Applications of Arrays (1A)

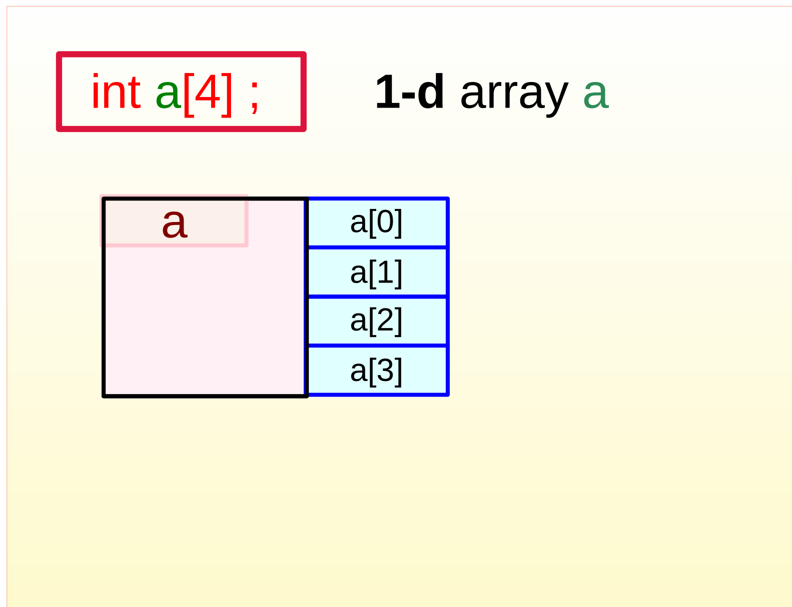
Young Won Lim
2/1/23

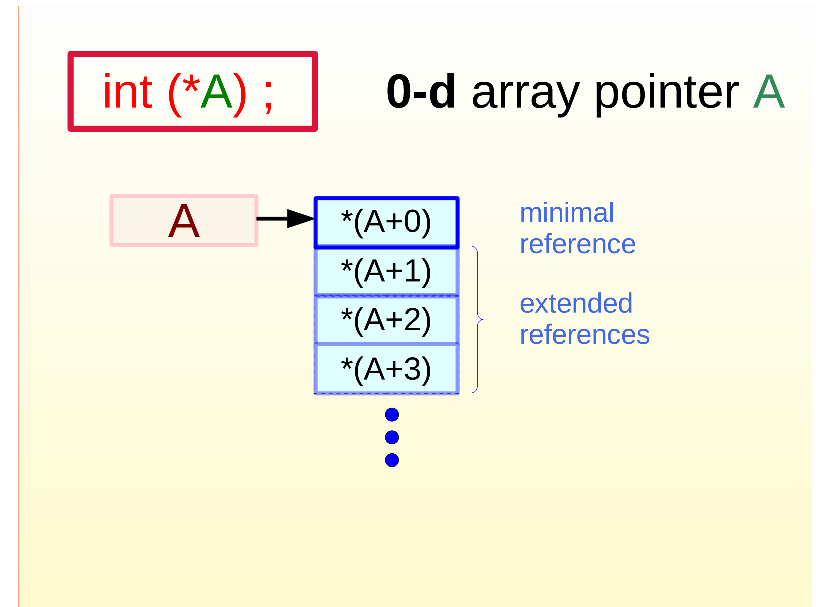
Copyright (c) 2023 - 2010 Young W. Lim.
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover 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.

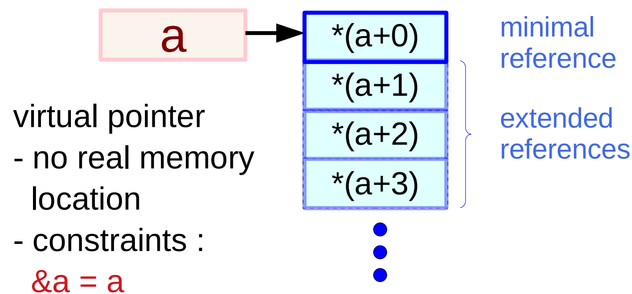
This document was produced by using LibreOffice.

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



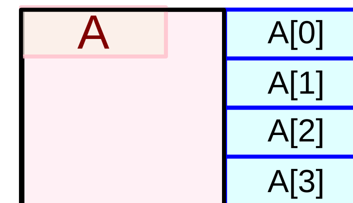


int (*) a as a **0-d** array pointer



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

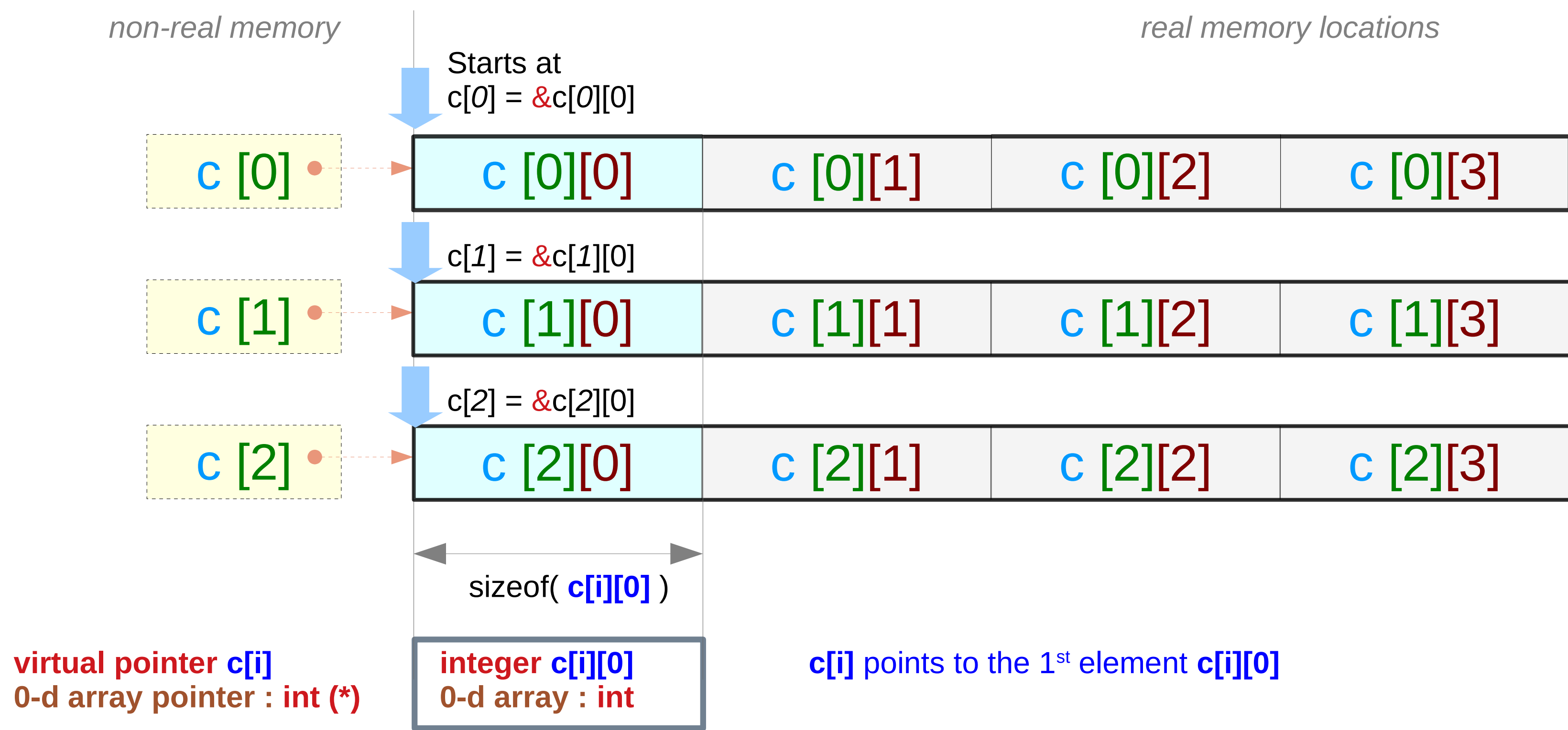
A is not a real array
- sizeof(A) = a pointer size
- A ≠ &A

Pointer **c[i]** and integer **c[i][0]**

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

non-real pointer **c[i]** : value(**c[i]**)= &**c[i][0]**

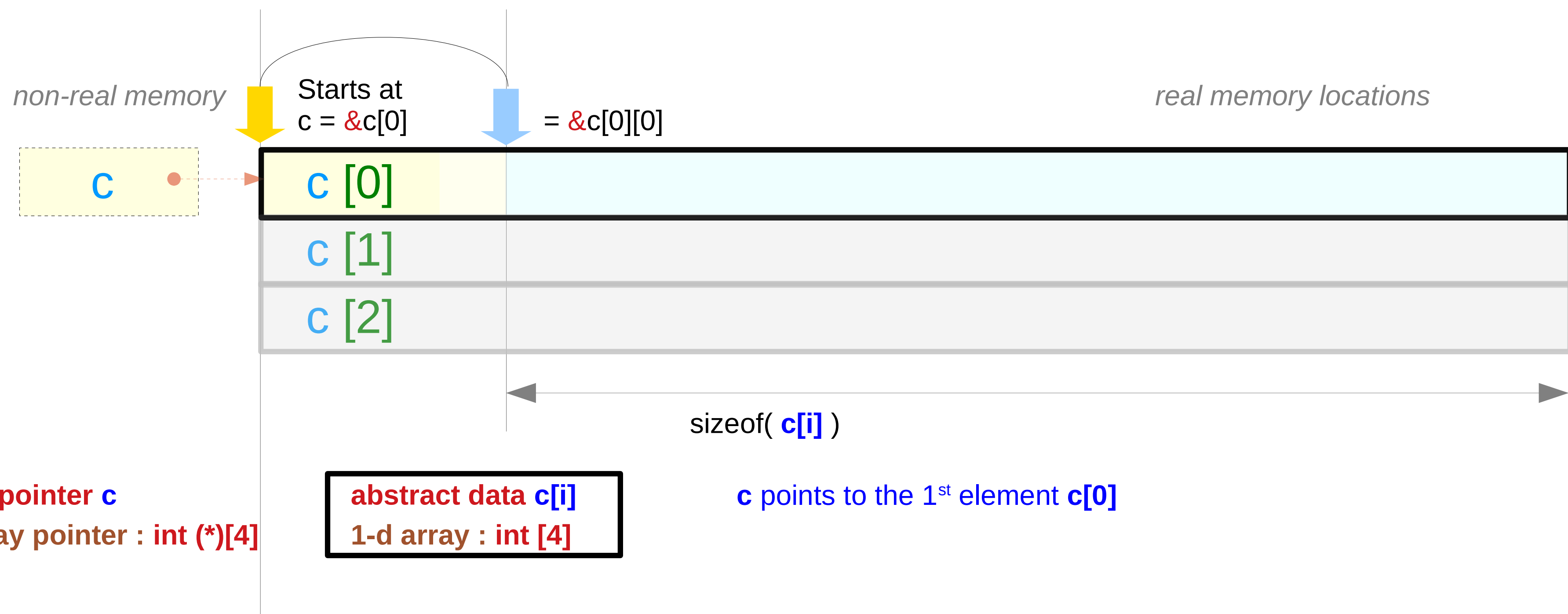
0-d array pointer



Pointer **c** and abstract data **c[i]**

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

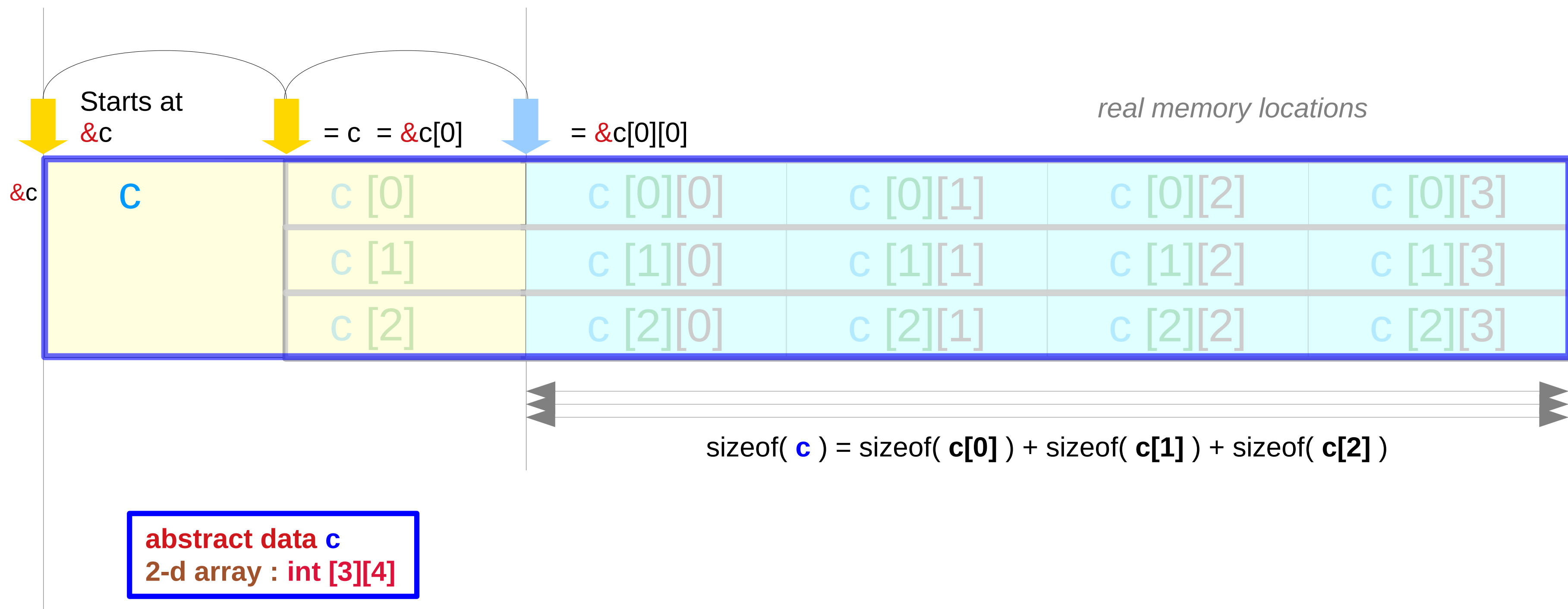
non-real pointer **c** : value(**c**) = &**c[0]** = &**c[0][0]** — **1-d** array pointer
abstract data **c[i]** : sizeof(**c[i]**) = 4 * sizeof(**int**) — **1-d** array



