Applications of Array Pointers (1A)

Young Won Lim
2/10/24

Please send corrections (or suggestions) to youngwlim@hotmail.com.
This document was produced by using LibreOffice.

Assumption

assume that

value(c) returns the hexadecimal number
that is obtained by printf(“%p”, c),
when the variable c contains
an address as its value

```
#include <stdio.h>
int main(void) {
  int c[3] ;
  printf ("c= %p \n", &c);
}

c= 0x7fffd923487c
```

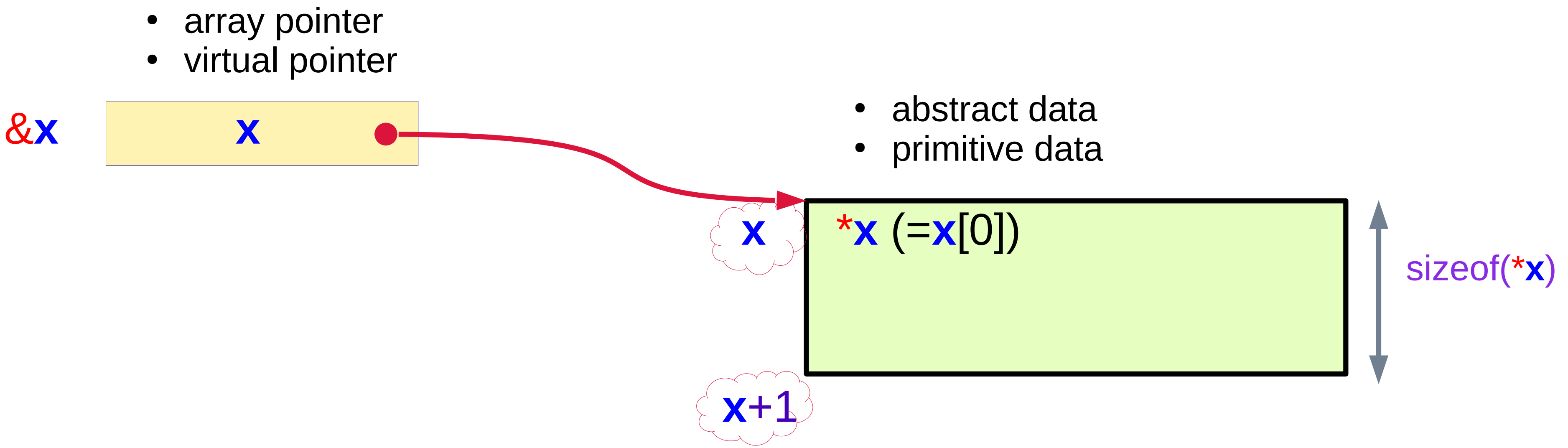
type(c) can be determined
by the warning messsage of printf(“%d”, c),
when the variable c contains
an address as its value

```
#include <stdio.h>
int main(void) {
  int c[3] ;
  printf ("c= %d \n", &c);
}

t.c: In function ‘main’:
t.c:5:16: warning: format ‘%d’ expects argument of type ‘int’,
but argument 2 has type ‘int (*)[3]’ [-Wformat=]
   printf ("c= %d \n", &c);
```

Byte Address

Pointer **x** and **x+1** relationship



- array pointer
- virtual pointer

- abstract data
- primitive data

value(**x**+1) = value(**x**) + sizeof(***x**)

Byte addresses in an array

value(x+1) = value(x) + 1 * sizeof(*x)
value(x+2) = value(x) + 2 * sizeof(*x)

byte address … byte address … byte size

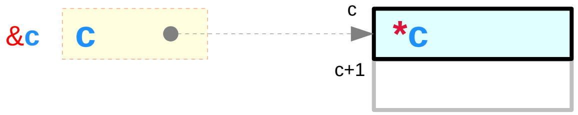
value(&x[1]) = value(x) + 1 * sizeof(x[0])
value(&x[2]) = value(x) + 2 * sizeof(x[0])

byte address … byte address … byte size

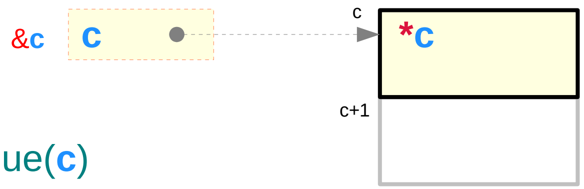
Four cases of array pointers

Virtual pointers vs. real pointers

Case 1 **virtual pointer c** **primitive data *c**



Case 2 **virtual pointer c** **abstract data *c**



subarray partitioning

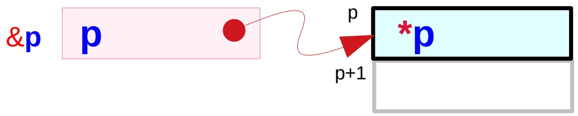
★ sizeof(c) = sizeof(*c) * N

address replication

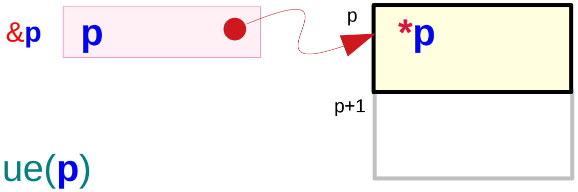
★ value(&c) = value(c)

value(c+1) = value(c) + sizeof(*c)

Case 3 **real pointer p** **primitive data *p**



Case 4 **real pointer p** **abstract data *p**



sizeof(p) = pointer size (4/8 bytes)

value(&p) ≠ value(p)

value(p+1) = value(p) + sizeof(*p)

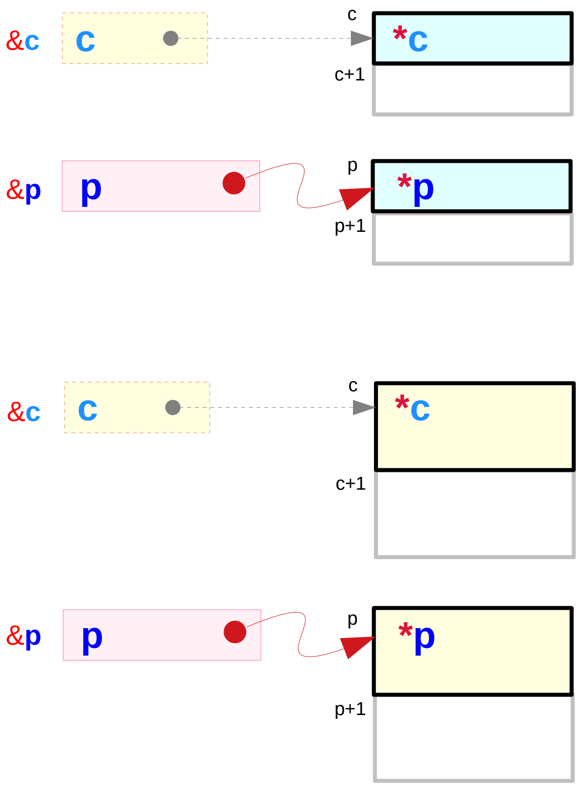
Primitive data vs. abstract data

Case 1 **virtual pointer c** **primitive data *c**

Case 3 **real pointer p** **primitive data *p**

Case 2 **virtual pointer c** **abstract data *c**

Case 4 **real pointer p** **abstract data *p**



subarray partitioning

★ sizeof(c) = sizeof(*c) * N

sizeof(p) = pointer size (4/8 bytes)

address replication

★ value(&c) = value(c)

value(&p) ≠ value(p)

value(c+1) = value(c) + sizeof(*c)

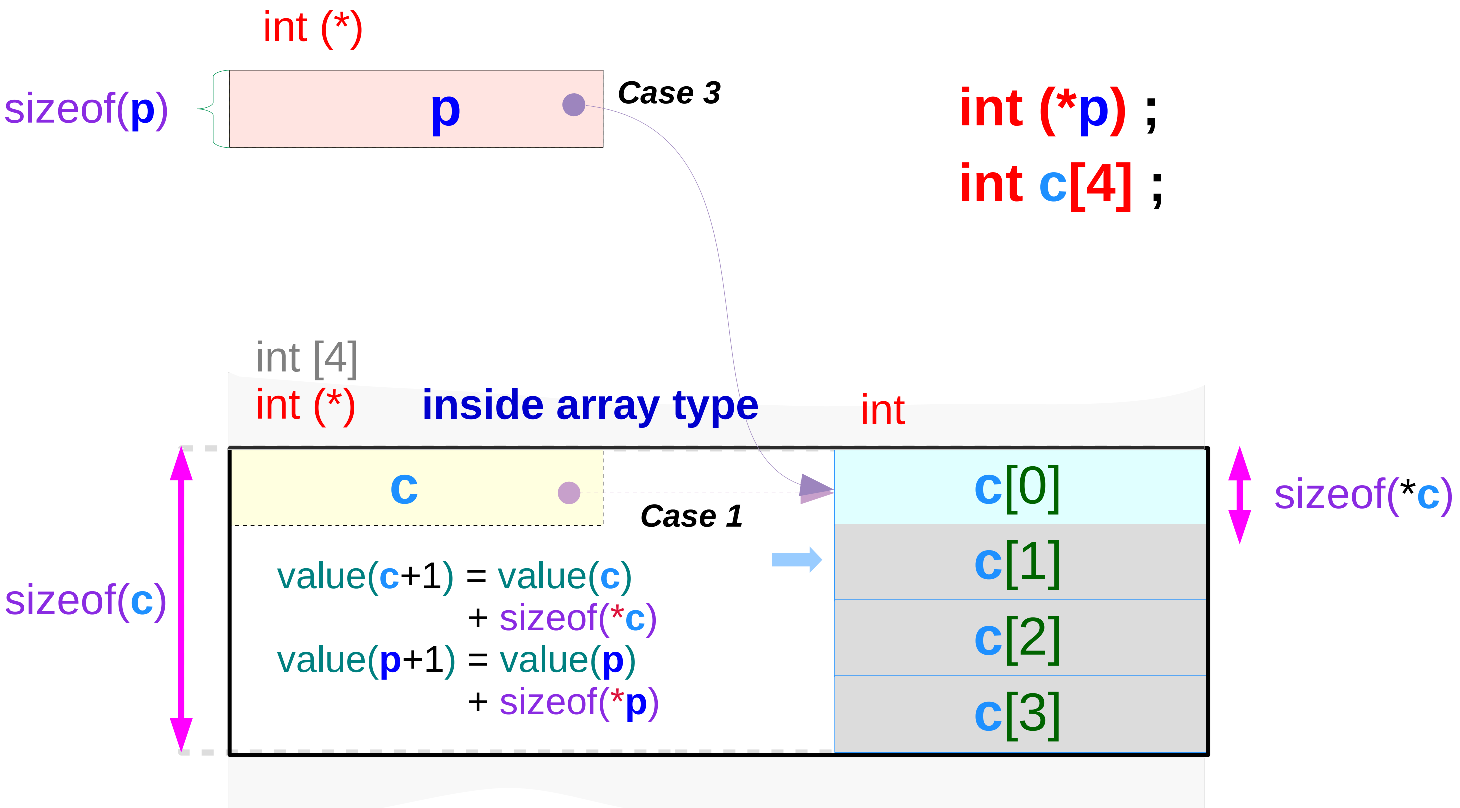
value(p+1) = value(p) + sizeof(*p)

Sizes of integer pointers

a pointer to an **int**

sizeof(**p**) = *pointer size*
= 8 bytes (64-bit CPU)
= 4 bytes (32-bit CPU)

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



int (*)

sizeof(**p**)

p

Case 3

int [4]
int (*) **inside array type** int

c

Case 1

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

sizeof(*c)

sizeof(**c**)

value(**c**+1) = value(**c**)
+ sizeof(*****c**)
value(**p**+1) = value(**p**)
+ sizeof(*****p**)

Case 1: virtual pointer c to primitive data *c
Case 3: real pointer p to primitive data *p

an **int array**

sizeof(**c**)
= sizeof(*****c**) * 4
= sizeof(int) * 4
= 4*4 = 16 bytes

value(**p**+1) = value(**p**) + sizeof(*****p**)

Sizes of integer pointers

int (*p) [4] ;

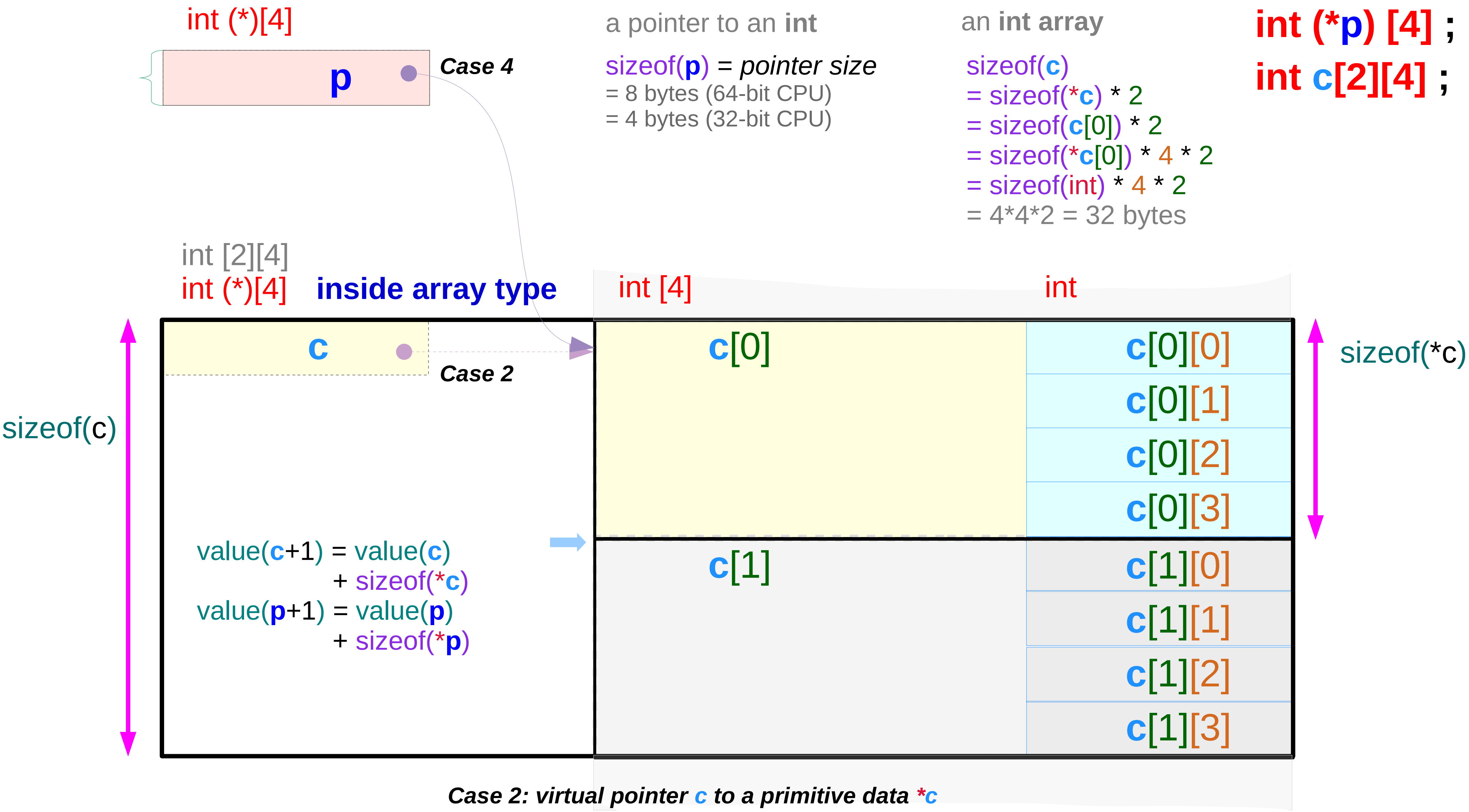
int c[2][4] ;

a pointer to an **int**

sizeof(**p**) = *pointer size*
= 8 bytes (64-bit CPU)
= 4 bytes (32-bit CPU)

an **int array**

sizeof(**c**)
= sizeof(***c**) * 2
= sizeof(**c**[0]) * 2
= sizeof(***c**[0]) * 4 * 2
= sizeof(int) * 4 * 2
= 4*4*2 = 32 bytes

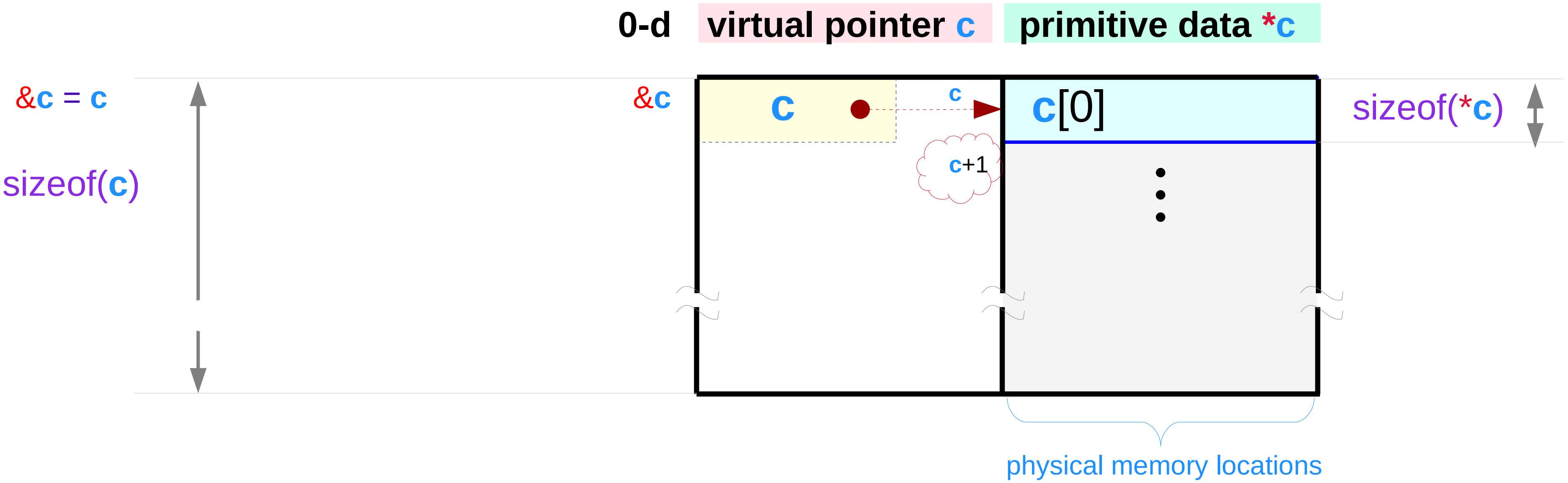


value(**c**+1) = value(**c**)
+ sizeof(***c**)

value(**p**+1) = value(**p**)
+ sizeof(***p**)

Case 2: virtual pointer c to a primitive data *c
Case 4: real pointer p to a primitive data *p

Case 1: virtual pointer **c** to a primitive data ***c**



Abstract data c subarray partitioning

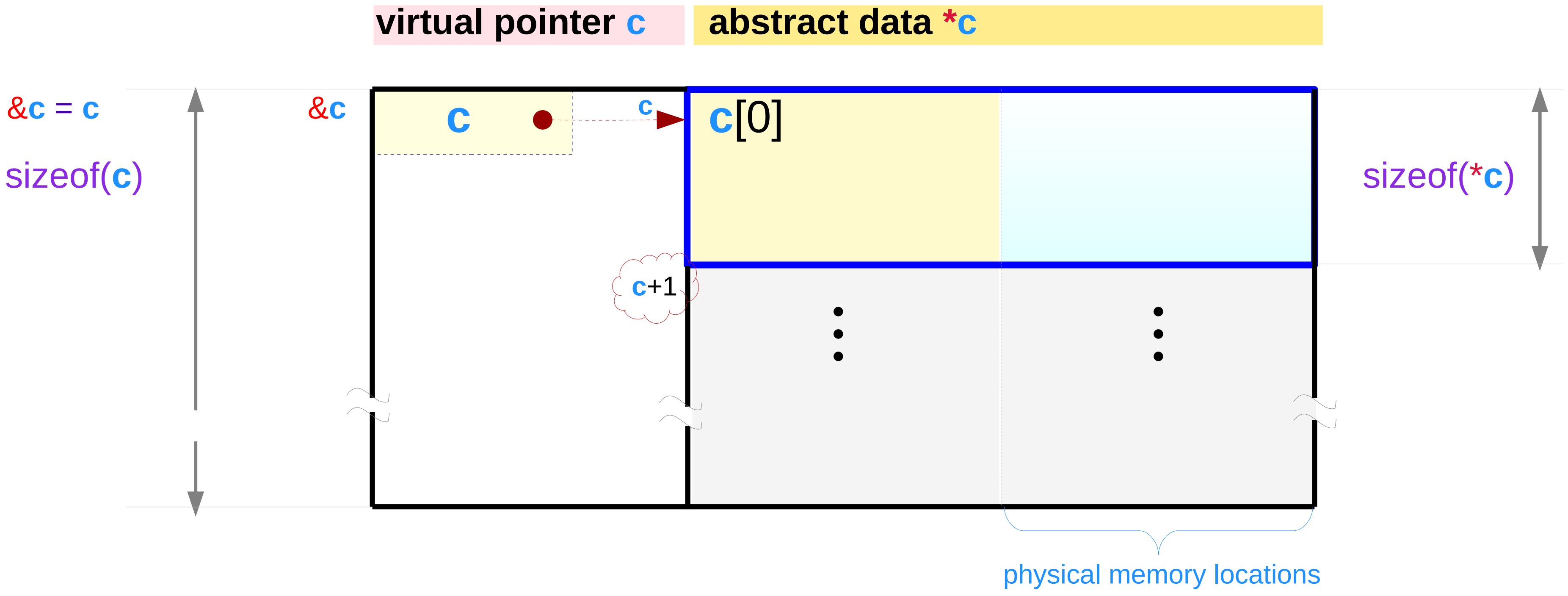
★ sizeof(c) = sizeof(*c) * N

value(c+1) = value(c) + sizeof(*c)

Virtual pointer c address replication

★ value(&c) = value(c)

Case 2: virtual pointer **c** to an abstract data ***c**



Abstract data c, *c subarray partitioning

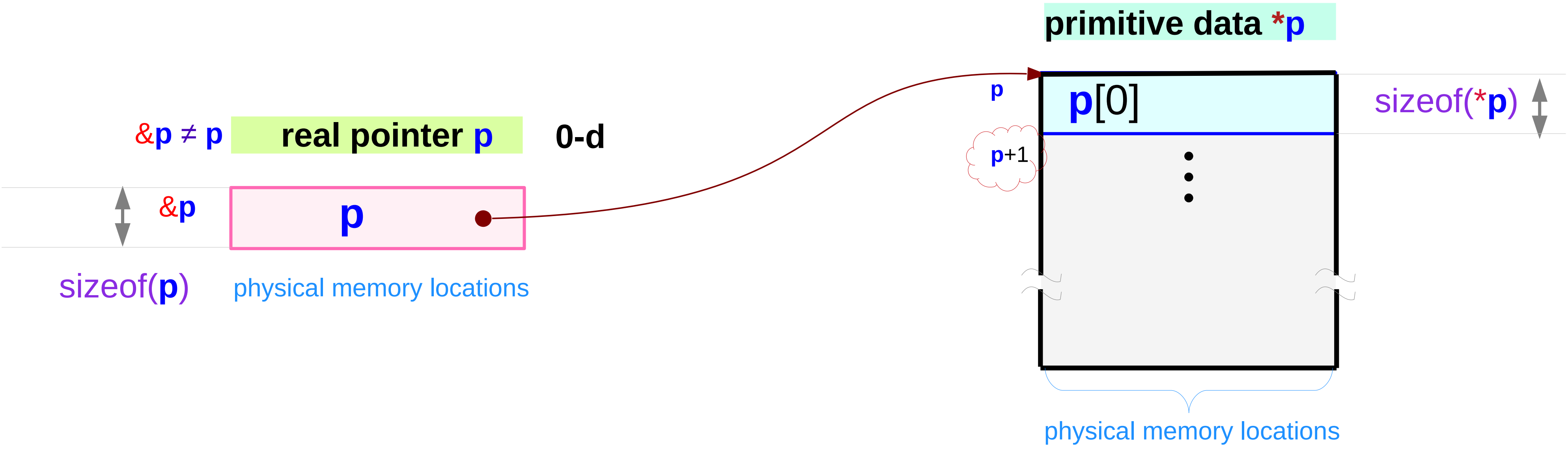
★ sizeof(c) = sizeof(*c) * N

value(c+1) = value(c) + sizeof(*c)

