Applications of Pointers (1A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

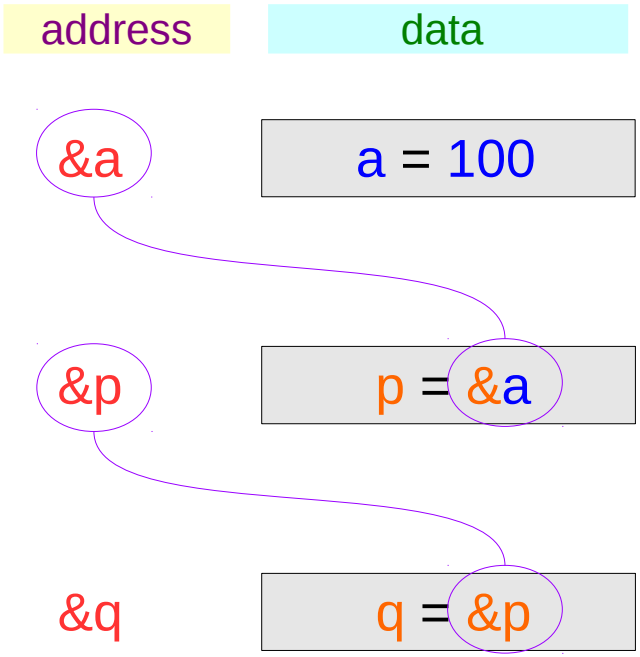
This document was produced by using LibreOffice.

Variables and their addresses

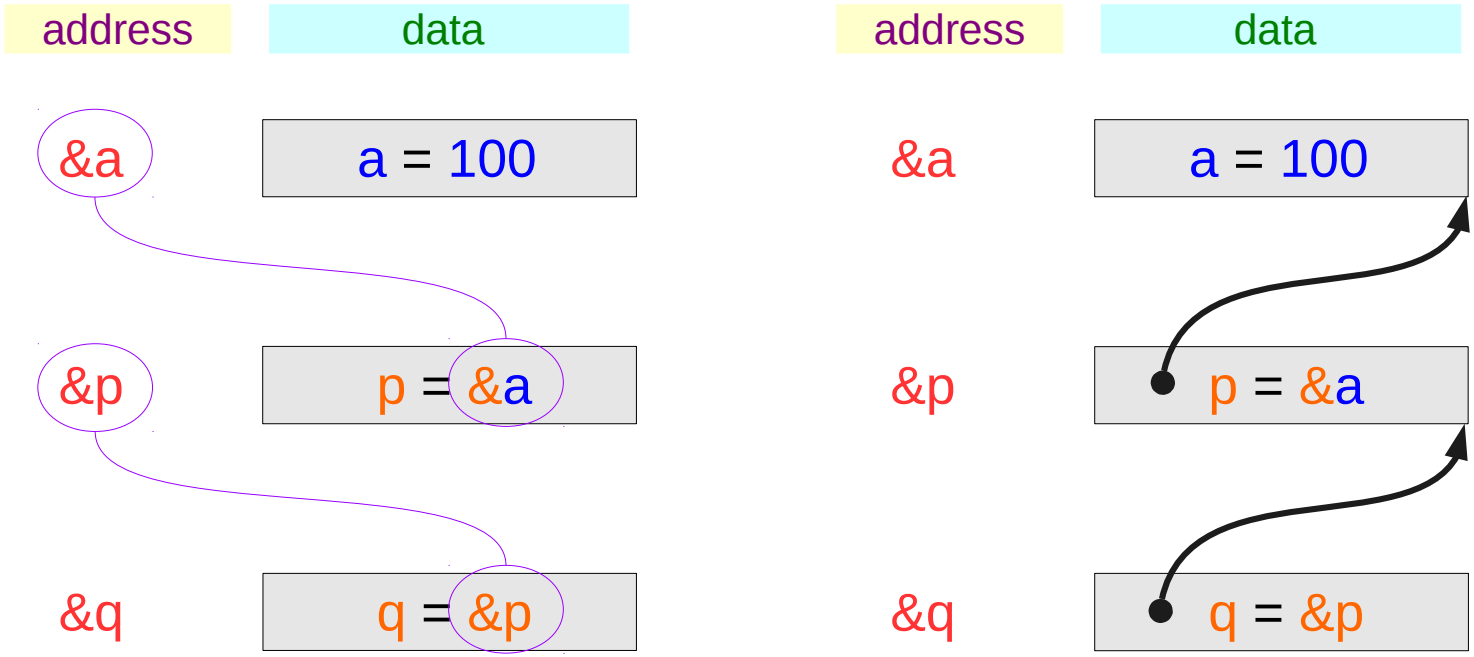
	address	data
int a;	&a	a
int * p;	&p	p
int ** q;	&q	q

Initialization of Variables

```
int     a = 100;

int   * p = &a;

int  ** q = &p;
```

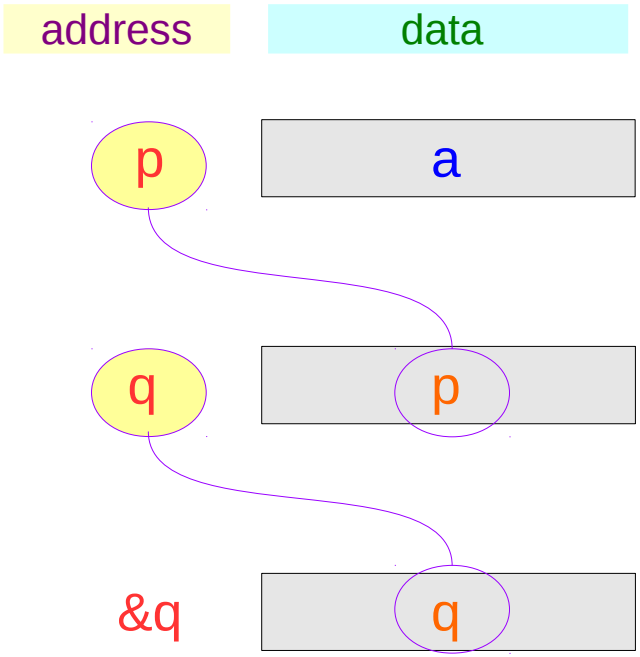


Traditional arrow notations



Pointed addresses : p, q

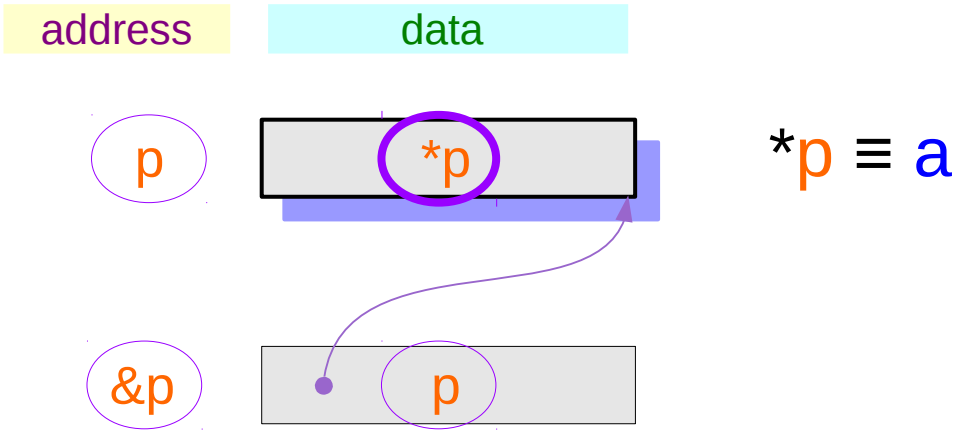
```
int      a;

int   *  p = &a;

int   ** q = &p;
```



p = &a
q = &p

Dereferenced Variables : *p

```
int     a;

int   * p = &a;

int   ** q = &p;
```



Dereferenced Variables : *p

```
int      a;

int   *  p = &a;

int   ** q = &p;
```

Address assignment

Variable aliasing

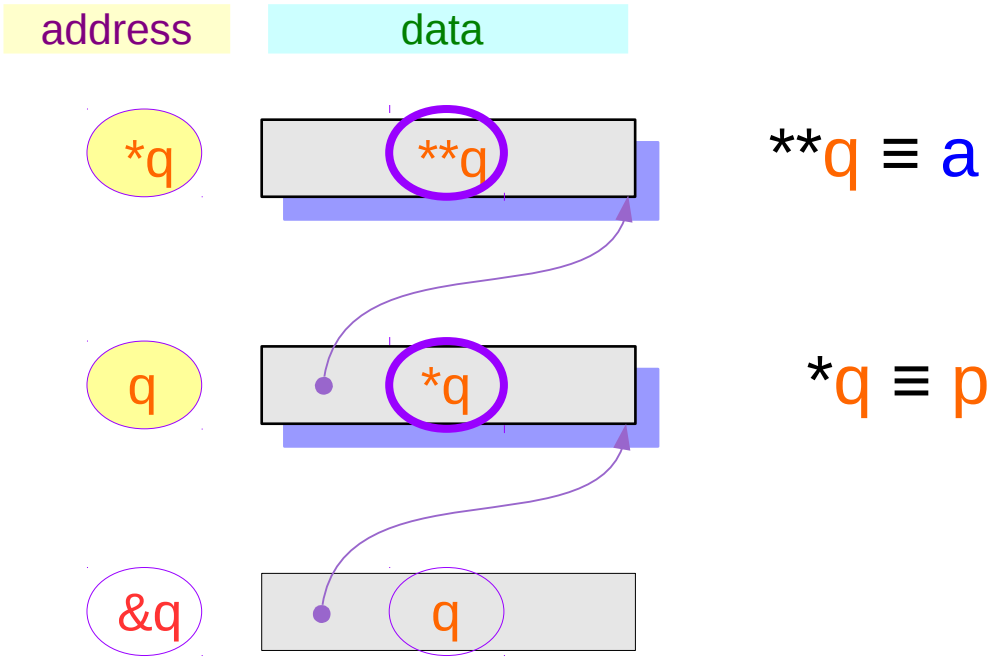
p = &a ➡ *p ≡ a

```
  p   ≡   &a
*(p)  ≡ *(&a)
* p   ≡    a
```

Relations after address assignment

Dereferenced Variables : *q, **q

```
int     a;

int   * p = &a;

int   ** q = &p;
```



**q ≡ a

*q ≡ p

Dereferenced Variables : *q, **q

```
int      a;

int   *  p = &a;

int   ** q = &p;
```

Address assignment

Variable aliasing

p = &a ➡ *p ≡ a

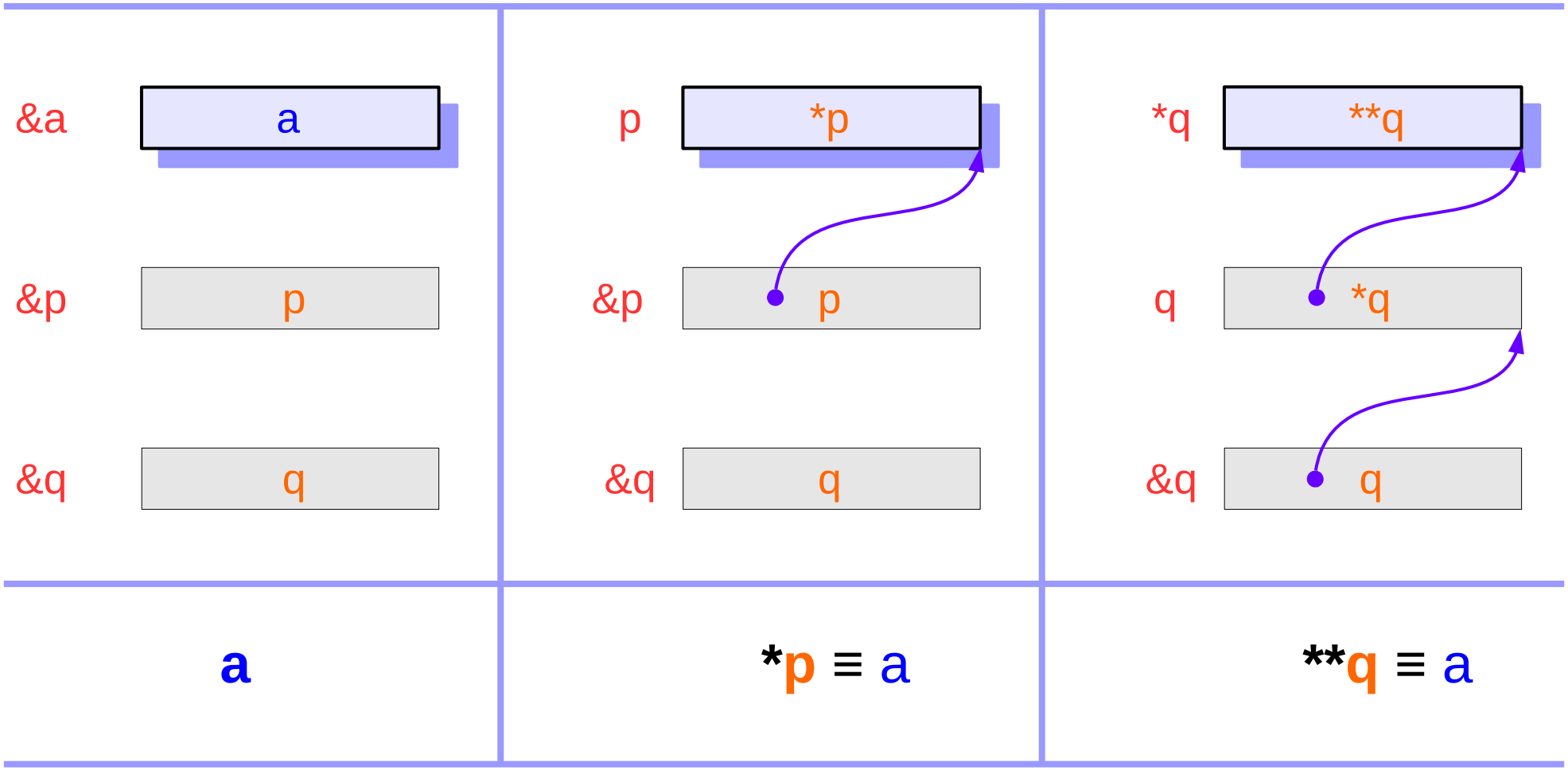
q = &p ➡ *q ≡ p

➡ **q ≡ a

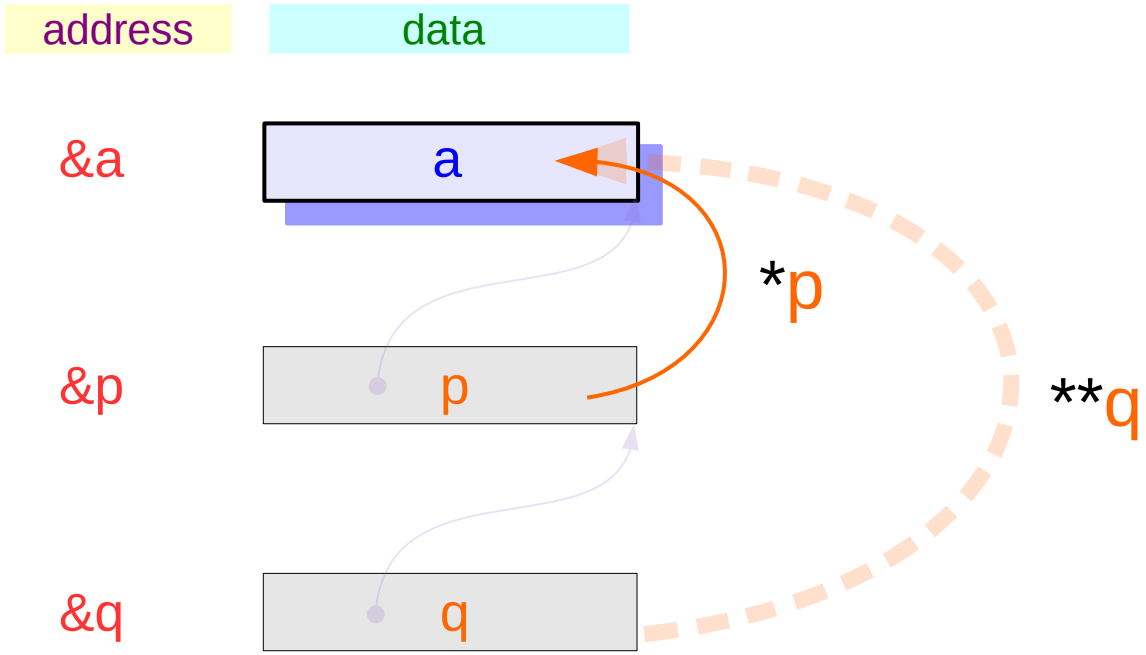
```
  q   ≡   &p
*(q)  ≡ *(&p)
* q   ≡    p
**q   ≡   *p
**q   ≡    a
```