Abstract data c

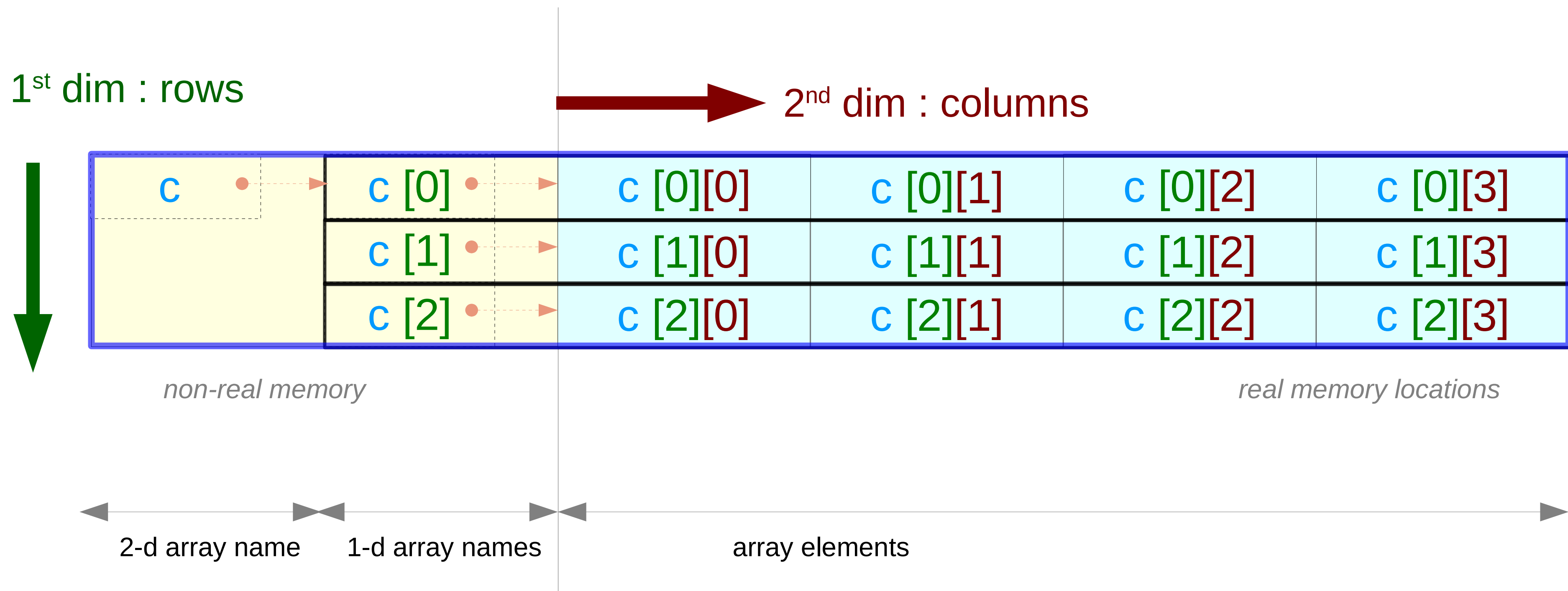
```
int c [3] [4];
```

abstract data **c:** sizeof(**c**) = 3 * sizeof(**c[i]**) **2-d** array



Rows and columns of a **2-d** array **c**

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



The name of a 2-d array

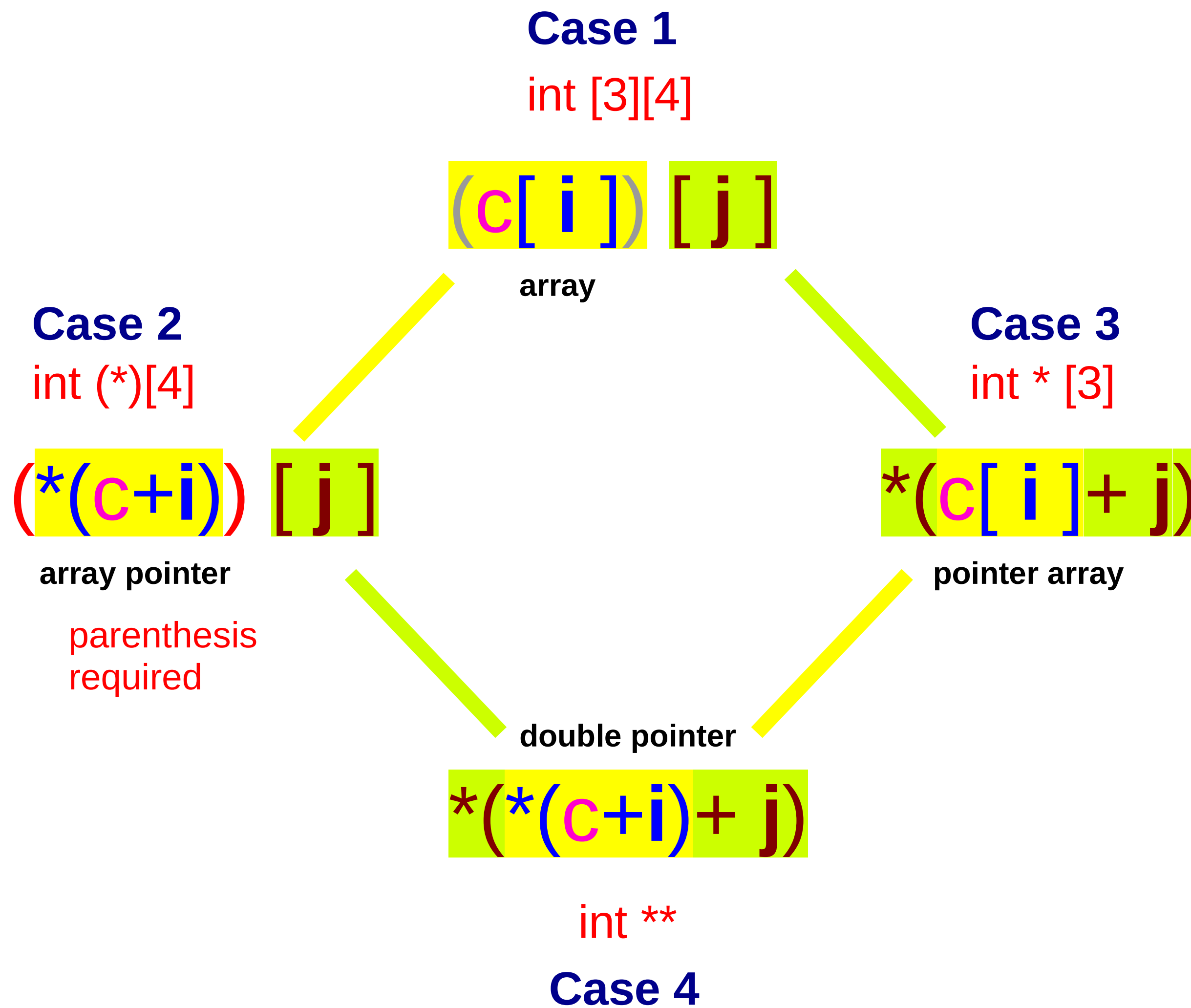
```
int          a [4];

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

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

2. a double pointer

3. a pointer to an array

Array-pointer conversions from a **2-d** array type



2-d array-pointer conversion types

Case	Type	Expression	Description
Case 1	int [3][4]	c[i][j]	**2-d array**
Case 2	int (*) [4]	(*(c+i))[j]	**1-d array pointer**
Case 3	int * [3]	*(c[i]+j)	**1-d array of pointers**
Case 4	int **	*(*(c+i)+j)	**double pointer**

Types of **c**

Case 1	**Case 2**	**Case 3**	**Case 4**
int [3][4]	int (*)[4]	int * [3]	int **
	array pointer	**pointer array**	**double pointer**
	c points to an array of 4 integers	**c** is an array of 3 integer pointers	**c** points to an integer pointer
c[i][j]	≡ **(*(c+i))[j]**	≡ ***(c[i]+j)**	≡ ***(*(c+i)+j)**
&c[i][j]	≡ ***(c+i)+j**	≡ **c[i]+j**	≡ ***(c+i)+j**

the address of **c[i][j]** is ***(c+i)+j** or **c[i]+j**

The row address is ***(c+i)** or **c[i]**

- Pointer conversions in array types
- Simulating array accesses by real pointers
- Dynamic memory allocation

Pointer conversions in **2-d** and **1-d** array types

Case 1 int [3][4]

c[i][j]

2-d array c

relaxing the 1st dimension

Case 2 int (*) [4]

(*(c+i))[j]

1-d array pointer

Case 3 int * [3]

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

1-d array c

relaxing the 1st dimension

Case 4 int **

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

double pointer

relaxing the 1st dimension

Relaxing the 1st dimension of an array

Virtual pointers

Case 1

int [3][4]

(c[i]) [j]

array

Integer Array

Case 2

int (*)[4]

(*(c+i)) [j]

array pointer

parenthesis required

