Array Pointers (1A)

Young Won Lim
10/12/20

Copyright (c) 2010 - 2020 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

```
#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);
```

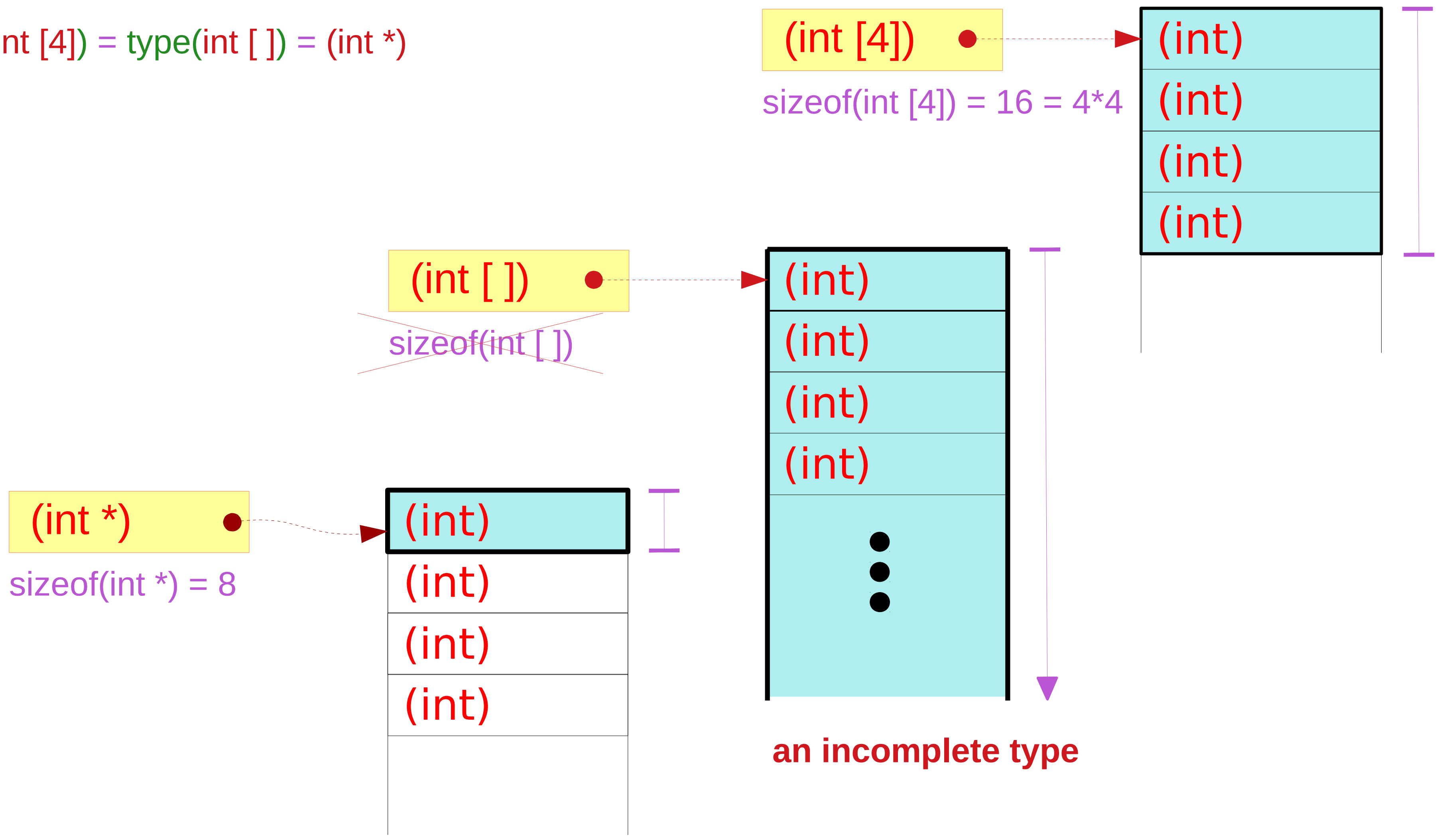
int *

int [N]

int [ ]

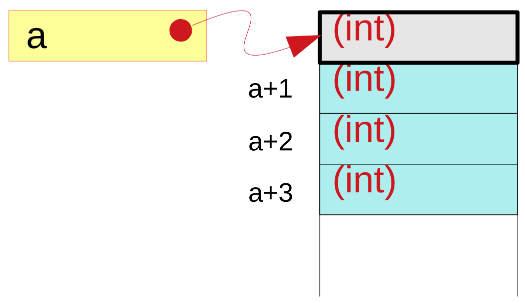
Differences in pointer types – **int [4]**, **int [ ]**, **int ***

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



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

int *a;

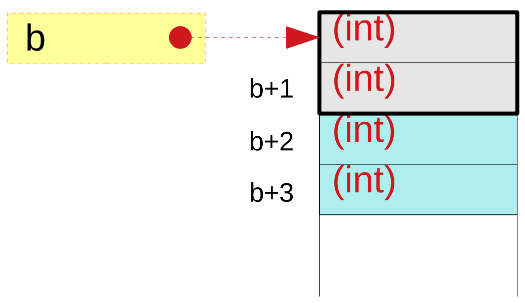


a[0] = *a
a[1] = *(a+1)
a[2] = *(a+2)
a[3] = *(a+3)

syntactically legitimate

programmers must ensure their validity

int b[2]

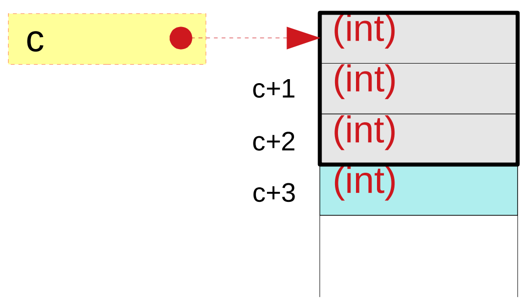


b[0] = *b
b[1] = *(b+1)
b[2] = *(b+2)
b[3] = *(b+3)

syntactically legitimate

programmers must ensure their validity

int c[3];



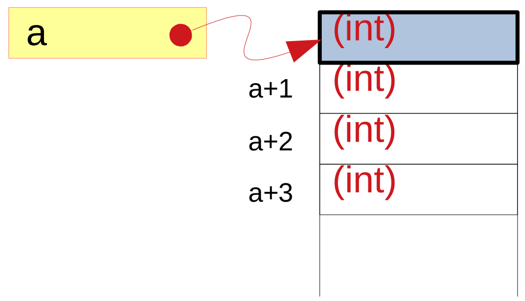
c[0] = *c
c[1] = *(c+1)
c[2] = *(c+2)
c[3] = *(c+3)

syntactically legitimate

programmers must ensure their validity

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

int *a;



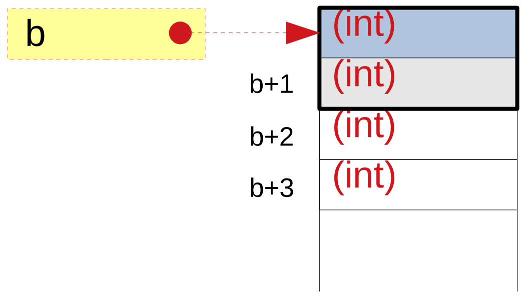
a[0] = *a

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

value(&a) ≠ value(a)

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

int b[2]



b[0] = *b

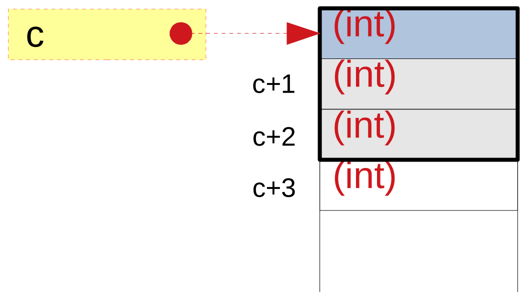
type(b) = int *
type(&b) = int (*) [2]

value(&b) = value(b)

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

&b and b evaluate
the <u>same</u> address
but have <u>different</u> types
and also <u>different</u> sizes

int c[3];



c[0] = *c

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

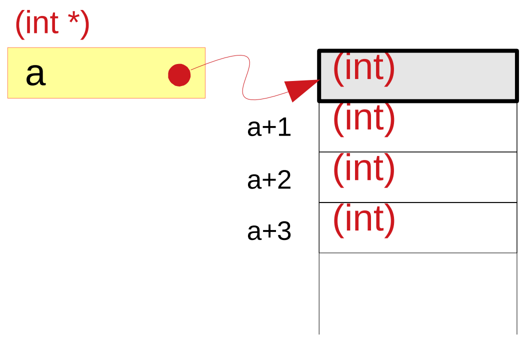
value(&c) = value(c)

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

&c and c evaluate
the <u>same</u> address
but have <u>different</u> types
and also <u>different</u> sizes

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

int *a;



sizeof (a) = pointer size

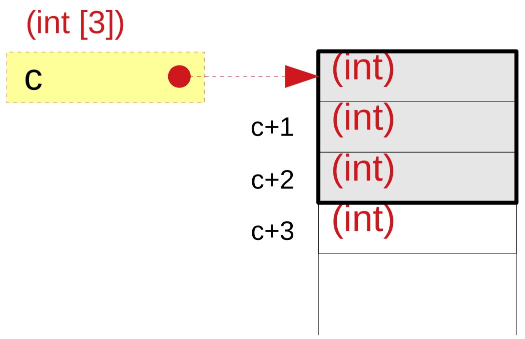
value(&a) ≠ value(a)

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

real memory location for a

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

int c[3];



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

value(&c) = value(c)

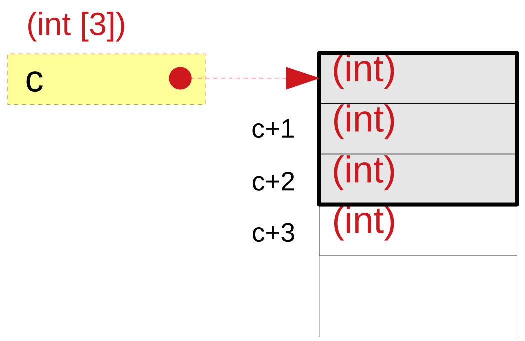
the starting address of array variable c
is equal to the address of the 1st element

no actual memory location for c

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

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

int c[3];



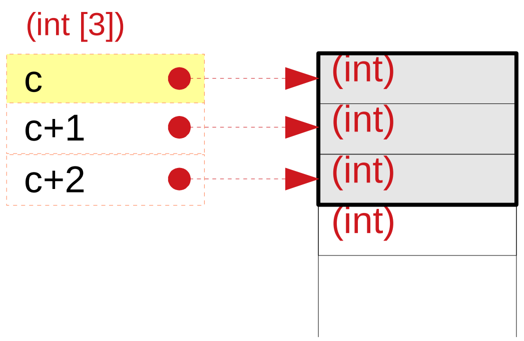
sizeof (c) = sizeof(int) * 3

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]

Types of multi-dimension array names

```
int      a ;
int      b [4];
int      c [4][5];
int      d [4][5][6];
```

array types		array pointer types	
a :: int	→	int	
b :: int [4]	→	int (*)	int *
c :: int [4][5]	→	int (*)[5]	
d :: int [4][5][6]	→	int (*)[5][6]	

array types — *specific types*

array pointer types — *general type*

Array pointers v.s. Array

int (*) | int (*)[5] | int (*)[5][6] — general type

- int [2]
- int [3]
- int [4]

⋮

- int [2][5]
- int [3][5]
- int [4][5]

⋮

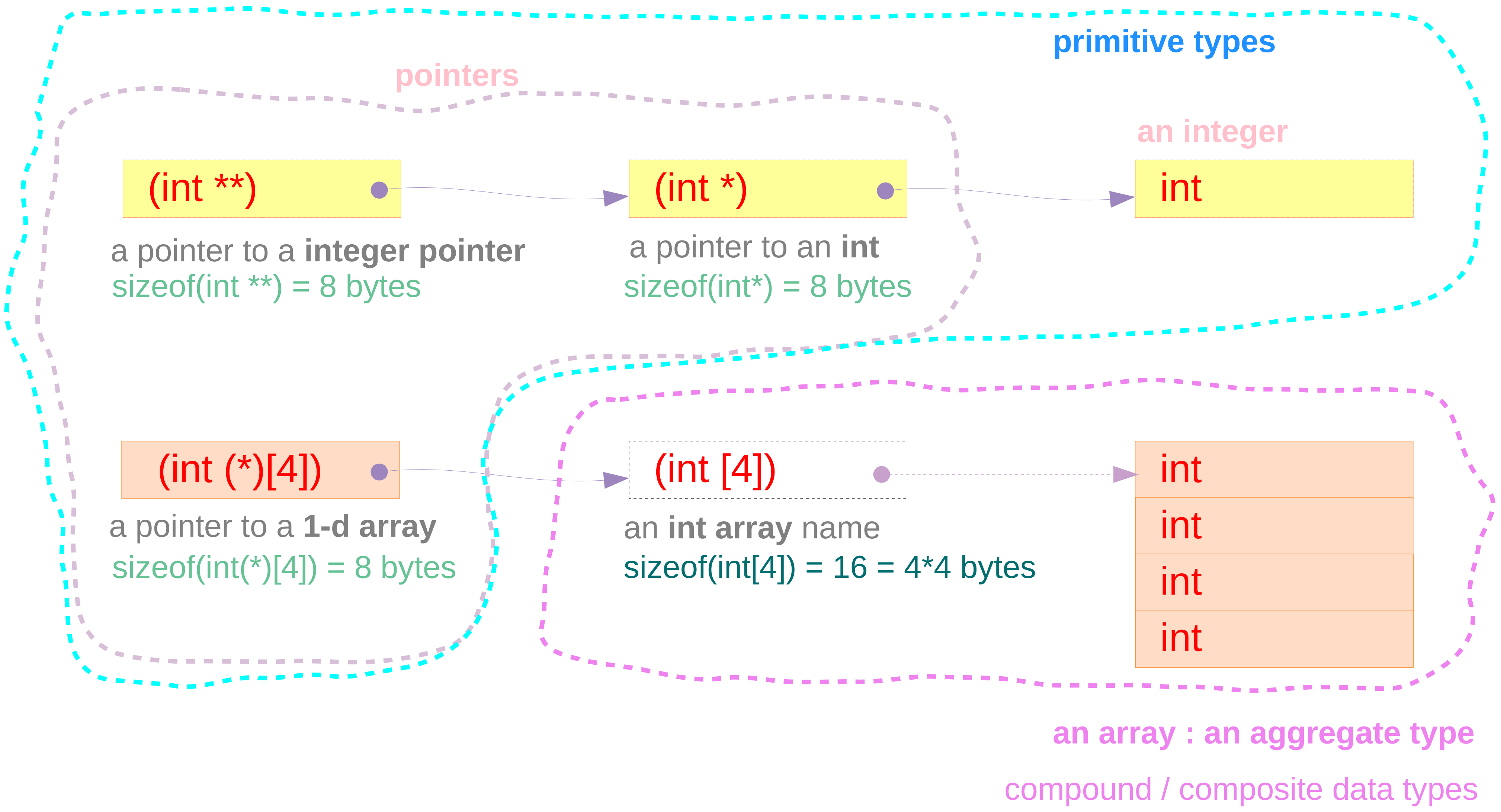
- int [2][5][6]
- int [3][5][6]
- int [4][5][6]

⋮

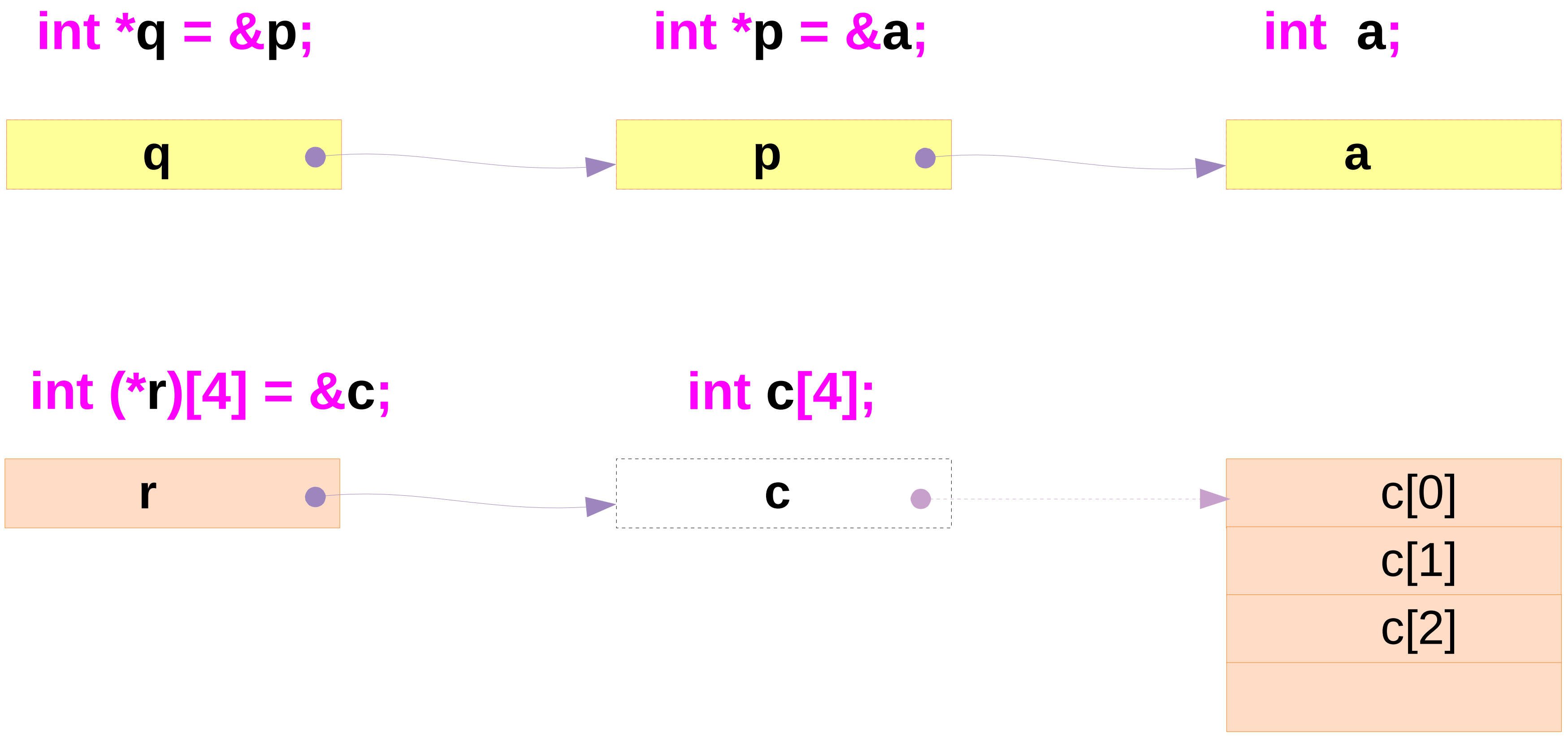
specific types

$$\begin{tikzcd}
\texttt{int **} \arrow[r] & \texttt{int *} \arrow[r] & \texttt{int} \\
\texttt{int (*) [4]} \arrow[r] & \texttt{int [4]} \arrow[r] & \texttt{int} \\
 & & \texttt{int} \\
 & & \texttt{int} \\
 & & \texttt{int}
\end{tikzcd}$$

Types of integer pointers


primitive types
pointers
an integer
(int **)
(int *)
int
a pointer to a **integer pointer**
sizeof(int **) = 8 bytes
a pointer to an **int**
sizeof(int*) = 8 bytes
(int (*)[4])
(int [4])
int
int
int
int
a pointer to a **1-d array**
sizeof(int(*)[4]) = 8 bytes
an **int array** name
sizeof(int[4]) = 16 = 4*4 bytes
an array : an aggregate type
compound / composite data types

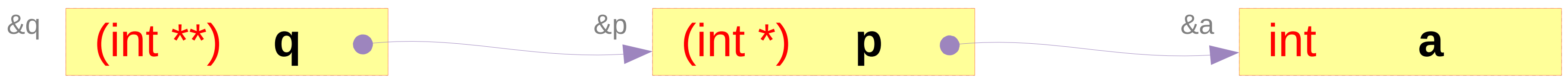
Variable declaration of integer pointers



