

Array Pointers (1A)

Copyright (c) 2022 - 2010 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

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

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

type(c) can be determined by the warning message 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= %p \n", &c);
}
```

`c= 0x7fffd923487c`

```
#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

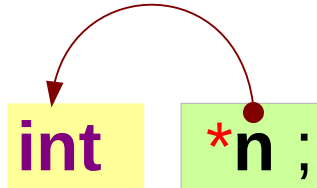
```
int func (int a, int b) ;
```

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

a function pointer

Pointer to an array – variable declarations

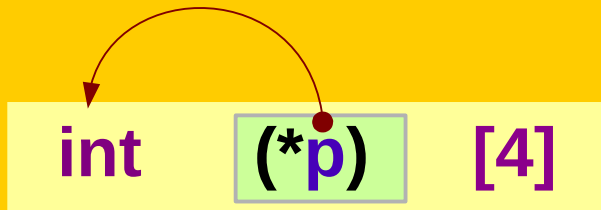
n points to a number (`int`)



an integer pointer

Array **Pointer Approach**
(**pointer to arrays**)

p points to an array (`int [4]`)



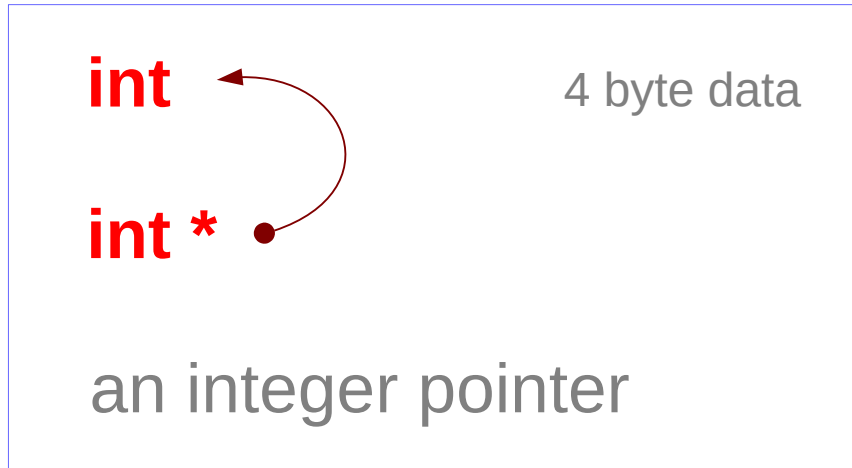
an array pointer

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

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

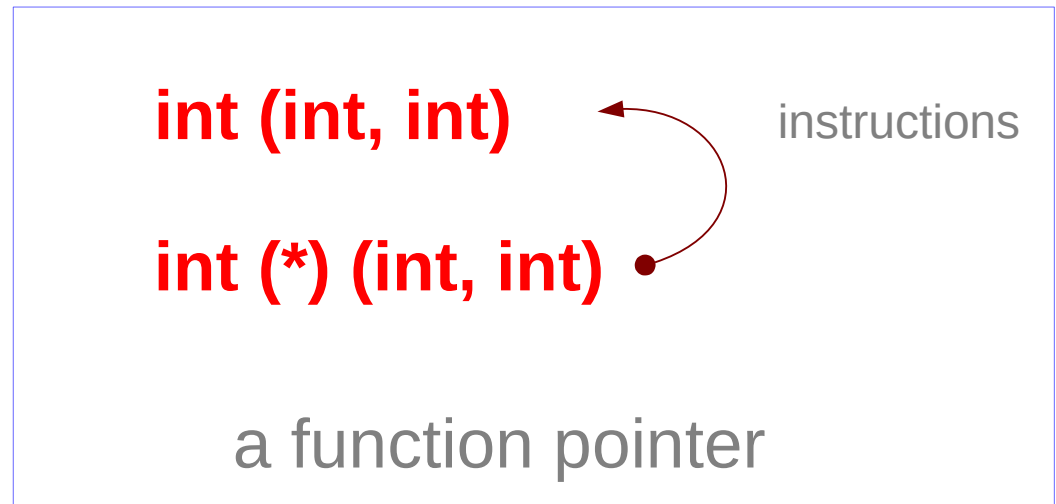
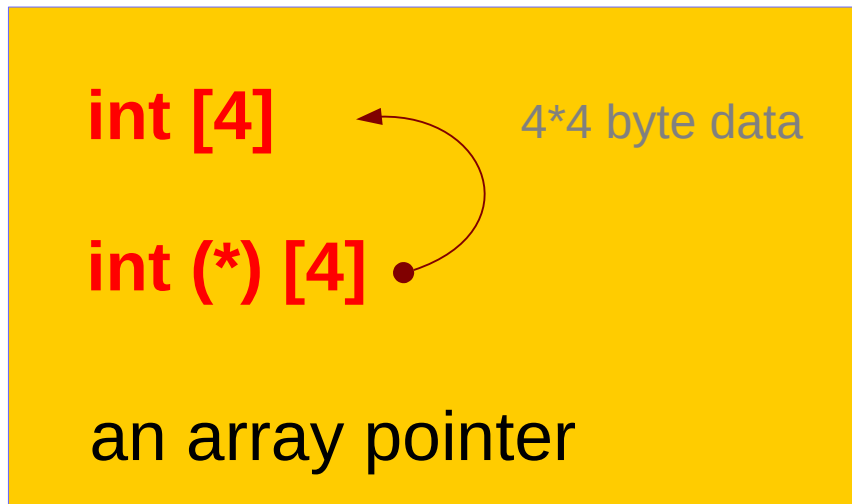
a function pointer

