Applications of Arrays (1A)

Young Won Lim
2/11/17

Copyright (c) 2009 - 2017 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.

This document was produced by using LibreOffice.

2-D Array Definition

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

	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]

row major ordering

Accessing 2-D Arrays

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

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

1. recursive pointers

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

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

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

2. linear array pointers

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

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

Array name : a pointer to the 1-d arrays

```
int        a [4];

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

int | a | [4] ;

a points to *a 4 integer element array*

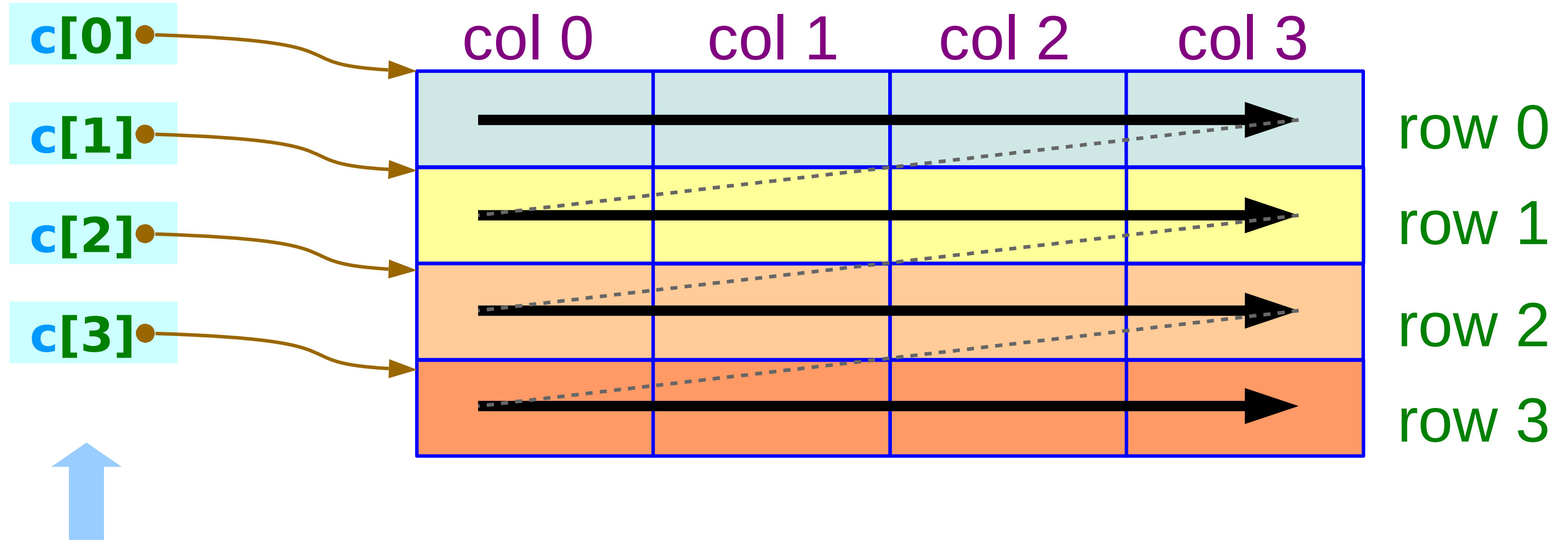
int | c[4] | [4] ;

each of **c[0]**, **c[1]**, **c[2]**, **c[3]**
points to *a 4 integer element array*

Row Major Ordering

int c[4] [4];

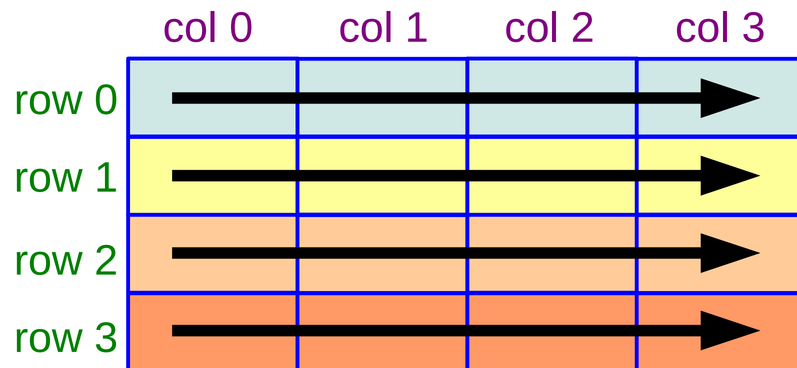
row major ordering

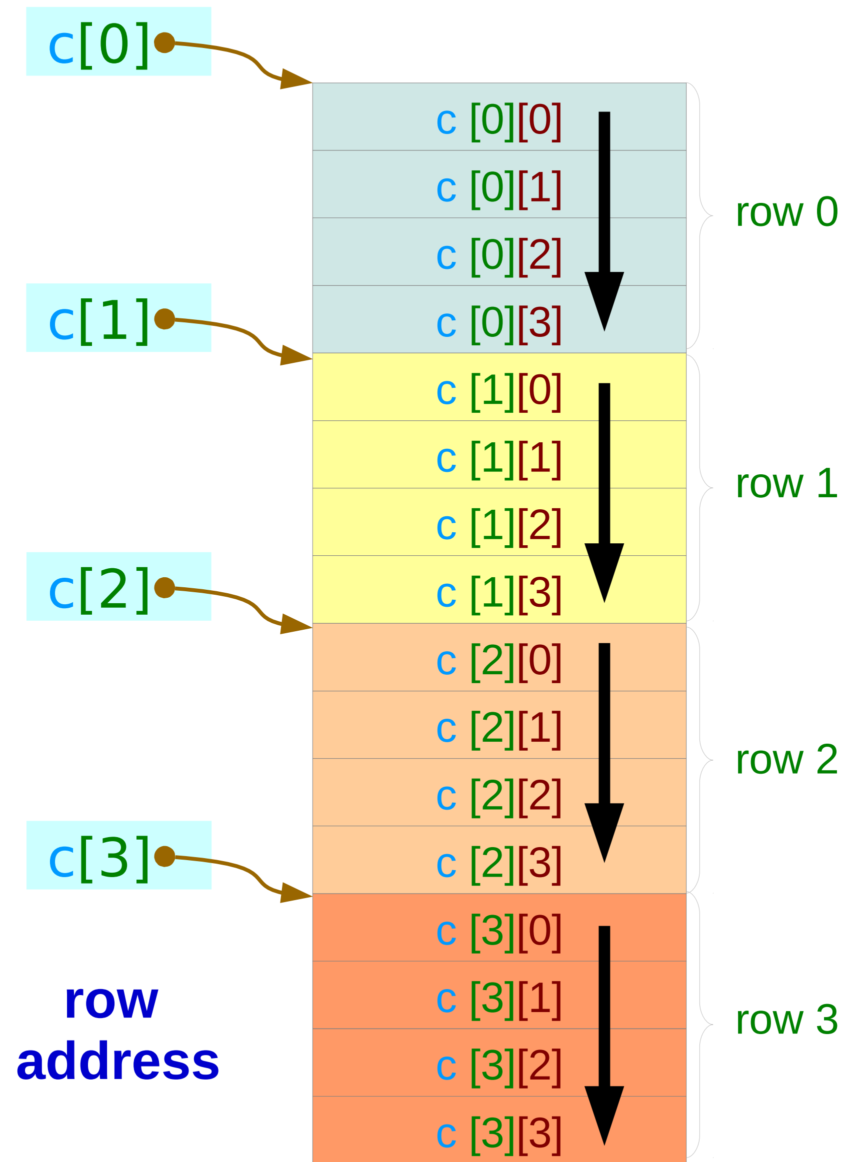


consider each c[i] as
the name of a 4 element array

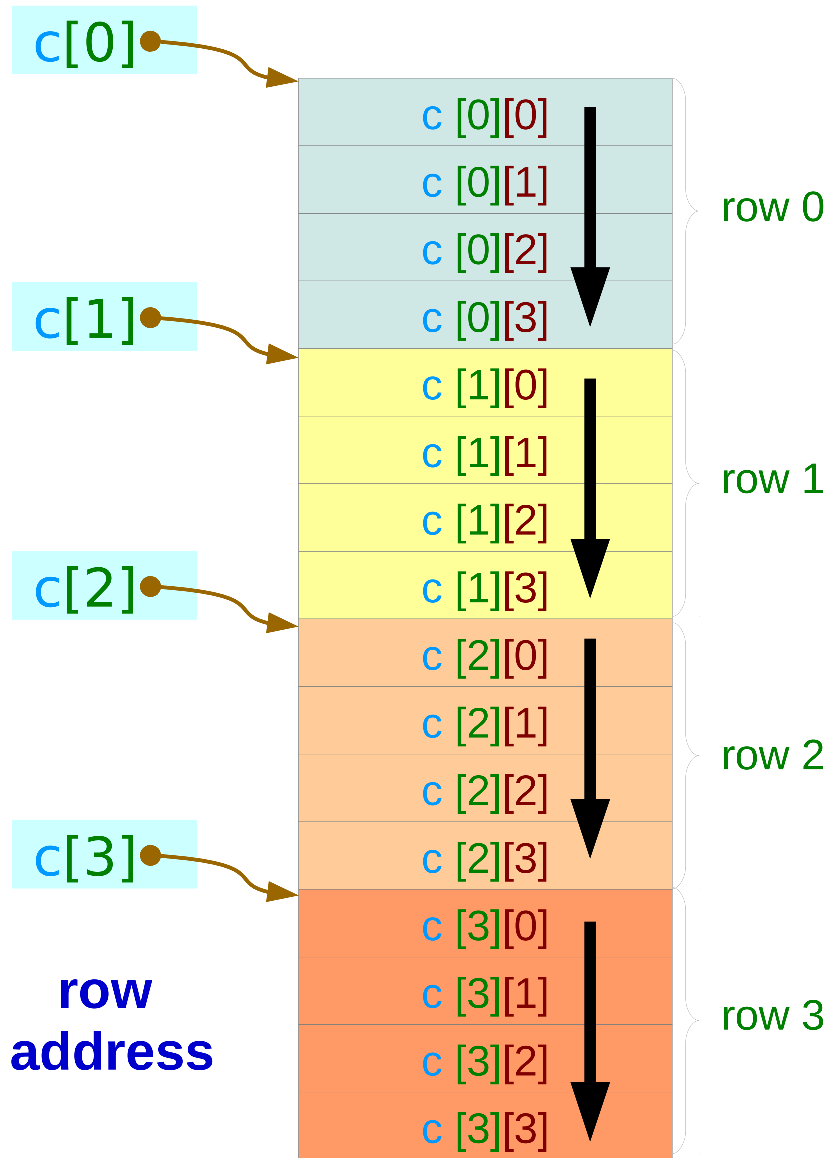
Linear Array Memory Layout