Case 3

int * [3]

*(c[i]+ j)

pointer array

double pointer

((c+i)+ j)

Case 4

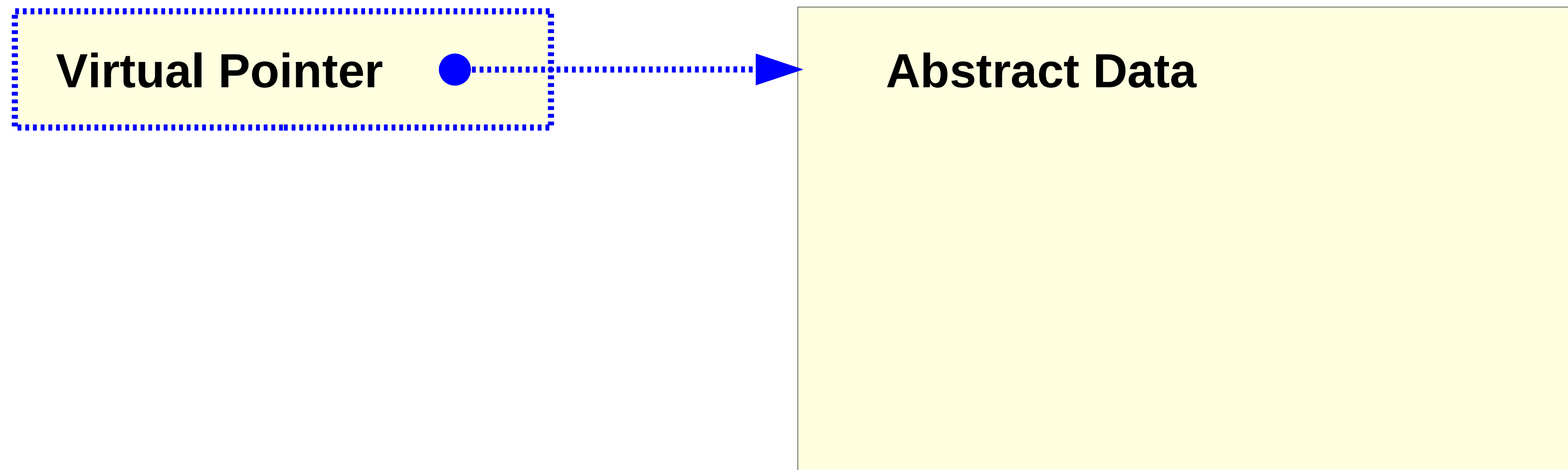
int **

Virtual pointers

Pointer Array

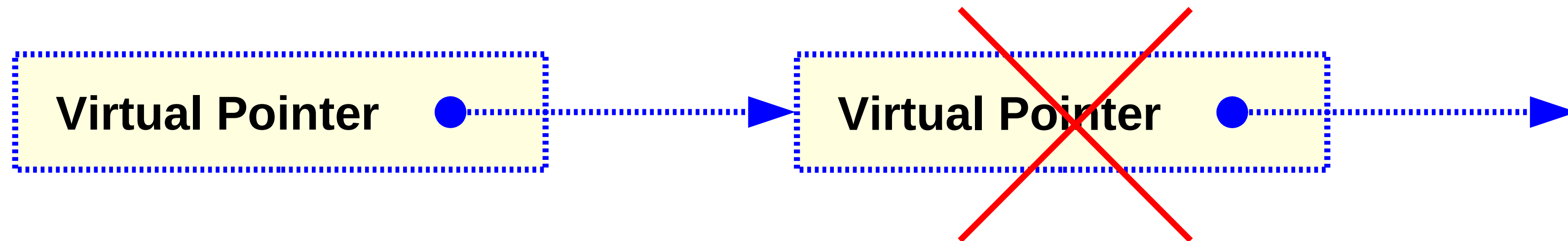
Pointer conversions in **2-d** and **1-d** array types

Only the 1st dimension can be relaxed



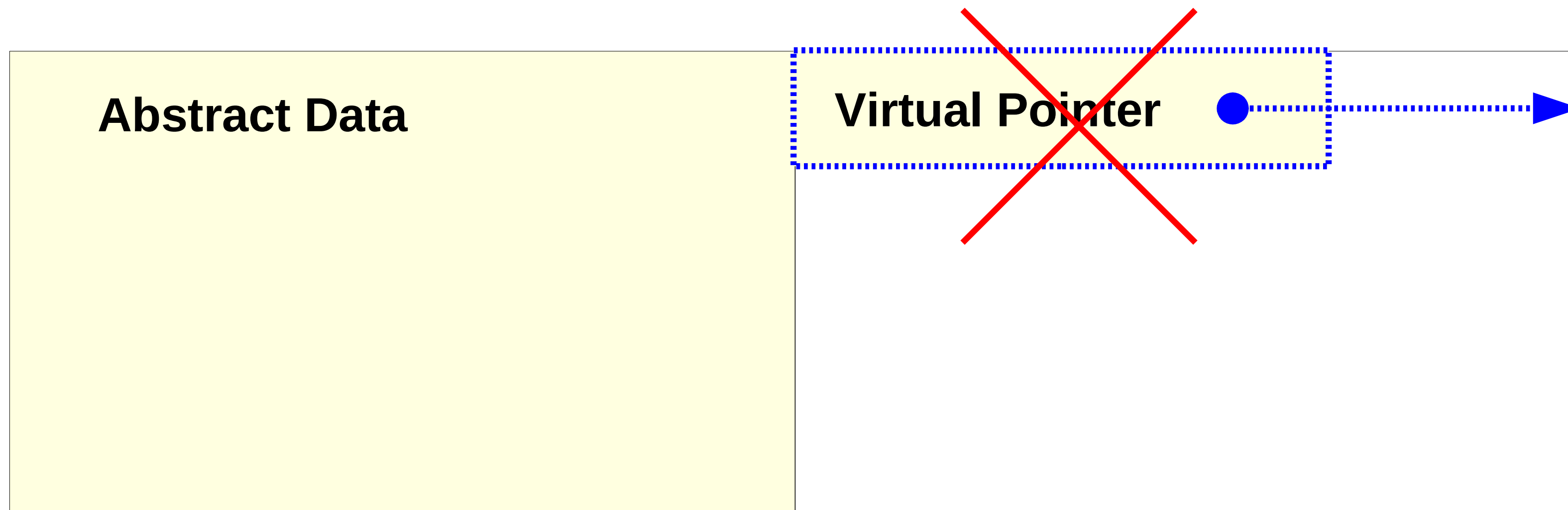
Pointer conversions in **2-d** and **1-d** array types

Only the 1st dimension can be relaxed

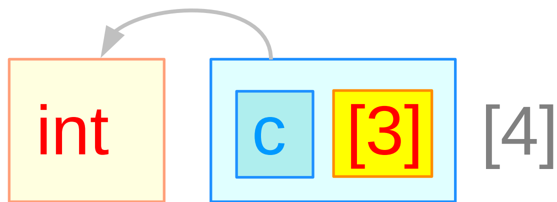


Pointer conversions in **2-d** and **1-d** array types

Only the 1st dimension can be relaxed

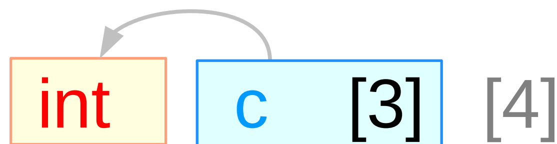


Case 3) 1-d array c, pointer c[i]



c **1-d array**

type : int * [3]

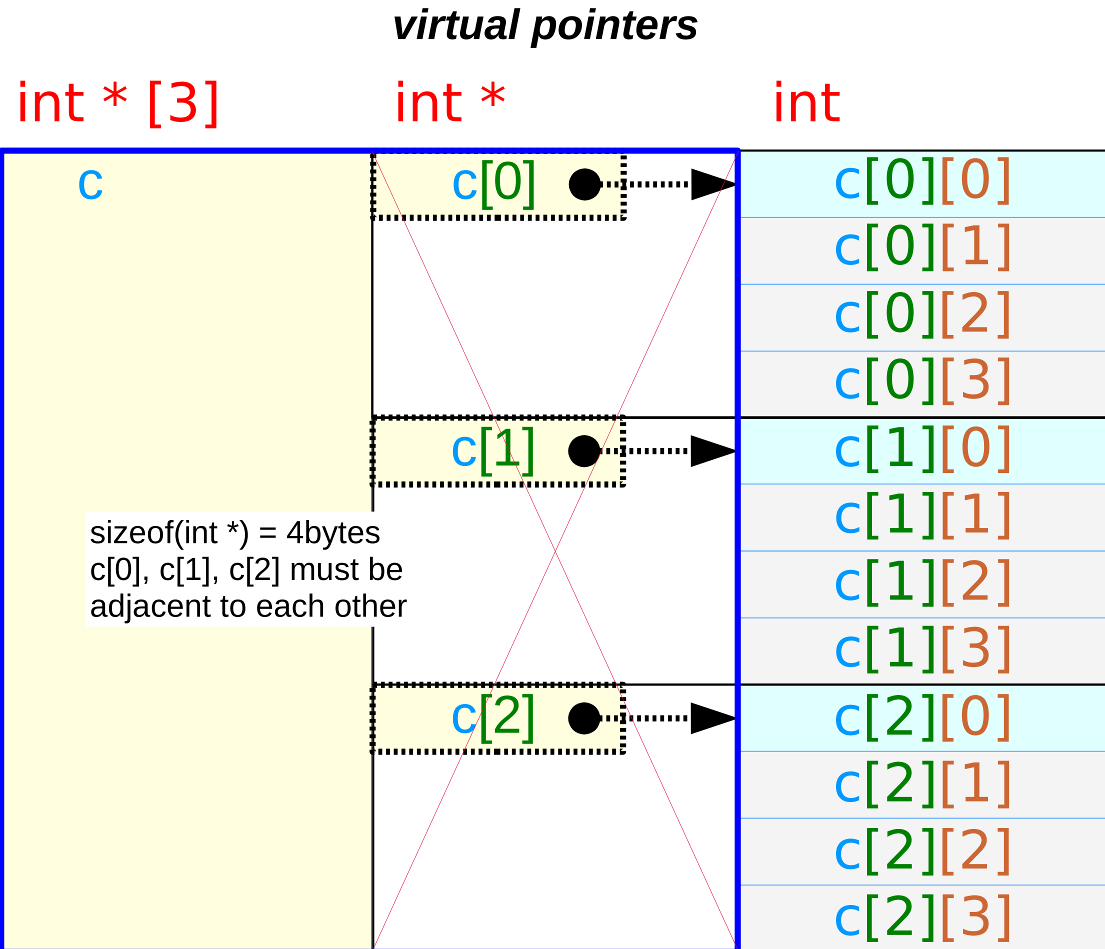


c[i] **pointer**

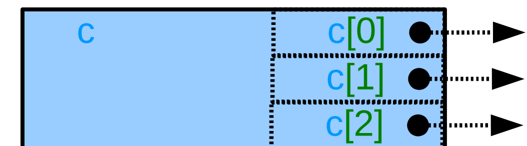
type : int *

Int pointer

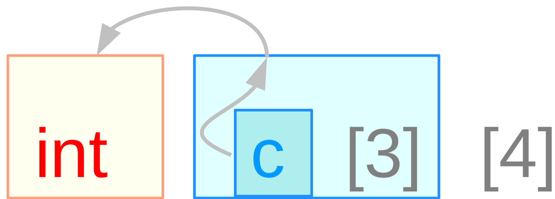
*(c[i]+ j)



c is an array of
3 integer pointers

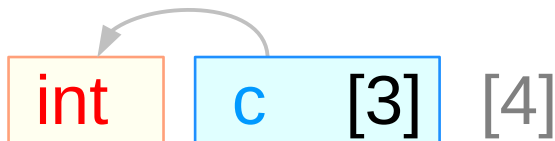


Case 4) double pointer **c**, pointer **c[i]**



c **double pointer**

type : int **

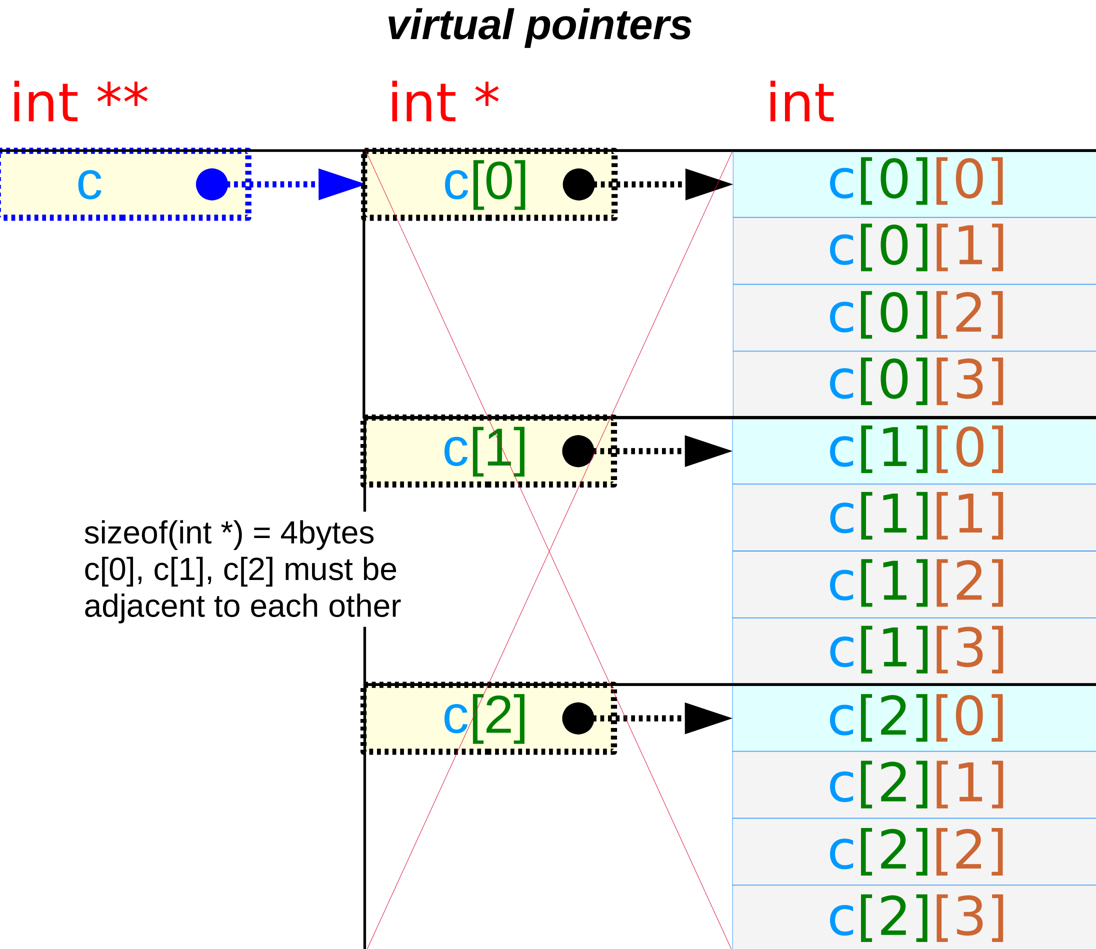


c[i] **pointer**

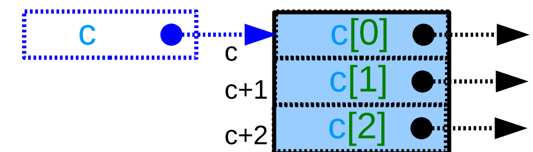
type : int *

Double pointer

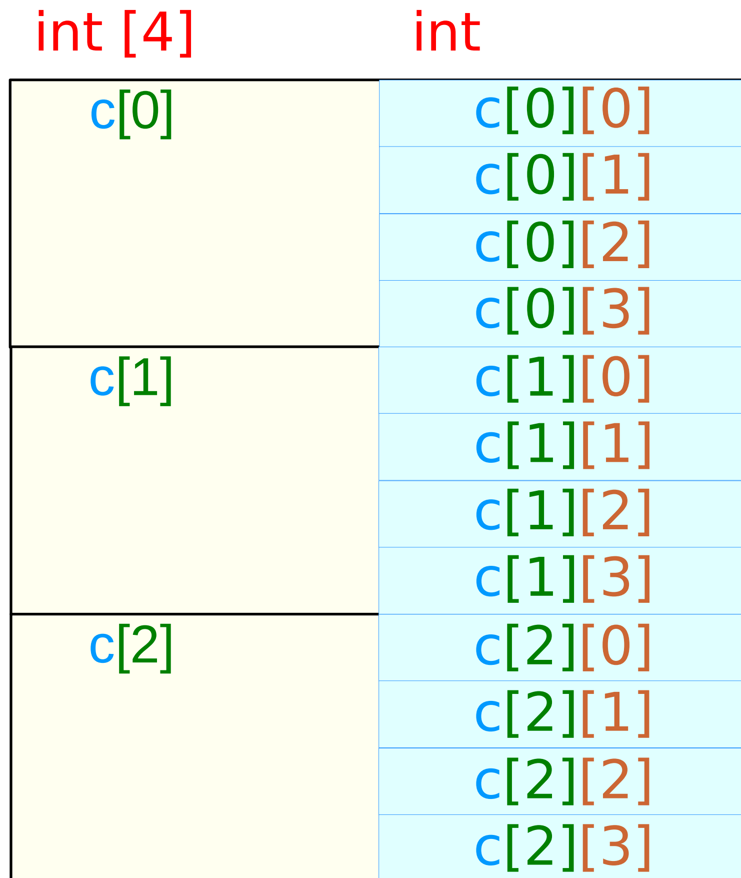
***(*(c+i)+ j)**

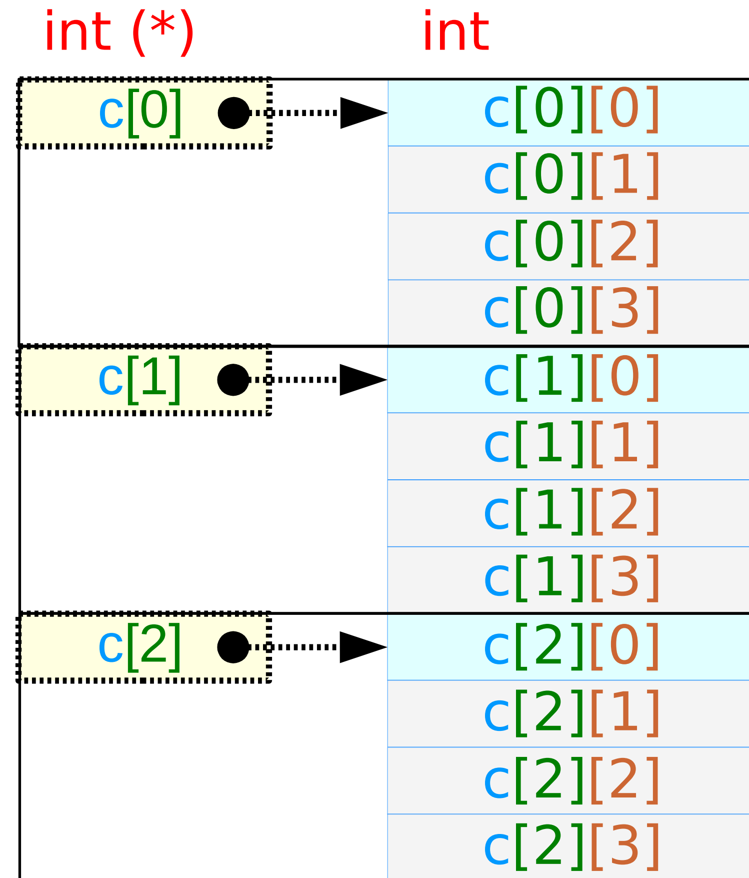


c points to an
integer pointer



Case





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

c points to an integer pointer

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]

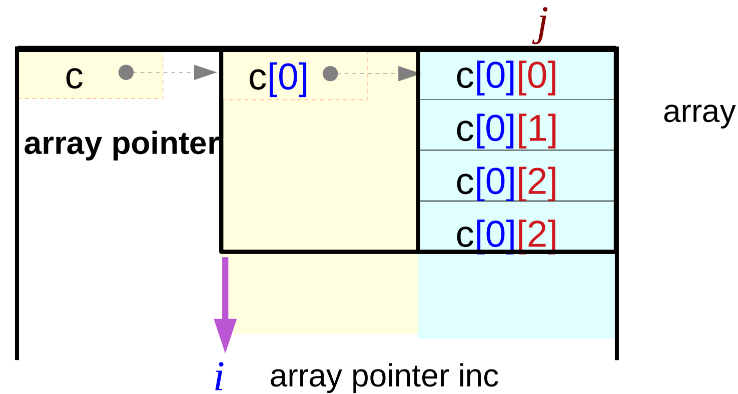
Abstract data and virtual pointer types

Case 2

(*(c+i))[j]

int (*)[4]

c points to a **1-d** array with 4 elements

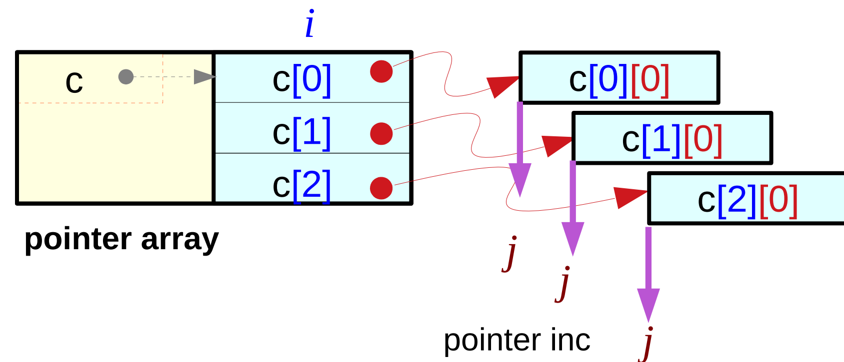


Case 3

*(c[i]+j)

int * [3]

c is a **1-d** array of integer pointers

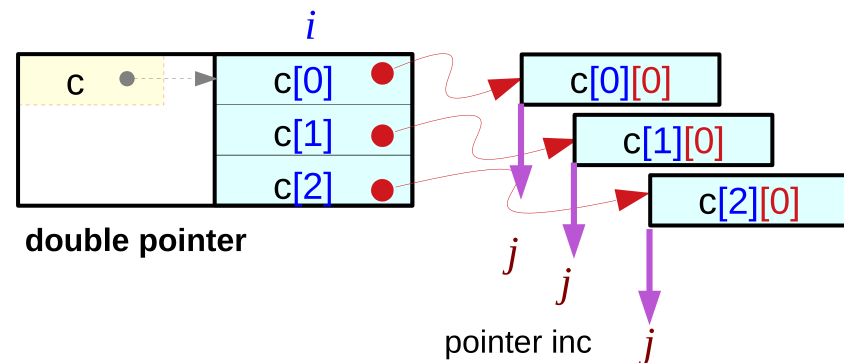


Case 4

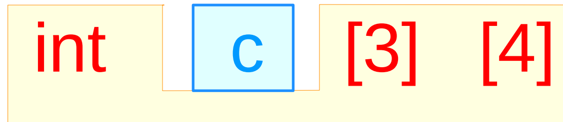
((c+i)+j)

int **

c points to an integer pointer

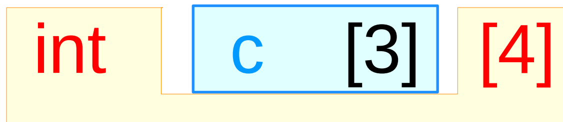


Case 1) 2-d array c, 1-d array c[i]