Pointer to an array – a type view

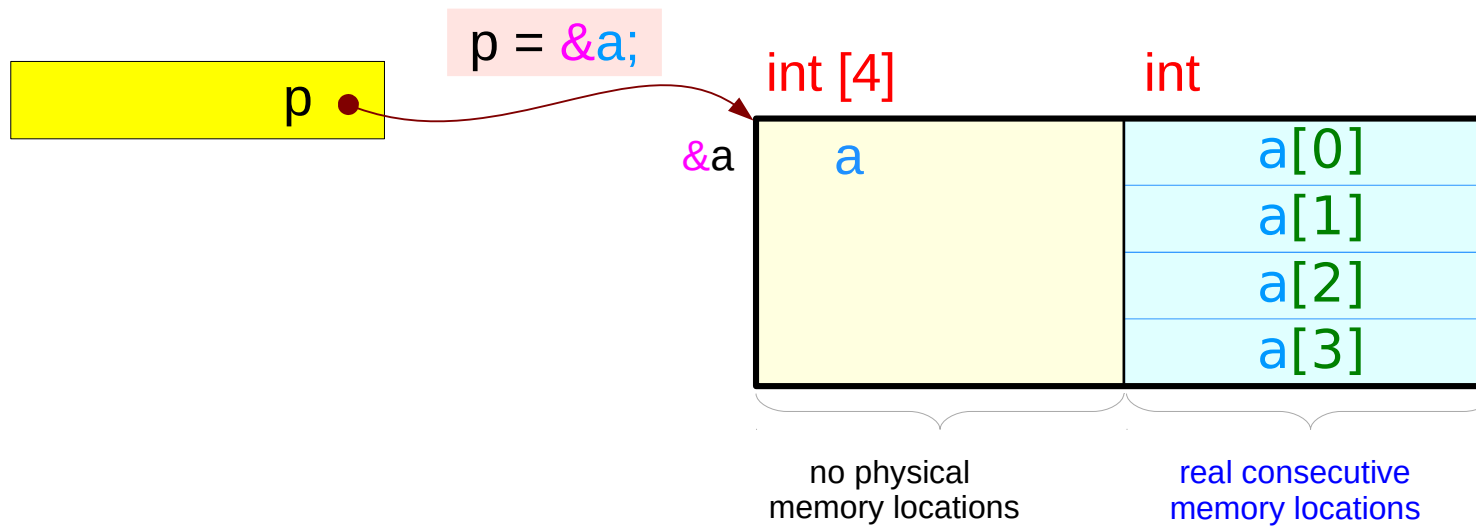


array pointer:
a pointer to an array

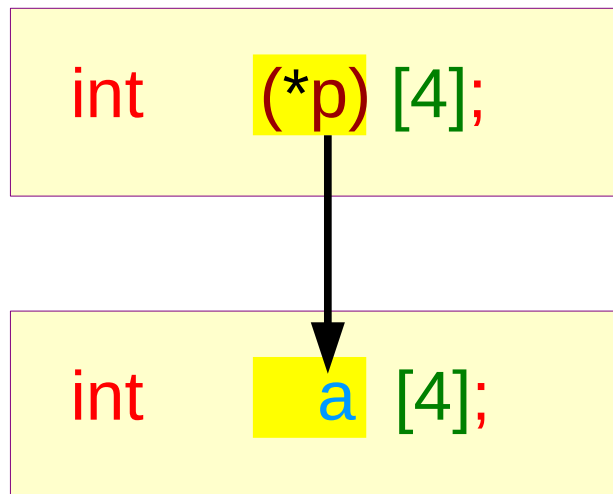
pointer array:
an array of pointers



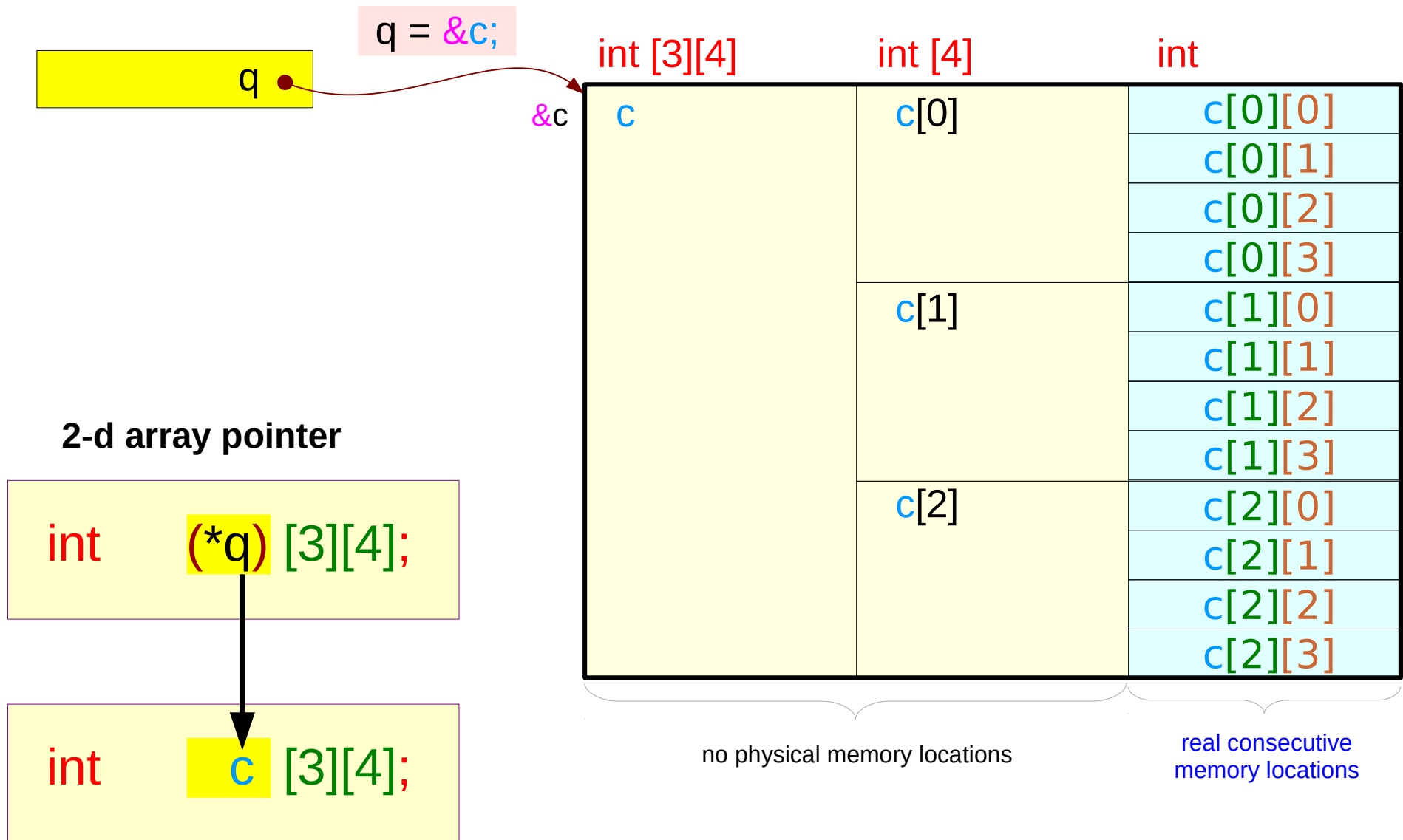
Pointer **q** to a **1-d** array **a**



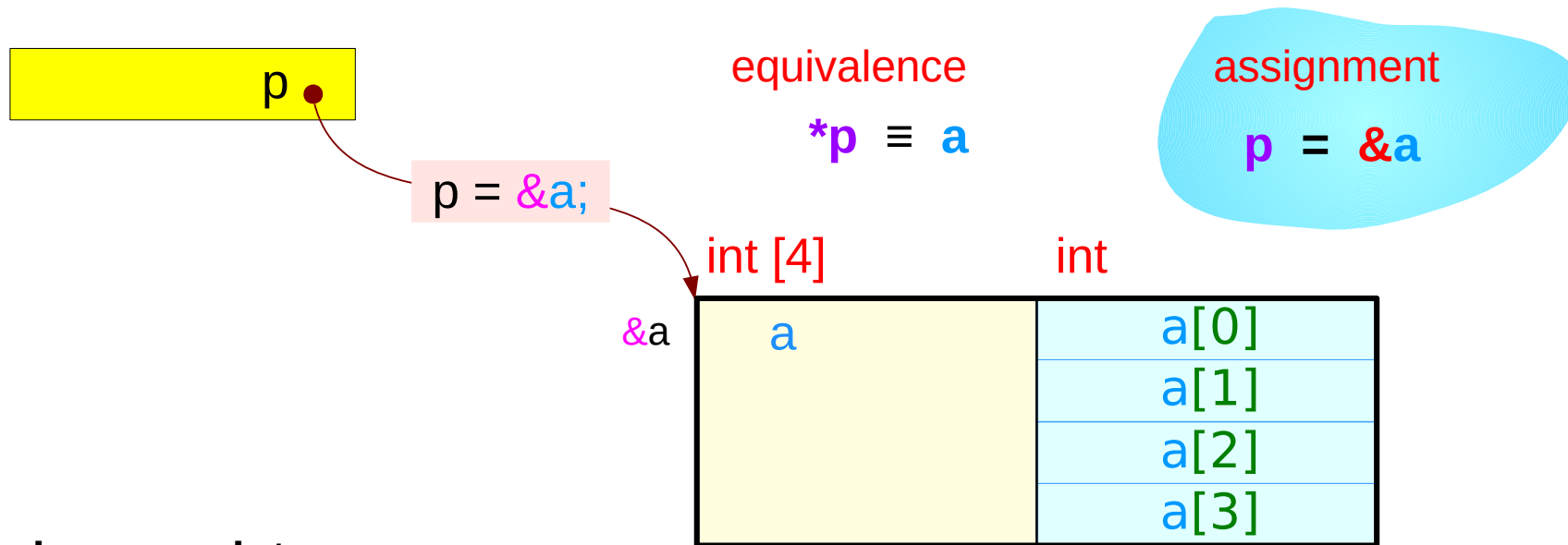
1-d array pointer



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



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



1-d array pointer

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

1-d array

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

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



```
q = c;
```

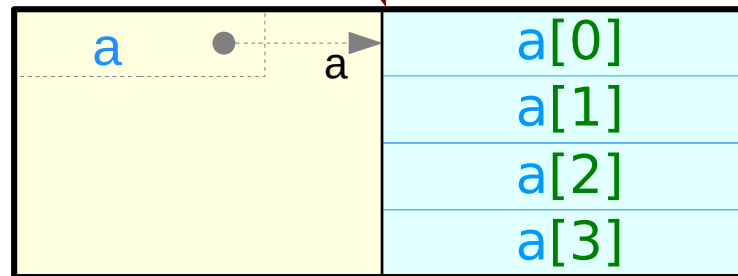
equivalence

$$*q \equiv a[0]$$

assignment

$$q = a$$

int (*) int



0-d array pointer

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

1-d array

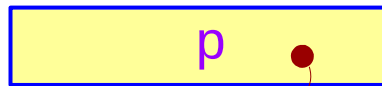
```
int a[4];
```

1-d array access using **p** and **q**

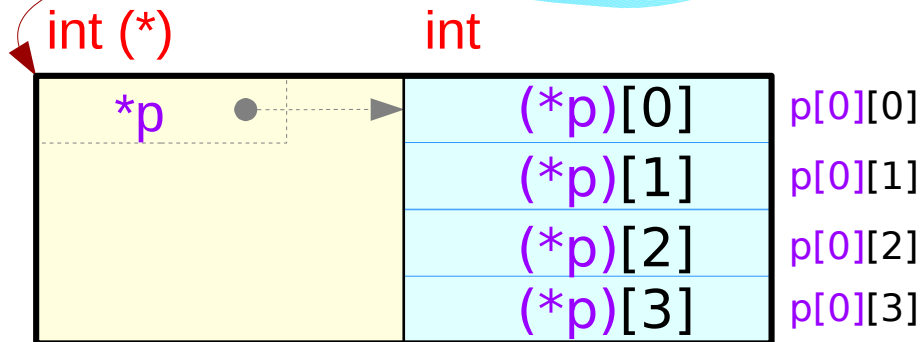
```
int (*p) [4] = &a;
```

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

1-d array pointer



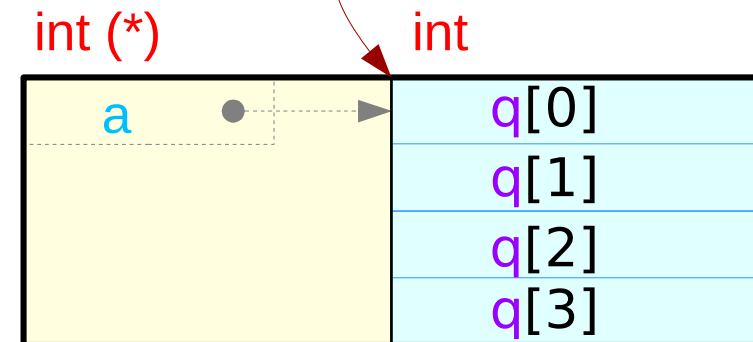
p = &a
assignment



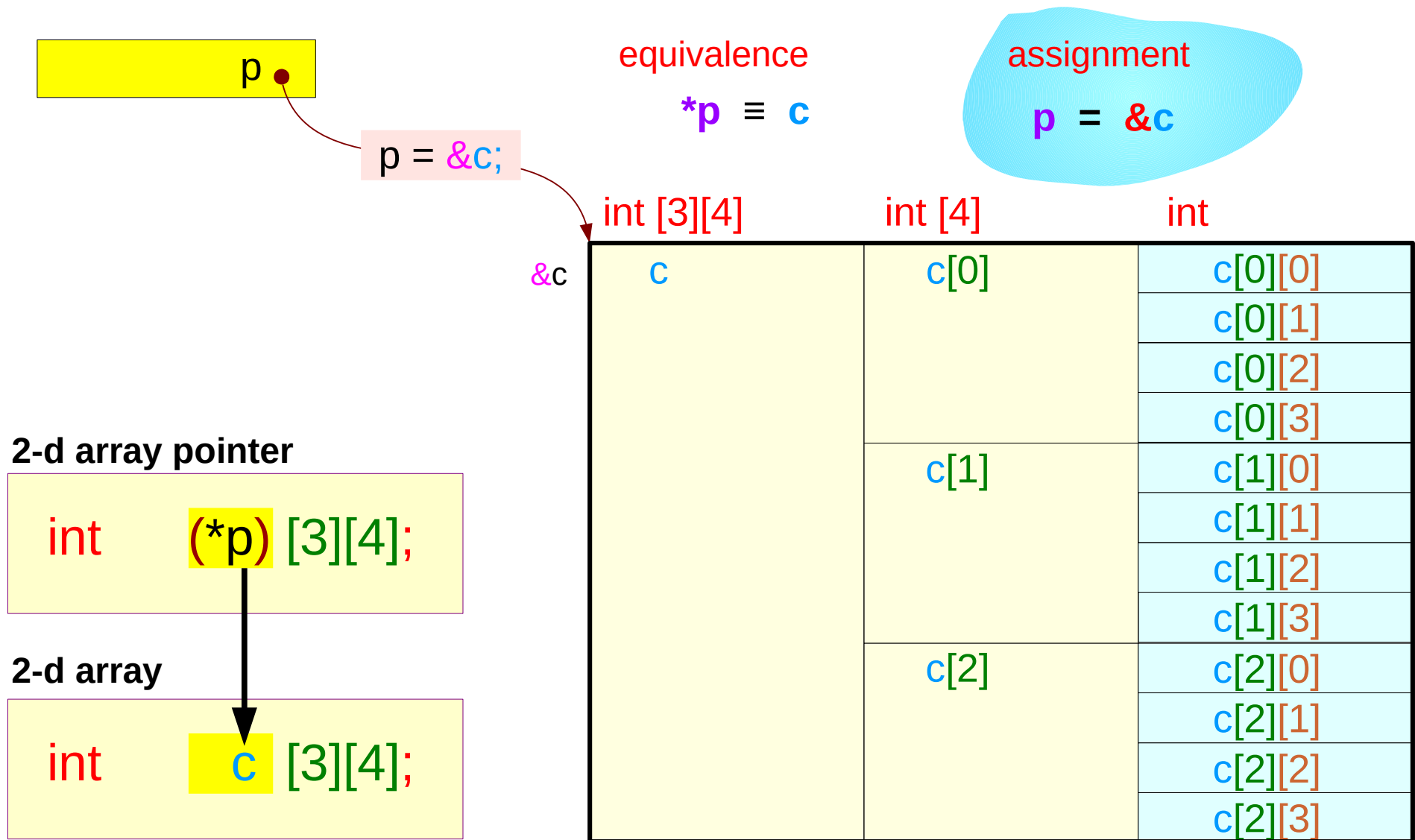
0-d array pointer



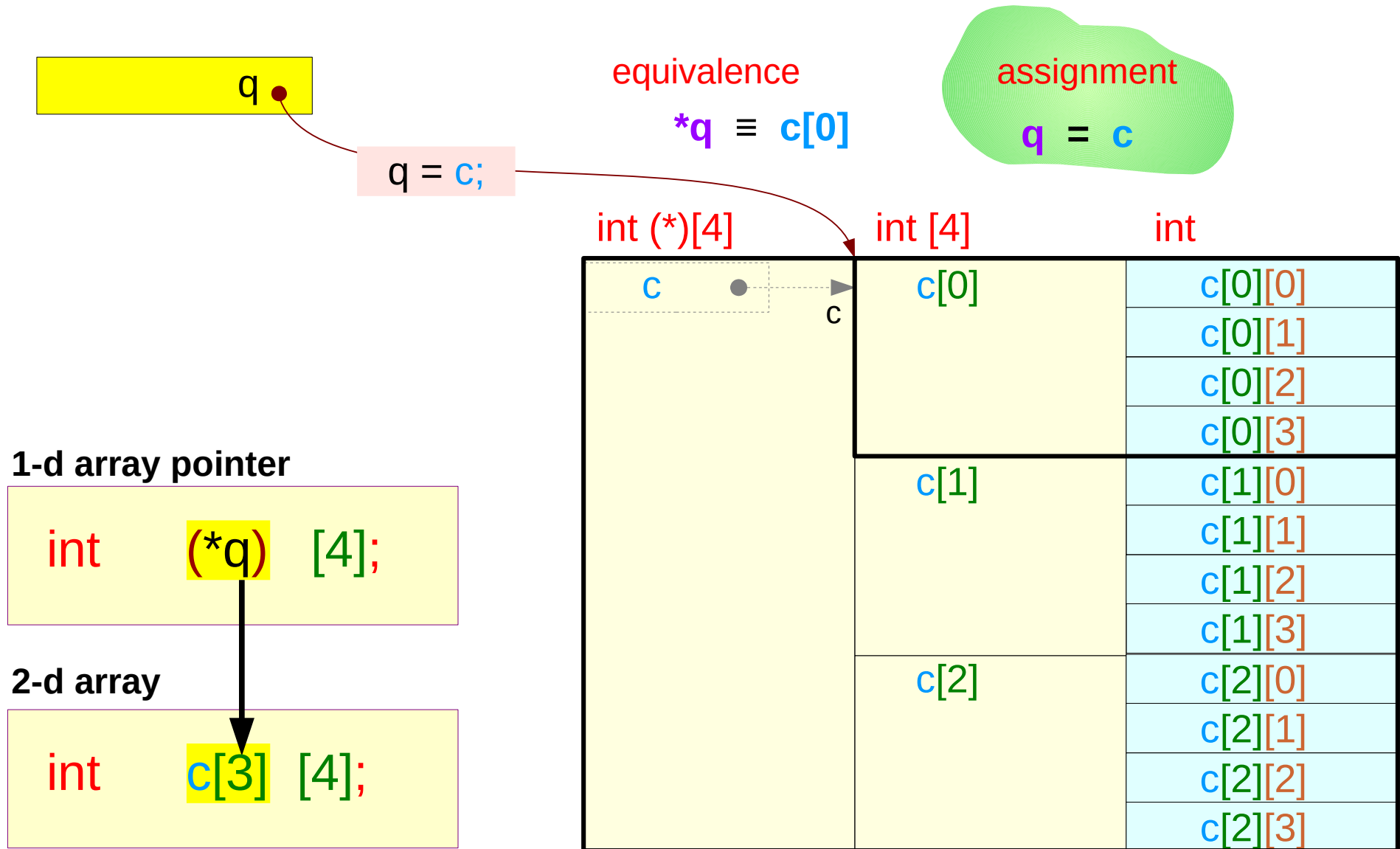
q = a
assignment



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



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



2-d array access using p



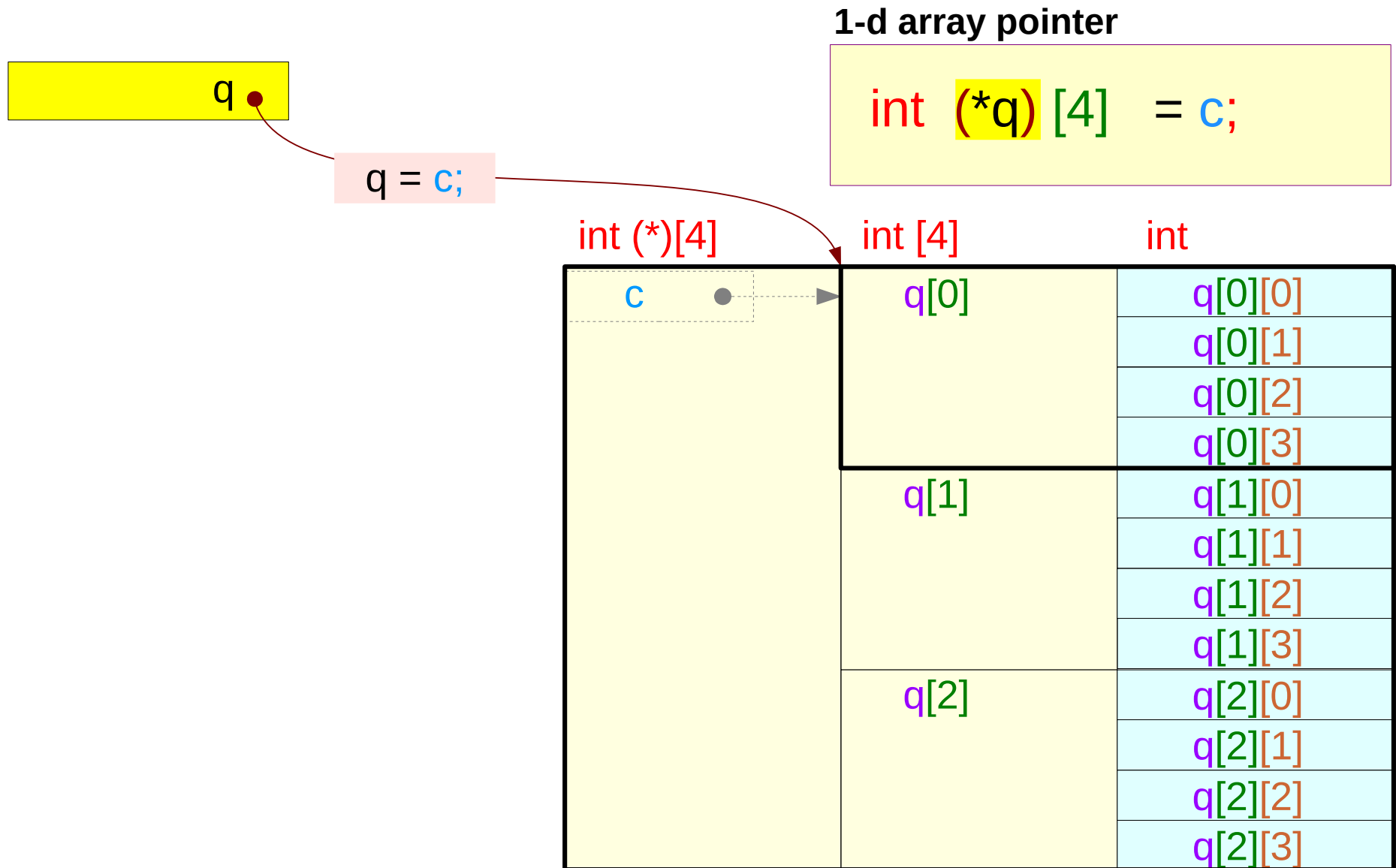
p = &c;

2-d array pointer

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

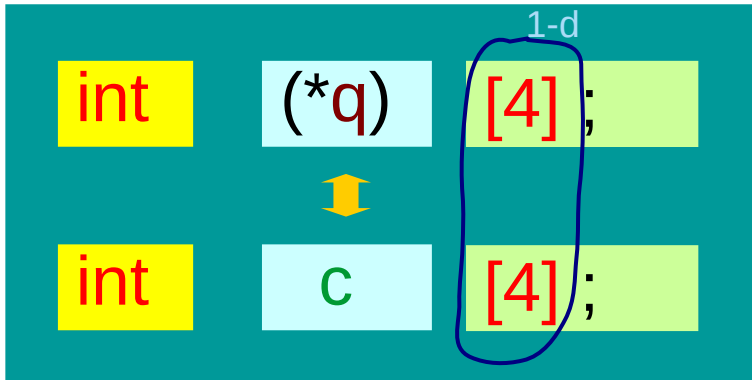
int (*)[4]	int (*)	int
*p	(*p)[0]	(*p)[0][0]
		(*p)[0][1]
		(*p)[0][2]
		(*p)[0][3]
	(*p)[1]	(*p)[1][0]
		(*p)[1][1]
		(*p)[1][2]
		(*p)[1][3]
	(*p)[2]	(*p)[2][0]
		(*p)[2][1]
		(*p)[2][2]
		(*p)[2][3]

2-d array access using q



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

1-d array pointer



`int (*) [4]`

equivalence

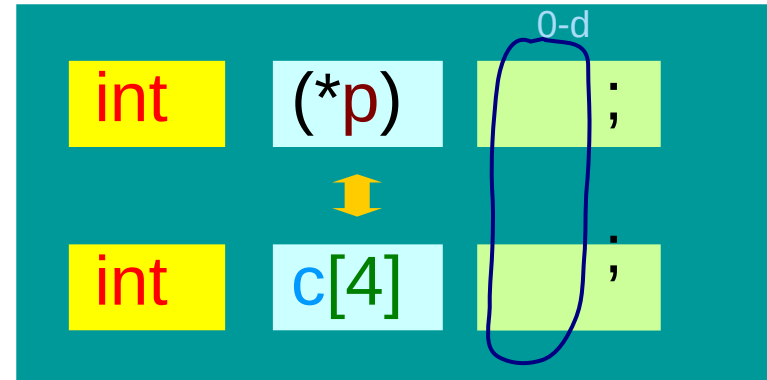
`*q ≡ c;`

assignment

`q = &c;`

`(*q)[i] ≡ q[0][i] ≡ c[i]`

0-d array pointer : int pointer



`int (*)`

equivalence

`*p ≡ *c;`

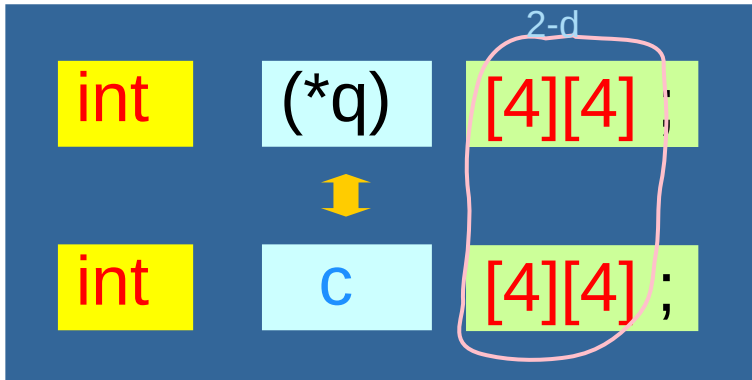
assignment

`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

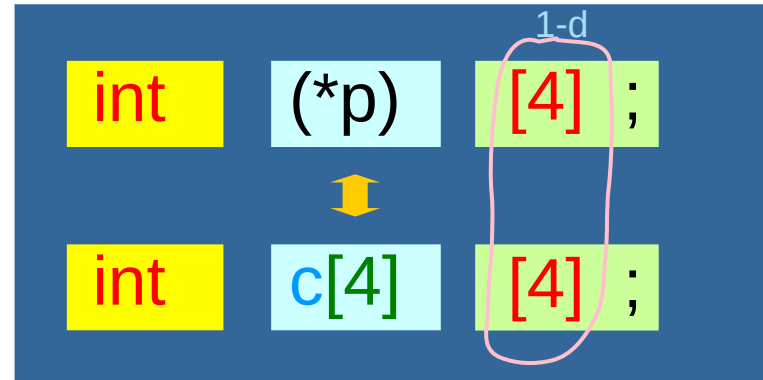
`*q ≡ c;`

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**

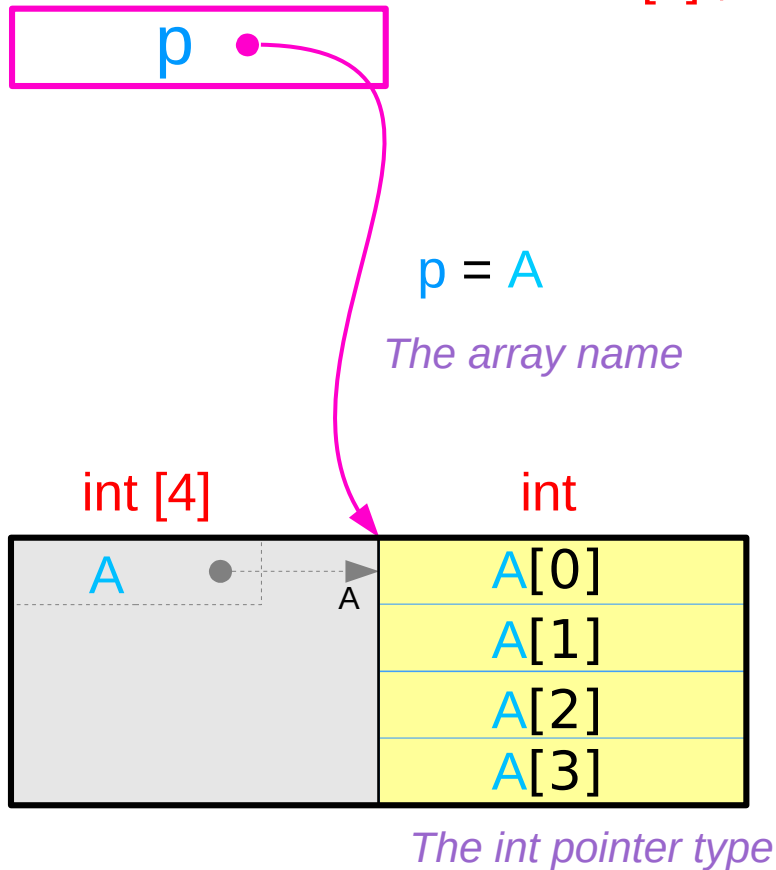
int (*) [4] **→** **int [4]** **→** **int**
int
int
int
int

Integer pointer **p** vs. array pointer **q**

```
int *p ;
```

```
p = A ;
```

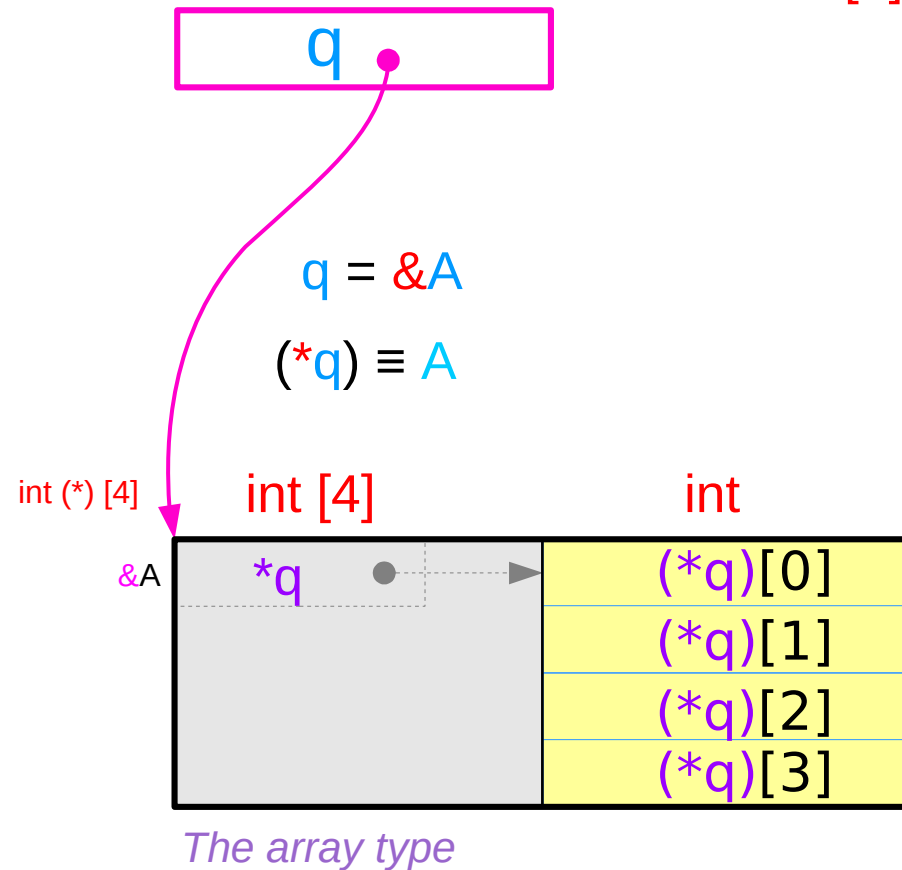
```
int A[4] ;
```



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

```
q = &A ;
```

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



Must point to an array type (array name)

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

```
q = A ;
```

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



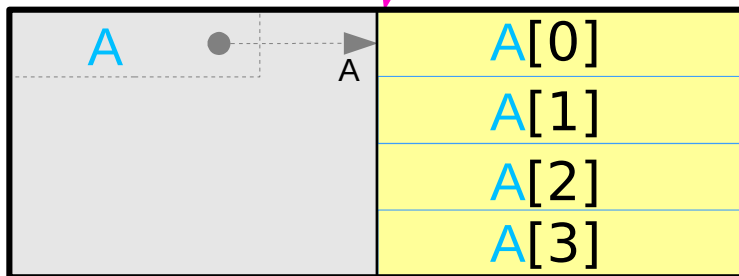
Warning!