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

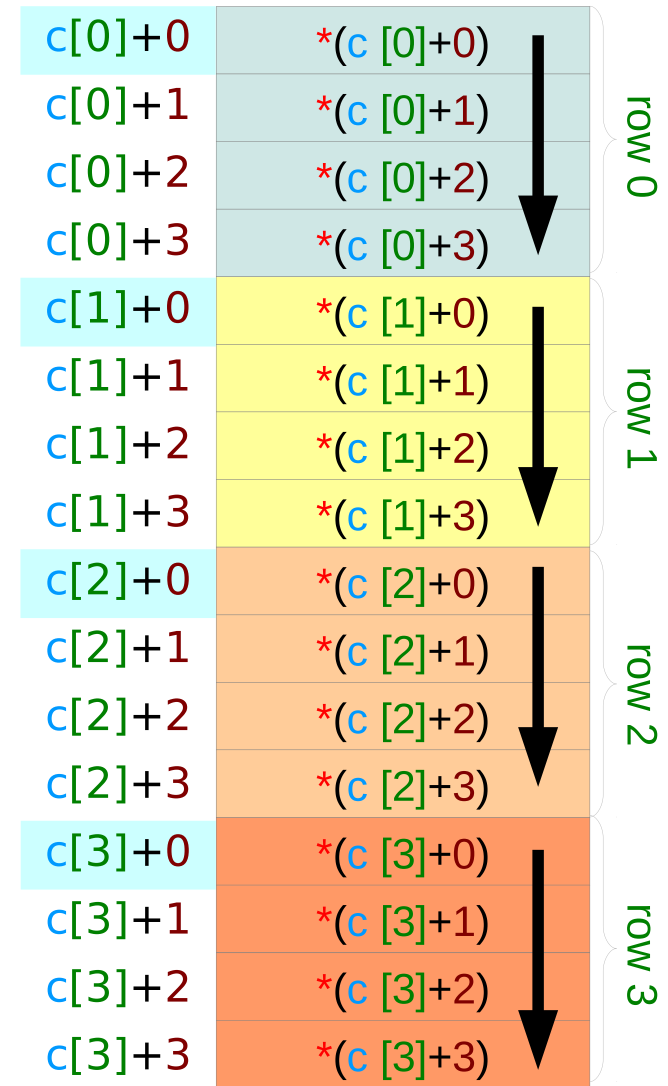




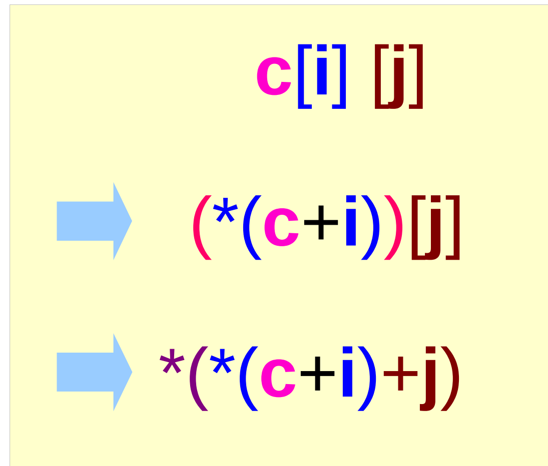
Row Address and Element Address



element address



Accessing 2-d arrays via recursive pointers



*(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]

Recursive Conversions to Pointers

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

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

assumption

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

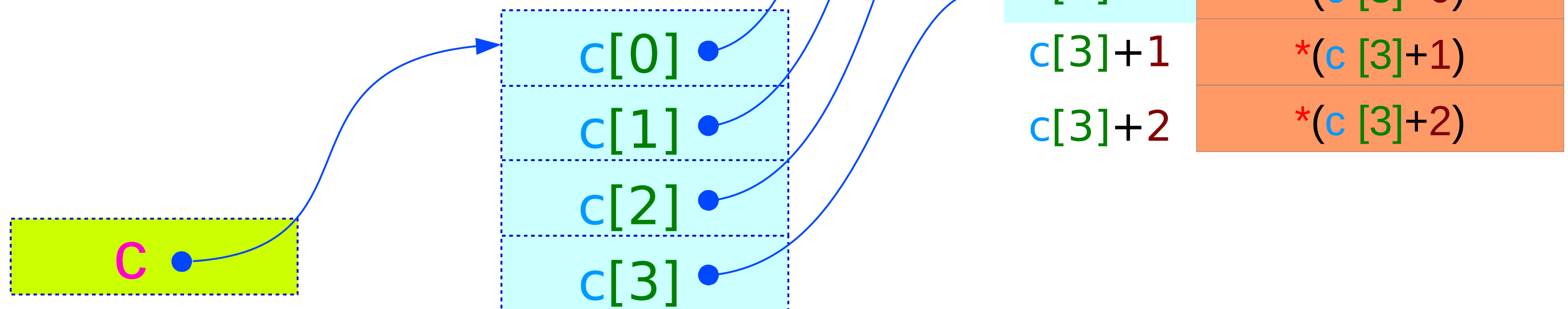
Nested Arrays

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

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

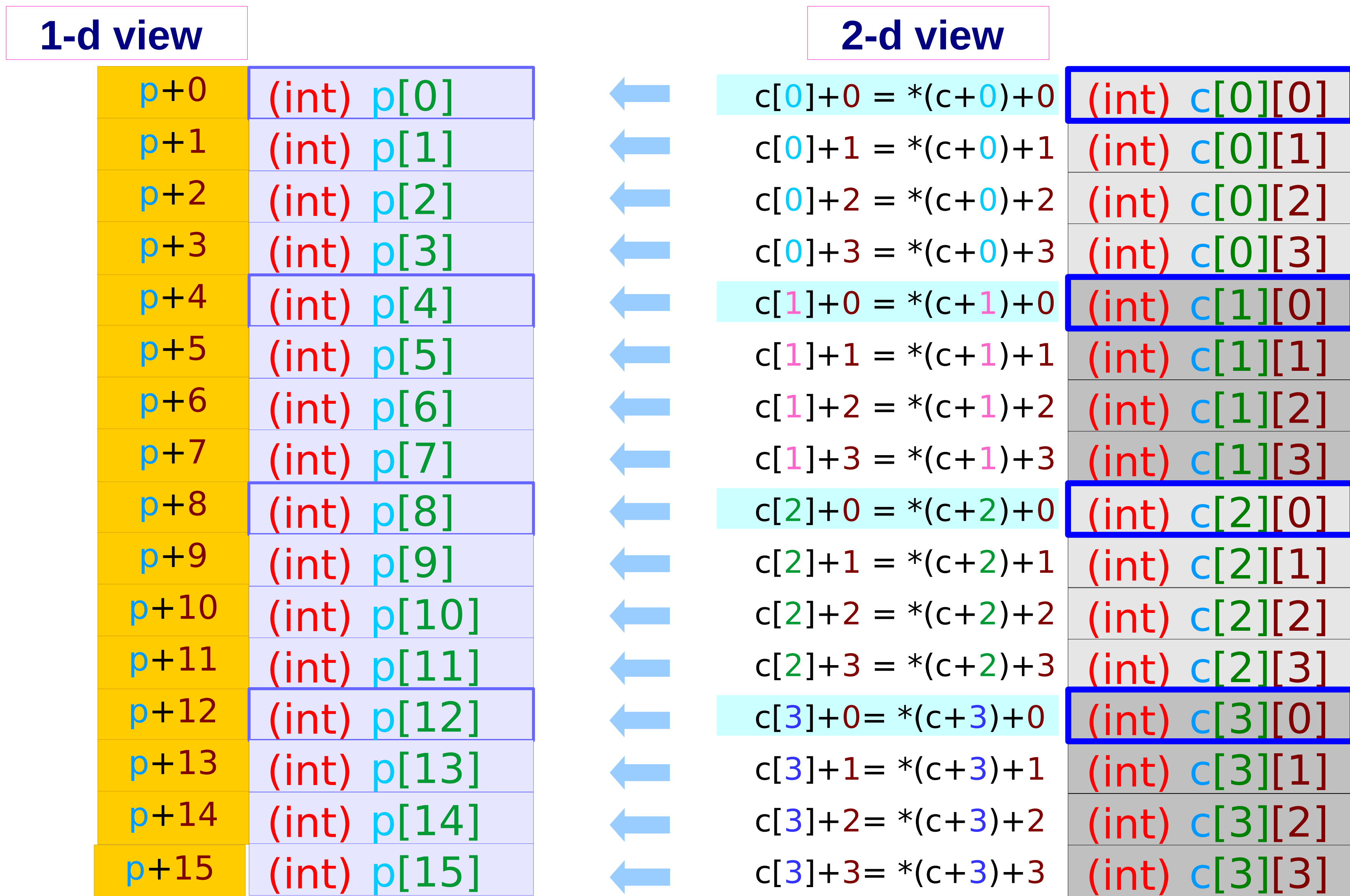
int c[4] [3] ;

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)



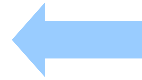
The intermediate array : not necessarily allocated in the memory

A linearization of a 2-D array



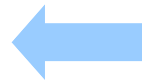
2-d array access via a single pointer

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



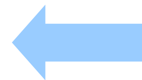
```
int c [4][4];
```

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



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

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



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