Relations after address assignment

Two more ways to access **a** : ***p**, ****q**



Two more ways to access a : *p, **q

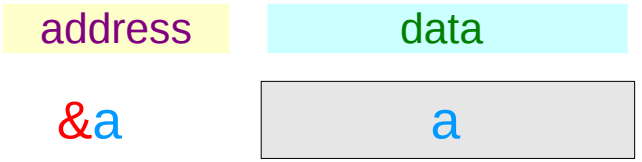


1) Read / Write a
2) Read / Write *p
3) Read / Write **q

Variables

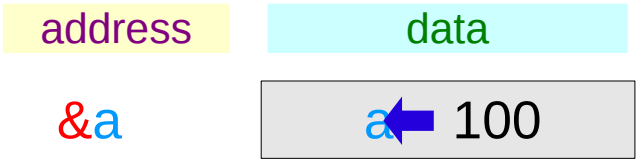
```
int     a;
```

a can hold an *integer*




```
a = 100;
```

a holds 100



Pointer Variables

```
int *       p;
```

p can hold an ***address***

```
int   * p;
```

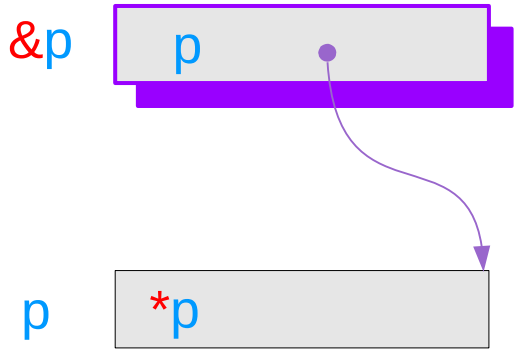
pointer to int

p holds an ***address***
of a int type data

```
int   * p;
```

int

***p** holds
a int type ***data***



Pointer to Pointer Variable

```
int **      q;
```

q holds an *address*

```
int  ** q;
```

pointer to
pointer to int

q holds an ***address*** of
a pointer to int type data

```
int  *   *q;
```

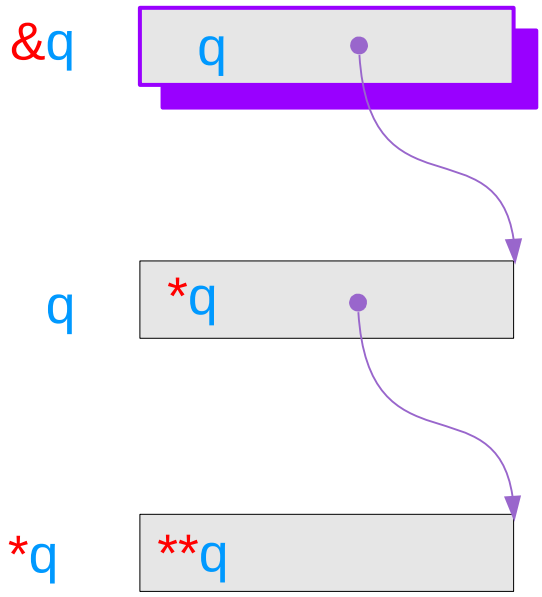
pointer to int

q** holds an ***address of
a int type data

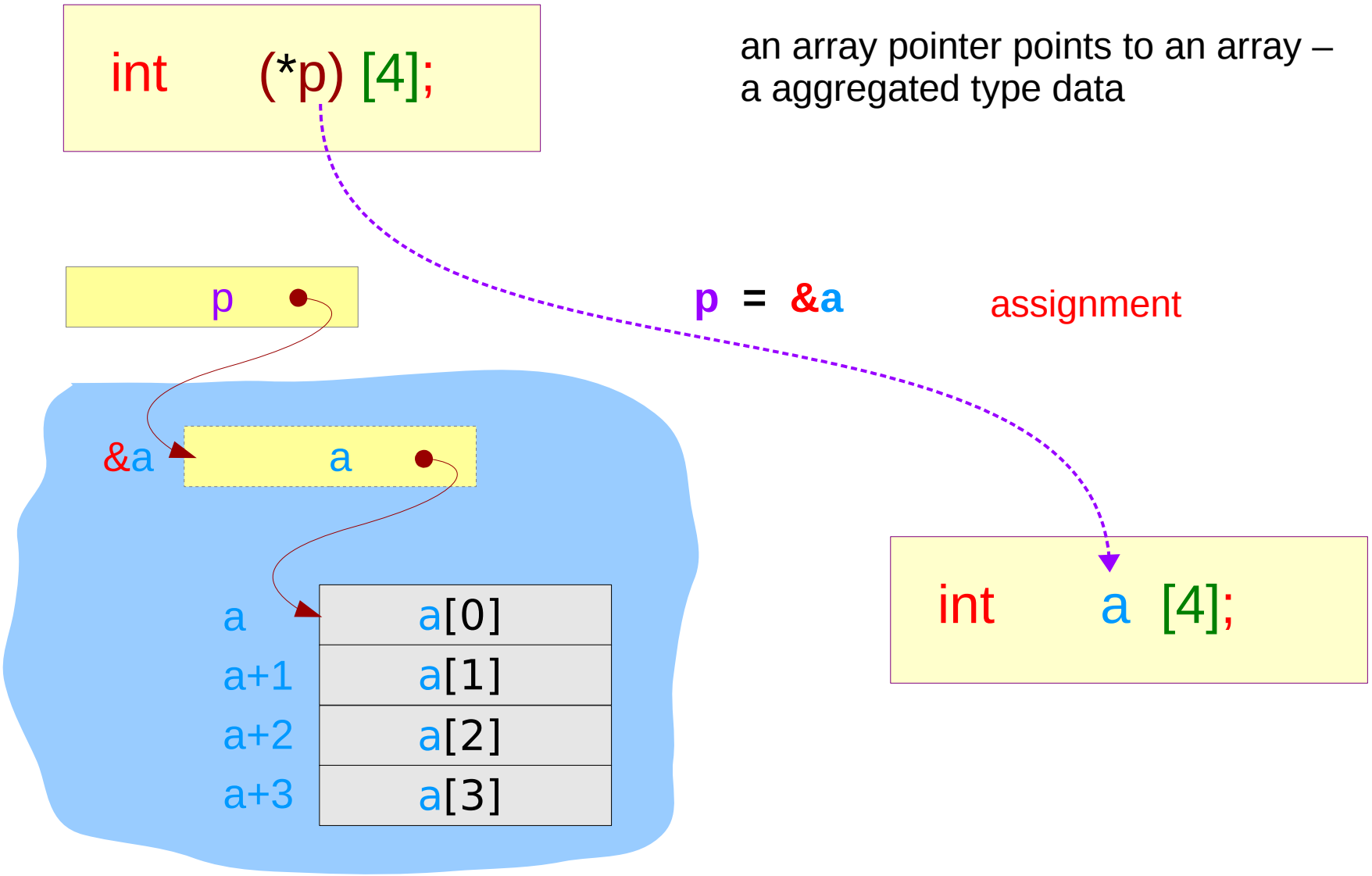
```
int    **q;
```

int

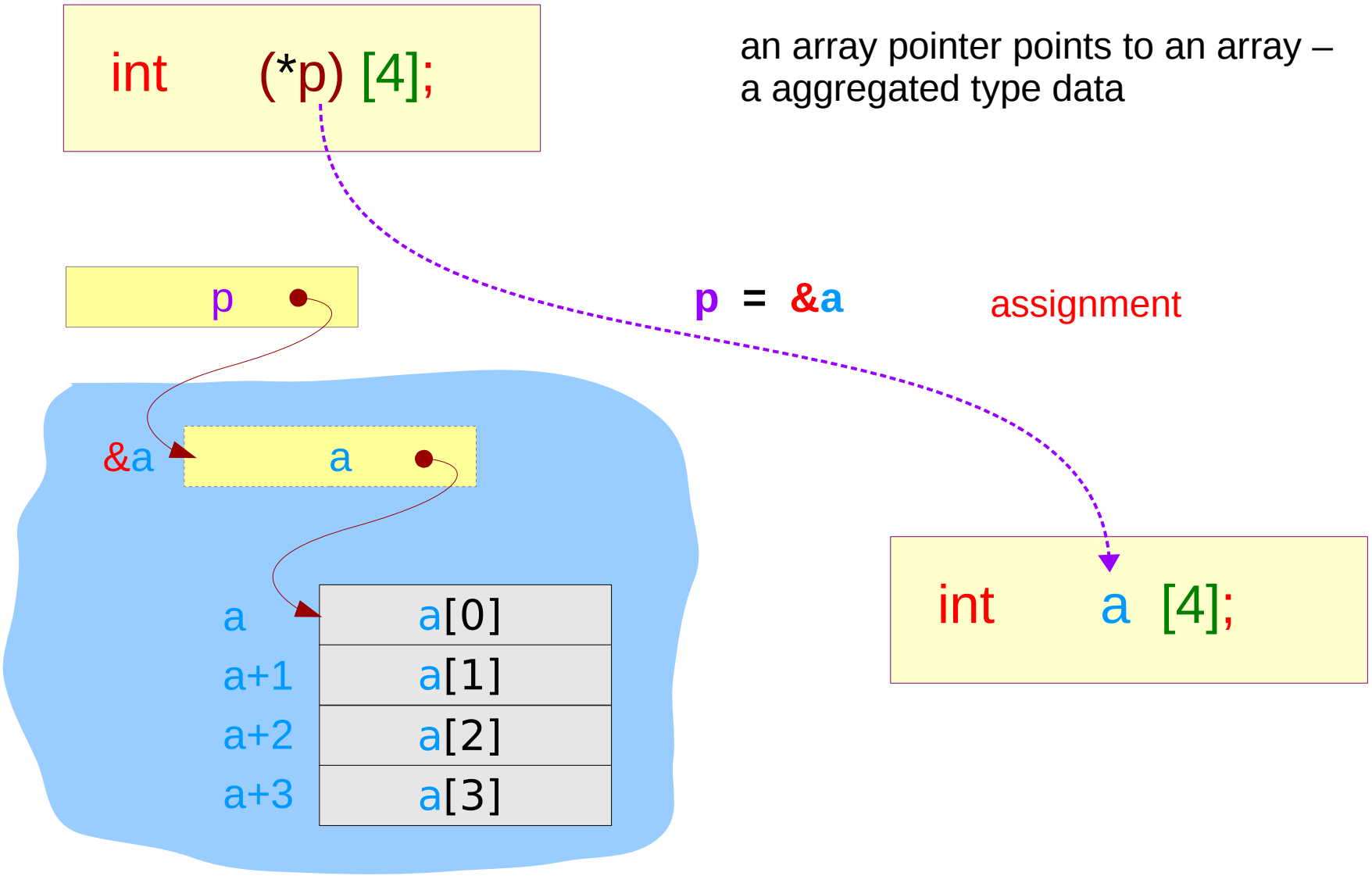
****q** holds a int type data



Pointer to an array – a variable view

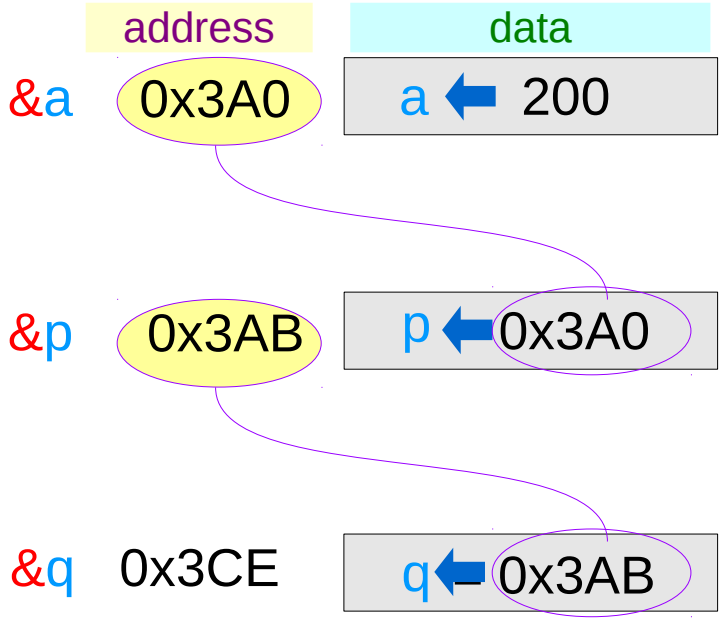


Pointer to an array – a variable view



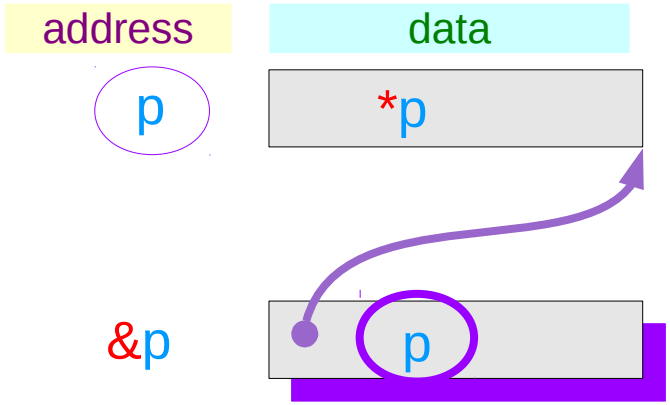
Pointer Variables Examples

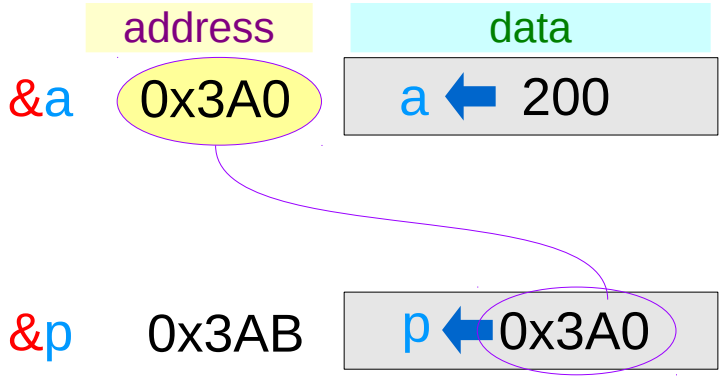
```
int      a = 200;

int *        p = & a;

int **       q = & p;
```




```
&q  ➡ 0x3CE
 q  ➡ 0x3AB
*q  ➡ 0x3A0
**q ➡ 200
```