$q = A$

The array name

int [4]

int



The array type

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

```
q = &A ;
```

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



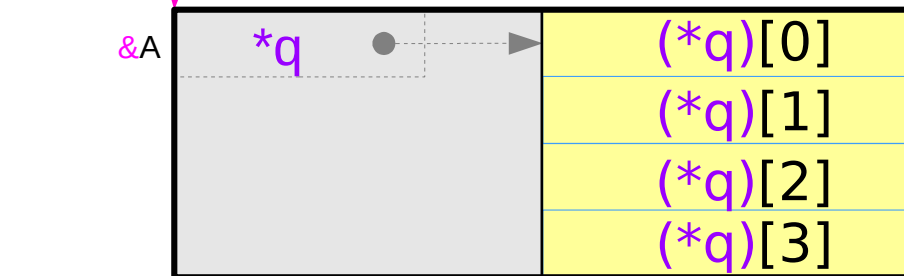
$q = \&A$

$(*q) \equiv A$

int (*) [4]

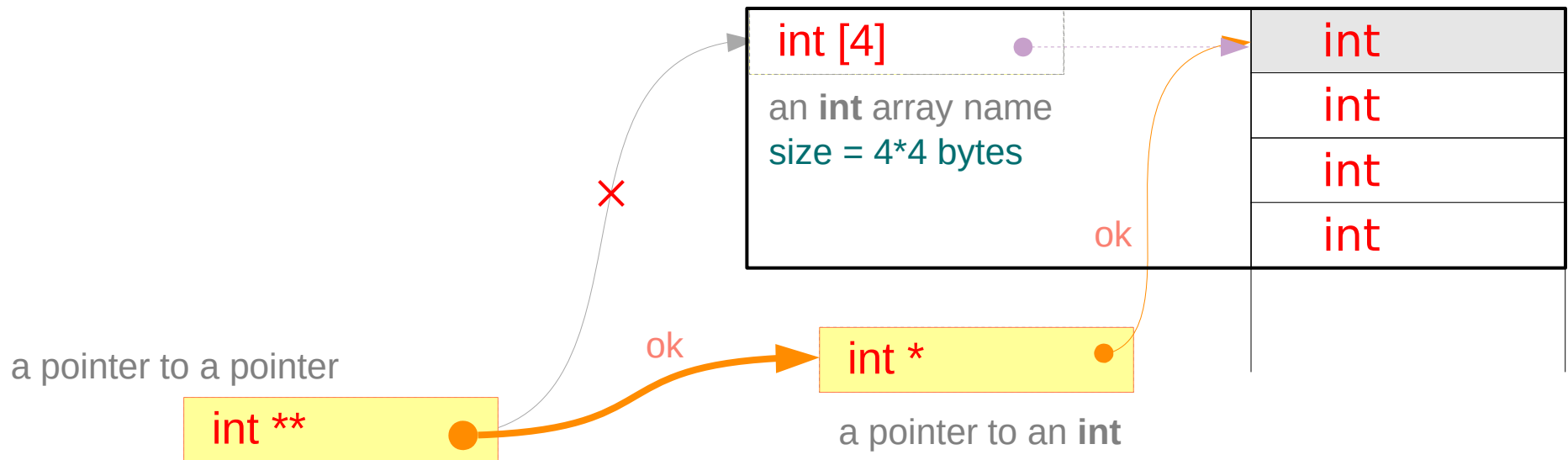
int [4]

int



The array type

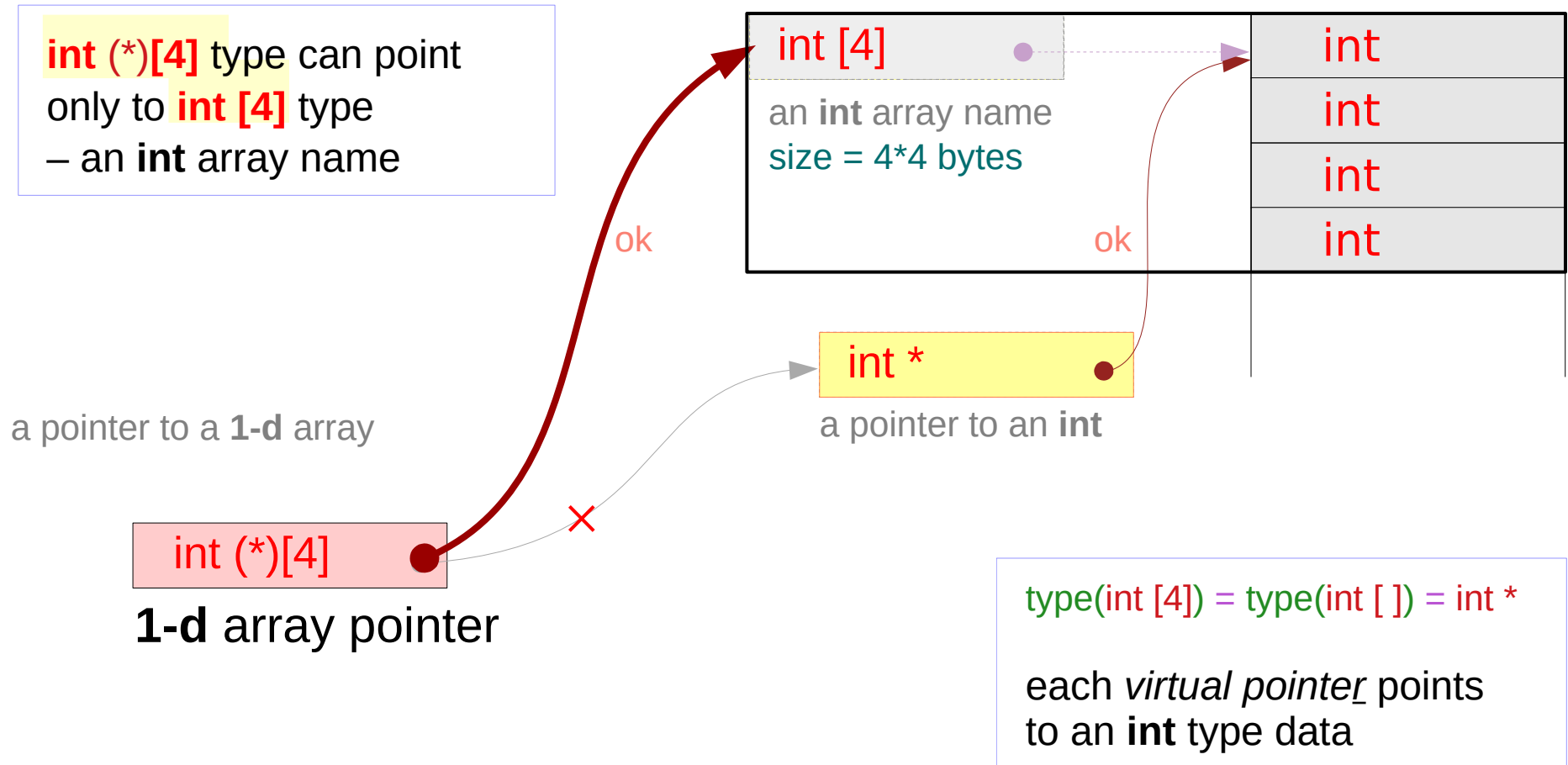
Double integer pointer type – `int **`



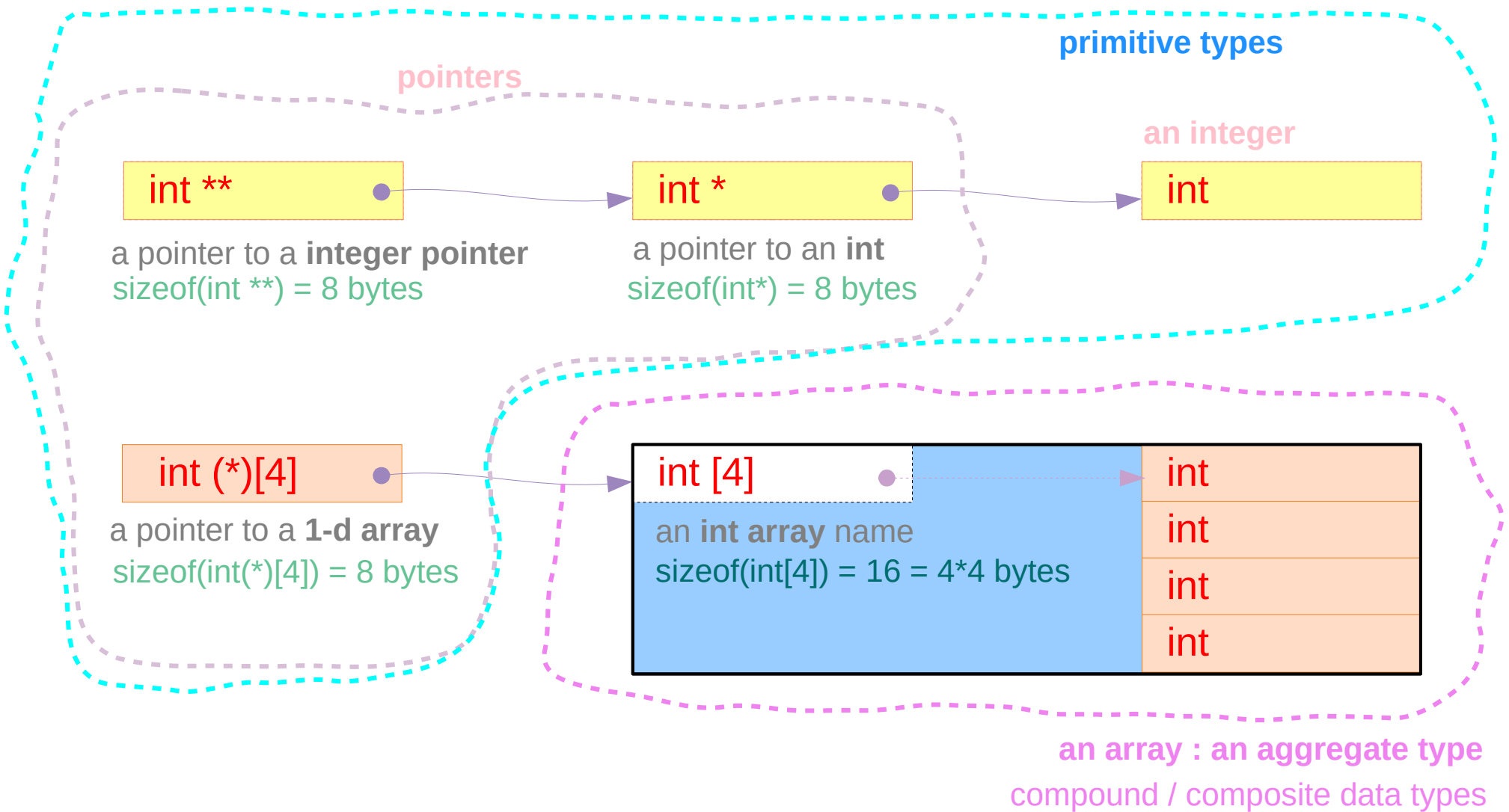
`int **` type can point only to `int *` type – an `int` array name (X)

`type(int [4]) = type(int []) = int *`
each *virtual pointer* points to an `int` type data

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



Types of integer pointers

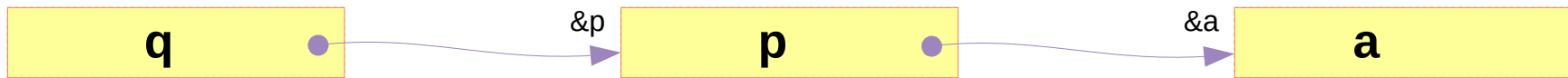


Variable declaration of integer pointers

`int *q = &p;`

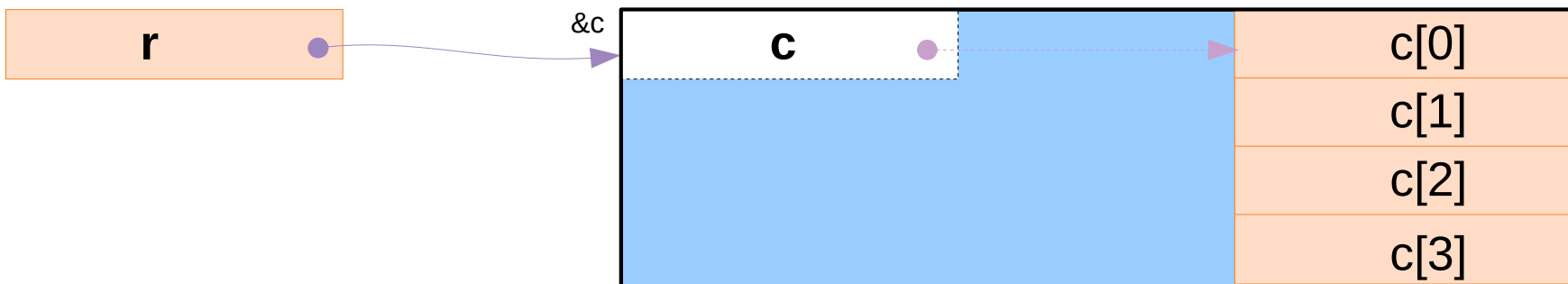
`int *p = &a;`

`int a;`



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

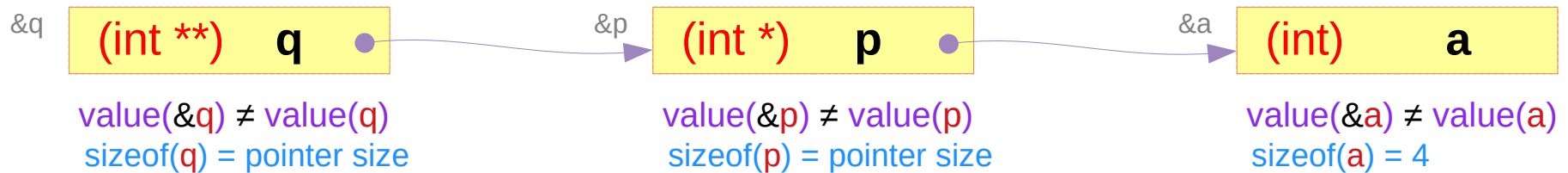
`int c[4];`



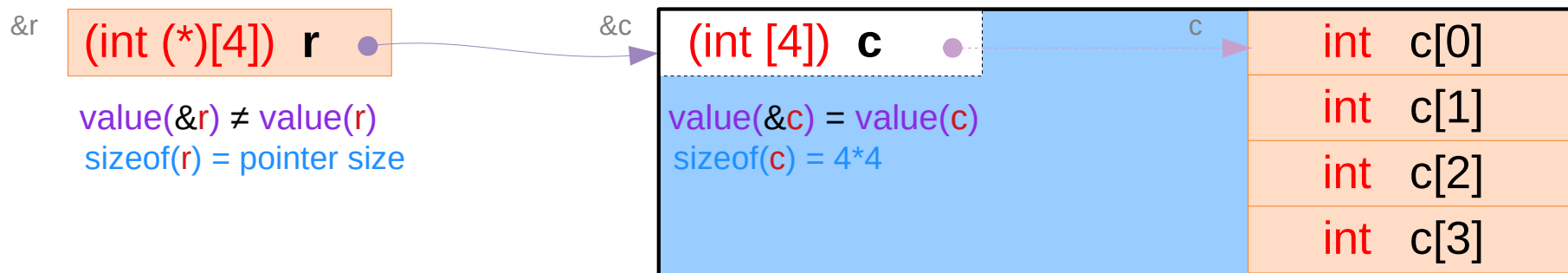
Types and sizes of integer pointers

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

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



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



Sizes of integer pointers

a pointer to an `int`

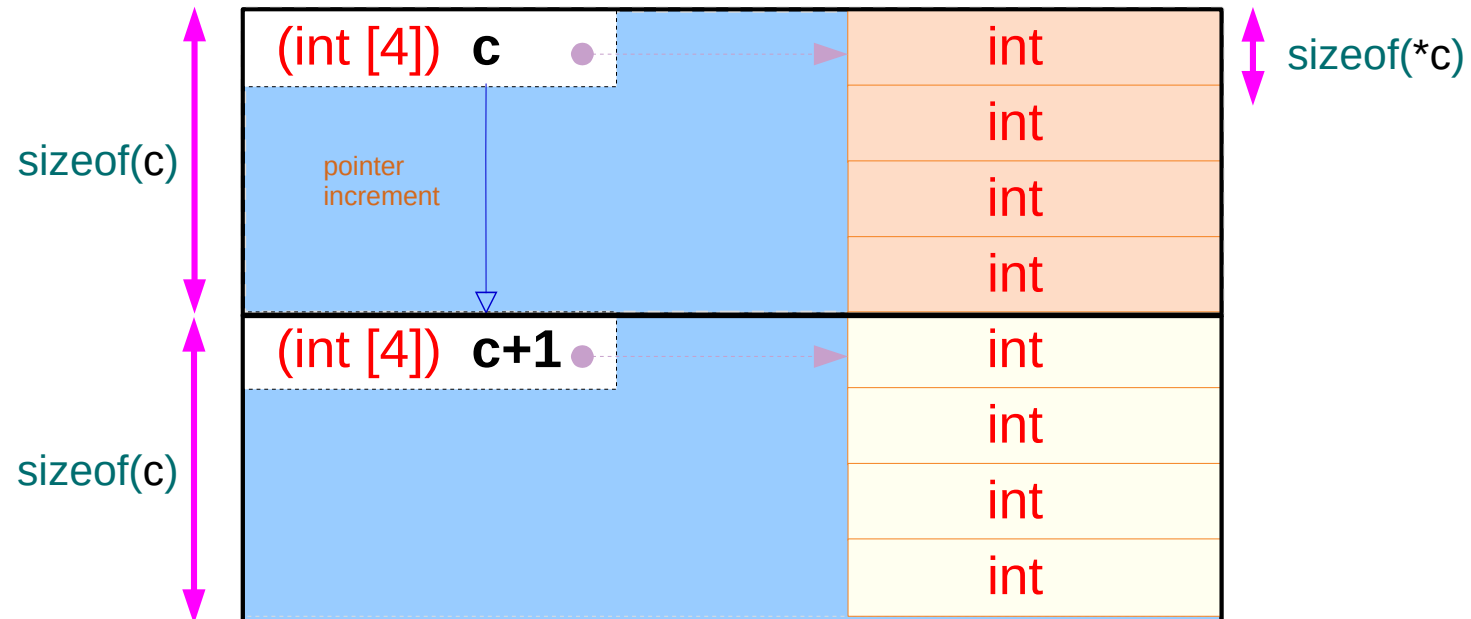
`sizeof(p)` = pointer size
= 8 bytes on 64-bit machine
= 4 bytes on 32-bit machine



an `int` array name

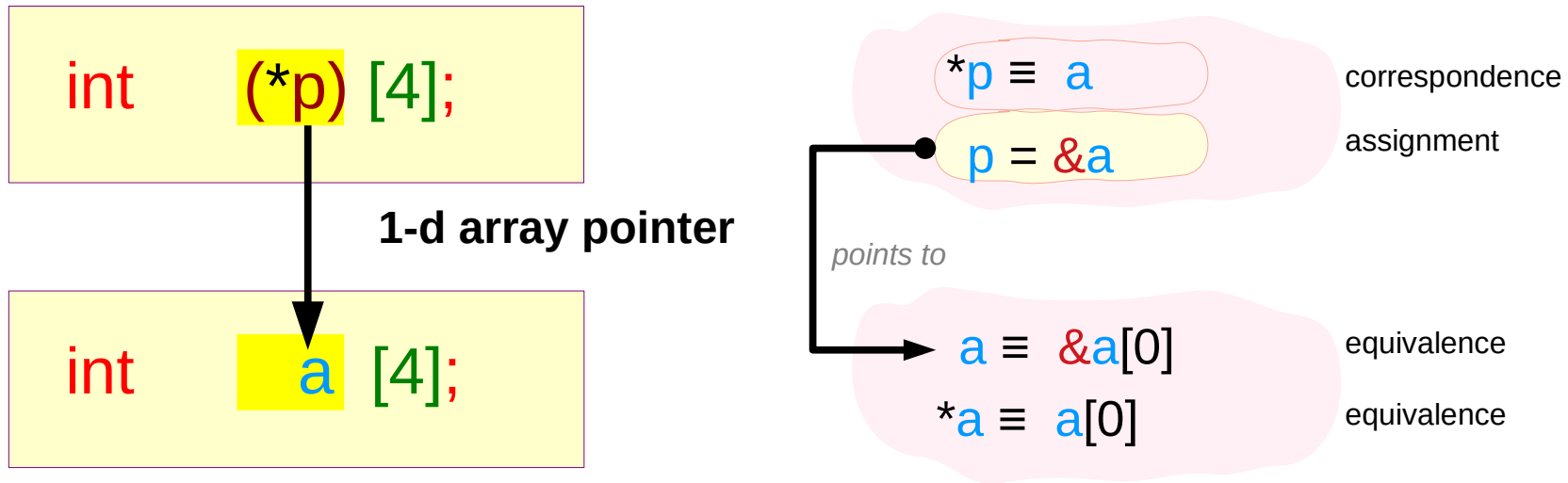
an array :
an aggregate type

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



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

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



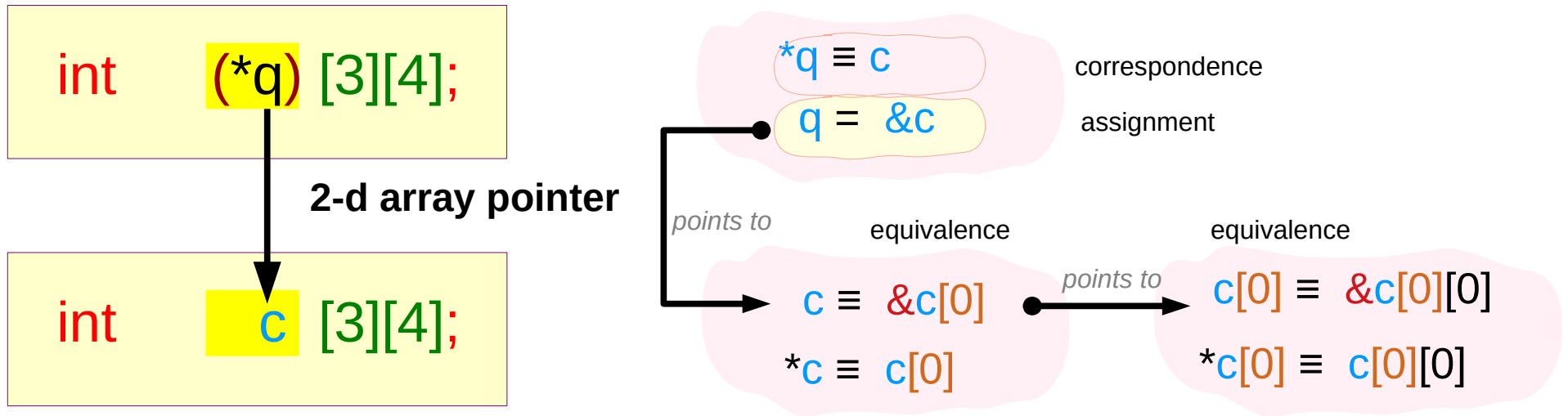
&a and a
print the same address
but have different types

