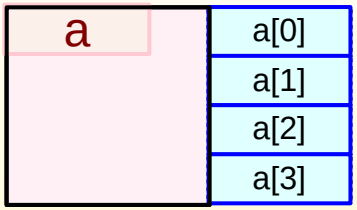
Applications of Arrays (1A)

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

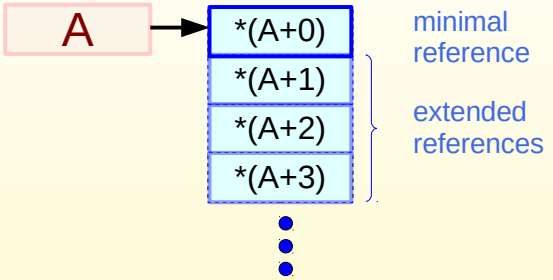
This document was produced by using LibreOffice.

Array **a** vs array pointer **A**

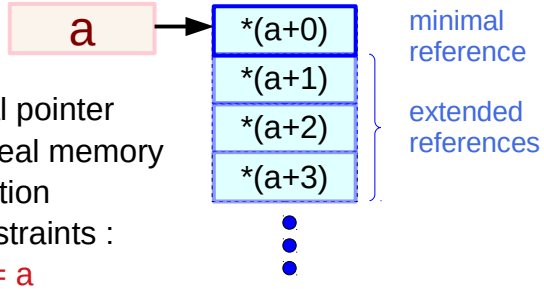
int a[4] ; **1-d** array a



int (*A) ; **0-d** array pointer A

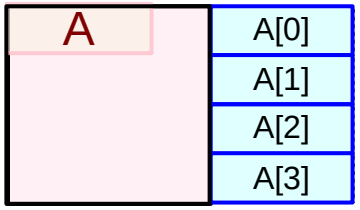


int (*) a as a **0-d** array pointer



virtual pointer
- no real memory location
- constraints :
  &a = a

int [N] A as a **1-d** array



N is not fixed to 4

sizeof(A) is not the size of the array but the size of a pointer variable

Array **a** and array pointers **A**

`int a[4] ;` **1-d** array a

- sizeof(a) = an array size
  = 4 * 4 bytes
- # of 0-d arrays = fixed
  = 4

`int (*A) ;` **0-d** array pointer A

- sizeof(A) = a pointer size
  = 4 / 8 bytes
- # of 0-d arrays = not fixed
  = at least 1