Pointer Variable **p** with an arrow notation





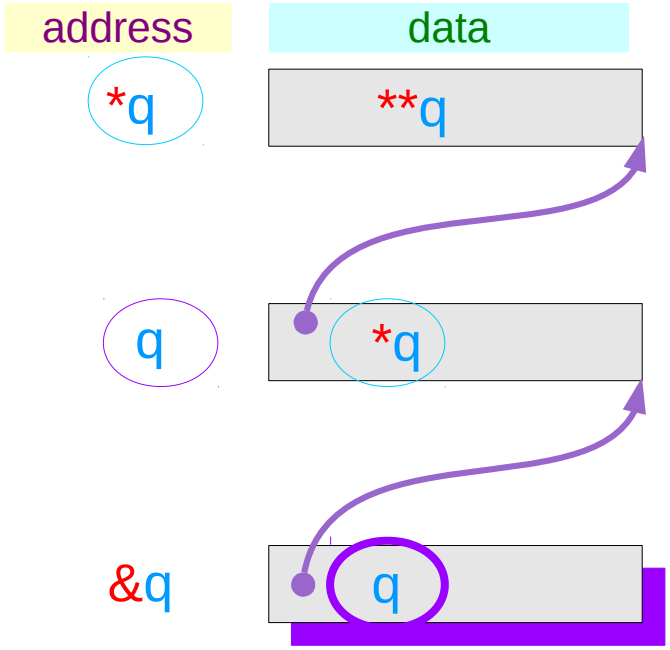
using an arrow notation

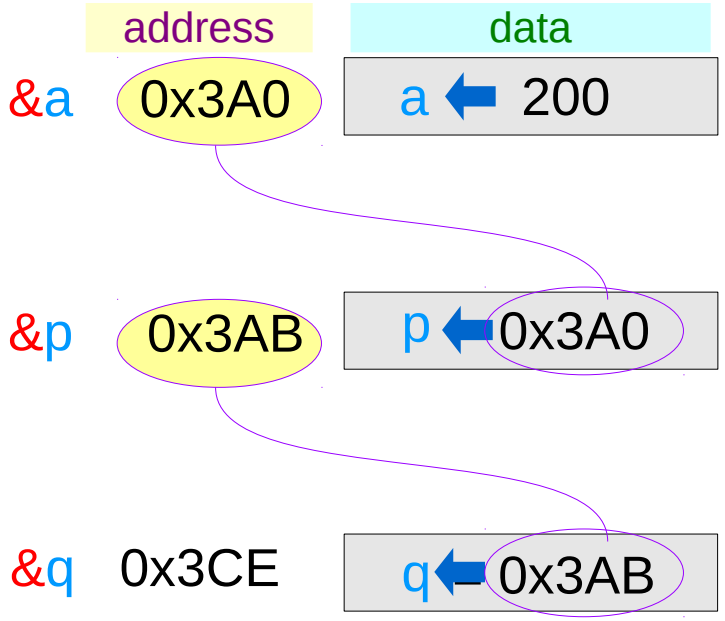
&p ➡ 0x3AB

p ➡ 0x3A0

*p ➡ 200

Pointer Variable **q** with an arrow notation





using an arrow notation

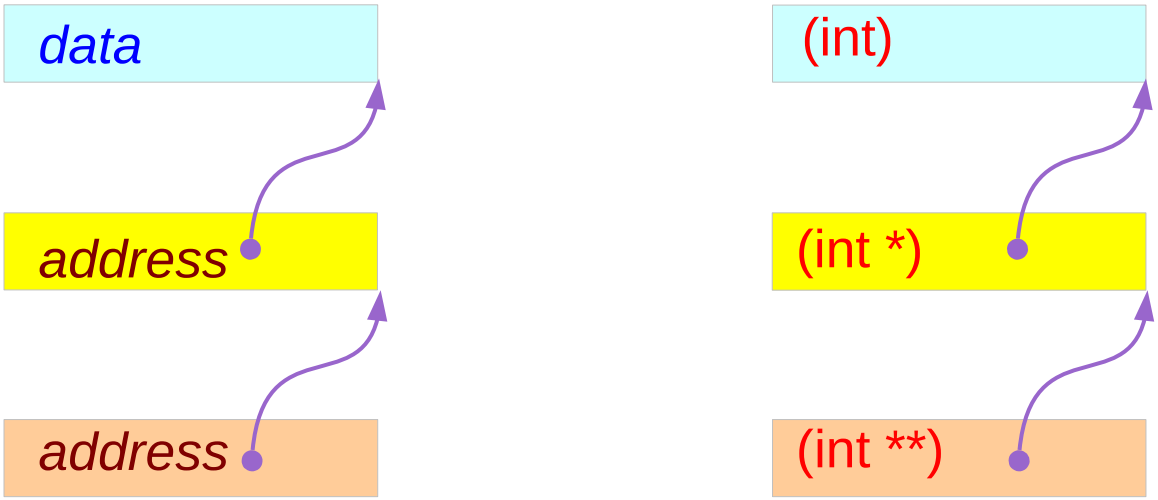
&q ➡ 0x3CE

q ➡ 0x3AB

*q ➡ 0x3A0

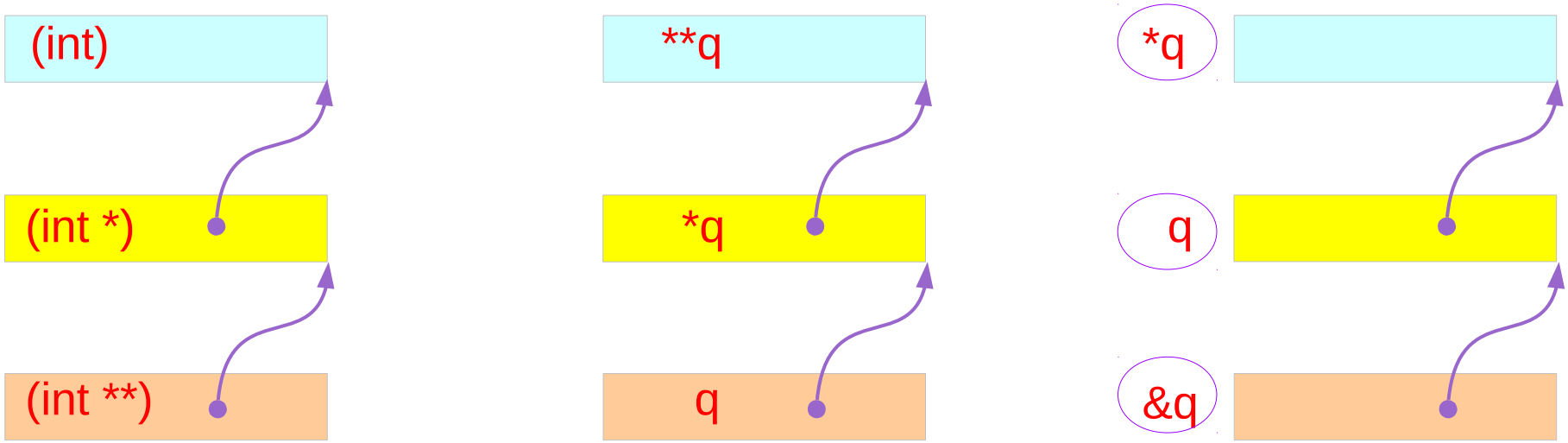
**q ➡ 200

The type view point of pointers



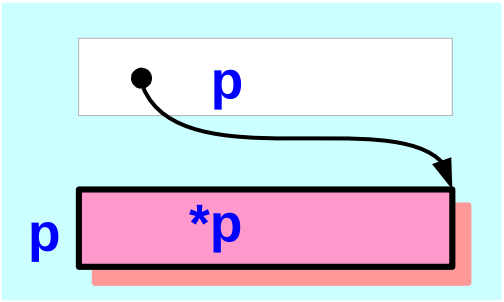
Types

The different view points of pointers

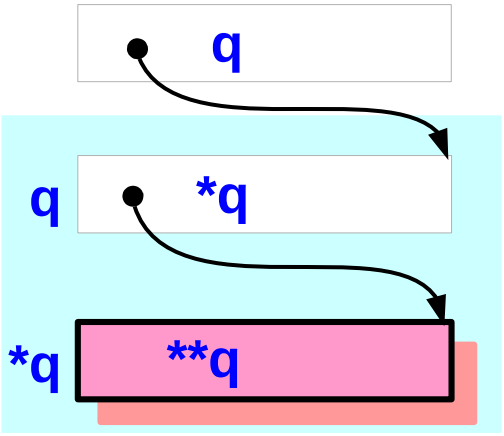


Single and Double Pointer Examples (1)

```
int    a ;
int  * p ;
int  **q ;
```

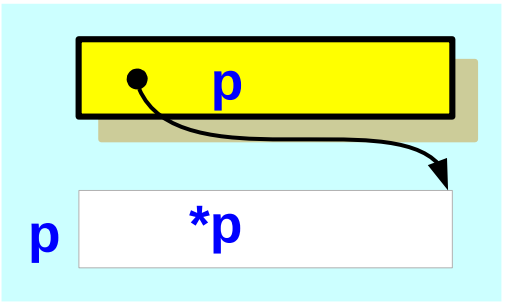


a, ***p**, and ****q**:
int variables

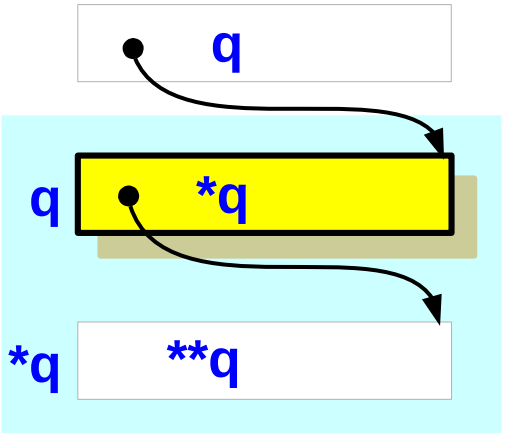


Single and Double Pointer Examples (2)

```
int      a ;
int *    p ;
int *   *q ;
```

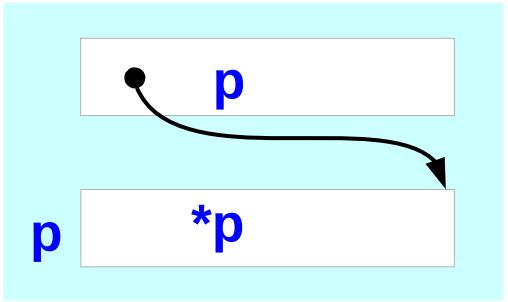


p and ***q** :
int pointer variables
(singlepointers)

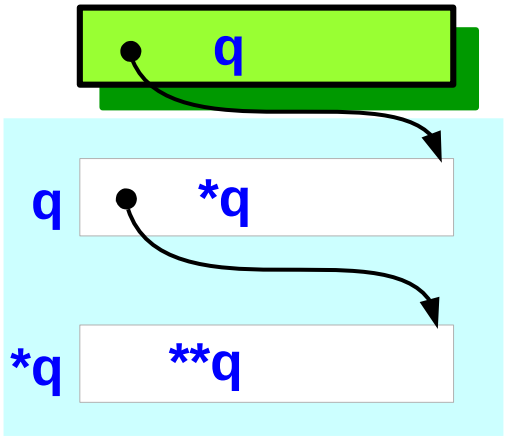


Single and Double Pointer Examples (3)

```
int        a  ;

int *      p  ;

int **     q  ;
```

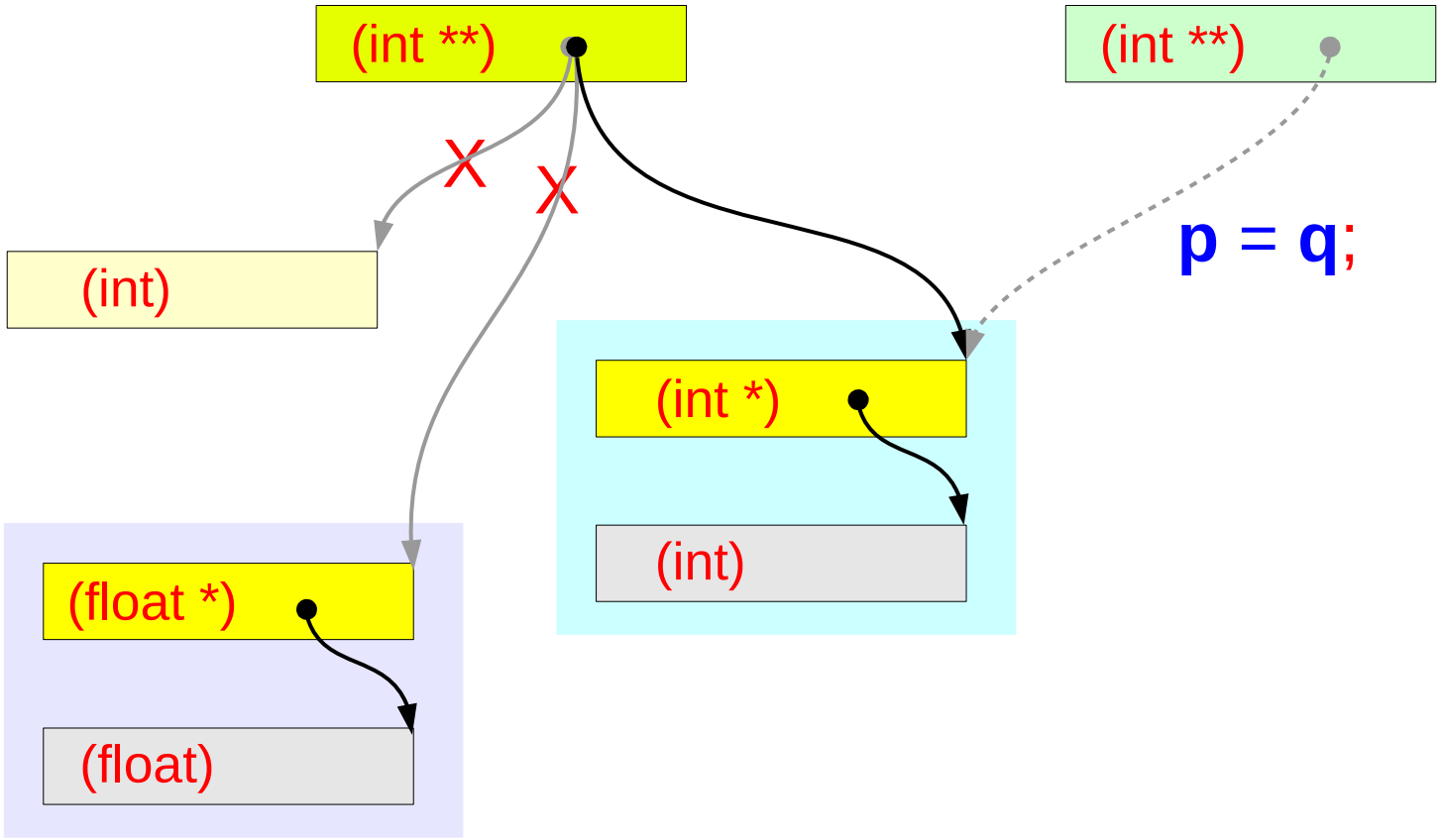


q :
double int pointer variables



Values of double pointer variables

```
int  ** p, **q ;
```



Pointed Addresses and Data

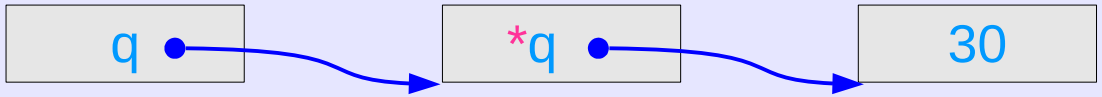
int a ; &a a =100

The variable a holds an integer data

int * p ; &p

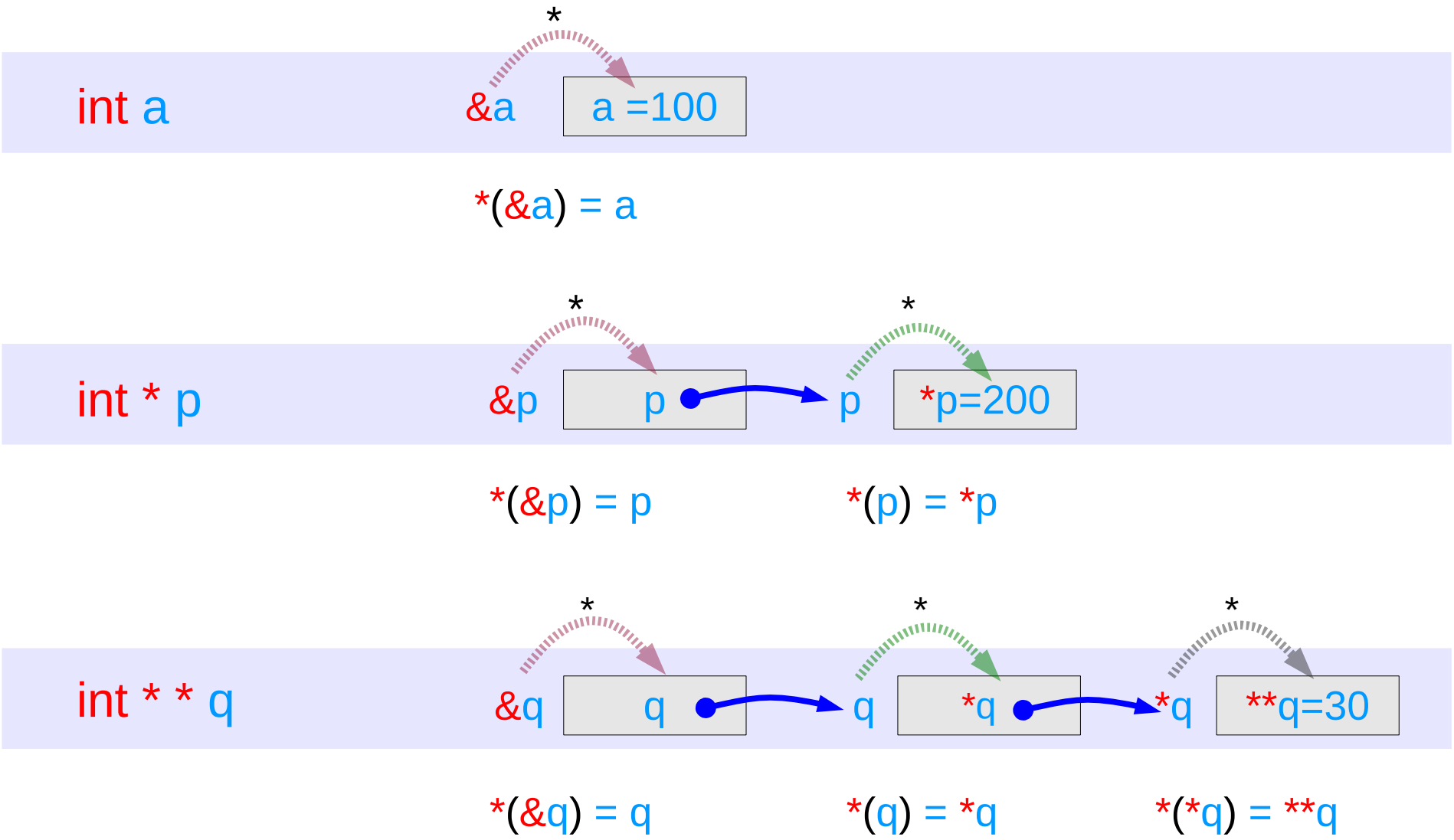
The **pointer** variable p holds an address,
at this address, an integer data is stored

int * * q ; &q



The **pointer** variable q holds an address,
at the address q, another address *q is stored,
at the address *q, an integer data **q is stored

Dereferencing Operations



Direct Access to an integer **a**

```
int a ;
```

&a | a =100

address	value
&a	a

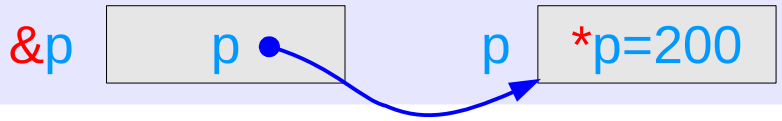
Direct Access

integer

1 memory access

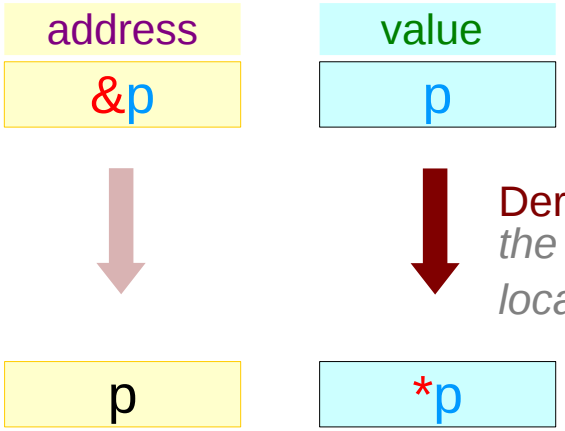
Indirect Access *p to an integer a

```
int * p ;
```