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

&a[0]
int (*)[4] ≠ int [4]

those values are evaluated as addresses

Pointer to a 2-d array – (1) type declarations



`&c, c, c[0]`
 print the same address
 but have different types

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

`&c[0][0]`

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

those values are evaluated as addresses

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

<code>int a [4];</code>	assignment	equivalence
<code>int (*p) [4];</code>	<code>p = &a</code>	<code>a ≡ &a[0]</code>

`int (*) [4]`

`sizeof(p) =`
4 or 8 bytes

size of a pointer

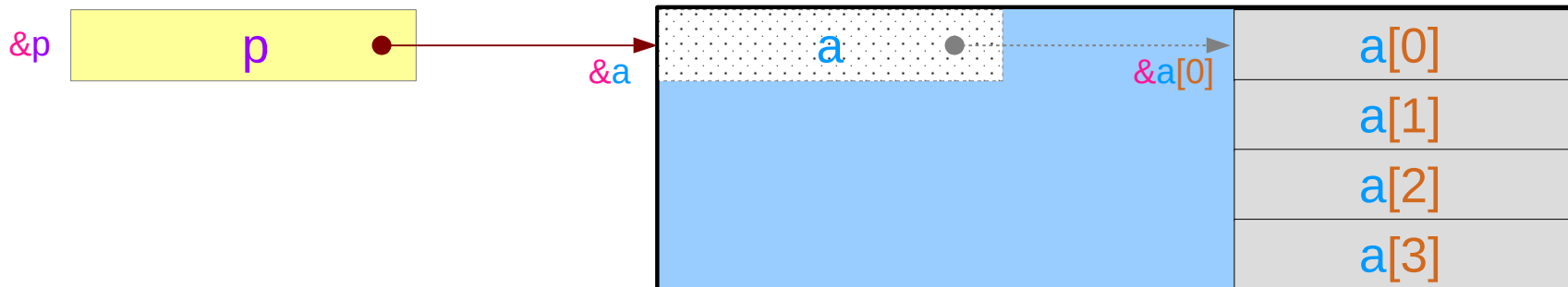
`int [4]` or `int (*)`

`sizeof(a) =`
4*4 bytes

not a real pointer `a`

`int`

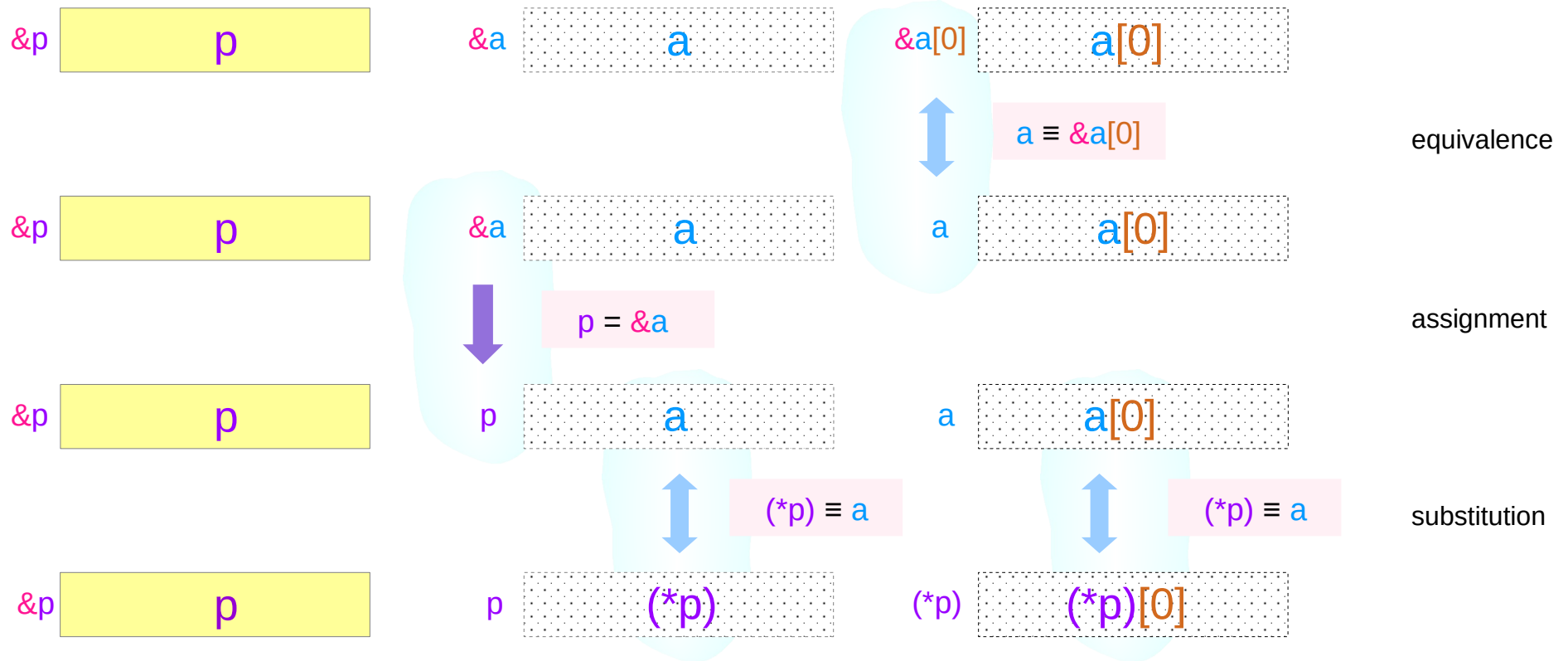
`sizeof(a[0]) =`
4 bytes



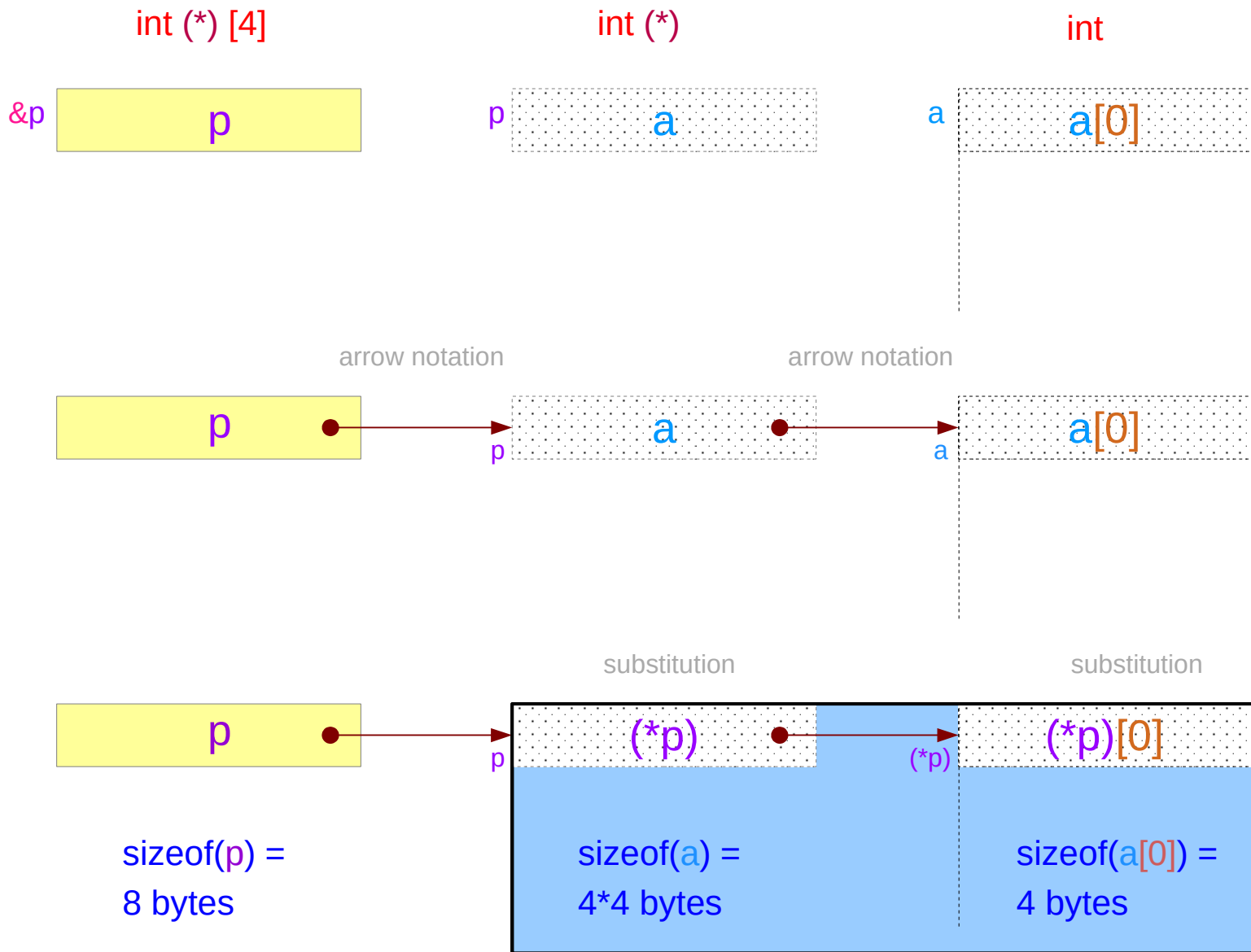
`value(&a) = value(a) = value(&a[0])`

not a real pointer `a`

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



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



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

<code>int c [3][4];</code>	assignment	equivalence	equivalence
<code>int (*q) [3][4];</code>	<code>q = &c</code>	<code>c ≡ &c[0]</code>	<code>c[0] ≡ &c[0][0]</code>

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

`sizeof(q) =`

8 bytes

size of a pointer

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

`sizeof(c) =`

3*4*4 bytes

not a real pointer `c`

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

`sizeof(c[0]) =`

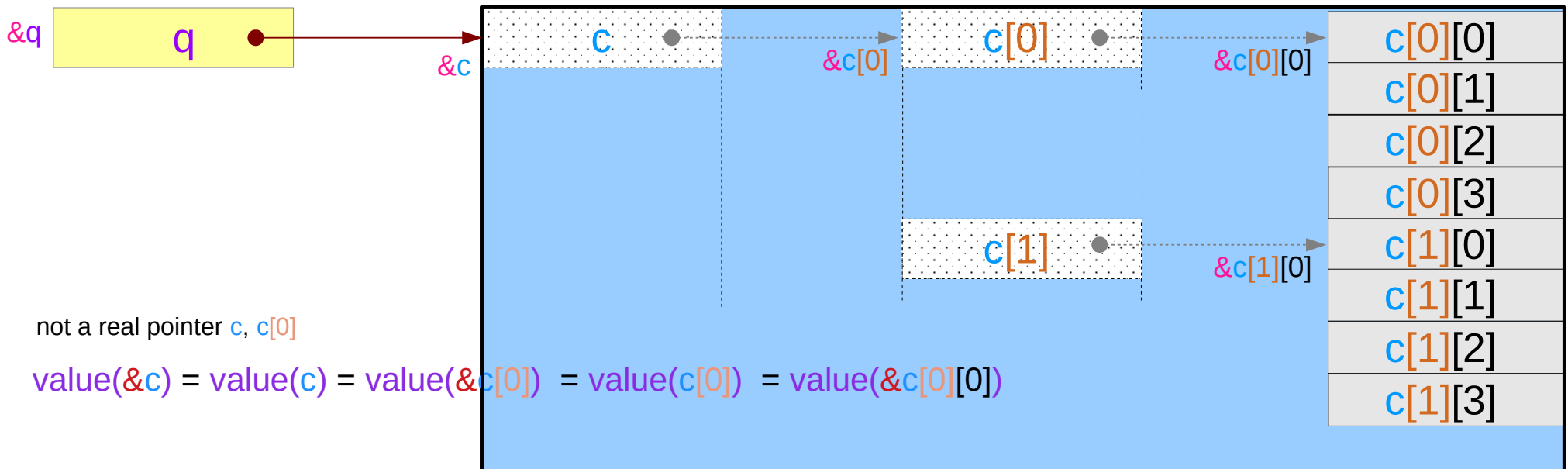
4*4 bytes

not a real pointer `c[0]`

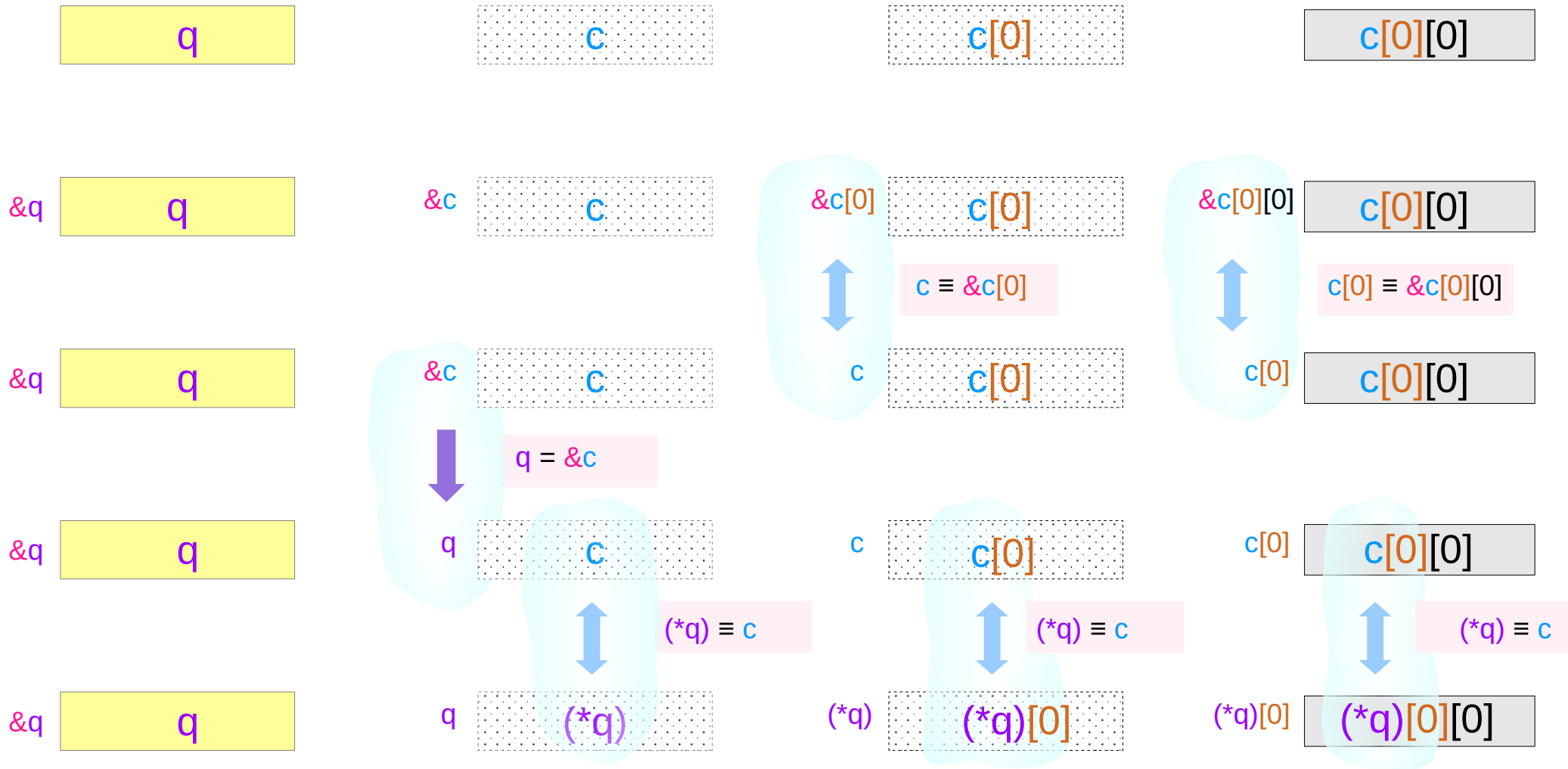
`int`

`sizeof(c[0][0]) =`

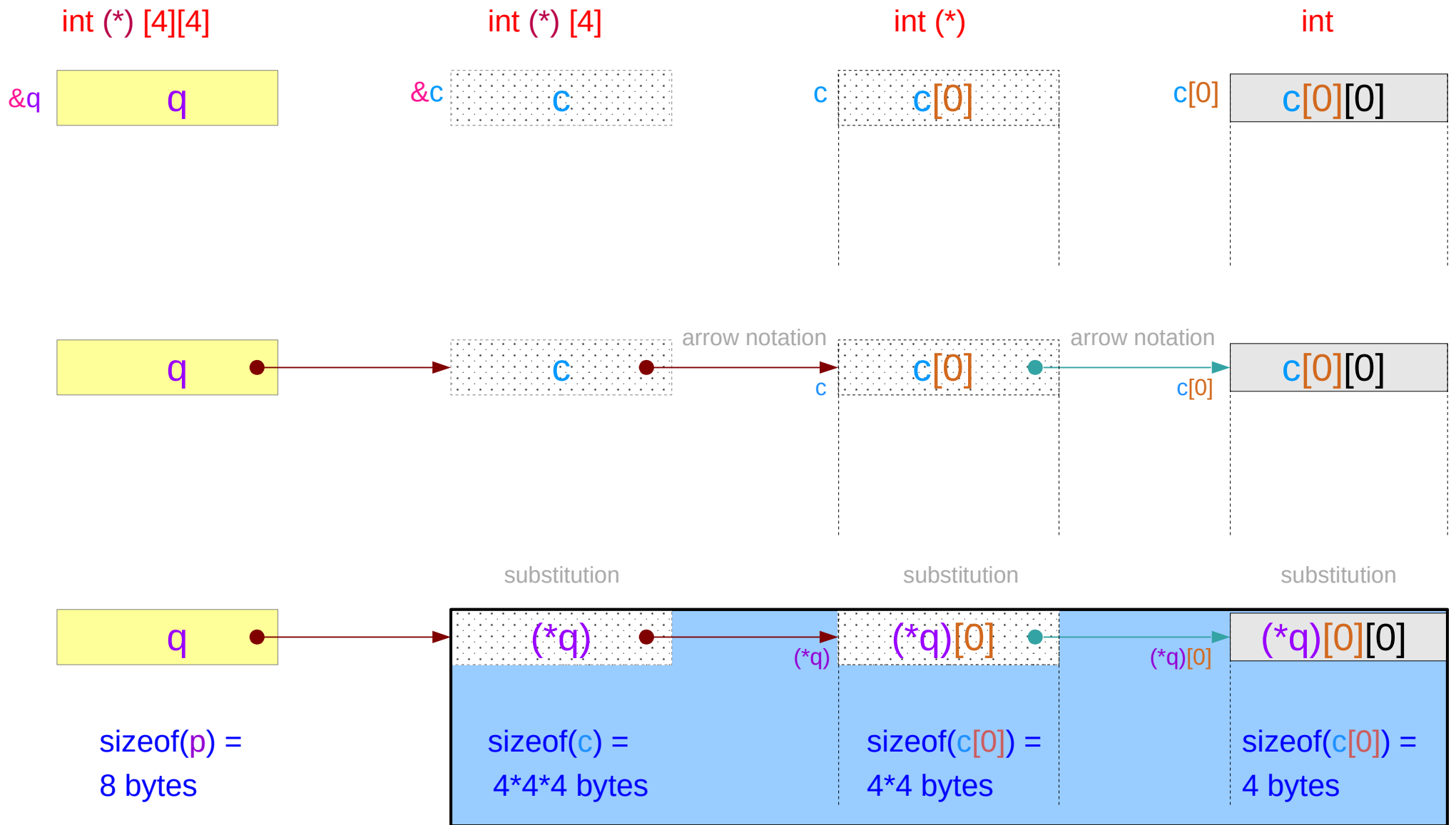
4 bytes



Pointer to a 2-d array – (3) an assignment & equivalences

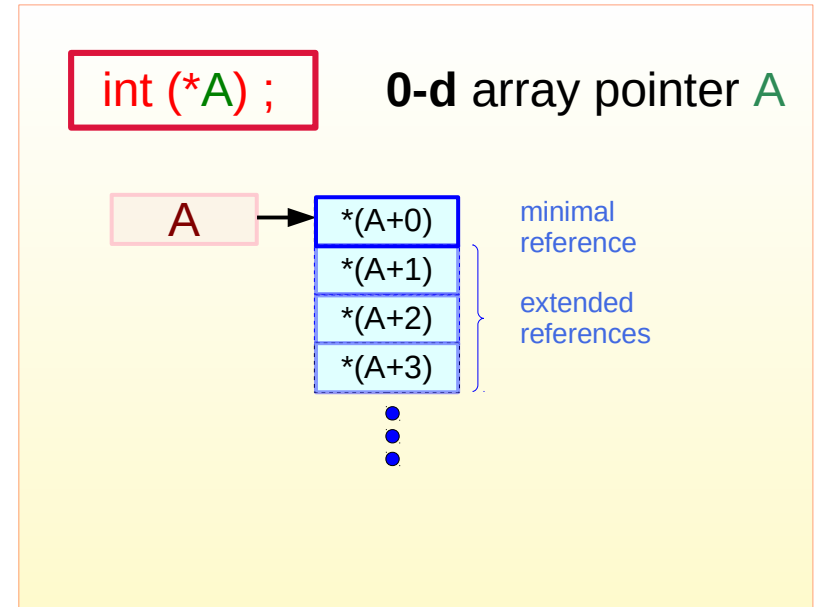
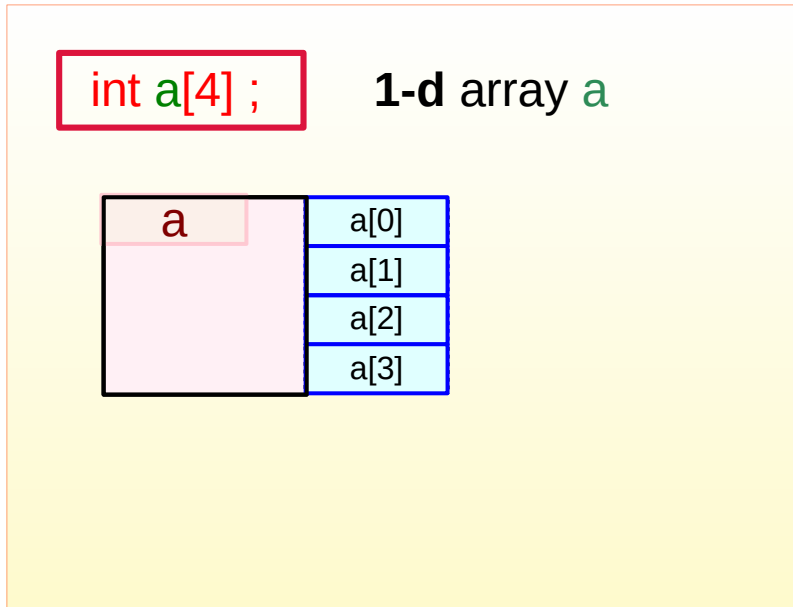