Virtual pointer c address replication

★ value(&c) = value(c)

Case 3: real pointer **p** to a primitive data ***p**



real pointer size

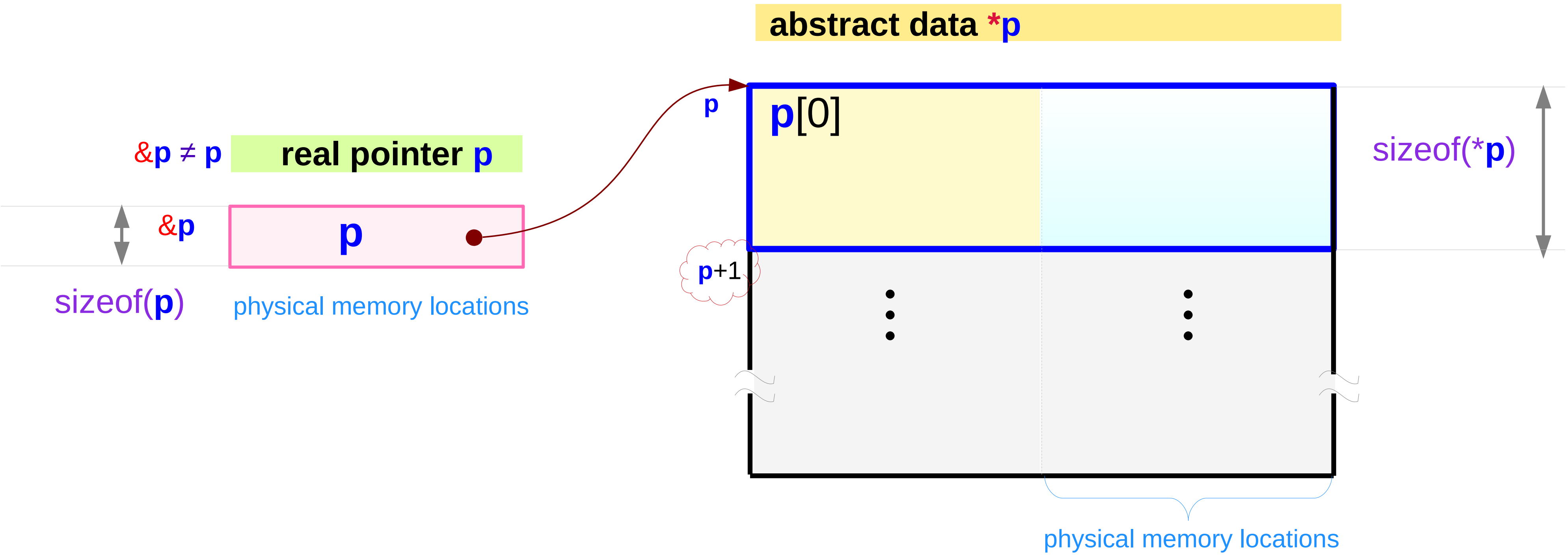
sizeof(**p**) = pointer size (4/8 bytes)

value(**p**+1) = value(**p**) + sizeof(***p**)

unique pointer value and address

value(&**p**) ≠ value(**p**)

Case 4: real pointer **p** to an abstract data ***p**



real pointer size

sizeof(**p**) = pointer size (4/8 bytes)

value(**p**+1) = value(**p**) + sizeof(***p**)

unique pointer value and address

value(&**p**) ≠ value(**p**)

Properties of array pointers

virtual pointer c in an array c **implicit array pointer**

Abstract data c[N][ ] … [ ] subarray partitioning

★ sizeof(c) = sizeof(*c) * N

value(c+1) = value(c) + sizeof(*c)

Virtual pointer (*c)[ ] … [ ] address replication

★ value(&c) = value(c)

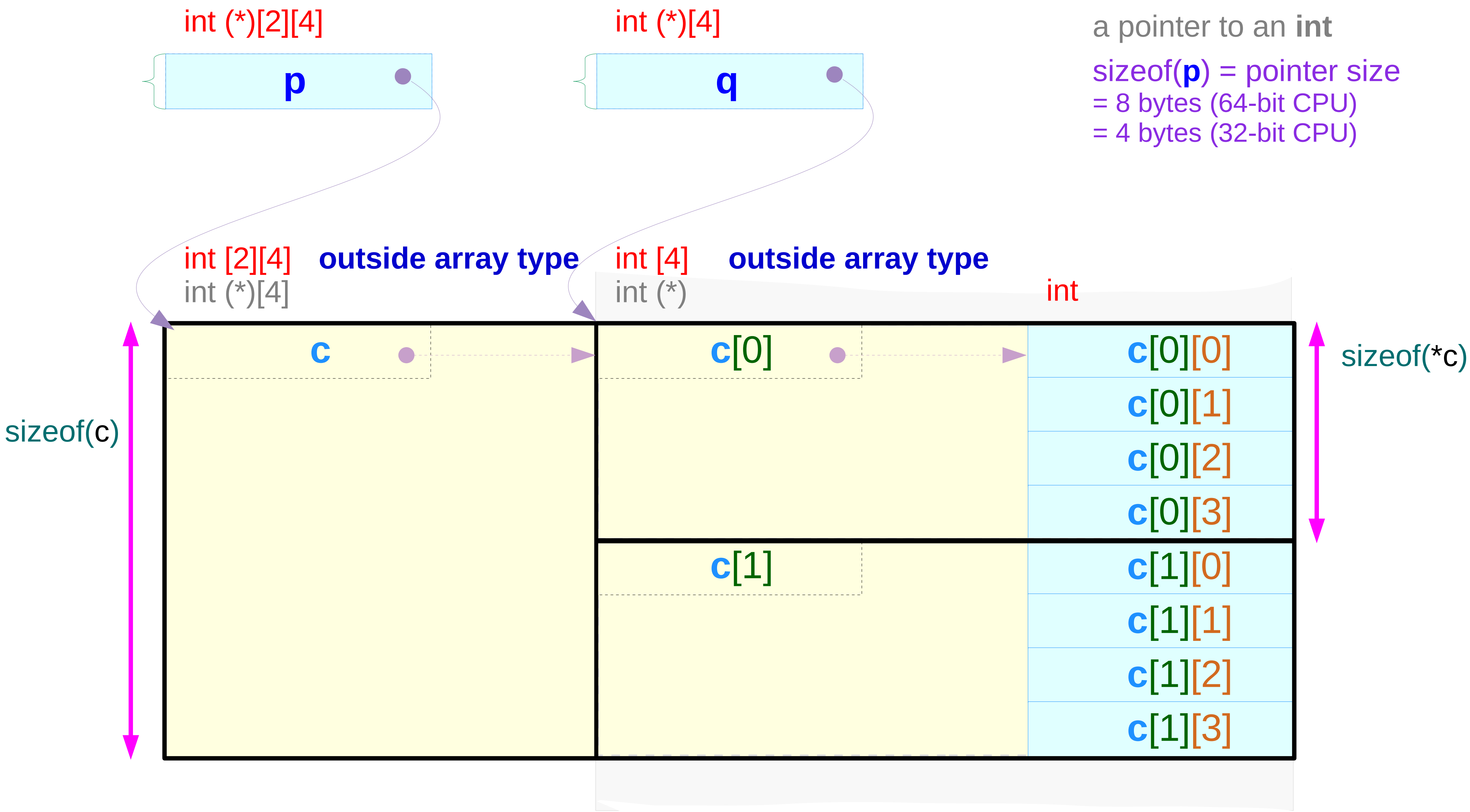
array pointer p **explicit array pointer**

sizeof(p) = pointer size (4/8 bytes)

value(p+1) = value(p) + sizeof(*p)

value(&p) ≠ value(p)

Sizes of integer pointers



a pointer to an **int**

sizeof(**p**) = pointer size
= 8 bytes (64-bit CPU)
= 4 bytes (32-bit CPU)

Array element notation `p[i]`

Dereference notation `*(p+i)`

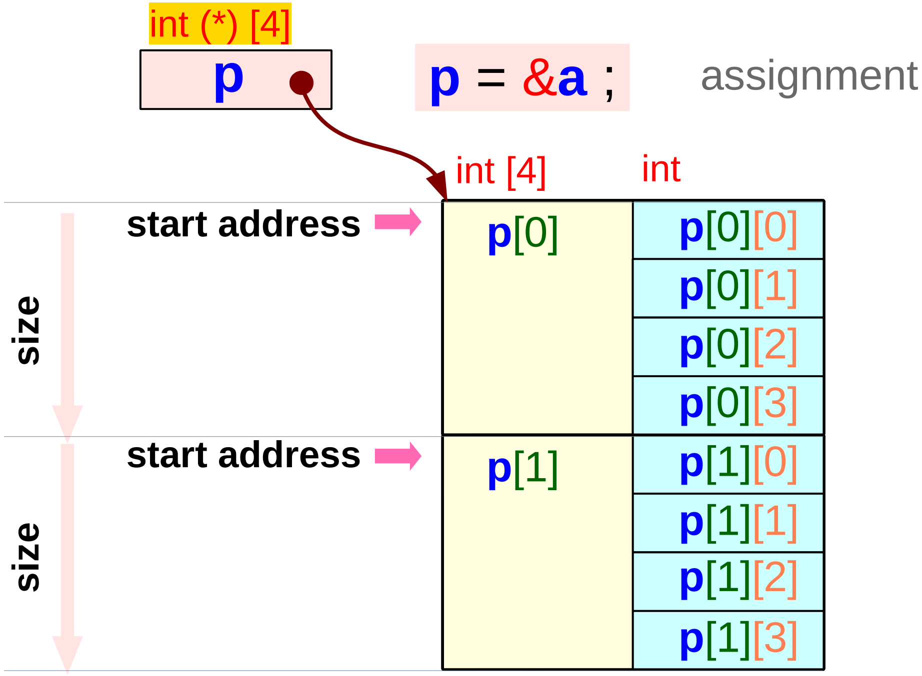
`int a[4];`	`int (*p) [4]    = &a;`	**1-d** array, **1-d** array pointer
`int c[3][4];`	`int (*p) [3][4] = &c;`	**2-d** array, **2-d** array pointer
`int c[3][4];`	`int ( *   q )[4]  =   c;`	**2-d** array, **1-d** array pointer

1-d array, **1-d** array pointer **p** – size and start address

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

value(**p**+1) – value(**p**) = sizeof(***p**) = sizeof(int [4])

= (long) (**p**+1) – (long) (**p**) = 4 * sizeof(int)



int (*) [4]

p

p = &a ;

assignment

int [4]

int

start address

size

p[0]

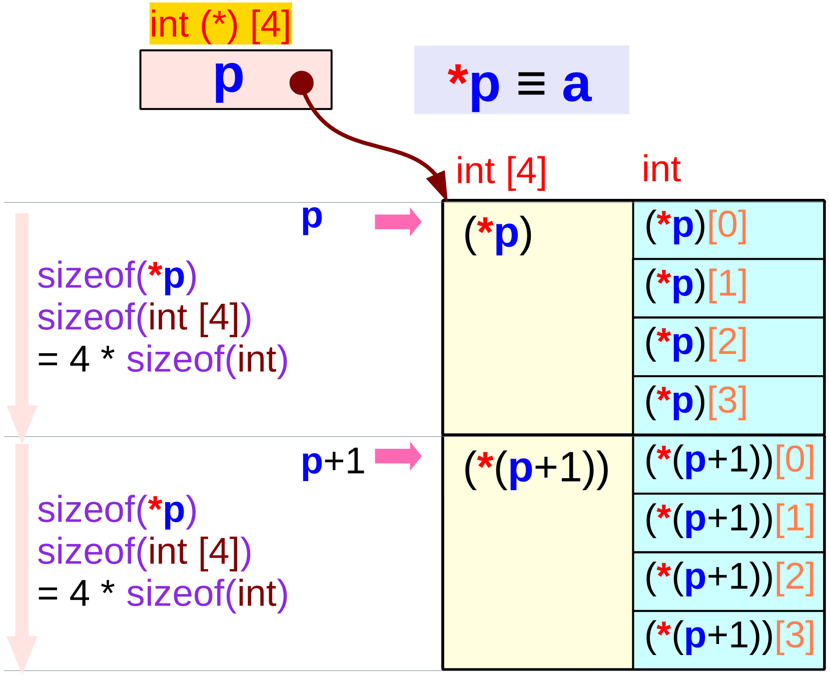
p[0][0]
p[0][1]
p[0][2]
p[0][3]

start address

size

p[1]

p[1][0]
p[1][1]
p[1][2]
p[1][3]



int (*) [4]

p

***p ≡ a**

int [4]

int

p

sizeof(***p**)
sizeof(int [4])
= 4 * sizeof(int)

(***p**)

(***p**)[0]
(***p**)[1]
(***p**)[2]
(***p**)[3]

p+1

sizeof(***p**)
sizeof(int [4])
= 4 * sizeof(int)

(*(**p**+1))

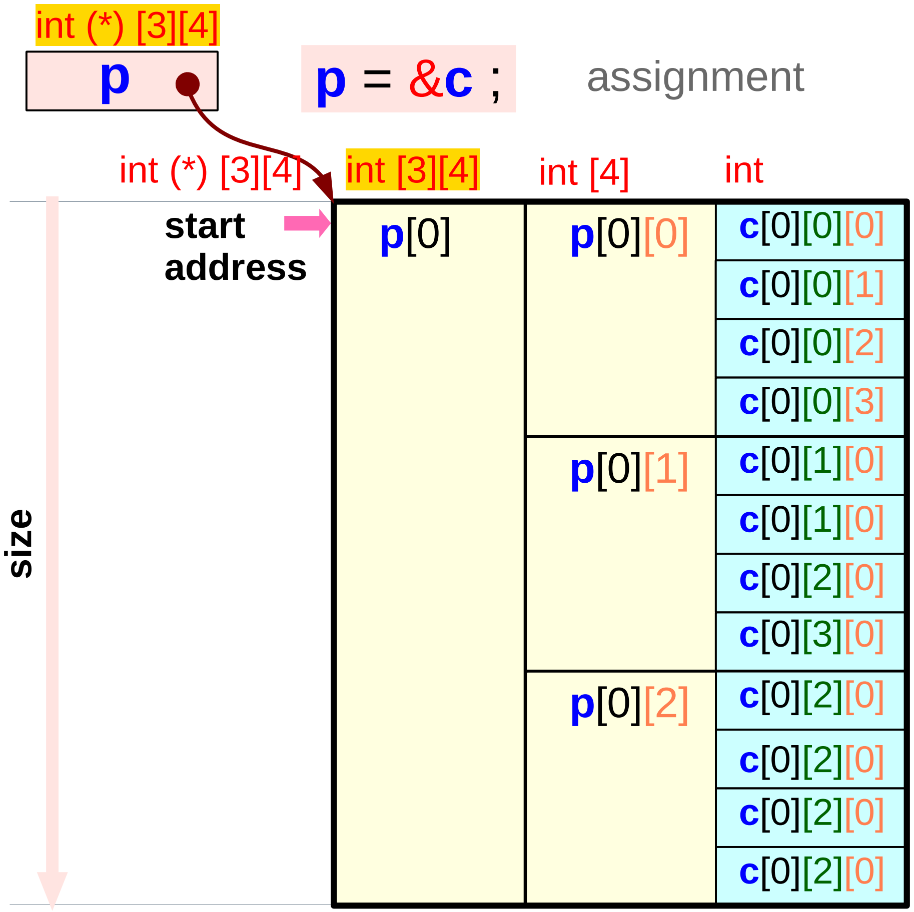
(*(**p**+1))[0]
(*(**p**+1))[1]
(*(**p**+1))[2]
(*(**p**+1))[3]

2-d array, **2-d** array pointer **p** – size and start address

int (***p**) [3][4] = &**c**;

value(**p**+1) – value(**p**) = sizeof(***p**) = sizeof(int [3][4])

= (long) (**p**+1) – (long) (**p**) = 3 * 4 * sizeof(int)



int (*) [3][4]

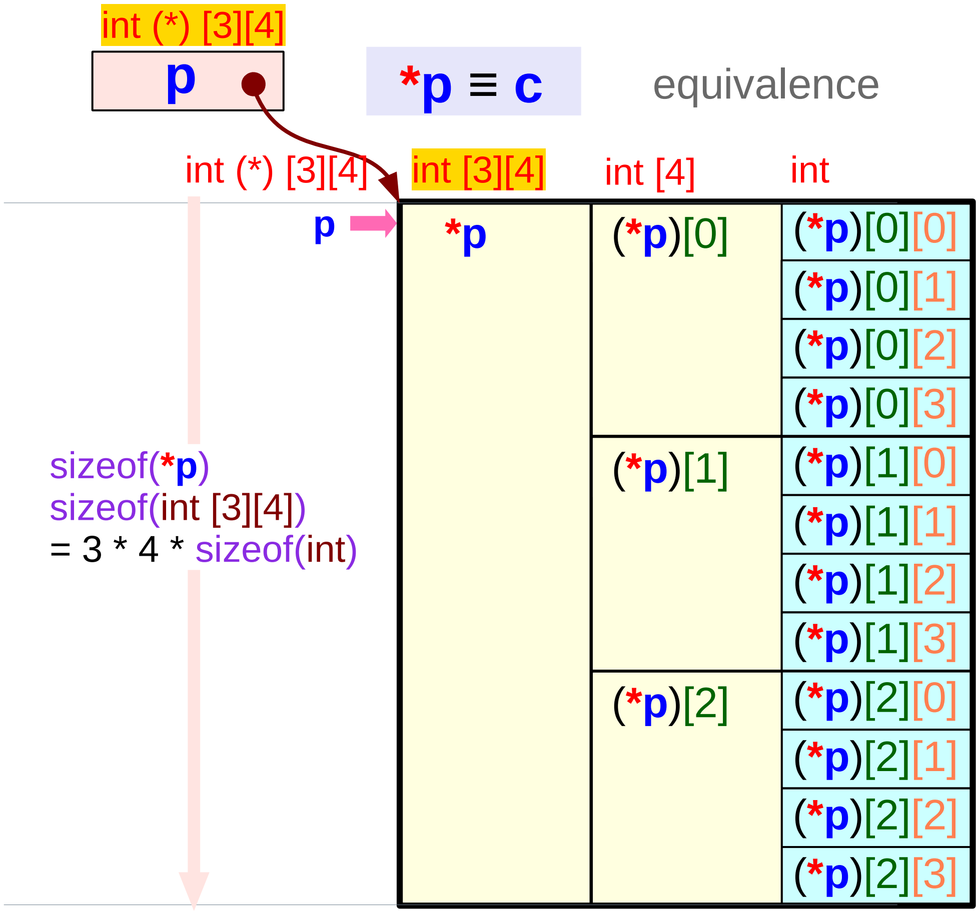
p

p = &**c** ;

assignment

int (*) [3][4]	int [3][4]	int [4]	int
start address	**p**[0]	**p**[0][0]	**c**[0][0][0]
			c[0][0][1]
			c[0][0][2]
			c[0][0][3]
		p[0][1]	**c**[0][1][0]
			c[0][1][0]
			c[0][2][0]
			c[0][3][0]
		p[0][2]	**c**[0][2][0]
			c[0][2][0]
			c[0][2][0]
			c[0][2][0]

size



int (*) [3][4]

p

***p** ≡ **c**

equivalence

sizeof(***p**)

sizeof(int [3][4])

= 3 * 4 * sizeof(int)

int (*) [3][4]	int [3][4]	int [4]	int
p	***p**	(***p**)[0]	(***p**)[0][0]
			(***p**)[0][1]
			(***p**)[0][2]
			(***p**)[0][3]
		(***p**)[1]	(***p**)[1][0]
			(***p**)[1][1]
			(***p**)[1][2]
			(***p**)[1][3]
		(***p**)[2]	(***p**)[2][0]
			(***p**)[2][1]
			(***p**)[2][2]
			(***p**)[2][3]

2-d array, **2-d** array pointer **p** – outside array type

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

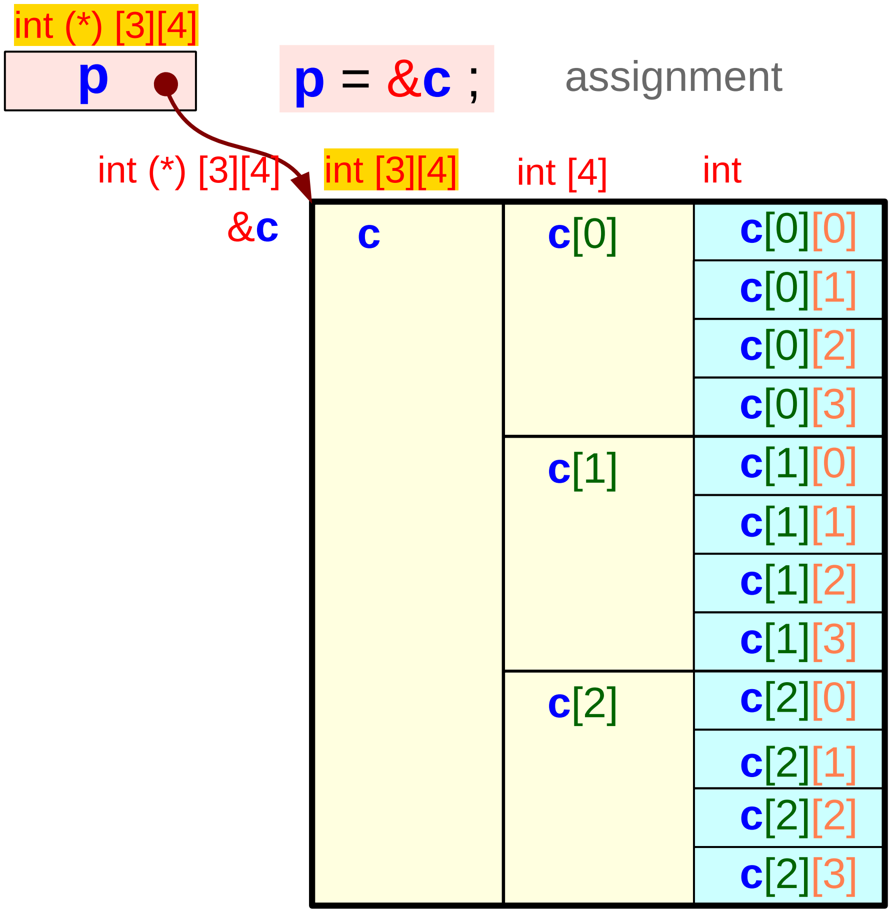
int (*__p__) [3][4];

assignment: **p** = &**c**

dereference: ***p** ≡ **c**

equivalence: **c** ≡ &**c**[0]

equivalence: **c**[0] ≡ &**c**[0][0]



int (*) [3][4]

p

p = &**c** ;

