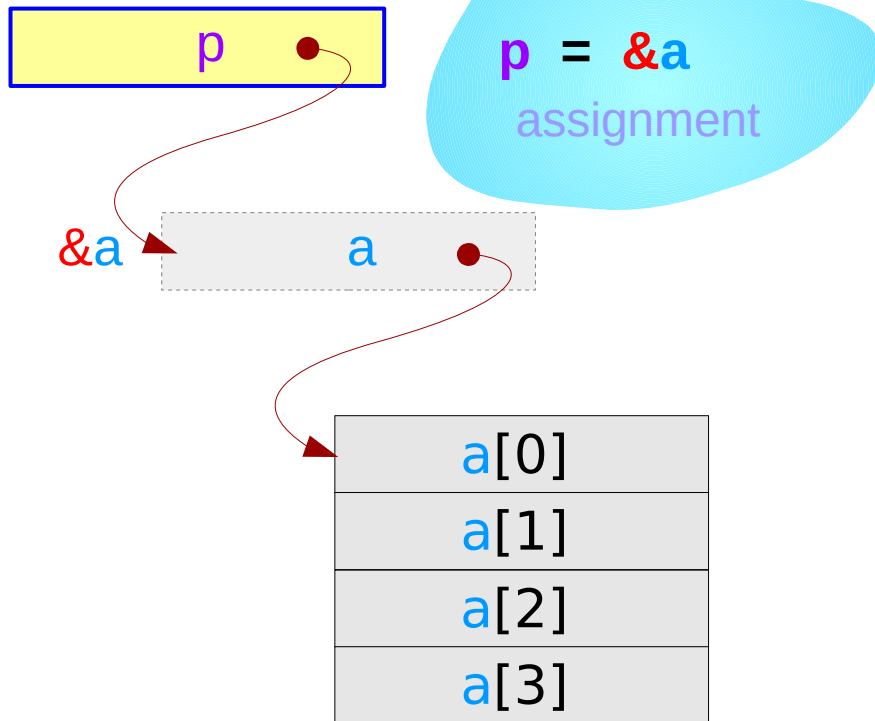
# Incrementing an array pointer – extending a dimension



# A 1-d array pointer and a 1-d array

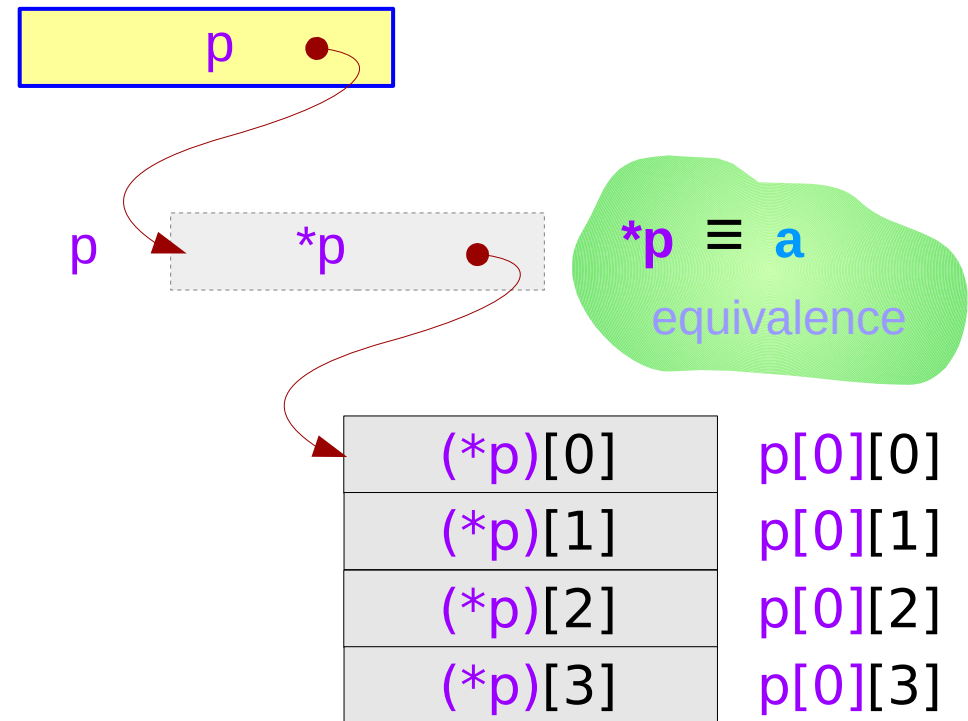
```
int    a [4];
```

1-d array pointer



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

1-d array pointer





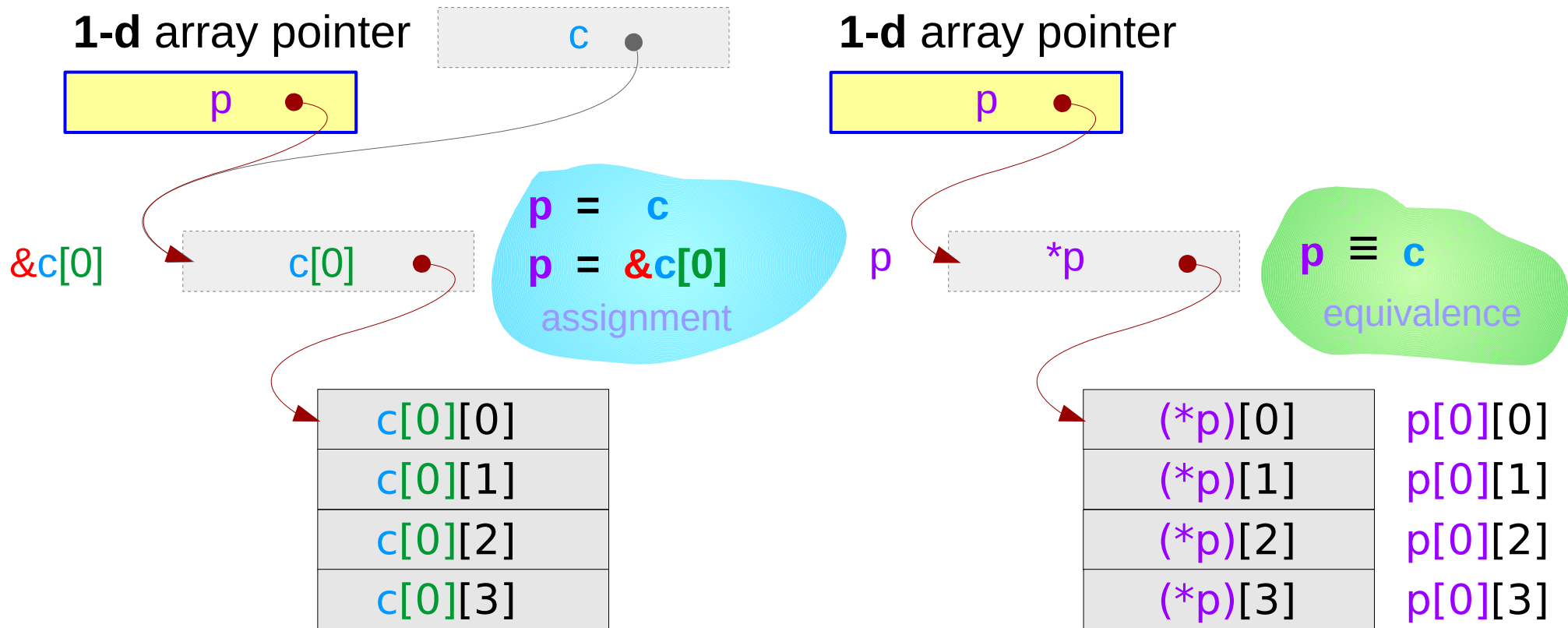
# A 1-d array pointer and a 2-d array

```
int c [4][4];
```

```
int (*p) [4] = &c[0];
```

1-d array pointer

1-d array pointer



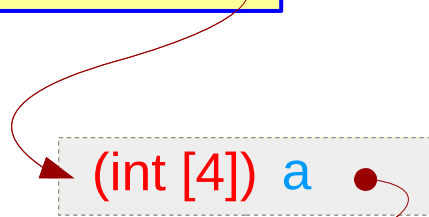
# A 1-d array pointer and a 1-d array – a type view

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

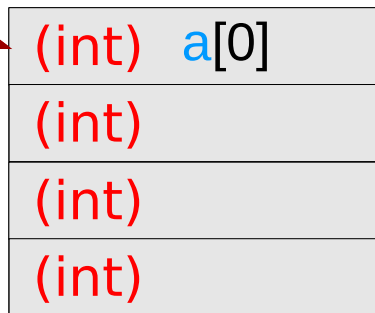
```
int (*p) [4] = &a;
```

1-d array pointer

`(int (*)[4])p` ●

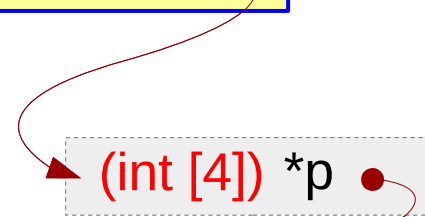


`(int *)`

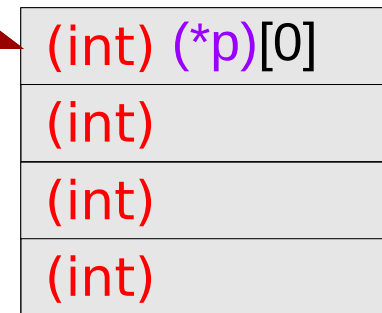


1-d array pointer

`(int (*)[4])p` ●



`p[0][0]`



# A 1-d array pointer and a 2-d array – a type view

```
int    c [4][4];
```

```
int (*p) [4] = &c[0];
```

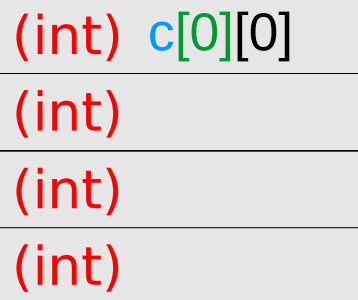
**1-d array pointer**

(int (\*)[4]) c ●

(int (\*)[4]) p ●

(int [4]) c[0] ●

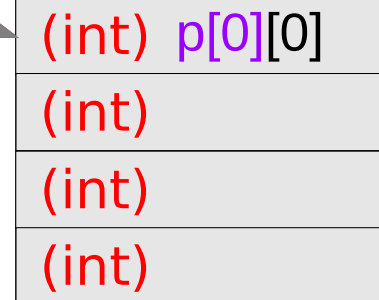
(int \*)



**1-d array pointer**

(int (\*)[4]) p ●

(int [4]) \*p ●

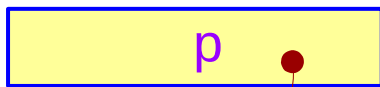


p[0][0]