Pointer to a 2-d array – (4) a chain of pointers view

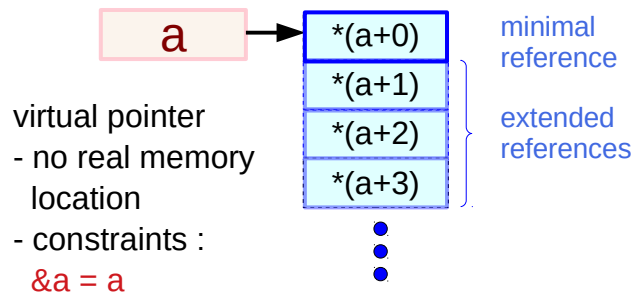


Array pointers in an array

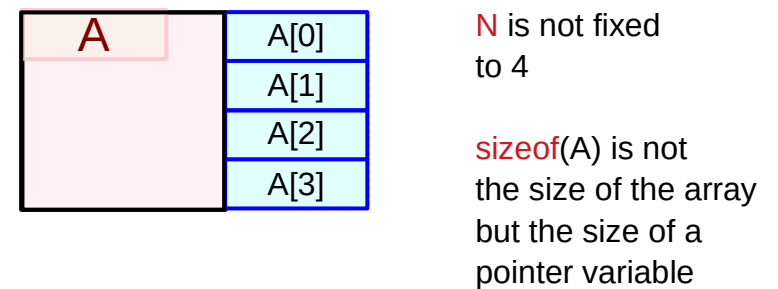
Array **a** vs array pointer **A**



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



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



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

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

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

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

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

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

a is not a real pointer

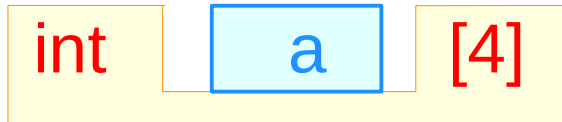
- `sizeof(a)` = an array size
- `a = &a`

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

A is not a real array

- `sizeof(A)` = a pointer size
- `A ≠ &A`

Array and pointer types in a 1-d array



a 1-d array

type : **int [4]**

size : **4 * 4**

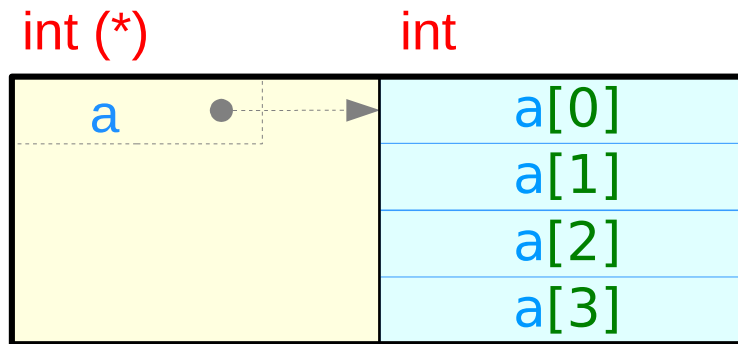
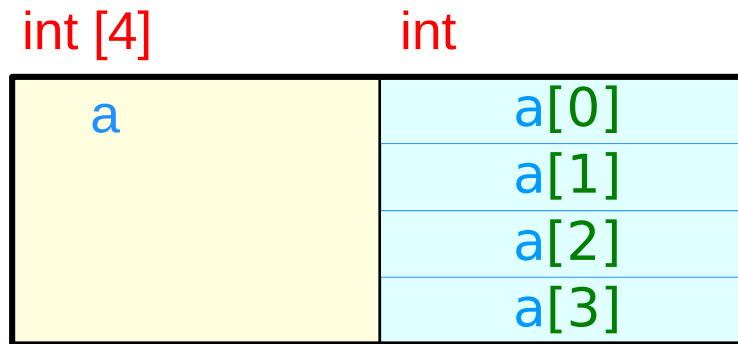


a 0-d array pointer (virtual)

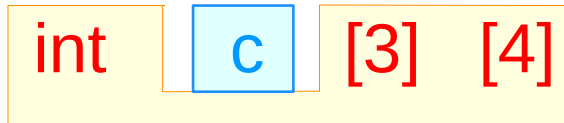
type : **int (*)**

size : **4 * 4**

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



2-d array type



C 2-d array

type : int [3][4]

size : 3 * 4 * 4

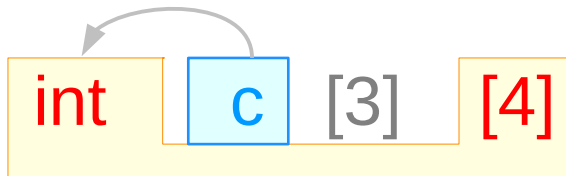
int [3][4]

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

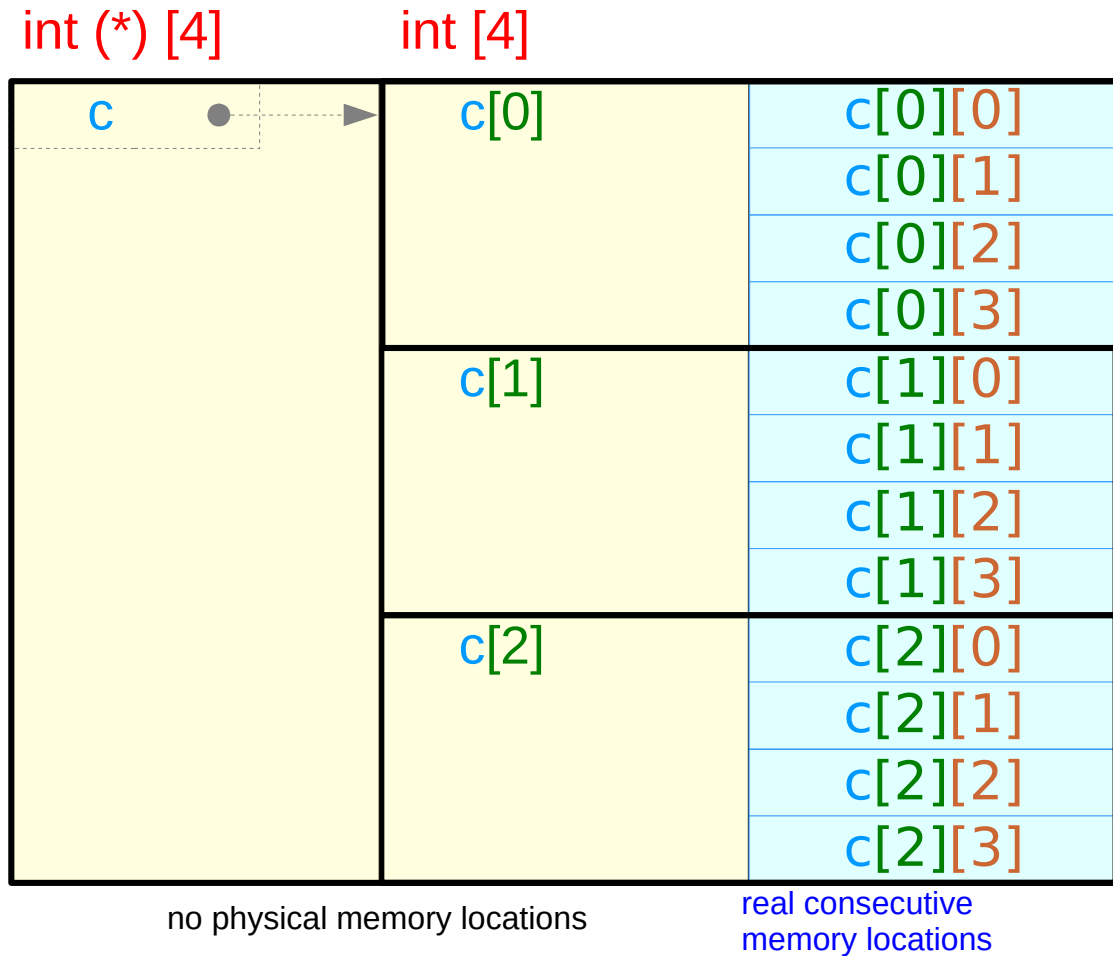


C 1-d array pointer (virtual)

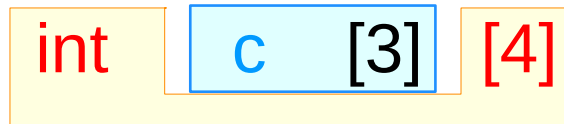
type : `int (*) [4]`

size : `3 * 4 * 4`

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



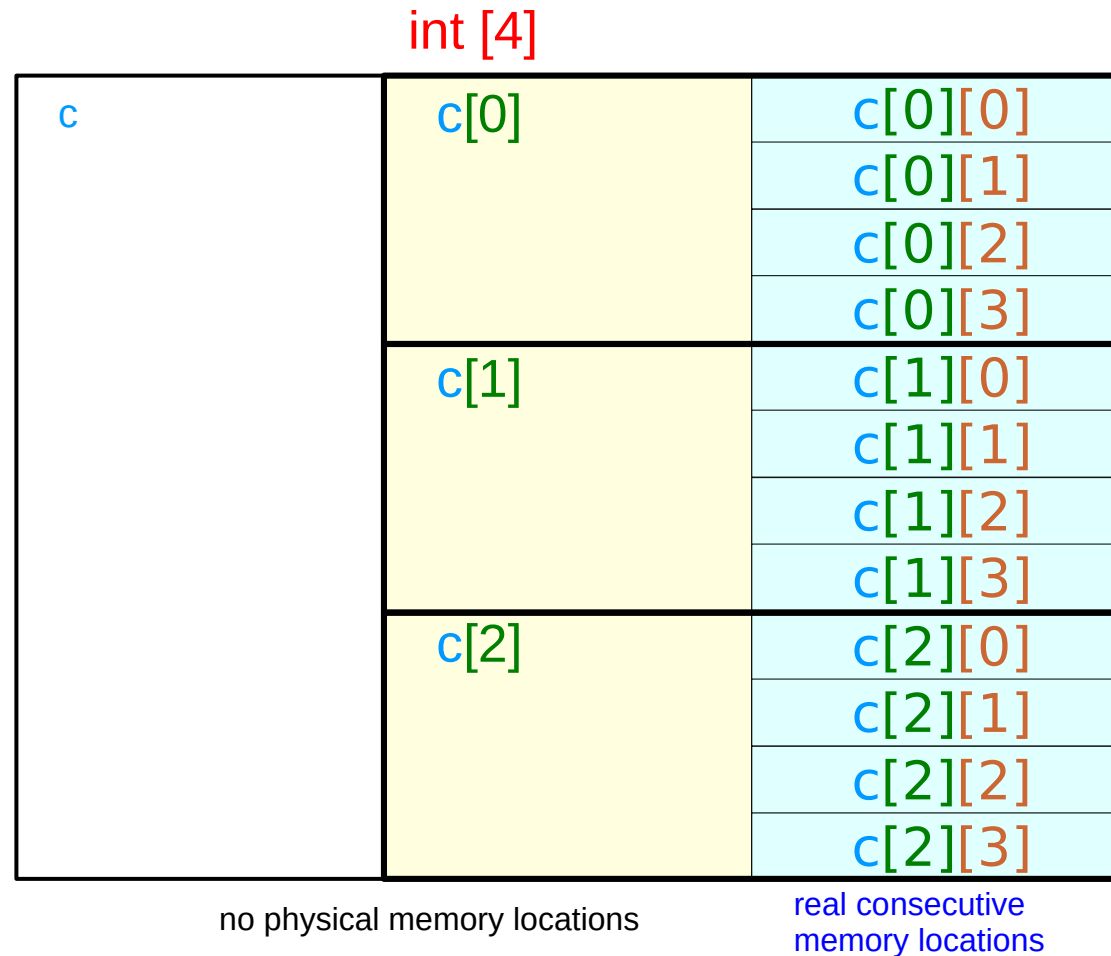
1-d array type



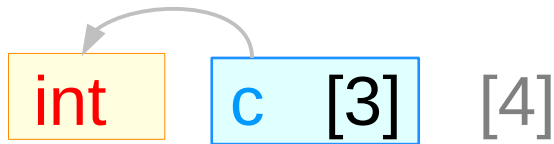
c[i] 1-d array

type : int [4]

size : 4 * 4



0-d array pointer type

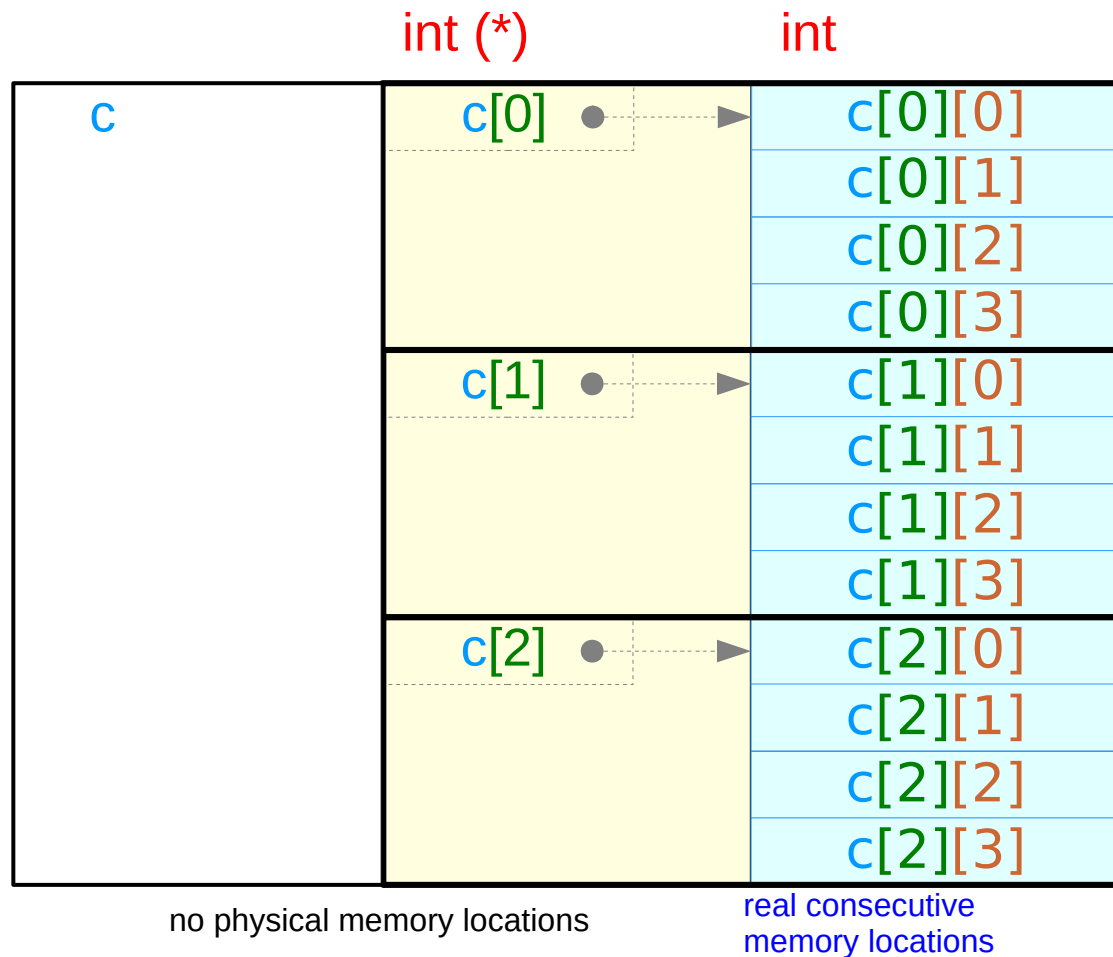


c[i] 0-d array pointer (virtual)

type : **int (*)**

size : **4 * 4**

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

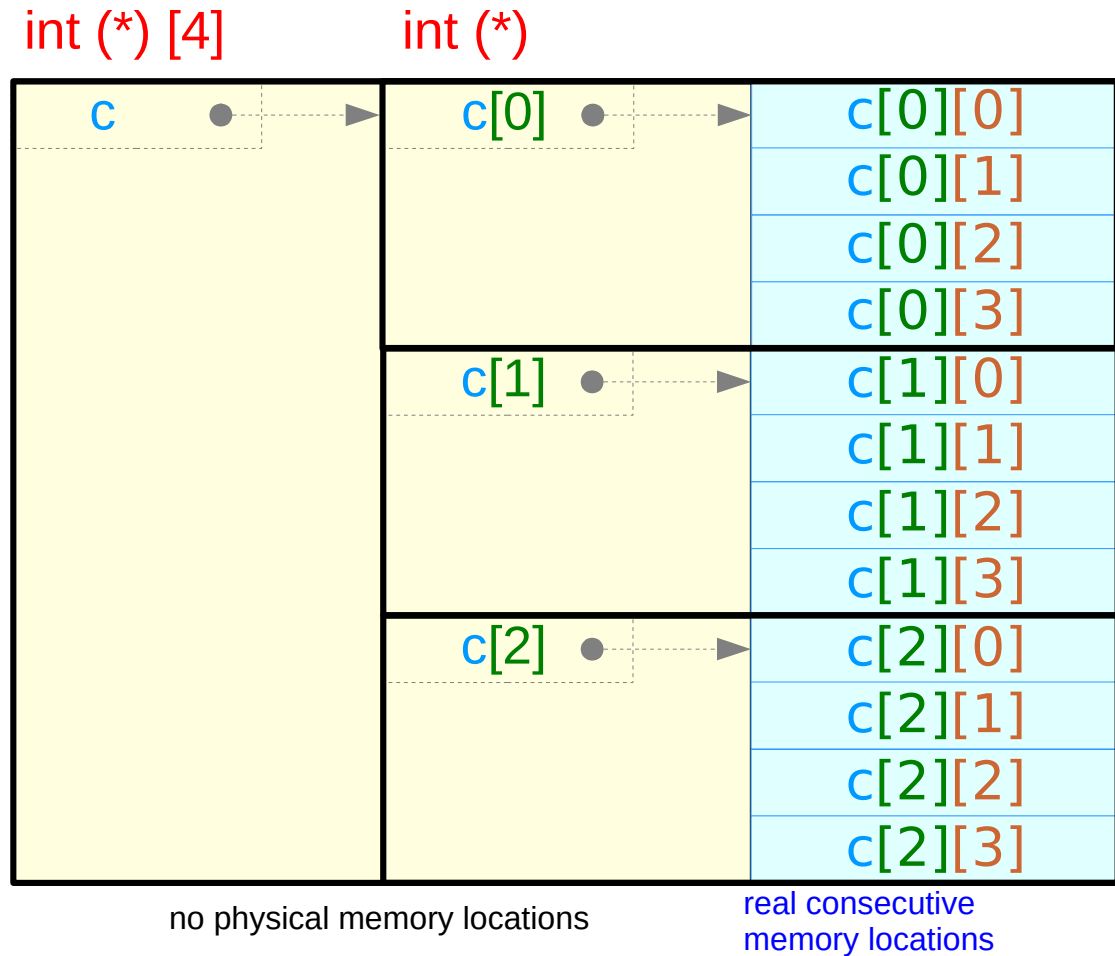


Values of virtual array pointer in a 2-d array

$c = c[0] = \&c[0][0]$

$c[1] = \&c[1][0]$

$c[2] = \&c[2][0]$



Types in a 2-d array

int c [3] [4]

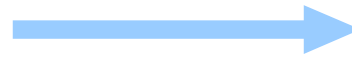
C 2-d array

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)