```
*(p +k)
```

```
i = k / 4;
j = k % 4;
```

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

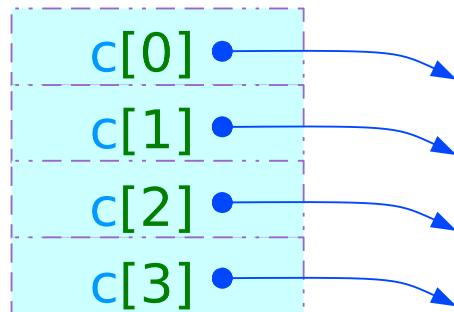
Static Allocation of a 2-d Array

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 intermediate array :
not necessarily allocated
in the memory

The name of a 2-d array

```
int        a [4];

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

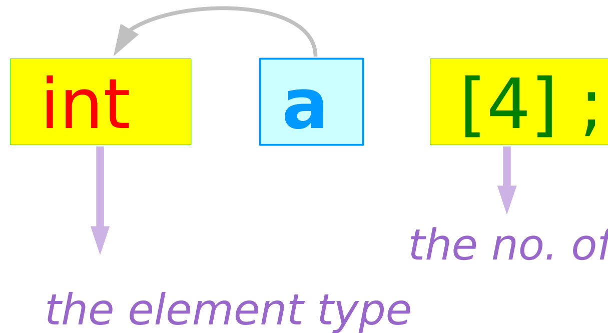
1. the name of the nested array (recursive definition)

2. a double pointer

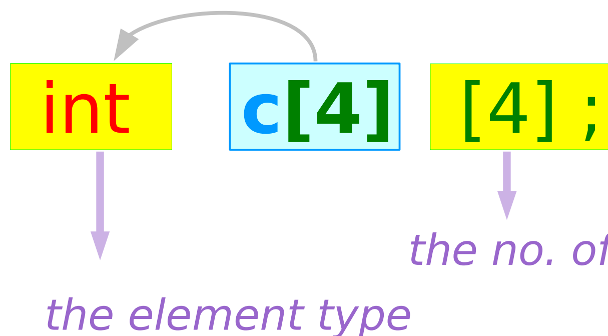
3. a pointer to an array

Nested array name

```
int        a [4];

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



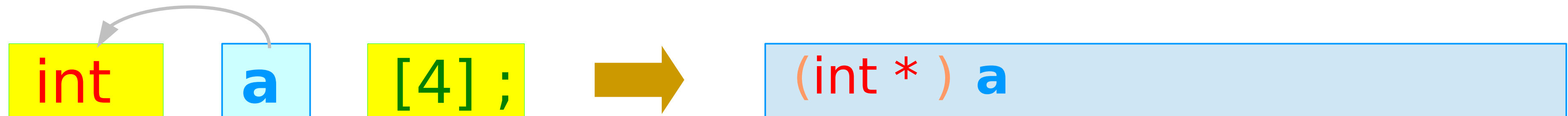
The array name **a** holds *the starting address of the 4 integer element array*



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

A 2-d array name as a double pointer

```
int        a [4];

