Array Pointers (1A)

Young Won Lim
6/21/22

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);
```

Array Pointers

Pointer to Arrays

1. pointer to 1-d arrays
int   (*p) [4];

2. pointer to 2-d arrays
int   (*q) [4][4];

Pointer to an array – variable declarations

int **m** ;

int ***n** ;

an integer pointer

Array Pointer Approach
(pointer to arrays)

int **a** [4]

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

an array pointer

int **func** (**int** a, **int** b) ;

int **(* fp)** (**int** a, **int** b) ;

a function pointer

Pointer to an array – variable declarations

n points to a number (int)

int   ***n ;**
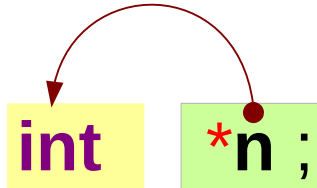
an integer pointer

Array Pointer Approach
(pointer to arrays)

p points to an array (int [4])

int   **(*p)**   **[4]**
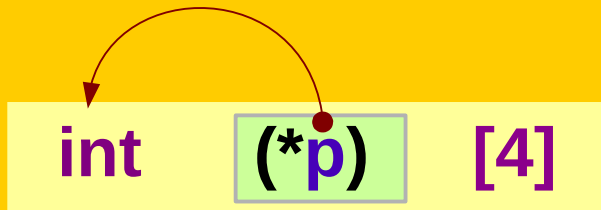
an array pointer

fp points to a function (int (int, int))

int (* fp)   **(int a, int b) ;**

a function pointer

Pointer to an array – a type view

int   4 byte data

int *

an integer pointer
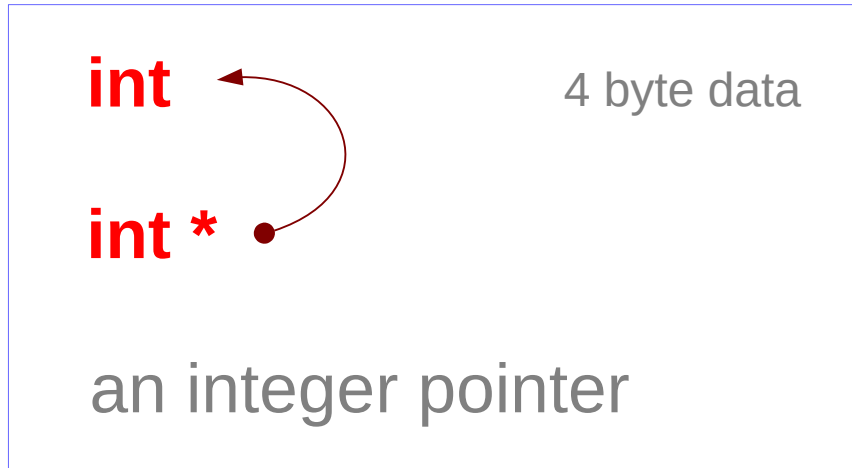
array pointer:
a pointer to an array

pointer array:
an array of pointers

int [4]   4*4 byte data

int (*) [4]

an array pointer

int (int, int)   instructions

int (*) (int, int)

a function pointer

Pointer **q** to a **1-d** array **a**

p = &a;

p •

int [4]   int

&a

a	a[0]
	a[1]
	a[2]
	a[3]

no physical
memory locations

real consecutive
memory locations

1-d array pointer

int   (*p) [4];

int   a [4];

Pointer **q** to a **2-d** array **c**

q = &c;

q ●

int [3][4]   int [4]   int

&c | 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 pointer

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

int   c [3][4];

no physical memory locations

real consecutive
memory locations
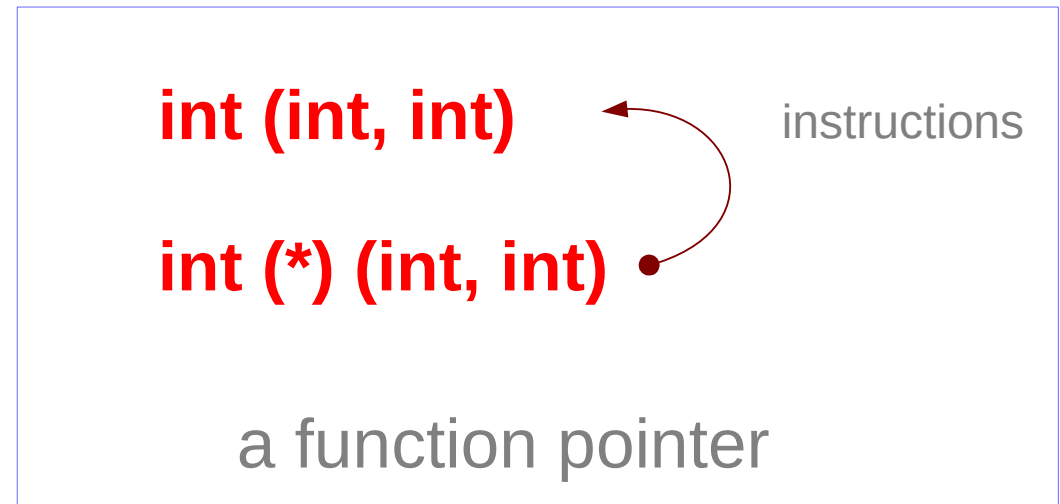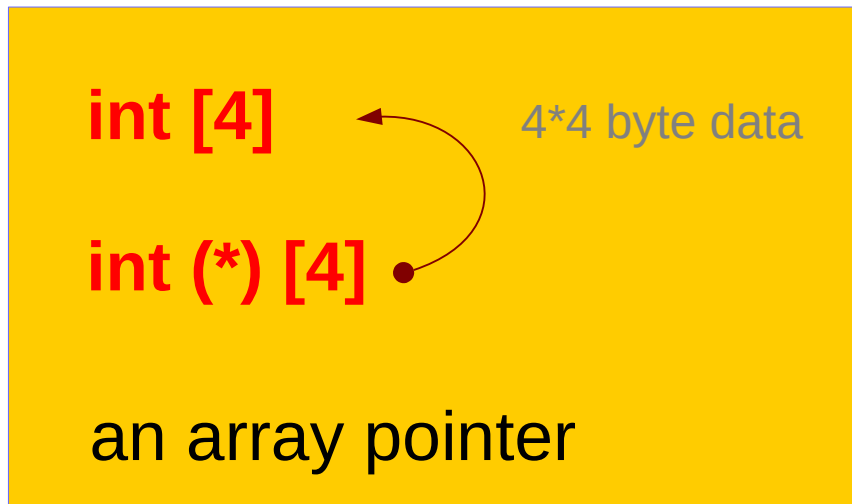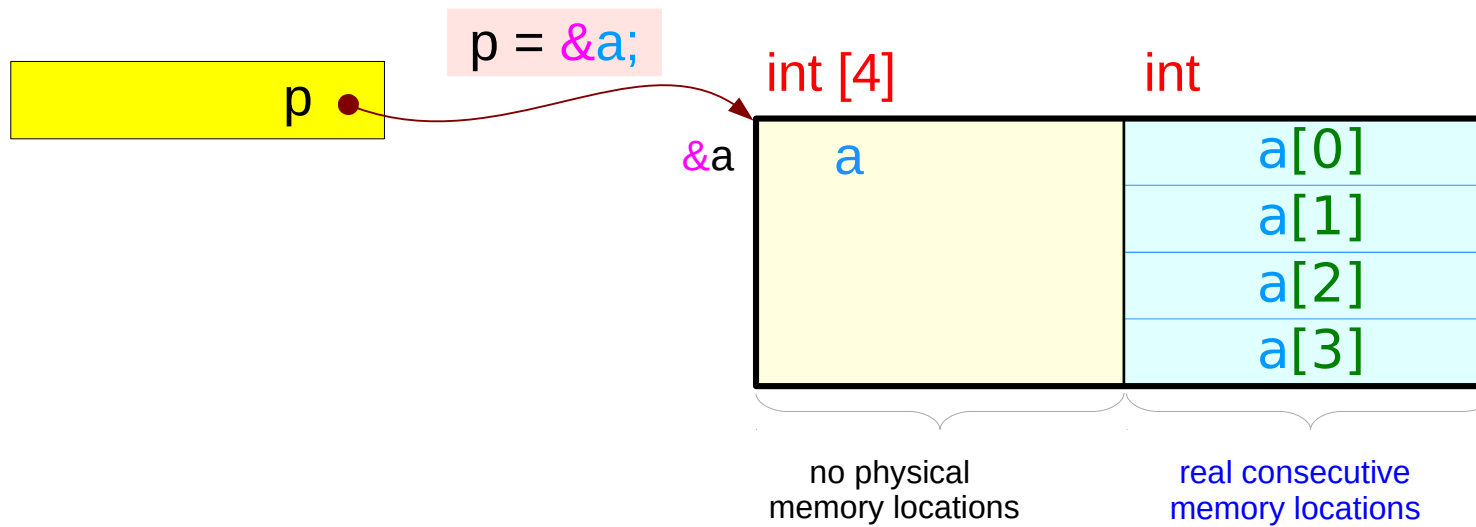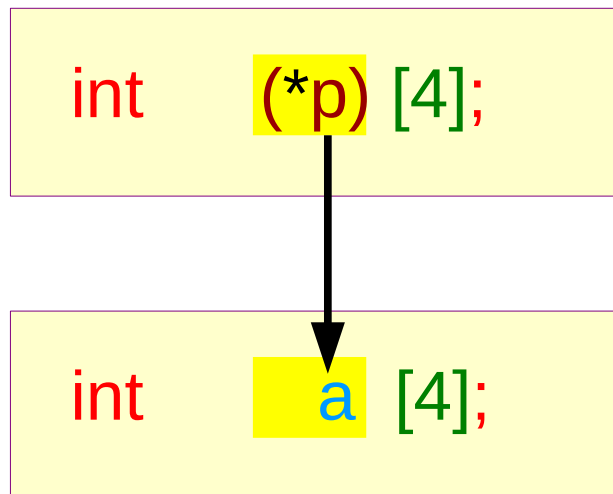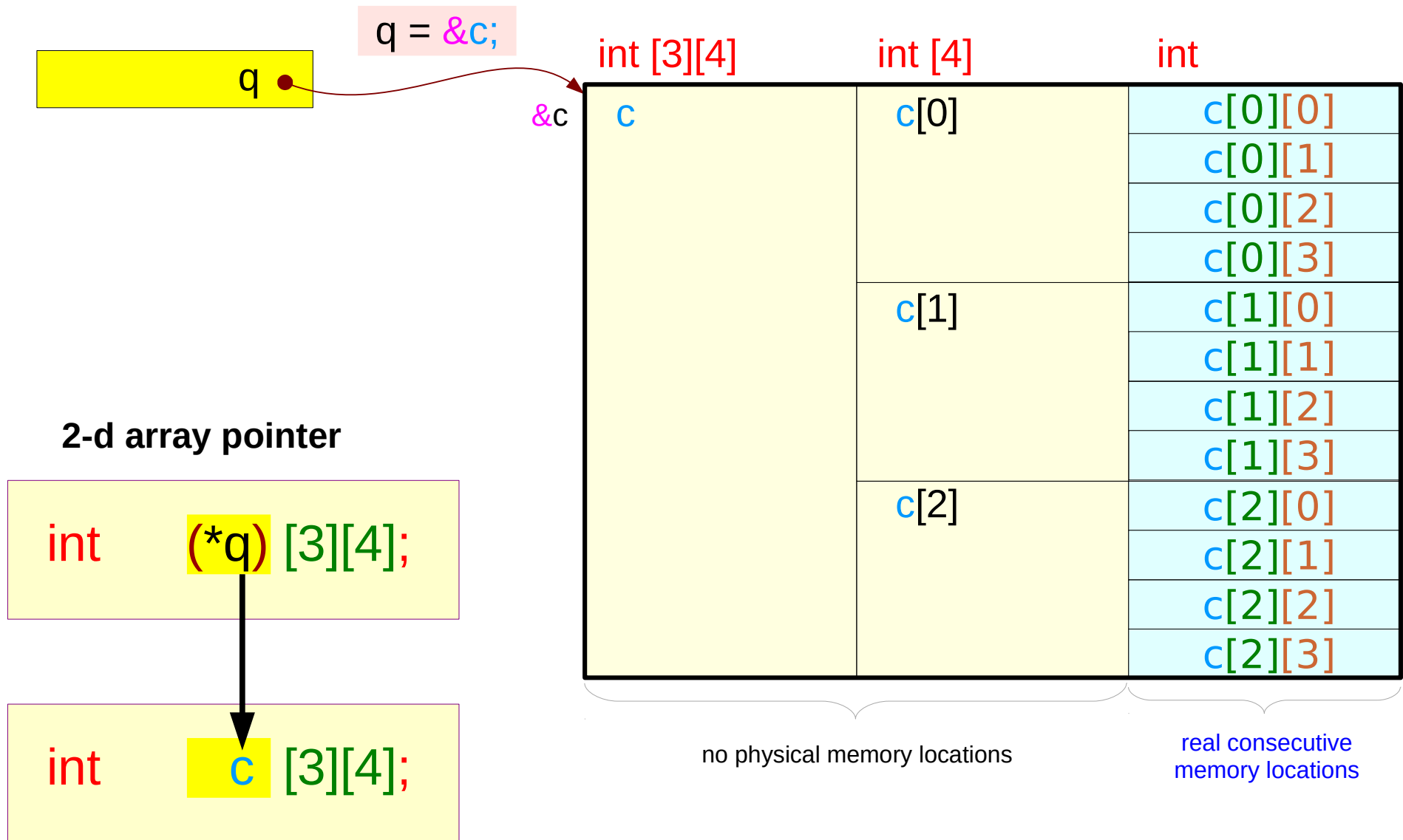
Array **a** vs array pointer **A**

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

a	a[0]
	a[1]
	a[2]
	a[3]

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

A → | *(A+0) | minimal reference
| *(A+1) |
| *(A+2) | extended references
| *(A+3) |

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

a → | *(a+0) | minimal reference
| *(a+1) |
| *(a+2) | extended references
| *(a+3) |

