Array Pointers (1A)

Young Won Lim
2/8/24

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

Young Won Lim
2/8/24

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

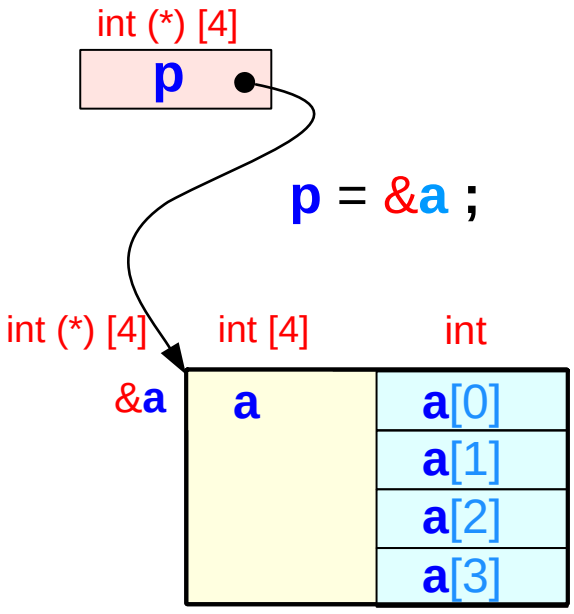
```
#include <stdio.h>
int main(void) {
  int c[3] ;
  printf ("c= %d \n", &c);
}
```

```
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);
```

int [4]	array type
int (*) [4]	array pointer type
int *	integer pointer type
int **	integer double pointer type

Outside and inside array types (1)

```
int a [4];        int (*p) [4];        int *q ;        int **r ;
```

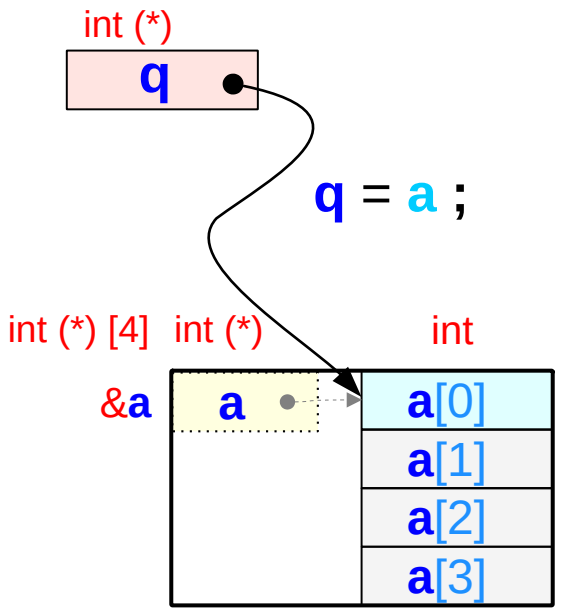


when accessing an array

outside of an array a

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

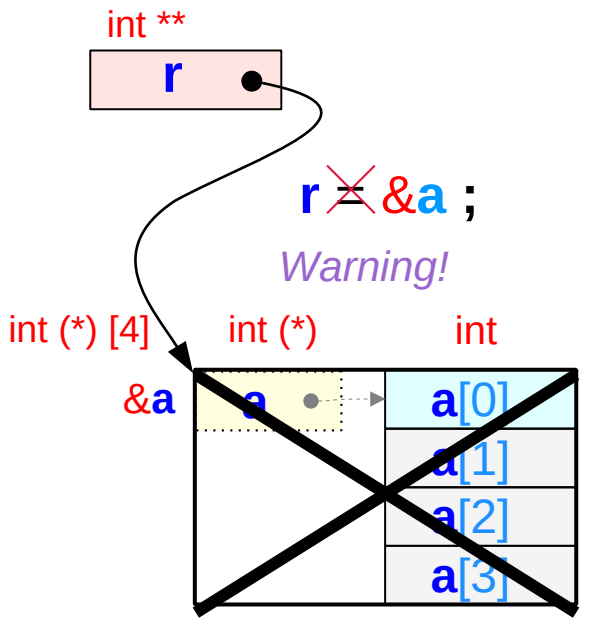
type(&**a**) = int (*) [4]



when accessing an element

inside of an array a

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



when accessing an array

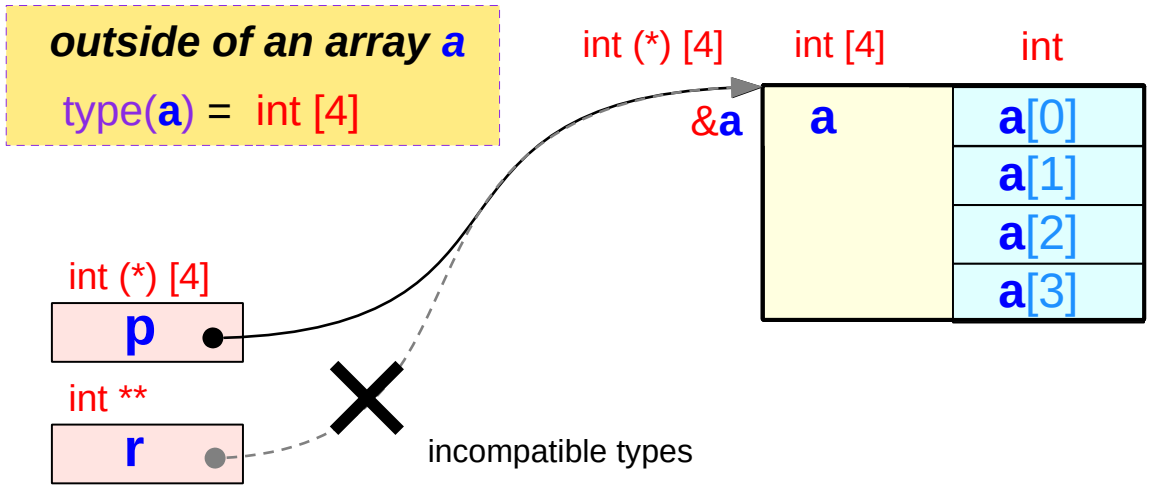
outside of an array a

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

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

type(&**a**) ≠ int **

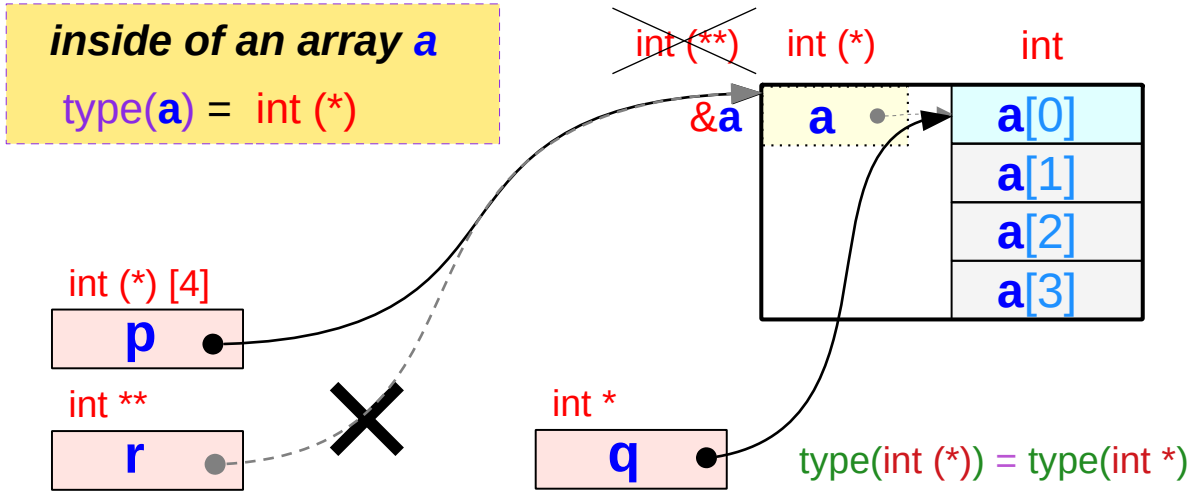
Outside and inside array types (2)



when an array **a** is referenced,
type(**a**) must be an array type,
in this example, int [4], therefore
type(&**a**) must be always int (*) [4]

int ** ⇏ int (*) [4]

*cannot point to an **outside array type***
due to incompatible type



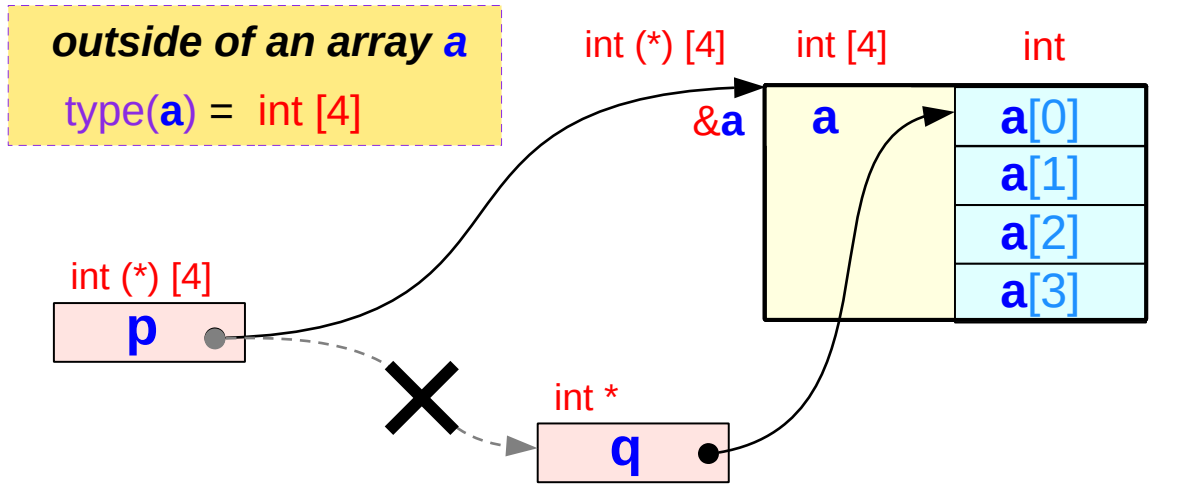
when an element of an array **a** is
referenced, type(**a**) can be relaxed
from an array type to a pointer type
here, int [4] can be relaxed to int (*)

but type(&**a**) must be always int (*) [4]

int ** ⇏ int (*)

*cannot point to an **inside array type***

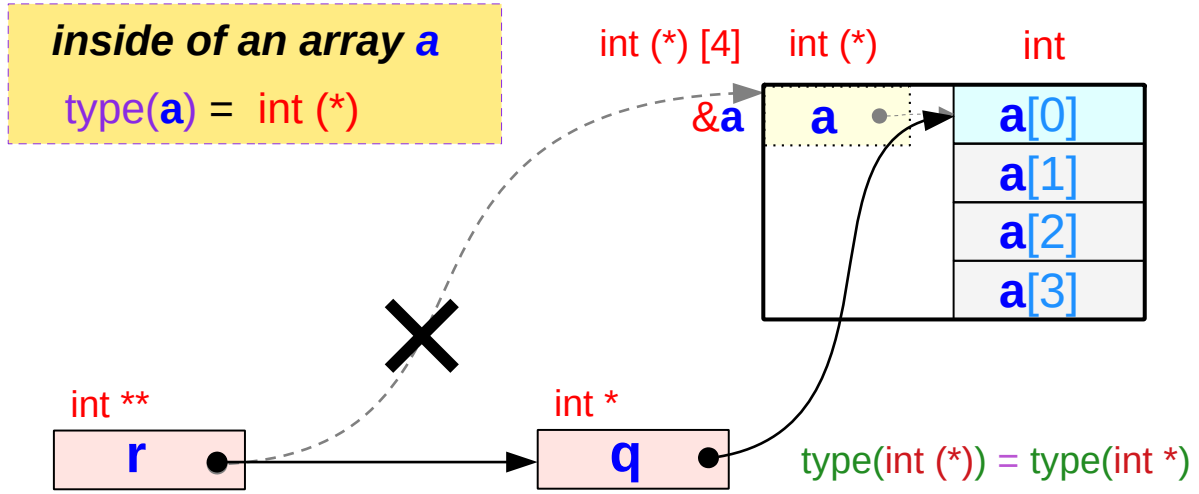
Outside and inside array types (3)



only abstract data type (int [4])
can be referenced by a pointer

int (*)[4] → int [4] *outside array type*

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



relaxed data types (int (*))
cannot be referenced by a pointer

int ** ✗→ int [4] *outside array type*

int ** ✗→ int (*) *inside array type*

int ** type can point
neither int [4] (incompatible type)
nor int (*) (inside array type)

int * an integer pointer

int [2] a **1-d** array with 2 integer elements

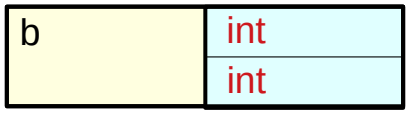
int [3] a **1-d** array with 3 integer elements

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

int ***a**;

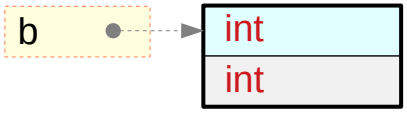
int **b**[2];

int [2] **outside type**



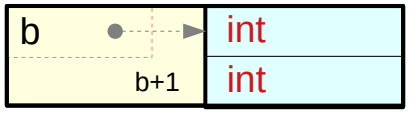
abstract data b *– size*

int (*) **inside type**



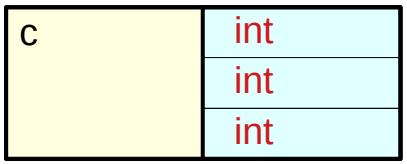
virtual pointer b *– address*

int [2] – size **outside type**
int (*) – address **inside type**



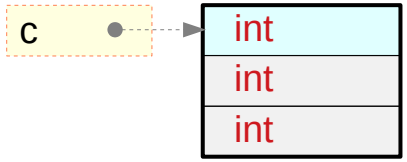
int **c**[3];

int [3] **outside type**



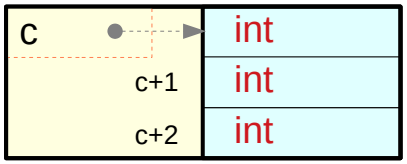
abstract data c *– size*

int (*) **inside type**



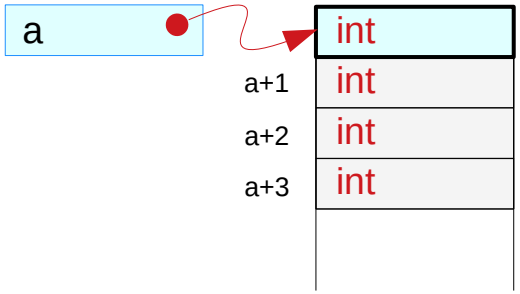
virtual pointer c *– address*

int [3] – size **outside type**
int (*) – address **inside type**



Incrementing pointers – **int *, int [2], int [3]**

int ***a**;

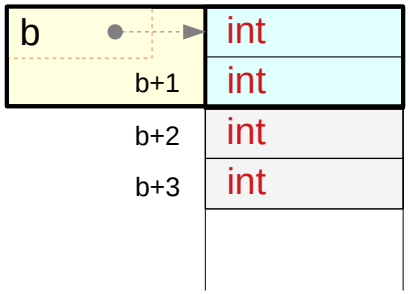


a[0] = *a
a[1] = *(a+1)
a[2] = *(a+2)
a[3] = *(a+3)

syntactically legitimate

programmers must
ensure their validity

int **b**[2]

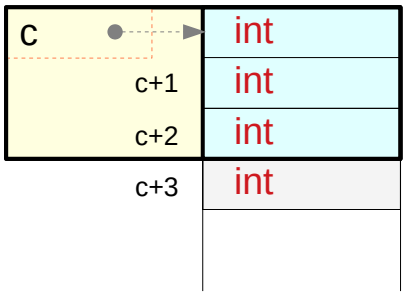


b[0] = *b
b[1] = *(b+1)
b[2] = *(b+2)
b[3] = *(b+3)

syntactically legitimate

programmers must
ensure their validity

int **c**[3];



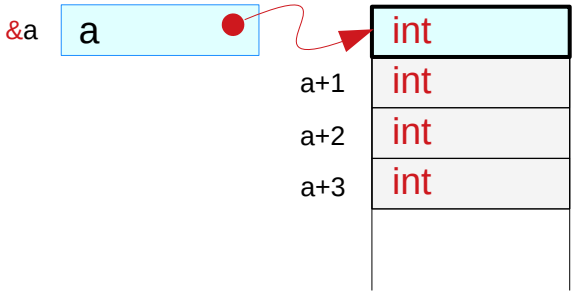
c[0] = *c
c[1] = *(c+1)
c[2] = *(c+2)
c[3] = *(c+3)

syntactically legitimate

programmers must
ensure their validity

Types and sizes – int *, int [2], int [3]

int ***a**;



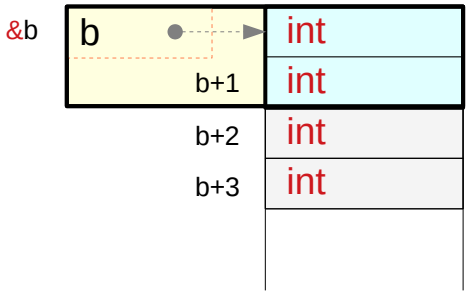
type(&a) = int **
type(a) = int *

type(*a) = int

value(&a) ≠ value(a)

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

int **b**[2]



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

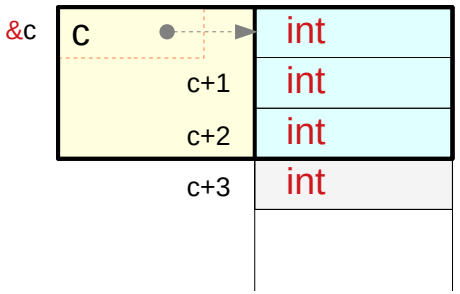
type(*b) = int

value(&b) = value(b) address replication

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

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

int **c**[3];



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

type(*c) = int

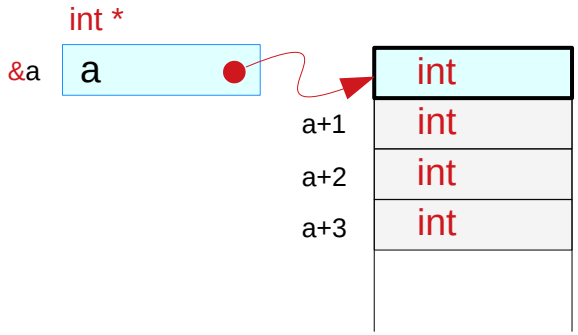
value(&c) = value(c) address replication

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

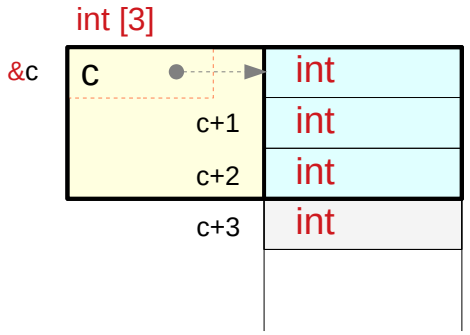
address replication
&c and c evaluate the same address
but have different types and sizes

Real pointer and virtual pointer types – **int *, int [3]**

int ***a**;



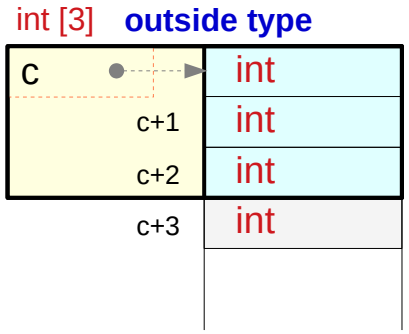
int **c**[3];



sizeof (a) = pointer size	int *	sizeof (c) = sizeof(*c) * 3	int [3]
value(&a) ≠ value(a)	int *	value(&c) = value(c)	int (*)
the address of **pointer** variable a is not equal to the pointed address real memory location for a		the starting address of **array** variable c is equal to the address of the 1st element no actual memory location for c	**address replication**
type(a) = int *		type(c) = int [3] outside type int (*) inside type	
type(&a) = int **		type(&c) = int (*) [3]	

Virtual pointer types of an array– **int [3]**

int **c**[3];



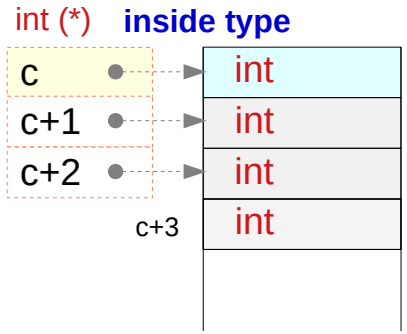
sizeof (c) = sizeof(int) * 3

value(&c) = value(c) ◄ address replication ►

type(c) = int [3] outside type

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

int **c**[3];



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

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

Array Pointers v.s. Pointer Arrays

Array pointers and pointer arrays

1. array pointer p – a pointer to an array of int [4] type

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

2. pointer array x – an array of pointers of int * type

```
int   *x  [4] ;
```

Types of array pointer **p** and pointer arrays **y**

array pointer:
a pointer to an array

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

(*p) is an array with 4 elements
each element is an integer
p is a pointer to such an array

the type of **(*p)** : **int [4]**
the type of **p** : **int (*) [4]**

pointer array:
an array of pointers

```
int * y [4] ;
```

y is an array with 4 elements
each element is an integer pointer

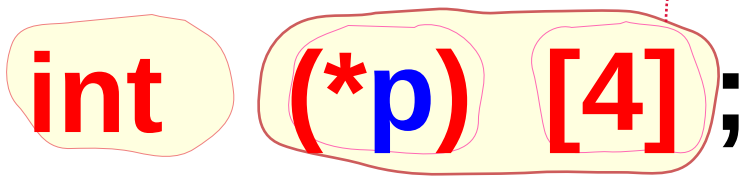
the type of **y** : **int * [4]**

[ ] has a higher priority than ***

(*p) must be grouped with **[4]**
y must be grouped with **[4]**

Types of array elements **(*p)[i]** and **y[i]**

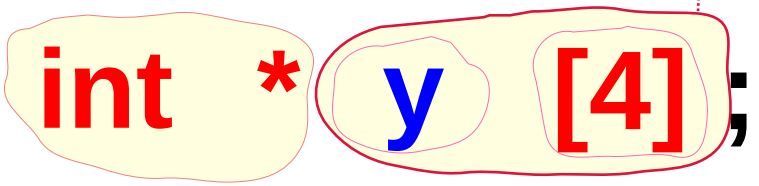
array pointer:
a pointer to an array



(*p)[i] in a statement

(*p) is an array with 4 elements
(*p)[0], (*p)[1], (*p)[2], (*p)[3]
the type of elements : **int**

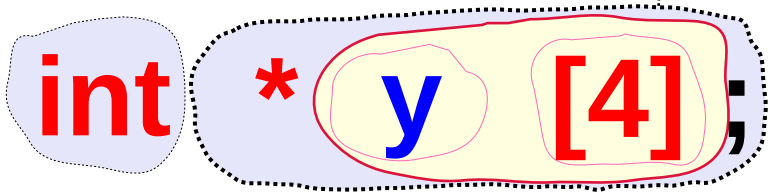
pointer array:
an array of pointers



y[i] in a statement

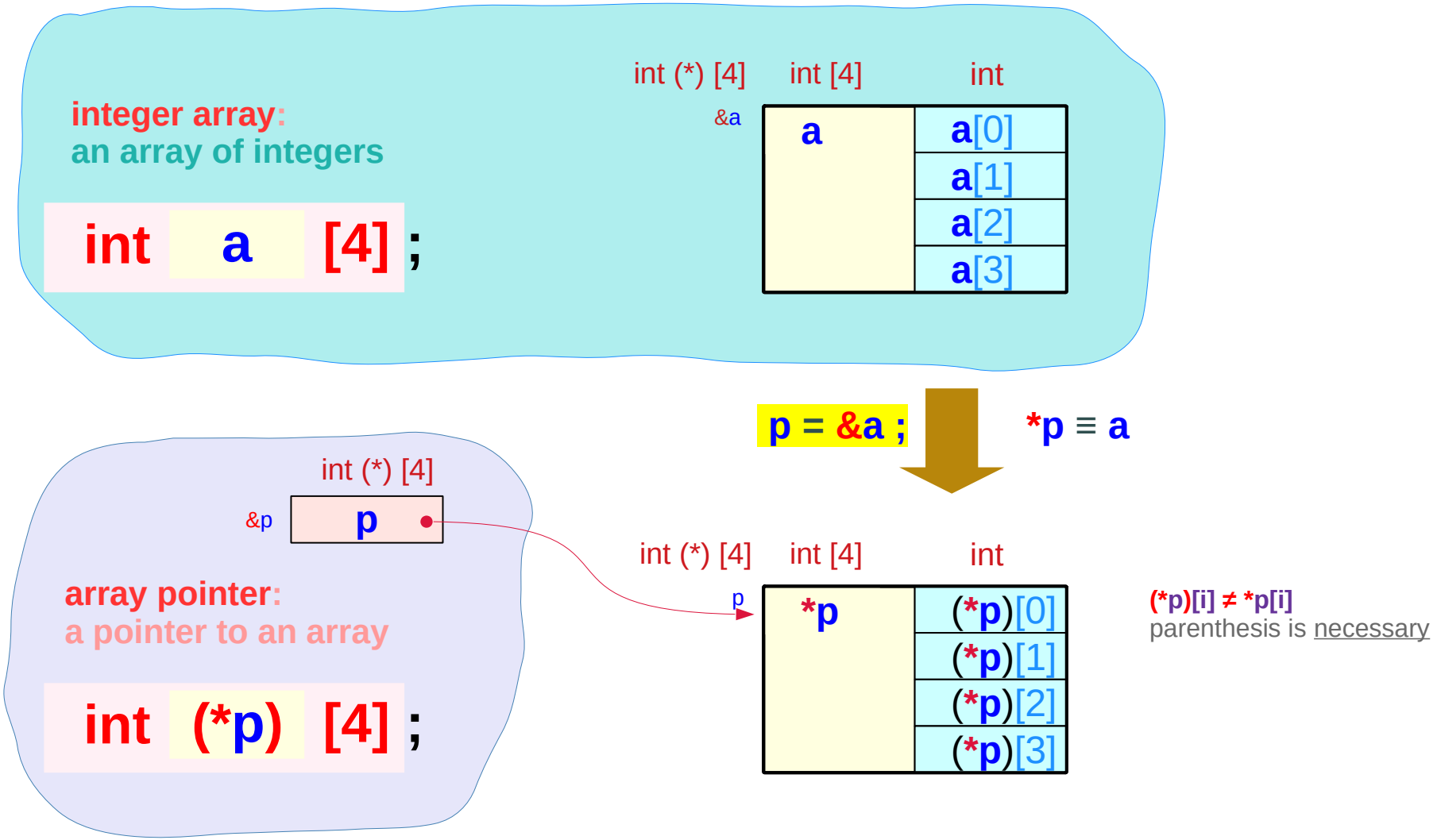
y is an array with 4 elements
y[0], y[1], y[2], y[3]
the type of elements : **int ***

***y[i]** in a statement



y is an array with 4 elements
***y[0], *y[1], *y[2], *y[3]**
the type of dereferenced elements : **int**

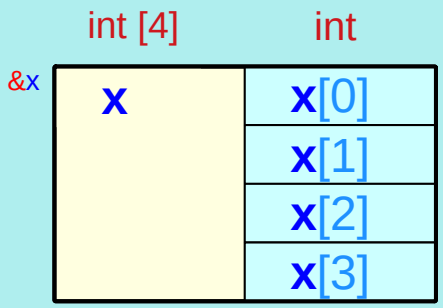
Array pointer **p**, array ***p**, integer **(*p)[i]**



Pointer array **y**, integer pointer **y[i]**, integer ***y[i]**

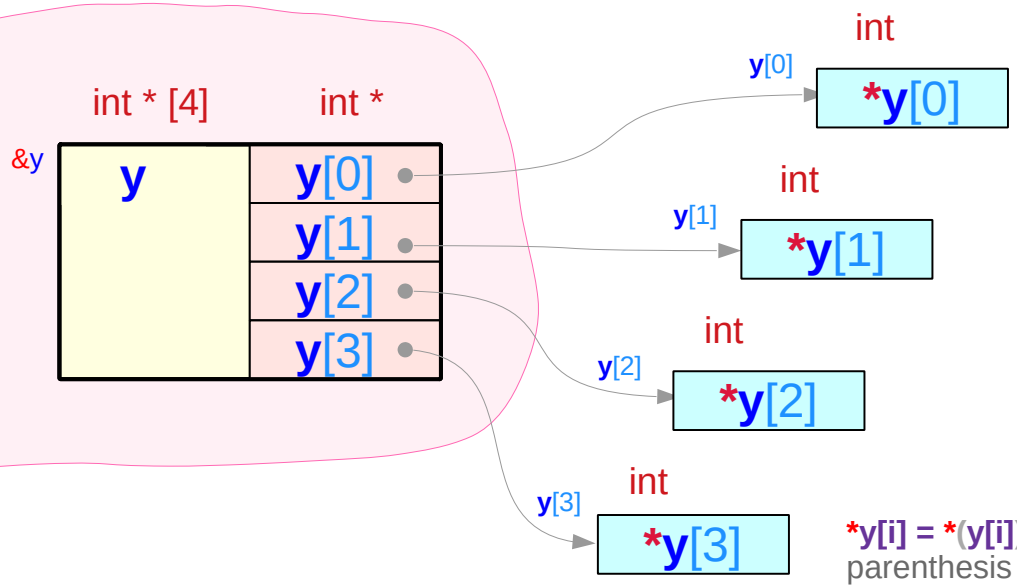
integer array:
an array of integers

```
int  x [4] ;
```



