Applications of Pointers (1A)

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

This document was produced by using LibreOffice.

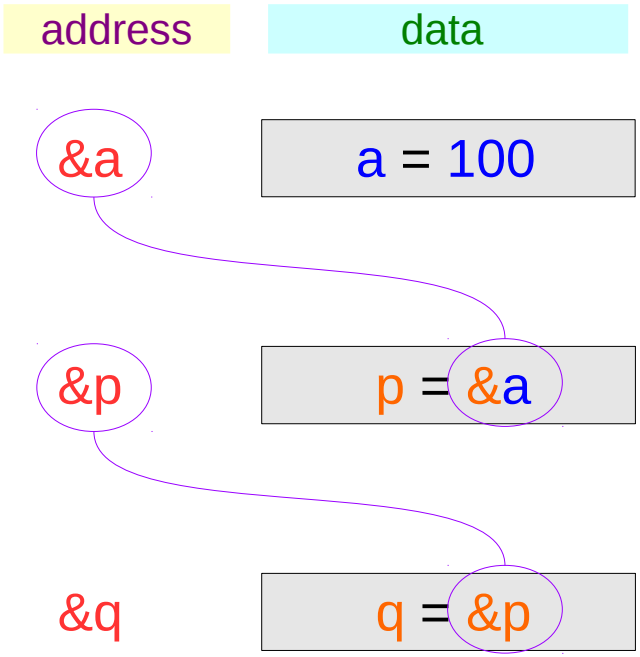
Double Pointers

Variables and their addresses

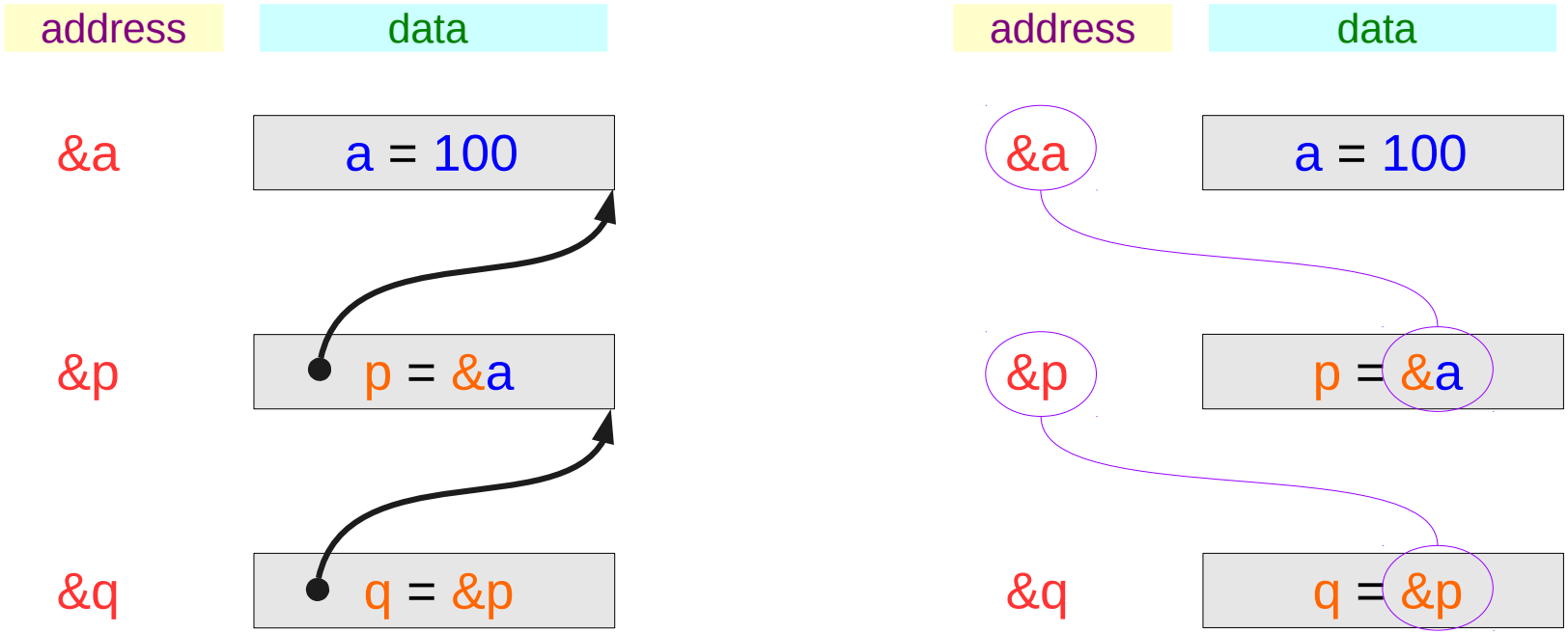
	address	data
int a;	&a	a
int * p;	&p	p
int ** q;	&q	q

Initialization of Variables

```
int     a = 100;

int   * p = &a;

int  ** q = &p;
```



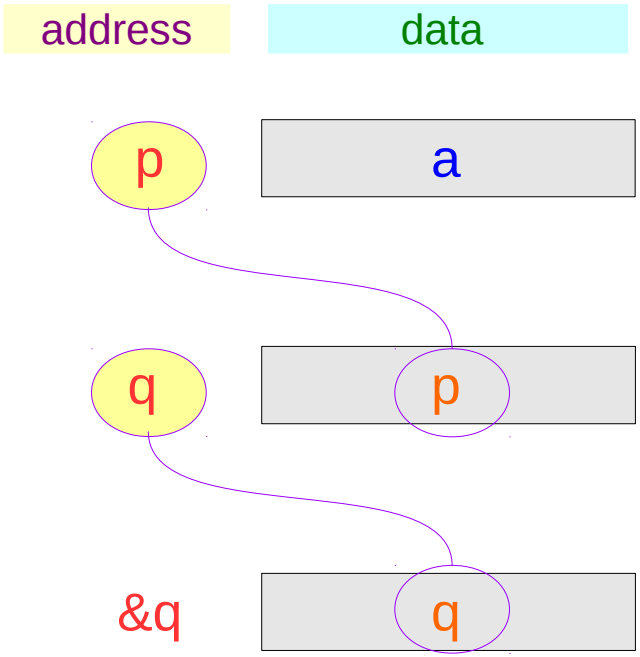
Traditional arrow notations



LSB, little endian

Pointed addresses : p, q

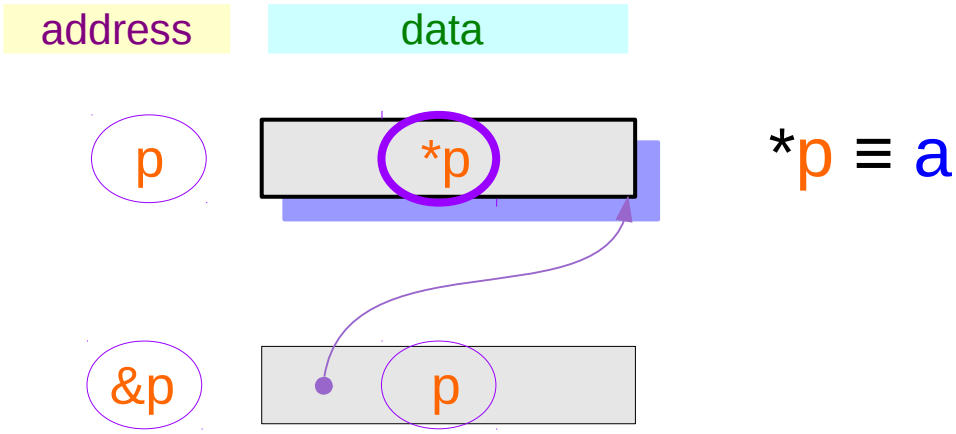
```
int      a;

int   *  p = &a;

int   ** q = &p;
```



p = &a
q = &p

A dereferenced variable : *p

```
int     a;

int   * p = &a;
```



An aliased variable : *p

```
int     a;

int  *  p = &a;
```

Address assignment

Variable aliasing

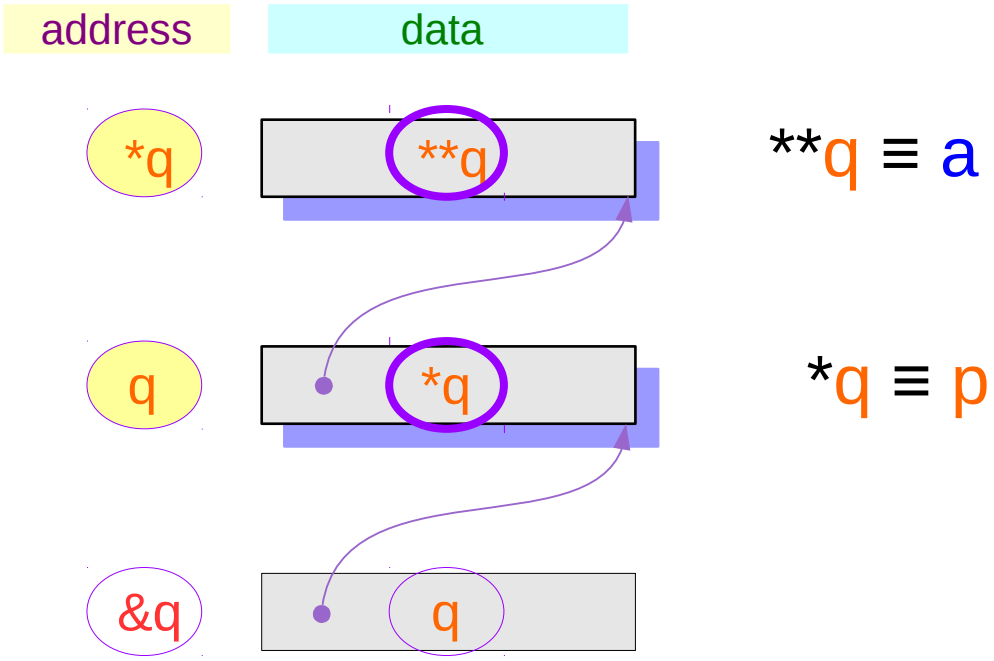
p = &a ➡ *p ≡ a

p	≡	&a
*(p)	≡	*(&a)
* p	≡	a

equivalent relations after address assignment

Dereferenced variables : *q, **q

```
int     a;

int   * p = &a;

int   ** q = &p;
```



**q ≡ a

*q ≡ p

Aliased variables : *q, **q

```
int      a;

int   *  p = &a;

int   ** q = &p;
```

Address assignment

Variable aliasing

p = &a ➡ *p ≡ a

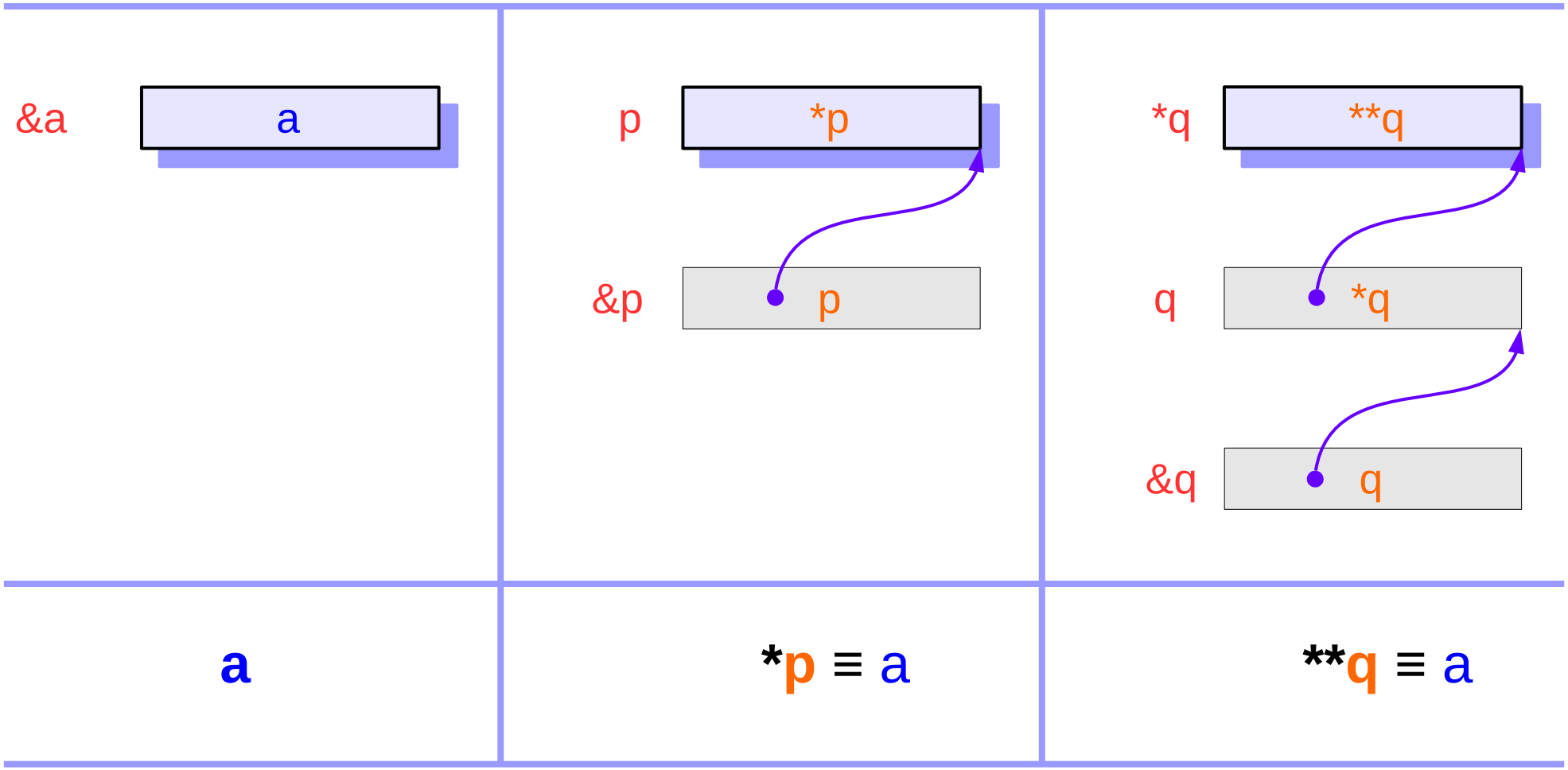
q = &p ➡ *q ≡ p

➡ **q ≡ a

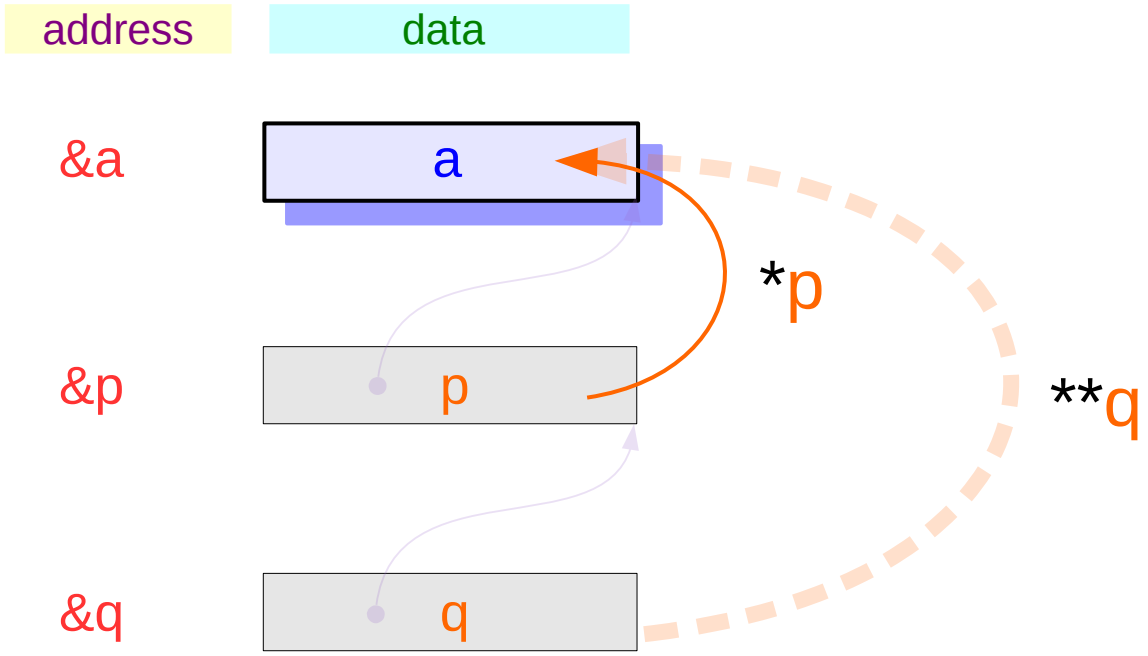
```
  q   ≡   &p
*(q)  ≡ *(&p)
* q   ≡    p
**q   ≡   *p
**q   ≡    a
```

equivalent relations after address assignment

Two aliased variables of **a** : ***p**, ****q**



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



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

Variable Definitions

```
int     a;
```

a can hold an *integer*

address	data
&a	a

```
a = 100;
```

a holds 100

address	data
&a	a ⬅ 100

Pointer Variable Definition

```
int *       p;
```

p can hold an ***address***

```
int   *   p;
```

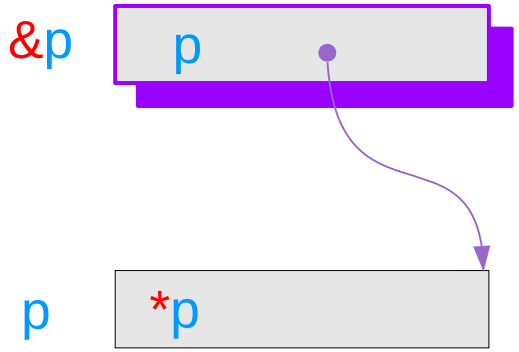
pointer to int

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

```
int   *   p;
```

int

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



Double Pointer Variable Definition

```
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;
```

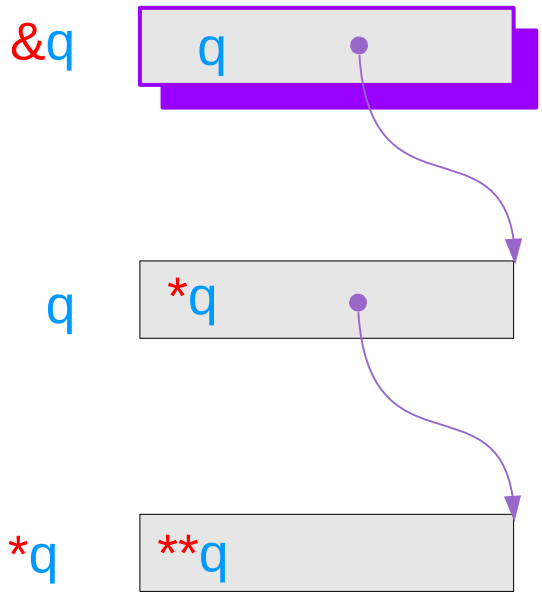
pointer to int

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

```
int         **q;
```

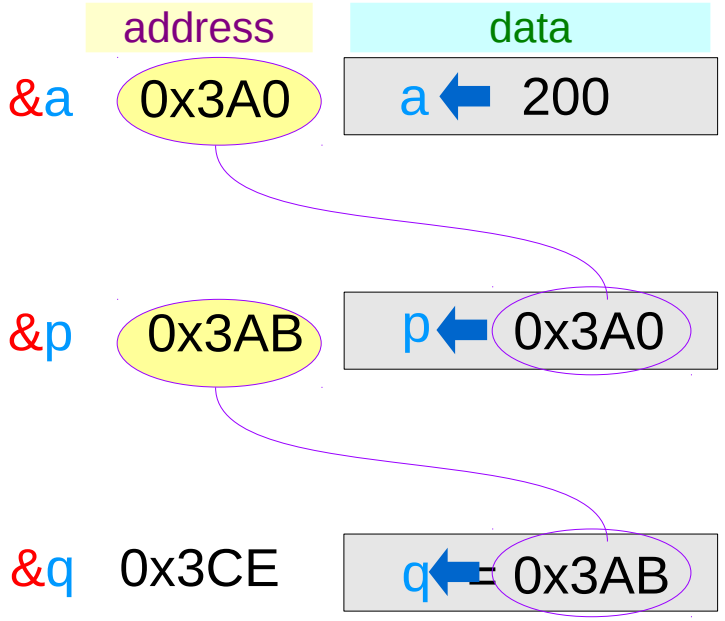
int

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



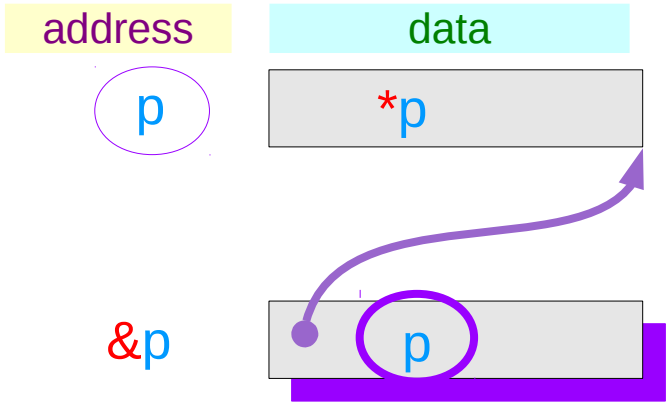
Pointer Variable Examples

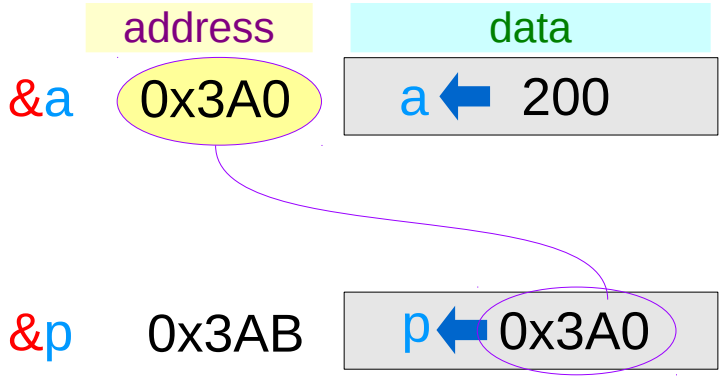
```
int      a = 200;

int *    p = & a;

int **   q = & p;
```




