Applications of Arrays (1A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

2-d Arrays

2-d array definition

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

A matrix view

	col 0	col 1	col 2	col 3
row 0	c [0][0]	c [0][1]	c [0][2]	c [0][3]
row 1	c [1][0]	c [1][1]	c [1][2]	c [1][3]
row 2	c [2][0]	c [2][1]	c [2][2]	c [2][3]
row 3	c [3][0]	c [3][1]	c [3][2]	c [3][3]

2-d array access via pointers

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

```
int      *p = c[0] ;
```

1. recursive pointers

2. linear array pointers

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

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

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

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

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

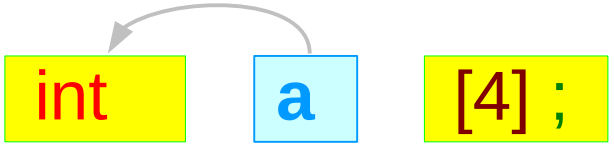
```
int **q;
```

```
p[ i*4 + j ]
```

```
*(p+ i*4 + j )
```

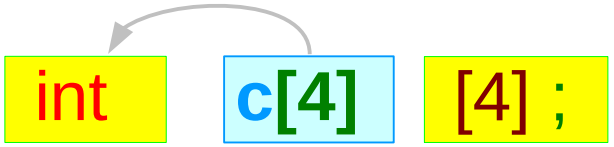
1-d array names

```
int         a [4];

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



The value of **a** is the starting address of *a 4 element array*

a: pointer to the first element



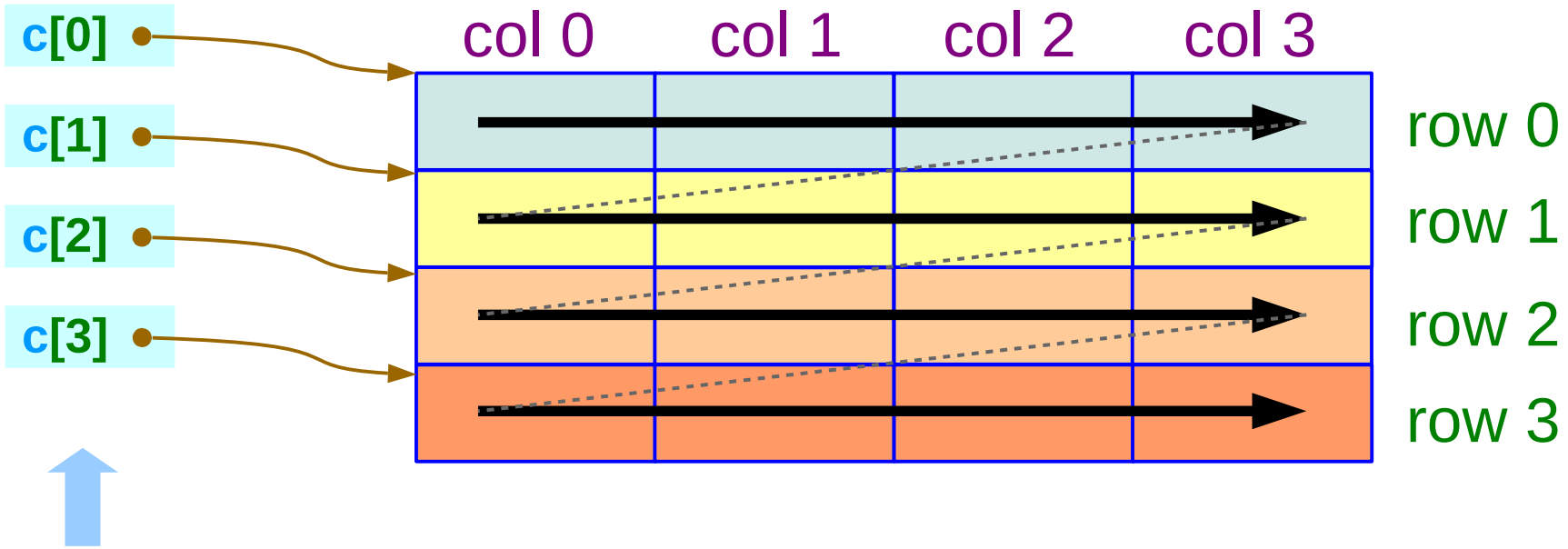
Each value of **c[i]** is the starting address of *a 4 element array*

c[i]: pointer to the first element

2-d array as a matrix

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

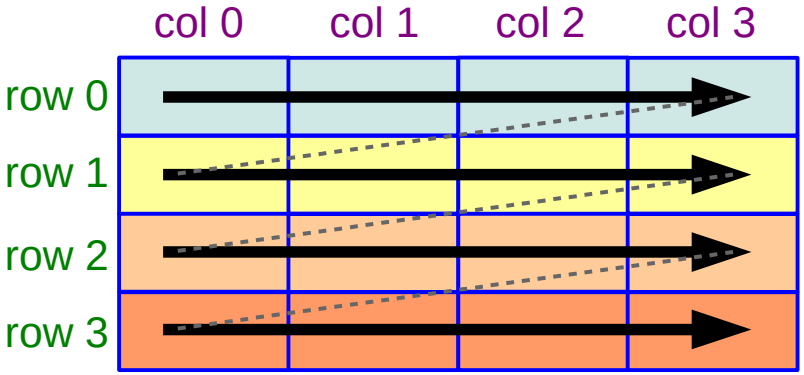
row major ordering

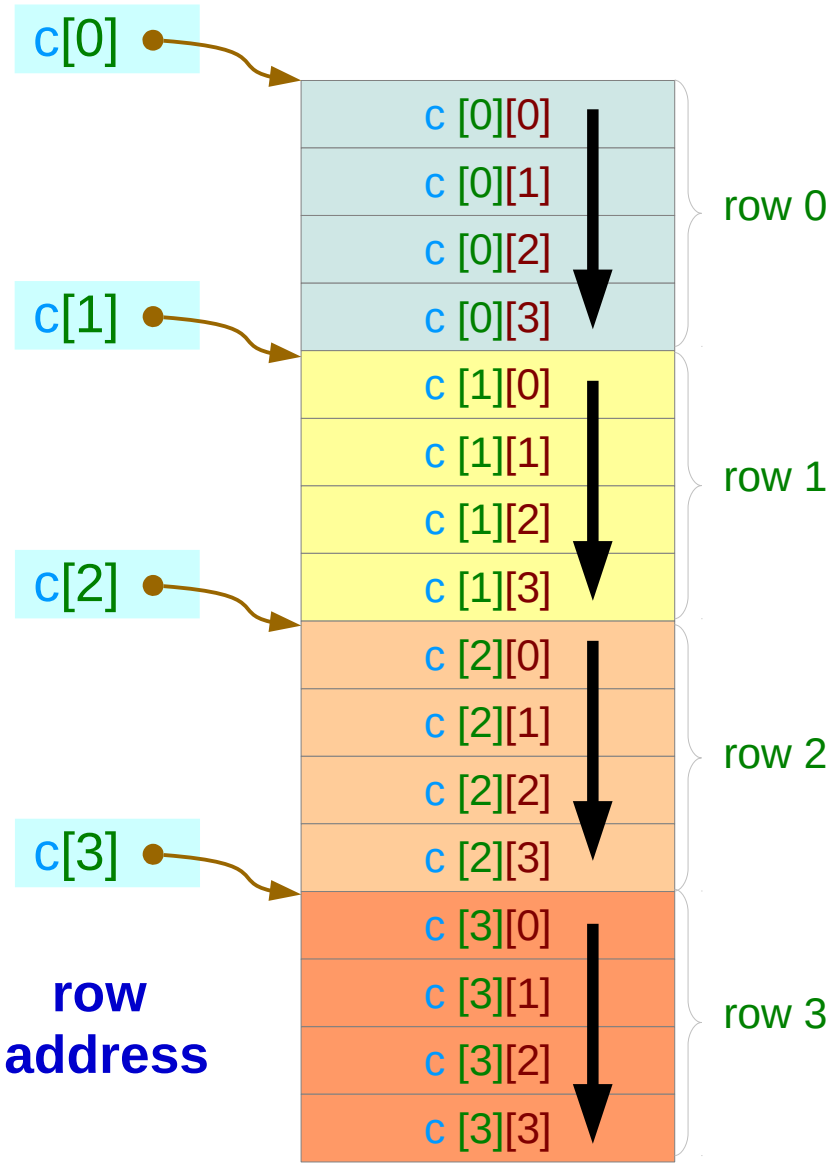


consider each c[i] as a pointer to the first element of each 4 element array

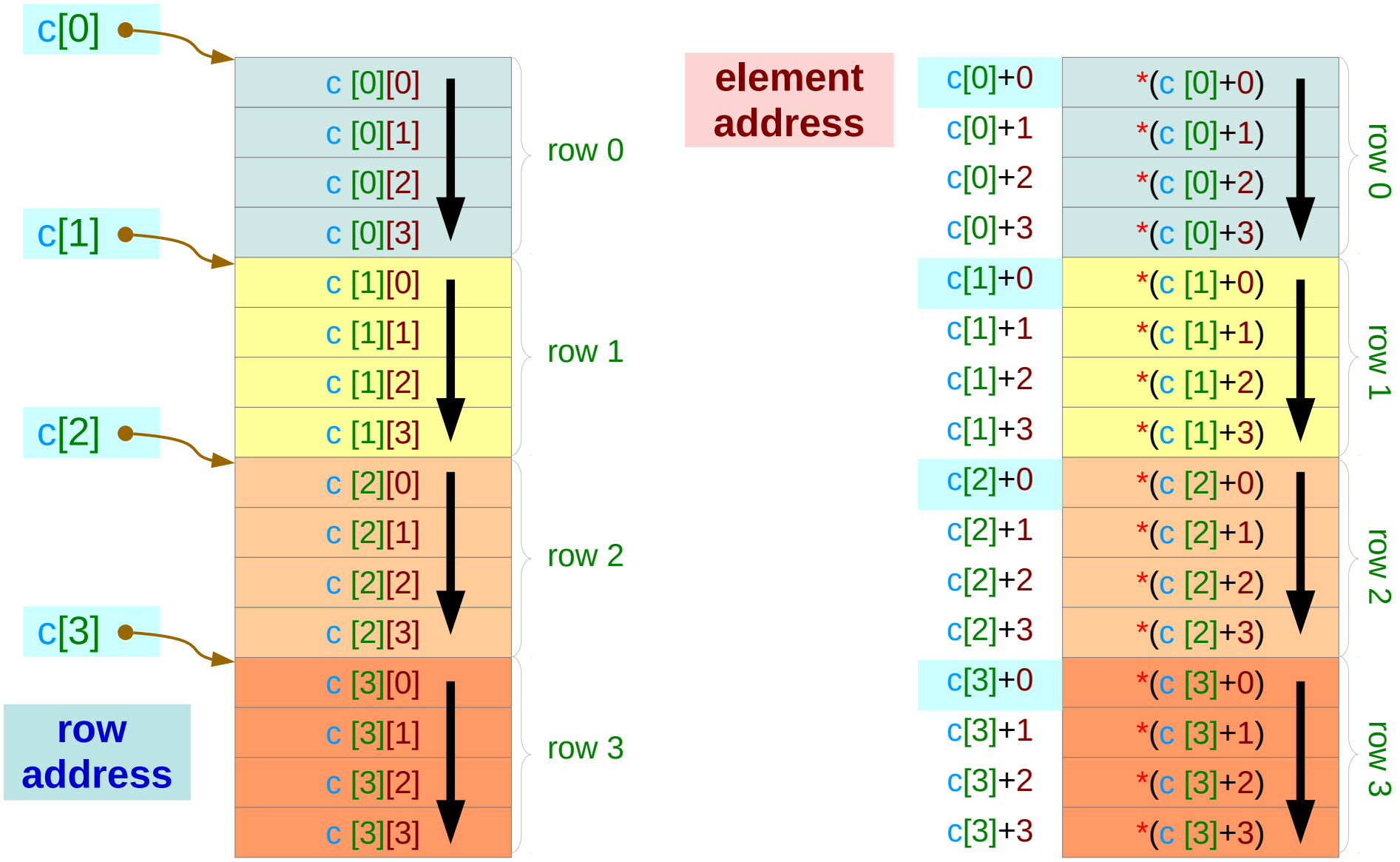
2-d array stored as a linear array

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

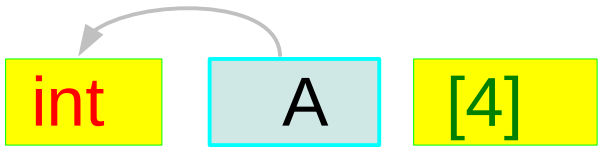


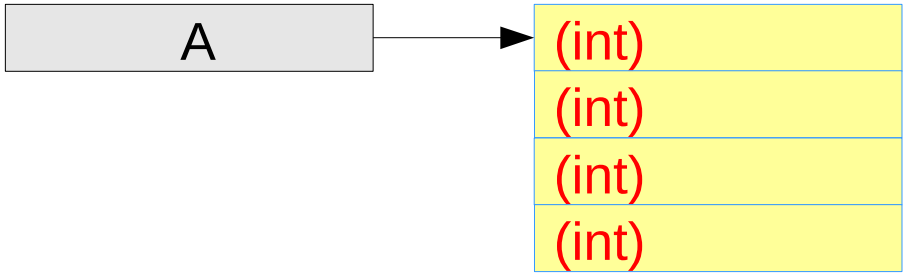


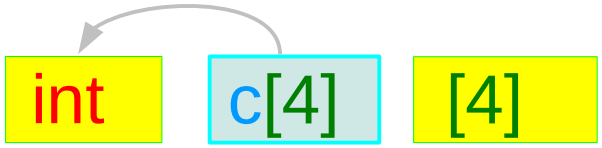
Row Address and Element Address

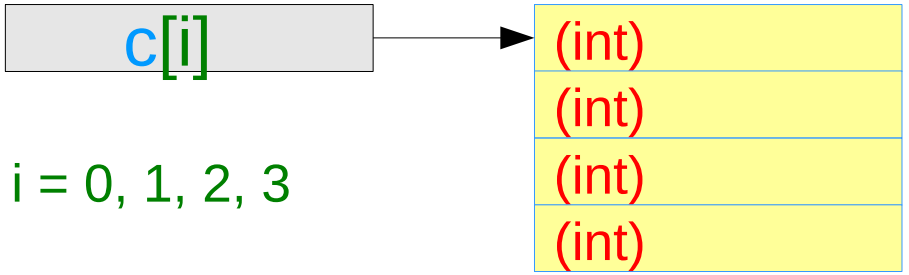


Addresses of **1-d** arrays with 4 integer elements





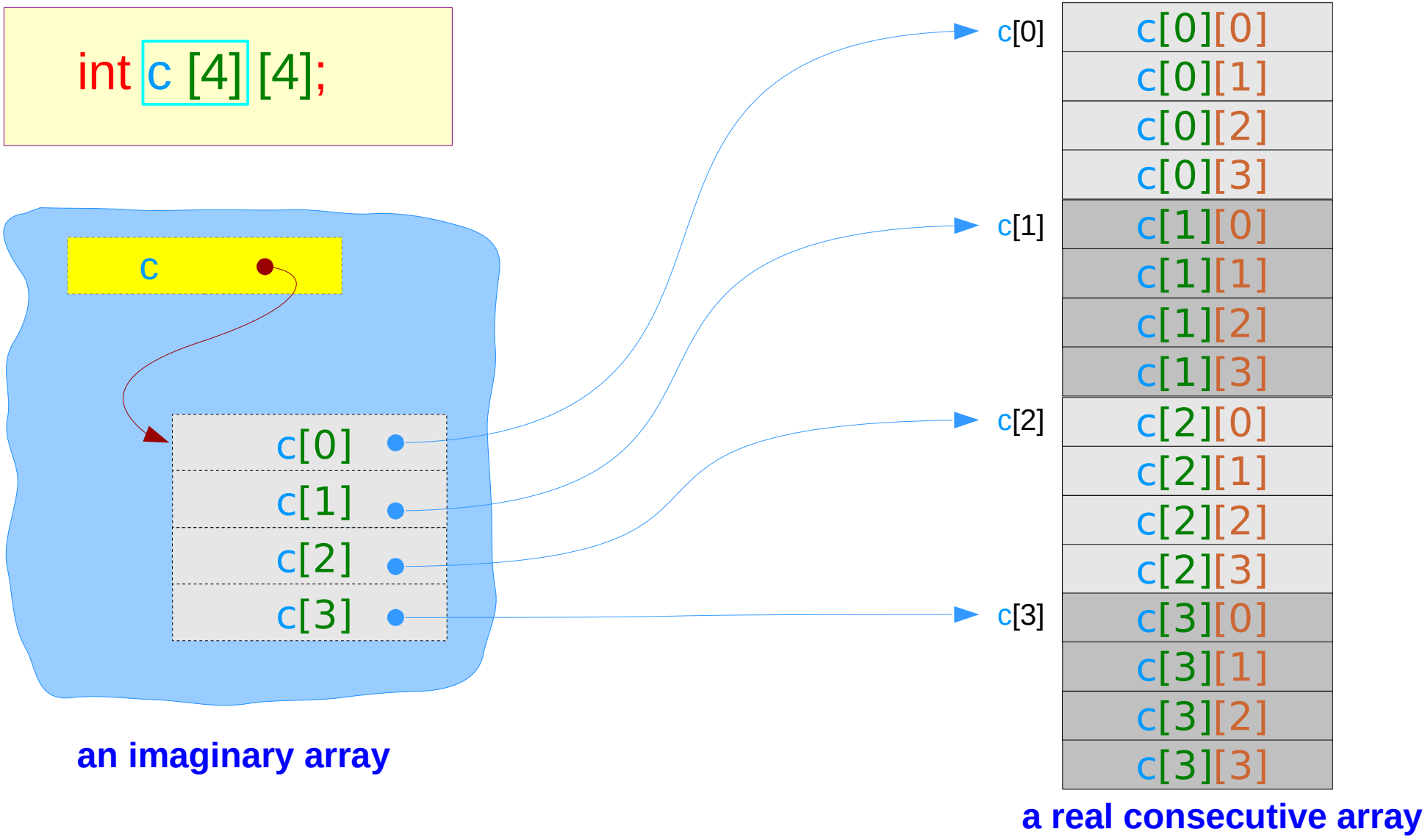




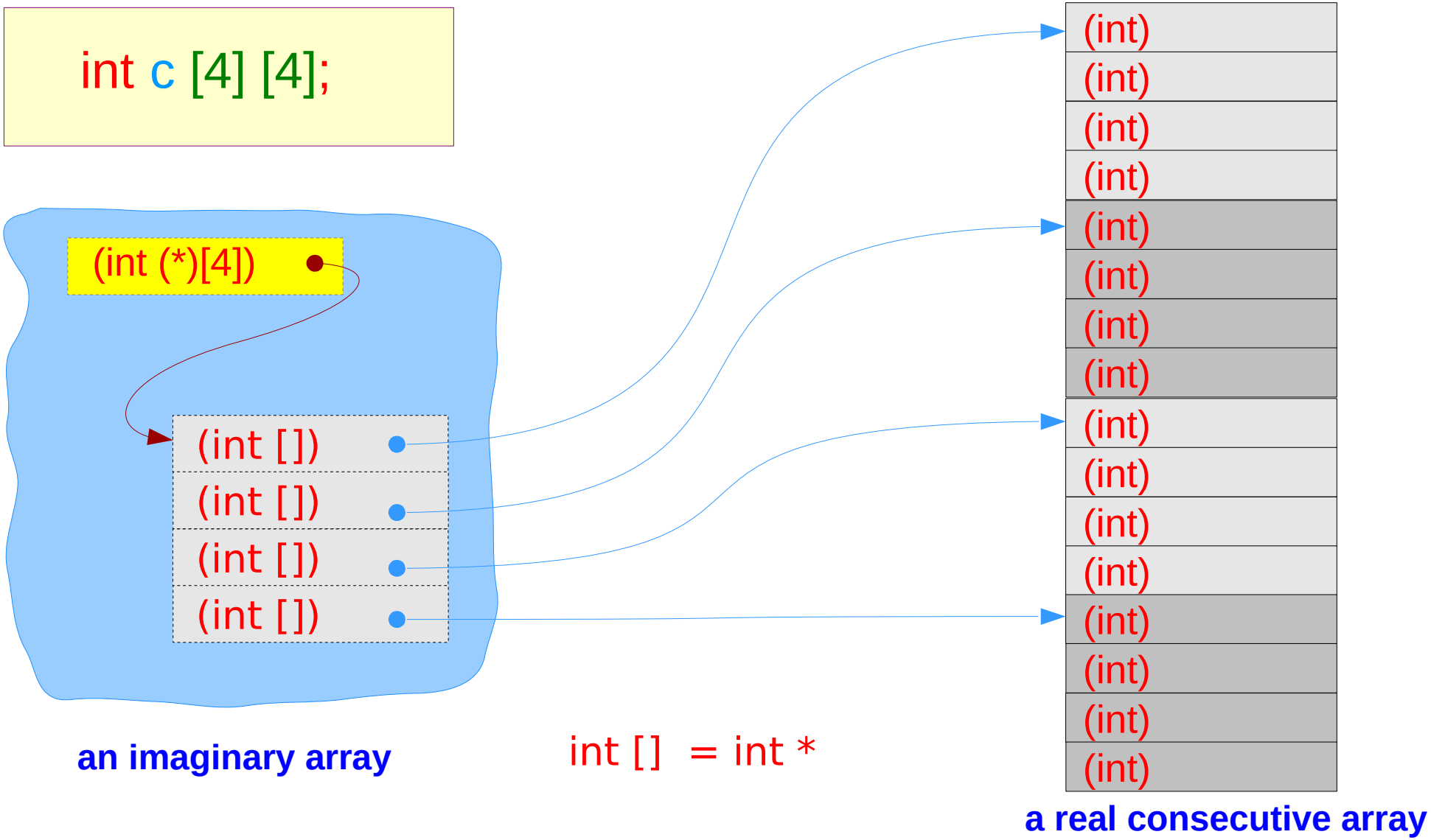
i = 0, 1, 2, 3

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

A **2-d** Array – a variable view

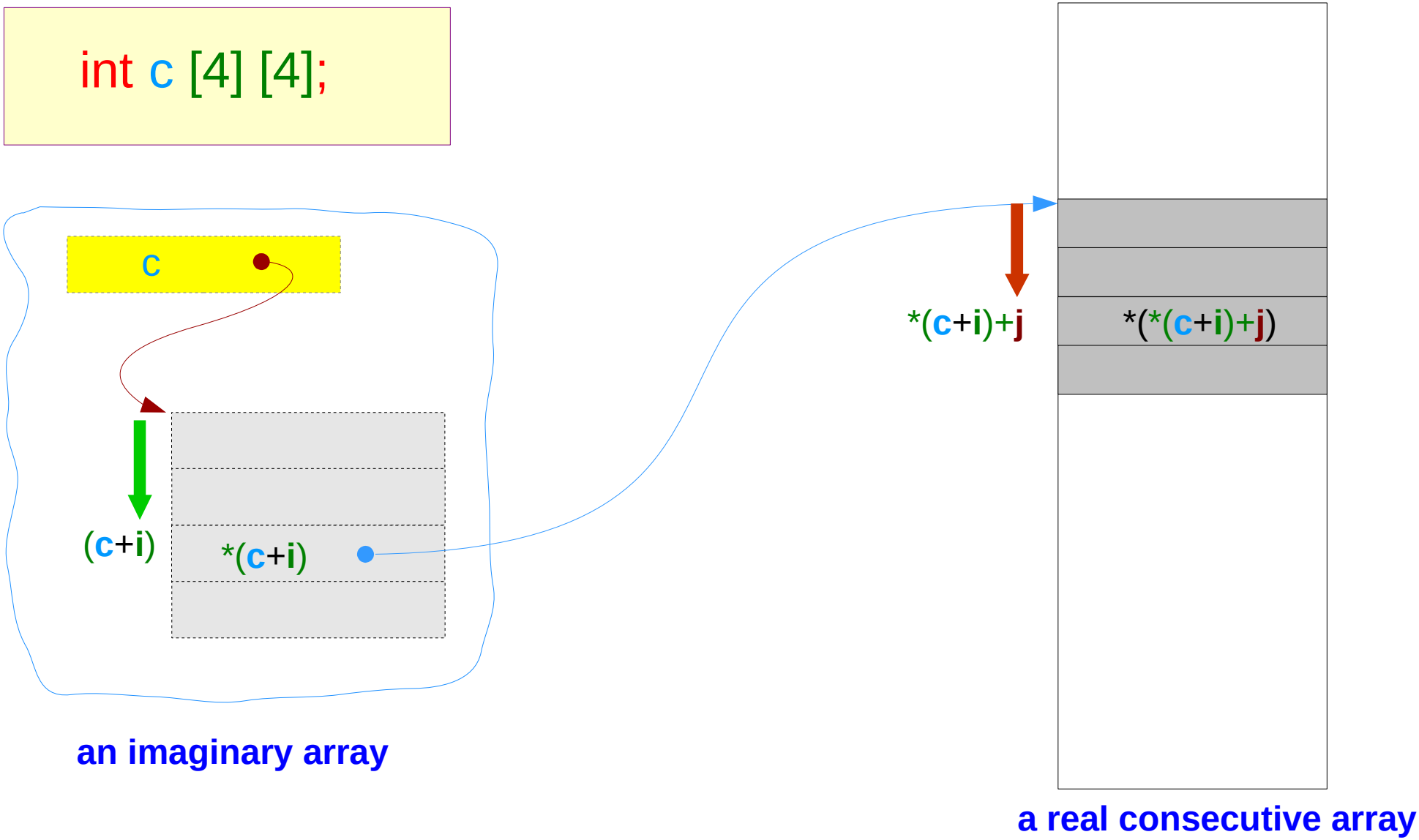


A **2-d** Array – a type view



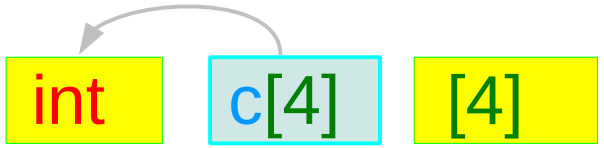
A **2-d** Array – an index view

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



2-d array access via a double indirection

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



(c [i])[j] → (*(c+i))[j] → *(*(c+i)+j)

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

contiguous memory
locations are assumed

(_)[j] = *((_)+j)

contiguous memory
locations are assumed

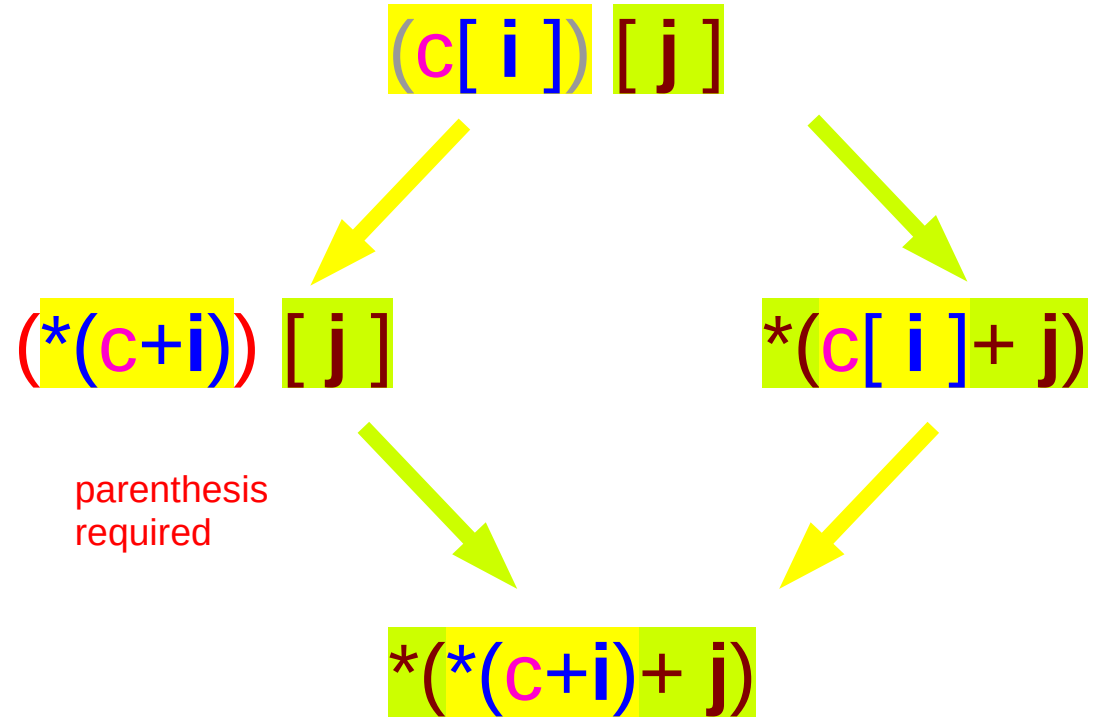
Recursive Pointer Conversions

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

c[i] = *(c+i) hold row addresses

assumption

contiguous memory
locations are assumed



array pointer

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

parenthesis
required

double pointer