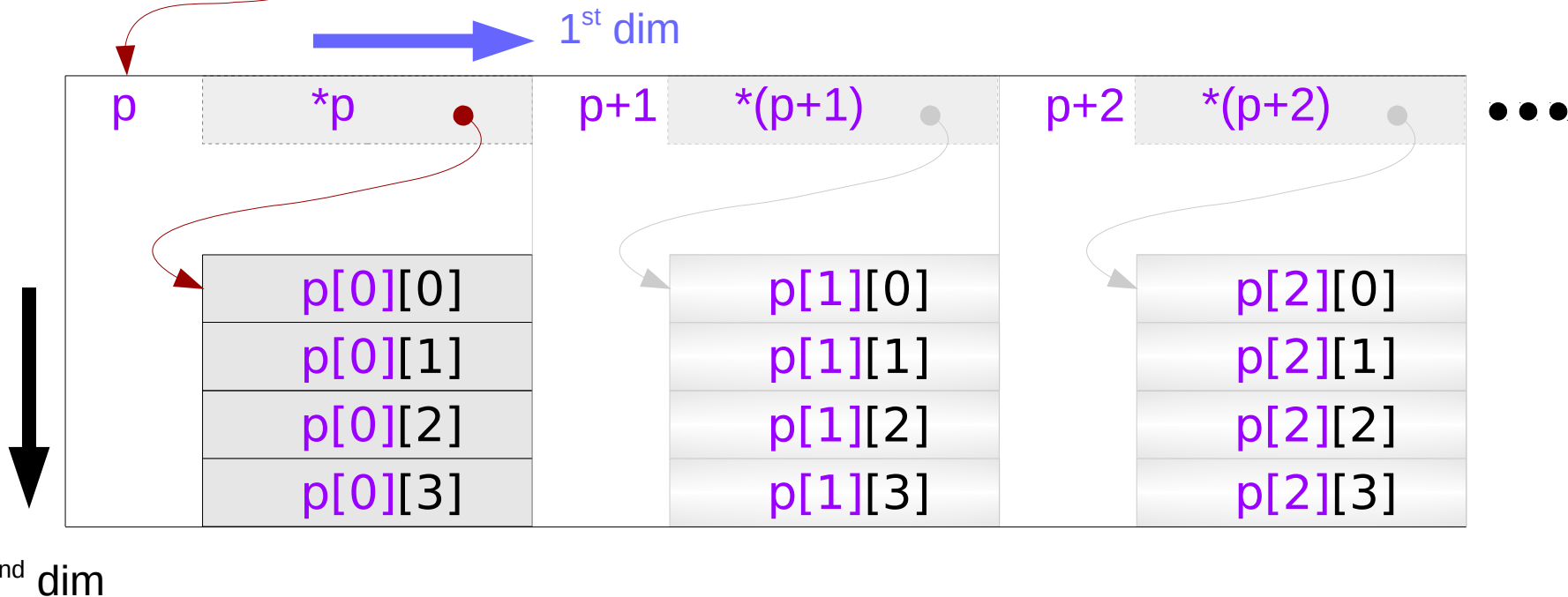
# A 1-d array pointer – extending a dimension

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

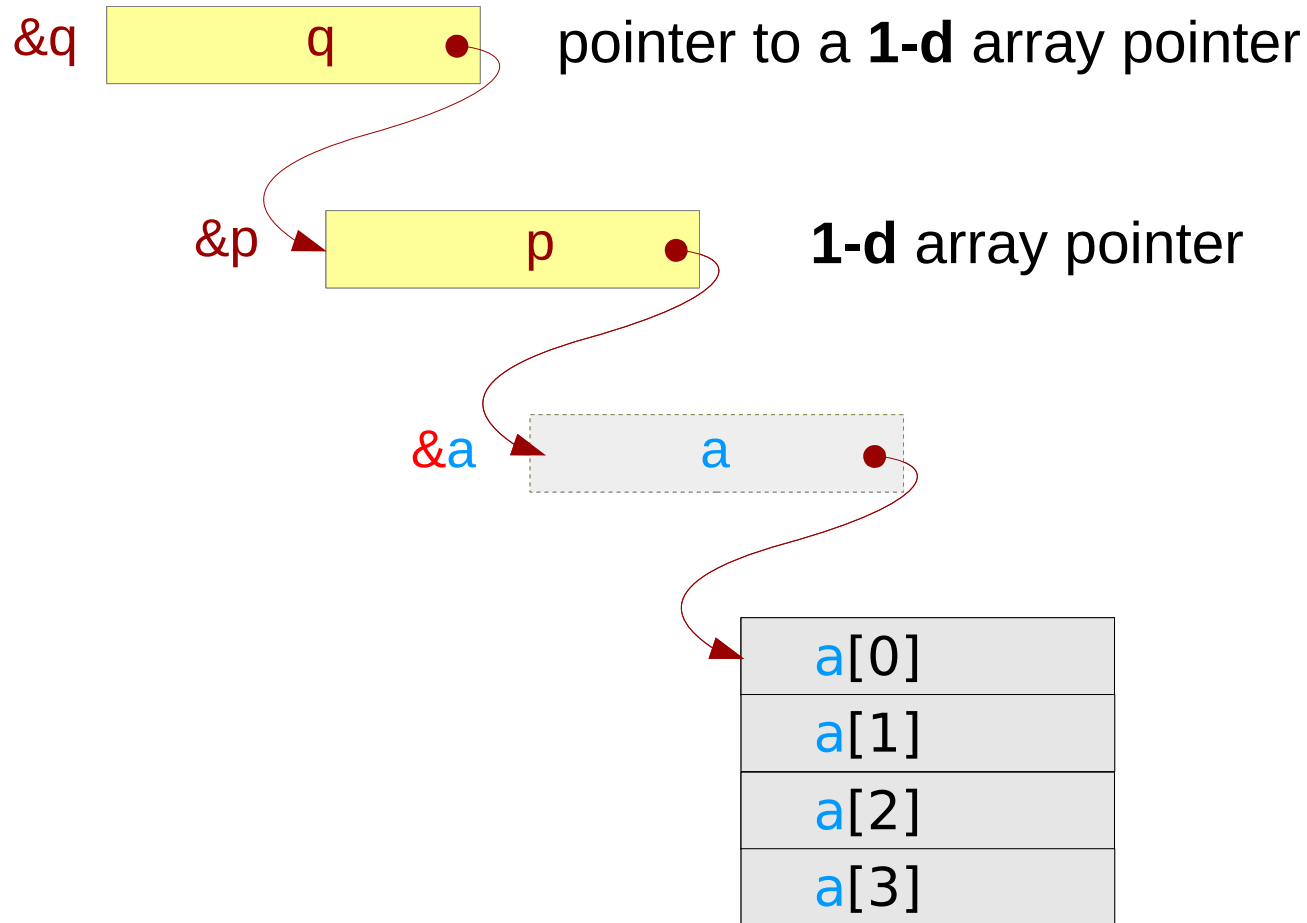
## 1-d array pointer



can be viewed as a 2-d array name  
: an additional dimension is added

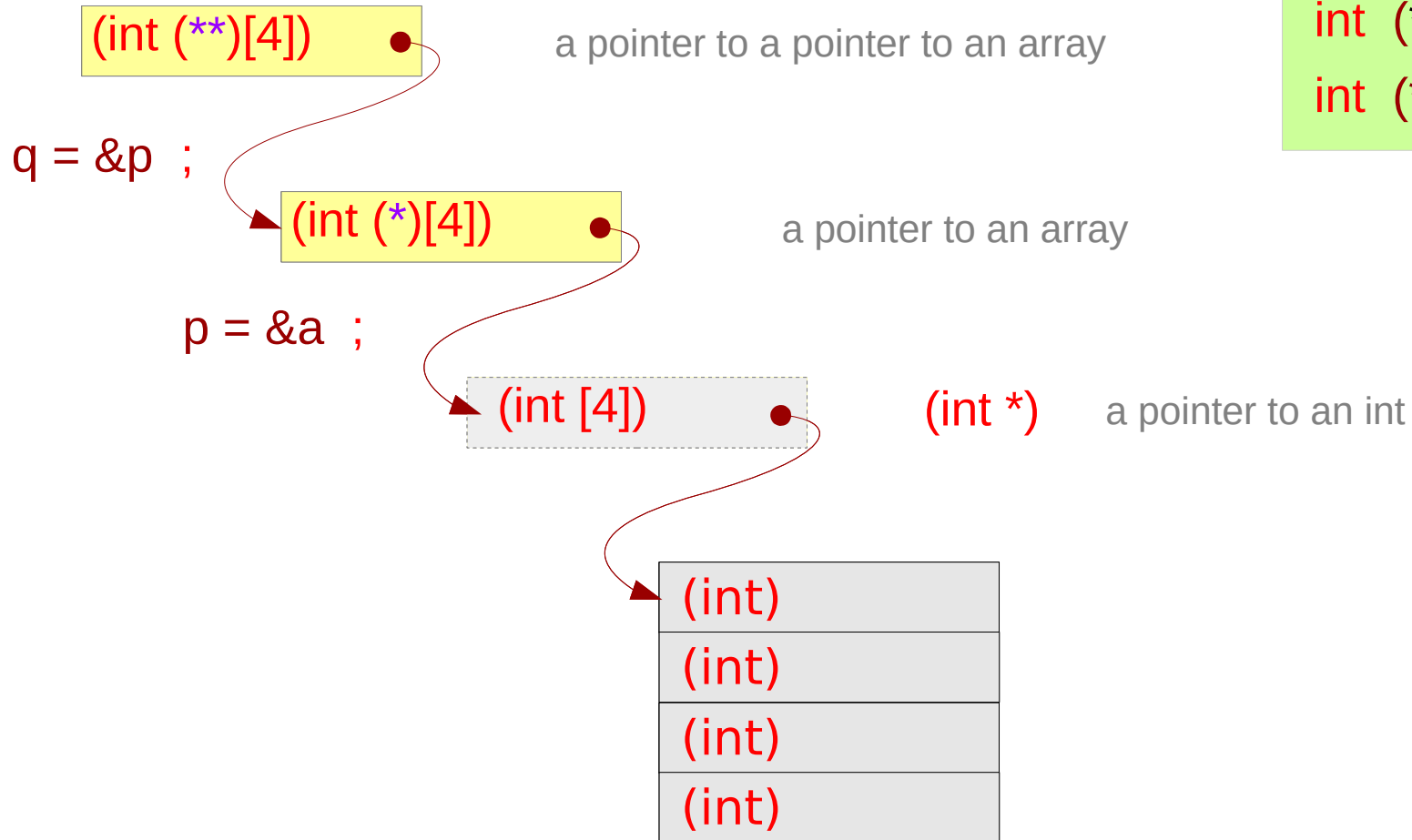


# Double pointer to a **1-d** array – a variable view



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

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

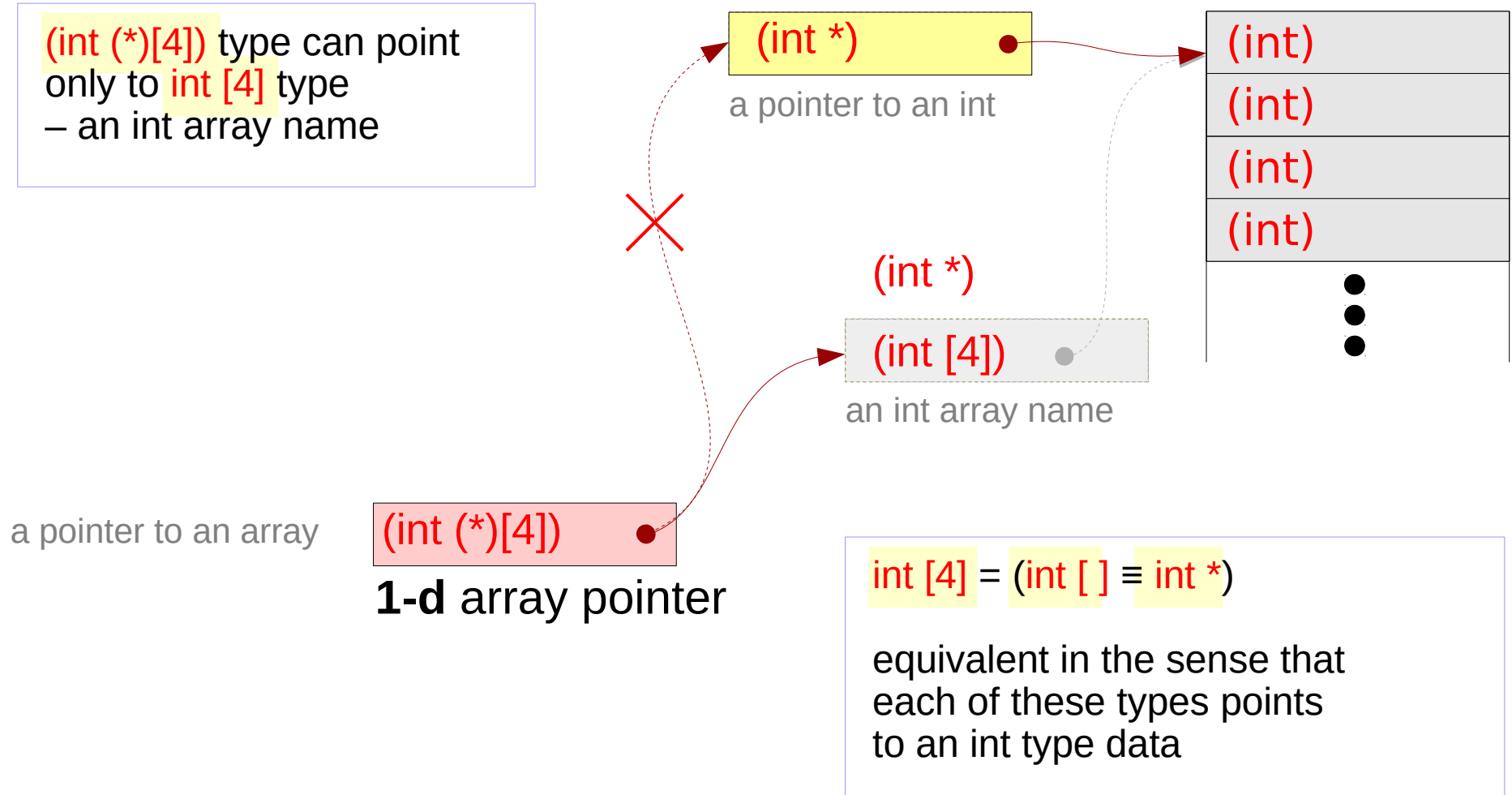


```
int a[4] ;  
int (*p) [4] = &a ;  
int (**q) [4] = &p ;
```