c **2-d array**

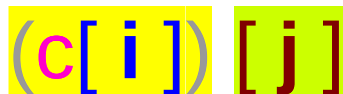
type : int [3][4]



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

type : int [4]

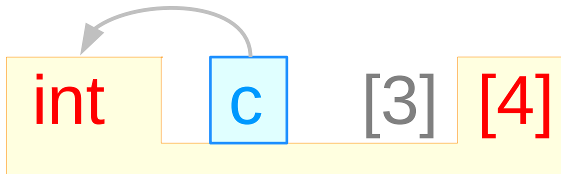
Abstract Data



int [3][4]	int [4]	int
c	c[0]	c[0][0]
		c[0][1]
		c[0][2]
		c[0][3]
	c[1]	c[1][0]
		c[1][1]
		c[1][2]
		c[1][3]
	c[2]	c[2][0]
		c[2][1]
		c[2][2]
		c[2][3]

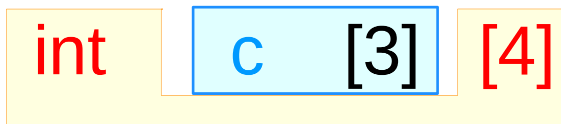
2-d array c

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



c **1-d array pointer**

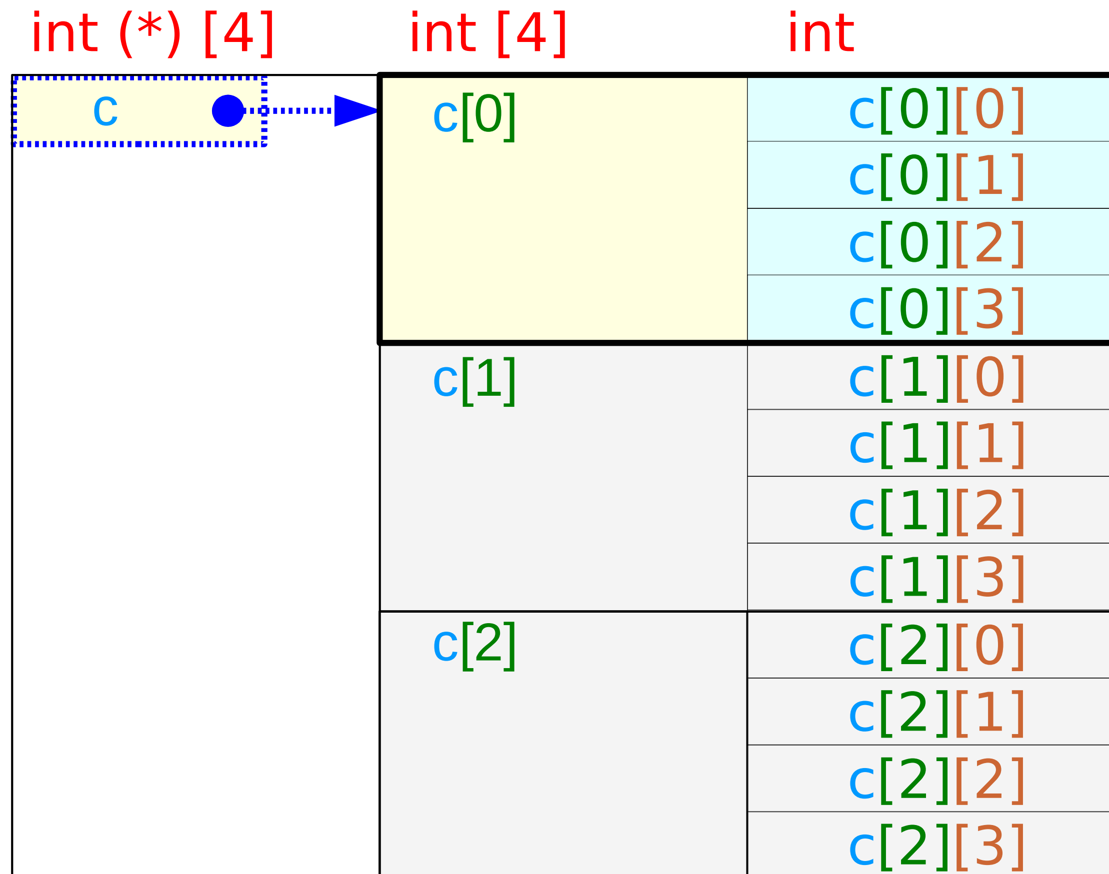
type : int (*) [4]



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

type : int [4]

Abstract Data

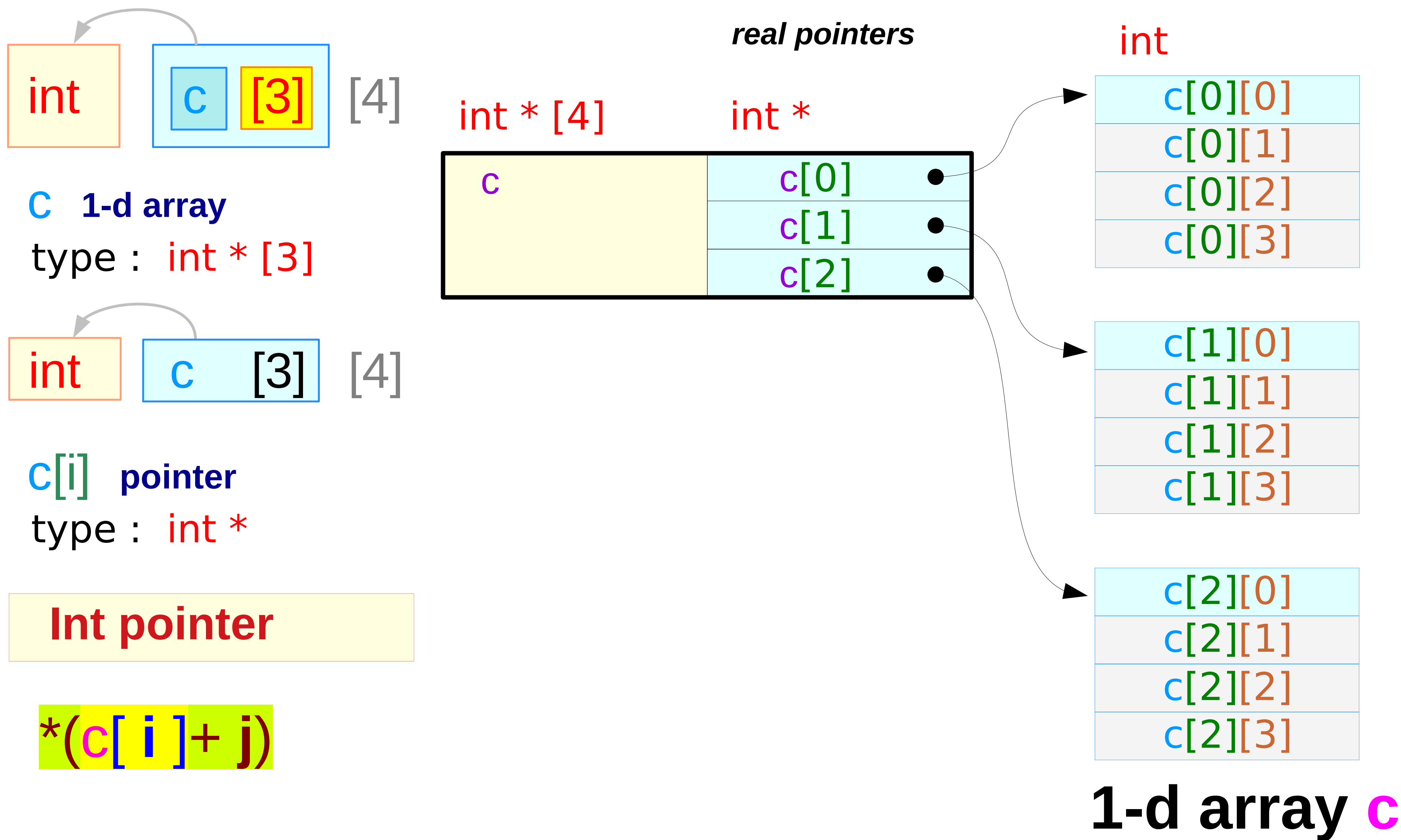


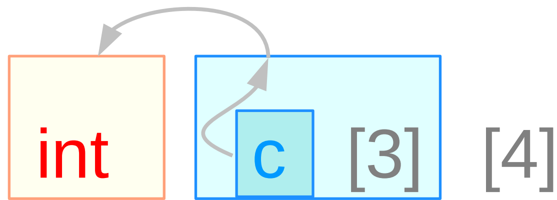
(*(c+i)) [j]

c points to an array
of 4 integers

2-d array c

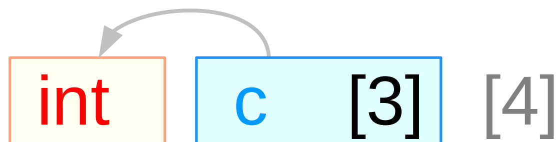
Case 3) 1-d array c, pointer c[i]





[4]

c **double pointer**
type : int **



[4]

c[i] **pointer**
type : int *

Double pointer

real pointers

int * [4]

int *

c

c[0]
c[1]
c[2]

int

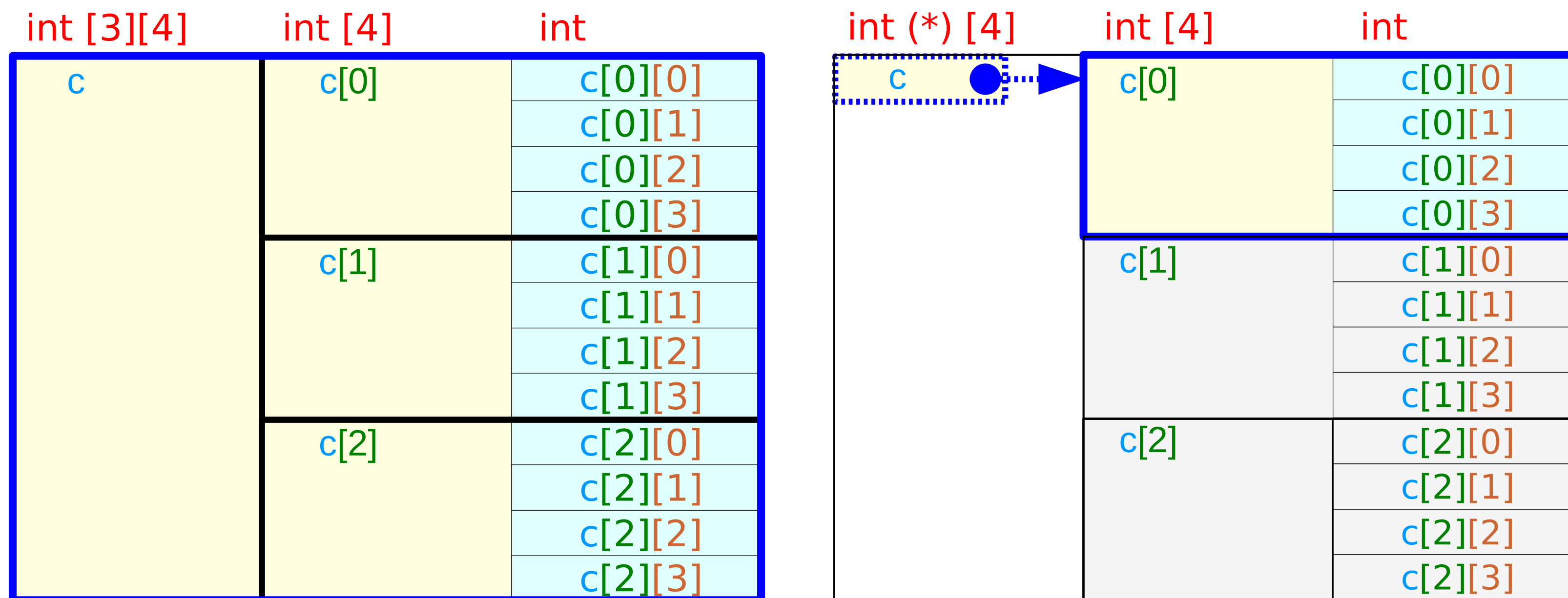
c[0][0]
c[0][1]
c[0][2]
c[0][3]

c[1][0]
c[1][1]
c[1][2]
c[1][3]

c[2][0]
c[2][1]
c[2][2]
c[2][3]

1-d array c

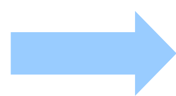
Nested Structure – abstract data, virtual pointer



Case 1

Case 2

(c[i]) [j]



(*(c+i)) [j]

nested structure

2-d array c

Nested structure – abstract data, virtual pointer

int * [4] int * int

c | c[0] | c[1] | c[2]

c[0][0] c[0][1] c[0][2] c[0][3]

c[1][0] c[1][1] c[1][2] c[1][3]

c[2][0] c[2][1] c[2][2] c[2][3]

int * * int * int

c | c[0] | c[1] | c[2]

c[0][0] c[0][1] c[0][2] c[0][3]

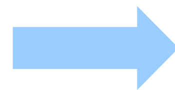
c[1][0] c[1][1] c[1][2] c[1][3]

c[2][0] c[2][1] c[2][2] c[2][3]

Case 3

Case 4

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



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

nested structure

1-d array c

- Pointer conversions in array types
- Simulating array accesses by real pointers
- Dynamic memory allocation