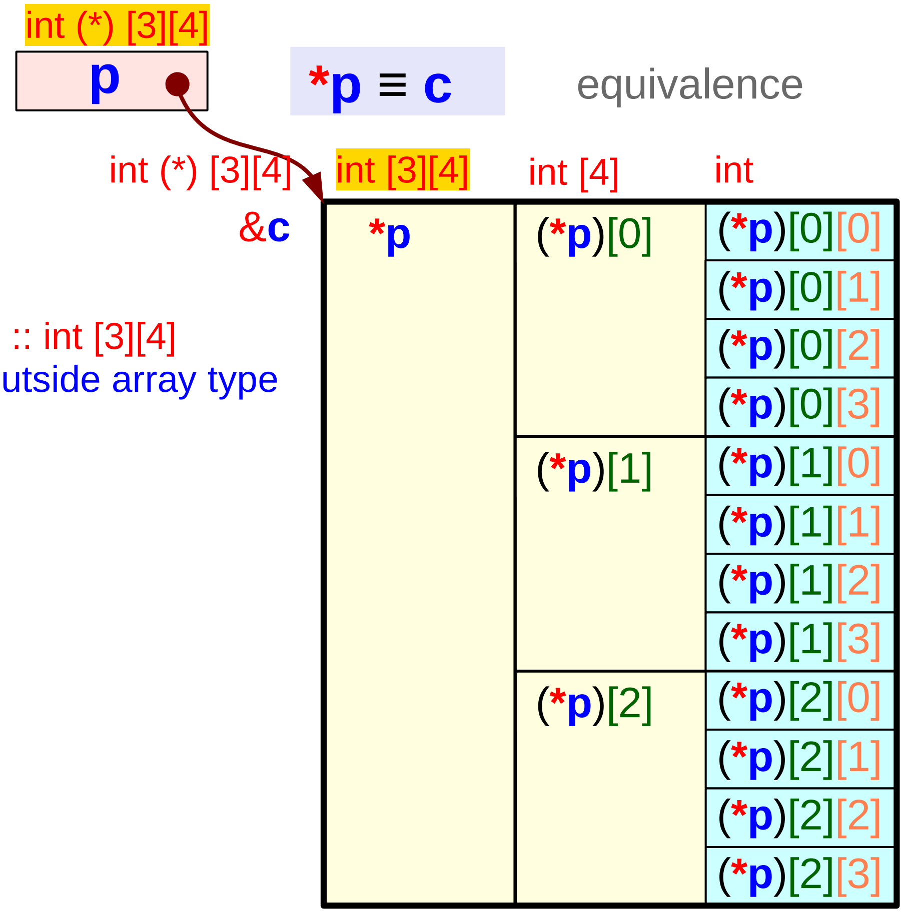
assignment

int (*) [3][4]

&**c**

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]

c :: int [3][4]
outside array type



int (*) [3][4]

p

***p** ≡ **c**

equivalence

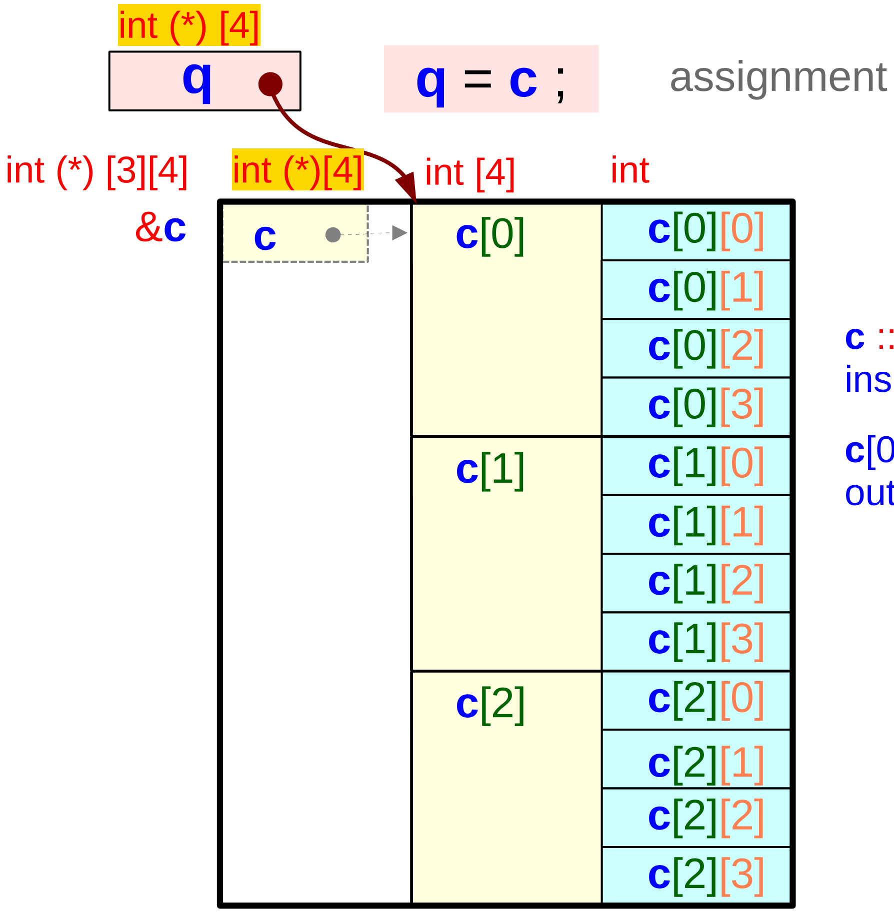
int (*) [3][4]

&**c**

int [3][4]	int [4]	int
***p**	(***p**)[0]	(***p**)[0][0]
		(***p**)[0][1]
		(***p**)[0][2]
		(***p**)[0][3]
	(***p**)[1]	(***p**)[1][0]
		(***p**)[1][1]
		(***p**)[1][2]
		(***p**)[1][3]
	(***p**)[2]	(***p**)[2][0]
		(***p**)[2][1]
		(***p**)[2][2]
		(***p**)[2][3]

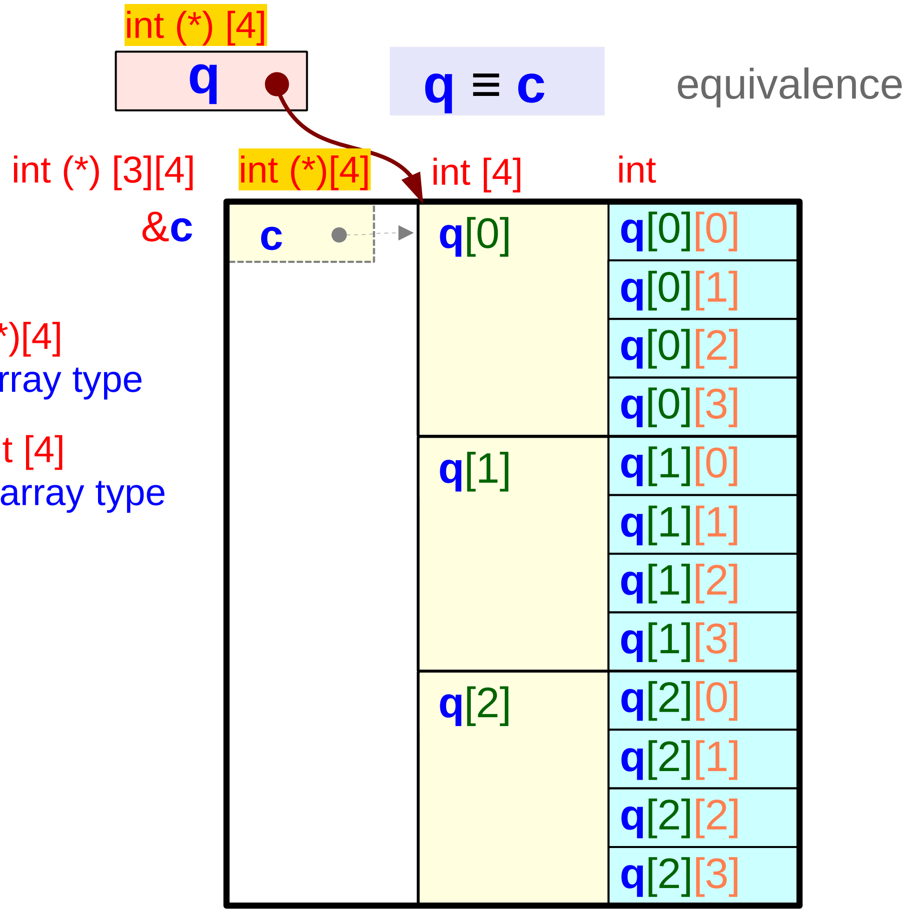
2-d array, **1-d** array pointer **q**– inside array type

	assignment	dereference	equivalence	equivalence
int **c** [3][4];				
int (***q**) [4];	**q** = **c**	**q** ≡ **c**	**c** ≡ &**c**[0]	**c**[0] ≡ &**c**[0][0]



c :: int (*)[4]
inside array type

c[0] :: int [4]
outside array type



Incrementing a 2-d array pointer

Incrementing a 1-d array pointer

Subarray sizes referenced by array pointers **p** and **q**

int **c** [3][4] ;	int (*****p**) [3][4] ;	int (*****q**) [4] ;
2-d array c	**2-d array pointer p**	**1-d array pointer q**
sizeof(**c**) = **array size** 3·4·4 bytes	sizeof(**p**) = **pointer size** 4 or 8 bytes	sizeof(**q**) = **pointer size** 4 or 8 bytes
int **c** [3][4] **c** : sizeof(int [3][4])	int (***p**) [3][4] (***p**) or **p**[0] : sizeof(int [3][4])	
int **c** [3][4] i = 0,1,2 **c**[**i**] : sizeof(int [4])	int (***p**) [3][4] i = 0,1,2 (***p**)[**i**] or **p**[0][**i**] : sizeof(int [4])	int (***q**) [4] i = 0,1,2 *(**q**+**i**) or **q**[**i**] : sizeof(int [4])
int **c** [3][4] i = 0,1,2 j = 0,1,2,3 **c**[**i**][**j**] : sizeof(int)	int (***p**) [3][4] i = 0,1,2 j = 0,1,2,3 (***p**)[**i**][**j**] or **p**[0][**i**][**j**] : sizeof(int)	int (***q**) [4] i = 0,1,2 j = 0,1,2,3 *(*(**q**+**i**)+**j**) or **q**[**i**][**j**] : sizeof(int)

2-d array **c**, **2-d** array pointer **p**, **1-d** array pointer **q**

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

```
int (*p) [3][4] = &c;
```

	Basic Reference			Extended Reference		
2-d	(*p)[i][j]	i = 0,1,2 j = 0,1,2,3	→	(*(p+m))[i][j]	i = 0,1,2 j = 0,1,2,3	2-d
3-d	p[0][i][j]	i = 0,1,2 j = 0,1,2,3	→	p[m][i][j]	i = 0,1,2 j = 0,1,2,3	3-d

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

	Basic Reference			Extended Reference		
1-d	(*q)[j]	j = 0,1,2,3	→	(*(q+i))[j]	i = 0,1,2 j = 0,1,2,3	1-d
2-d	q[0][j]	j = 0,1,2,3	→	q[i][j]	i = 0,1,2 j = 0,1,2,3	2-d

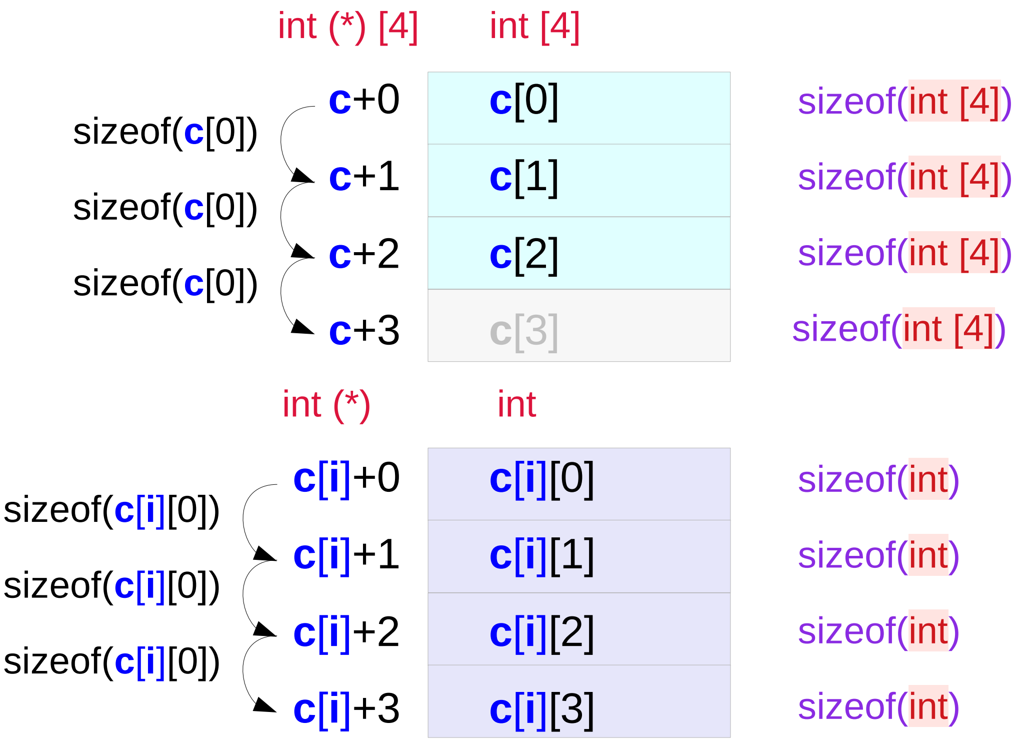
2-d array **c**

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

(int [4]) **c[i]** i = 0,1,2

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

(int) **c[i][j]** i = 0,1,2
j = 0,1,2,3



	int (*) [4]	int [4]	
	c+0	**c**[0]	sizeof(int [4])
sizeof(**c**[0])	**c**+1	**c**[1]	sizeof(int [4])
sizeof(**c**[0])	**c**+2	**c**[2]	sizeof(int [4])
sizeof(**c**[0])	**c**+3	**c**[3]	sizeof(int [4])

	int (*)	int	
	c[i]+0	**c[i]**[0]	sizeof(int)
sizeof(**c[i]**[0])	**c[i]**+1	**c[i]**[1]	sizeof(int)
sizeof(**c[i]**[0])	**c[i]**+2	**c[i]**[2]	sizeof(int)
sizeof(**c[i]**[0])	**c[i]**+3	**c[i]**[3]	sizeof(int)

value(**c** + **i**) = value(**c**) + **i** * sizeof(***c**)

value(**c[i]** + **j**) = value(**c[i]**) + **j** * sizeof(***c**[0])

value(**c** + **i**) = value(**c[i]**) *(address replication)*

&**c[i][j]** = value(**c**) + **i** * sizeof(***c**) + **j** * sizeof(***c**[0])

Incrementing a **2-d** array pointer **p**

`int (*p) [3][4] = &c ;`

`int c [3][4] ;`

p+0

sizeof(*p) sizeof(int [3][4]) sizeof(**2-d array**)

p+1

sizeof(*p) sizeof(int [3][4]) sizeof(**2-d array**)

p+2

sizeof(*p) sizeof(int [3][4]) sizeof(**2-d array**)

p+3

(*p)[i][j] i = 0,1,2 j = 0,1,2,3

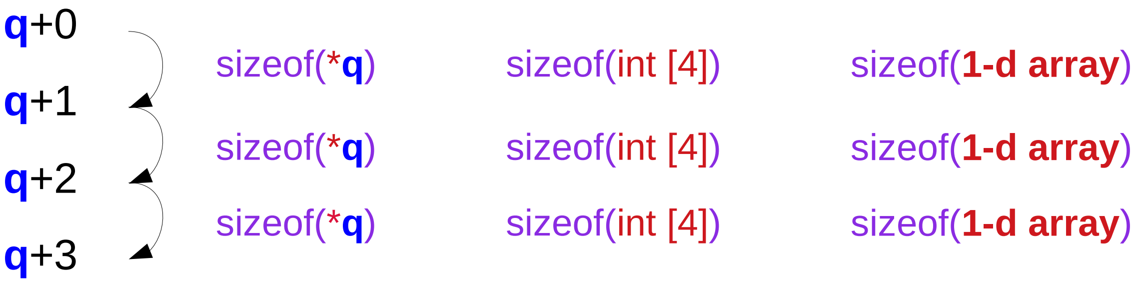
p[0][i][j] i = 0,1,2 j = 0,1,2,3

*(p+0)	⟺	p[0]	(*(p+0))[i][j]	⟺	p[0][i][j]	used for c[3][4]
*(p+1)	⟺	p[1]	(*(p+1))[i][j]	⟺	p[1][i][j]	
*(p+2)	⟺	p[2]	(*(p+2))[i][j]	⟺	p[2][i][j]	
*(p+3)	⟺	p[3]	(*(p+3))[i][j]	⟺	p[3][i][j]	

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

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

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



q+0

q+1

q+2

q+3

sizeof(*q)   sizeof(int [4])   sizeof(**1-d array**)

sizeof(*q)   sizeof(int [4])   sizeof(**1-d array**)

sizeof(*q)   sizeof(int [4])   sizeof(**1-d array**)

(*(**q**+**i**))[**j**]   i = 0,1,2   j = 0,1,2,3

q[**i**][**j**]   i = 0,1,2   j = 0,1,2,3

*(**q**+0) ⟷ **q**[0]

*(**q**+1) ⟷ **q**[1]

*(**q**+2) ⟷ **q**[2]

*(**q**+3) ⟷ **q**[3]

(*(**q**+0))[**j**] ⟷ **q**[0][**j**]

(*(**q**+1))[**j**] ⟷ **q**[1][**j**]

(*(**q**+2))[**j**] ⟷ **q**[2][**j**]

(*(**q**+3))[**j**] ⟷ **q**[3][**j**]

used for c[3][4]

2-d array pointer **p**

int (*p) [3][4] ;

(int [3][4]) (*p)

int [3][4]	int (*) [3][4]	int [3][4]	
sizeof(*p)	p+0	*p	sizeof(int [3][4])
sizeof(*p)	p+1	*(p+1)	sizeof(int [3][4])
sizeof(*p)	p+2	*(p+2)	sizeof(int [3][4])
	p+3	*(p+3)	sizeof(int [3][4])

int (*p) [3][4] ;

(int [4]) (*p)[i]   i = 0,1,2

int [4]	int (*) [4]	int [4]	
sizeof((*p)[0])	(*p)+0	(*p)[0]	sizeof(int [4])
sizeof((*p)[0])	(*p)+1	(*p)[1]	sizeof(int [4])
sizeof((*p)[0])	(*p)+2	(*p)[2]	sizeof(int [4])
	(*p)+3	(*p)[3]	sizeof(int [4])

int (*p) [3][4] ;

(int) (*p)[i][j]   i = 0,1,2
j = 0,1,2,3

int	int (*)	int	
sizeof((*p)[0][0])	(*p)[i]+0	(*p)[i][0]	sizeof(int)
sizeof((*p)[0][0])	(*p)[i]+1	(*p)[i][1]	sizeof(int)
sizeof((*p)[0][0])	(*p)[i]+2	(*p)[i][2]	sizeof(int)
	(*p)[i]+3	(*p)[i][3]	sizeof(int)

2-d array pointer **p**

int (*p) [3][4] ;

(int [3][4]) **p**[0]

int [3][4]	int (*) [3][4]	int [3][4]	
	p+0	**p**[0]	sizeof(int [3][4])
sizeof(**p**[0])	**p**+1	**p**[1]	sizeof(int [3][4])
sizeof(**p**[0])	**p**+2	**p**[2]	sizeof(int [3][4])
sizeof(**p**[0])	**p**+3	**p**[3]	sizeof(int [3][4])

int (*p) [3][4] ;

(int [4]) **p**[0][**i**] i = 0,1,2

int [4]	int (*) [4]	int [4]	
	p[0]+0	**p**[0][0]	sizeof(int [4])
sizeof(**p**[0][0])	**p**[0]+1	**p**[0][1]	sizeof(int [4])
sizeof(**p**[0][0])	**p**[0]+2	**p**[0][2]	sizeof(int [4])
sizeof(**p**[0][0])	**p**[0]+3	**p**[0][3]	sizeof(int [4])

int (*p) [3][4] ;

(int) **p**[0][**i**][**j**] i = 0,1,2
j = 0,1,2,3

int	int (*)	int	
	p[0][**i**]+0	**p**[0][**i**][0]	sizeof(int)
sizeof(**p**[0][0][0])	**p**[0][**i**]+1	**p**[0][**i**][1]	sizeof(int)
sizeof(**p**[0][0][0])	**p**[0][**i**]+2	**p**[0][**i**][2]	sizeof(int)
sizeof(**p**[0][0][0])	**p**[0][**i**]+3	**p**[0][**i**][3]	sizeof(int)

1-d array pointer **q**

int (*q) [4] ;

(int [4]) ***(q+i)** i = 0,1,2

int [4]	int (*) [4]	int [4]	
	q+0	***q**	sizeof(int [4])
sizeof(***q**)	**q**+1	*(**q**+1)	sizeof(int [4])
sizeof(***q**)	**q**+2	*(**q**+2)	sizeof(int [4])
sizeof(***q**)	**q**+3	*(**q**+3)	sizeof(int [4])

int (*q) [4] ;

(int) **(*(q+i))[j]** i = 0,1,2 j = 0,1,2,3

int	int (*)	int	
	*(**q**+**i**)+0	(*(**q**+**i**))[0]	sizeof(int)
sizeof((***q**)[0])	*(**q**+**i**)+1	(*(**q**+**i**))[1]	sizeof(int)
sizeof((***q**)[0])	*(**q**+**i**)+2	(*(**q**+**i**))[2]	sizeof(int)
sizeof((***q**)[0])	*(**q**+**i**)+3	(*(**q**+**i**))[3]	sizeof(int)

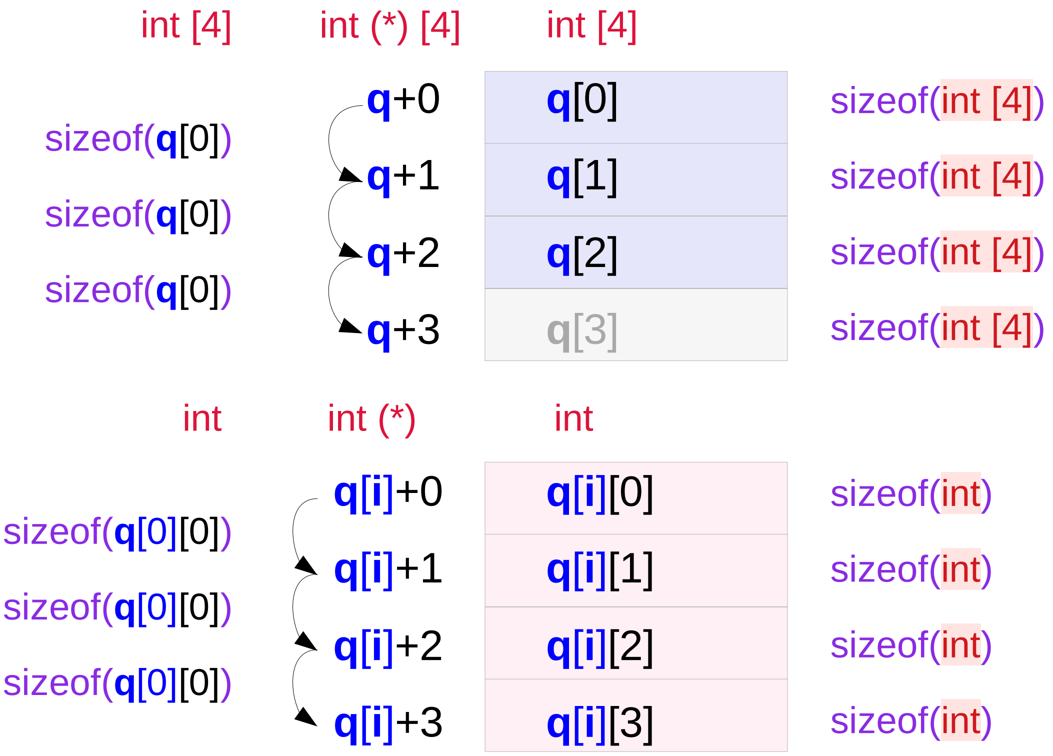
1-d array pointer **q**

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

(int [4]) **q**[i] i = 0,1,2

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

(int) **q**[i][j] i = 0,1,2
j = 0,1,2,3



int [4]	int (*) [4]	int [4]	
	q+0	q[0]	sizeof(int [4])
sizeof(q[0])	q+1	q[1]	sizeof(int [4])
sizeof(q[0])	q+2	q[2]	sizeof(int [4])
sizeof(q[0])	q+3	q[3]	sizeof(int [4])

int	int (*)	int	
	q[i]+0	q[i][0]	sizeof(int)
sizeof(q[0][0])	q[i]+1	q[i][1]	sizeof(int)
sizeof(q[0][0])	q[i]+2	q[i][2]	sizeof(int)
sizeof(q[0][0])	q[i]+3	q[i][3]	sizeof(int)