---

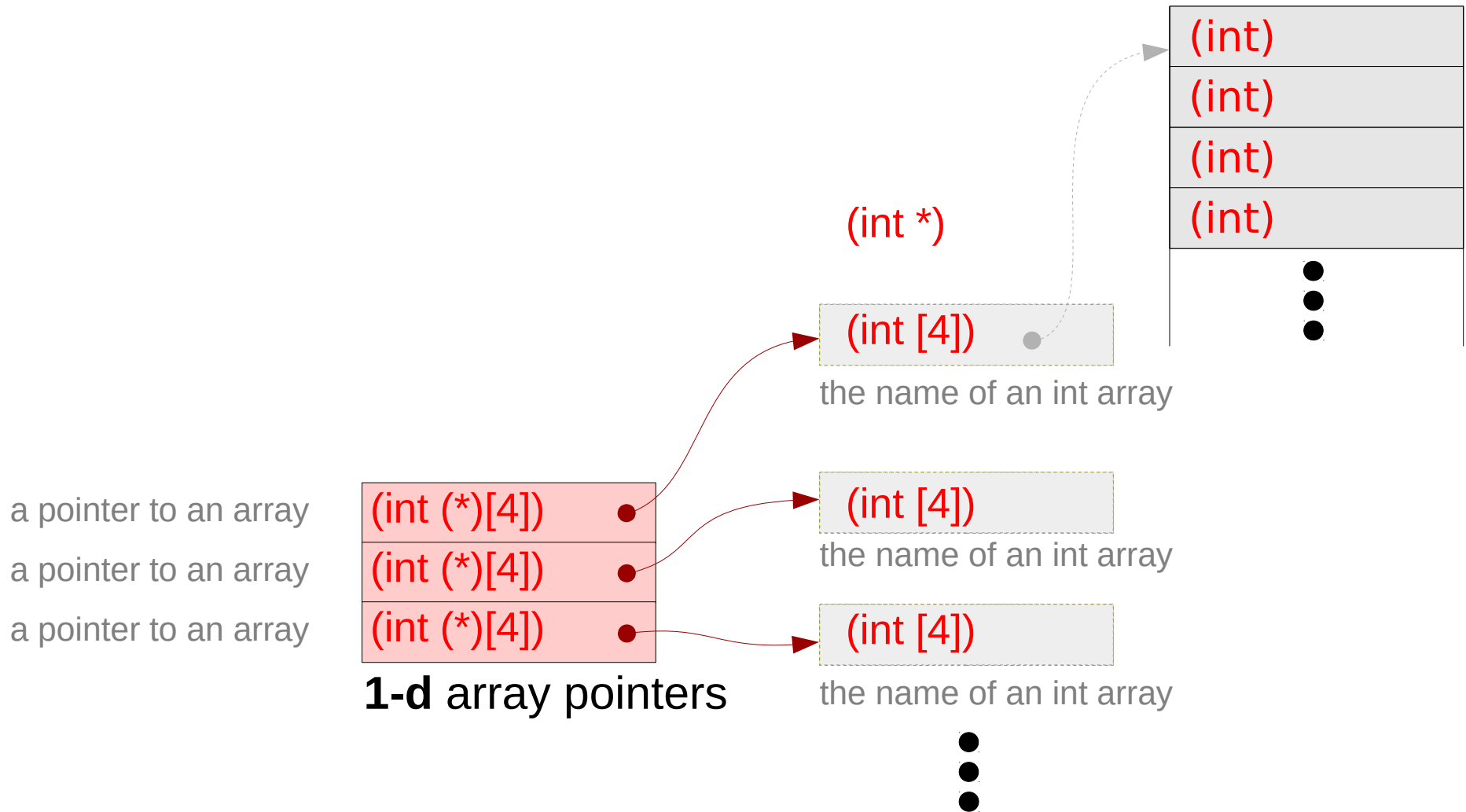
# Pointer to Multi-dimensional Arrays

# Integer pointer type





# Series of array pointers – a type view



# Series of array pointers – a variable view

```
int a[4]; int (*p1)[4]; int (*r);  
int b[4]; int (*p2)[4];  
int c[4]; int (*p3)[4];
```

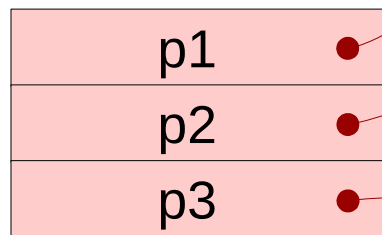
assignment

```
p1 = &a  
p2 = &b  
p3 = &c
```

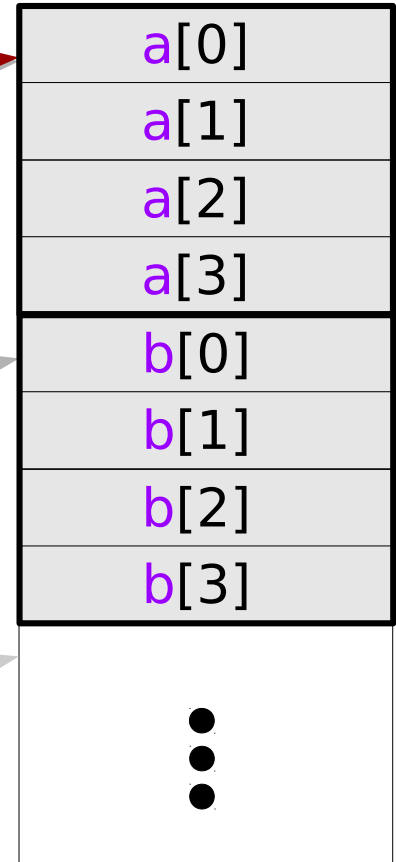
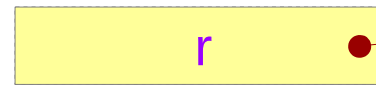
equivalence

```
(*p1) ≡ p1[0] ≡ a  
(*p2) ≡ p2[0] ≡ b  
(*p3) ≡ p3[0] ≡ c
```