```
int **q;
```

2-d array access via recursive pointers (double indirection)

c[i] [j]

➡ (*(c+i))[j]

➡ *(*(c+i)+j)

*(c+i) : row address

*(c+i)+j : element address

first select a row, then a column

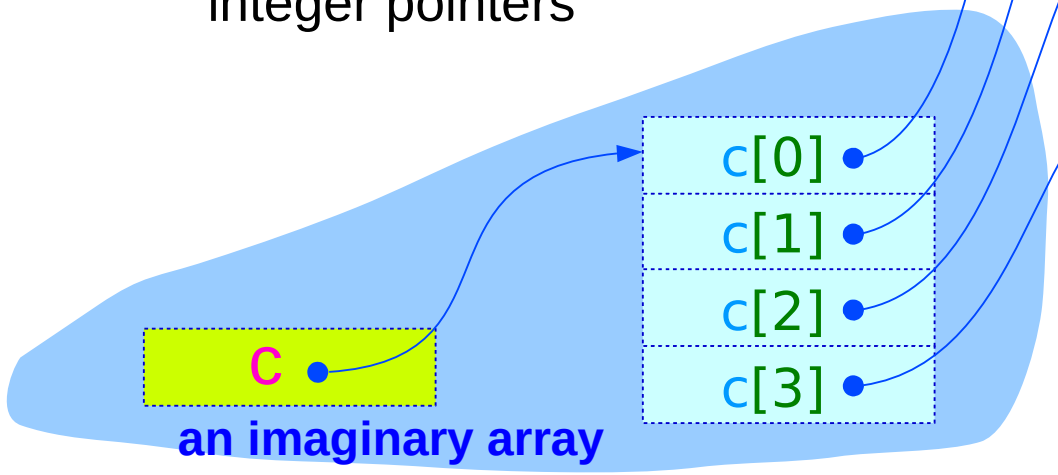
c[0]+0 = *(c+0)+0	c[0][0]
c[0]+1 = *(c+0)+1	c[0][1]
c[0]+2 = *(c+0)+2	c[0][2]
c[0]+3 = *(c+0)+3	c[0][3]
c[1]+0 = *(c+1)+0	c[1][0]
c[1]+1 = *(c+1)+1	c[1][1]
c[1]+2 = *(c+1)+2	c[1][2]
c[1]+3 = *(c+1)+3	c[1][3]
c[2]+0 = *(c+2)+0	c[2][0]
c[2]+1 = *(c+2)+1	c[2][1]
c[2]+2 = *(c+2)+2	c[2][2]
c[2]+3 = *(c+2)+3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

Nested arrays

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

int c[4] [3] ;

c: an array of integer pointers



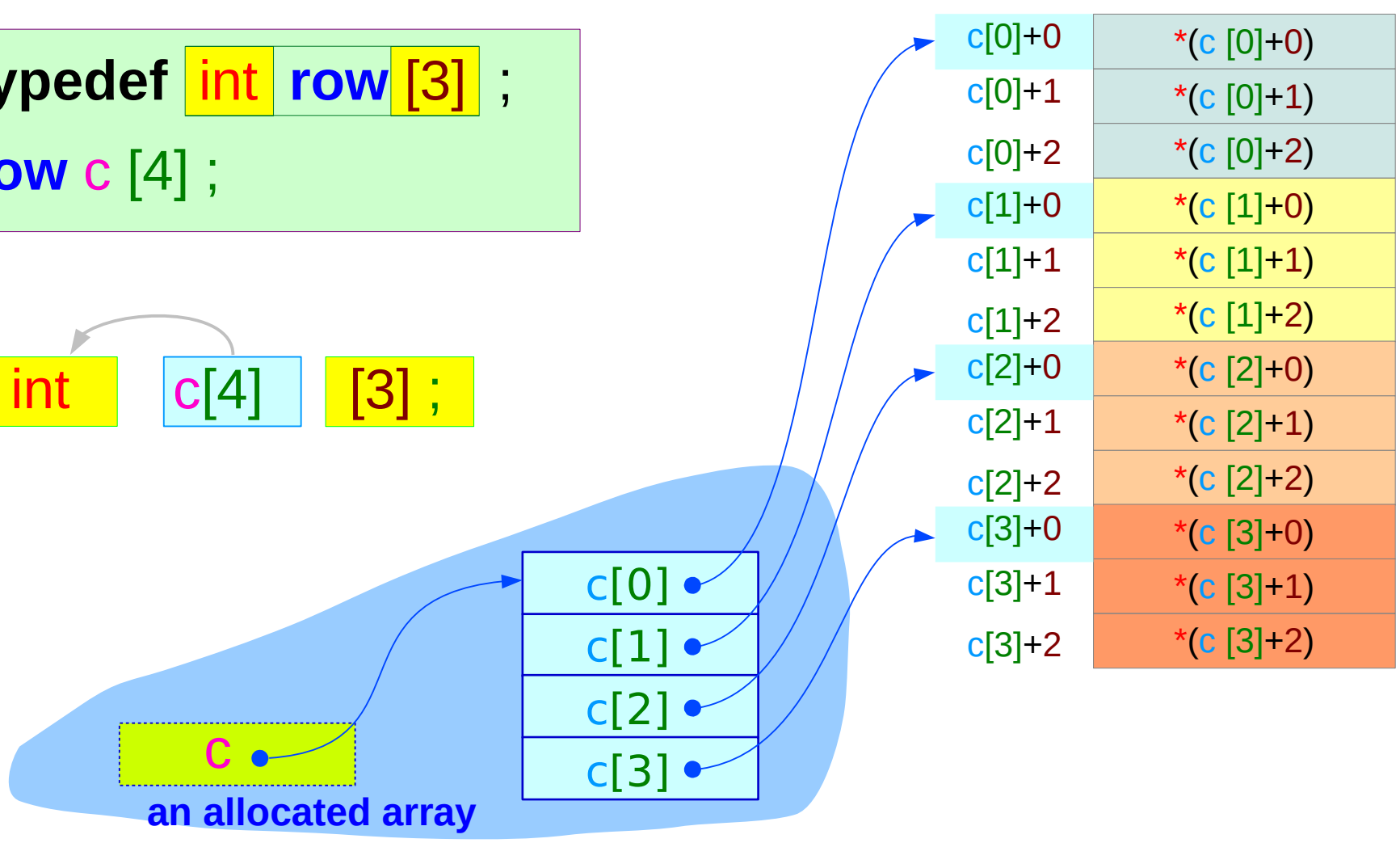
c[0]+0	*(c [0]+0)
c[0]+1	*(c [0]+1)
c[0]+2	*(c [0]+2)
c[1]+0	*(c [1]+0)
c[1]+1	*(c [1]+1)
c[1]+2	*(c [1]+2)
c[2]+0	*(c [2]+0)
c[2]+1	*(c [2]+1)
c[2]+2	*(c [2]+2)
c[3]+0	*(c [3]+0)
c[3]+1	*(c [3]+1)
c[3]+2	*(c [3]+2)

int c[4][3] does not allocate c[0], c[1], c[2], c[3] in the memory

Nested array declared explicitly

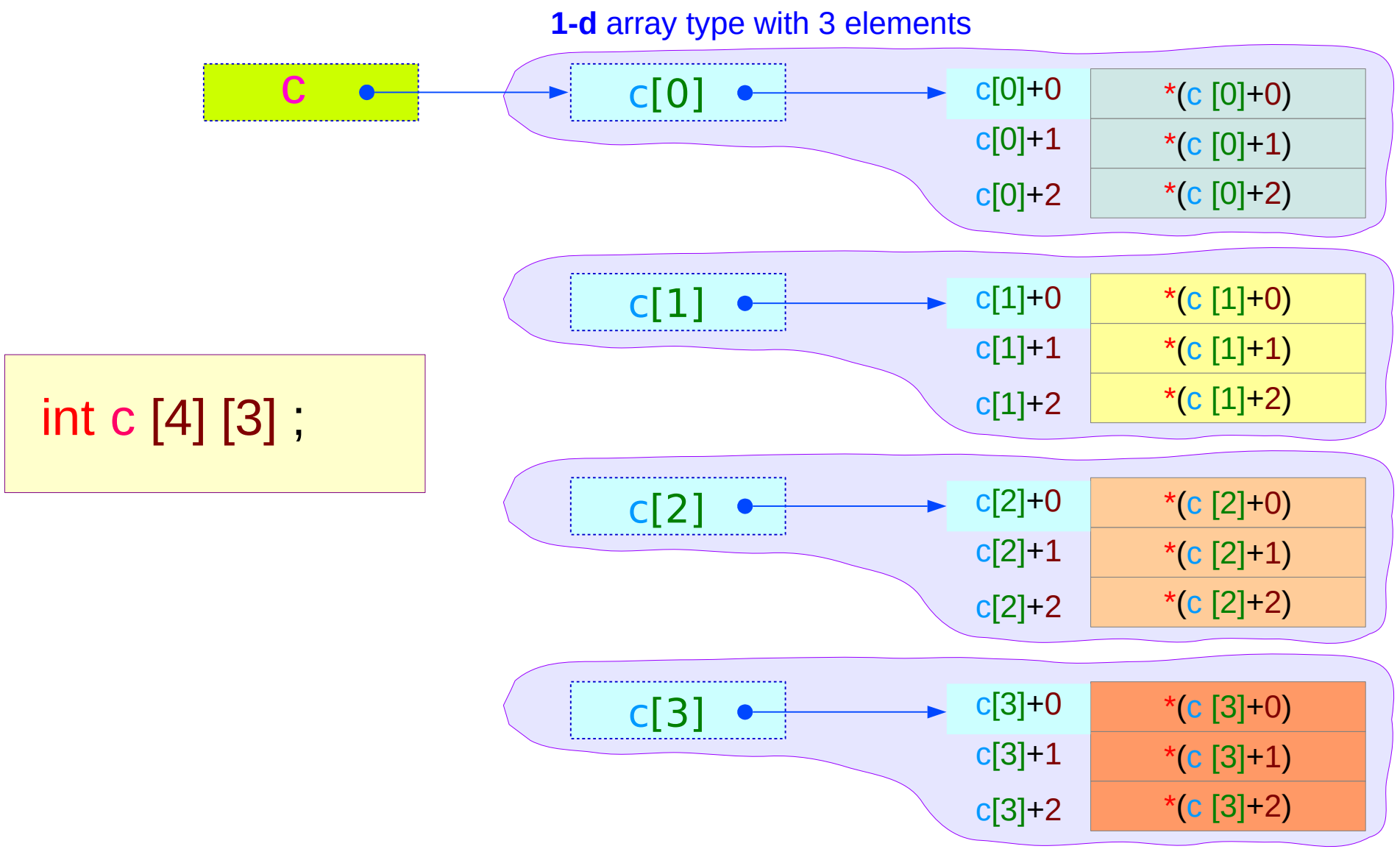
```
typedef int row [3] ;
row c [4] ;
```

int c[4] [3] ;



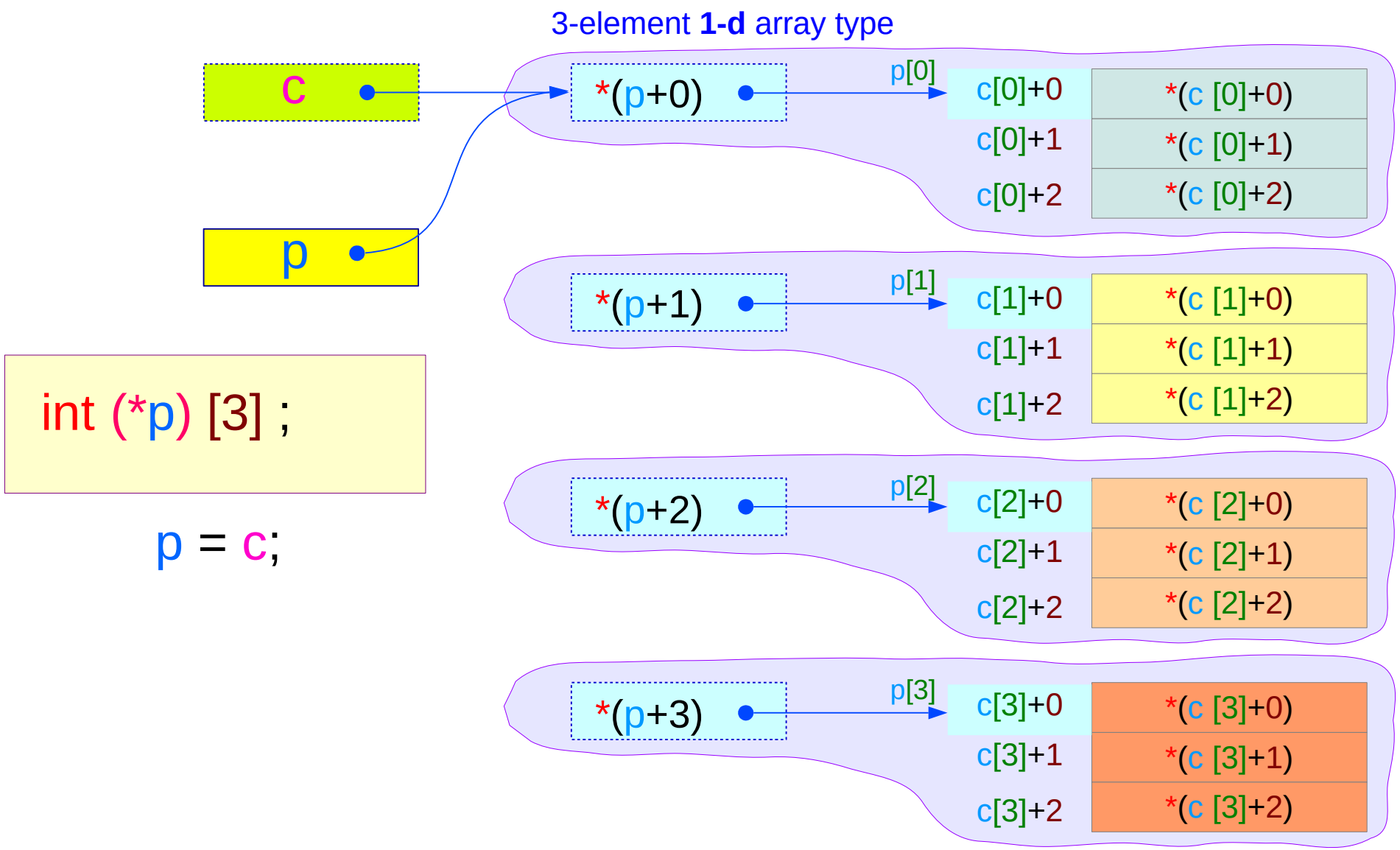
row c[4] allocates 4 array pointers in the memory

2-d Array : rows of **1-d** arrays




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

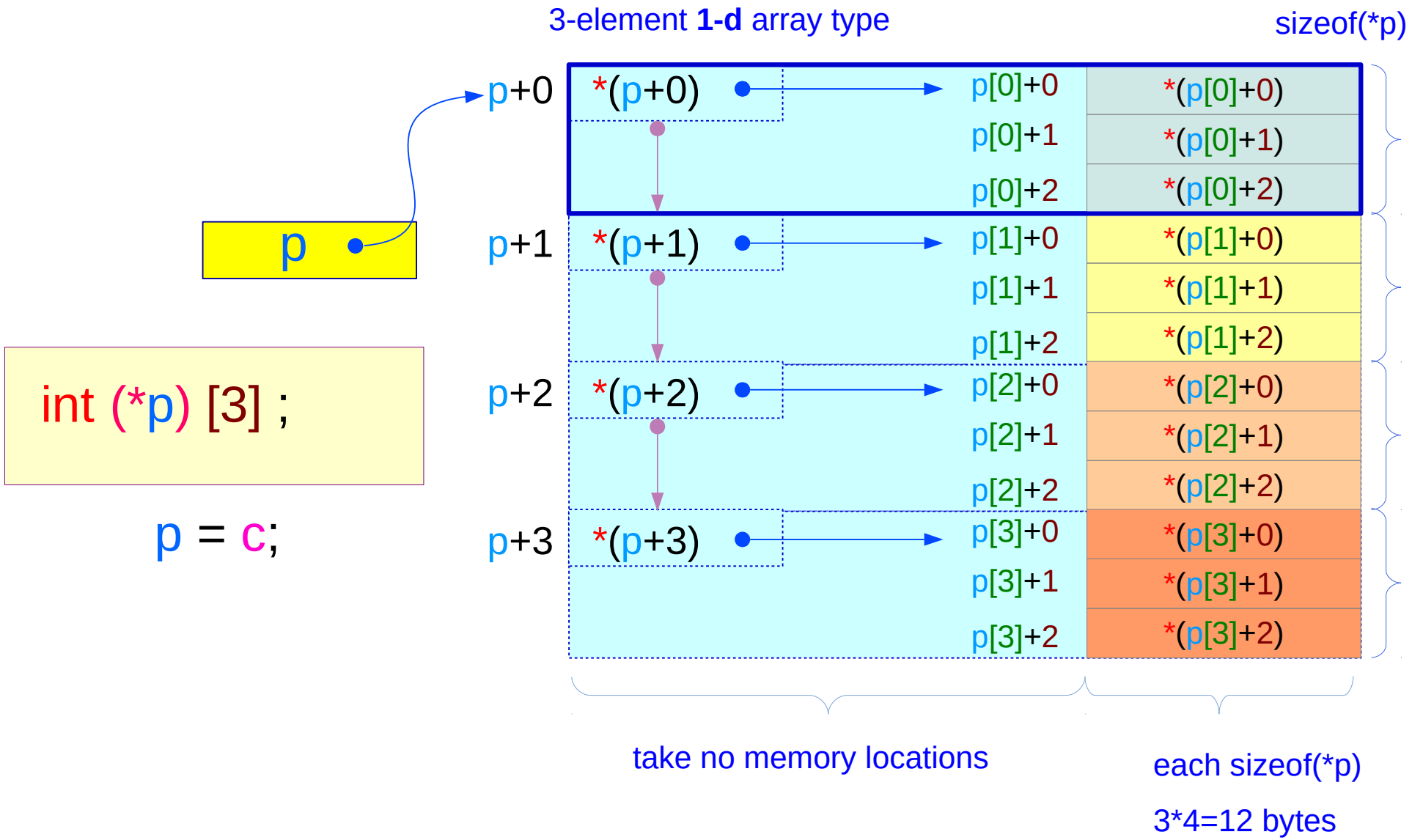
Pointer to **1-d** arrays