Array element address

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

```
value(c + i) =
value(c) + i * sizeof(*c)
```

```
value(c[i] + j) =
value(c[i]) + j * sizeof(*c[i])
```

```
value(c + i) = value(c[i])
```

address replication

```
&c[i][j] = value(c[i] + j)
= value(c[i]) + j * sizeof(*c[i])
= value(c + i) + j * sizeof(*c[i])
= value(c) + i * sizeof(*c)
                + j * sizeof(*c[i])
= value(c) + i * 4 * 4 + j * 4
```

```
int (*p) [3][4] = &c;
```

```
value(*p + i) =
value(*p) + i * sizeof(**p)
```

```
value((*p)[i] + j) =
value((*p)[i]) + j * sizeof(*(*p)[i])
```

```
value(*p + i) = value((*p)[i])
```

address replication

```
&(*p)[i][j] = value((*p)[i] + j)
= value((*p)[i]) + j*sizeof(*(*p)[i])
= value((*p) + i) + j*sizeof(*(*p)[i])
= value(*p) + i * sizeof(**p)
                + j * sizeof(*(*p)[i])
= value(*p) + i * 4 * 4 + j * 4
```

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

```
value(q + i) =
value(q) + i * sizeof(*q)
```

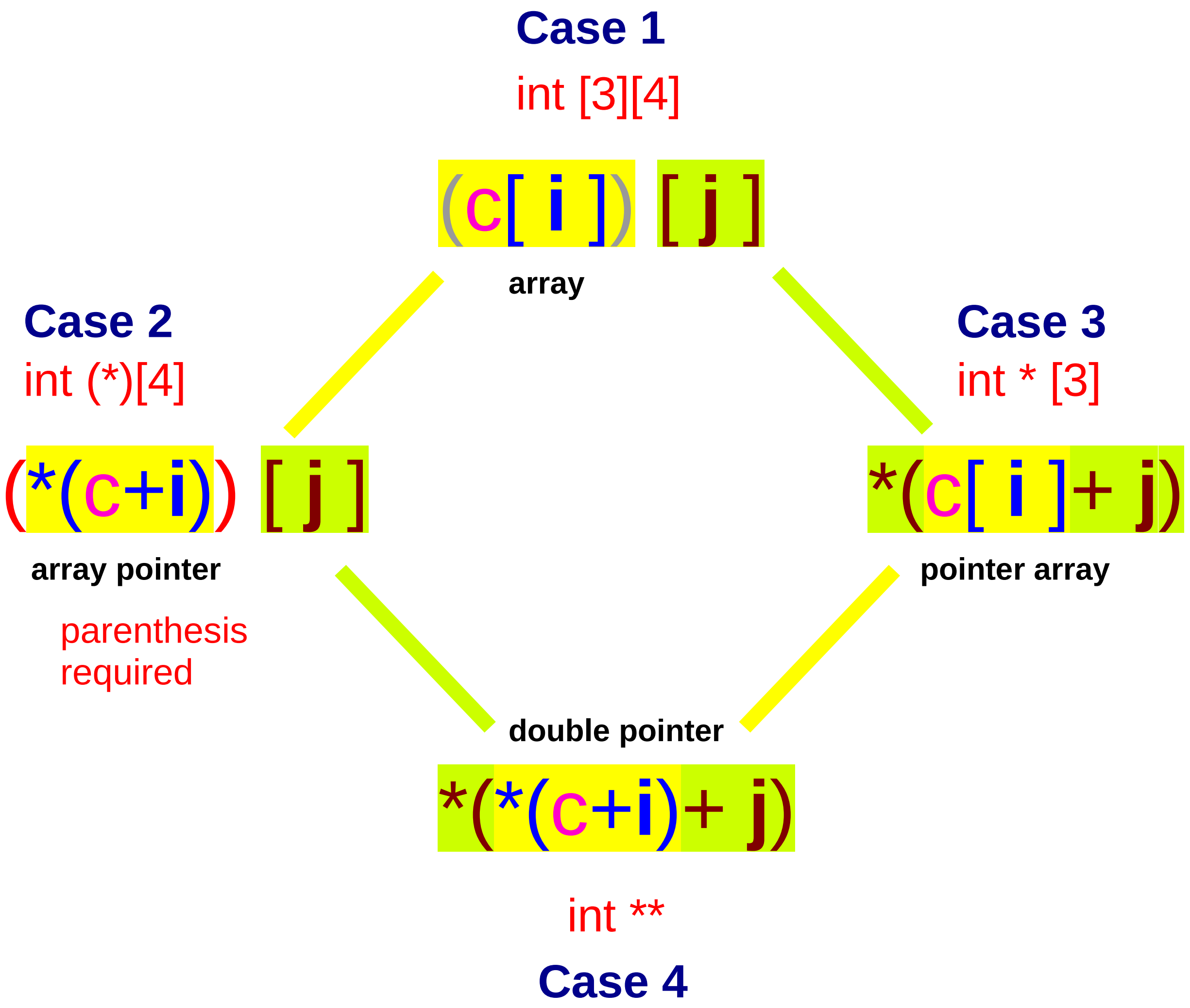
```
value(q[i] + j) =
value(q[i]) + j * sizeof(*q[i])
```

```
value(q + i) = value(q[i])
```

address replication

```
&q[i][j] = value(q[i] + j)
= value(q[i]) + j * sizeof(*q[i])
= value(q + i) + j * sizeof(*q[i])
= value(q) + i * sizeof(*q)
                + j * sizeof(*q[i])
= value(q) + i * 4 * 4 + j * 4
```

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



Case 1

int [3][4]

(c[i]) [j]

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)

int **

Case 4

2-d array-pointer conversion types

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**

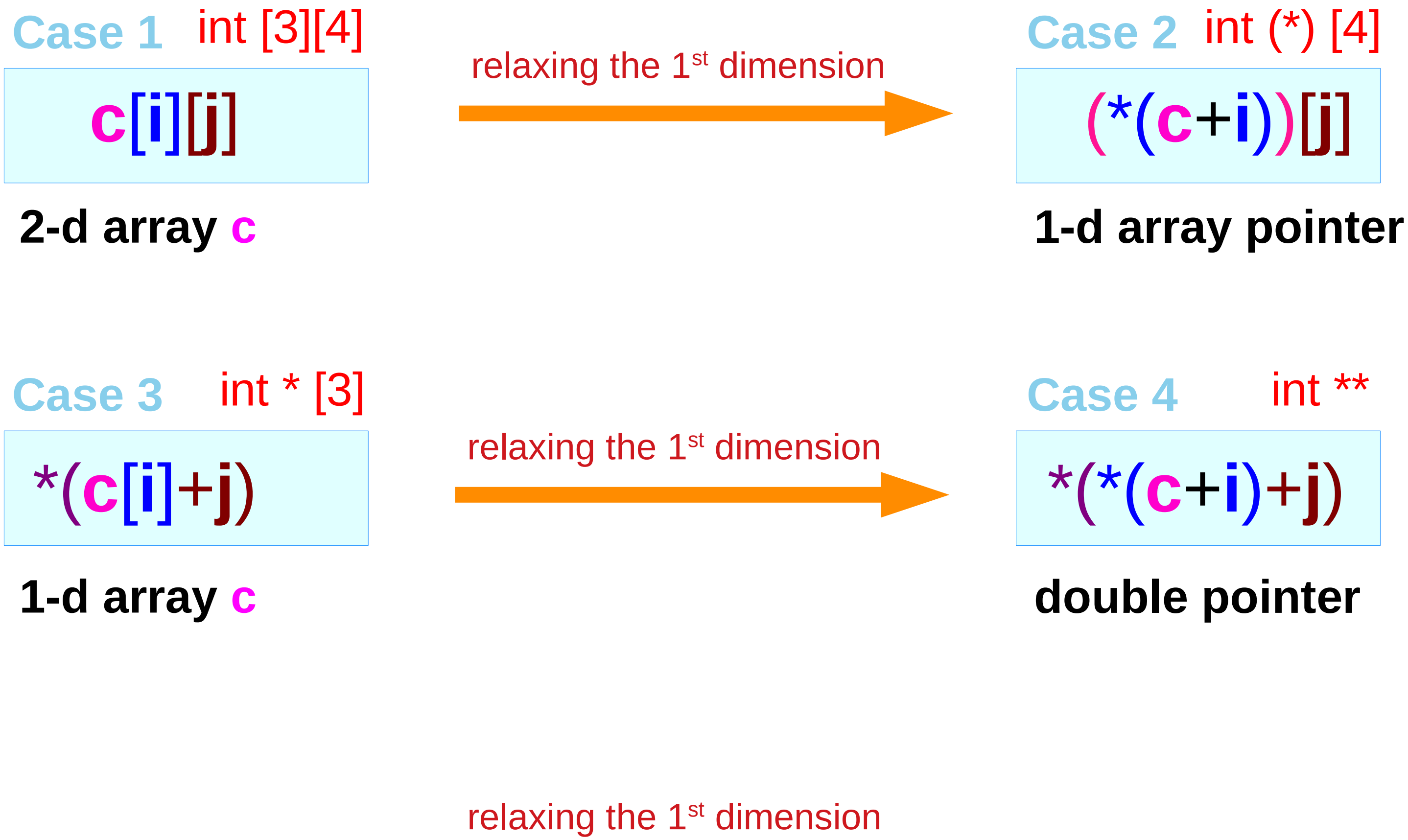
Case 1	**Case 2**	**Case 3**	**Case 4**
int [3][4]	int (*)[4]	int * [3]	int **
	array pointer	**pointer array**	**double pointer**
	c points to an array of 4 integers	c is an array of 3 integer pointers	c points to an integer pointer
c[i][j]	≡ **(*(c+i))[j]**	≡ ***(c[i]+j)**	≡ ***(*(c+i)+j)**
&c[i][j]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

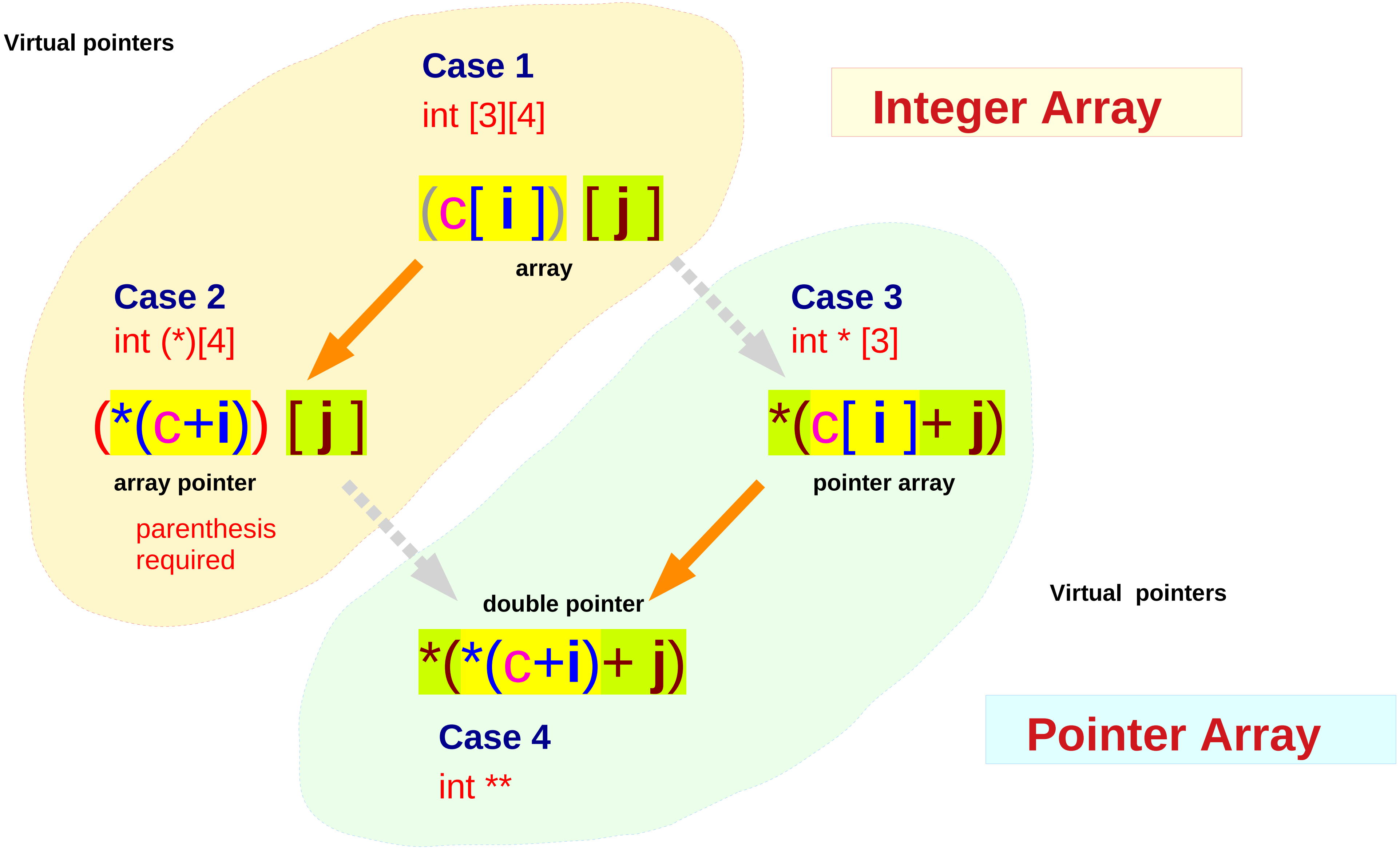
Case 4 int **

`*(*(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

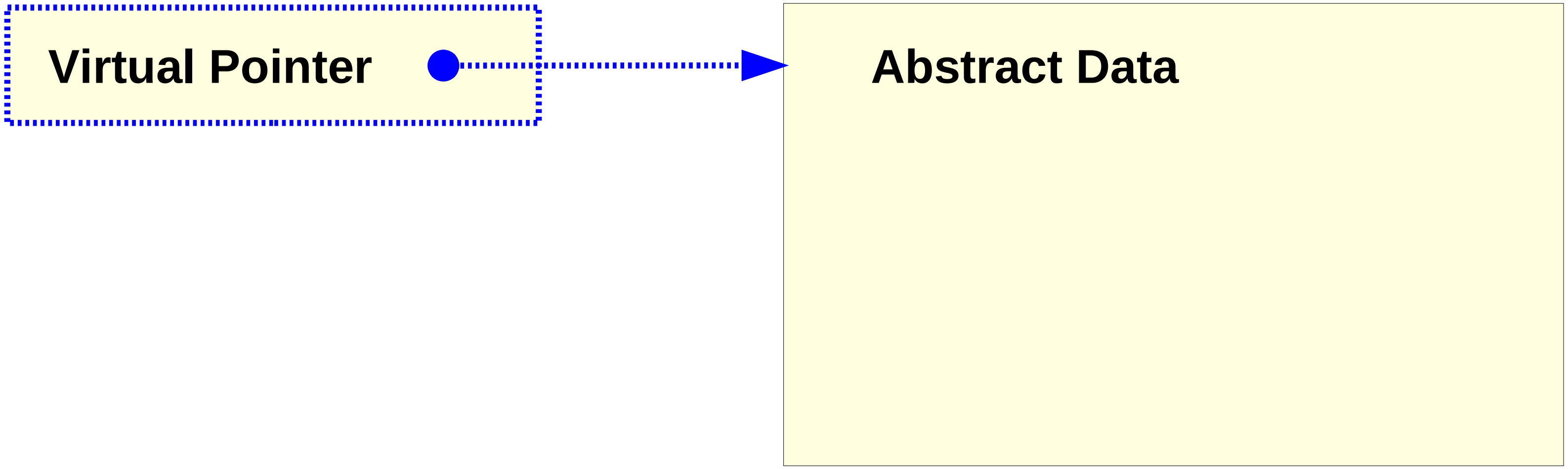
int **

Virtual pointers

Pointer Array

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

Only the 1^{st} dimension can be relaxed

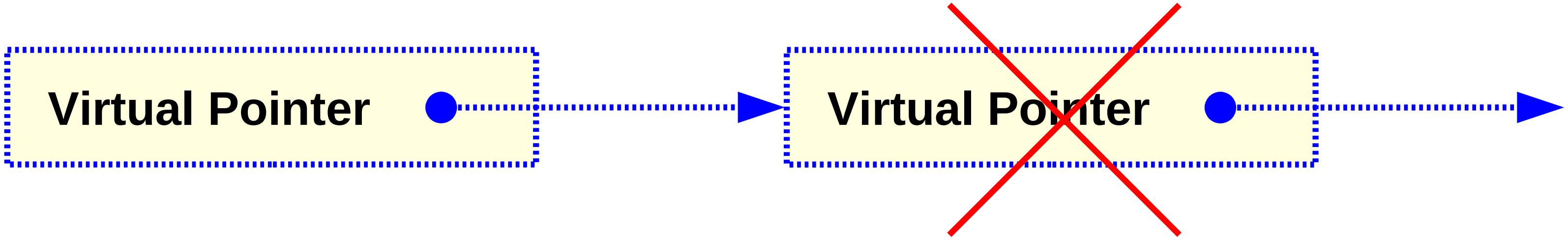


Virtual Pointer

Abstract Data

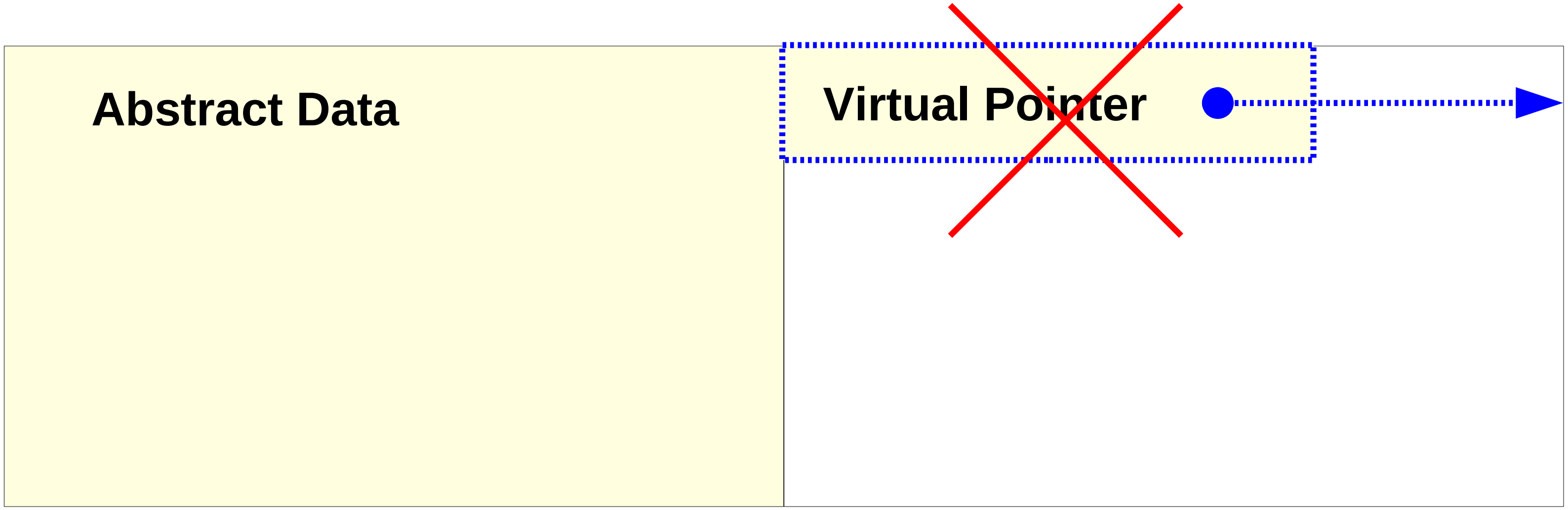
Pointer conversions in **2-d** and **1-d** array types

Only the 1^{st} dimension can be relaxed

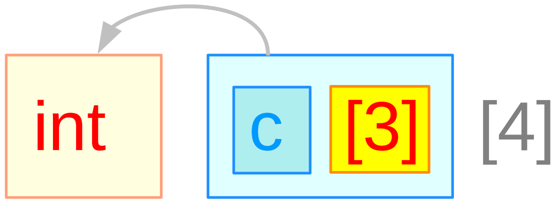


Pointer conversions in **2-d** and **1-d** array types

Only the 1st dimension can be relaxed

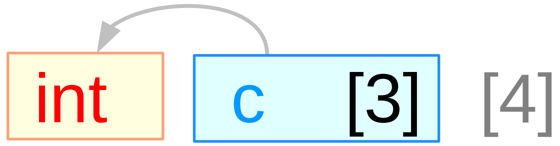


Case 3) 1-d array **c**, pointer **c[i]**



c **1-d array**

type : int * [3]

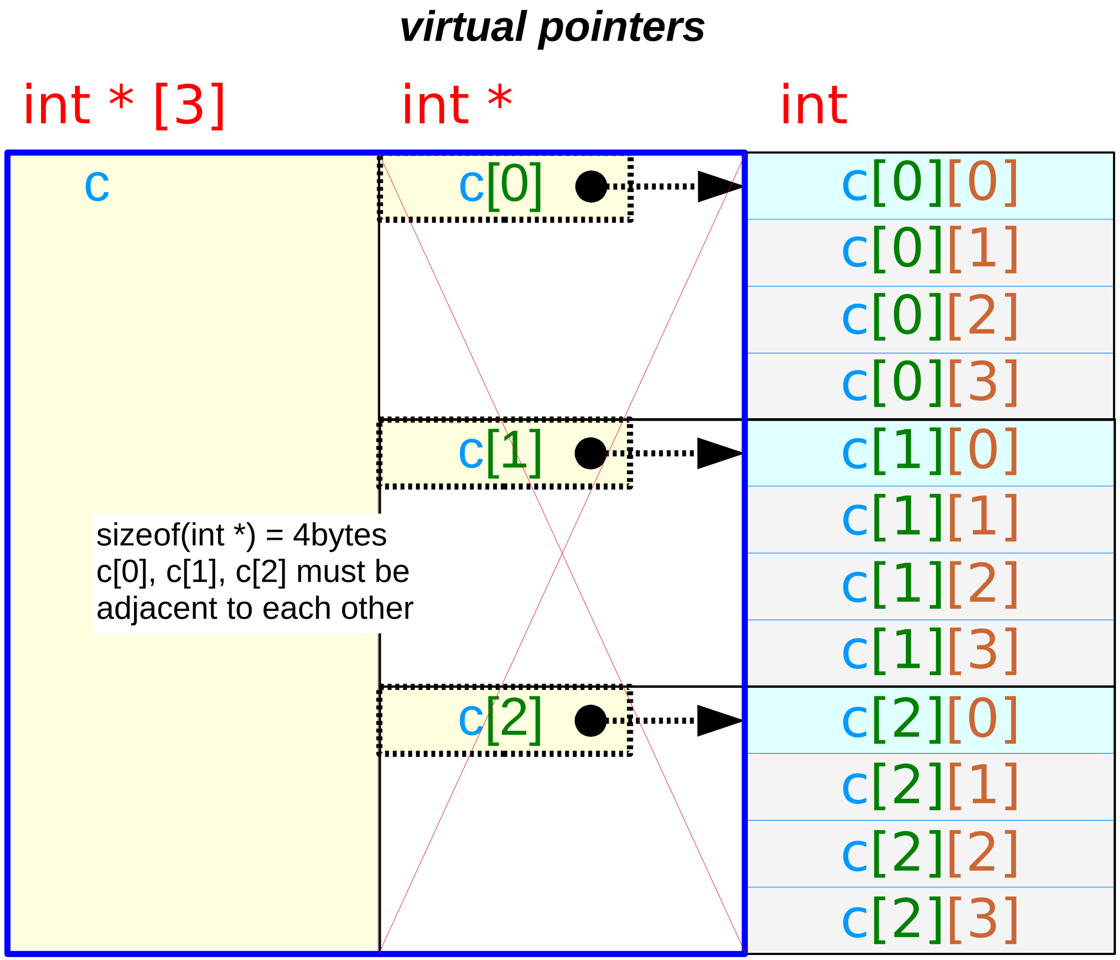


c[i] **pointer**

type : int *

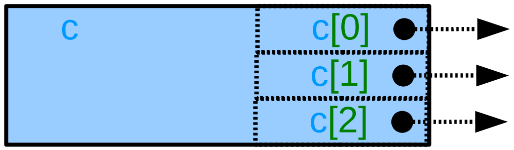
Int pointer

*(c[i]+ j)