`int (*)` a as a **0-d** array pointer

a is not a real pointer
- sizeof(a) = an array size
- a = &a

`int [N]` A as a **1-d** array

A is not a real array
- sizeof(A) = a pointer size
- A ≠ &A

The name of a 2-d array

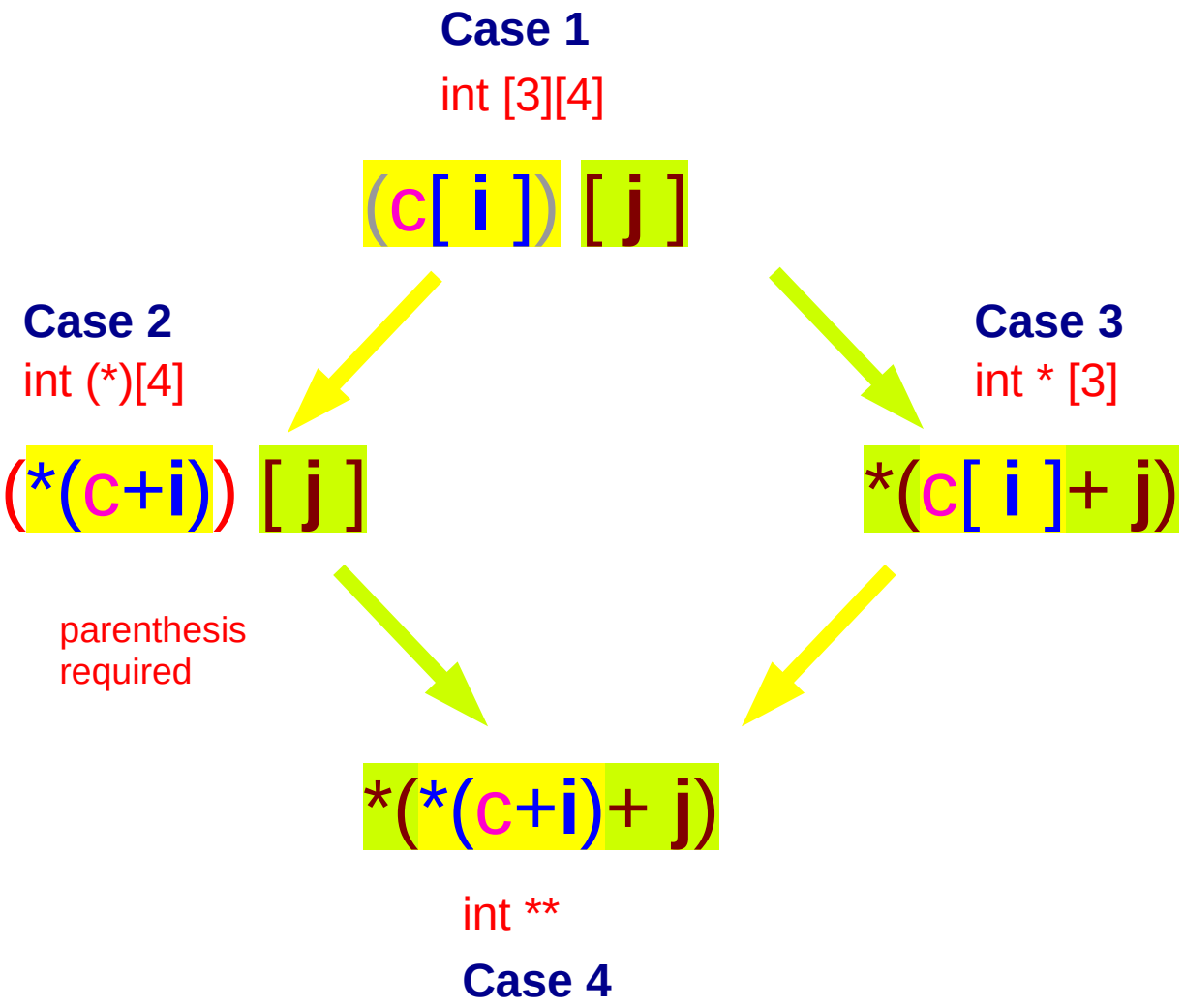
```
int         a [4];

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

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

2. a double pointer

3. a pointer to an array

2-d array access



Case 1) 2-d array c, 1-d array c[i]

int c [3] [4]

c **2-d array**

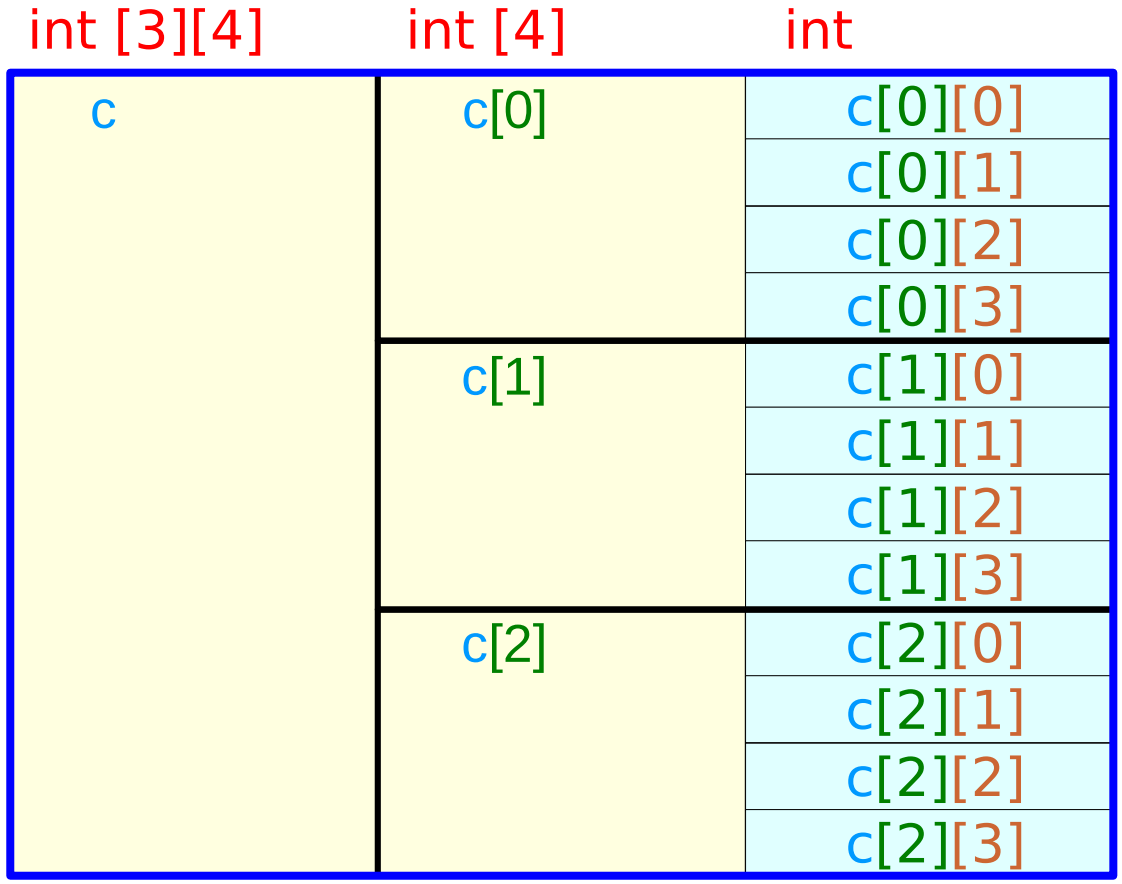
type : int [3][4]

int c [3] [4]

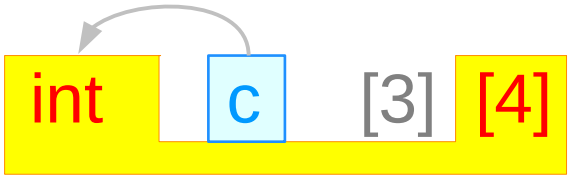
c[i] **1-d array**

type : int [4]

(c[i]) [j]



Case 2) 1-d array pointer c, 1-d array c[i]



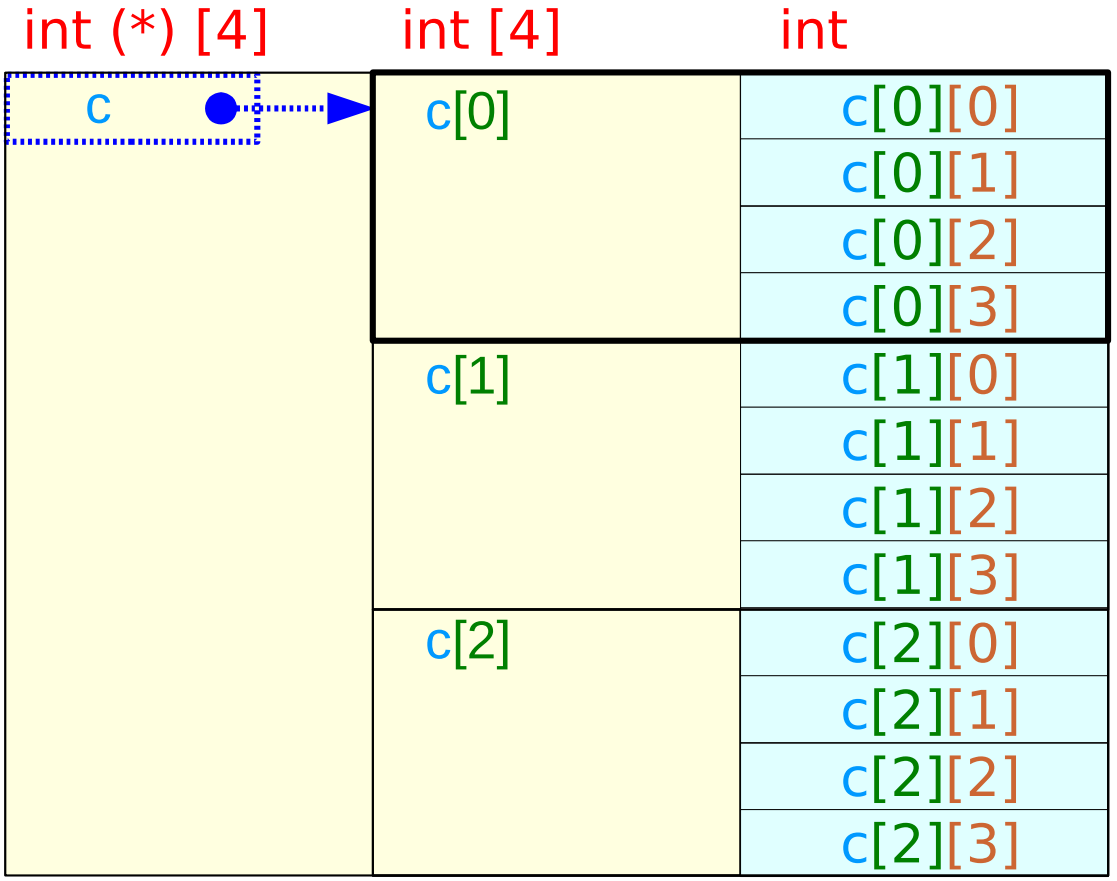
c **1-d array pointer**

type : int (*) [4]

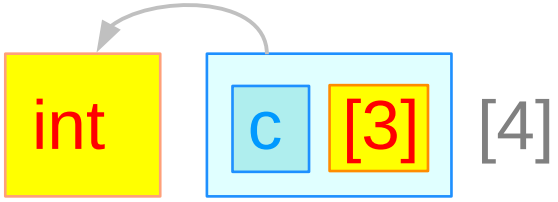
c[i] **1-d array**

type : int [4]

(*(c+i)) [j]

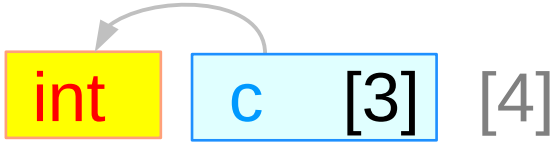


Case 3) 1-d array c, pointer c[i]



c **1-d array**

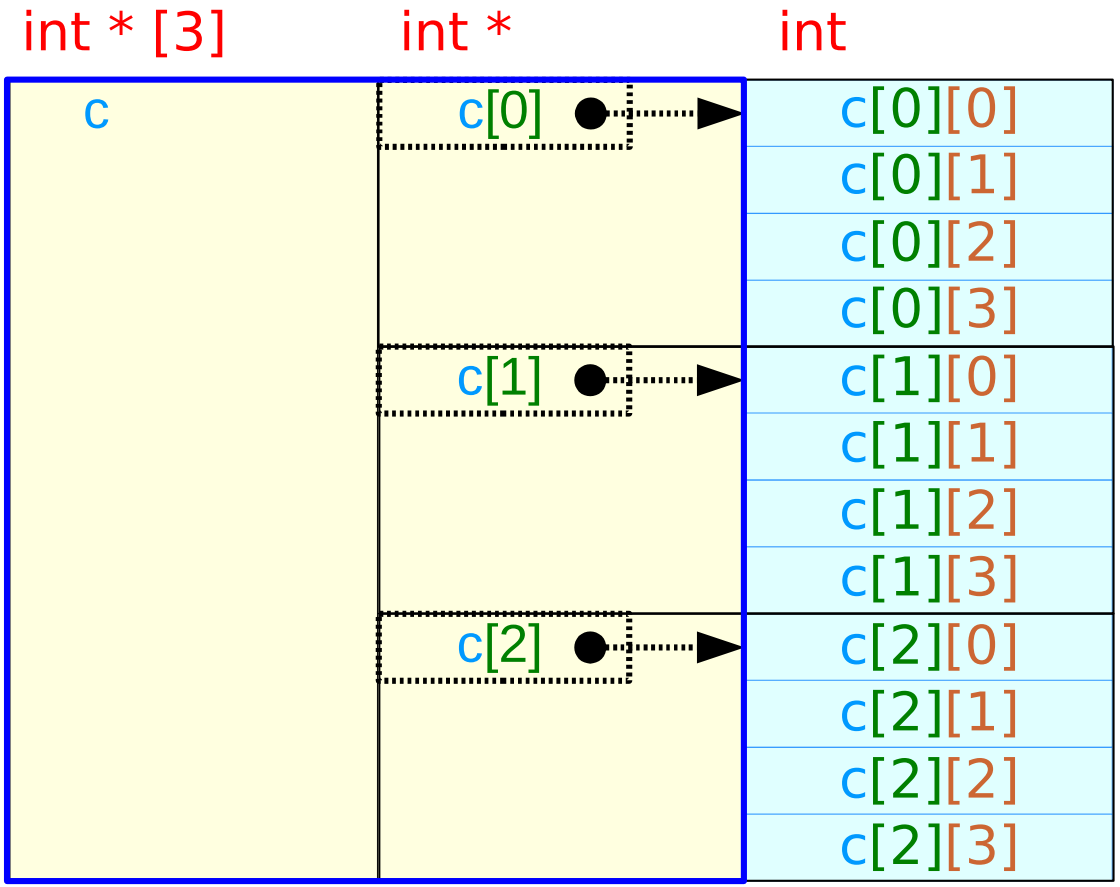
type : int * [3]



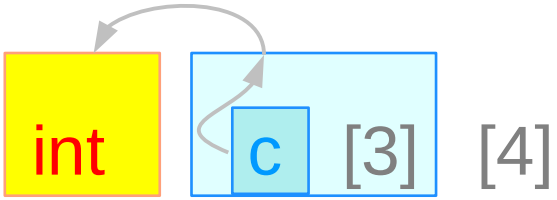
c[i] **pointer**

type : int *

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

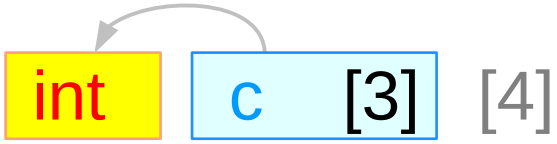


Case 4) double pointer **c**, pointer **c[i]**



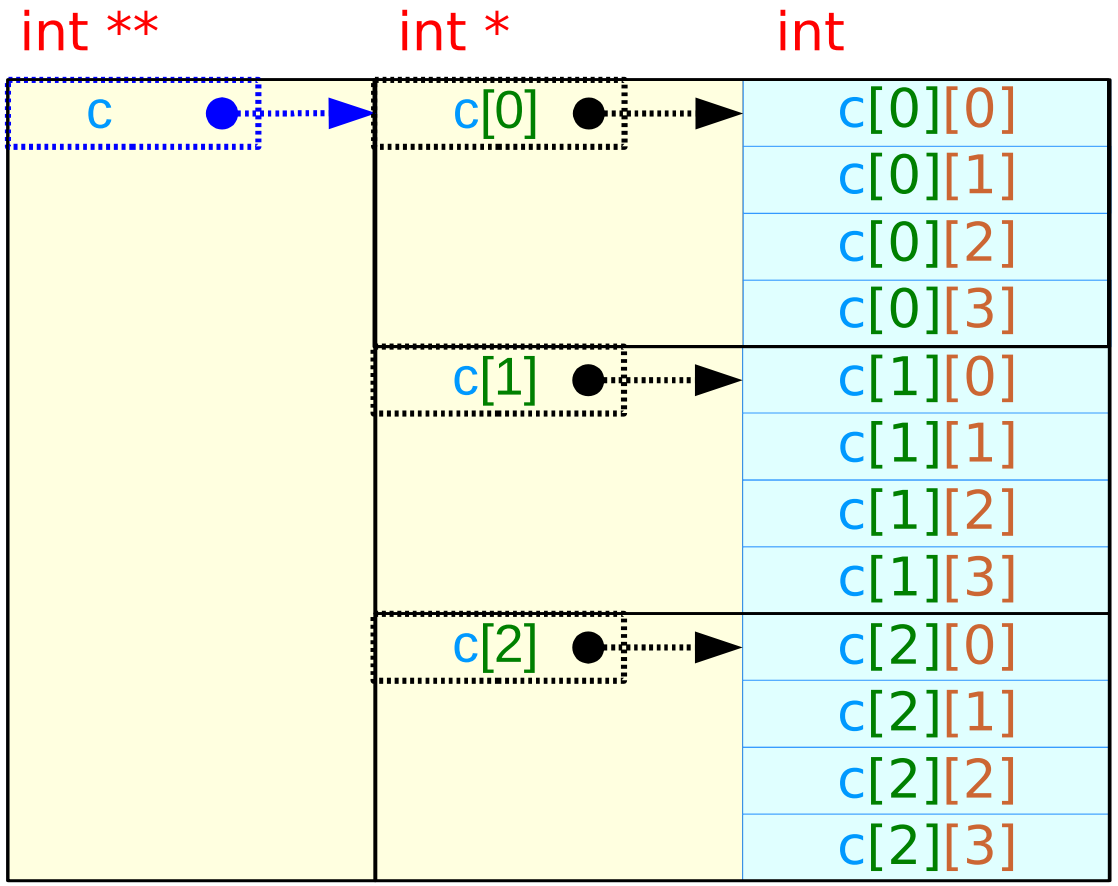
c **double pointer**