```
int (*p) [3] ;
```

p = c;

Incrementing the **1-d** array pointer **p**

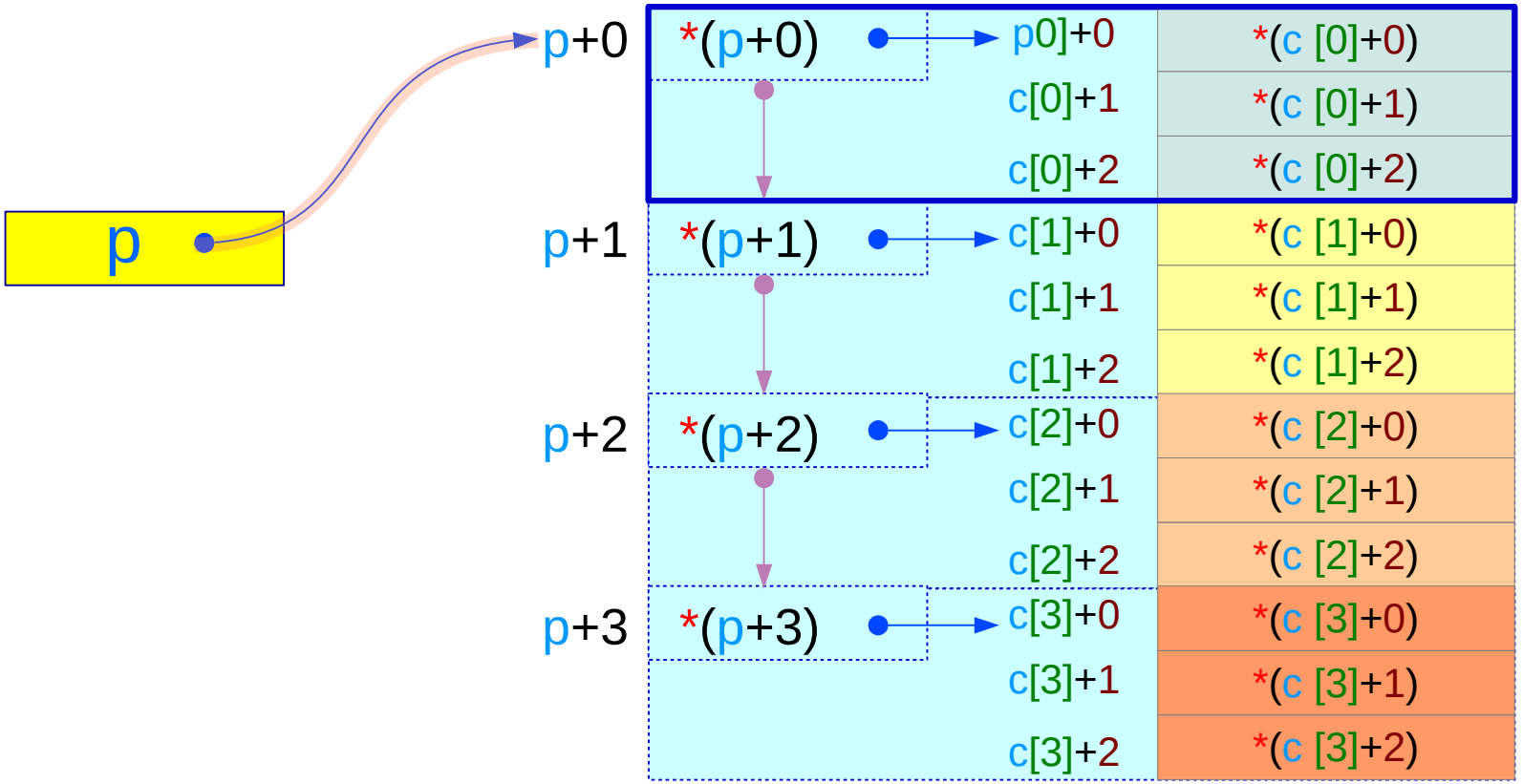


2-d array access via a 1-d array pointer

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

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

```
p = c;
```



2-d array access via a 1-d array pointer

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

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

(c [i])[j] → (*(c+i))[j] → *(*(c+i)+j)

p = c

p[0]=c[0],
p[1]=c[1],
p[2]=c[2],
p[3]=c[3];

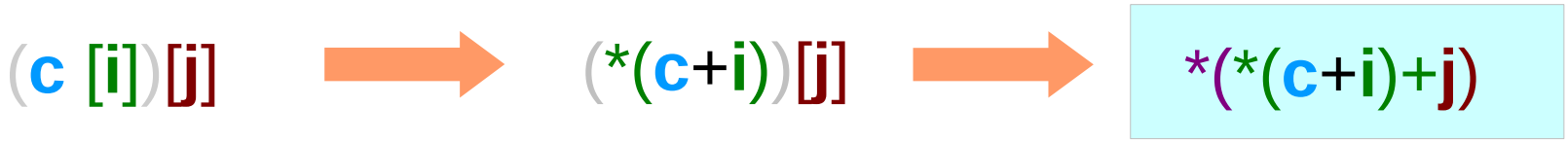
equivalence

(p [i])[j] → (*(p+i))[j] → *(*(p+i)+j)

2-d array access via a double pointer

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