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

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

Case 1

int [3][4]

Incrementing the 1st dimension pointer

Case 2

array pointer

int (*)[4]

Incrementing the 2nd dimension pointer

Case 4

double pointer

int * *

Case 1

int [3][4]

Incrementing the 2nd dimension pointer

Case 3

pointer array

int * [3]

Incrementing the 1st dimension pointer

Case 4

double pointer

int * *

2-d array access via a double indirection

Case 1	Case 2	Case 4
int [3][4]	int (*)[4]	int **
	pointer array	**double pointer**
(c [i])[j]	(*(c+i))[j]	*(*(c+i)+j)

i — expand the 1st dimension

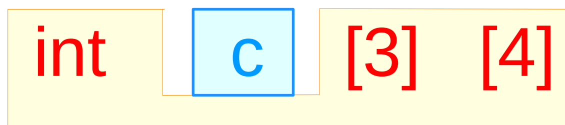
(c [i]) = (*(c+i))

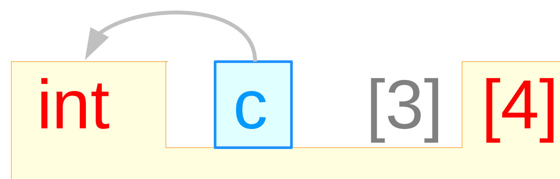
contiguous memory locations are assumed

j — expand the 2nd dimension

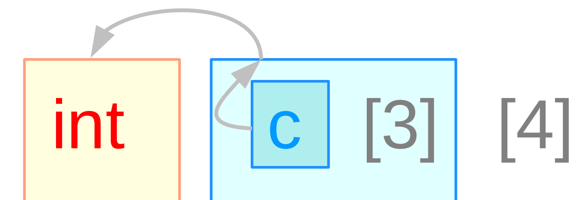
(_)[j] = *((_)+j)

contiguous memory locations are assumed





c points to an array of 4 integers



c points to an integer pointer

2-d array access via a double indirection

Case 1

int [3][4]

Case 3

int * [3]

array pointer

Case 4

int **

double pointer

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

expand the 2nd dimension

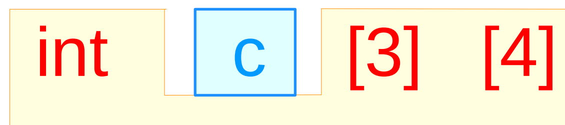
(_)[j] = *((_)+j)

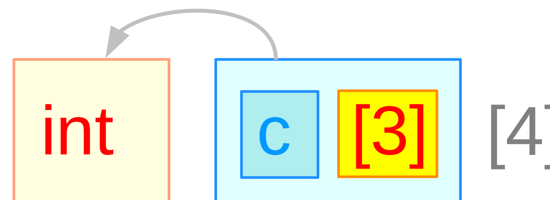
contiguous memory locations are assumed

expand the 1st dimension

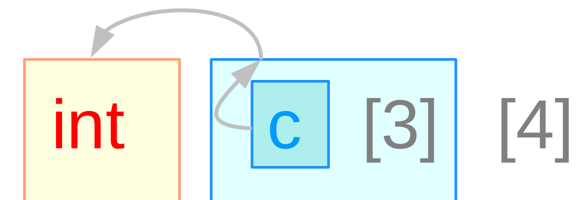
(c [i]) = (*(c+i))

contiguous memory locations are assumed





c is an array of
3 integer pointers



c points to an
integer pointer

Cases 1, 2, 4

int c [3] [4];

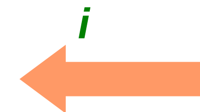
int (*p) [4];

Case 1
int [3][4]

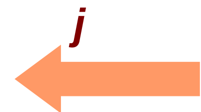
Case 2
int (*) [4]

Case 4
int **

(c [i])[j]



(*(c+i))[j]



((c+i)+j)

p = c

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

equivalence

(p [i])[j]

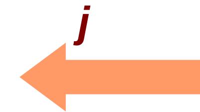
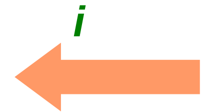
(*(p+i))[j]

((p+i)+j)