type : int **



c[i] **pointer**

type : int *



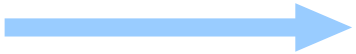
((c+i)+ j)

Types in a **2-d** array

int c [3] [4]

c **2-d array**
type : int [3][4]
size : 3 * 4 * 4
value : &c[0][0]

relaxing the 1st dimension



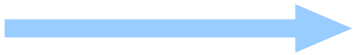
int c [3] [4]

c **1-d array pointer** (virtual)
type : int (*) [4]
size : 3 * 4 * 4
value : &c[0][0]

int c [3] [4]

c[i] **1-d array**
type : int [4]
size : 4 * 4
value : &c[i][0]

relaxing the 1st dimension



int c [3] [4]

c[i] **0-d array pointer** (virtual)
type : int (*)
size : 4 * 4
value : &c[i][0]

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

((c+0)+0) ➡ **c a double pointer

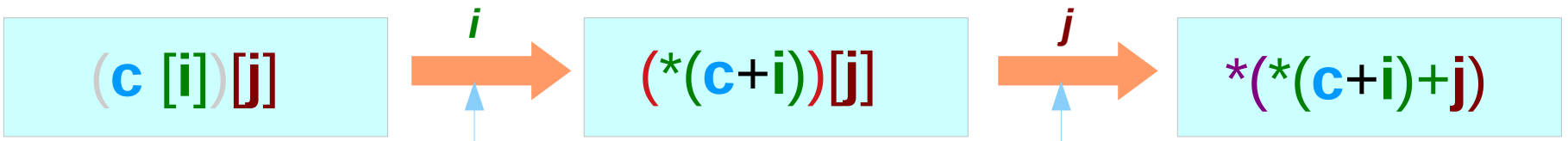
(*(c+0))[0] ➡ (*c)[0] a **1-d** array pointer

2-d array access via a double indirection

Case 1
int [3][4]

Case 2
int (*)[4]

Case 4
int **



relax the 1st dimension

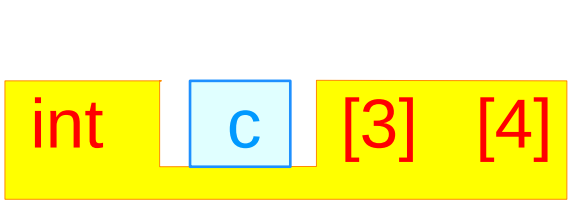
(c [i]) = (*(c+i))

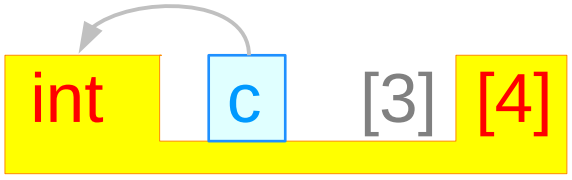
contiguous memory locations are assumed

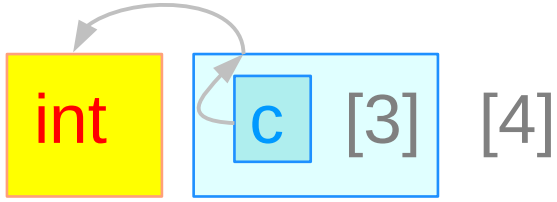
relax the 2nd dimension

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

contiguous memory locations are assumed





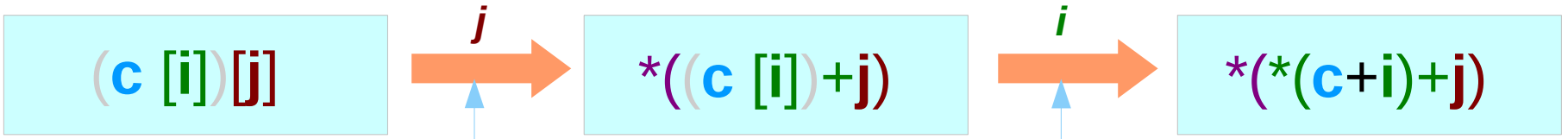


2-d array access via a double indirection

Case 1
int [3][4]

Case 3
int * [3]

Case 4
int **