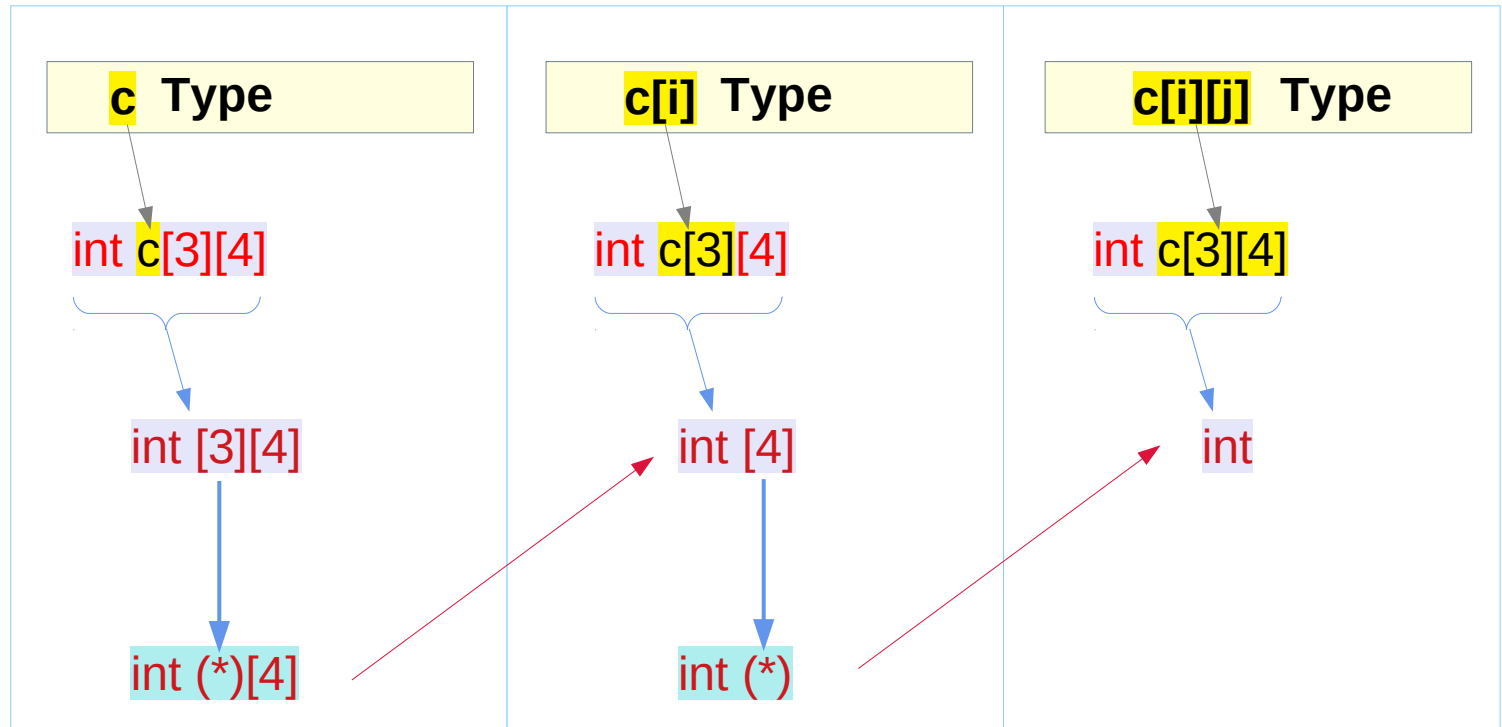
type : int (*)

size : 4 * 4

value : &c[i][0]

Subarray types in a 2-d array

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



Dual Types

Multi-dimensional array types

```
int a ;
```

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

```
int c [4][5];
```

```
int d [4][5][6];
```

```
a int → int
```

```
b int [4] → int (*) or int *
```

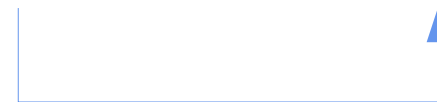
```
c int [4][5] → int (*) [5]
```

```
d int [4][5][6] → int (*) [5][6]
```

array name
(variable)

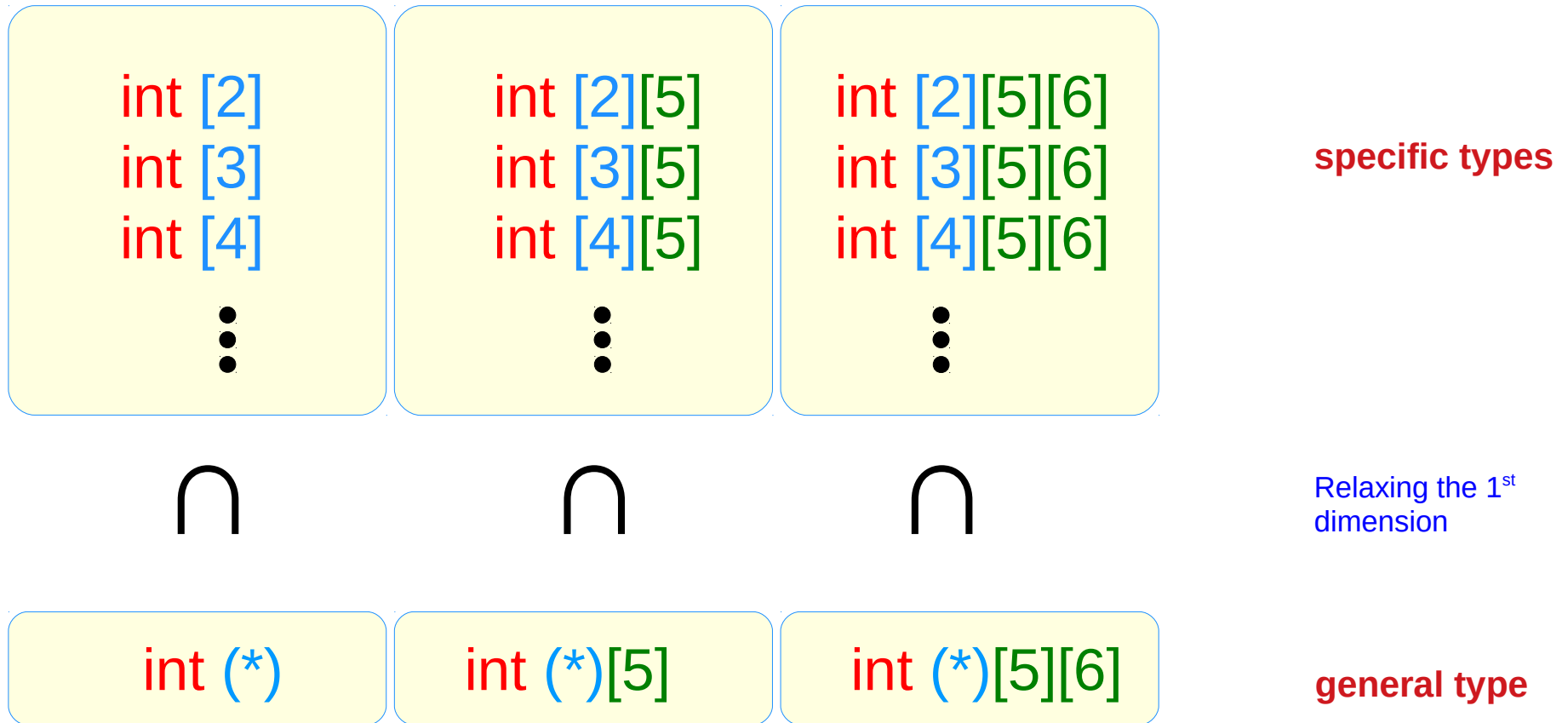
array types
specific types

array pointer types
general type



Relaxing the 1st dimension

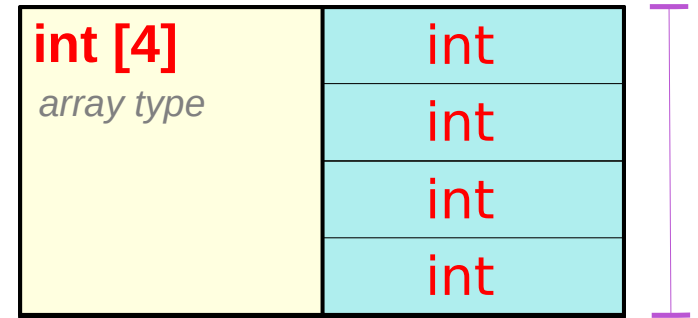
Super types and sub types



Sizes of arrays and array pointers

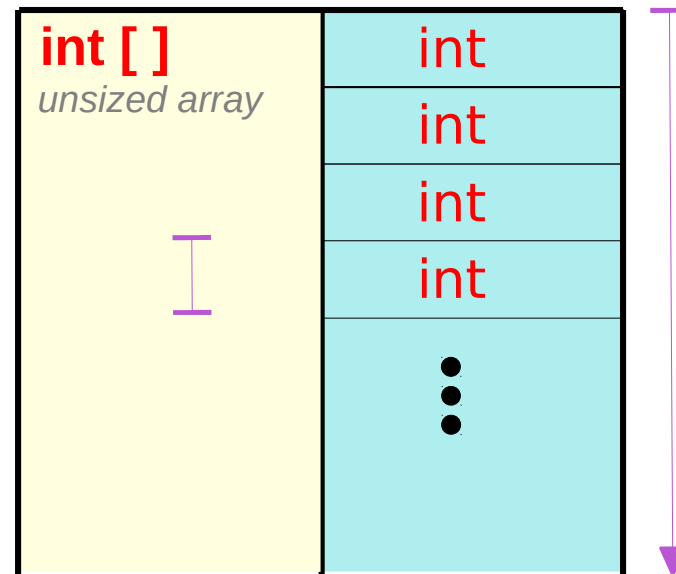
Sizes of `int [4]`, `int []`, `int *` types

$\text{sizeof}(\text{int } [4]) = \text{sizeof}(\text{int}) * 4 = 16$



undefined size

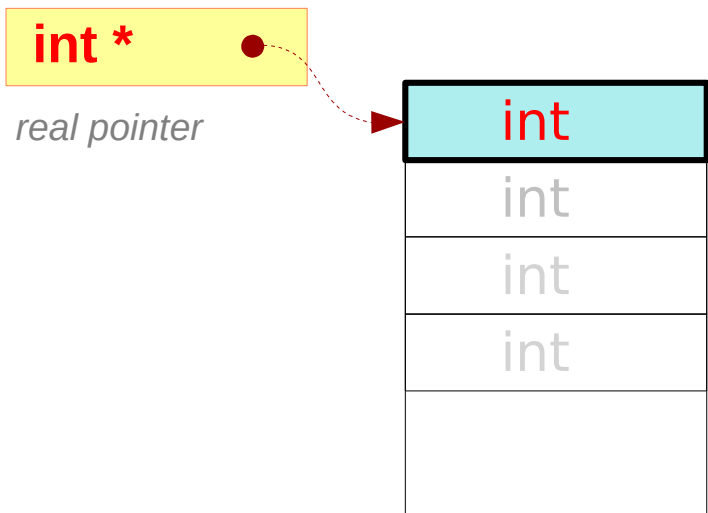
~~`sizeof(int [])`~~



an incomplete type

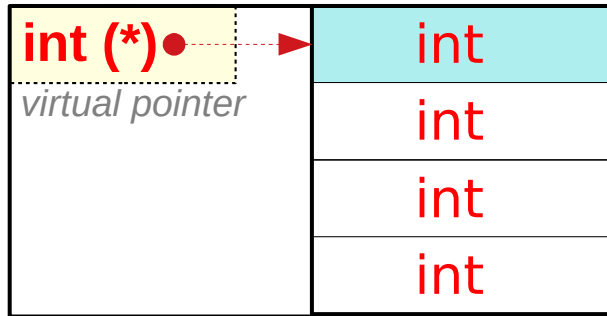
when used to pass arrays to a function

$\text{sizeof}(\text{int } *) = 4 \text{ or } 8$



Sizes of `int [4]` and `int (*)` types

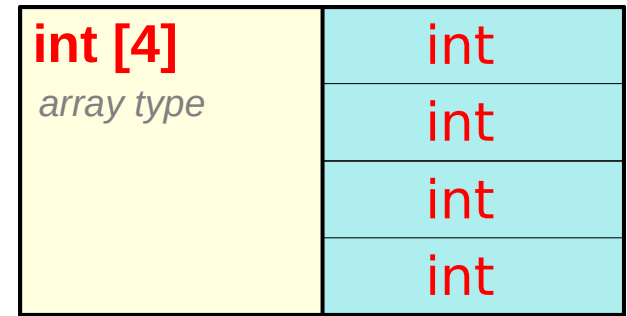
`sizeof(int (*)) = 4 or 8`



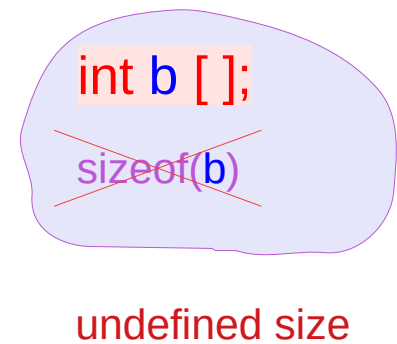
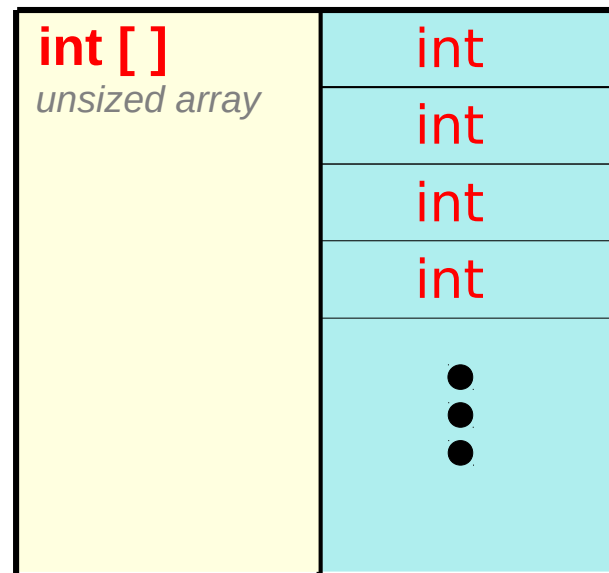
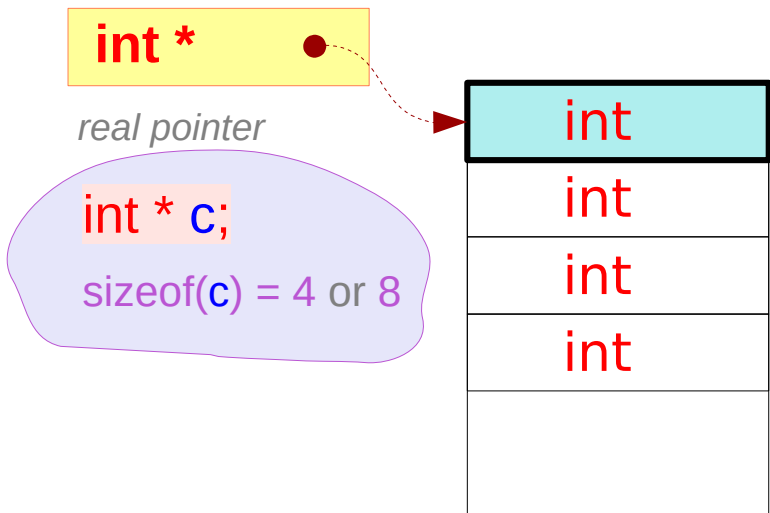
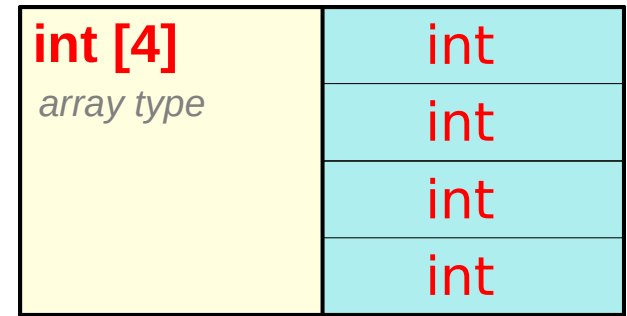
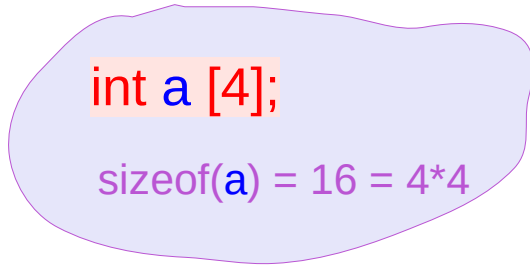
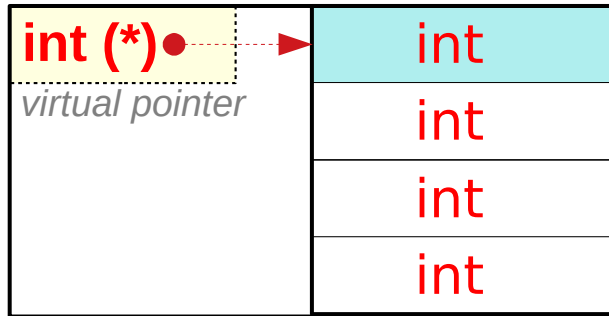
Relaxing the 1st dimension



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



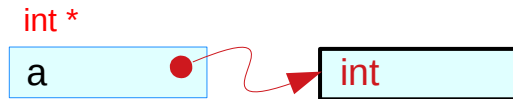
Sizes of variables of `int [4]`, `int []`, `int *` types



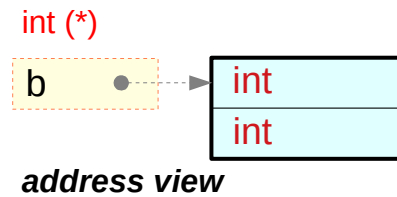
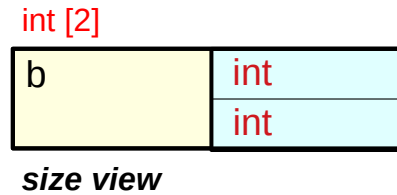
an incomplete type
when used to pass arrays to a function

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

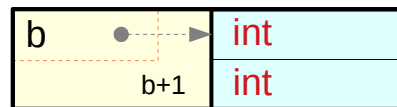
`int *a;`



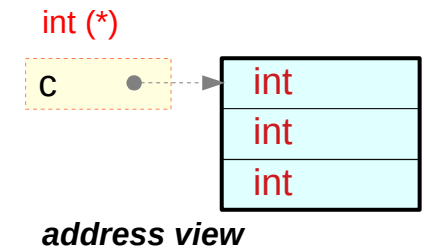
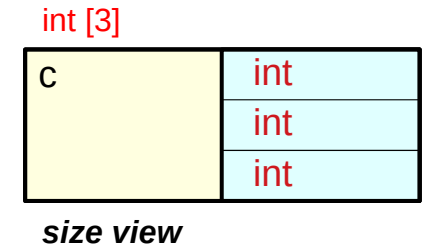
`int b[2];`



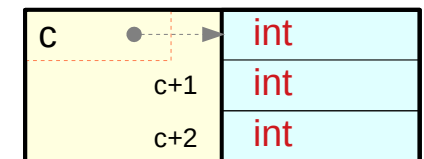
`int [2]` – size view
`int (*)` – address view