```
int **p, *q[4];
```




```
p = q;
```

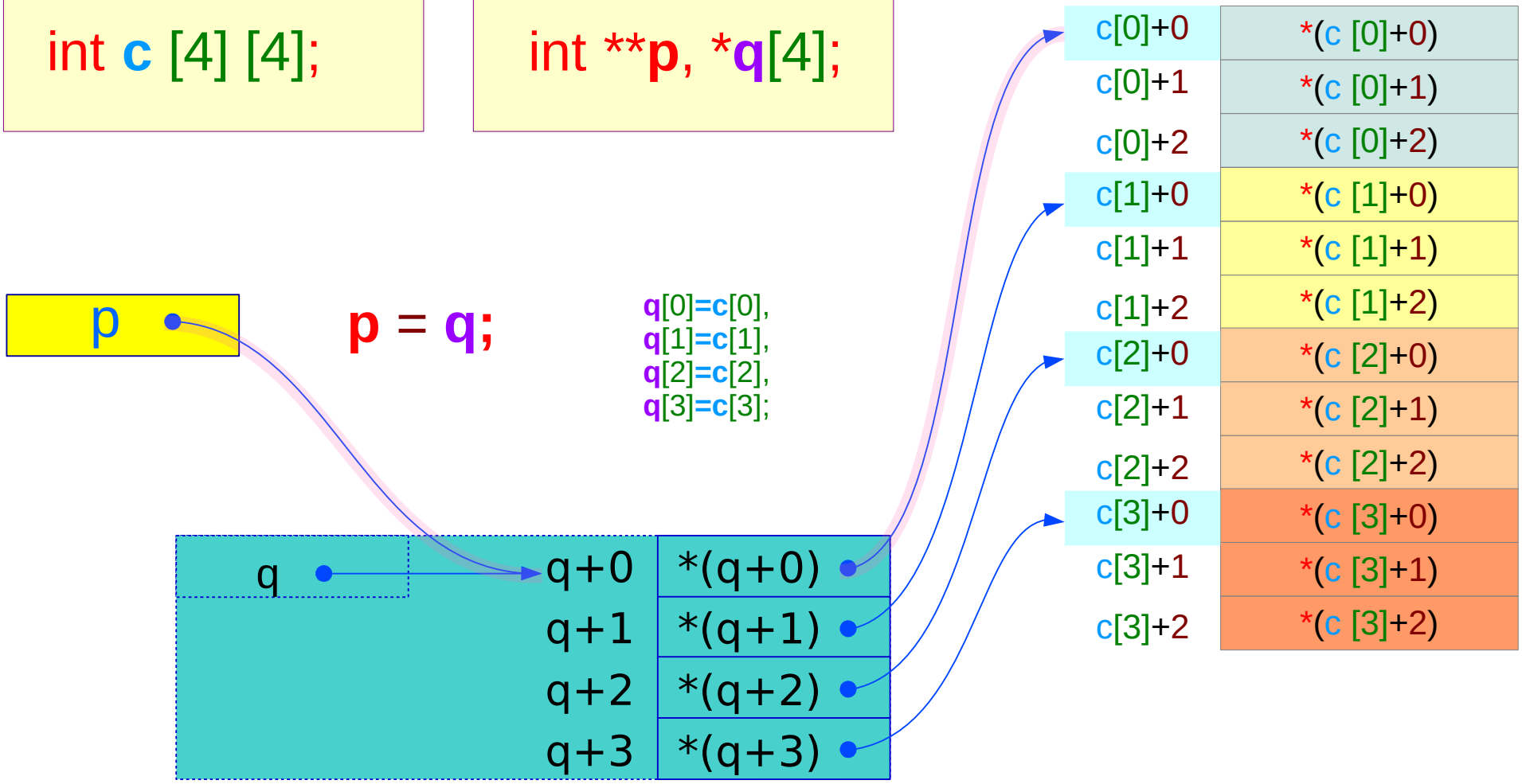
```
q[0]=c[0],
q[1]=c[1],
q[2]=c[2],
q[3]=c[3];
```

must be allocated and initialized

2-d array access via a double pointer

int c [4] [4];

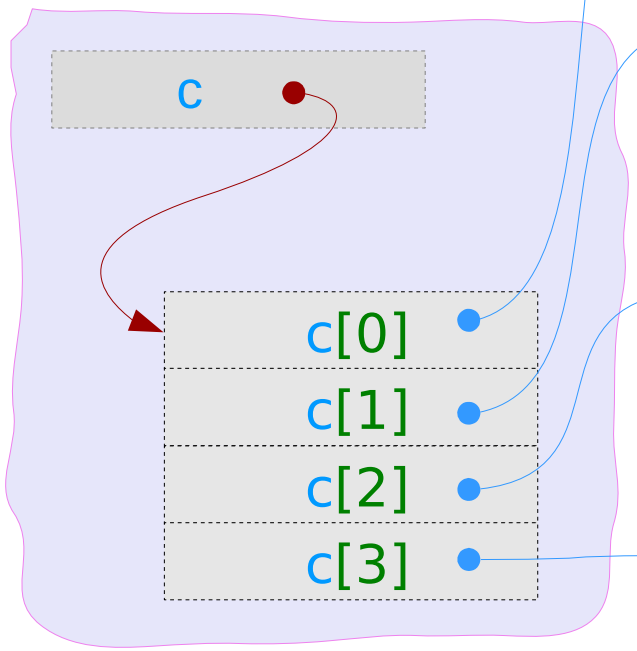
int **p, *q[4];



to use a double pointer p, a pointer array q must be allocated and initialized

A **2-d** array stored as a **1-d** array (row major order)

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

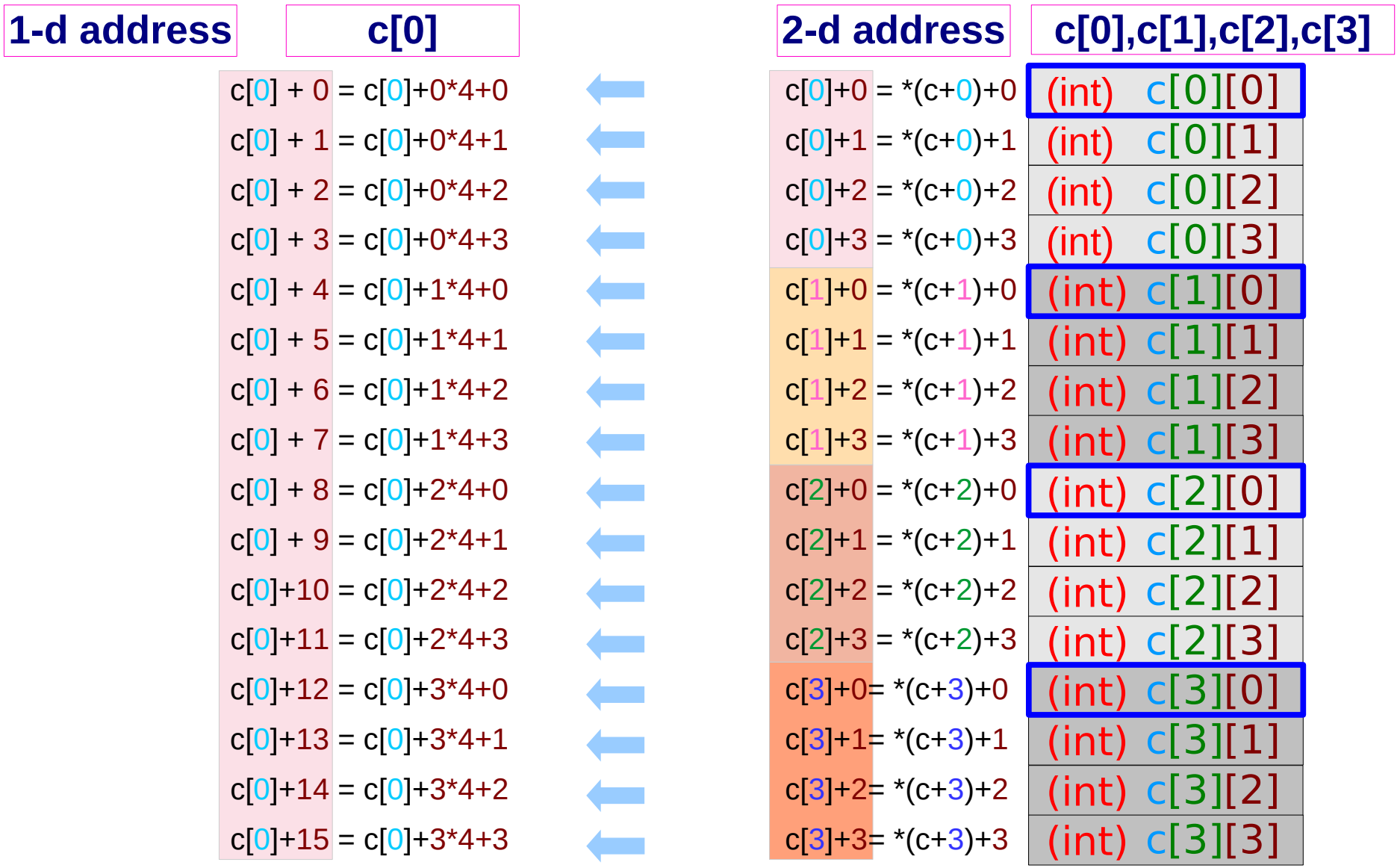


an imaginary array

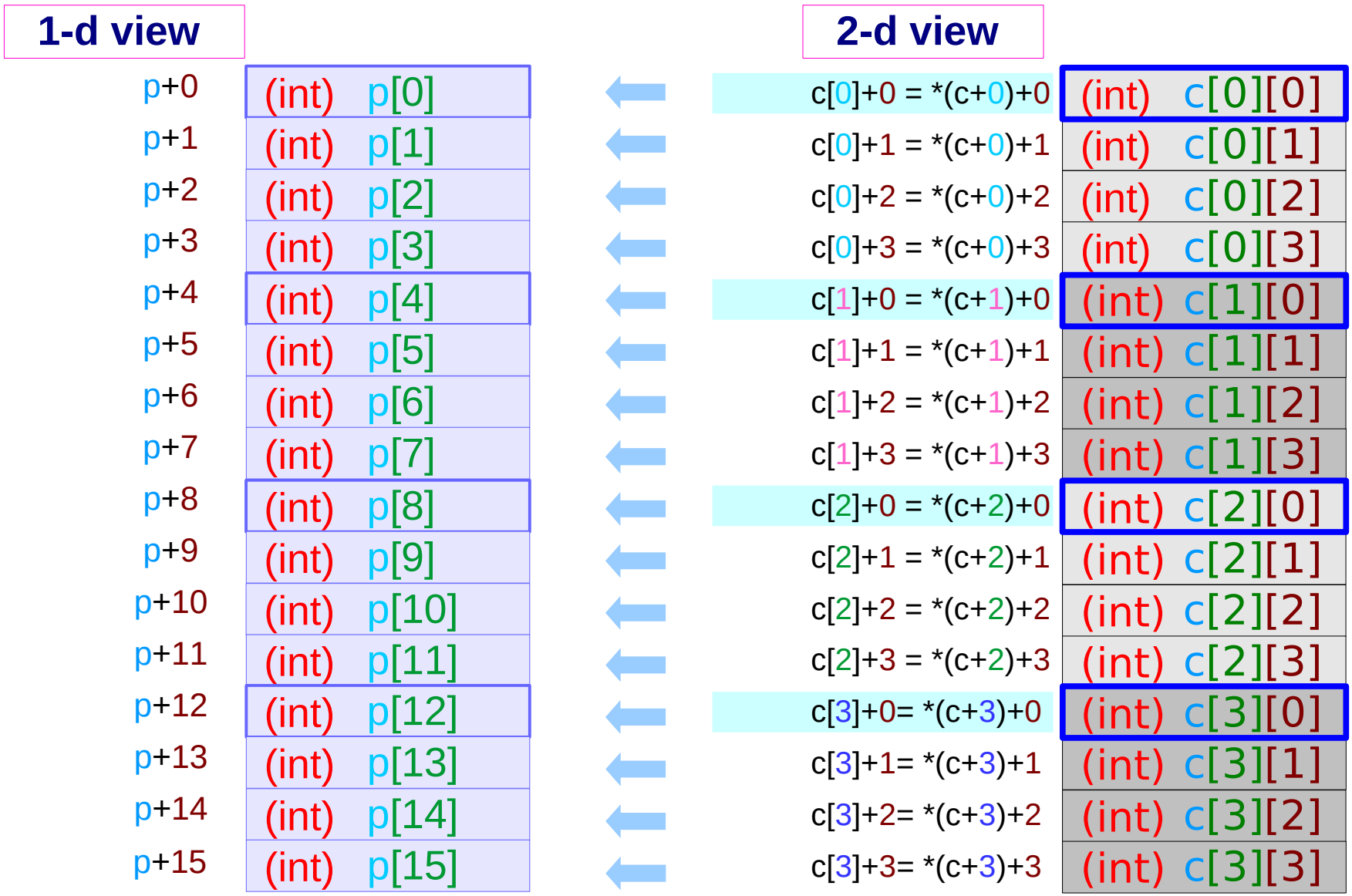
		[i*4+j]
c[0]	c[0][0]	0=[0*4+0]
	c[0][1]	1=[0*4+1]
	c[0][2]	2=[0*4+2]
	c[0][3]	3=[0*4+3]
c[1]	c[1][0]	4=[1*4+0]
	c[1][1]	5=[1*4+1]
	c[1][2]	6=[1*4+2]
	c[1][3]	7=[1*4+3]
c[2]	c[2][0]	8=[2*4+0]
	c[2][1]	9=[2*4+1]
	c[2][2]	10=[2*4+2]
	c[2][3]	11=[2*4+3]
c[3]	c[3][0]	12=[3*4+0]
	c[3][1]	13=[3*4+1]
	c[3][2]	14=[3*4+2]
	c[3][3]	15=[3*4+3]

c[i][j]

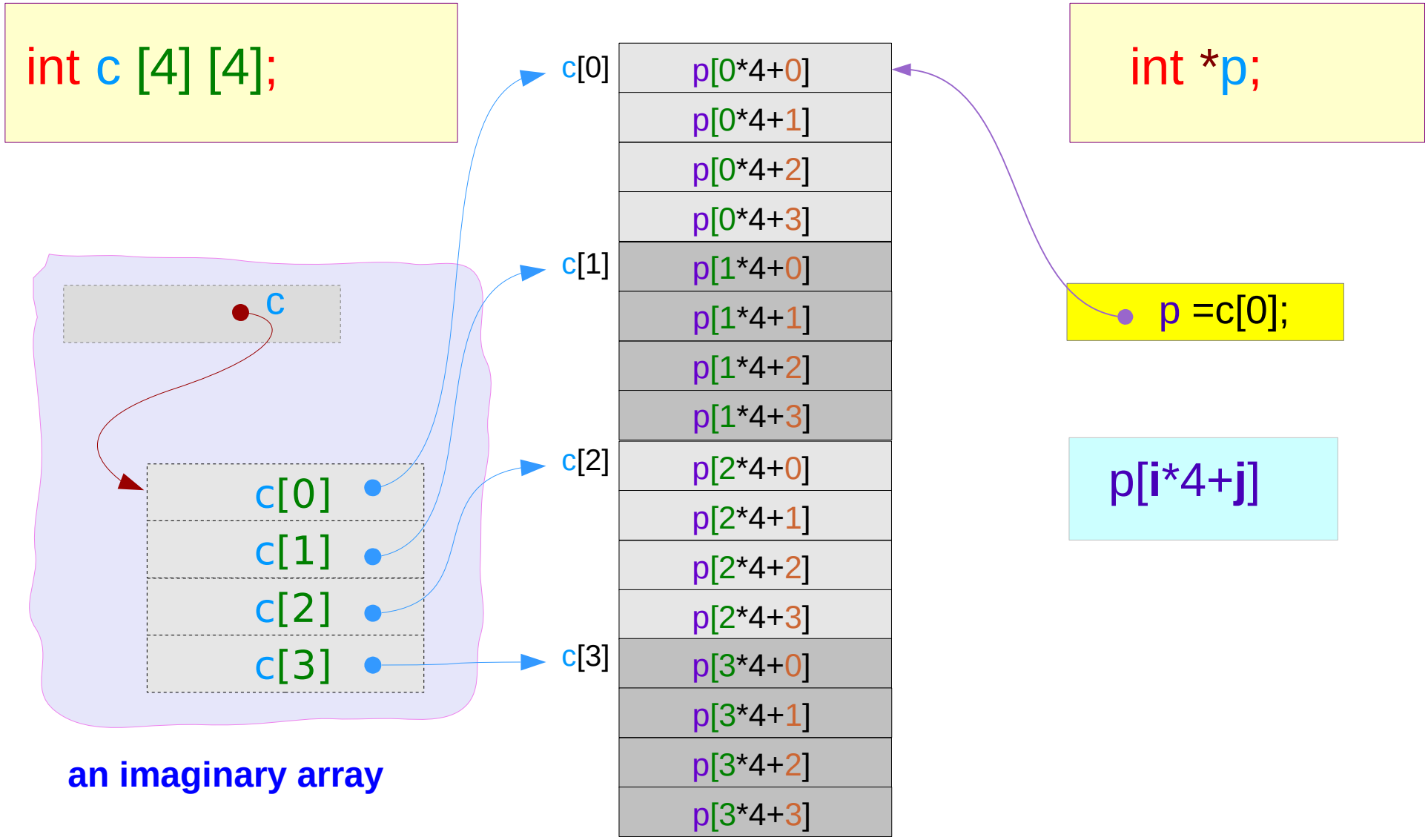
A Linear Memory Address



A linearization of a 2-D array



2-d array access via a single pointer



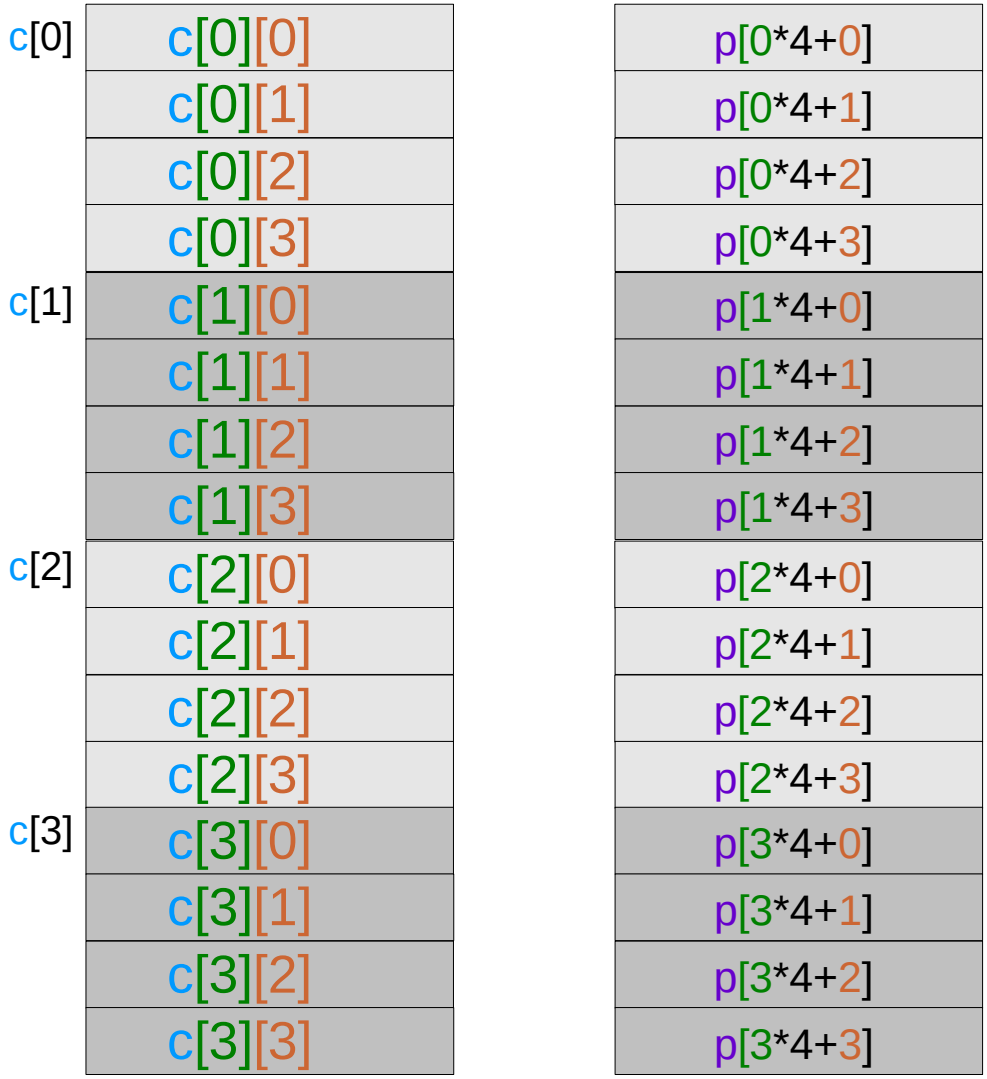
2-d array index vs **1-d** array index

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