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

Pointer Variable **p** with an arrow notation





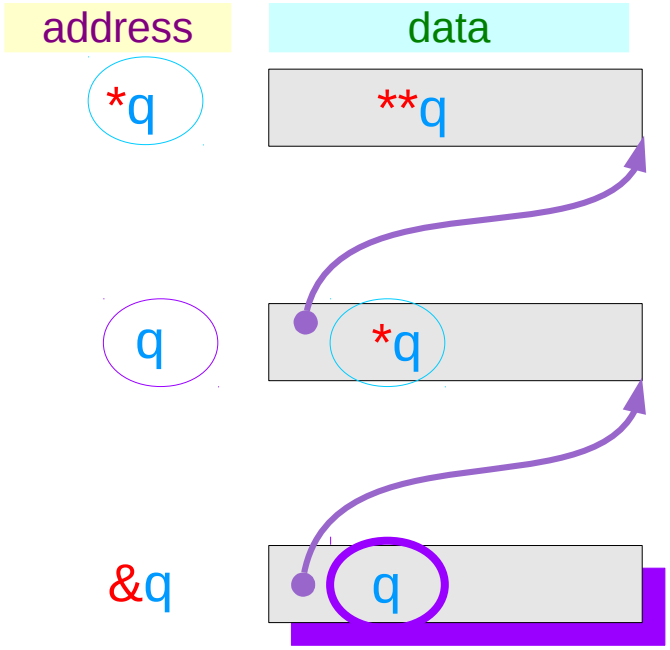
using an arrow notation

&p ➡ 0x3AB

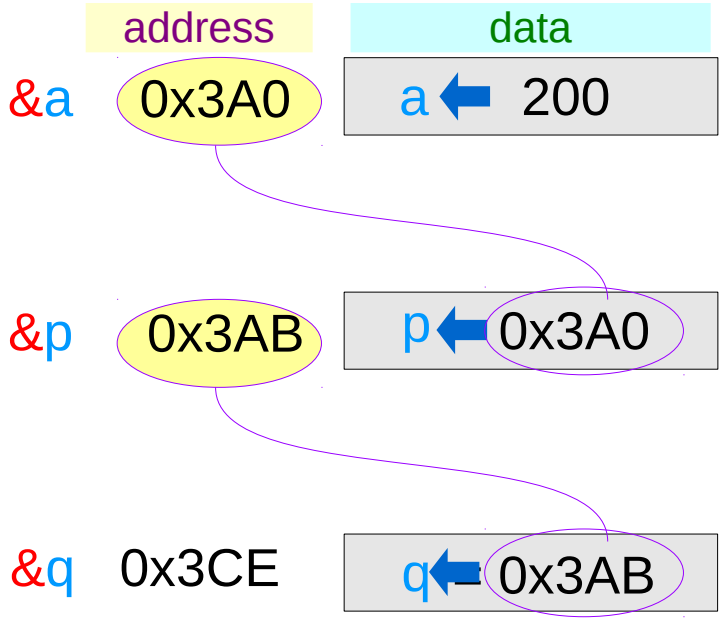
p ➡ 0x3A0

*p ➡ 200

Pointer Variable q with an arrow notation



using an arrow notation



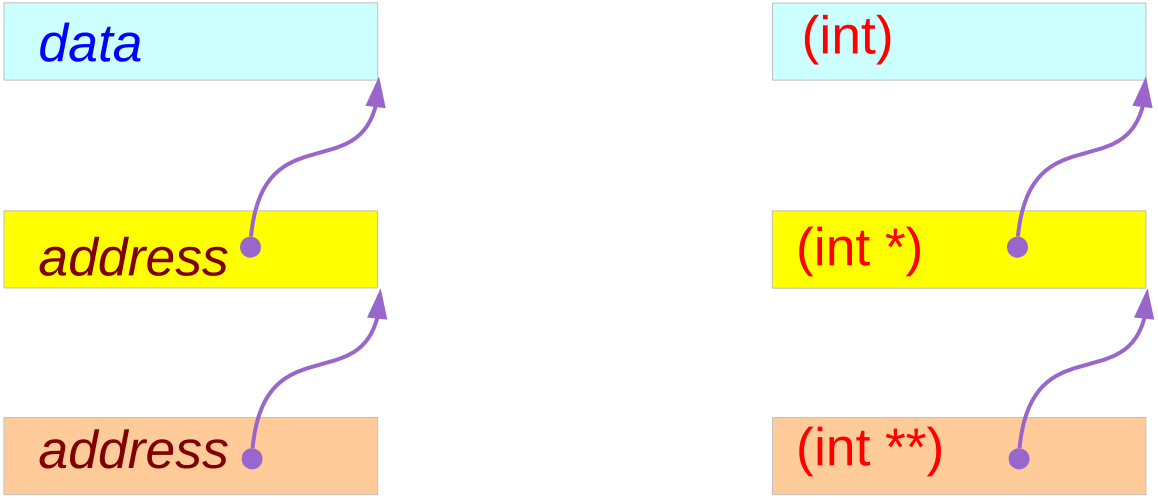
&q ➡ 0x3CE

q ➡ 0x3AB

*q ➡ 0x3A0

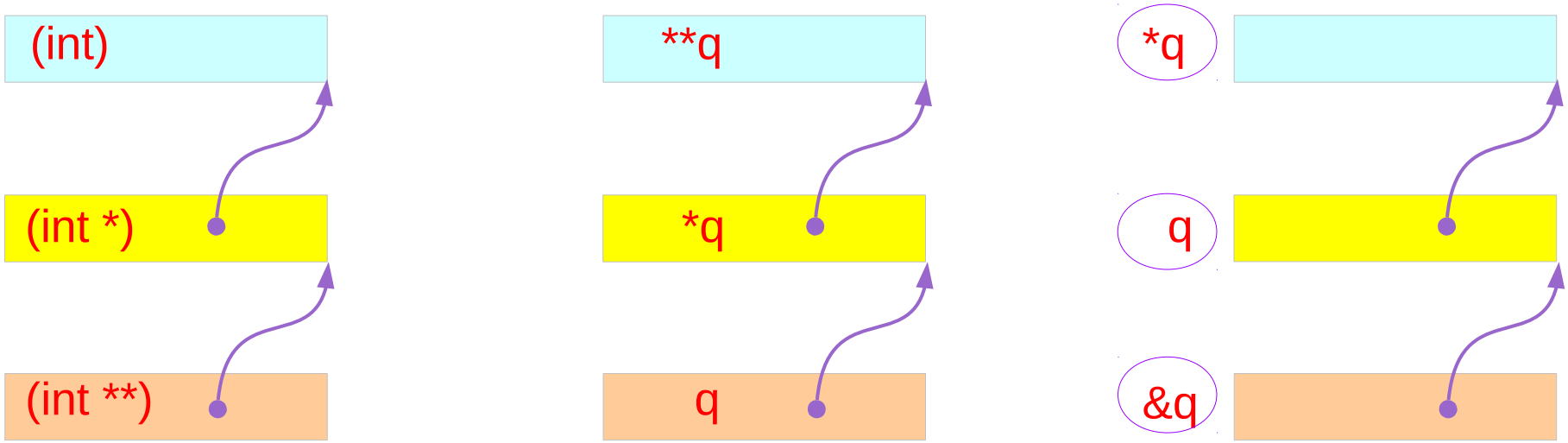
**q ➡ 200

Pointers – a type view



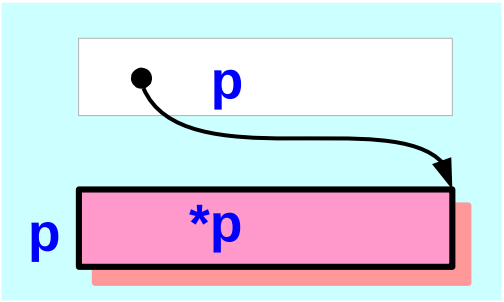
Types

Pointers – other view

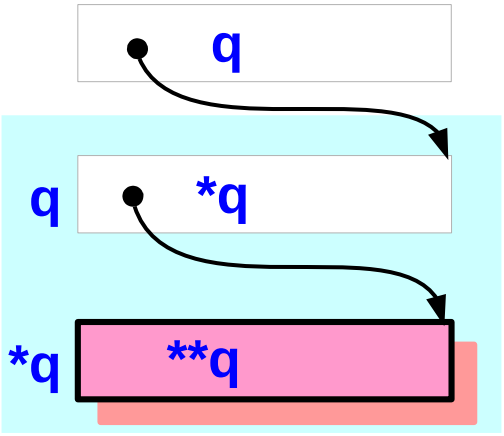


Single and double pointer examples (1)

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

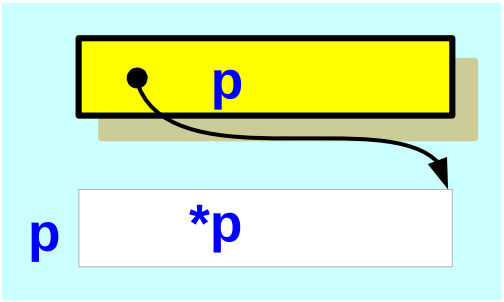


a, ***p**, and ****q**:
int variables

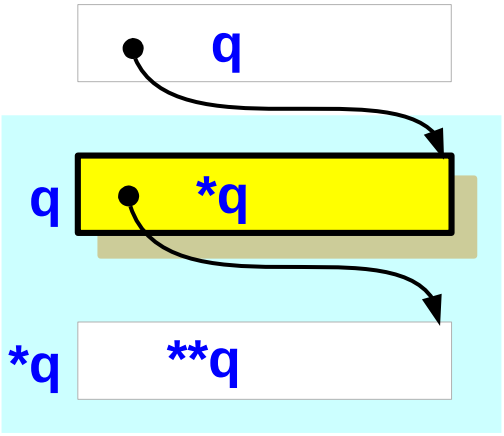


Single and double pointer examples (2)

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

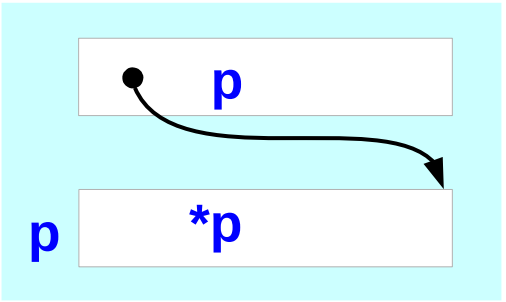


p and ***q** :
int pointer variables
(singlepointers)

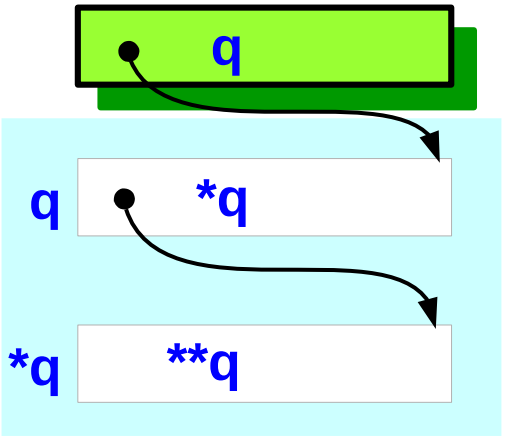


Single and double pointer examples (3)

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



q :
double int pointer variables

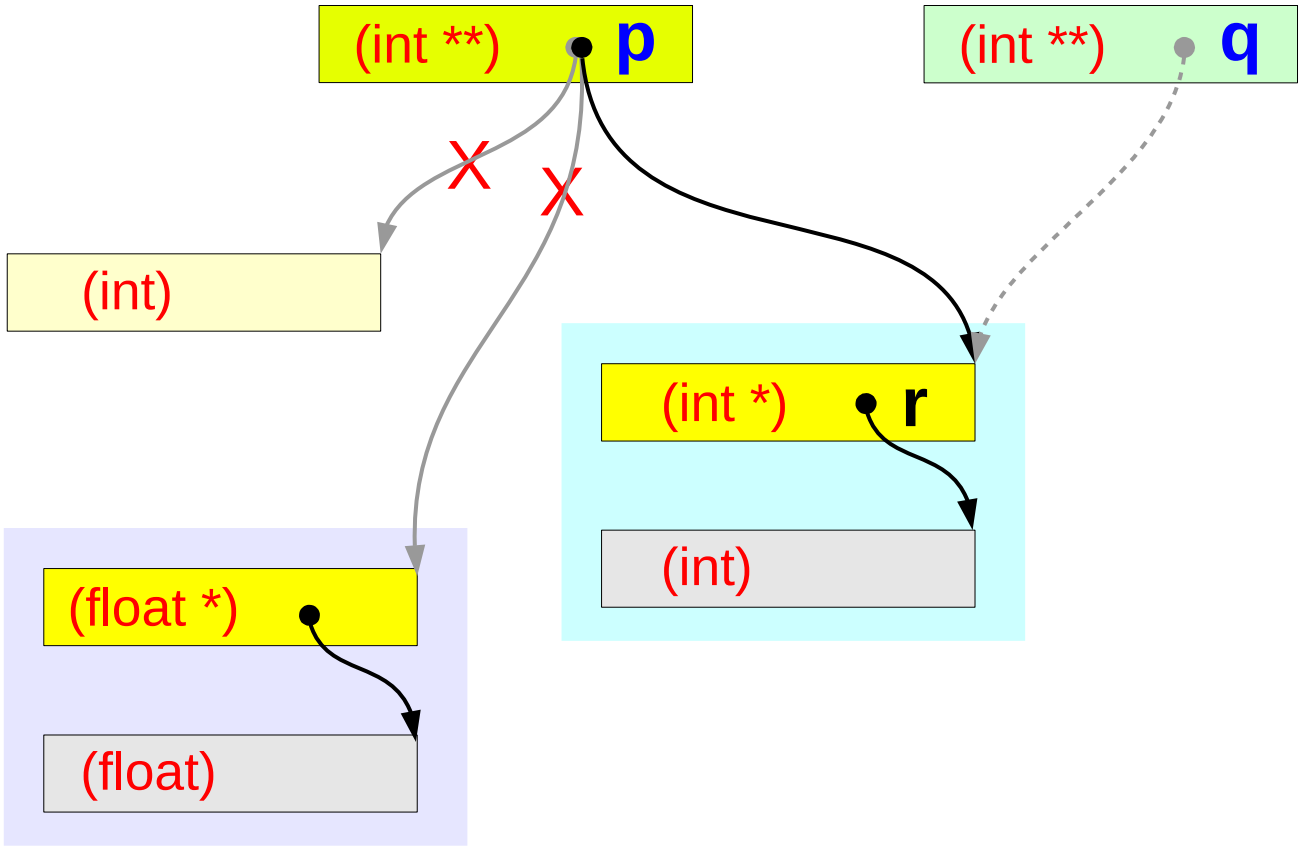


Double pointer variable assignments

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

p = &r;

q = p;



Pointed Addresses and Data

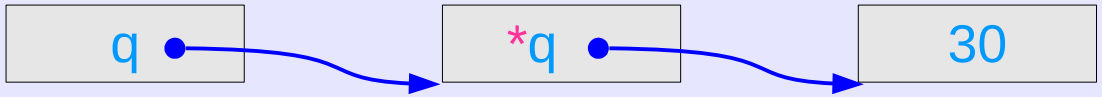
int a ; &a a =100

The variable a holds an integer data

int * p ; &p

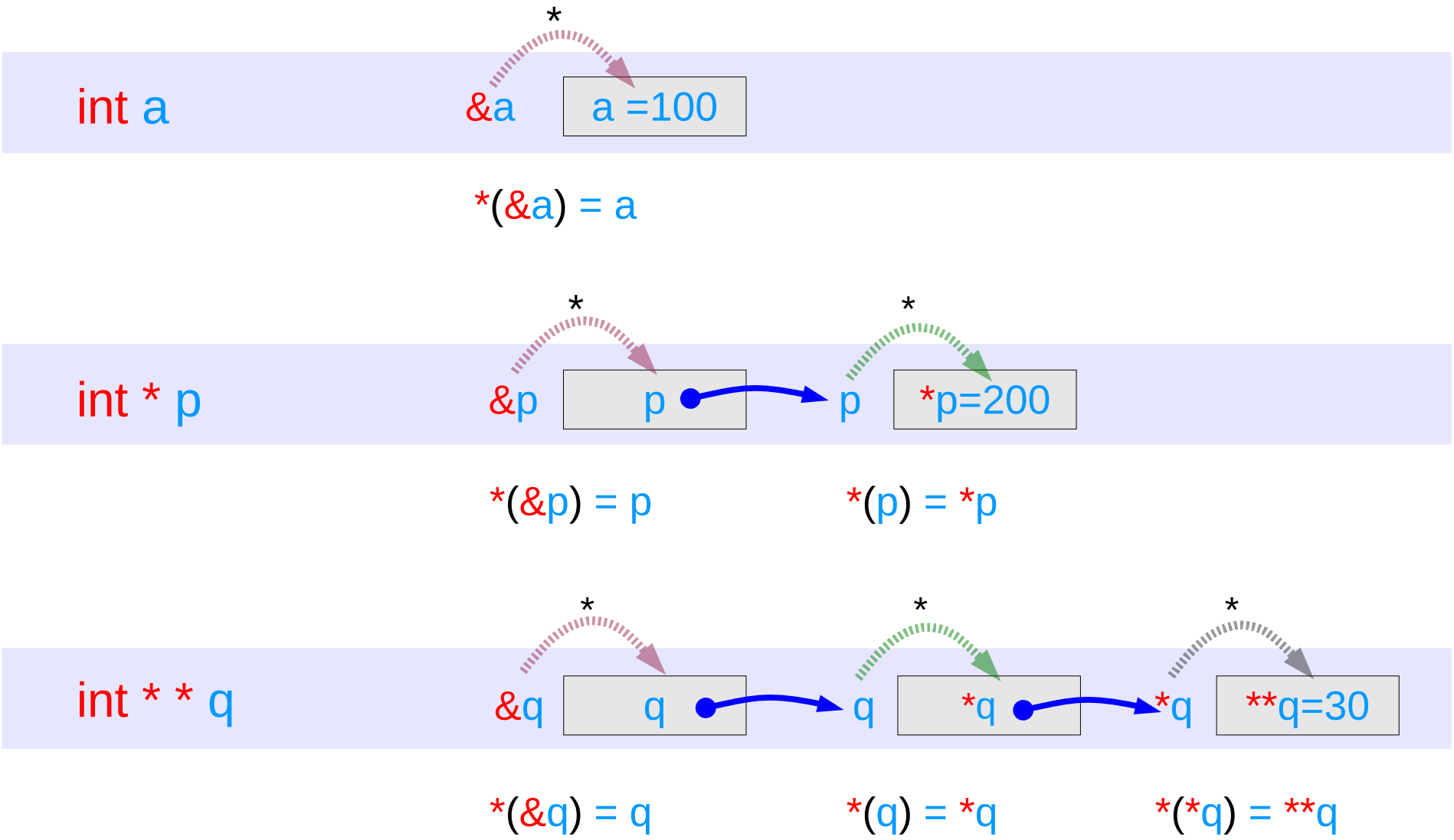
The **pointer** variable p holds an address,
at this address, an integer data is stored