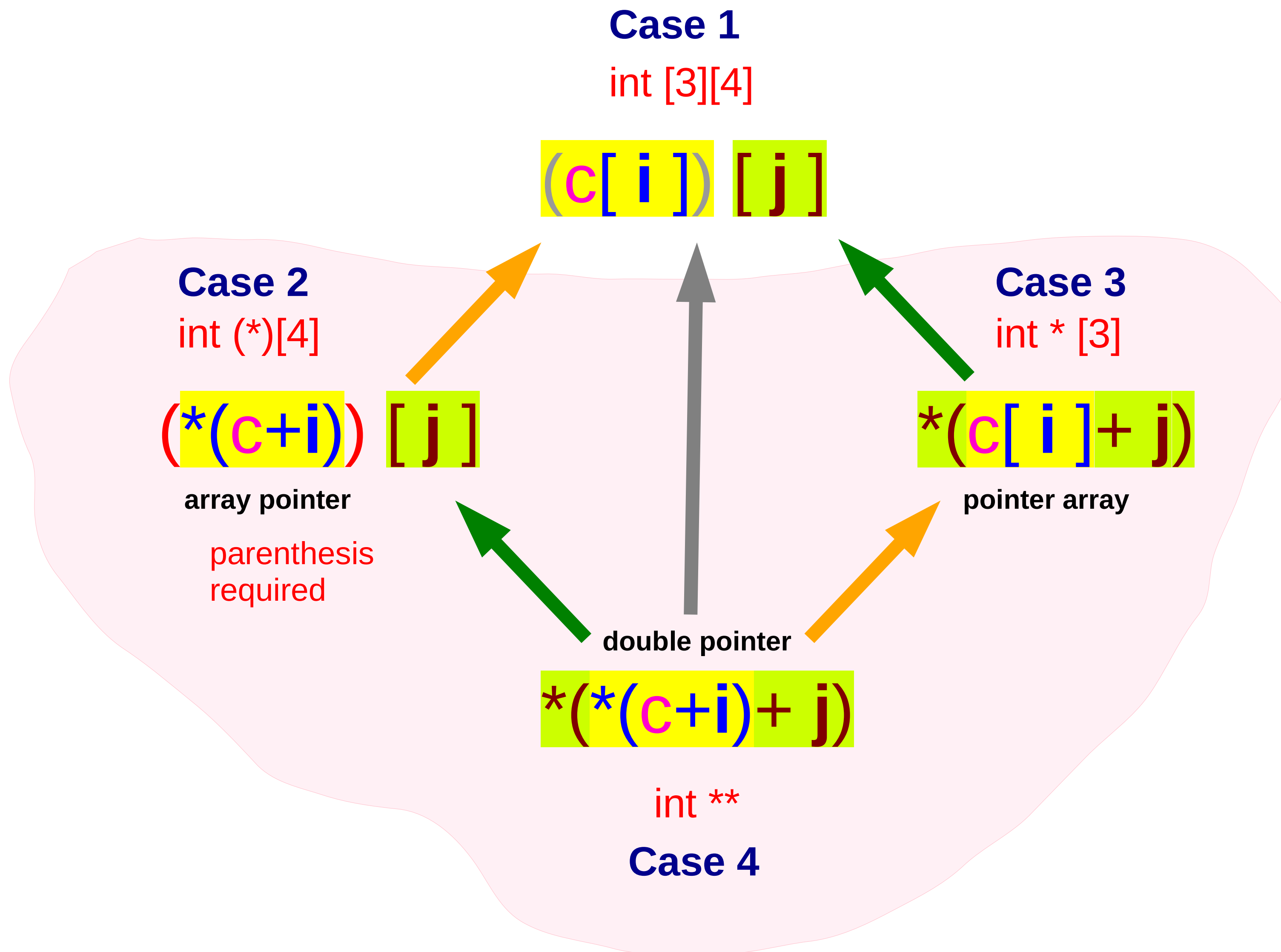
Cases 1, 3, 4

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

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

Case 1		Case 3		Case 4
int [3][4]		int * [3]		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)

Simulating **2-d** array accesses by real pointers



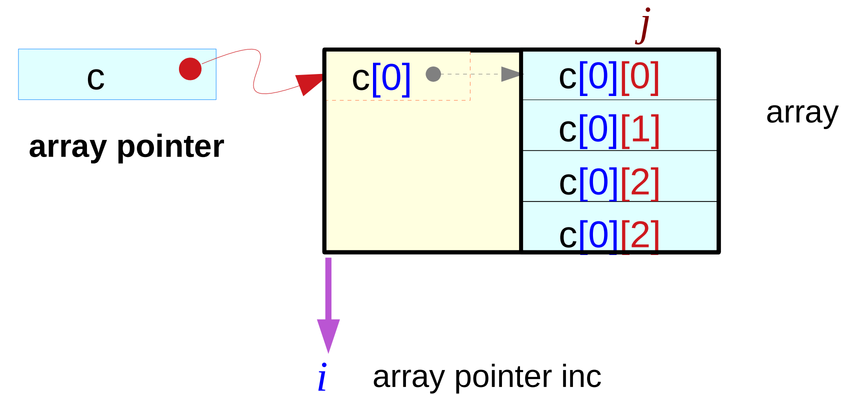
Incrementing pointers

Case 2

int (*)[4]

(*(c+i))[j]

c points to a **1-d** array with 4 elements

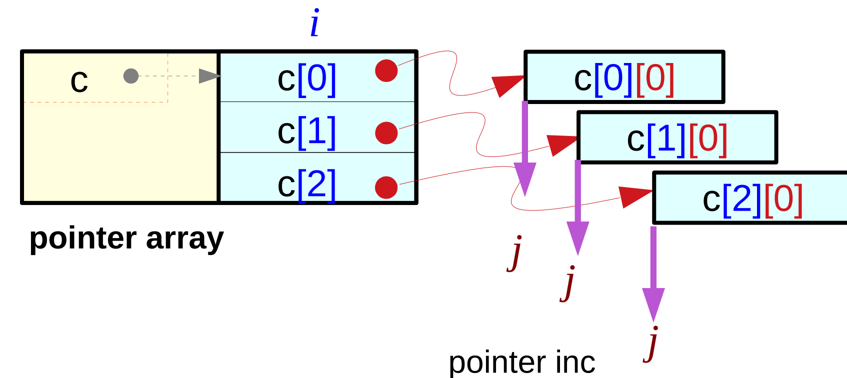


Case 3

int * [3]

*(c[i]+j)

c is a **1-d** array of integer pointers

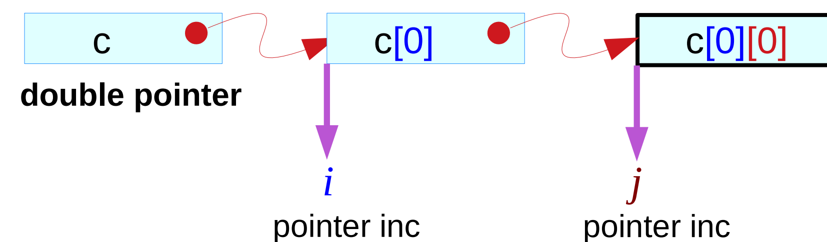


Case 4

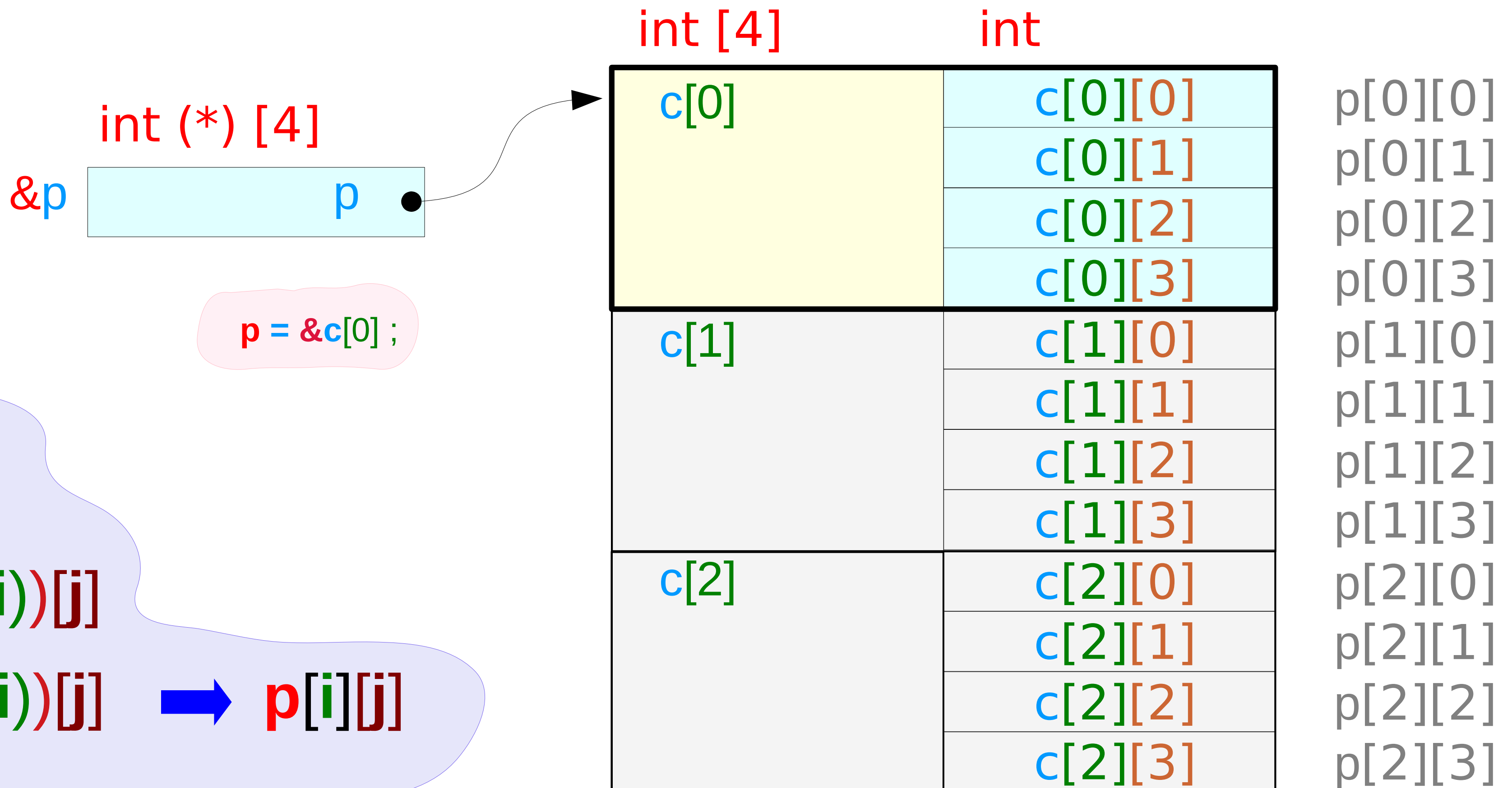
int **

((c+i)+j)

c points to an integer pointer



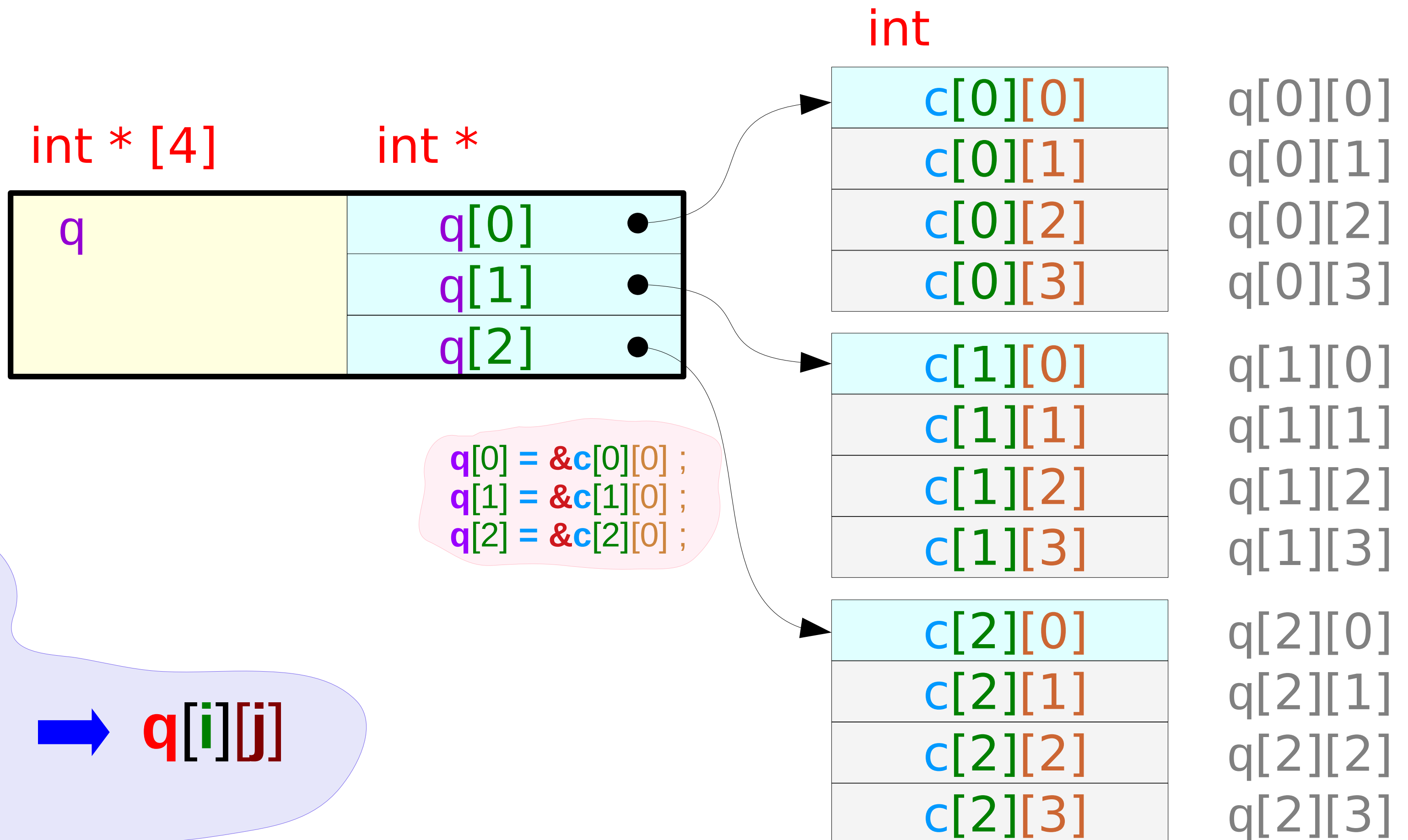
2-d array access using an array pointer p



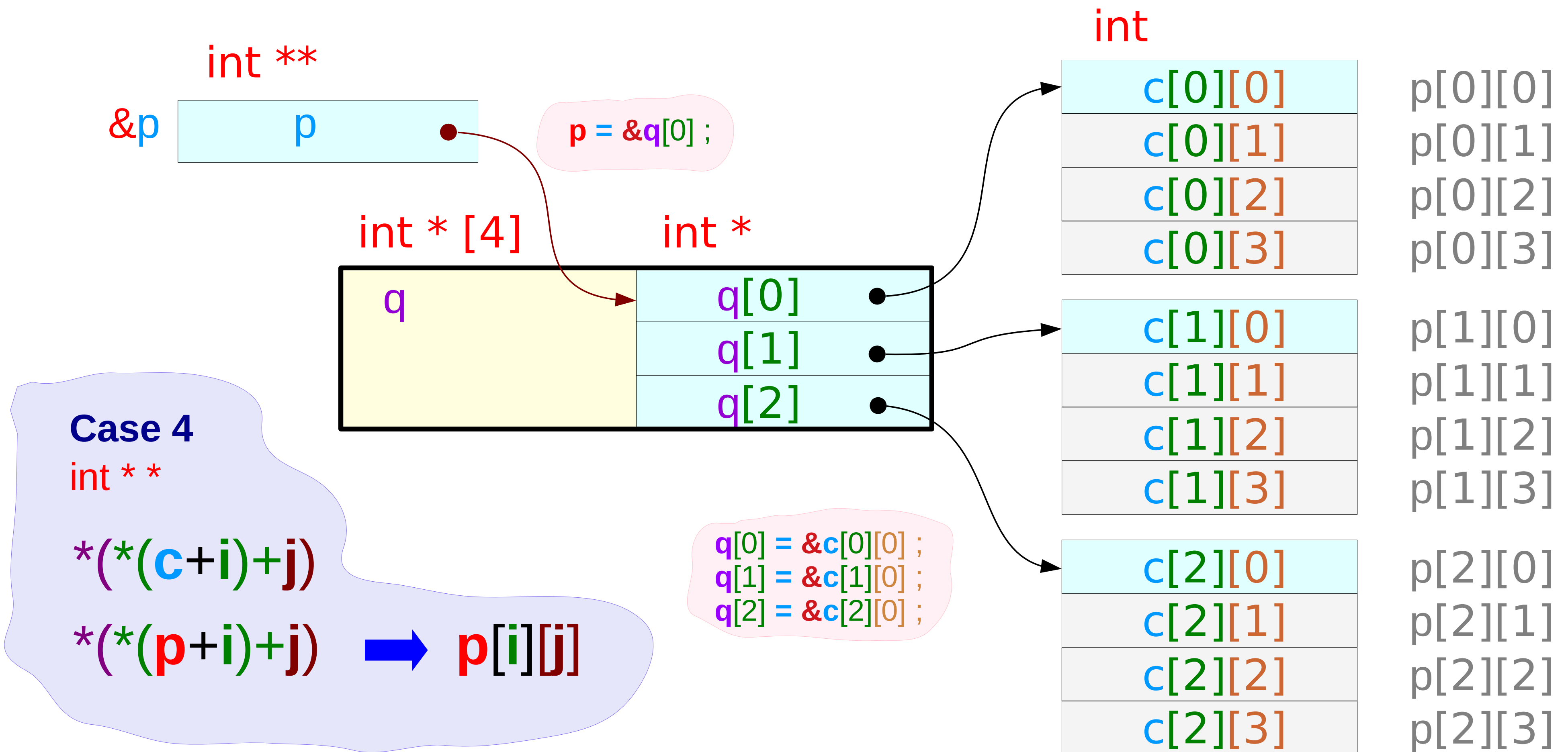
2-d array access using a pointer array q

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

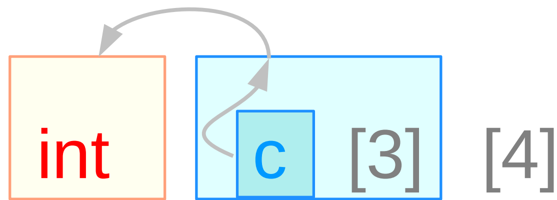
```
int *q[3];
```



2-d array access using double pointers q



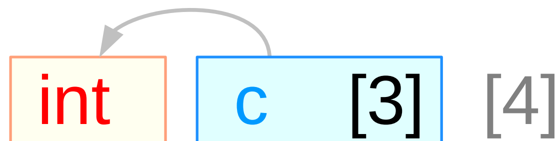
Case 4) double pointer **c**, pointer **c[i]**



[4]

c **double pointer**

type : int **



[4]

c[i] **pointer**

type : int *

Double pointer

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

real pointers

int * *

int *

int

c

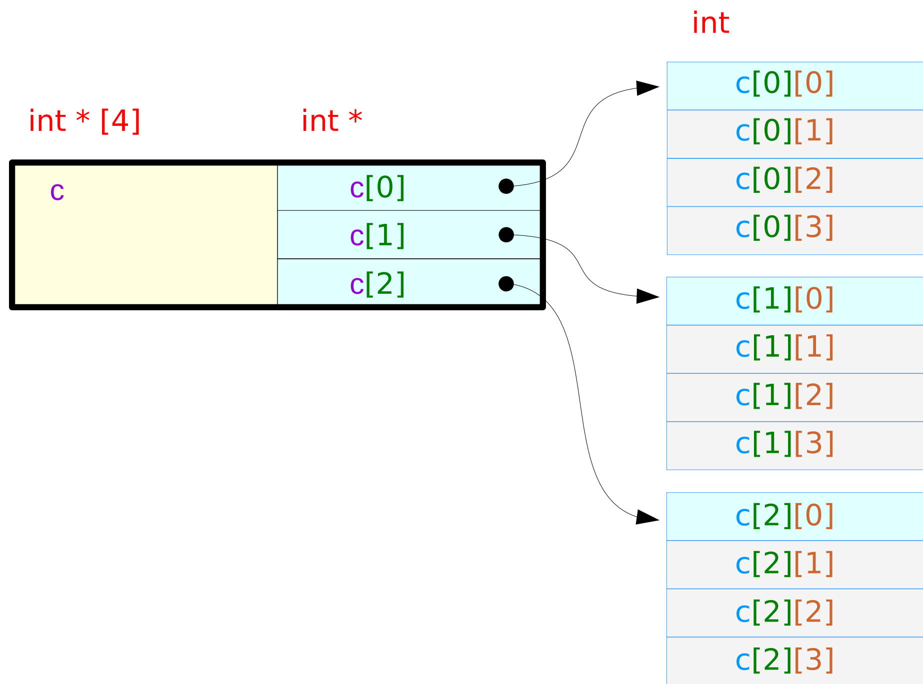
c[0]
c[1]
c[2]

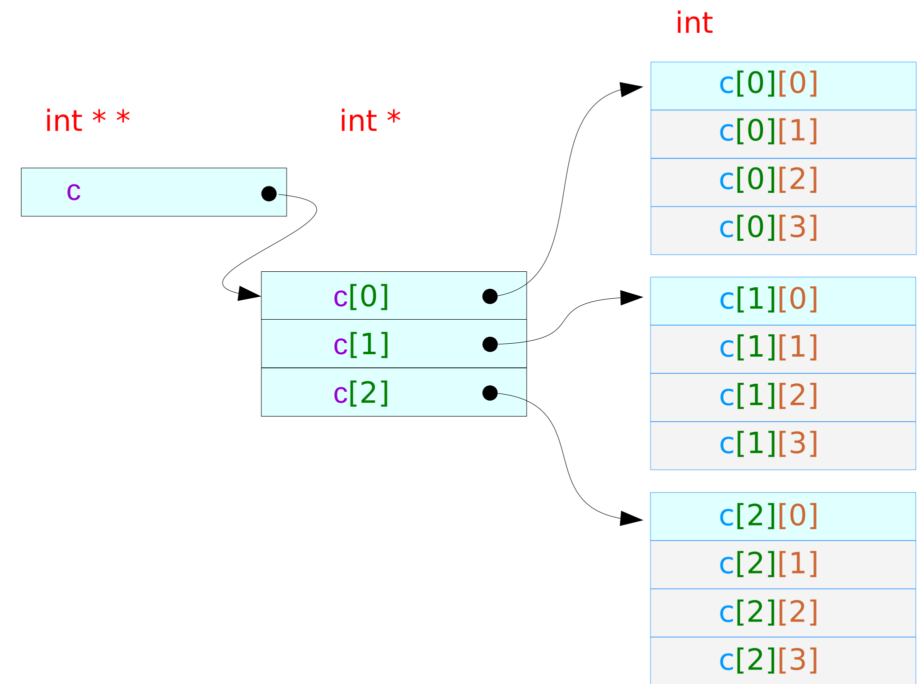
c[0][0]
c[0][1]
c[0][2]
c[0][3]

c[1][0]
c[1][1]
c[1][2]
c[1][3]

c[2][0]
c[2][1]
c[2][2]
c[2][3]

Nested structure - Pointers

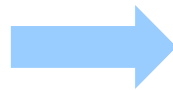




Case 3

Case 4

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



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

nested structure

- Pointer conversions in array types
- Simulating array accesses by real pointers
- Dynamic memory allocation

Dynamic Memory Allocation of 2-d Arrays

method 1

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

Case 4

int * *

method 2