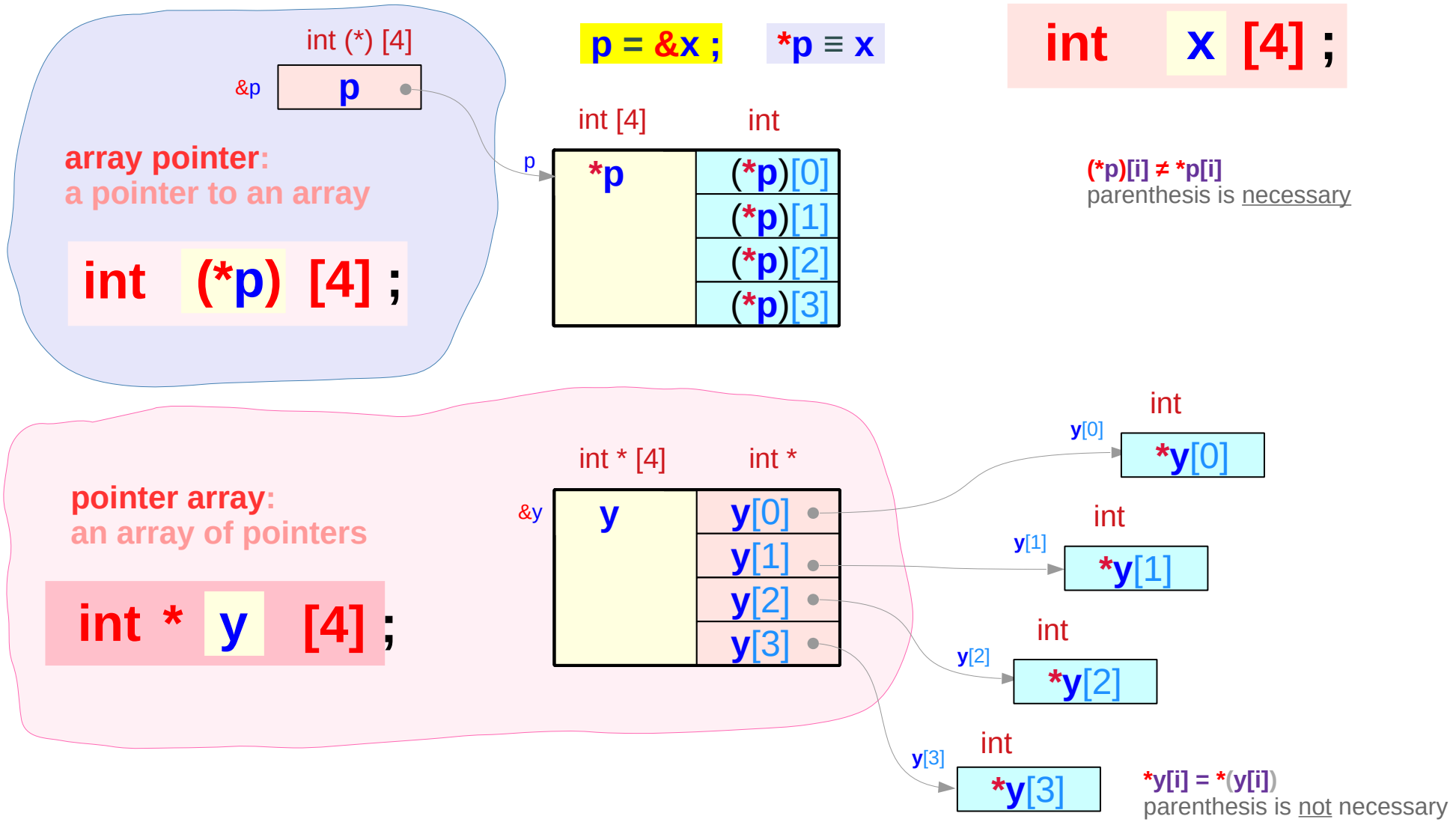
pointer array:
an array of pointers

```
int * y [4] ;
```

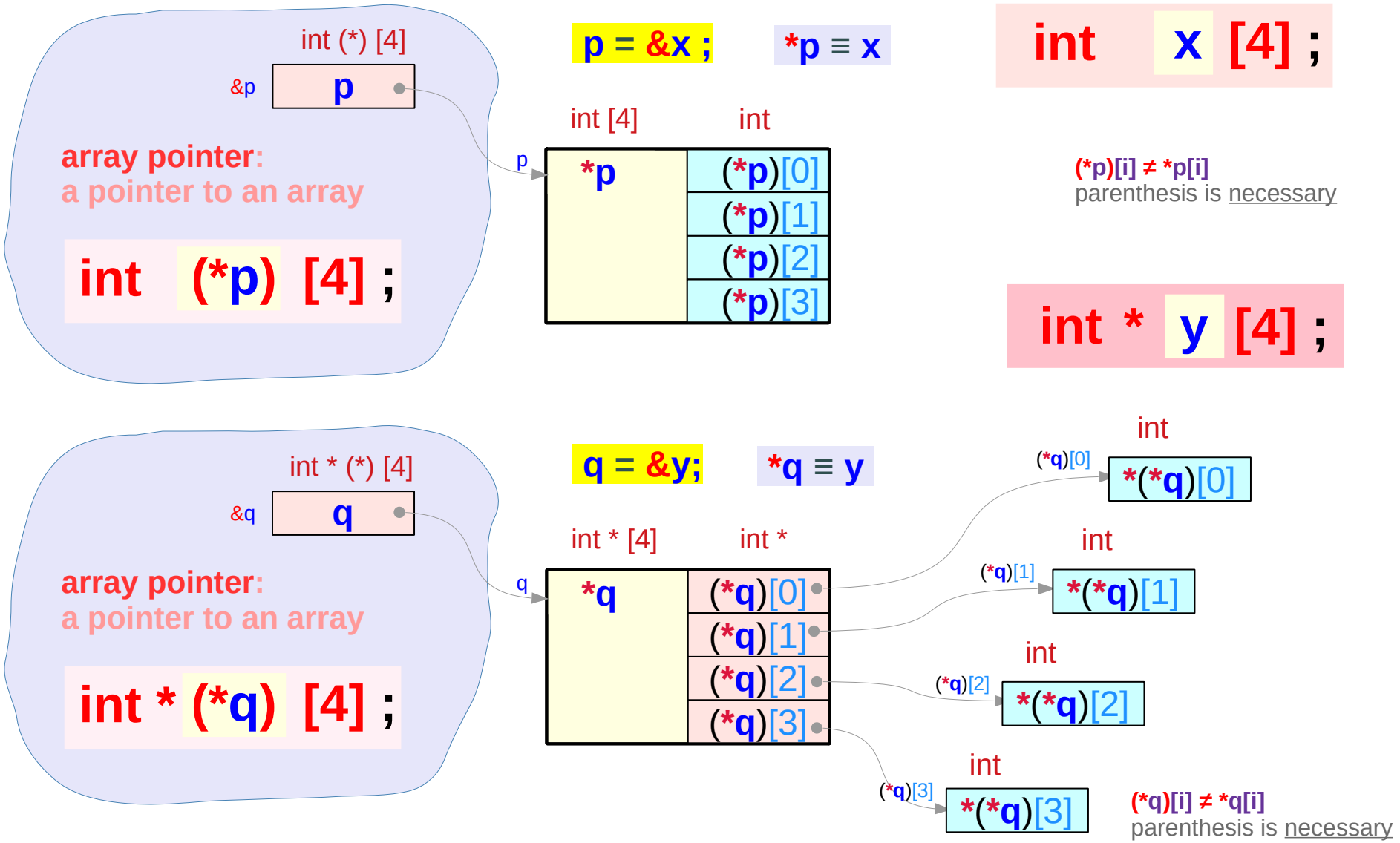


***y[i] = *(y[i])**
parenthesis is not necessary

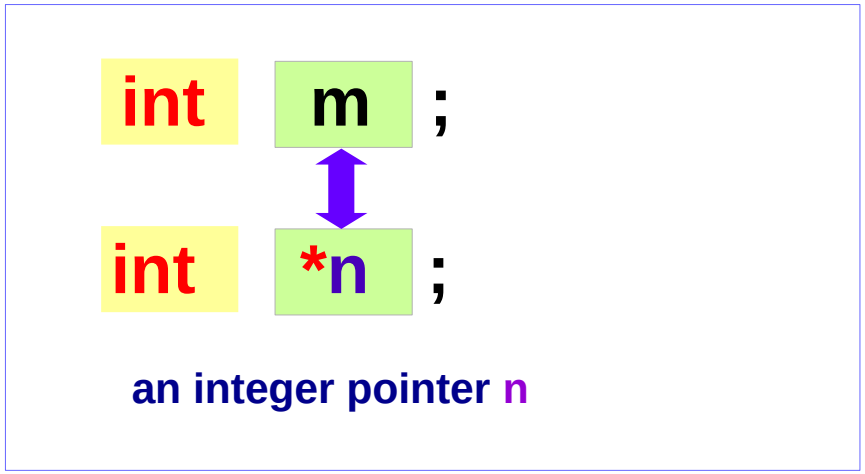
Array pointers **p** v.s. pointer arrays **y**



Array pointers **p** v.s. array pointer **q**



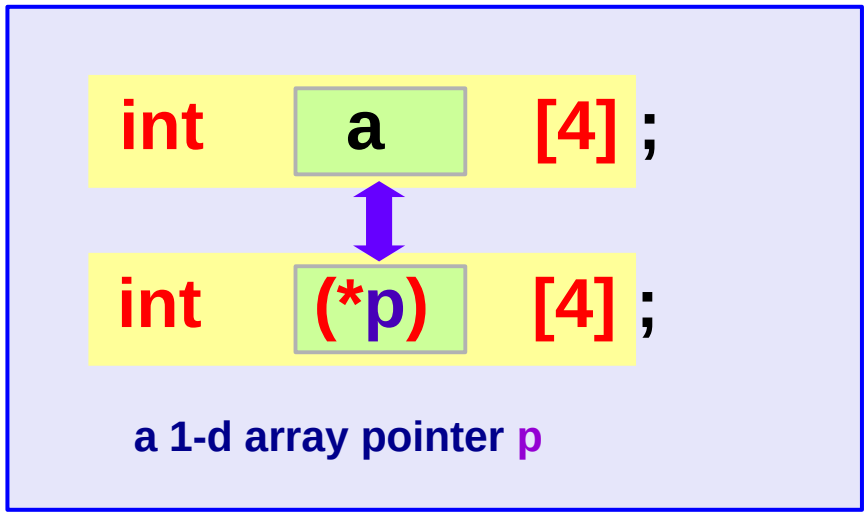
Correspondence of pointer dereference *n, *p, *fp

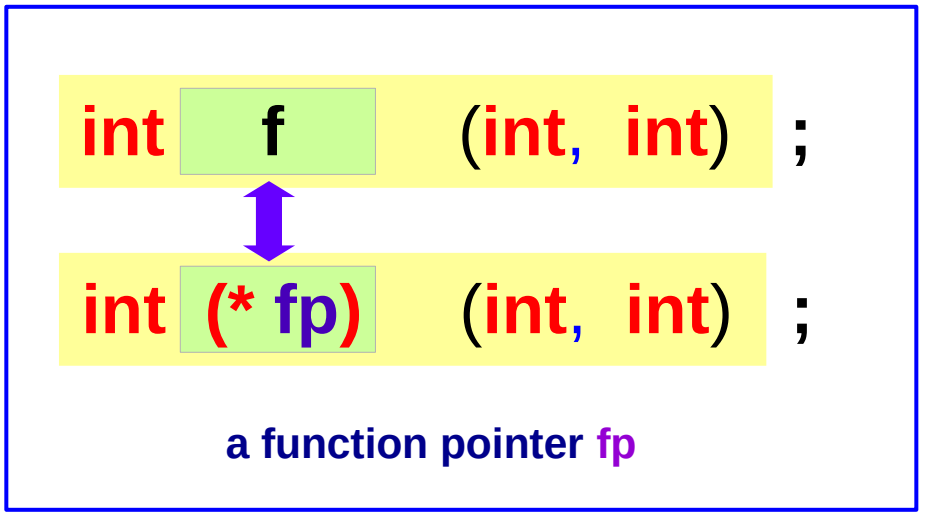


m and ***n** : an **integer** (**int**)

a and ***p** : a **1-d array** (**int [4]**)
with 4 integer elements

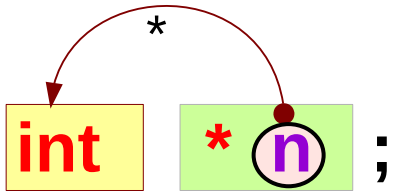
f and ***fp** : a **function** (**int (int, int)**)
taking two integers
returning an integer





Variables and pointed types

n points to a integer
***n** dereference of **n** (int)



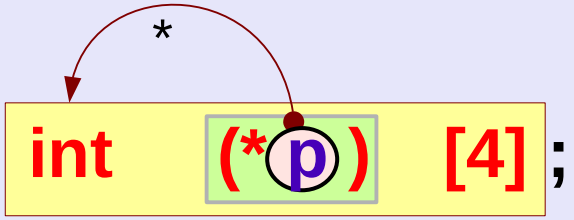
an integer pointer n int *

n points to an **integer**

p points to a **1-d array**
that has 4 integer elements

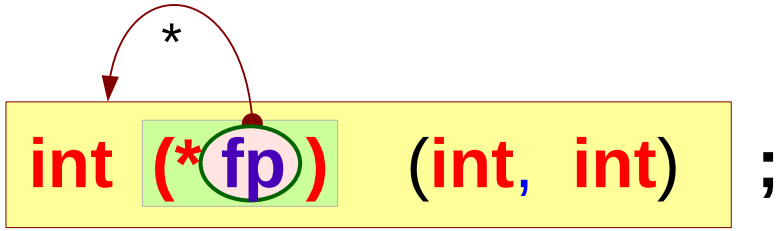
fp points to a **function**
that takes two integers
returns an integer

p points to a 1-d array
***p** dereference of **p** (int [4])



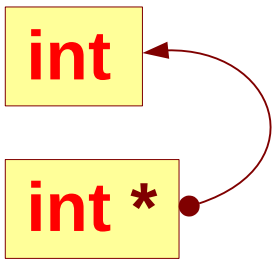
a 1-d array pointer p int (*) [4]

fp points to a function
***fp** dereference of **fp** (int (int, int))



a function pointer fp int (*) (int, int)

Referring and referenced types



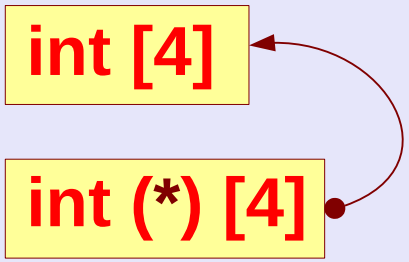
4 byte data

an integer pointer

int * type variable
points to an **int** type data

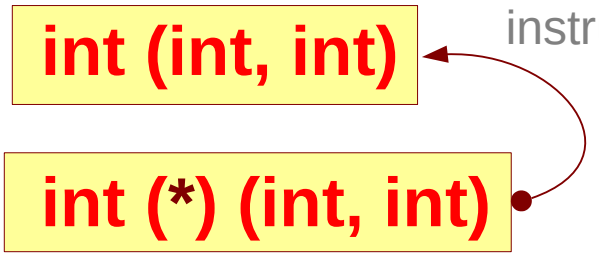
int (*) [4] type variable
points to an **int [4]** type data

int (*) (int, int) type variable
points to an **int (int, int)** type function



4*4 byte data

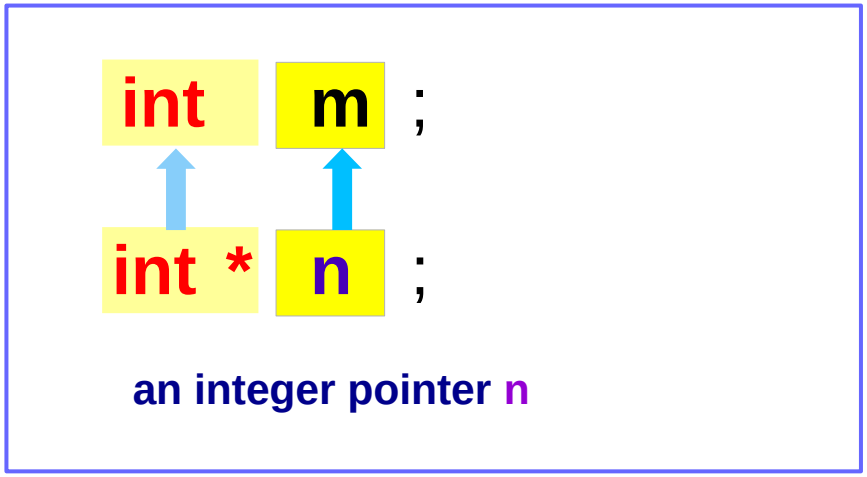
a 1-d array pointer

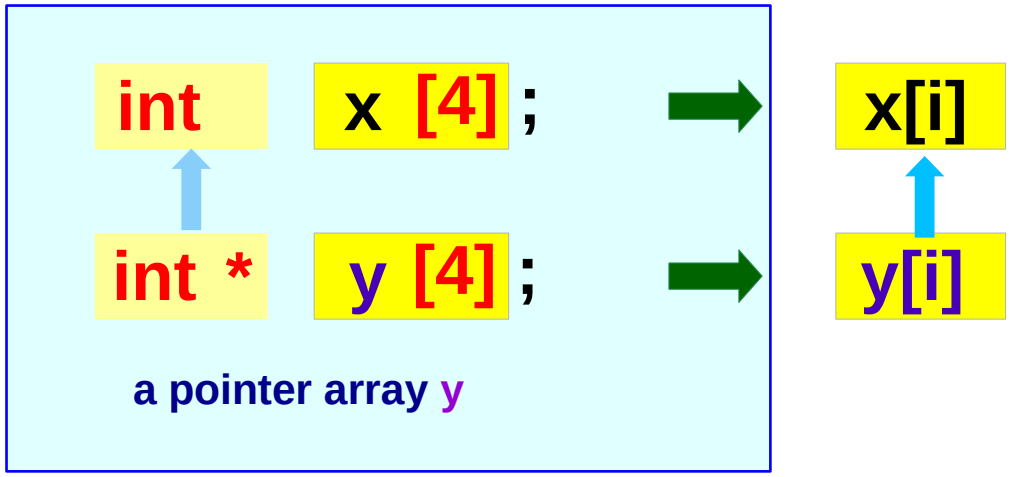


instructions

a function pointer

Correspondence of pointer reference **n**, **y[i]**





m : an **integer**

n : an **integer pointer**

n can point to **m**

n = &m ;

***n ≡ m**

x[i] : an **integer**

y[i] : an **integer pointer**

y[i] can point to **x[i]**

y[i] = &x[i] ;

***y[i] ≡ x[i]**

[ ] has a higher priority than ***

***y[i] = *(y[i])** unnecessary parenthesis

Element types of a pointer array **y**

int * y [4] ;

[ ] has a higher priority than ***

y must be grouped with **[4]**

int x [4] ; ➡ **x[i]**

x is a **1-d array** with 4 elements

each element **x[i]**, **i=0**, **1**, **2**, **3**
has the type of an **integer** (**int**)

int * y [4] ; ➡ **y[i]**

integer pointer

y is a **1-d array** with 4 elements

each element **y[i]**, **i=0**, **1**, **2**, **3**
has the type of an **integer pointer**
(**int ***)

int *y [4] ; ➡ ***y[i]**

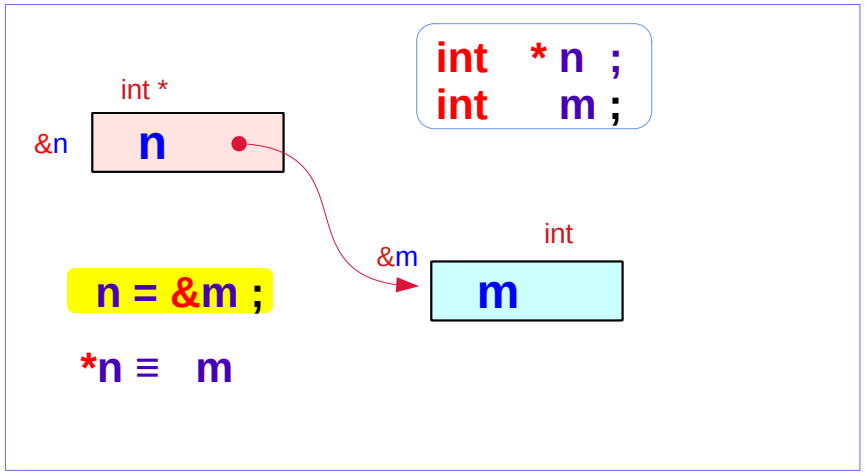
integer

y is a **1-d array** with 4 elements

the dereference of each element
***y[i]**, **i=0**, **1**, **2**, **3** is an **integer**

here, ***y[i]** = ***(y[i])**

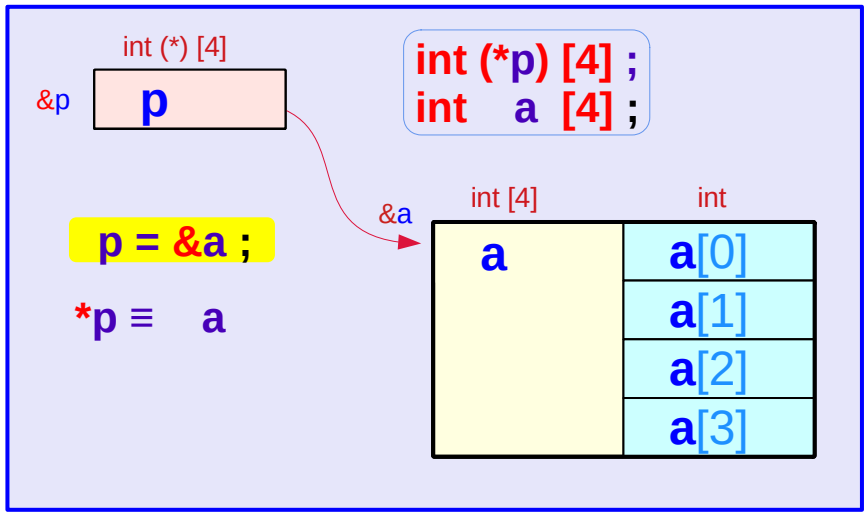
Assigning pointer variables **n**, **p**, **fp**

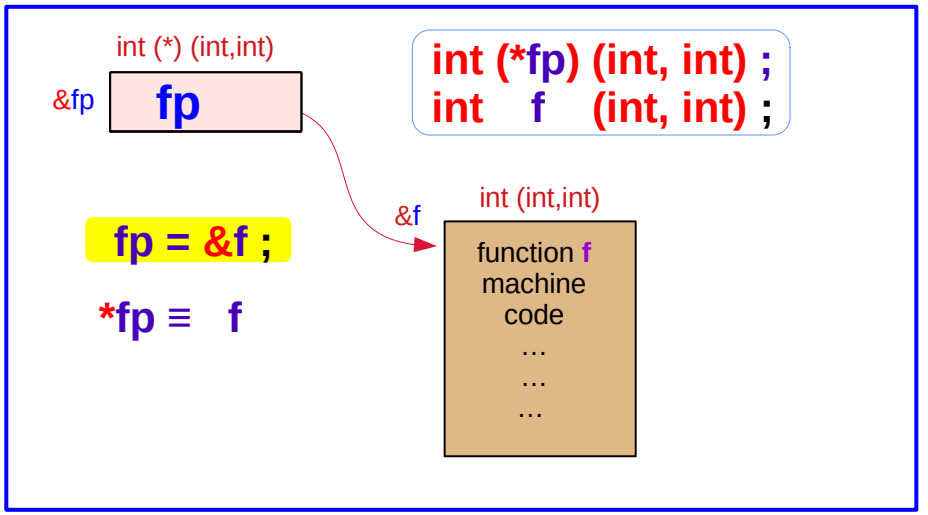


n : a **pointer to** an **integer** **int ***

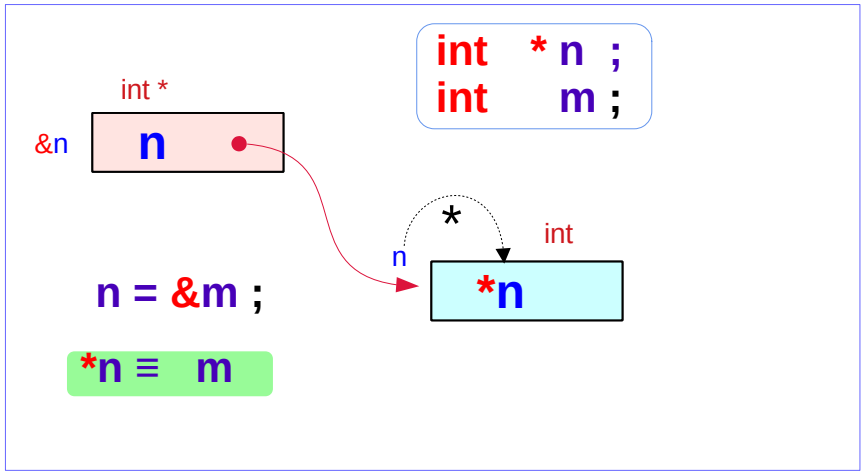
p : a **pointer to** a **1-d array** **int (*) [4]**
with 4 integer elements

fp : a **pointer to** a **function** **int (*) (int, int)**
that takes two integers
and returns an integer





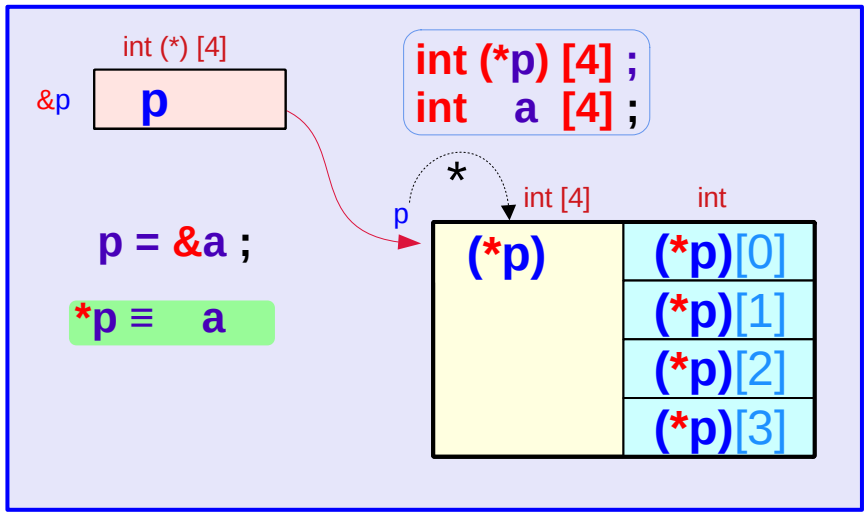
Dereferencing pointer variables **n**, **p**, **fp**

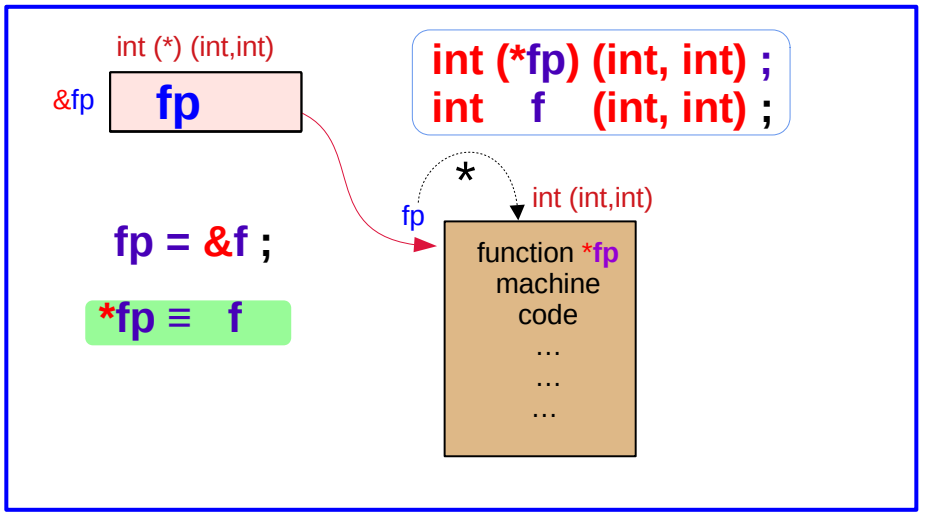


***n** : an **integer** **int**

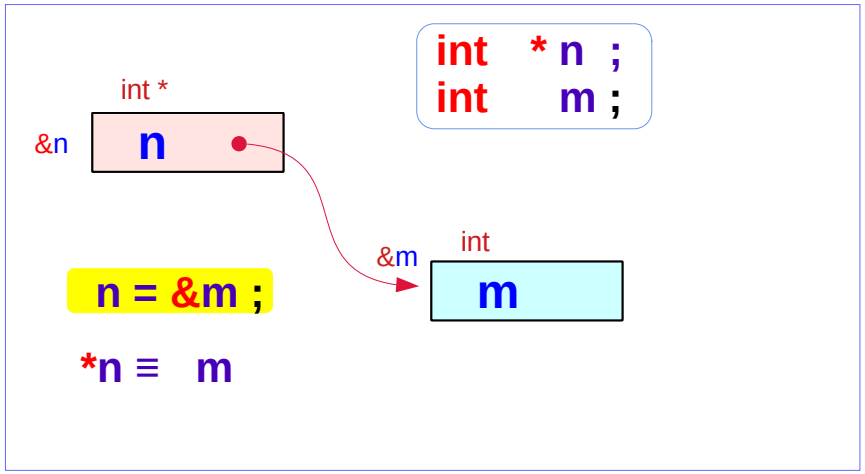
***p** : a **1-d array** **int [4]**
with 4 integer elements