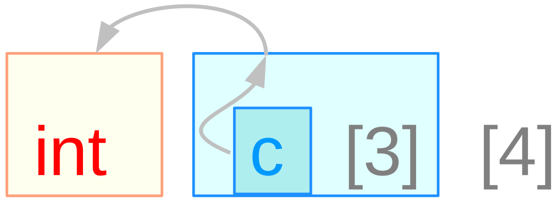


sizeof(int *) = 4bytes
c[0], c[1], c[2] must be
adjacent to each other

c is an array of
3 integer pointers

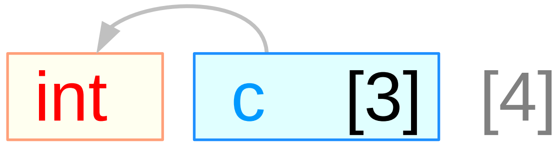


Case 4) double pointer **c**, pointer **c[i]**



c **double pointer**

type : int **

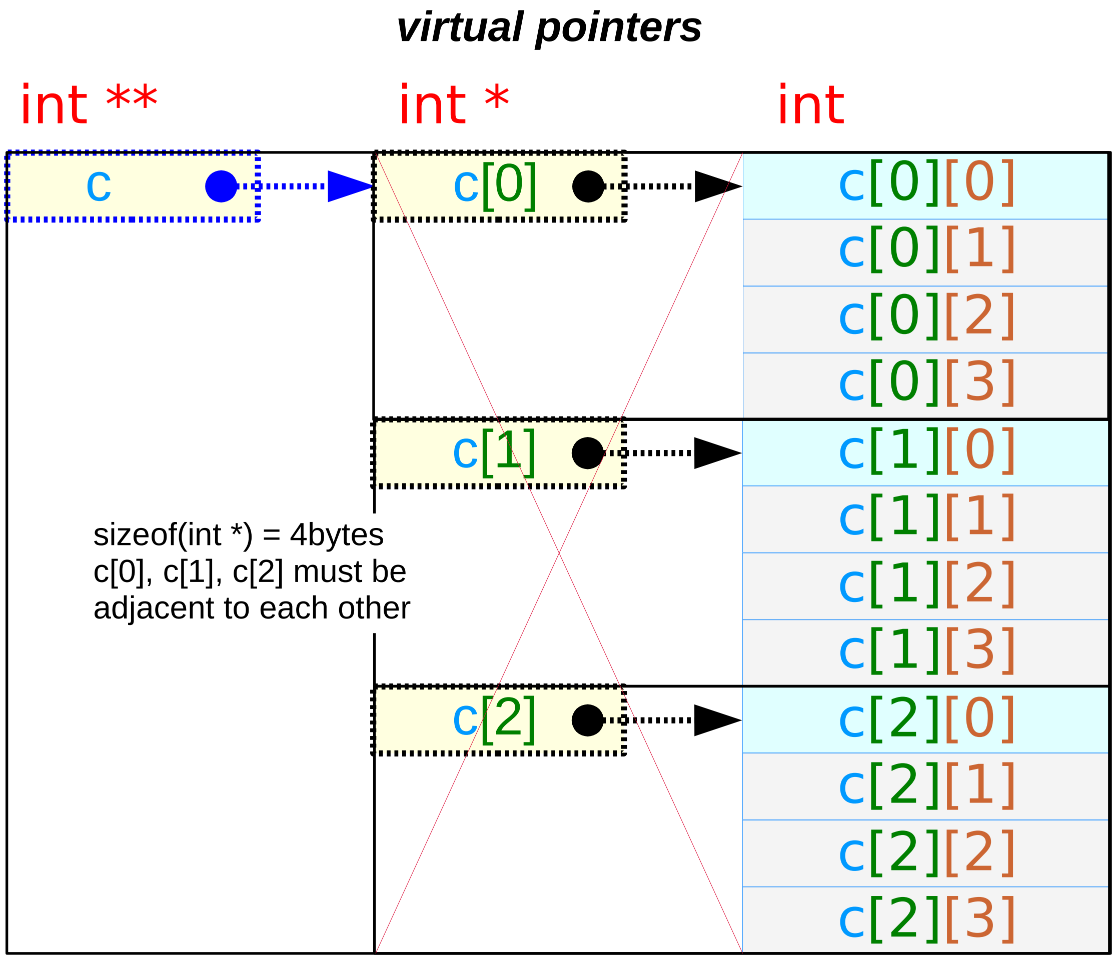


c[i] **pointer**

type : int *

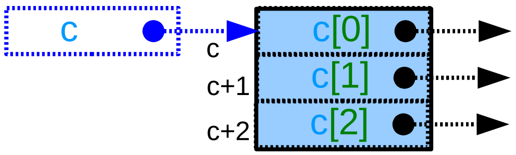
Double pointer

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

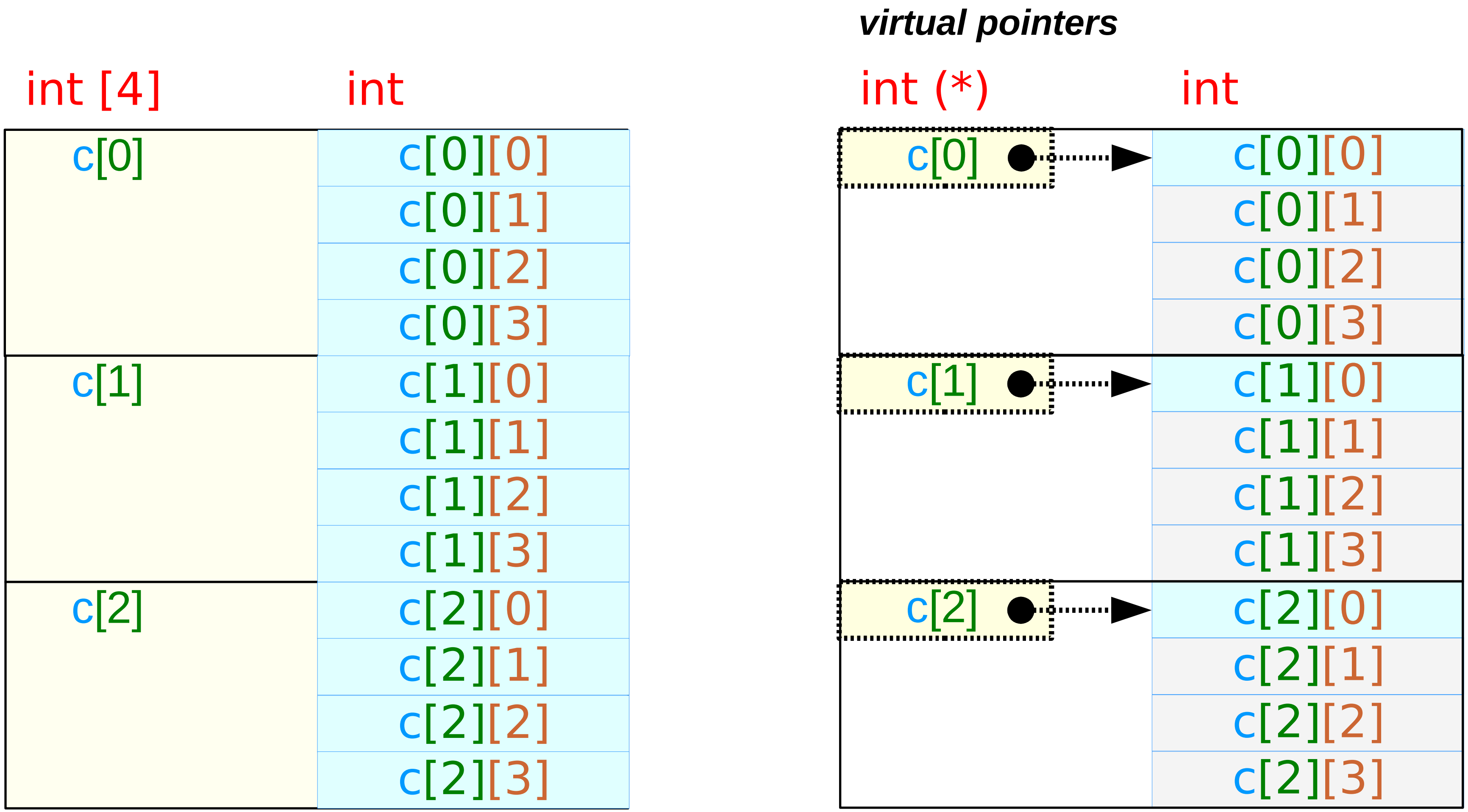


sizeof(int *) = 4bytes
c[0], c[1], c[2] must be
adjacent to each other

c points to an integer pointer



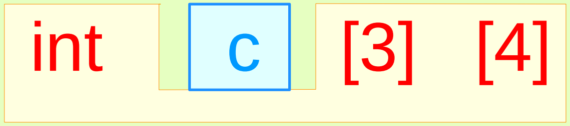
Case



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

c points to an
integer pointer

Types in a **2-d** array



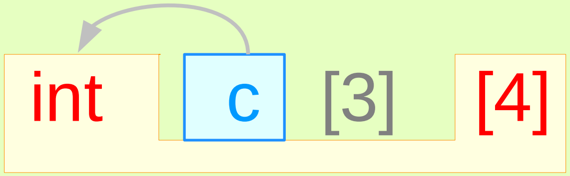
c **2-d array**

type : int [3][4]

size : 3 * 4 * 4

value : &c[0][0]

relaxing the 1st dimension

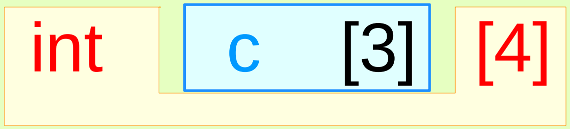


c **1-d array pointer** (virtual)

type : int (*) [4]

size : 3 * 4 * 4

value : &c[0][0]



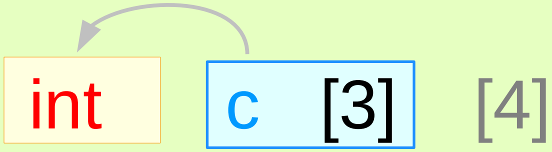
c[i] **1-d array**

type : int [4]

size : 4 * 4

value : &c[i][0]

relaxing the 1st dimension



c[i] **0-d array pointer** (virtual)

type : int (*)

size : 4 * 4

value : &c[i][0]

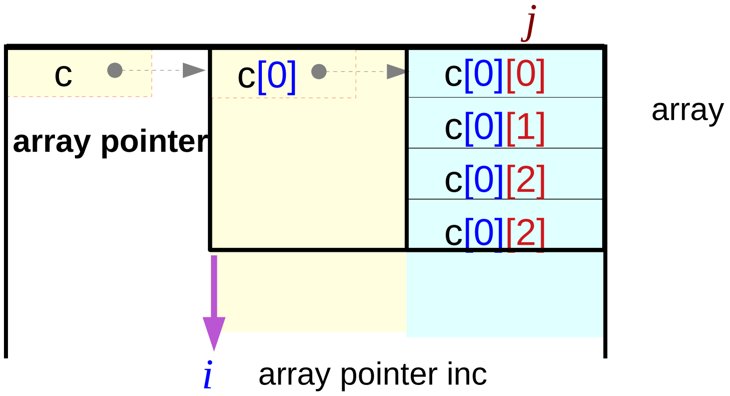
Abstract data and virtual pointer types

Case 2

int (*)[4]

(*(c+i))[j]

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

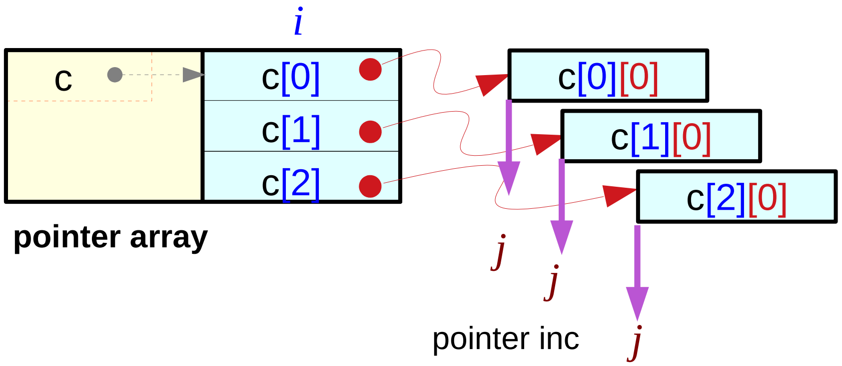


Case 3

int * [3]

*(c[i]+j)

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

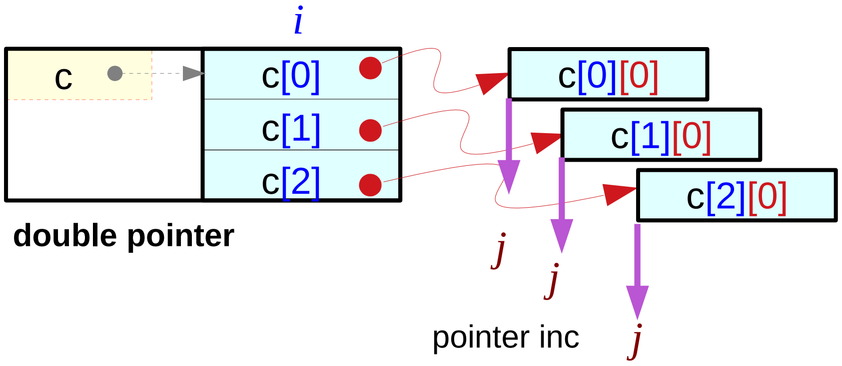


Case 4

int **

((c+i)+j)

c points to
an integer pointer



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

int c [3] [4]

c **2-d array**

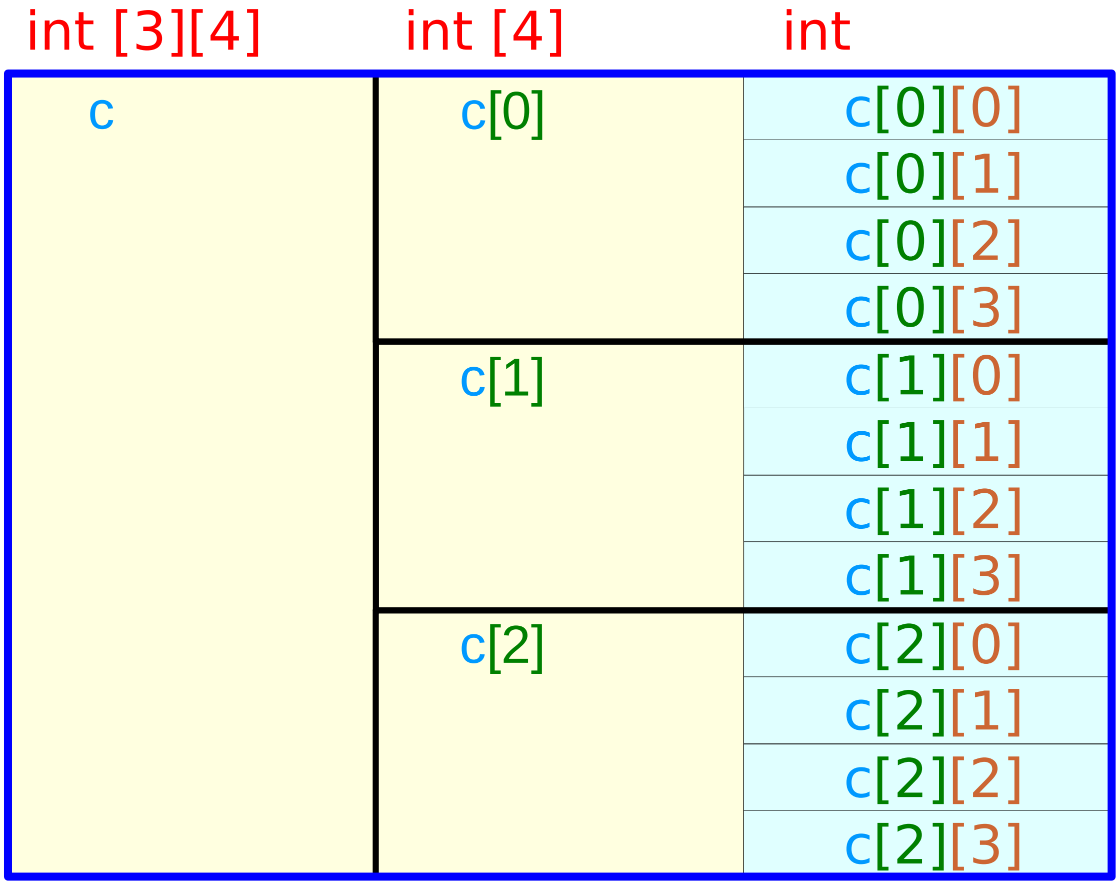
type : int [3][4]

int c [3] [4]

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

type : int [4]

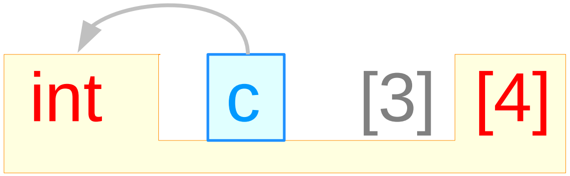
Abstract Data



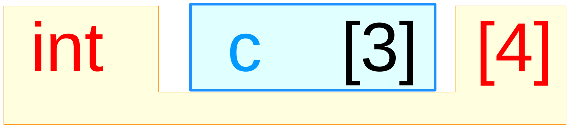
(c[i]) [j]

2-d array c

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



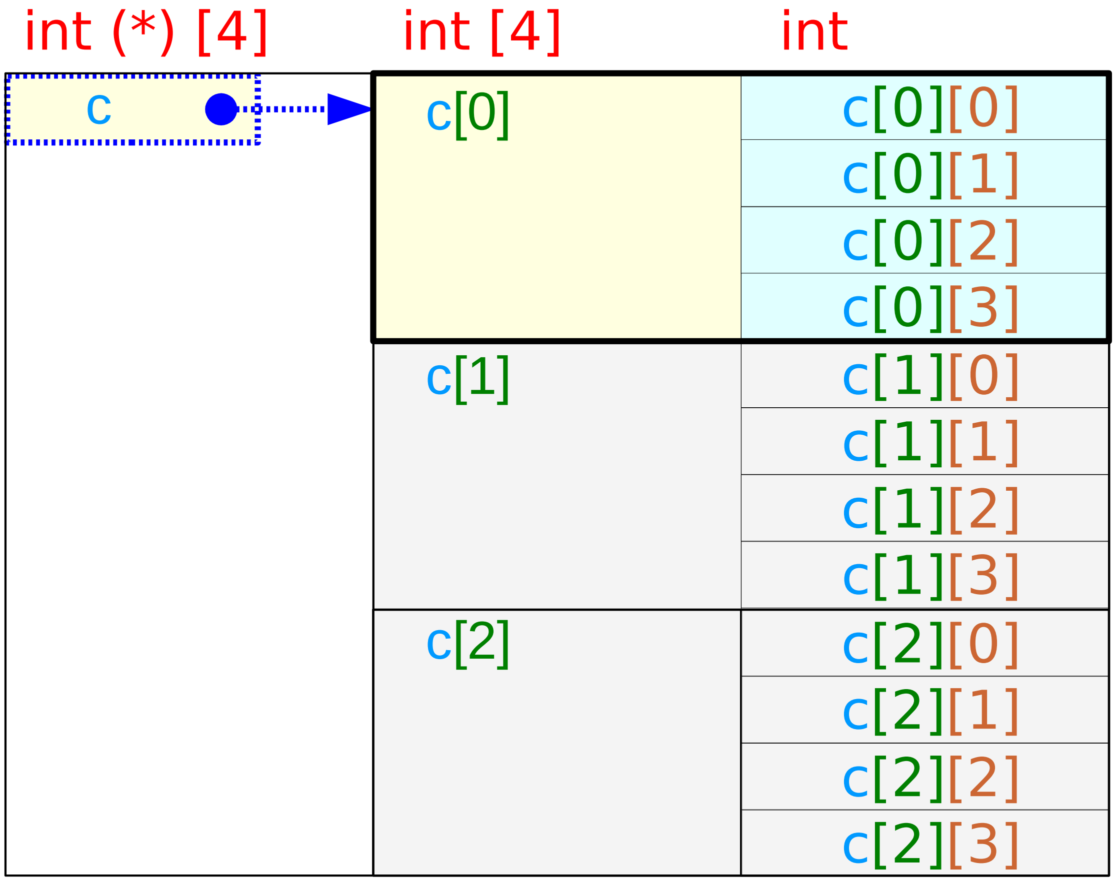
c **1-d array pointer**
type : int (*) [4]



c[i] **1-d array**
type : int [4]

Abstract Data

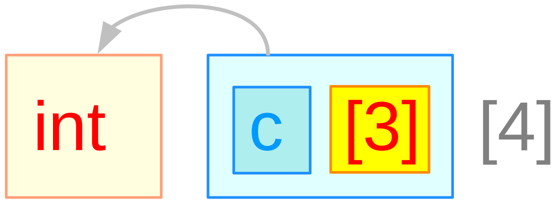
(*(c+i)) [j]



c points to an array
of 4 integers

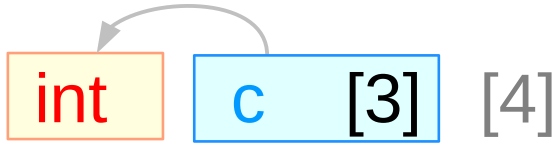
2-d array c

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



c **1-d array**

type : int * [3]