Indirect Access

2 memory accesses



Dereference Operator *
the content of the pointed location

Double Indirect Access **q to an integer a

```
int * * q ;
```

Double Indirect Access

3 memory accesses

address	value
&q	q
q	*q
*q	**q

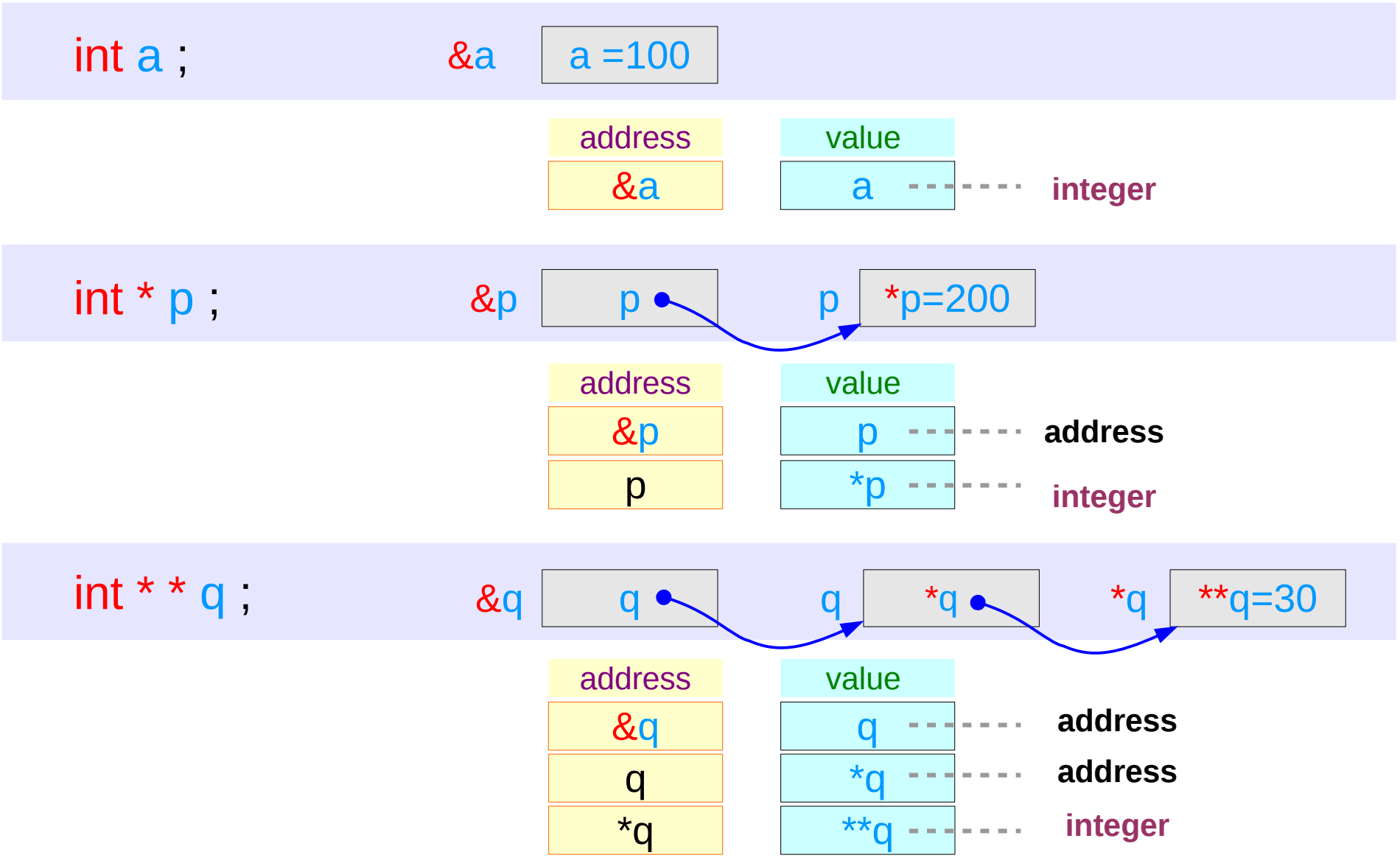
Dereference Operator *

the content of the pointed location

Dereference Operator *

the content of the pointed location

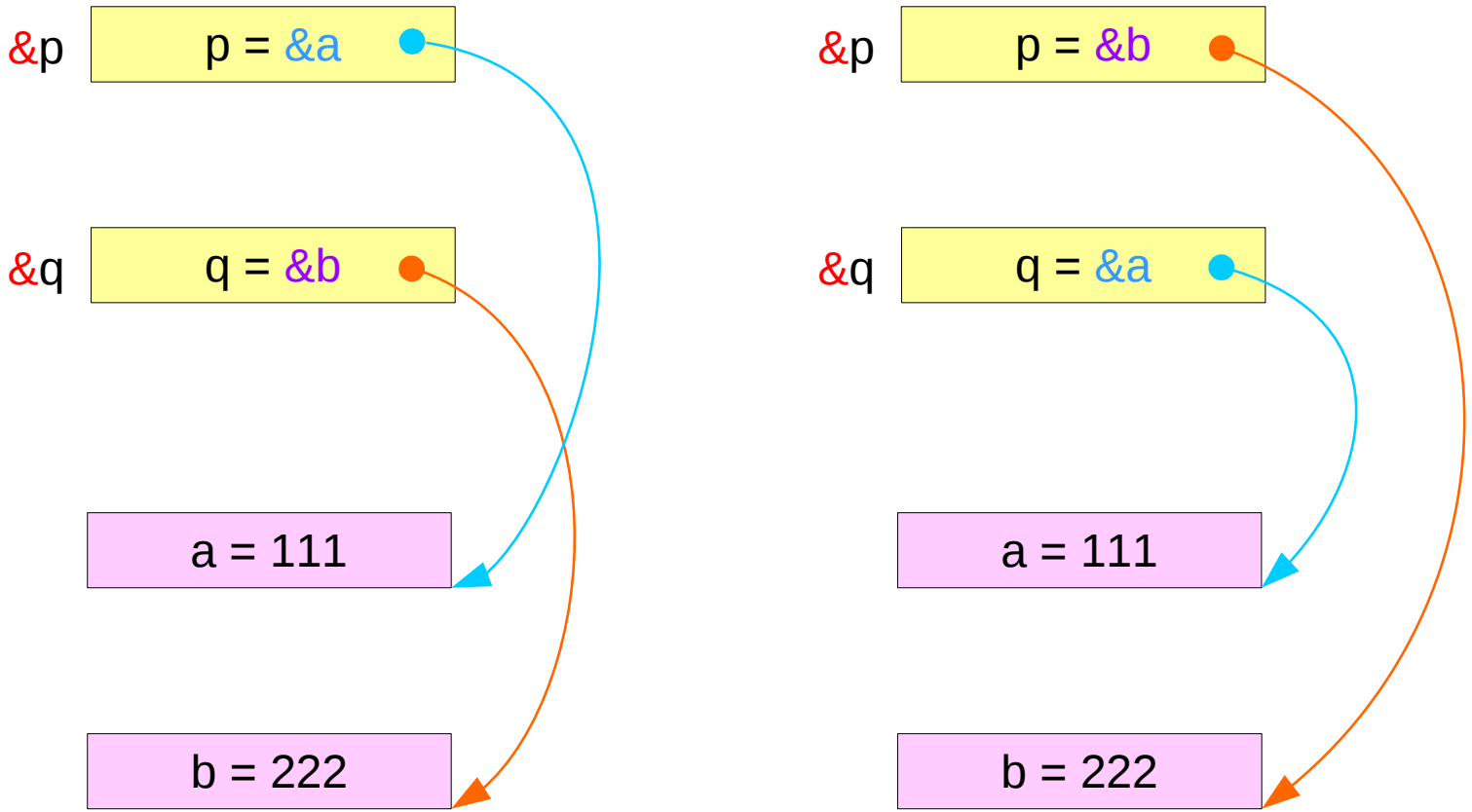
Values of Variables



Swapping pointers

- pass by reference
- double pointers

Swapping integer pointers



Swapping integer pointers


```
int *p, *q;

swap_pointers( &p, &q );
```

function call

```
swap_pointers( int **, int ** );
```

function prototype

Pass by integer pointer reference

```
void swap_pointers (int **m, int **n)
{
      int* tmp;

      tmp = *m;
      *m = *n;
      *n = tmp;
}
```

int **	m
int *	*m

int **	n
int *	*n

int *	tmp

```
int   a,  b;
int *p, *q;          p=&a, q=&b;
      ...
swap_pointers( &p, &q );
```

Array of Pointers

Array of Pointers

```
int     a [4];

int *   b [4];
```

No. of elements = 4

int a [4]

Type of each element

No. of elements = 4

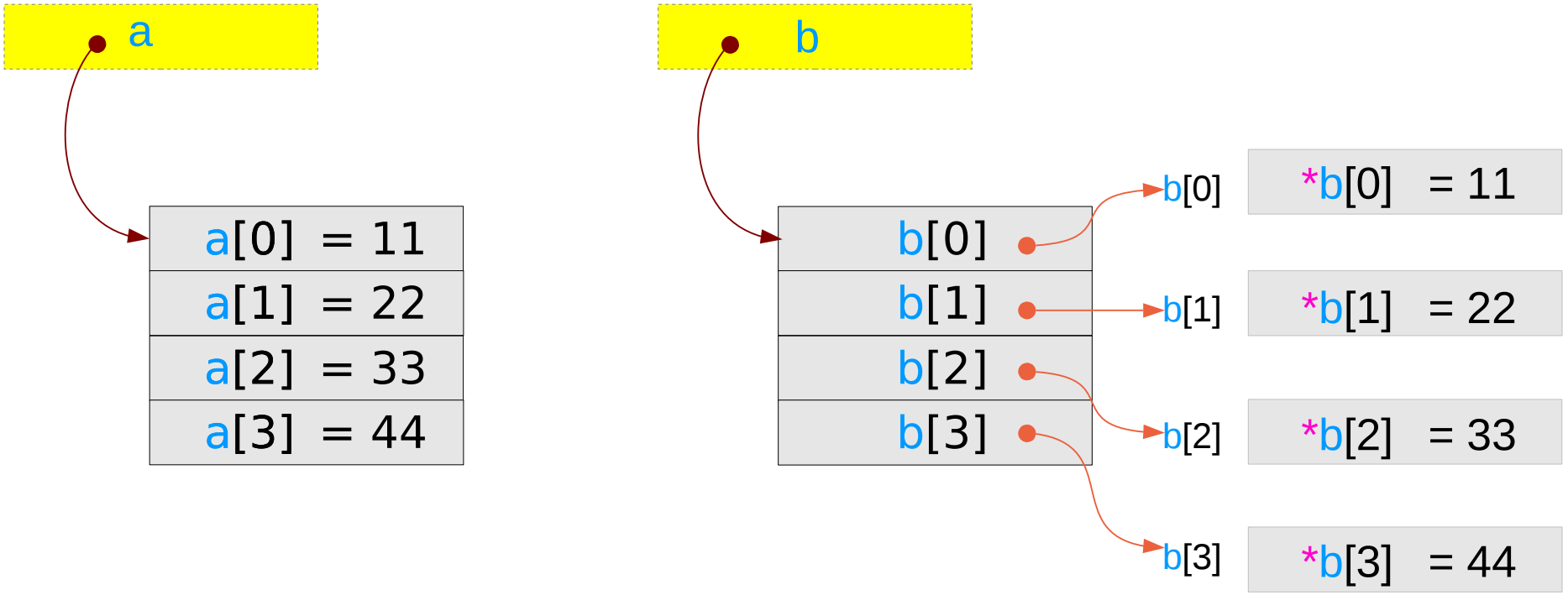
int * b [4]

Type of each element

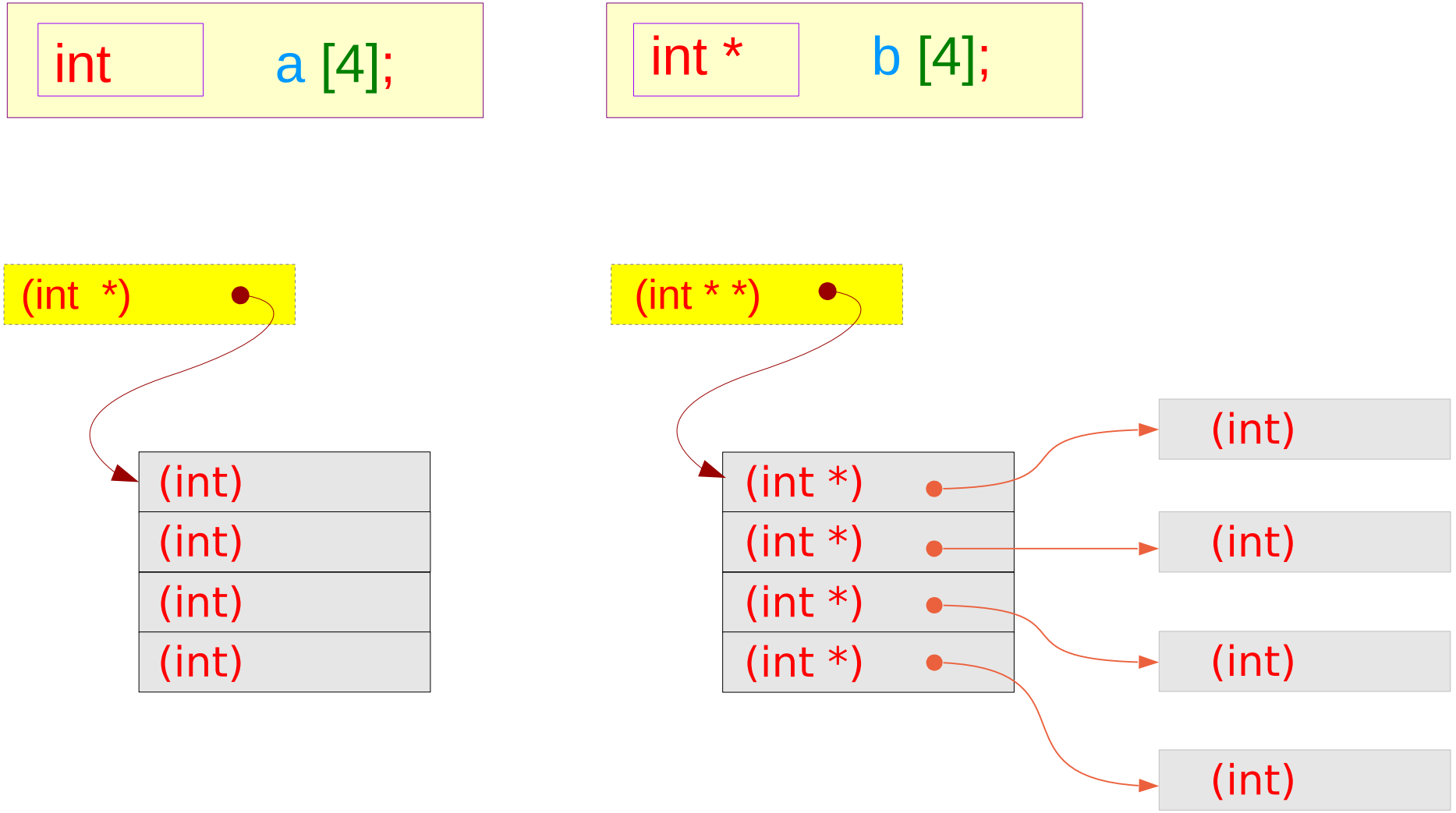
Array of Pointers – variable view

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

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



Array of Pointers – type view



Pointer to Arrays

Pointer to an array – variable declarations

```
int   m ;
int  *n ;
```

an integer pointer

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

an integer array pointer

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

a function pointer

Pointer to an array – a type view

int

int *

an integer pointer

Int [4] ≡ int [ ]

int (*) [4]

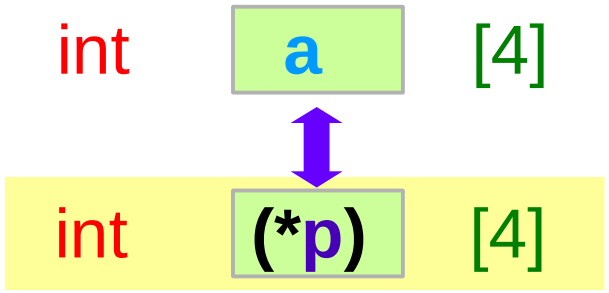
an integer array pointer

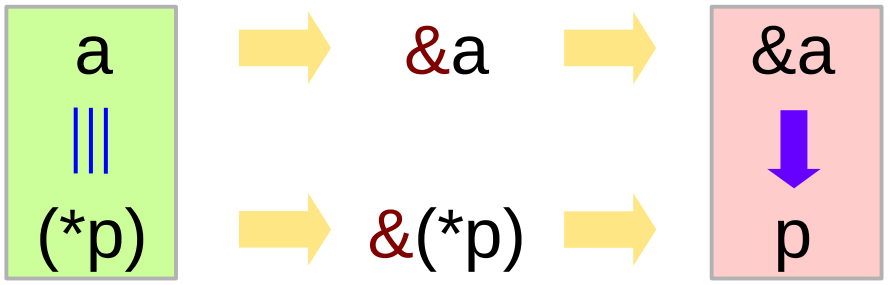
int (int, int)

int (*) (int, int)