relax the 2nd dimension

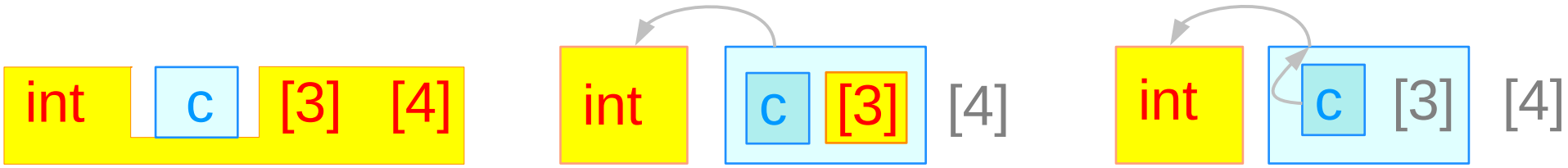
(_)[j] = *((_)+j)

contiguous memory locations are assumed

relax the 1st dimension

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

contiguous memory locations are assumed



Cases 1, 2, 4

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

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

Case 1
int [3][4]

Case 2
int (*) [4]

Case 4
int **

p = c

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

equivalence

Cases 1, 3, 4

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

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

Case 1
int [3][4]

Case 3
int * [3]

Case 4
int **

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

p = q;

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

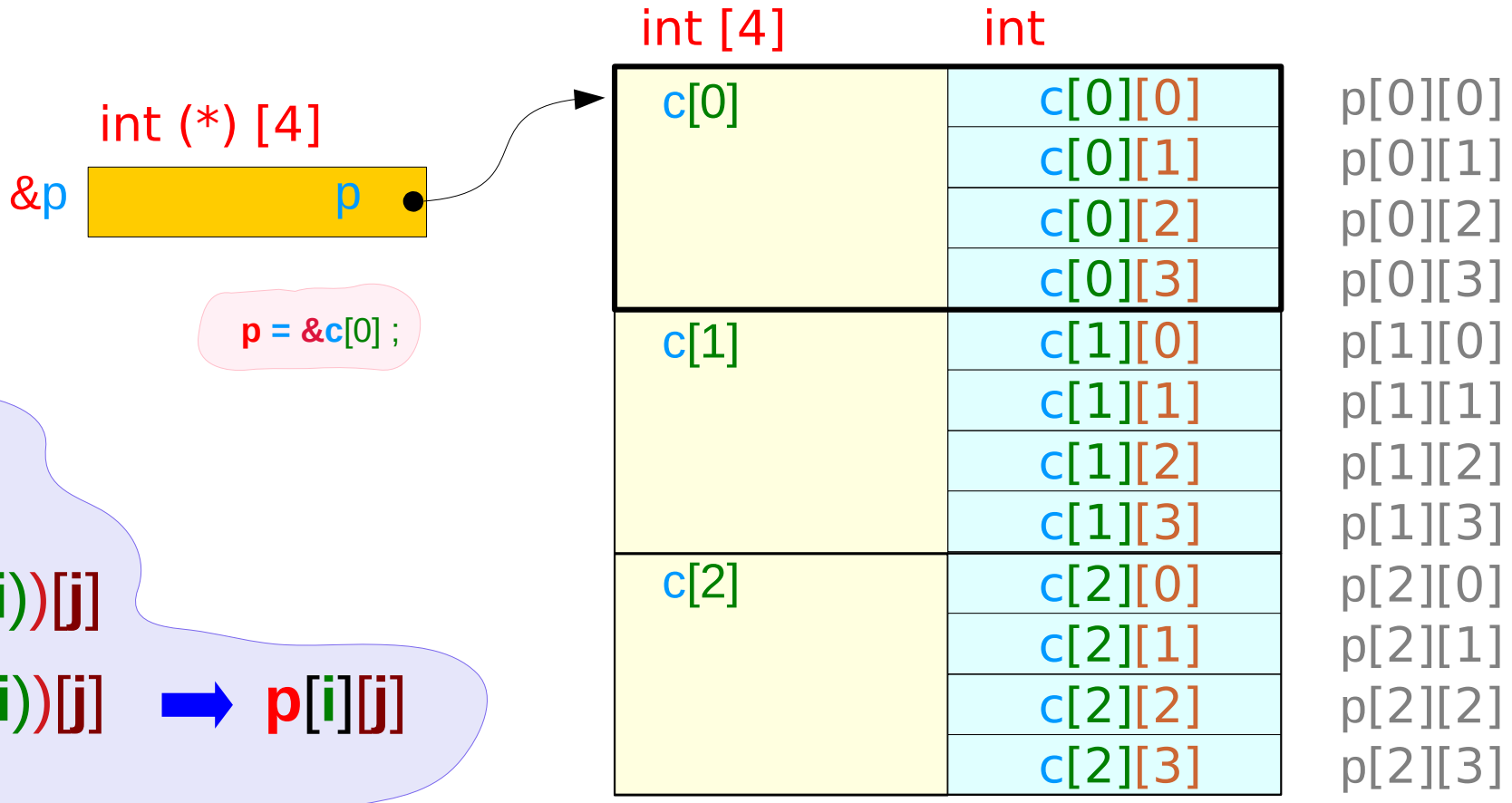
must be allocated and initialized

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

2-d array access using an array pointer p

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

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



Case 2
int (*)[4]

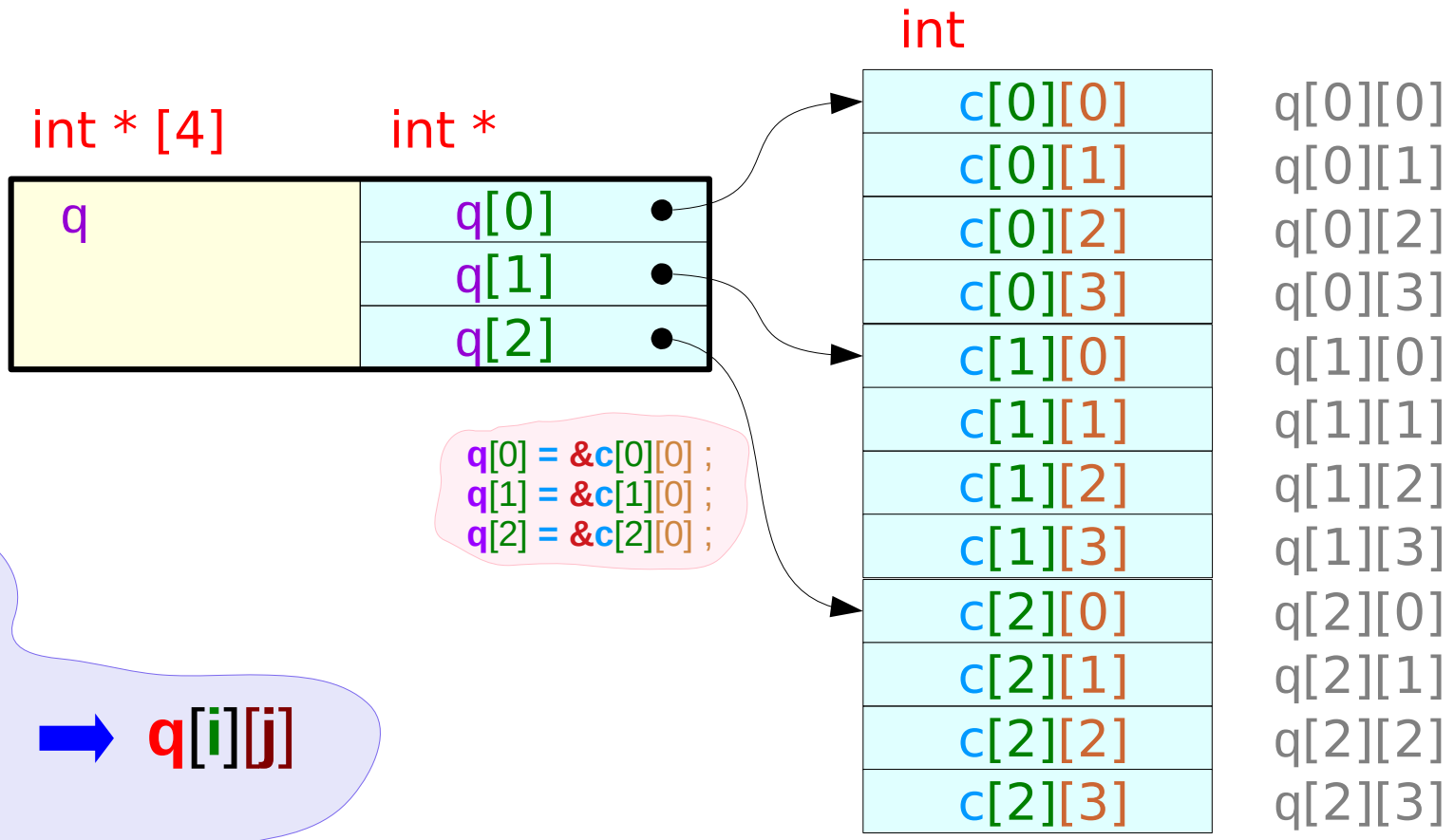
(*(c+i))[j]

(*(p+i))[j] ➡ p[i][j]

2-d array access using a pointer array q

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

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



Case 3

int * [3]