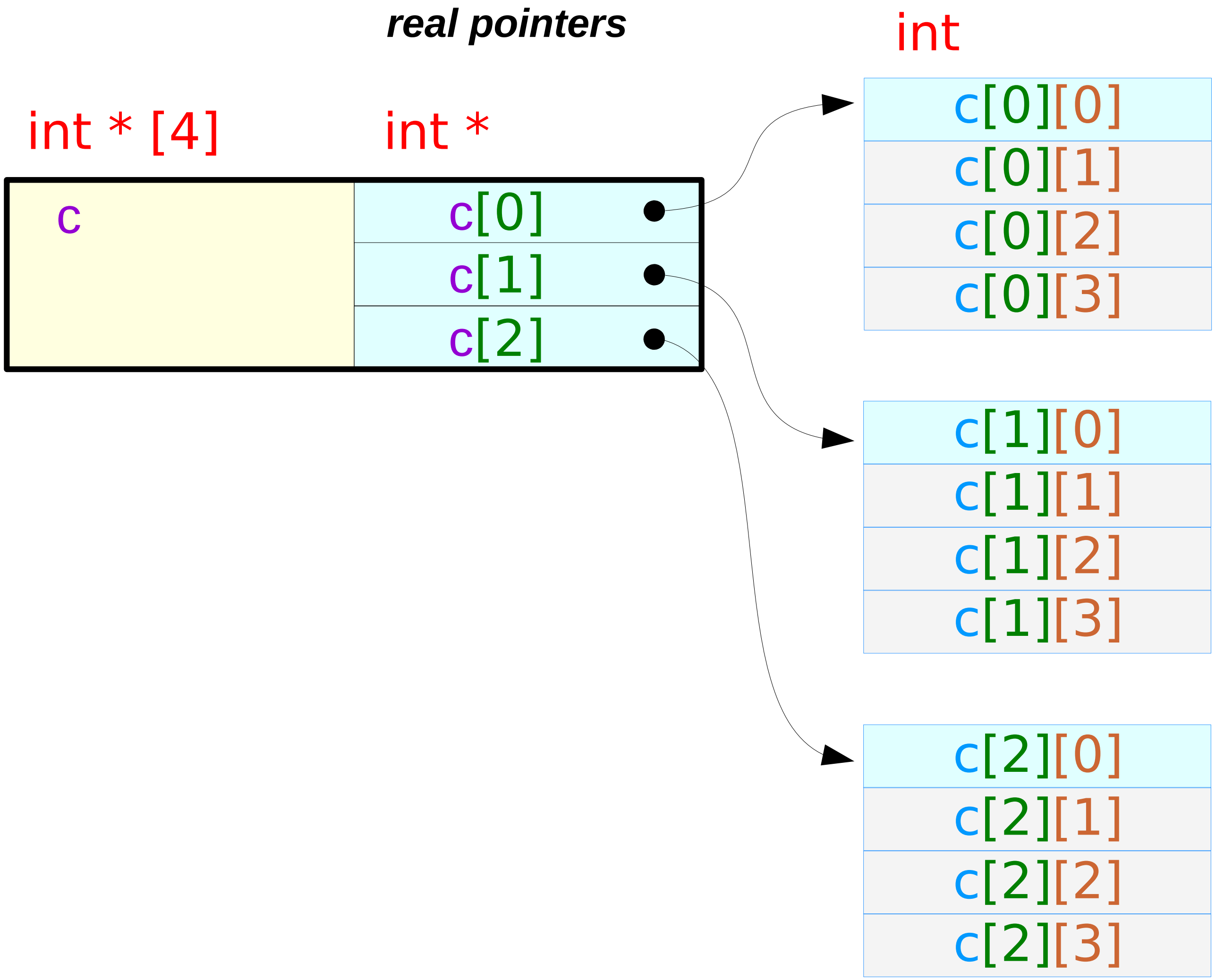


c[i] **pointer**

type : int *

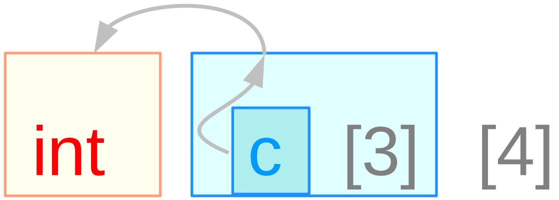
Int pointer

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

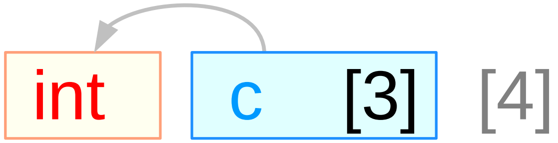


1-d array c

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

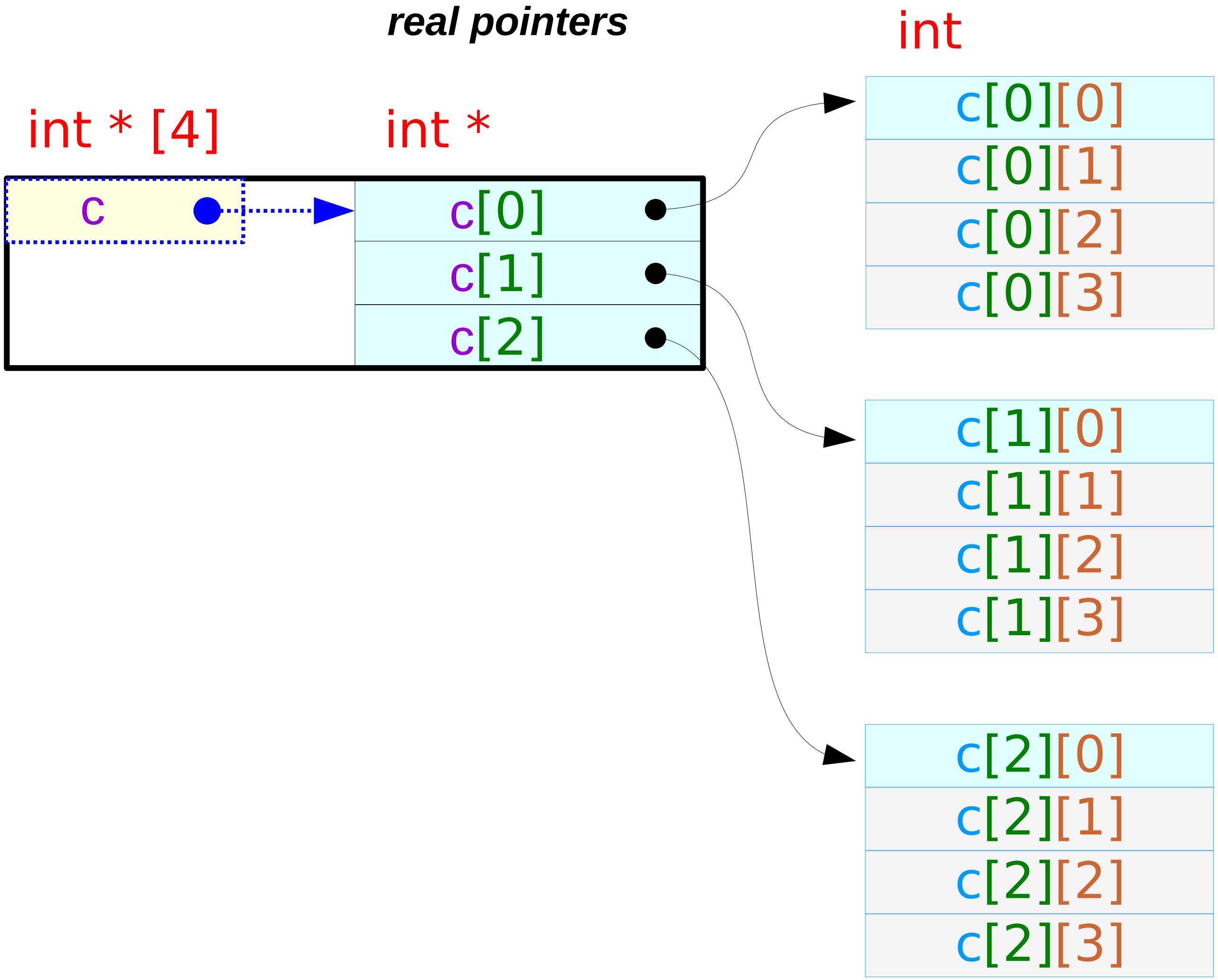


c **double pointer**
type : int **



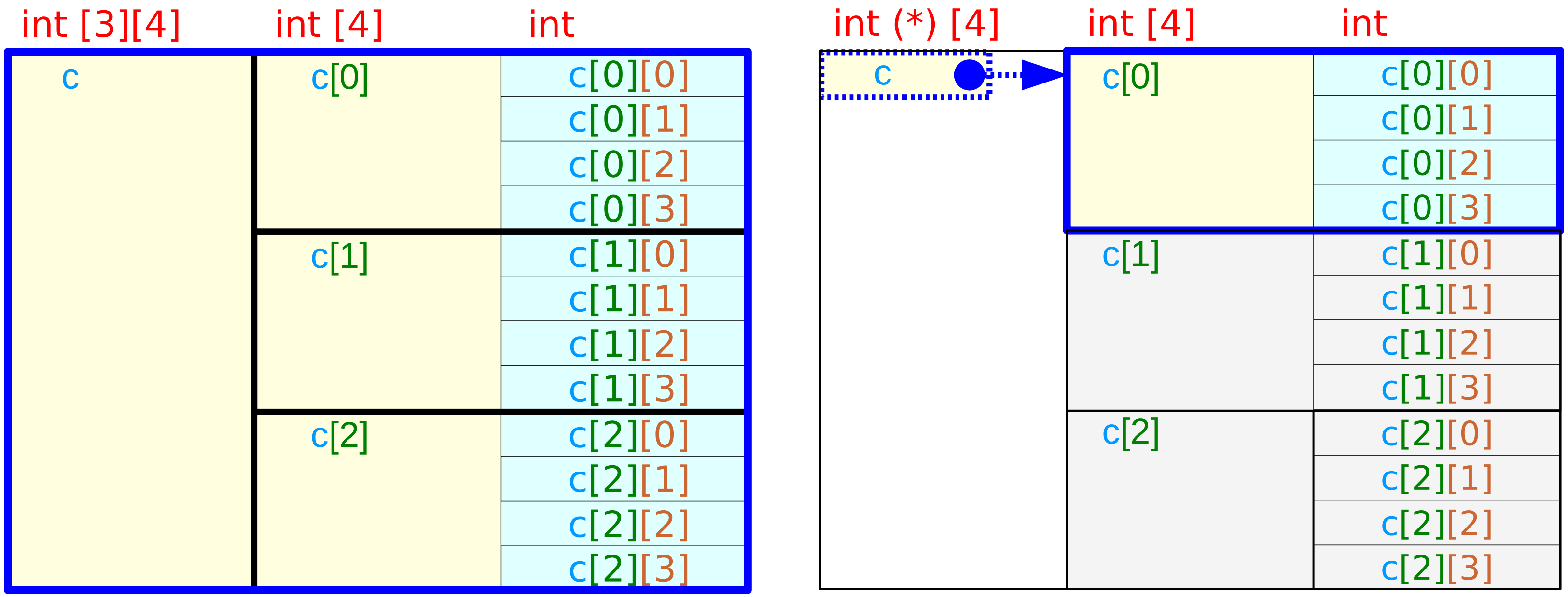
c[i] **pointer**
type : int *

Double pointer



1-d array c

Nested Structure – abstract data, virtual pointer



Case 1

(c[i]) [j]

→

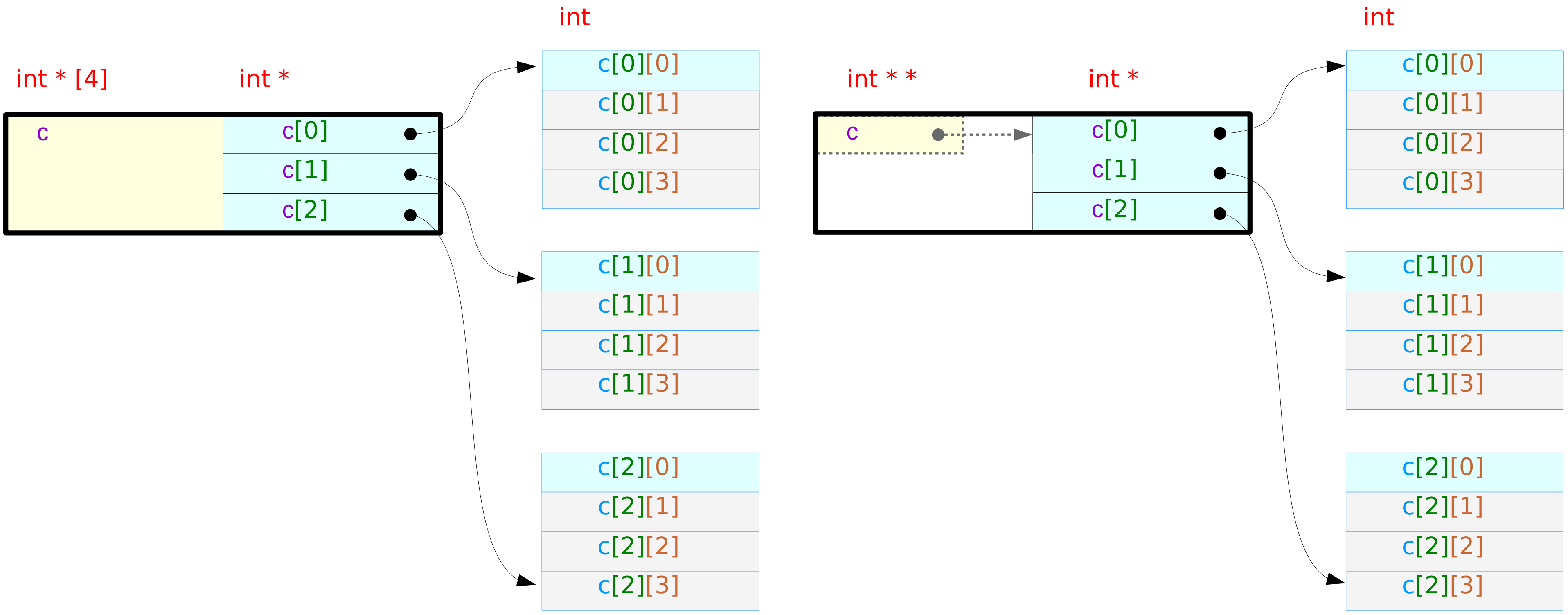
Case 2

(*(c+i)) [j]

nested structure

2-d array c

Nested structure – abstract data, virtual pointer



Case 3

Case 4

***(c[i]+ j)** → ***(*(c+i)+ j)**

nested structure

1-d array c

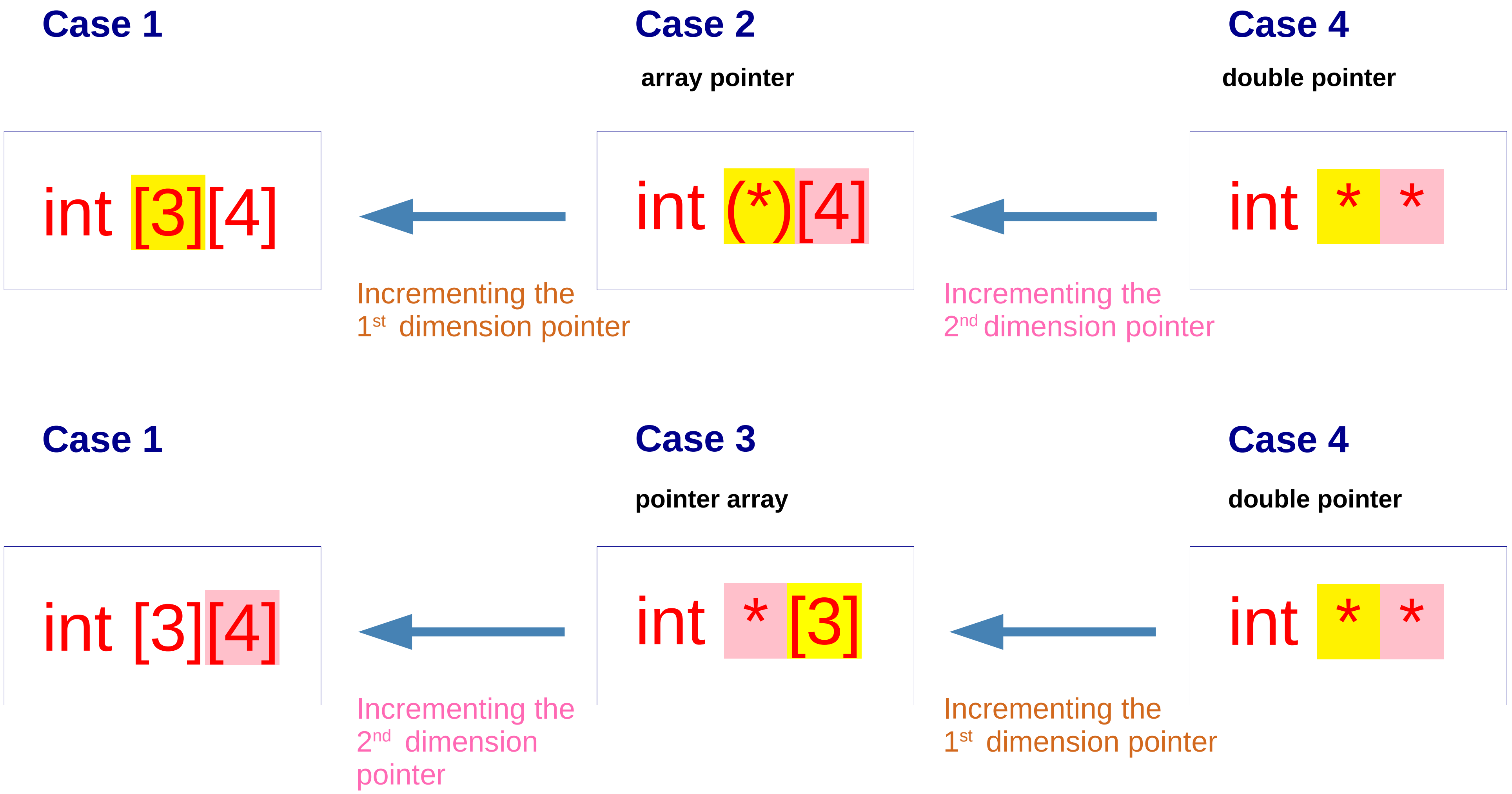
- Pointer conversions in array types
- Simulating array accesses by real pointers
- Dynamic memory allocation

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]

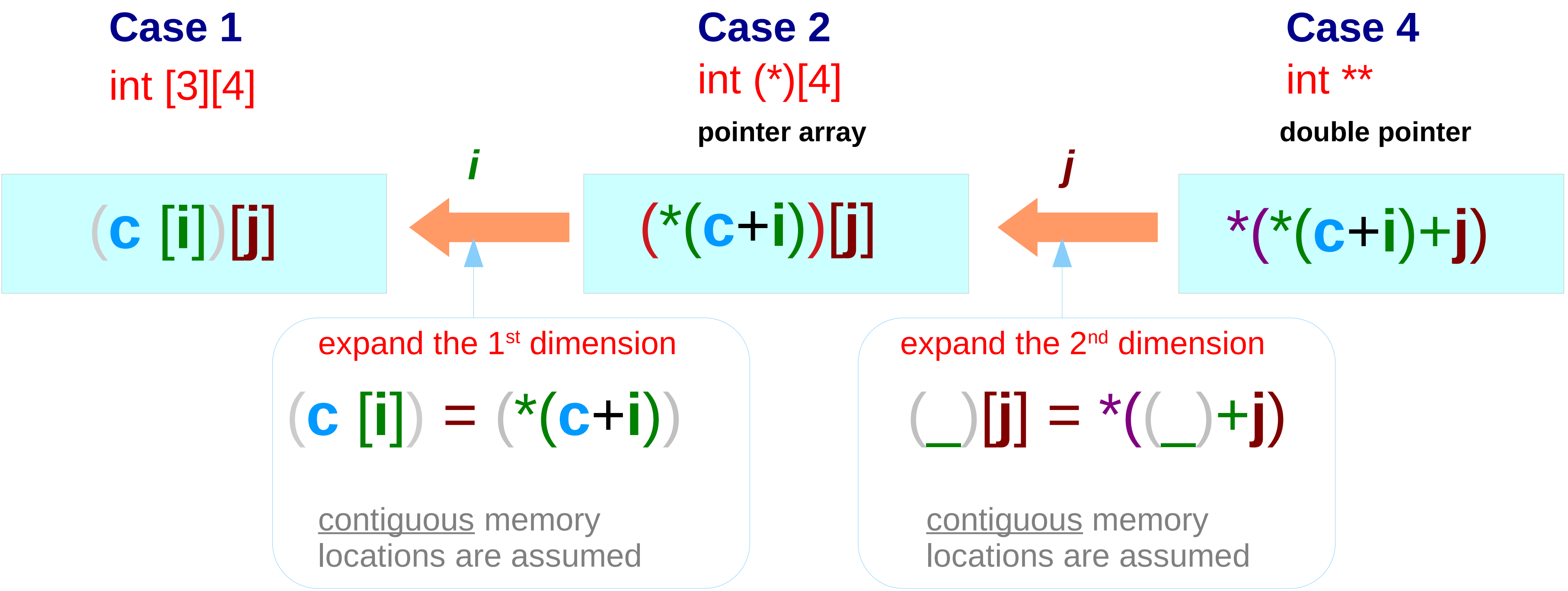
Incrementing the 1st dimension pointer

Case 4

double pointer

int * *

2-d array access via a double indirection



Case 1

int [3][4]

`(c [i])[j]`

Case 2

int (*)[4]

pointer array

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

Case 4

int **

double pointer

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

expand the 1st dimension

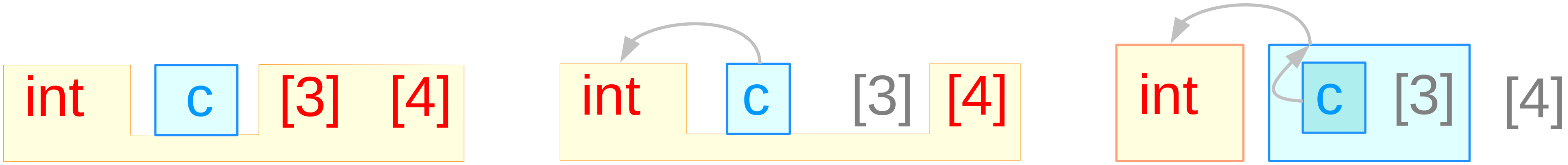
`(c [i]) = (*(c+i))`

contiguous memory locations are assumed

expand the 2nd dimension

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

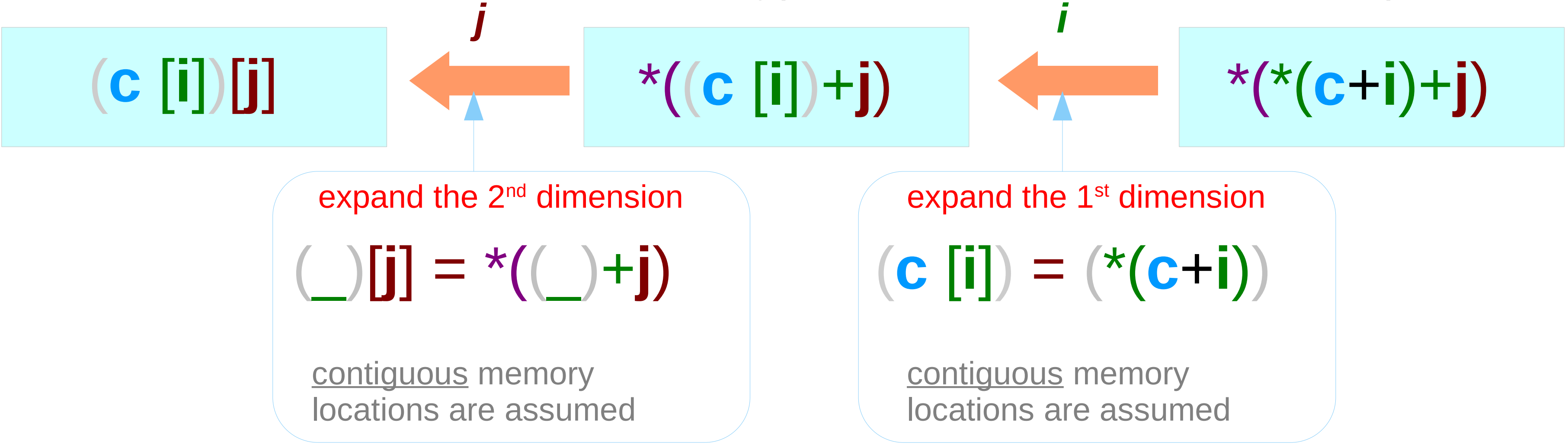
contiguous memory locations are assumed



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]