a function pointer

Pointer to an Array : Assignment and Dereference





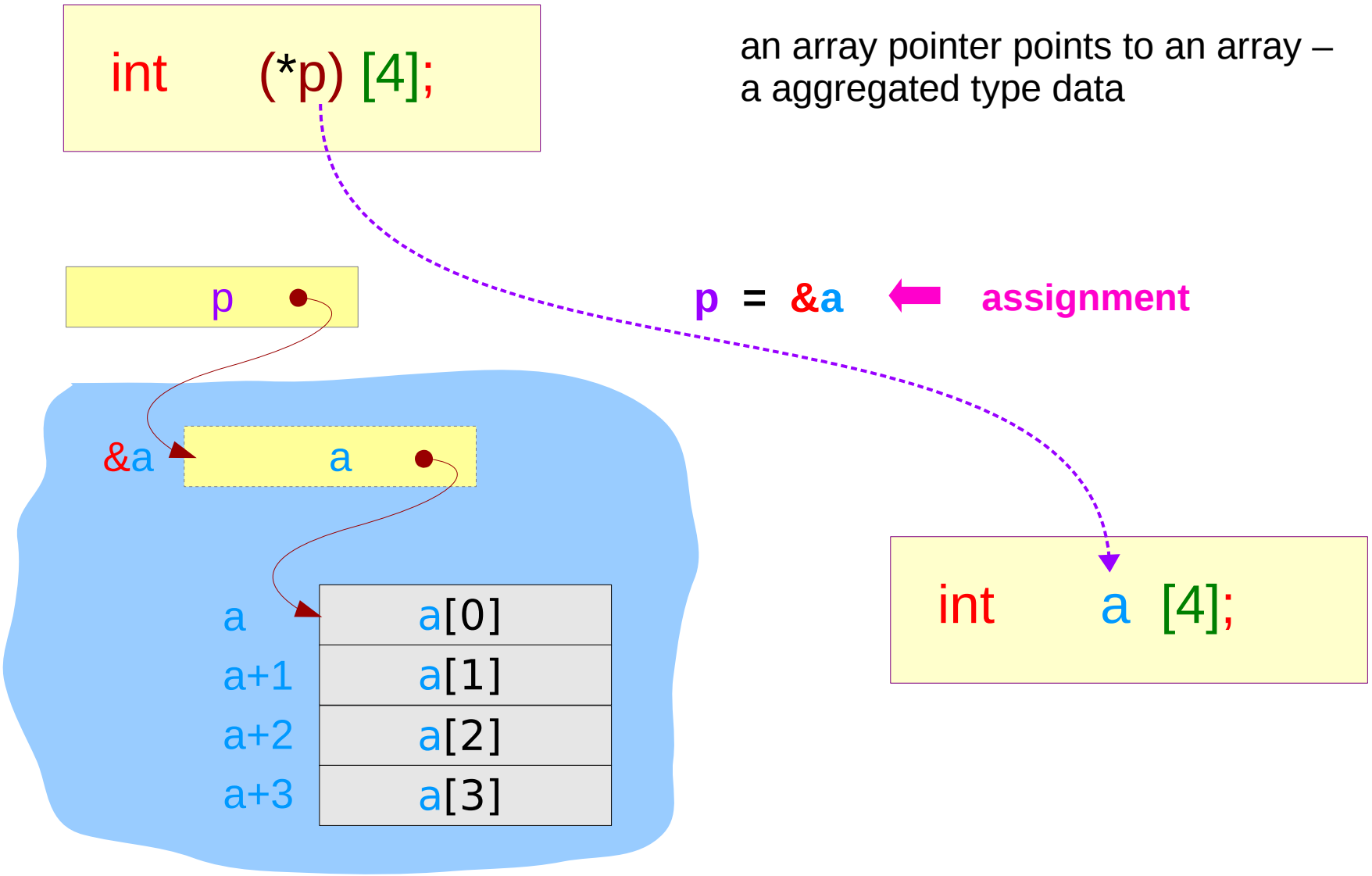
equivalence **assignment**

usages *initialization*

sizeof(p)= 8 bytes : the size of a pointer

sizeof(*p)= 16 bytes : the whole size of the pointed array

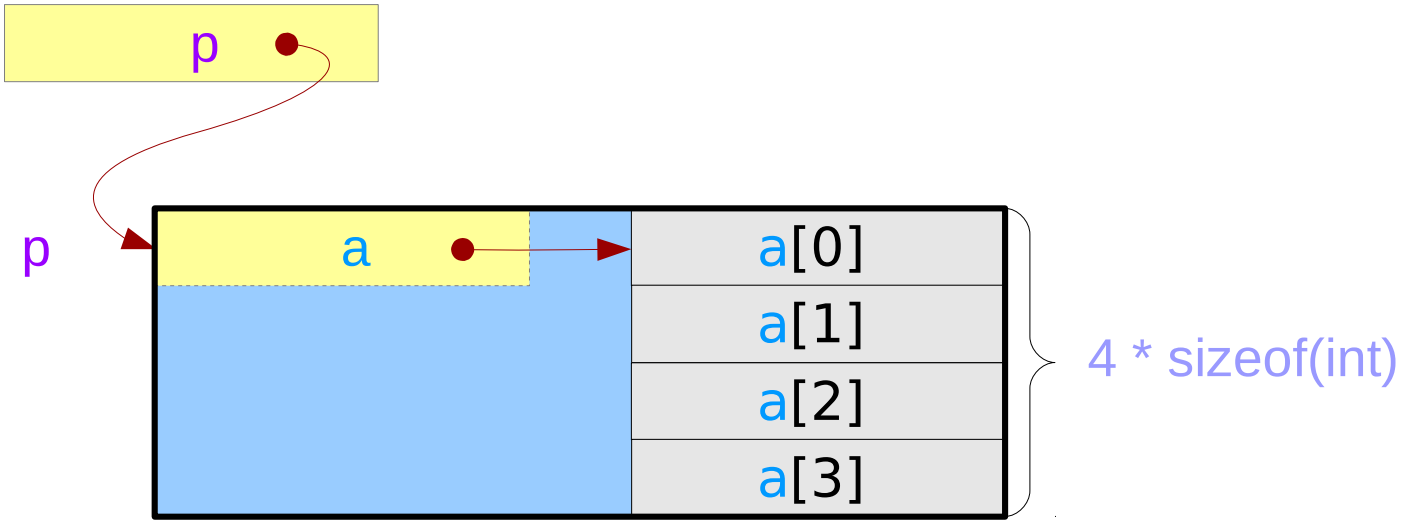
Pointer to an array – a variable view



Pointer to an array – a aggregated type view

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

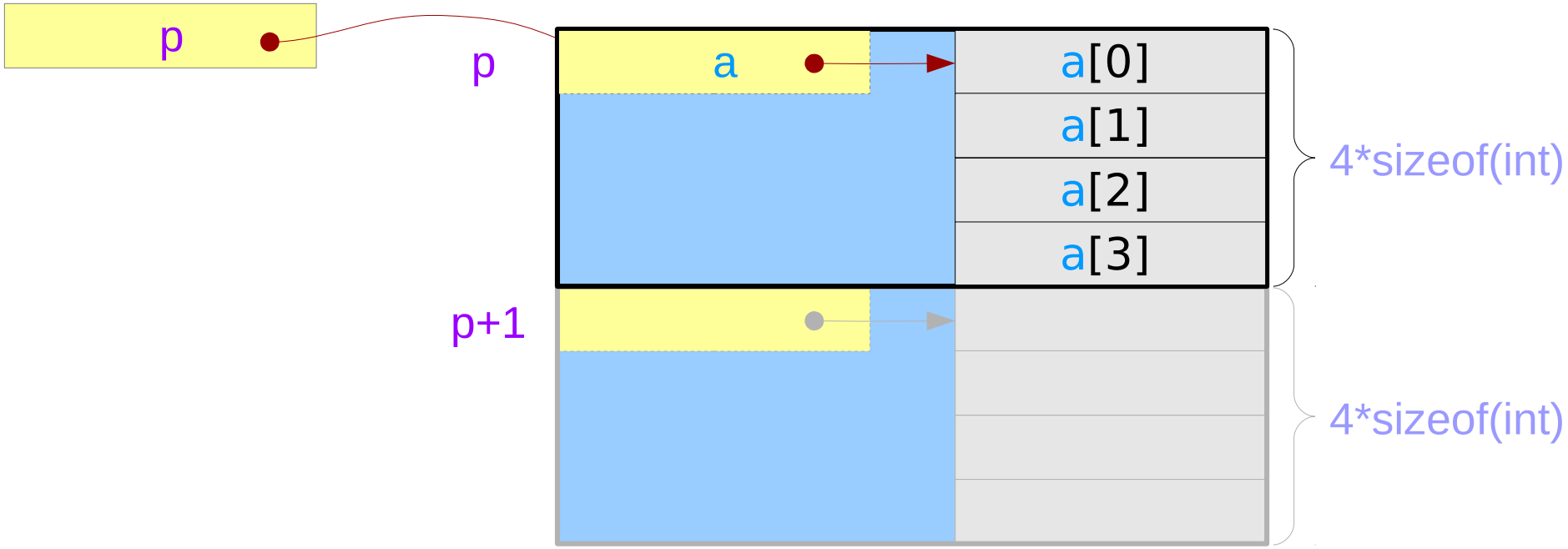
An aggregated type
- starting address (&a)
- size of all the array elements (16 bytes)



Pointer to an array – incrementing a pointer

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

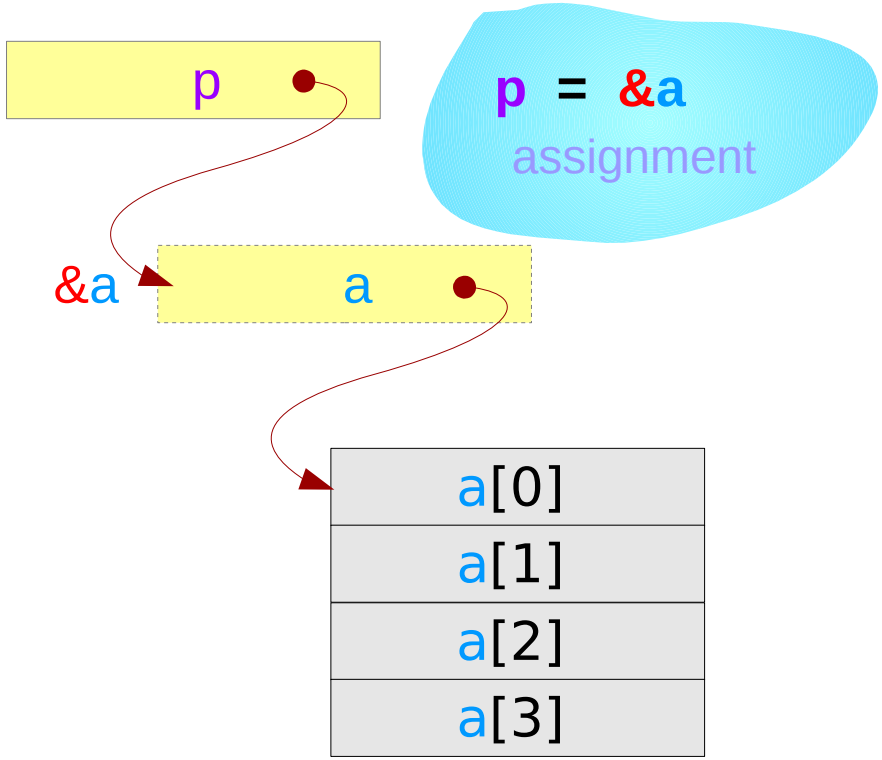
Address value (p+1) – **Address value** (p)
= (long) (p+1) - (long) (p) = 4 * sizeof(int)