Types and sizes of integer pointers

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

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

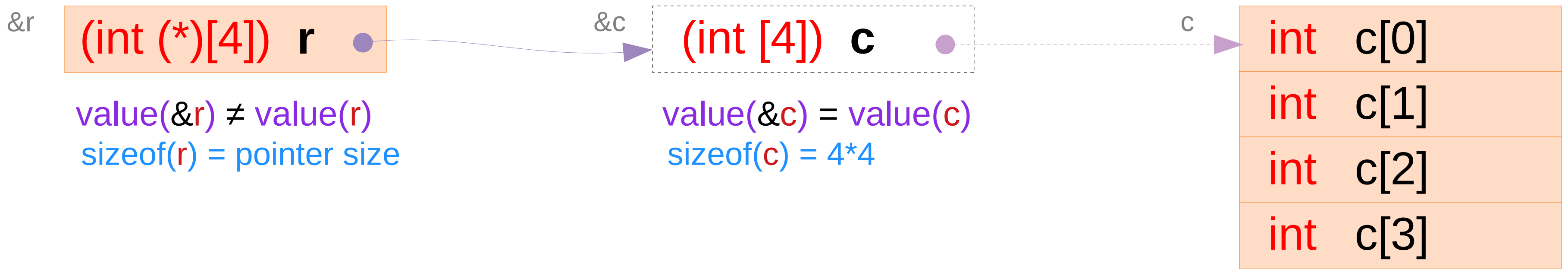


value(&q) ≠ value(q)
sizeof(q) = pointer size

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

value(&a) ≠ value(a)
sizeof(a) = 4

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



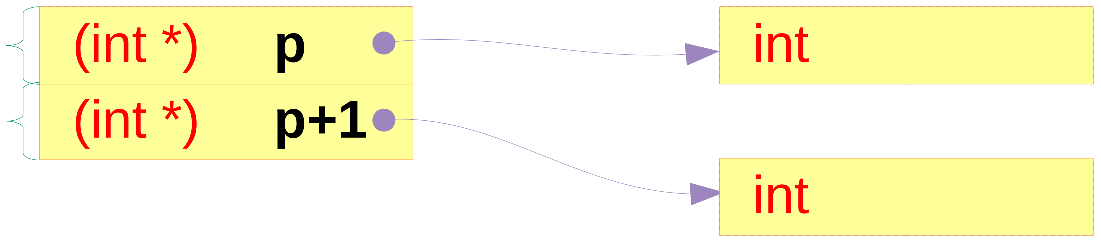
value(&r) ≠ value(r)
sizeof(r) = pointer size

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

Sizes of integer pointers

a pointer to an **int**

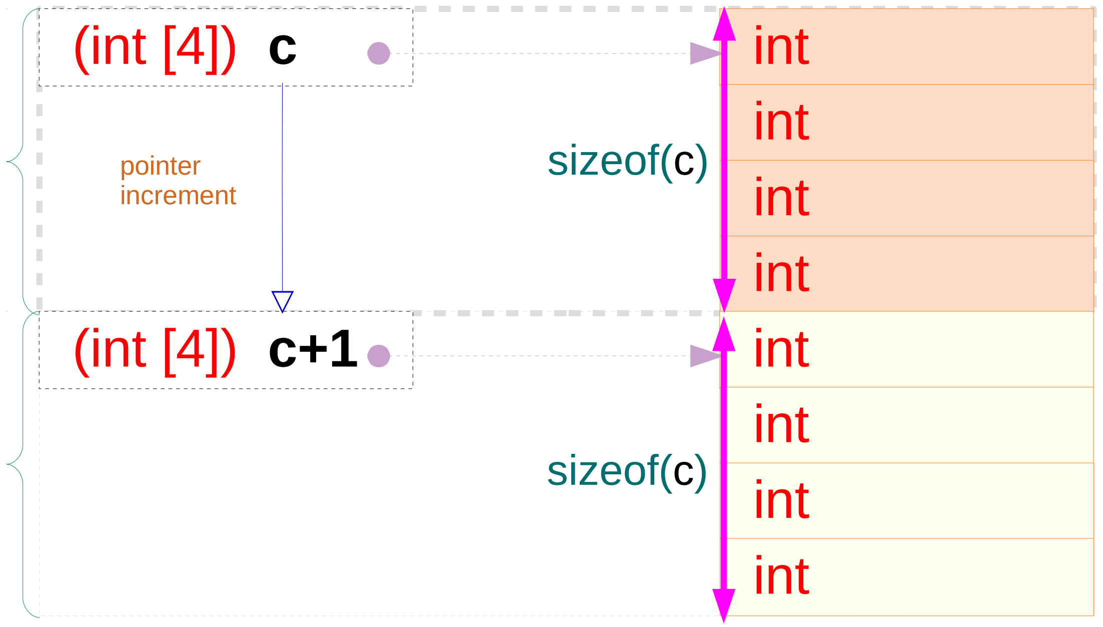
sizeof(p) = pointer size
= 8 bytes on 64-bit machine
= 4 bytes on 32-bit machine



an **int array** name

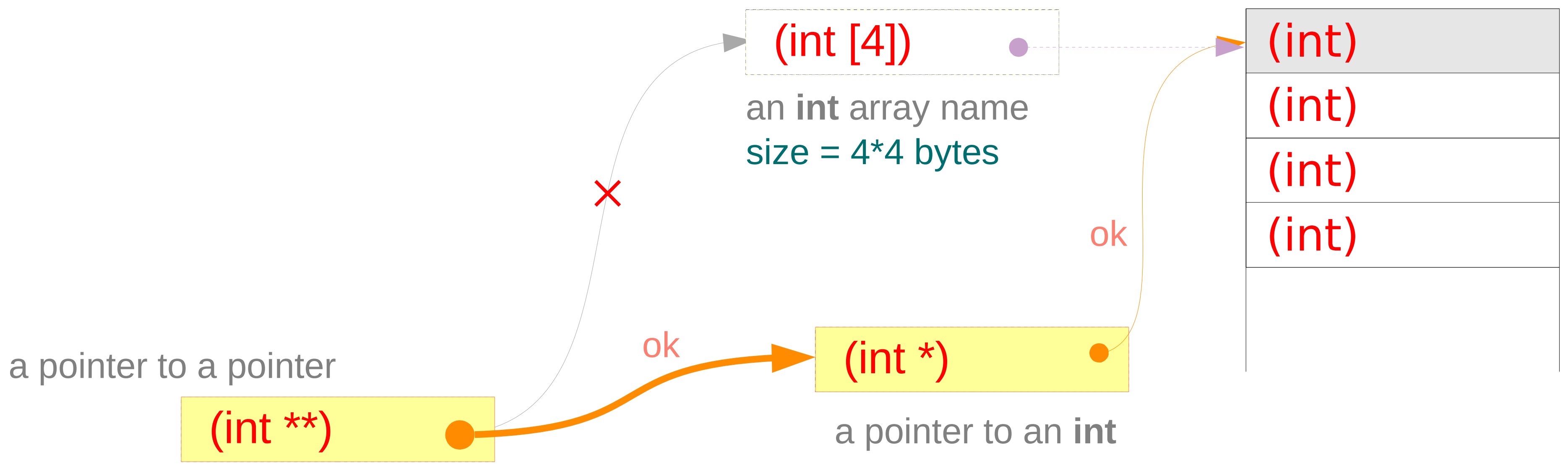
an array :
an aggregate type

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



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

Double integer pointer type – (int **)



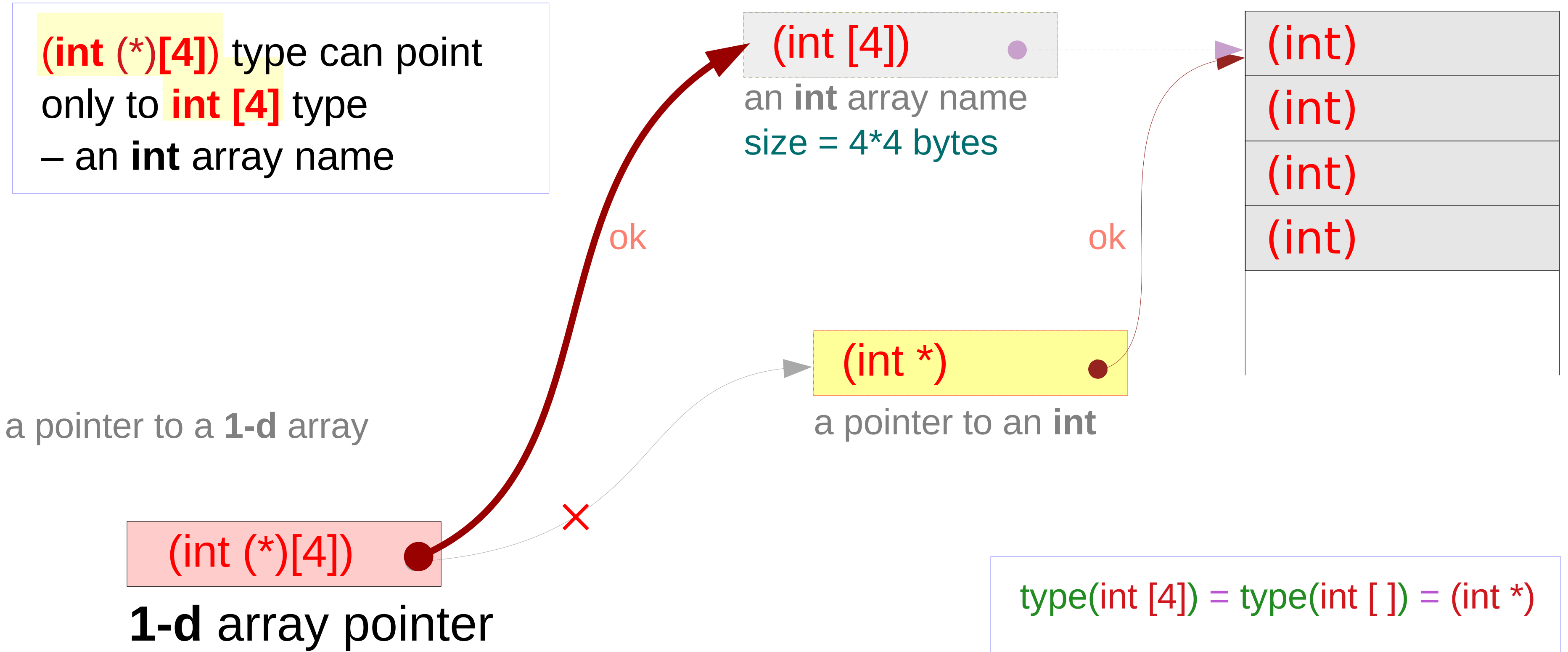
(int **) type can point only to **(int *)** type – an **int** array name (X)

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

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

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

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



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

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

Array Pointers

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 – 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]

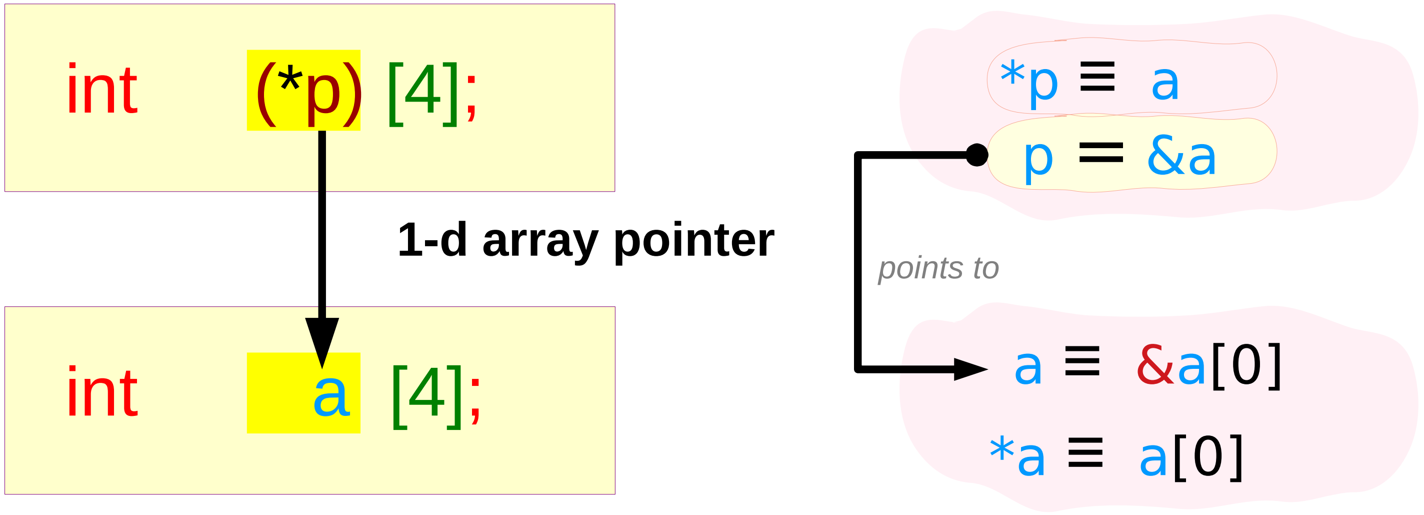
an array pointer

int (int, int) instructions

int (*) (int, int)

a function pointer

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



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

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

$*p \equiv a$ correspondence

$p = \&a$ assignment

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

$*a \equiv a[0]$

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

value(&a) = value(a)

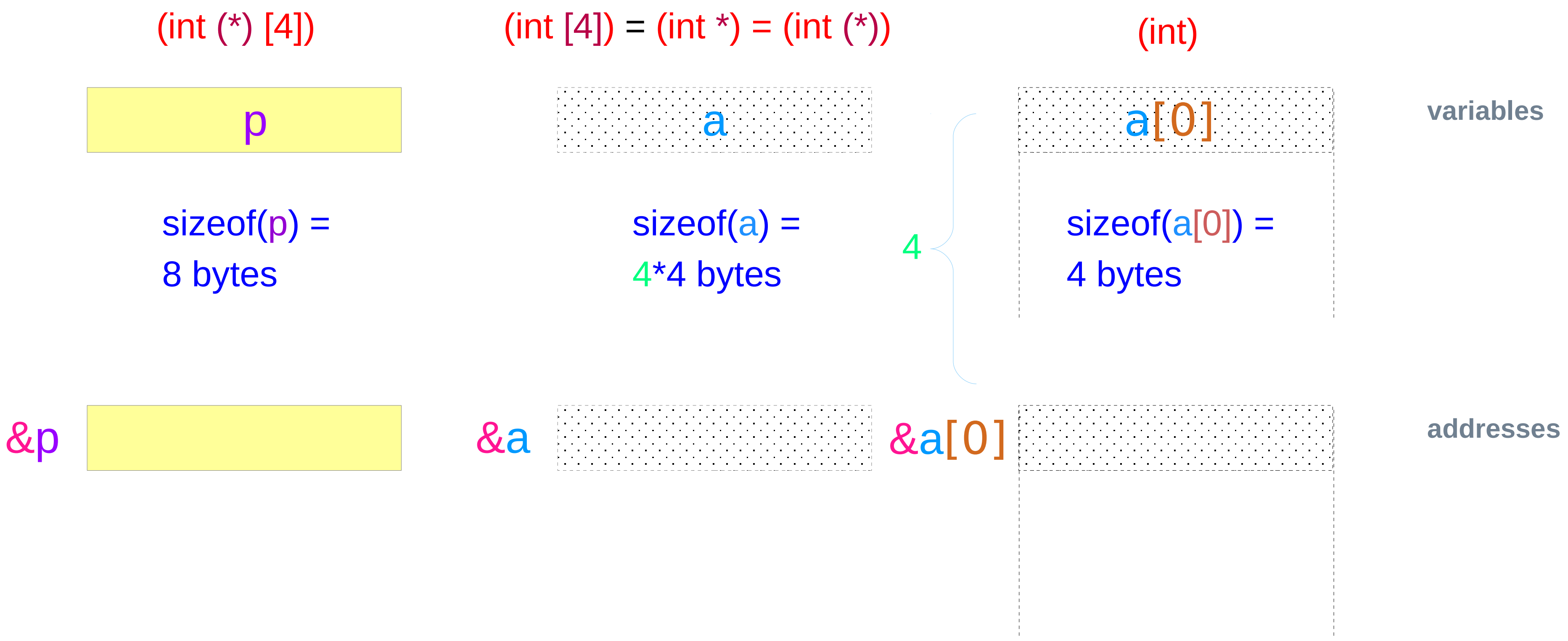
type(&a) ≠ type(a)

int (*)[4] ≠ int [4]