a pointer to an array  
a pointer to an array  
a pointer to an array



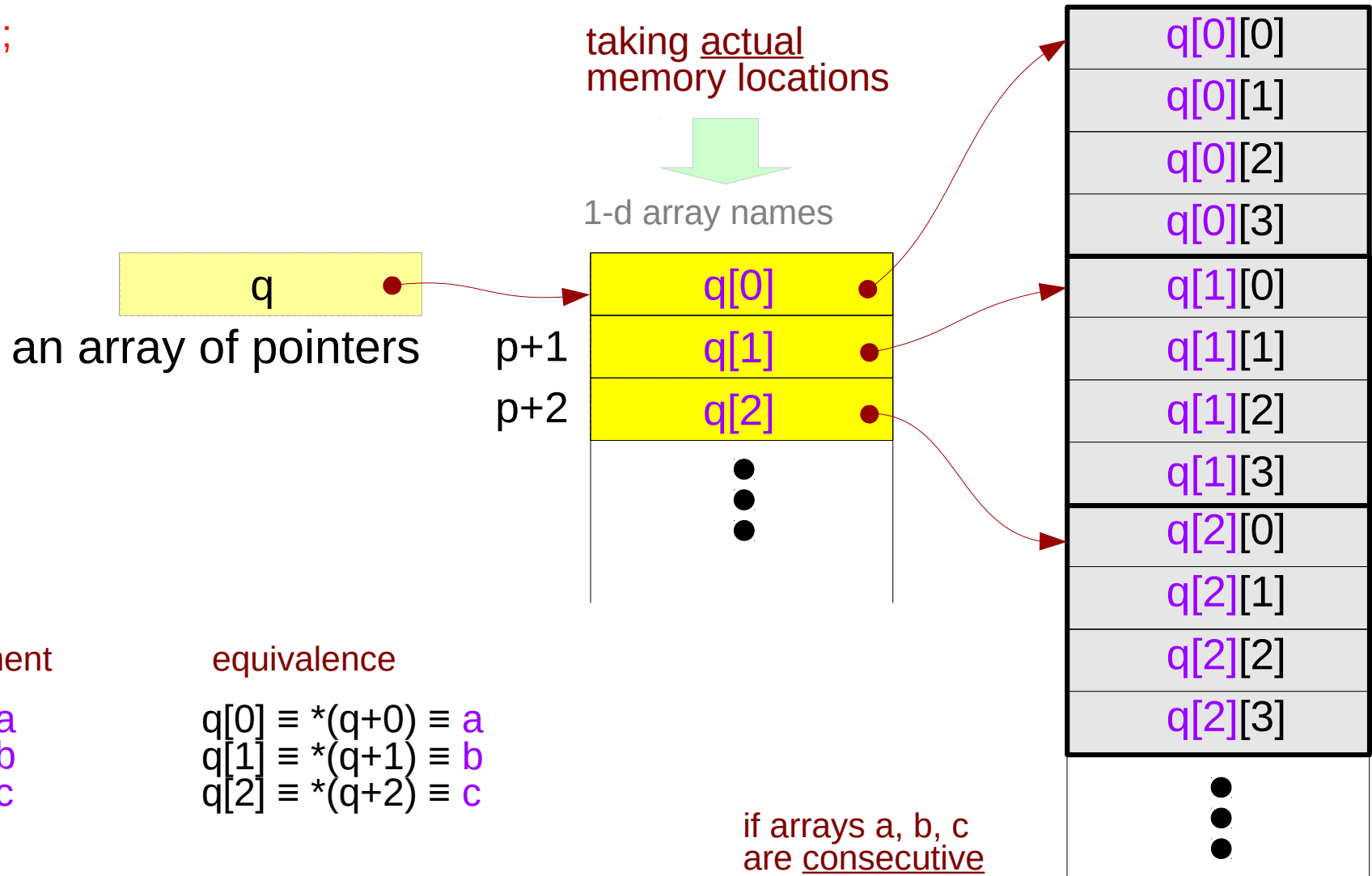
**1-d** array pointers



assume that  
array a, b, and c  
are contiguous  
in the memory

# Pointer array – a variable view

```
int *q[3];
```



# Array pointer to consecutive **1-d** arrays

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

a pointer to an array



**1-d** array pointer

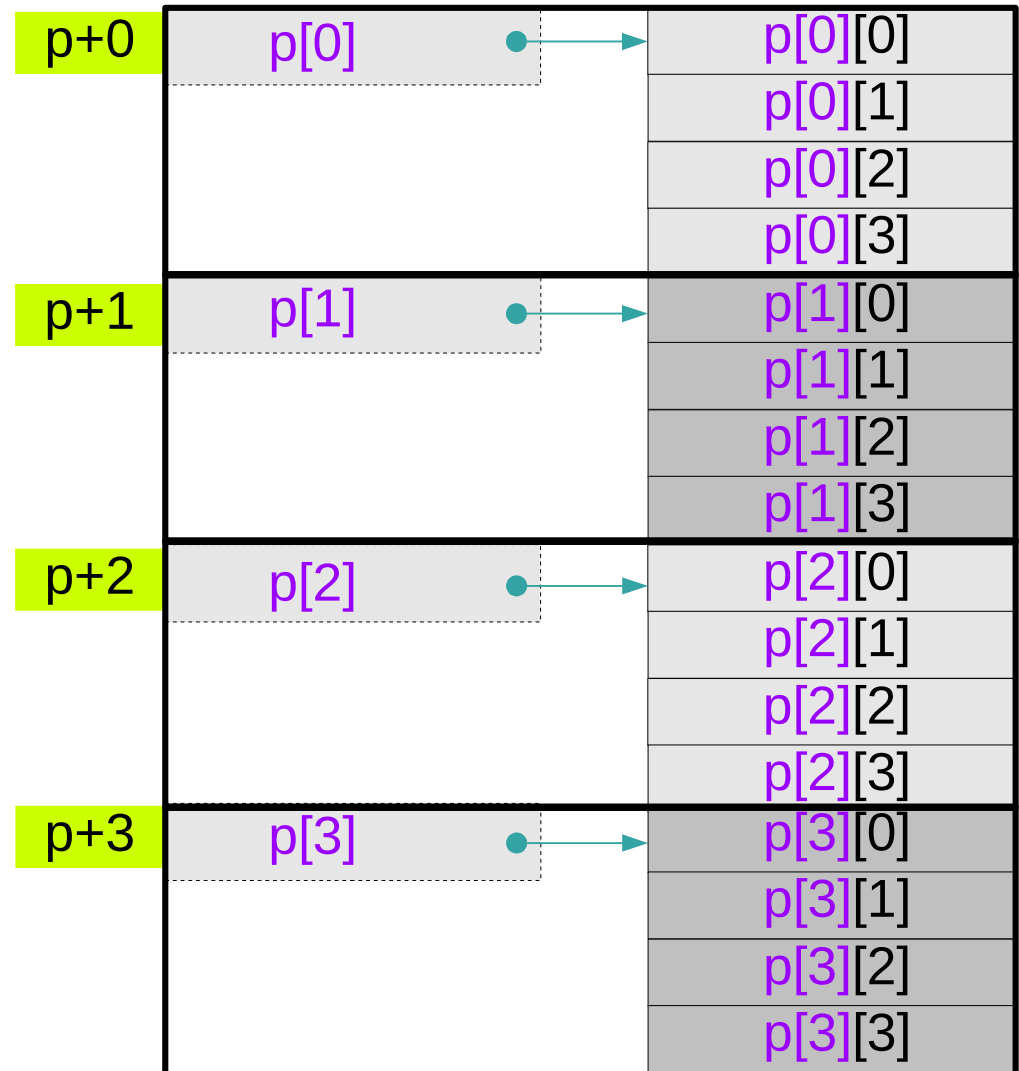
assignment

```
p = &a
```

equivalence

```
*(p+0) ≡ p[0] ≡ a  
*(p+1) ≡ p[1] ≡ b  
*(p+2) ≡ p[2] ≡ c  
*(p+3) ≡ p[3] ≡ d
```

if arrays a, b, c, d  
are consecutive



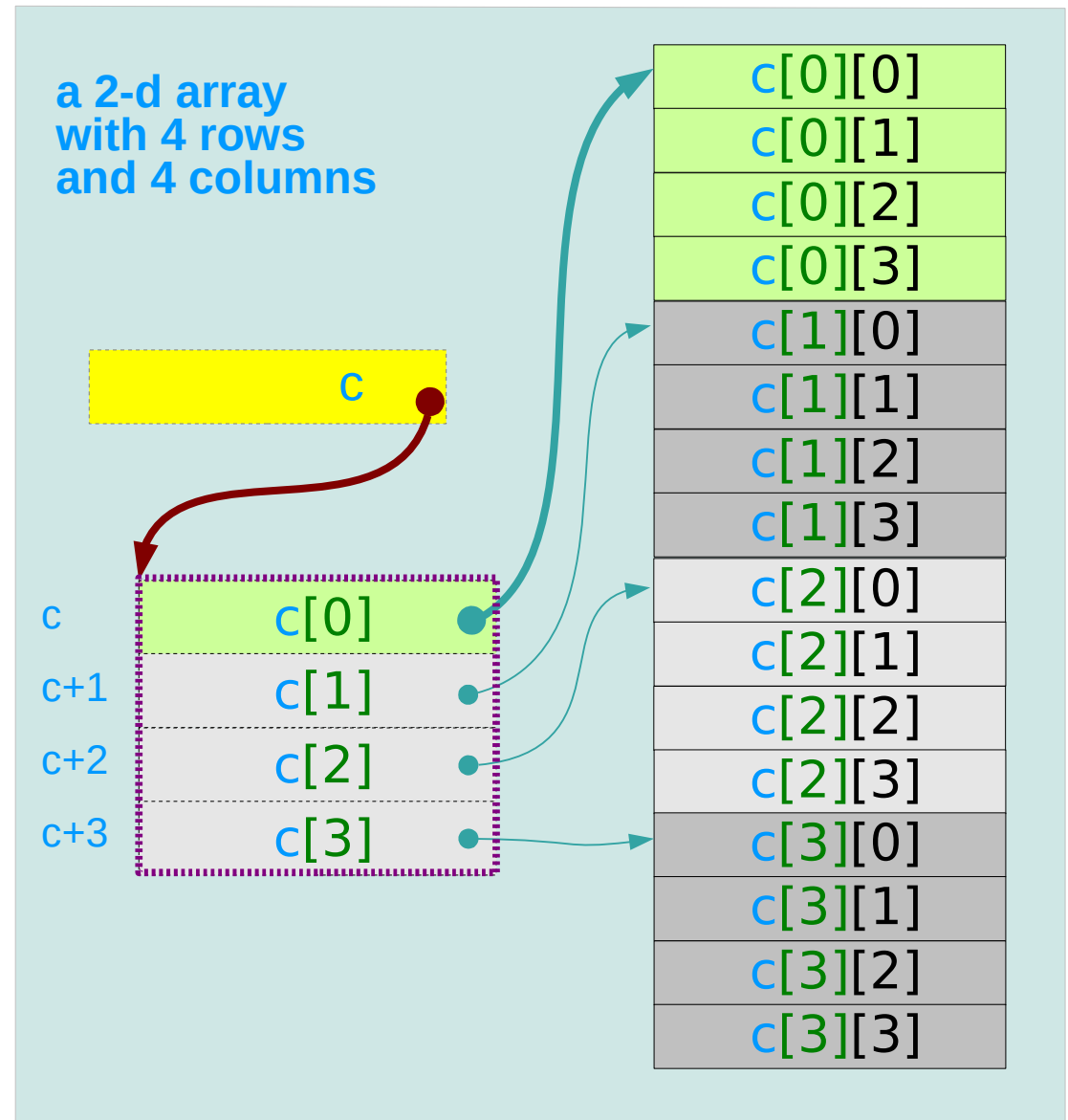
# A 2-d array and its sub-arrays – a variable view

the array name **c** of a **2-d** array  
as a **1-d** array pointer which  
points to its 1<sup>st</sup> **1-d** sub-array

**c** is the **1-d** array pointer  
**c[i]**'s are the **1-d** sub-array name

<b>c[0]</b>	the 1 <sup>st</sup>	1-d sub-array name
<b>c[1]</b>	the 2 <sup>nd</sup>	1-d sub-array name
<b>c[2]</b>	the 3 <sup>rd</sup>	1-d sub-array name
<b>c[3]</b>	the 4 <sup>th</sup>	1-d sub-array name