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

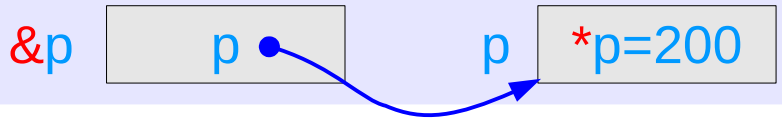
Direct Access

integer

1 memory access

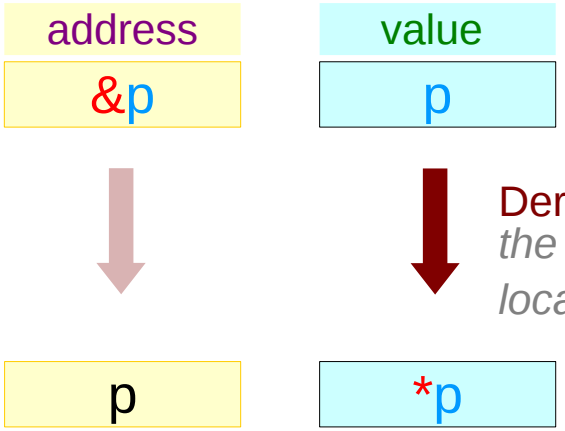
Indirect access *p to an integer a

```
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

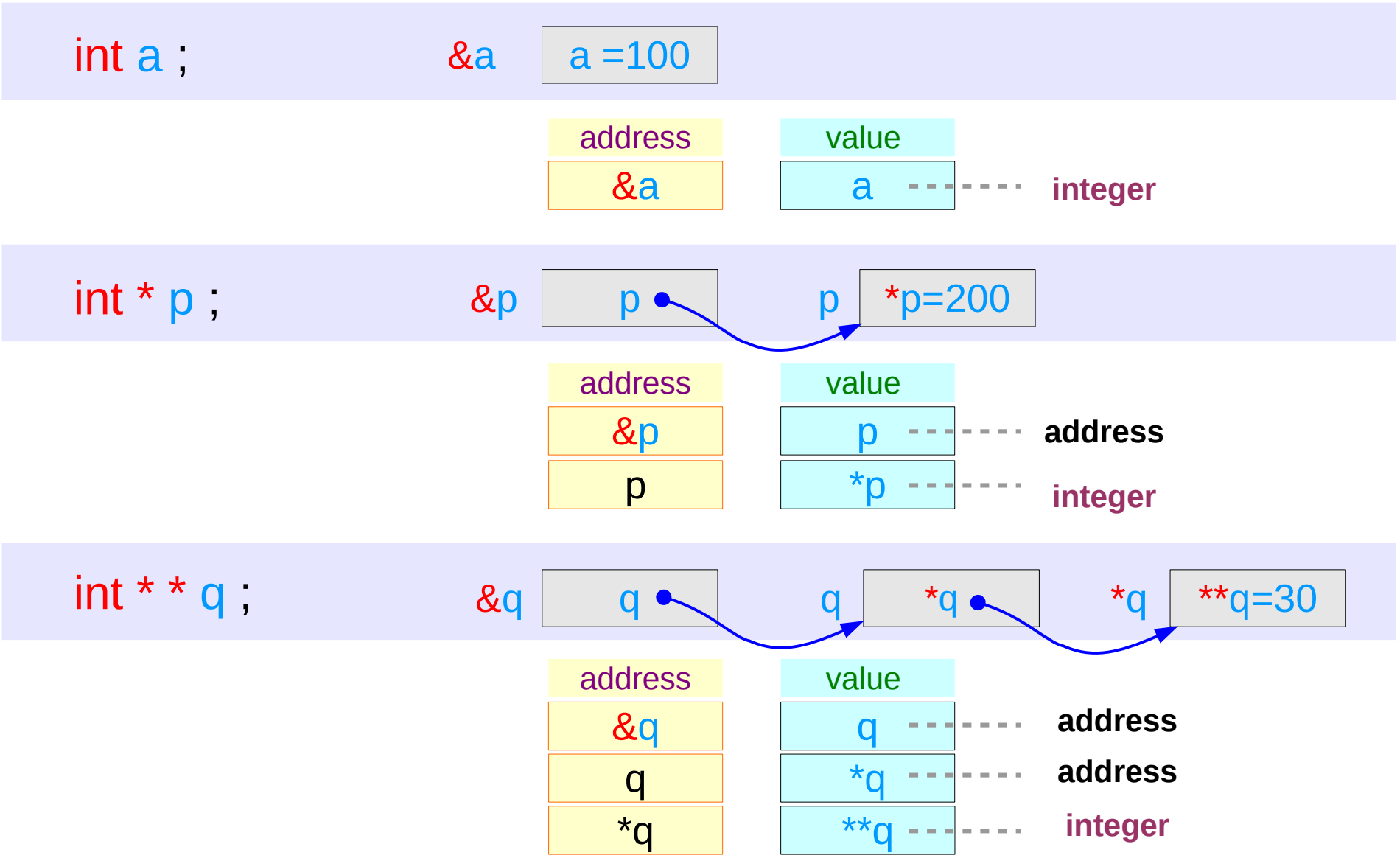
Dereference Operator *

the content of the pointed location

Dereference Operator *

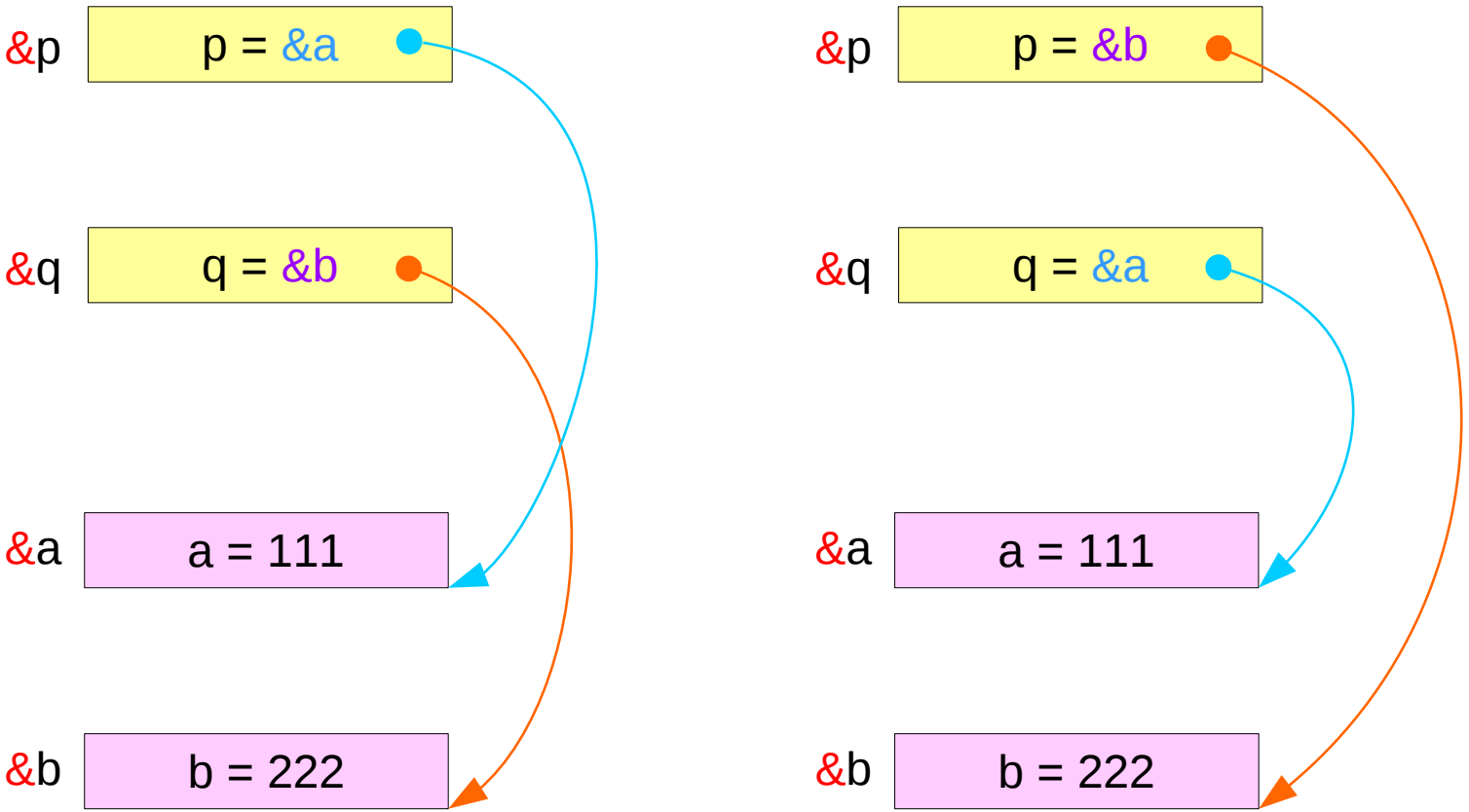
the content of the pointed location

Values of variables



Swapping pointers

Swapping integer pointers



Swapping integer pointers


```
int   *p,   *q  ;
swap_pointers(   &p,   &q );          function call
void 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

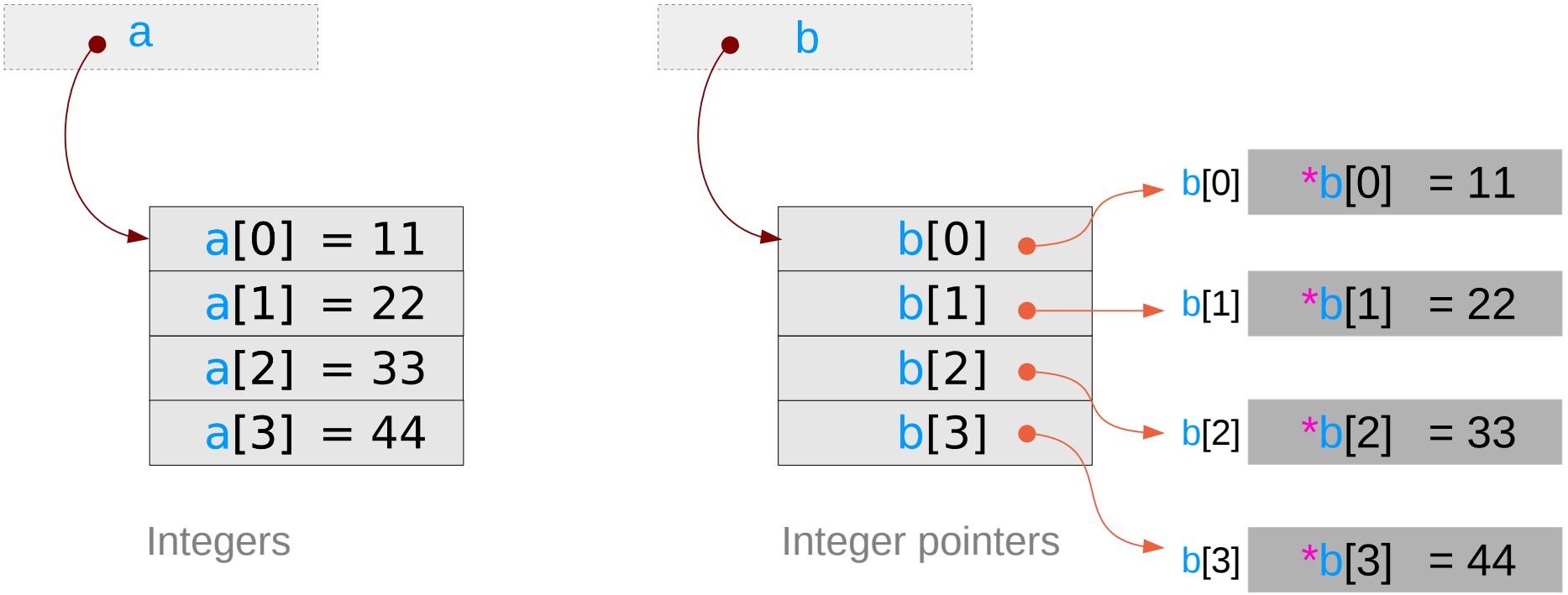
No. of elements = 4

int * b [4]

Type of each element

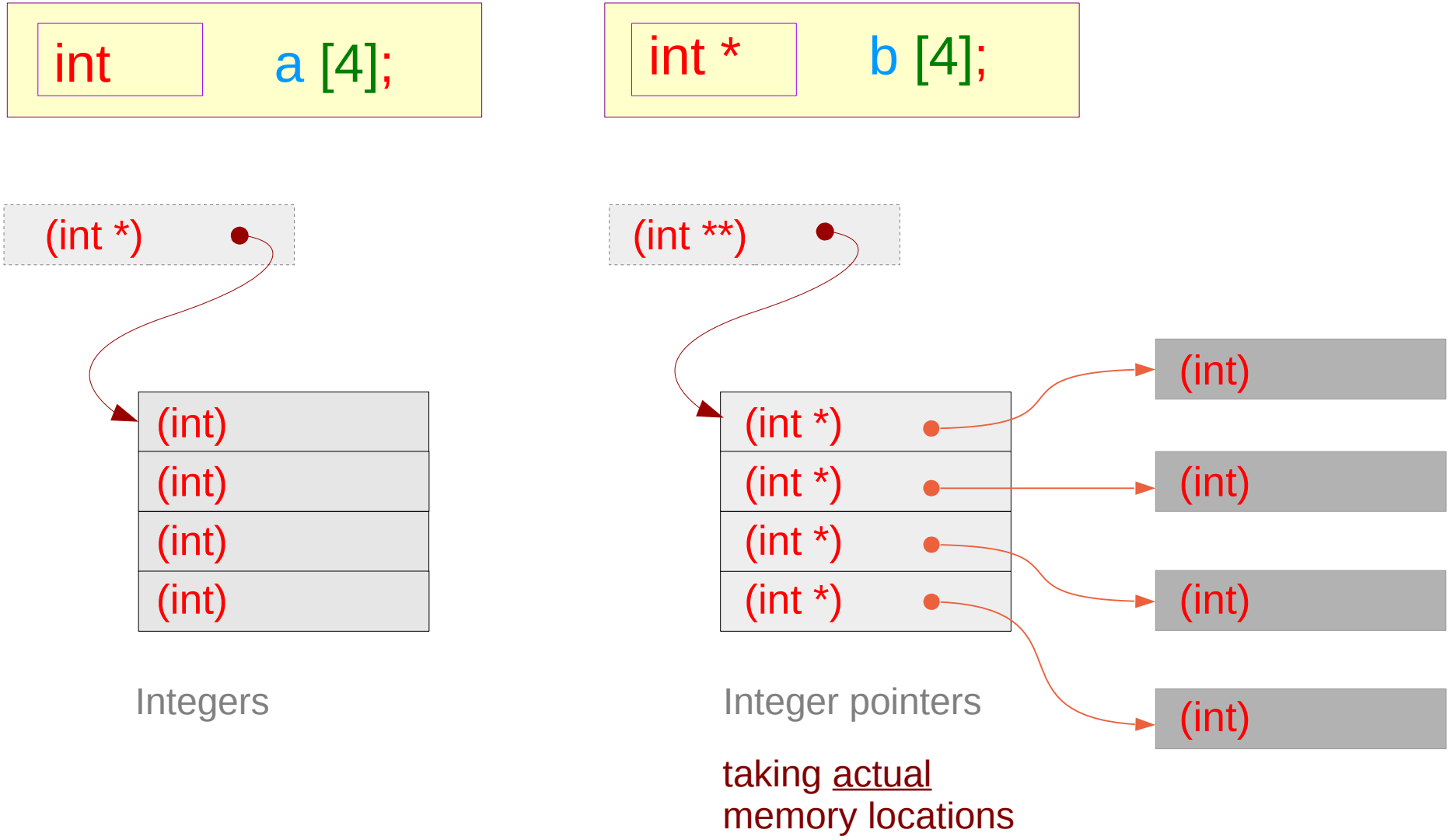
Array of Pointers – a variable view

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



taking actual
memory locations

Array of Pointers – a type view



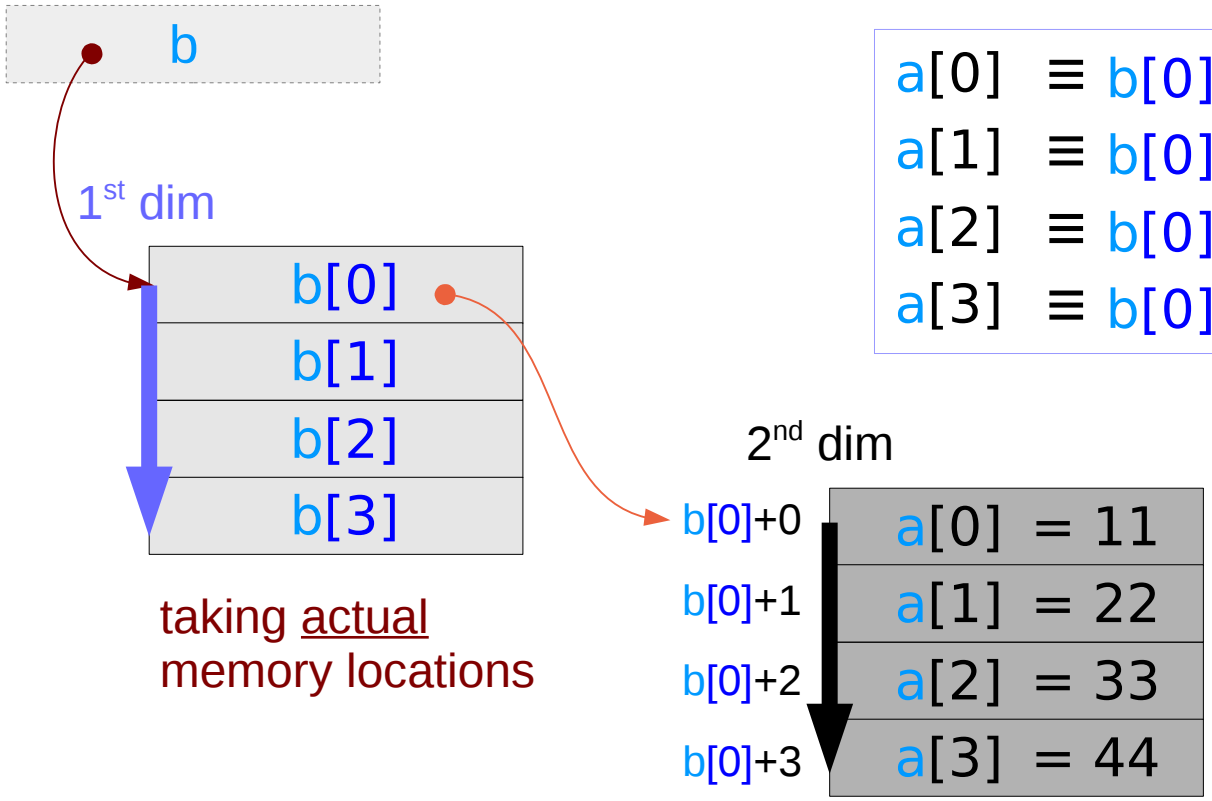
Array of Pointers – extending a dimension