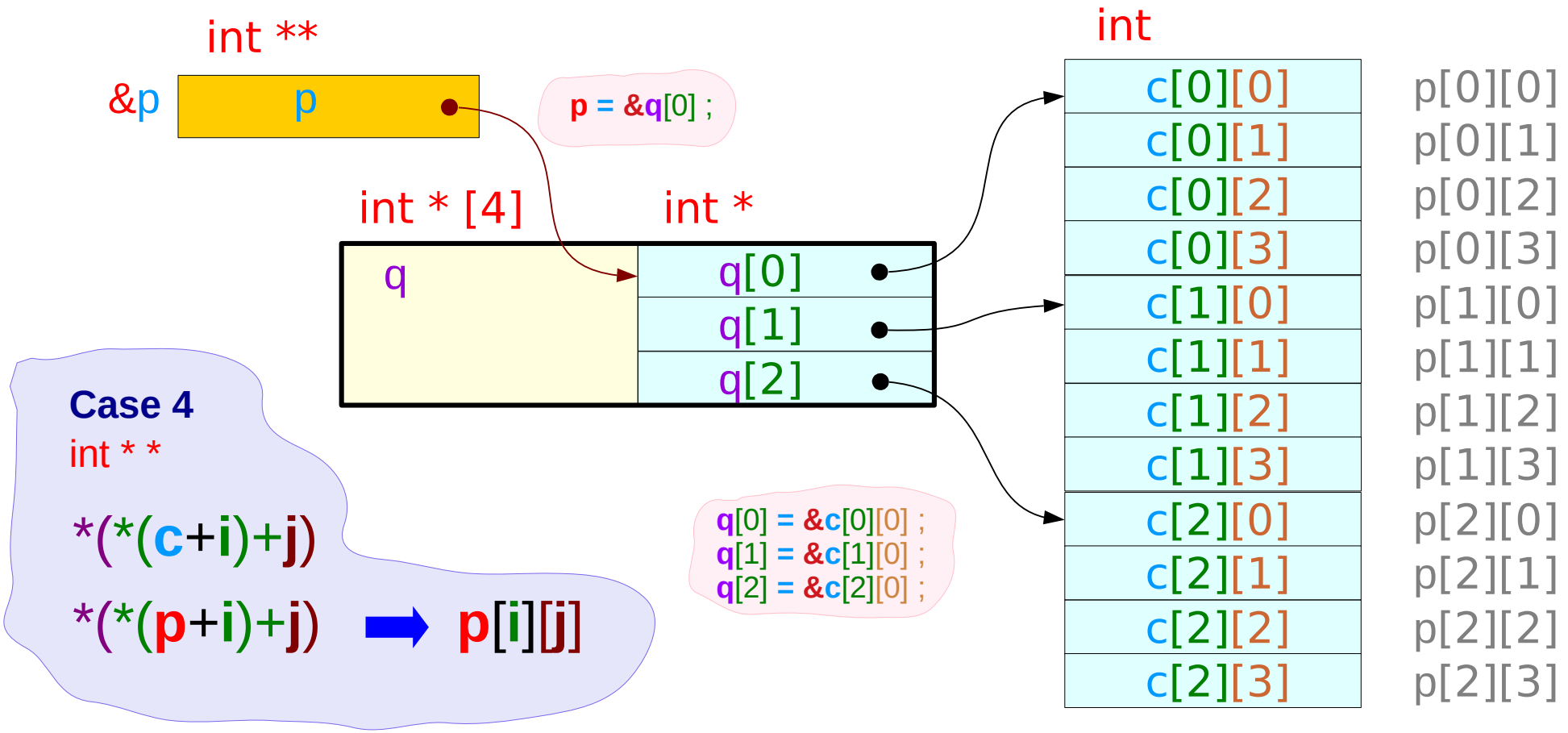
*((c [i])+j)

*((q [i])+j) ➡ q[i][j]

2-d array access using double pointers q

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

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

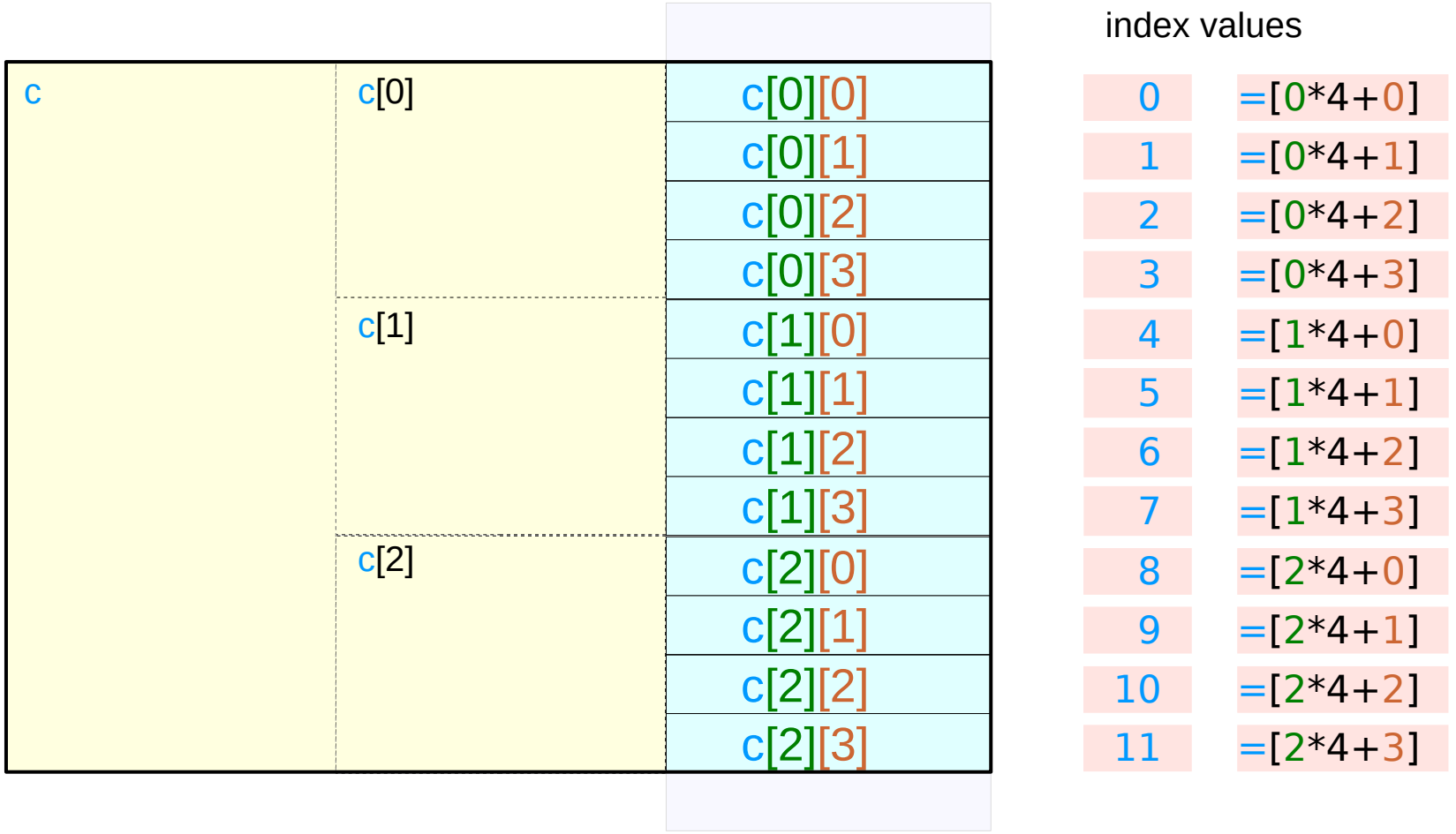


A **2-d** array stored as a **1-d** array (row major order)

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

c[i][j]

[i*4+j]



index values

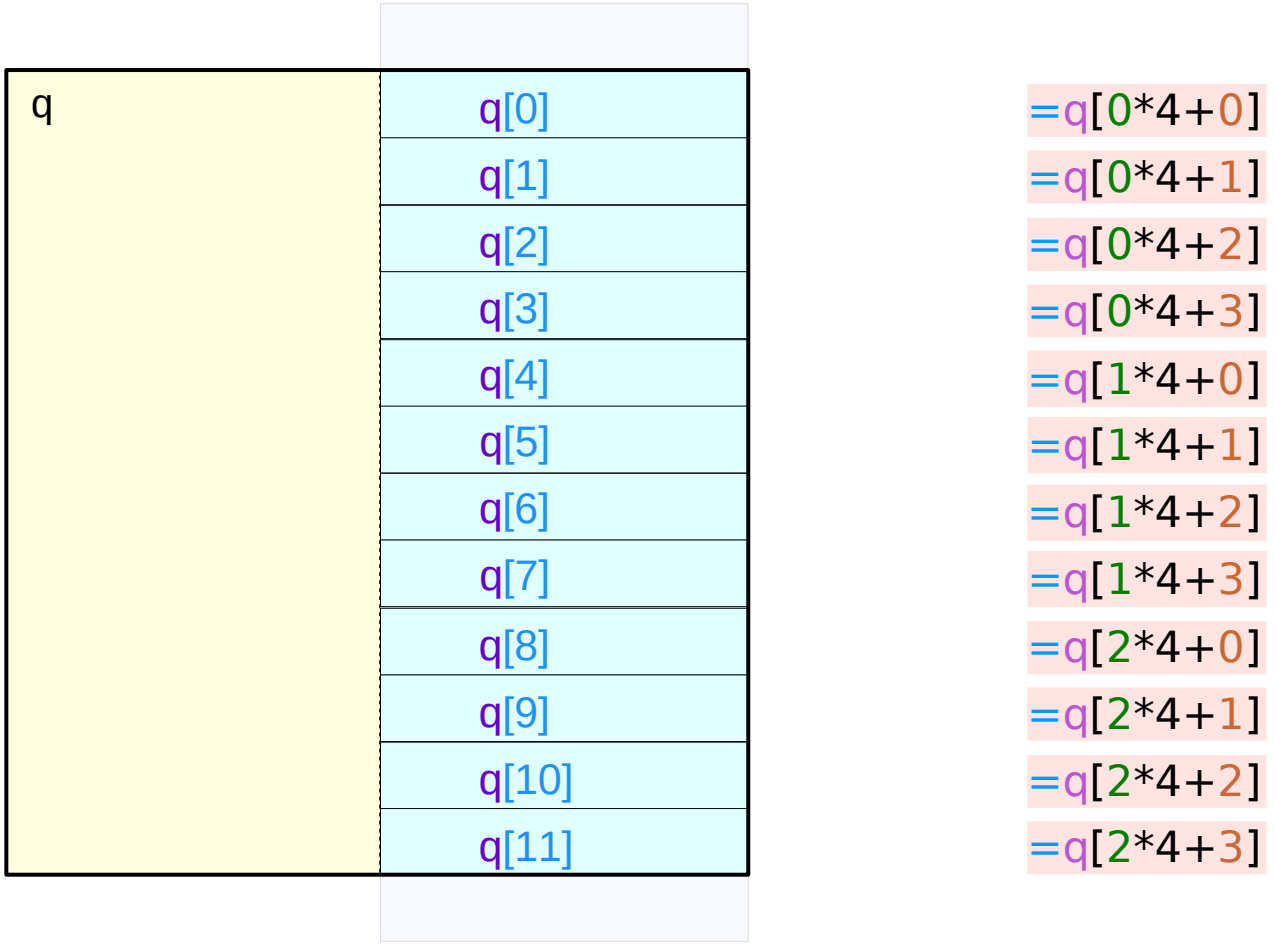
0	=[0*4+0]
1	=[0*4+1]
2	=[0*4+2]
3	=[0*4+3]
4	=[1*4+0]
5	=[1*4+1]
6	=[1*4+2]
7	=[1*4+3]
8	=[2*4+0]
9	=[2*4+1]
10	=[2*4+2]
11	=[2*4+3]

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

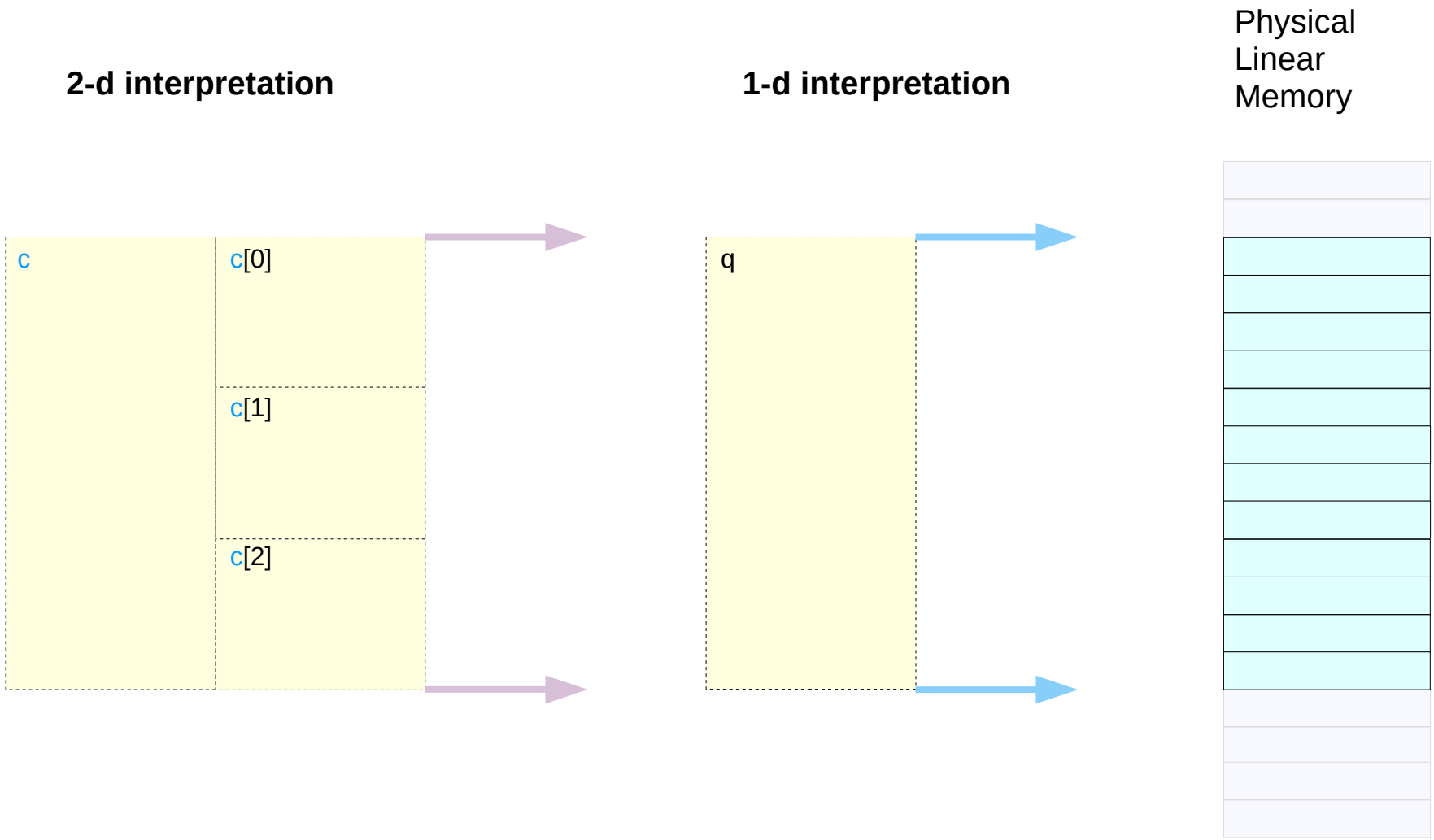
```
int q [4*4];
```

[k]

[i*4+j]



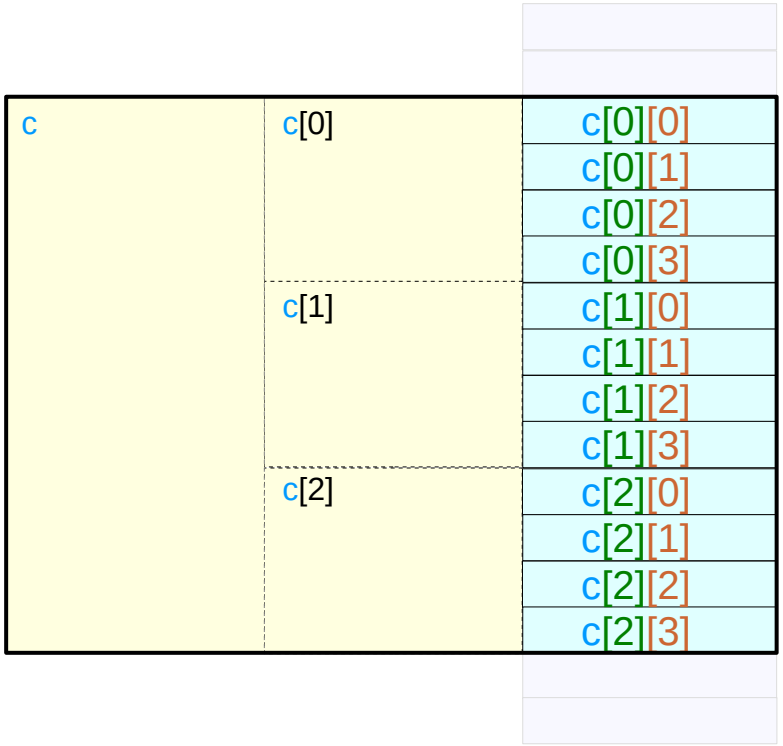
2-d and 1-d interpretations of linear memory

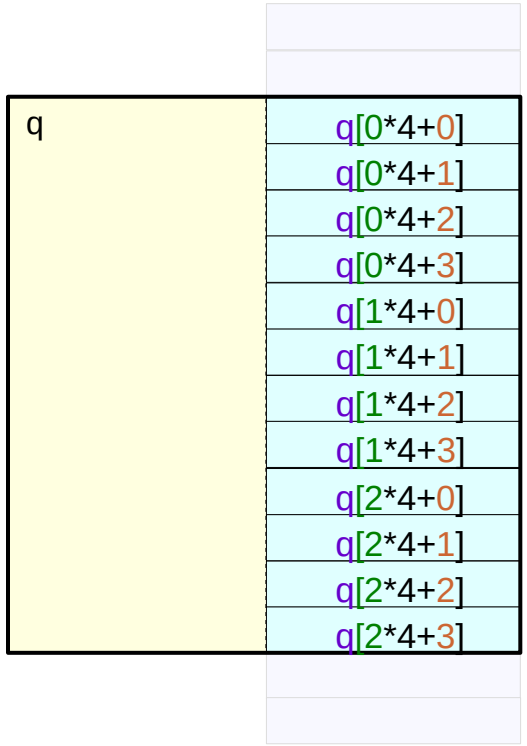


A **2-d** array stored as a **1-d** array (row major order)

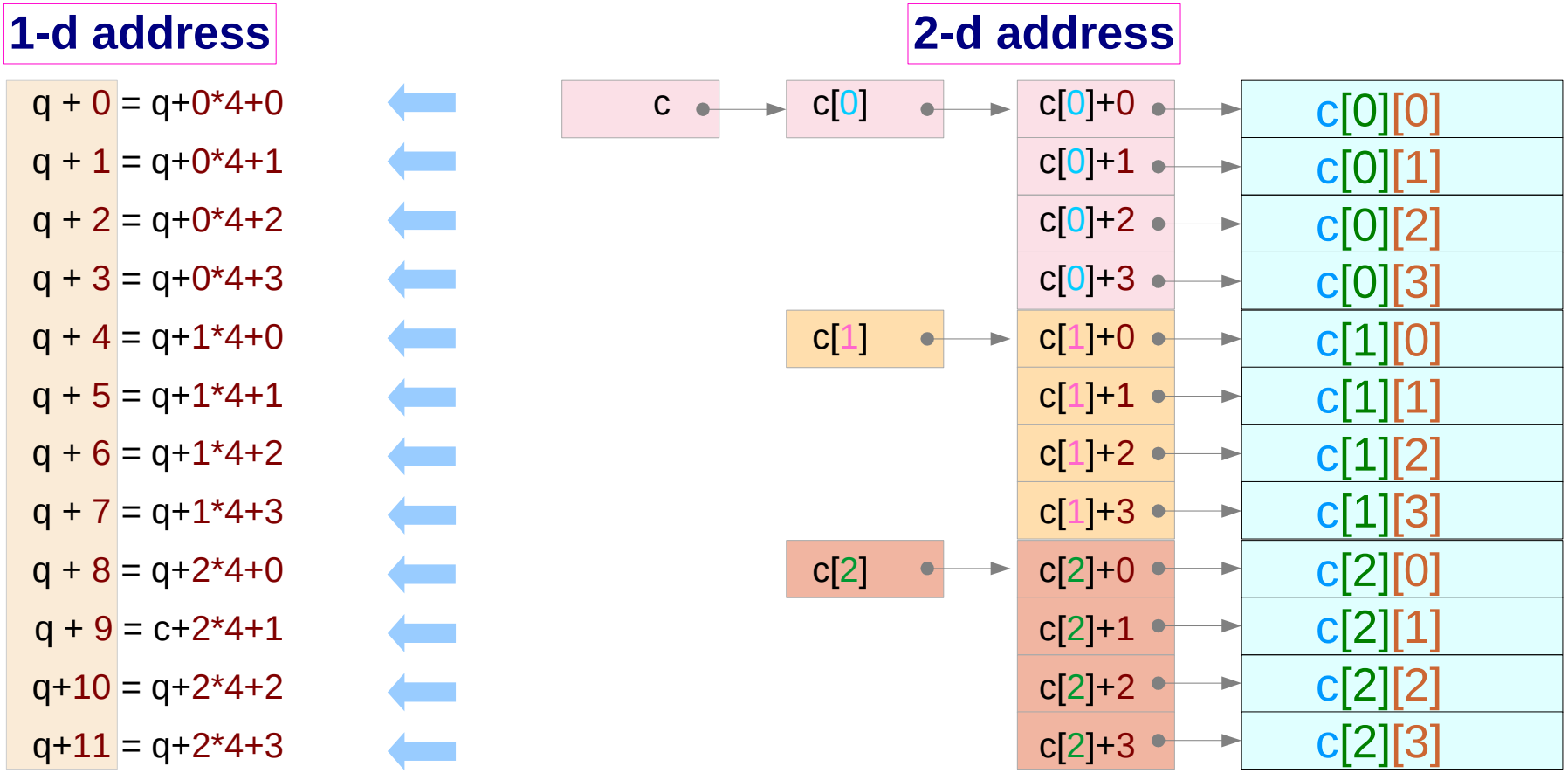
```
int c [4] [4];
```

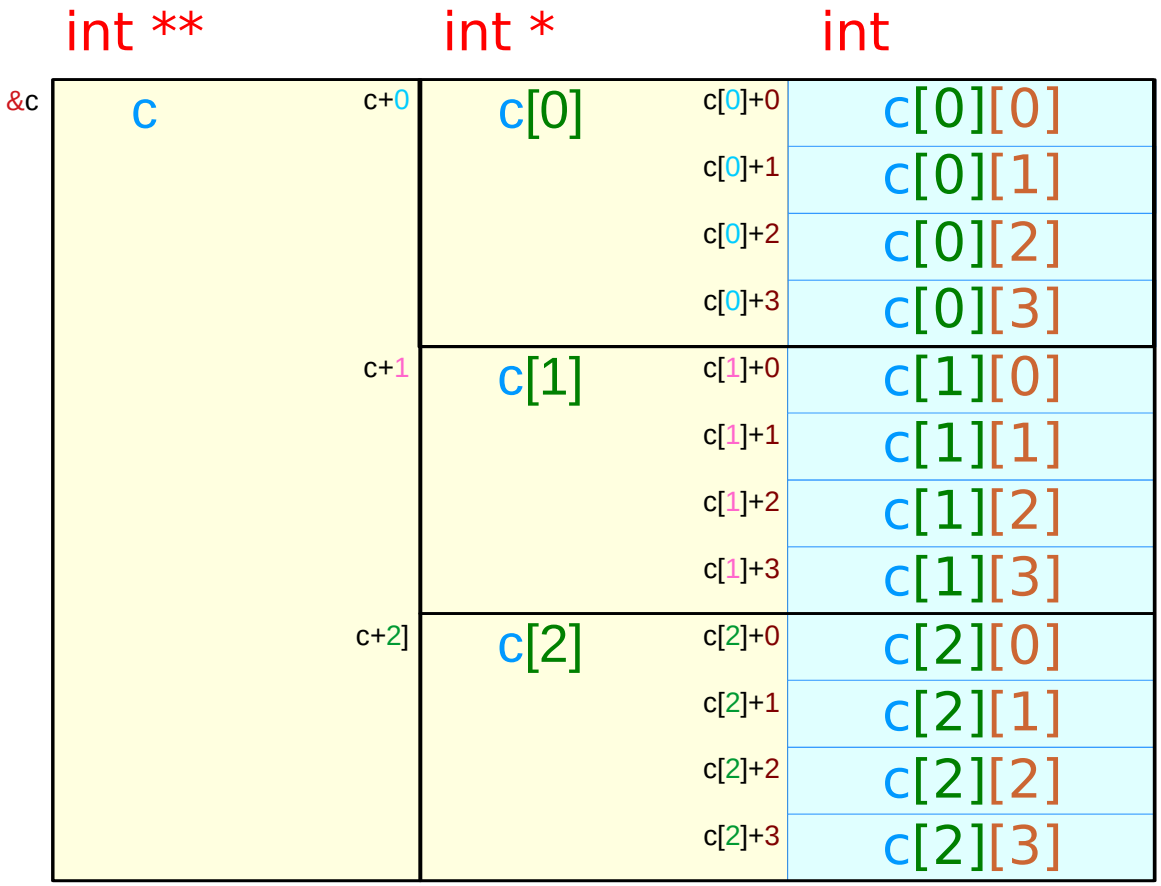
```
int q [4*4];
```





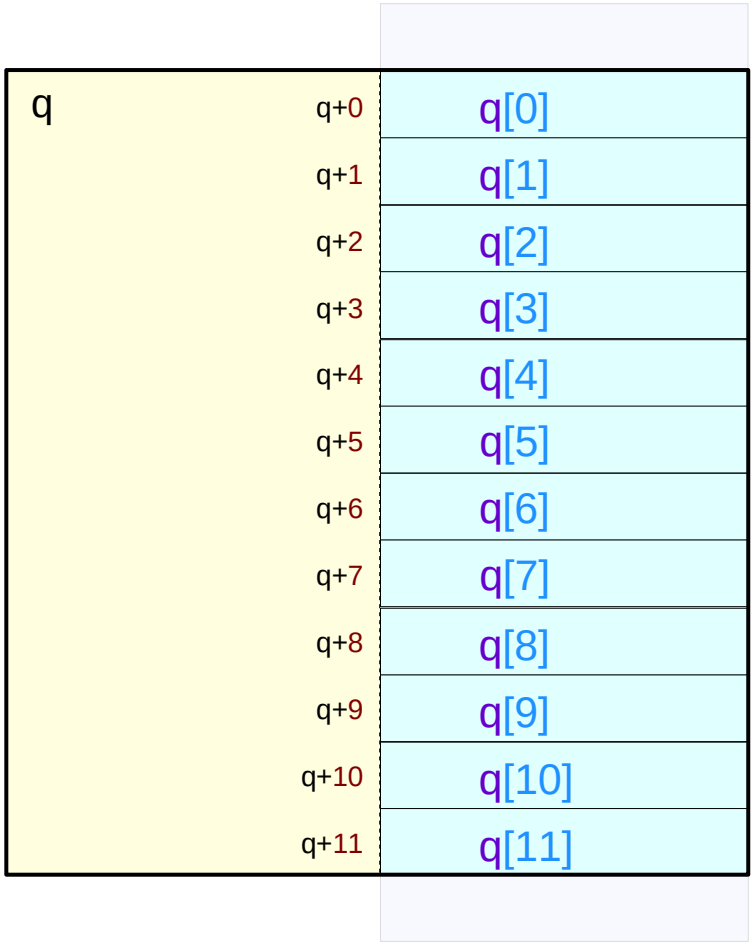
A Linear Memory Address



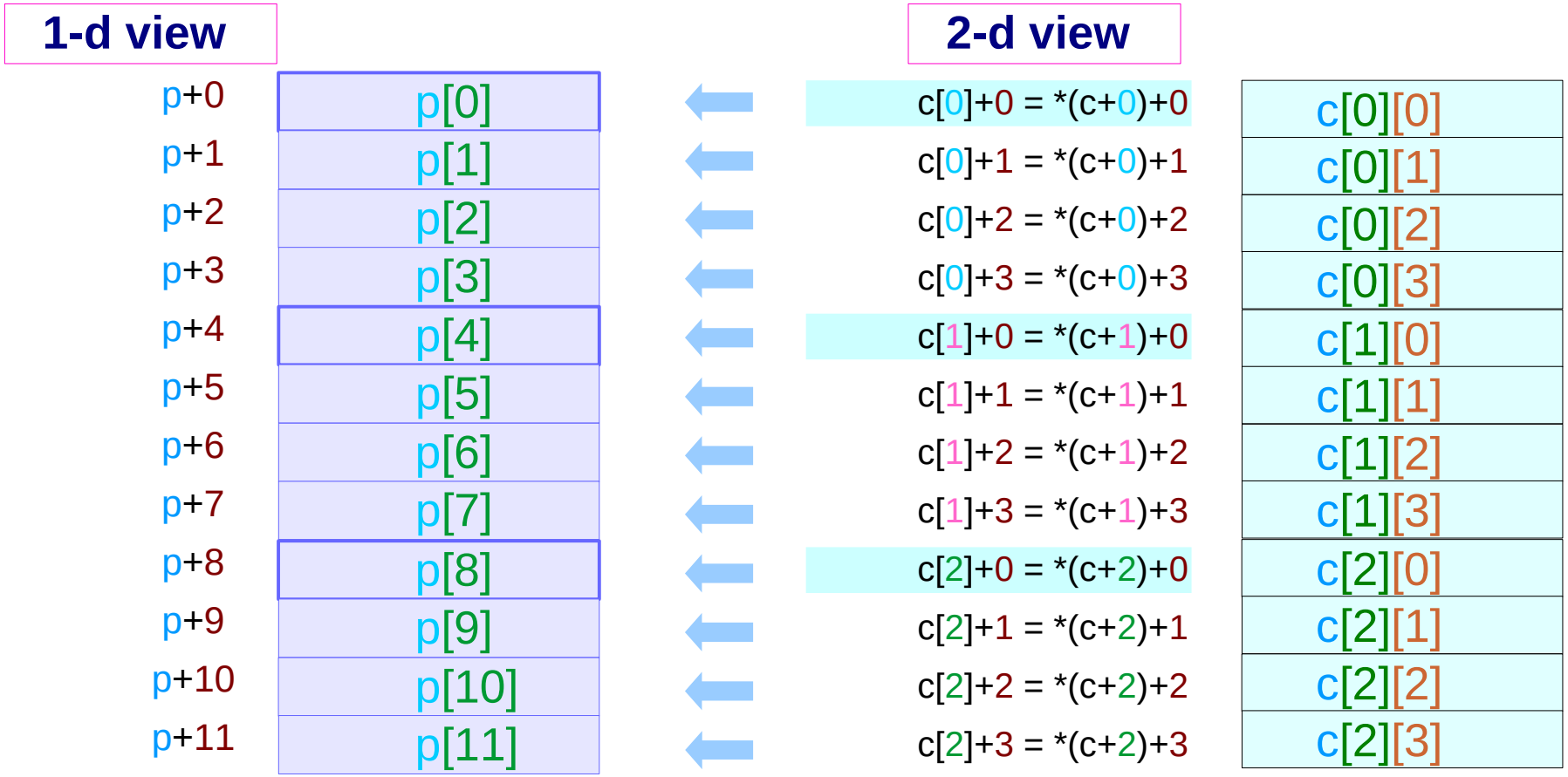

int **
int *
int
&c
c
c+0
c[0]
c[0]+0
c[0][0]
c[0]+1
c[0][1]
c[0]+2
c[0][2]
c[0]+3
c[0][3]
c+1
c[1]
c[1]+0
c[1][0]
c[1]+1
c[1][1]
c[1]+2
c[1][2]
c[1]+3
c[1][3]
c+2]
c[2]
c[2]+0
c[2][0]
c[2]+1
c[2][1]
c[2]+2