`int c[3];`

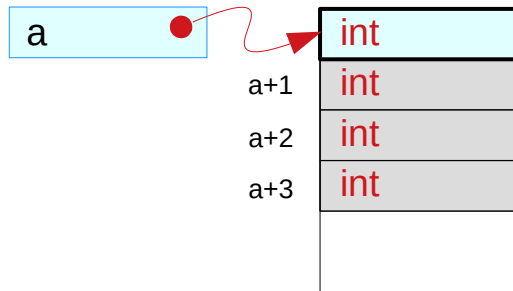


`int [3]` – size view
`int (*)` – address view



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

`int *a;`

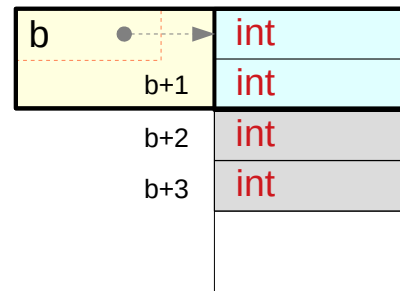


```
a[0] = *a
a[1] = *(a+1)
a[2] = *(a+2)
a[3] = *(a+3)
```

syntactically legitimate

programmers must ensure their validity

`int b[2];`

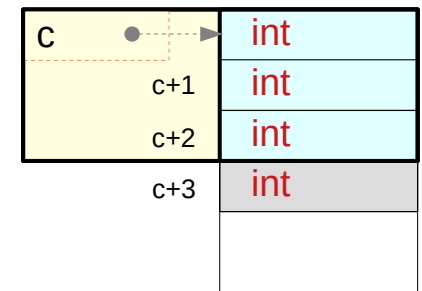


```
b[0] = *b
b[1] = *(b+1)
b[2] = *(b+2)
b[3] = *(b+3)
```

syntactically legitimate

programmers must ensure their validity

`int c[3];`



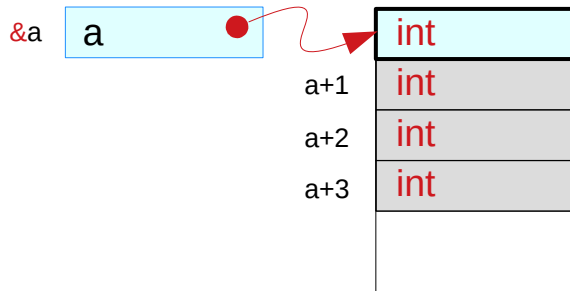
```
c[0] = *c
c[1] = *(c+1)
c[2] = *(c+2)
c[3] = *(c+3)
```

syntactically legitimate

programmers must ensure their validity

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

`int *a;`



`type(&a) = int **`

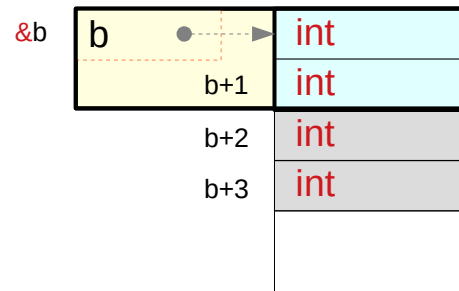
`type(a) = int *`

`type(*a) = int`

`value(&a) ≠ value(a)`

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

`int b[2]`



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

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

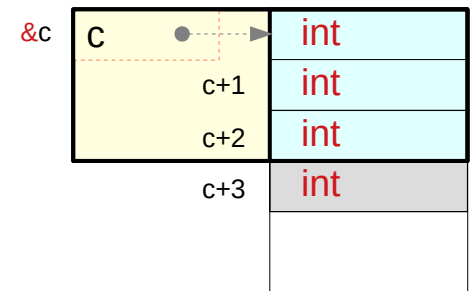
`type(*b) = int`

`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];`



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

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

`type(*c) = int`

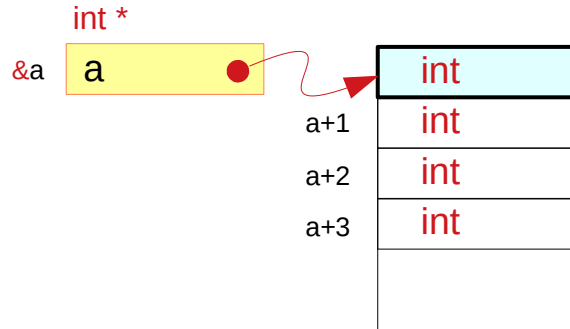
`value(&c) = value(c)`

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

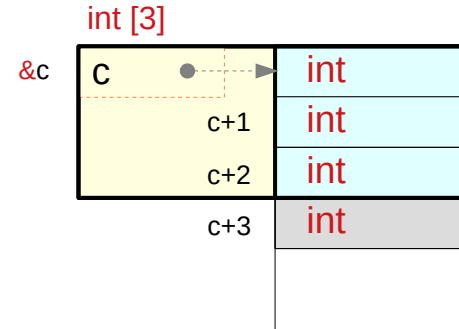
`&c` and `c` evaluate the same address but have different types and also different sizes

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

`int *a;`



`int c[3];`



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

`value(&a) ≠ value(a)` `int *`

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

real memory location for `a`

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

`sizeof (c) = sizeof(*c) * 3` `int [3]`

`value(&c) = value(c)` `int (*)`

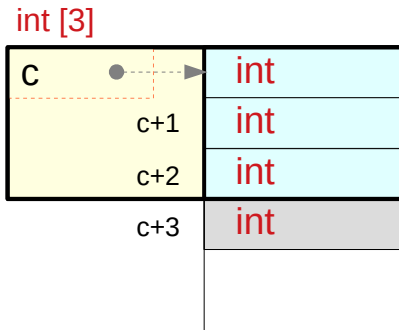
the starting address of array variable `c` is equal to the address of the 1st element

no actual memory location for `c`

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

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

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



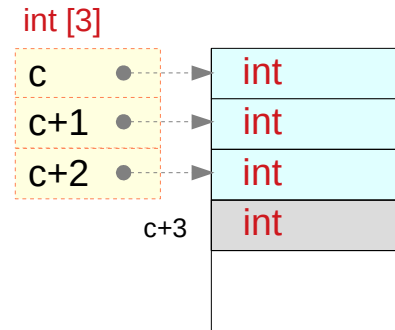
`sizeof (c) = sizeof(int) * 3`

`value(&c) = value(c)`

`type(c) = int *`

`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 *`

`type(c+1) = int *`

`type(c+2) = int *`

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

Virtual pointer p in an aggregated data

Virtual pointers

size view `p[N][] ... []`

`sizeof(p) = sizeof(*p) * N`

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

address view `(*p)[] ... []`

`value(&p) = value(p)`

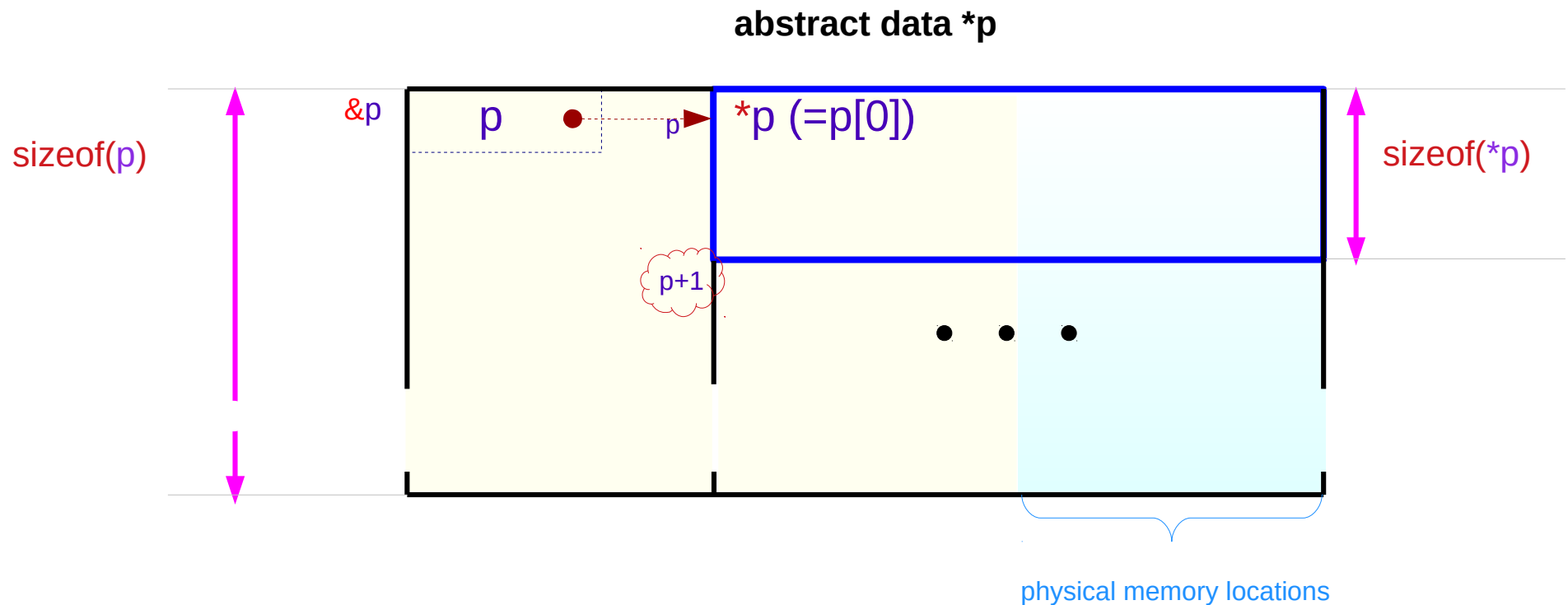
Real pointers

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

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

`value(&p) ≠ value(p)`

Virtual pointer p in an aggregated data



size view $p[N][] \dots []$

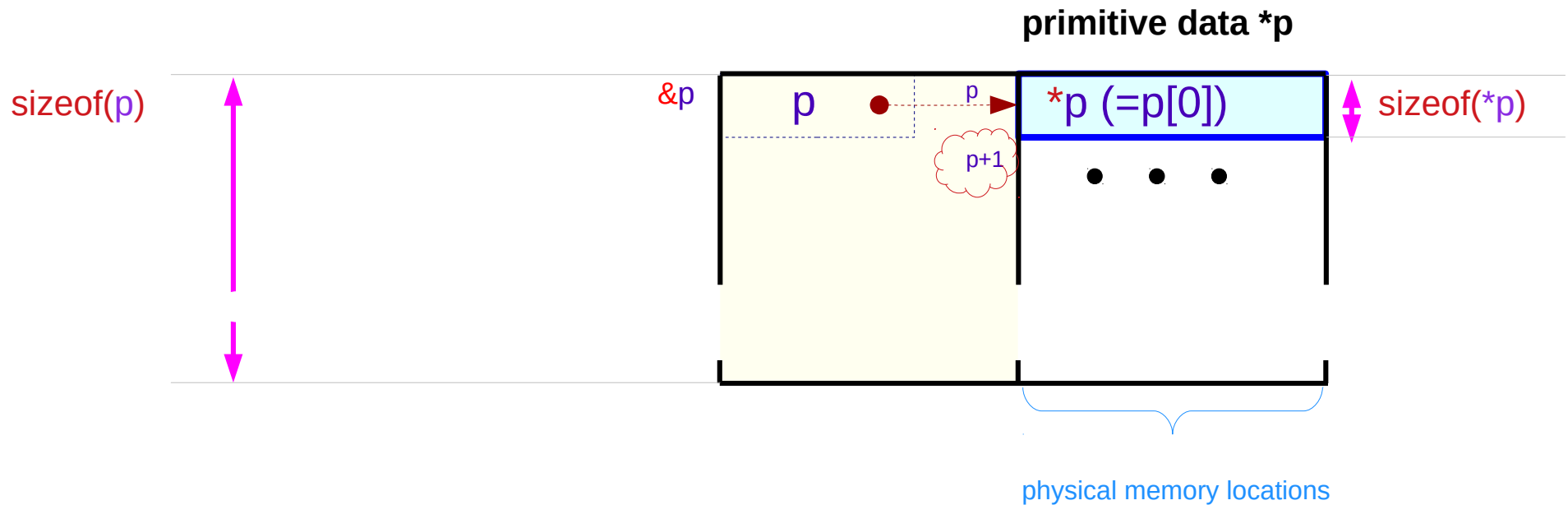
$$\text{sizeof}(p) = \text{sizeof}(*p) * N$$

$$\text{value}(p+1) = \text{value}(p) + \text{sizeof}(*p)$$

address view $(*p)[] \dots []$

$$\text{value}(\&p) = \text{value}(p)$$

Virtual pointer p in an aggregated data



size view $p[N][] \dots []$

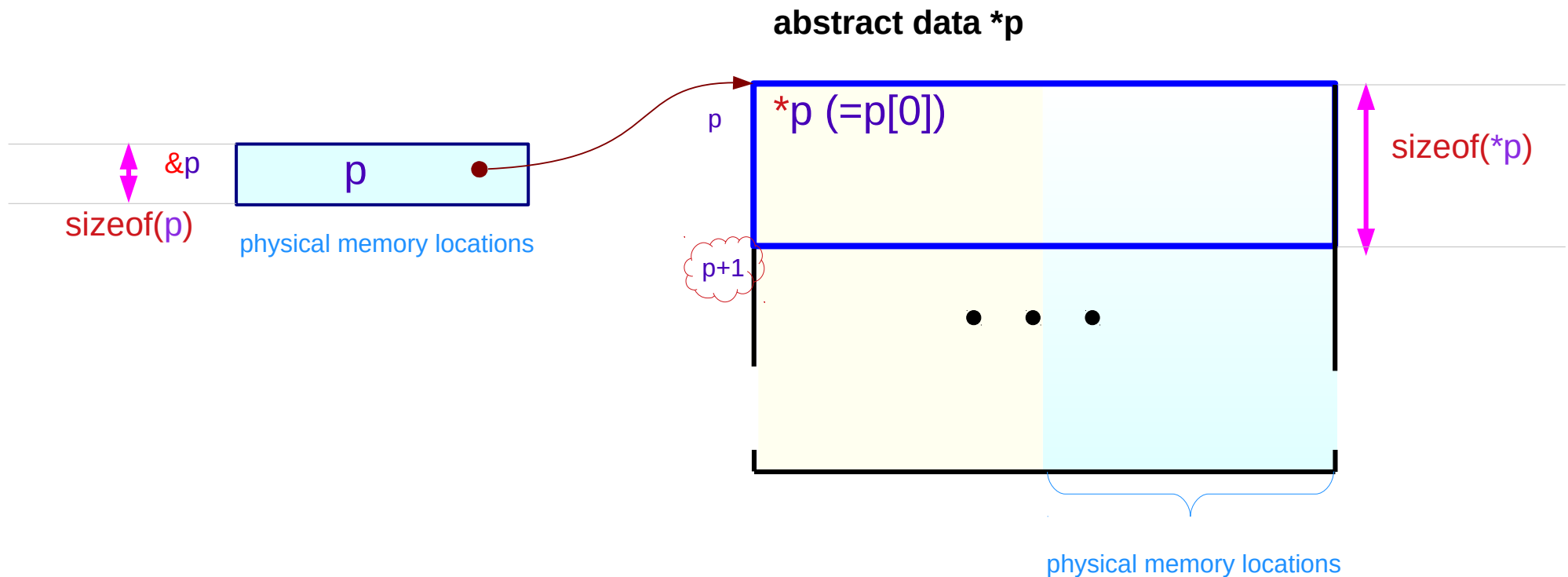
$$\text{sizeof}(p) = \text{sizeof}(*p) * N$$

$$\text{value}(p+1) = \text{value}(p) + \text{sizeof}(*p)$$

address view $(*p)[] \dots []$

$$\text{value}(\&p) = \text{value}(p)$$

Real pointers

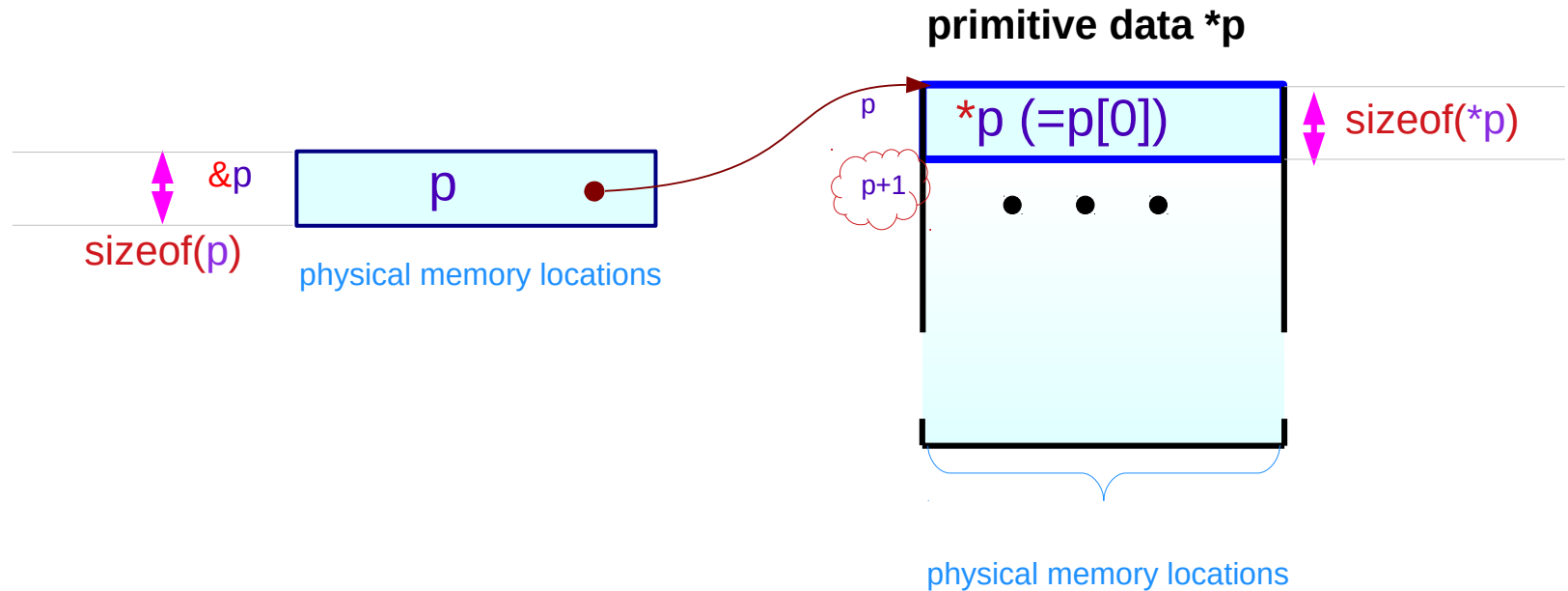


$sizeof(p)$ = pointer size (4/8 bytes)

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

$value(\&p) \neq value(p)$

Real pointers



$\text{sizeof}(p)$ = pointer size (4/8 bytes)

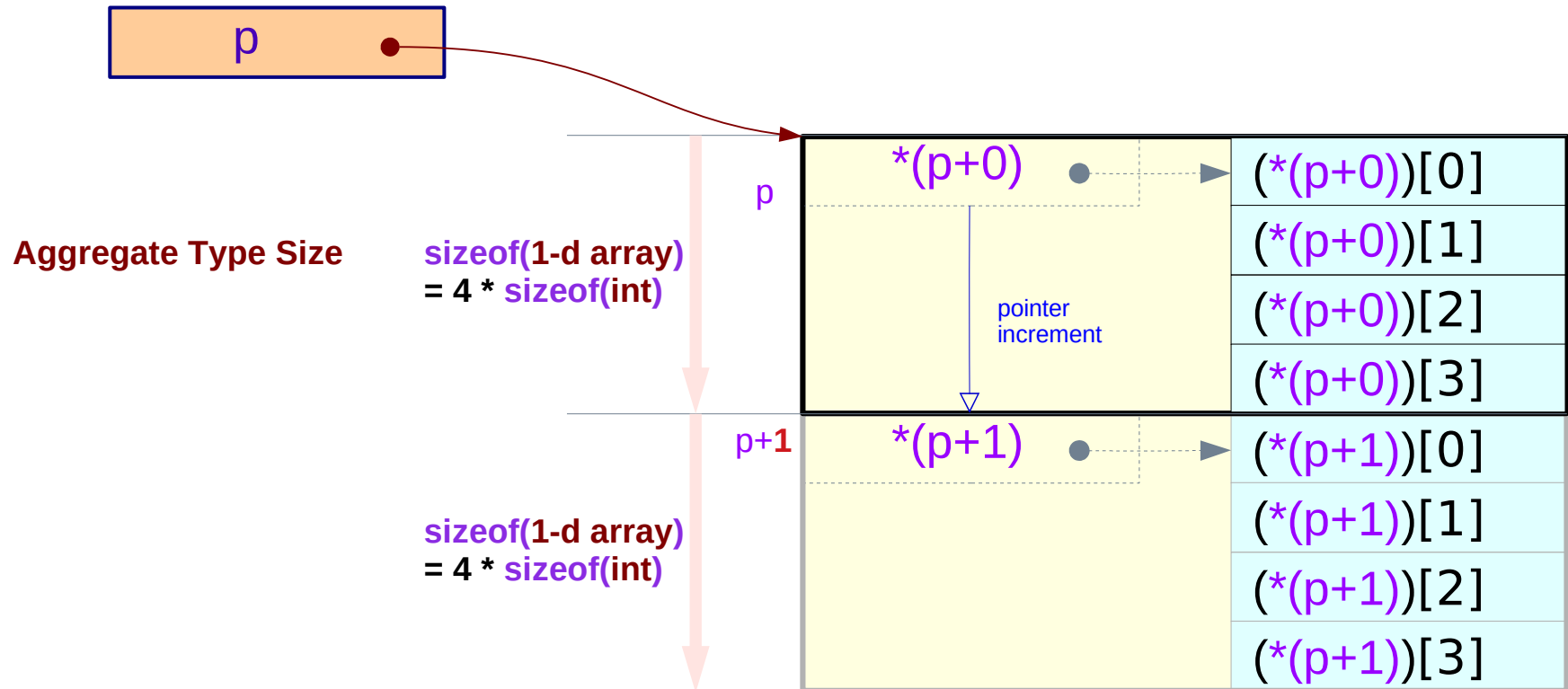
$\text{value}(p+1) = \text{value}(p) + \text{sizeof}(*p)$