***fp** : a **function** **int (int, int)**
that takes two integers
and returns an integer





Assigning pointer variables **n** and **y[i]**

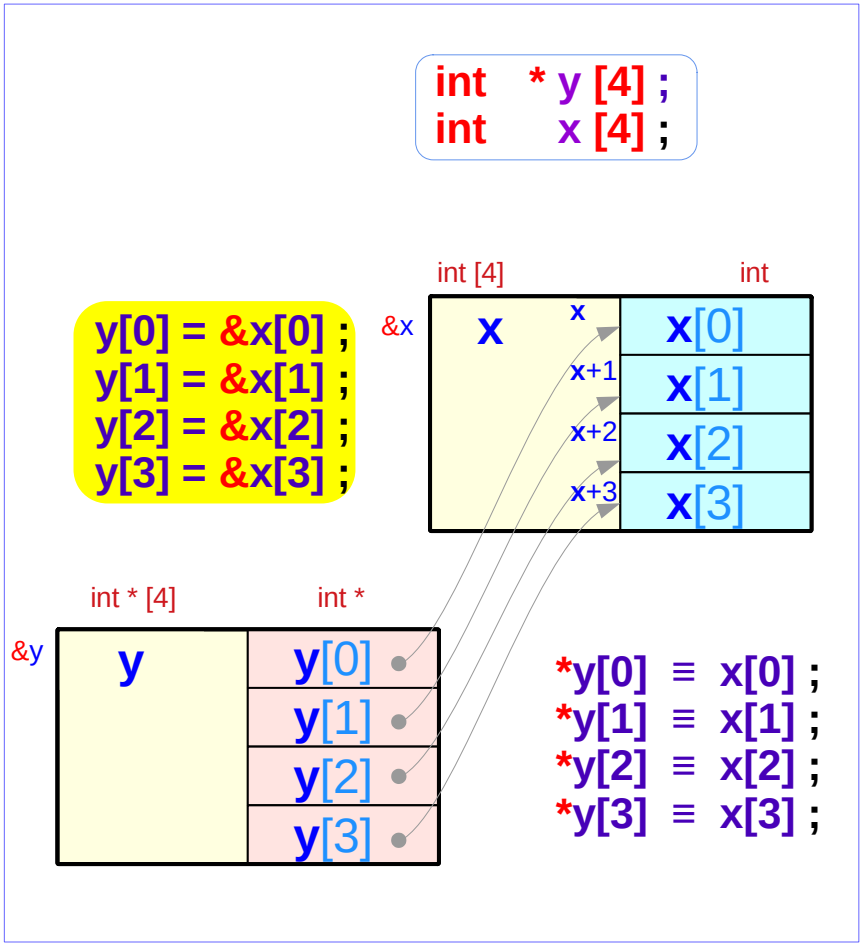


m : an **integer** — **int**

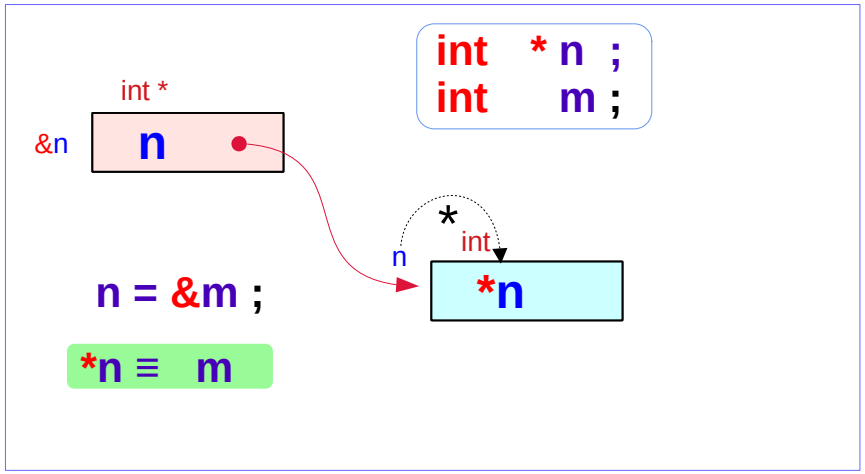
n : an **integer pointer** — **int ***

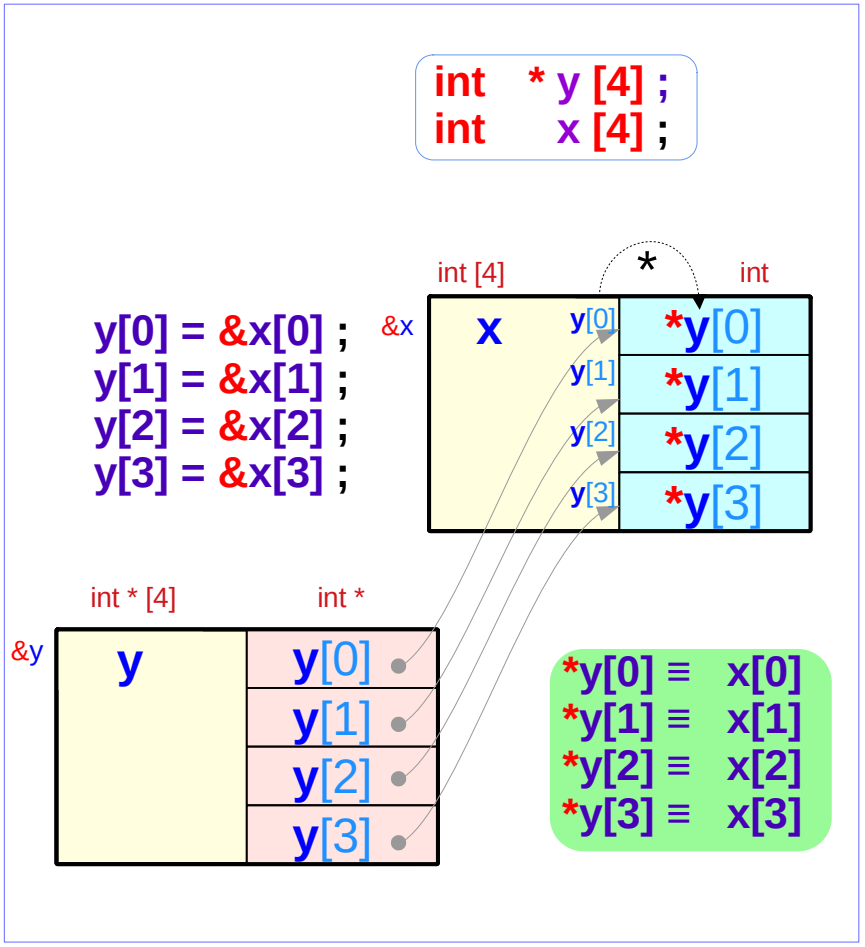
x : a **1-d array** with 4 **integer** elements — **int [4]**

y : a **1-d array** with 4 **integer pointer** elements — **int * [4]**



Dereferencing pointer variables **n** and **y[i]**





n : an **integer pointer** **int ***

***n** : an **integer** **int**

y[i] : an **integer pointer** **int ***

***y[i]** : an integer **int**

Array pointers

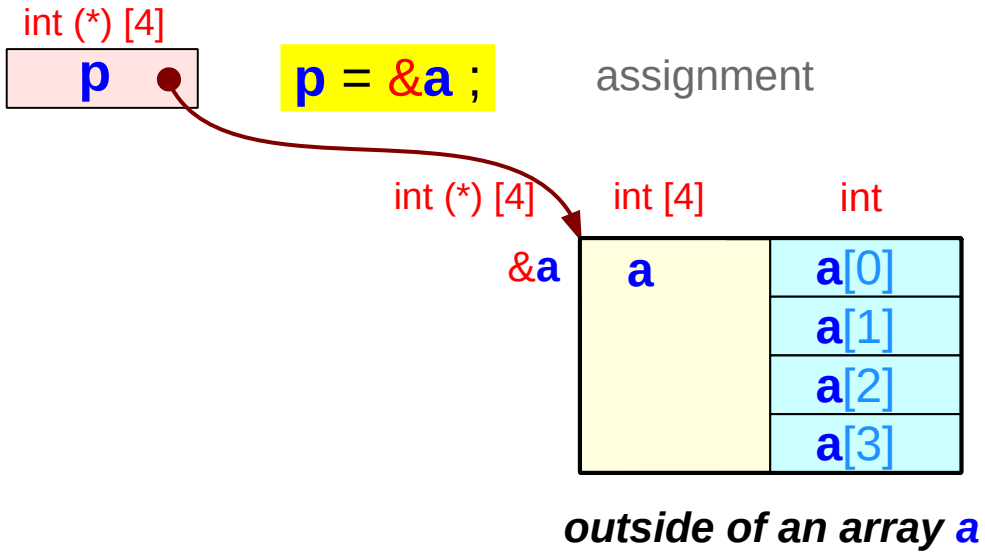
pointer to a 1-d array a and its 0-d sub-array a[0]

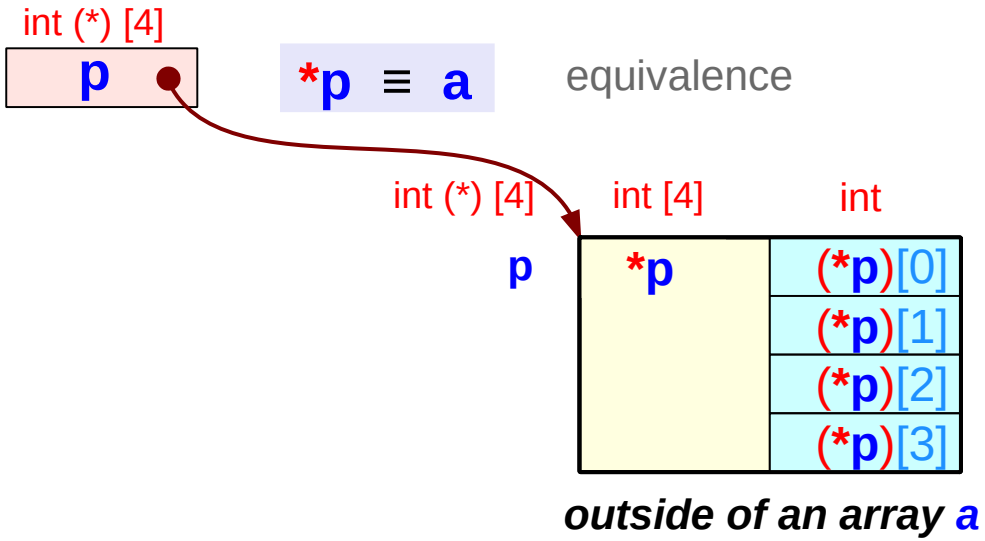
```
int  (*p) [4];     (1-d array pointer)      int a [4];     (1-d array)
int  (*q) ;        (0-d array pointer)
```

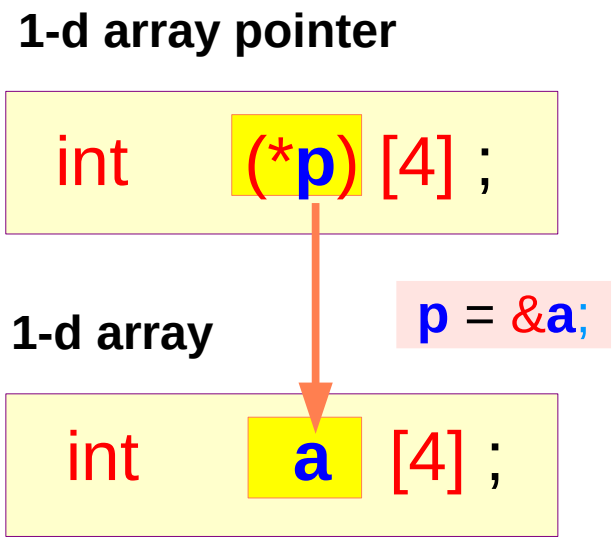
pointer to a 2-d array c and its 1-d sub-array c[0]

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

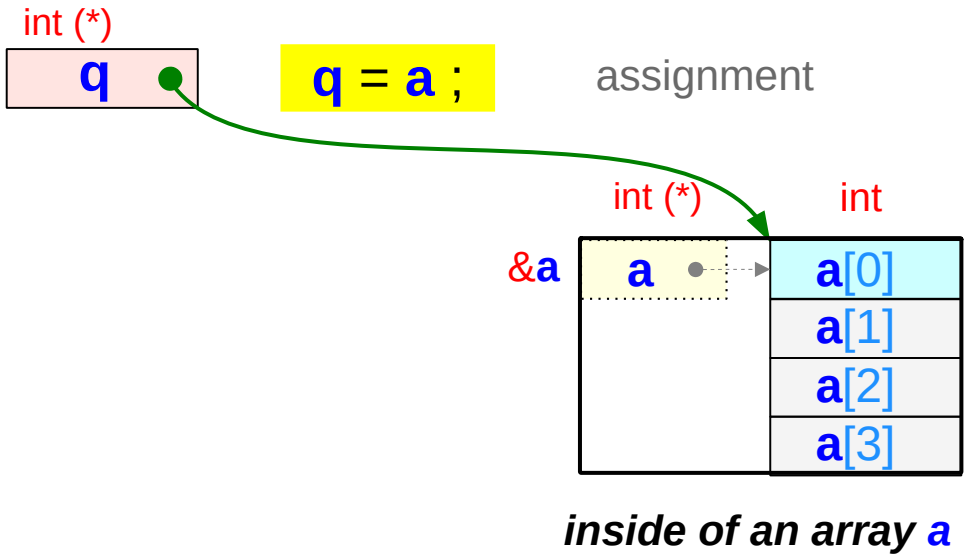
1-d array pointer p to a 1-d array a

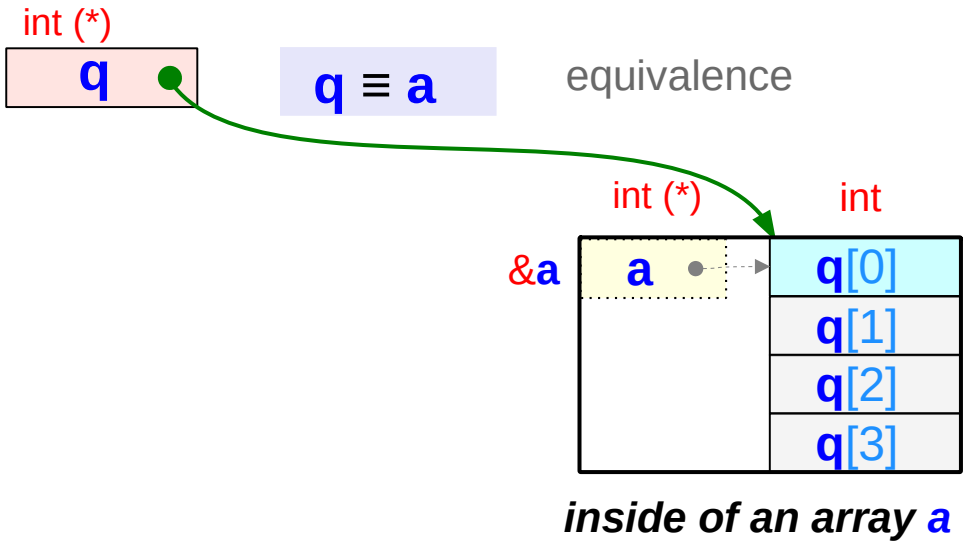


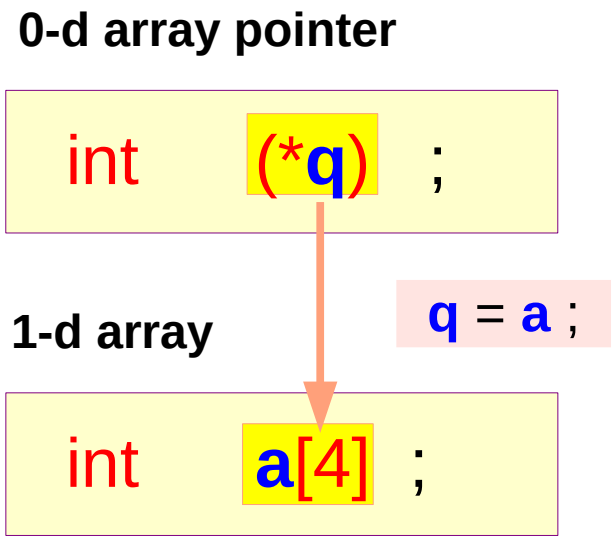




0-d array pointer q to a 0-d sub-array a[0]







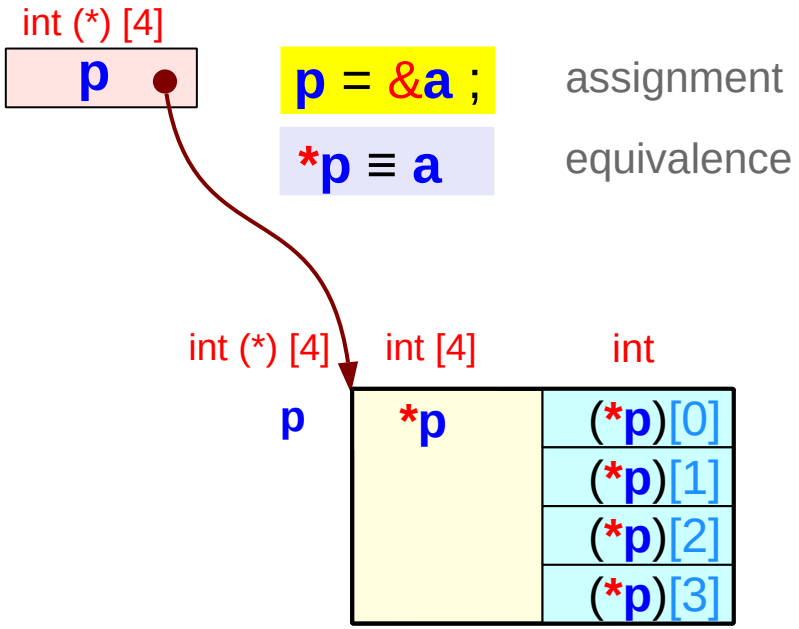
a[i]
a[0]

&a[0] = &*(a+0) = a

1-d array access using p and q

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

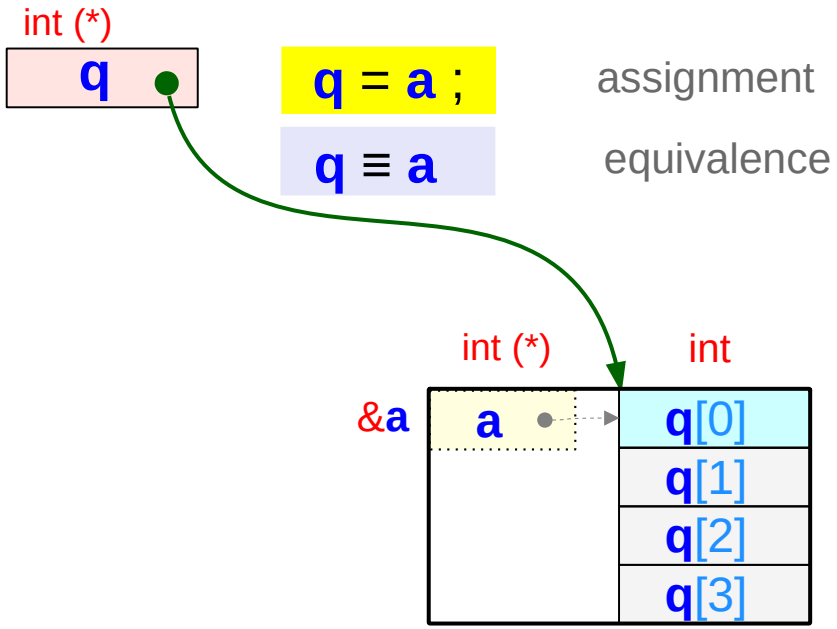
1-d array pointer



outside of an array a

```
int  (*q)   = a;
```

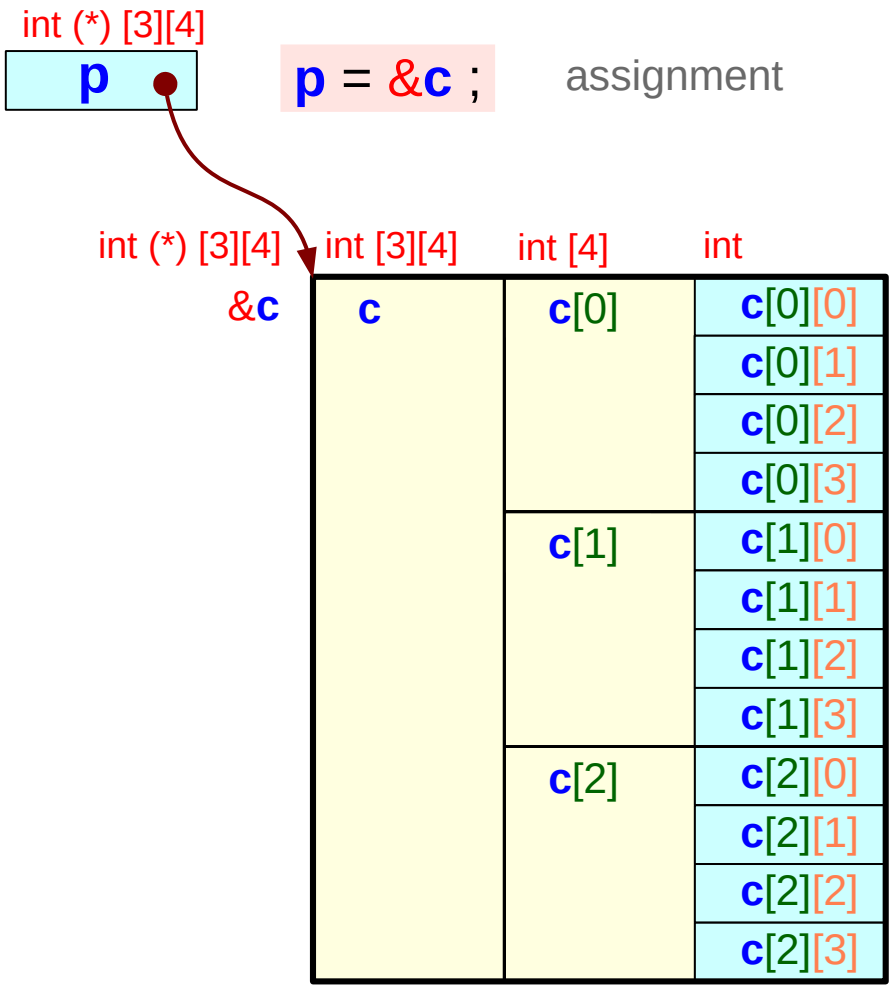
0-d array pointer



inside of an array a

2-d array pointer p to a 2-d array c – reference

2-d array pointer



outside of an array c

2-d array pointer

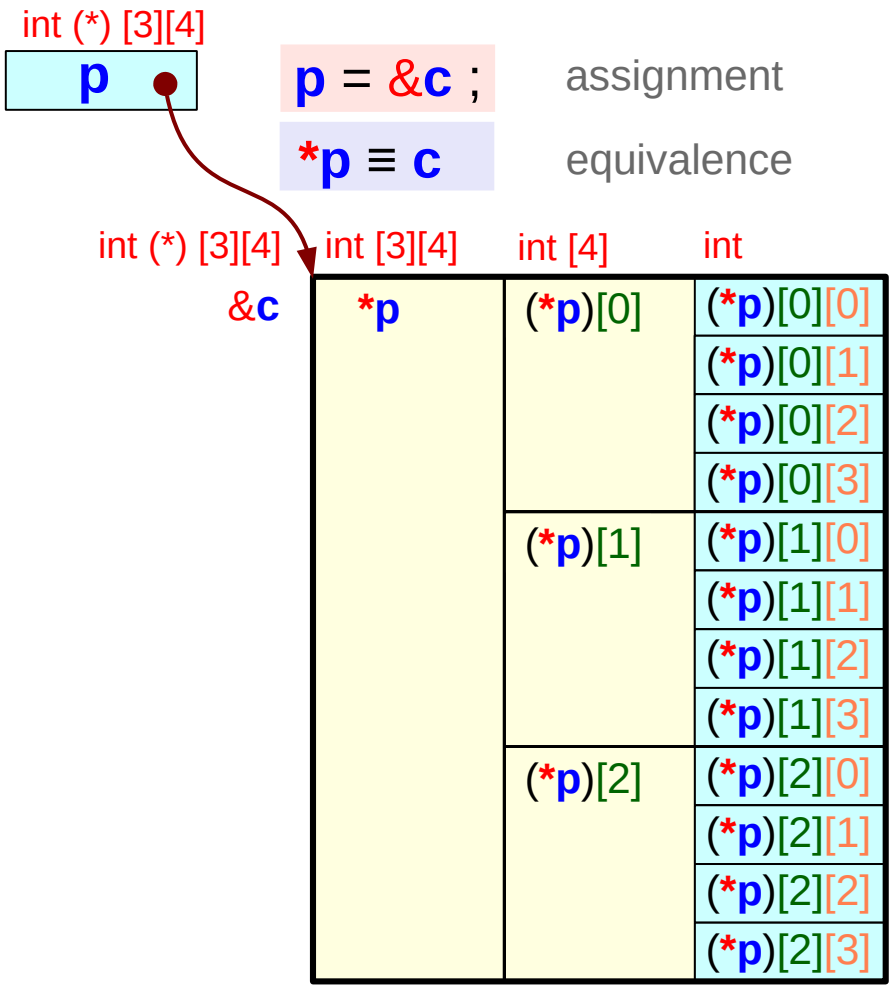
```
int (*p) [3][4] ;
```

2-d array

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

2-d array pointer p to a 2-d array c – dereference

2-d array pointer



outside of an array c

2-d array pointer

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

2-d array

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

1-d array pointer q to a 1-d subarray c[0] – reference

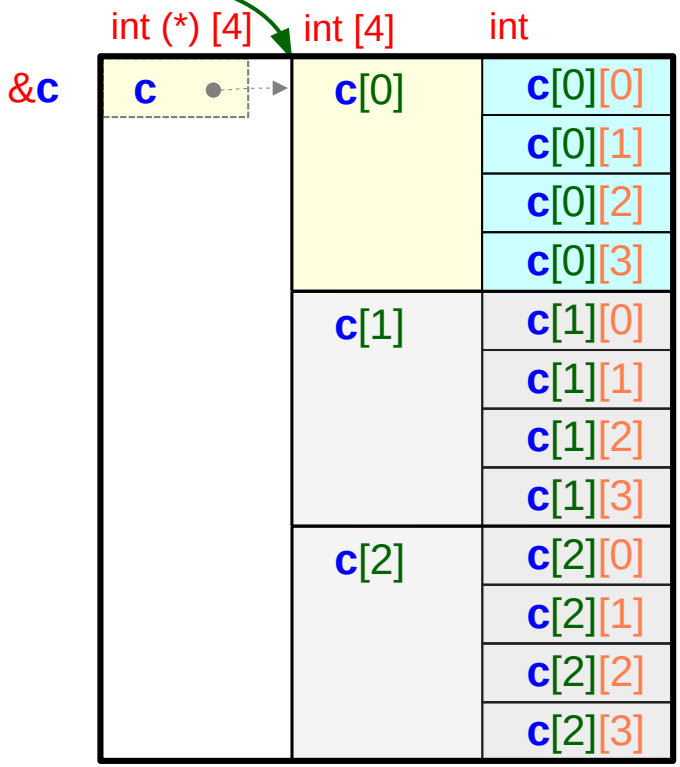
1-d array pointer

int (*) [4]

q

&**c**[0] = &***(c**+0) = **c**

q = **c** ; assignment



inside of an array c ***outside of an array c[0]***

1-d array pointer

int (*q) [4] ;

1-d subarray

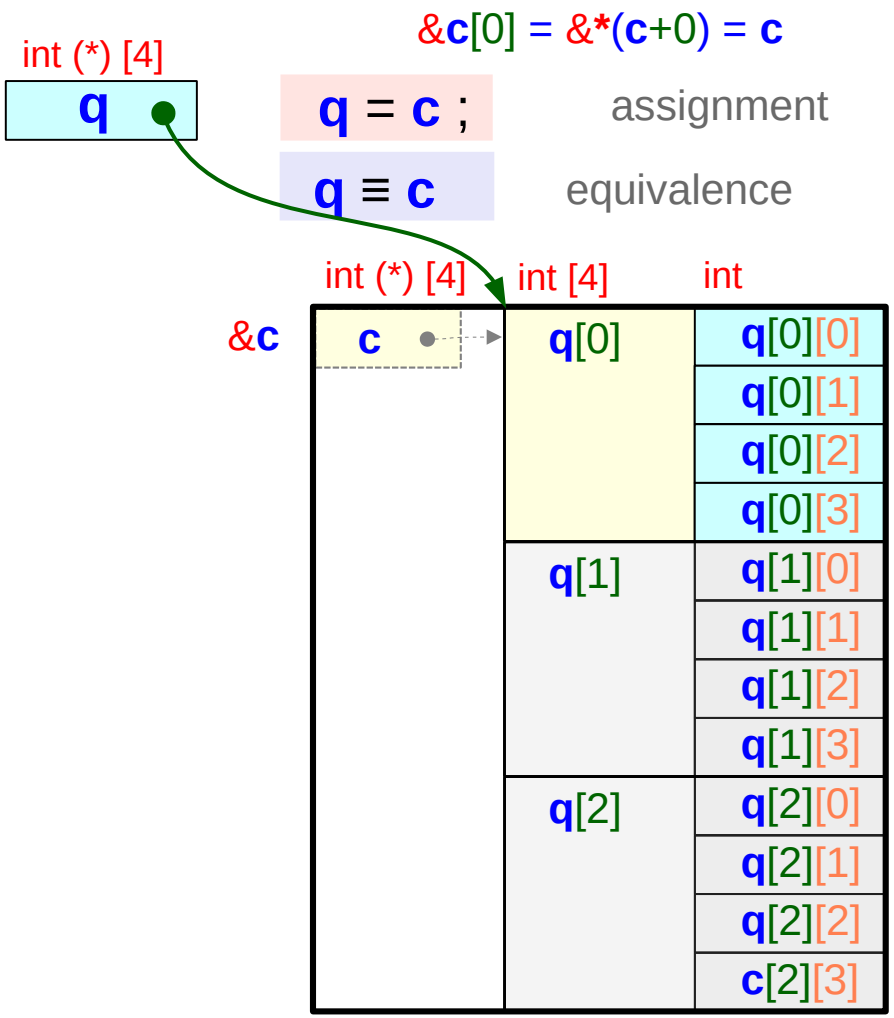
int c[3] [4] ;

Declaration:
[3] means there are 3 elements
Expression:
[0], [1], [2] are used

among 3 elements c[0], c[1], c[2],
consider the first one c[0]

1-d array pointer q to a 1-d subarray c[0] – dereference

1-d array pointer



1-d array pointer

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

1-d subarray

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

Declaration:
[3] means there are 3 elements
Expression:
[0], [1], [2] are used

among 3 elements c[0], c[1], c[2],
consider the first one c[0]

inside of an array c ***outside of an array c[0]***

2-d array access using p and q

2-d array pointer

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

p = &c ;