Compilers can make **c[i]**'s require  
no actual memory locations



# A 2-d array and its sub-arrays – a type view

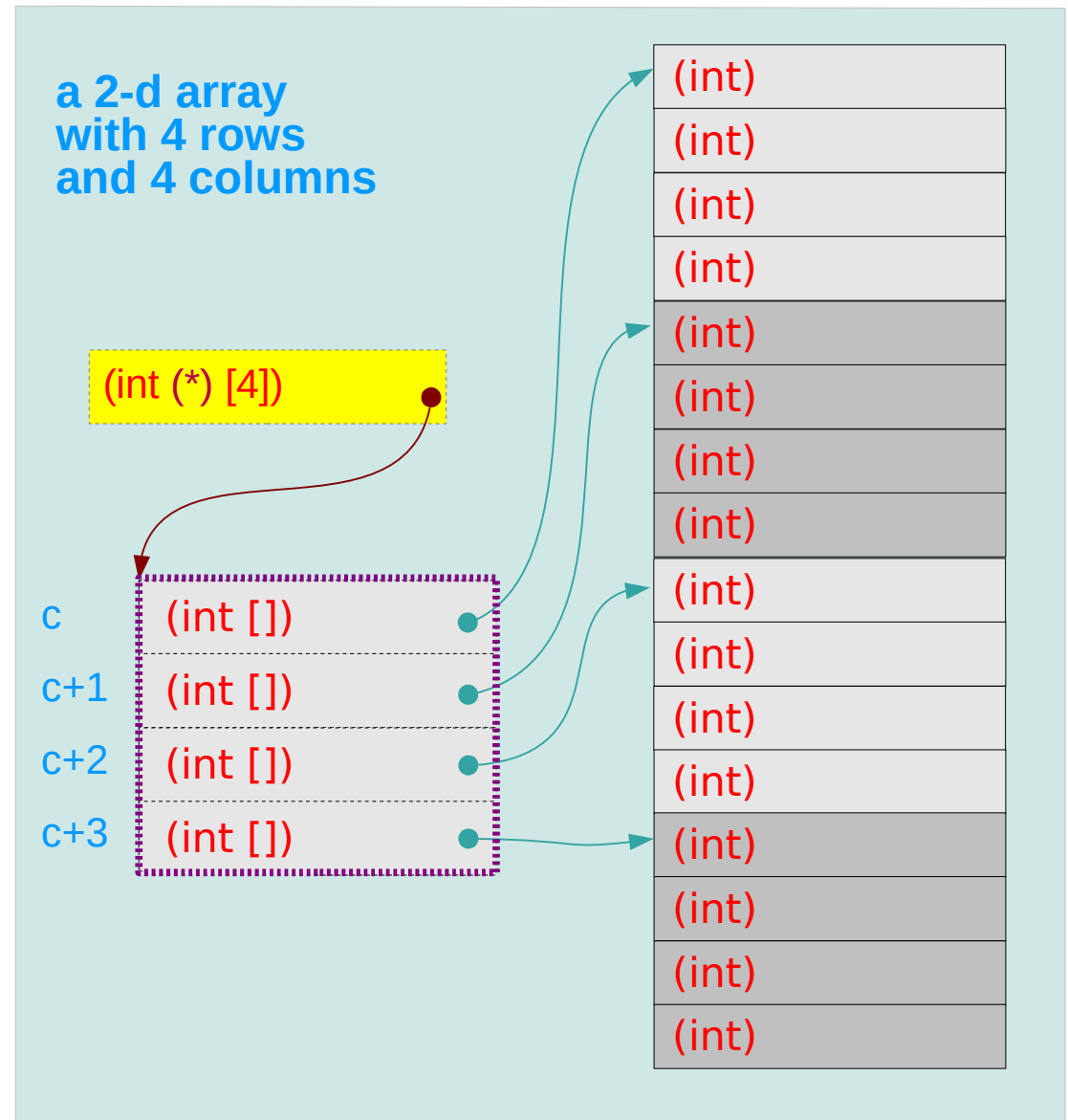
**1-d** array pointer

**1-d** array name

**1-d** array name

**1-d** array name

**1-d** array name



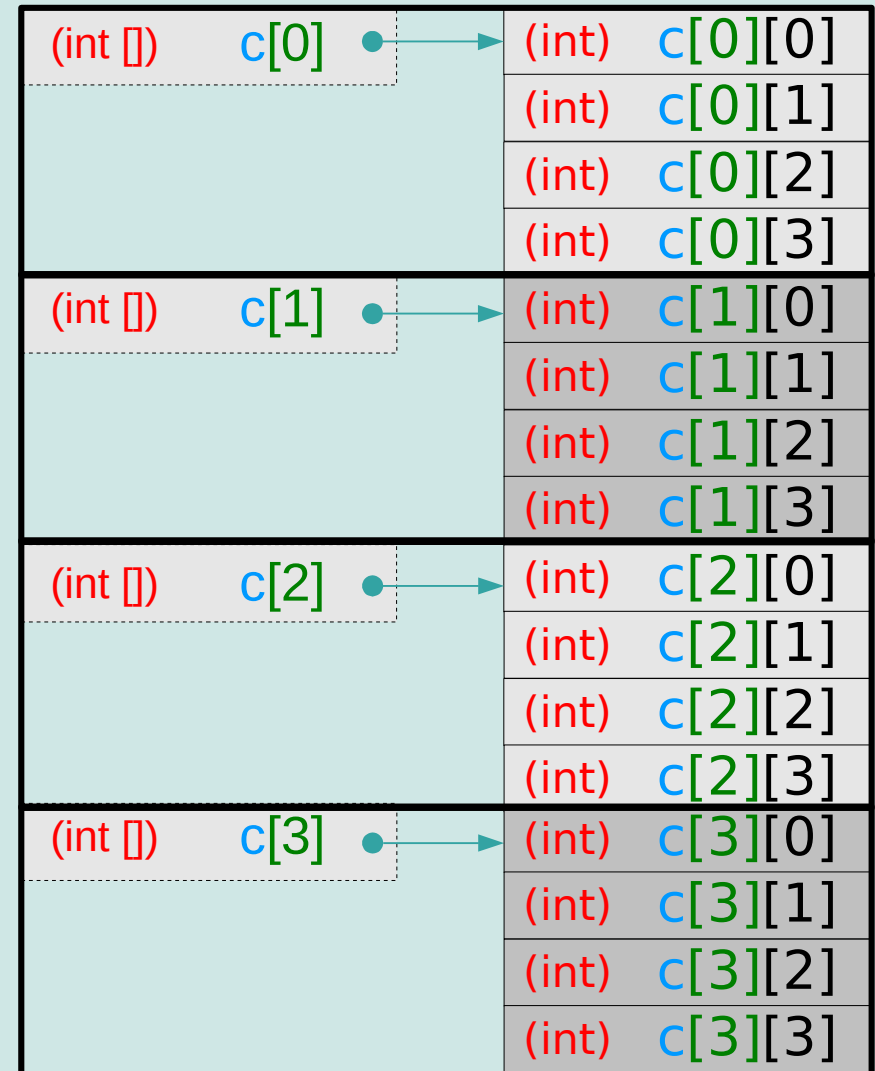
# 1-d subarray aggregated data type

The 1<sup>st</sup> subarray **c[0]** (=array name)  
sizeof(**c[0]**) = 16 bytes

The 2<sup>nd</sup> subarray **c[1]** (=array name)  
sizeof(**c[1]**) = 16 bytes

The 3<sup>rd</sup> subarray **c[2]** (=array name)  
sizeof(**c[2]**) = 16 bytes

The 4<sup>th</sup> subarray **c[3]** (=array name)  
sizeof(**c[3]**) = 16 bytes



# 2-d array name as a pointer to a 1-d subarray

1-d array pointer

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

`(int []) c[0]`

<code>(int) c[0][0]</code>
<code>(int) c[0][1]</code>
<code>(int) c[0][2]</code>
<code>(int) c[0][3]</code>

The 1<sup>st</sup> subarray

1-d array pointer

`(int (*) [4]) c+1`

`(int []) c[1]`

<code>(int) c[1][0]</code>
<code>(int) c[1][1]</code>
<code>(int) c[1][2]</code>
<code>(int) c[1][3]</code>

The 2<sup>nd</sup> subarray

1-d array pointer

`(int (*) [4]) c+2`

`(int []) c[2]`

<code>(int) c[2][0]</code>
<code>(int) c[2][1]</code>
<code>(int) c[2][2]</code>
<code>(int) c[2][3]</code>

The 3<sup>rd</sup> subarray

1-d array pointer

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

`(int []) c[3]`

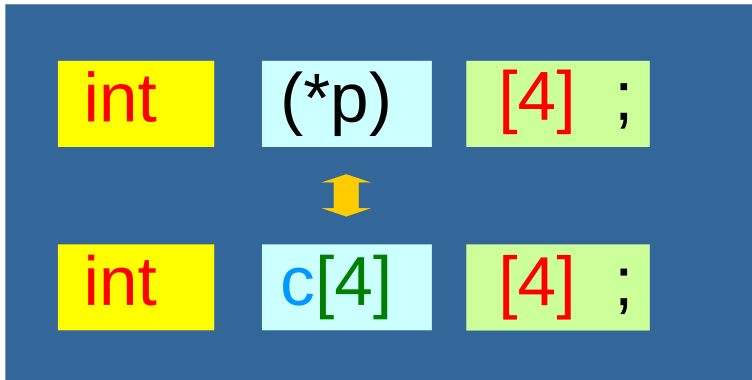
<code>(int) c[3][0]</code>
<code>(int) c[3][1]</code>
<code>(int) c[3][2]</code>
<code>(int) c[3][3]</code>

The 4<sup>th</sup> subarray



# 2-d array and 1-d and 2-d array pointers

## 1-d array pointer



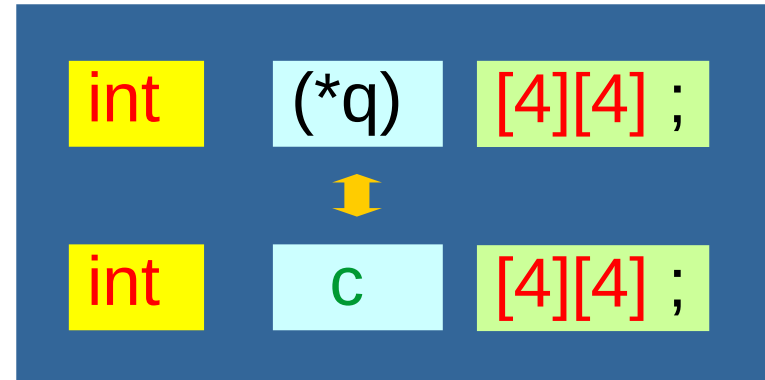
(int (\*) [4])

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

```
p = c;
```

```
p[0] ≡ c[0]  
p[1] ≡ c[1]  
p[2] ≡ c[2]  
p[3] ≡ c[3]
```

## 2-d array pointer



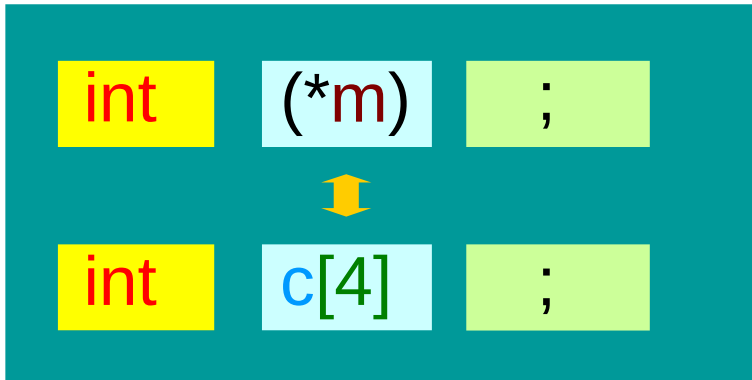
(int(\*)[4][4])

```
q = &c;
```

```
(*q)[0] ≡ q[0][0] ≡ c[0]  
(*q)[1] ≡ q[0][0] ≡ c[1]  
(*q)[2] ≡ q[0][0] ≡ c[2]  
(*q)[3] ≡ q[0][0] ≡ c[3]
```

# 1-d array and 0-d and 1-d array pointers

0-d array pointer : int pointer



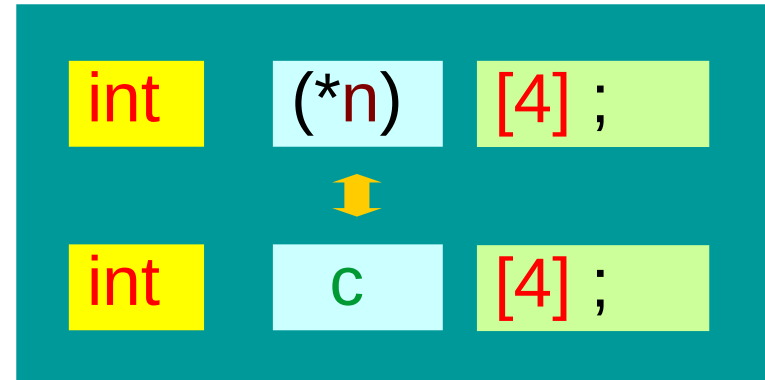
(int (\*))

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

```
m = c;
```

```
m[0] ≡ c[0]  
m[1] ≡ c[1]  
m[2] ≡ c[2]  
m[3] ≡ c[3]
```

1-d array pointer

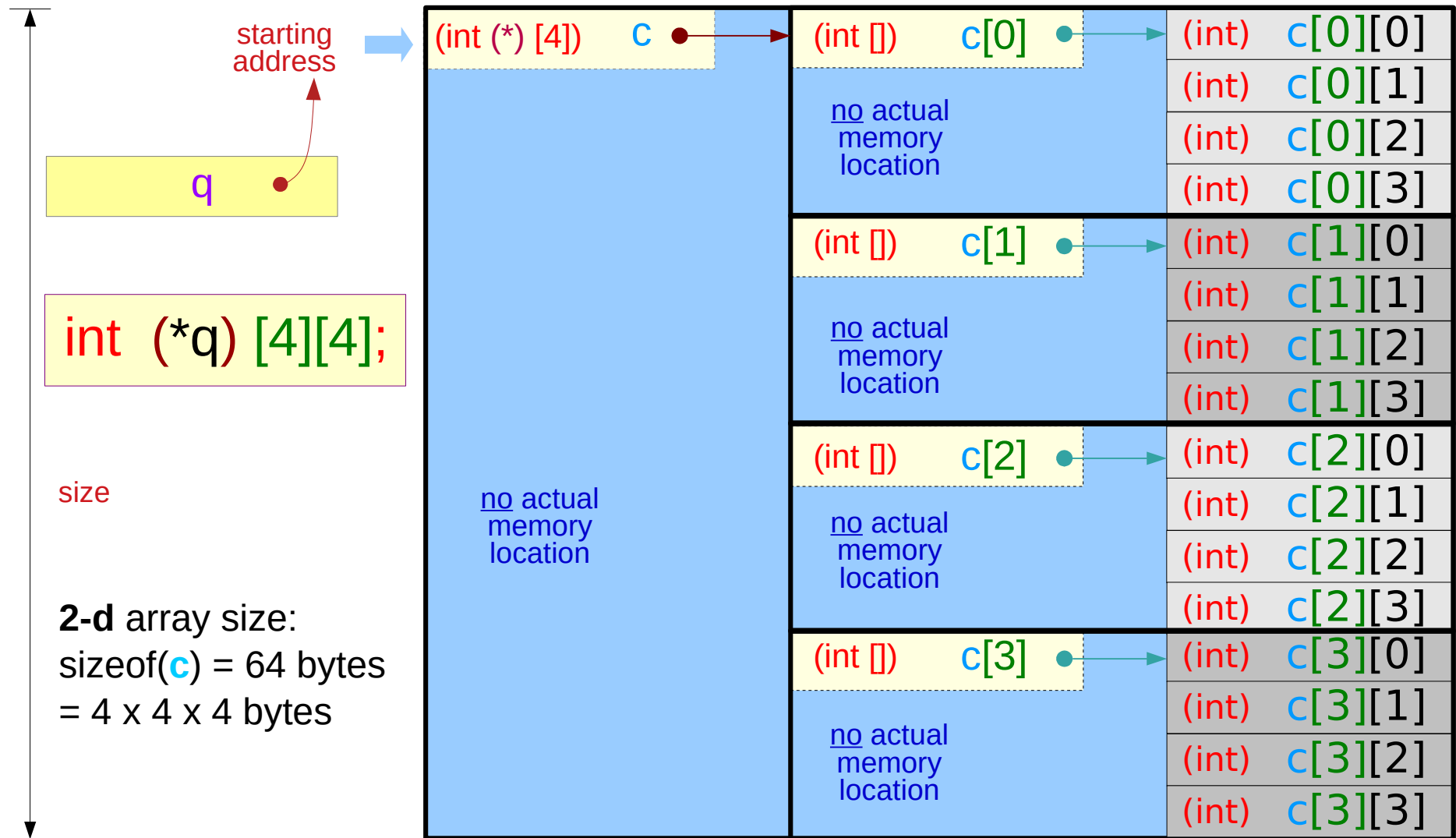


(int(\*)[4])