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

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

A	A[0]
	A[1]
	A[2]
	A[3]
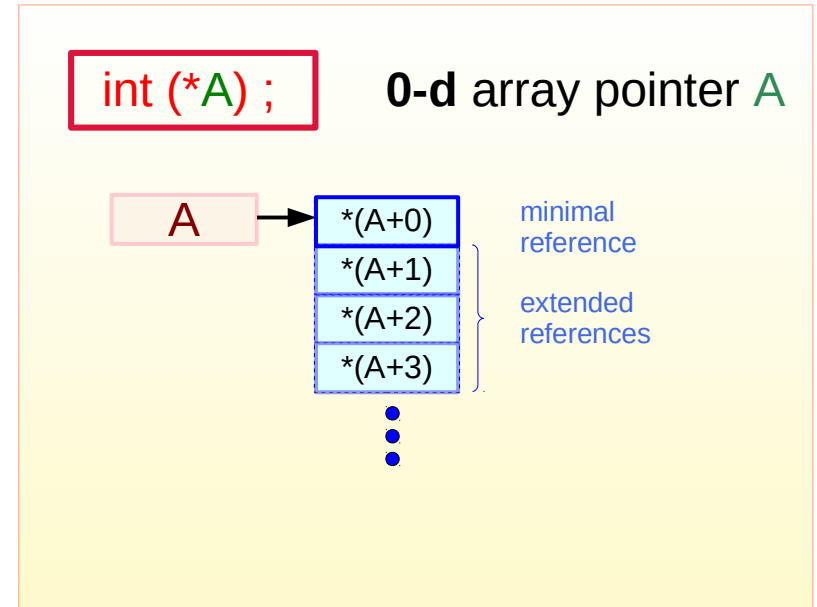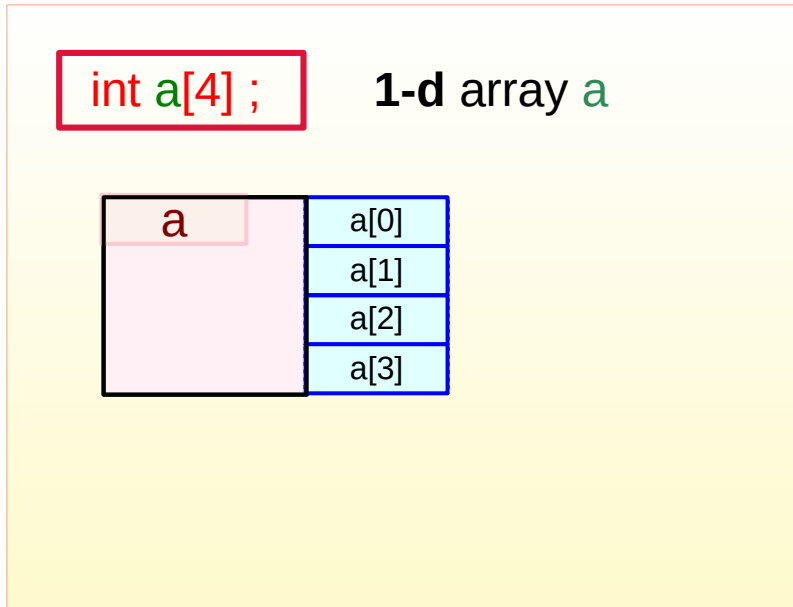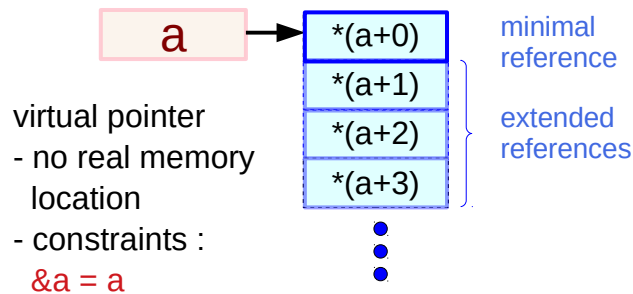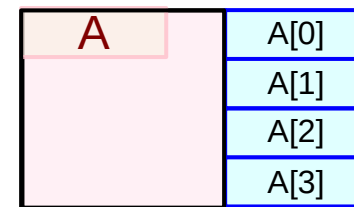
N is not fixed
to 4

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 <u>not</u> a real <u>pointer</u>
- sizeof(a) = an array size
- a = &a

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

A is <u>not</u> a real <u>array</u>
- sizeof(A) = a pointer size
- A ≠ &A

Array and pointer types in a **1-d** array

int [4]   int

int   a   [4]

a **1-d array**

type : int [4]

size : 4 * 4

a	a[0]
	a[1]
	a[2]
	a[3]

no physical memory locations   real consecutive memory locations

int (*)   int

int   a   [4]

a **0-d array pointer** (virtual)

type : int (*)

size : 4 * 4

a ●	a[0]
	a[1]
	a[2]
	a[3]

no physical memory locations   real consecutive memory locations

a points to the 1st **int** element
there are 4 **int** elements

2-d array type

int [3][4]

int   c   [3]   [4]
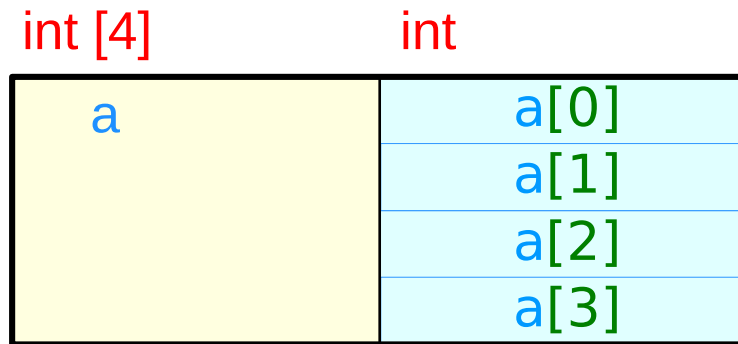
c   2-d array

type :   int [3][4]

size :   3 * 4 * 4

int [3][4]

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]

no physical memory locations

real consecutive
memory locations

1-d array pointer type

int (*) [4]   int [4]

int   c   [3]   [4]

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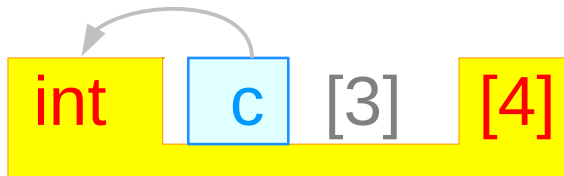
c   **1-d array pointer** (virtual)

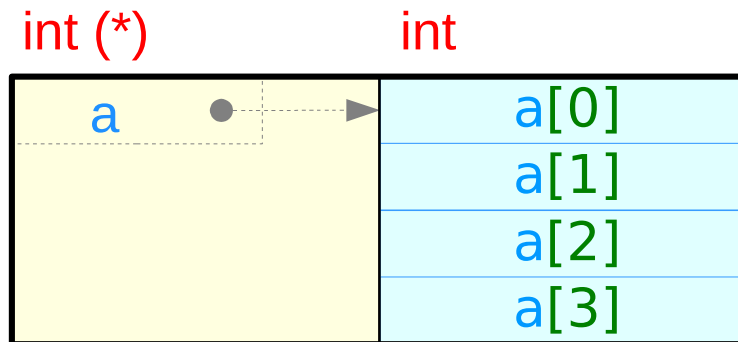type :   int (*) [4]

size :   3 * 4 * 4

c points to the 1st **int [4]** element
There are 3 **int [4]** elements

no physical memory locations

real consecutive
memory locations

1-d array type

int [4]

int   c   [3]   [4]

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

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]

no physical memory locations

real consecutive
memory locations
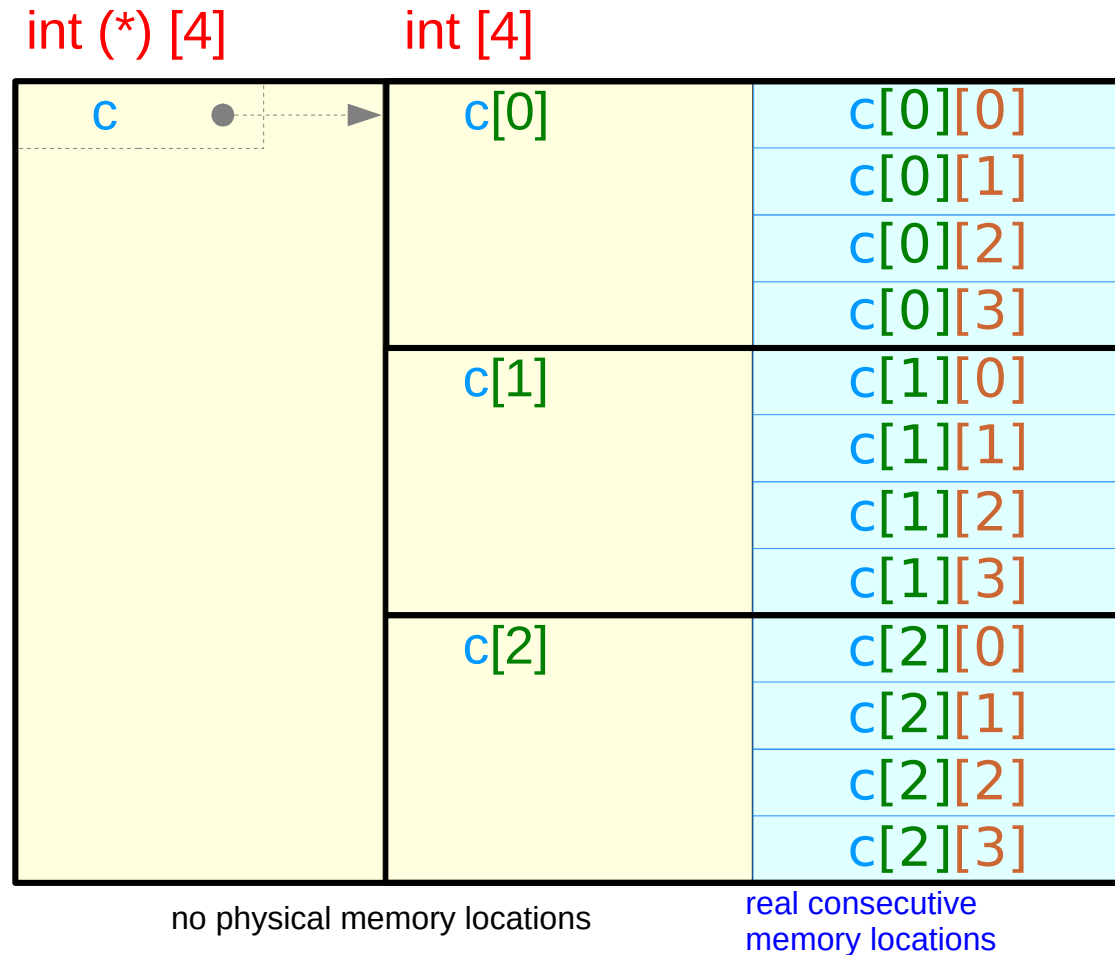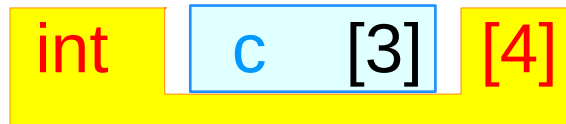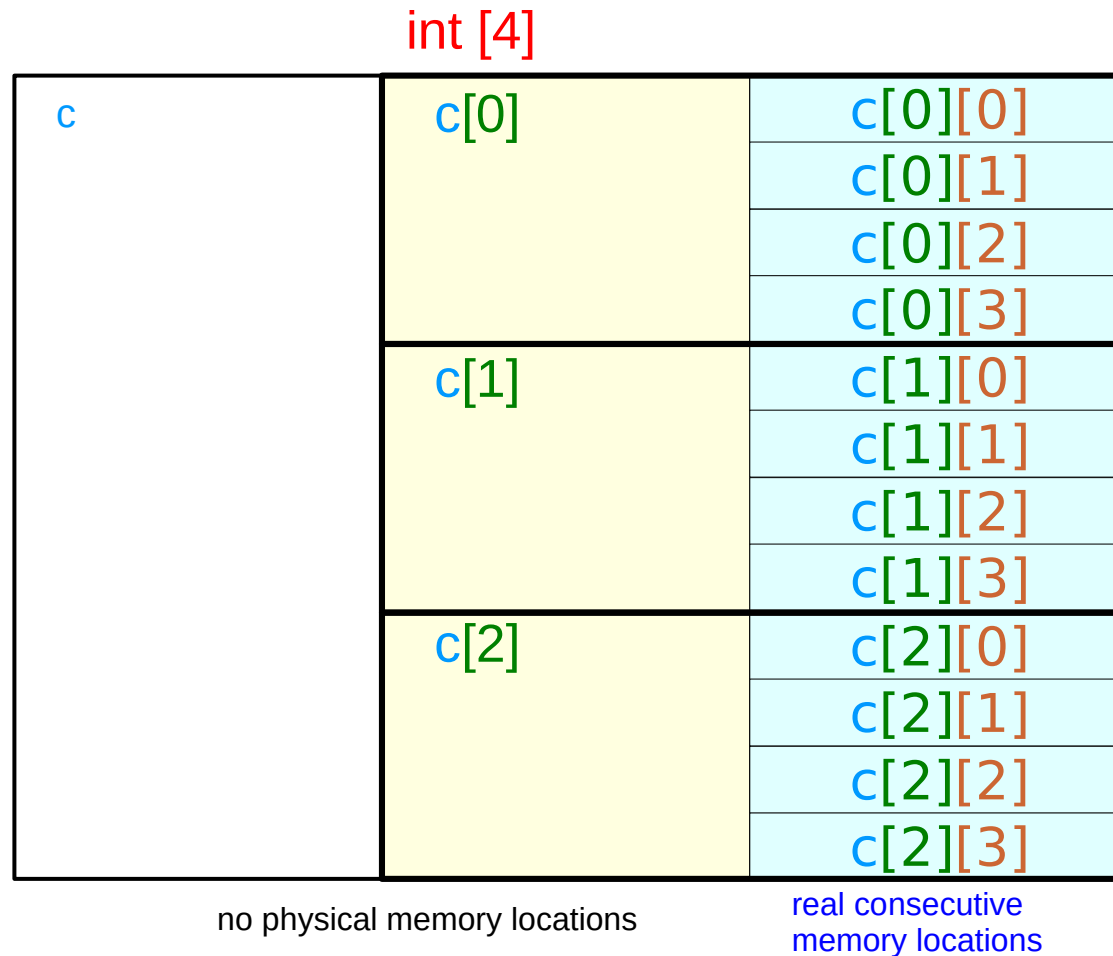
0-d array pointer type

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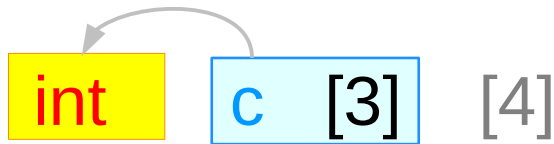
int   c   [3]   [4]

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

type :   int (*)

size :   4 * 4

c[i] points to the 1st **int** element
There are 4 **int** elements