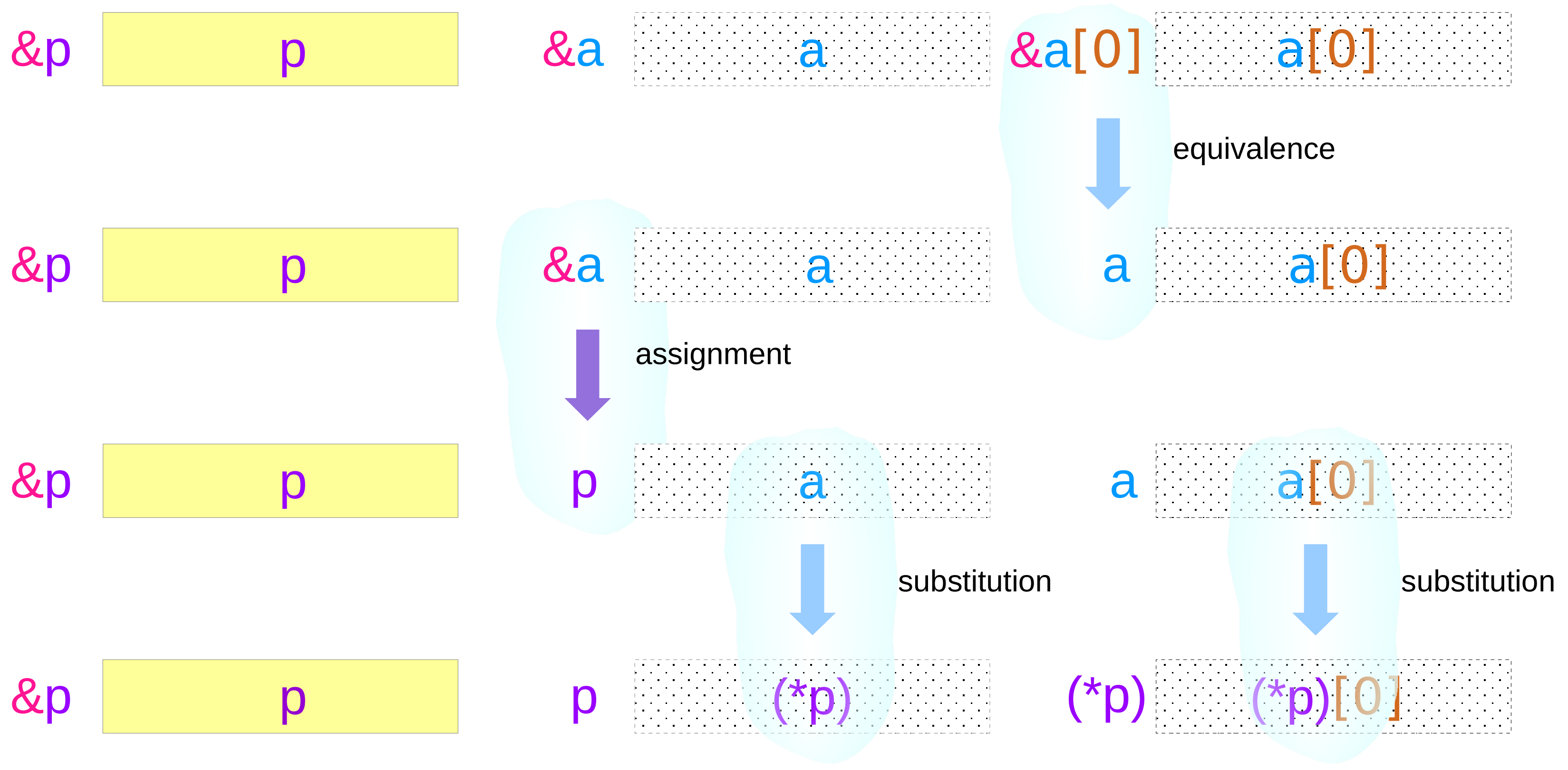
those values are evaluated as addresses

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

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



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

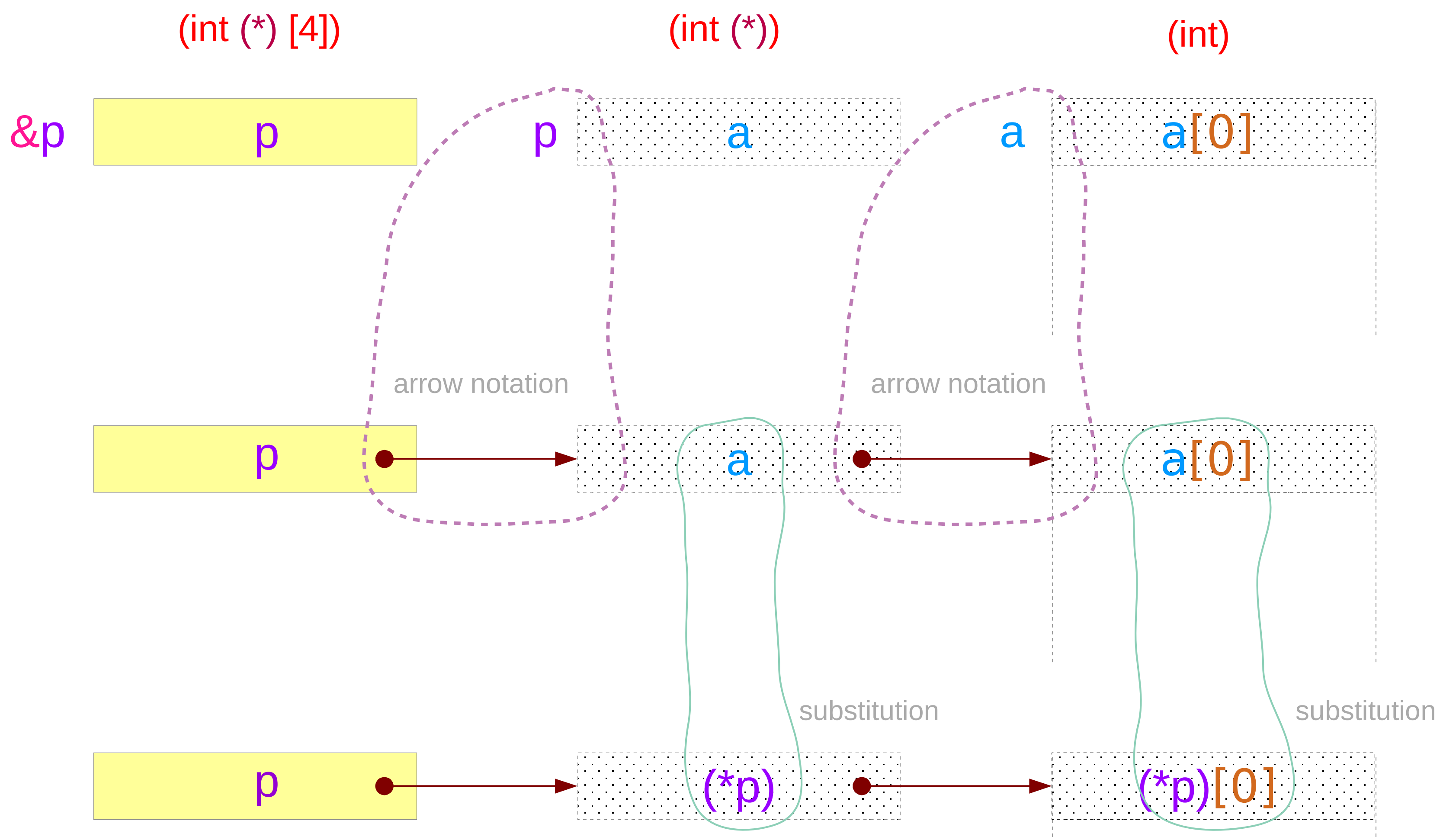


$a \equiv \&a[0]$

$p = \&a$

$(*p) \equiv a$

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

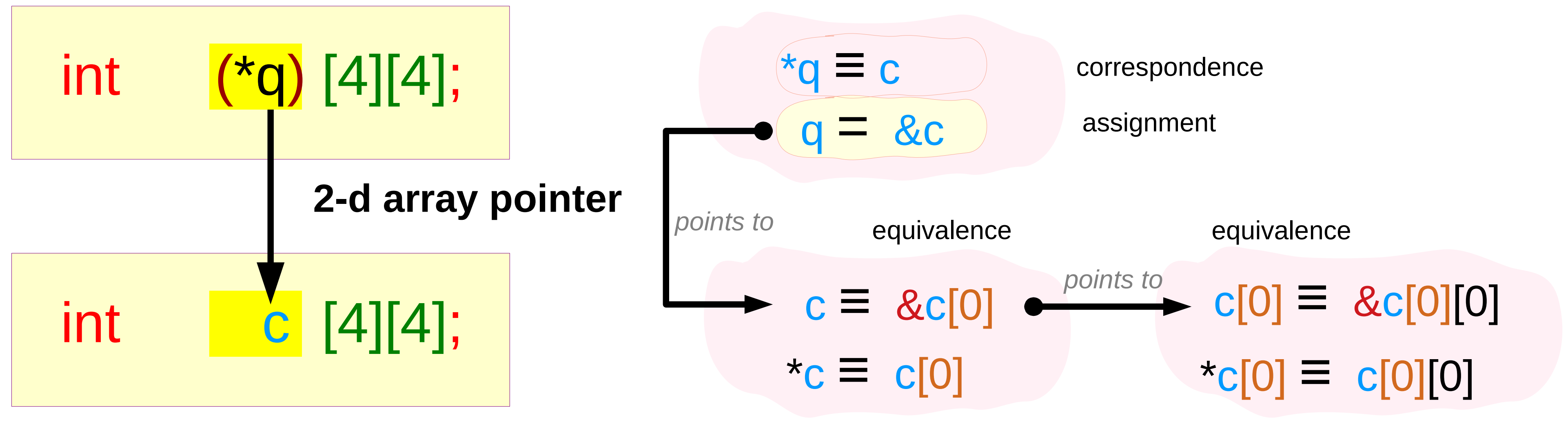


sizeof(p) =
8 bytes

sizeof(a) =
4*4 bytes

sizeof(a[0]) =
4 bytes

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



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

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

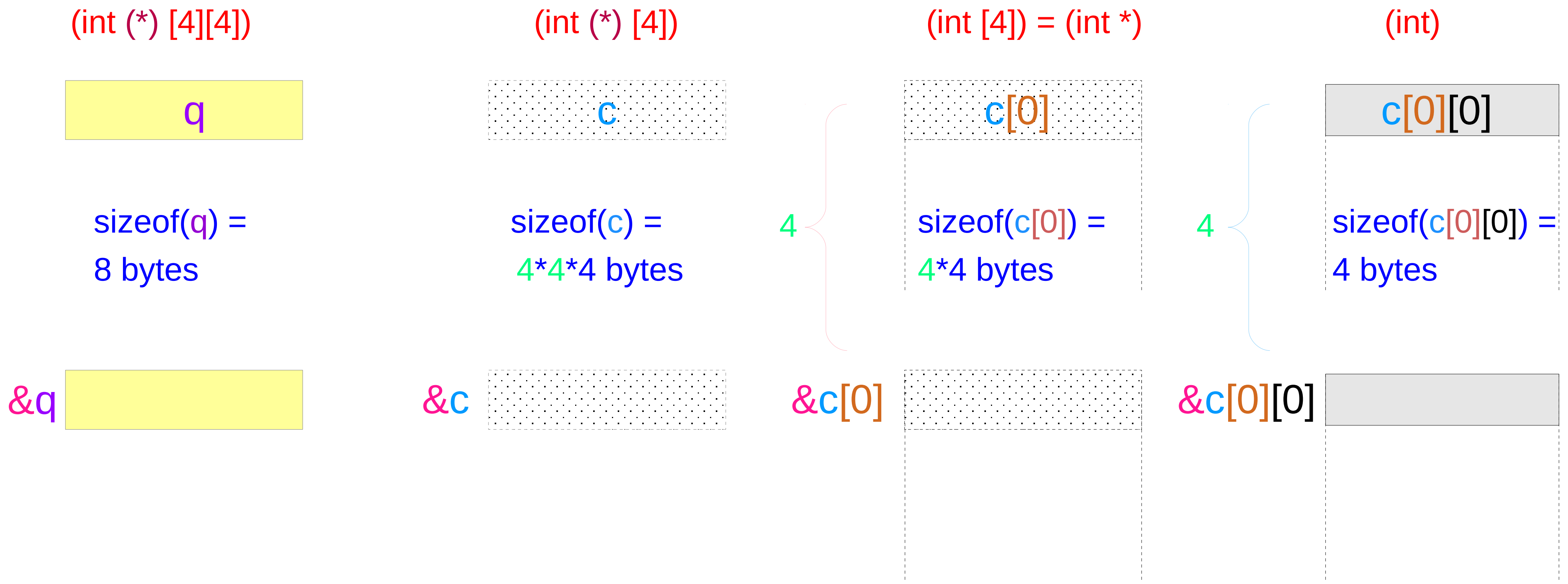
type(&c) ≠ type(c) ≠ type(c[0])

int (*)[4][4] ≠ int [4][4] ≠ int [4]

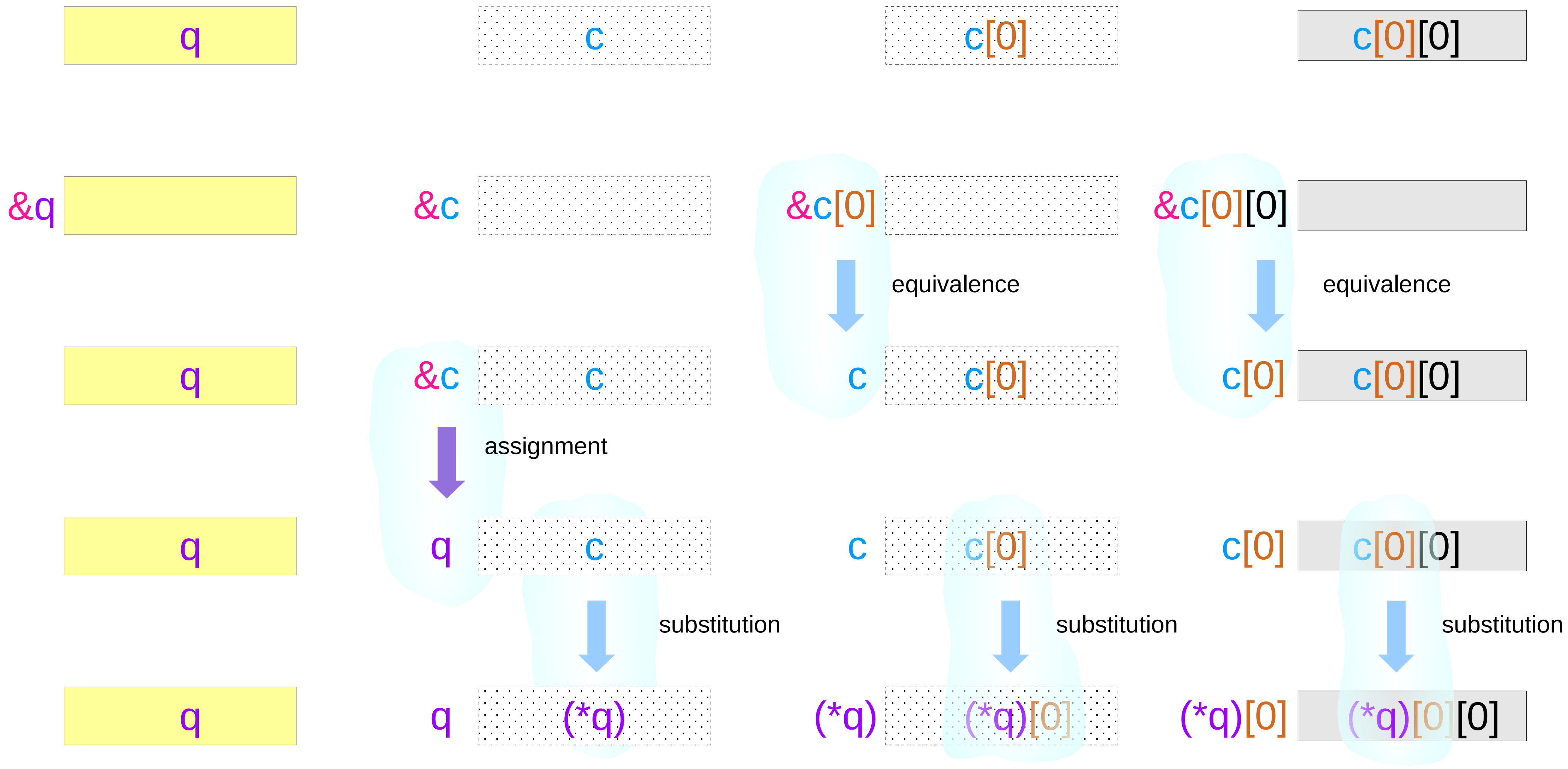
those values are evaluated as addresses

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

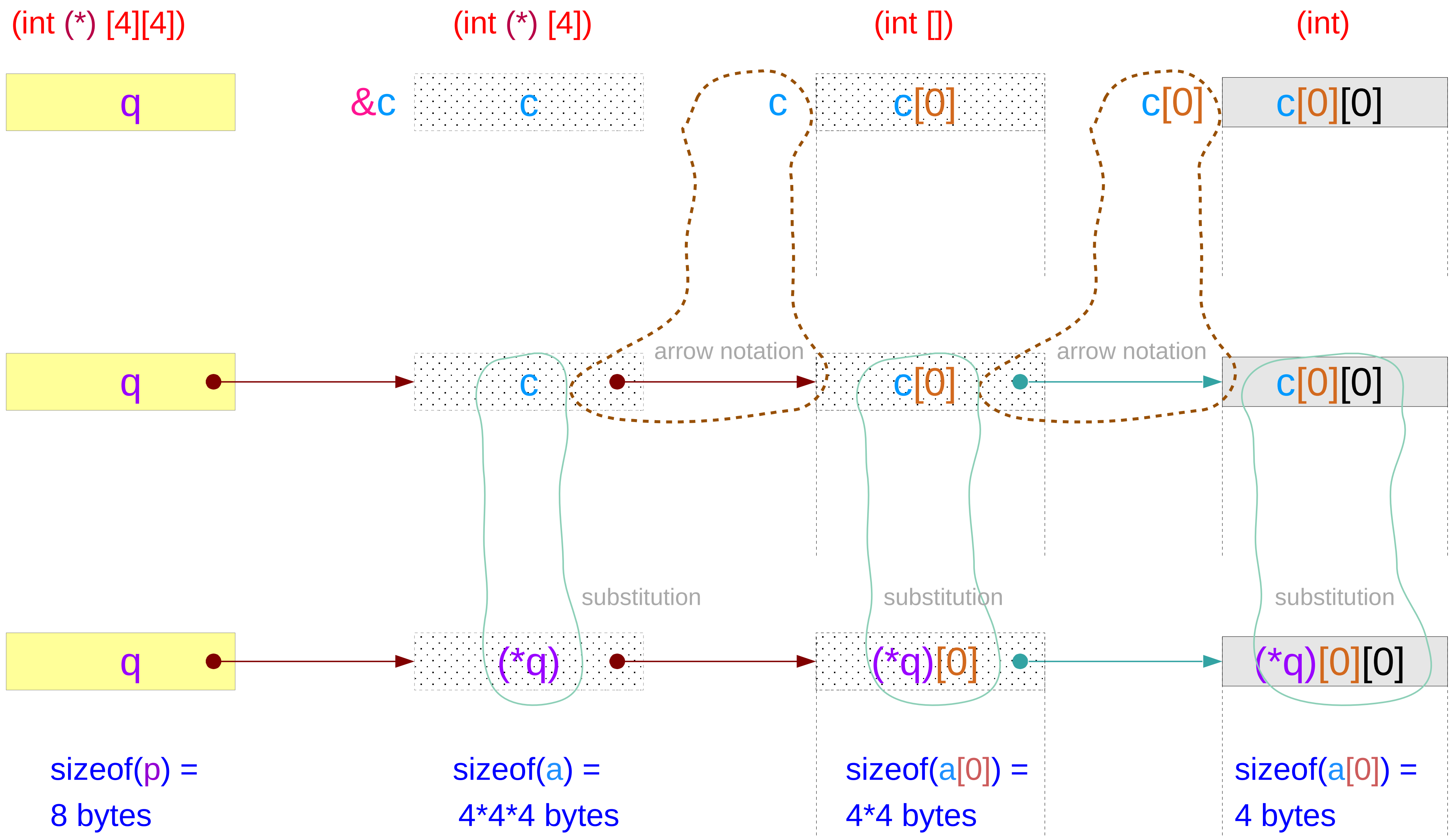
	assignment	equivalence	equivalence
int c [4][4];			
int (*q) [4];	q = &c	c ≡ &c[0]	c[0] ≡ &c[0][0]



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



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



sizeof(p) =
8 bytes

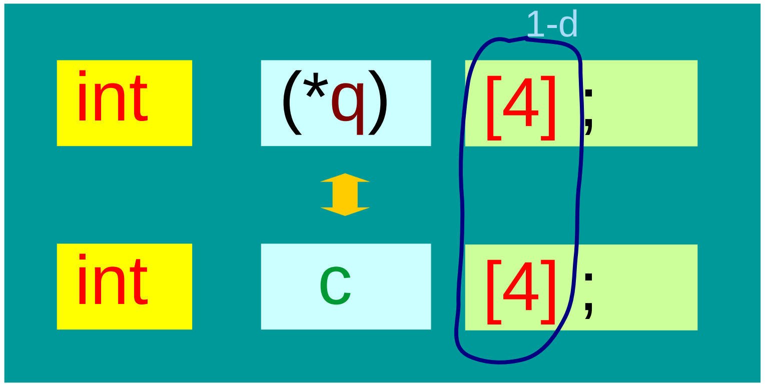
sizeof(a) =
4*4*4 bytes

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

sizeof(a[0]) =
4 bytes

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

1-d array pointer

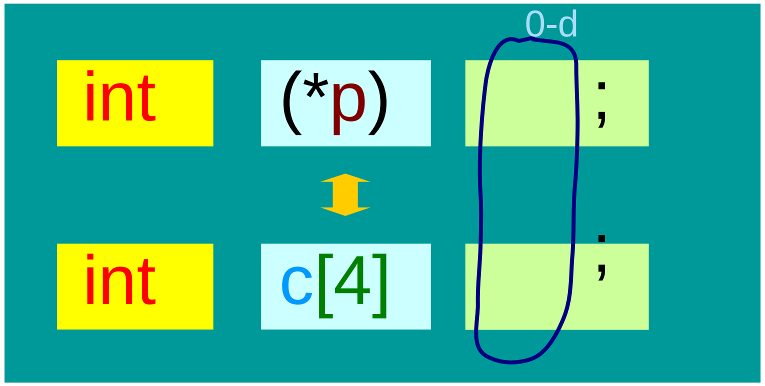


correspondence *q ≡ c;

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

(*q)[**i**] ≡ q[0][**i**] ≡ c[**i**]

0-d array pointer : int pointer



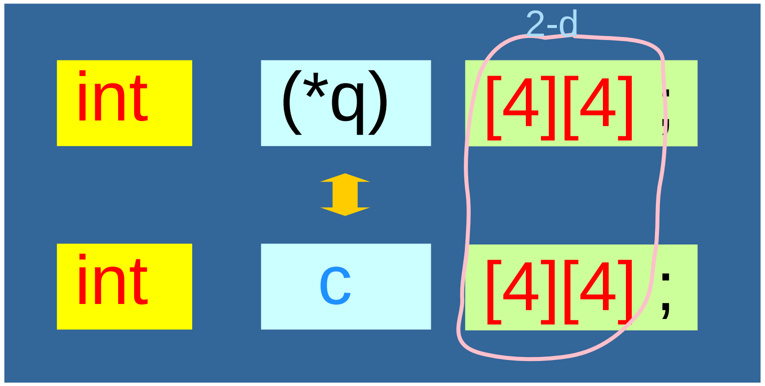
correspondence *p ≡ *c;