```
int *p=c[0];
```

c[i][j]

p[i*4+j]



2-d array access via a single pointer

```
int      *p = c[0] ;
```

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

```
p[ i*4 + j ]
```

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

```
*(p+ i*4 + j)
```

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

```
*(p +k)        i = k / 4;
               j = k % 4;
```

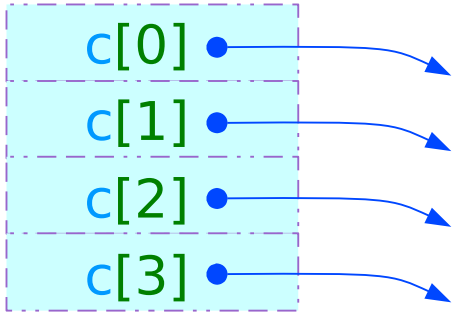
Static Allocation of a 2-d Array

```
int A [4][3];
```

A in %eax,
i in %edx,
j in %ecx

```
sall    $2, %ecx                    ;; j * 4
leal    (%edx, %edx, 2), %edx       ;; i * 3
leal    (%ecx, %edx, 4), %edx       ;; j * 4 + i * 12
movl    (%eax, %edx), %eax          ;; read M[ X_A+4(3i +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 [1]+1)
c[1]+2	*(c [1]+2)
c[2]+0	*(c [2]+0)
c[2]+1	*(c [2]+1)
c[2]+2	*(c [2]+2)
c[3]+0	*(c [3]+0)
c[3]+1	*(c [3]+1)
c[3]+2	*(c [3]+2)



The pointer array :
not allocated
in the memory

The name of a 2-d array

```
int         a [4];

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

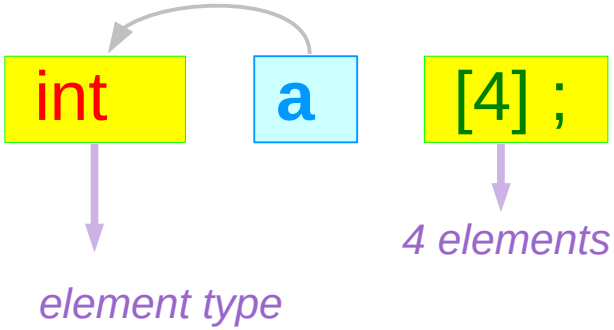
1. the name of the nested array (recursive definition)

2. a double pointer

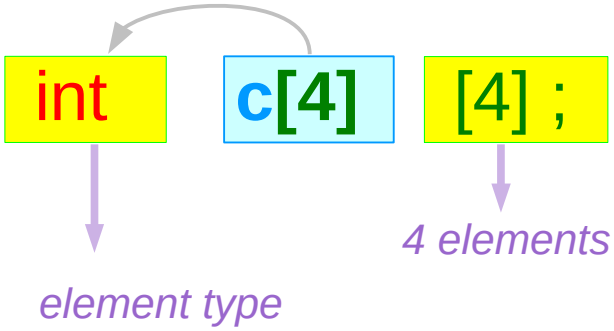
3. a pointer to an array

c: the nested array name

```
int         a [4];

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



The array name **a** holds *the starting address of the 4 integer element array*



c[0], **c[1]**, **c[2]**, **c[3]** holds *the starting address of the 4 integer element array*

The array name **c** is the name of the nested array

c is a double pointer and a **1-d** array pointer

```
int         a [4];

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

((c+0)+0) ➡ **c a double pointer

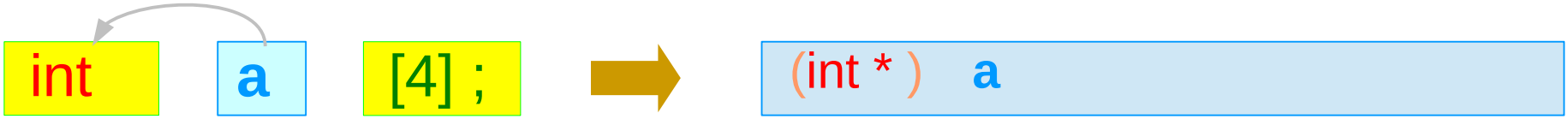
(*(c+0))[j] ➡ (*c)[j] a **1-d** array pointer

c is a double pointer

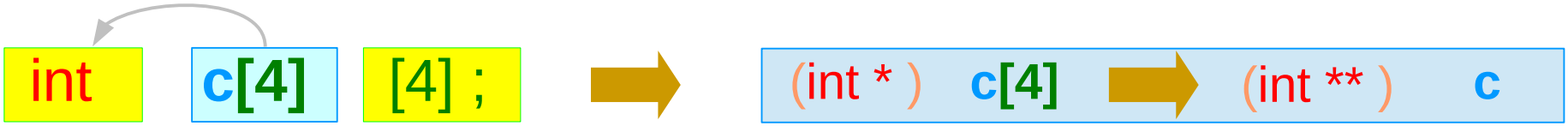
```
int     a [4];

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

**c

((c+0)+0)