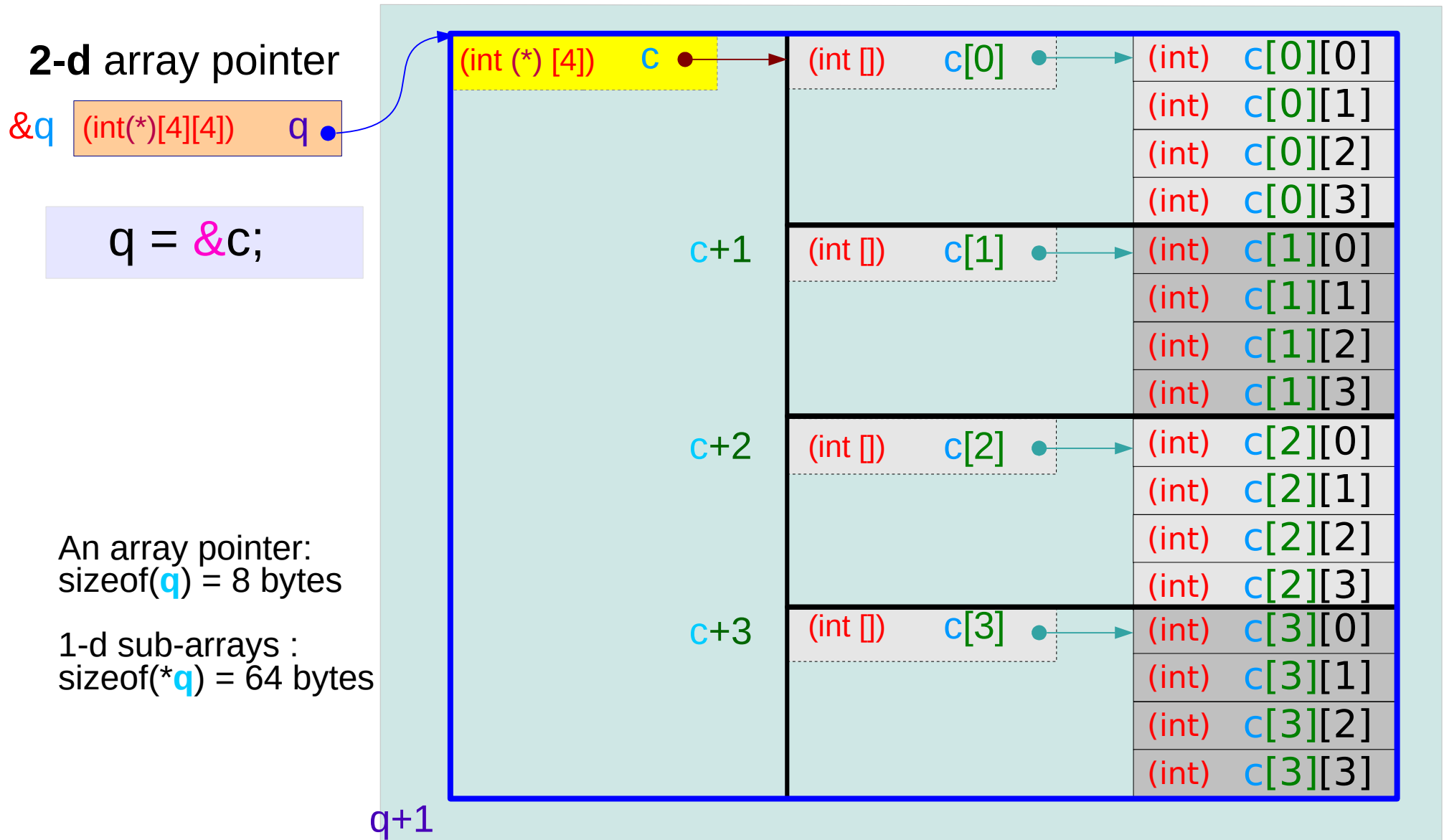
```
n = &c;
```

```
(*n)[0] ≡ n[0][0] ≡ c[0]  
(*n)[1] ≡ n[0][0] ≡ c[1]  
(*n)[2] ≡ n[0][0] ≡ c[2]  
(*n)[3] ≡ n[0][0] ≡ c[3]
```

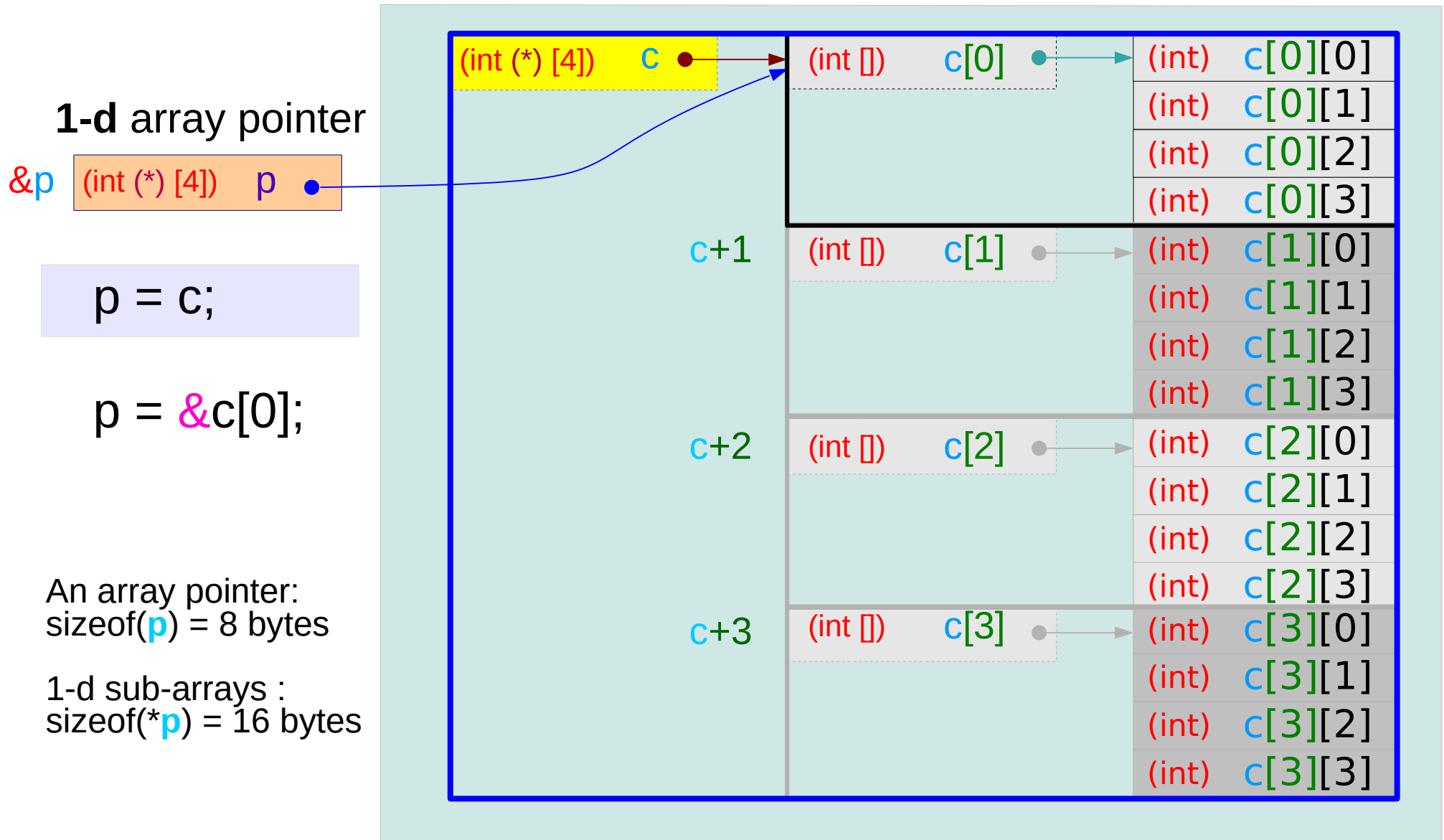
# Pointer to a 2-d array – an aggregated type view