(int (*)) p = c;

p[**i**] ≡ c[**i**]

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

2-d array pointer

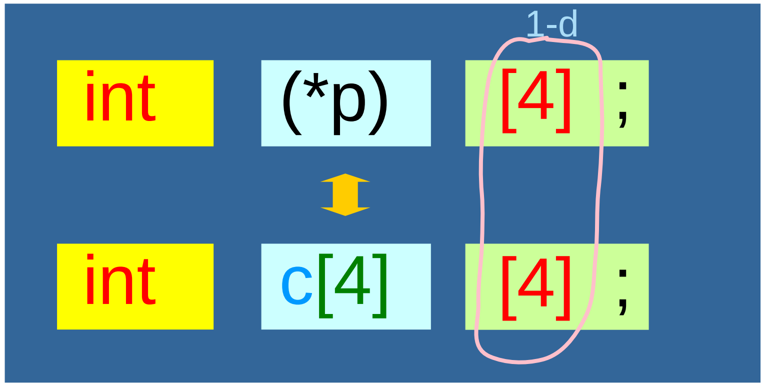


correspondence
*q ≡ c;

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

(*q)[i][j] ≡ q[0][i][j] ≡ c[i][j]

1-d array pointer

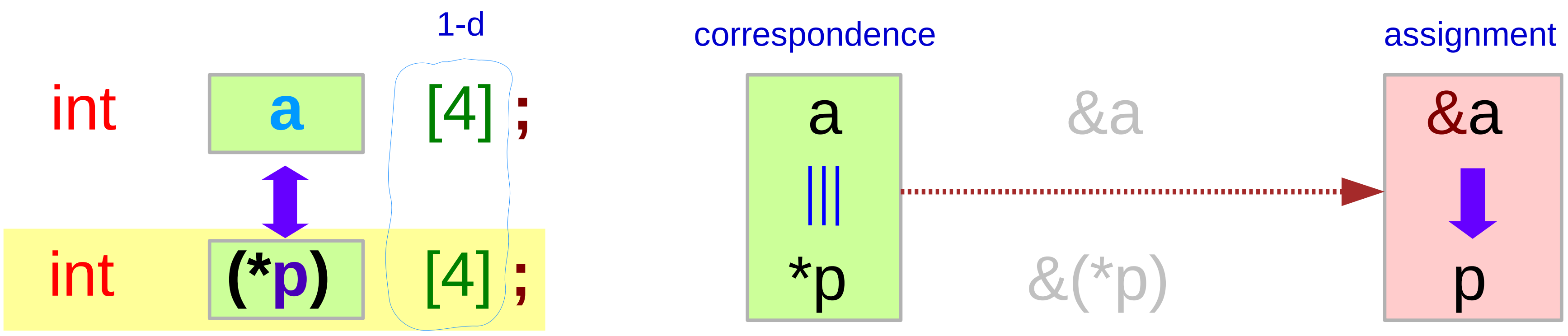


correspondence
*p ≡ *c;

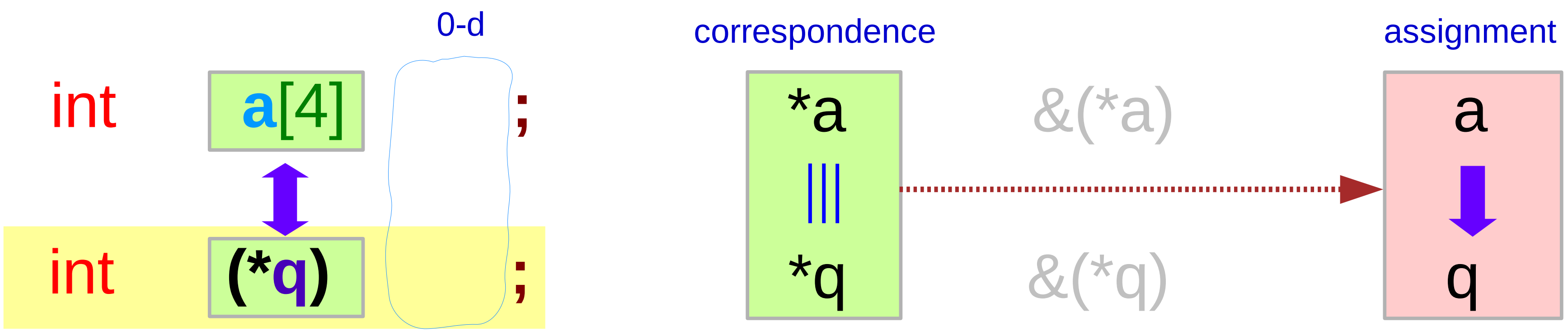
(int (*) [4])
p = c;

p[i] ≡ c[i]

Pointer types to a **1-d** array : 2 cases

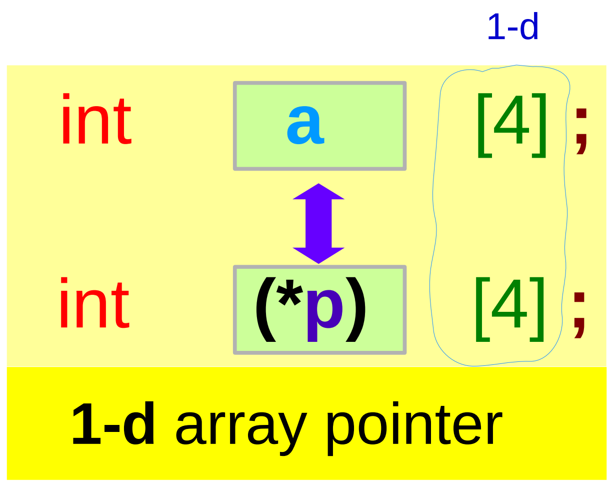


1-d array pointer **p**



0-d array pointer **q** (= integer pointer)

Pointer types to a **1-d** array : sizes of pointer dereferences



assignment

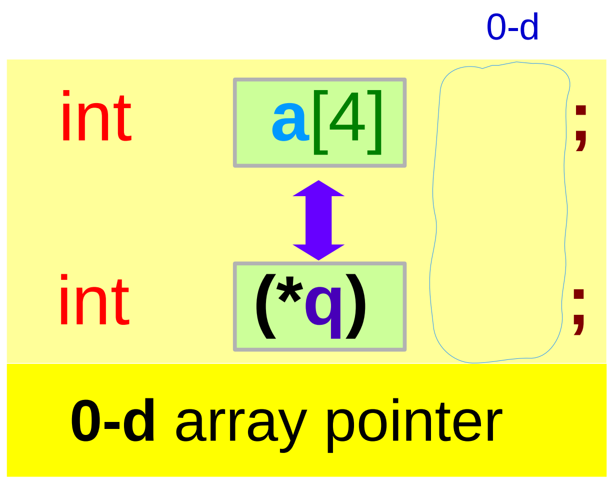
p = &a;

substitution

(*p)[i] ≡ p[0][i] ≡ a[i]

sizeof(p) = 4 or 8 bytes : the size of a pointer

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



assignment

q = a;

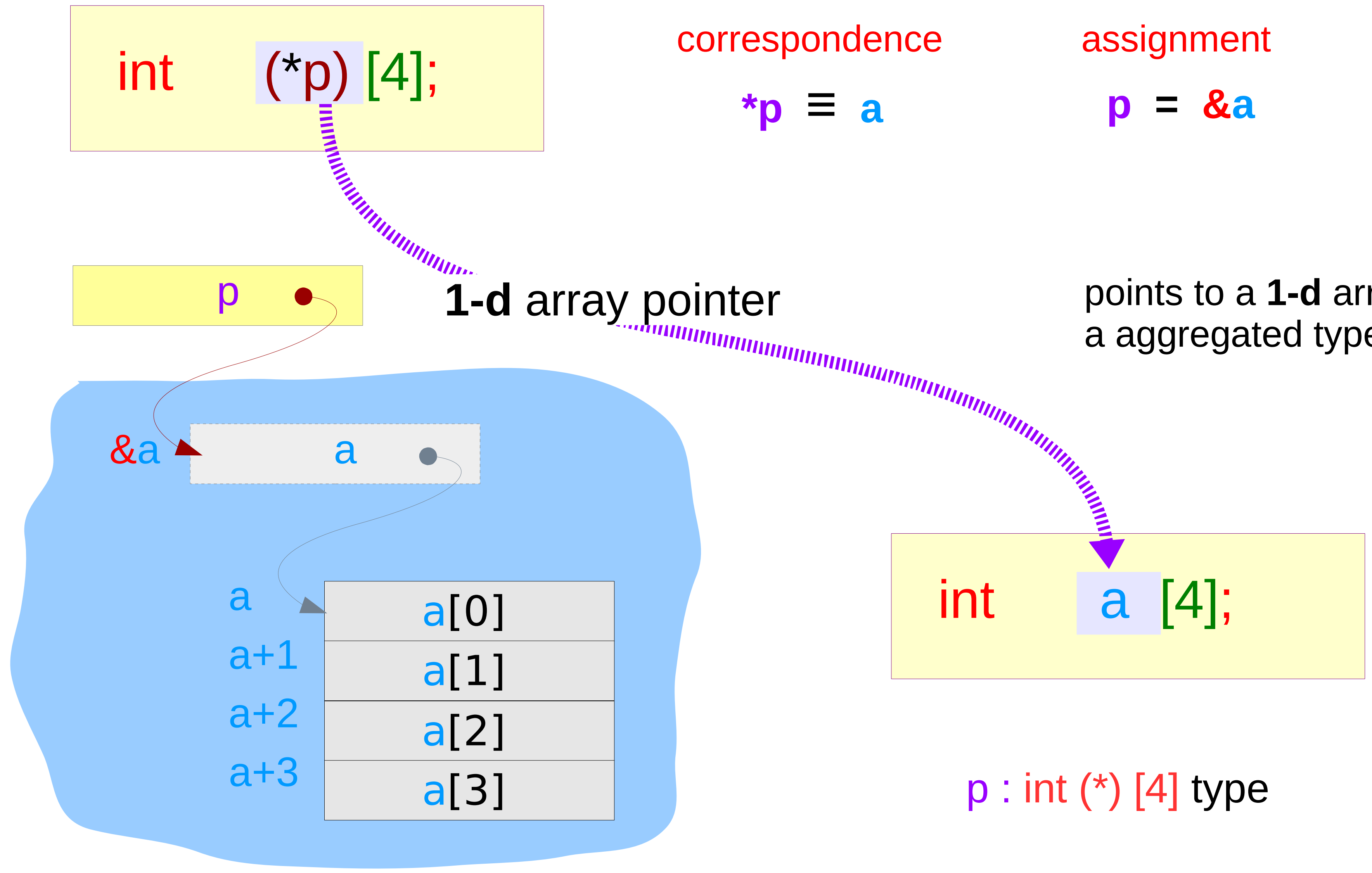
substitution

q[i] ≡ a[i]

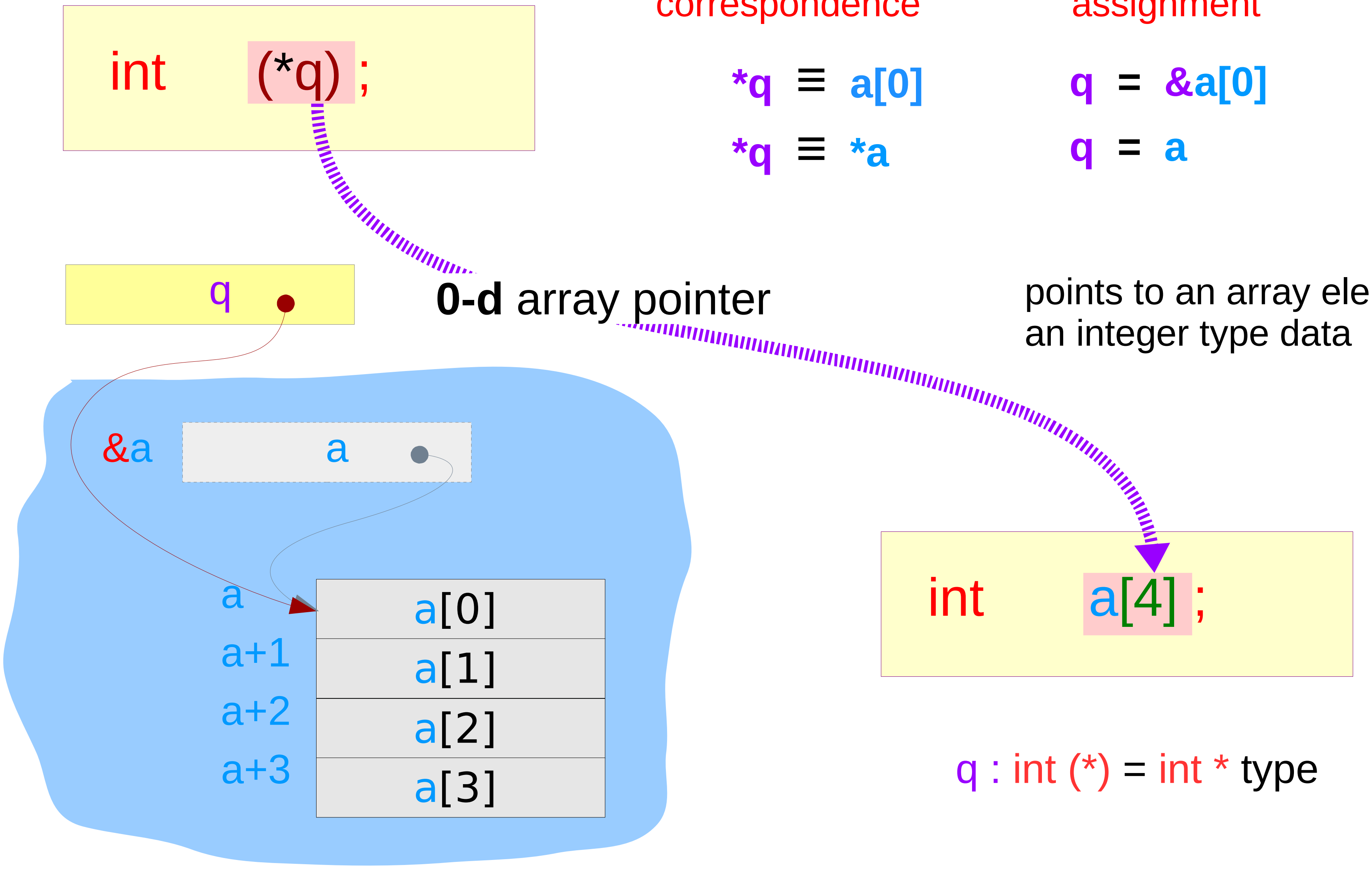
sizeof(q) = 4 or 8 bytes : the size of a pointer