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

assignment `b[0] = a`

equivalence

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



Pointer to Arrays

Pointer to an array – variable declarations

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

an integer pointer

```
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 – a type view

int 4 byte data

int *

an integer pointer

array pointer:
a pointer to an array

pointer array:
an array of pointers

int [4] 4*4 byte data

int (*) [4]

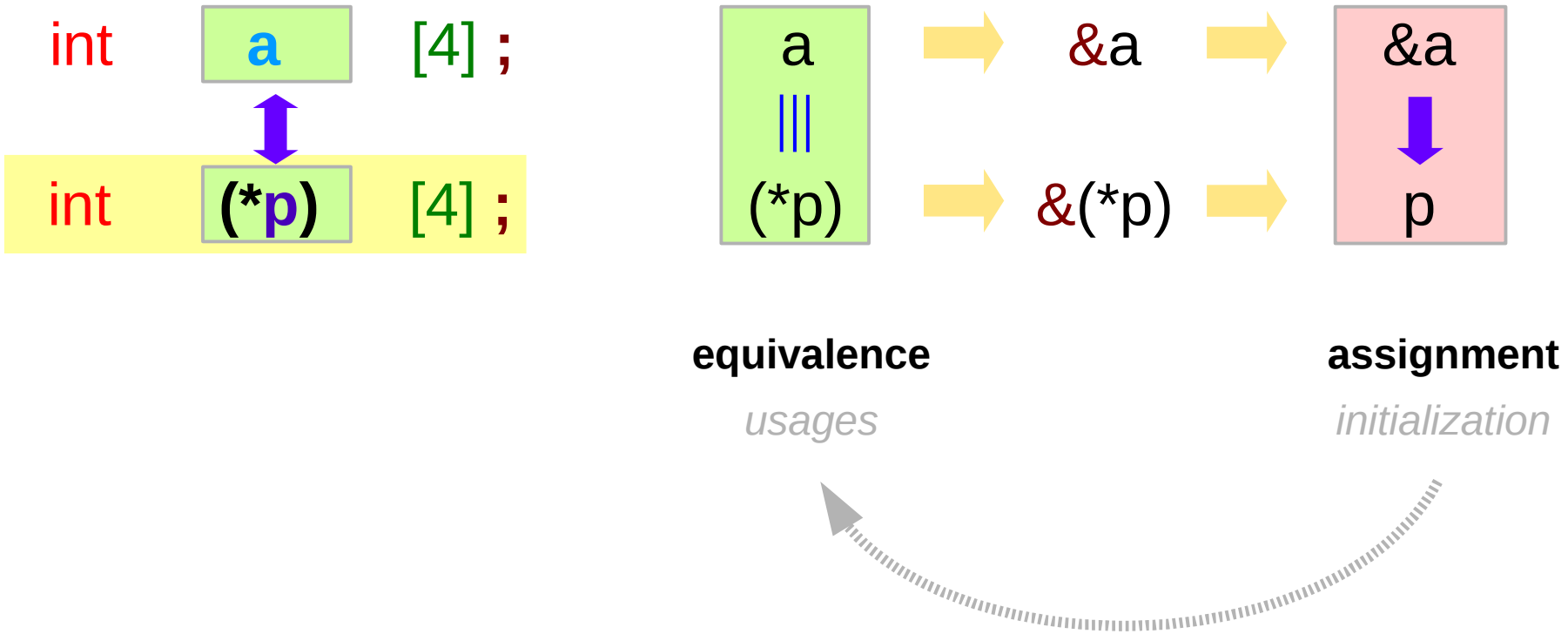
an array pointer

int (int, int) instructions

int (*) (int, int)

a function pointer

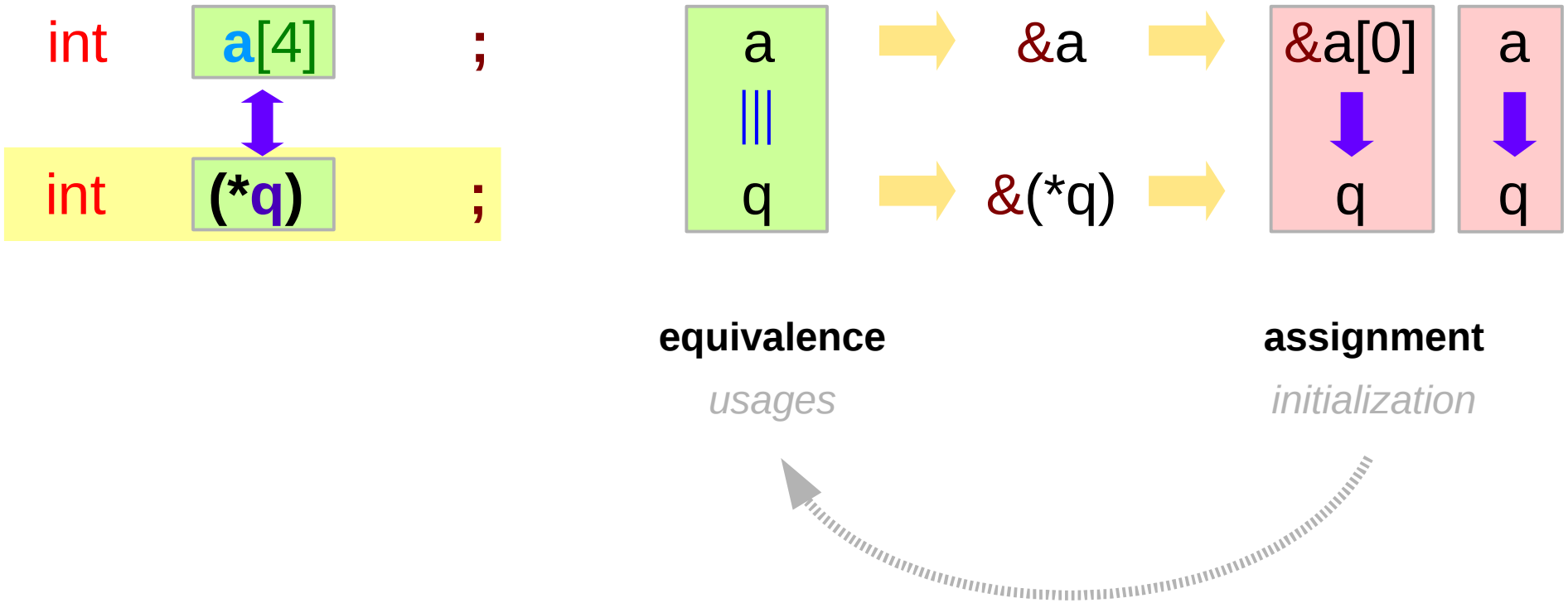
Pointer to an Array : Assignment and Dereference (1)



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

sizeof(*p)= 4*4 bytes : the whole size of the pointed 1-d array

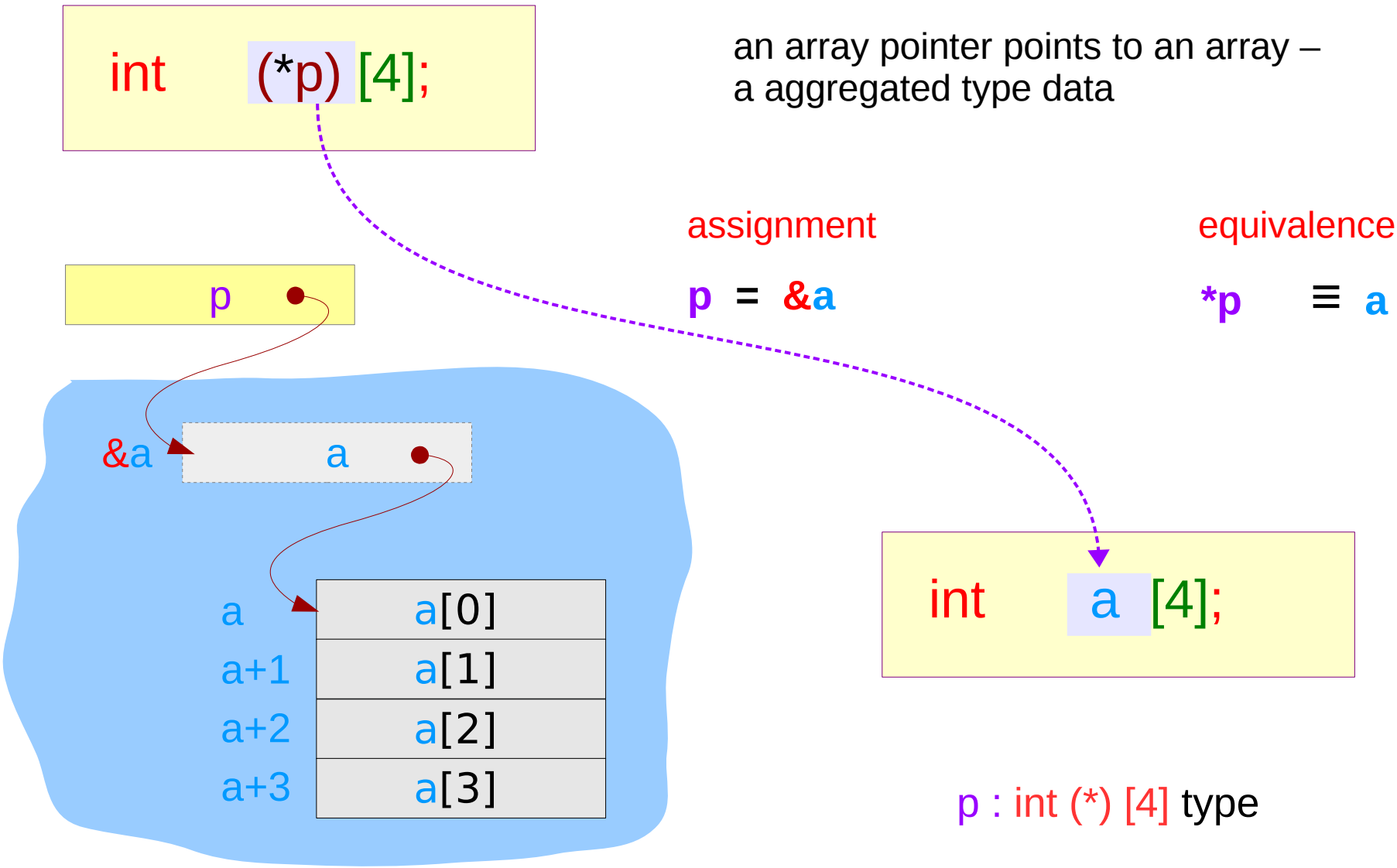
Pointer to an Array : Assignment and Dereference (2)



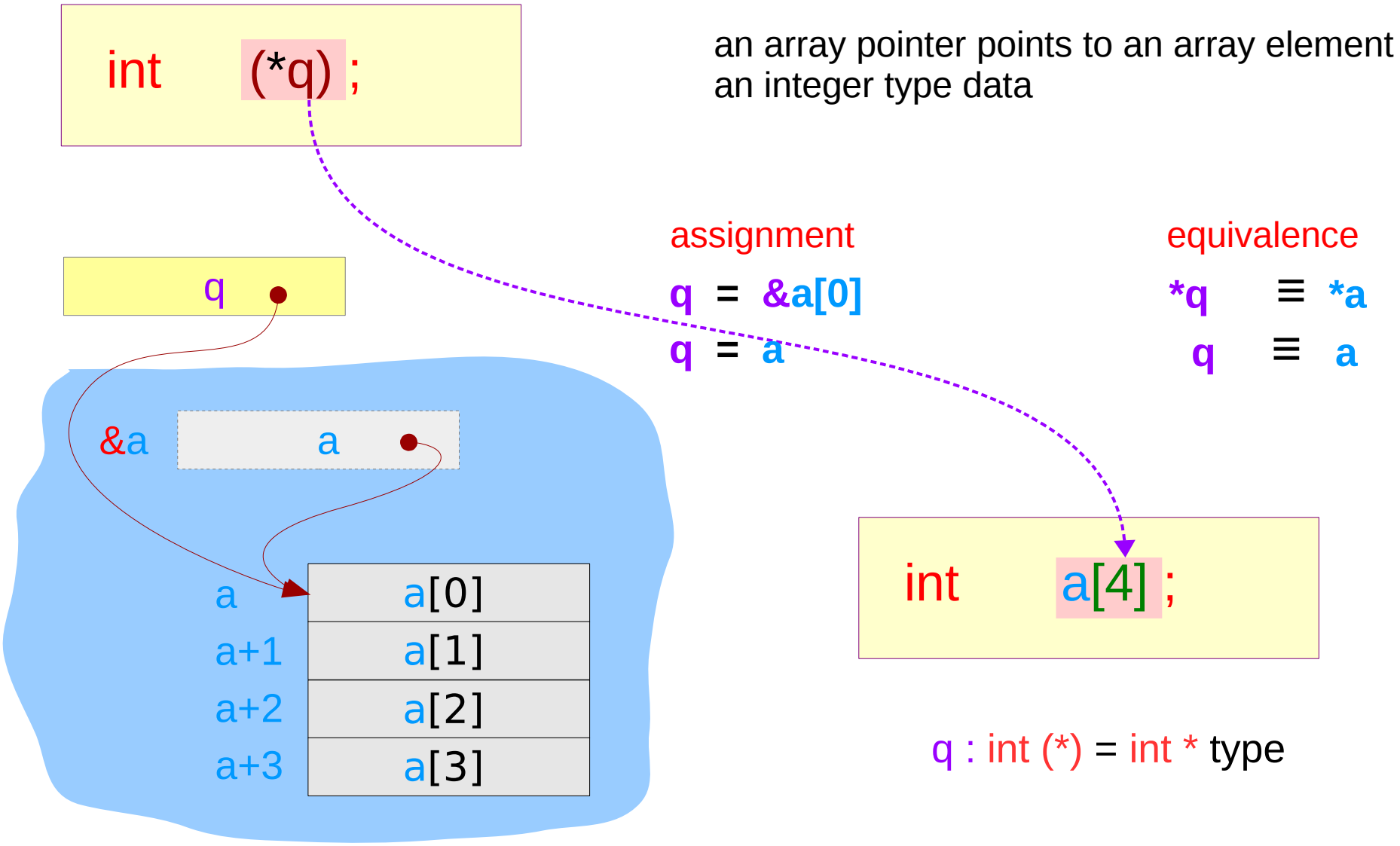
sizeof(q)= 8 bytes : the size of a pointer

sizeof(*q)= 4 bytes : the whole size of the pointed 0-d array

Pointer to an array – a variable view (1)



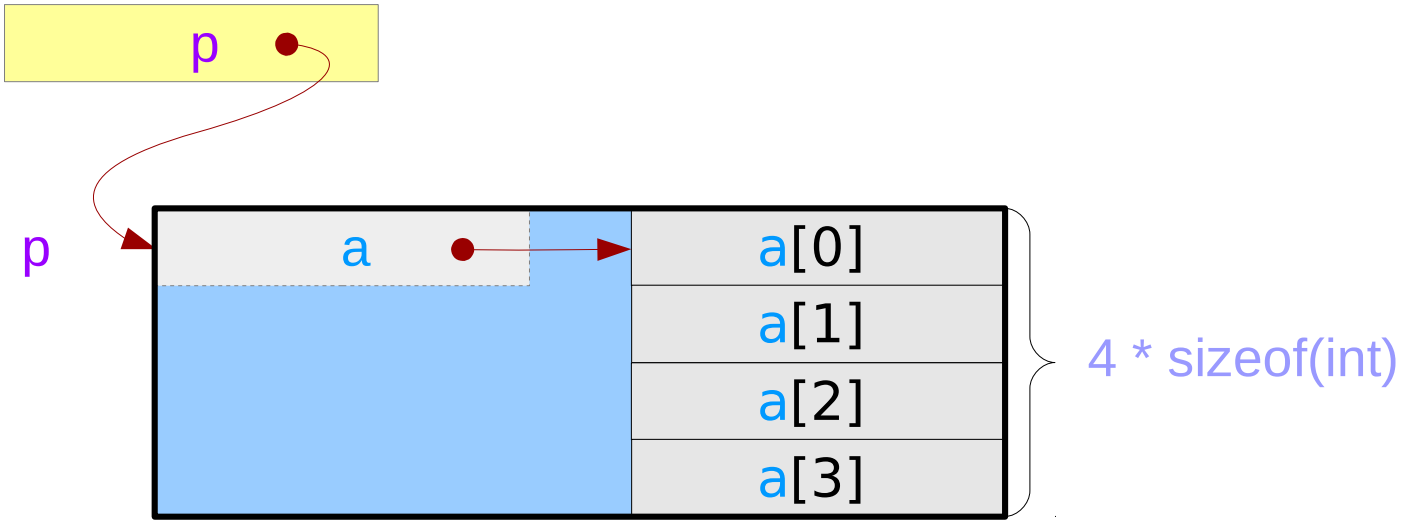
Pointer to an array – a variable view (2)



Pointer to an array – an aggregated type view

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

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

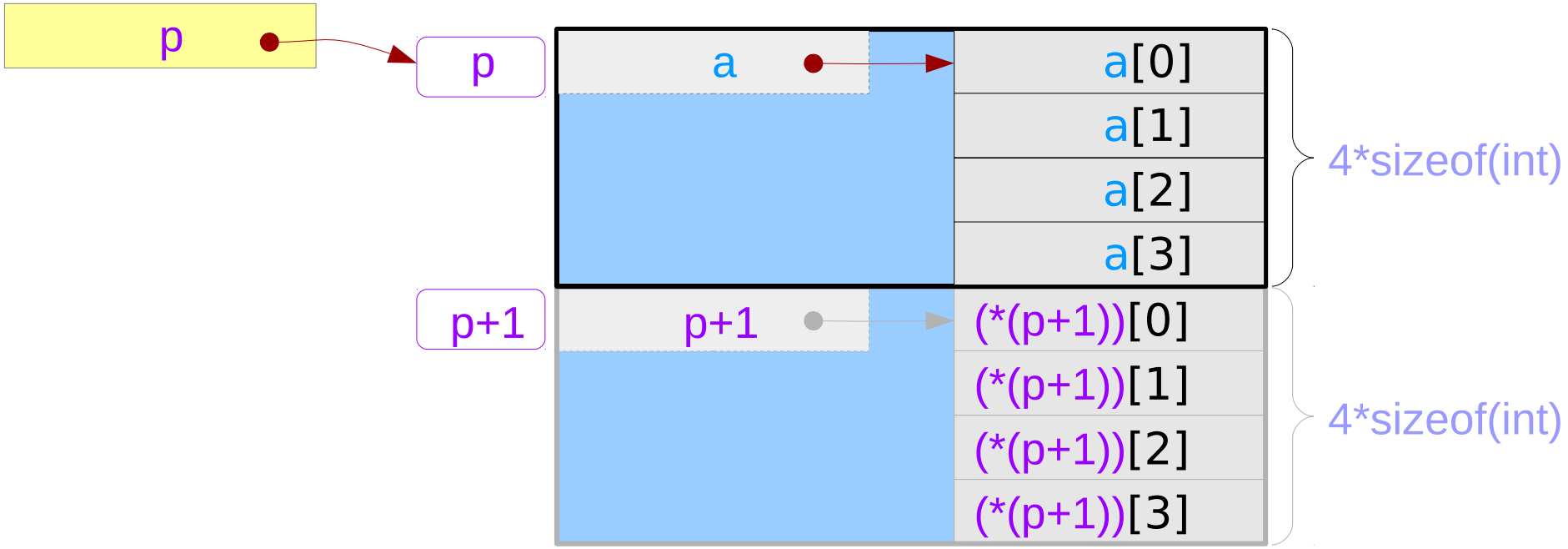


Incrementing an array pointer

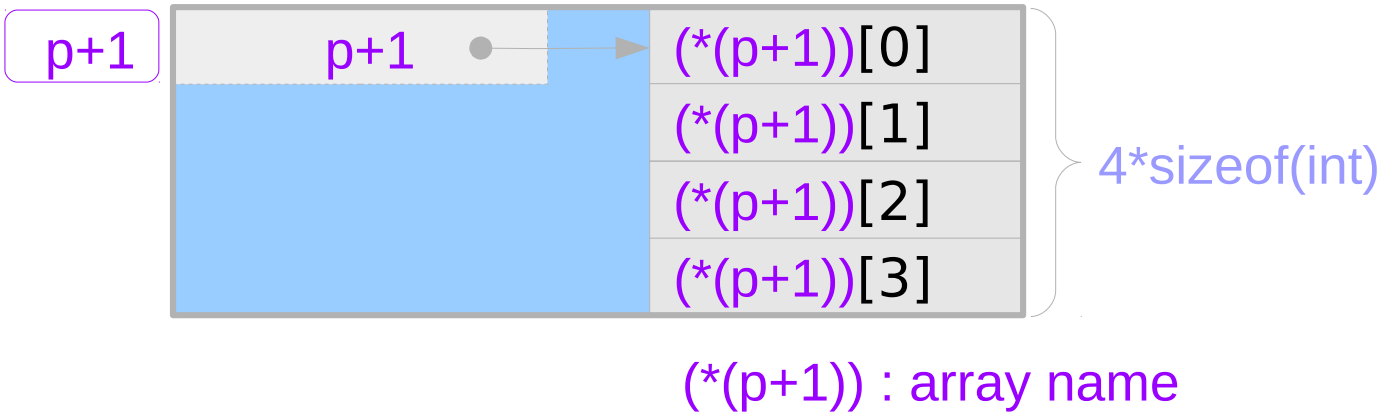
```
int     (*p) [4];
```

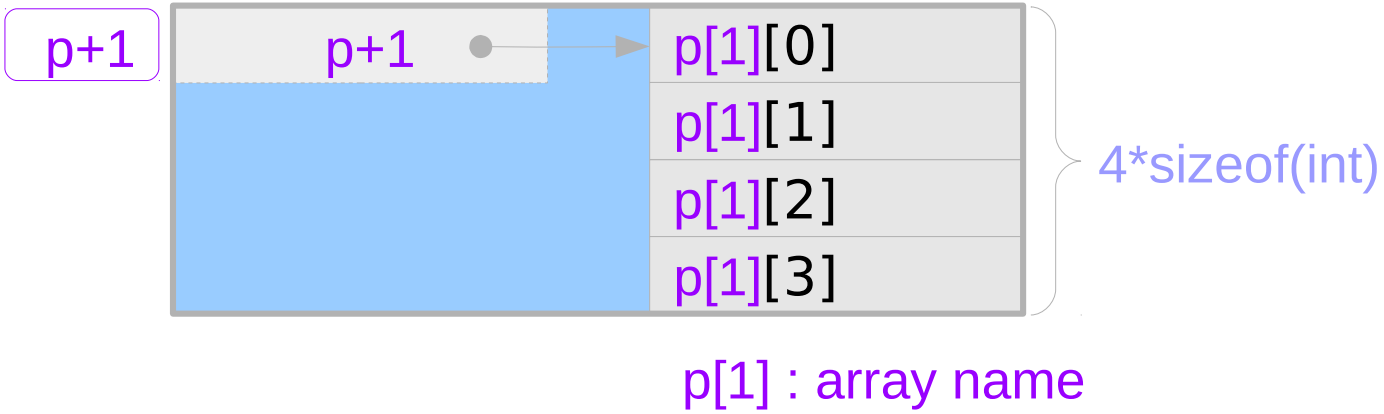
Address value (p+1) – **Address value** (p)
= (long) (p+1) - (long) (p) = 4 * sizeof(int)

Aggregated Type Size



Incrementing an array pointer – extending a dimension

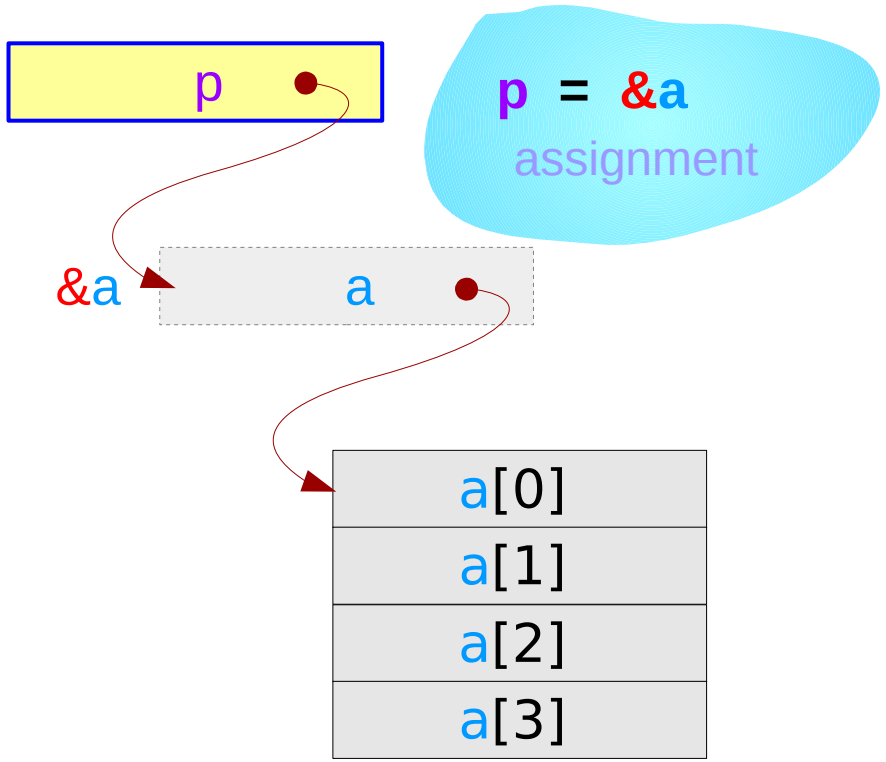


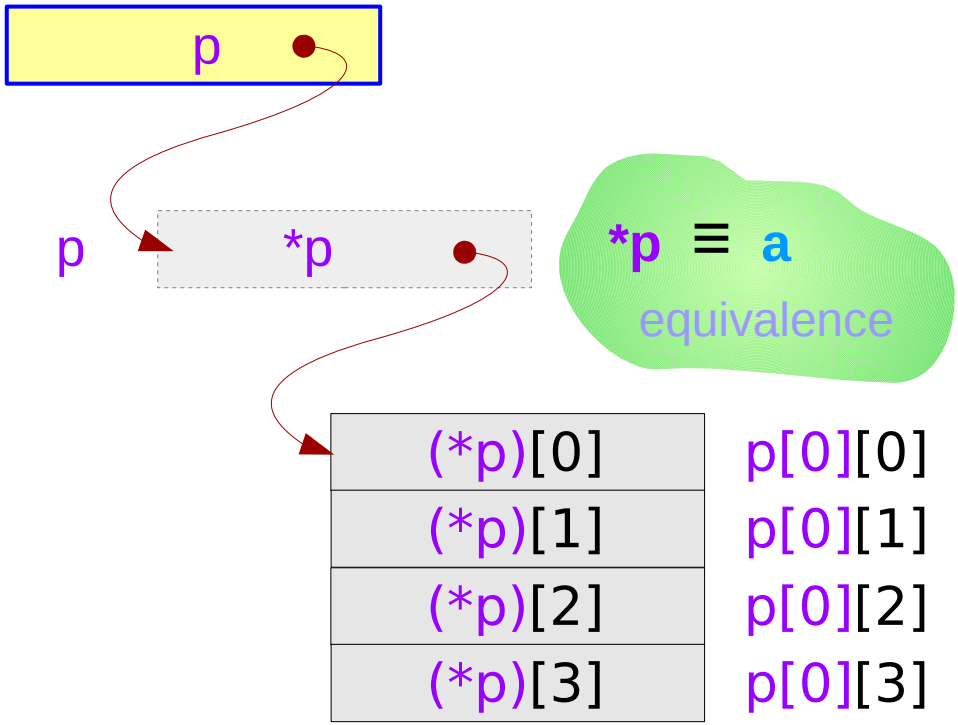


An array pointer and a 1-d array

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

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

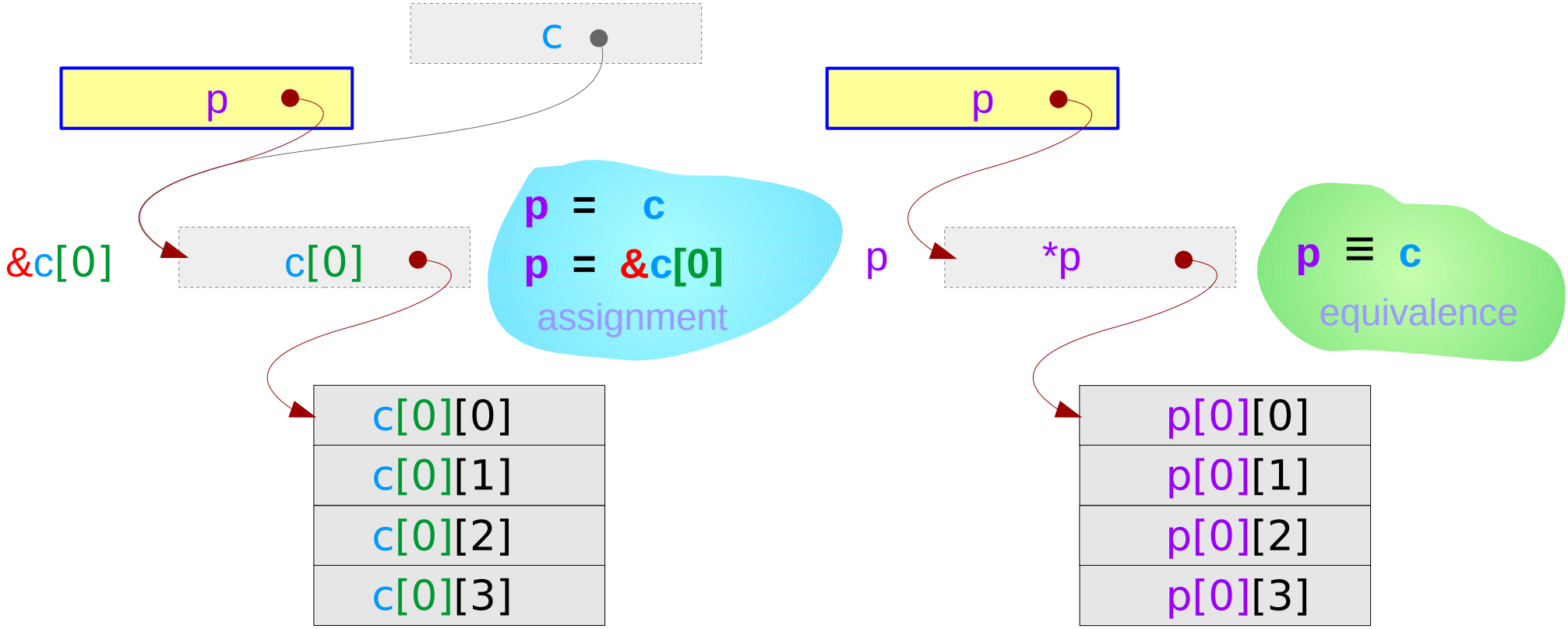




An array pointer and a 2-d array

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

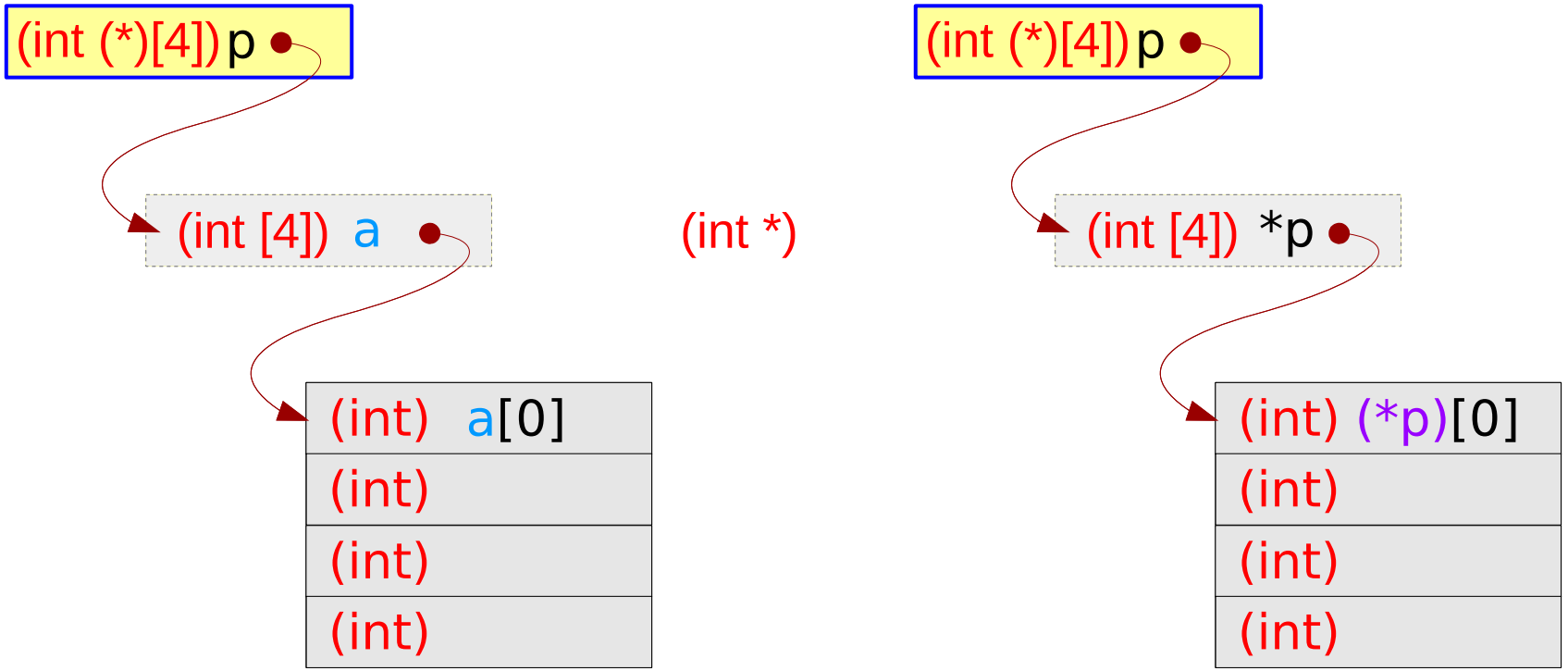
```
int  (*p) [4]  = &c[0];
```



An array pointer and a 1-d array – a type view

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

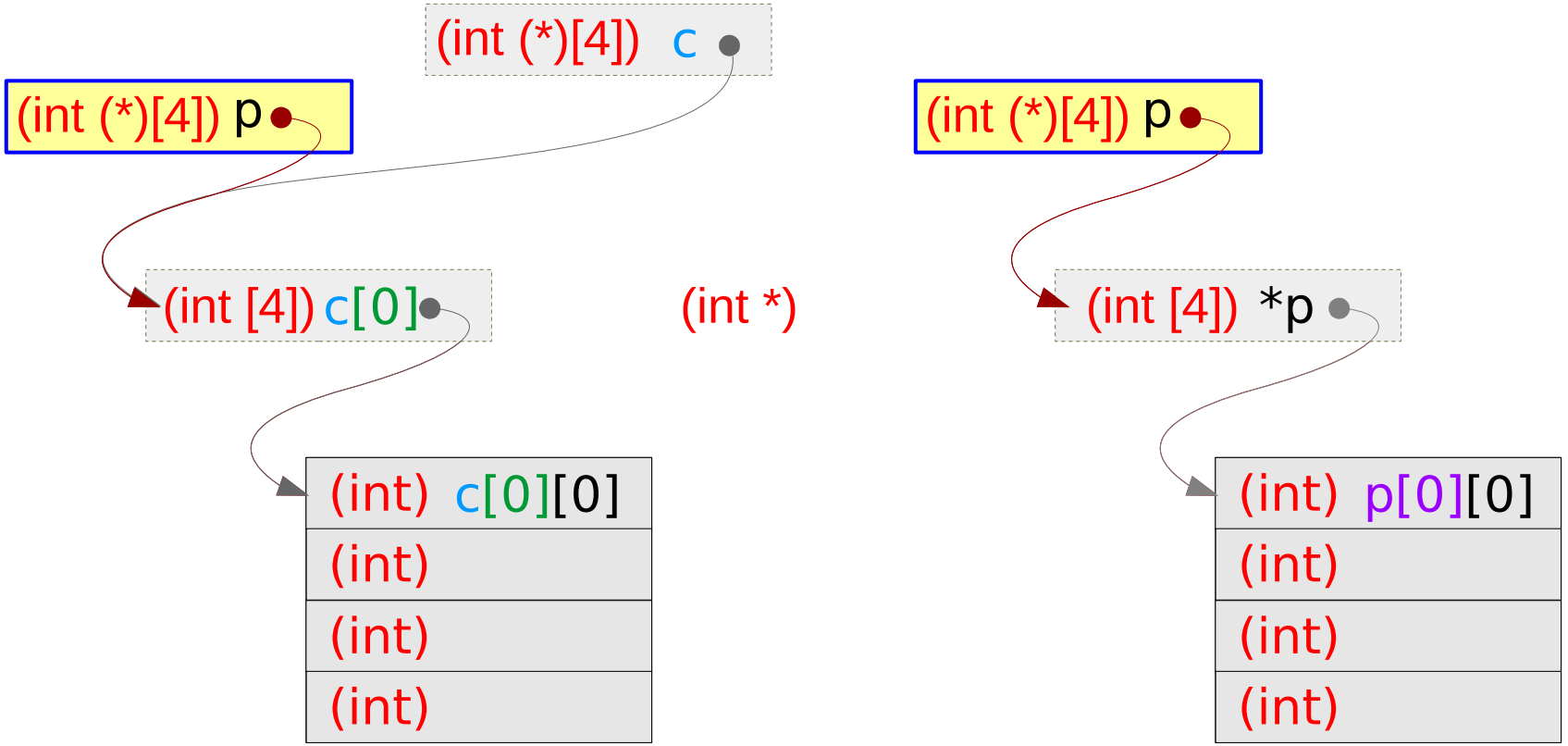
```
int  (*p) [4]  = &a;
```



An array pointer and a 2-d array – a type view

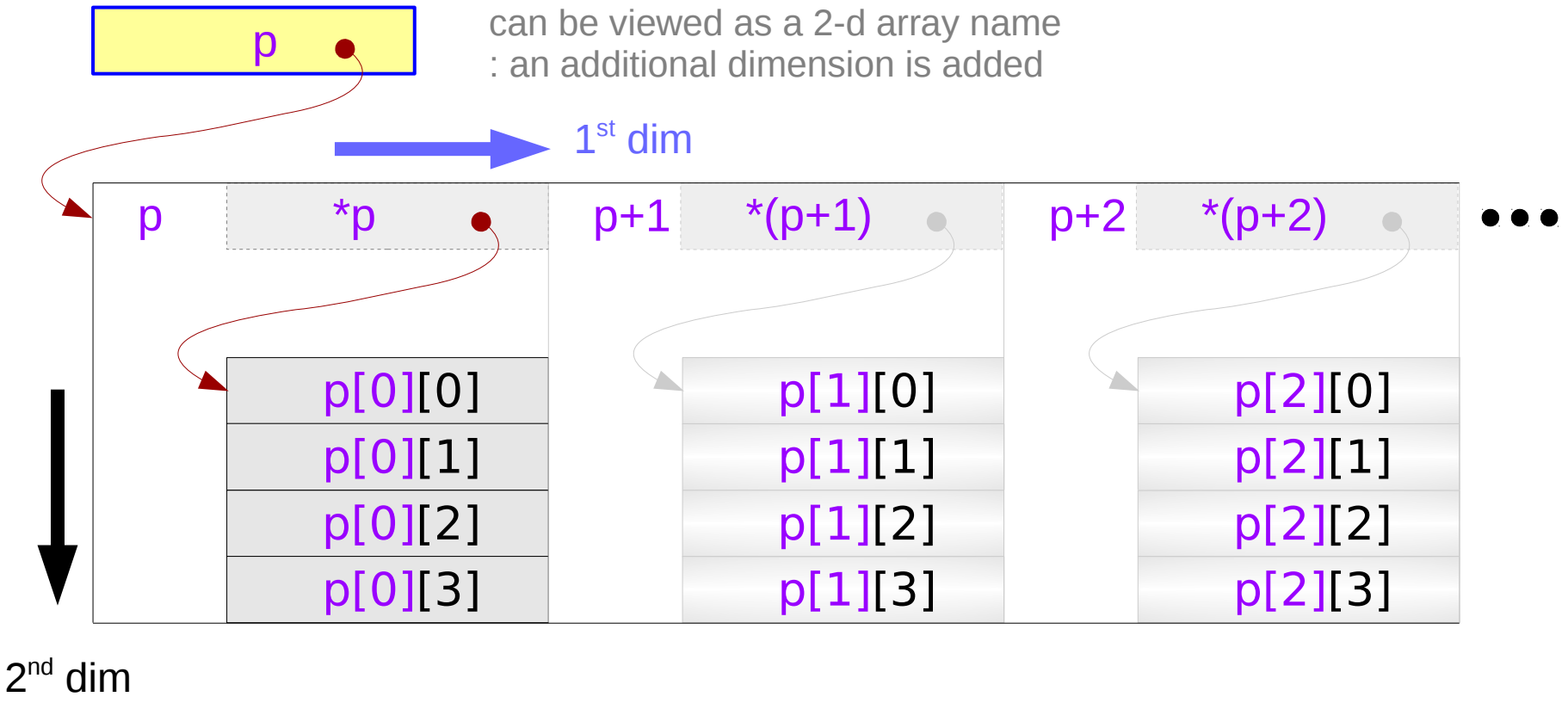
```
int          c [4][4];
```

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

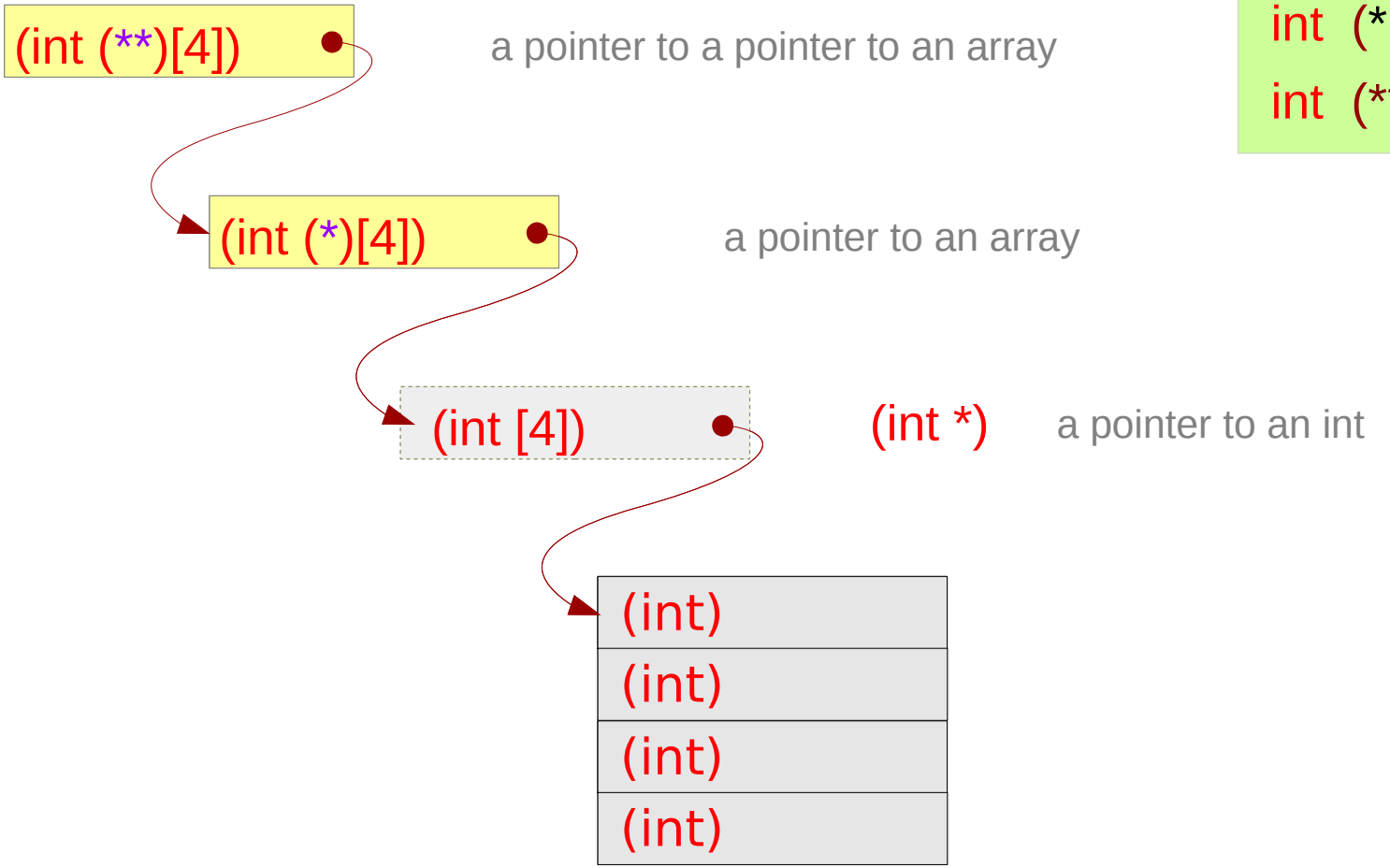


An array pointer – extending a dimension

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



Double pointer to an array – a type view

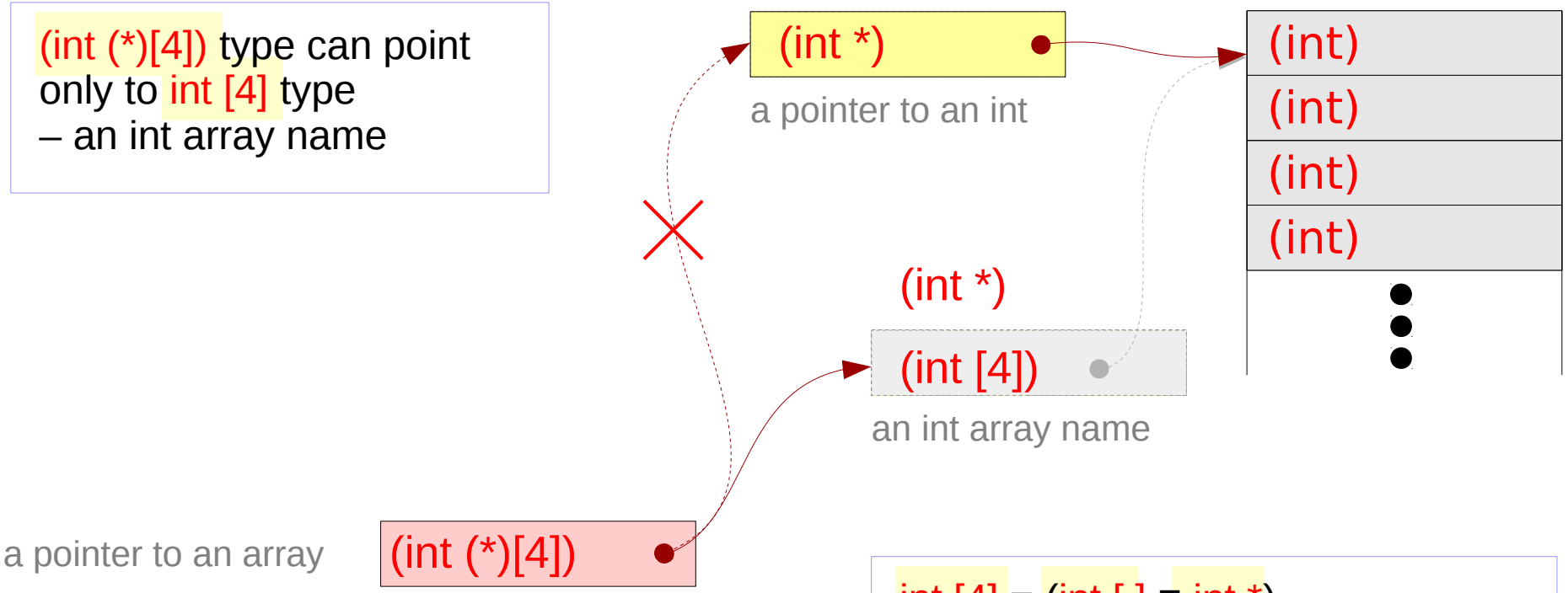