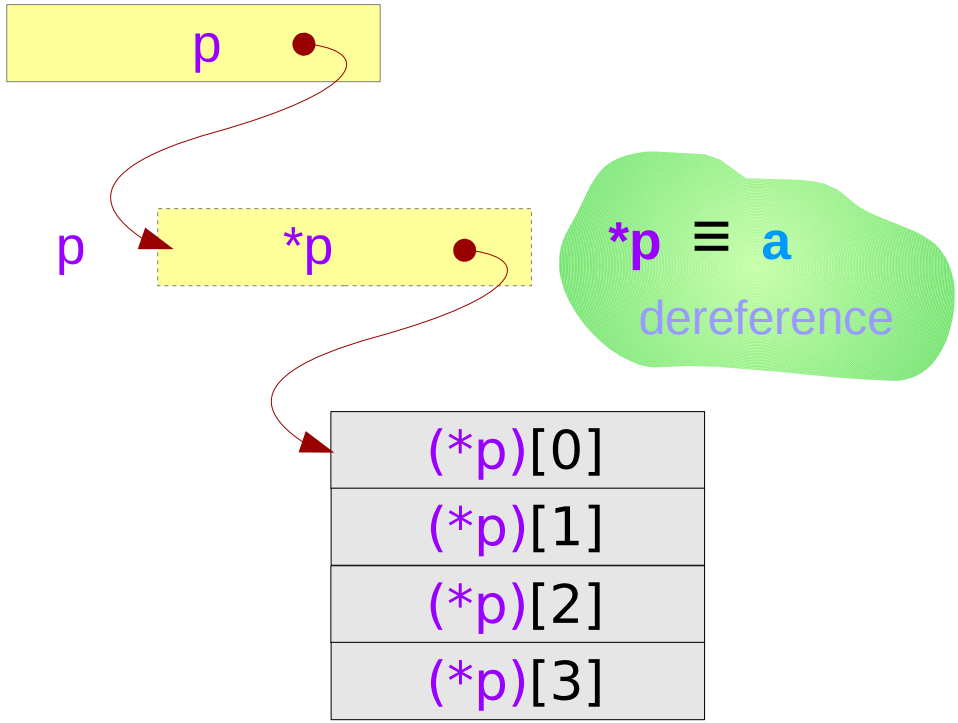


Pointer to an array – dereferencing

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

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

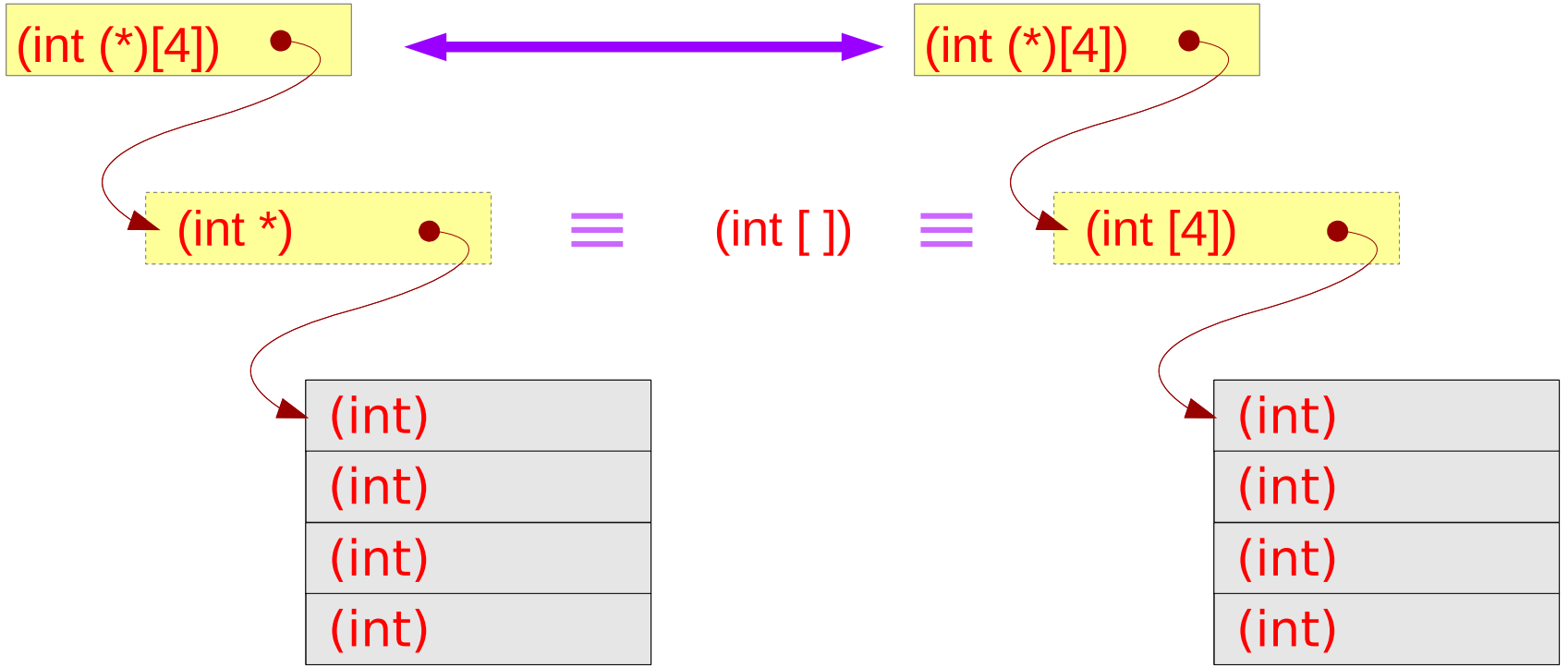




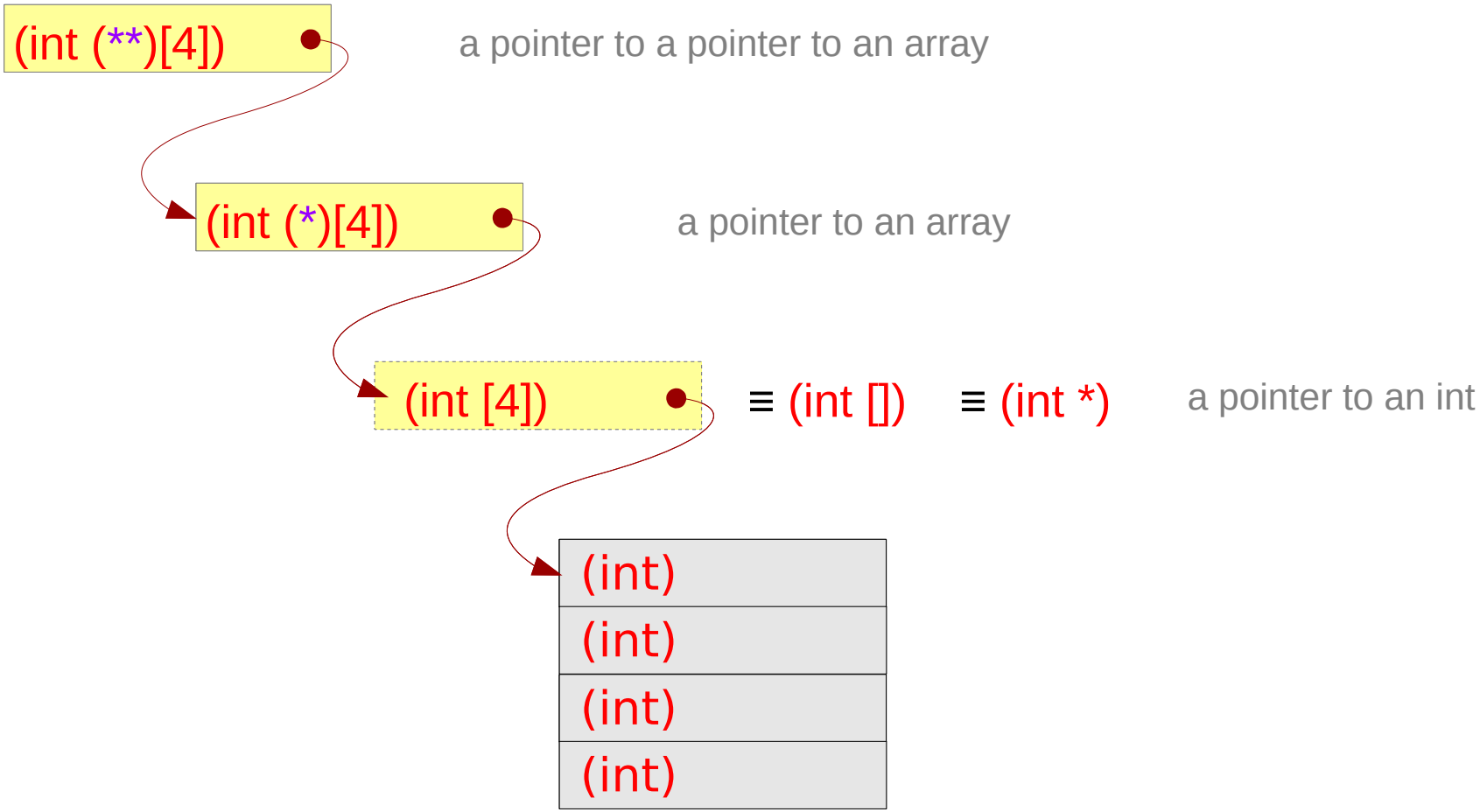
Pointer to an array – a type view

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

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



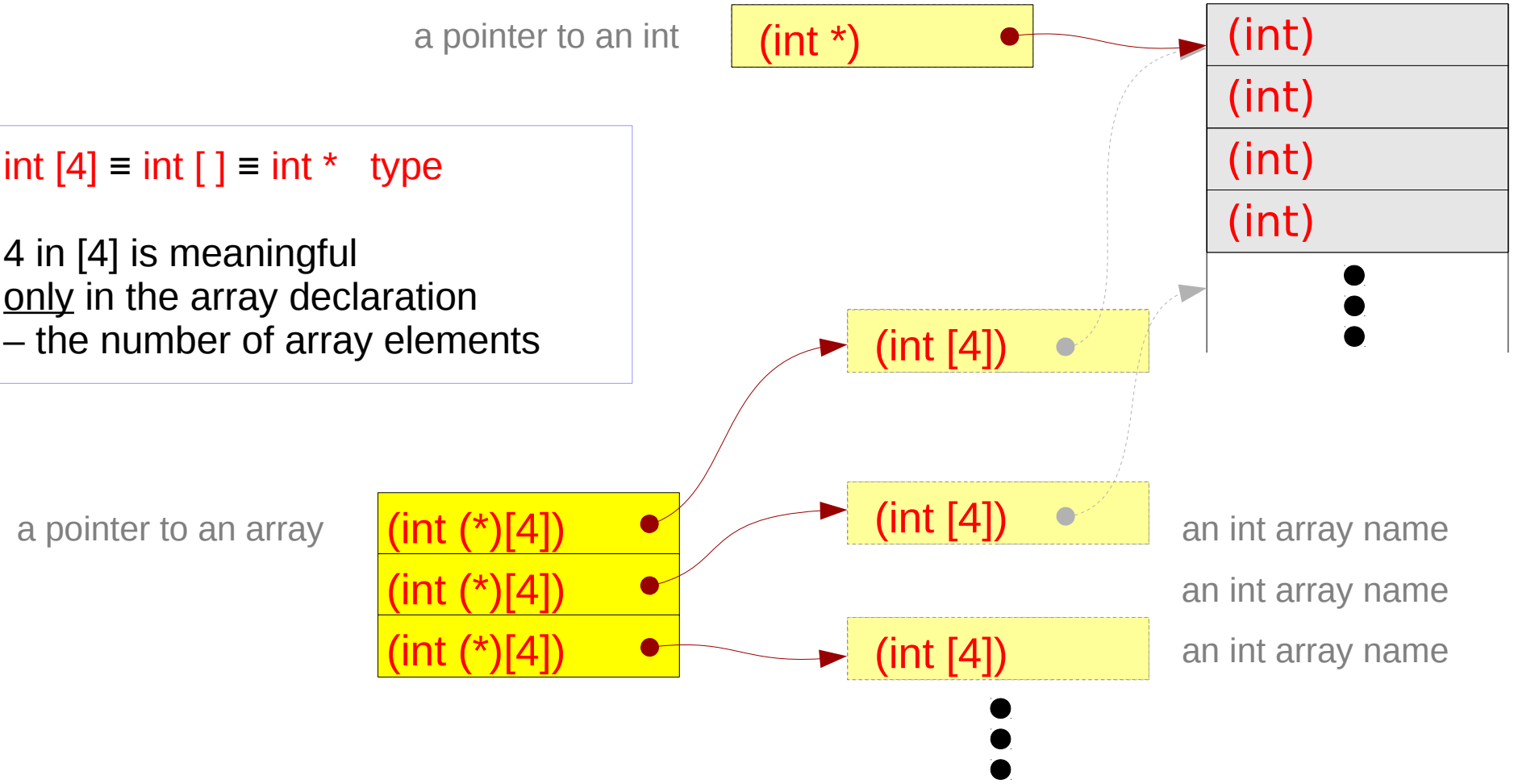
Double pointer to an array – a type view



Series of array pointers – a type view

int [4] ≡ int [ ] ≡ int * type

4 in [4] is meaningful
only in the array declaration
– the number of array elements



Series of array pointers – a variable view

```
int (*p1)[4];
int (*p2)[4];
int (*p3)[4];
```

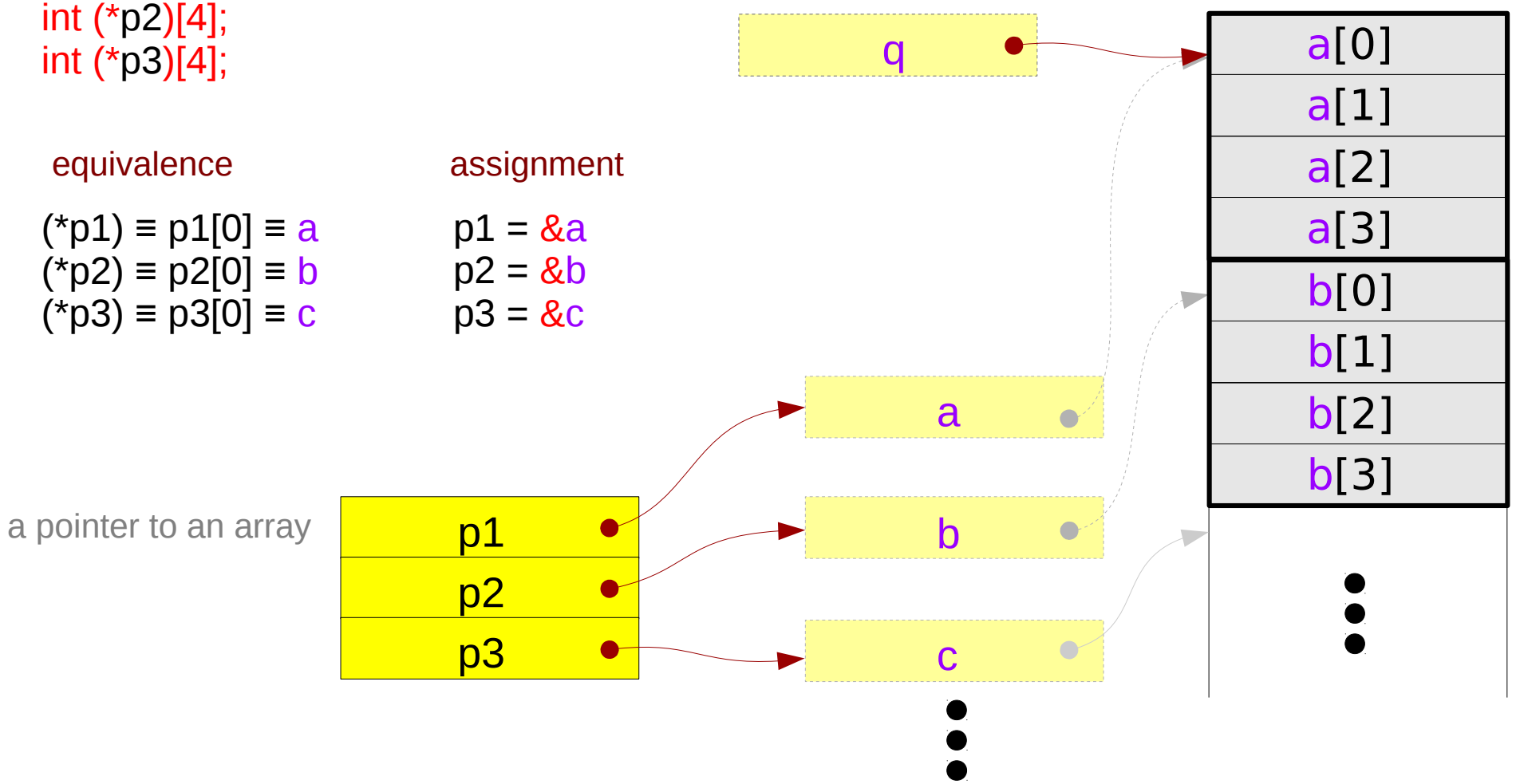
```
int  (*q);
```

equivalence

(*p1) ≡ p1[0] ≡ a
(*p2) ≡ p2[0] ≡ b
(*p3) ≡ p3[0] ≡ c

assignment

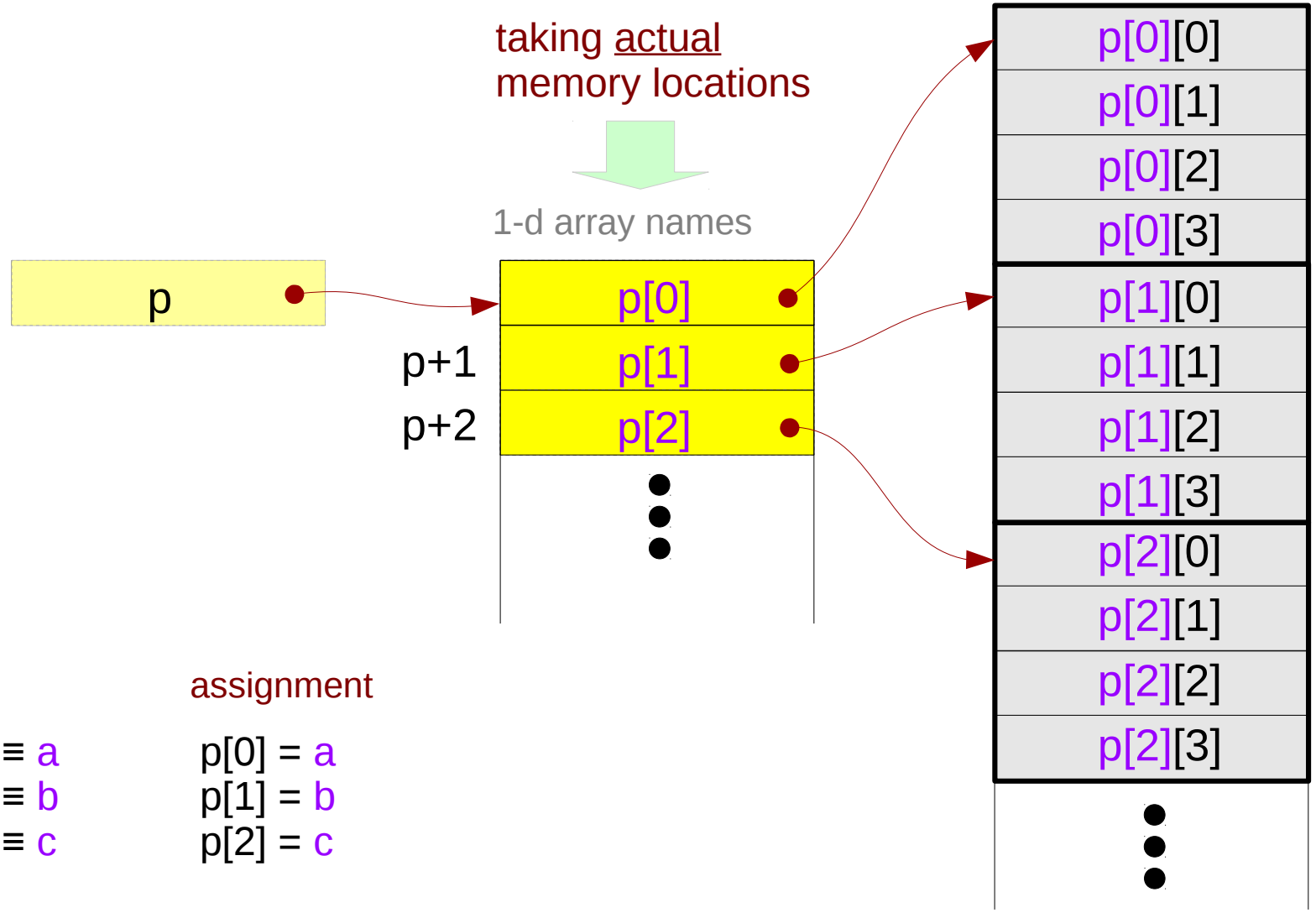
p1 = &a
p2 = &b
p3 = &c



Pointer array – a variable view

int *p[3];

taking actual memory locations



equivalence

p[0] ≡ *(p+0) ≡ a
p[1] ≡ *(p+1) ≡ b
p[2] ≡ *(p+2) ≡ c

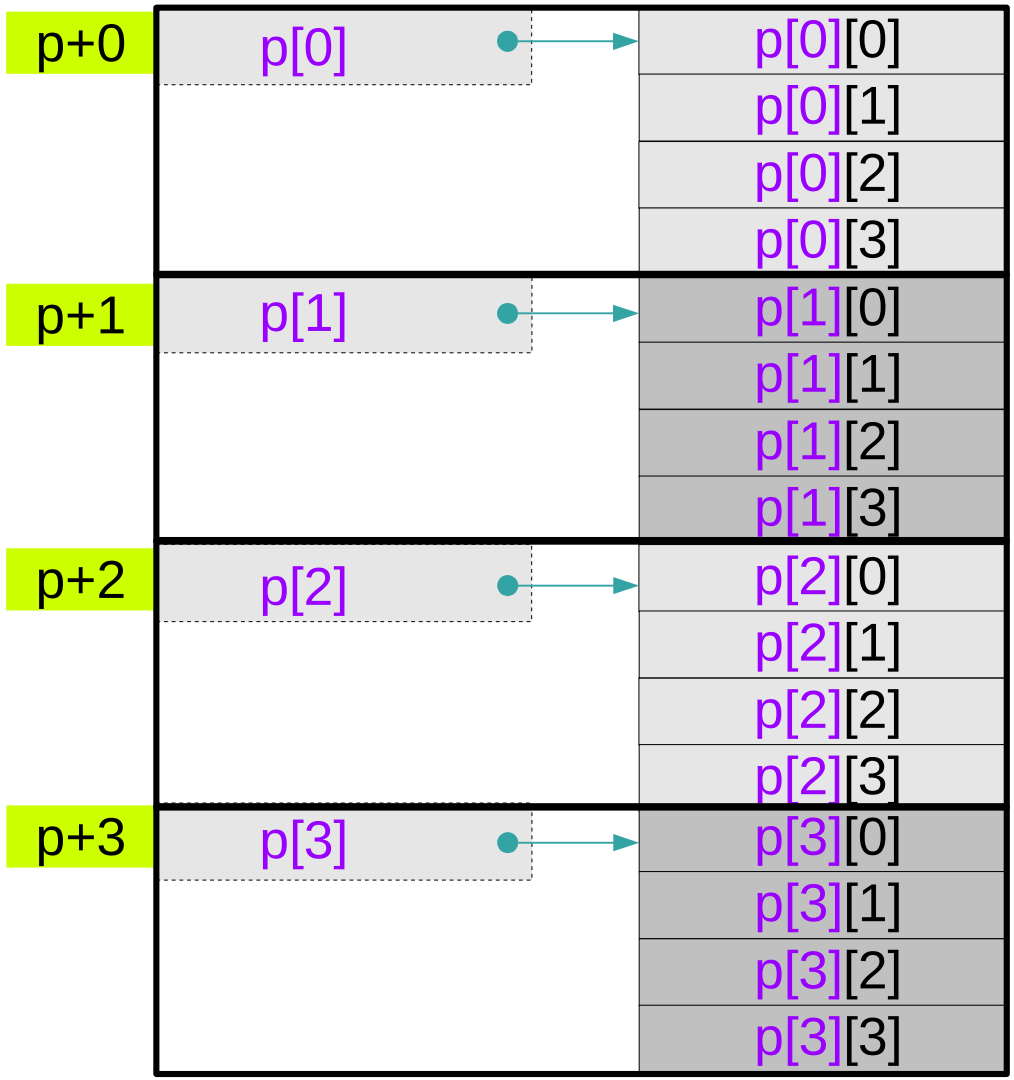
assignment

p[0] = a
p[1] = b
p[2] = c

Pointer to consecutive 1-d arrays

int (*p)[4];