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

1-d pointer to a **1-d** array – a variable view



0-d pointer to a **1-d** array – a variable view



correspondence	assignment
***q ≡ a[0]**	**q = &a[0]**
***q ≡ *a**	**q = a**

points to an array element –
an integer type data

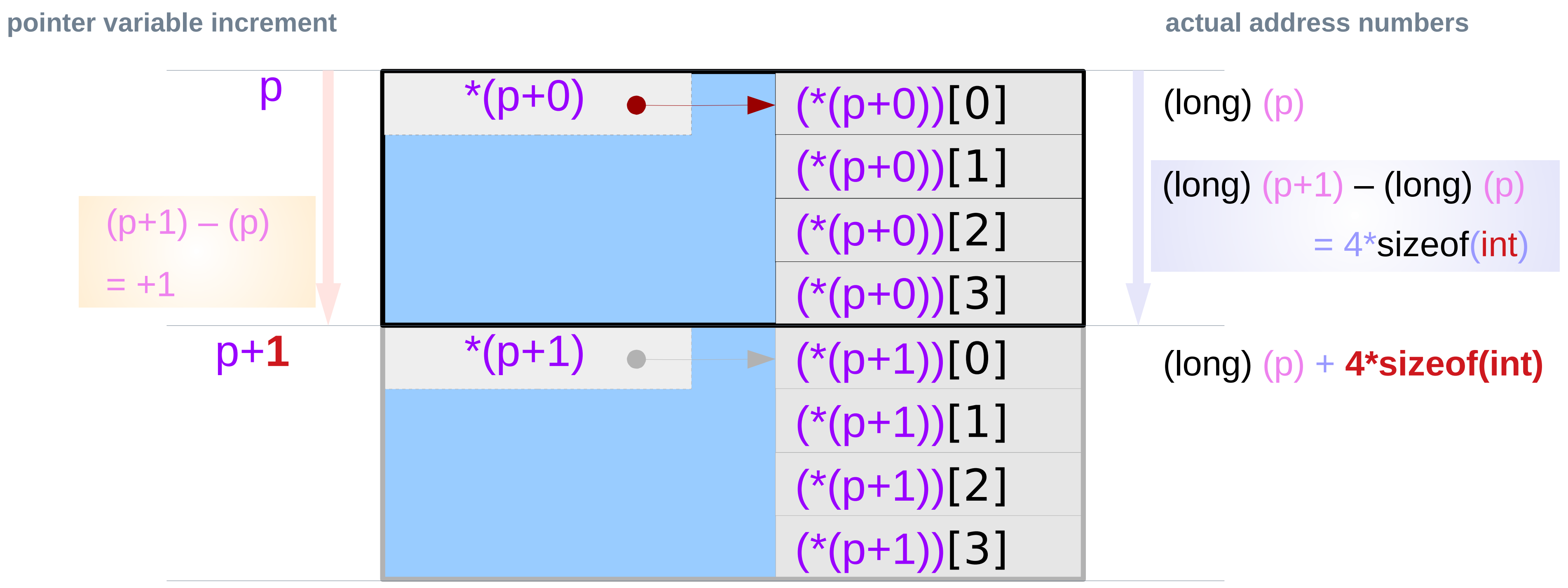
q : int (*) = int * type

Incrementing a **1-d** array pointer

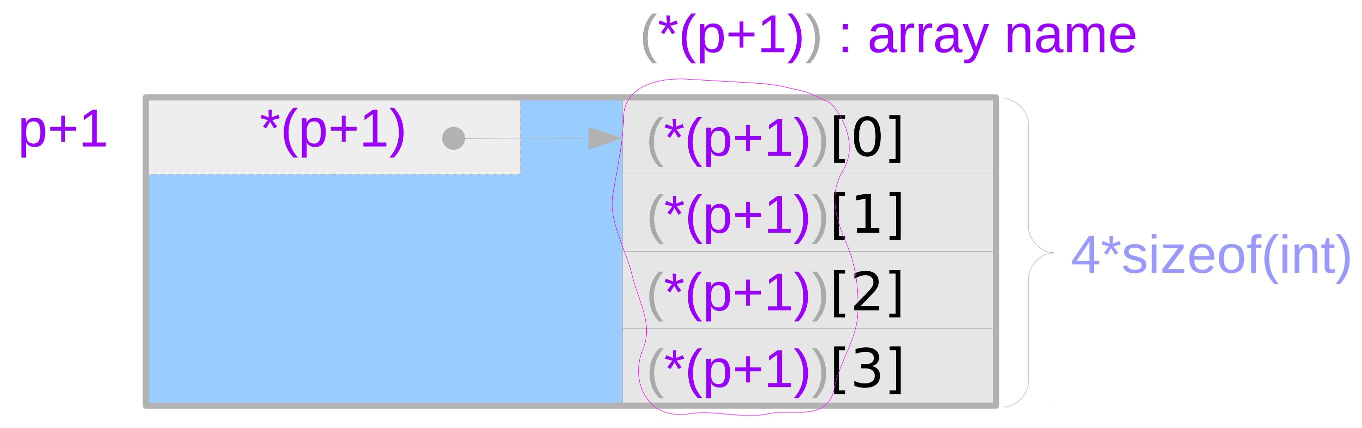
```
int     (*p) [4];
```

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

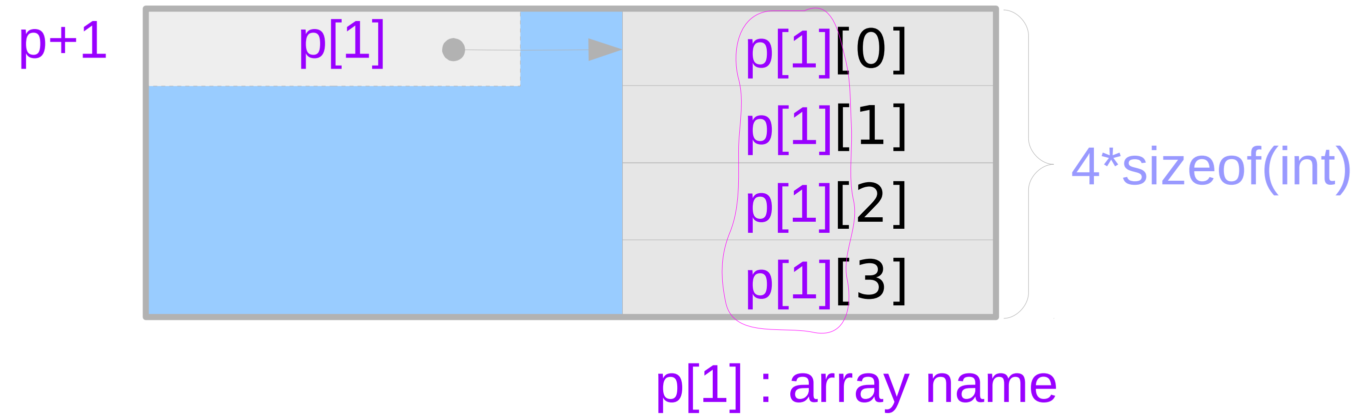
Aggregate Type Size



Incrementing a **1-d** array pointer – extending a dimension

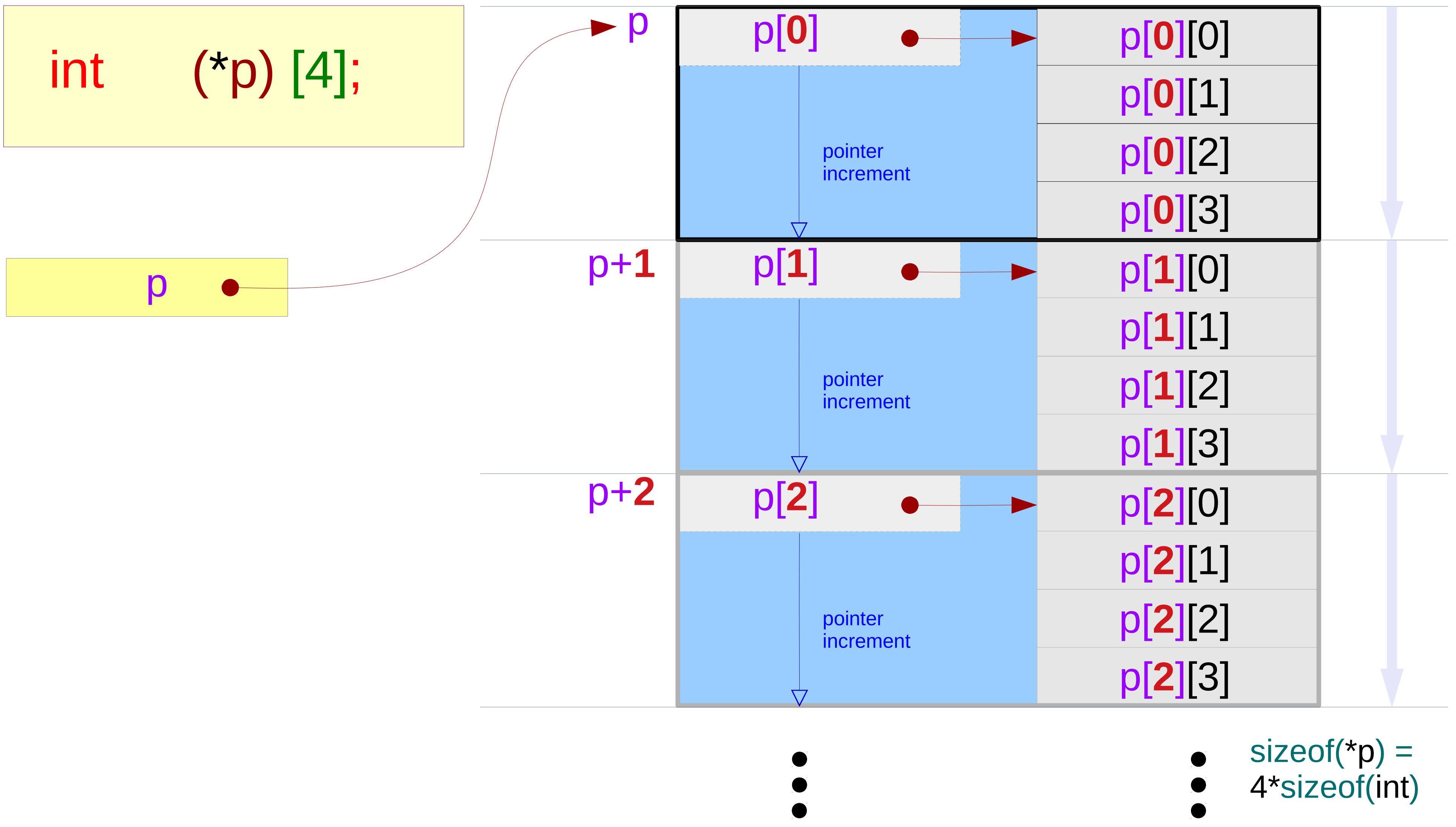


(*(p+1)) ≡ p[1] || equivalence



Substitution using a **1-d** array pointer

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

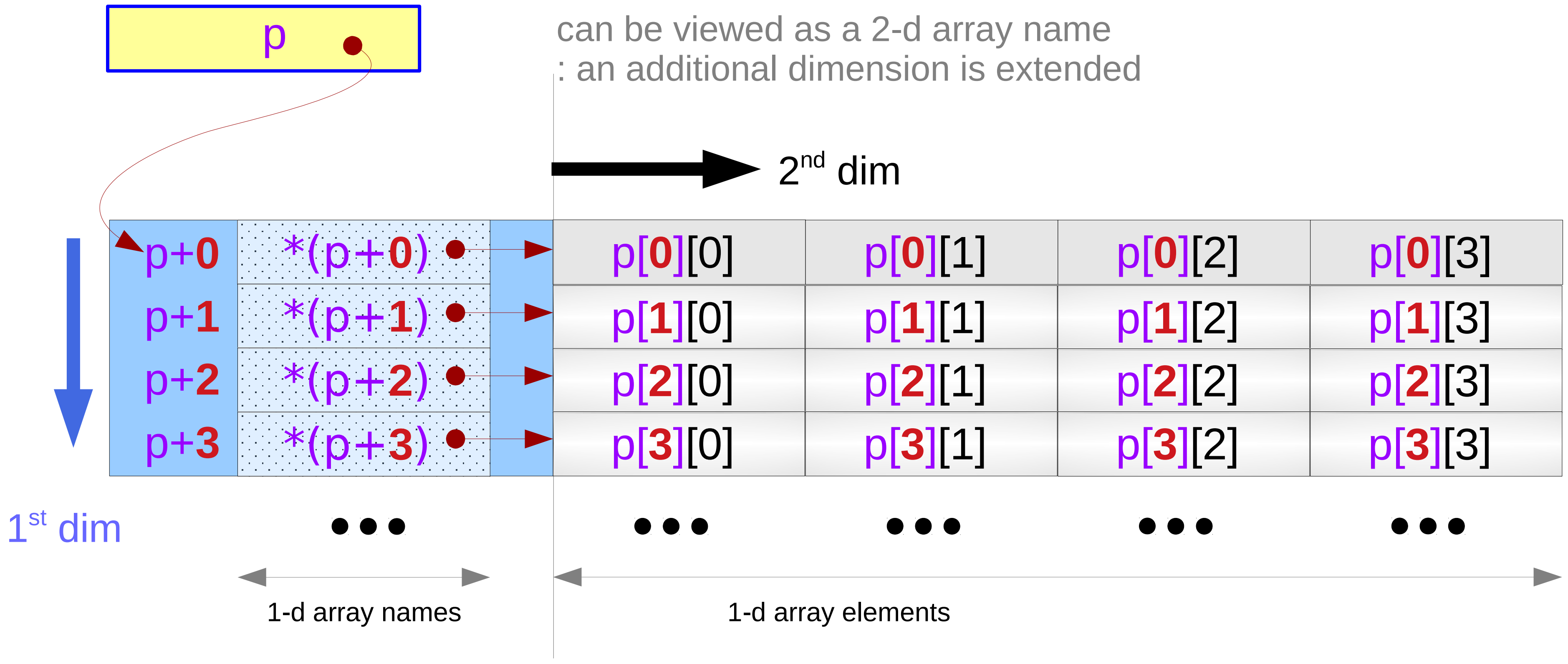


sizeof(*p) = 4*sizeof(int)

A **1-d** array pointer – extending a dimension

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

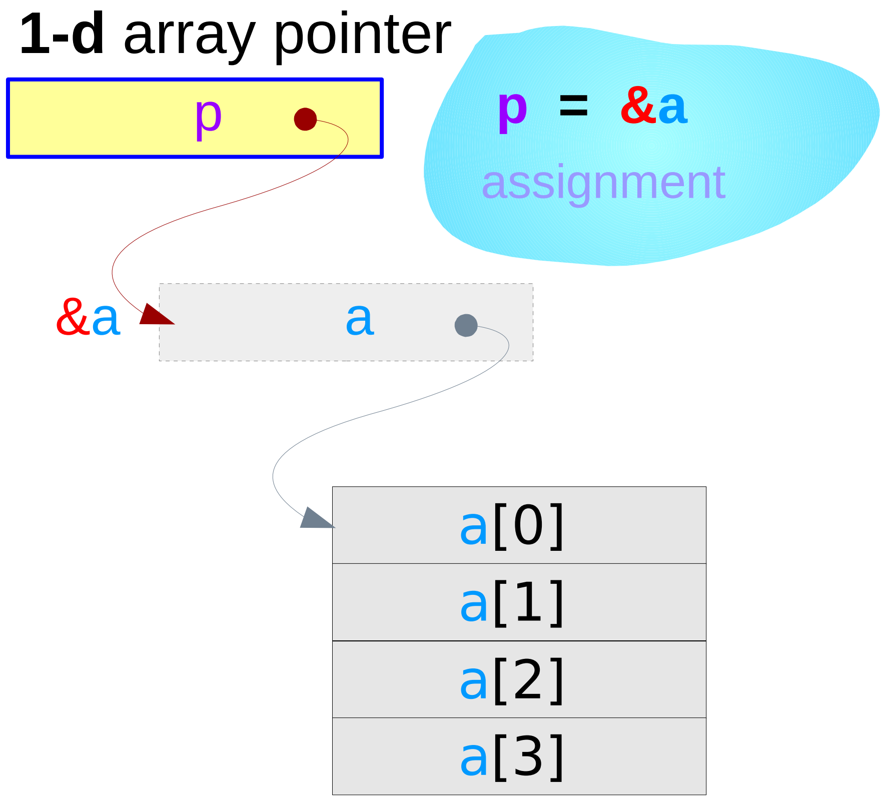
1-d array pointer

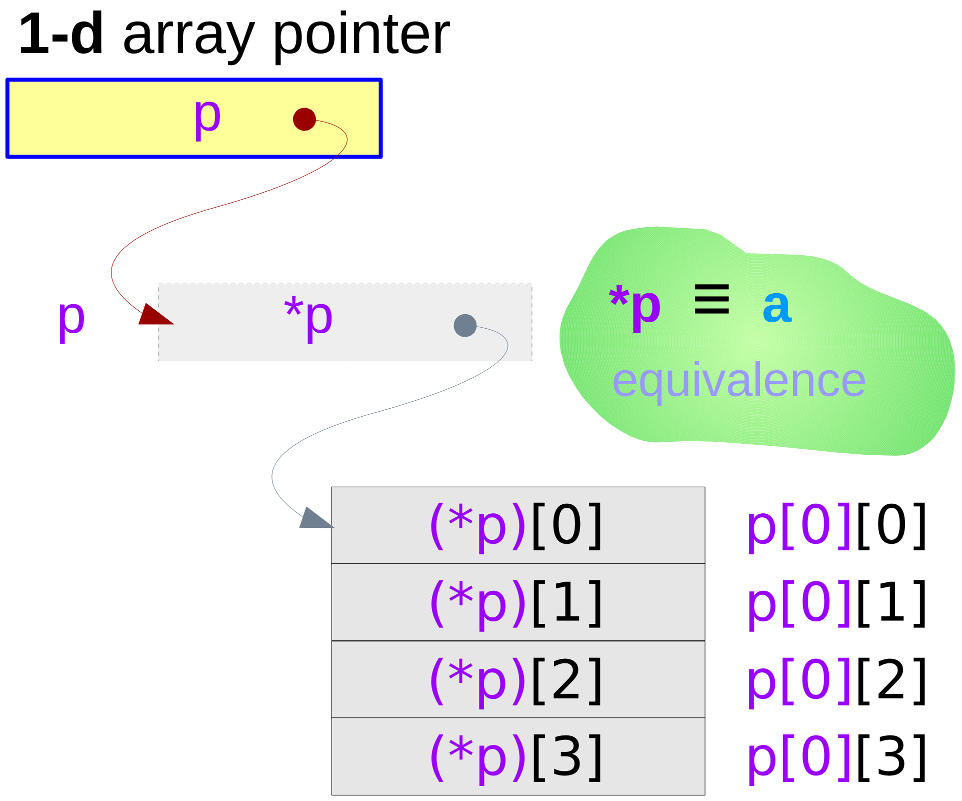

p
can be viewed as a 2-d array name
: an additional dimension is extended
2nd dim
1st dim
p+0
p+1
p+2
p+3
*(p+0)
*(p+1)
*(p+2)
*(p+3)
p[0][0] p[0][1] p[0][2] p[0][3]
p[1][0] p[1][1] p[1][2] p[1][3]
p[2][0] p[2][1] p[2][2] p[2][3]
p[3][0] p[3][1] p[3][2] p[3][3]
1-d array names
1-d array elements