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]

no physical memory locations   real consecutive
memory locations
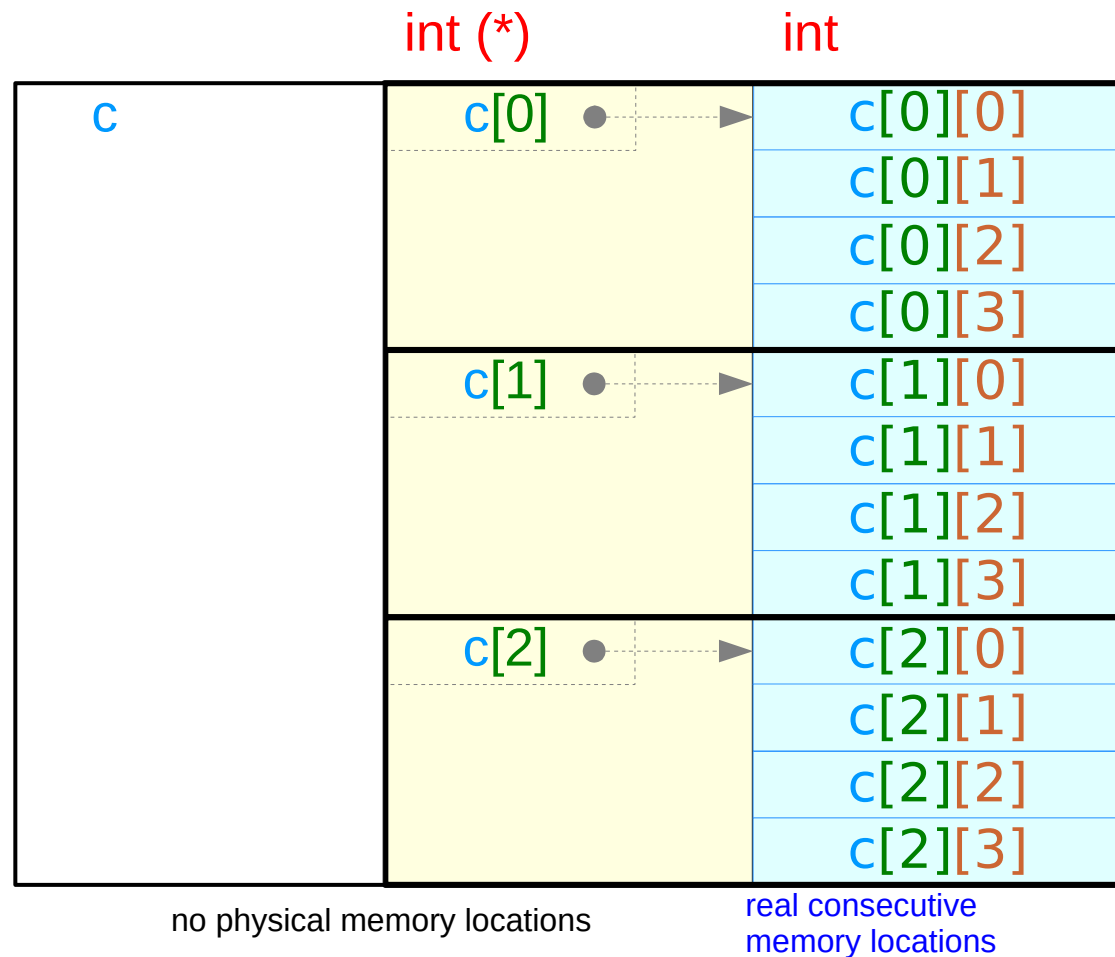
Values of virtual array pointer in a **2-d** array

c = c[0] = &c[0][0]

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

c[2] = &c[2][0]

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

no physical memory locations

real consecutive
memory locations
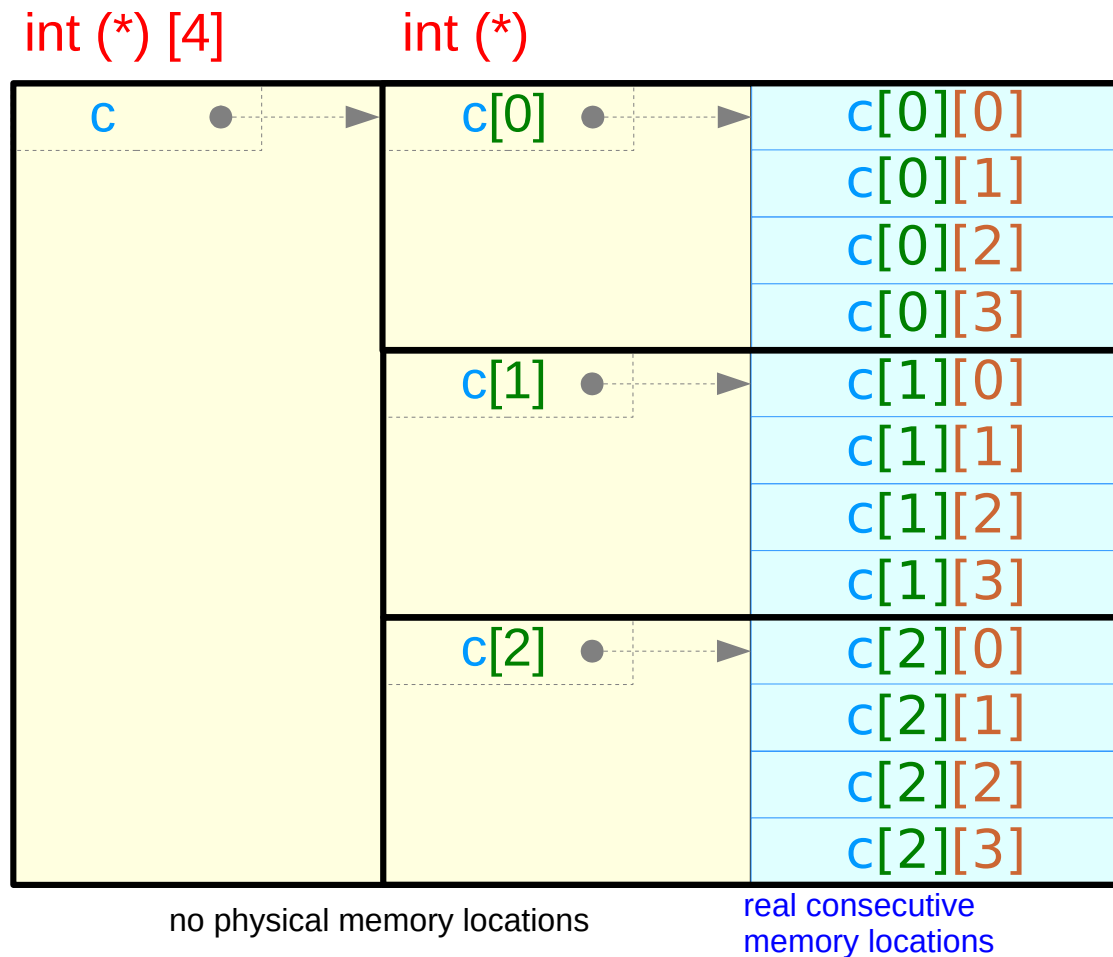
Types in a **2-d** array

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]

Subarray types in a **2-d** array

int c[3][4];   **2-d** array c

c Type	c[i] Type	c[i][j] Type	
int c[3][4]	int c[3][4]	int c[3][4]	
int [3][4]	int [4]	int	Array Type
int (*)[4]	int (*)		virtual Array Pointer Type

Dual Types

Pointer **p** to a **1-d** array **a**

p

p = &a;

equivalence

*p ≡ **a**

assignment

p = **&a**

int [4]   int

&a

a	a[0]
	a[1]
	a[2]
	a[3]
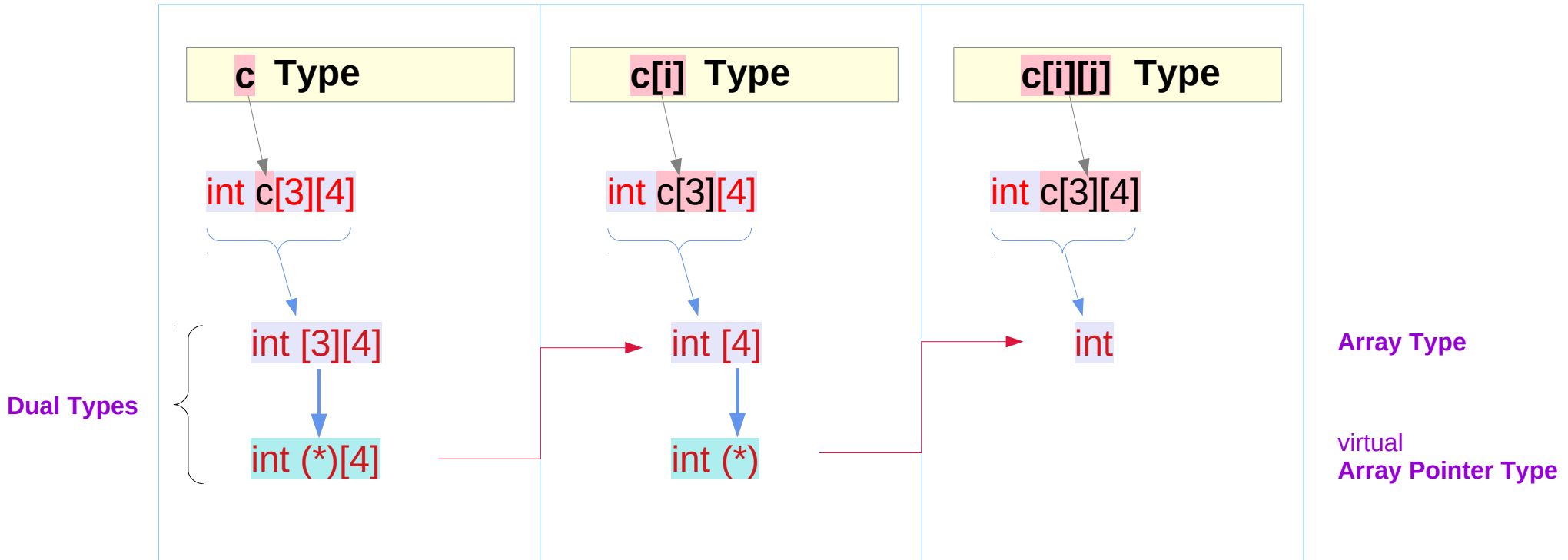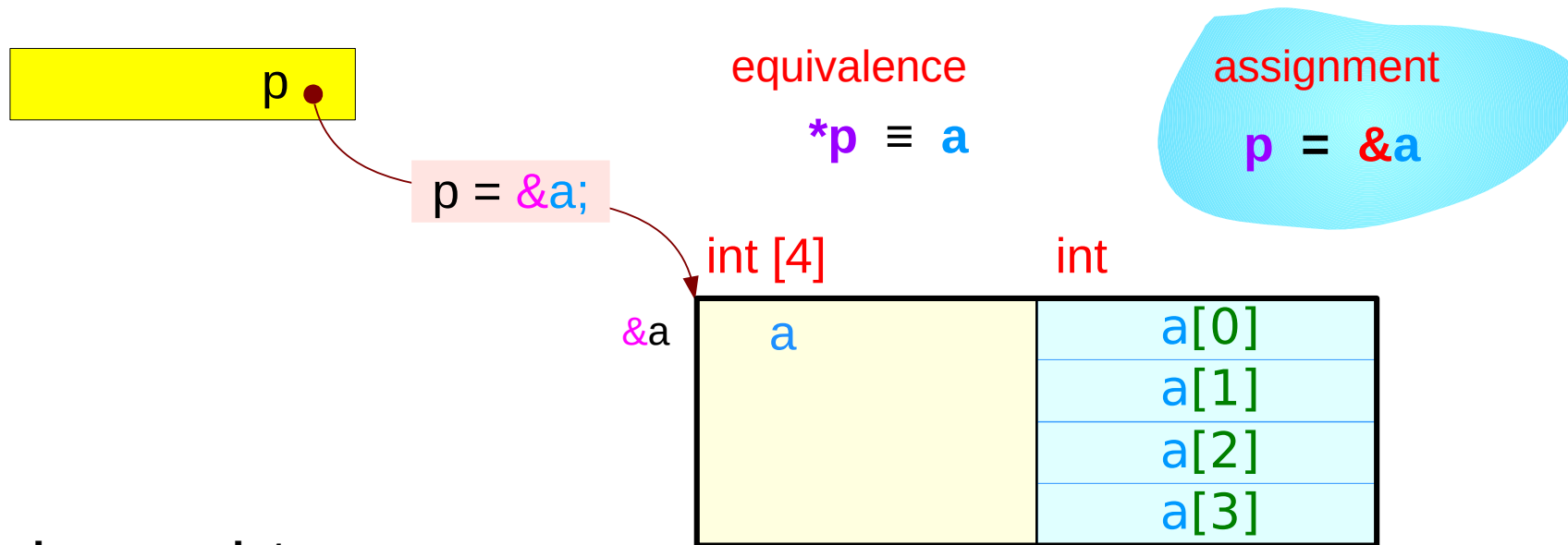
1-d array pointer

int   (*p) [4];

1-d array

int   c [4];

Pointer **q** to a **0-d** array **a[0]**

q

q = c;

equivalence

$*q \equiv a[0]$

assignment

$q = a$

int (*)   int

a ●	a[0]
a	a[1]
	a[2]
	a[3]
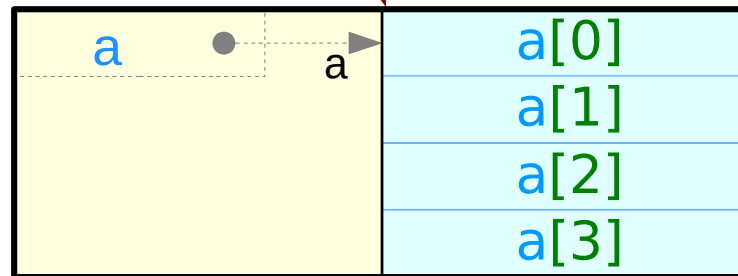
0-d array pointer

int   (*q)   [4];

1-d array

int   a[4]   ;

1-d array access using p and q

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

int (*q)   = a;

1-d array pointer
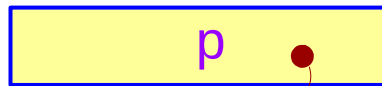
| p   ● |
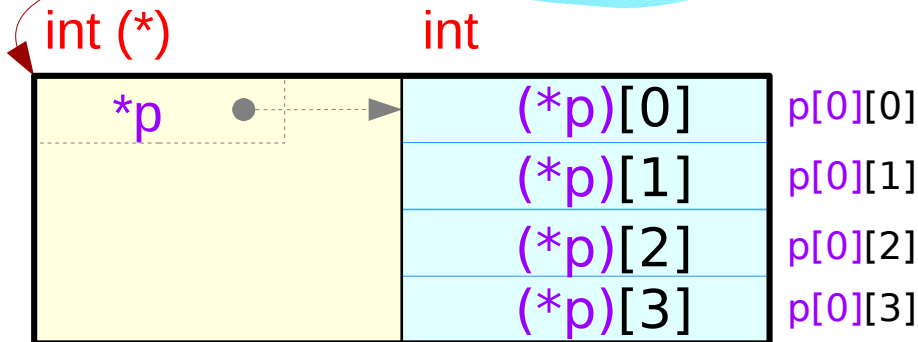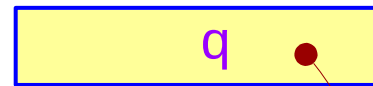
p = &a

assignment

int (*)   int

*p ● ┄┄►	(*p)[0]	p[0][0]
	(*p)[1]	p[0][1]
	(*p)[2]	p[0][2]
	(*p)[3]	p[0][3]

0-d array pointer

| q   ● |

q = a

assignment

int (*)   int

a ● ┄┄►	q[0]
	q[1]
	q[2]
	q[3]

Pointer **p** to a **2-d** array **c**

p

p = &c;

equivalence

*p ≡ **c**

assignment

p = **&c**

int [3][4]   int [4]   int

&c

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 pointer

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

2-d array

int   c [3][4];

Pointer **q** to a **1-d** array **c[0]**

q

q = c;

equivalence

$*q \equiv c[0]$

assignment

$q = c$

int (*)[4]   int [4]   int

c   •----▶   c[0] | c[0][0] |
c | 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] |

1-d array pointer

int (*q) [4];

2-d array

int c[3] [4];