a pointer to an array



equivalence

*(p+0) ≡ p[0] ≡ a
*(p+1) ≡ p[1] ≡ b
*(p+2) ≡ p[2] ≡ c
*(p+2) ≡ p[2] ≡ d

if arrays a, b, c, d
are consecutive

assignment

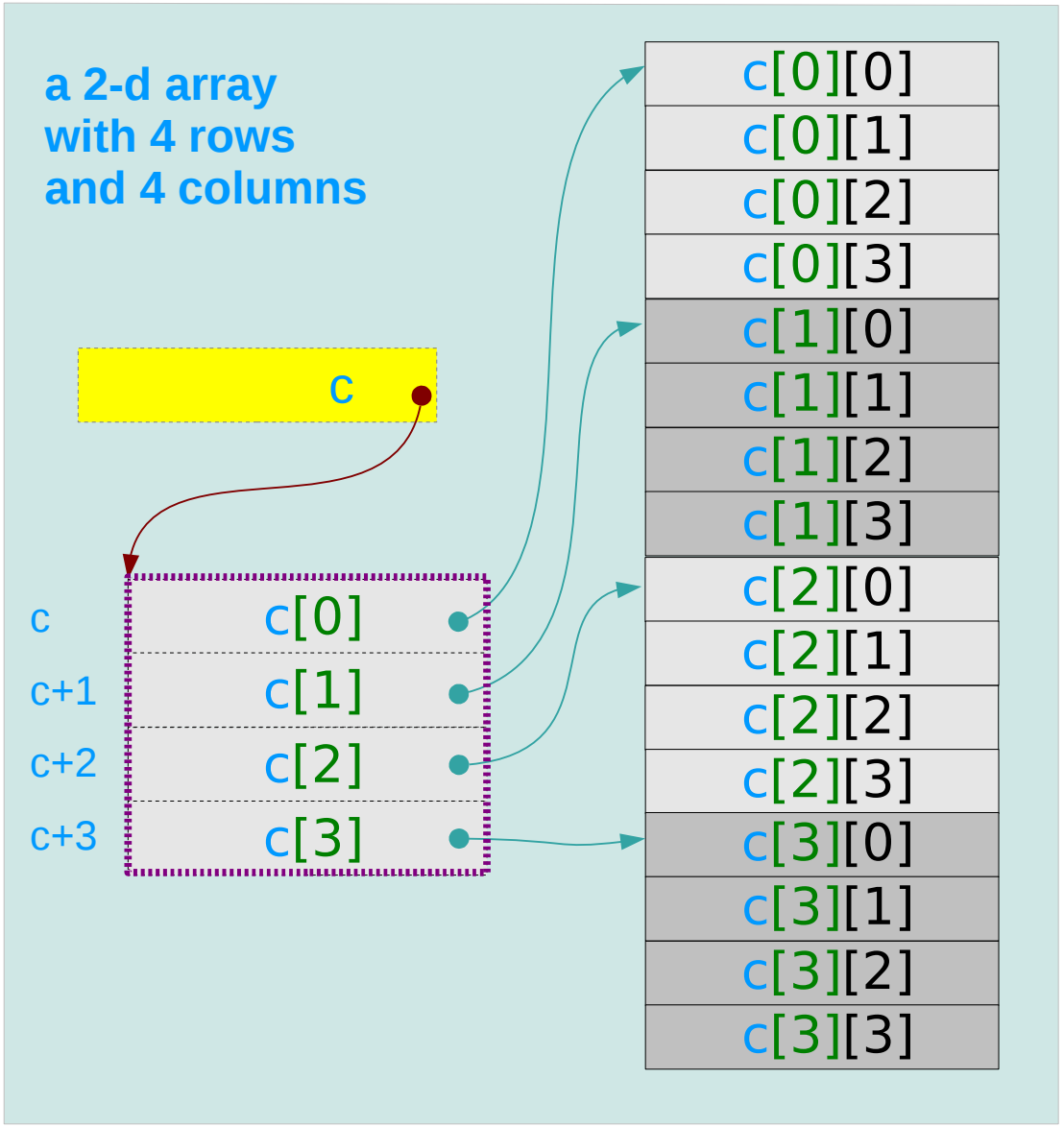
p = &a

A 2-d array and its sub-arrays

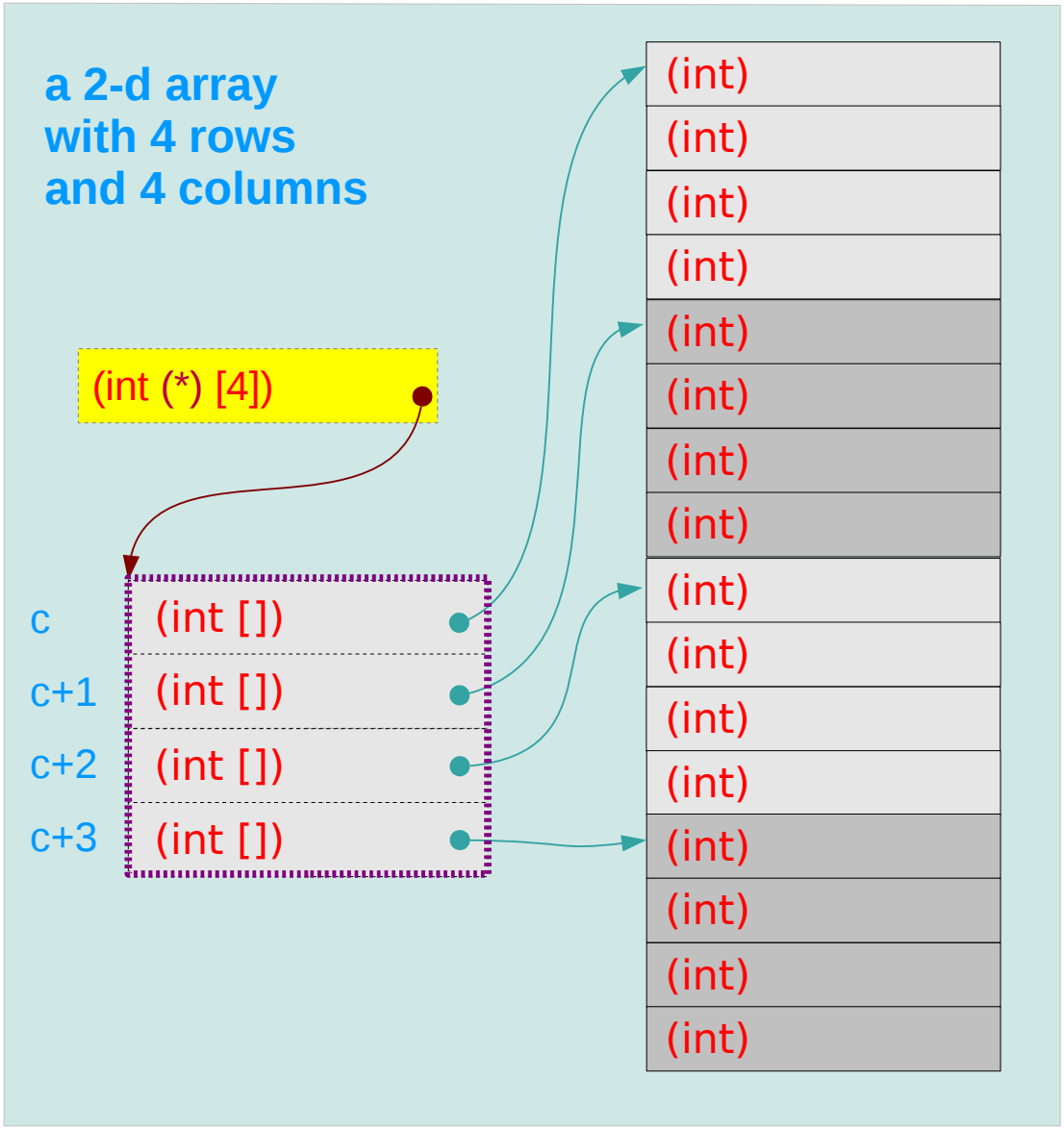
the array name c of a 2-d array as an array pointer which points to its 1st 1-d sub-array of 4 elements.

c[0]	the 1st 1-d sub-array name
c[1]	the 2nd 1-d sub-array name
c[2]	the 3rd 1-d sub-array name
c[3]	the 4th 1-d sub-array name

c[0], c[1], c[2], c[3] can be implemented without taking actual memory locations

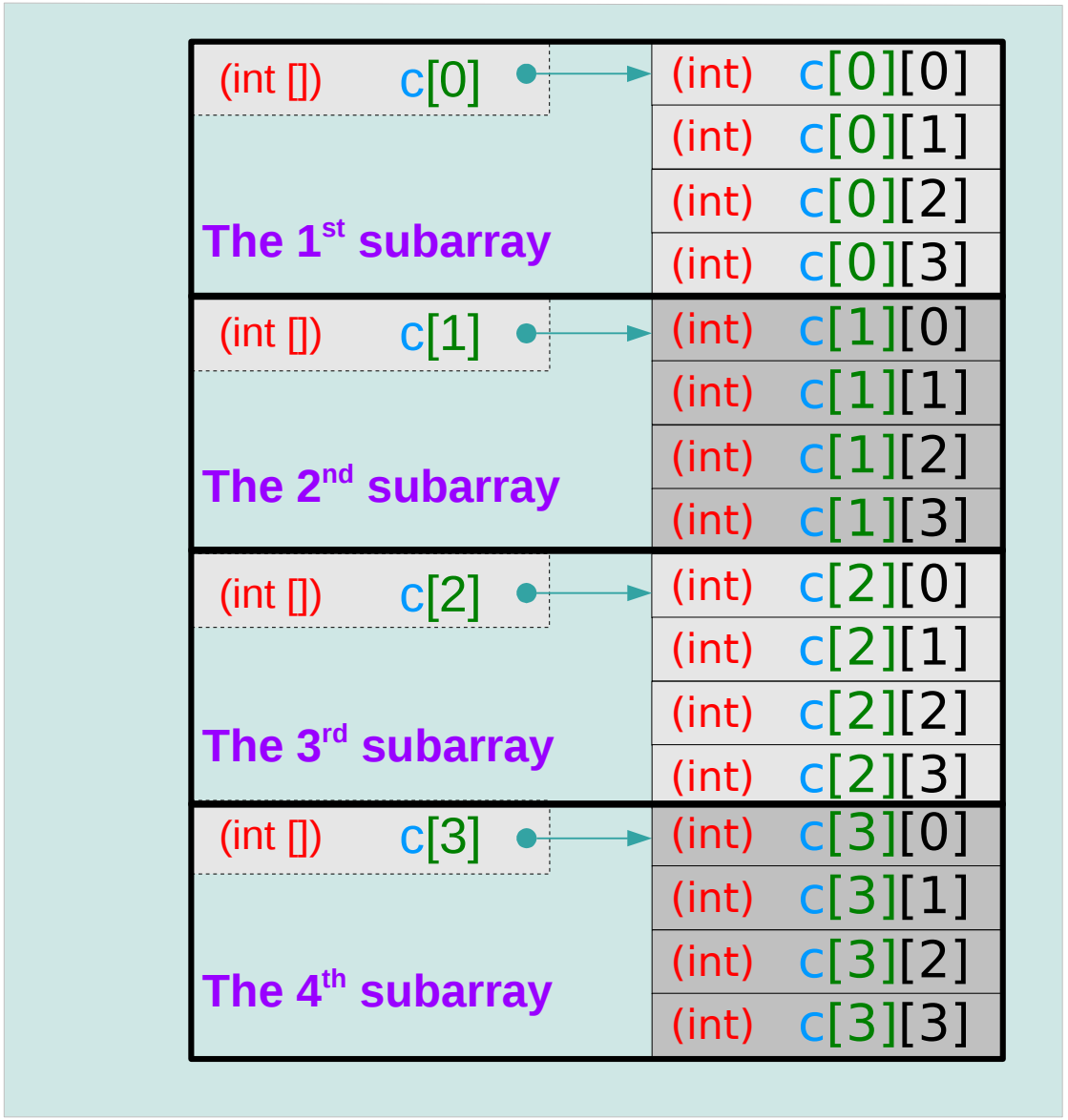


A 2-d array and its sub-arrays – a type view



1-d subarray aggregated data type

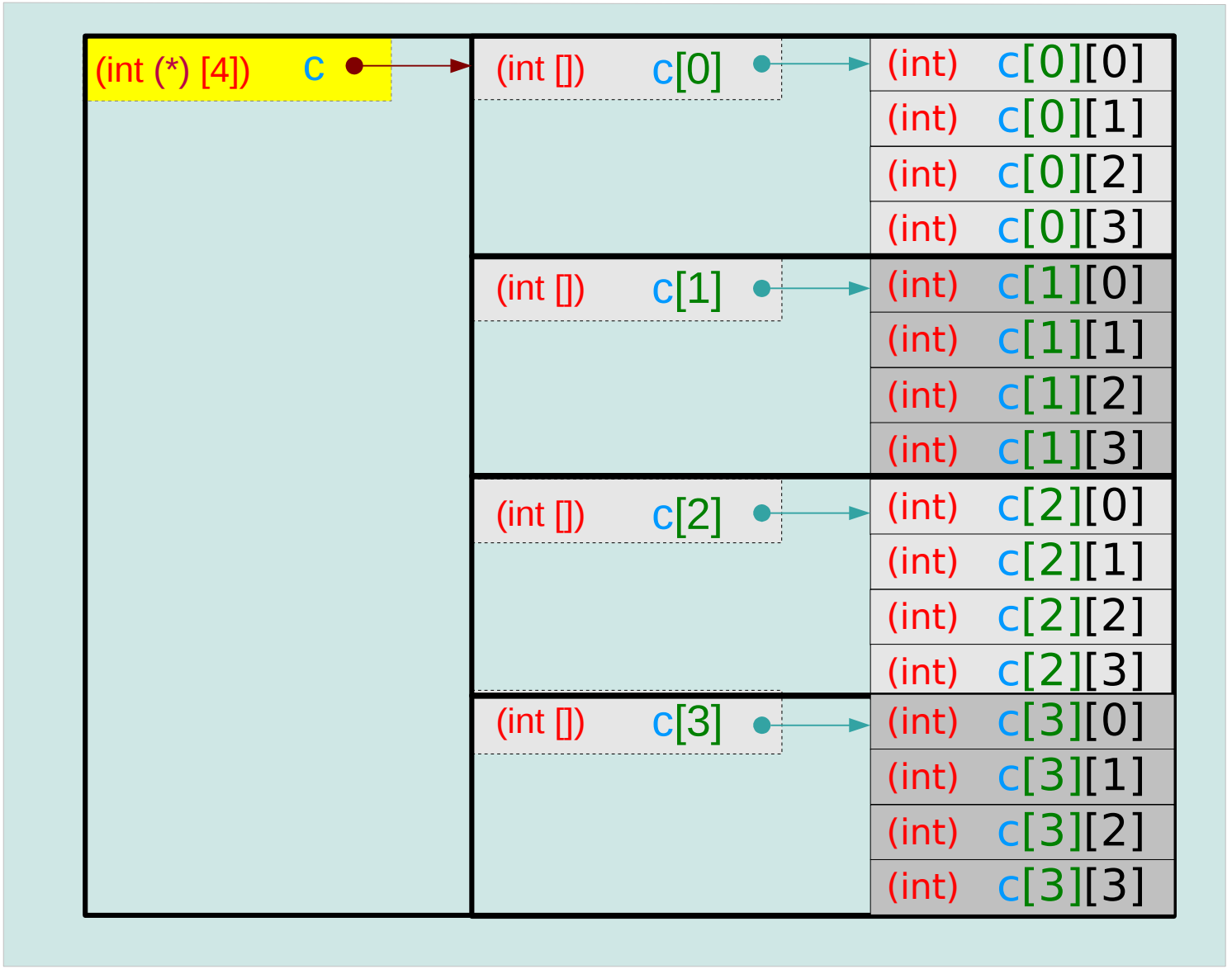
sizeof(c[0]) = 16 bytes
sizeof(c[1]) = 16 bytes
sizeof(c[2]) = 16 bytes
sizeof(c[3]) = 16 bytes