```
int   **   c ;
int   *    p ;
c = (int **) malloc( 3 * sizeof(int *) ) ;
p = (int *)  malloc( 4 * 4 * sizeof(int) ) ;
for (i=0; i<M; i++)    c[i] = p + i*N;
```

Case 4

int * *

method 3

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

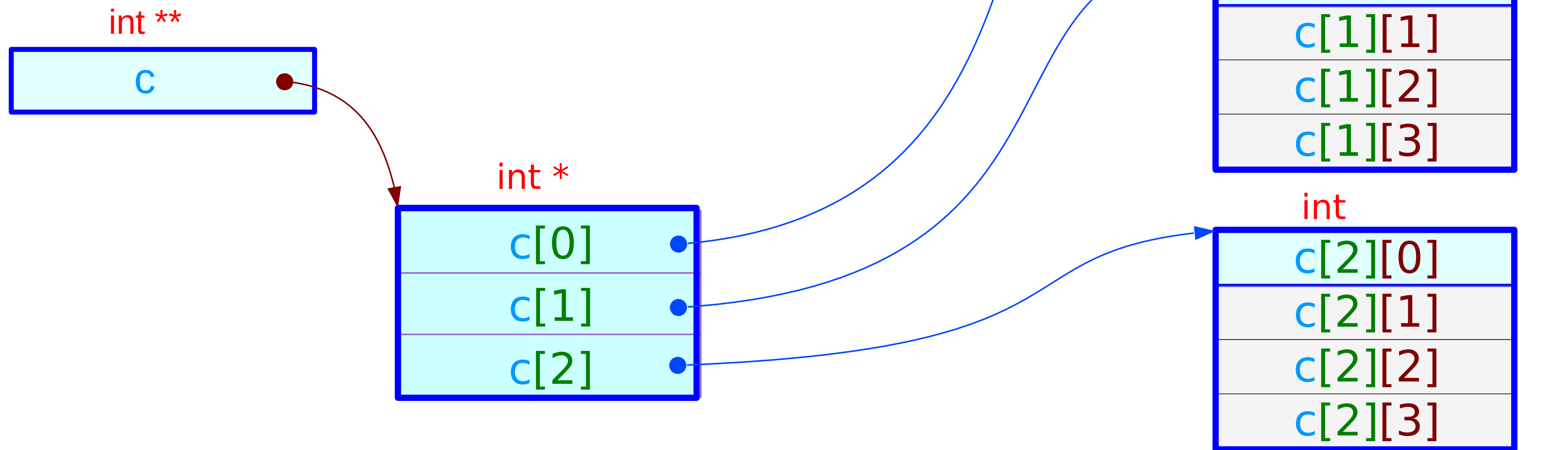
Case 2

int (*)[4]

2-d array dynamic allocation : method 1

method 1

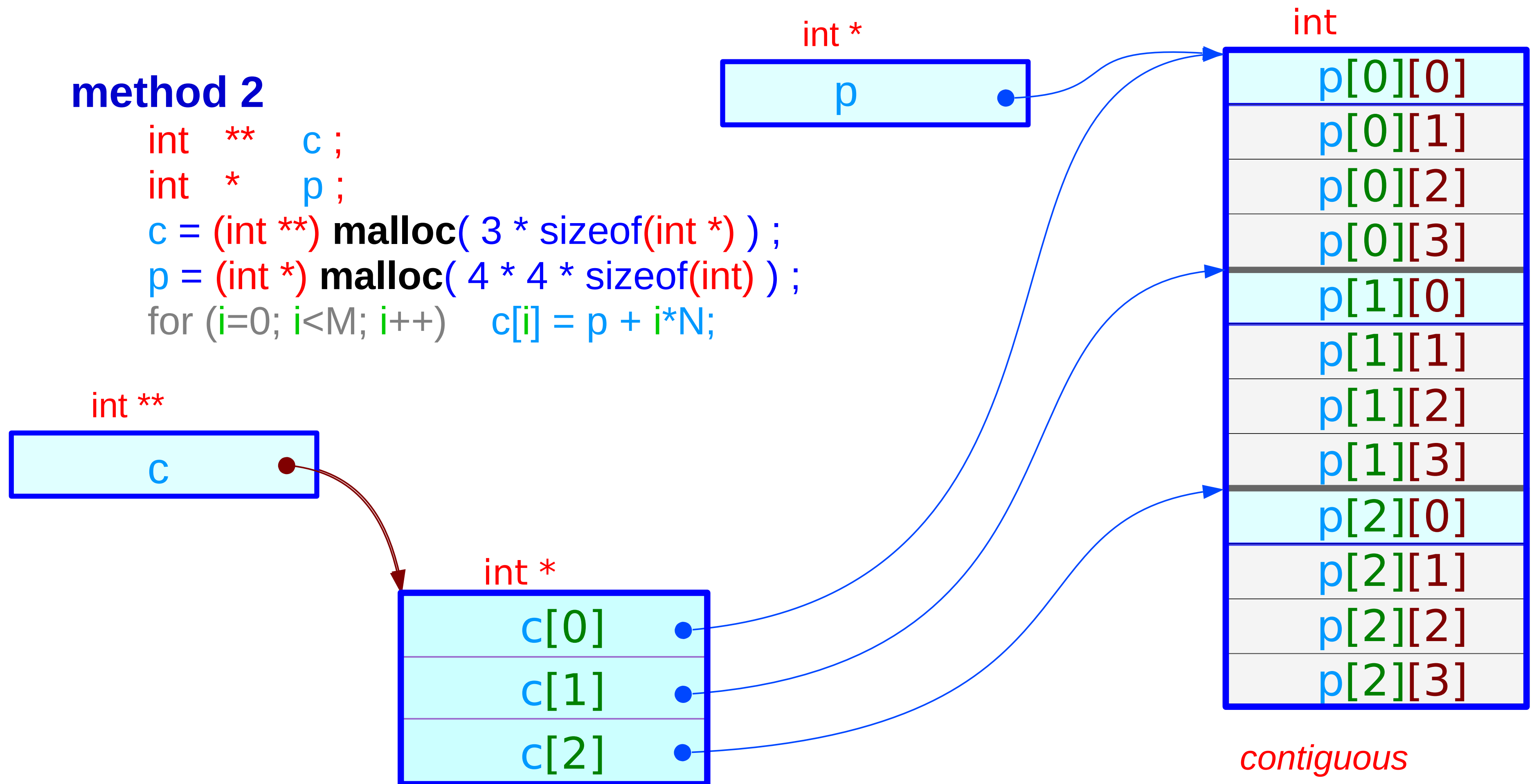
```
int ** c ;
c    = (int **) malloc(3 * sizeof (int *)) ;
c[0] = (int *) malloc(4 * sizeof (int)) ;
c[1] = (int *) malloc(4 * sizeof (int)) ;
c[2] = (int *) malloc(4 * sizeof (int)) ;
```



c: an array of integer pointers

may not be contiguous because of memory alignments

2-d array dynamic allocation : method 2

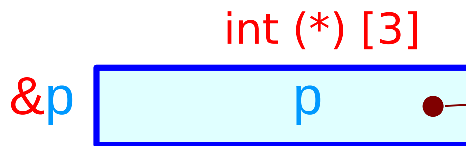


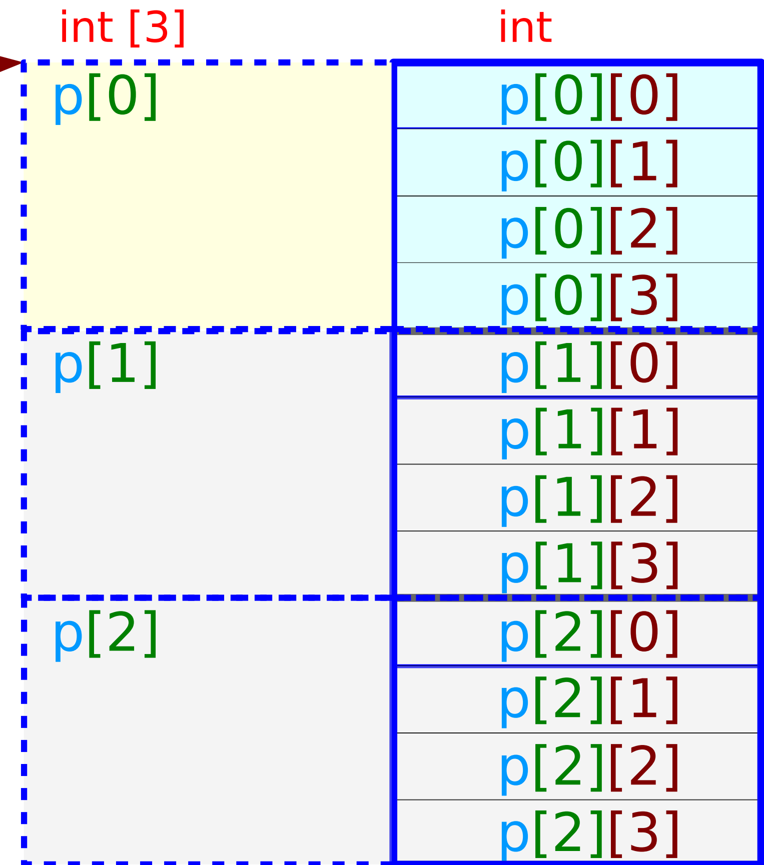
array of pointers
allocated physically in memory

2-d array dynamic allocation : method 3

method 3

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



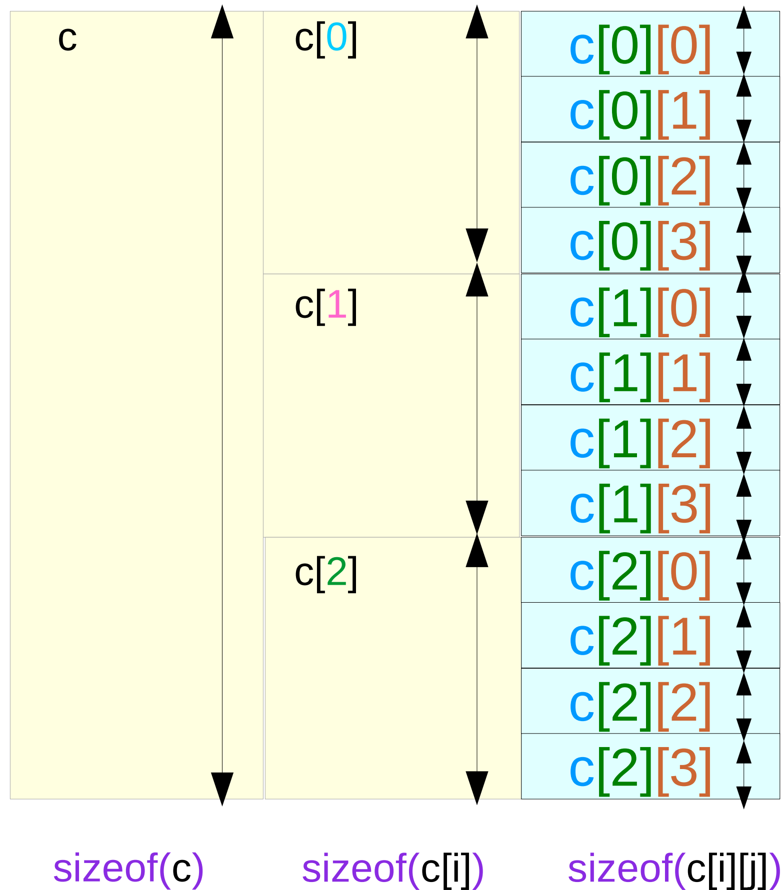


utilize pointer
addition property

Pointer to Arrays :
No physical allocation

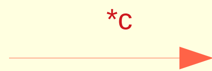
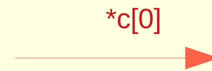
Size and address views of a **2-d** array

size view



address view

Sub-array sizes of a **2-d** array

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

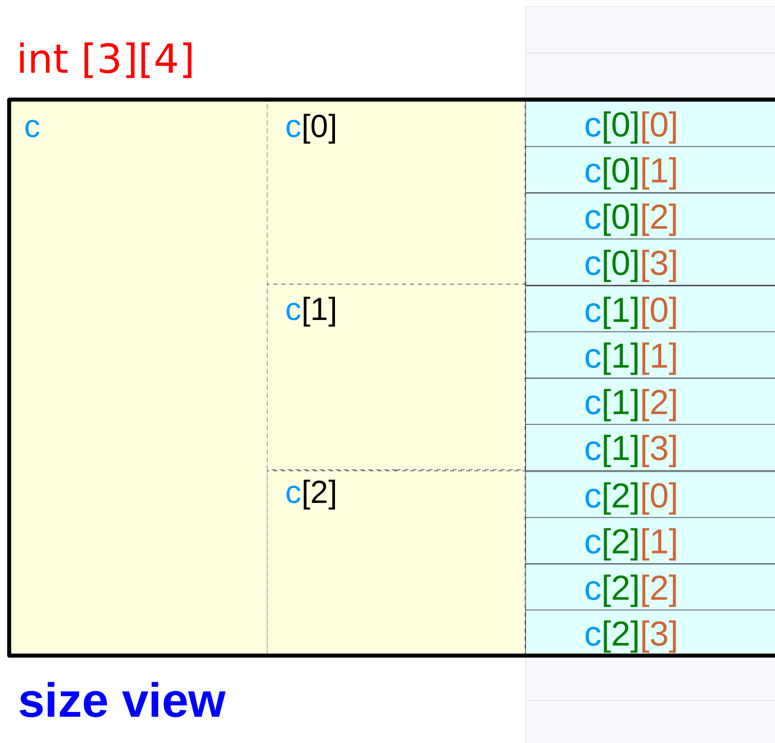
Size and address views of a **2-d** array **c**

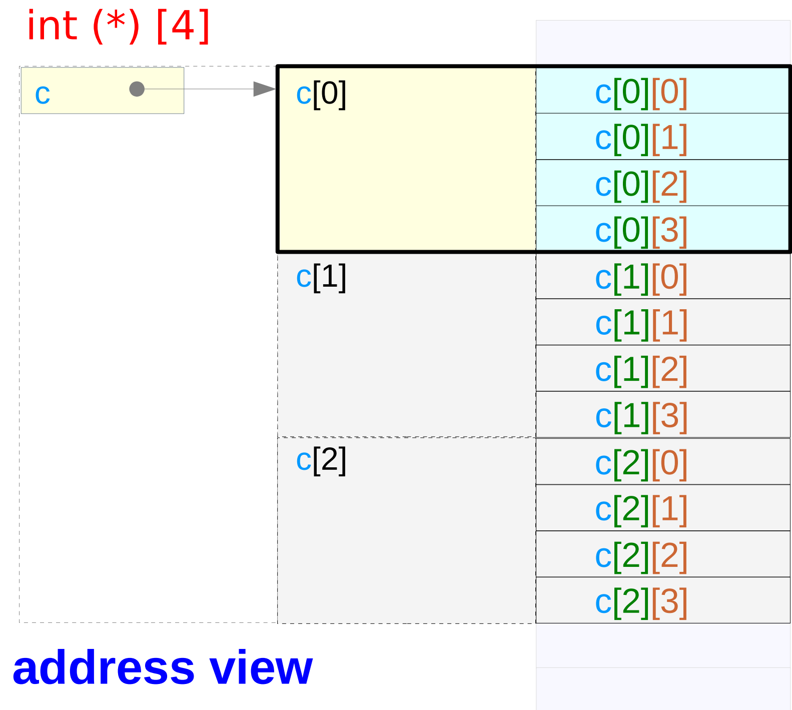
```
int c [3] [4];
```

int c[3][4];
sizeof(c)
sizeof(int [3][4])

*c →

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





Size and address views of a **1-d** array **c[i]**

int c [3] [4];

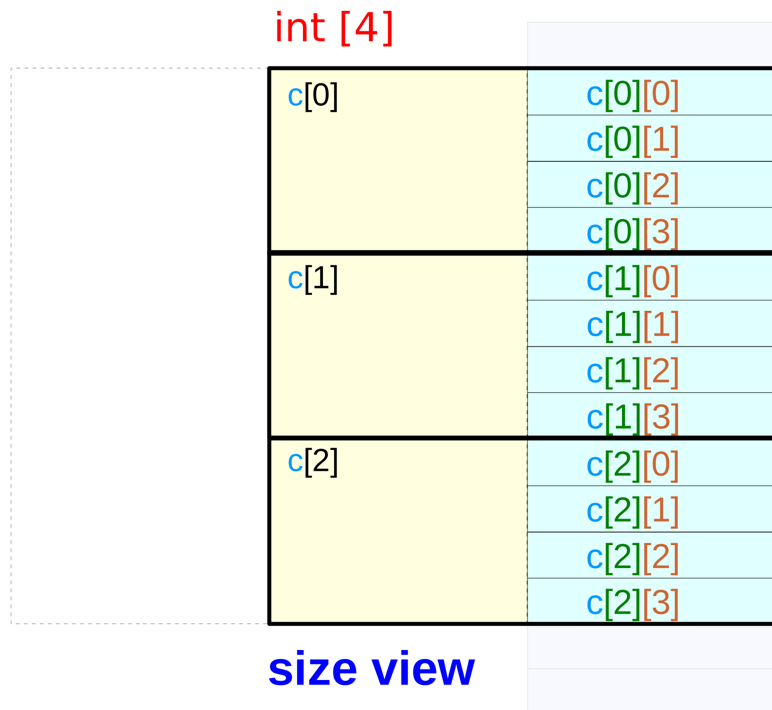
int **c[3]**[4];

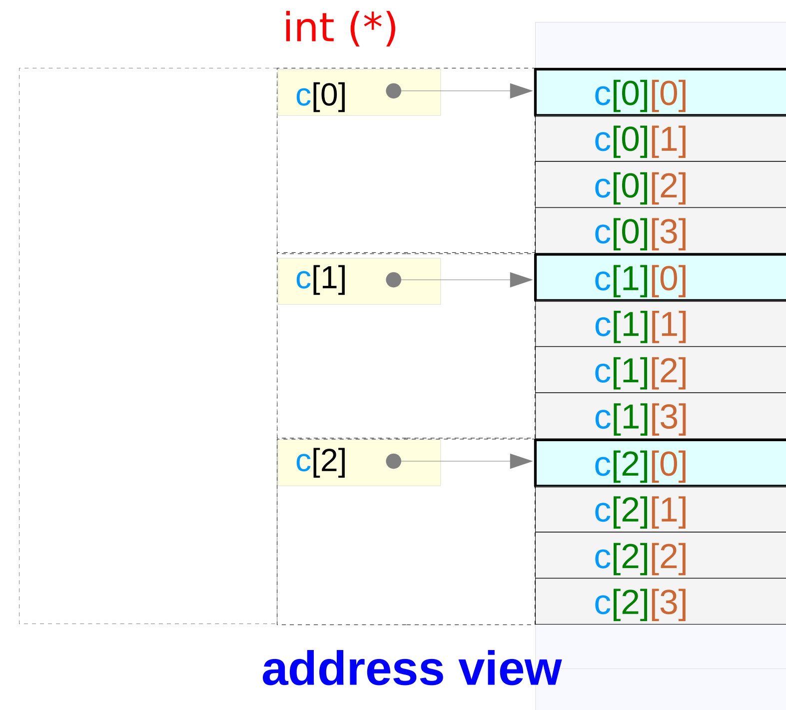
sizeof(c[0]) → *c[0] → sizeof(c[0][0]) * 4

sizeof(int [4])

int **c[3][4]**;

sizeof(int) * 4





2-d array c and 1-d array q

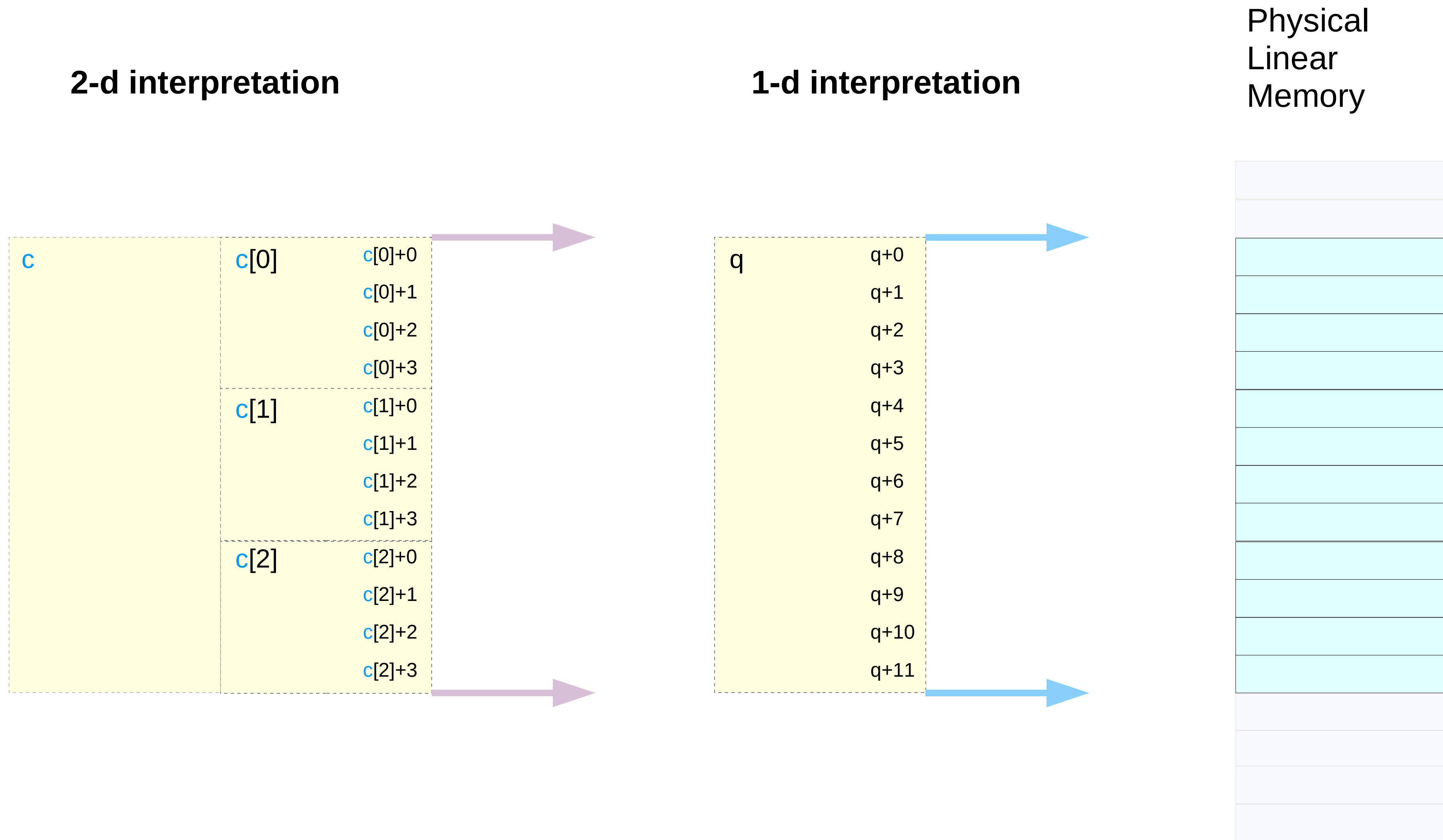
```
int c [3] [4];
```