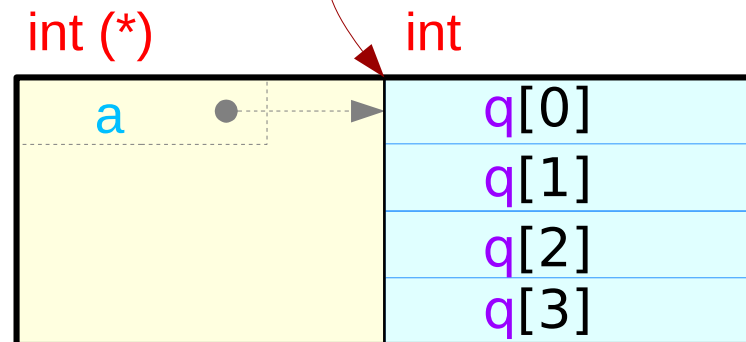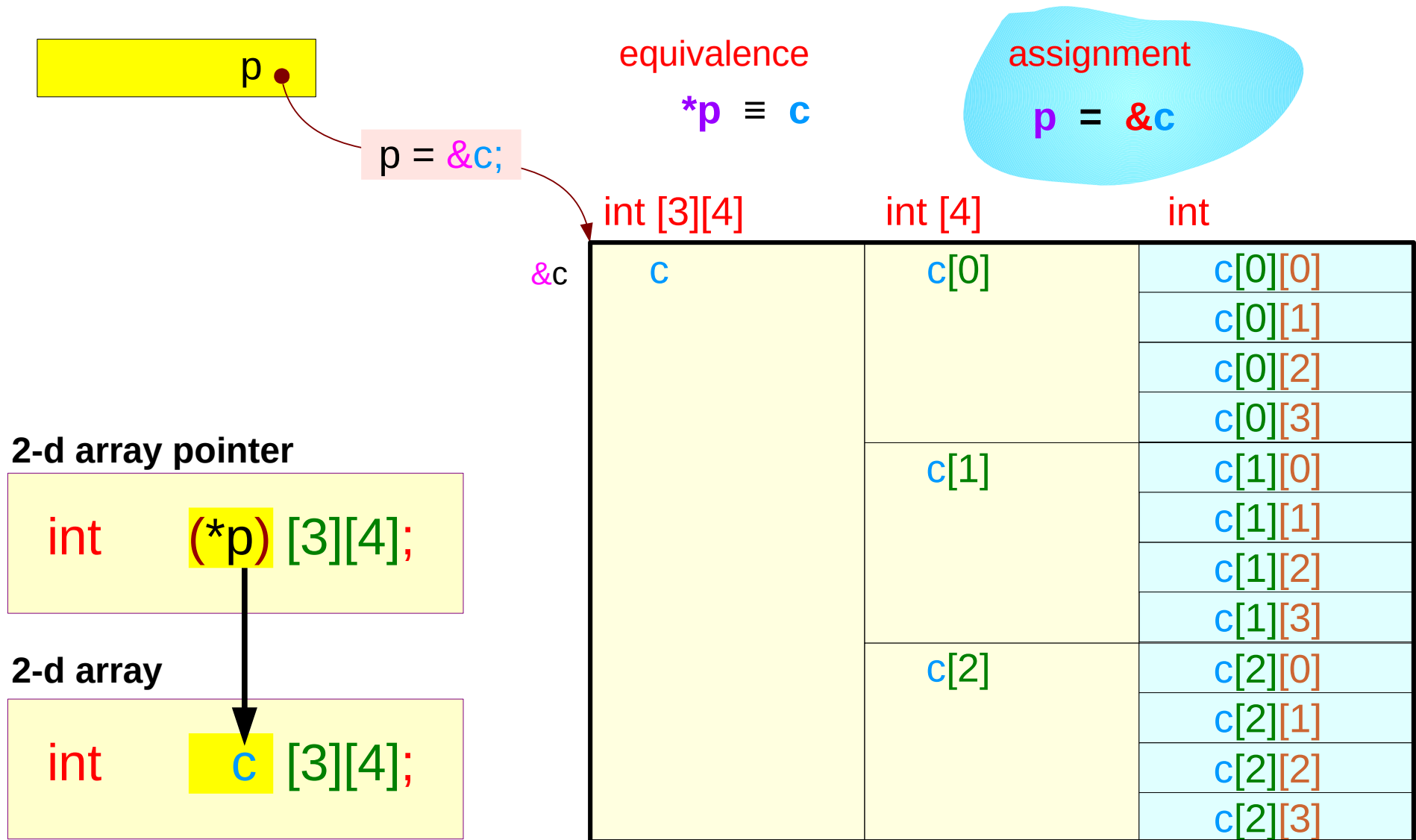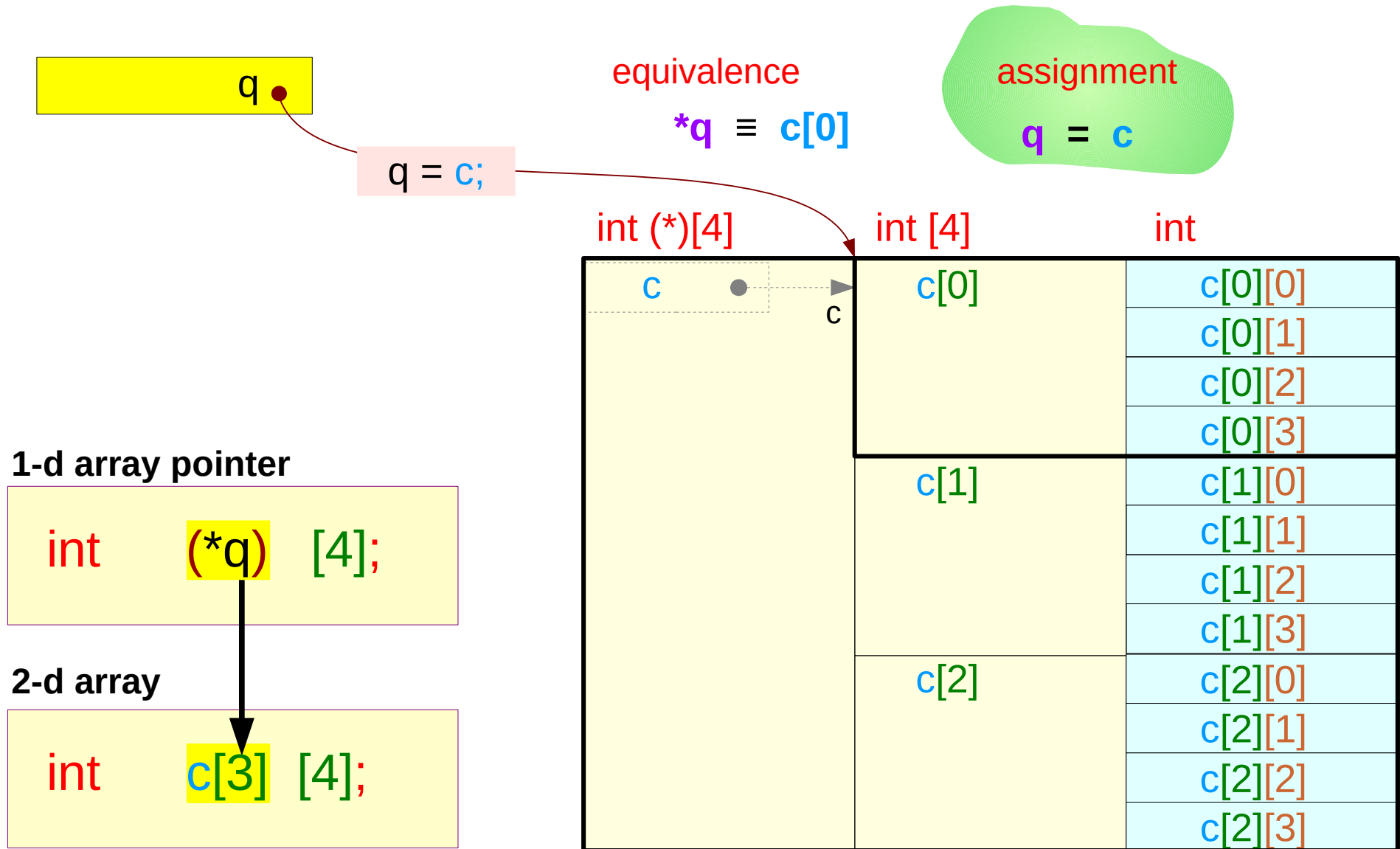
2-d array access using **p**

2-d array pointer

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

p

p = &c;

int (*)[4]   int (*)   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 access using **q**

1-d array pointer

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

q •

q = c;

int (*)[4]   int [4]   int

c	q[0]	q[0][0]
		q[0][1]
		q[0][2]
		q[0][3]
	q[1]	q[1][0]
		q[1][1]
		q[1][2]
		q[1][3]
	q[2]	q[2][0]
		q[2][1]
		q[2][2]
		q[2][3]
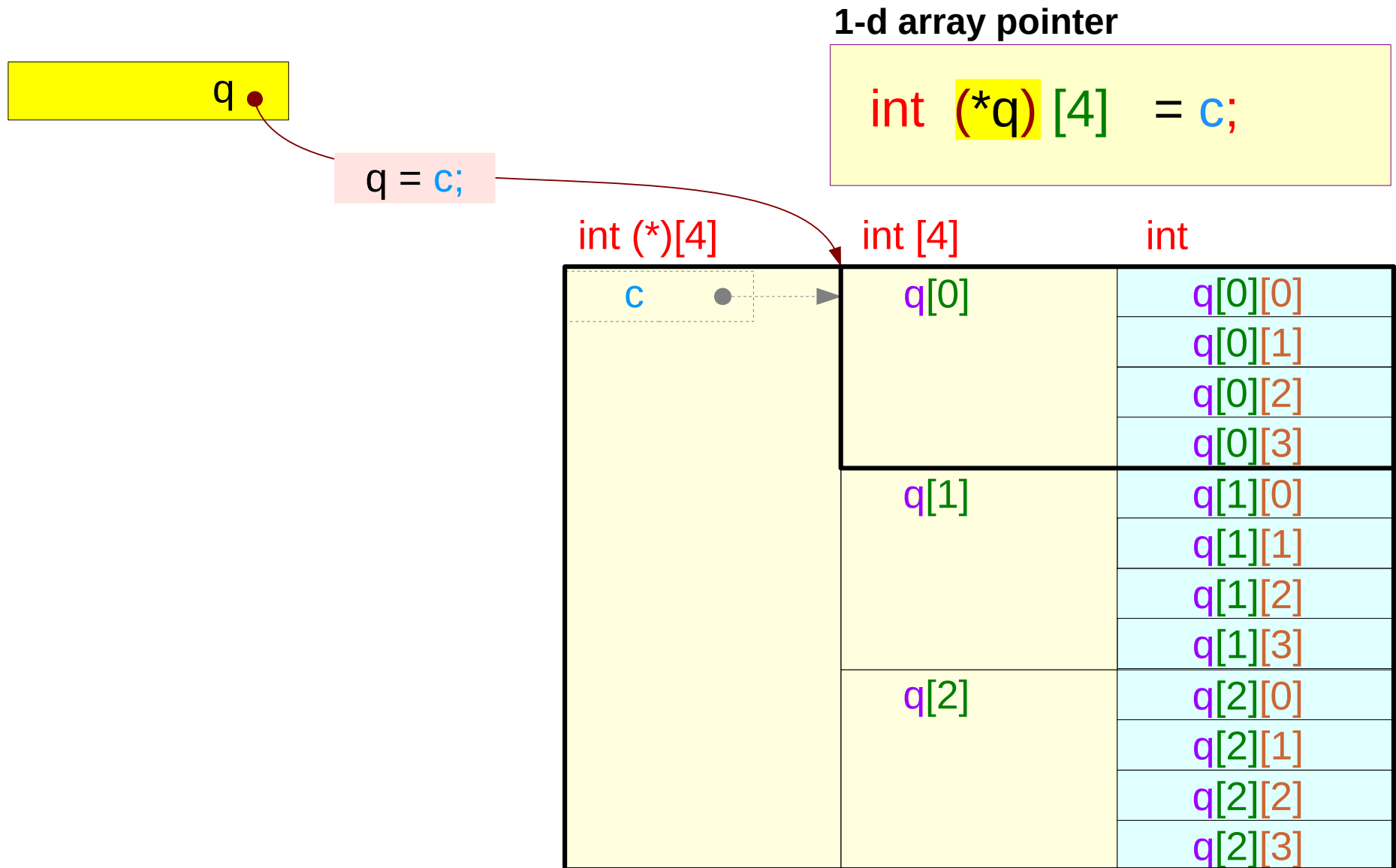
1-d and **0-d** array pointers to an **1-d** array

1-d array pointer

int	(*q)	[4] ;
⇕		
int	c	[4] ;

1-d
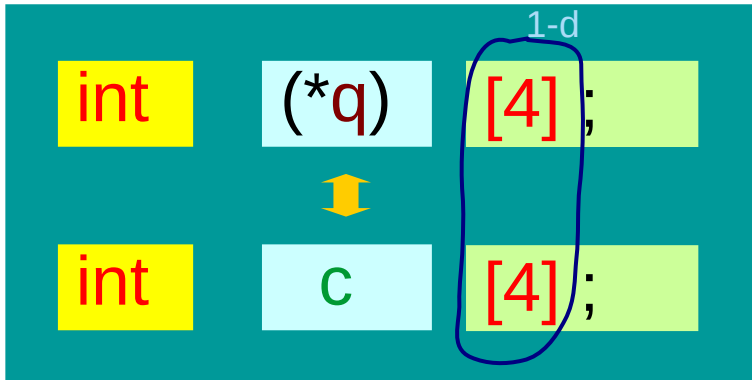
int (*) [4]

equivalence	*q ≡ c;
assignment	q = &c;

(*q)[**i**] ≡ q[0][**i**] ≡ c[**i**]

0-d array pointer : int pointer

int	(*p)	;
⇕		
int	c[4]	;

0-d
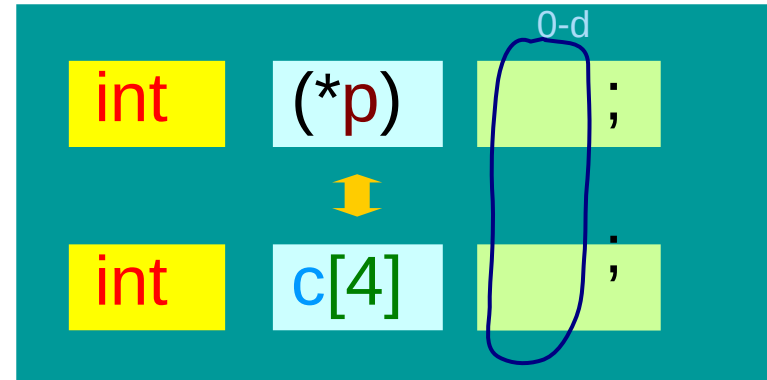
int (*)

equivalence	*p ≡ *c;
assignment	p = c;

p[**i**] ≡ c[**i**]

2-d and **1-d** array pointers to a **2-d** array

2-d array pointer
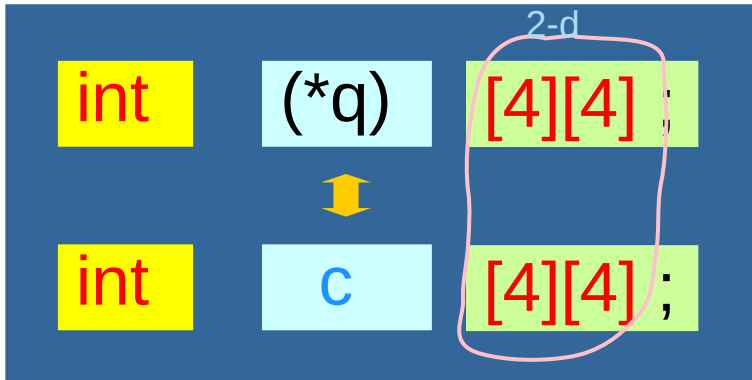
int (*) [4][4]

equivalence	*q ≡ c;
assignment	q = &c;

(*q)[i][j] ≡ q[0][i][j] ≡ c[i][j]

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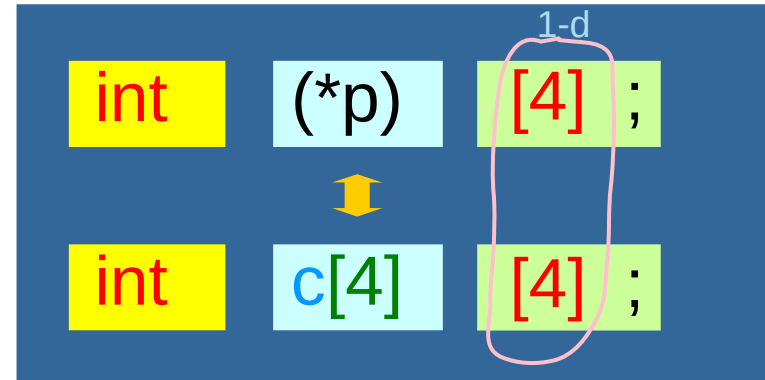
int (*) [4]

equivalence	*p ≡ *c;
assignment	p = c;

p[i] ≡ c[i]

Array pointers to a **2-d** array

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

2-d array c

2-d array pointer q

1-d array pointer p

c[i][j]   ➡   (*q)[i][j]

c[i][j]   ➡   p[i][j]

int **   ⟶   int *   ⟶   int

int (*) [4]   ⟶   int [4]   ⟶   int
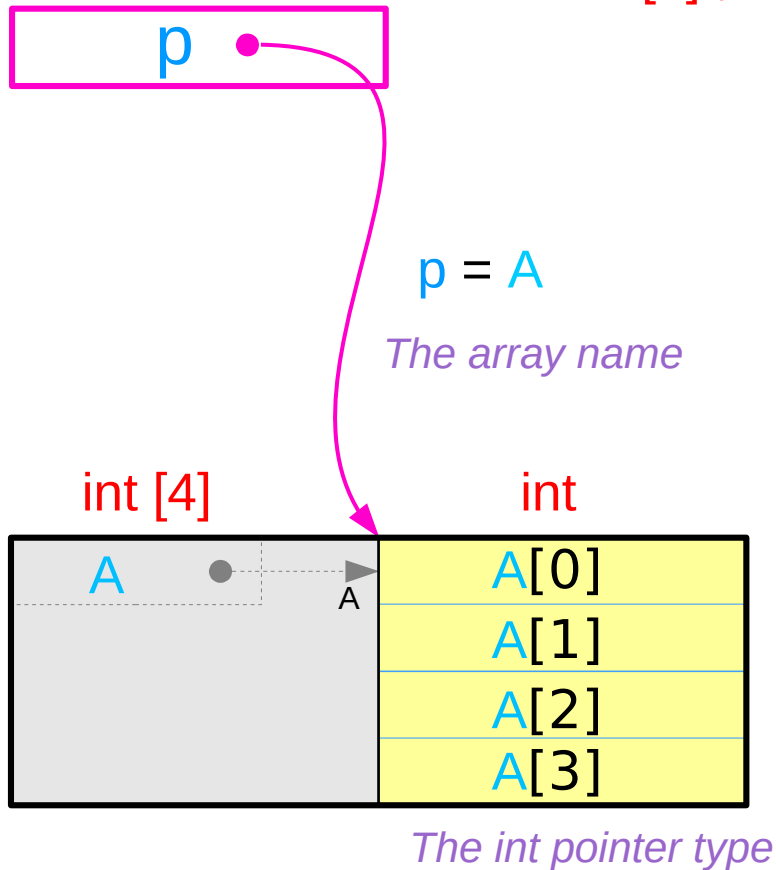   int
                                                            int
                                                            int
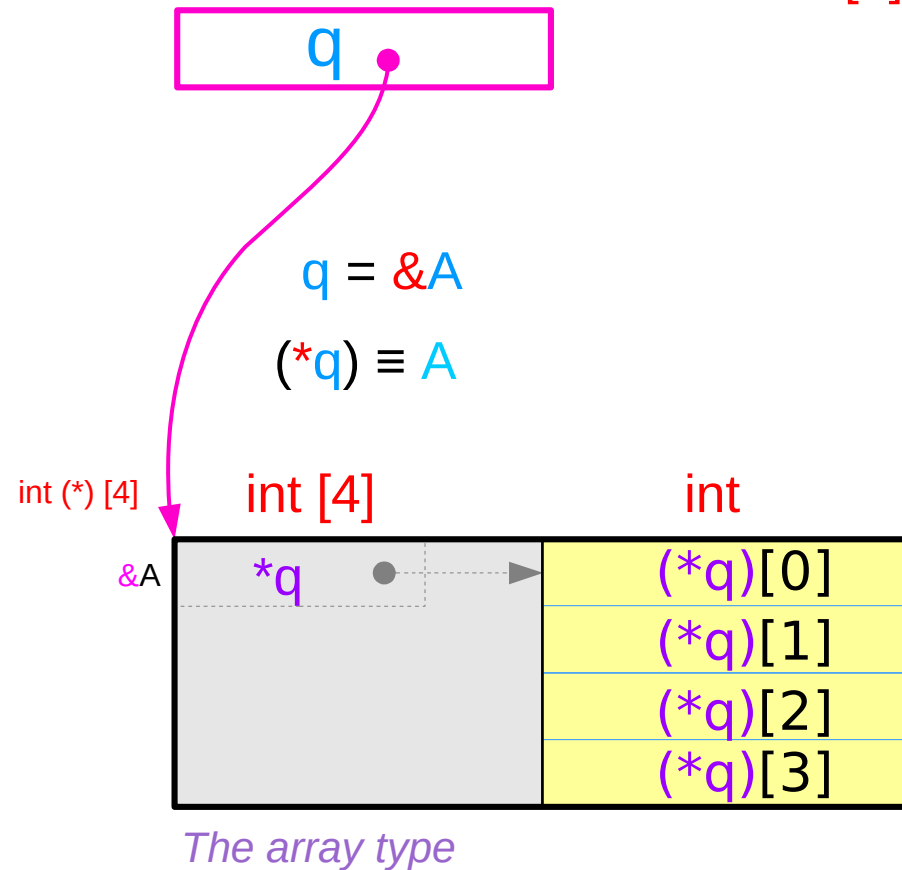
Integer pointer **p** vs. array pointer **q**

int *p ;   p = A ;

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

int A[4] ;

int A[4] ;

p

q

p = A

q = &A

The array name

(*q) ≡ A

int [4]   int

int (*) [4]   int [4]   int

A		A[0]
		A[1]
		A[2]
		A[3]

&A   *q

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

The int pointer type

The array type

Must point to an array type (array name)

int (*q) [4];   q ≠ A ;

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

int A[4] ;

int A[4] ;

q ●

q ●

Warning!