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



a points to an integer data

c[i] points to an integer data
c points to an integer pointer

A 2-d array name as a pointer to an array

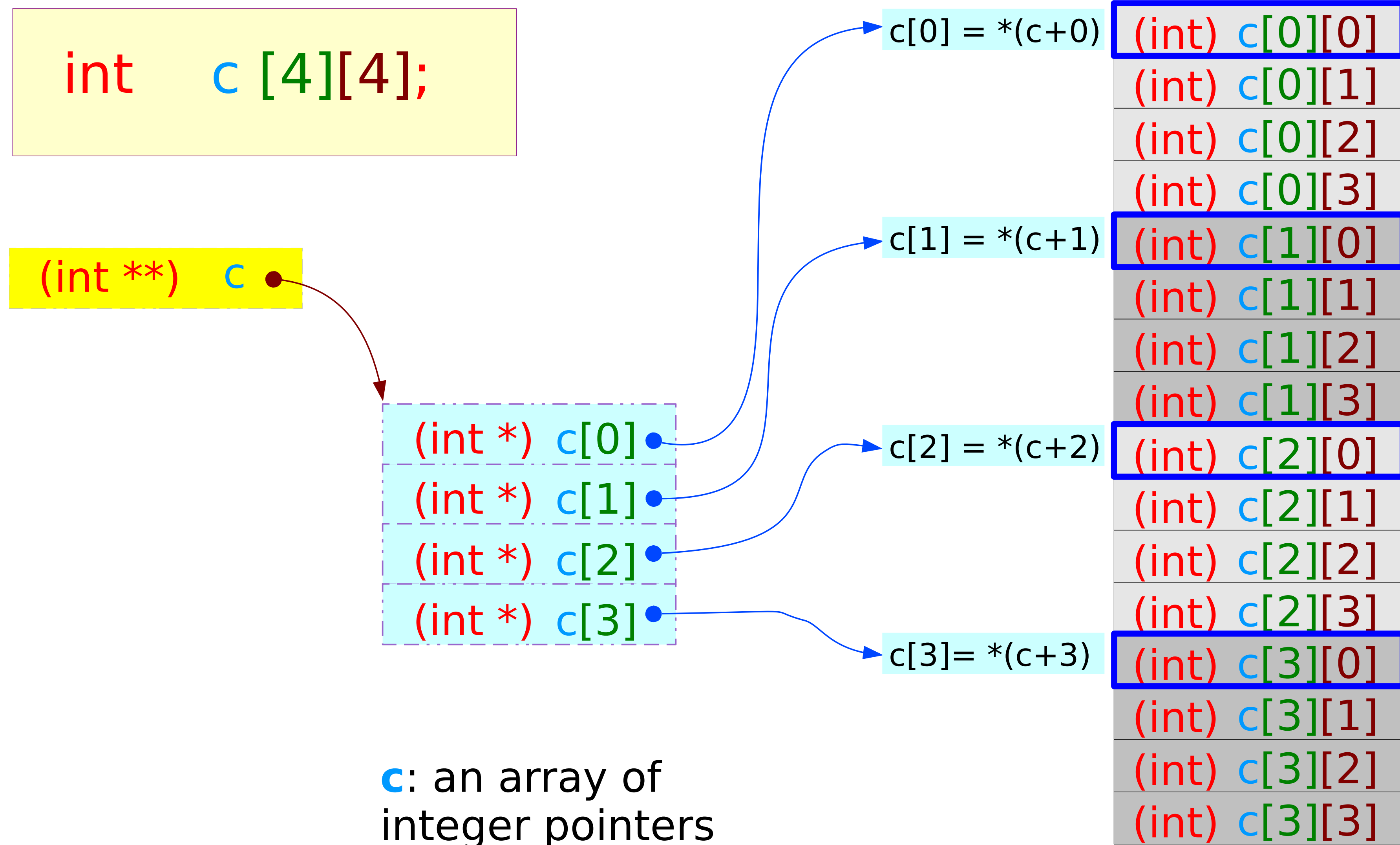
```
int        a [4];

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

a : an array name

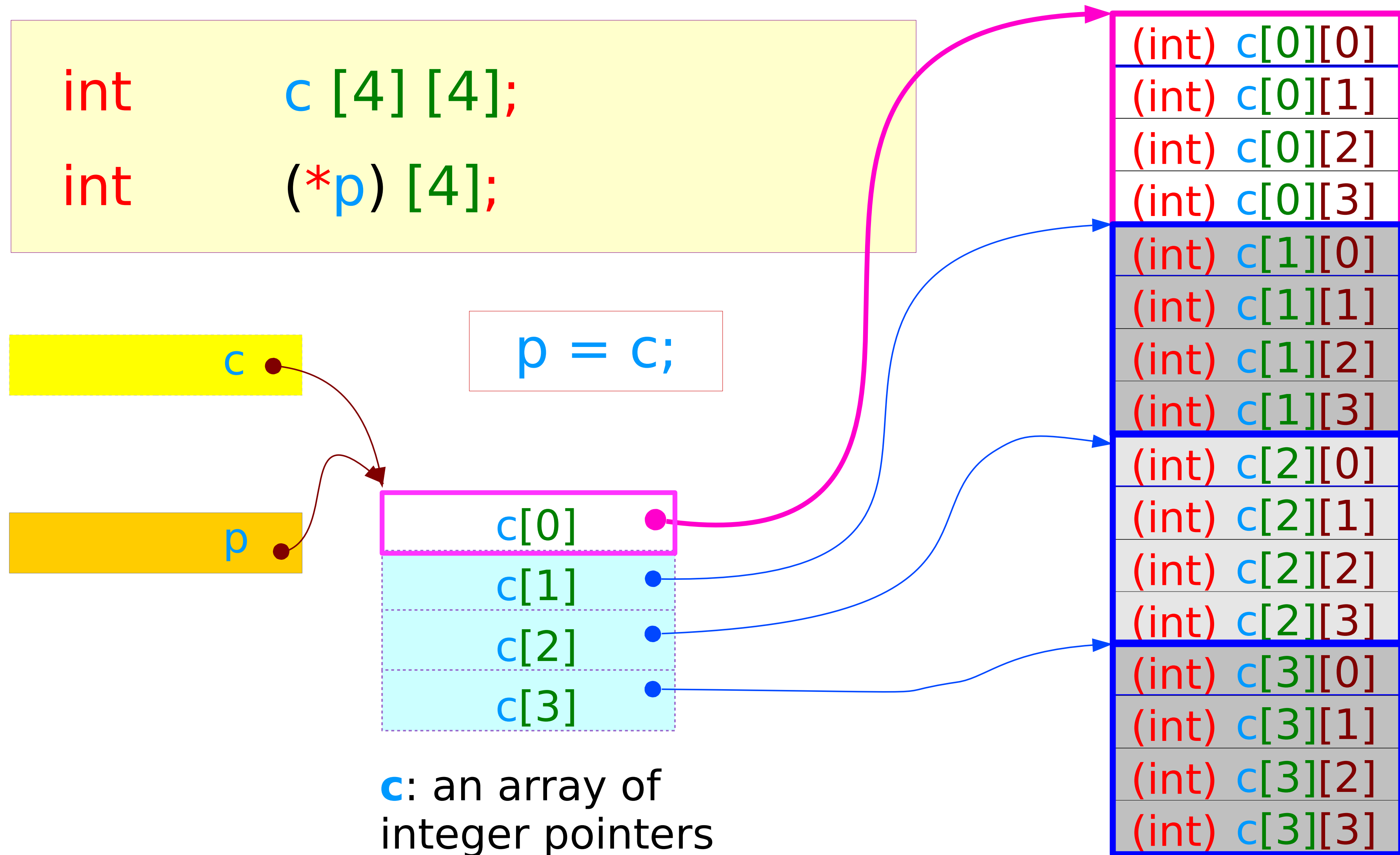
c[i] : array names

c : the intermediate array name

An intermediate array in a 2-d array



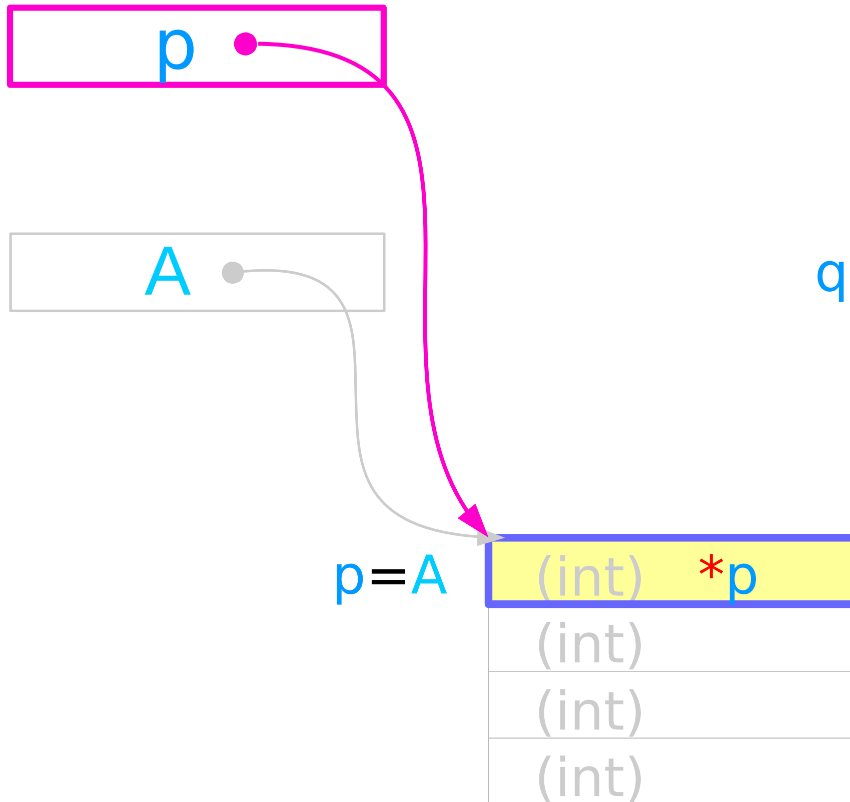
A 2-d array and a pointer to a 1-d array

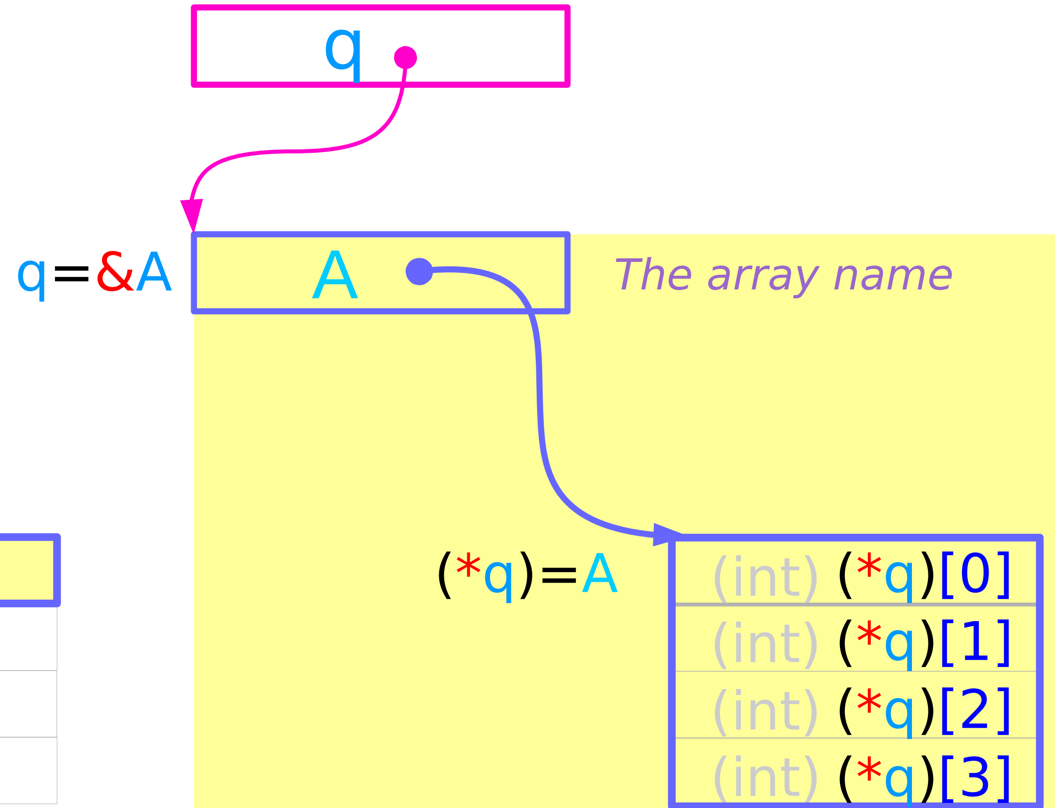


Pointer to Integer v.s. Pointer to Array