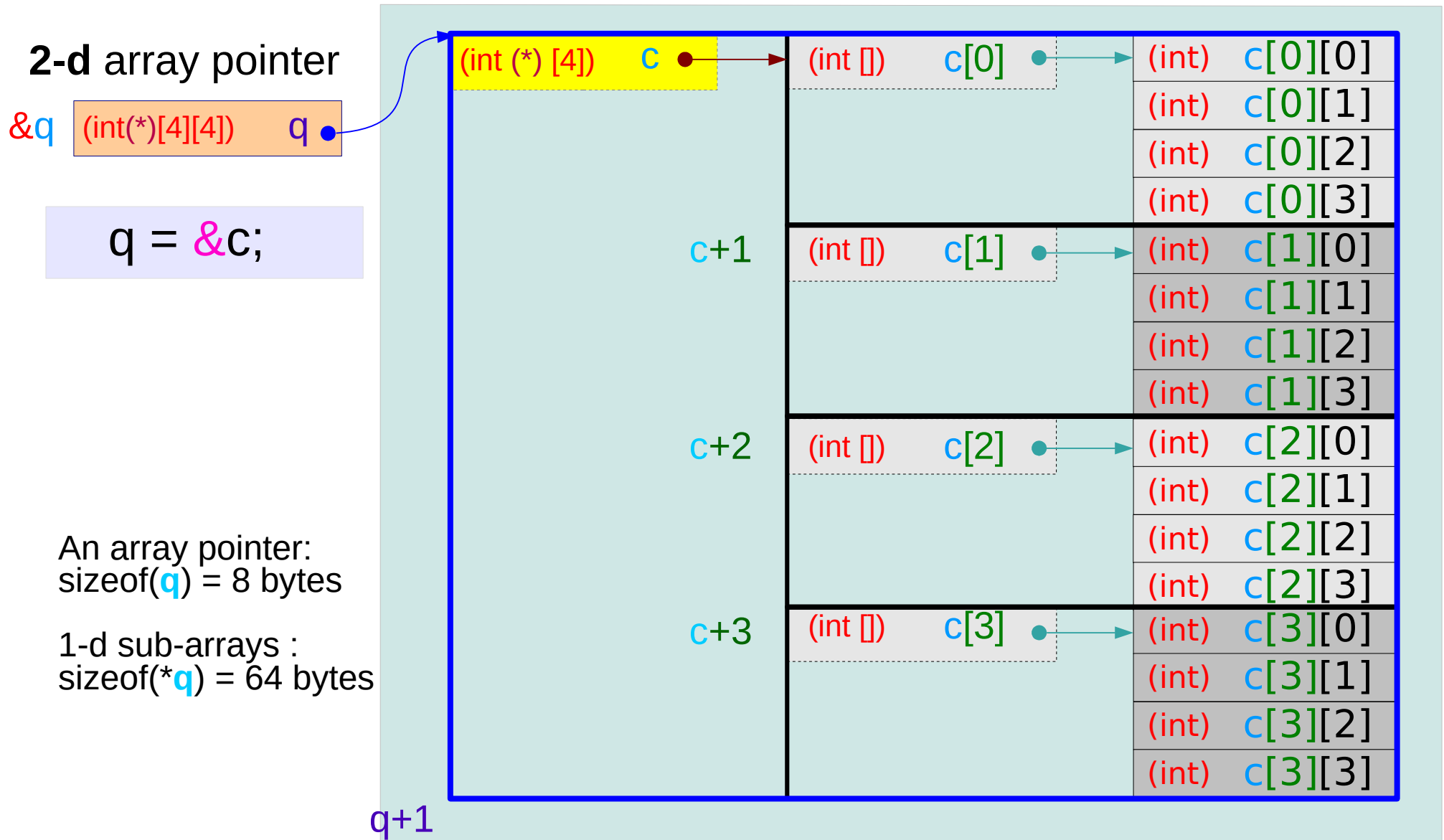
# 2-d array pointer to a 2-d array



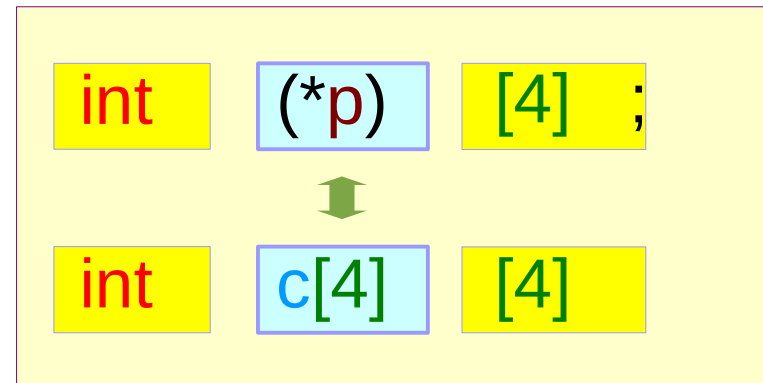
# 1-d array pointer to a 2-d array



# 2-d array pointer to a 2-d array



# Using a 1-d array pointer to a 2-d array

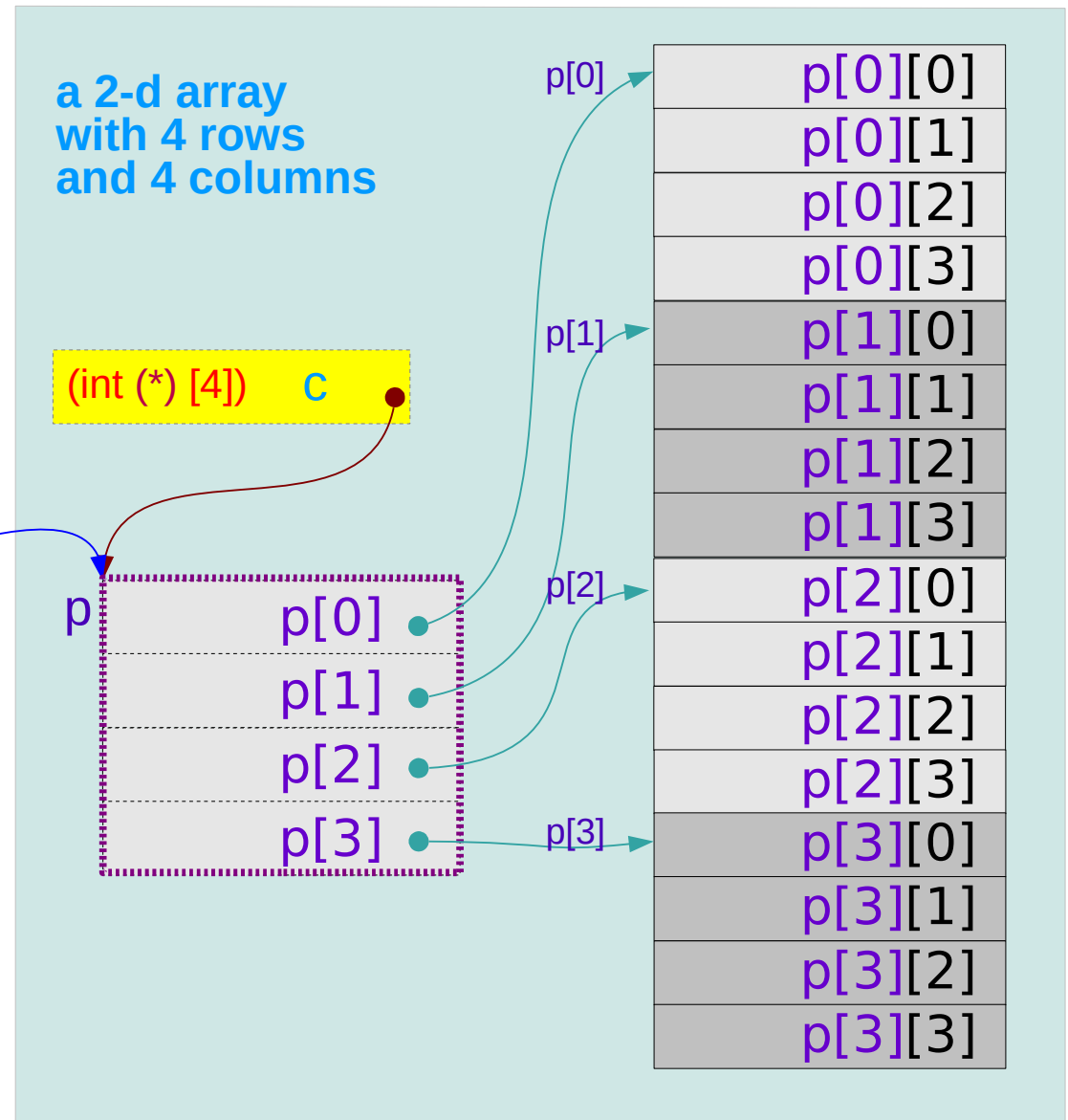


1-d array pointer

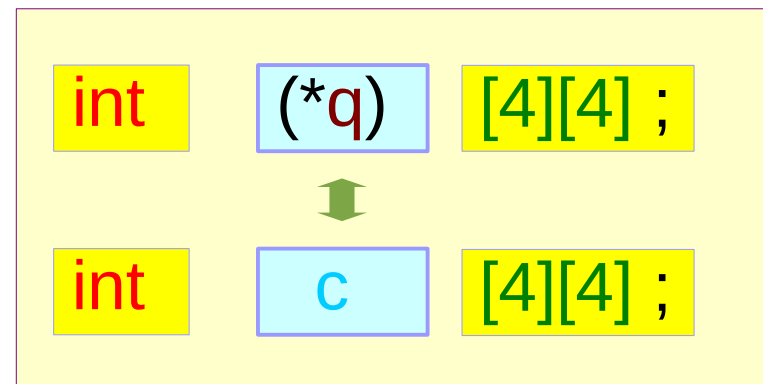
`&p` `(int (*) [4]) p`

`p = c;`

`p[0] ≡ c[0]`  
`p[1] ≡ c[1]`  
`p[2] ≡ c[2]`  
`p[3] ≡ c[3]`



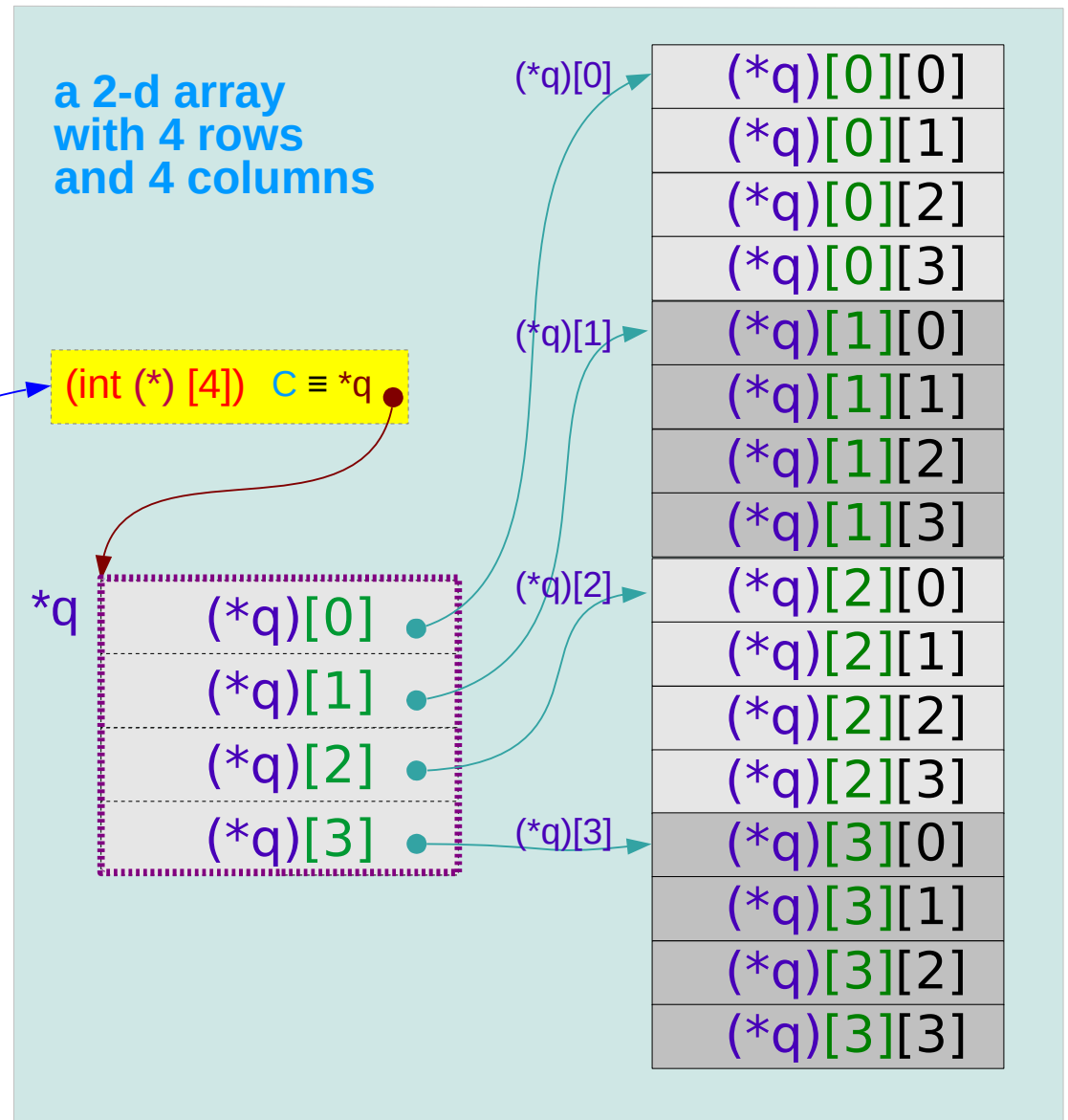
# Using a 2-d array pointer to a 2-d array



2-d array pointer  
&p (int(\*)[4][4]) q

q = &c;

$$\begin{pmatrix} (*q)[0] \\ (*q)[1] \\ (*q)[2] \\ (*q)[3] \end{pmatrix} \equiv c \begin{pmatrix} [0] \\ [1] \\ [2] \\ [3] \end{pmatrix}$$





# $(n-1)$ -d array pointer to a $n$ -d array

`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

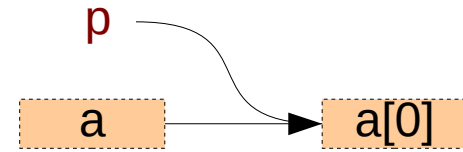
`int c[4][2][3];`                    **3-d** array  
`int (*r)[2][3];`                    **2-d** array pointer

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

# $n$ -d array name : $(n-1)$ -d array pointer

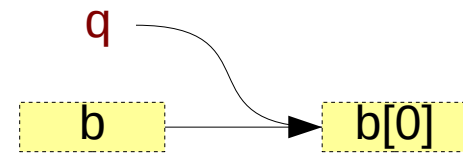
```
int a[4];  
int (*p);
```

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



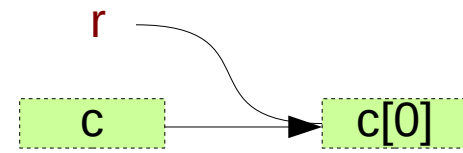
```
int b[4][2];  
int (*q)[2];
```

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



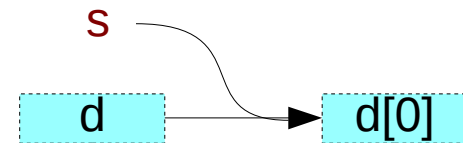
```
int c[4][2][3];  
int (*r)[2][3];
```