q=A

The array name

q=&A

(*q) ≡ A

int [4]   int

int (*) [4]   int [4]   int

A	A[0]
	A[1]
	A[2]
	A[3]

&A

*q	(*q)[0]
	(*q)[1]
	(*q)[2]
	(*q)[3]
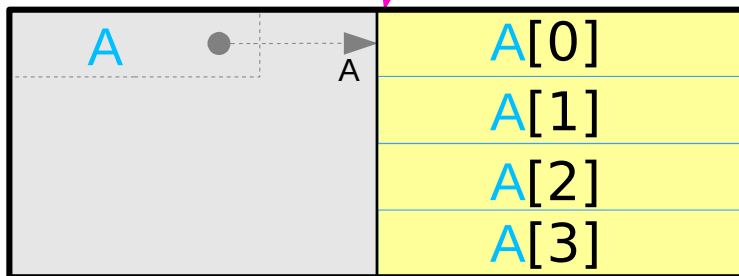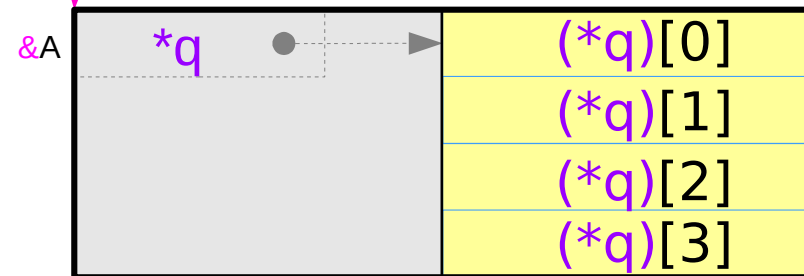
The array type

The array type

Double integer pointer type – **int ****

int [4]

an **int** array name
size = 4*4 bytes

int

int

int

int

×

ok

ok

a pointer to a pointer

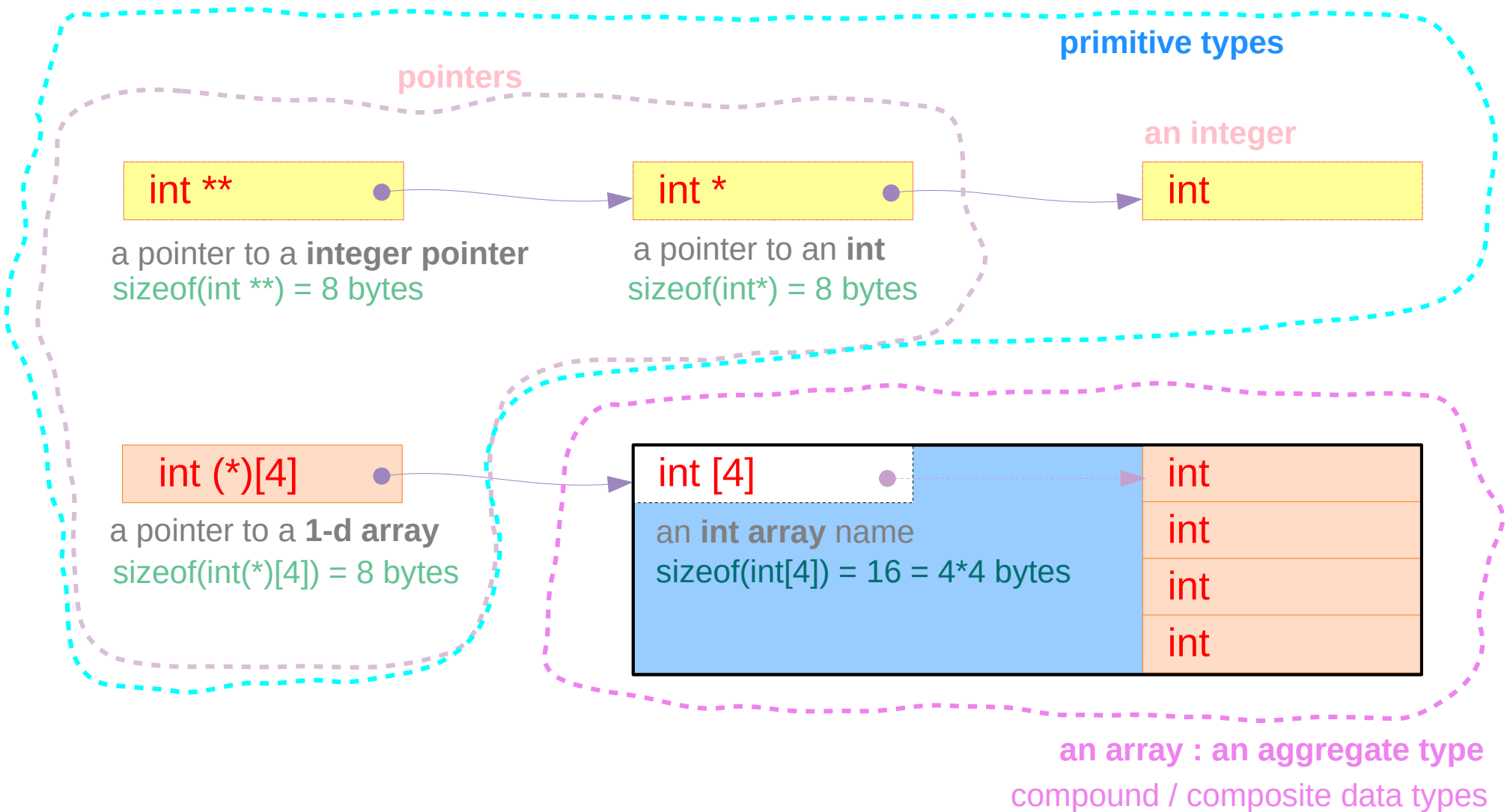int **

int *
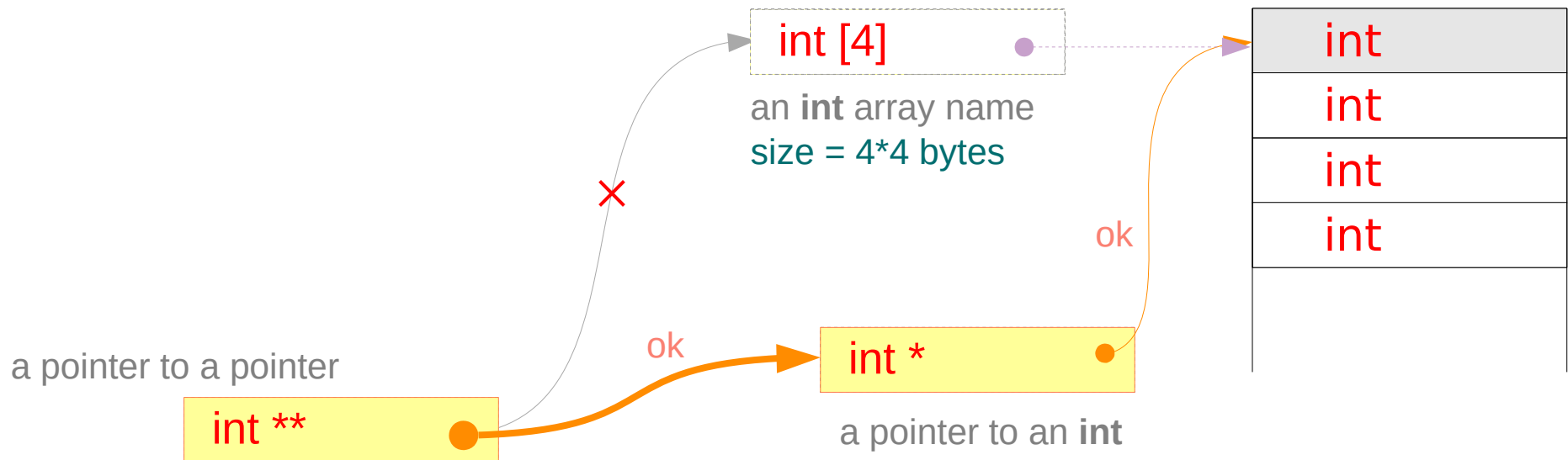
a pointer to an **int**

int ** type can point
only to **int** * type
– an **int** array name (X)

type(int [4]) = type(int [ ]) = int *

each of these types points
to an **int** type data

Integer array pointer type – **int (*)[4]**

int (*)**[4]** type can point
only to **int [4]** type
– an **int** array name

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

int [4]   •

an **int** array name
size = 4*4 bytes

ok

ok

int
int
int
int

int *   ●

a pointer to an **int**

int (*)[4]   ●

1-d array pointer

×

type(int [4]) = type(int [ ]) = int *

each of these types points
to an **int** type data

Types of integer pointers

pointers

primitive types

an integer

int **	●

a pointer to a **integer pointer**
sizeof(int **) = 8 bytes

int *	●

a pointer to an **int**
sizeof(int*) = 8 bytes

int

int (*)[4]	●

a pointer to a **1-d array**
sizeof(int(*)[4]) = 8 bytes

int [4]	●	int

an **int array** name
sizeof(int[4]) = 16 = 4*4 bytes

int
int
int

an array : an aggregate type

compound / composite data types

Variable declaration of integer pointers

int *q = &p;   int *p = &a;   int a;

| q ● | &p | p ● | &a | a |

int (*r)[4] = &c;   int c[4];

| r ● | &c | c ● |

	c[0]
	c[1]
	c[2]
	c[3]
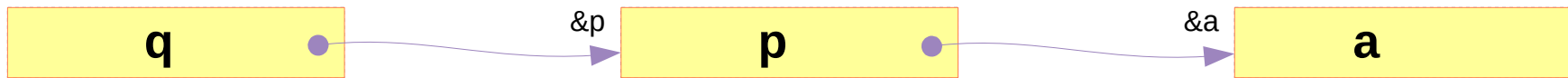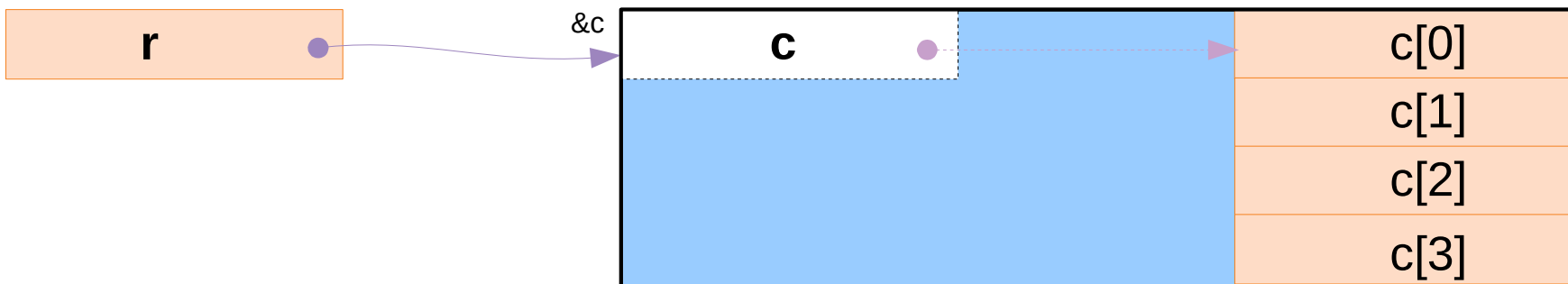
Types and sizes of integer pointers

type(int [4]) = type(int [ ]) = (int *)

int **a;**
int *p = &a;
int *q = &p;

&q (int **) **q** ● &p (int *) **p** ● &a (int) **a**

value(&q) ≠ value(q)
sizeof(q) = pointer size

value(&p) ≠ value(p)
sizeof(p) = pointer size

value(&a) ≠ value(a)
sizeof(a) = 4

int c[4];
int (*r)[4] = &c;

&r (int (*)[4]) **r** ● &c (int [4]) **c** ● c

value(&r) ≠ value(r)
sizeof(r) = pointer size

value(&c) = value(c)
sizeof(c) = 4*4

int c[0]
int c[1]
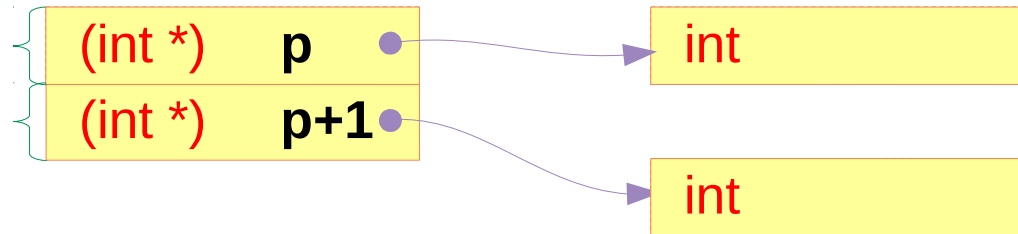int c[2]
int c[3]
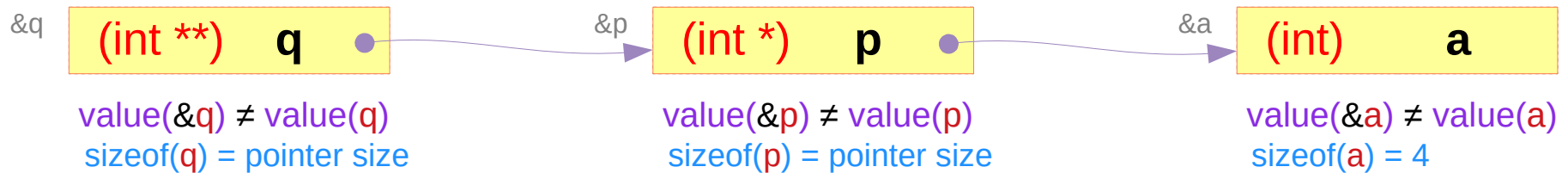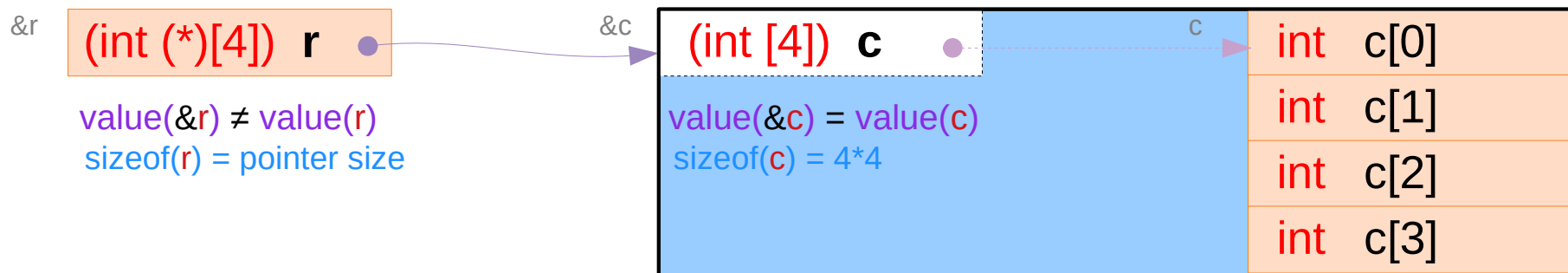
Sizes of integer pointers

a pointer to an **int**

sizeof(p) = pointer size
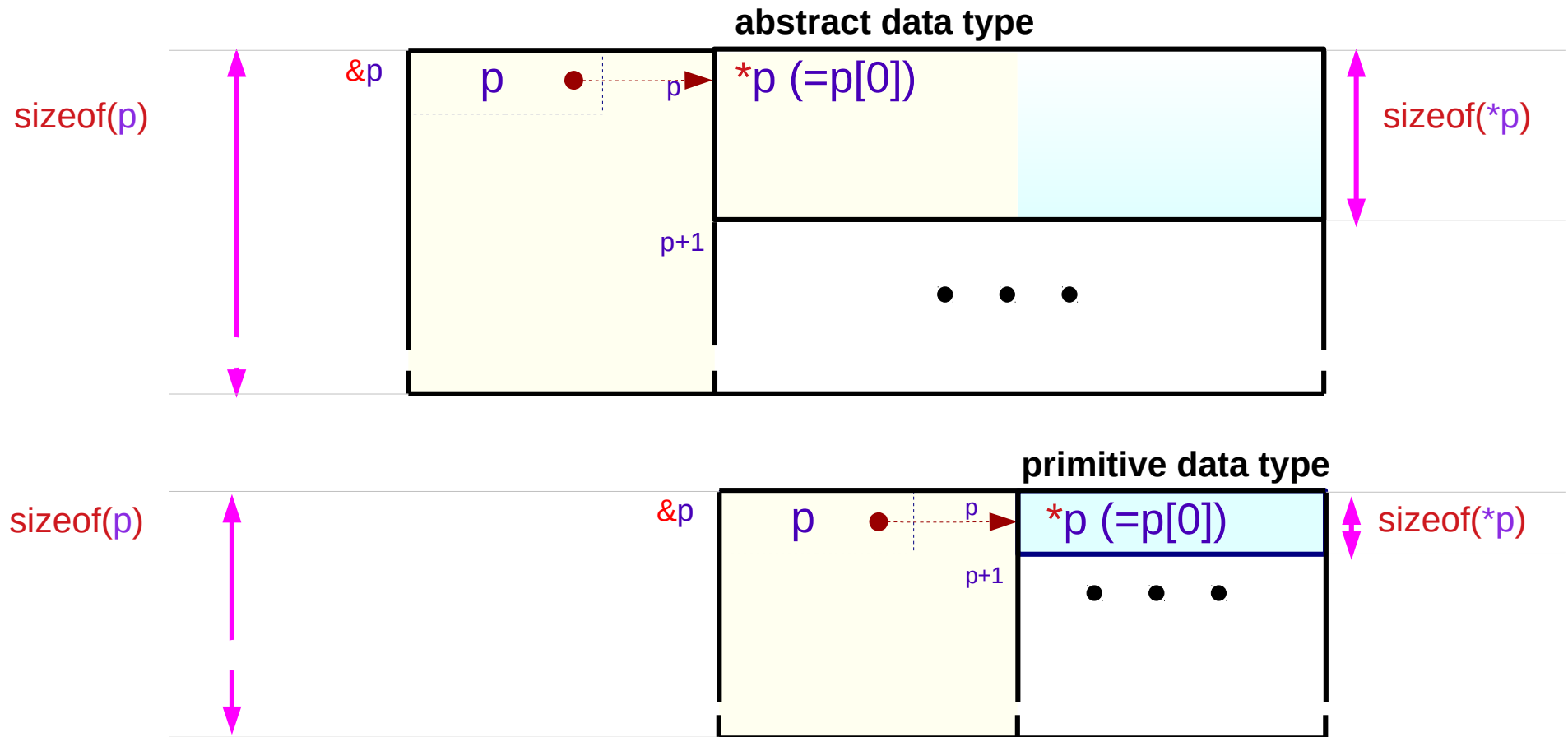= 8 bytes on 64-bit machine
= 4 bytes on 32-bit machine

(int *)	**p** ●
(int *)	**p+1** ●

int

int

an **int array** name

an array :
an aggregate type

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

sizeof(c)

(int [4]) **c** ●

pointer
increment

int

int

int

int

sizeof(*c)

sizeof(c)

(int [4]) **c+1** ●

int

int

int

int
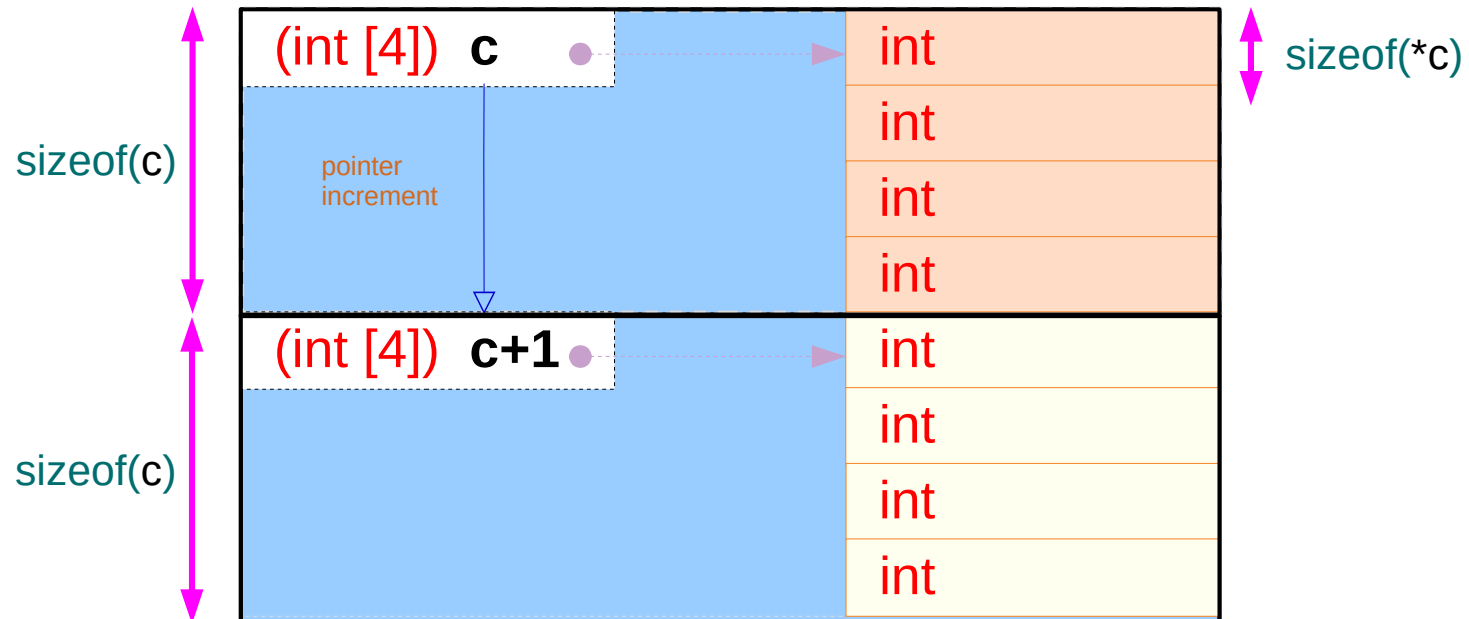
type(int [4]) = type(int [ ]) = (int *)

Virtual pointer p in an aggregated data

abstract data type

sizeof(p)

&p

p   ●   p

*p (=p[0])

sizeof(*p)

p+1

● ● ●

primitive data type

sizeof(p)

&p   p   ●   p   *p (=p[0])   sizeof(*p)

p+1

● ● ●

$$sizeof(p) = sizeof(*p) * N$$

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

Real pointers

abstract data type