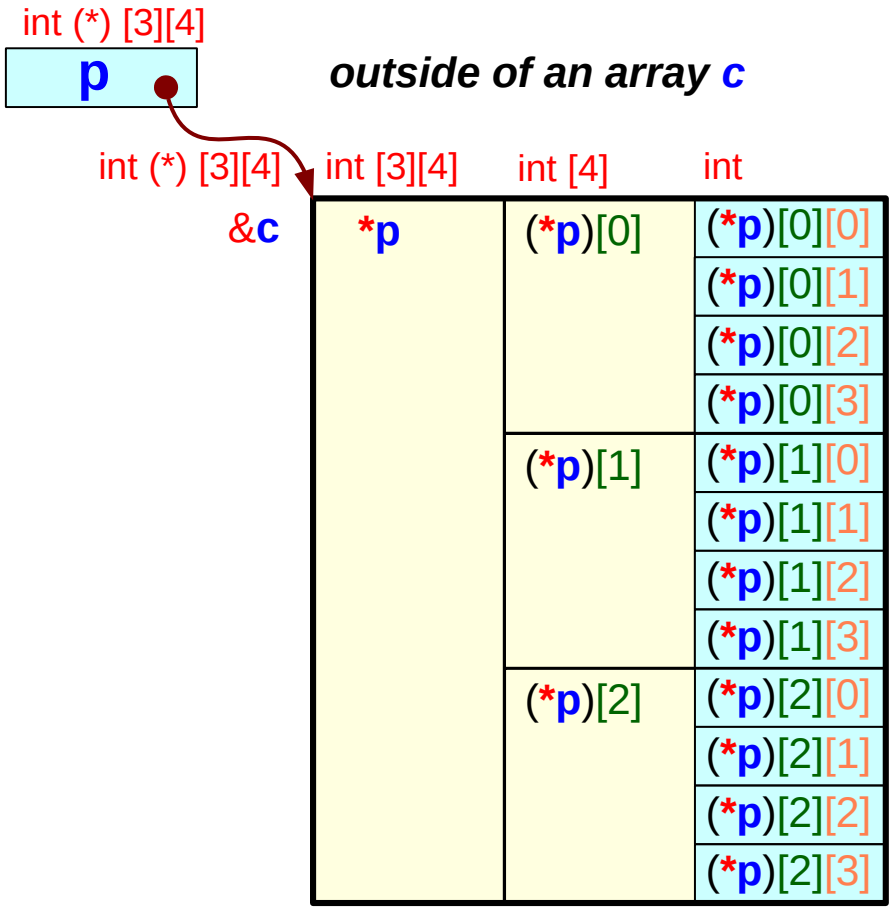
***p ≡ c**

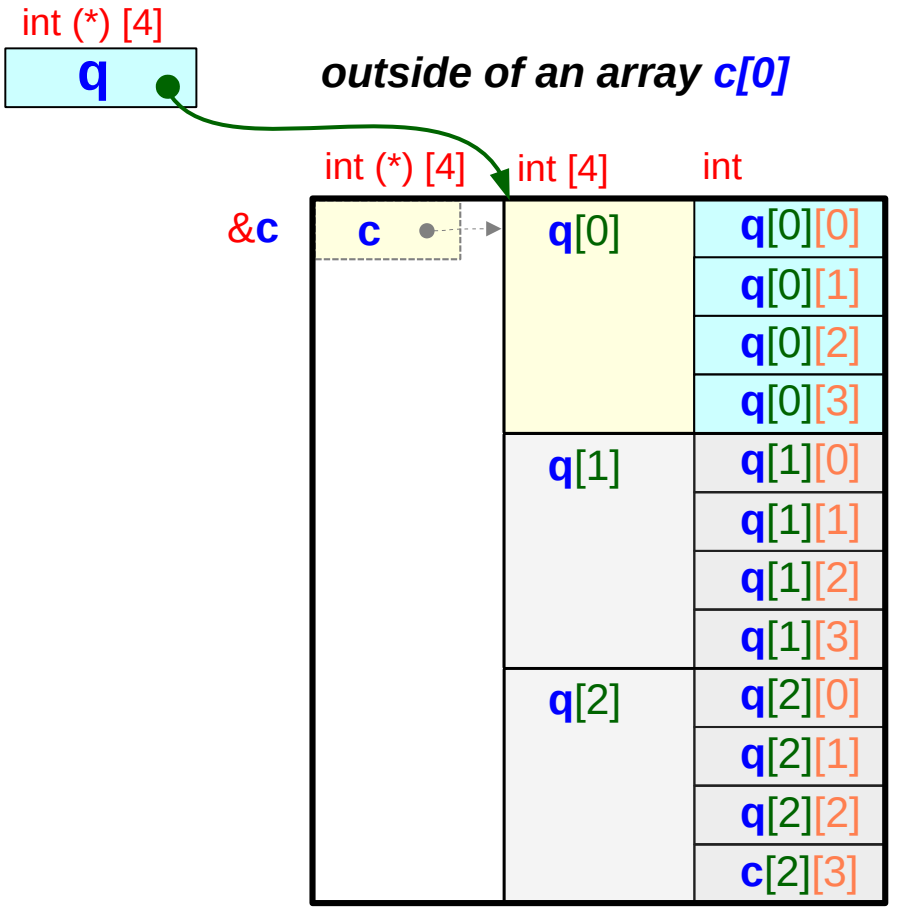
1-d array pointer

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

q = &c[0] ;

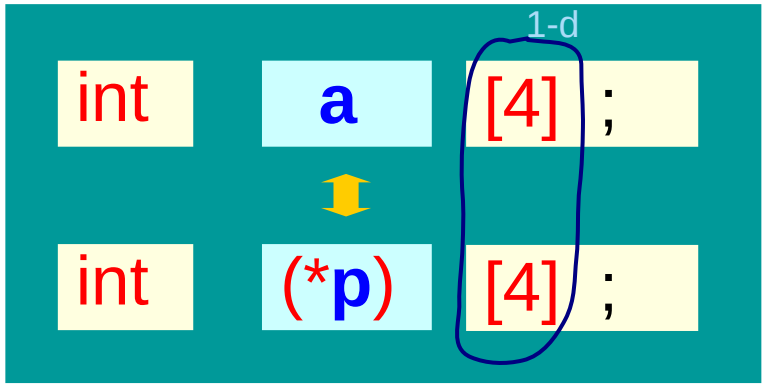
q ≡ c





1-d and 0-d array pointers to an 1-d array

1-d array pointer

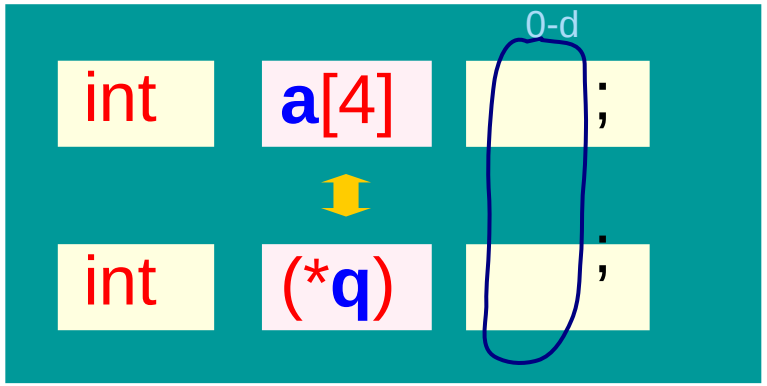


int (*) [4]

***p** ≡ **a**	equivalence
p = &**a**;	assignment

(***p**)[**i**] ≡ **a**[**i**] ≡ **p**[0][**i**]

0-d array pointer : int pointer



int (*)

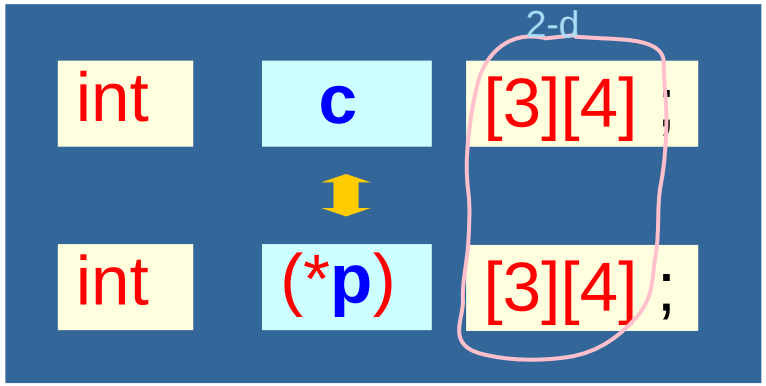
***q** ≡ **a**[0]	equivalence
q = **a**;	assignment

q[**i**] ≡ **a**[**i**] ≡ *(**q**+**i**)

among 4 elements a[0], a[1], a[2], a[3],
consider the first one a[0] = *a

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

2-d array pointer

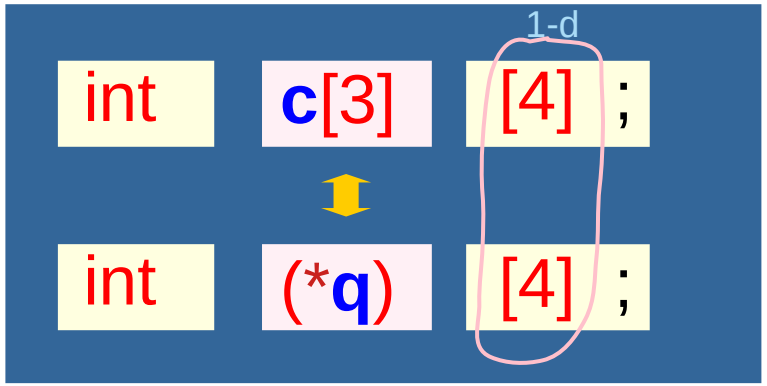


int (*) [3][4]

*p ≡ c	equivalence
p = &c;	assignment

(*p)[i][j] ≡ c[i][j] ≡ p[0][i][j]

1-d array pointer



int (*) [4]

*q ≡ c[0]	equivalence
q = c;	assignment

q[i][j] ≡ c[i][j] ≡ (*(q+i))[j]

among 3 elements c[0], c[1], c[2],
consider the first one c[0] = *c

Extended references of **1-d** & **2-d** array pointers

1-d array pointer

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

(*p)[j] ≡ c[0][j]

(*p)

minimal reference

p[i][j] ≡ c[i][j]

(*p+i)

extended reference

2-d array pointer

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

(*q)[i][j] ≡ c[i][j]

(*q)

minimal reference

q[m][i][j]

(*q+m)

extended reference

Accessing a **2-d** array using a **2-d** & **1-d** array pointers

1-d array pointer

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

2-d array pointer

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

```
(*p)[j]
```

1-d array access
minimal

```
p [i][j]
```

2-d array access
extended

```
(*q) [i][j]
```

2-d array access
minimal

```
q[m] [i][j]
```

3-d array access
extended

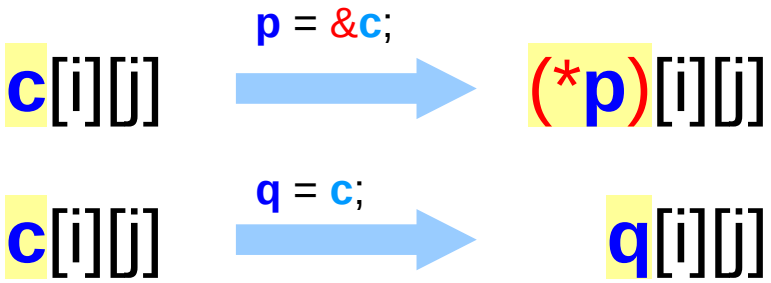
Array pointers to a **2-d** array and its sub-array

```
int   c   [3]  [4] ;
int (*p)  [3]  [4] = &c ;
int (  * q  )  [4] = &c[0] ;   (= c)
```

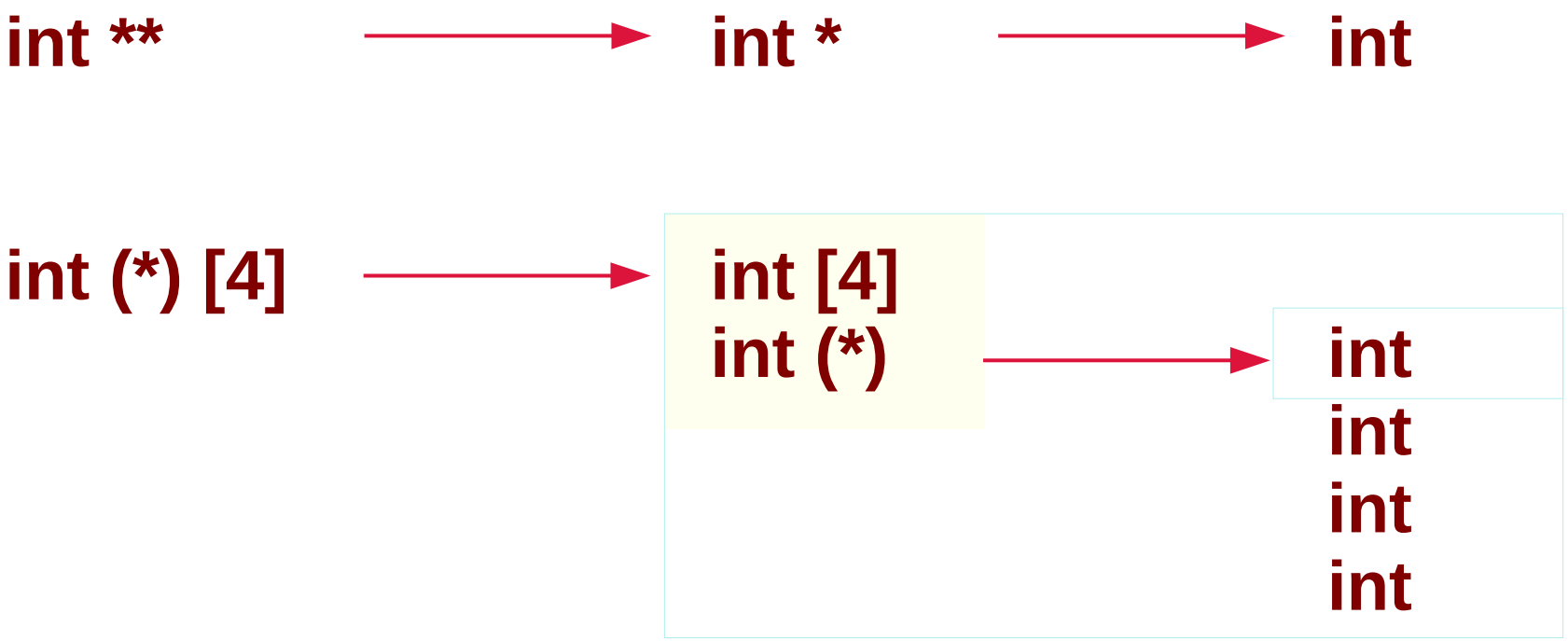
2-d array c

2-d array pointer p

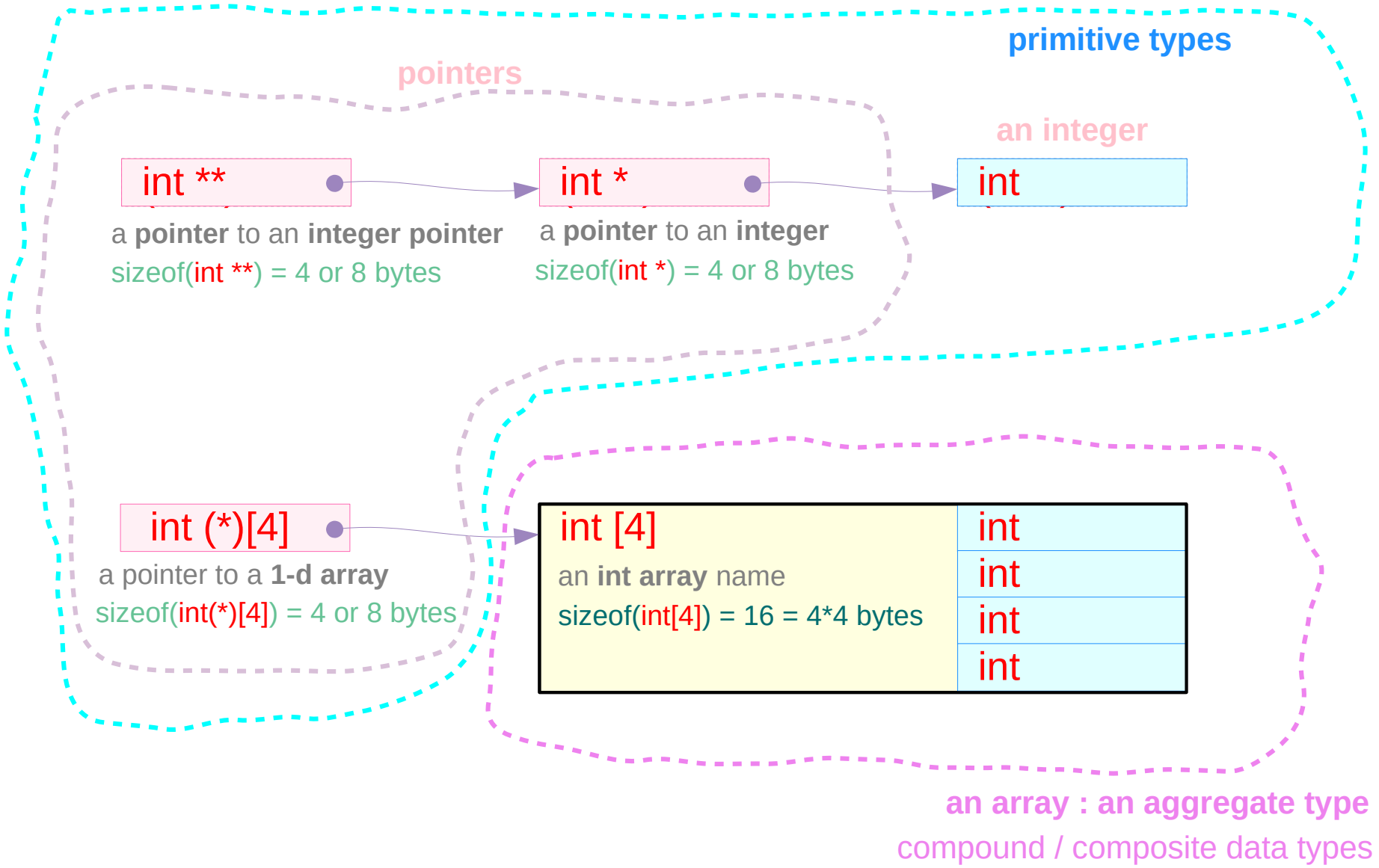
1-d array pointer q



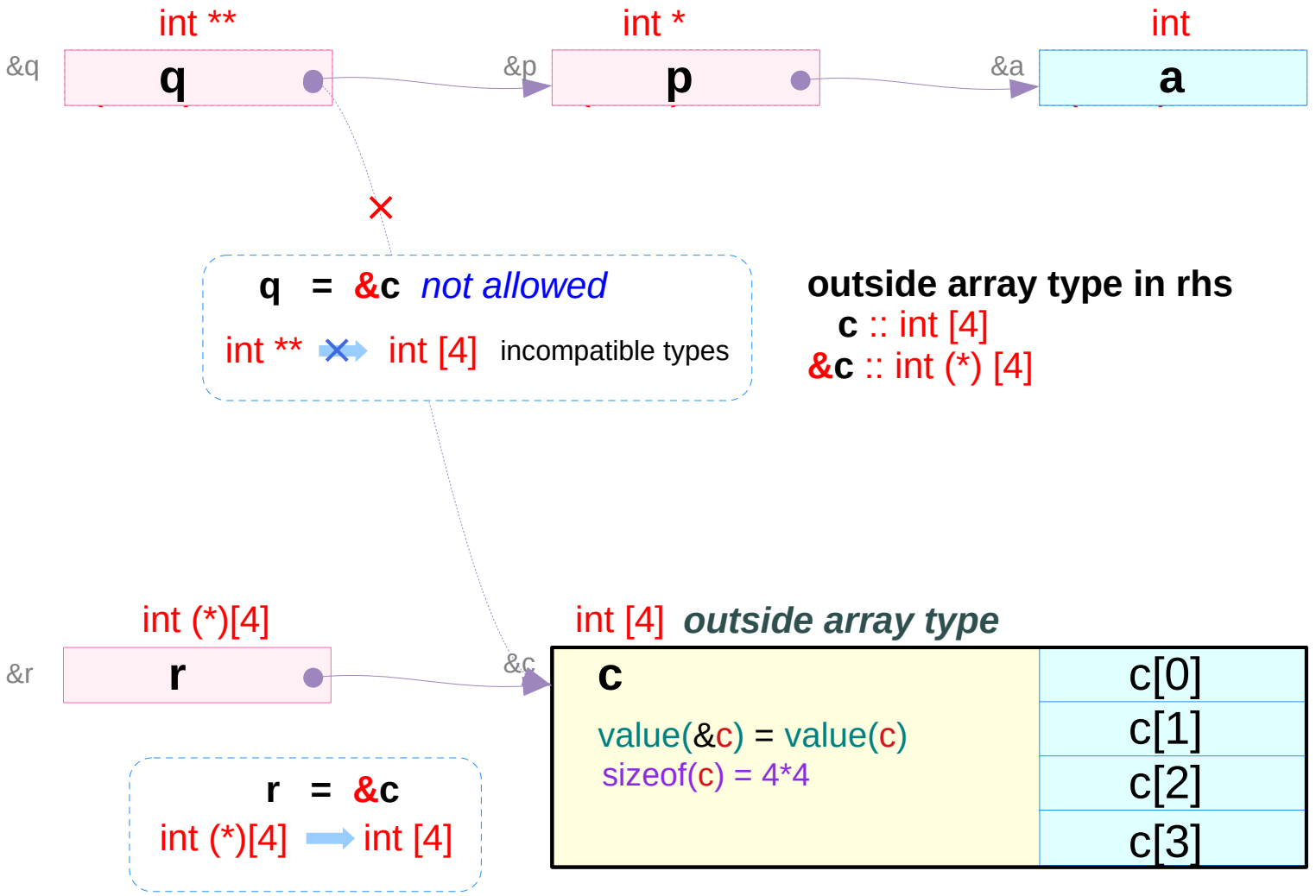
Pointer chains and nested pointers



Types of integer pointers



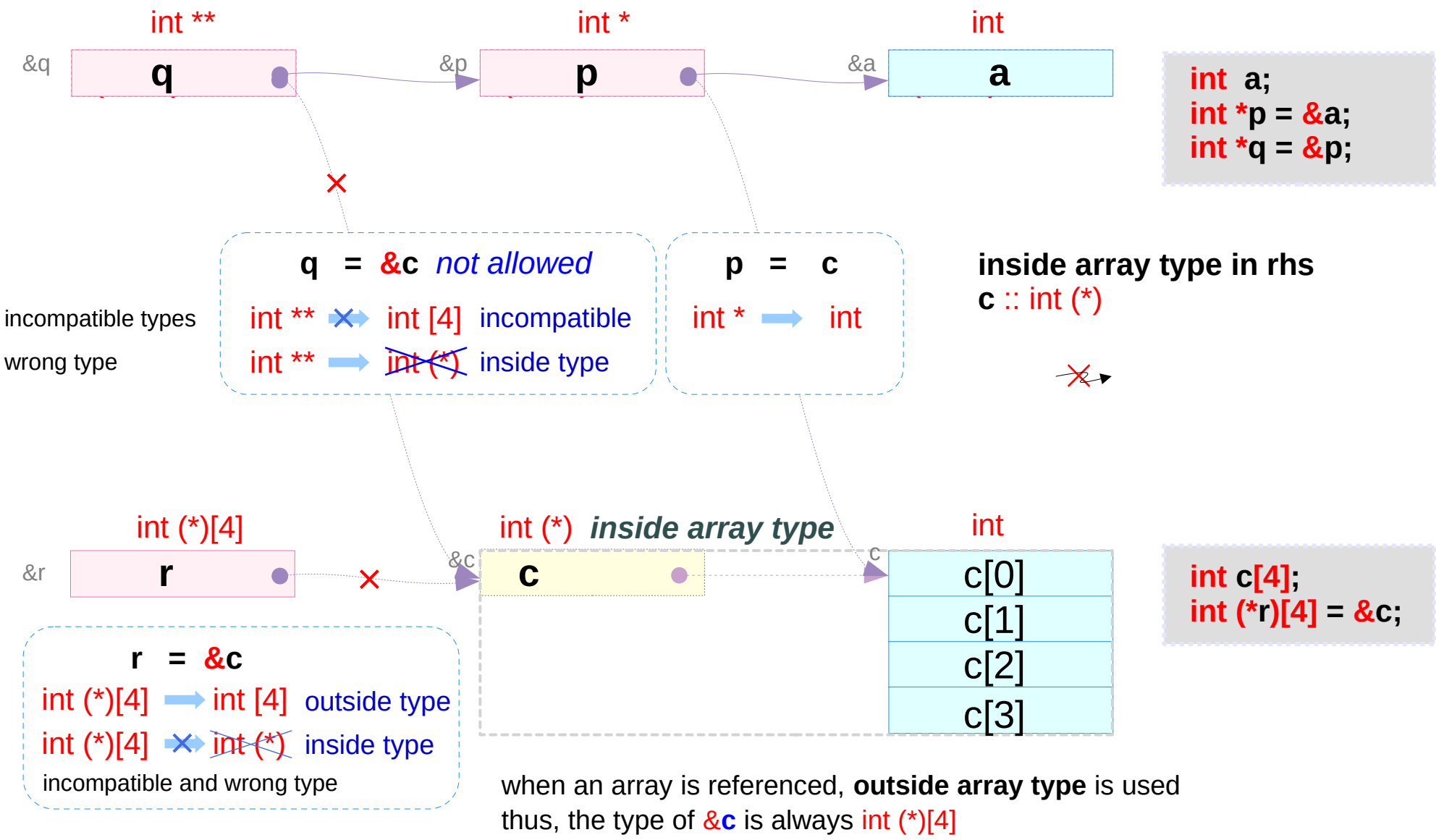
Outside array type **int [4]**



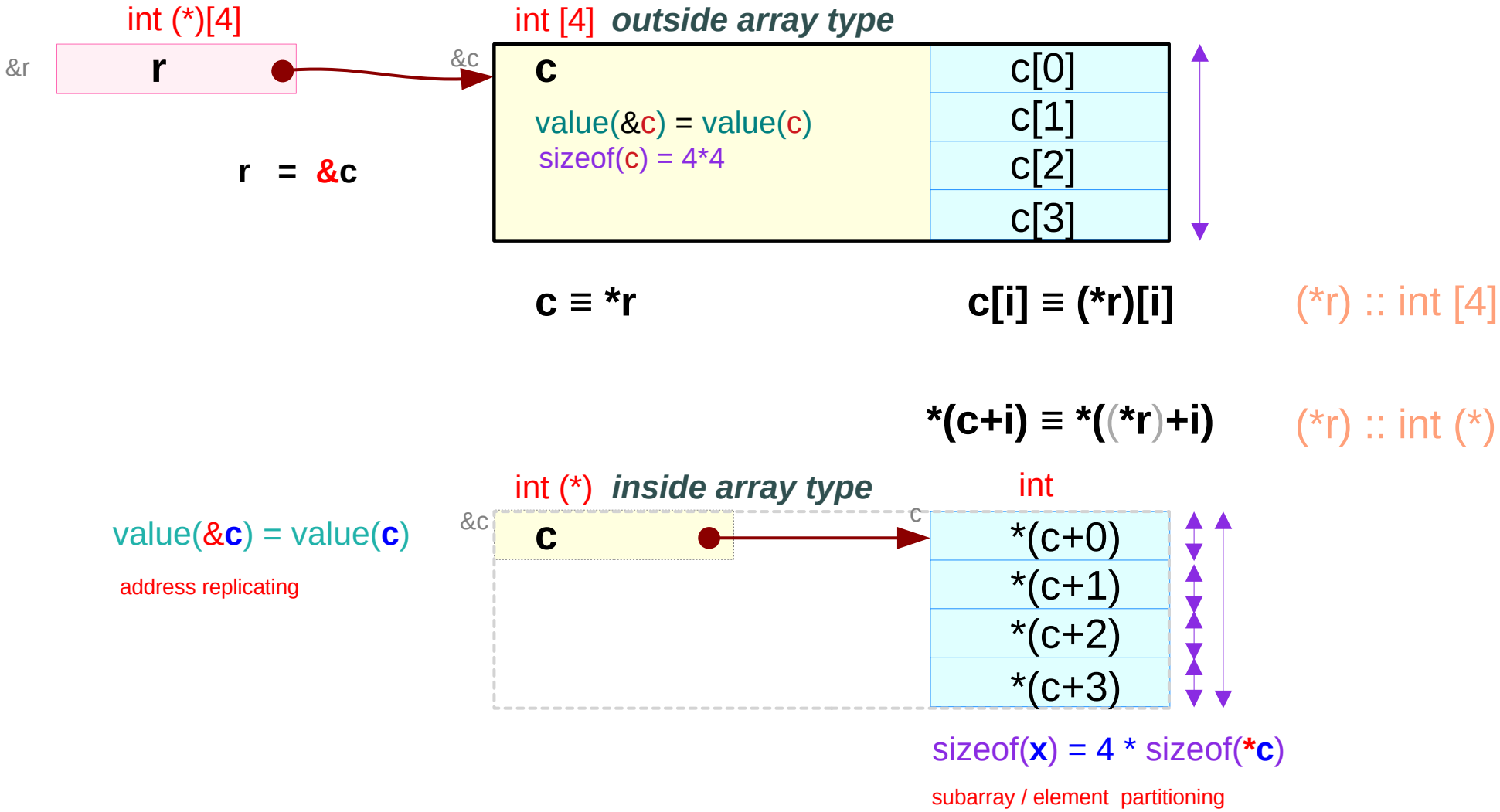

```
int  a;
int *p = &a;
int *q = &p;
```

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

Inside array type **int (*)**



Outside array type **int [4]**



Explicit and implicit array pointers

- **Explicit array pointers**

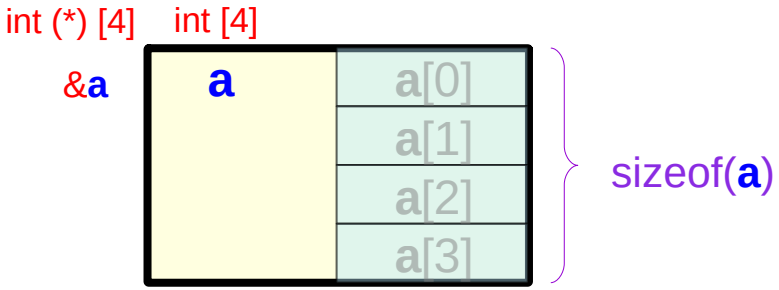
- **Implicit array pointers in an array**

- **Sizes**
- **Types**
- **Values**

Type, size, and value of **a** and &**a** for a **1-d** array **a**

```
int a [4] ;
```

outside of an array a –
a** as an **abstract data



abstract data a

value(**a**) = value(&**a**[0])
sizeof(**a**) = 4 * sizeof(int)
type(**a**) = int [4] (outside)
type(**a**) = int (*) (inside)

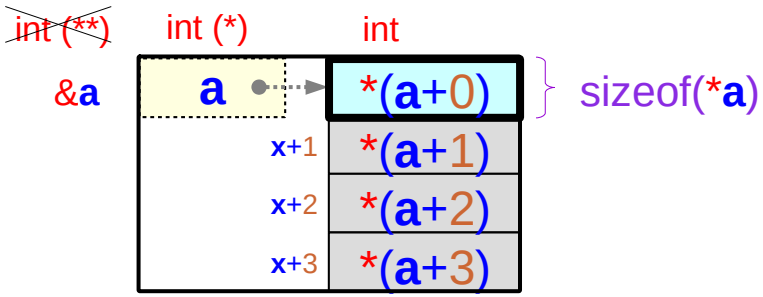
&a : the address of a

value(&**a**) = value(**a**)
sizeof(&**a**) = 4 or 8 bytes
type(&**a**) = int (*) [4]