```
int    *p ;        p = A ;
```

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

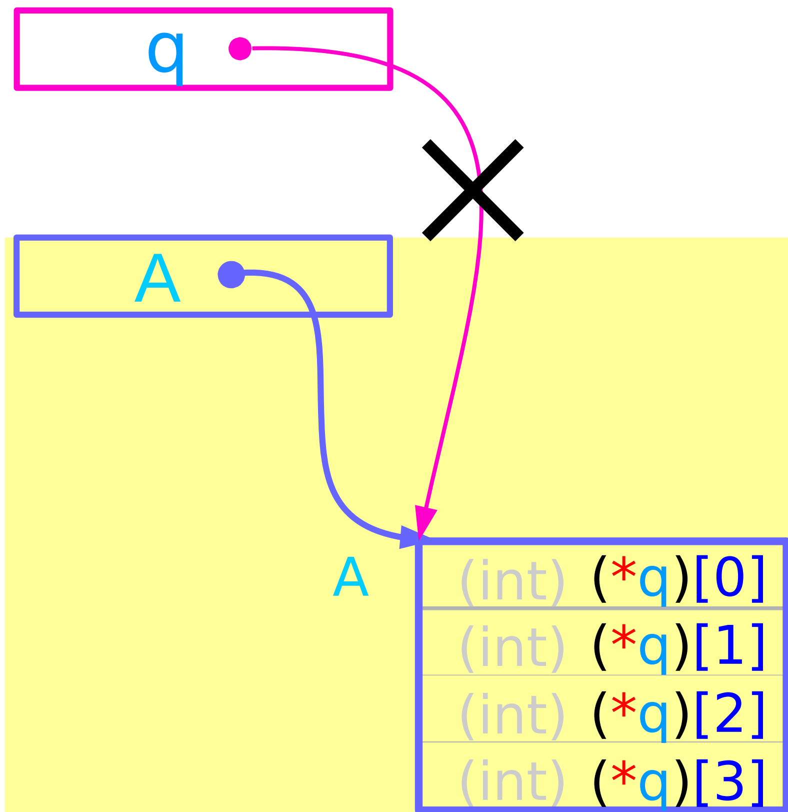




Must point to an array type (array name)

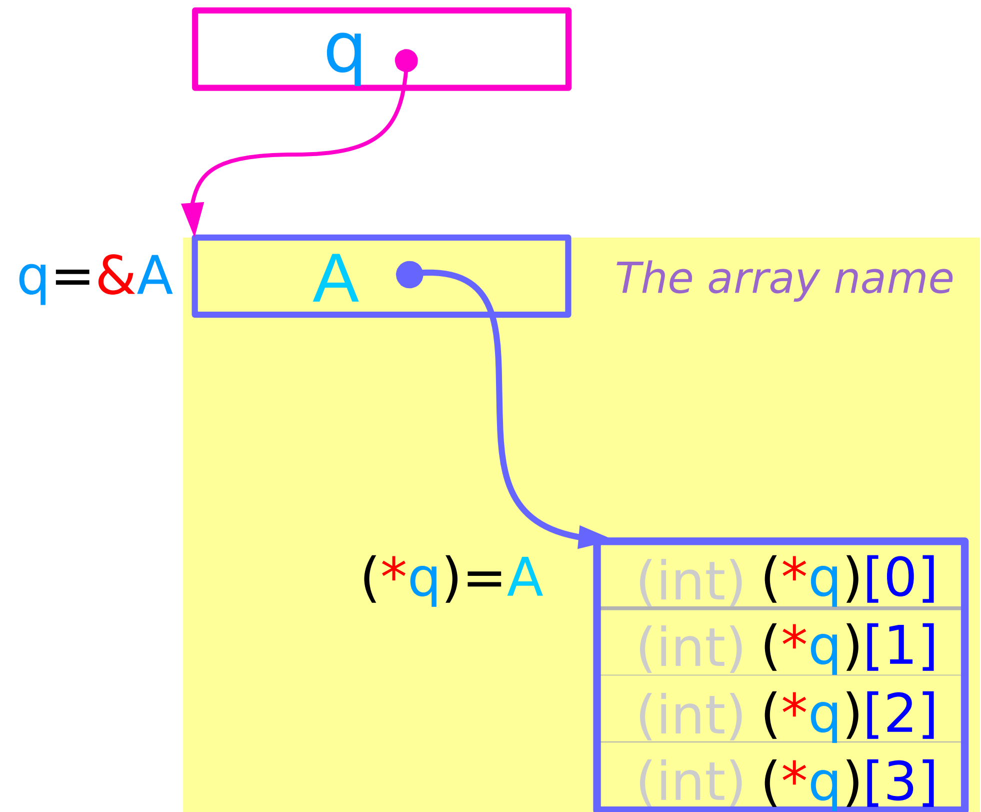
```
int    (*q) [4];    q ≠ A ;
```

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



The array type

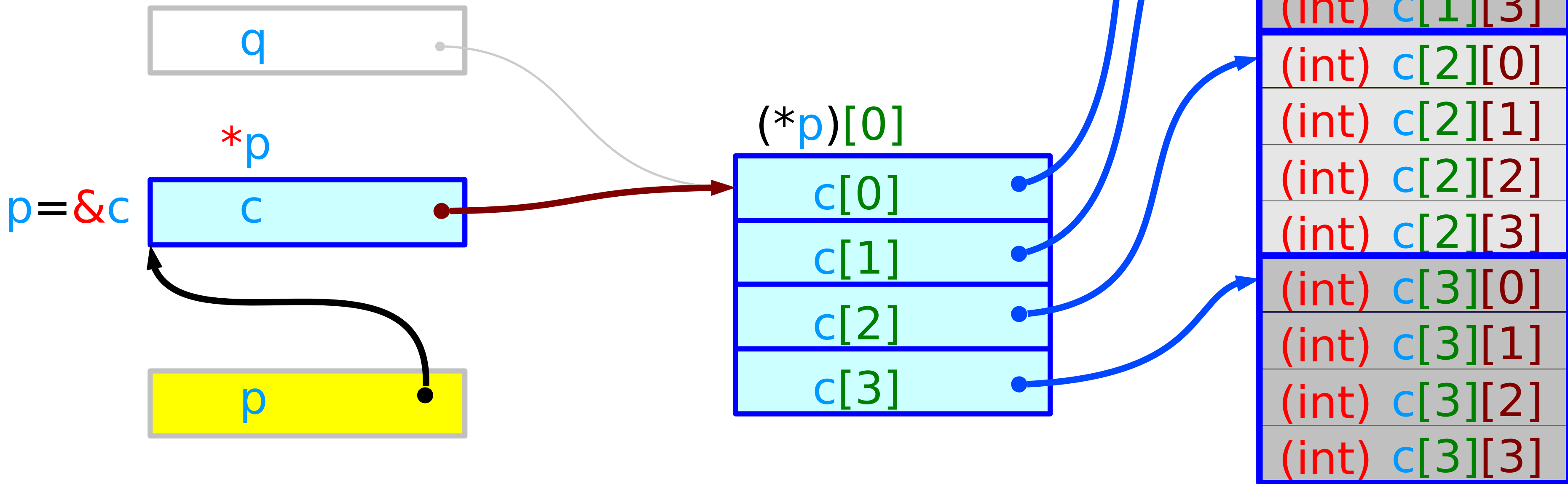
q=&A



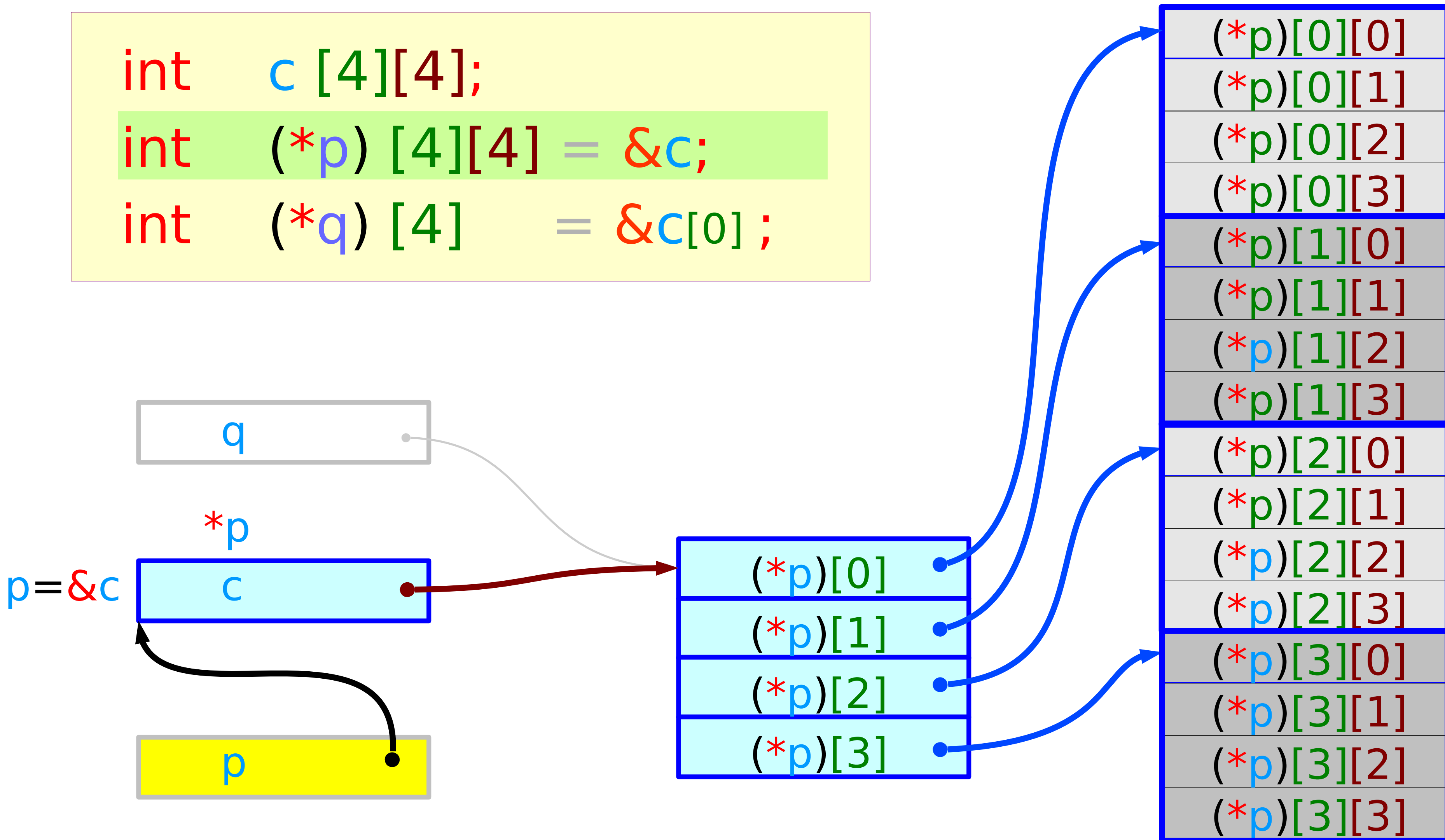