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

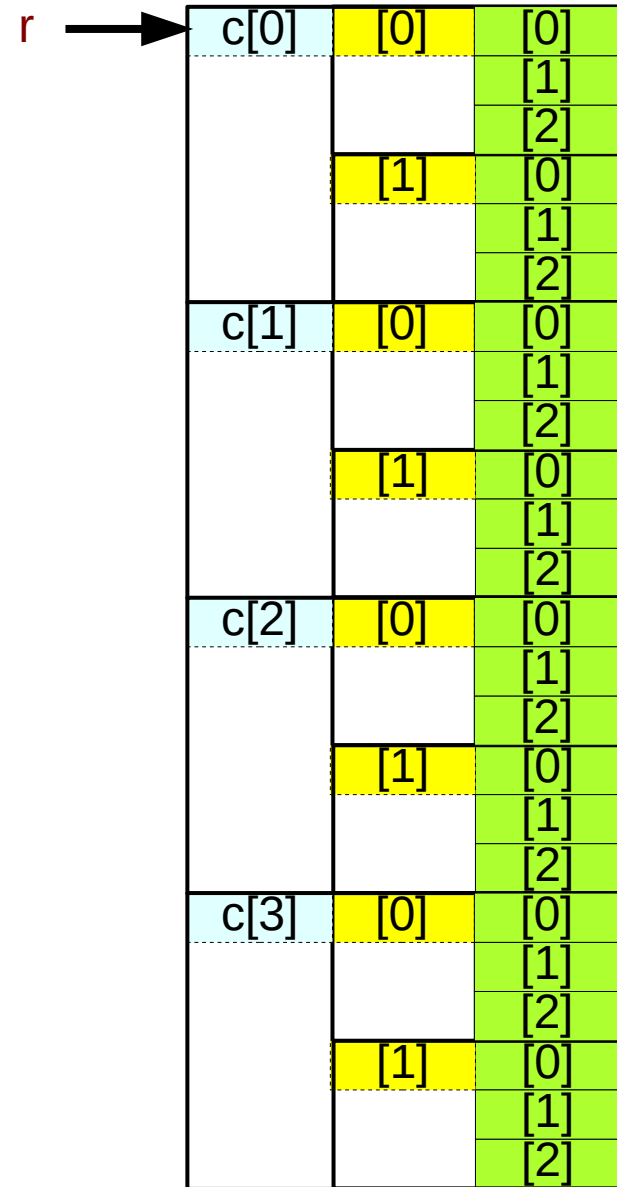
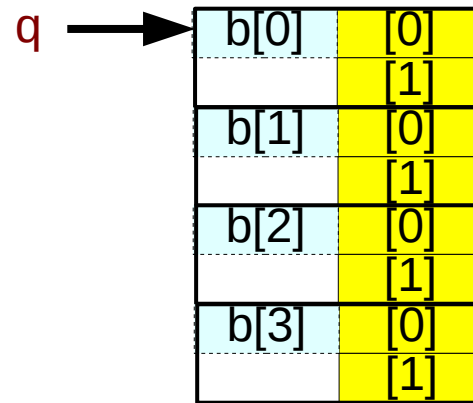
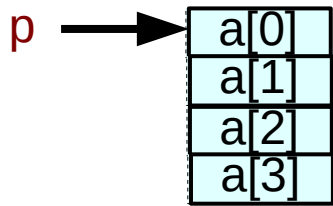


```
int d[4][2][3][4];  
int (*s)[2][3][4];
```

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



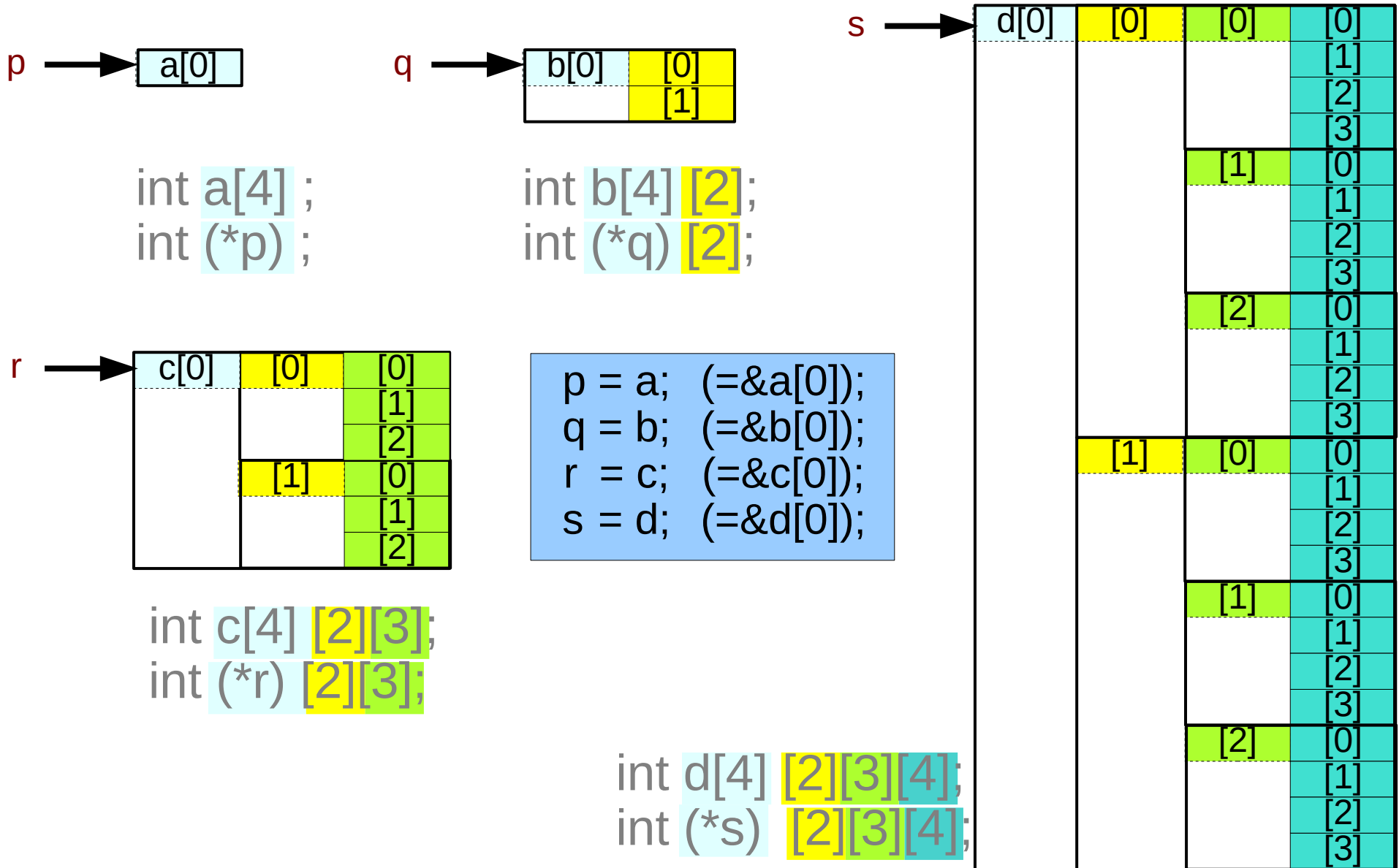
# multi-dimensional array pointers



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

```
int (*p) ;  
int (*q) [2];  
int (*r) [2][3];  
int (*s) [2][3][4];
```

# multi-dimensional array pointers



# multi-dimensional array pointers

d[1]	[0]	[0]	[0]
			[1]
			[2]
			[3]
		[1]	[0]
			[1]
			[2]
			[3]
		[2]	[0]
			[1]
			[2]
			[3]
	[1]	[0]	[0]
			[1]
			[2]
			[3]
		[1]	[0]
			[1]
			[2]
			[3]
		[2]	[0]
			[1]
			[2]
			[3]

d[2]	[0]	[0]	[0]
			[1]
			[2]
			[3]
		[1]	[0]
			[1]
			[2]
			[3]
		[2]	[0]
			[1]
			[2]
			[3]
	[1]	[0]	[0]
			[1]
			[2]
			[3]
		[1]	[0]
			[1]
			[2]
			[3]
		[2]	[0]
			[1]
			[2]
			[3]

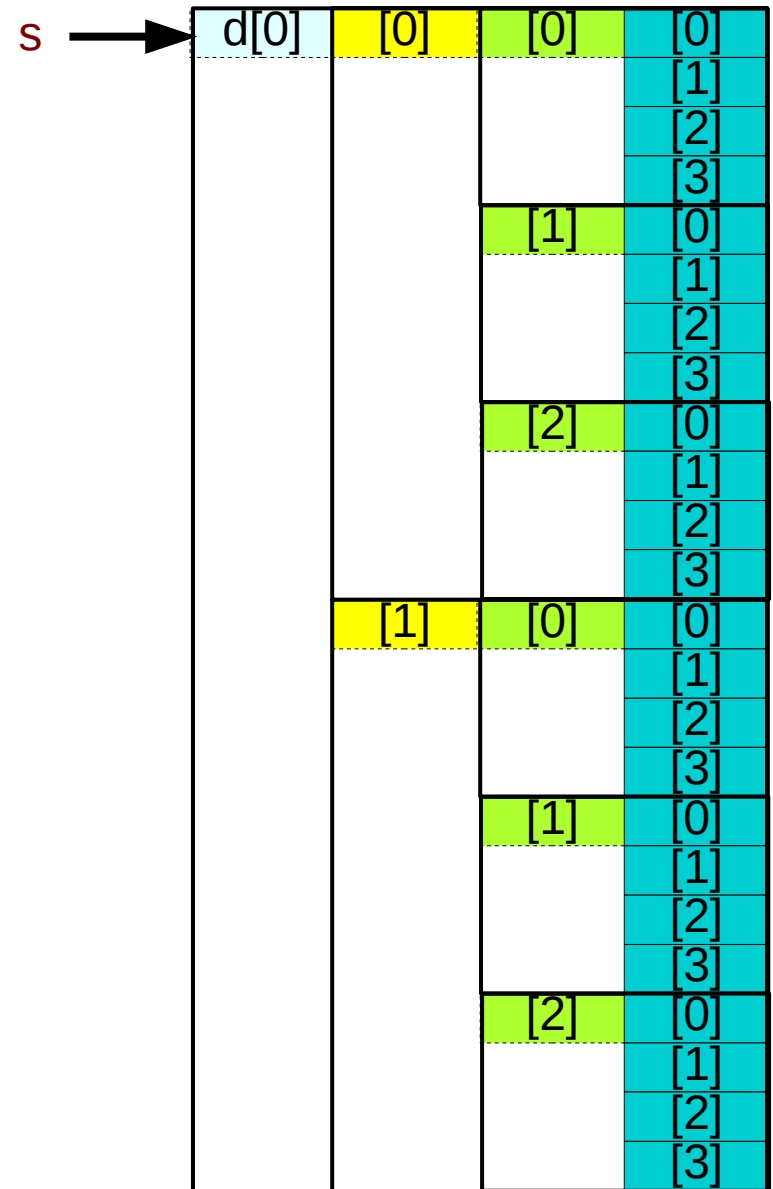
d[3]	[0]	[0]	[0]
			[1]
			[2]
			[3]
		[1]	[0]
			[1]
			[2]
			[3]
		[2]	[0]
			[1]
			[2]
			[3]
	[1]	[0]	[0]
			[1]
			[2]
			[3]
		[1]	[0]
			[1]
			[2]
			[3]
		[2]	[0]
			[1]
			[2]
			[3]

# multi-dimensional array pointers

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

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

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



# To pass 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], ...);

## 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