A **1-d** array pointer and a **1-d** array

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

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

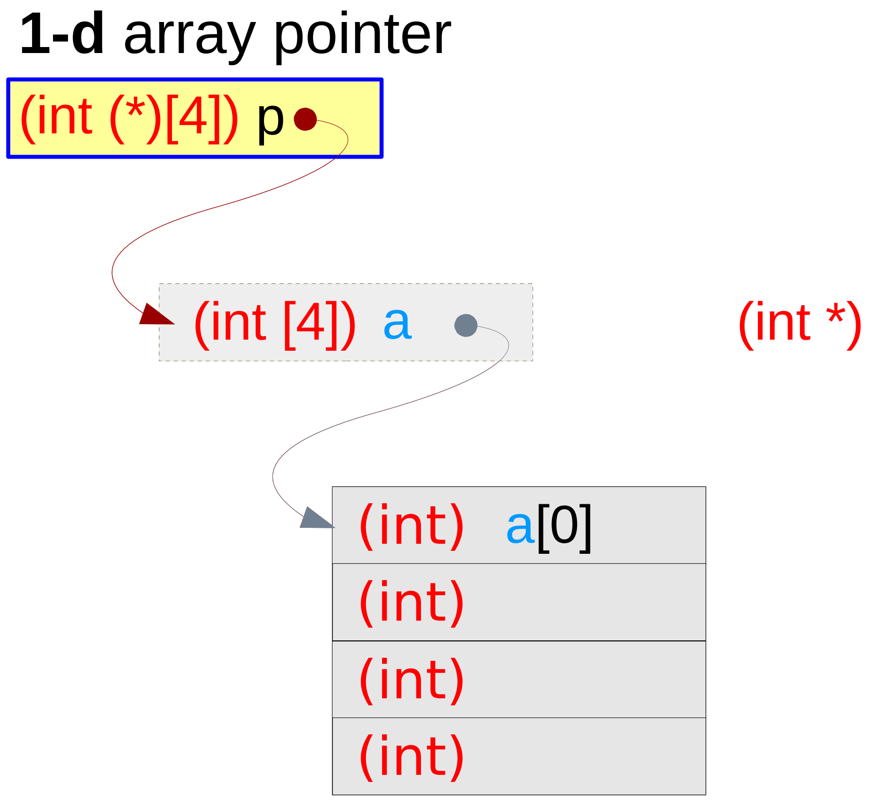


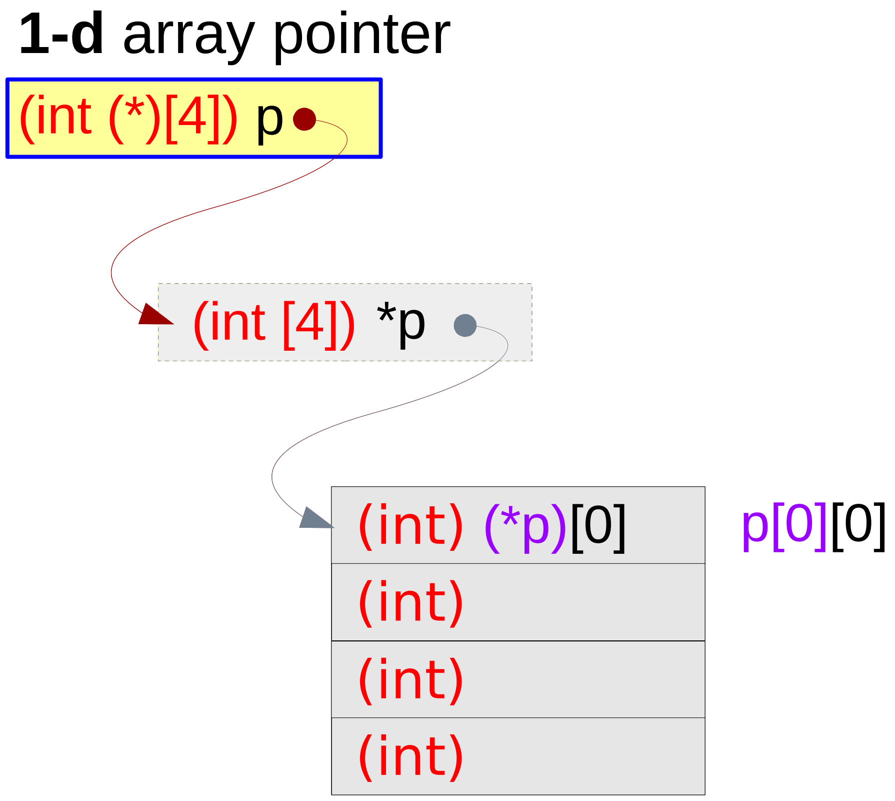


A **1-d** array pointer and a **1-d** array – a type view

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

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

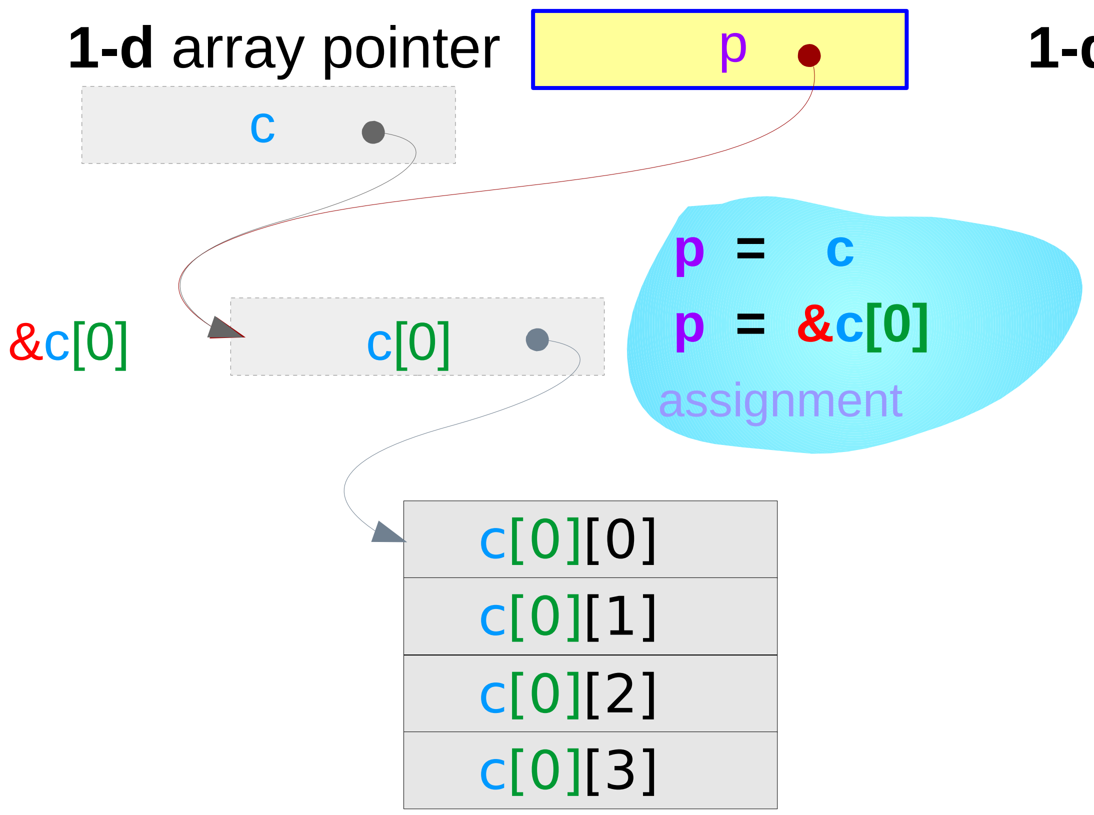


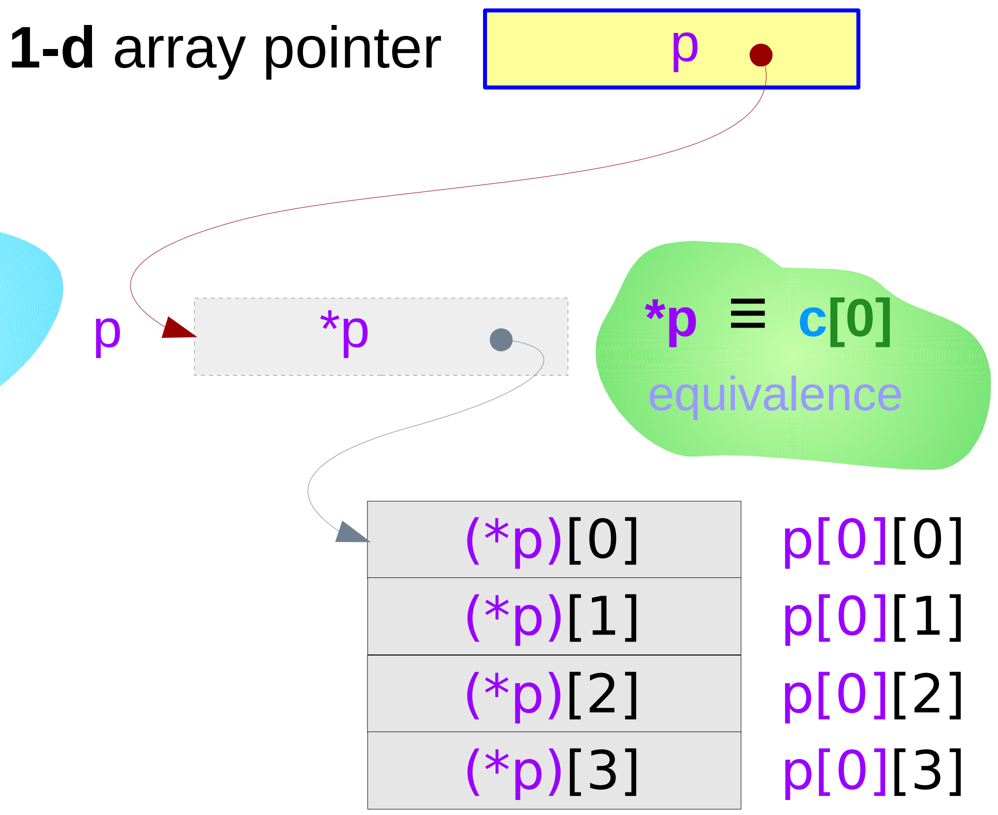


A **1-d** array pointer and a **2-d** array

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

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

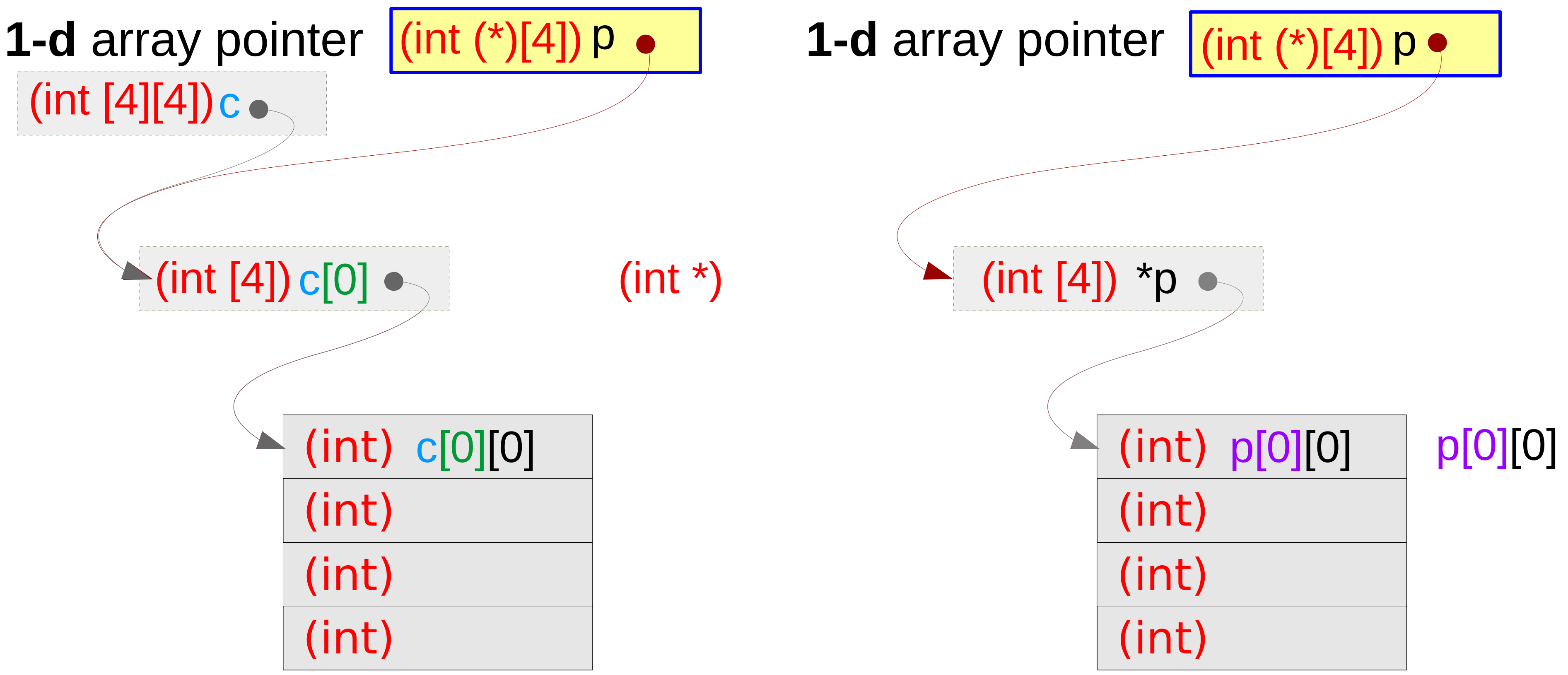




A **1-d** array pointer and a **2-d** array – a type view

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

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

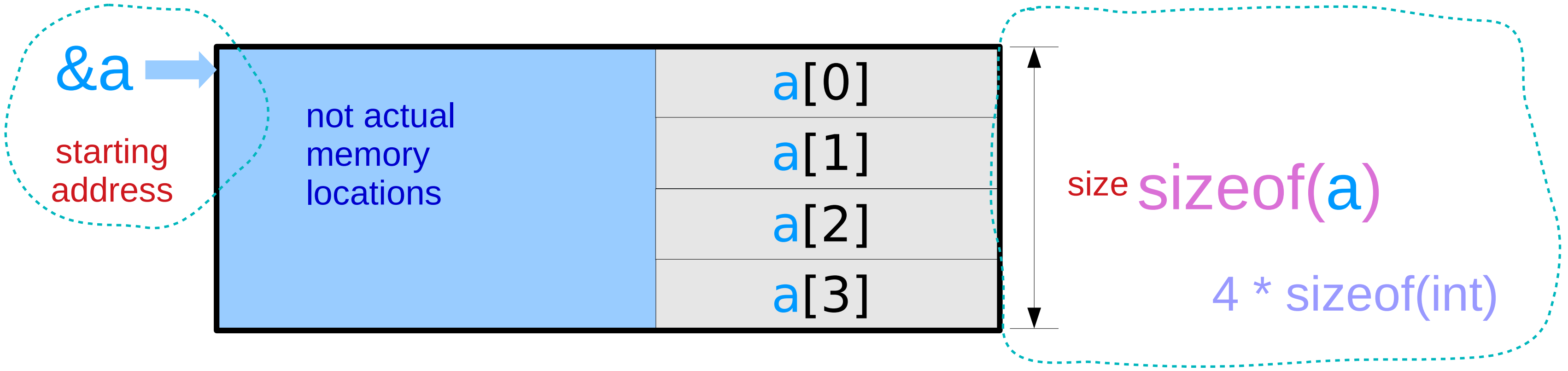


A 2-d array and array pointers

1-d array – an aggregate type view

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

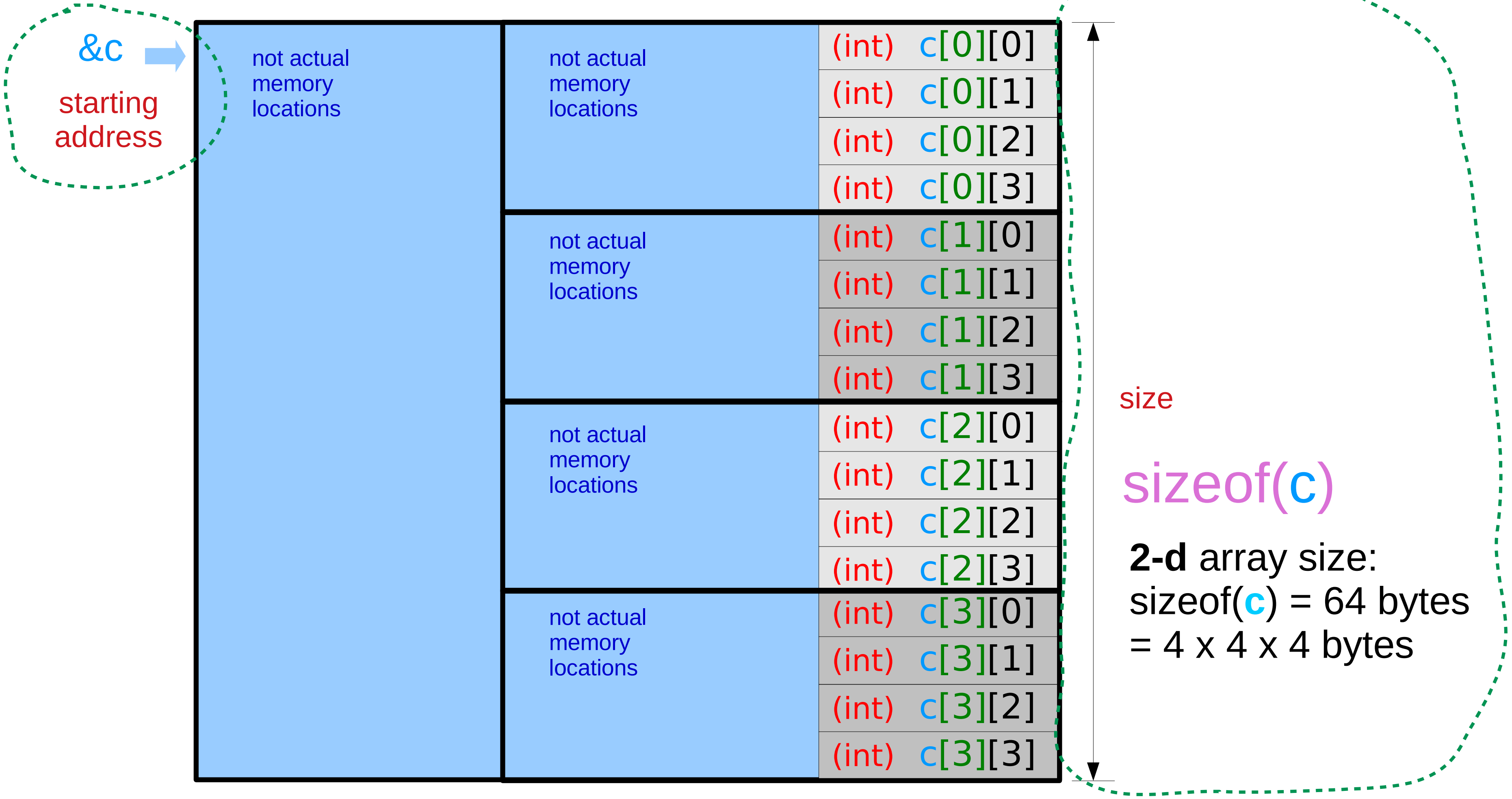
An aggregate type	a
- starting address	&a
- size	sizeof(a)



2-d array – an aggregate type view

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

An aggregate type c
- starting address &c
- size sizeof(c)



2-d array size:
sizeof(c) = 64 bytes
= 4 x 4 x 4 bytes

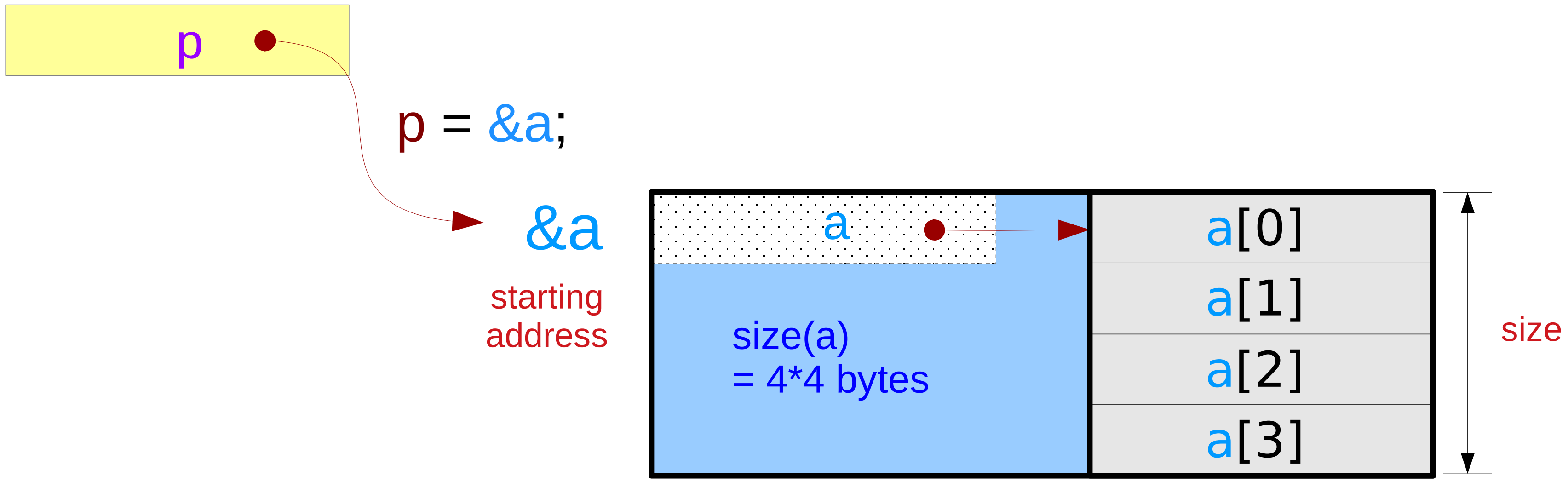
Pointer to a **1-d** array – an aggregate type view

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

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

An aggregate type	a
- starting address	&a
- size	sizeof(a)

1-d array pointer



sizeof(*p) = sizeof(a)

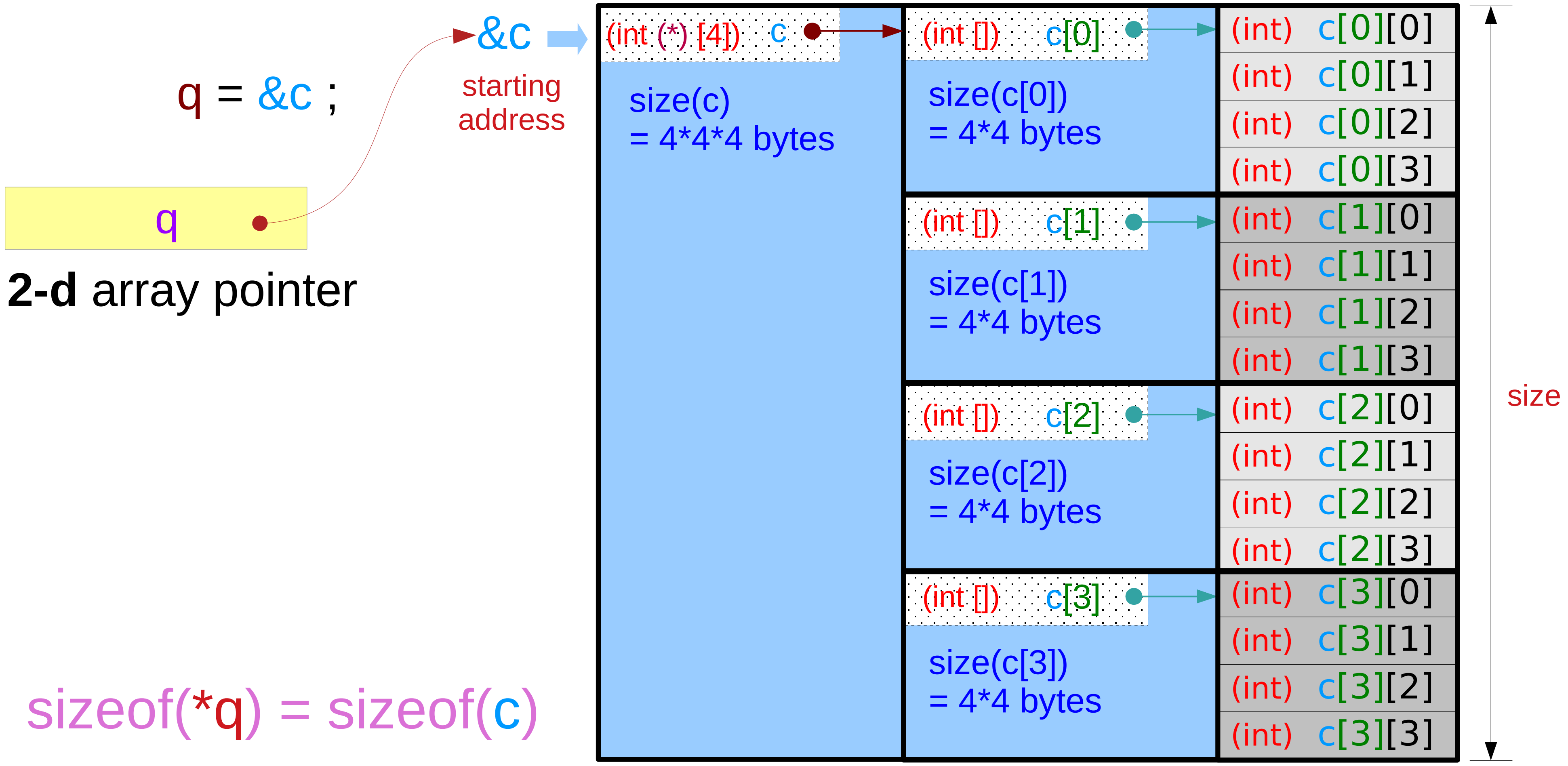
Pointer to a **2-d** array – an aggregate type view

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

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

An aggregate type	c
- starting address	&c
- size	sizeof(c)

q = &c ;



2-d array pointer

sizeof(*q) = sizeof(c)