a points to an integer data



c[i] points to an integer data

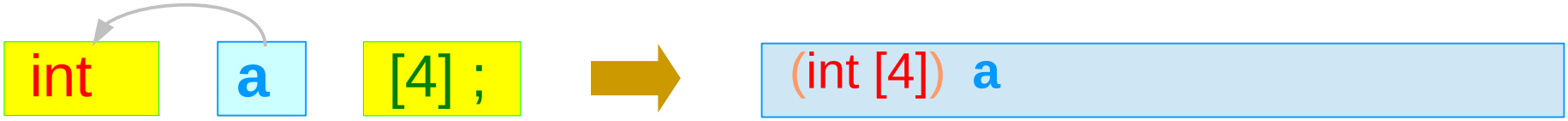
c points to an integer pointer

c is also a pointer to an array

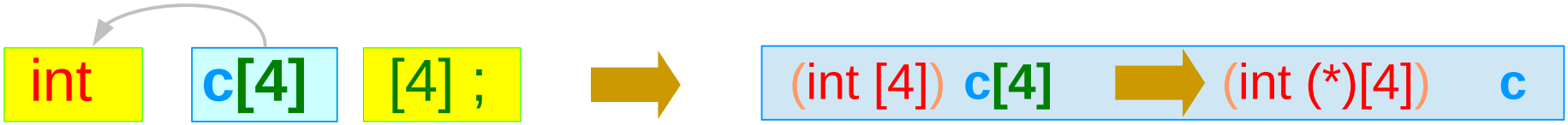
```
int       a [4];

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

(*c)[j]

(*(c+0))[j]



a : an array name



c[i] : array names

c : the pointer array name

Pointer to Arrays

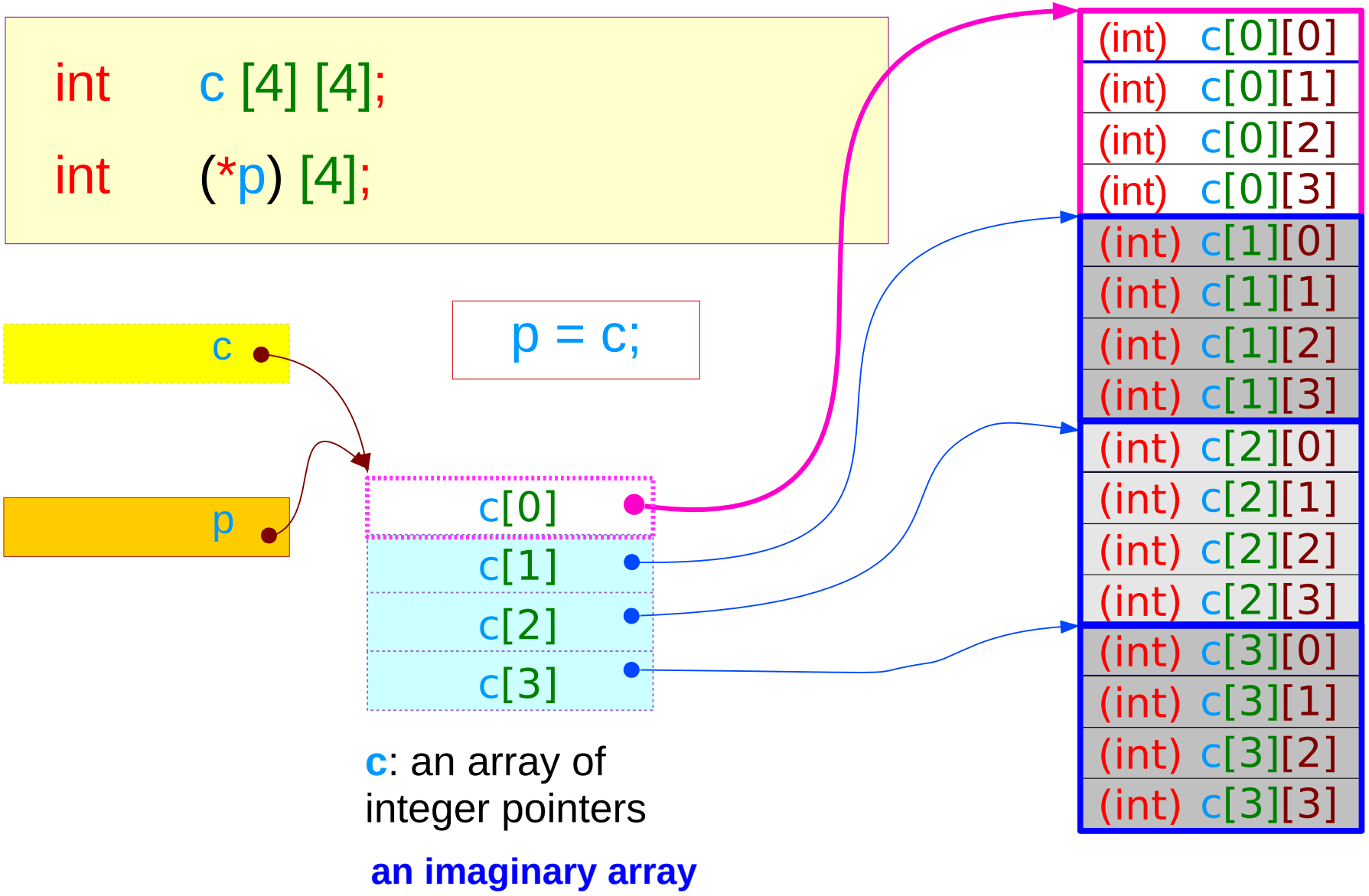
1. pointer to 1-d arrays

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

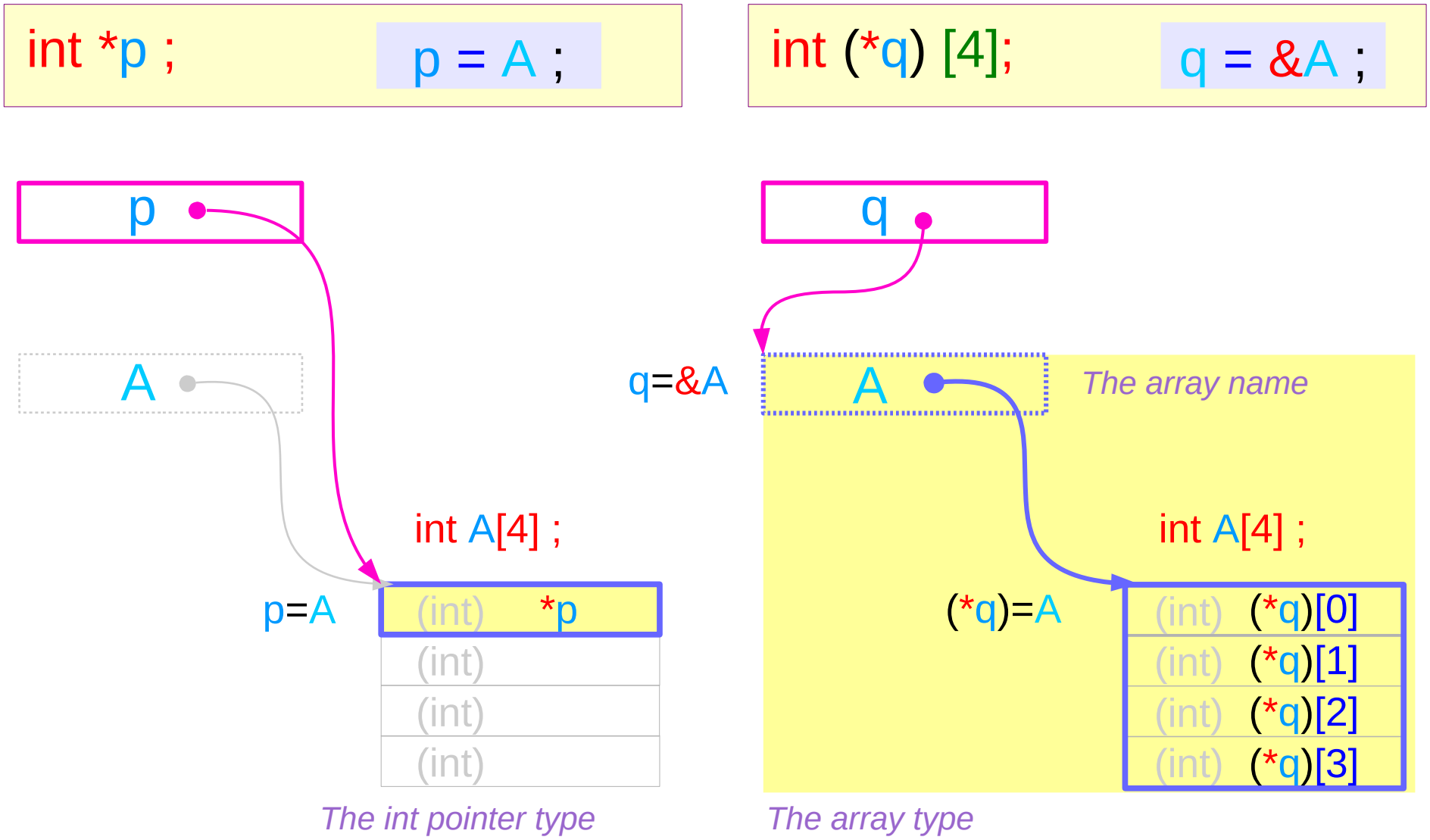
2. pointer to 2-d arrays

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

Pointer to **1-d** arrays



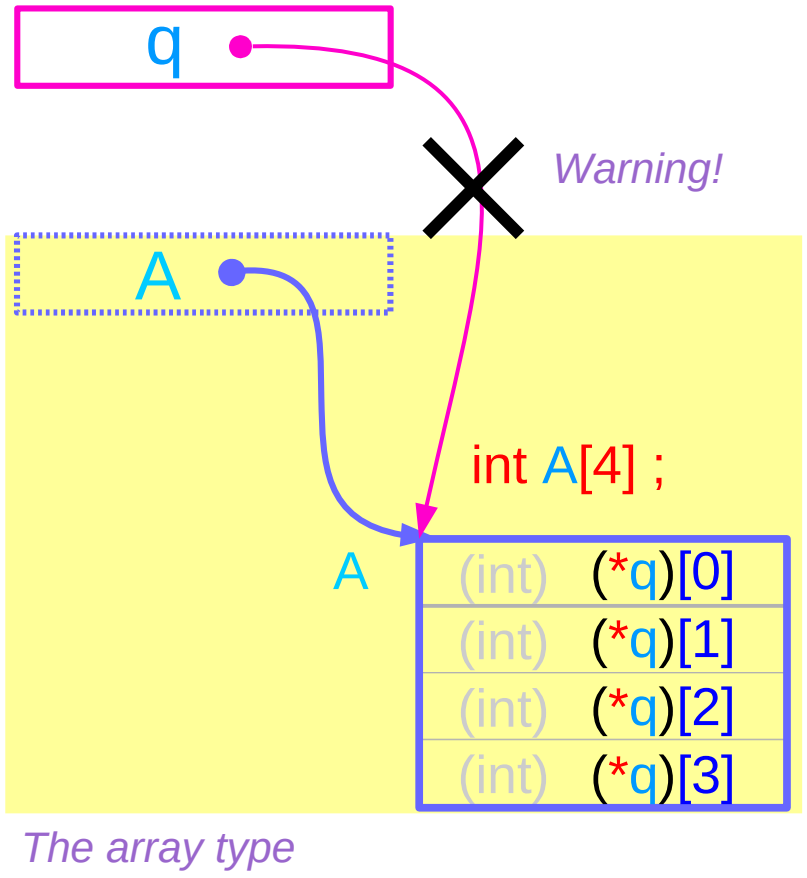
Pointer to Integer vs. Pointer to Array



Must point to an array type (array name)

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

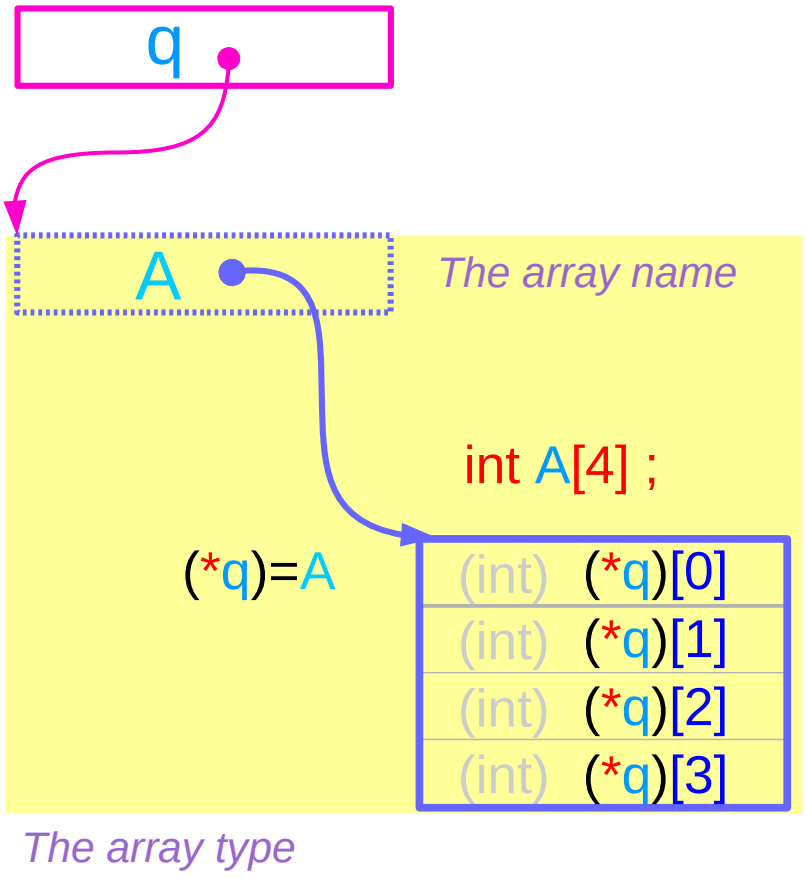
q ≠ A ;




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

q = &A ;

q=&A



Pointer to a **2-d** array

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

2-d array c

2-d array pointer p

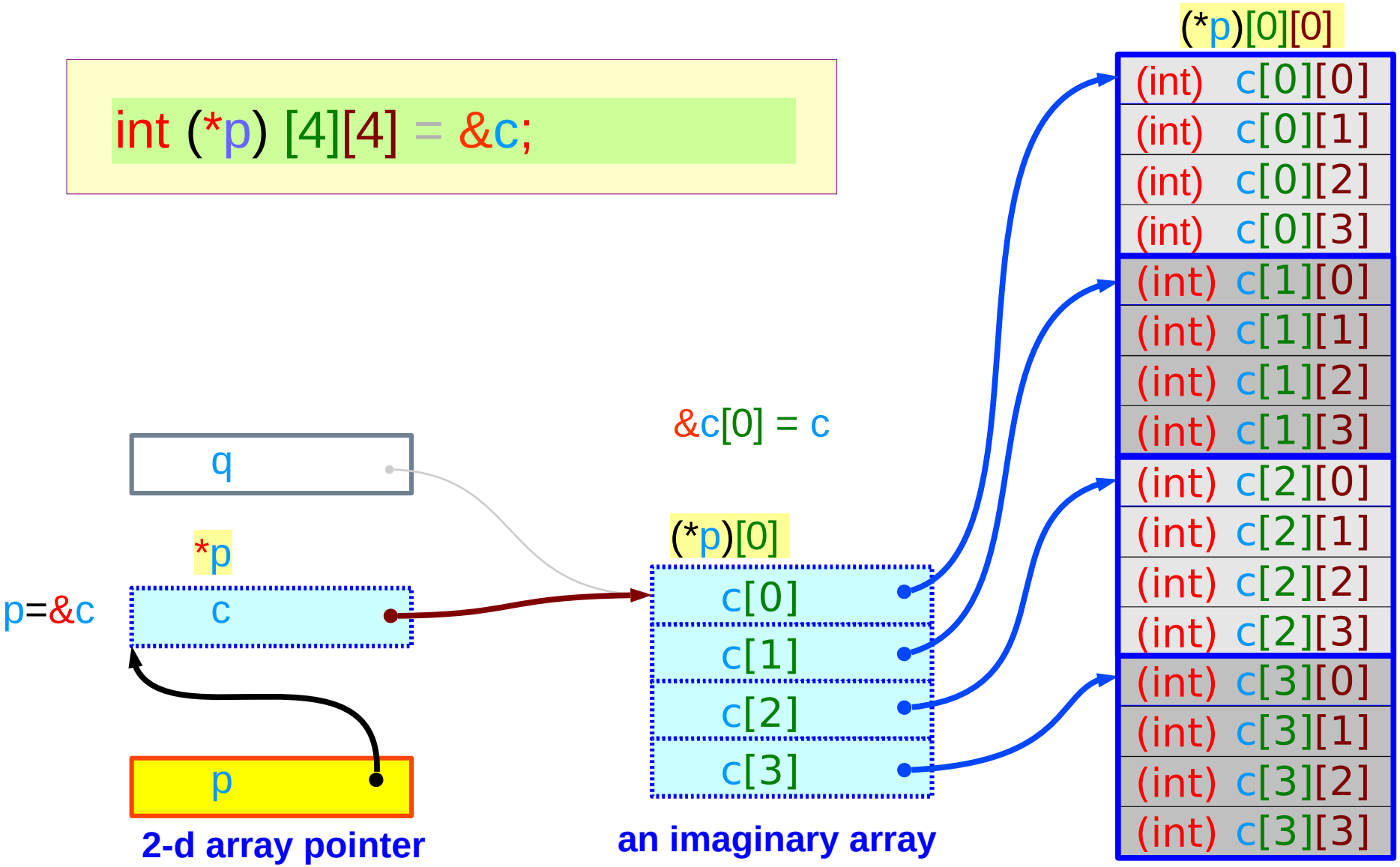
1-d array pointer q

(*p)[i][j] ➡ c[i][j]

q[i][j] ➡ c[i][j]

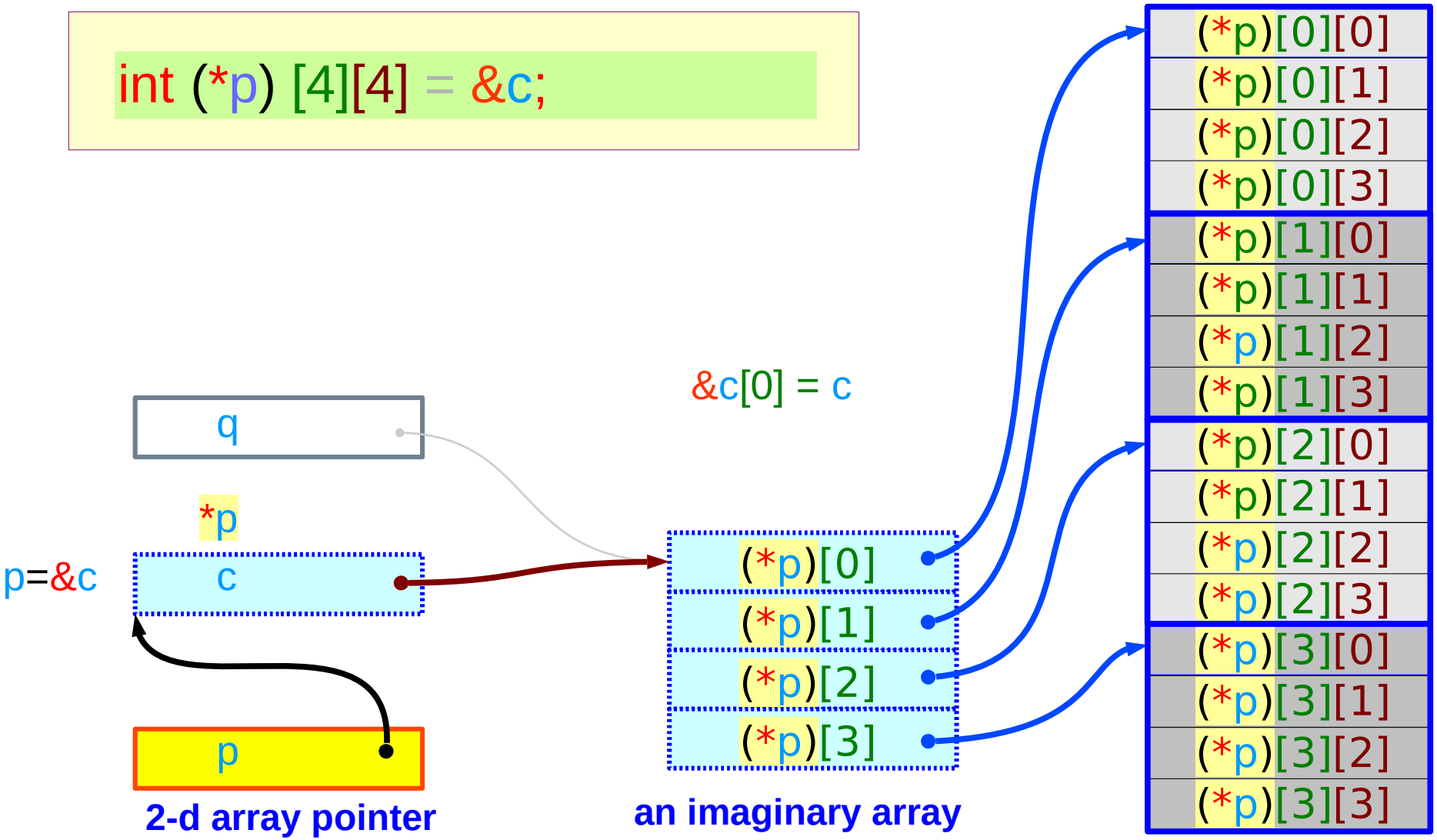
Pointer to a **2-d** array

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



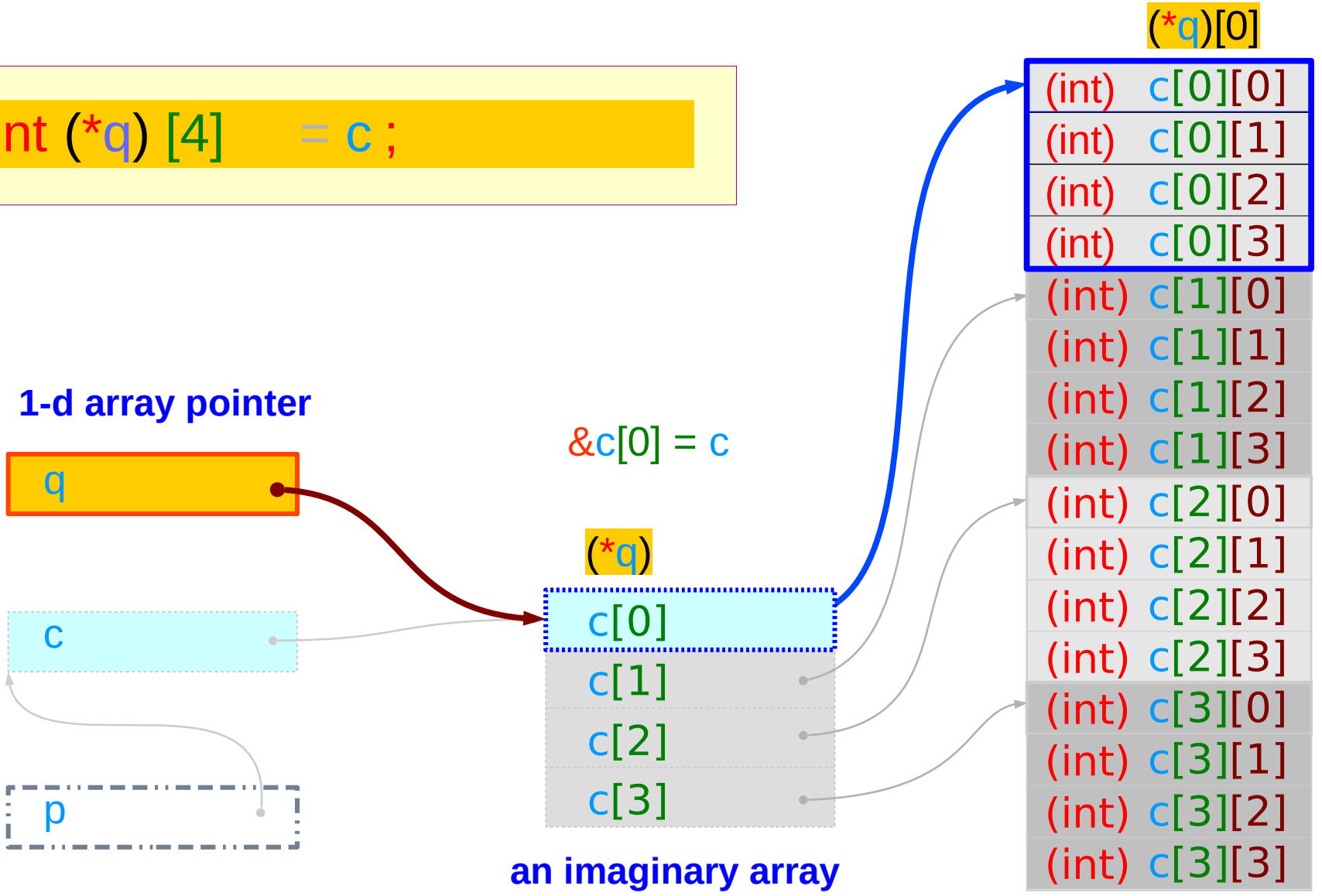
2-d array access using a pointer to a 2-d array

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



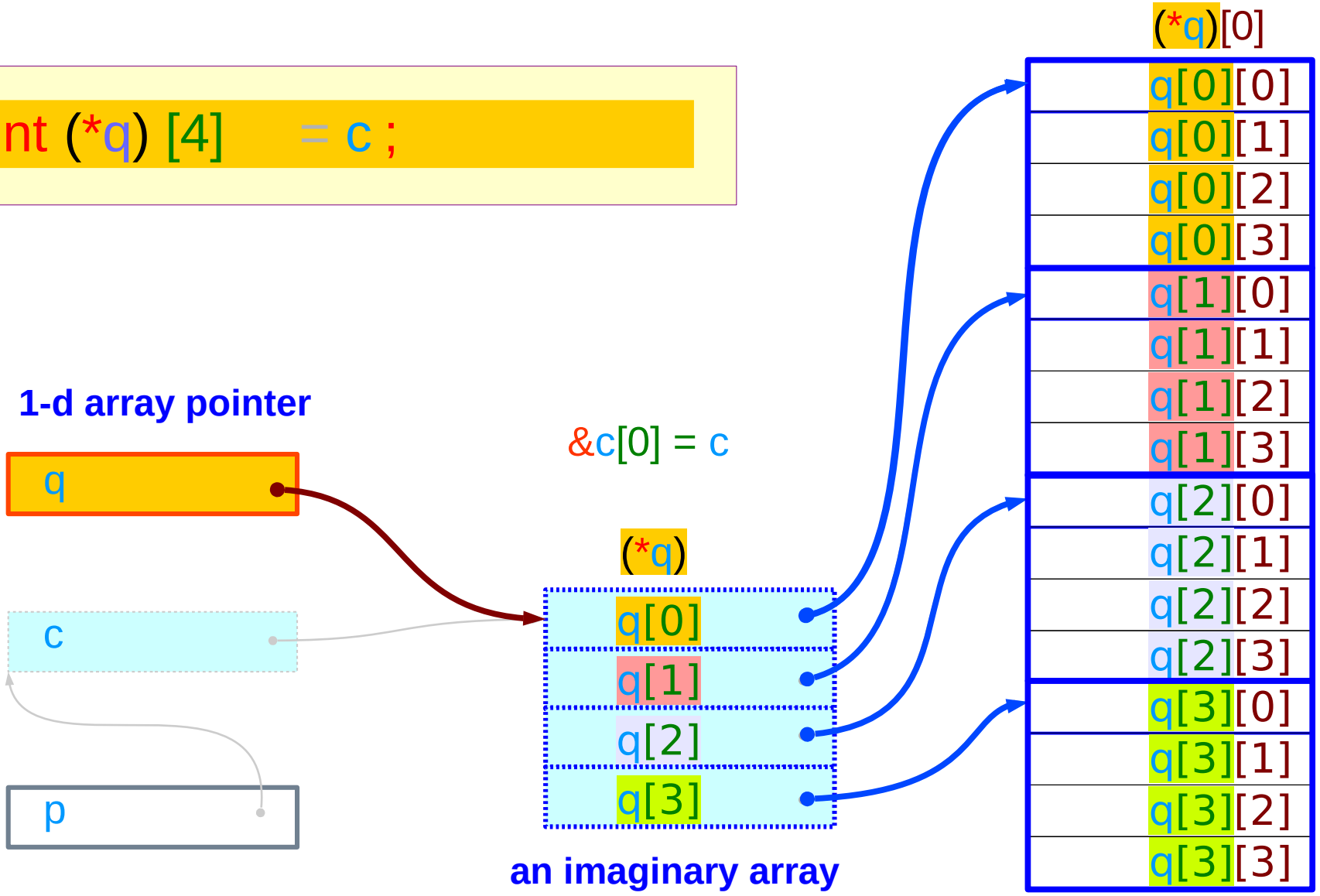
Pointer to a **1-d** array