c[2][2]
c[2]+3
c[2][3]

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

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



A linearization of a 2-D array



2-d array access via a single pointer

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

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

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

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

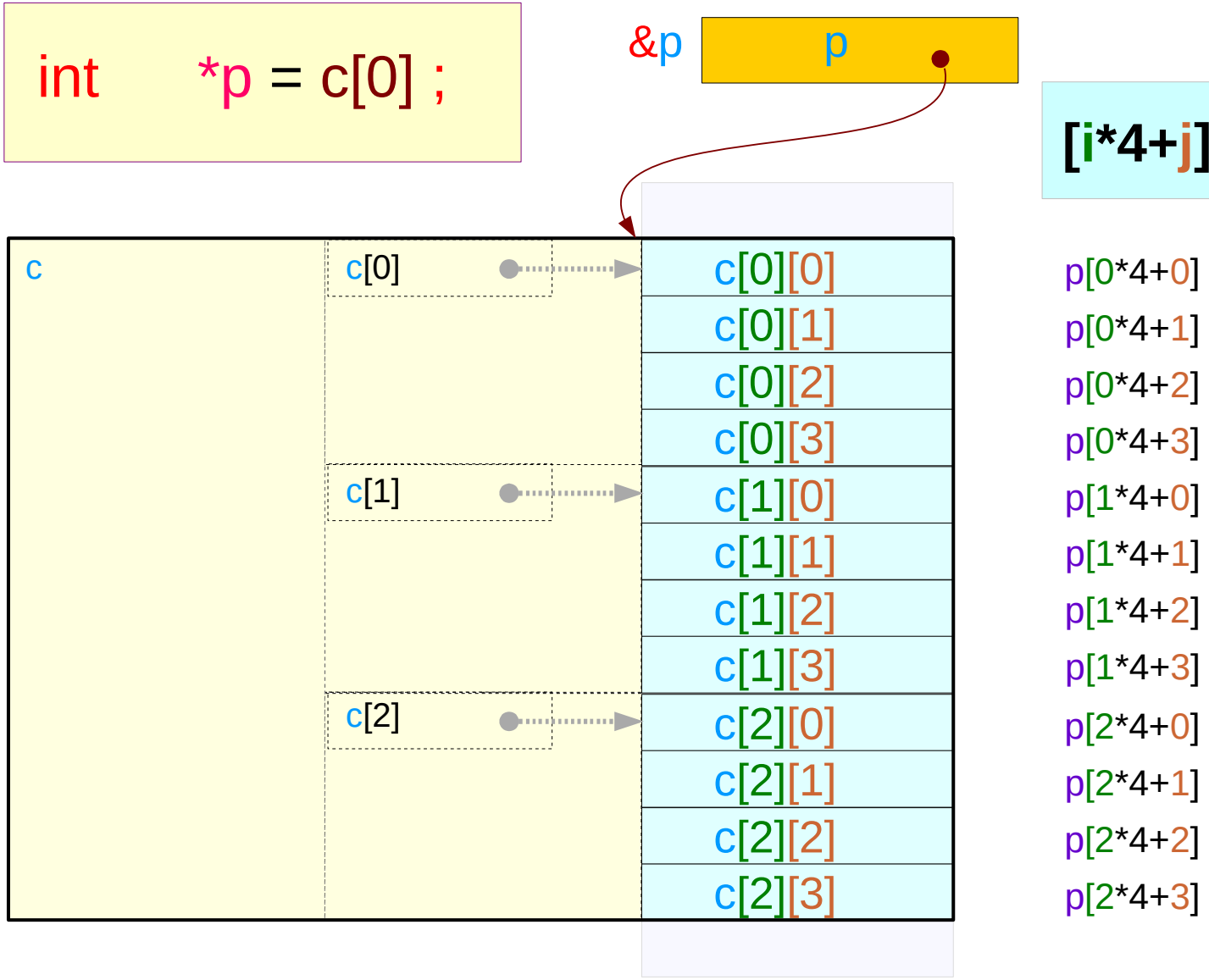
```
*(p+ i*4 + j)
```

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

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

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

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



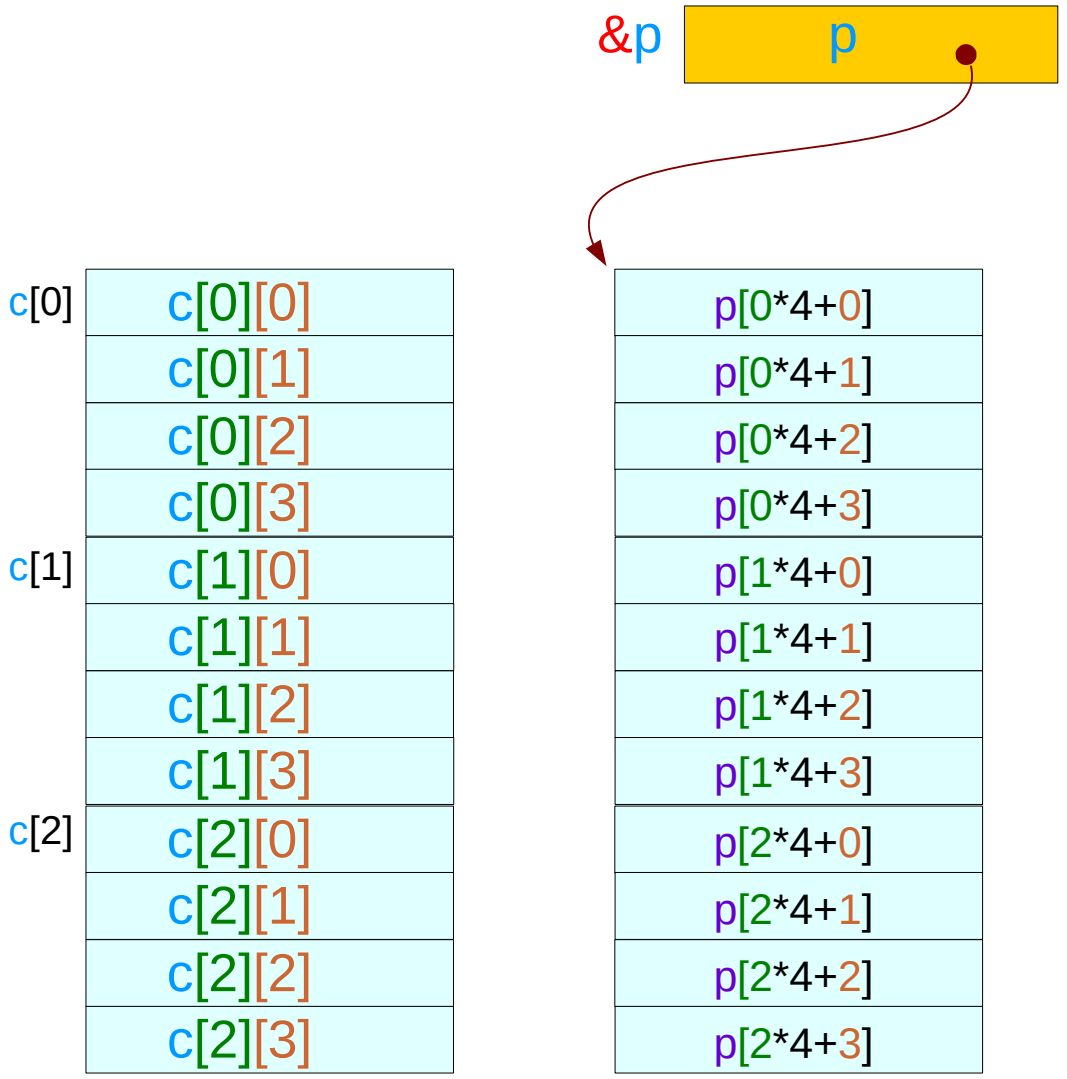
2-d array index vs 1-d array index

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

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

c[i][j]

p[i*4+j]



2-d array access via pointers

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

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

1. recursive pointers

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

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

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

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

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