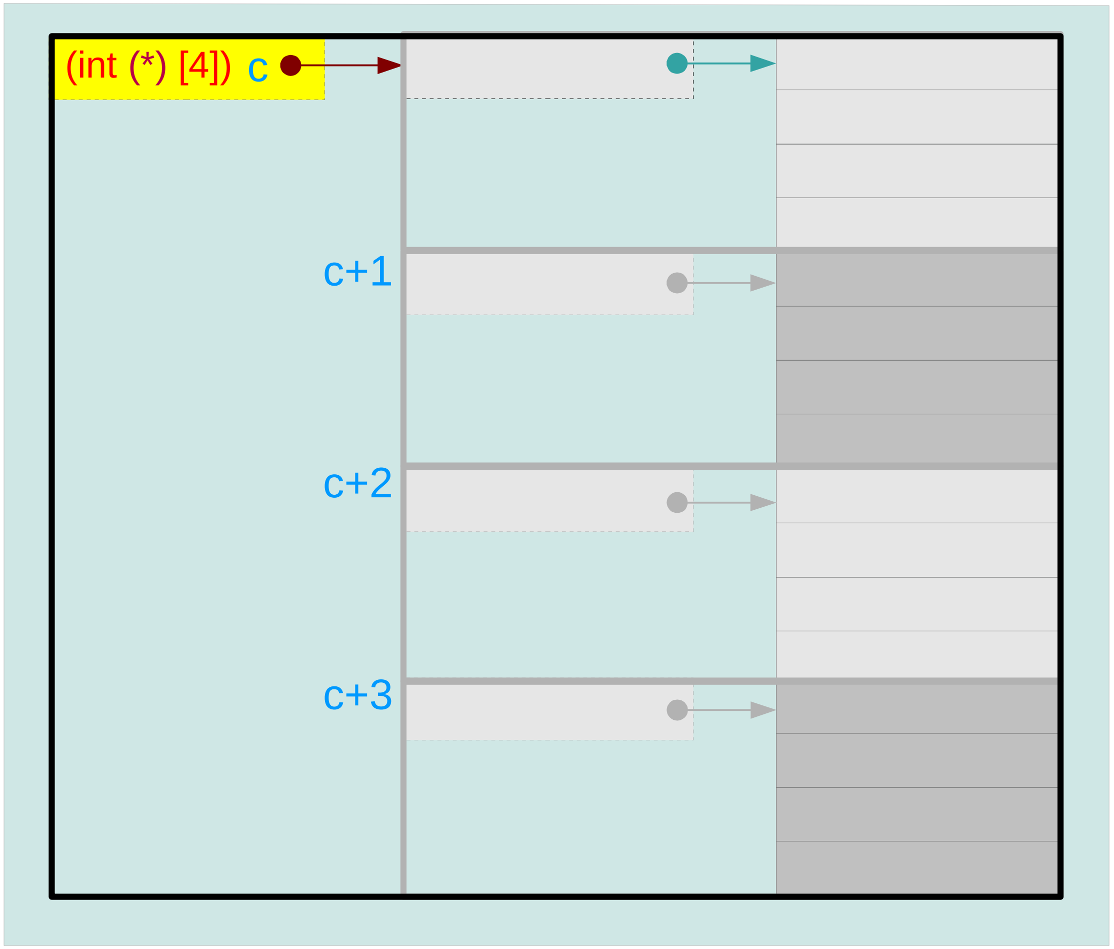
A **2-d** array and its sub-arrays – array name

int c[4][4];

c :

- the **2-d** array name
- the **2-d** array starting address
- the **1-d** array pointer which
  points to its 1st **1-d** sub-array

compilers do not allocate
a memory location for **c**



A **2-d** array and its sub-arrays – subarray names

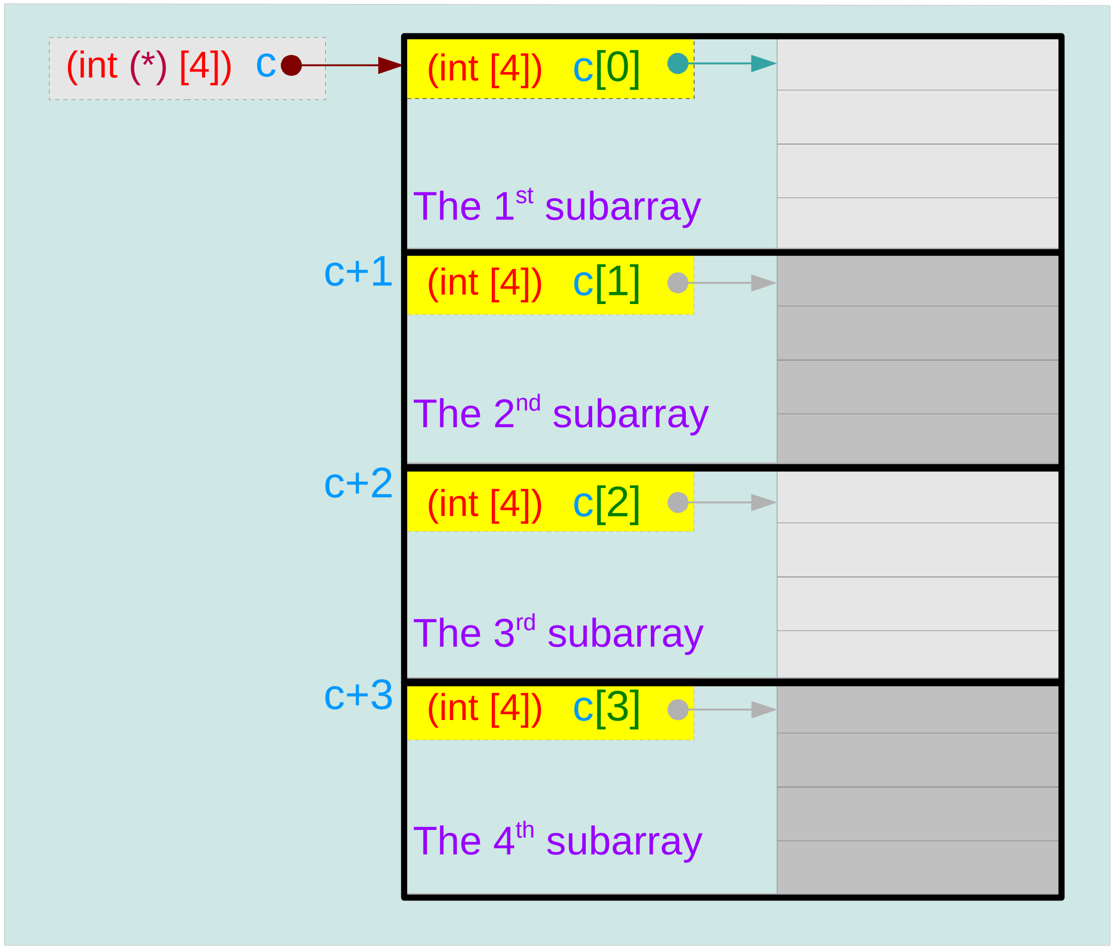
int c[4][4];

c[i]

- a **1-d** array name
- a **1-d** array starting address
- a **0-d** array pointer which points to its scalar integer

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

compilers do not allocate memory locations for **c[i]**'s

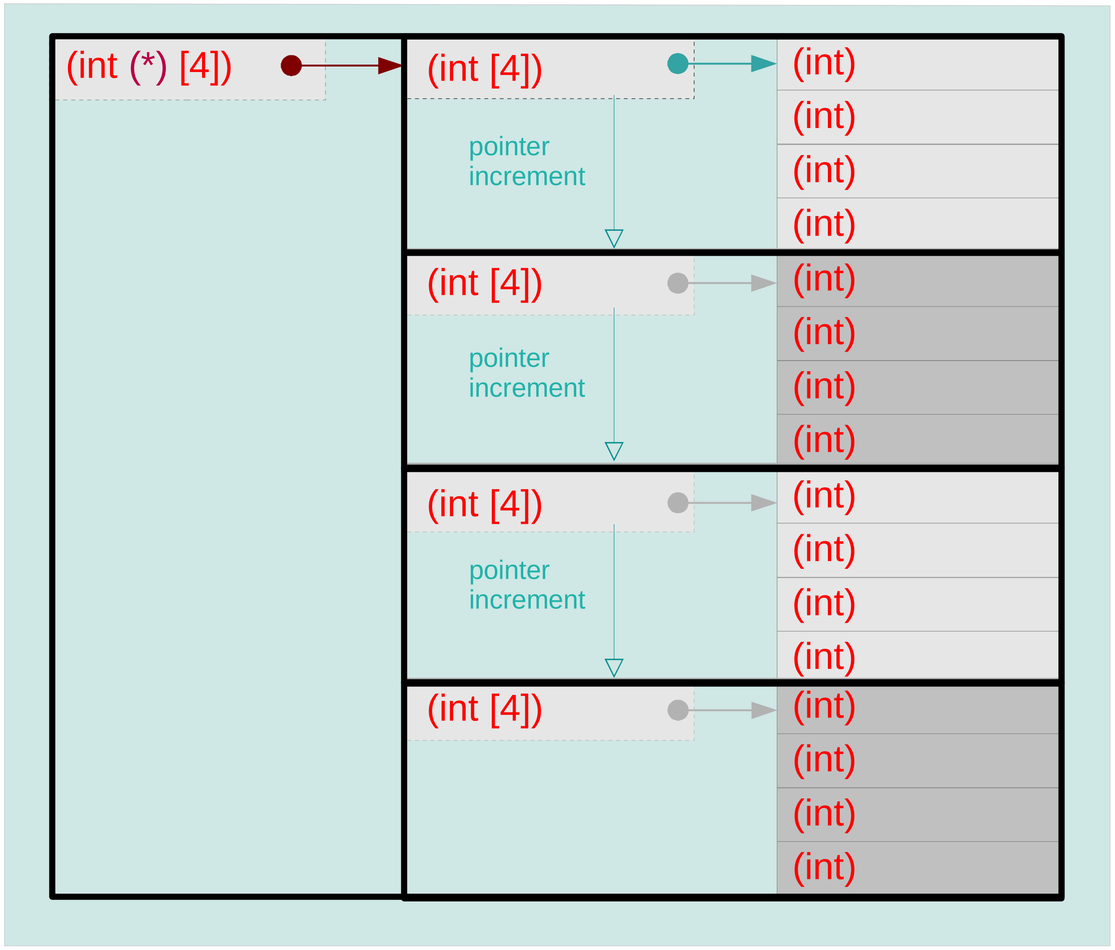


A **2-d** array and its **1-d** sub-arrays – a type view

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

1-d subarray name c[0]	int [4]
1-d subarray name c[1]	int [4]
1-d subarray name c[2]	int [4]
1-d subarray name c[3]	int [4]

c and c[0]
- different types
- the same address of the starting element



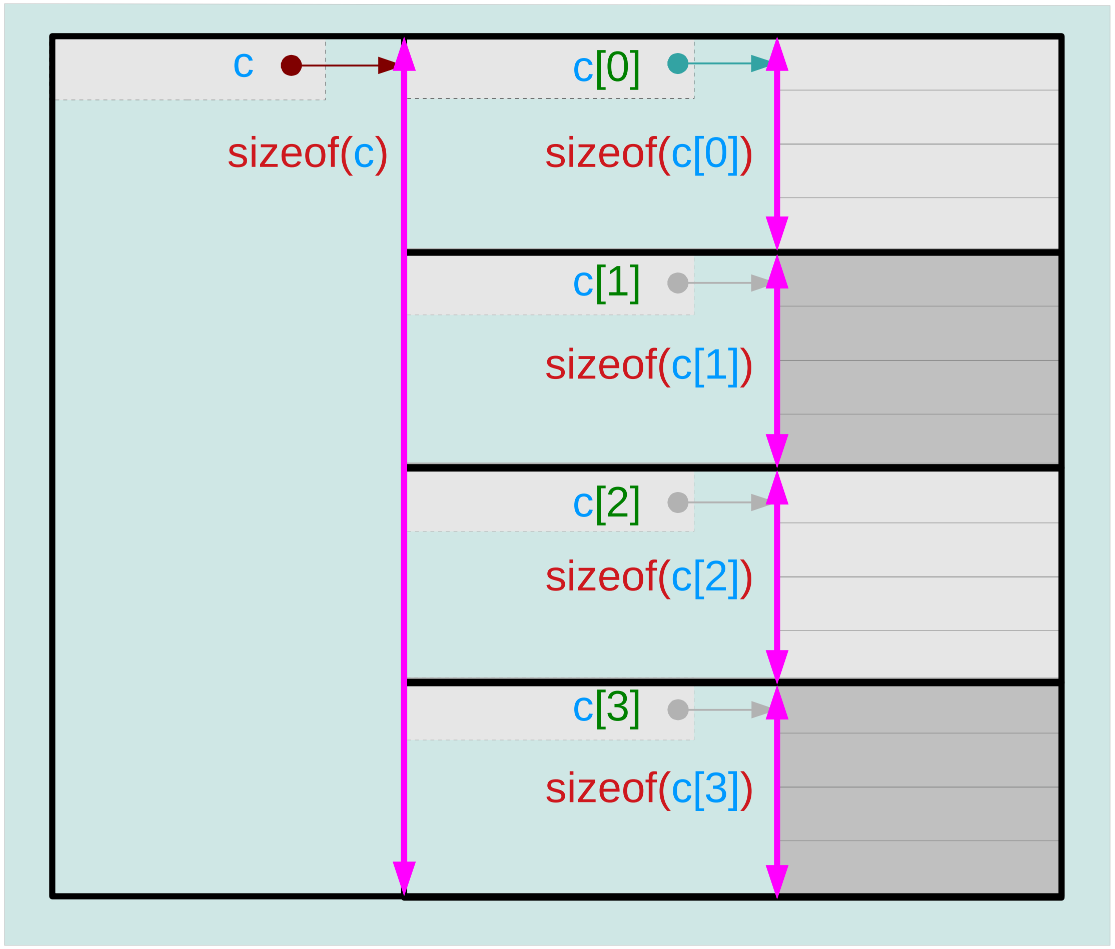
A **2-d** array and its sub-arrays – type sizes

sizeof(c) = 4*4*4 bytes

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

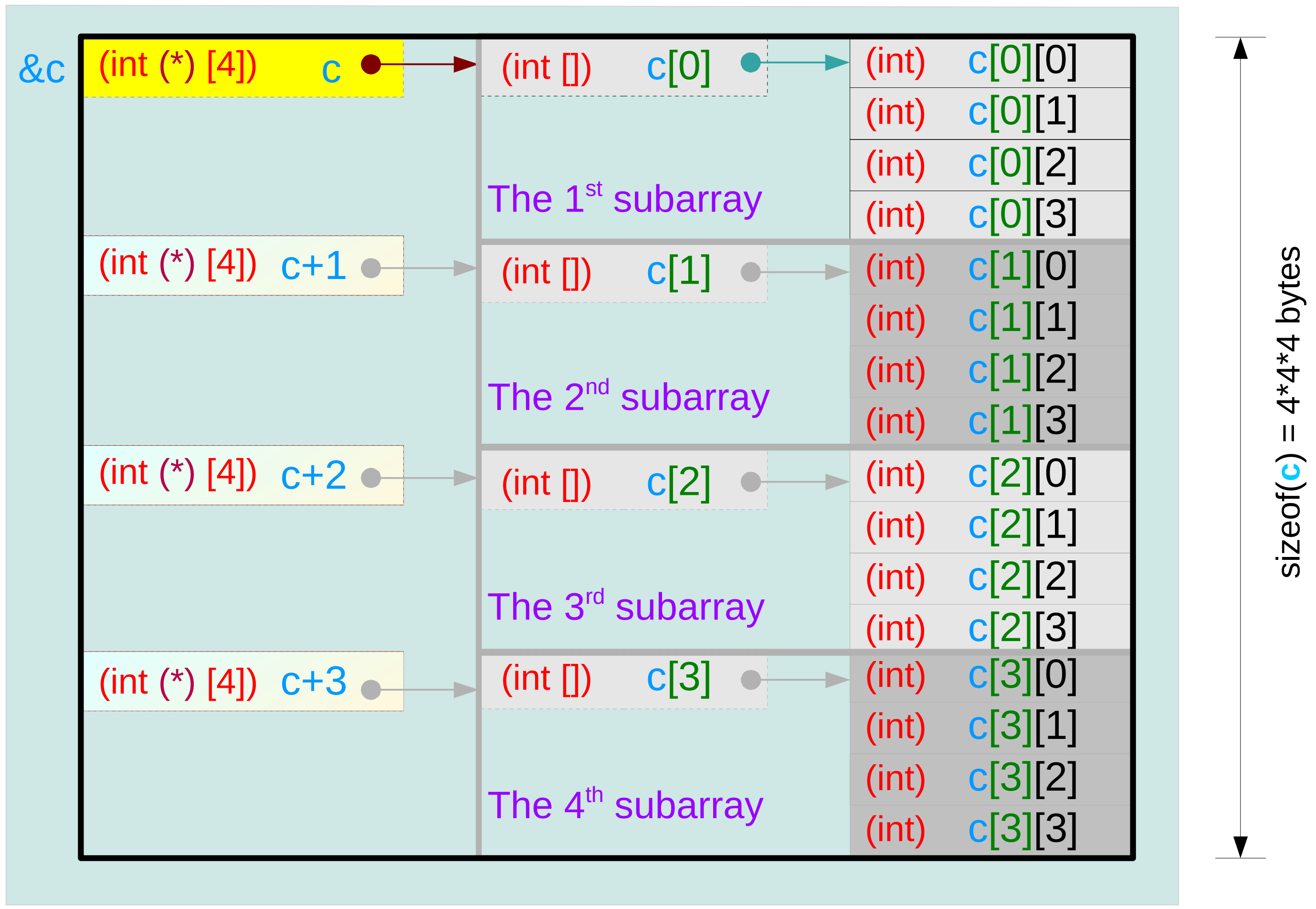
sizeof(c[i][j]) = 4 bytes

c	: the **2-d** array name
c[i]	: the **1-d** array name
c[i][j]	: the **0-d** array name (a scalar integer)



2-d array aggregate data type

The array c (=subarray name)
sizeof(c) = 4*4*4 bytes
&c : start address



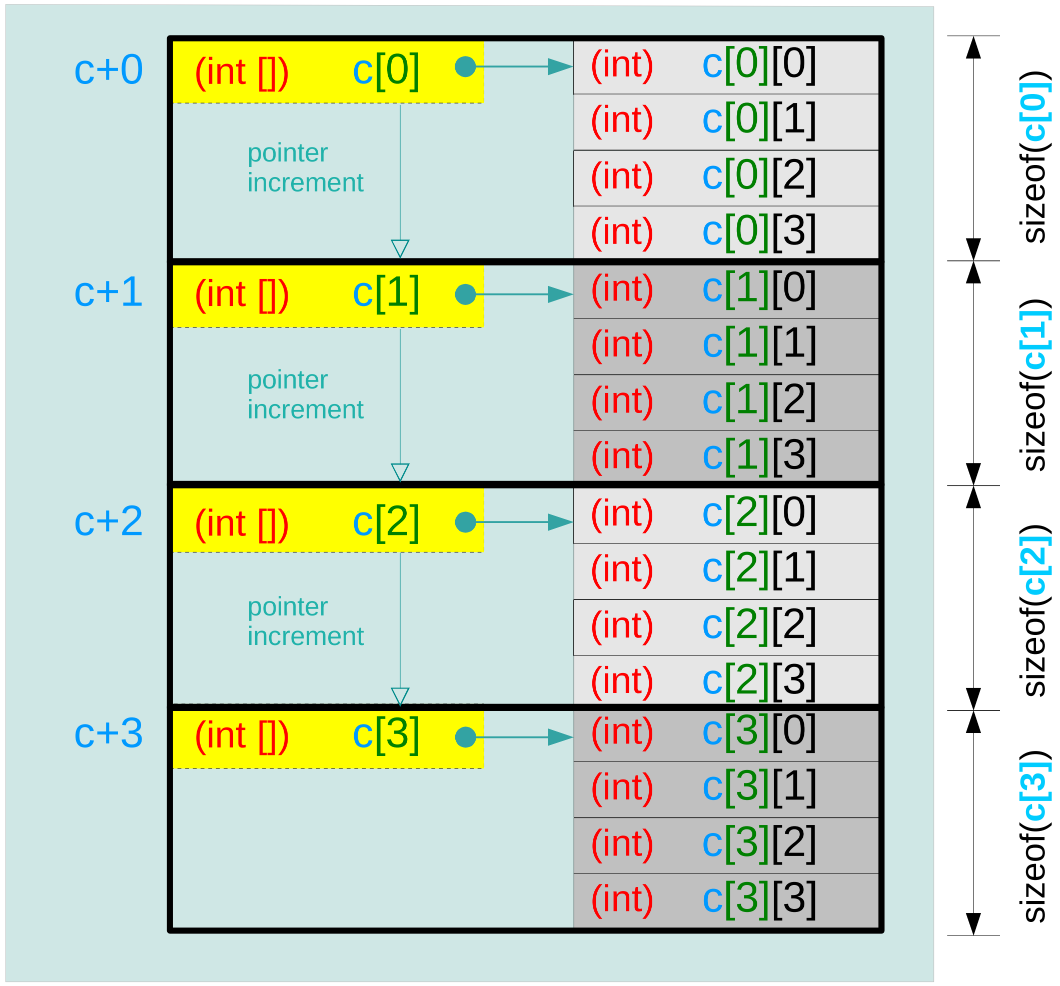
1-d subarray aggregate data type

The 1st subarray c[0] (=subarray name)
sizeof(c[0]) = 4*4 bytes
(c+0) : start address