```
int (*q) [4]     = c ;
```



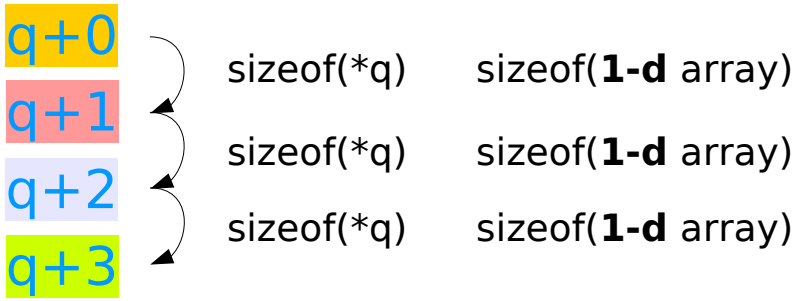
2-d array access using a pointer to a 1-d array

```
int (*q) [4]     = c ;
```



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

```
int (*q) [4]     = c ;
```



(*(q+0))	⟺	q[0]	(*(q+0))[j]	⟺	q[0][j]
(*(q+1))	⟺	q[1]	(*(q+1))[j]	⟺	q[1][j]
(*(q+2))	⟺	q[2]	(*(q+2))[j]	⟺	q[2][j]
(*(q+3))	⟺	q[3]	(*(q+3))[j]	⟺	q[3][j]

Dynamic Memory Allocation of 2-d Arrays

1. method 1

```
int   **    c ;
c = malloc(4 * sizeof (int *)) ;
c[0] = malloc(4 * sizeof (int)) ;
c[1] = malloc(4 * sizeof (int)) ;
c[2] = malloc(4 * sizeof (int)) ;
c[3] = malloc(4 * sizeof (int)) ;
```

2. method 2

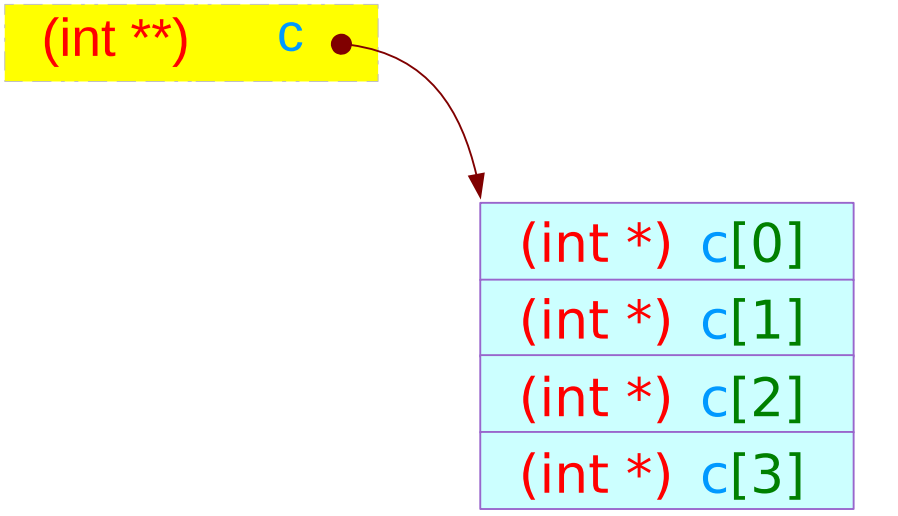
```
int   (*p) [4] ;
p = malloc(4 * 4 * sizeof (int)) ;
```

3. method 3

```
int   **    c ;
int   *     p ;
c = malloc( 4 * sizeof(int *) ) ;
p = malloc( 4 * 4 * sizeof(int) ) ;
for (i=0; i<M; i++)    c[i] = p + i*N;
```

2-d array dynamic allocation : method 1 (a)

```
int **  c ;

c = malloc(4 * sizeof (int *)) ;
```

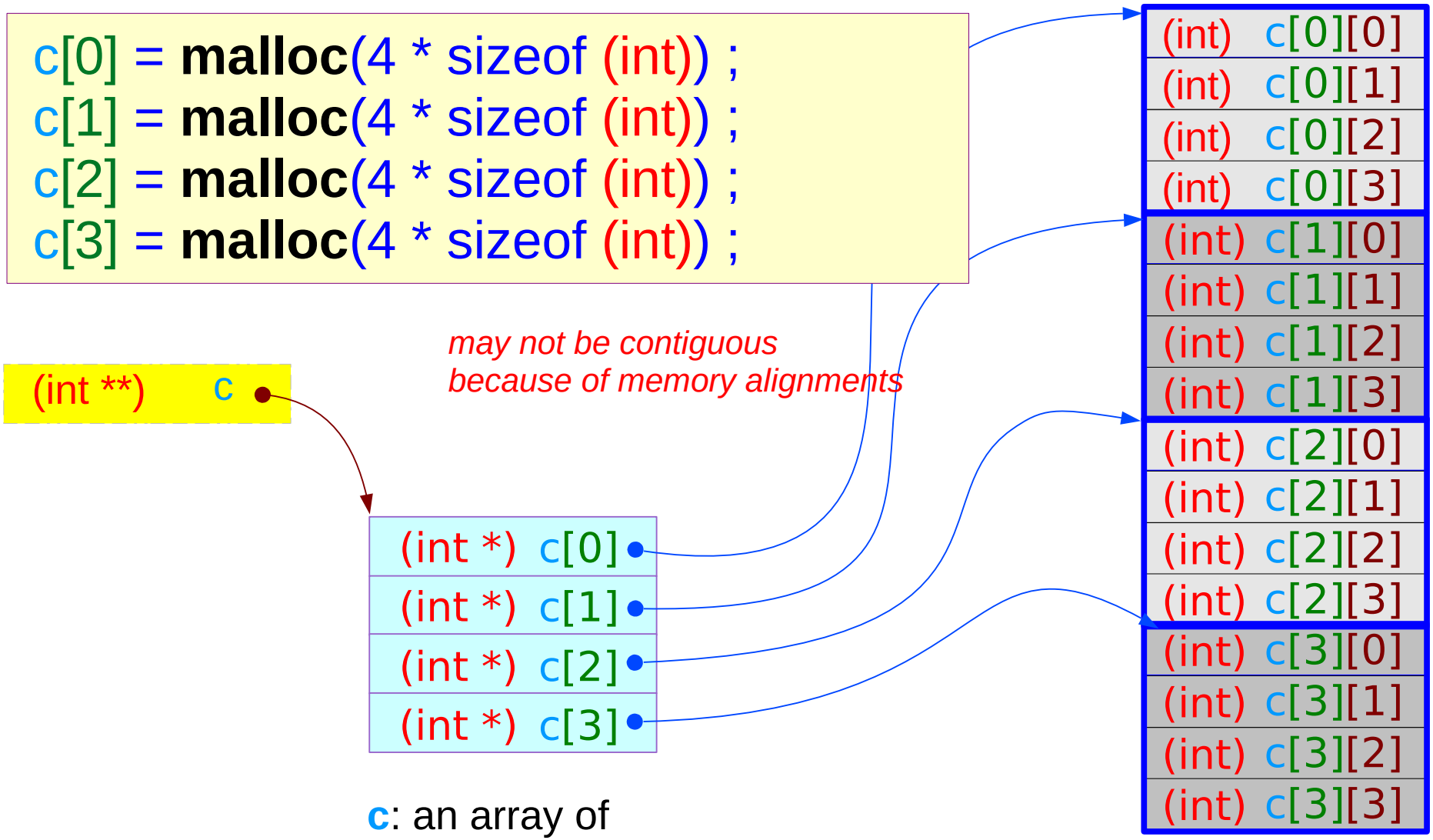


array of pointers :
allocated physically
in memory

c: an array of
integer pointers

2-d array dynamic allocation : method 1 (b)

```
c[0] = malloc(4 * sizeof (int)) ;
c[1] = malloc(4 * sizeof (int)) ;
c[2] = malloc(4 * sizeof (int)) ;
c[3] = malloc(4 * sizeof (int)) ;
```

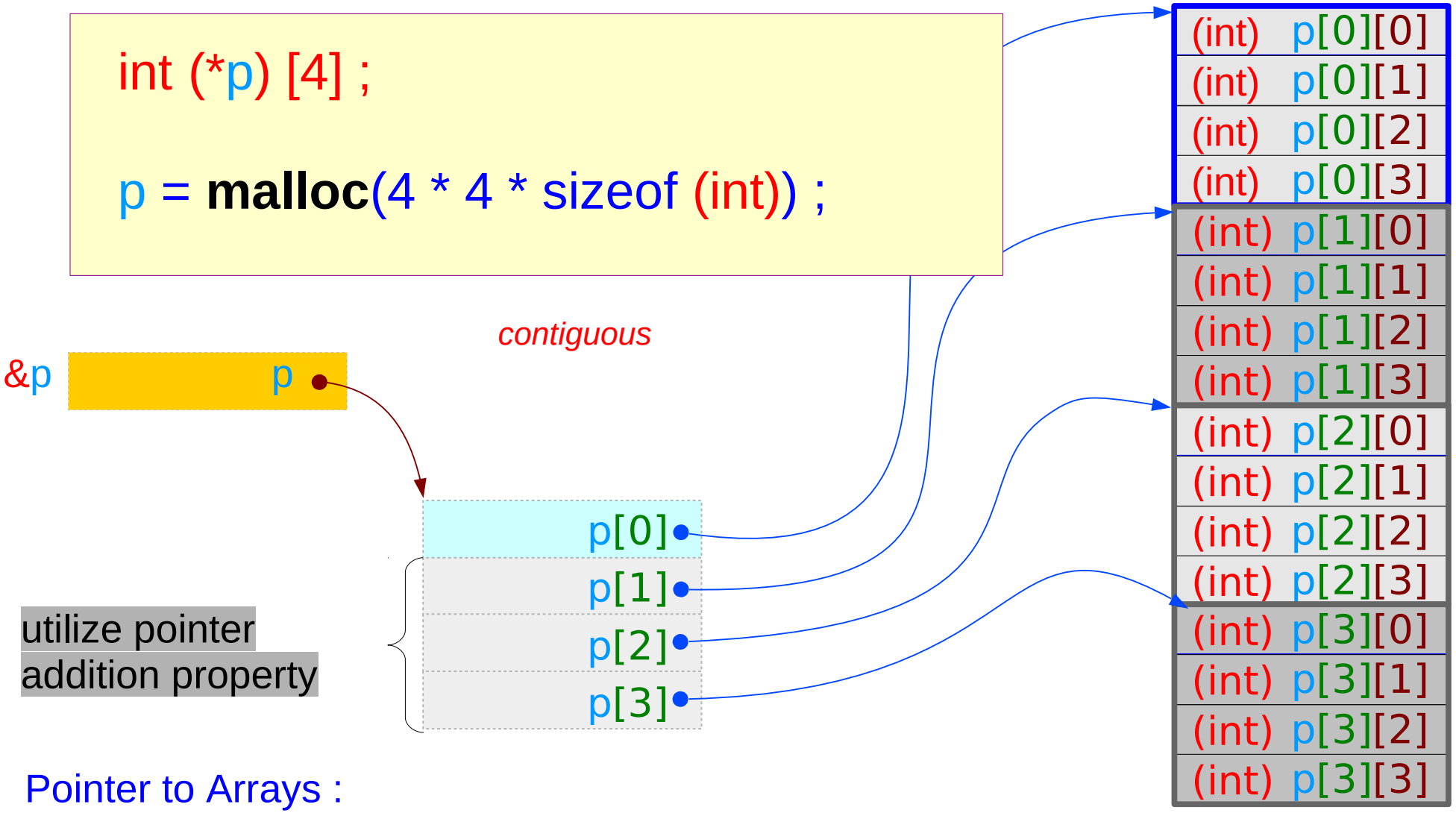


may not be contiguous
because of memory alignments

c: an array of
integer pointers

2-d array dynamic allocation : method 2

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

p = malloc(4 * 4 * sizeof (int)) ;
```

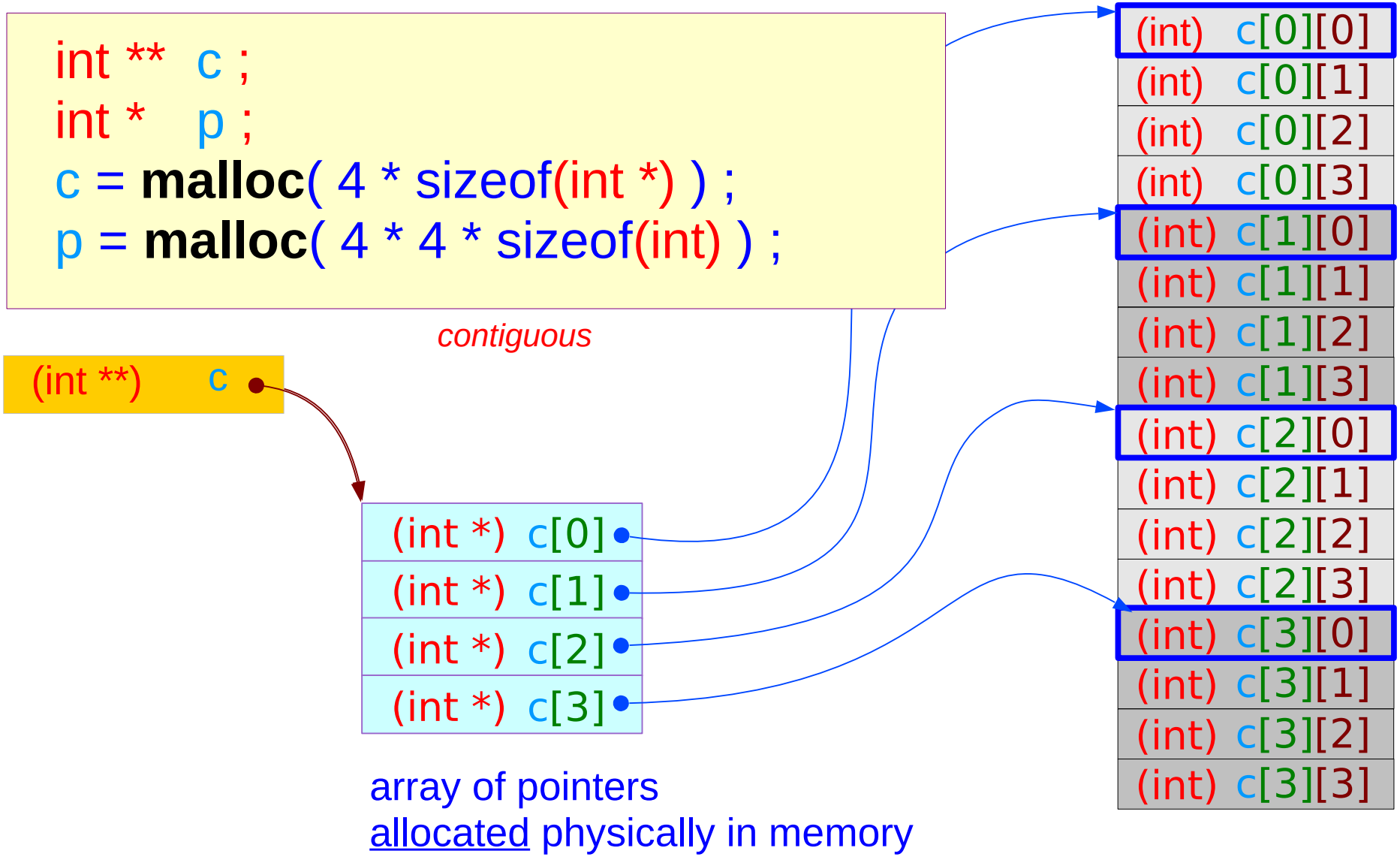


Pointer to Arrays :
No physical allocation

2-d array dynamic allocation : method 3 (a)

```
int **  c ;
int *   p ;
c = malloc( 4 * sizeof(int *) ) ;
p = malloc( 4 * 4 * sizeof(int) ) ;
```