The array type

Pointer to 2-d Arrays

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



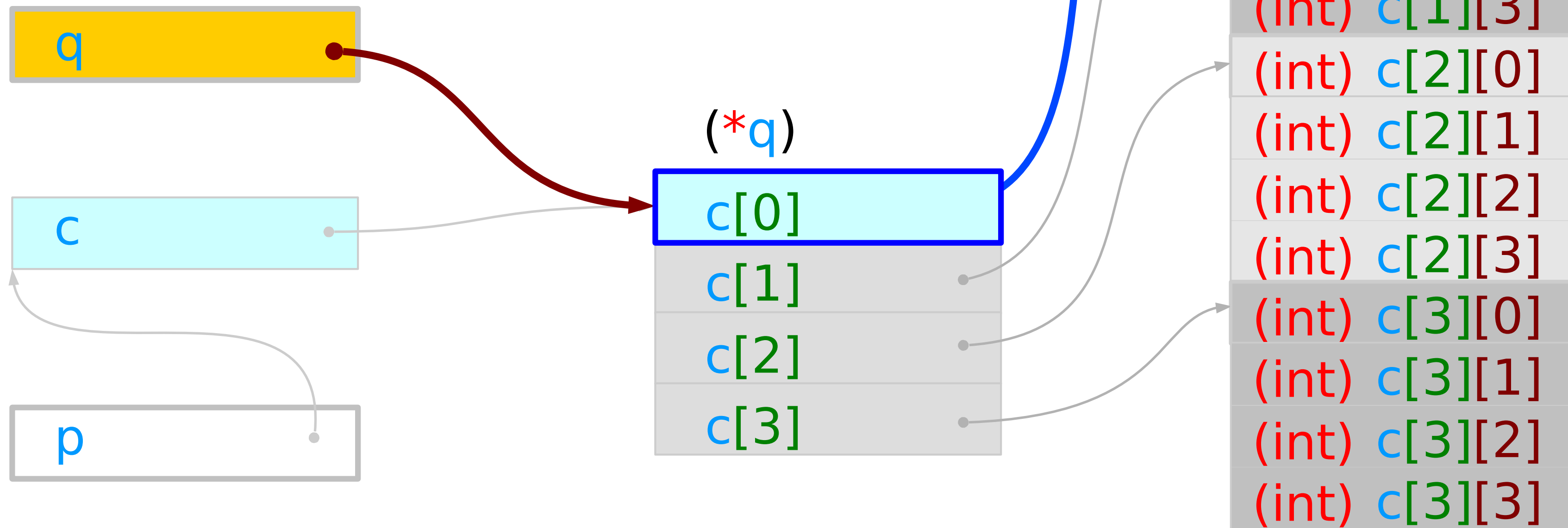
Pointer to 2-d Arrays – accessing elements



Pointer to 1-d Arrays

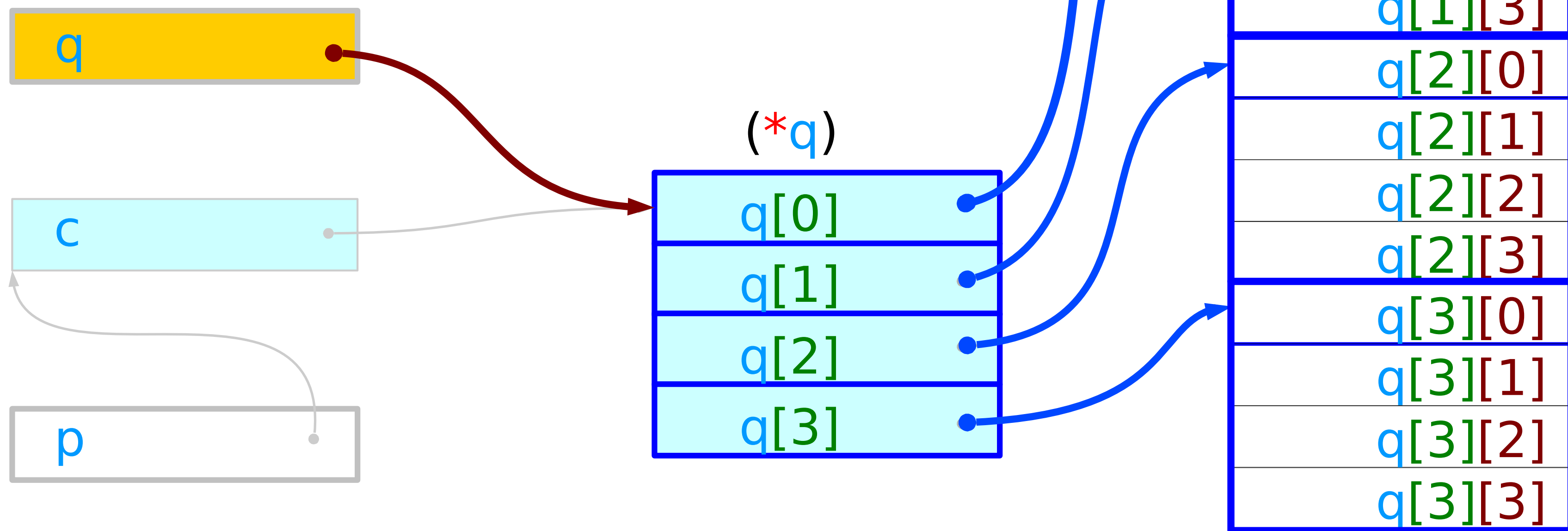
```
int    c [4][4];
int    (*p) [4][4] = &c;
int    (*q) [4]    = &c[0] ;
```

(*q)[0]



Pointer to 1-d Arrays – accessing 2-d elements