```
int q [3*4];
```

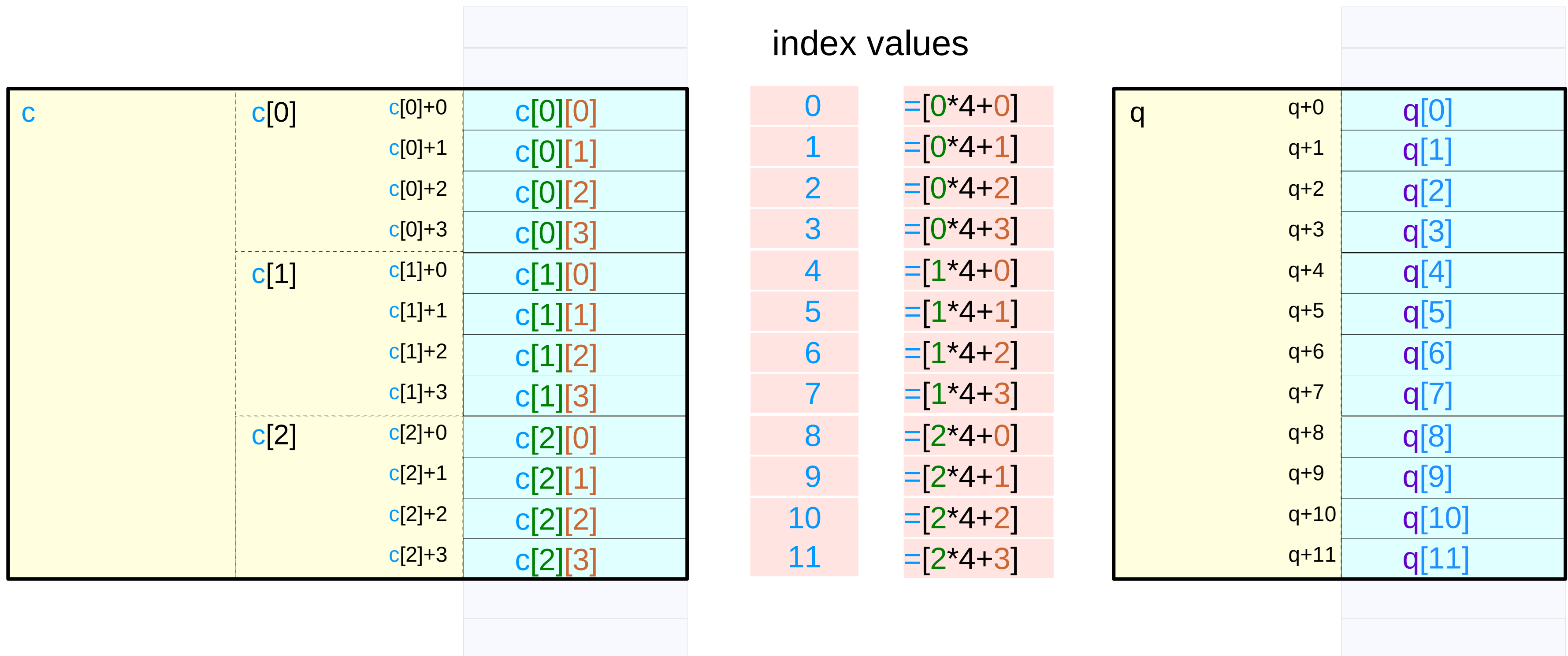
c	c[0]	c[0]+0	c[0][0]
		c[0]+1	c[0][1]
		c[0]+2	c[0][2]
		c[0]+3	c[0][3]
	c[1]	c[1]+0	c[1][0]
		c[1]+1	c[1][1]
		c[1]+2	c[1][2]
		c[1]+3	c[1][3]
	c[2]	c[2]+0	c[2][0]
		c[2]+1	c[2][1]
		c[2]+2	c[2][2]
		c[2]+3	c[2][3]

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

2-d and 1-d interpretations of linear memories

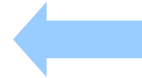


A **2-d** array stored as a **1-d** array (row major order)



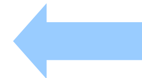
2-d array access via a single pointer

int *p = c[0] ;



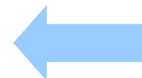
int c [3][4];

p[i*4 + j]



c[i][j]

*(p+ i*4 + j)



((c+i)+ j)

*(p +k)

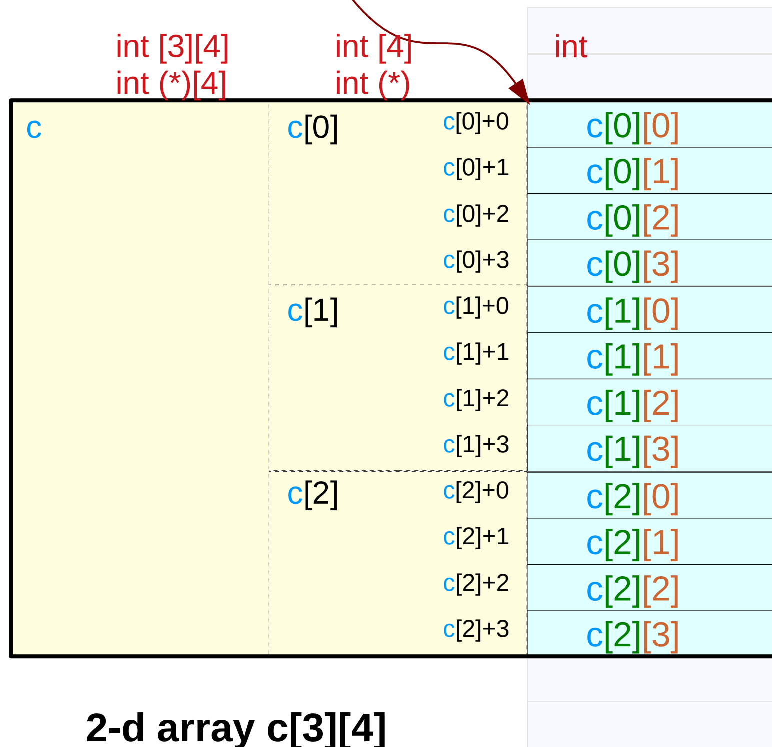
i = k / 4;
j = k % 4;

View a **2-d** array as a **1-d** array

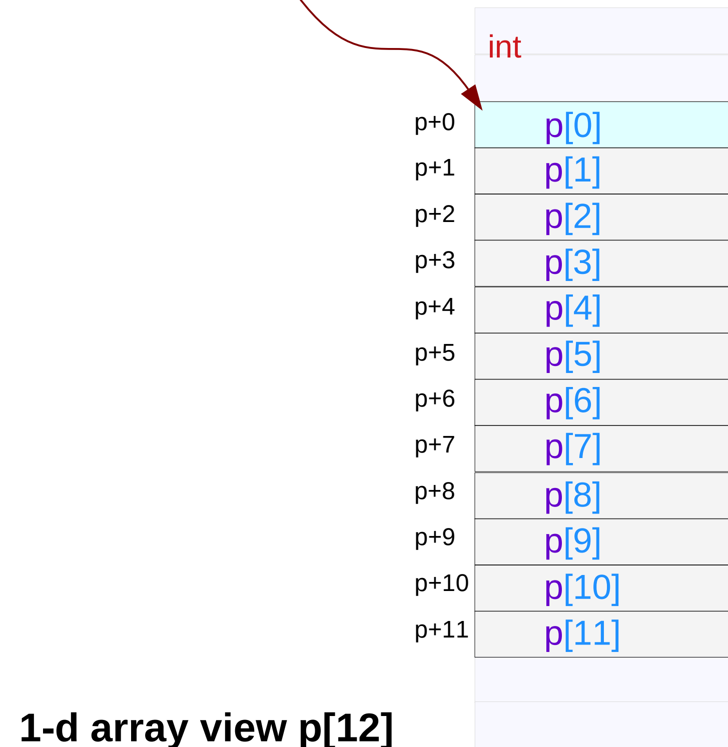
int c [3][4];

int *p = c[0];

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



2-d array c[3][4]



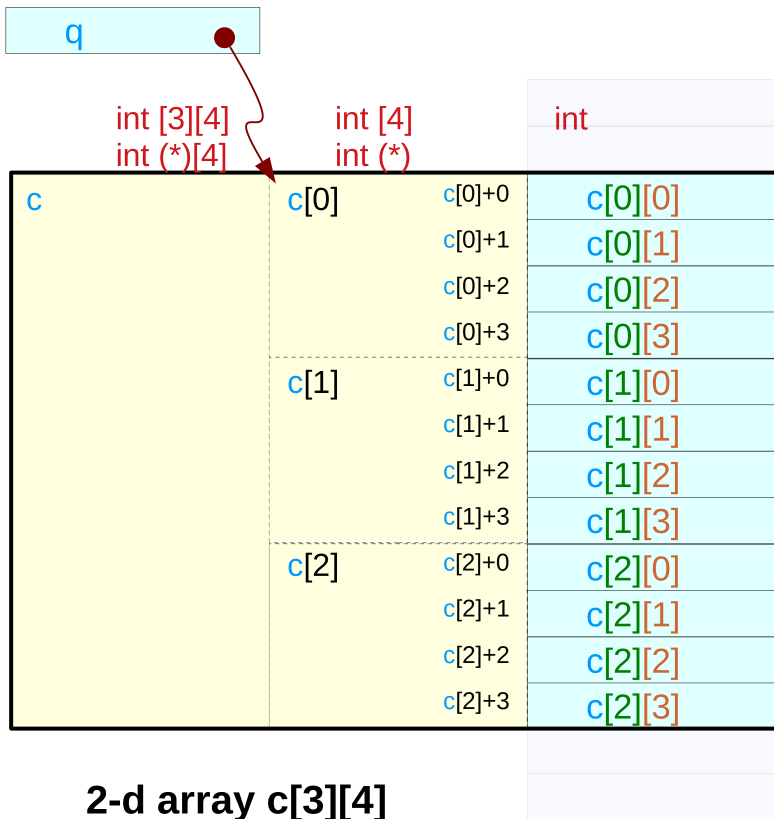
1-d array view p[12]

View a **2-d** array as another **2-d** array

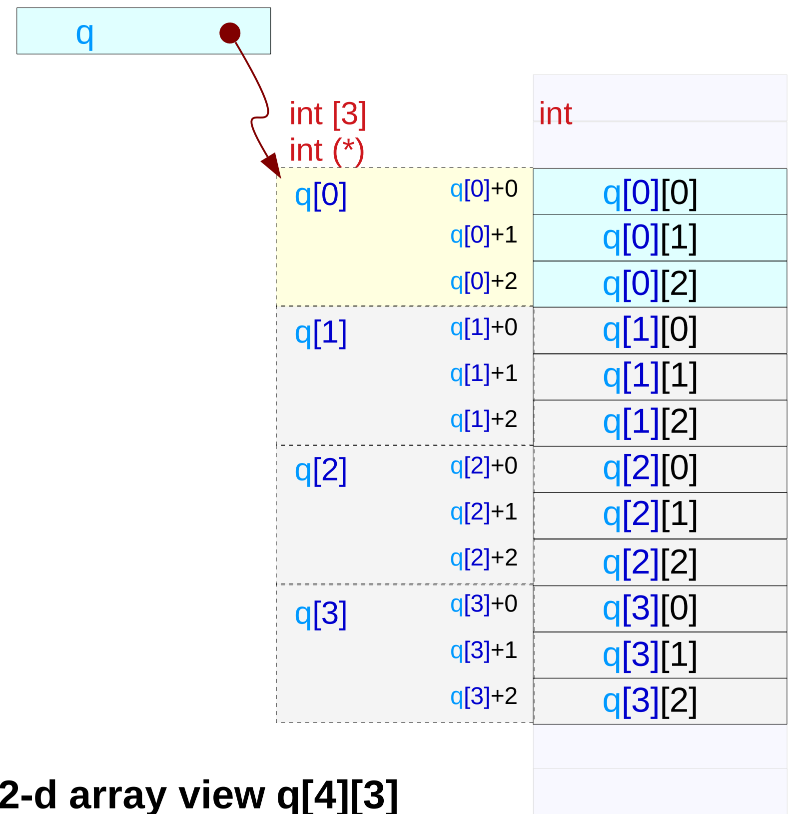
int c [3][4];

int (*q) [3] = (int (*) [3]) c;

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



2-d array c[3][4]



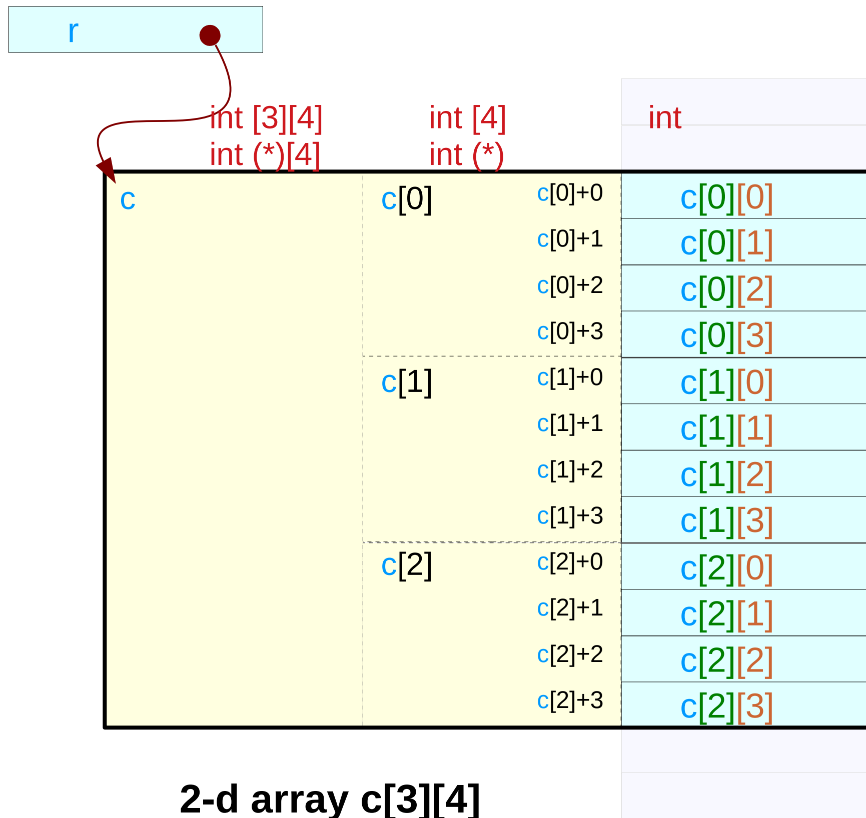
2-d array view q[4][3]

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

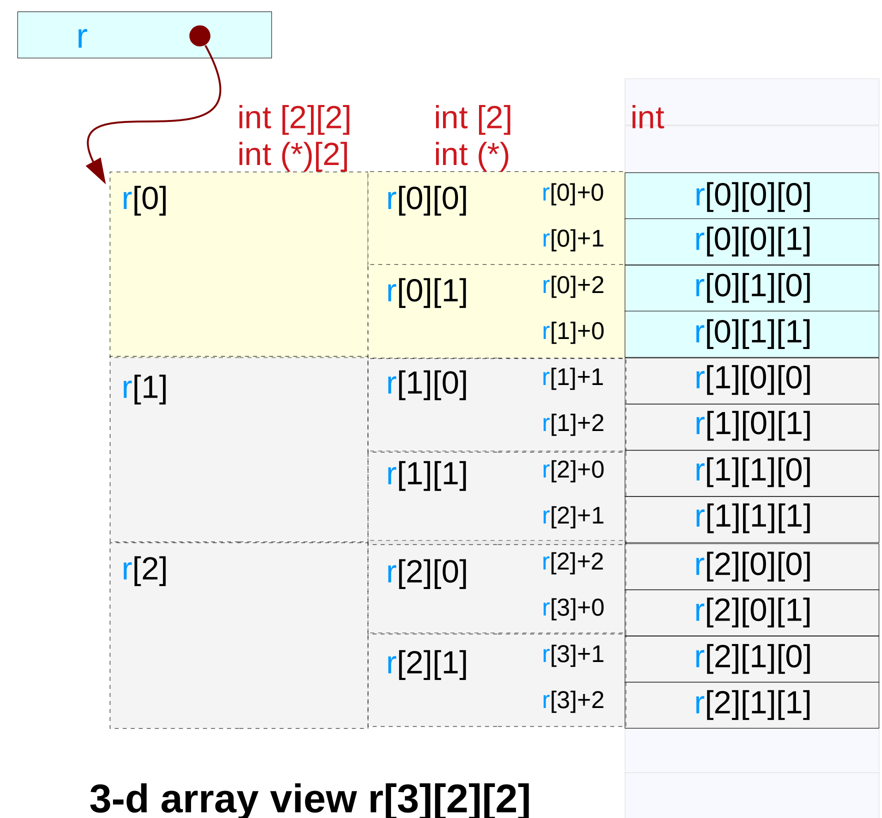
int c [3] [4];

int (*r) [2][2] = (int (*) [2][2]) c;

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



2-d array c[3][4]



3-d array view r[3][2][2]

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 ]
```

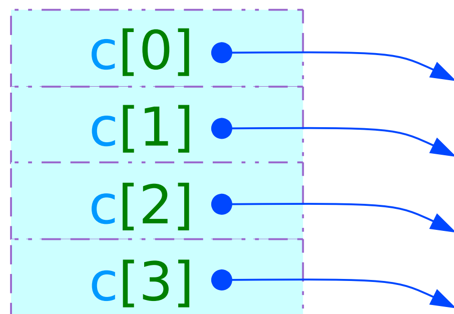
```
*(p+ i*4 + j )
```

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[ XA+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)

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