(c [i])[j]

Case 3

int * [3]

array pointer

*((c [i])+j)

Case 4

int **

double pointer

((c+i)+j)

expand the 2nd dimension

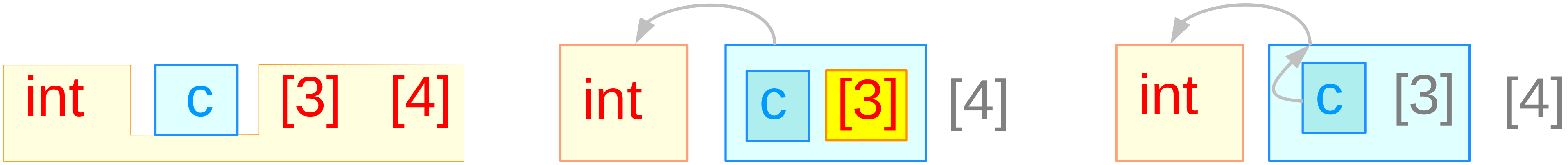
(_)[j] = *((_)+j)

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

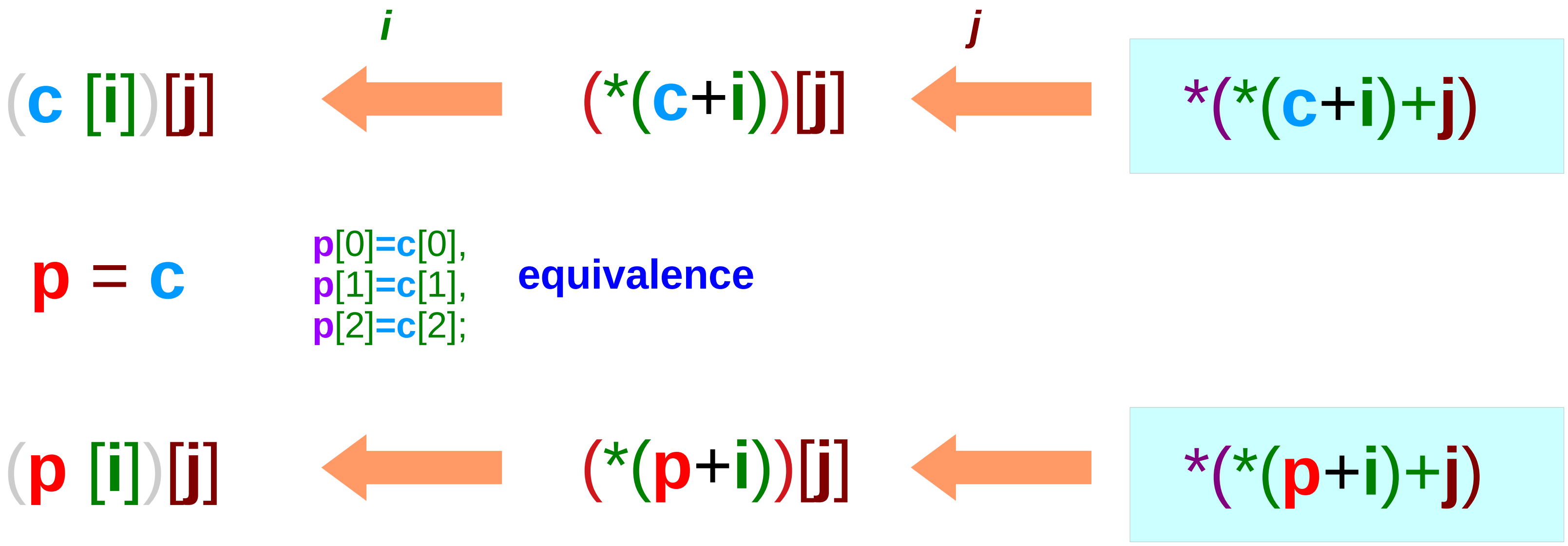
int [3][4]

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];

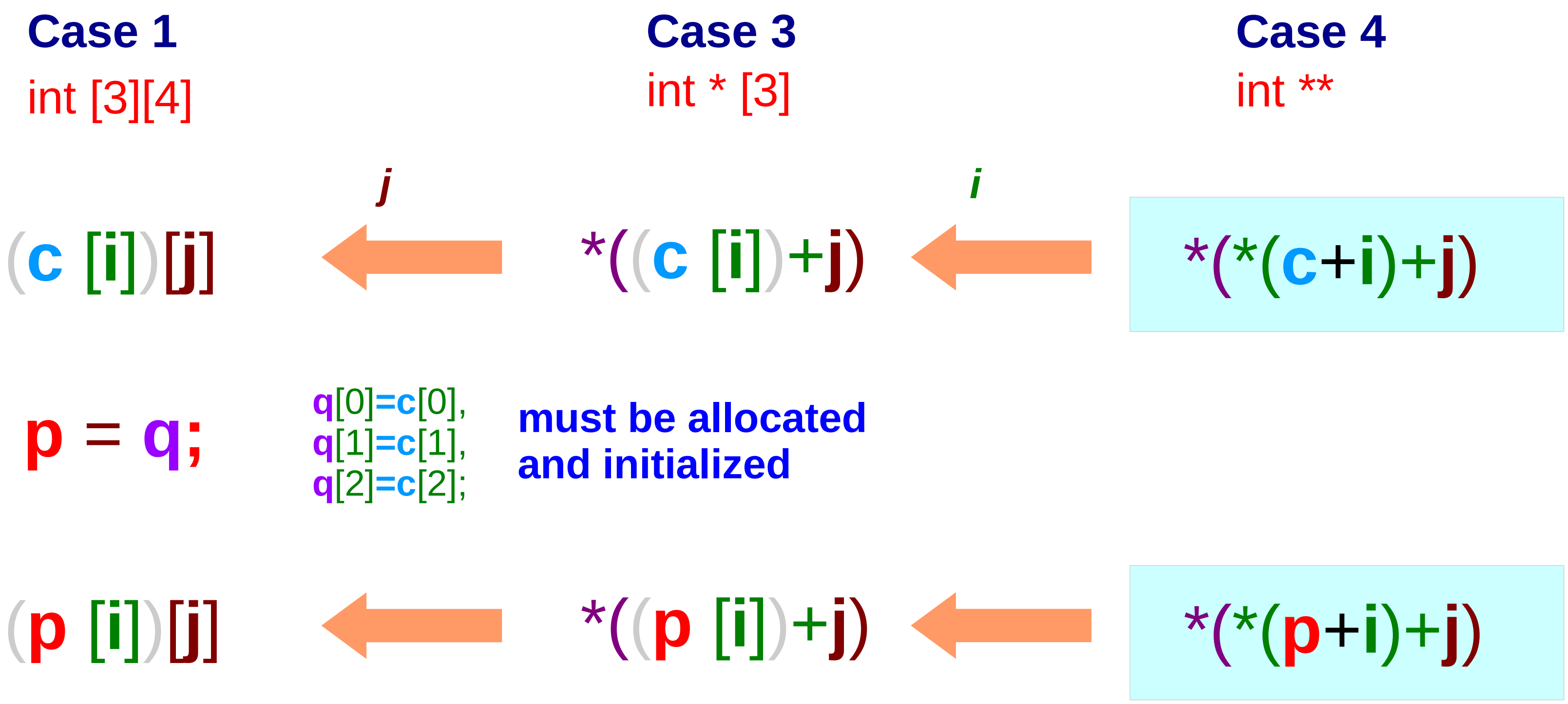
equivalence

`(p [i])[j]` ← `(*(p+i))[j]` ← `*(*(p+i)+j)`

Cases 1, 3, 4

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

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



Case 1

int [3][4]

Case 3

int * [3]

Case 4

int **

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

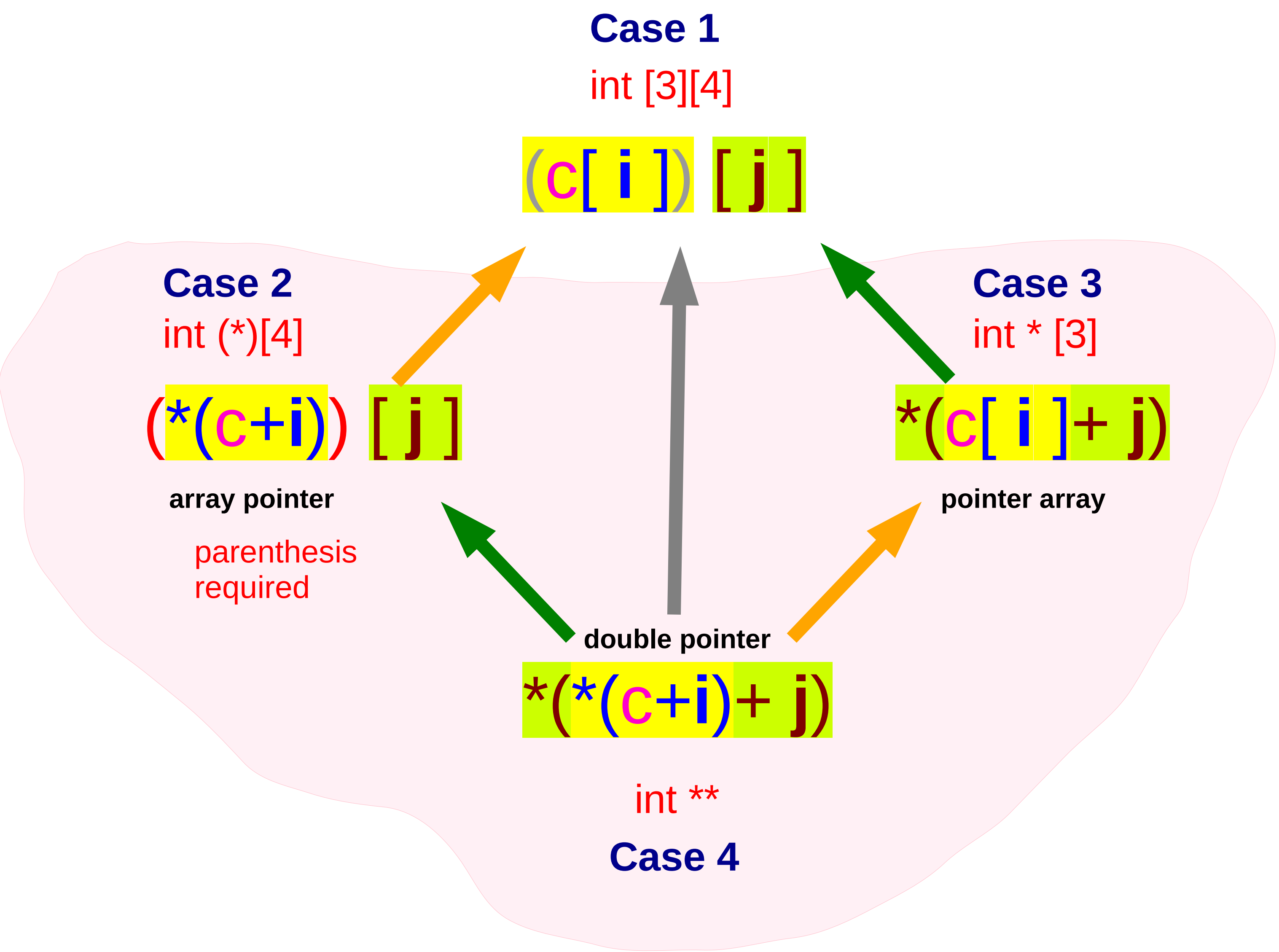
p = q;

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



Case 1

int [3][4]

`(c[ i ]) [ j ]`

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)`

int **

Case 4

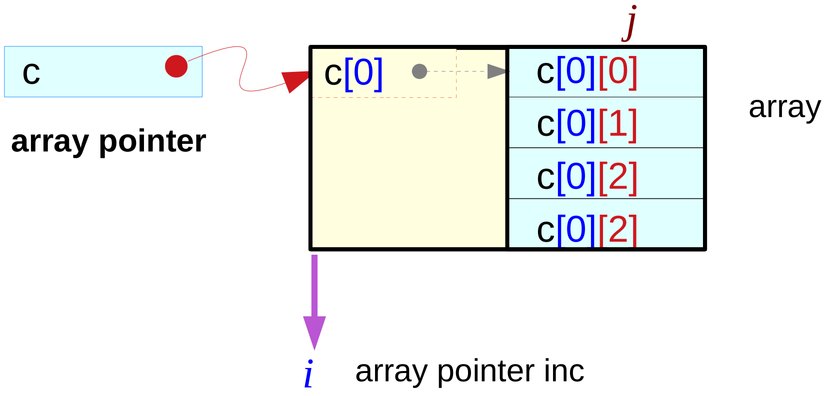
Incrementing pointers

Case 2

(*(c+i))[j]

int (*)[4]

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

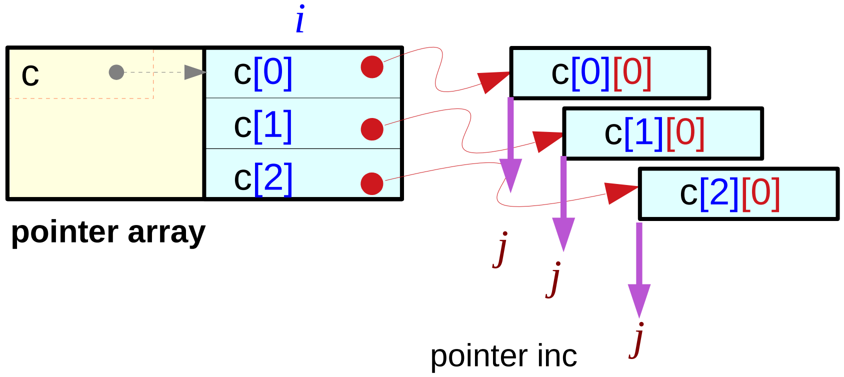


Case 3

*(c[i]+j)

int * [3]

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

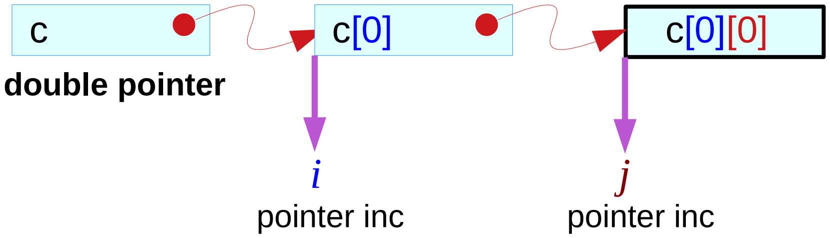


Case 4

((c+i)+j)

int **

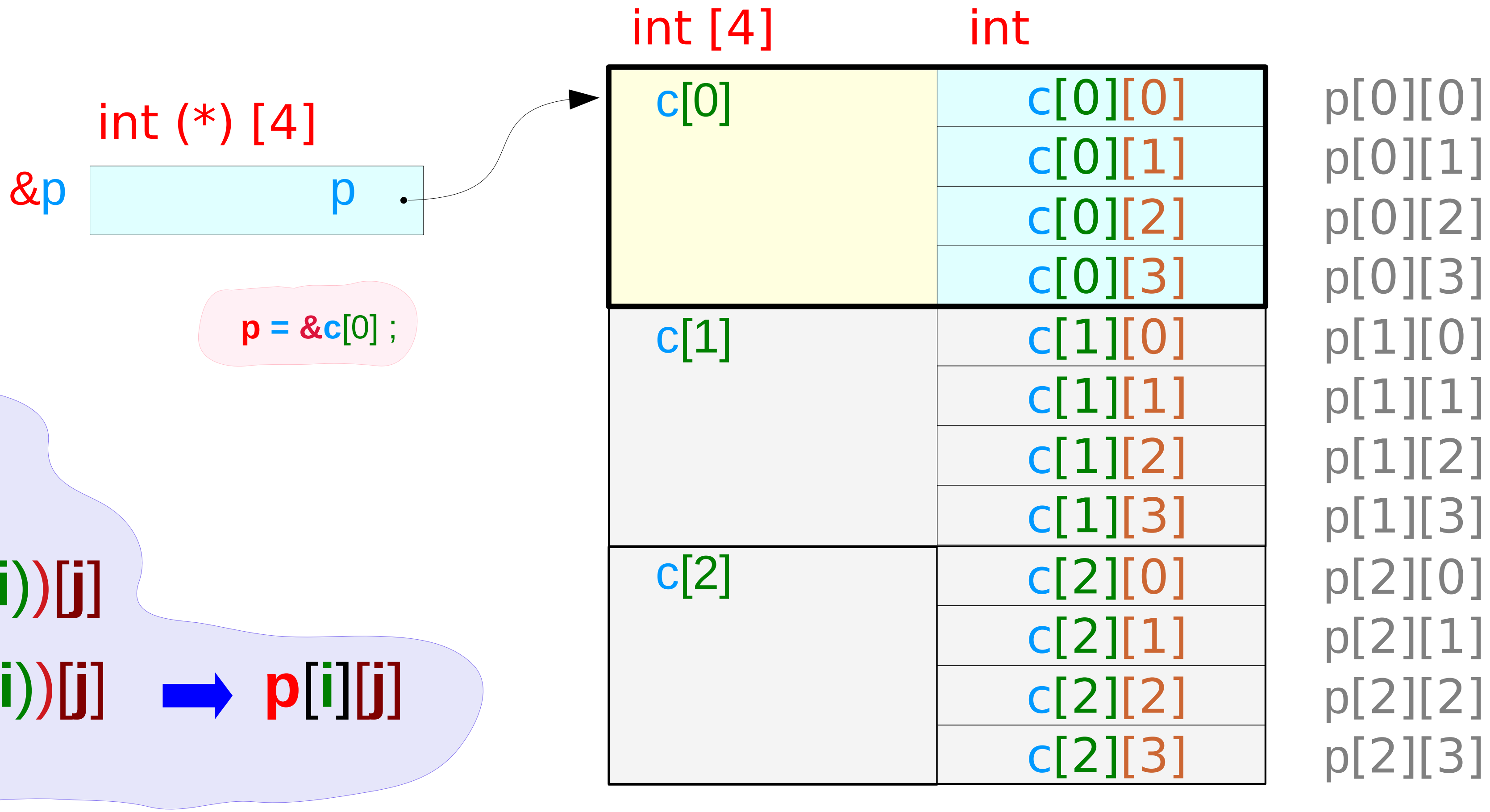
c points to
an integer pointer



2-d array access using an array pointer **p**

`int c [3] [4];`

`int (*p) [4];`



`p = &c[0] ;`

Case 2

int (*)[4]

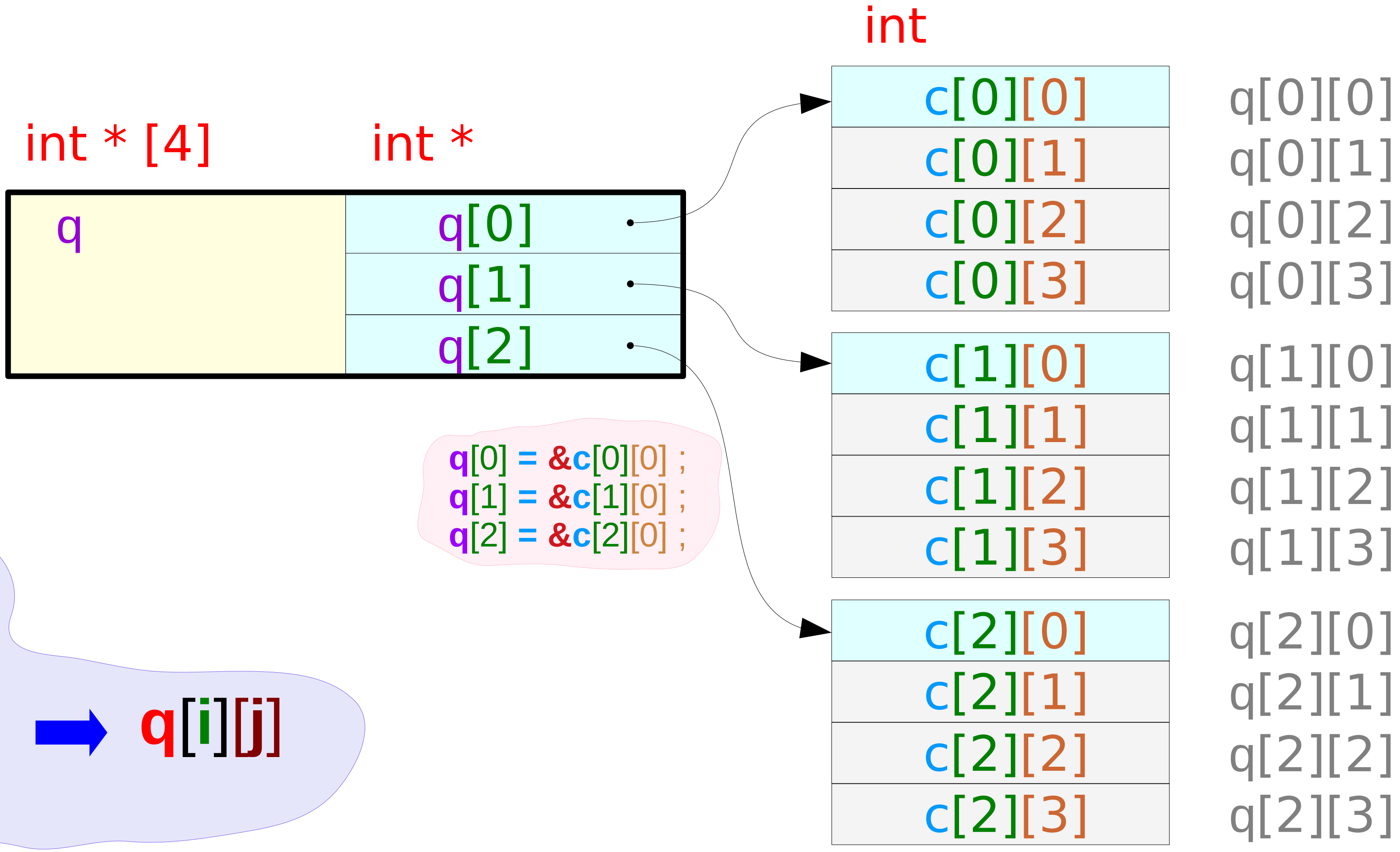
`(*(c+i))[j]`

`(*(p+i))[j]` ➡ `p[i][j]`

2-d array access using a pointer array **q**

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

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



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

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

Case 3

int * [3]