$\text{value}(\&p) \neq \text{value}(p)$

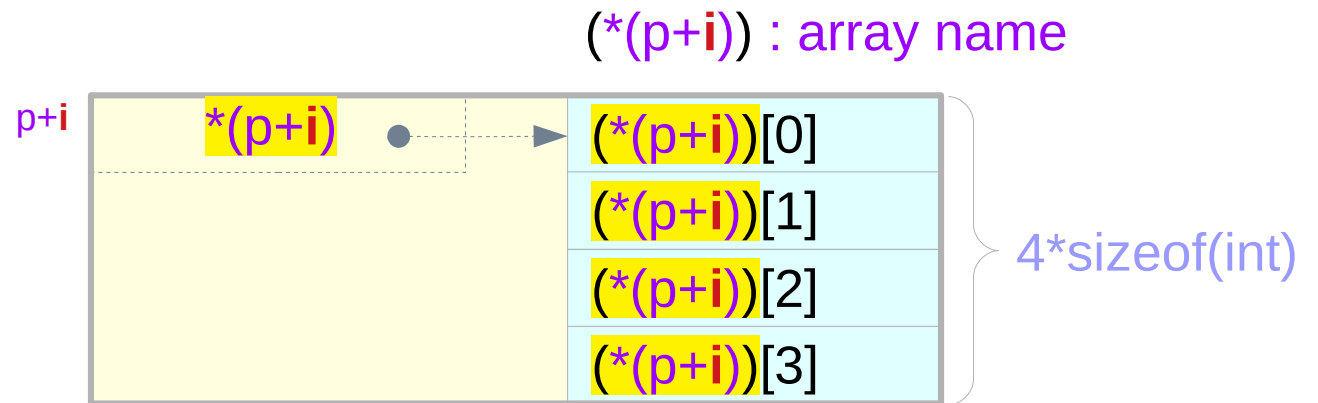
Incrementing a 1-d array pointer

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

$$\begin{aligned} \text{value}(p+1) - \text{value}(p) &= \text{sizeof}(*p) \\ &= (\text{long}) (p+1) - (\text{long}) (p) &= 4 * \text{sizeof}(\text{int}) \end{aligned}$$

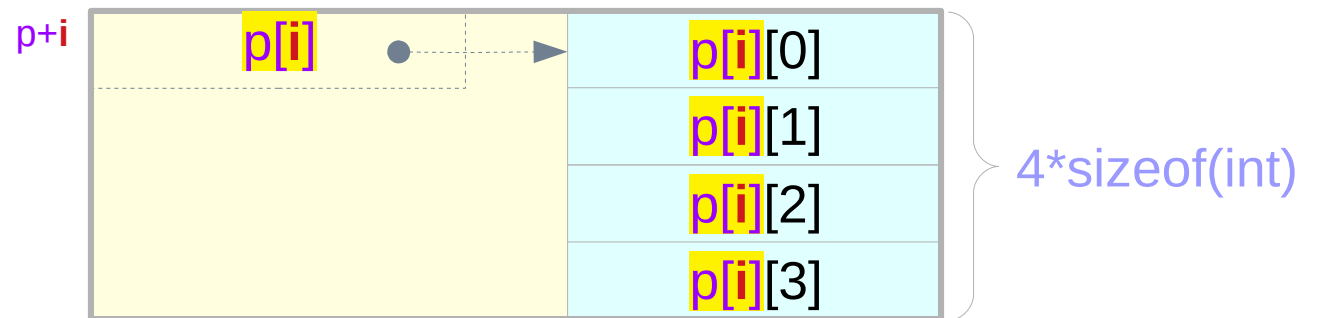


Equivalence : $*(p+i) = p[i]$



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

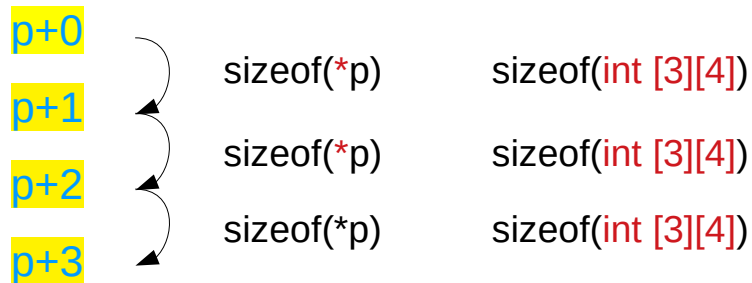
|| equivalence



$p[i]$: 1-d array name

Incrementing a pointer to a 1-d array

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

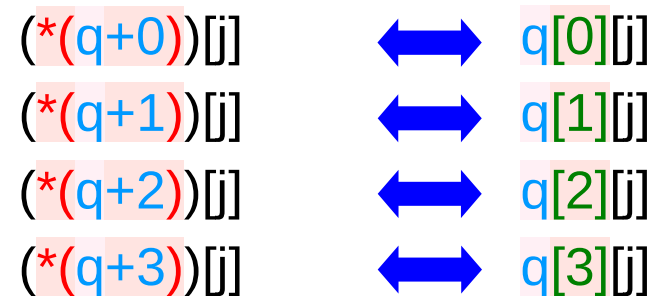
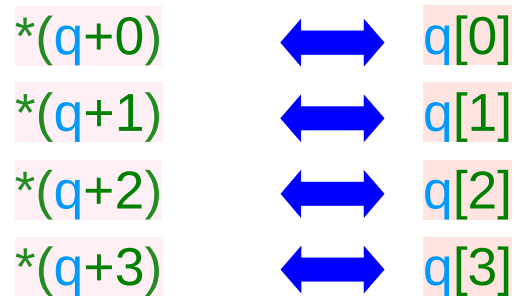
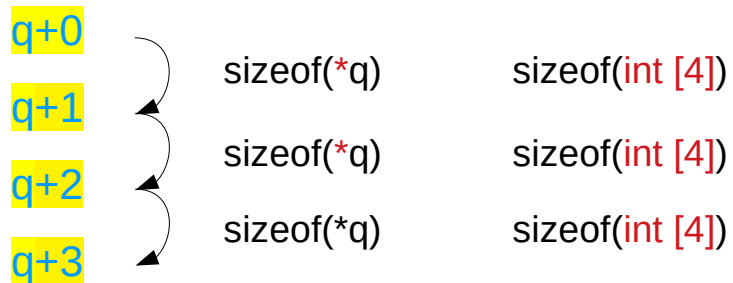


$*((*p)+0)$ ↔ $(*p)[0]$
 $*((*p)+1)$ ↔ $(*p)[1]$
 $*((*p)+2)$ ↔ $(*p)[2]$
 $*((*p)+3)$ ↔ $(*p)[3]$

$(*((*p)+0))[j]$ ↔ $(*p)[0][j]$
 $(*((*p)+1))[j]$ ↔ $(*p)[1][j]$
 $(*((*p)+2))[j]$ ↔ $(*p)[2][j]$
 $(*((*p)+3))[j]$ ↔ $(*p)[3][j]$

Incrementing a pointer to a 1-d array

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



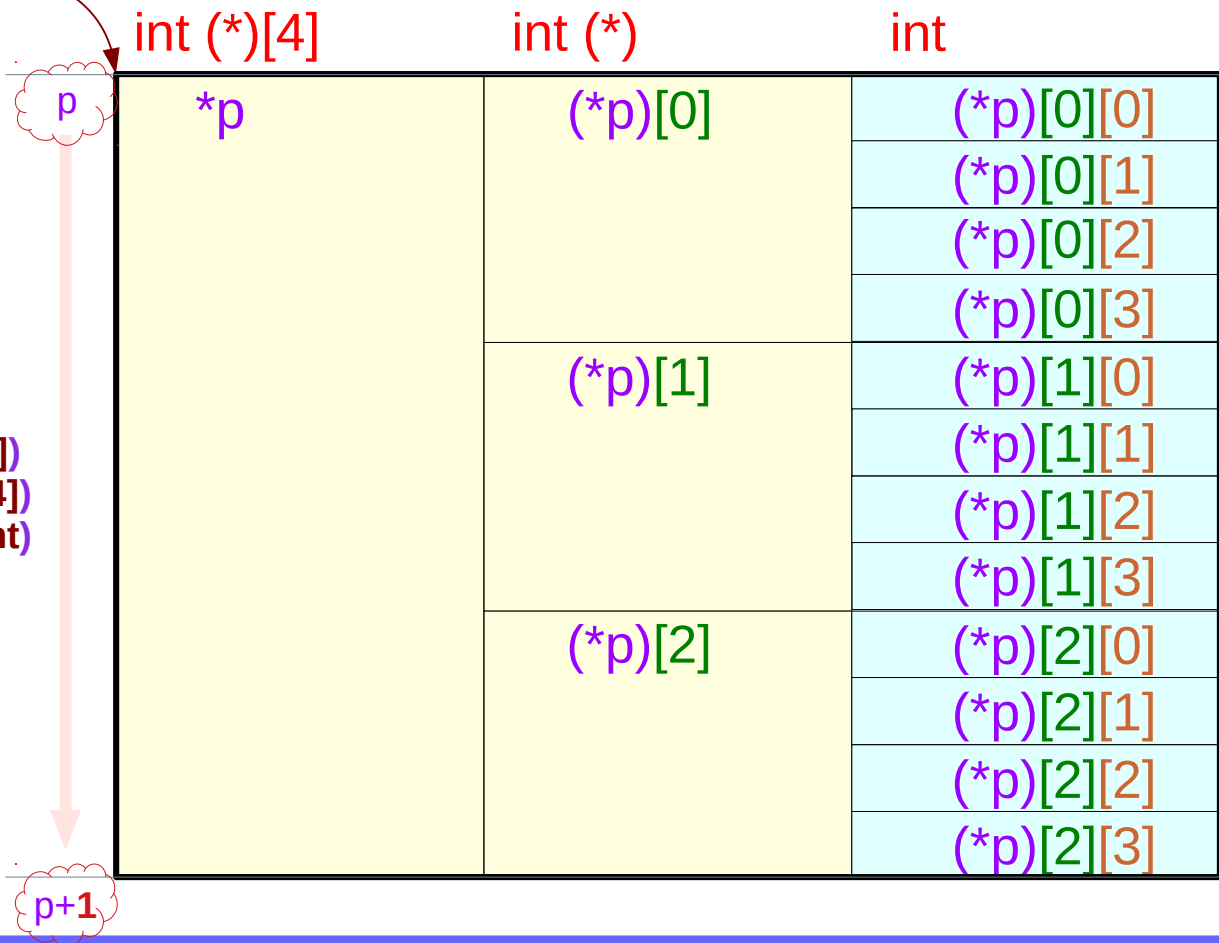
2-d array access using p

2-d array pointer

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



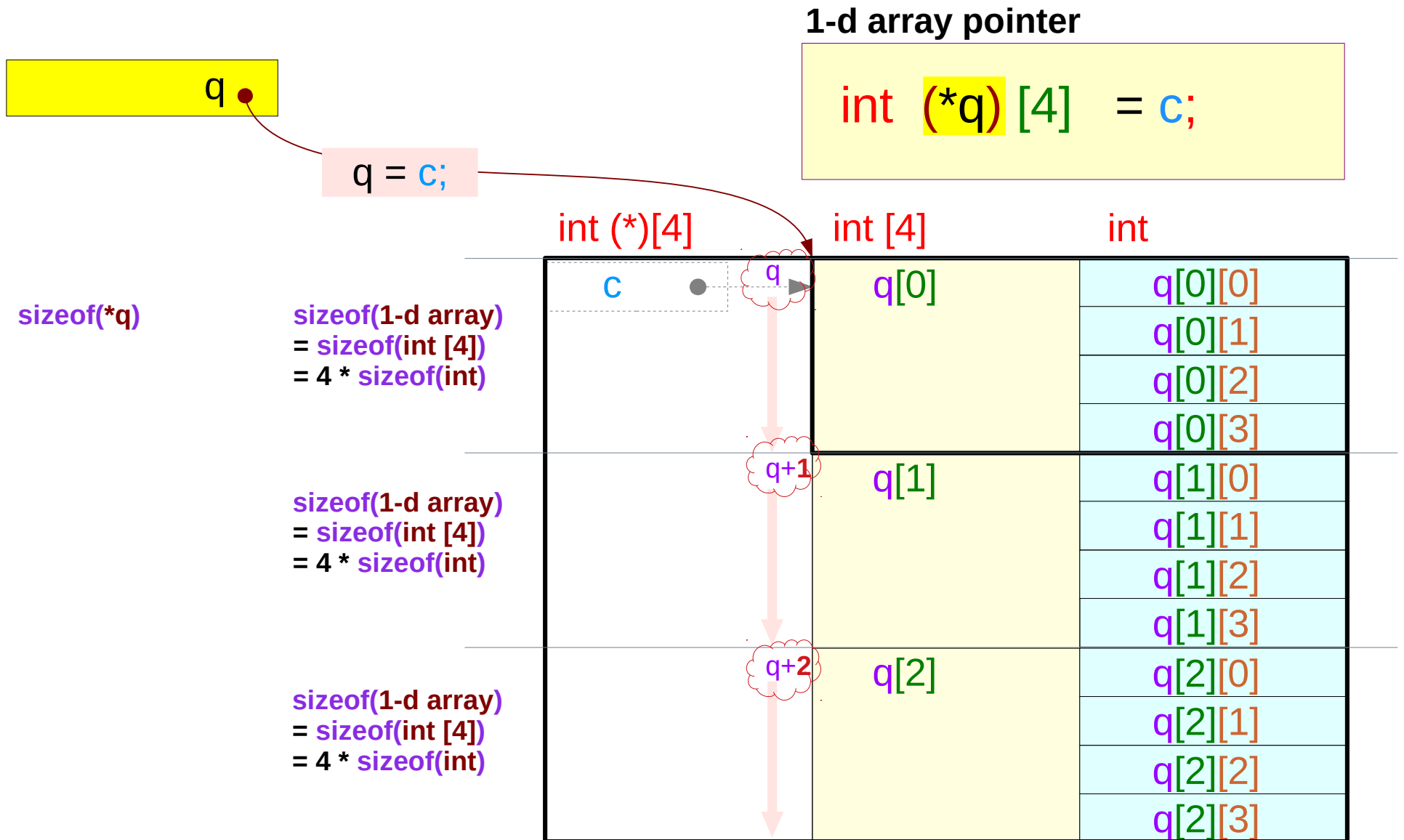
p = &c;



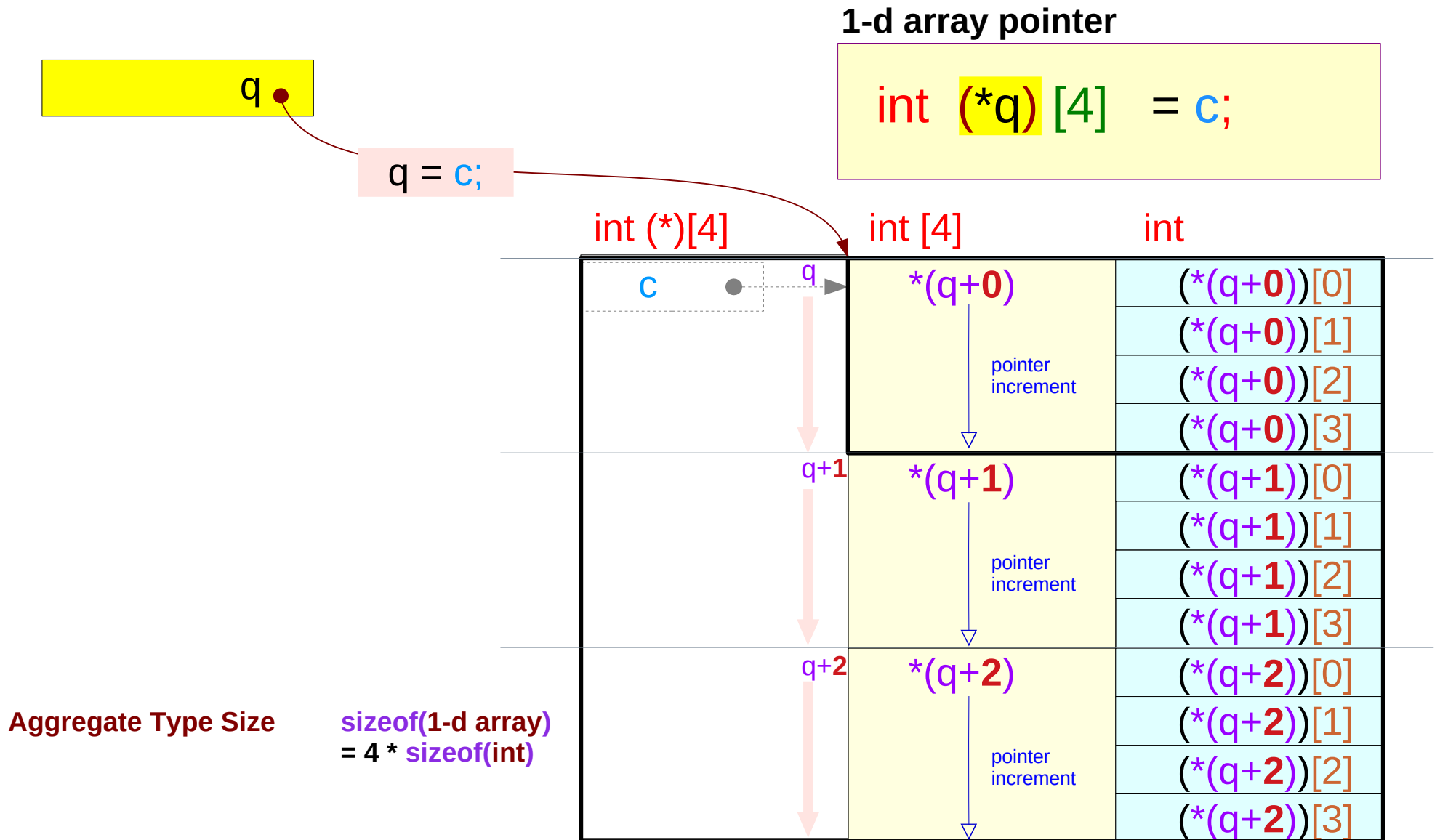
sizeof(*p)

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

2-d array access using q



2-d array access using q



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

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

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

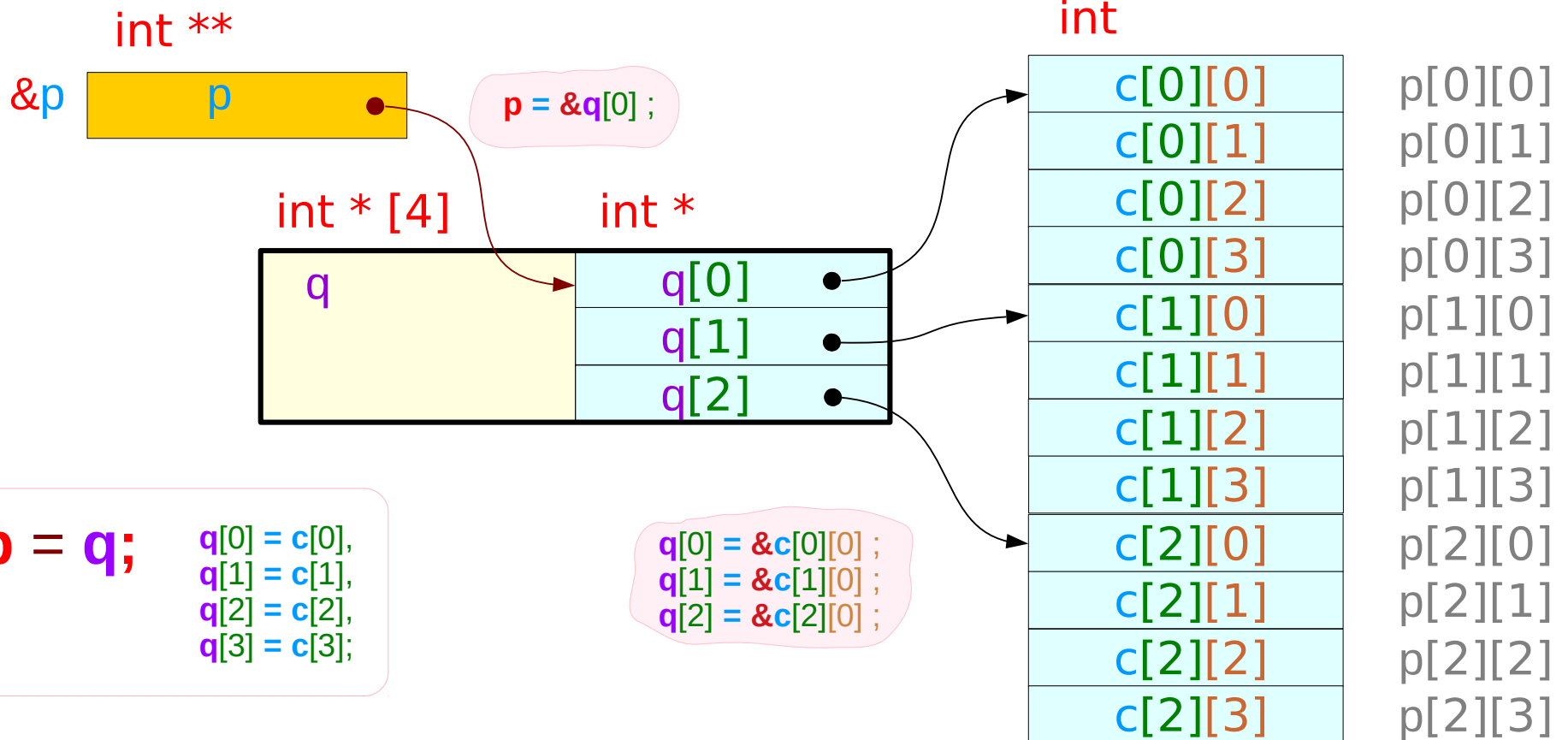
int

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]

2-d array access using double pointers q

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

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



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