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



Pointer to 1-d Arrays – pointer notation

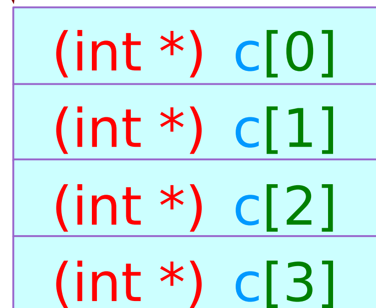
```
int    c [4][4];
int    (*p) [4][4] = &c;
int    (*q) [4]    = &c[0] ;
```

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

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

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

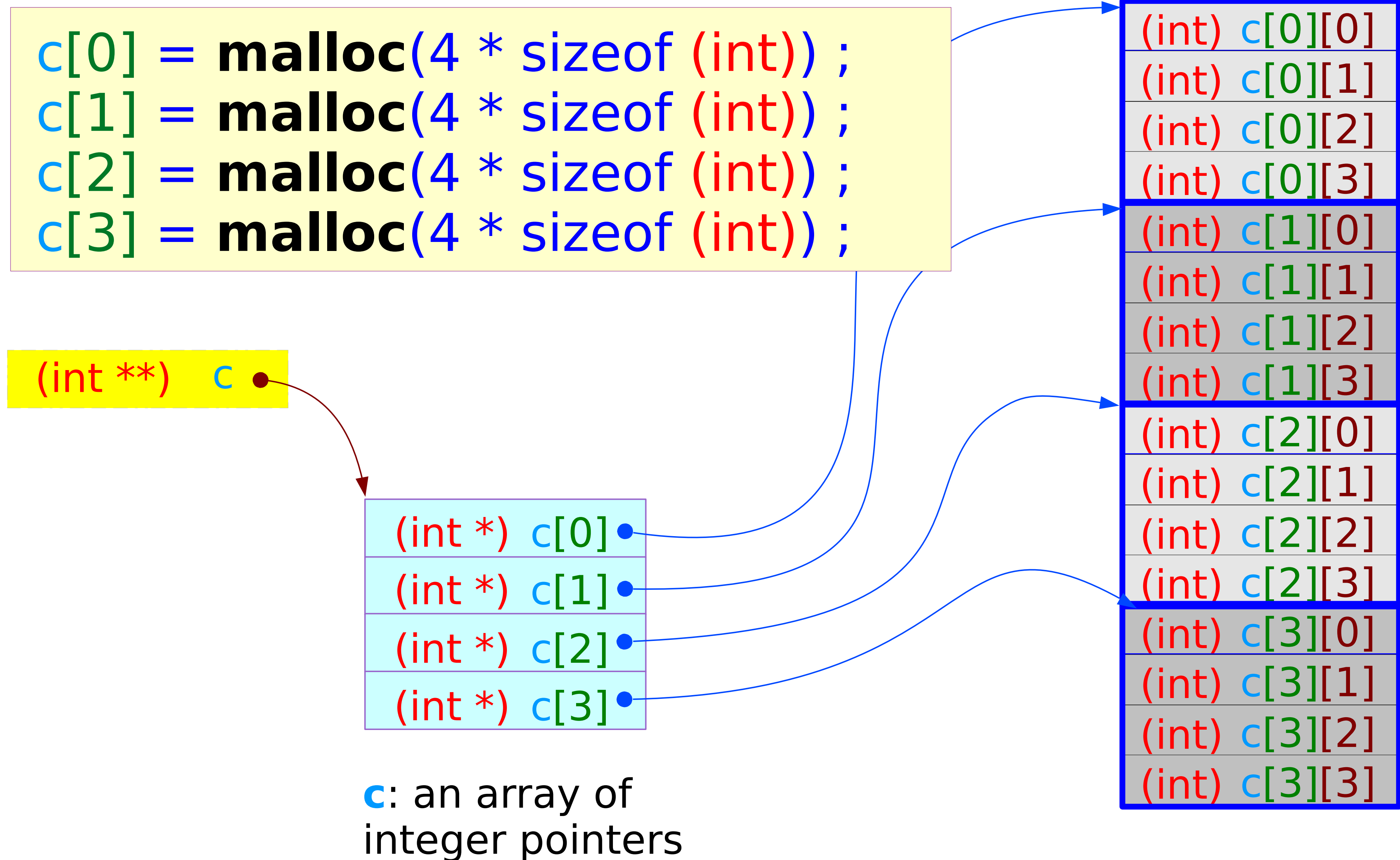
```
int    ** c ;