Type, size, and value of *a and a for a 1-d array a

```
int a [4] ;
```

inside of an array a –
a as a virtual pointer



primitive data *a

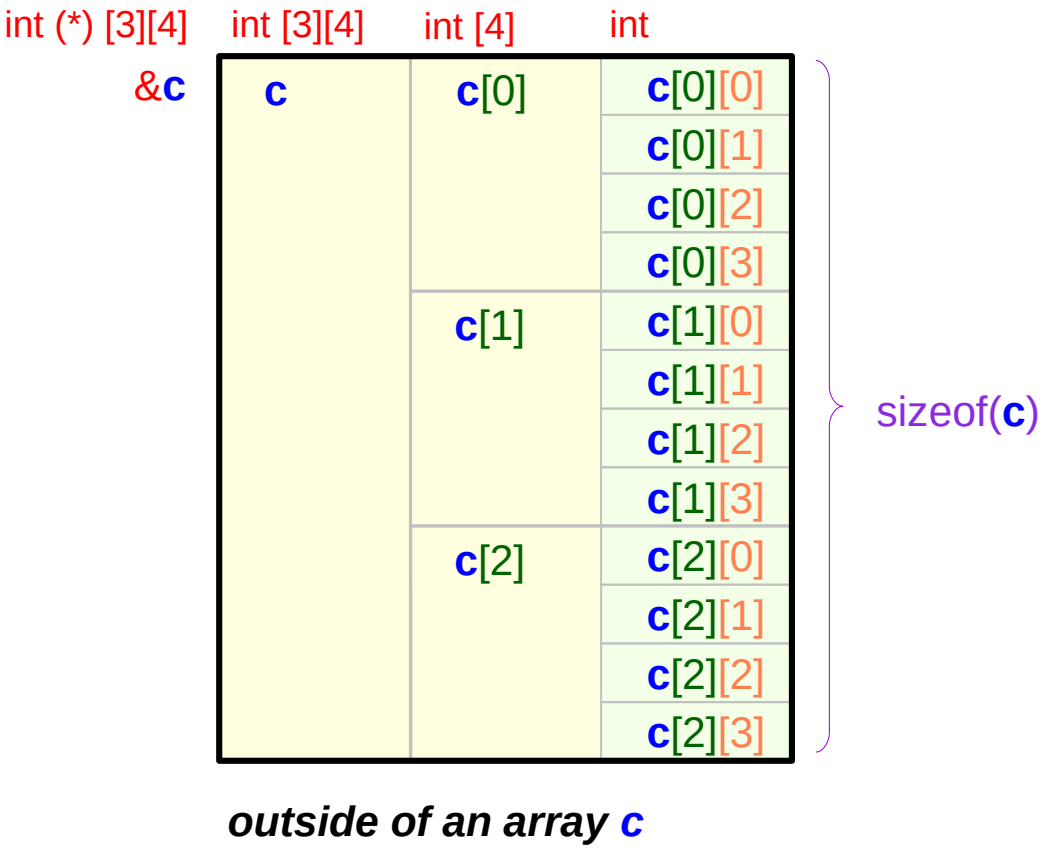
value(*a) = value(a[0])
sizeof(*a) = sizeof(int)
type(*a) = int

a : the address of *a

value(a) = value(&a[0])
sizeof(a) = 4 * sizeof(int)
type(a) = int [4] (outside)
type(a) = int (*) (inside)

Type, size, and value of **c** and &**c** for a **2-d** array **c**

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



outside of an array c

outside of an array c –
c** as an **abstract data

c : abstract data

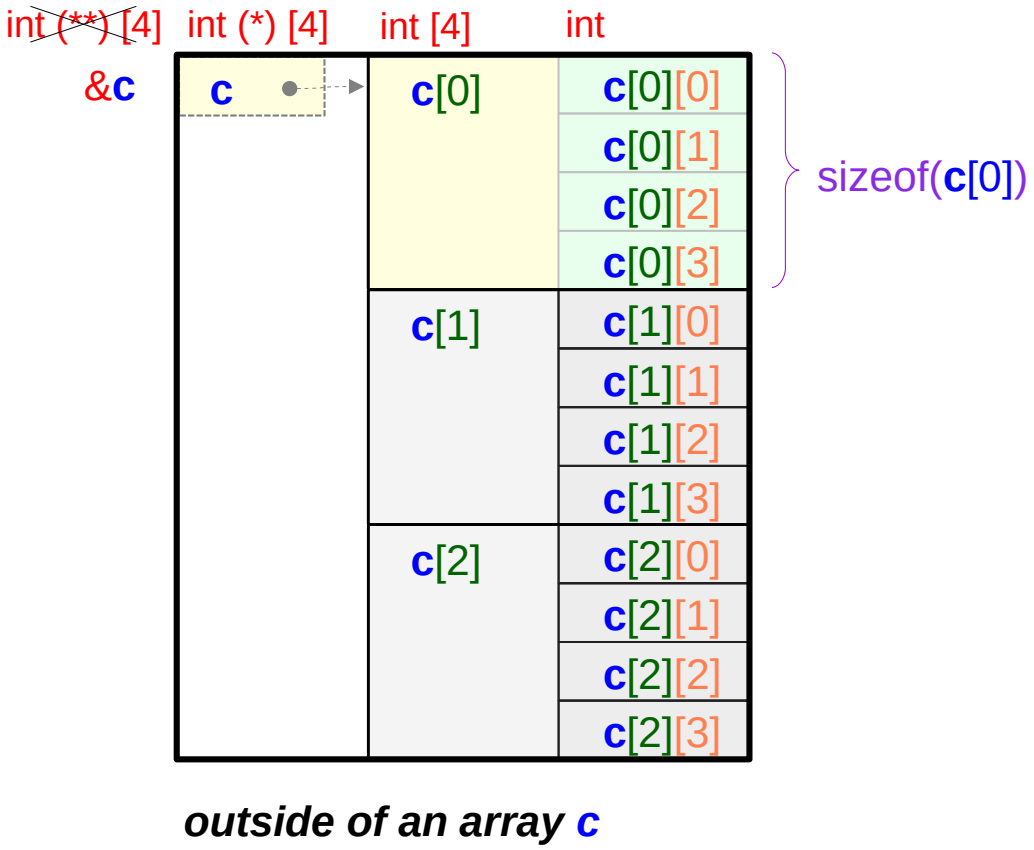
value(**c**) = value(&**c**[0])
sizeof(**c**) = 3 * 4 * sizeof(int)
type(**c**) = int [3][4] (outside)
type(**c**) = int (*)[4] (inside)

&c : the address of c

value(&**c**) = value(**c**)
sizeof(&**c**) = 4 or 8 bytes
type(&**c**) = int (*) [3][4]

Type, size, and value of **c[0]** and **c** for a **2-d** array **c**

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



outside of an array c

inside of an array c –
c as a virtual pointer

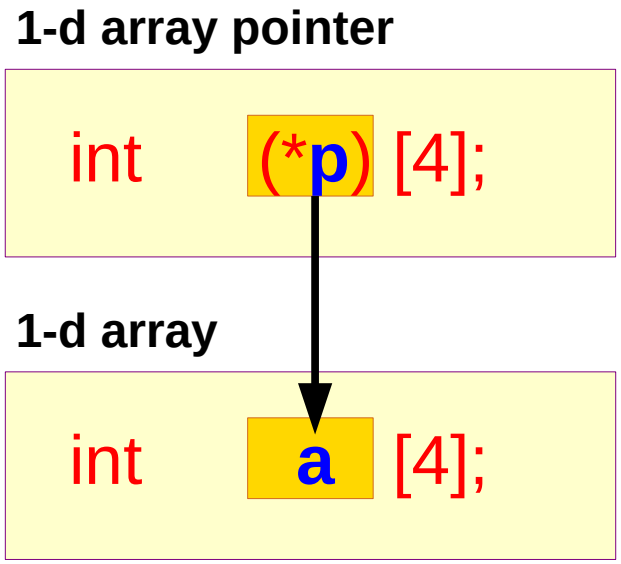
c[0] : abstract data *c

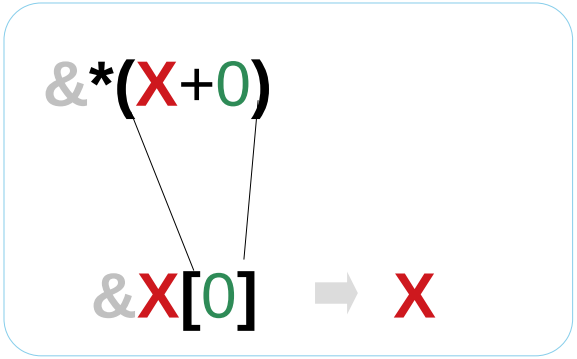
value(**c**[0]) = value(&**c**[0][0])
sizeof(**c**[0]) = 4 * sizeof(int)
type(**c**[0]) = int [4] (outside)
type(**c**[0]) = int (*) (inside)

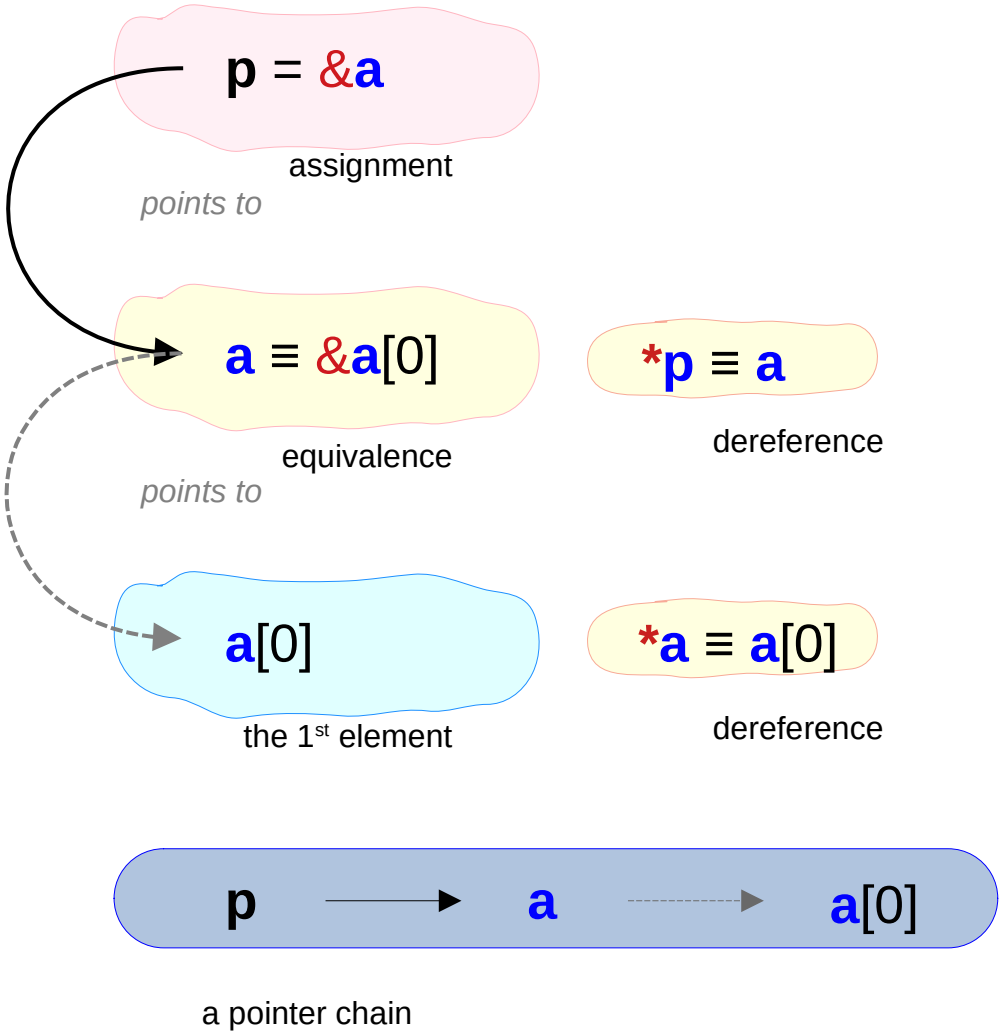
c : the address of c[0]

value(**c**) = value(&**c**[0])
sizeof(**c**) = 3 * 4 * sizeof(int)
type(**c**) = int [3][4] (outside)
type(**c**) = int (*)[4] (inside)

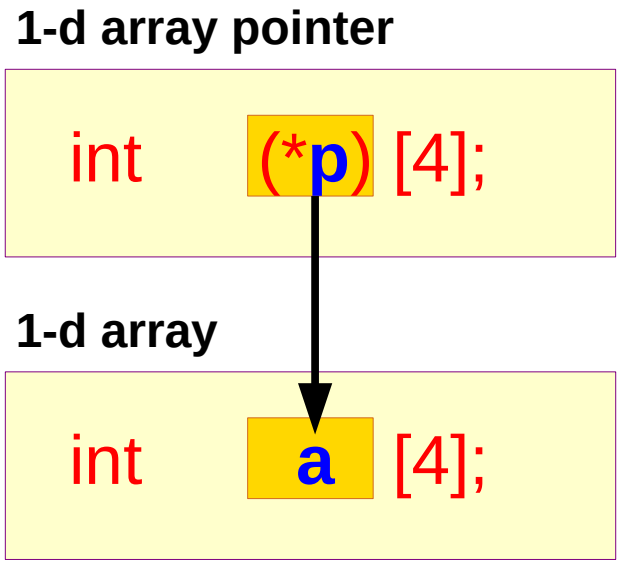
1-d array pointer p – (1) pointer chain

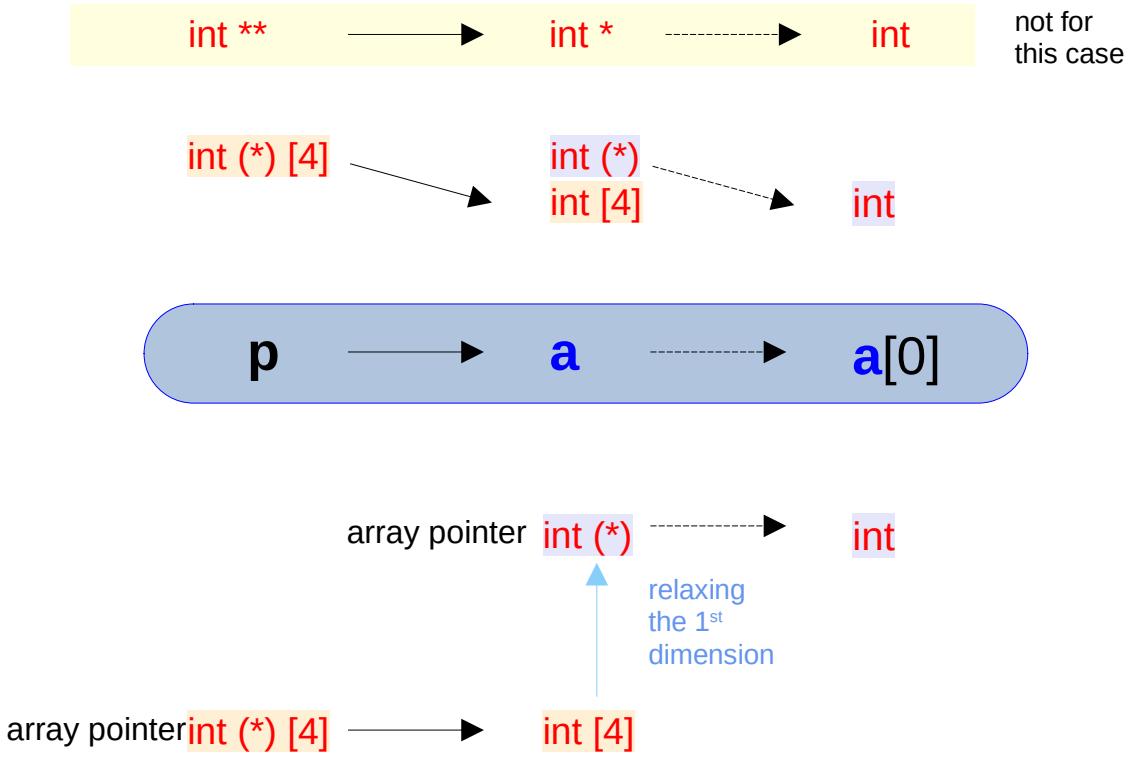




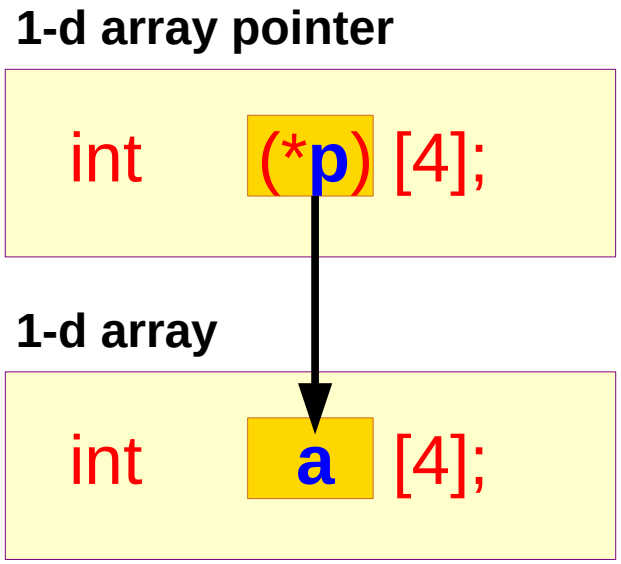


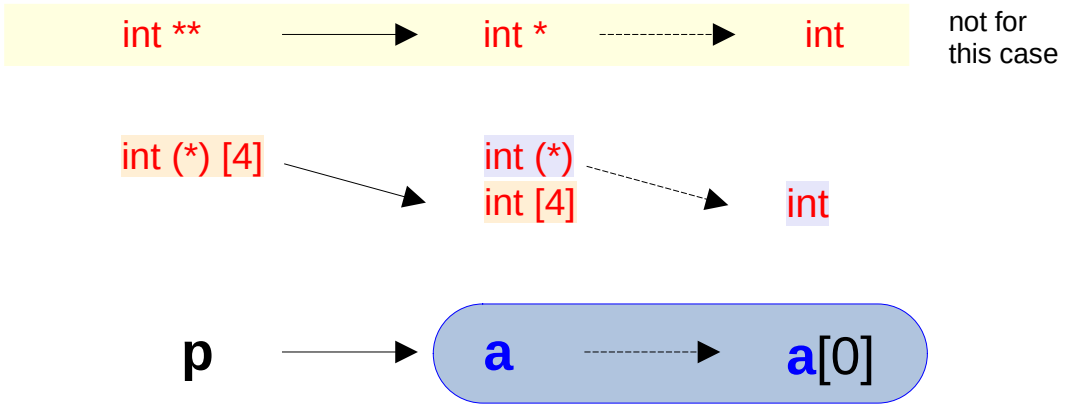
1-d array pointer p – (2) types in a pointer chain

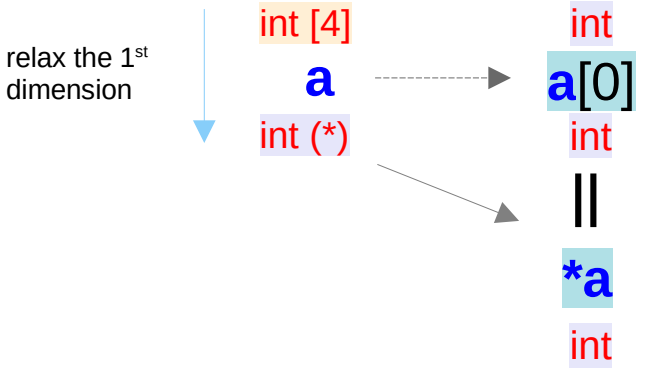




1-d array pointer p – (3) relaxing the 1st dimension

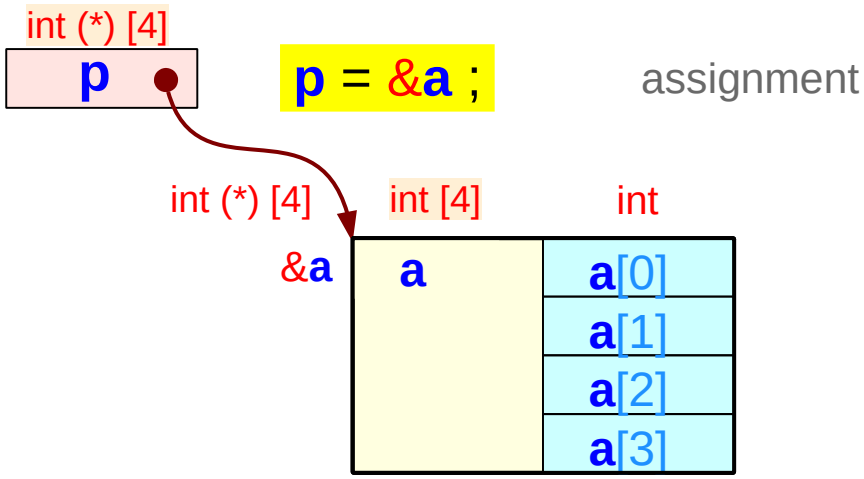


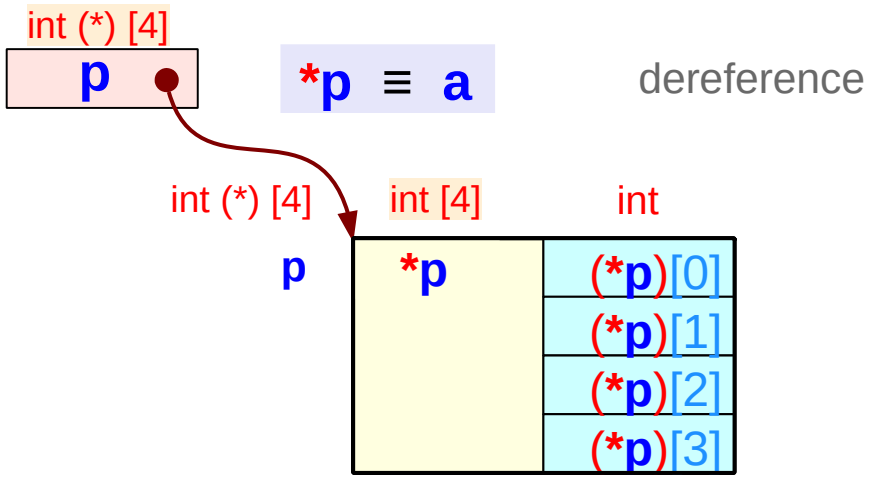




1-d array pointer p – (4) assignment and dereference

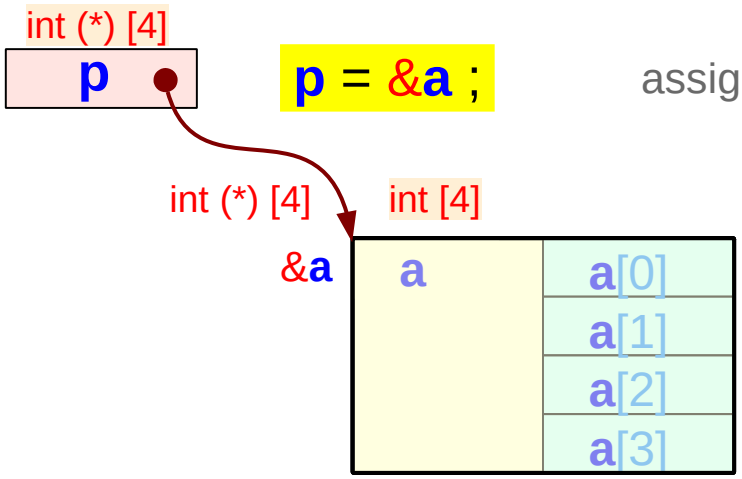
	assignment	dereference	equivalence	dereference
int a [4];				
int (*p) [4];	p = &a	*p ≡ a	a ≡ &a[0]	*a ≡ a[0]

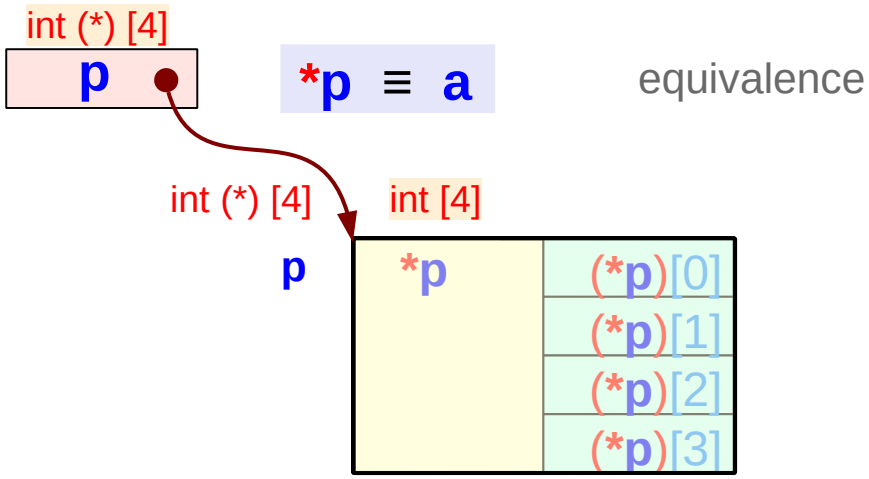




1-d array pointer p – 1-d array a

outside of an array a (***a*** *as an* ***abstract data***)

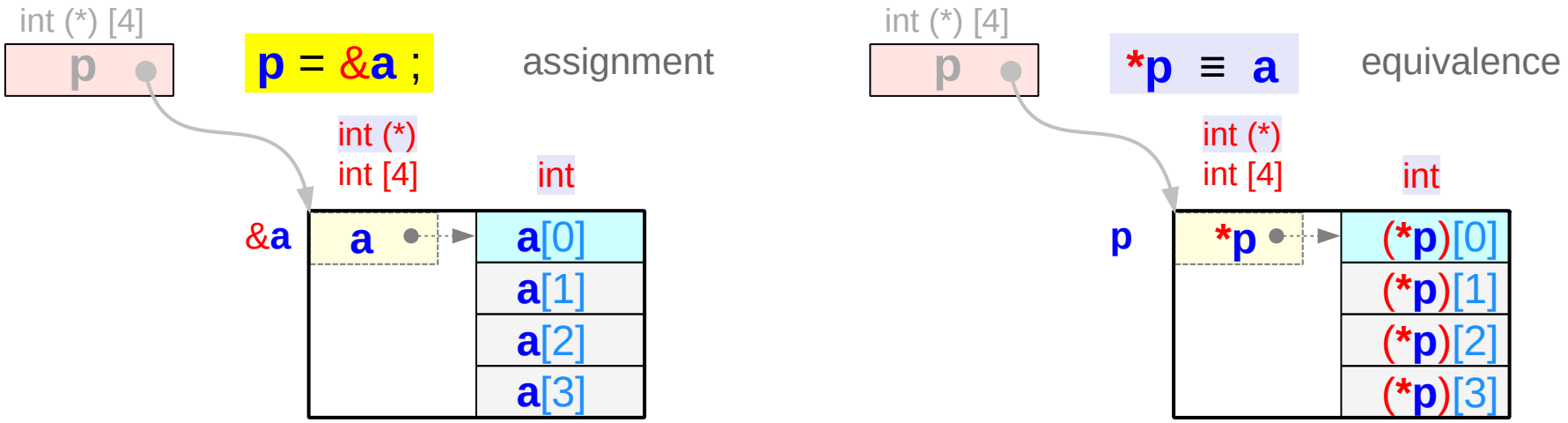




sizeof(&a) = 4 or 8 bytes	size of a pointer	value(&a)	address value of a 1-d array a
sizeof(a) = 4 * sizeof(int)	size of a **1-d** array	= value(a)	data value of a 1-d array a

0-d array pointer a – 0-d array a[0]

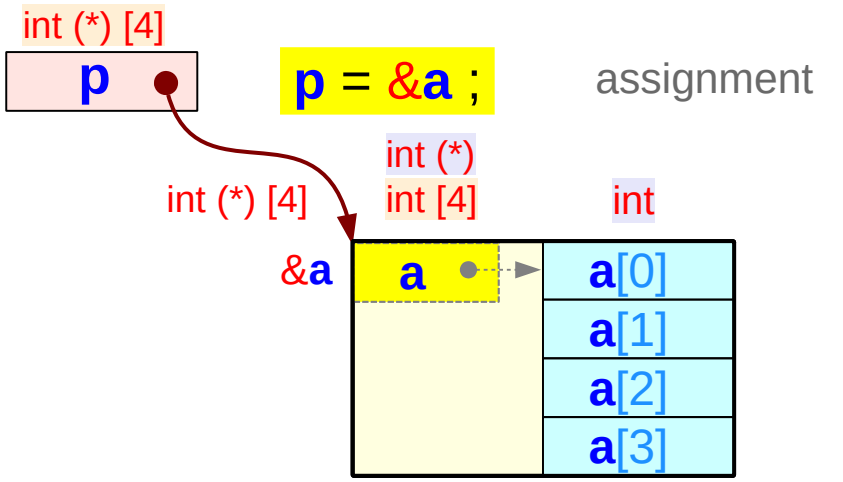
inside of an array a (***a*** *as a* ***virtual pointer***)

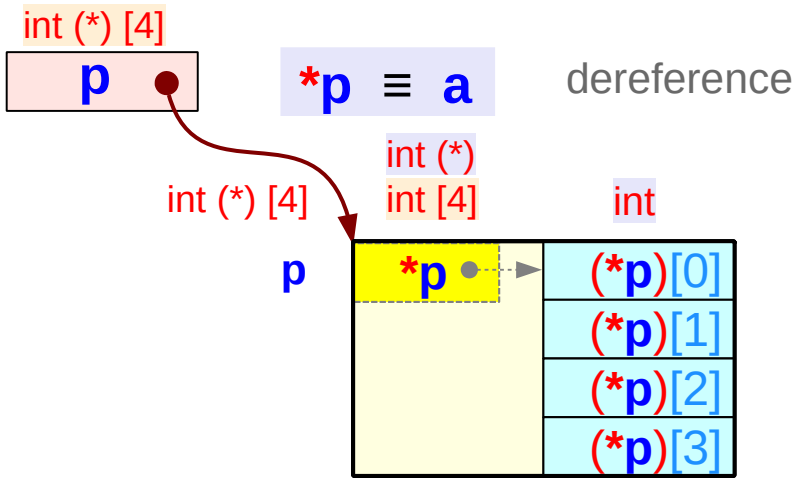


sizeof(**a**) = 4 * sizeof(int)	size of a **1-d** array	value(**a**) = value(&**a**[0])	address value of an integer **a**[0]
sizeof(**a**[0]) = 4 bytes	size of an integer	value(**a**[0])	data value of an integer **a**[0]

Overlayed representation

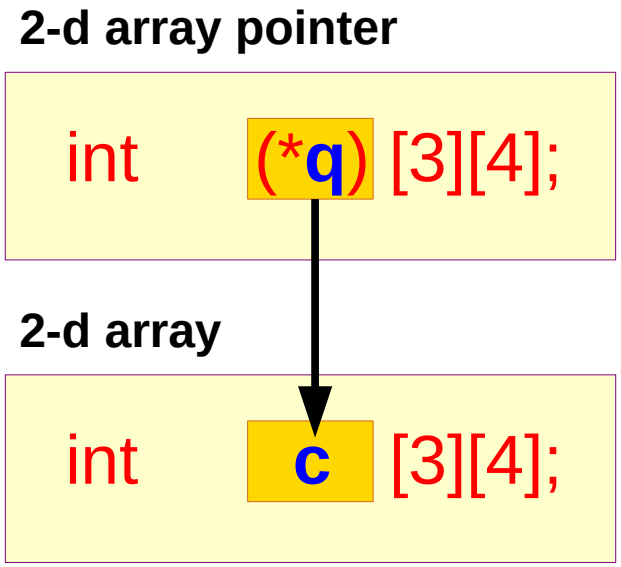
outside of an array a (***a*** *as an* ***abstract data***)
inside of an array a (***a*** *as a* ***virtual pointer***)