```
int **q;
```

2. linear array pointers

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

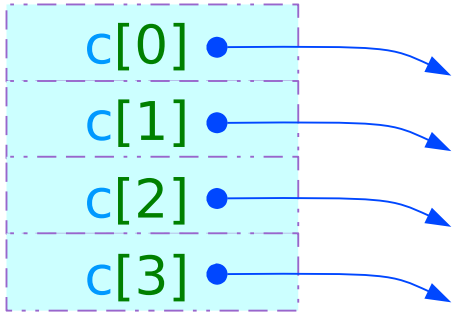
```
*(p+ i*4 + j )
```

Static Allocation of a 2-d Array

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

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) ]
```



The pointer array :
not allocated
in the memory

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

Dynamic Memory Allocation of 2-d Arrays

1. method 1

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

2. method 2

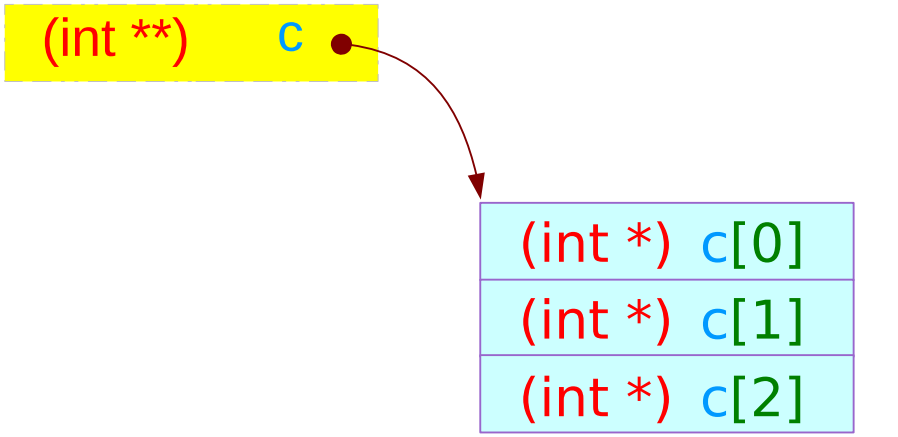
```
int   (*p) [3] ;
p = malloc(3 * 4 * sizeof (int)) ;
```

3. method 3

```
int   **    c ;
int   *     p ;
c = malloc( 3 * 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(3 * sizeof (int *)) ;
```