```
int  a[4]  ;
int  (*p) [4]  ;
int  (**q) [4]  ;
```

Pointer to Multi-dimensional Arrays

Integer pointer type

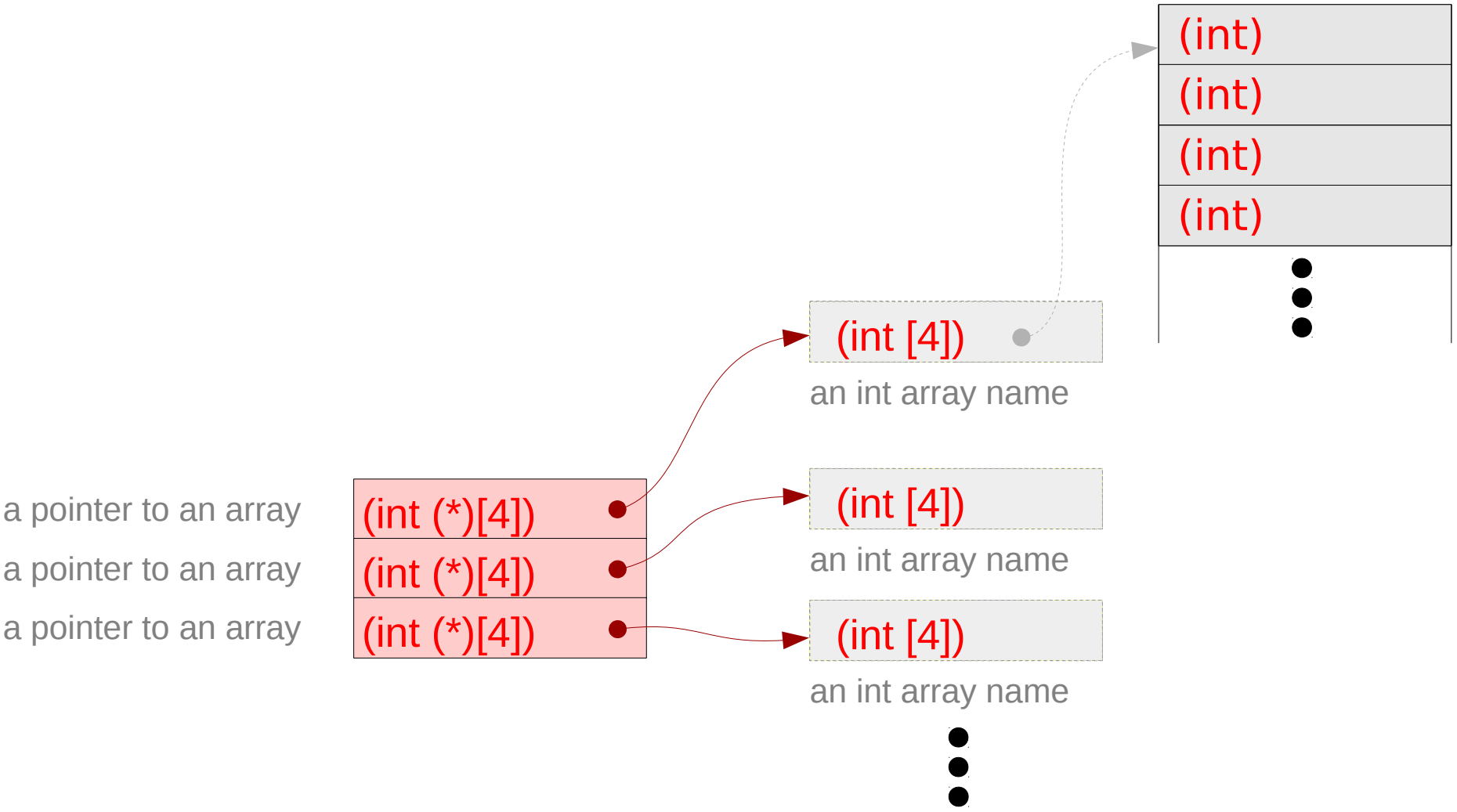
(int (*)[4]) type can point only to int [4] type
– an int array name



int [4] = (int [ ] ≡ int *)

equivalent in a sense that each of these types points to an int type data

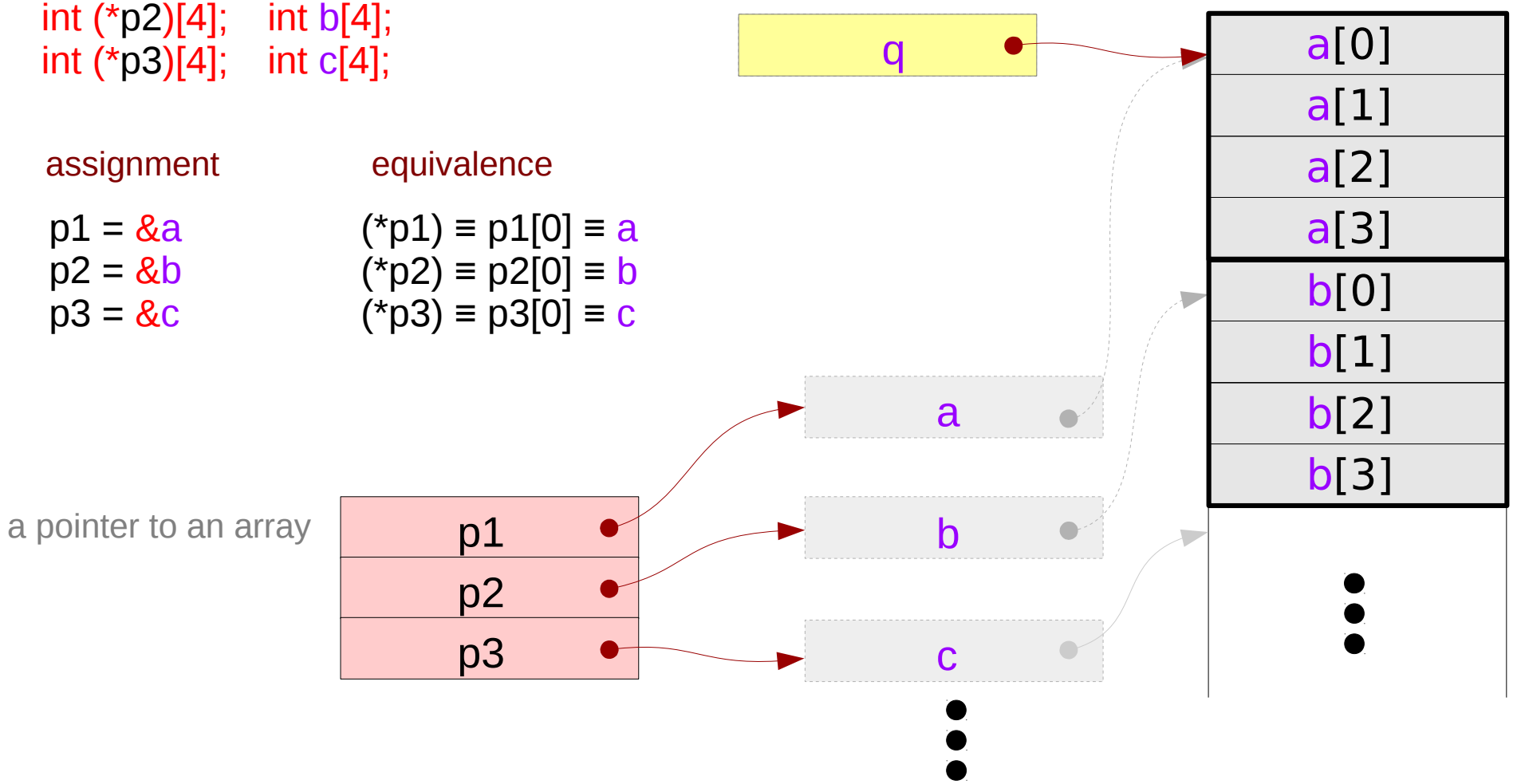
Series of array pointers – a type view



Series of array pointers – a variable view