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



The intermediate array :
Allocated physically
in memory

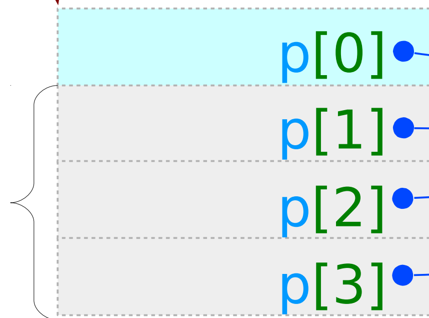
c: an array of
integer pointers

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

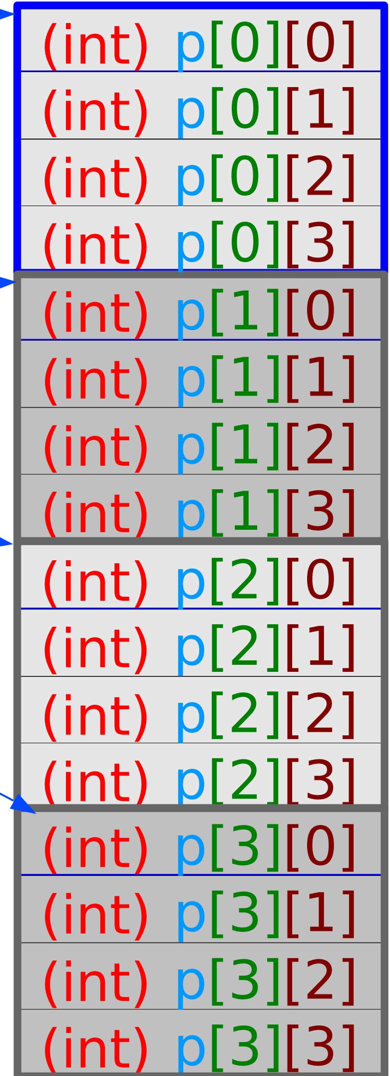


2-d array dynamic allocation : method 2

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

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



utilize pointer addition property

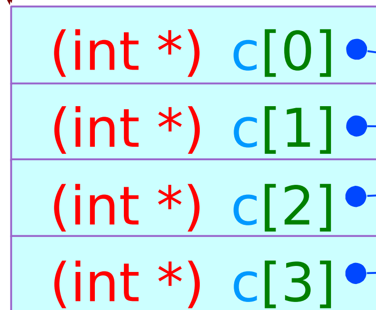


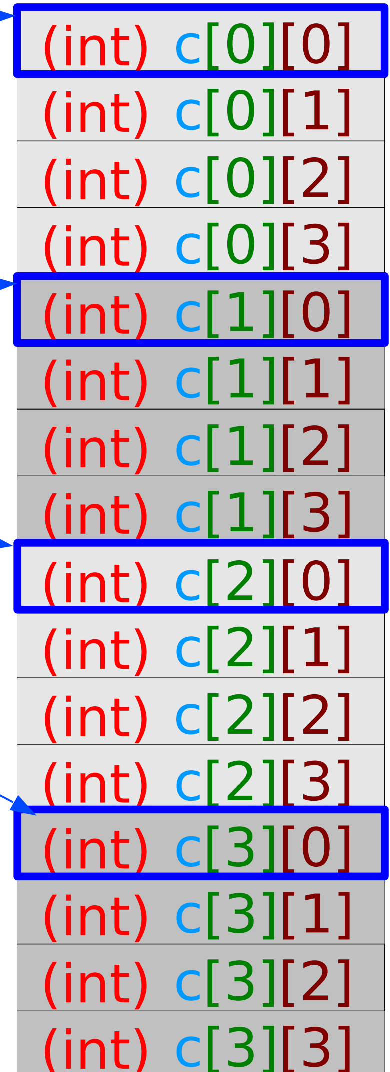
The intermediate array :
No physical allocation

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

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

(int **) c





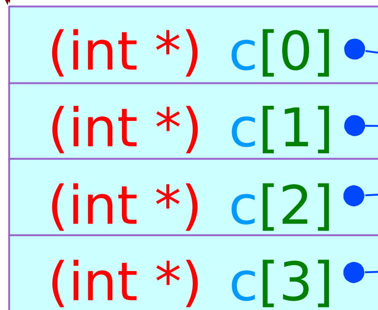
The intermediate array :
Allocated physically in memory

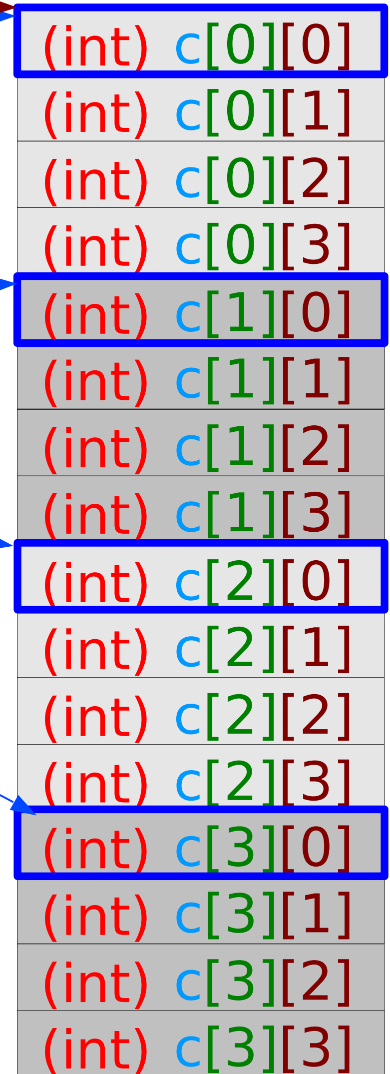
2-d array dynamic allocation : method 3 (b)

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

(int *) p

(int **) c





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

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