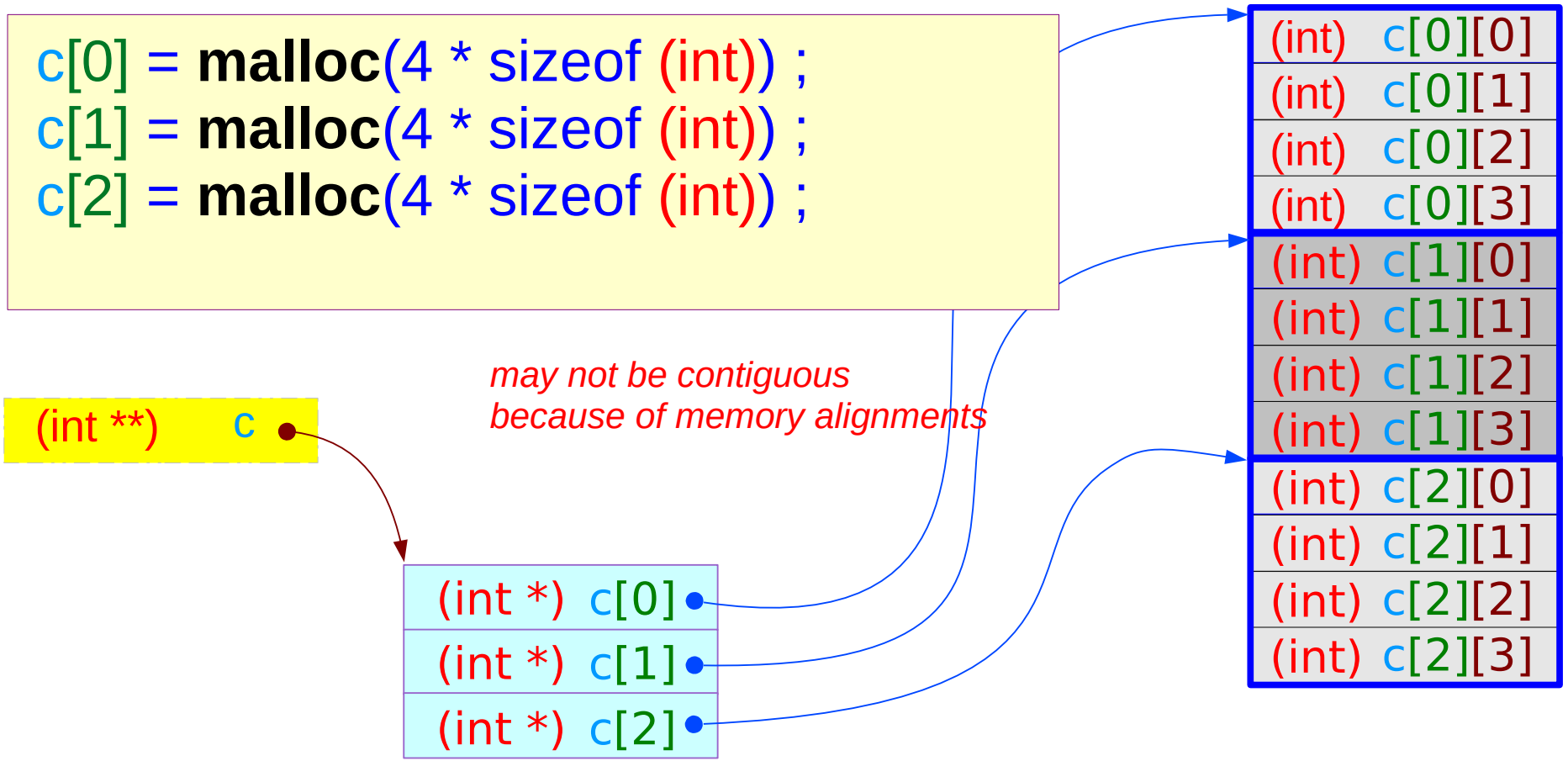


array of pointers :
allocated physically
in memory

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)) ;
```

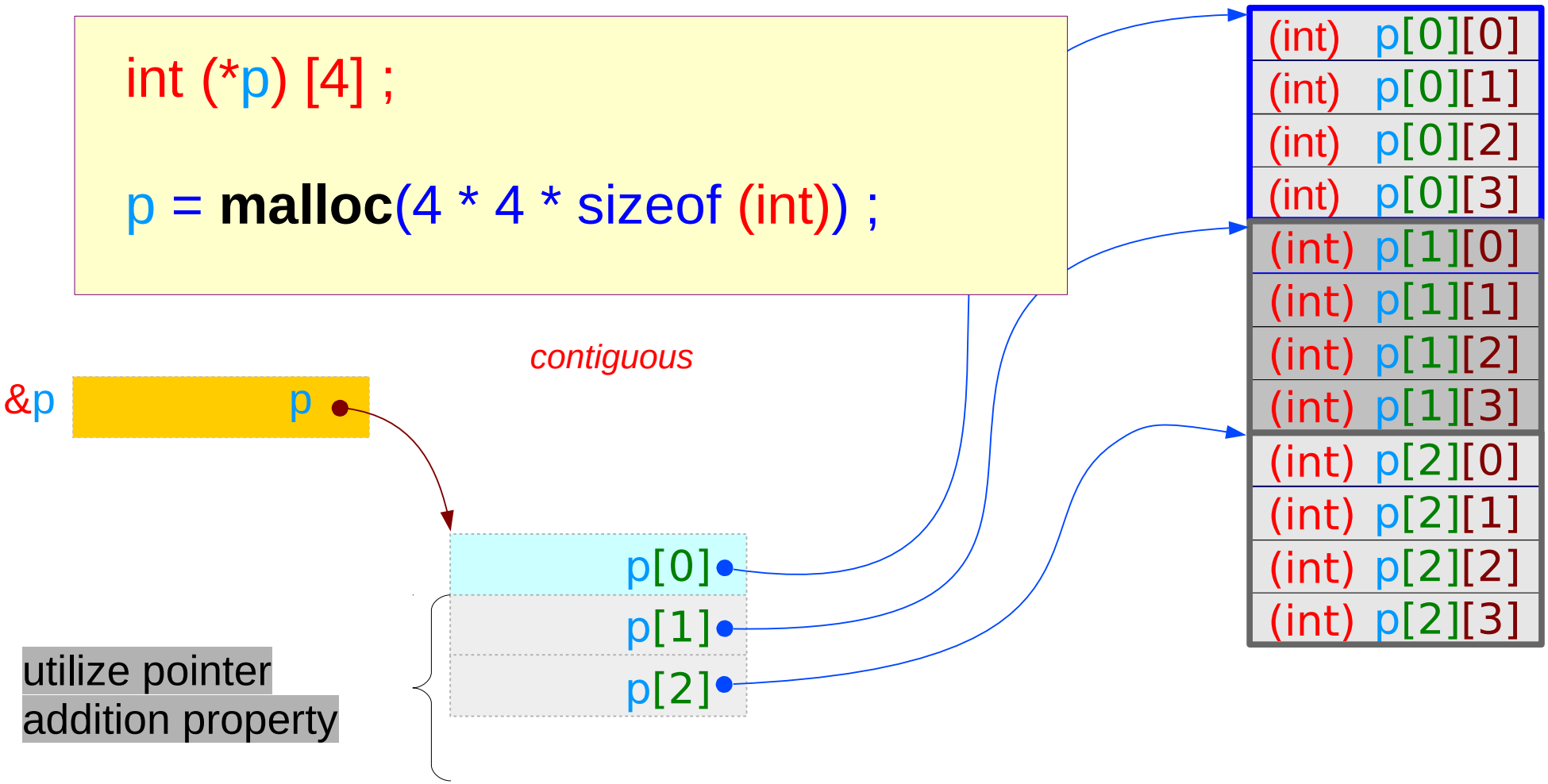


may not be contiguous
because of memory alignments

c: an array of
integer pointers

2-d array dynamic allocation : method 2

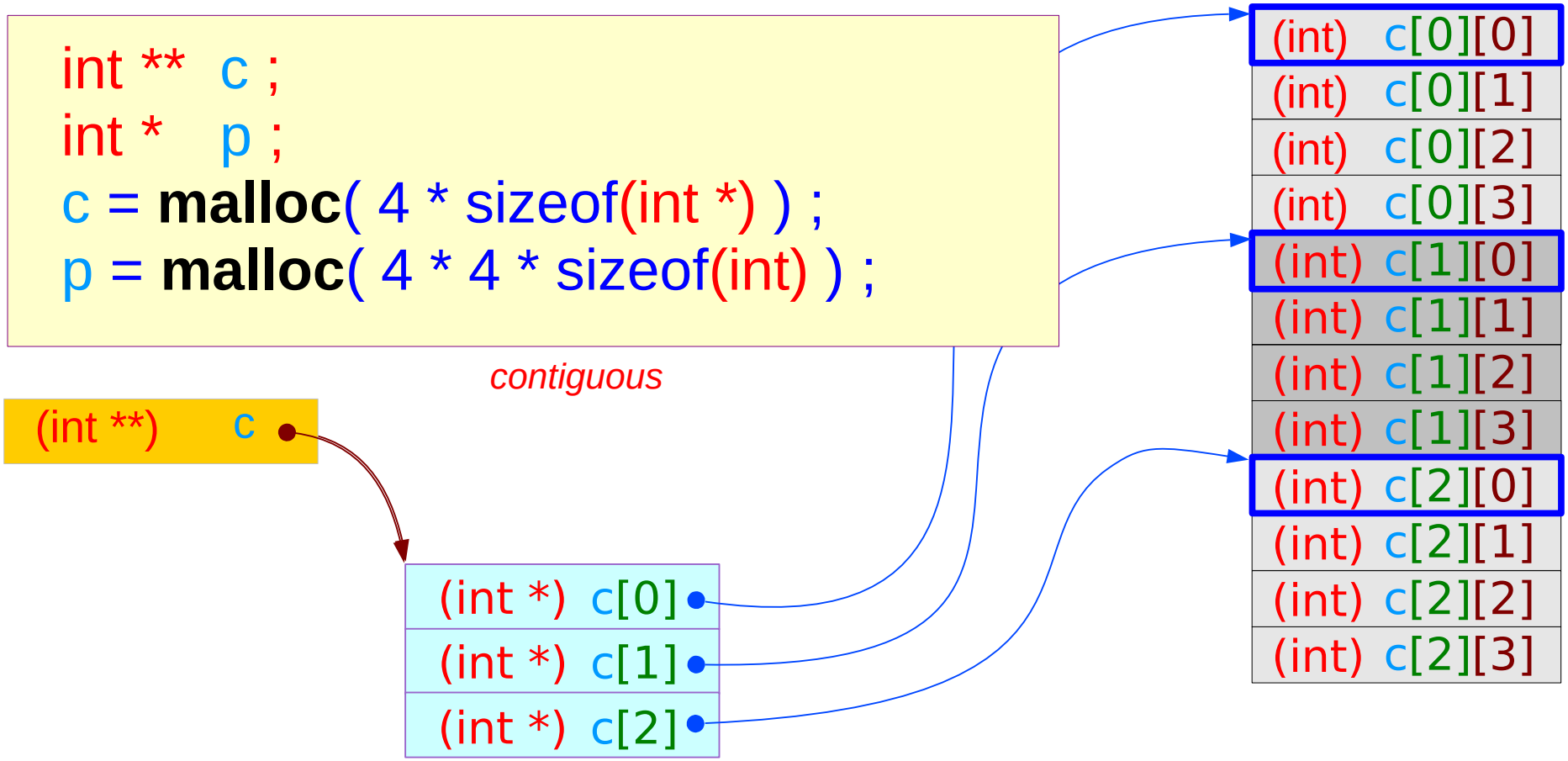
```
int (*p) [4] ;

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) ) ;
```

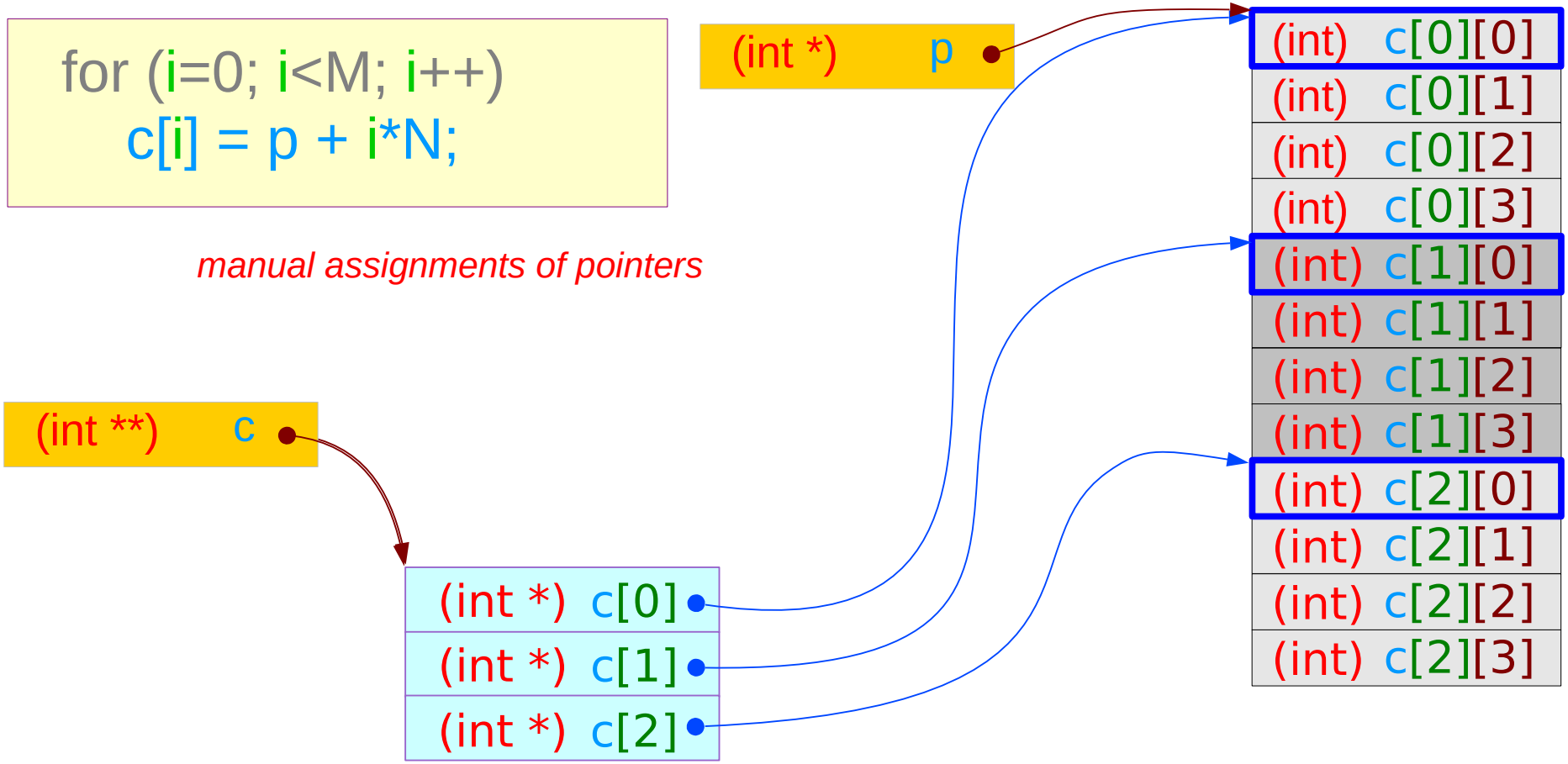


array of pointers
allocated physically in memory

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

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