```
int (*p1)[4];   int a[4];   int  (*q);
int (*p2)[4];   int b[4];
int (*p3)[4];   int c[4];
```

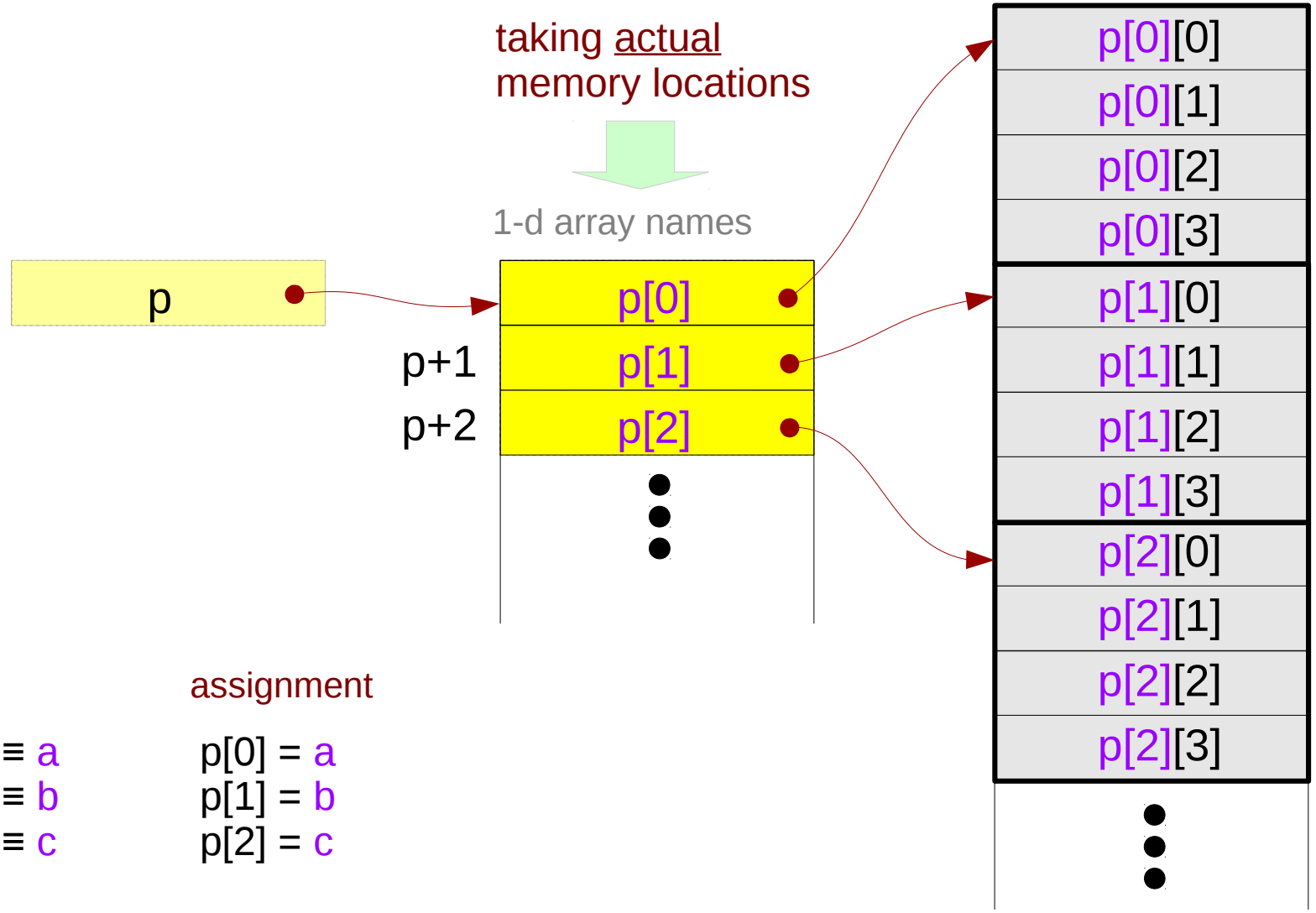
assignment	equivalence
p1 = &a	(*p1) ≡ p1[0] ≡ a
p2 = &b	(*p2) ≡ p2[0] ≡ b
p3 = &c	(*p3) ≡ p3[0] ≡ c



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

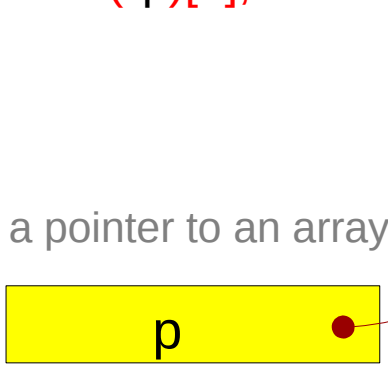
assignment

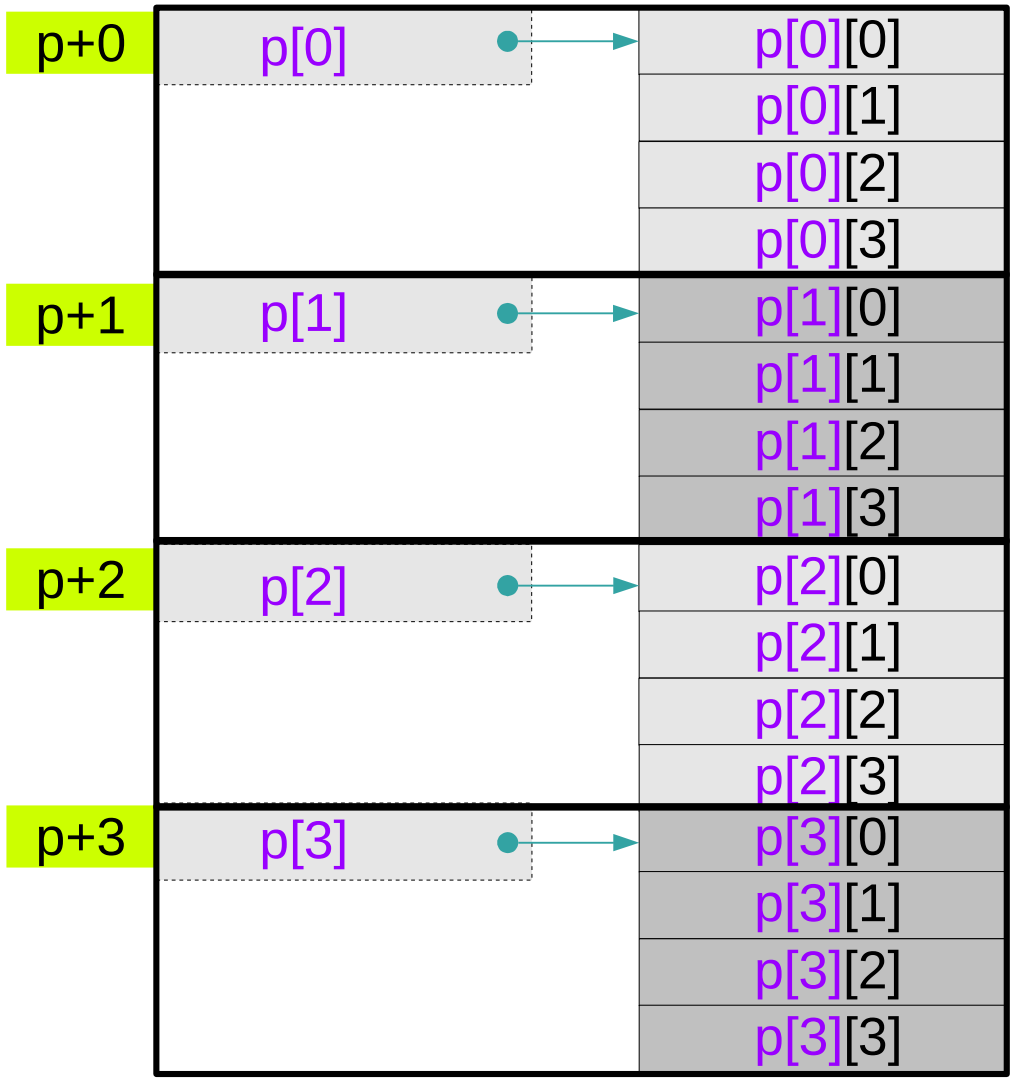
p[0] = a
p[1] = b
p[2] = c

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

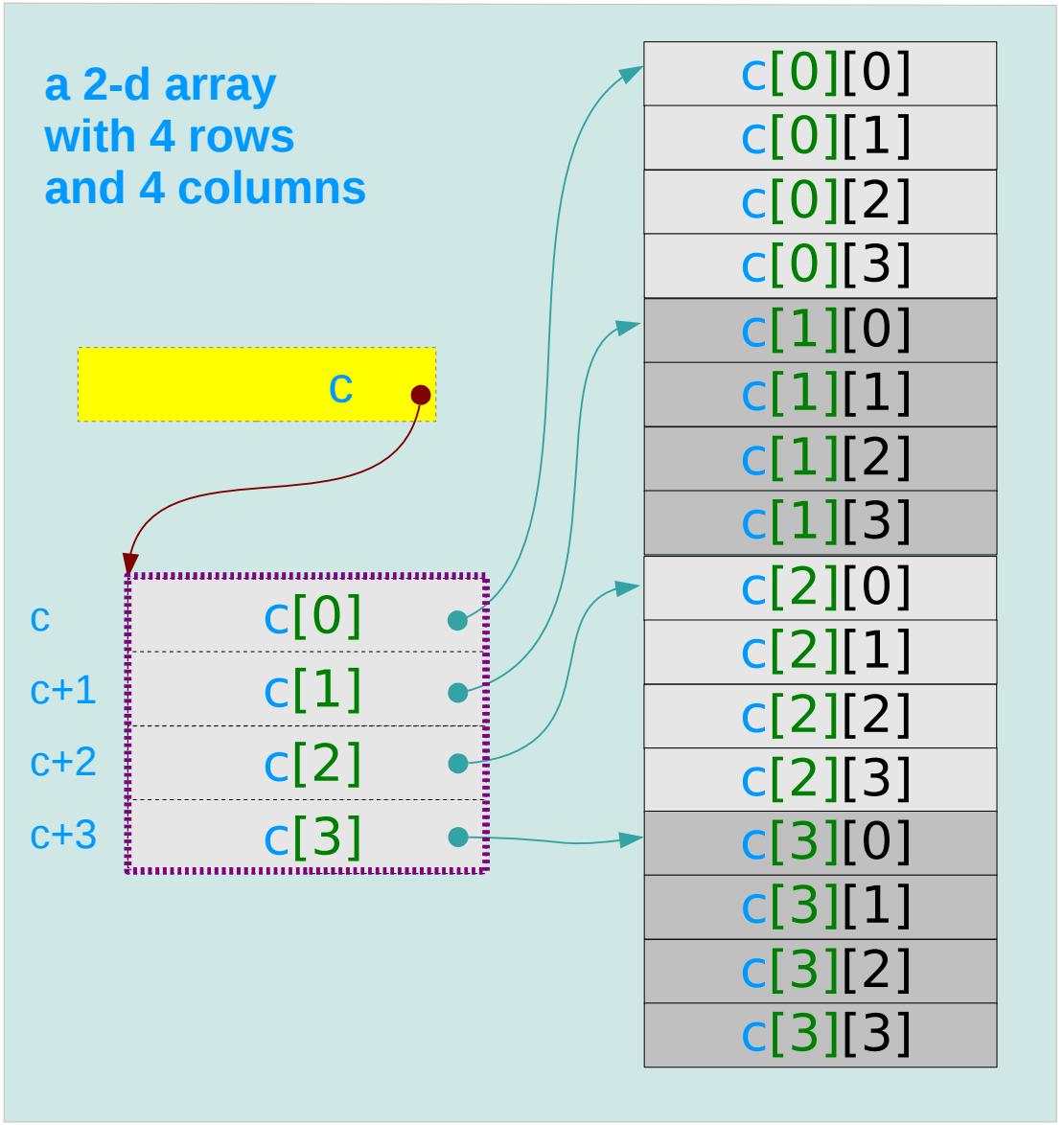
p = &a

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

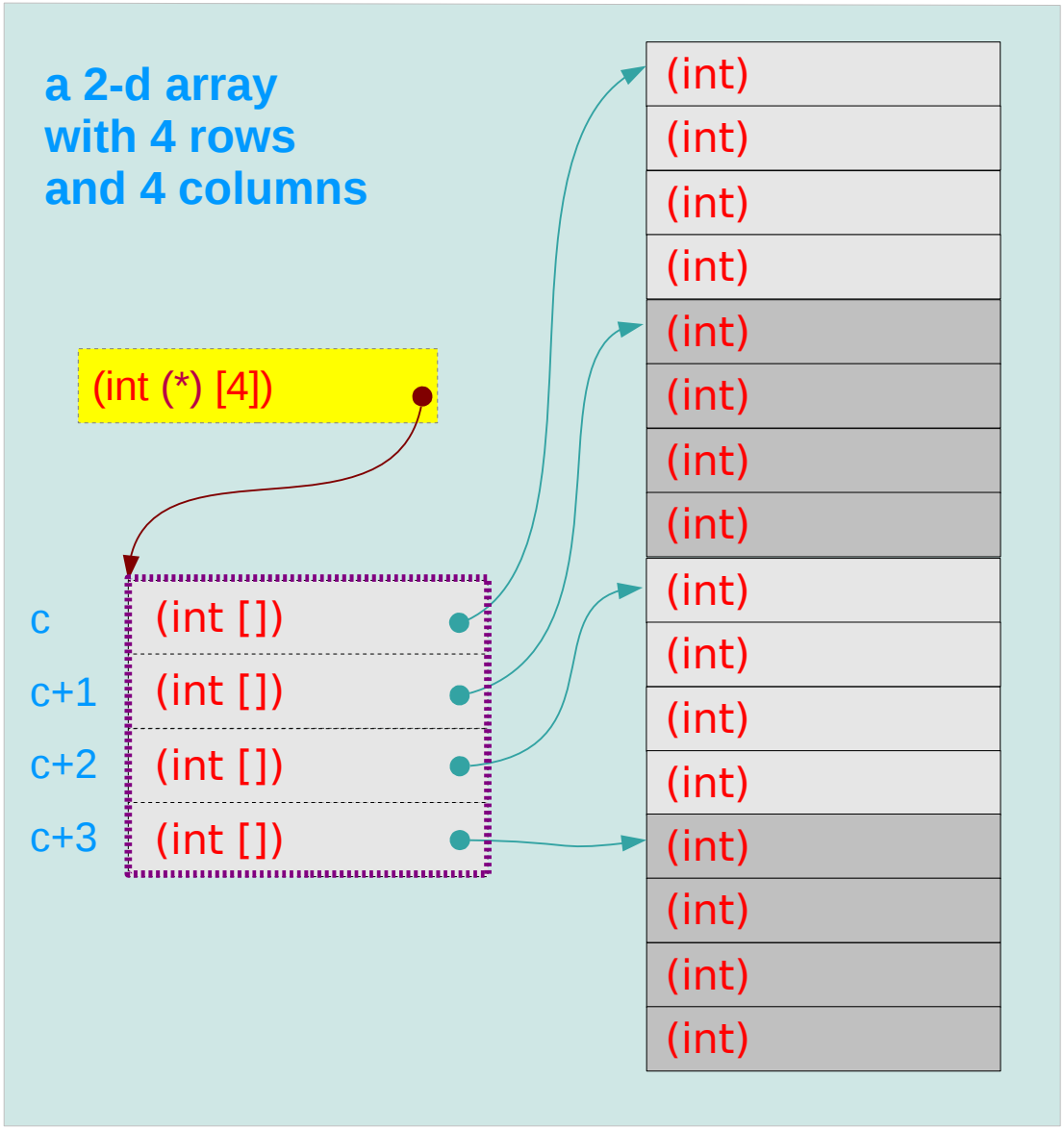
the array name **c** of a 2-d array as an array pointer which points to its 1st 1-d sub-array of 4 elements.

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

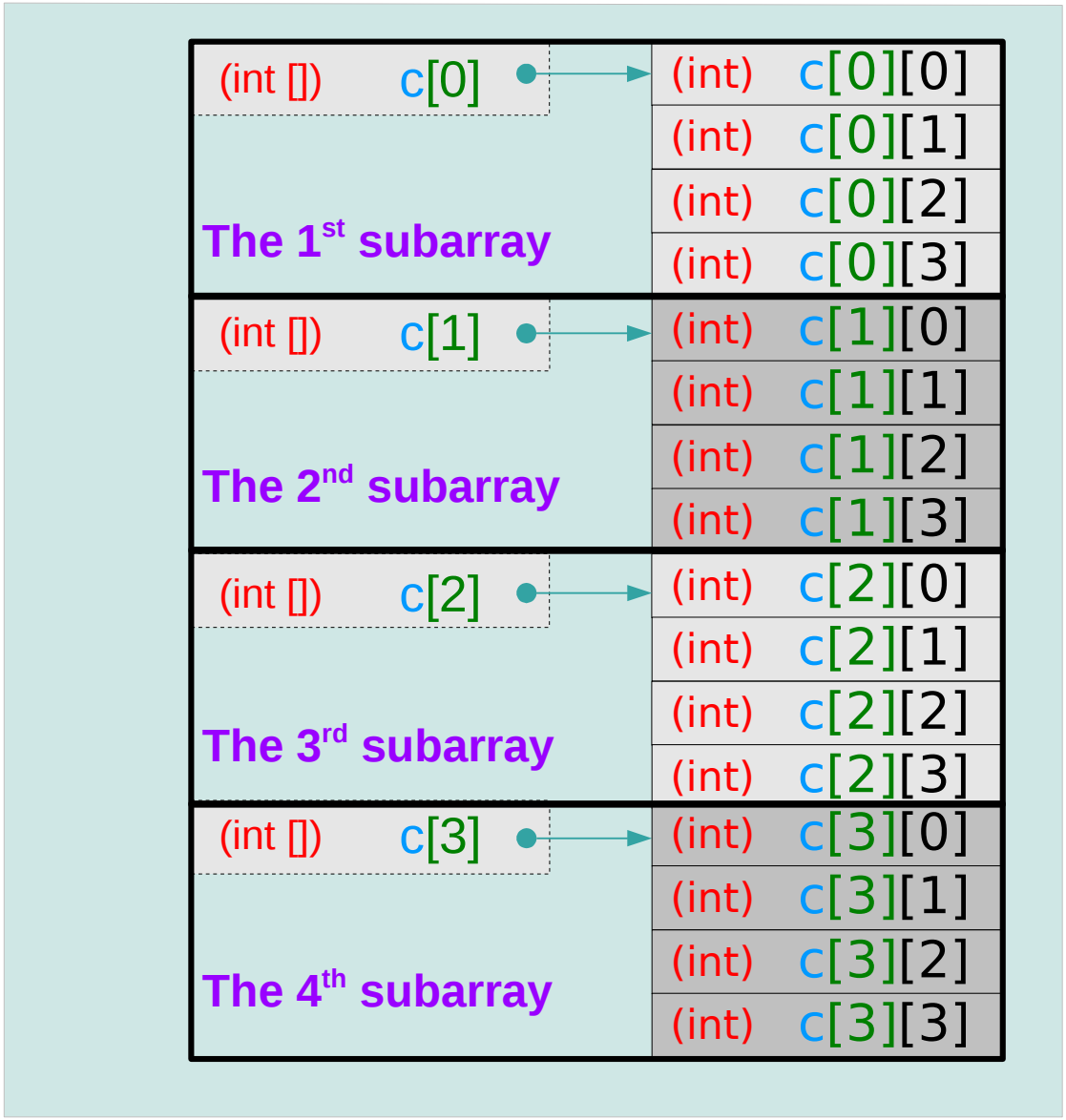


A 2-d array and its sub-arrays – a type view



1-d subarray aggregated data type

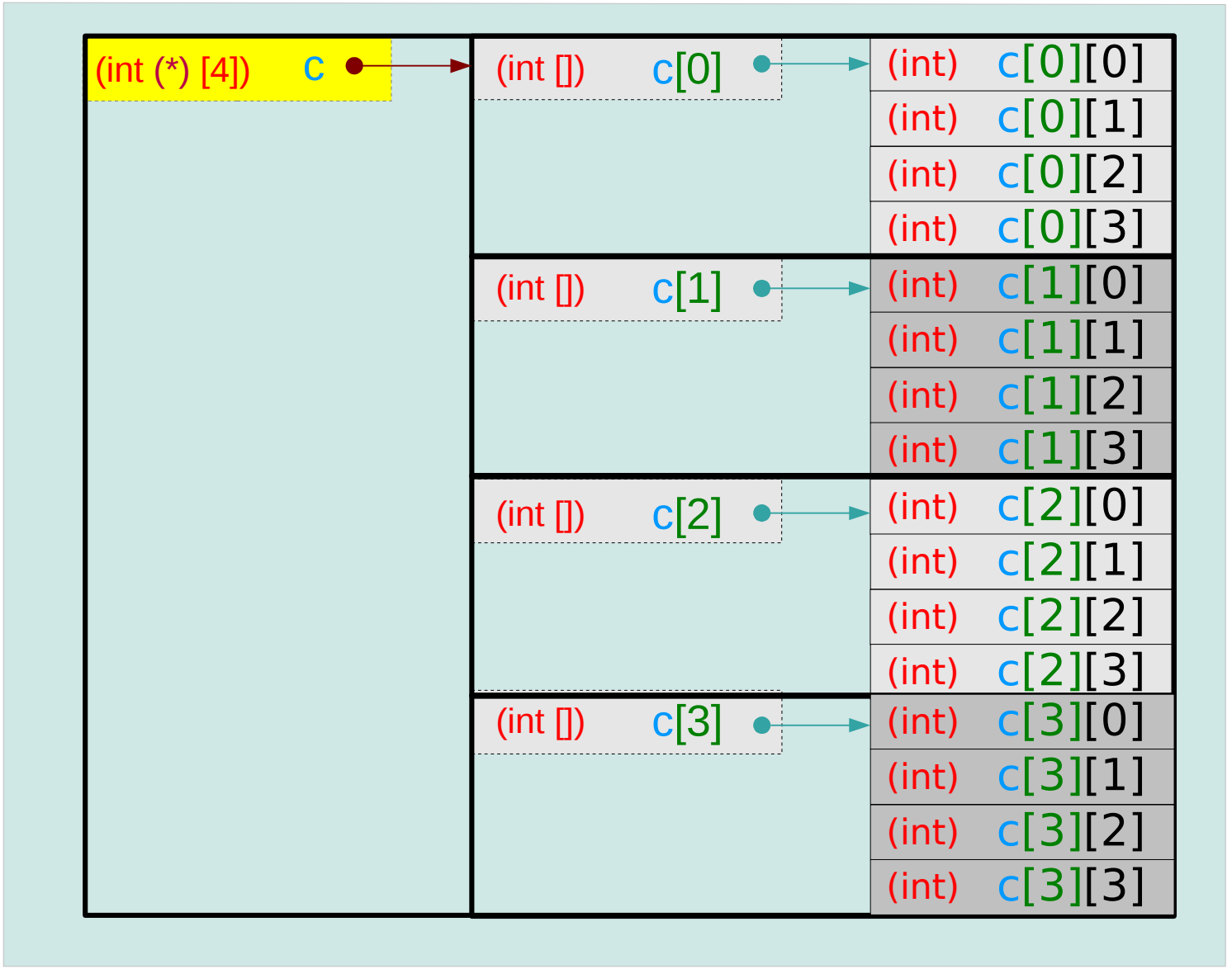
sizeof(c[0]) = 16 bytes
sizeof(c[1]) = 16 bytes
sizeof(c[2]) = 16 bytes
sizeof(c[3]) = 16 bytes



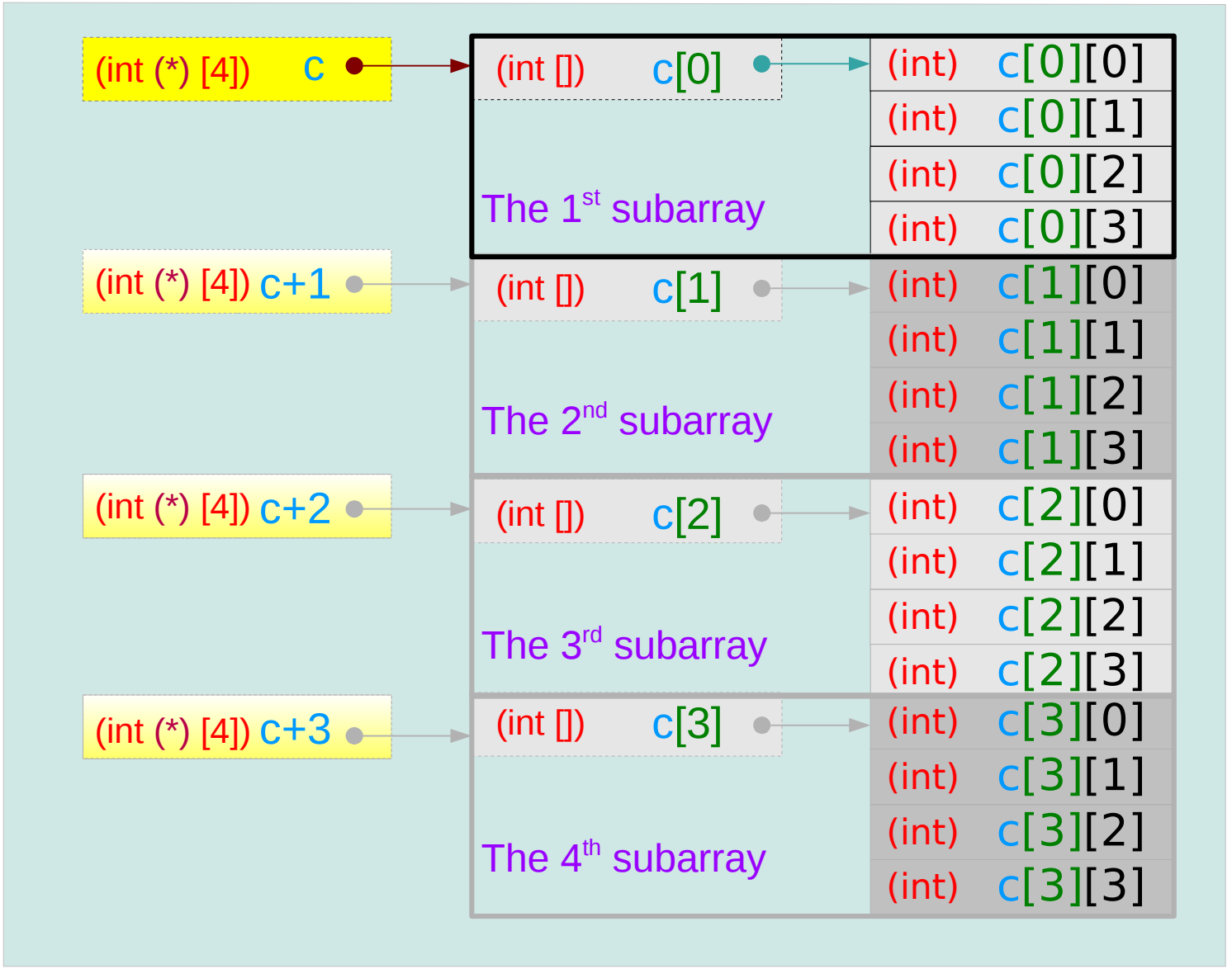
2-d subarray aggregated data type

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

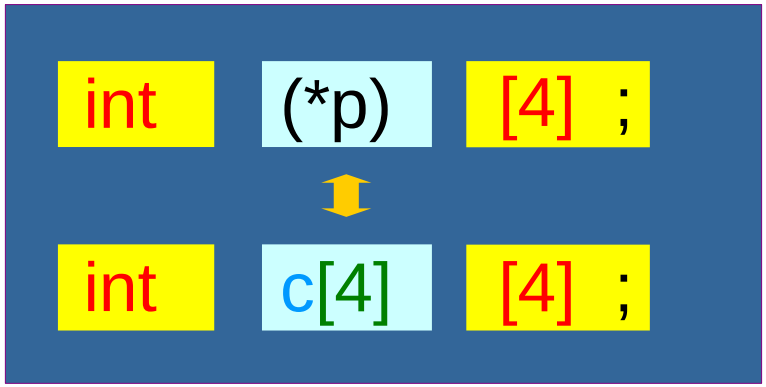
1-d sub-arrays :
sizeof(*c) = 16 bytes

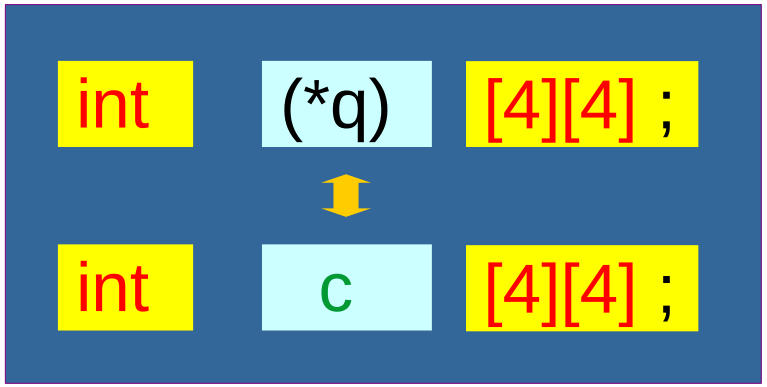


2-d array name as a pointer to a 1-d subarray



Assignment of array pointer variables





(int (*) [4])