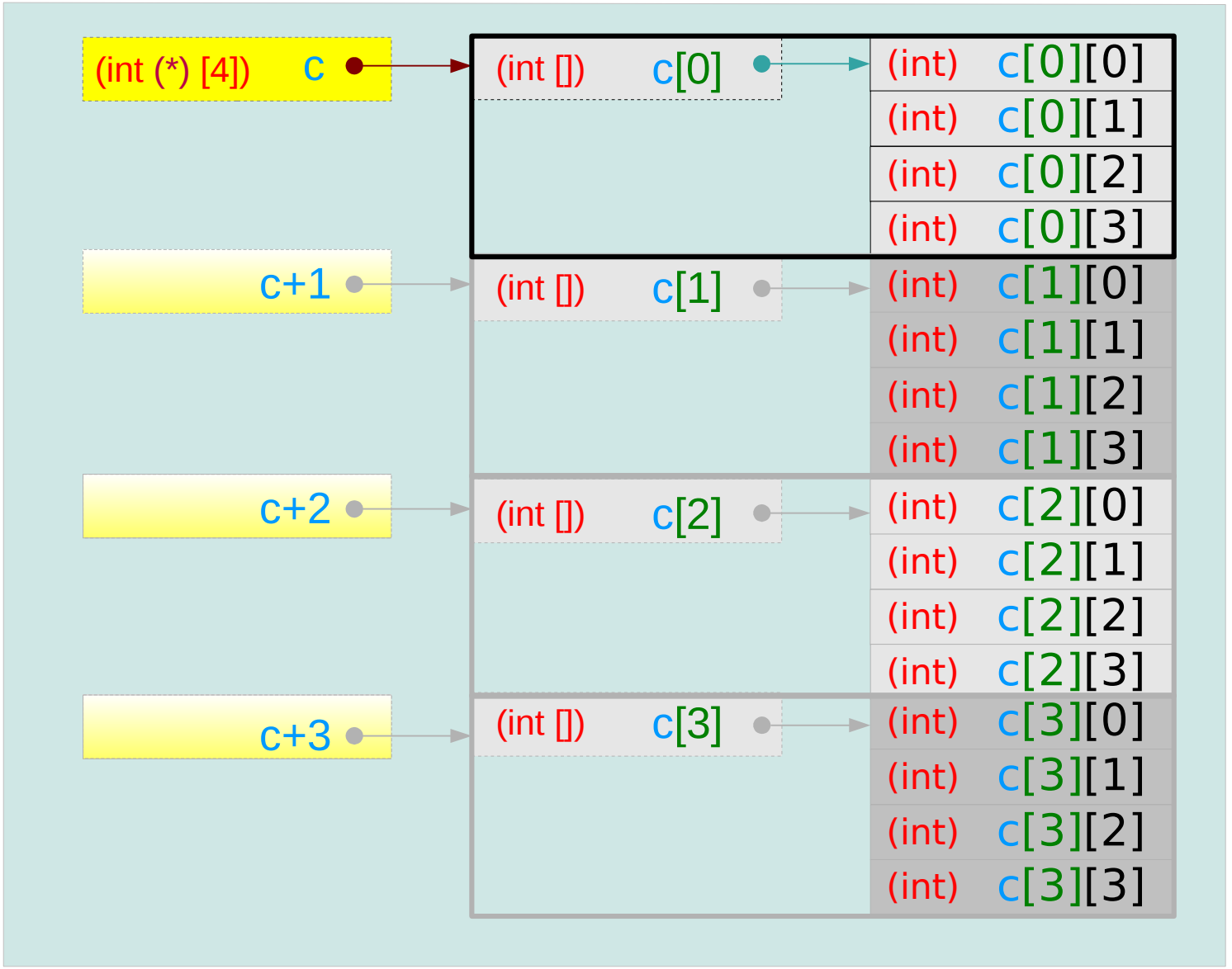
2-d subarray aggregated data type

2-d array :
sizeof(c) = 64 bytes

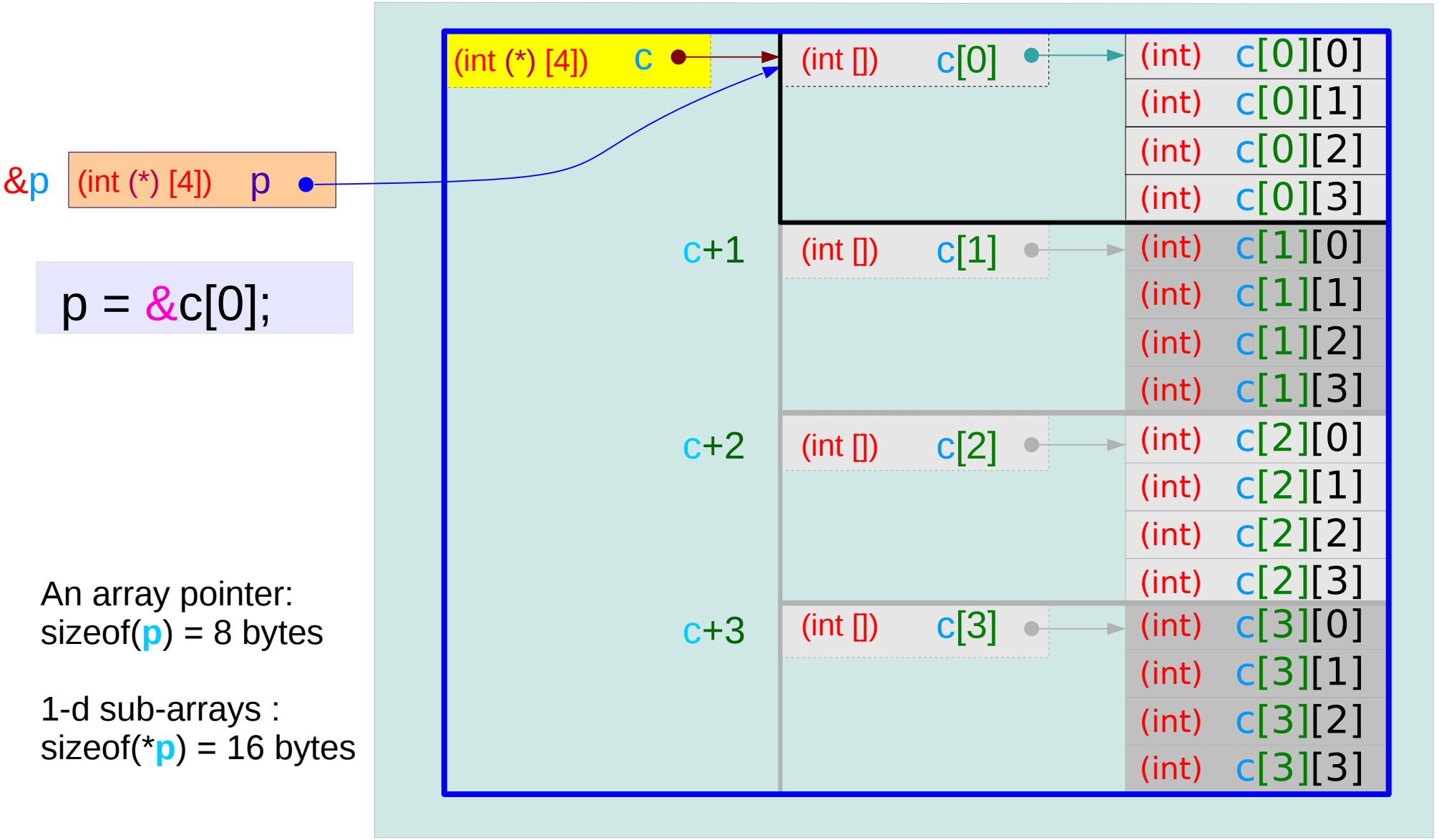
1-d sub-arrays :
sizeof(*c) = 16 bytes



2-d array name as a pointer to a 1-d subarray



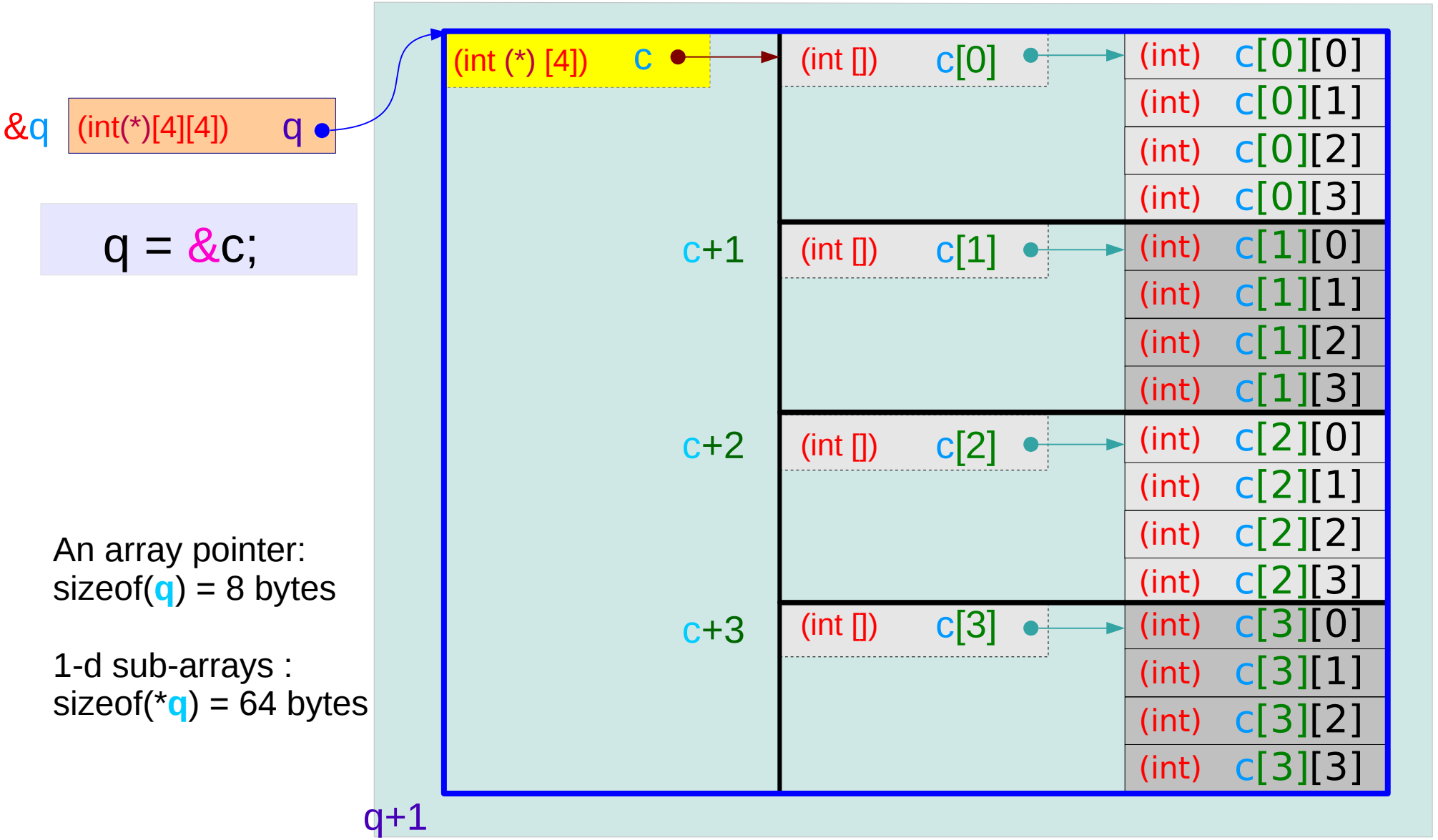
Pointer variable to a 1-d array



An array pointer:
sizeof(p) = 8 bytes

1-d sub-arrays :
sizeof(*p) = 16 bytes

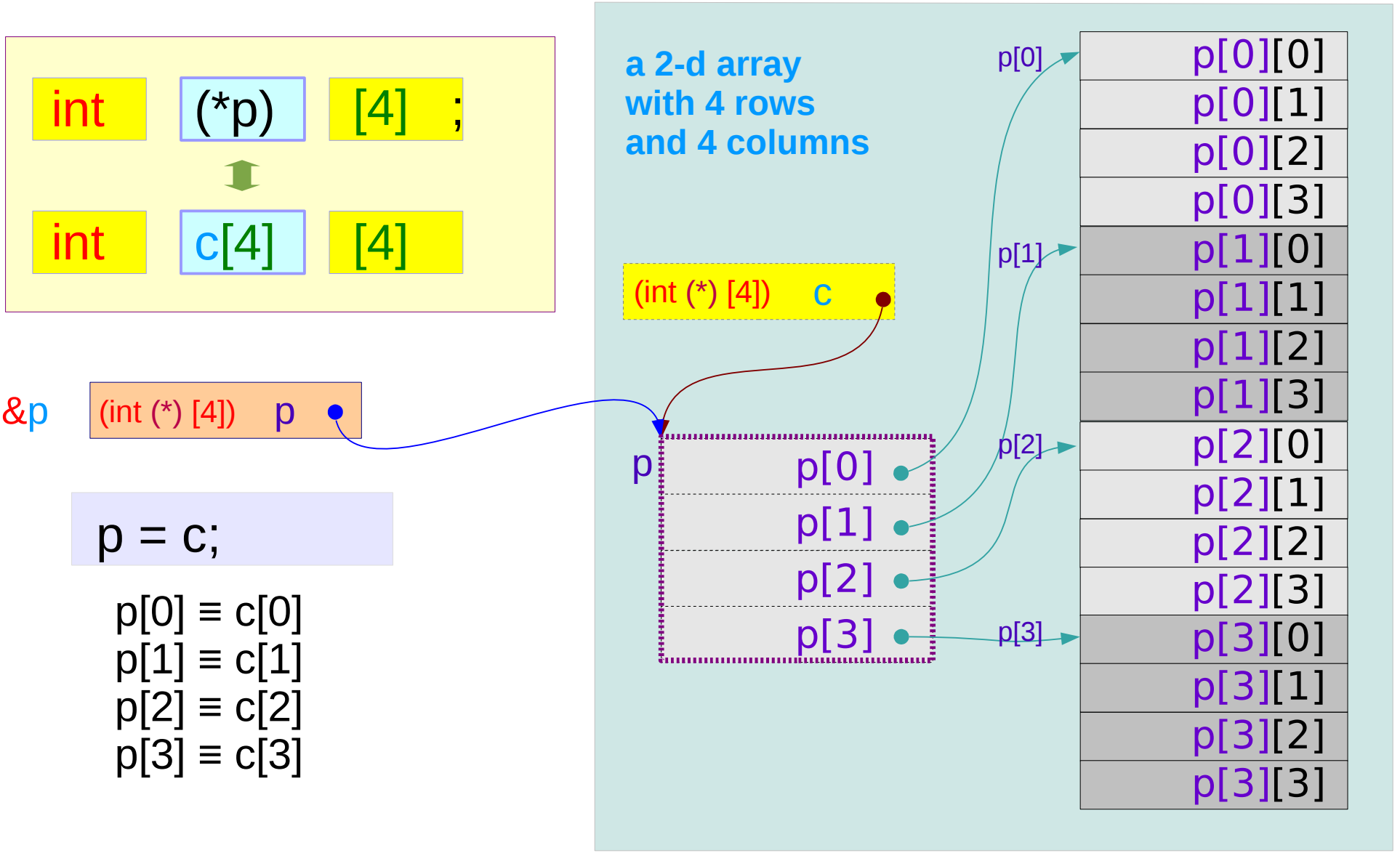
Pointer variable to a 2-d array



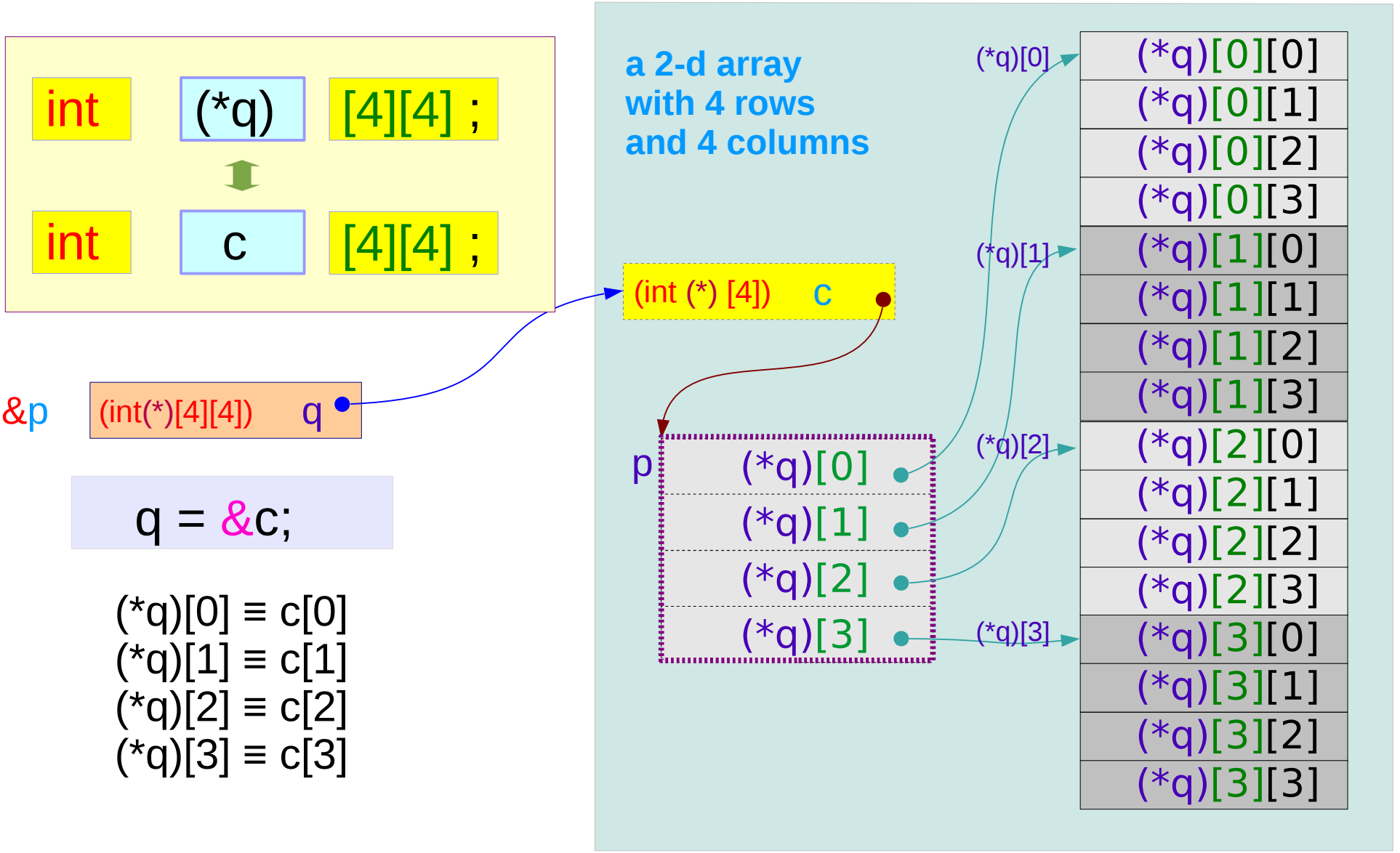
An array pointer:
sizeof(**q**) = 8 bytes

1-d sub-arrays :
sizeof(***q**) = 64 bytes

Using a a pointer to a 1-d array



Using a pointer to a 2-d array



Pointer to array arguments

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

    p = c;

    func(p, ... );
```

```
void func(int (*x)[4], ... )
{

    x[r][c] =

}
```

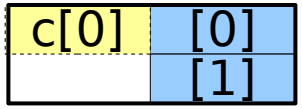
```
int x[ ] [4];
int (*x) [4];

    p = c;

    func(p, ... );
```

```
void func(int x[ ][4], ... )
{

    x[r][c] =

}
```

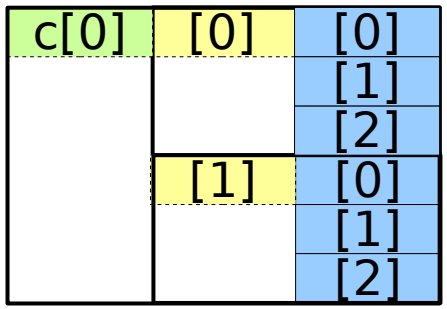
Pointer to array arguments

```
int a[4] [2];
int (*p) [2];                    p = a;

int b[4] [2][3];
int (*q) [2][3];                 q = b;

int c[4] [2][3][4];
int (*r)  [2][3][4];             r = c;
```

Pointer to array arguments

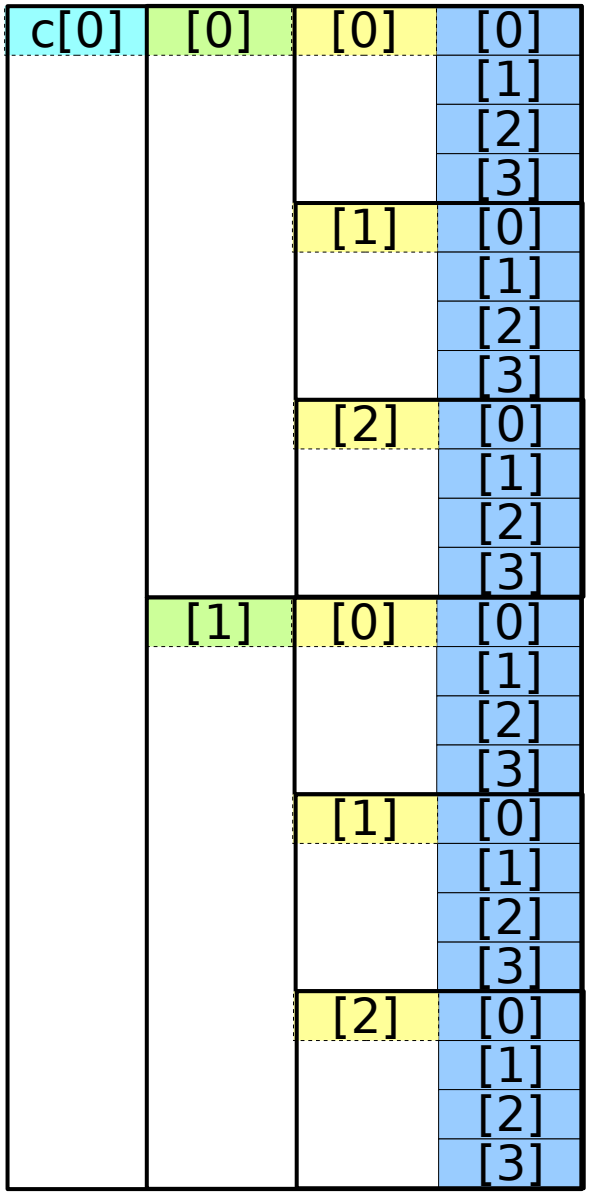



```
int a[4] [2];
int (*p) [2];
```




```
int b[4] [2][3];
int (*q) [2][3];
```

```
int c[4] [2][3][4];
int (*r)  [2][3][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