p   *p (=p[0])

sizeof(*p)

&p

p

sizeof(p)

p+1

• • •

primitive data type

p   *p (=p[0])

sizeof(*p)

&p

p

sizeof(p)

p+1

• • •
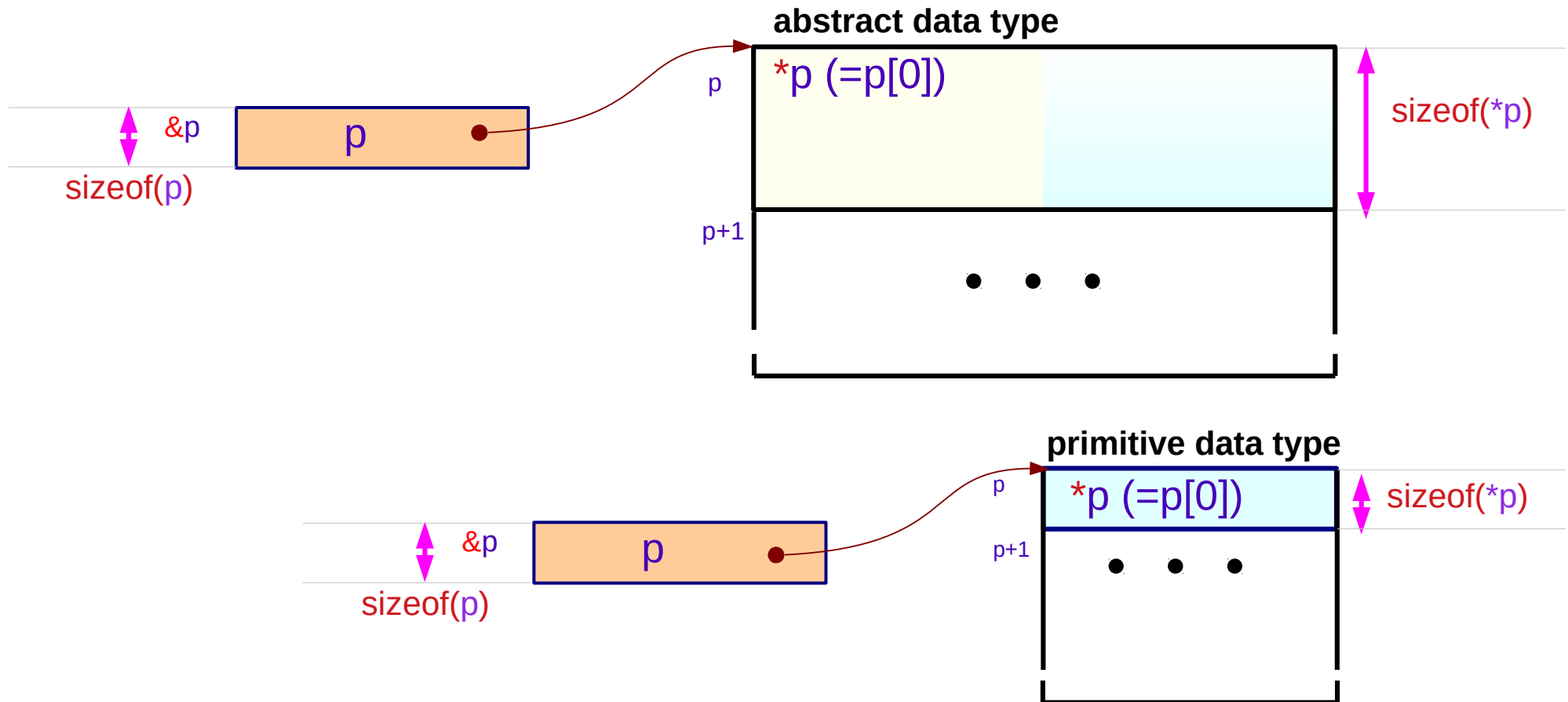
sizeof(p) = 4 / 8 bytes

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

Incrementing a **1-d** array pointer

int   (*p) [4];

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

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

p

Aggregate Type Size   **sizeof(1-d array)**
= 4 * sizeof(int)

p

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

p+**1**   *(p+1)   (*(p+1))[0]
(*(p+1))[1]
sizeof(1-d array)
= 4 * sizeof(int)   (*(p+1))[2]
(*(p+1))[3]

Equivalence : *(p+i) = p[i]

(*(p+i)) : array name

p+i	*(p+i) ● - - - →

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

4*sizeof(int)

$$*(p+i) \equiv p[i]$$

‖ equivalence

p+i	p[i] ● - - - →

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

4*sizeof(int)
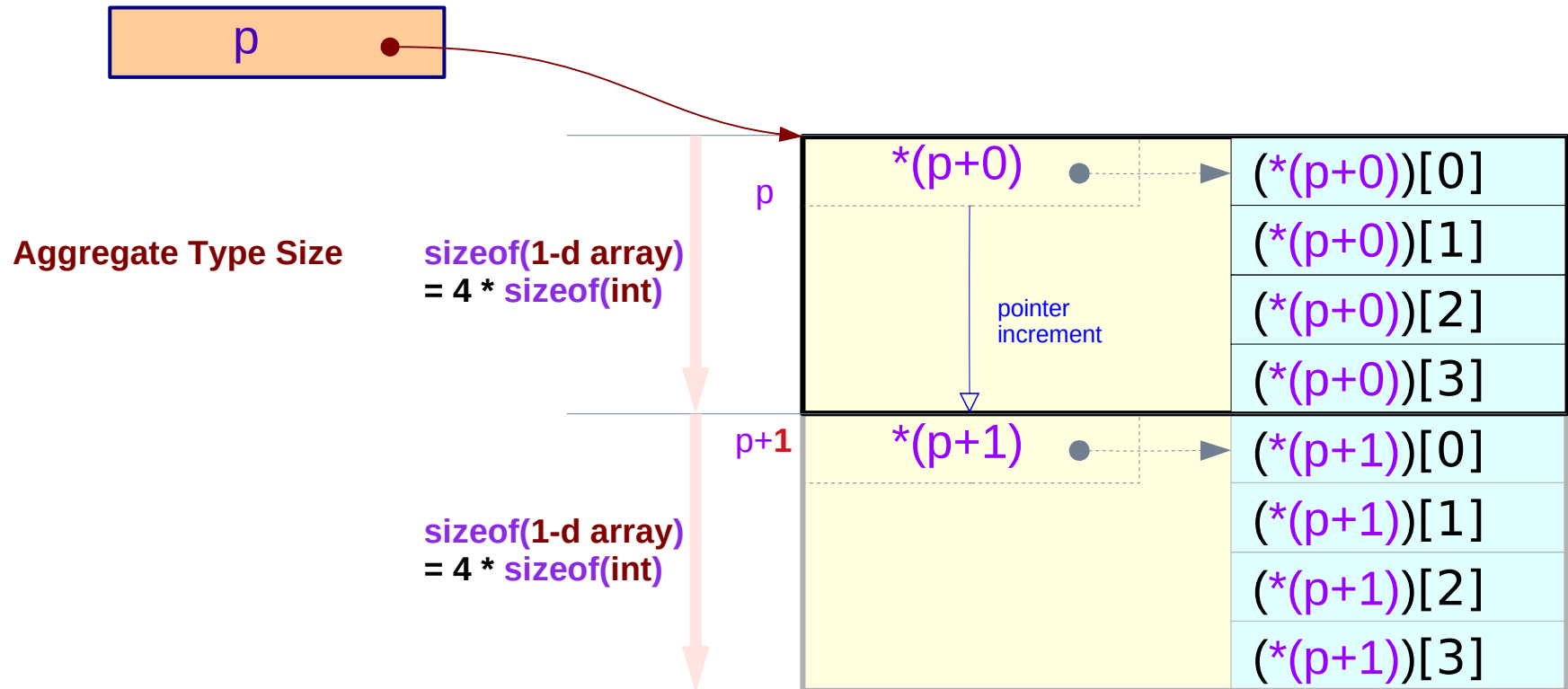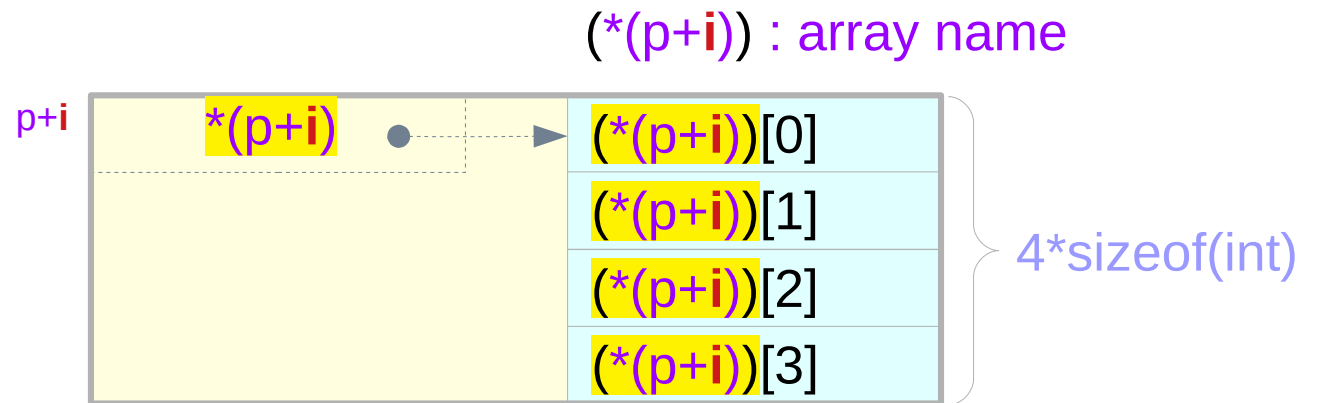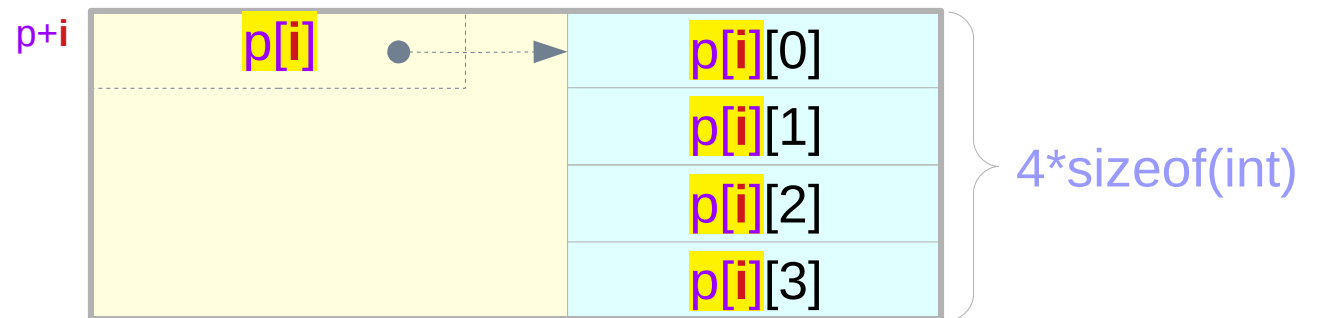
p[i] : **1-d** array name

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

int (*p) [4]   = c ;

p+0
p+1   sizeof(*p)   sizeof(**1-d** array)
p+2   sizeof(*p)   sizeof(**1-d** array)
p+3   sizeof(*p)   sizeof(**1-d** array)

(p+0) ⟷ p[0]   ((p+0))[j] ⟷ p[0][j]
(p+1) ⟷ p[1]   ((p+1))[j] ⟷ p[1][j]
(p+2) ⟷ p[2]   ((p+2))[j] ⟷ p[2][j]
(p+3) ⟷ p[3]   ((p+3))[j] ⟷ p[3][j]
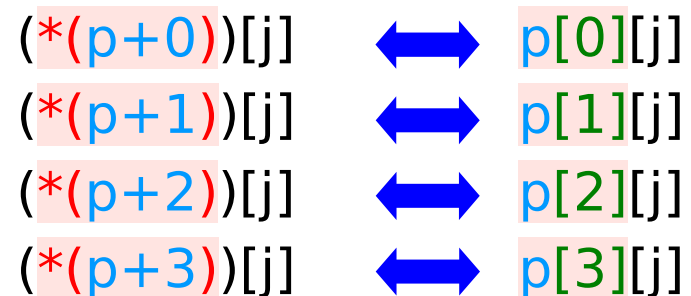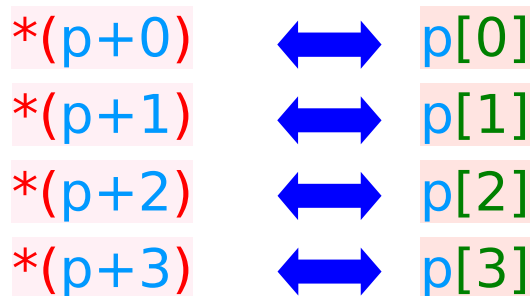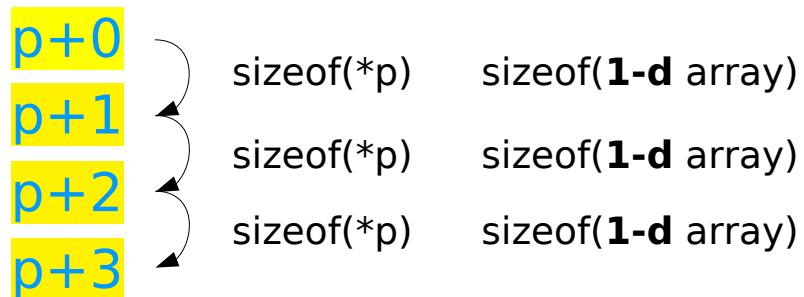
Accessing a **2-d** array using a **1-d** array pointer

int   (*p) [4];

p

p

*(p+**0**)

pointer
increment

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

p+**1**

*(p+**1**)

pointer
increment

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

p+**2**

*(p+**2**)

Aggregate Type Size

sizeof(1-d array)
= 4 * sizeof(int)

pointer
increment

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

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

int   (*p) [4];

p

p[**0**] ●——→ p[**0**][0]

p[**0**][1]

pointer increment

p[**0**][2]

p[**0**][3]

p

p+**1**

p[**1**] ●——→ p[**1**][0]

p[**1**][1]

pointer increment

p[**1**][2]

p[**1**][3]

p+**2**

p[**2**] ●——→ p[**2**][0]

p[**2**][1]

Aggregate Type Size

sizeof(1-d array)
= 4 * sizeof(int)

pointer increment

p[**2**][2]

p[**2**][3]

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

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

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

int **

&p

int * [4]

int *

int

| p = &q[0] ; |

&p p

q

q[0]
q[1]
q[2]

p = &q[0] ;

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

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

Array	Pointer
c[0][0]	p[0][0]
c[0][1]	p[0][1]
c[0][2]	p[0][2]
c[0][3]	p[0][3]
c[1][0]	p[1][0]
c[1][1]	p[1][1]
c[1][2]	p[1][2]
c[1][3]	p[1][3]
c[2][0]	p[2][0]
c[2][1]	p[2][1]
c[2][2]	p[2][2]
c[2][3]	p[2][3]
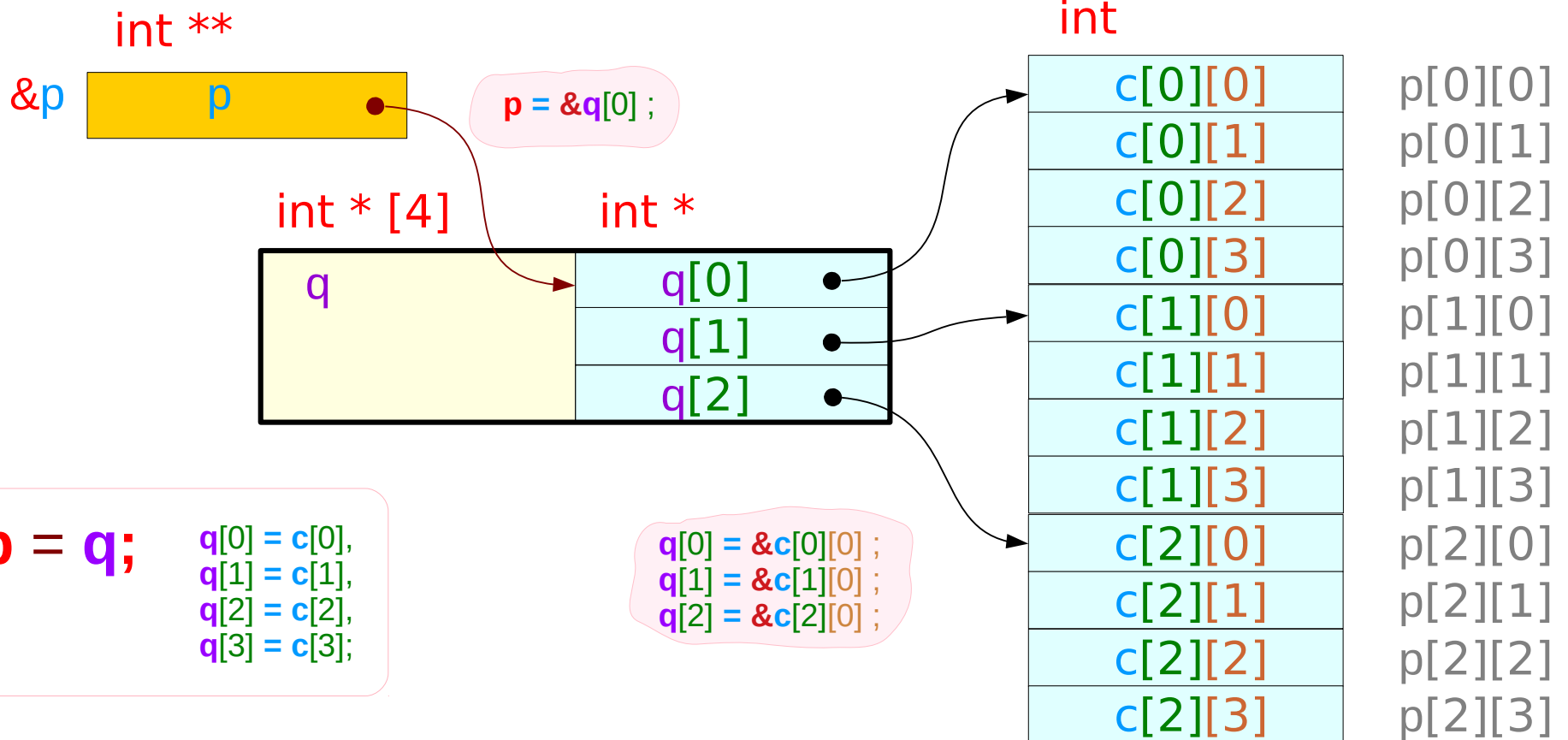
int *

int [N]

int [ ]

Integer pointer and array types – **int *, int [2]**, **int [3]**

int *a;

int *

a	int
a+1	int
a+2	int
a+3	int

int b[2];

int [2]

b	int
	int
	int
	int

int (*)

b	int
b+1	int
b+2	int
b+3	int

int c[3];

int [3]

c	int
	int
	int
	int

int (*)