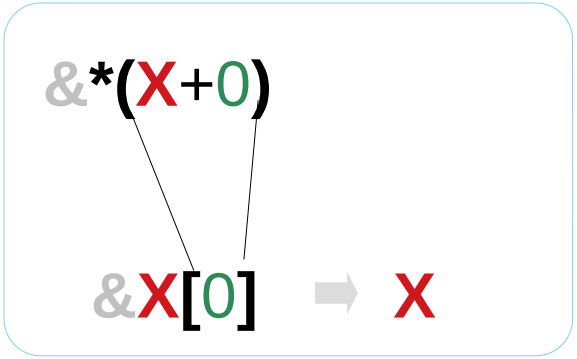


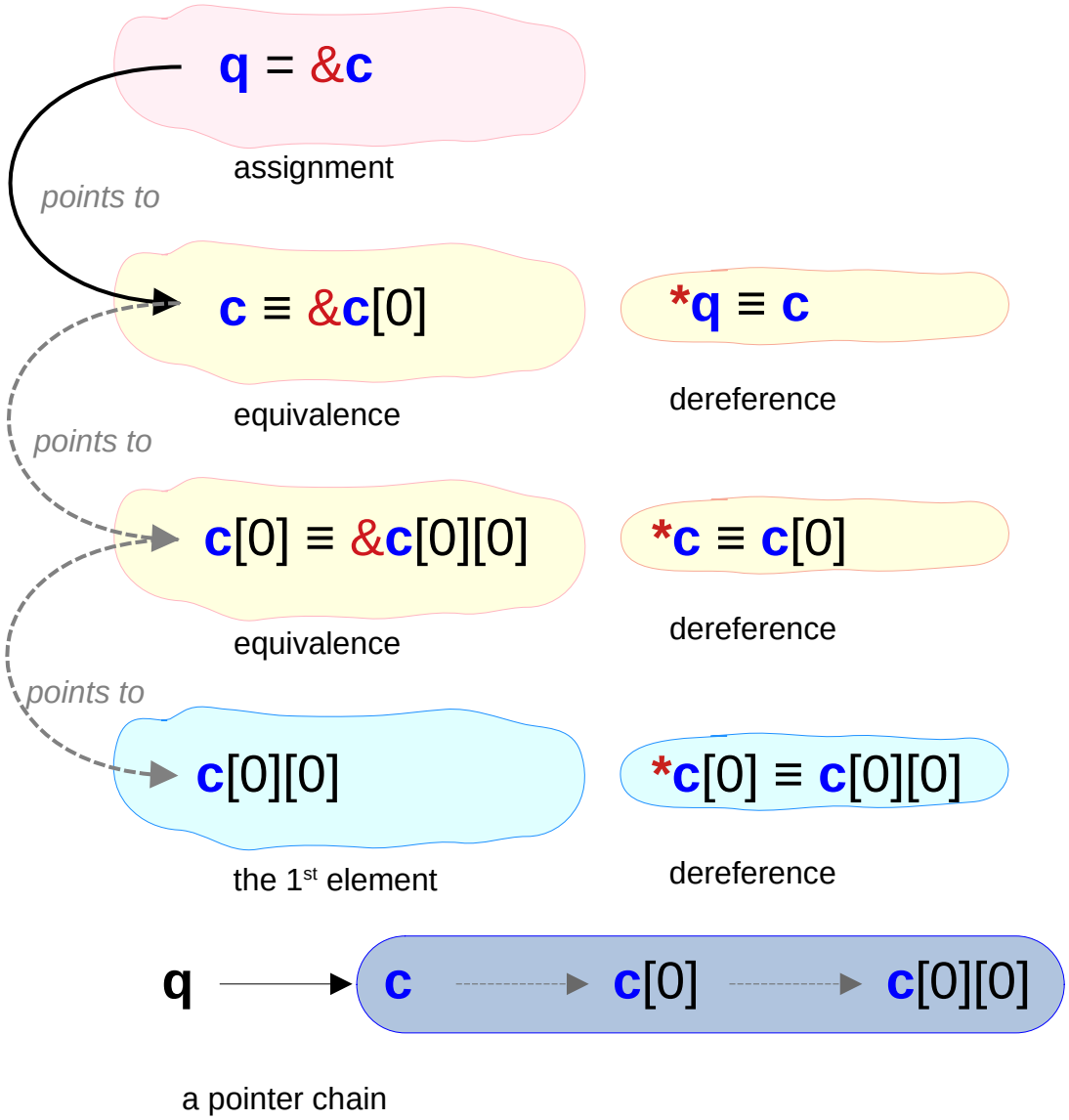


not a real pointer a value(&a) = value(a) = value(&a[0])

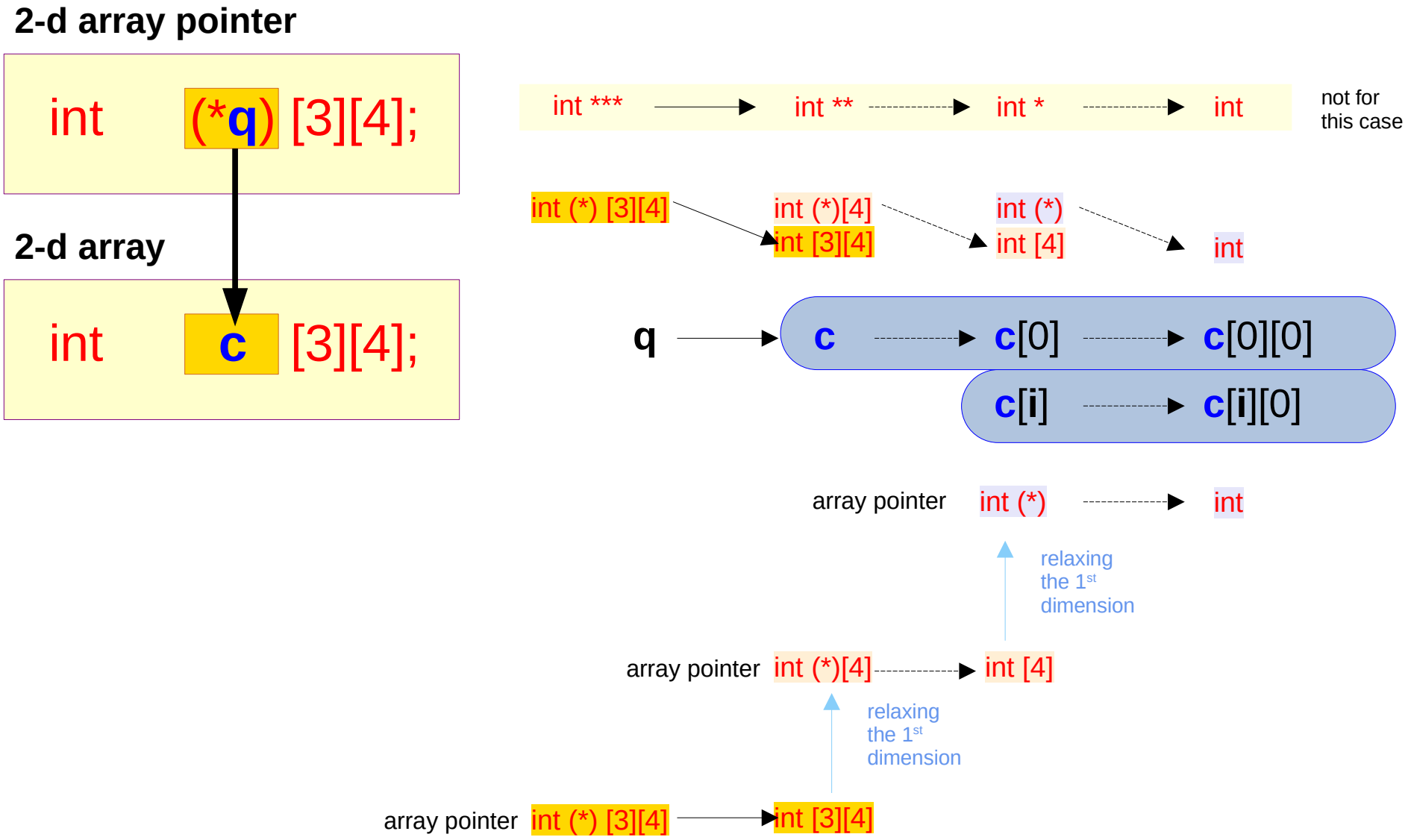
2-d array pointer q – (1) pointer chains



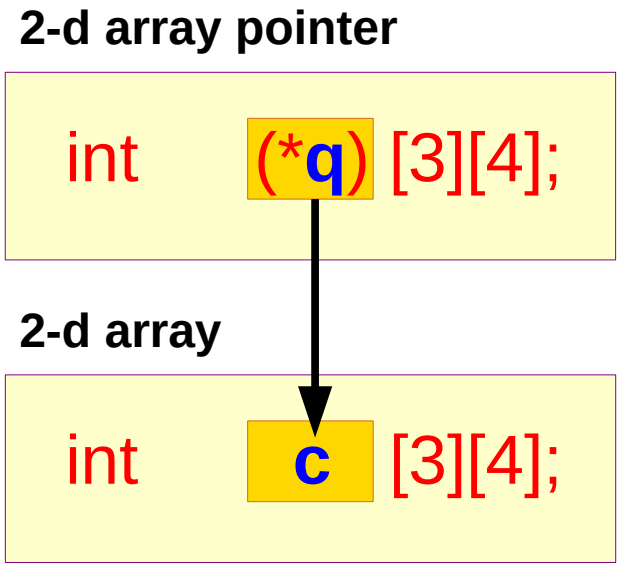


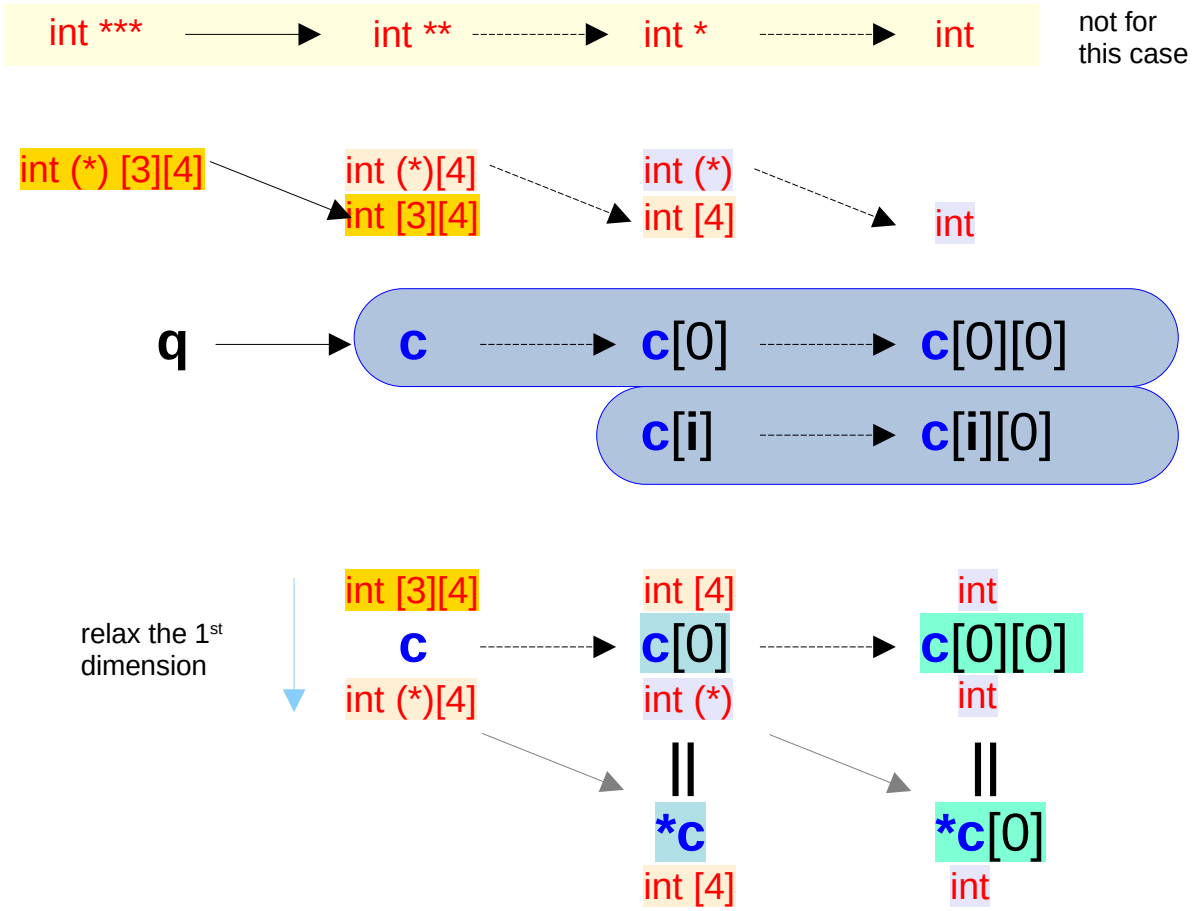


2-d array pointer q – (2) types in a pointer chain



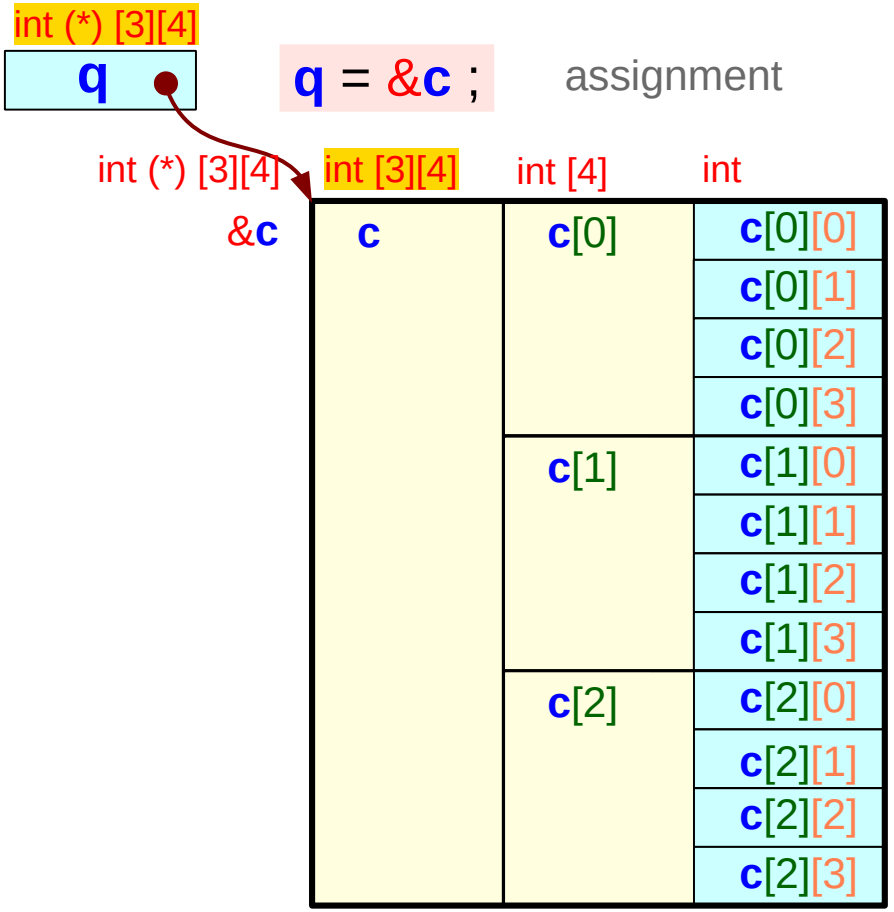
2-d array pointer q – (3) relaxing the 1st dimension

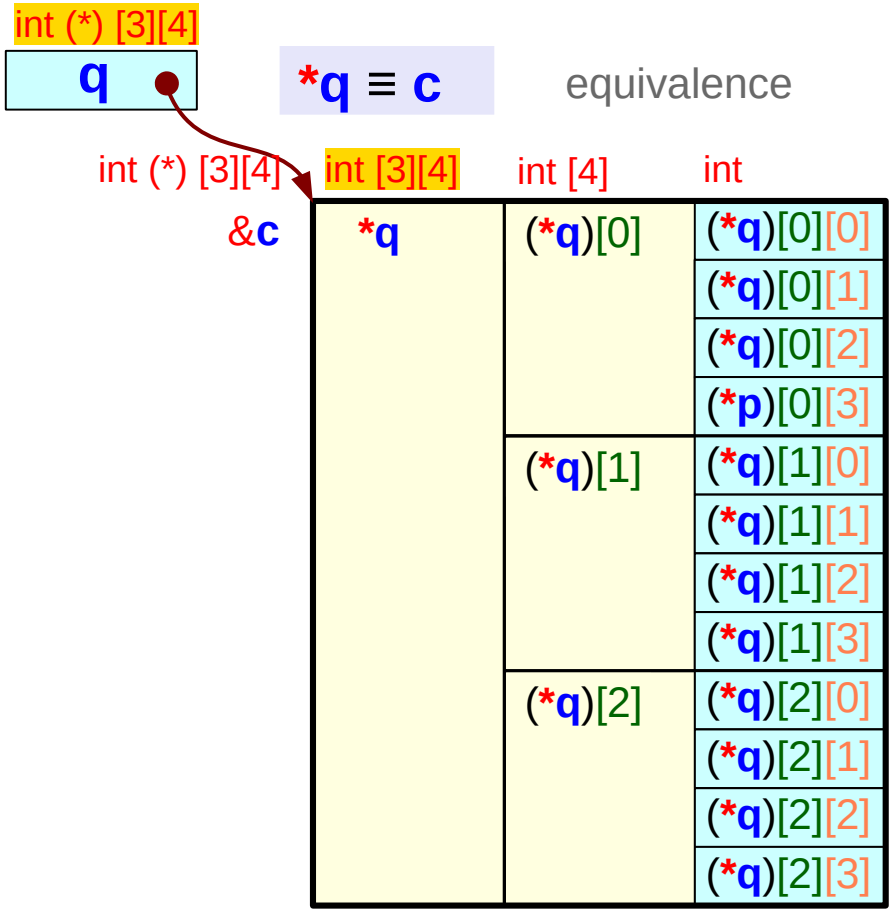




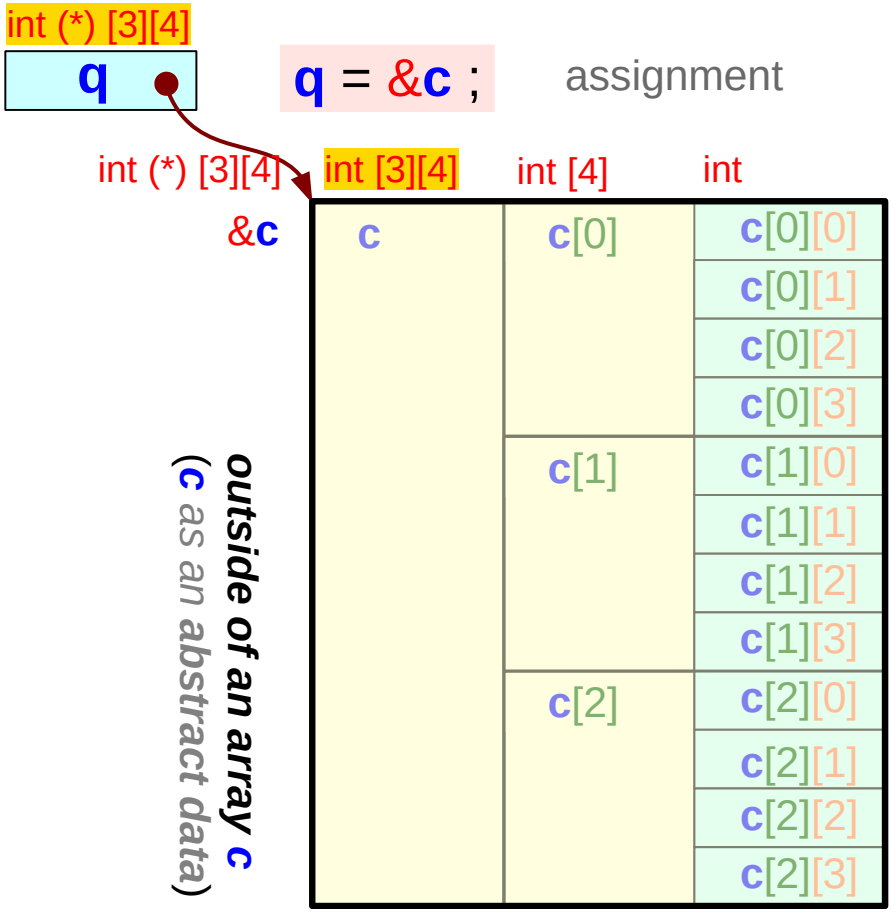
2-d array pointer q – (4) assignment and dereference

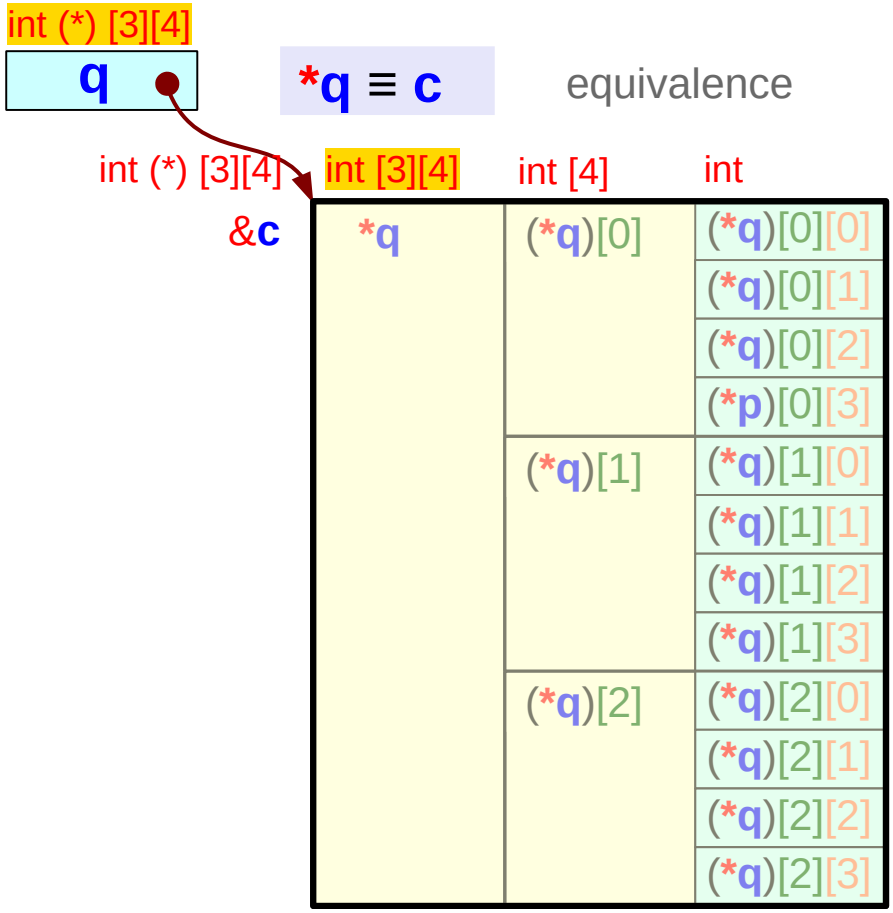
	assignment	dereference	equivalence	equivalence
int c [3][4];				
int (*q) [3][4];	q = &c	*q ≡ c	c ≡ &c[0]	c[0] ≡ &c[0][0]





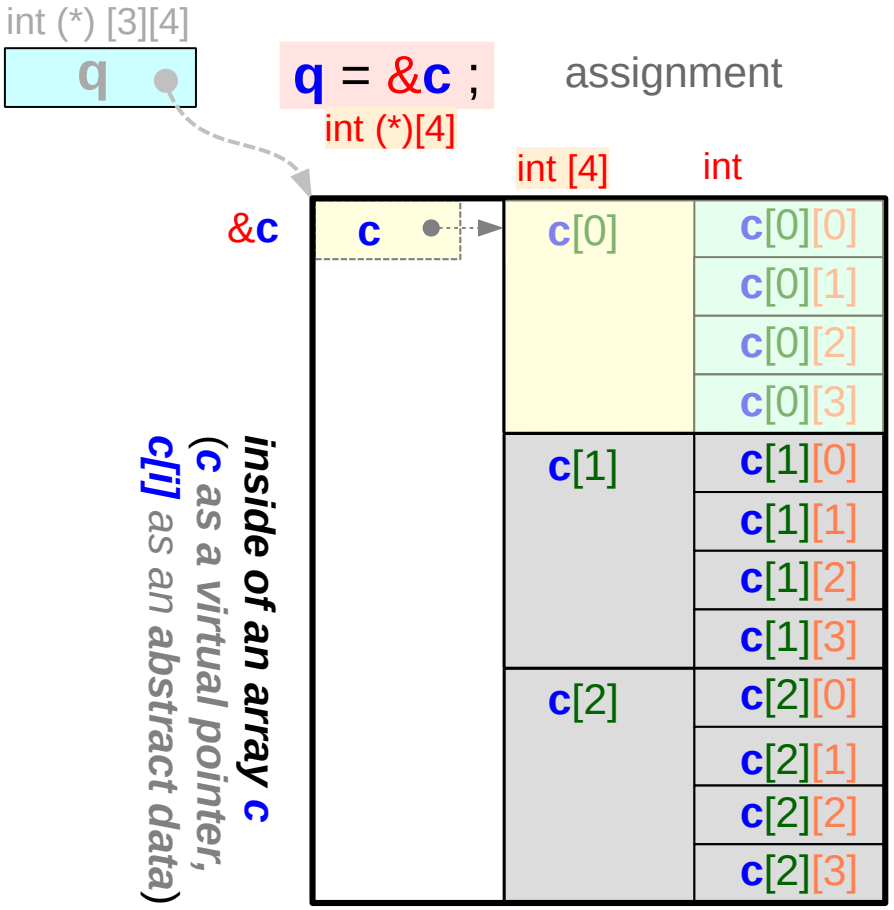
2-d array pointer q – 2-d array c

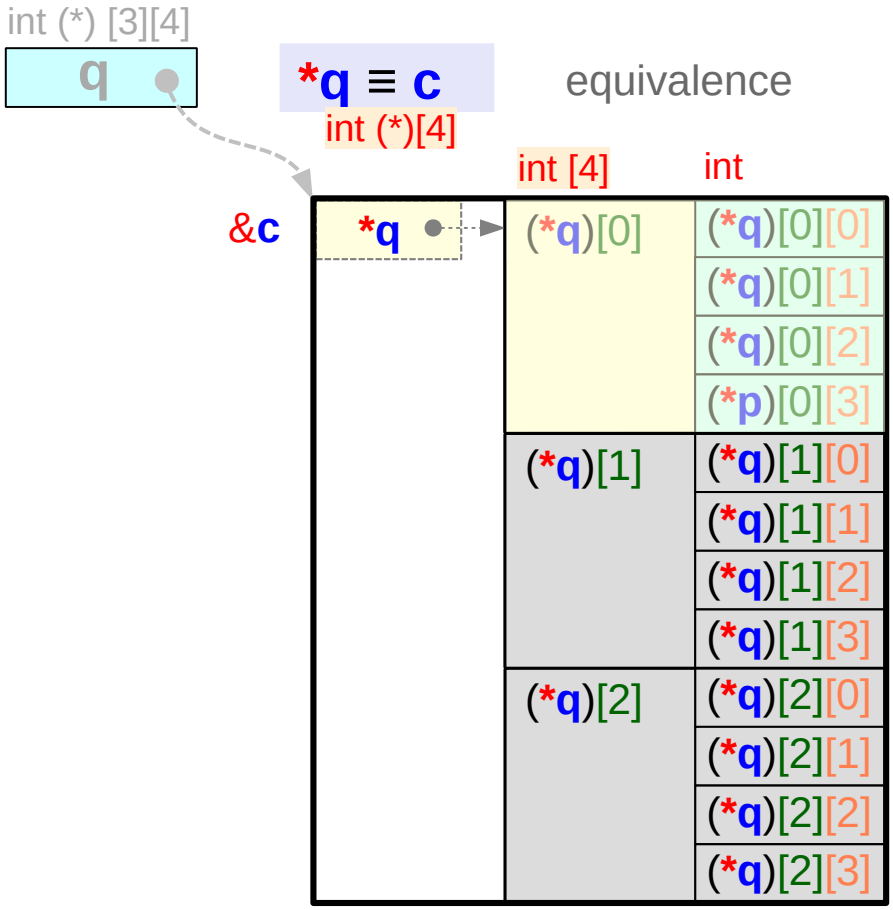




sizeof(&c)	= 4 or 8 bytes	size of a pointer	value(&c) = address value of a **2-d** array **c**
sizeof(**c**)	= 3 * 4 * sizeof(int)	size of a **2-d** array	value(**c**) data value of a **2-d** array **c**

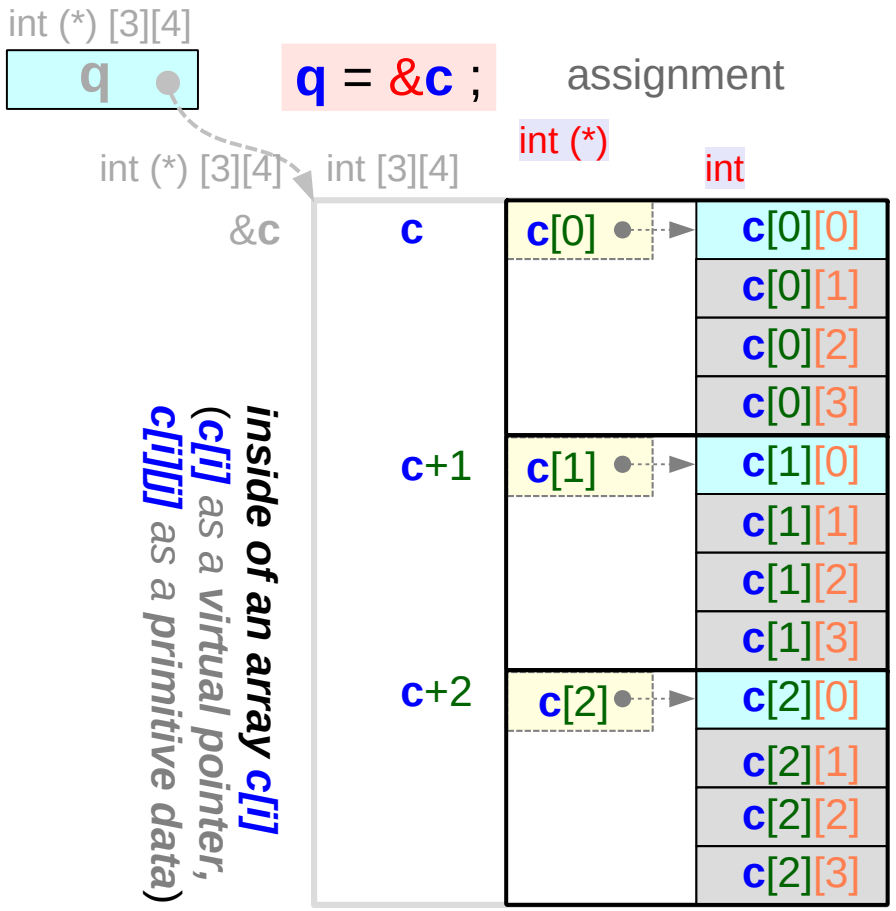
1-d array pointer c – 1-d array c[0]