contiguous

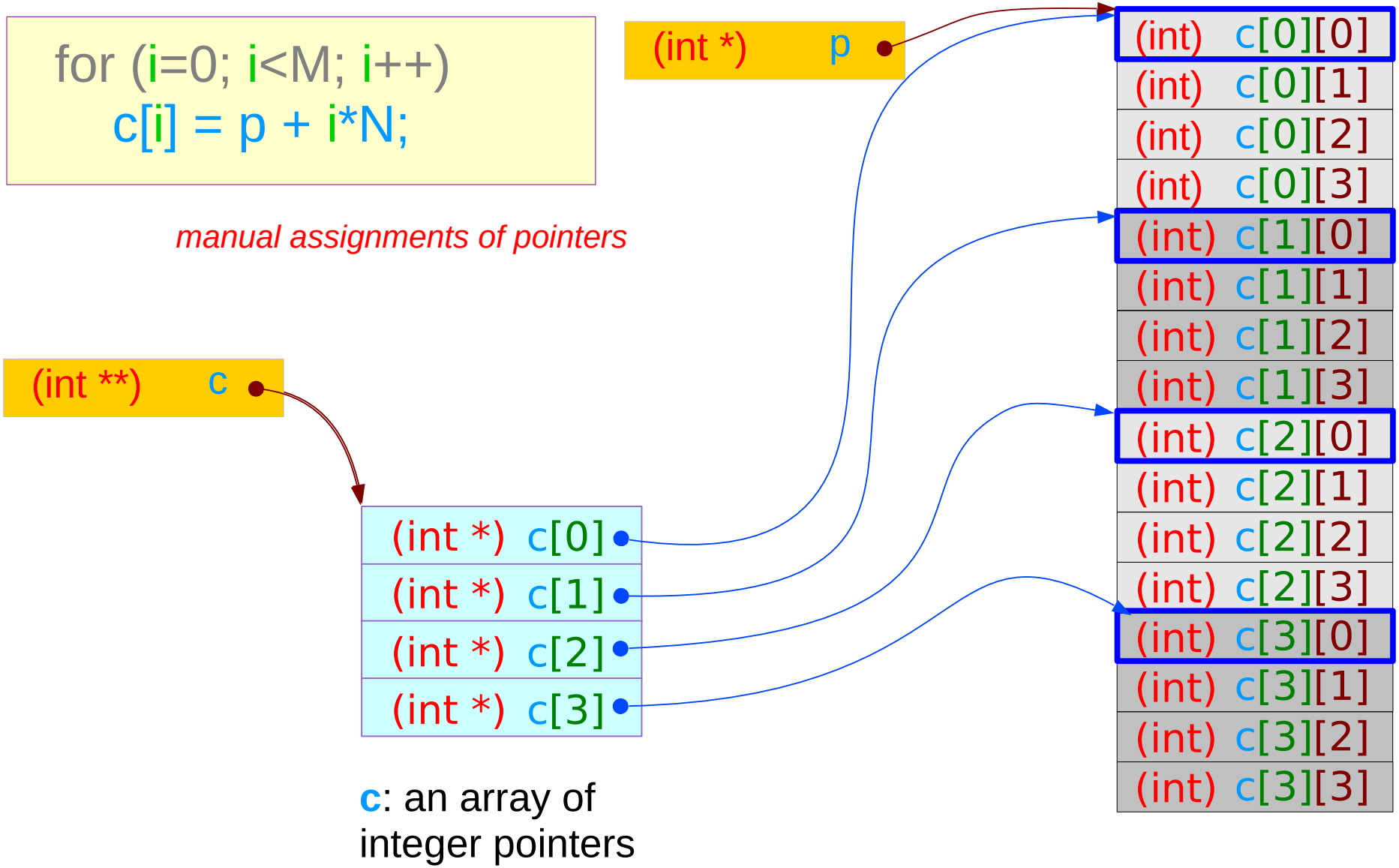


array of pointers
allocated physically in memory

2-d array dynamic allocation : method 3 (b)

```
for (i=0; i<M; i++)
    c[i] = p + i*N;
```

manual assignments of pointers



c: an array of integer pointers

Limitations

No index Range Checking

Array Size must be a constant expression

    Variable Array Size

Arrays cannot be Copied or Compared

Aggregate Initialization and Global Arrays

Precedence Rule

Index Type Must be Integral

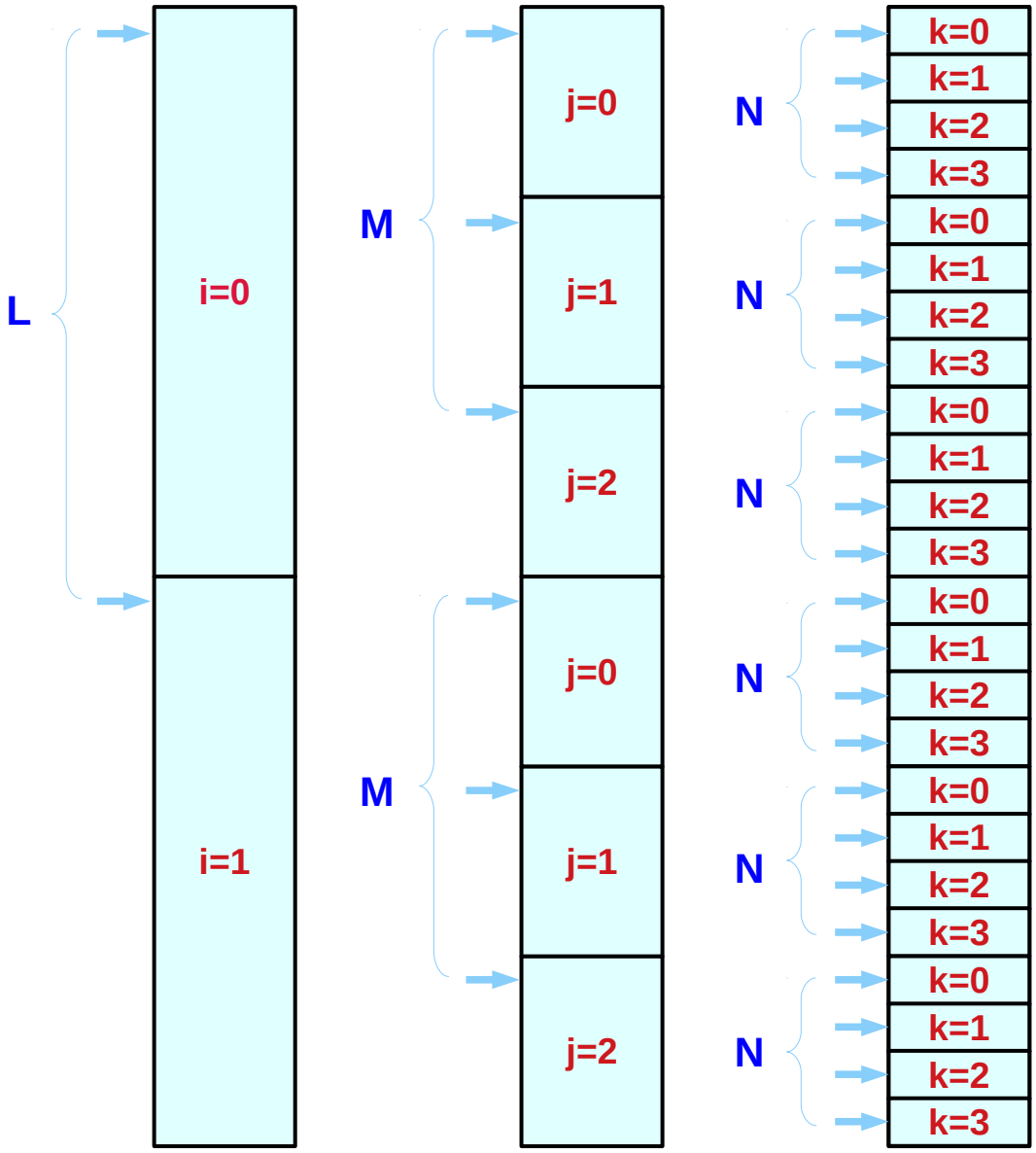
3-d Array Index

c [i][j][k]

interpret as recursive indirections
interpret as hierarchical sub-arrays

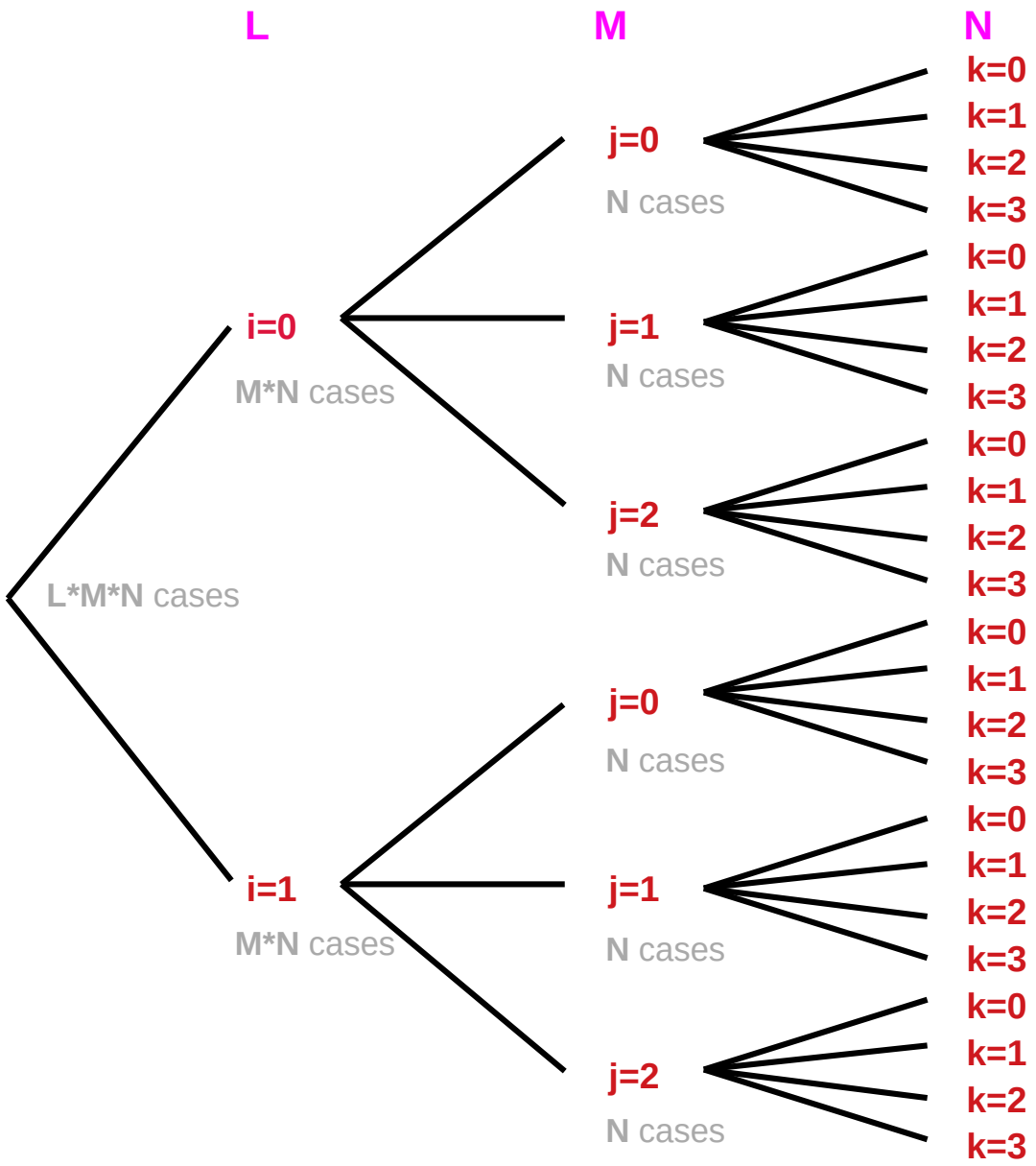
L, M, N – the number of index values

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

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



Index value tree – all possible combinations

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

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



The number of elements of subarrays

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

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

c ..... **L*M*N** elements

c[i] ..... **M*N** elements

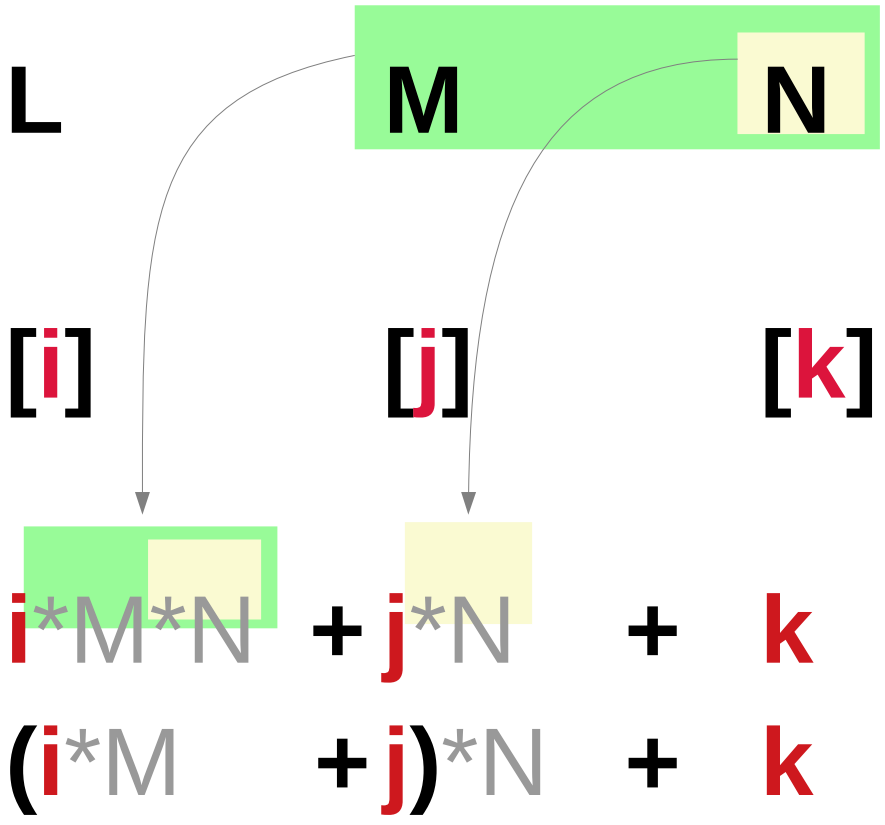
c[i][j] ..... **N** elements

array names

covering elements

From a **3-d** index to a **1-d** index

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

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



i*M*N, j*N, k – index offset values

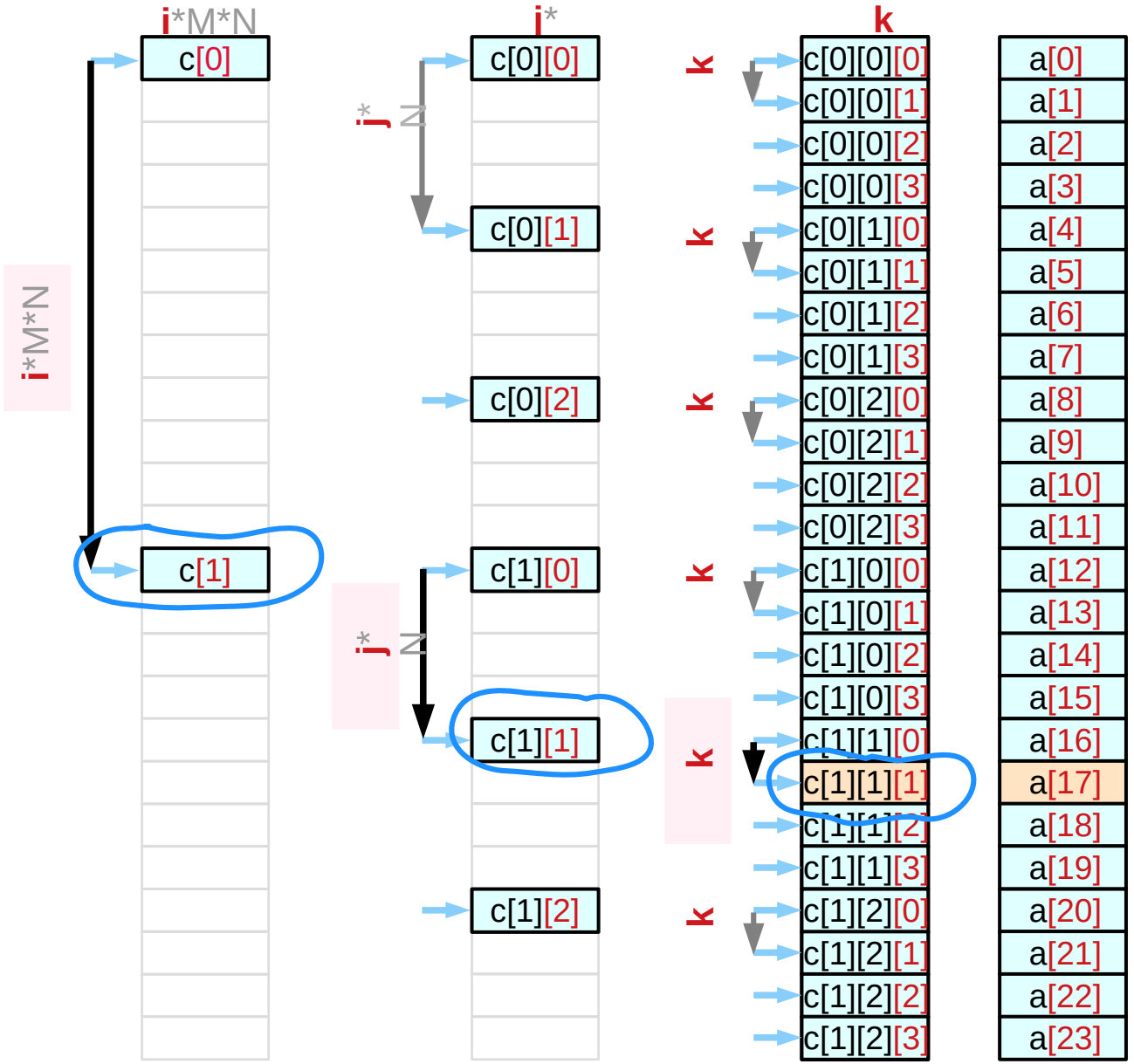
```
int     c [L][M][N];
```

c [i][j][k]

c [1][1][1]

i=1	j=1	k=1

a [(1*3 + 1)*4 + 1]



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
[5] https://pdos.csail.mit.edu/6.828/2008/readings/pointers.pdf