```
p = &c[0];
```

```
p = c;
```

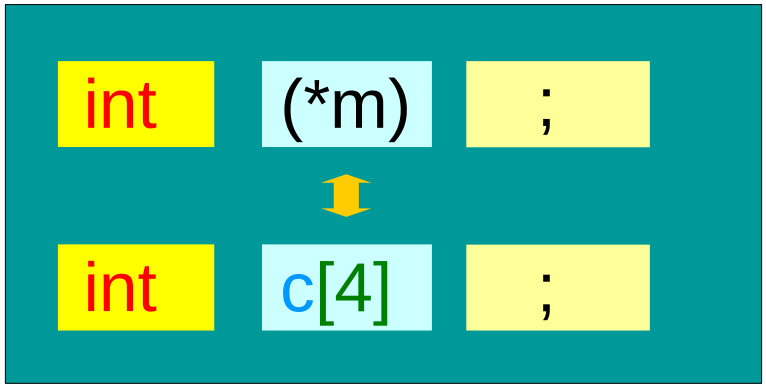
(int(*)[4][4])

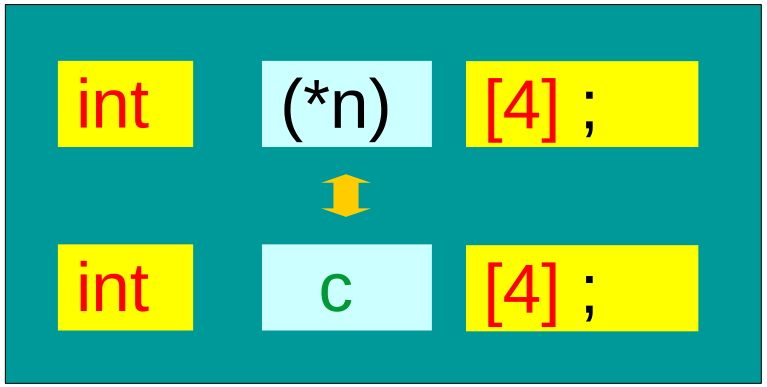
```
q = &c;
```

p[0] ≡ c[0]
p[1] ≡ c[1]
p[2] ≡ c[2]
p[3] ≡ c[3]

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

Assignment of array pointer variables





(int (*))