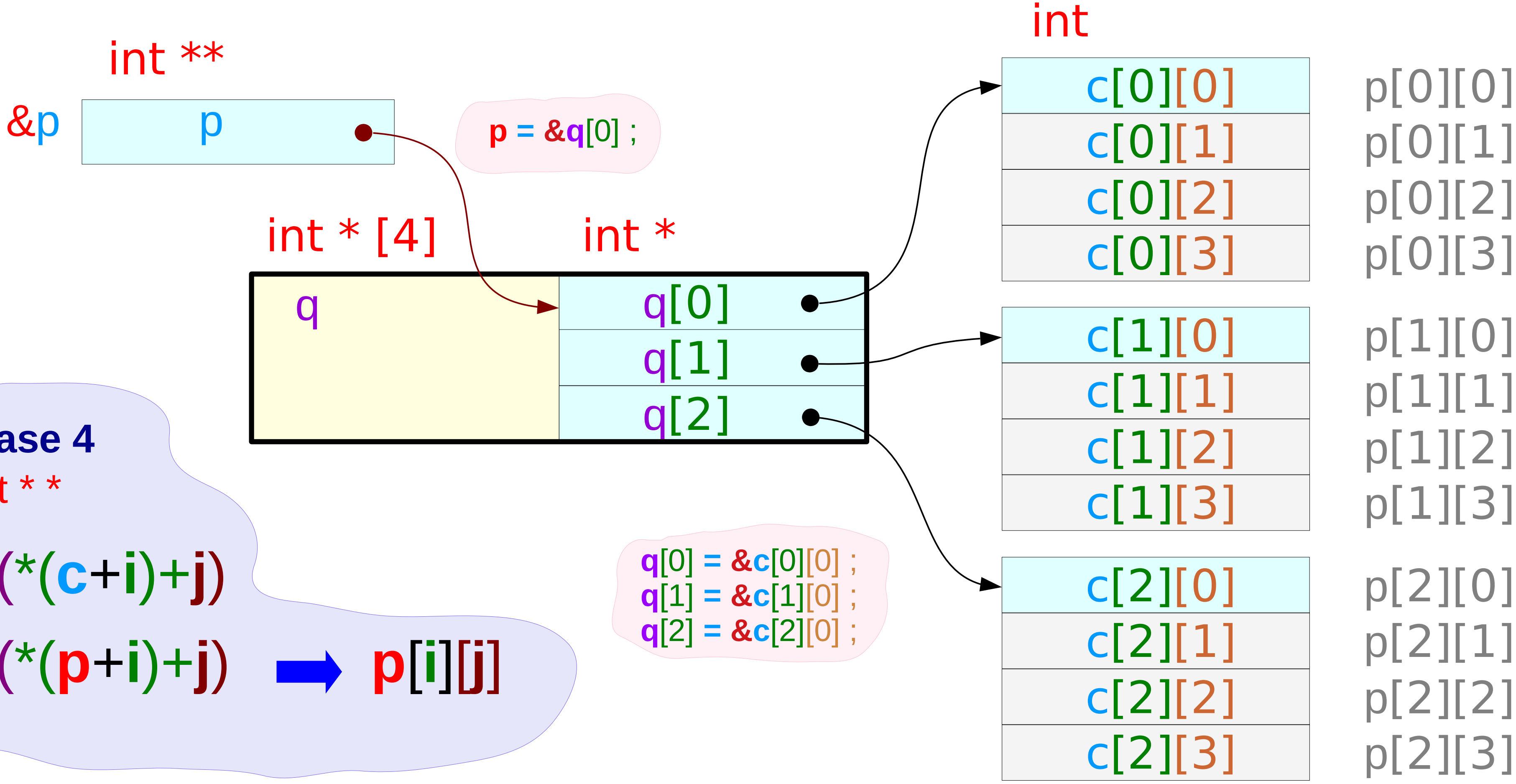
*((c [i])+j)

*((q [i])+j) ➡ q[i][j]

2-d array access using double pointers **q**

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

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



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

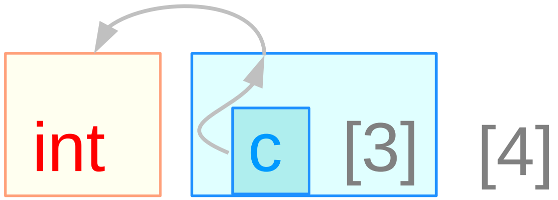
Case 4

int * *

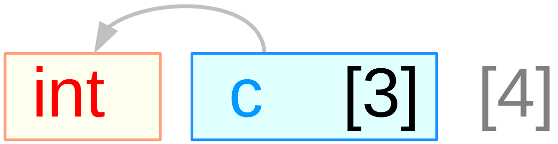
((c+i)+j)

((p+i)+j) ➡ p[i][j]

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



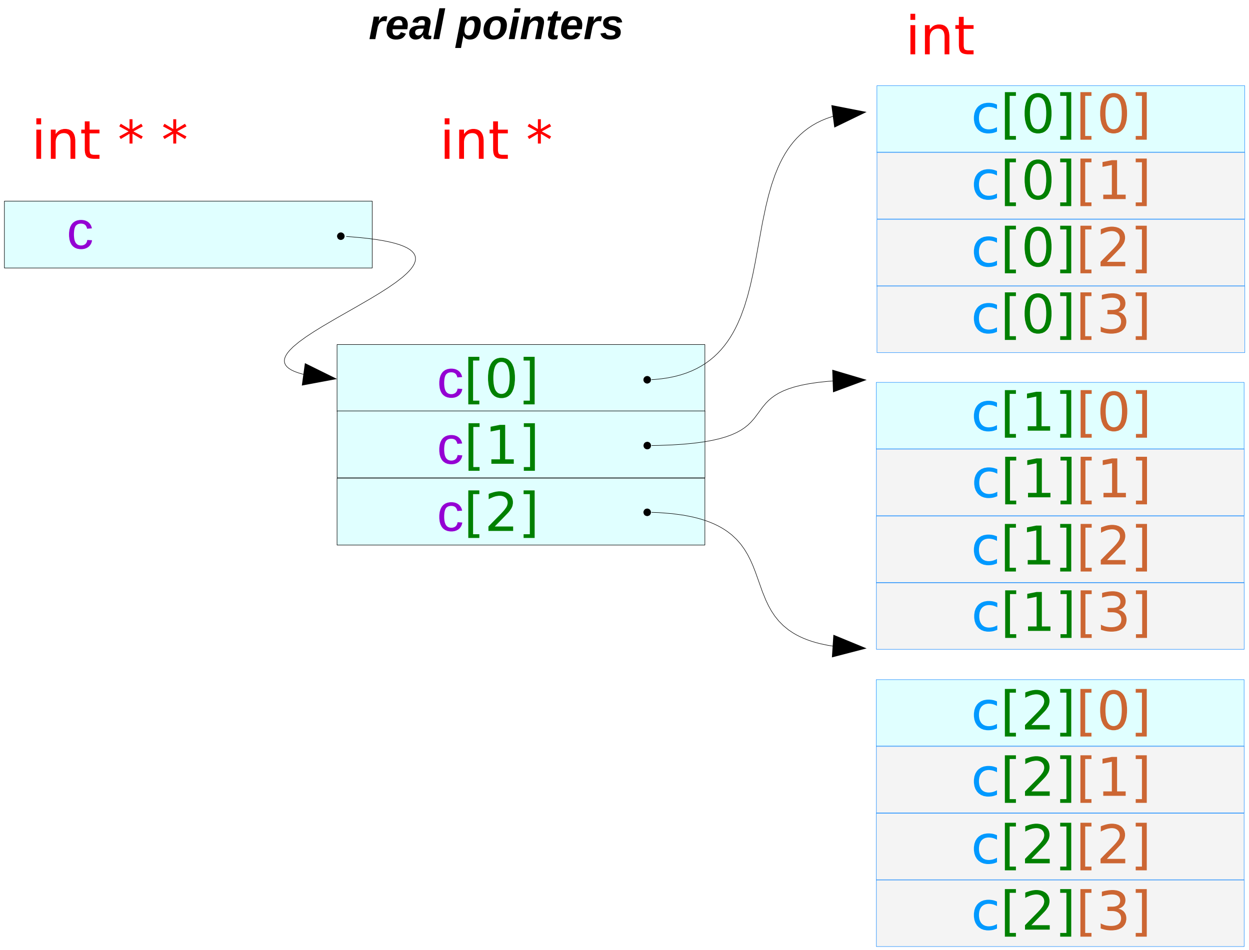
c **double pointer**
type : int **



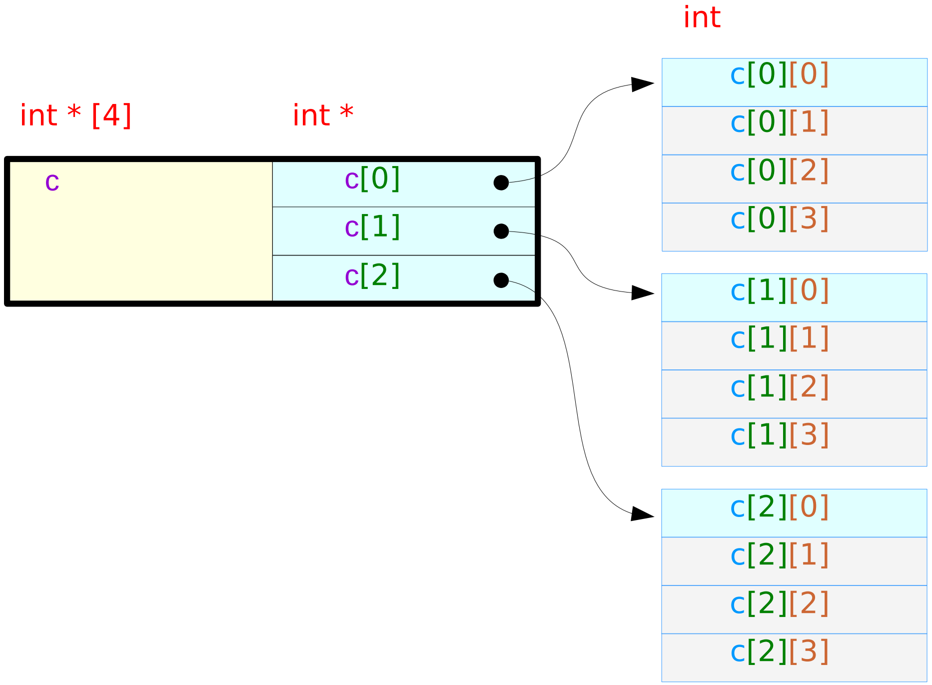
c[i] **pointer**
type : int *

Double pointer

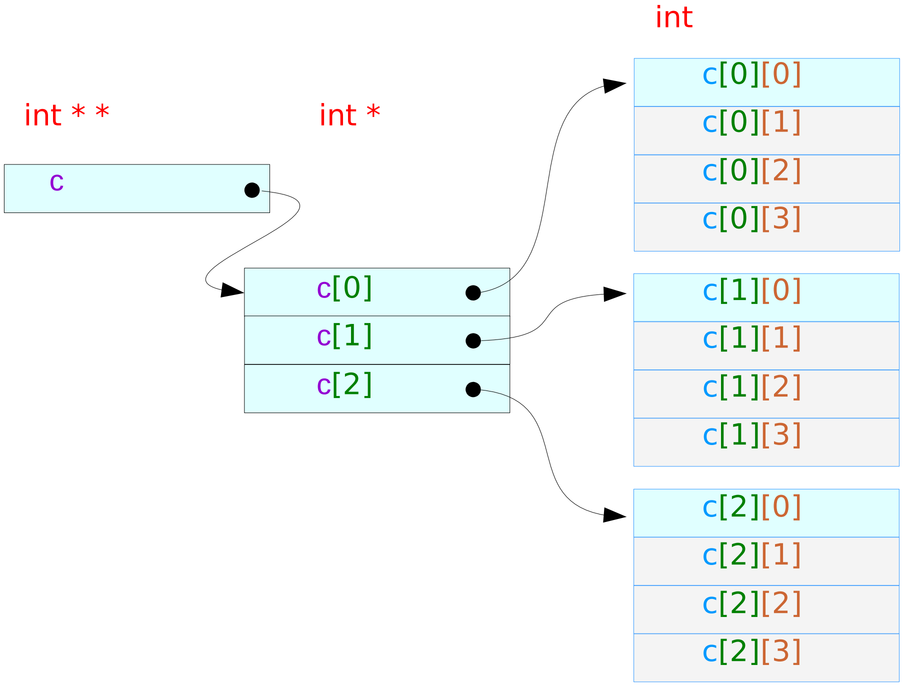
((c+i)+ j)



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)) ;
```

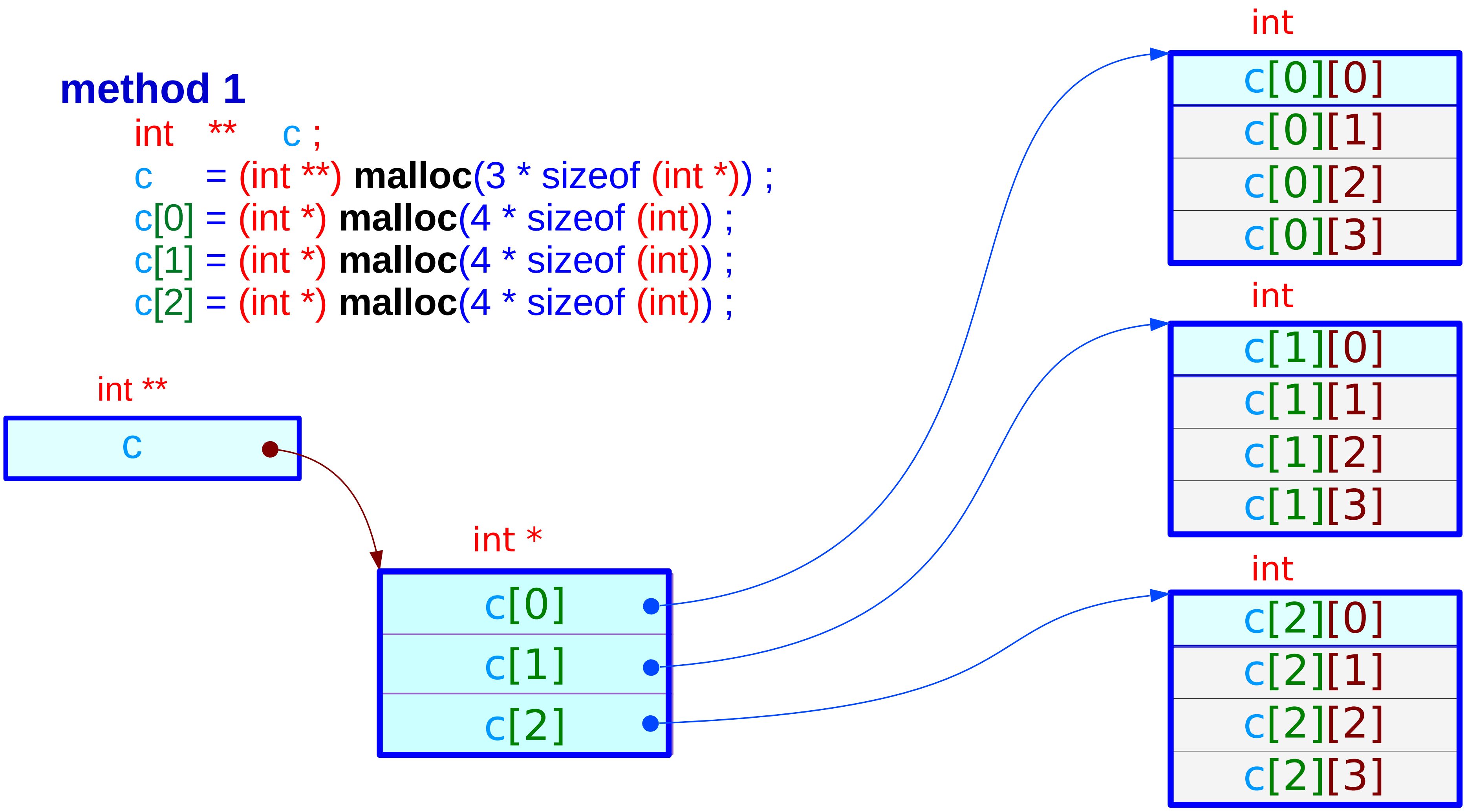
Case 2

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)) ;
```



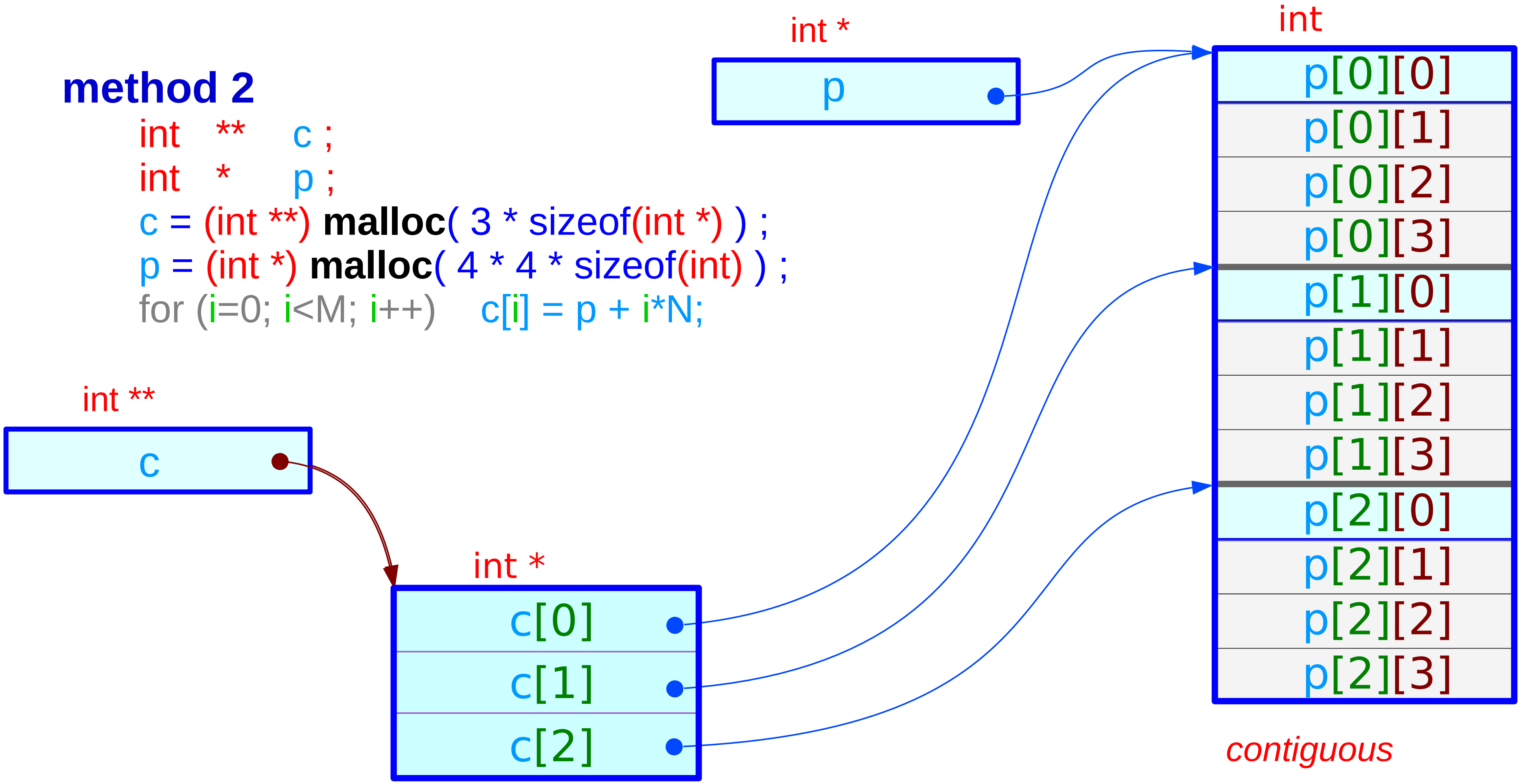
c: an array of integer pointers

may not be contiguous because of memory alignments

2-d array dynamic allocation : method 2

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;
```



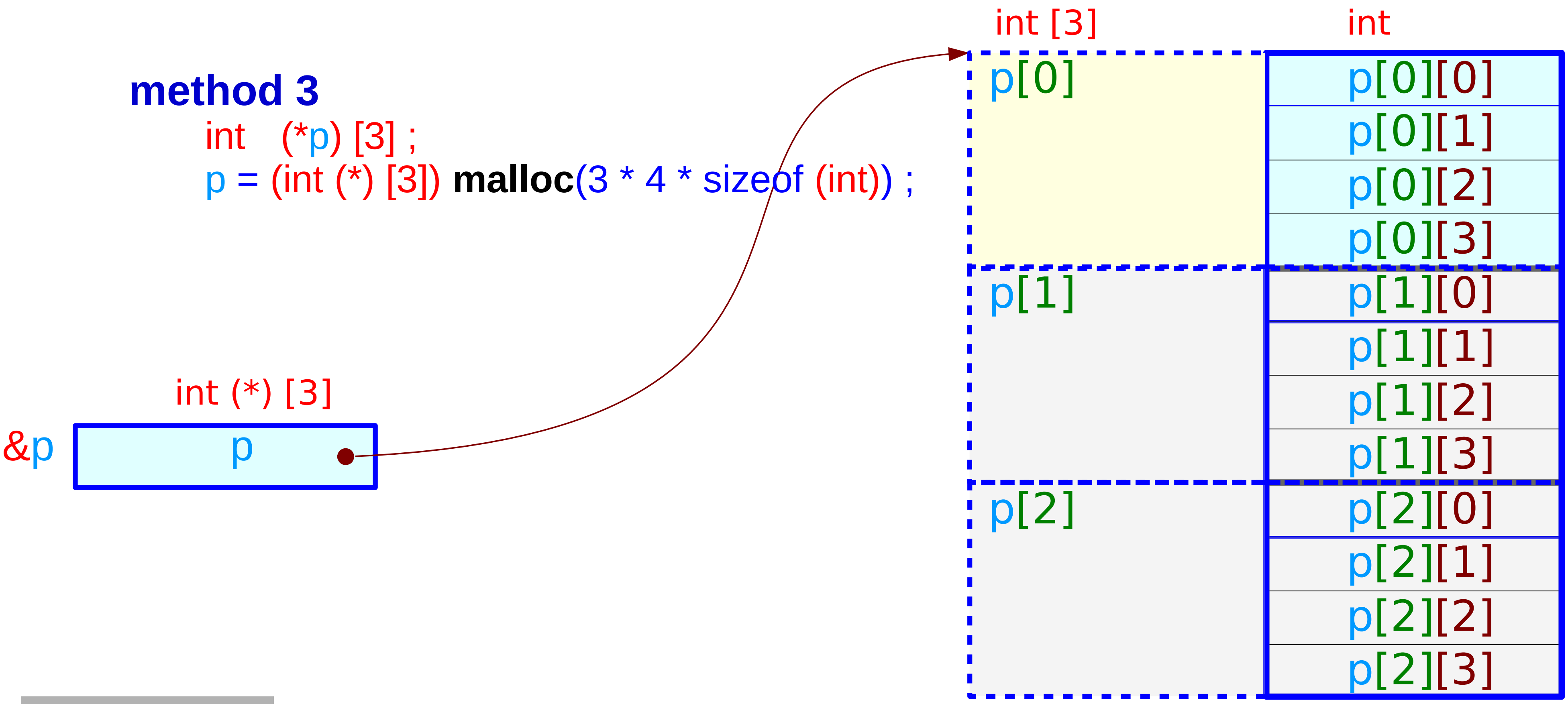
contiguous

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

C scalar data types

Arithmetic types and **pointer** types
- collectively called **scalar types**
- hold single data item
- size and format

Array and **structure** / **union** types
- collectively called **aggregate types**
- hold more than one data items

C scalar data types provide their size and format
The alignment of a scalar data types is equal to its size
Scalar alignment shows scalar alignments that apply to individual scalars and to scalars that are elements of an array or members of a structure or union.
Wide characters are supported (character constants prefixed with an L)
The size of each wide character is 4 bytes

https;//stackoverflow.com/questions/35722514

Data Types

- Void
- Scalar Types
  - Arithmetic Types
    - Integral Types
    - Floating Types
  - Pointer
  - Enum
- Aggregate Types
  - Array
  - Union
  - Union

https;//stackoverflow.com/questions/35722514

C abstract data types

Array as an **Abstract Data Type** and
as a **Data Structure**

Abstract Data Types, **ADT**s are
a way of classifying **data structures**
based on how they are used and
the behaviors they provide.

They do not specify
how the data structure must be implemented
but simply provide a minimal expected interface
and set of behviors.

Data structure is
a concrete implementation of a data type.

It’s possible to analyze the time and memory complexity of a Data Structure but not from a data type.

The Data Structure can be implemented in several ways and its implementation may vary from language to language

Lucasmagnum.edium.com/sidenotes-array-sbstract-data-type-data-structure…

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