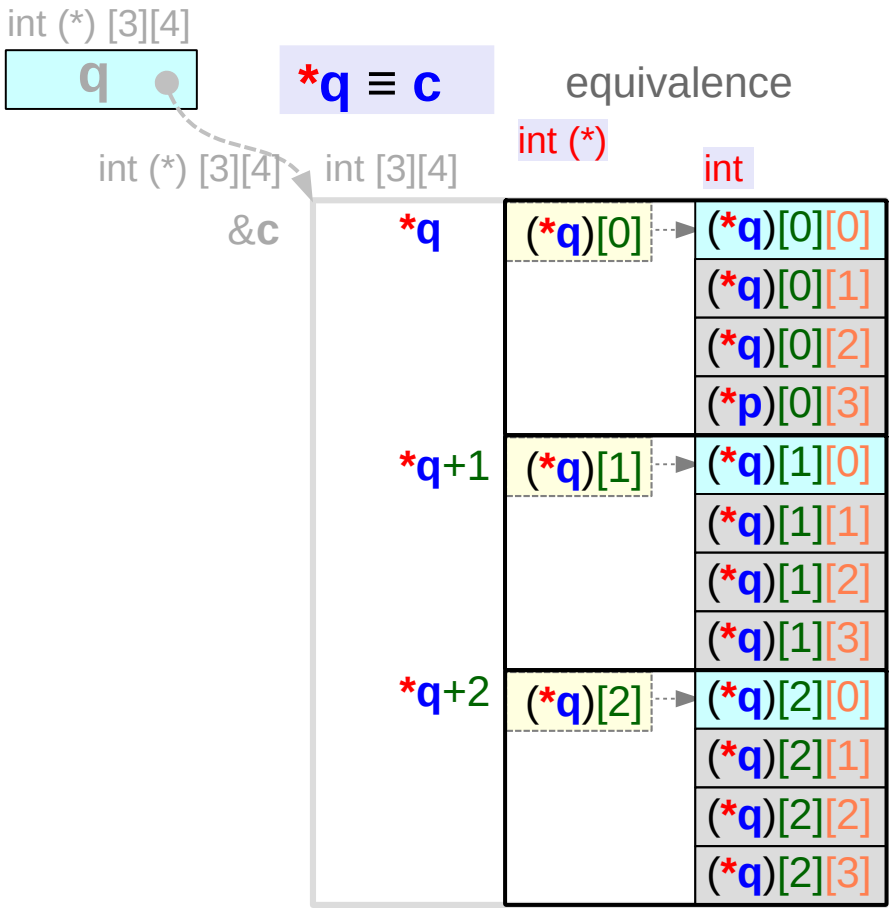




sizeof(**c**) = 3 * 4 * sizeof(int)	size of a **2-d** array	value(**c**) =	address value of a **1-d** array **c**[0]
sizeof(**c**[0]) = 4 * sizeof(int)	size of a **1-d** array	value(**c**[0])	data value of a **1-d** array **c**[0]

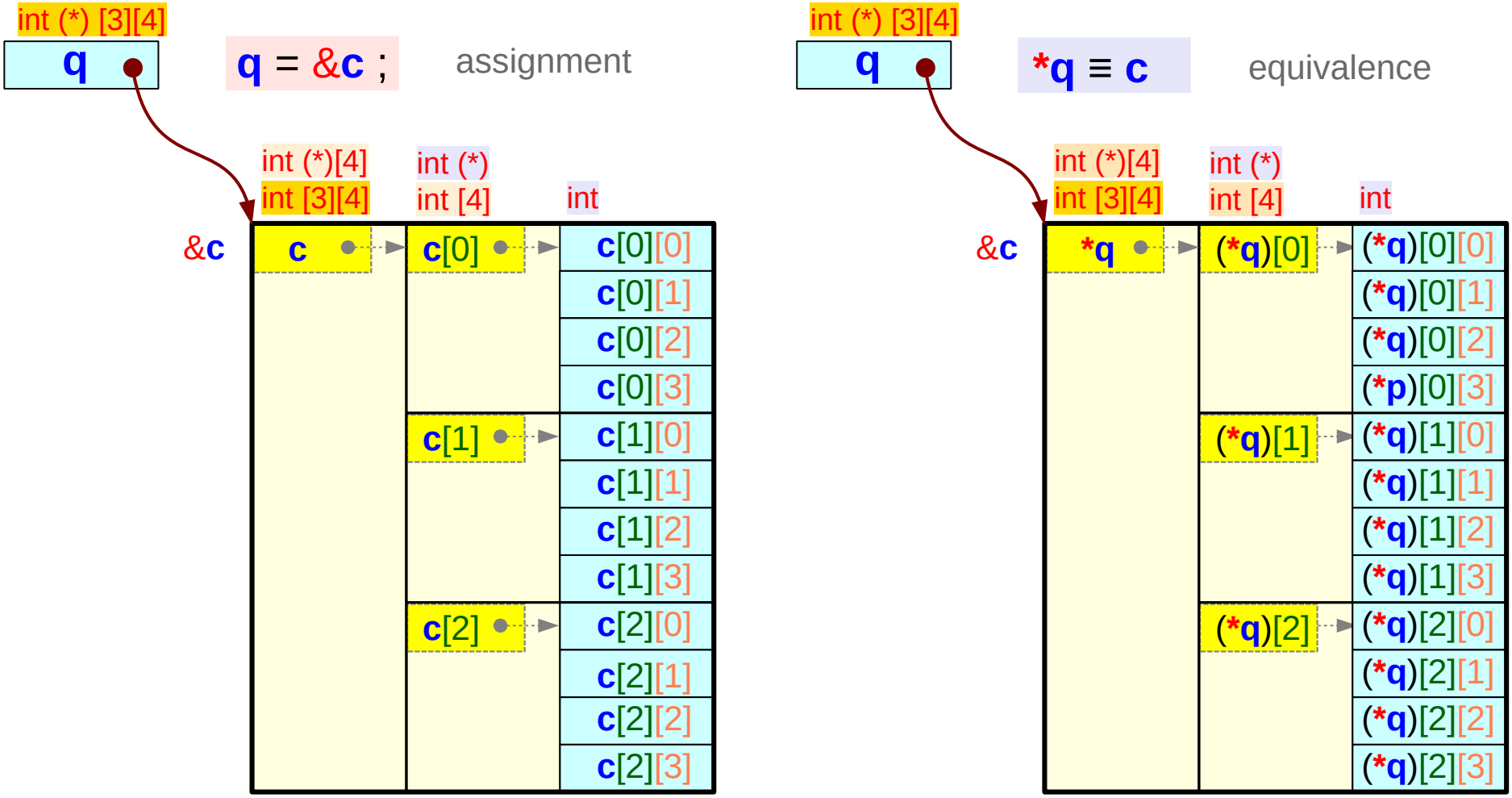
0-d array pointer c[i] – 0-d array c[i][0]





sizeof(**c**[i]) = 4 * sizeof(int)	size of a **1-d** array	value(**c**[i]) = value(&**c**[i][0])	address value of an integer **c**[i][0]
Sizeof(**c**[i][0]) = sizeof(int)	size of an integer	value(**c**[i][0])	data value of an integer **c**[i][0]

Overlayed representation



not a real pointer c

not a real pointer c[i]

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

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

int [4] a **1-d** 4-element integer array

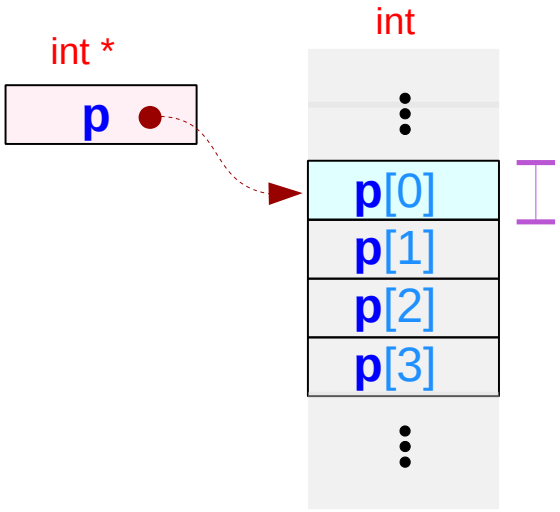
int [ ] a **1-d** unsized integer array

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

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

Sizes of **int [4]**, **int [ ]**, **int *** types

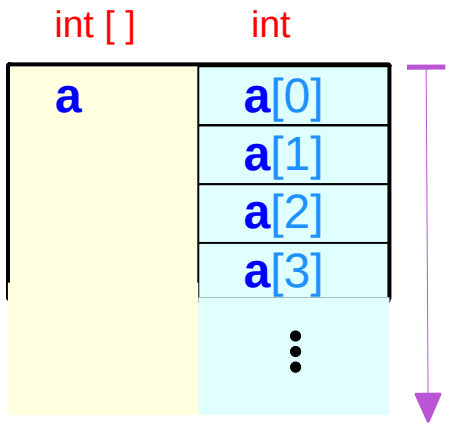
sizeof(int *) = 4 or 8



an incomplete type

undefined size

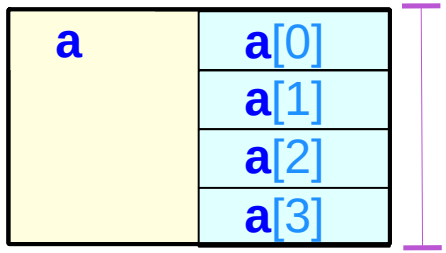
~~sizeof(int [ ])~~



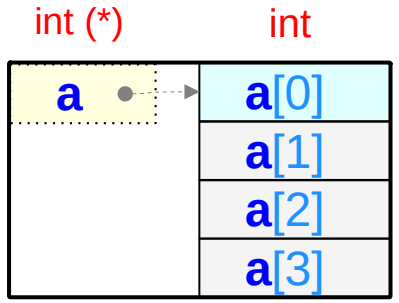
used to pass arrays
to a function

sizeof(int [4])
= sizeof(int) *4 = 16

outside array type



inside array type

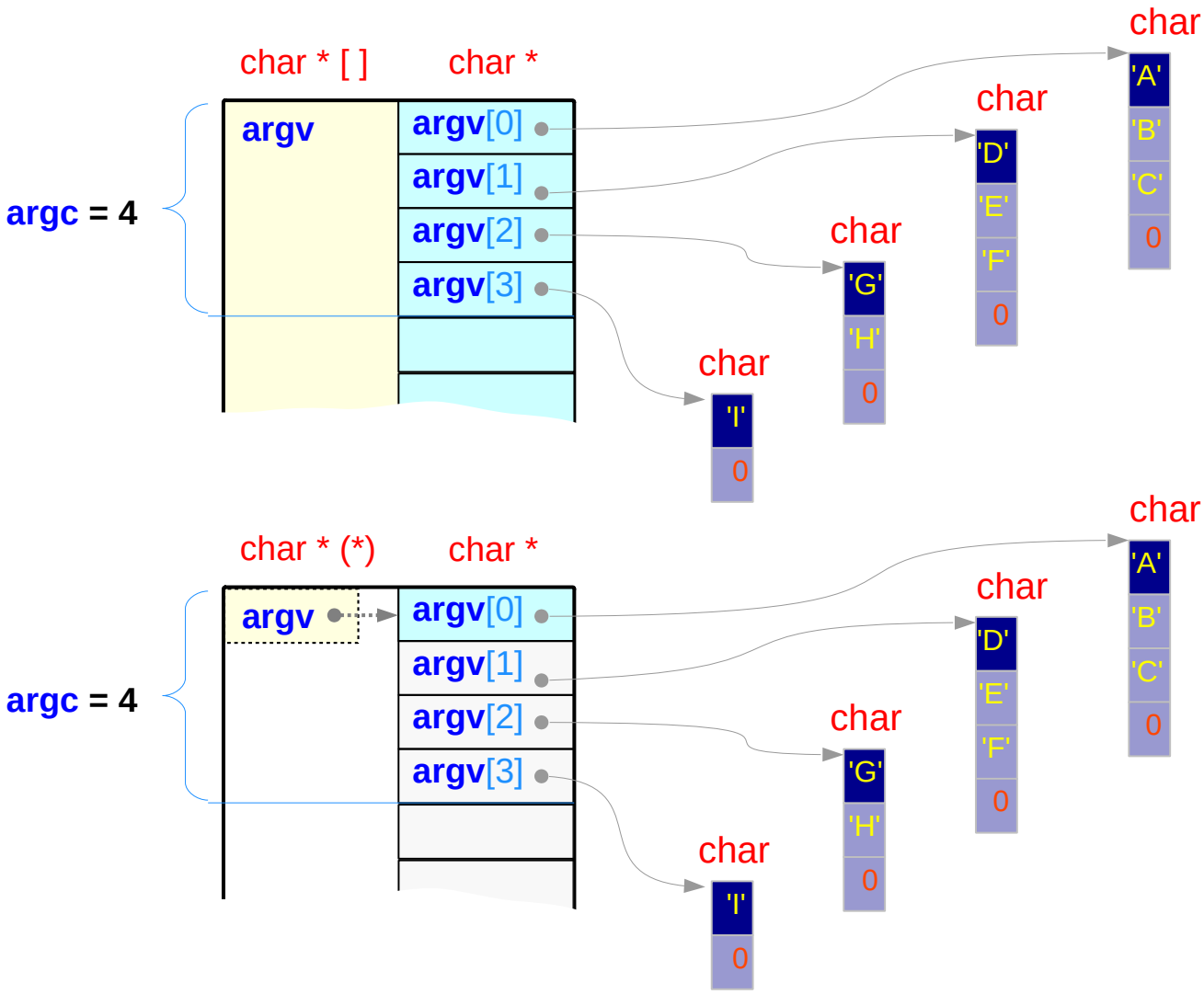


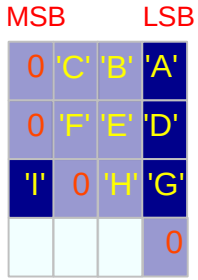
&**a** = value(**a**)

type(**int [4]**) ⊂ type(**int [ ]**) ≈ type(**int (*)**) ≡ **int ***

abstract data — **relaxed data** — **0-d array pointer** — **int pointer**

(int argc, char * argv[ ]) example





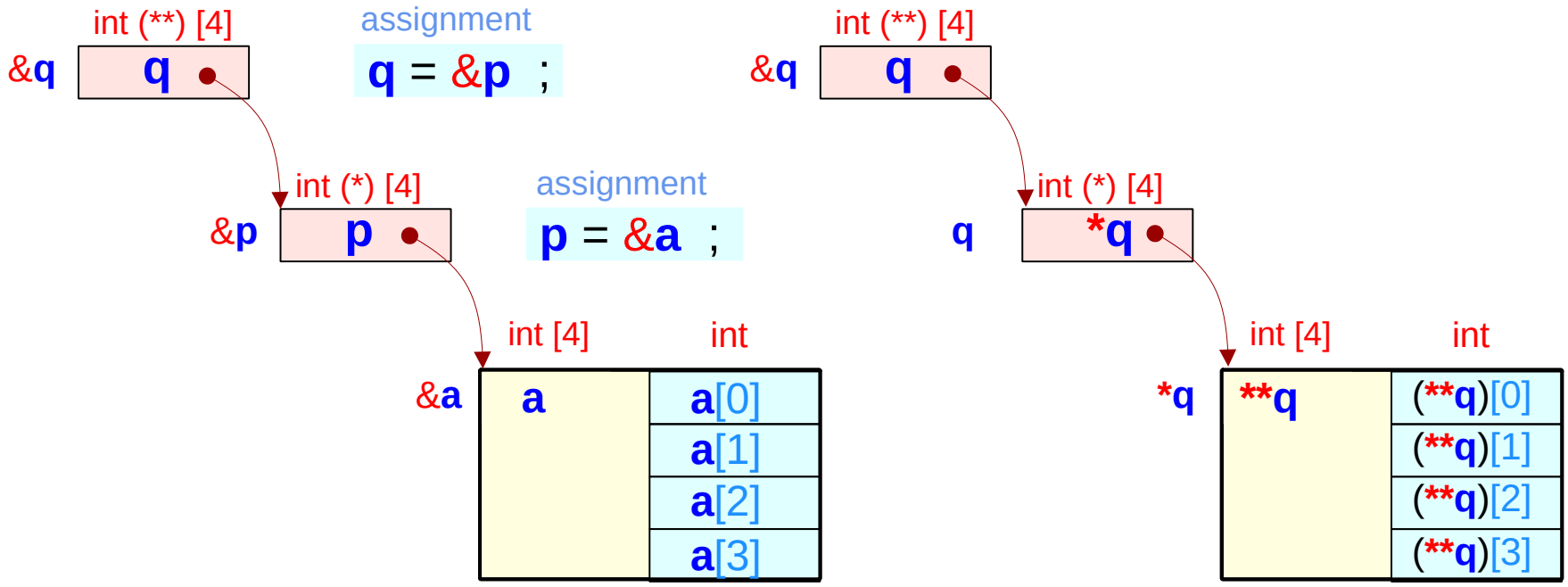
char * [ ] relaxed 1st dimension

char * (*) array pointer

char * *

Double pointer to a **1-d** array – a variable view (p, q)

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

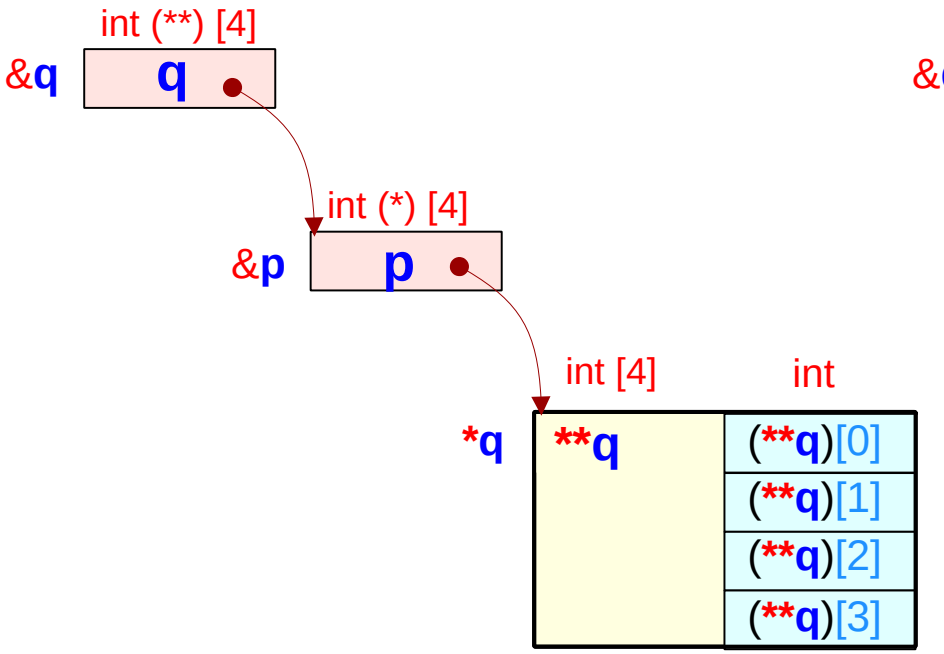


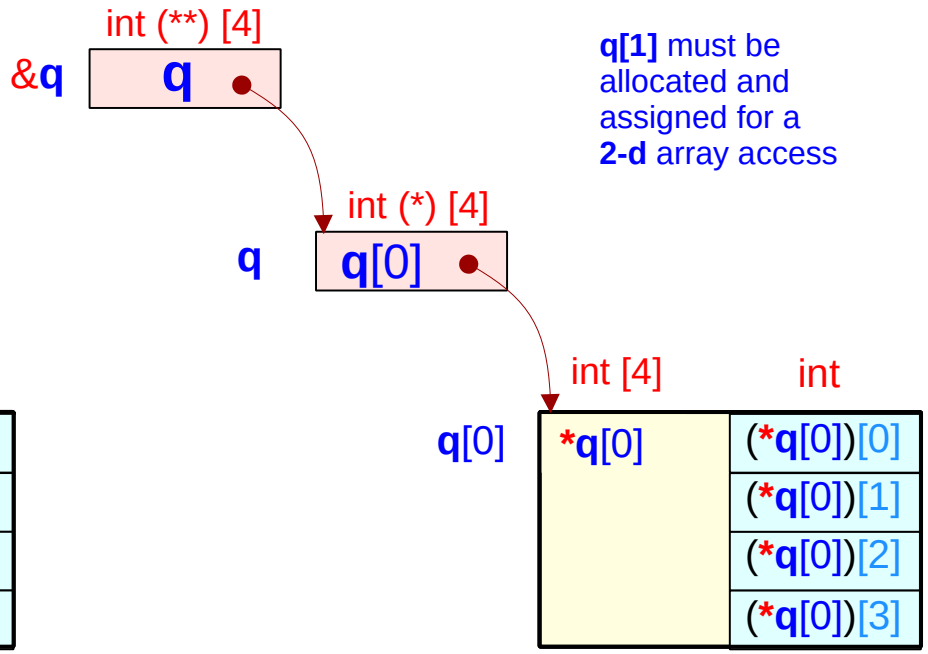
Double array pointer

```
int a[4] ;
```

1-d array a

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

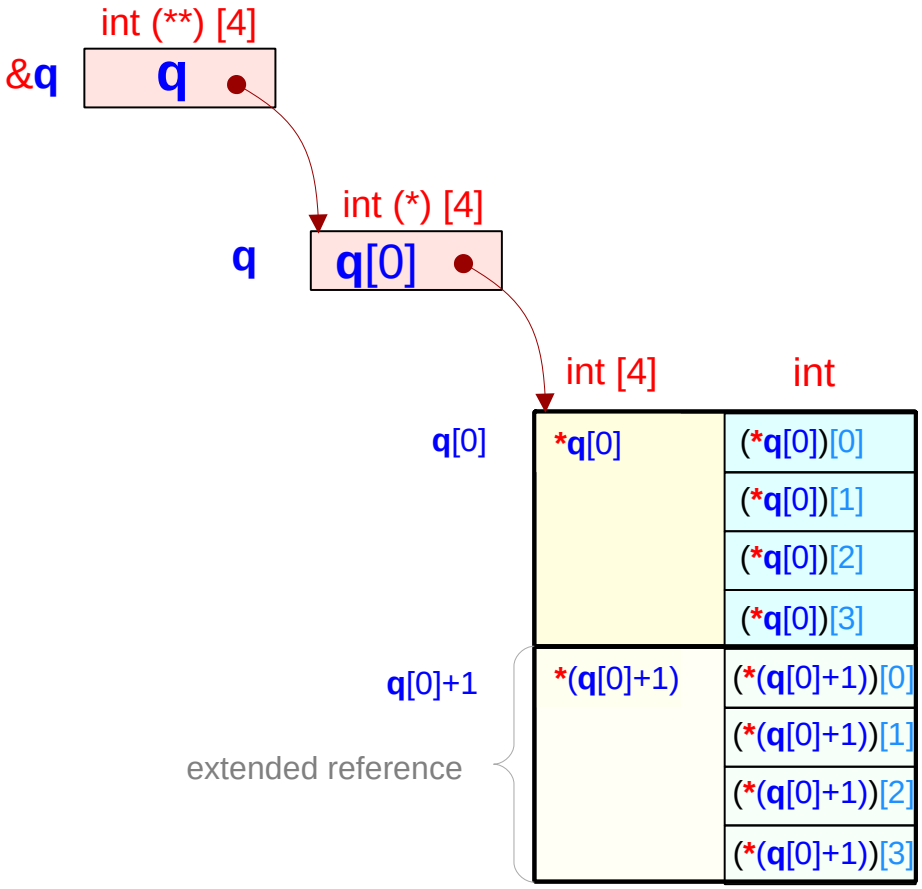




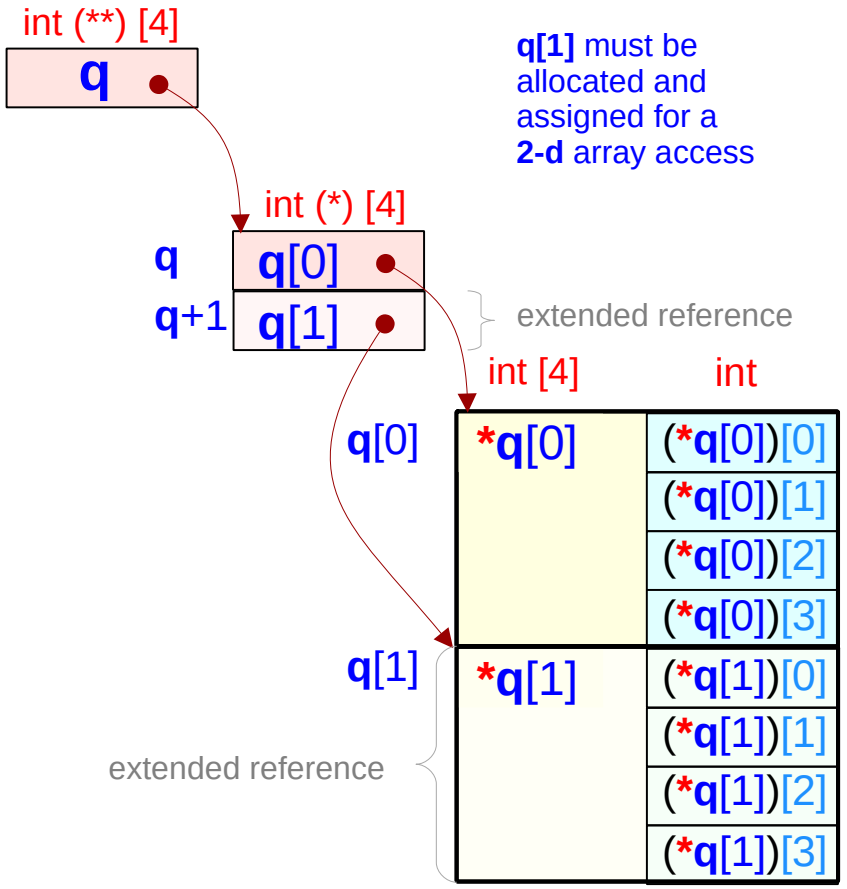
q[1] must be allocated and assigned for a **2-d** array access

2-d array access with a double array pointer

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




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

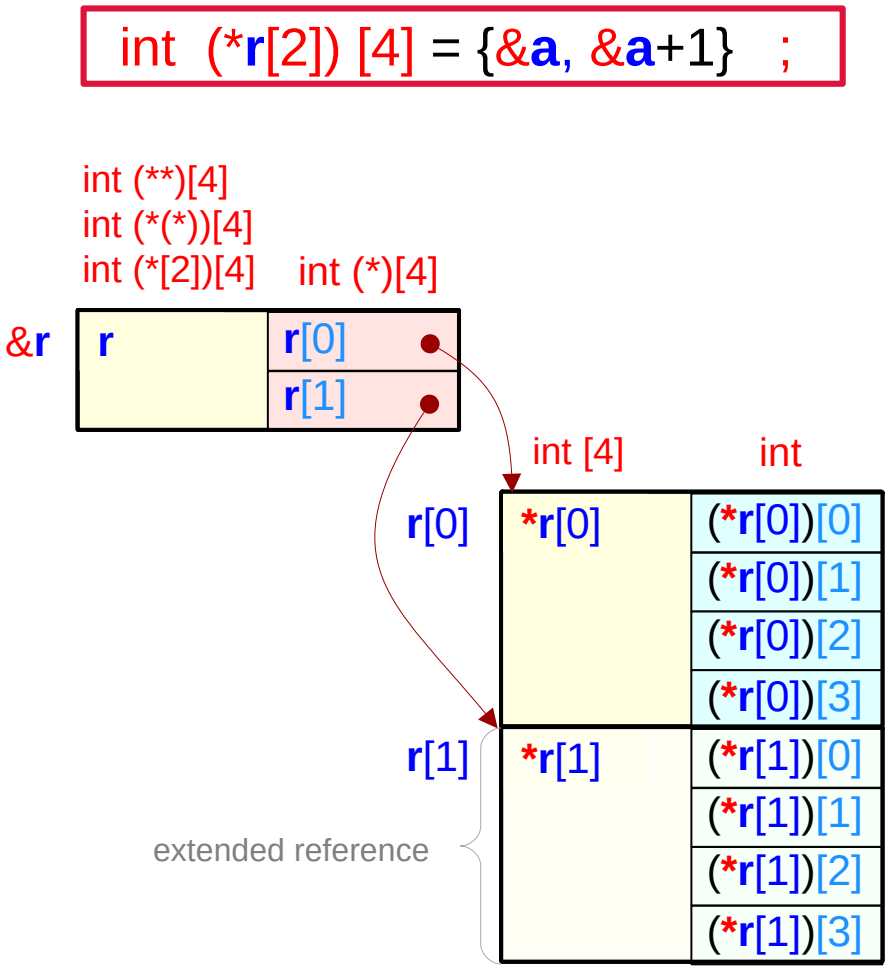


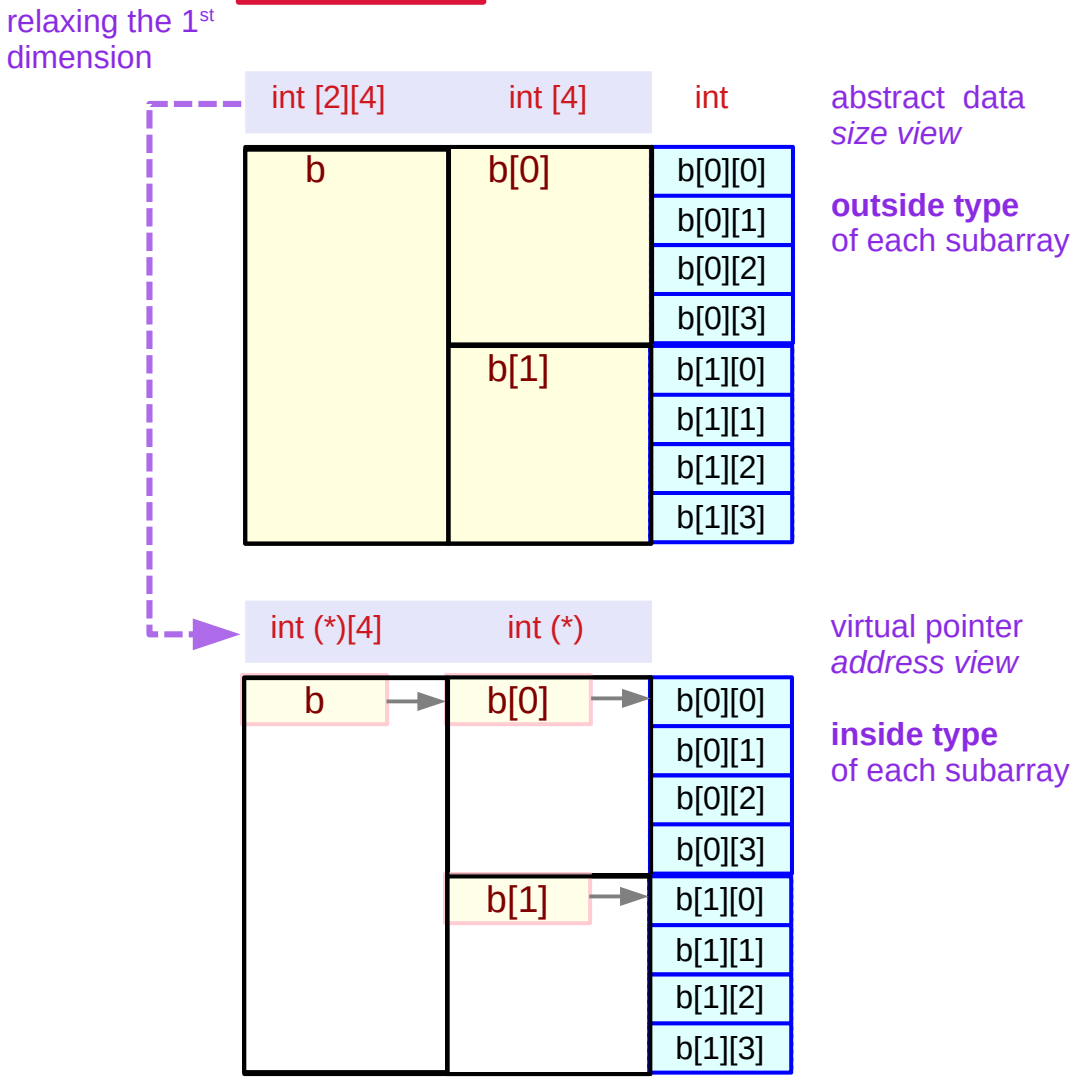
Array pointer and subarray types

```
int  (*r[2]) [4] = {&a, &a+1}  ;
```

```
int b[2][4];
```