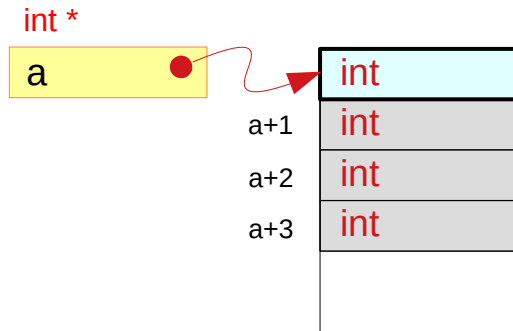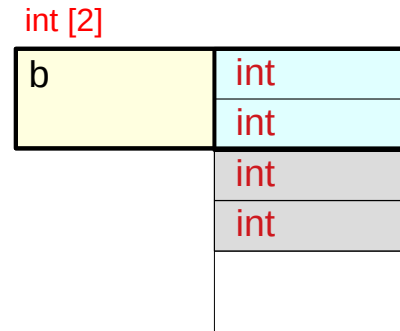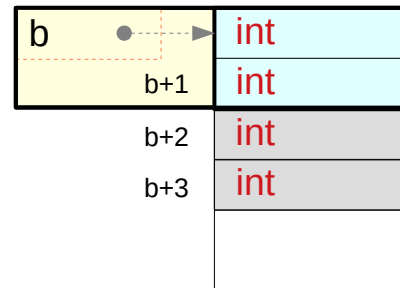
c	int
c+1	int
c+2	int
c+3	int

Integer pointer and array types – **int *, int [2]**, **int [3]**

int *a;

int b[2]

int c[3];

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

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

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

syntactically legitimate
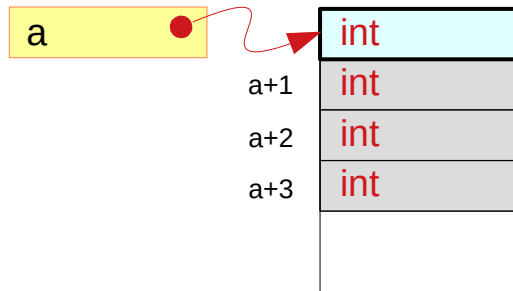
syntactically legitimate

syntactically legitimate

programmers must
ensure their validity

programmers must
ensure their validity

programmers must
ensure their validity

Integer pointer and array types – **int *, int [2]**, **int [3]**

int *a;

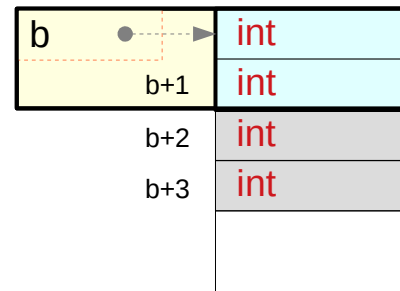|	
a	int
a+1	int
a+2	int
a+3	int
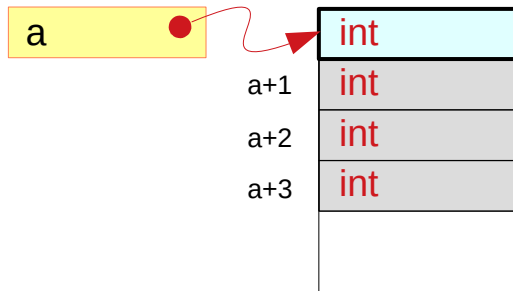
a[0] = *a

type(a)   = int *
type(&a)   = int **

value(&a) ≠ value(a)

sizeof(a)
= pointer size
= sizeof(int *)

int b[2]

b	int
b+1	int
b+2	int
b+3	int

b[0] = *b

type(b)   = int [2]
type(&b)   = int (*) [2]

value(&b) = value(b)

sizeof(b)
= sizeof(*b) * 2
= sizeof(int) * 2

&b and b evaluate
the same address
but have different types
and also different sizes

int c[3];

c	int
c+1	int
c+2	int
c+3	int

c[0] = *c
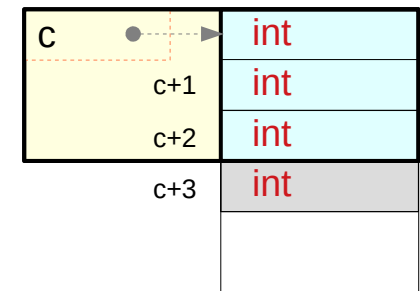
type(c)   = int [3]
type(&c)   = int (*) [3]
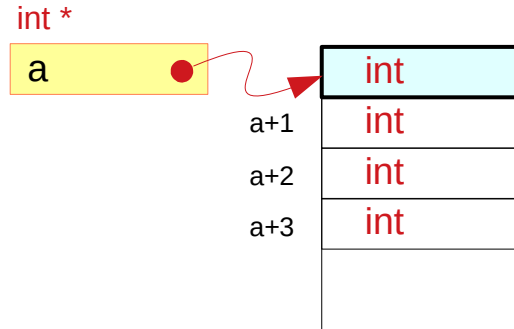
value(&c) = value(c)

sizeof (c)
= sizeof (*c) * 3
= sizeof(int) * 3

&c and c evaluate
the same address
but have different types
and also different sizes

Integer pointer and array types – **int *, int [3]**

int *a;

int *

a		int
	a+1	int
	a+2	int
	a+3	int

sizeof (a) = pointer size

value(&a) ≠ value(a)

the address of pointer variable a is
not equal to the pointed address

real memory location for a

a   :: int *
&a   :: int **

---

int c[3];

int [3]

c		int
	c+1	int
	c+2	int
	c+3	int
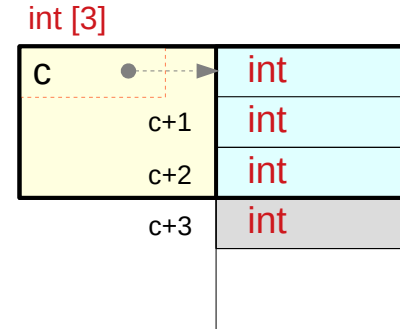
sizeof (c) = sizeof(*c) * 3

value(&c) = value(c)

the starting address of array variable c
is equal to the address of the 1st element

no actual memory location for c

c   :: int [3]
&c   :: int (*) [3]

Integer pointer and array types – **int [3]**

int c[3];

int [3]

c ●

c+1

c+2

c+3

int
int
int
int

sizeof (c) = sizeof(int) * 3

value(&c) = value(c)

type(c)   = int *

type(&c)   = int (*) [3]

---

int c[3];
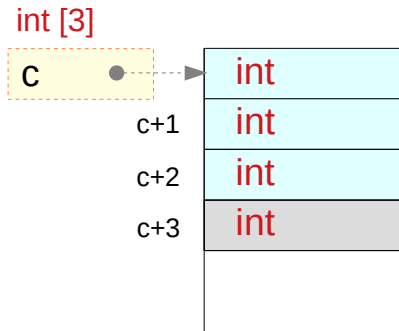
int [3]

c ●
c+1 ●
c+2 ●

c+3

int
int
int
int
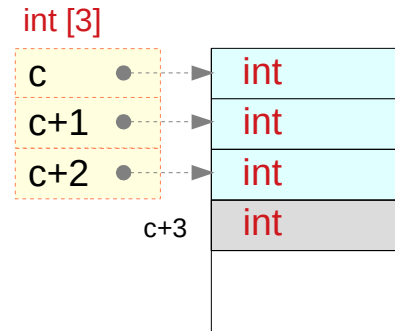
sizeof (c)   = sizeof(*c) * 3 … *leading element*
sizeof (c+1) = pointer size
sizeof (c+2) = pointer size

value(&c)   = value(c) … *leading element*
value(c+1) = value(c) + sizeof(*c) *1
value(c+2) = value(c) + sizeof(*c) *2

type(c)   = int *
type(c+1) = int *
type(c+2) = int *

type(&c)   = int (*) [3]

Pointer to a **1-d** array – (1) type declarations

int   (*p) [4];

1-d array pointer

int   a [4];

$*p \equiv a$ correspondence

$p = \&a$ assignment

points to

$a \equiv \&a[0]$ equivalence
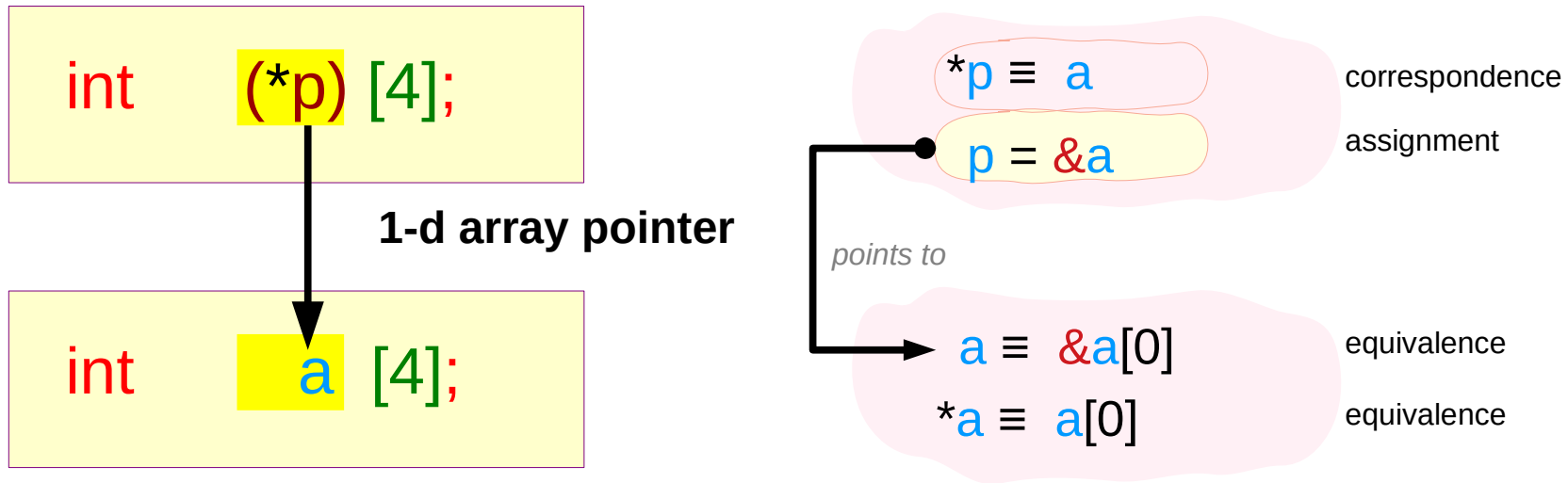
$*a \equiv a[0]$ equivalence

&a and a
print the <u>same</u> address
but have <u>different</u> types

value(&a) = value(a)
type(&a) ≠ type(a)

&a[0]
int (*)[4] ≠ int [4]

those values are evaluated as addresses

Pointer to a **1-d** array – (2) types and sizes

int   a   [4];   assignment   equivalence