```
m = &c[0];
```

```
m = c;
```

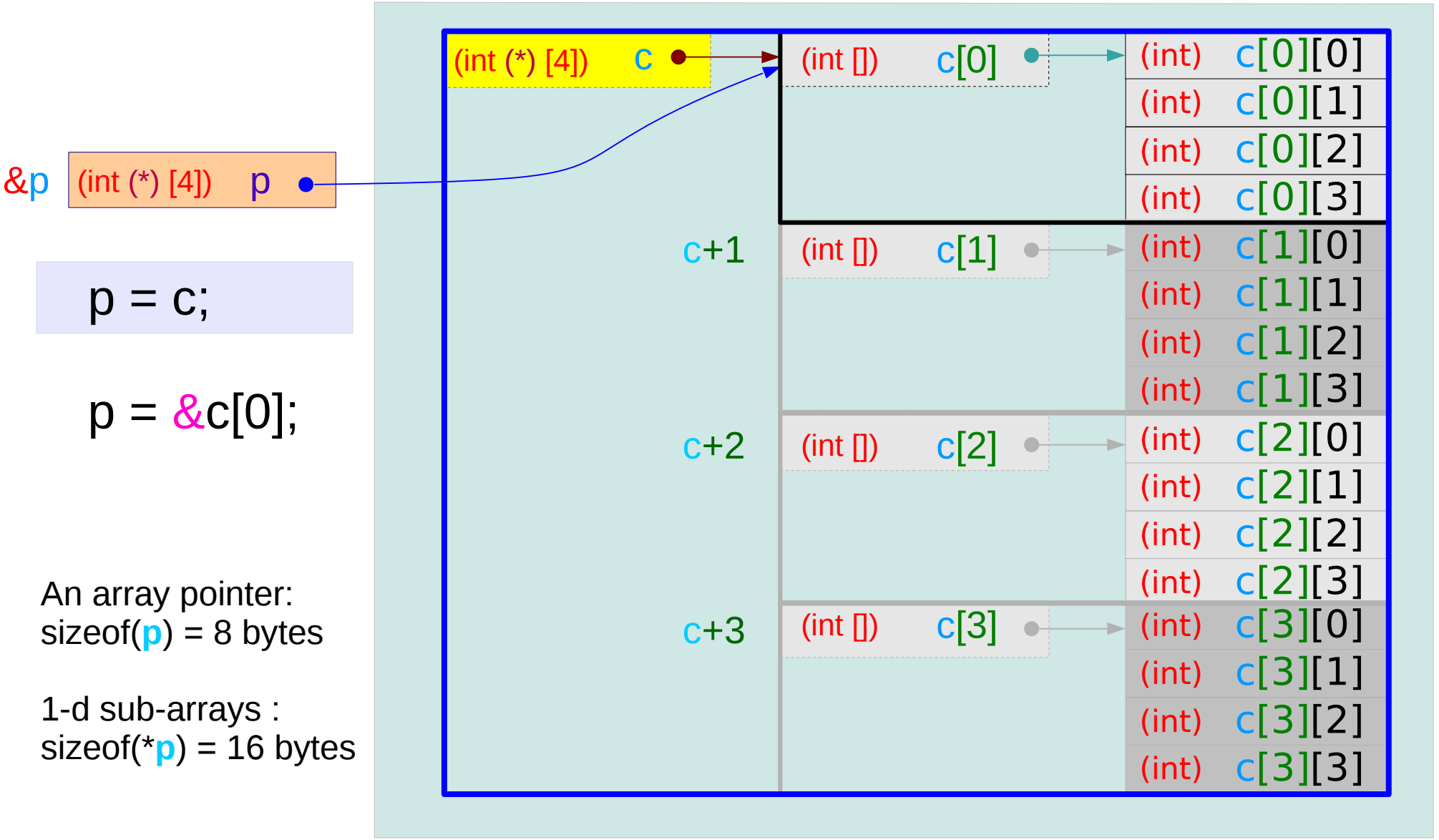
(int(*)[4])

```
n = &c;
```

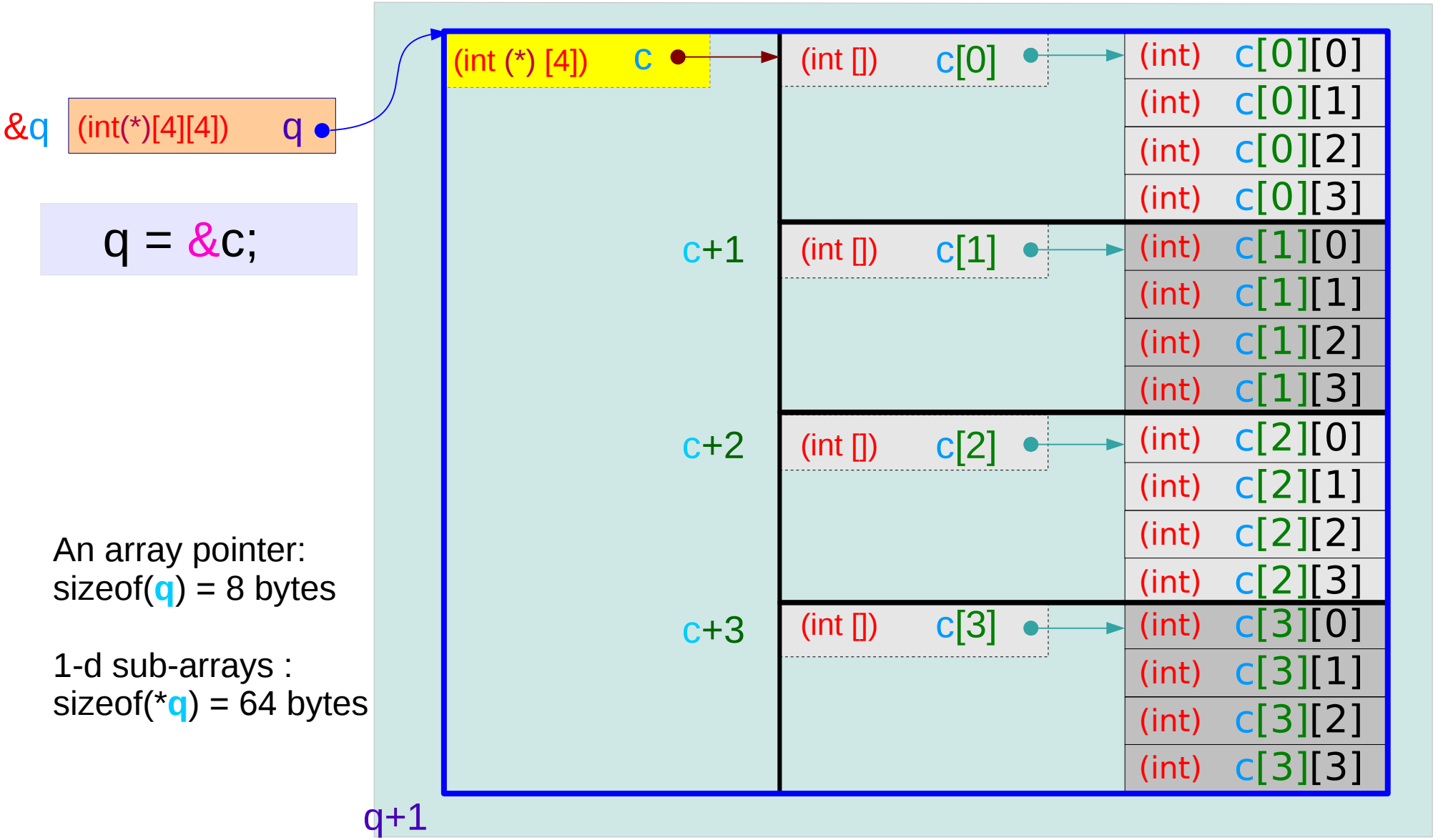
m[0] ≡ c[0]
m[1] ≡ c[1]
m[2] ≡ c[2]
m[3] ≡ c[3]

(*n)[0] ≡ c[0]
(*n)[1] ≡ c[1]
(*n)[2] ≡ c[2]
(*n)[3] ≡ c[3]

Pointer variable to a 1-d array



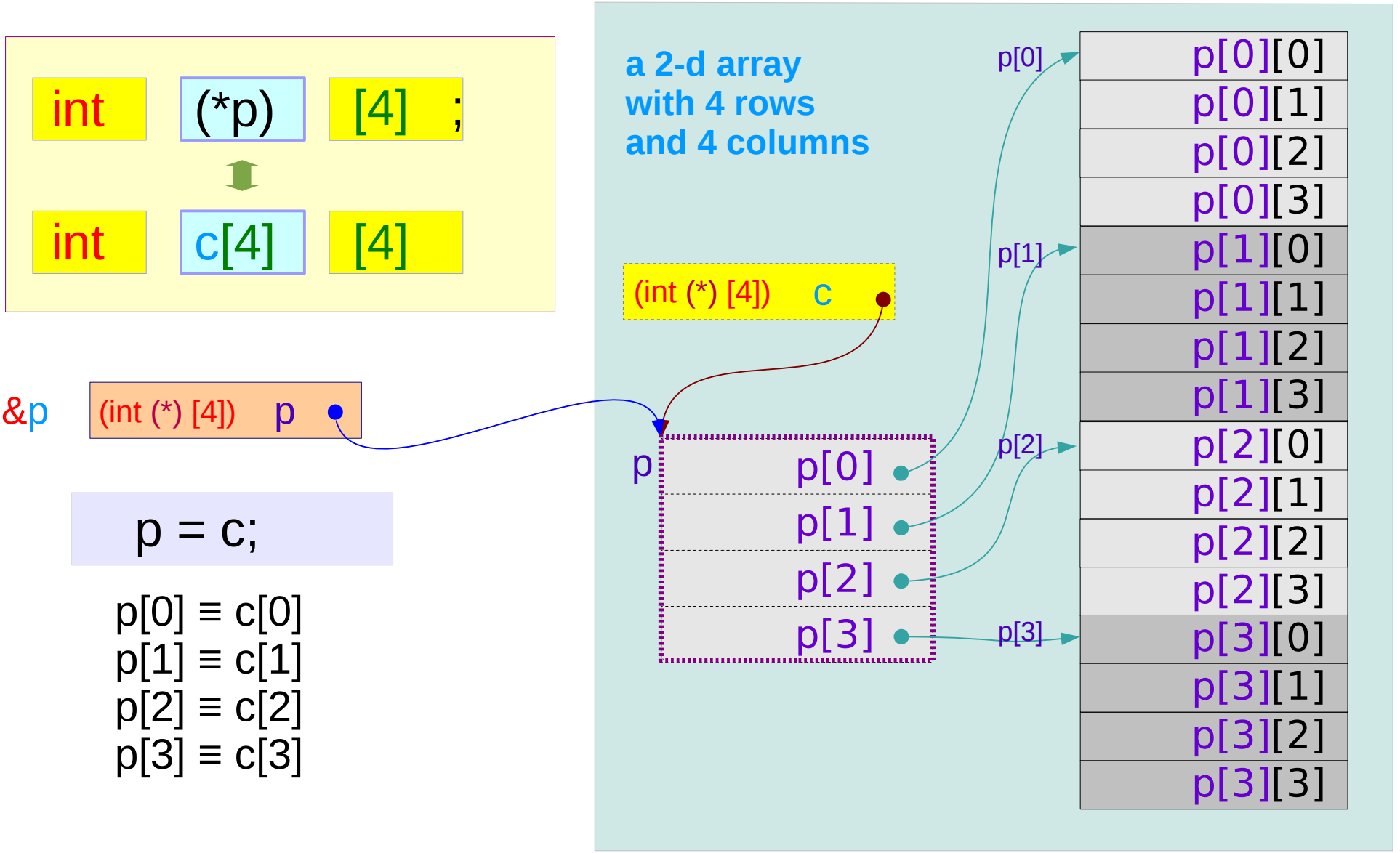
Pointer variable to a 2-d array



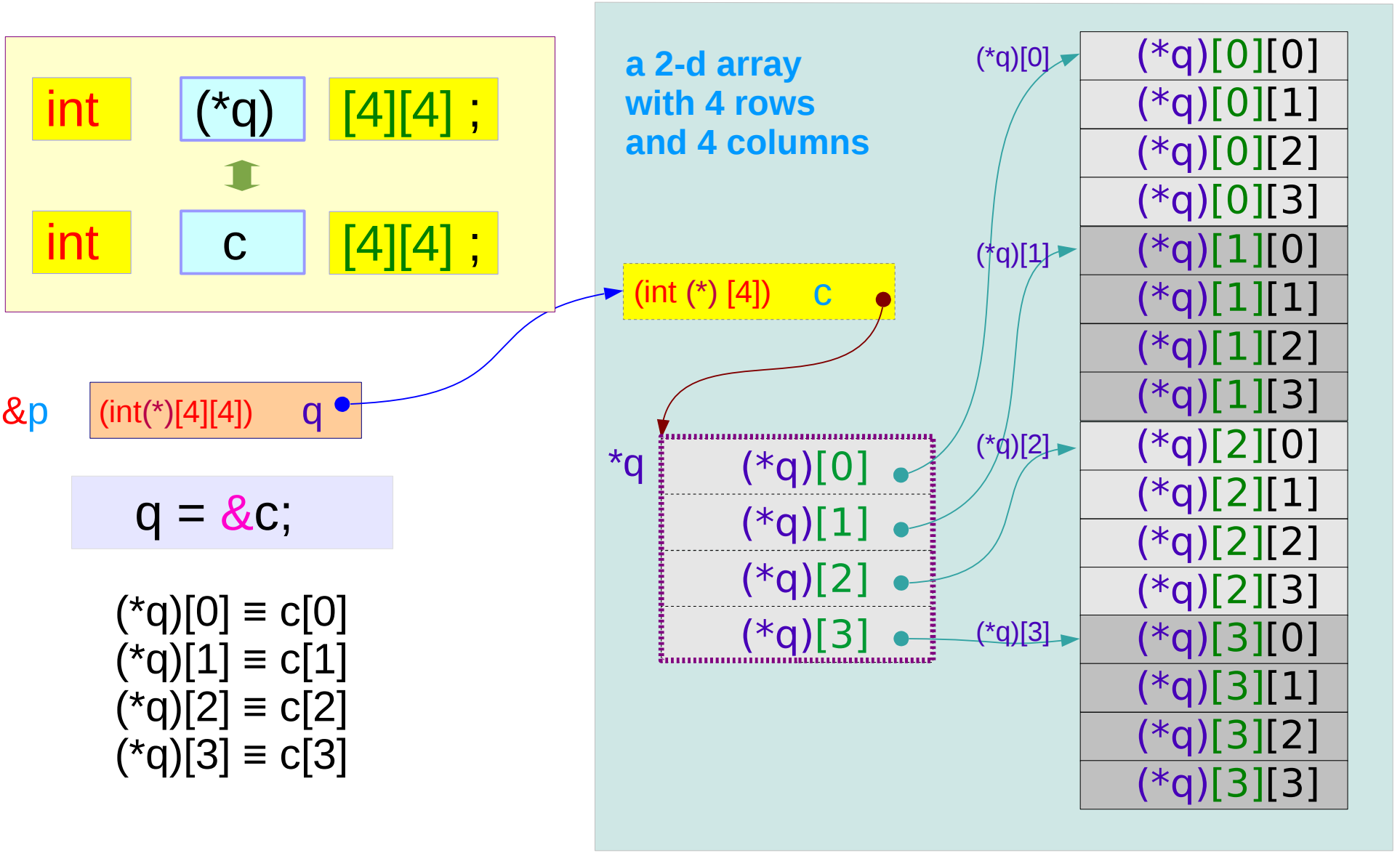
An array pointer:
sizeof(**q**) = 8 bytes

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 multi-dimensional arrays (1)

```
int a[4] ;
int (*p) ;
```
A pointer to a 0-d array (an integer)
can be viewed as a 1-d array name

```
int b[4] [2];
int (*q) [2];
```
A pointer to a 1-d array
can be viewed as a 2-d array name

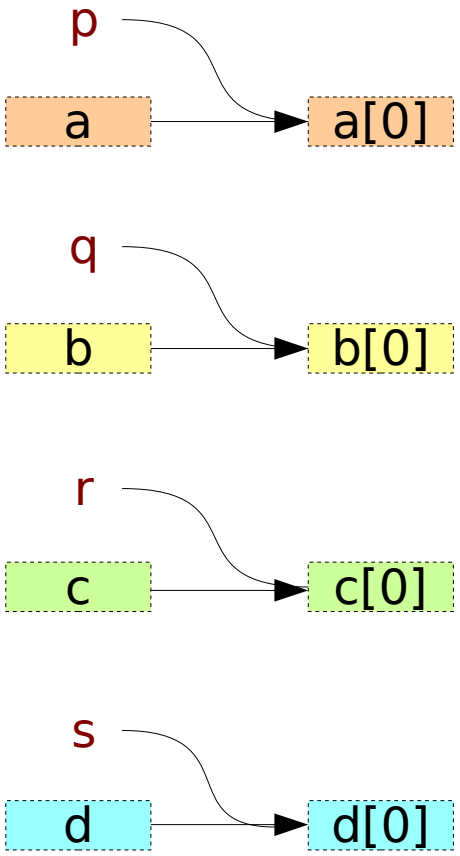
```
int c[4] [2][3];
int (*r) [2][3];
```
A pointer to a 2-d array
can be viewed as a 3-d array name

```
int d[4] [2][3][4];
int (*s)  [2][3][4];
```
A pointer to a 3-d array
can be viewed as a 4-d array name

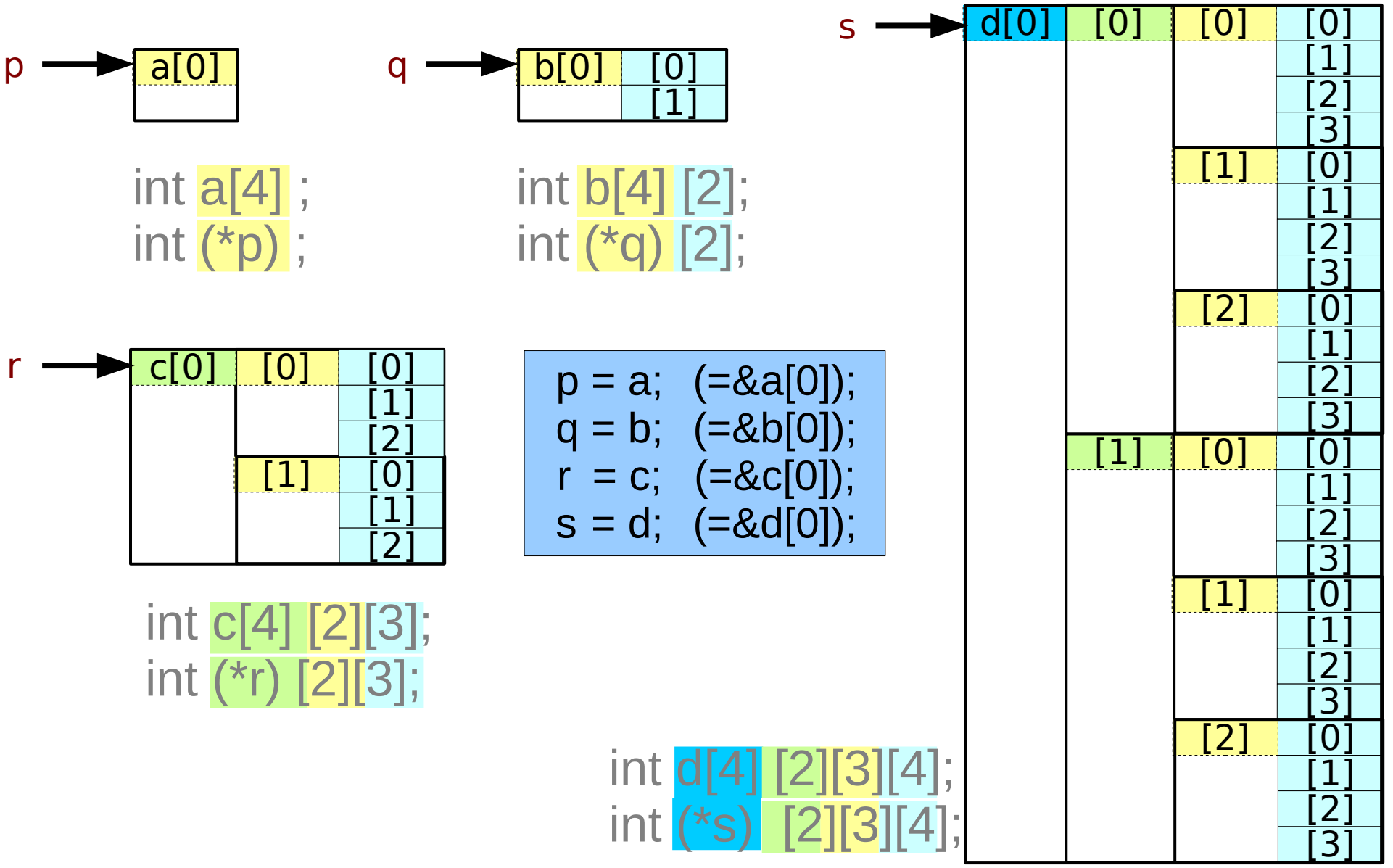
Pointer to multi-dimensional arrays (2)

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

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

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

int d[4] [2][3][4];
int (*s)  [2][3][4];
```

```
p = &a[0];
p = a;

q = &b[0];
q = b;

r = &c[0];
r = c;

s = &d[0];
s= d;
```



Pointer to multi-dimensional arrays (3)



To pass array name

```
int a[4] ;                 prototype  void func(int (*p), …);
int (*p) ;                     call   func(a, …);

int b[4] [2];              prototype  void func(int (*q)[2], …);
int (*q) [2];                  call   func(b, …);

int c[4] [2][3];           prototype  void func(int (*r)[2][3], …);
int (*r)  [2][3];              call   func(c, …);

int d[4] [2][3][4];        prototype  void func(int (*s)[2][3][4], …);
int (*s) [2][3][4];            call   func(d, …);
```

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