2-d array **b**



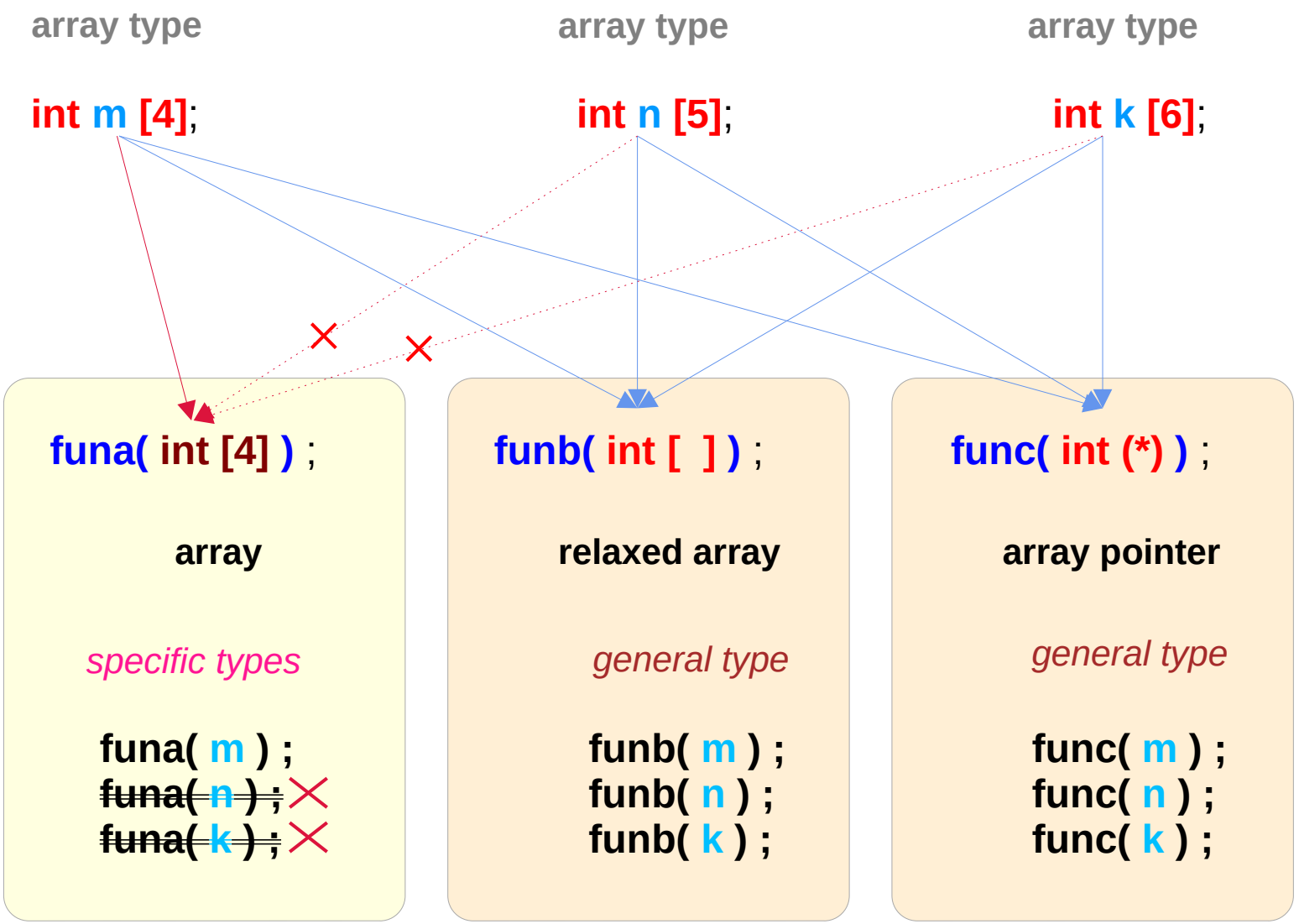


Relaxing the 1st dimension of an array

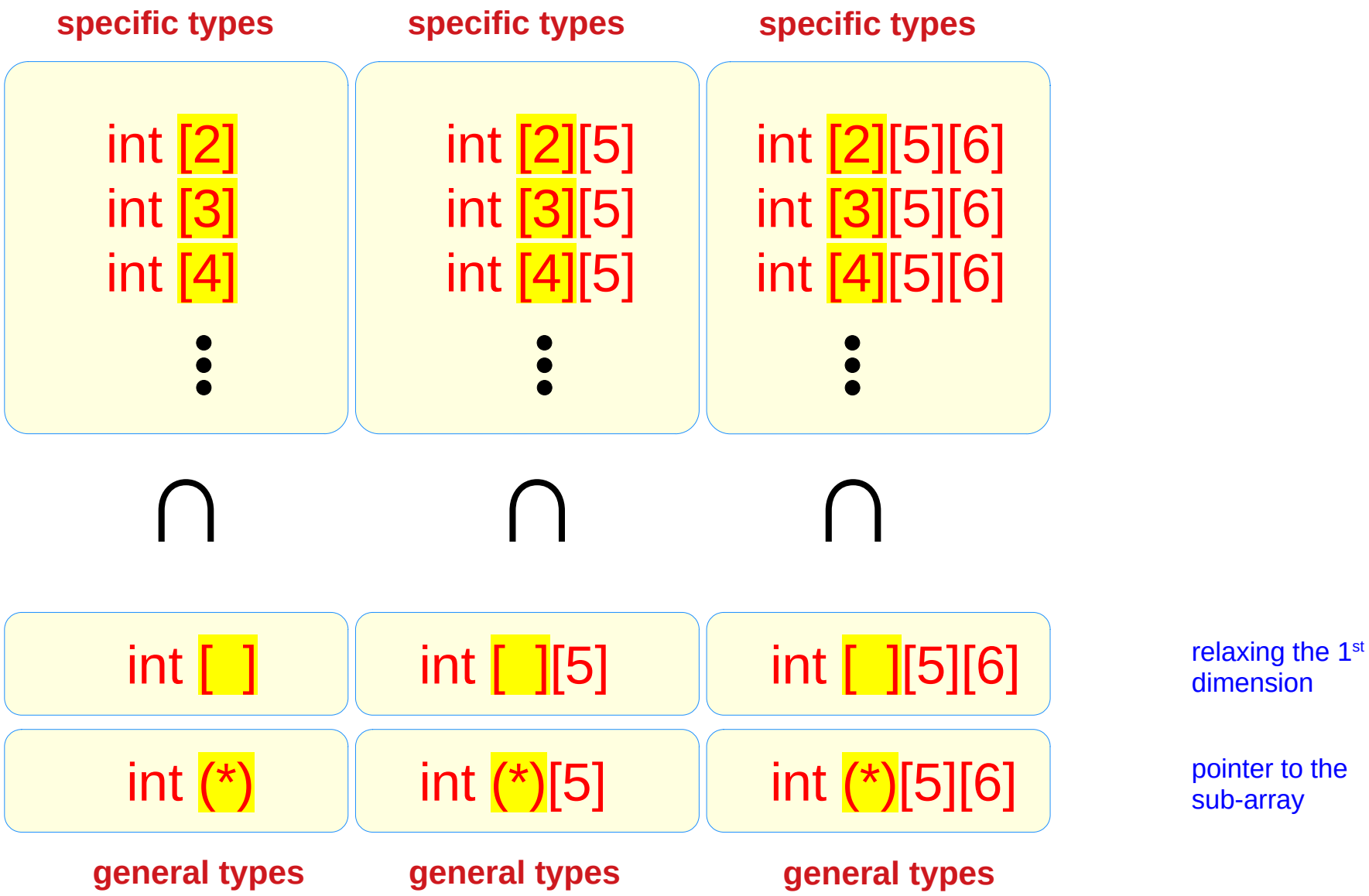
Multi-dimensional array types

array types	function calls	possible function prototypes		
int a [4];	funa(a);	funa(int [4]) ;	array type	*specific type*
		funa(int [ ]) ;	relaxed array	*general type*
		funa(int (*)) ;	array pointer	*general type*
int b [4][5];	funb(b);	funb(int [4][5]) ;	array type	*specific type*
		funb(int [ ][5]) ;	relaxed array	*general type*
		funb(int (*)[5]) ;	array pointer	*general type*
int c [4][5][6];	func(c);	func(int [4][5][6]) ;	array type	*specific type*
		func(int [ ][5][6]) ;	relaxed array	*general type*
		func(int (*)[5][6]) ;	array pointer	*general type*

Multi-dimensional array types



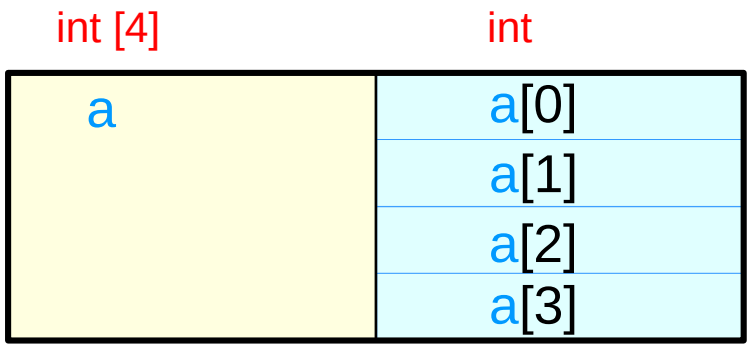
Super types and sub types

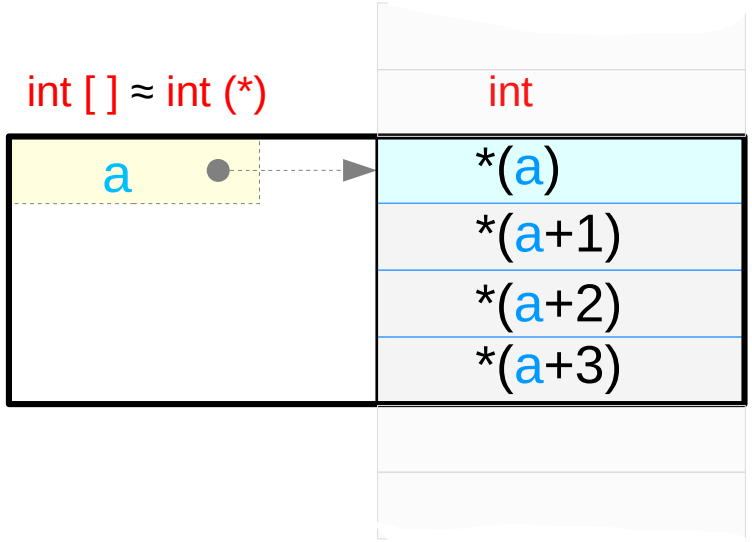


Relaxing array types

int [3][4][5] **3-d** array	int [ ][4][5] the 1st dimension relaxed	int (*)[4][5] **2-d** array pointer
int [4][5] **2-d** array	int [ ][5] the 1st dimension relaxed	int (*)[5] **1-d** array pointer
int [5] **1-d** array	int [ ] the 1st dimension relaxed	int (*) **0-d** array pointer
	undefined size	pointer size

Passing an individual element by reference





int [4] ⊂

int [ ]

≀≀

int (*)

|||

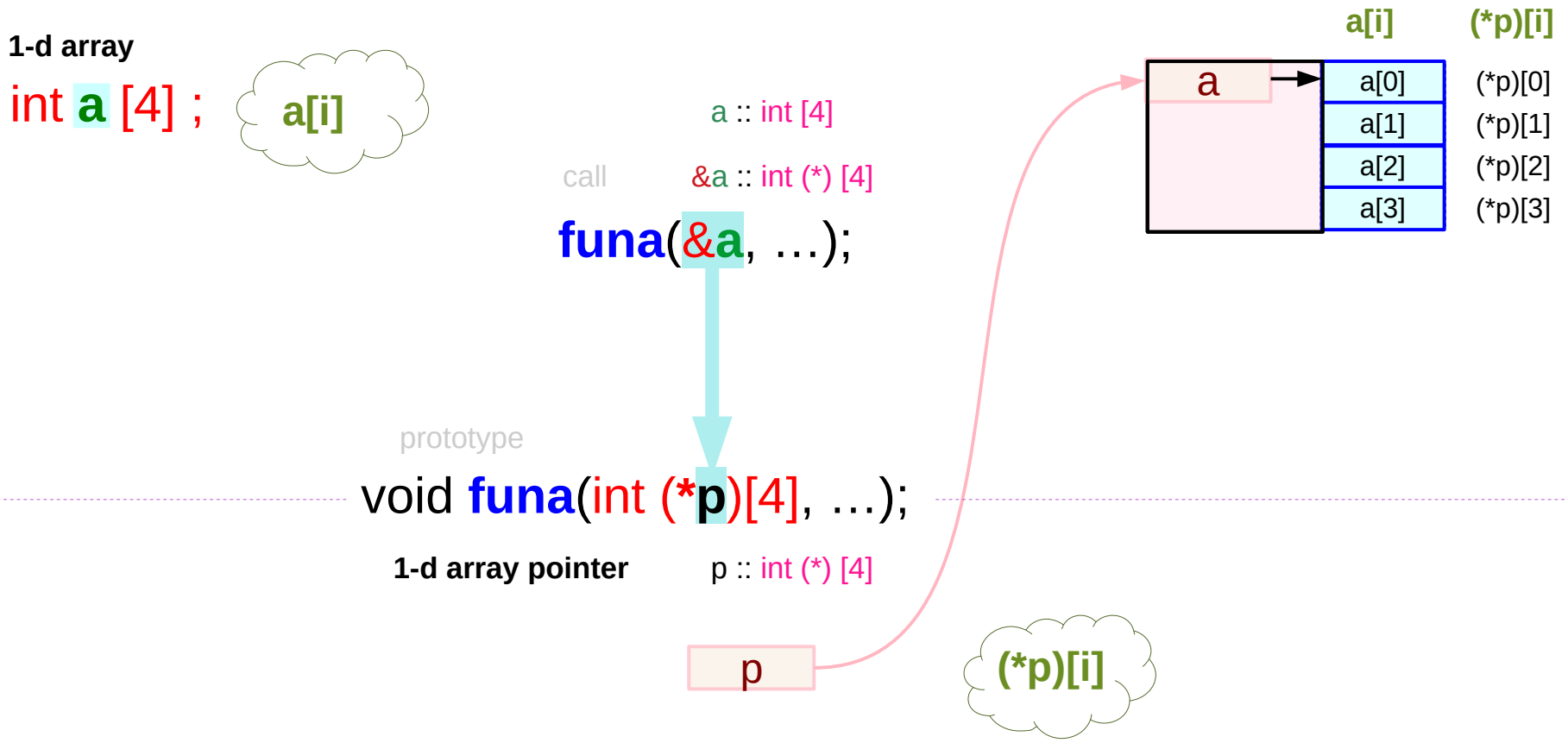
int *

relaxing the 1st dimension

pointer to a **0-d array**

pointer to an **integer**

Passing a *n*-d array pointer



Passing a (*n-1*)-**d** array pointer

1-d array

int **a** [4] ;

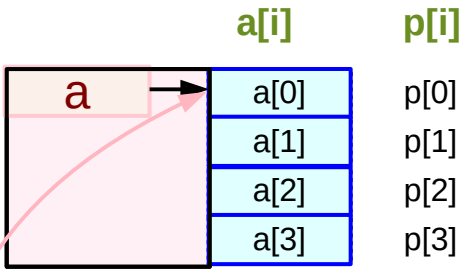
call a :: int [4]

funa(**a**, …);

prototype

void **funa**(int (***p**), …) {

0-d array pointer p :: int (*)



p

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

}

Passing *n*-d array pointers

1-d array	call	prototype 1-d array pointer
int **a** [4] ;	**funa**(&**a**, …);	void **funa**(int (***p**)[4], …);
a[i]		(*p)[i]

2-d array	call	prototype 2-d array pointer
int **b** [4][2];	**funb**(&**b**, …);	void **funb**(int (***q**)[4][2], …);
b[i][j]		(*q)[i][j]

3-d array	call	prototype 3-d array pointer
int **c** [4][2][3];	**func**(&**c**, …);	void **func**(int (***r**)[4][2][3], …);
c[i][j][k]		(*r)[i][j][k]

4-d array	call	prototype 4-d array pointer
int **d** [4][2][3][4];	**fund**(&**d**, …);	void **fund**(int (***s**)[4][2][3][4], …);
d[i][j][k][l]		(*s)[i][j][k][l]

Passing (*n*-1)-**d** array pointers

1-d array	call	prototype 0-d array pointer
int **a**[4] ;	**funa**(**a**, …);	void **funa**(int (***p**), …);
a[i]		p[i]

2-d array	call	prototype 1-d array pointer
int **b**[4] [2];	**funb**(**b**, …);	void **funb**(int (***q**)[2], …);
b[i][j]		q[i][j]

3-d array	call	prototype 2-d array pointer
int **c**[4] [2][3];	**func**(**c**, …);	void **func**(int (***r**)[2][3], …);
c[i][j][k]		r[i][j][k]

4-d array	call	prototype 3-d array pointer
int **d**[4] [2][3][4];	**fund**(**d**, …);	void **fund**(int (***s**)[2][3][4], …);
d[i][j][k][l]		s[i][j][k][l]

Receiving (*n*-1)-**d** array pointers

1-d array

```
int a [4] ;
```

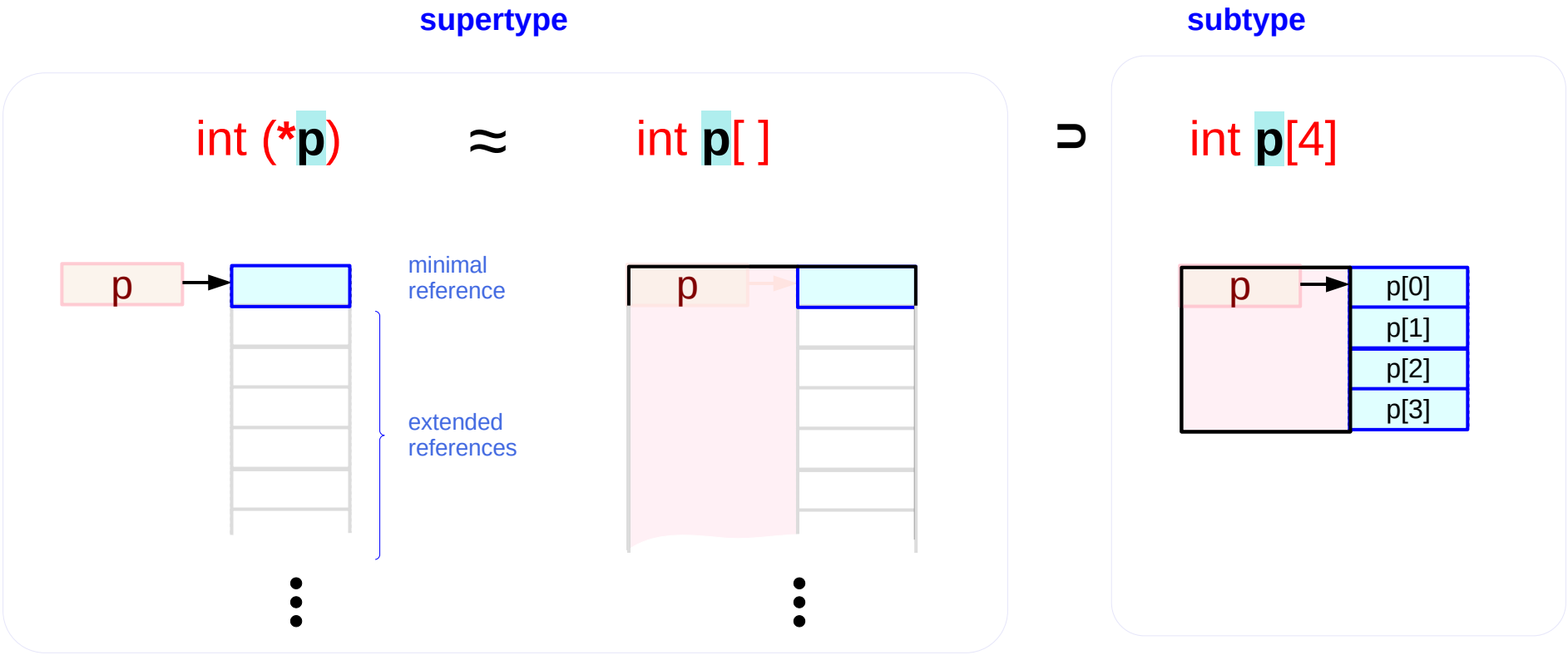
call a :: int [4]

```
funa(a, …);
```


prototype

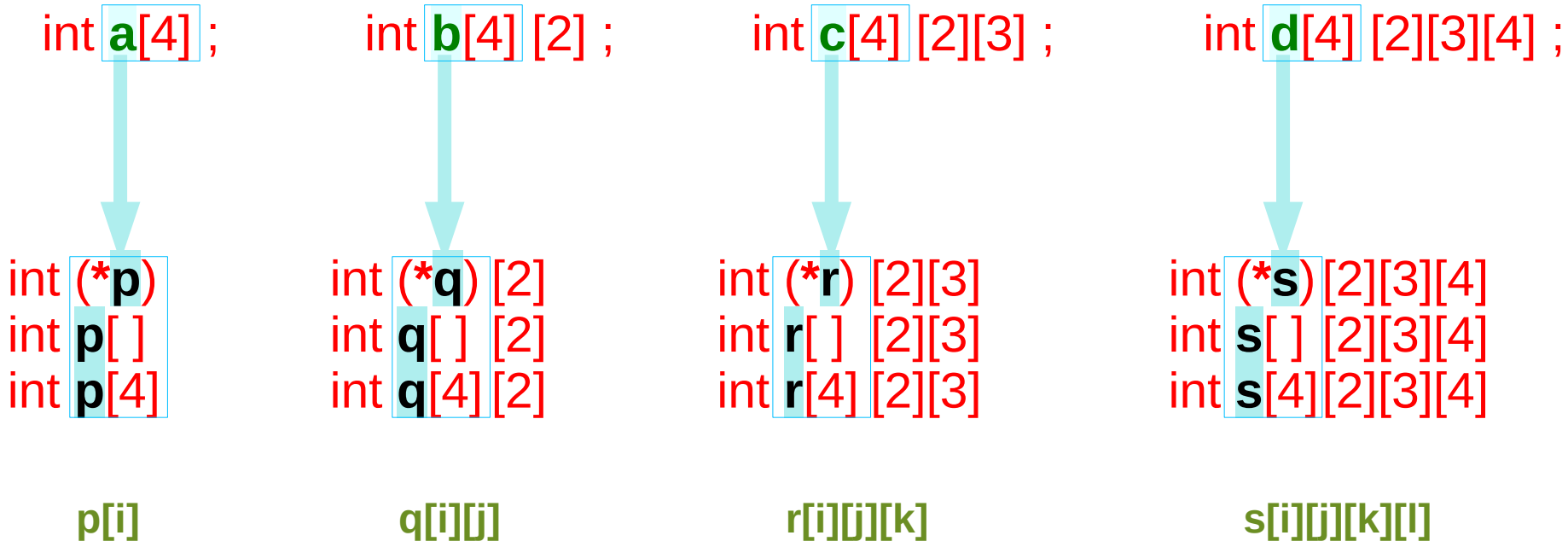
`void funa(int (*p), …) {`	**0-d array pointer**	p :: int (*)	**a[i]**
`void funa(int p[ ], …) {`	**1-d array, relaxed**	p :: int [ ]	**a[i]**
`void funa(int p[4], …) {`	**1-d array**	p :: int [4]	**a[i]**

int (*) and int [ ] types



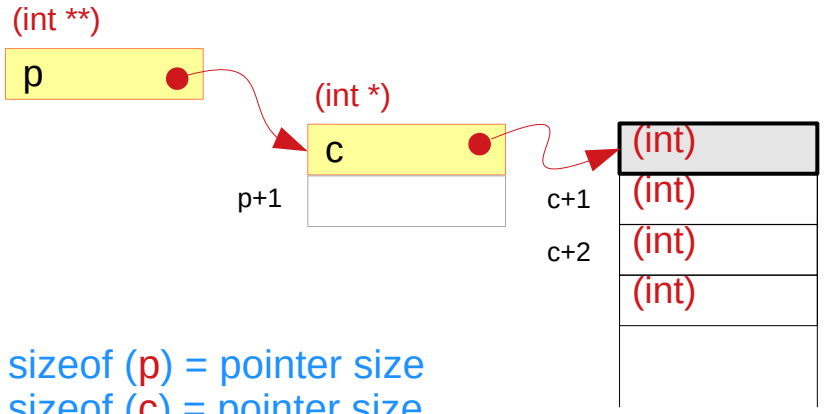
int p[ ] unsized array expression is only allowed
- in a function definition
- in an initialization

The 1st dimensions



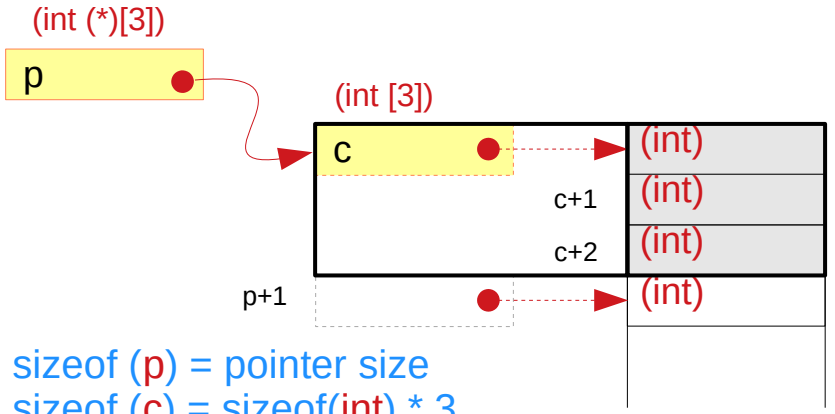
Integer pointer and array types – int **, int (*)[3], int[2][3]

int **p; **int *c;** v(&c) ≠ v(c)



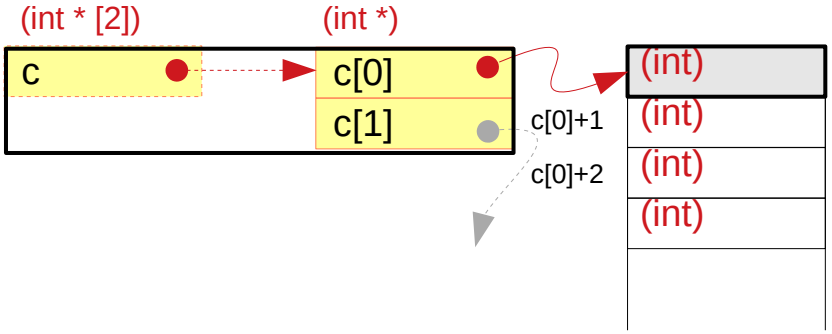
sizeof (p) = pointer size
sizeof (c) = pointer size

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



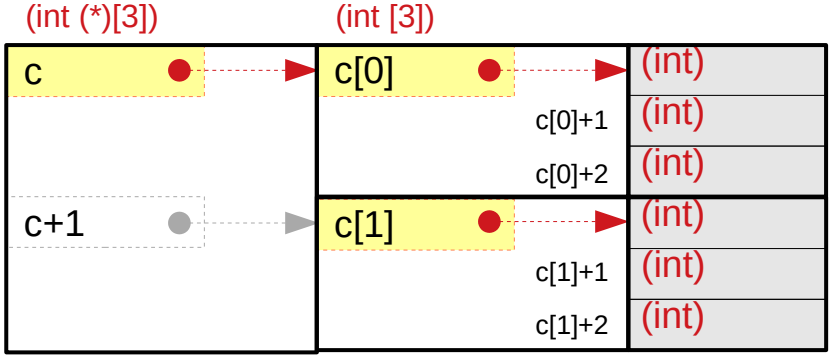
sizeof (p) = pointer size
sizeof (c) = sizeof(int) * 3

int* c[2]; v(&c[0]) ≠ v(c[0])



sizeof (c) = pointer size * 2
sizeof (c[0]) = pointer size

int c[2][3]; v(&c)=v(c) = v(&c[0])=v(c[0])



sizeof (c) = sizeof(int) * 2 * 3
sizeof (c[0]) = sizeof(int) * 3

Integer pointer types

```
#include <stdio.h>

void func(int d[ ])
{

}

int main(void) {
  int a[4];
  int *b;
  int **c;

  int (*p)[4];

  func(a);


}
```

```
sizeof(a)     = 16 = 4*4    // array size
sizeof(*a)    = 4           // int size

sizeof(b)     = 8           // pointer size
sizeof(*b)    = 4           // int size

sizeof(c)     = 8           // pointer size
sizeof(*c)    = 8           // pointer size
sizeof(**c)   = 4           // pointer size

sizeof(d)     = 8           // pointer size
sizeof(*d)    = 4           // int size

sizeof(p)     = 8           // pointer size
sizeof(*p)    = 16 = 4*4    // array size
```

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