int (*p) [4];   p = &a   a ≡ &a[0]

int (*) [4]   int [4] or int (*)   int

sizeof(p) =   sizeof(a) =   sizeof(a[0]) =

4 or 8 bytes   4*4 bytes   4   4 bytes

size of a pointer   not a real pointer a

&p   p   a   •   a[0]

&a   &a[0]   a[1]

                                                                a[2]

                                                                a[3]
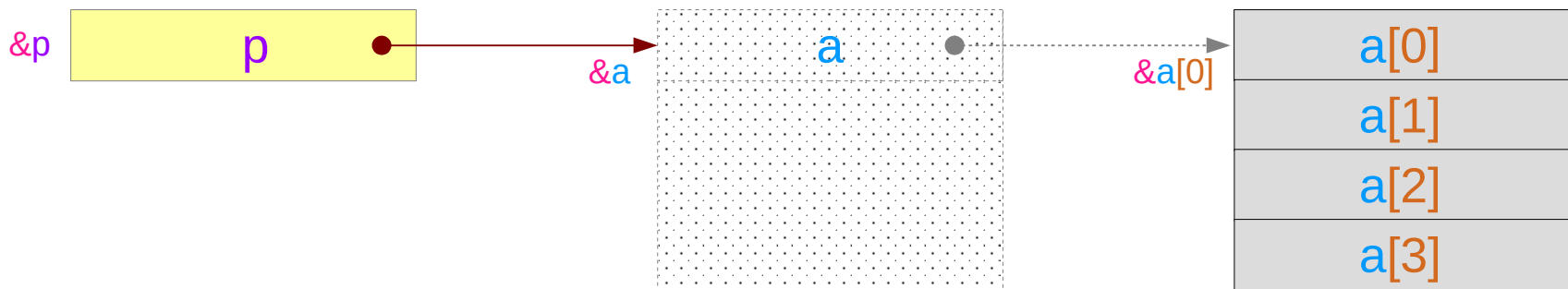
value(&a) = value(a) = value(&a[0])

not a real pointer a

Pointer to a **1-d** array – (3) an assignment & equivalences

&p | p

&a | a &a[0] | a[0]

a ≡ &a[0]   equivalence

&p | p

&a | a a | a[0]

p = &a   assignment

&p | p

p | a a | a[0]

(*p) ≡ a   (*p) ≡ a   substitution

&p | p

p | (*p) (*p) | (*p)[0]

Pointer to a **1-d** array – (4) a chain of pointers view

int (*) [4]   int (*)   int

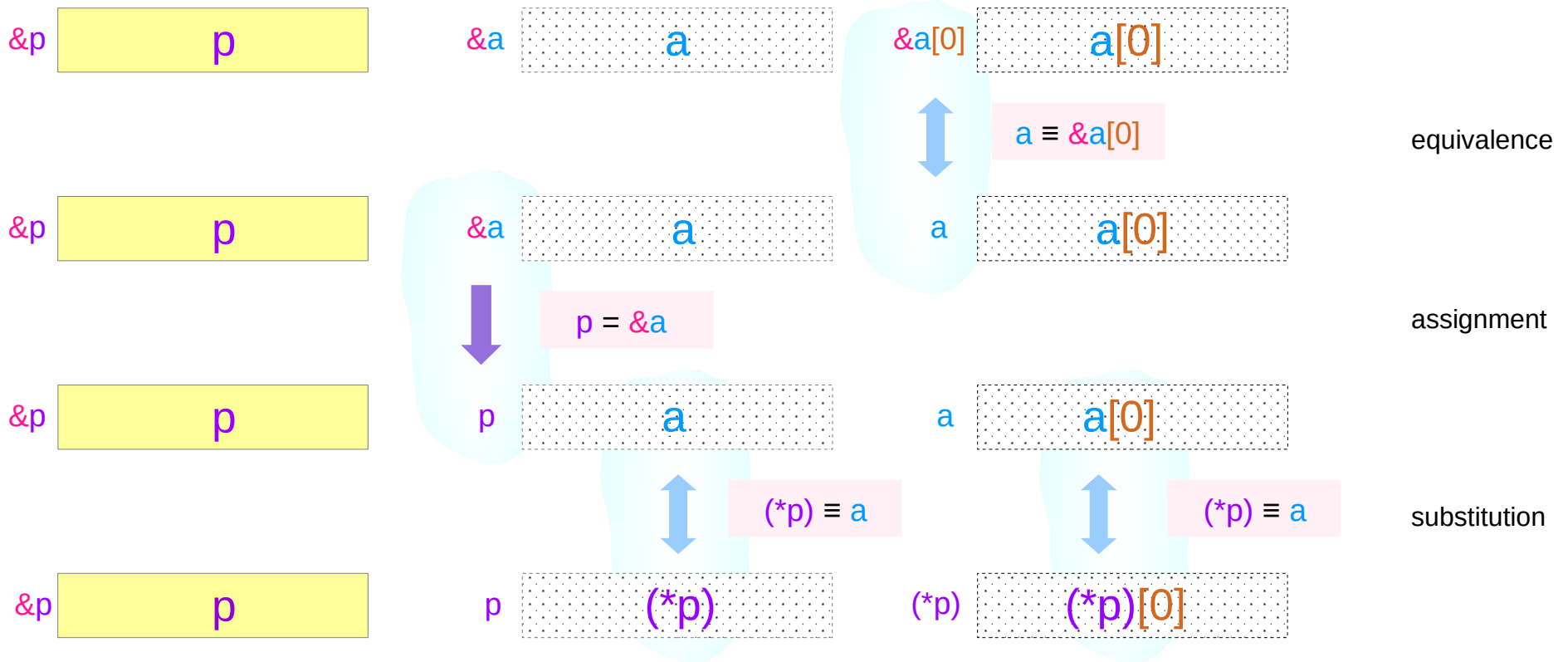&p [p]   p [a]   a [a[0]]

arrow notation   arrow notation

[p]●———→ p [a]●———→ a [a[0]]

substitution   substitution

[p]●———→ p [(*p)]●———→ (*p) [(*p)[0]]

sizeof(p) =   sizeof(a) =   sizeof(a[0]) =
8 bytes   4*4 bytes   4 bytes
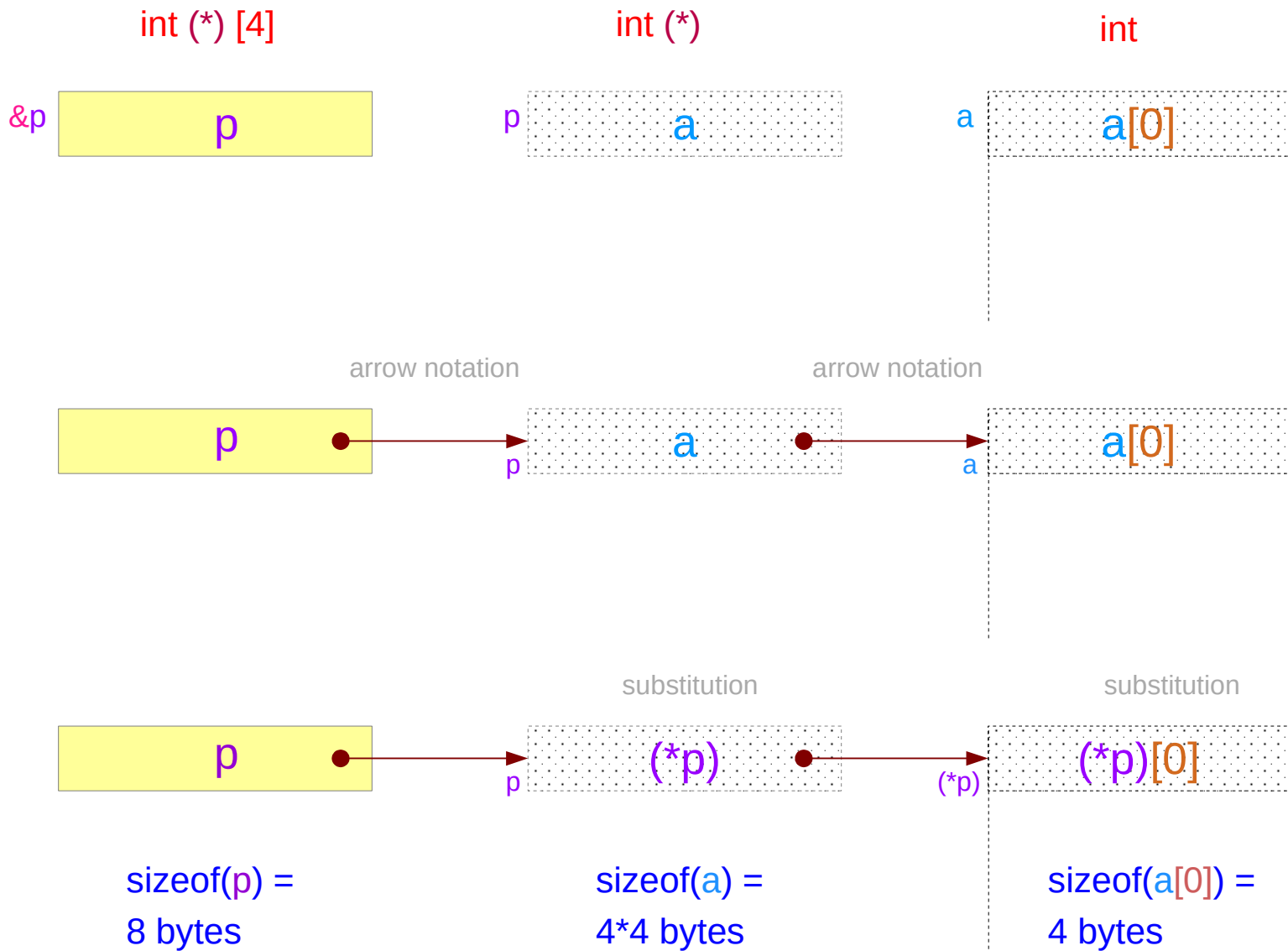
Pointer to a **2-d** array – (1) type declarations

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

2-d array pointer

int   c [3][4];

points to

*q ≡ c   correspondence

q = &c   assignment

equivalence   equivalence

c ≡ &c[0]   *points to*   c[0] ≡ &c[0][0]
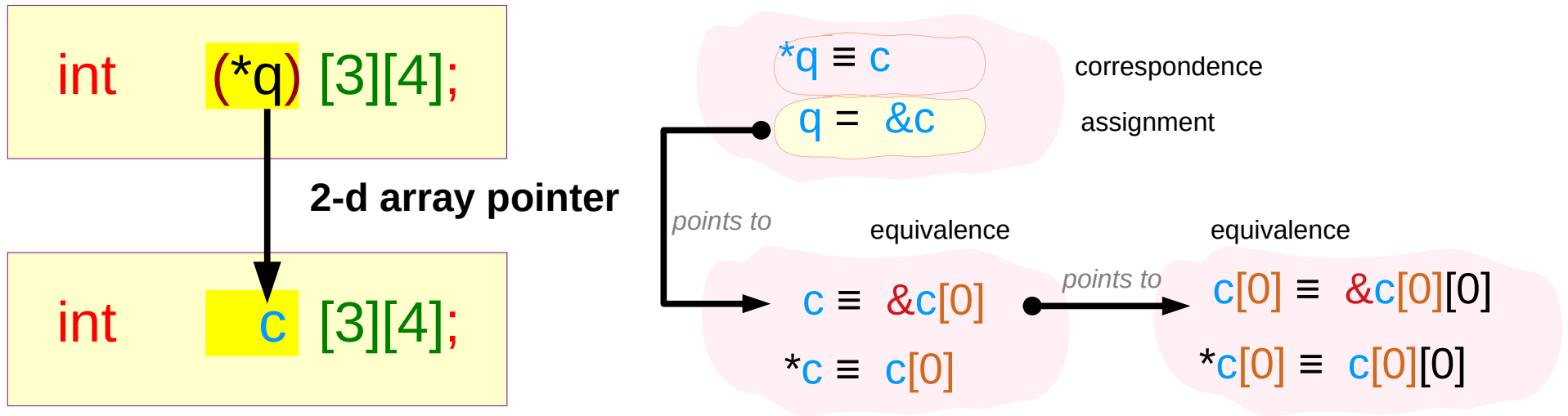
*c ≡ c[0]   *c[0] ≡ c[0][0]

&c, c, c[0]
print the <u>same</u> address
but have <u>different</u> types

value(&c) = value(c) = value(c[0])   &c[0][0]
type(&c) ≠ type(c) ≠ type(c[0])

int (*)[4][4] ≠ int [4]4]
               int (*)[4] ≠ int [4]

those values are evaluated as addresses

Pointer to a **2-d** array – (2) types and sizes

int   c   [3][4];

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

assignment	equivalence	equivalence
q = &c	c ≡ &c[0]	c[0] ≡ &c[0][0]

int (*) [3][4]

sizeof(q) =

8 bytes

size of a pointer

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

sizeof(c) =

3*4*4 bytes   3

not a real pointer c

int [4] or int (*)
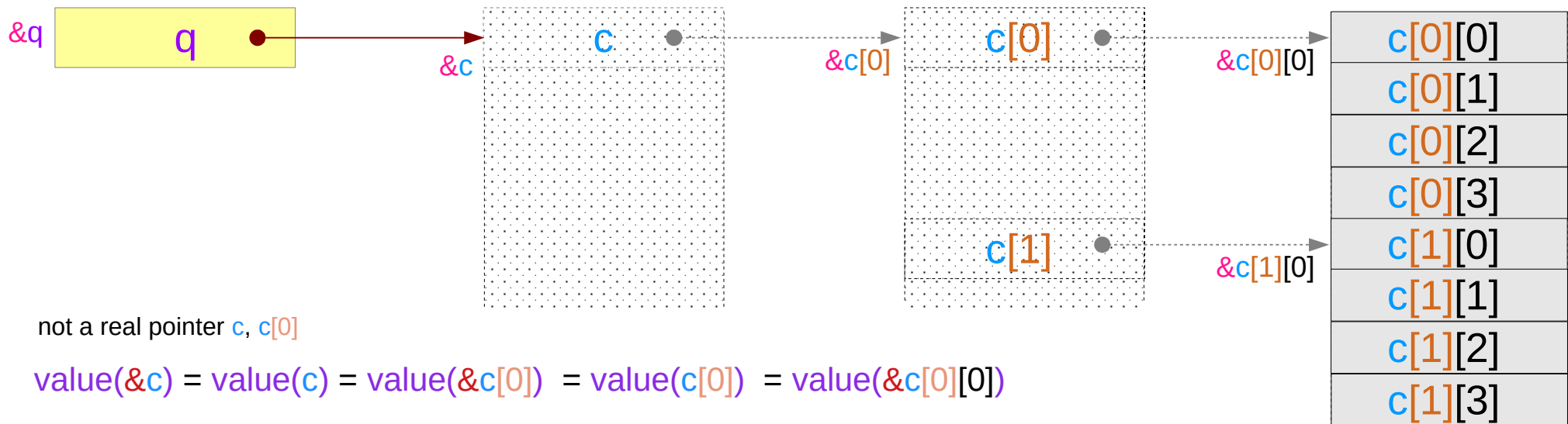
sizeof(c[0]) =

4*4 bytes   4

not a real pointer c[0]

int

sizeof(c[0][0]) =

4 bytes

&q

q ●——————→

&c

c ●

&c[0]

c[0] ●

&c[0][0]

c[1] ●

&c[1][0]

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

not a real pointer c, c[0]

value(&c) = value(c) = value(&c[0]) = value(c[0]) = value(&c[0][0])

q

c

c[0]

c[0][0]

&q | q

&c | c

&c[0] | c[0]

&c[0][0] | c[0][0]

c ≡ &c[0]

c[0] ≡ &c[0][0]

&q | q

&c | c

c | c[0]

c[0] | c[0][0]

q = &c

&q | q

q | c

c | c[0]

c[0] | c[0][0]

(*q) ≡ c

(*q) ≡ c

(*q) ≡ c

&q | q

q | (*q)

(*q) | (*q)[0]

(*q)[0] | (*q)[0][0]

Pointer to a **2-d** array – (4) a chain of pointers view

int (*) [4][4]   int (*) [4]   int (*)   int

&q   q   &c   c   c   c[0]   c[0]   c[0][0]

q   c   c[0]   c[0][0]

       arrow notation   c   arrow notation   c[0]

substitution   substitution   substitution

q   (*q)   (*q)[0]   (*q)[0][0]

          (*q)   (*q)[0]

sizeof(p) =
8 bytes

sizeof(c) =
4*4*4 bytes

sizeof(c[0]) =
4*4 bytes

sizeof(c[0]) =
4 bytes
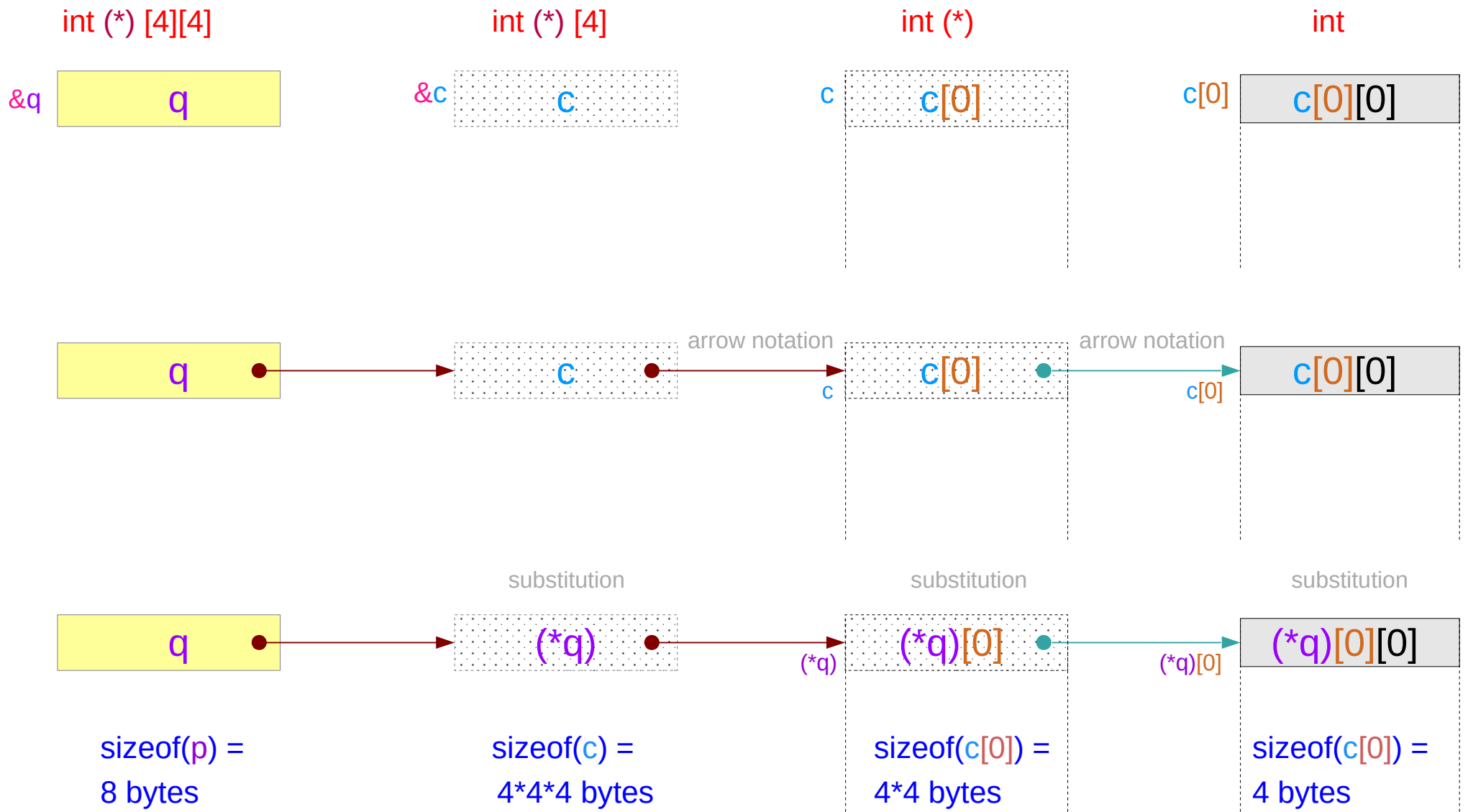
References

[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