The 2nd subarray c[1] (=subarray name)
sizeof(c[1]) = 4*4 bytes
(c+1) : start address

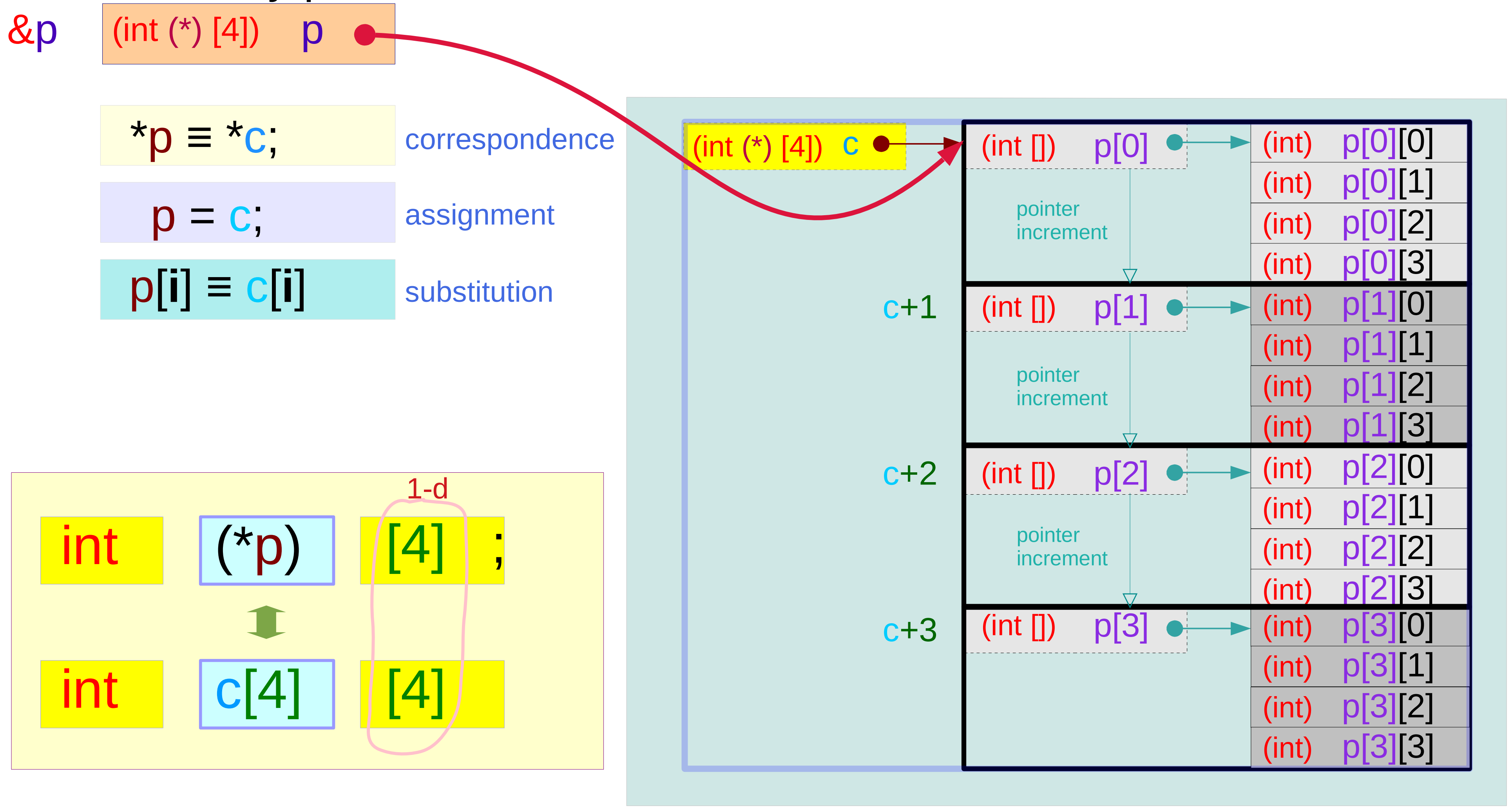
The 3rd subarray c[2] (=subarray name)
sizeof(c[2]) = 4*4 bytes
(c+2) : start address

The 4th subarray c[3] (=subarray name)
sizeof(c[3]) = 4*4 bytes
(c+3) : start address

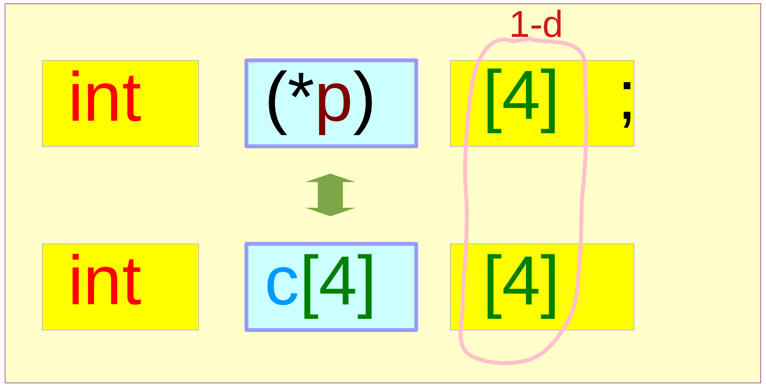


Using a **1-d** array pointer to a **2-d** array

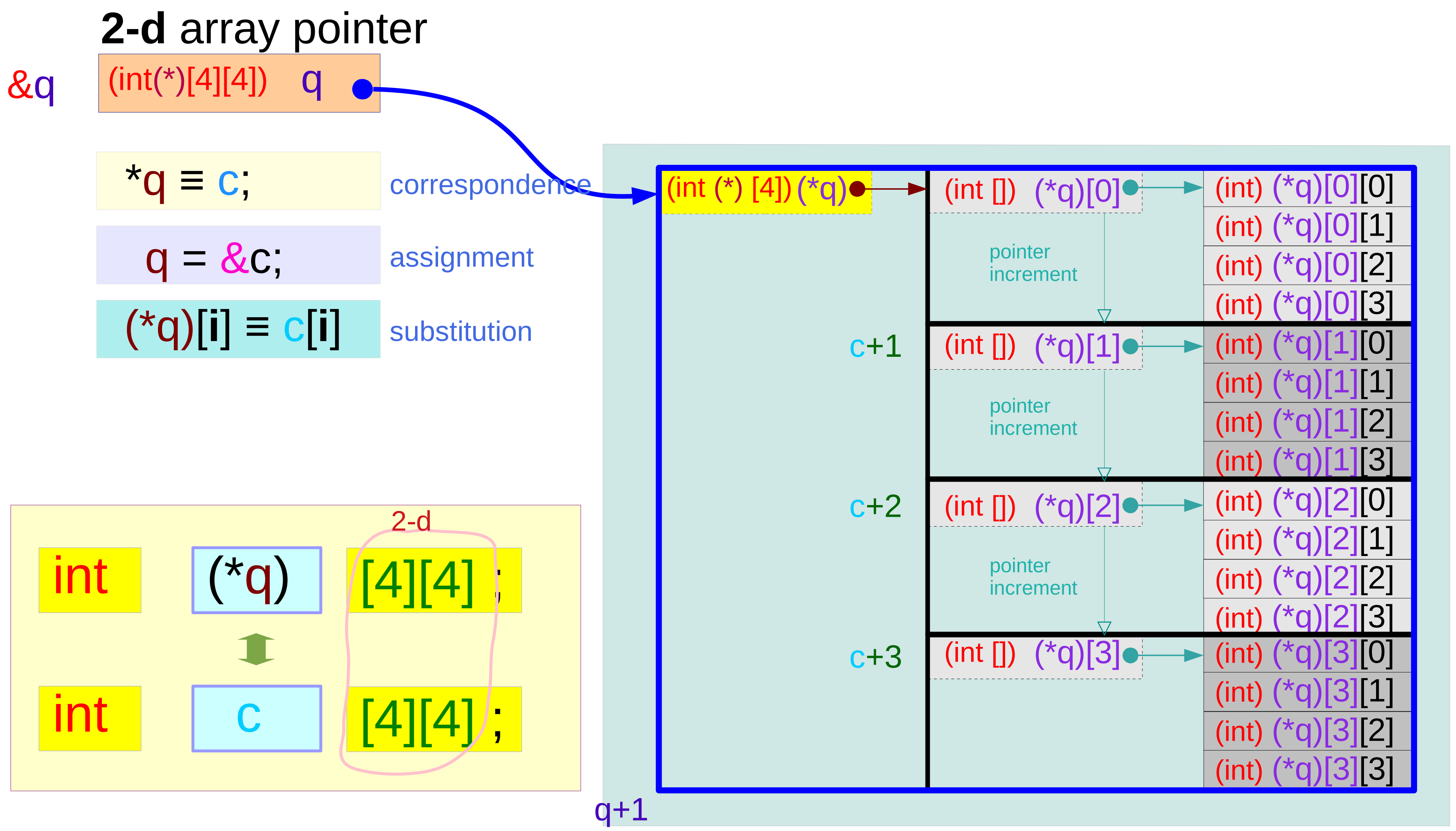
1-d array pointer



*p ≡ *c;	correspondence
p = c;	assignment
p[i] ≡ c[i]	substitution

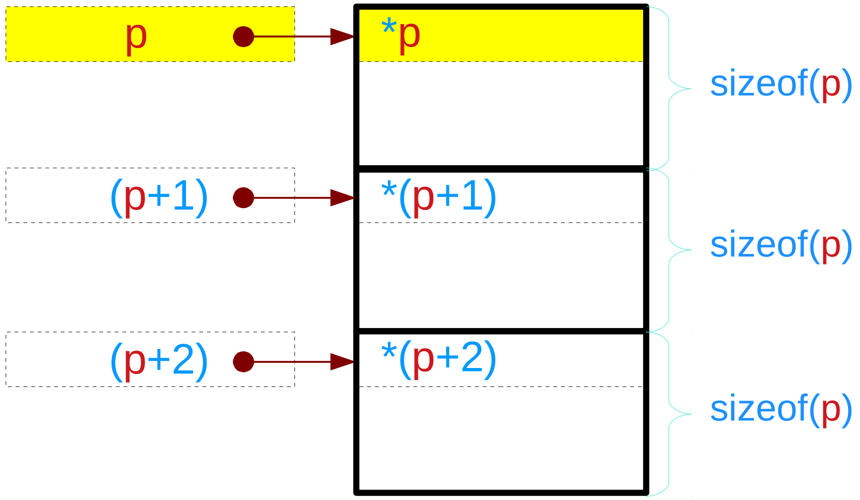


Using a **2-d** array pointer to a **2-d** array



An **n-d** array pointers

p: pointer to an *n*-d array

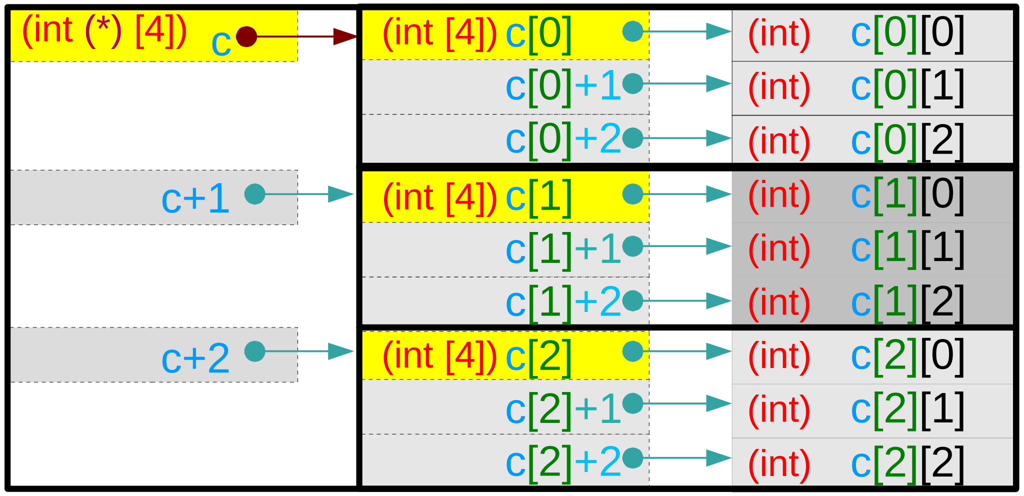


sizeof(p) = sizeof(*p) * 3 … *leading element*
sizeof(p+1) = pointer size
sizeof(p+2) = pointer size

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

A **2-d** array and its **1-d** sub-arrays – a type view

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



sizeof(c[0]) = sizeof(c[0][0]) * 3 … *leading element*
sizeof(c[0]+1) = pointer size
sizeof(c[0]+2) = pointer size

sizeof(c[1]) = sizeof(c[1][0]) * 3 … *leading element*
sizeof(c[1]+1) = pointer size
sizeof(c[1]+2) = pointer size

sizeof(c[2]) = sizeof(c[2][0]) * 3 … *leading element*
sizeof(c[2]+1) = pointer size
sizeof(c[2]+2) = pointer size

sizeof(c) = sizeof(c[0]) * 3 … *leading element*
sizeof(c+1) = pointer size
sizeof(c+2) = pointer size

value(c) = value(c[0]) … *leading element*
value(c+1) = value(c[0]) + sizeof(c[0]) * 1
value(c+2) = value(c[0]) + sizeof(c[0]) * 2

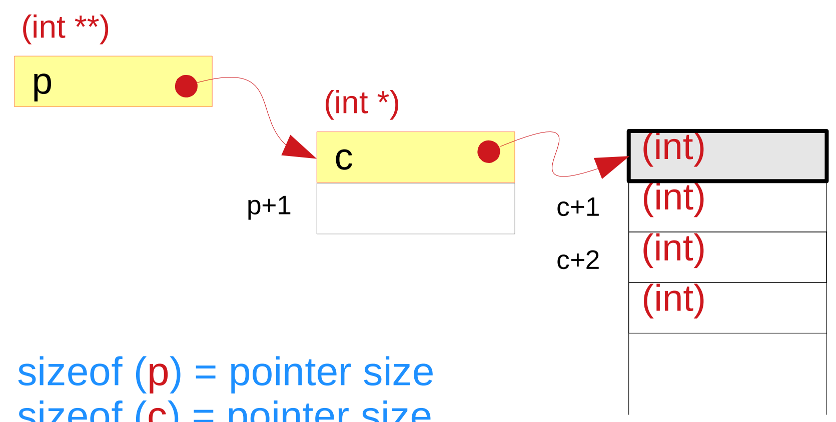
value(c[0]) = value(c[0][0]) … *leading element*
value(c[0]+1) = value(c[0][0]) + sizeof(c[0][0]) * 1
value(c[0]+2) = value(c[0][0]) + sizeof(c[0][0]) * 2

value(c[1]) = value(c[1][0]) … *leading element*
value(c[1]+1) = value(c[1][0]) + sizeof(c[1][0]) * 1
value(c[1]+2) = value(c[1][0]) + sizeof(c[1][0]) * 2

value(c[2]) = value(c[2][0]) … *leading element*
value(c[2]+1) = value(c[2][0]) + sizeof(c[2][0]) * 1
value(c[2]+2) = value(c[2][0]) + sizeof(c[2][0]) * 2

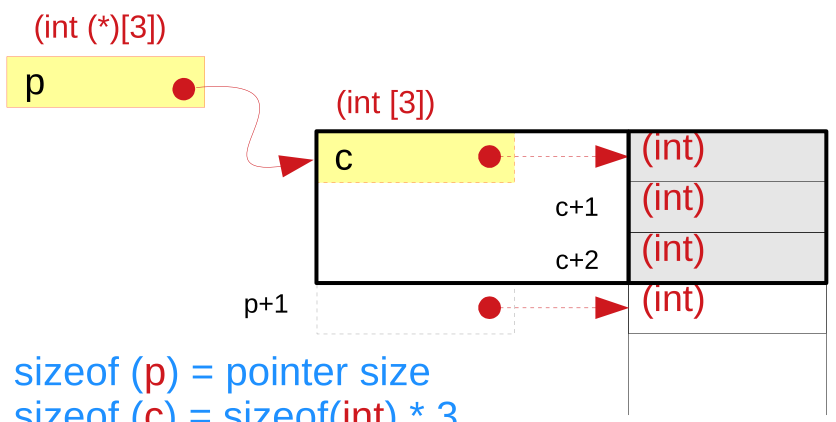
Integer pointer and array types – **int **, int (*)[3], int[2][3]**

int **p; **int *c;** v(&c) ≠ v(c)



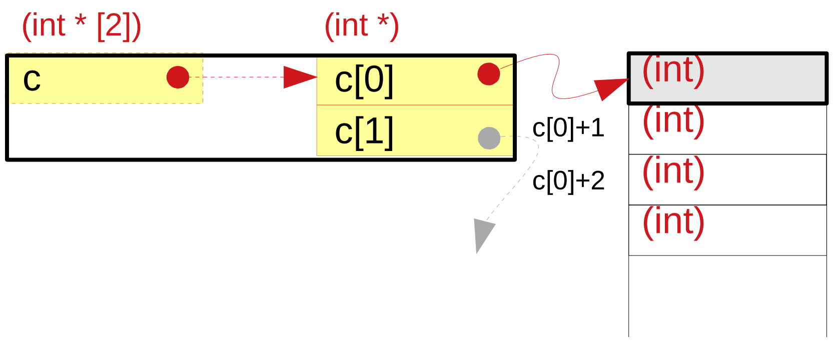
sizeof (p) = pointer size
sizeof (c) = pointer size

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



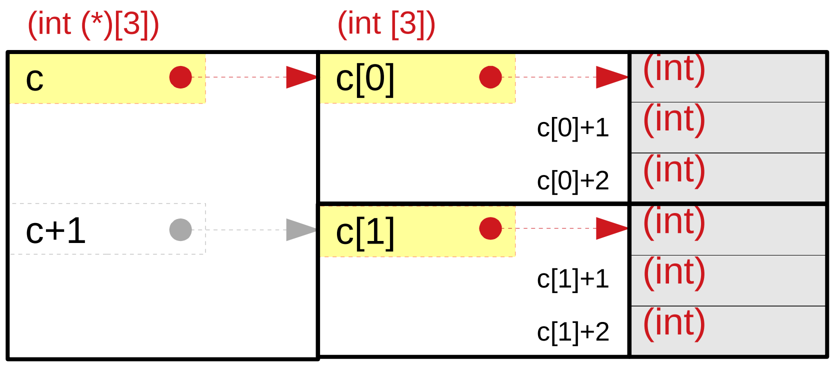
sizeof (p) = pointer size
sizeof (c) = sizeof(int) * 3

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

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

Multi-dimensional Array Pointers

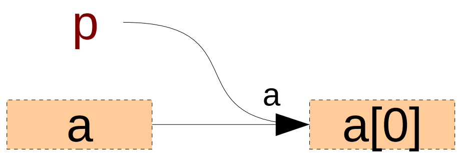
(*n-1*)-d array pointer to a *n*-d array

`int a[4] ;`	**1-d** array	
`int (*p) ;`	**0-d** array pointer	(p = a)
`int b[4] [2];`	**2-d** array	
`int (*q) [2];`	**1-d** array pointer	(q = b)
`int c[4] [2][3];`	**3-d** array	
`int (*r) [2][3];`	**2-d** array pointer	(r = c)
`int d[4] [2][3][4];`	**4-d** array	
`int (*s) [2][3][4];`	**3-d** array pointer	(s = d)

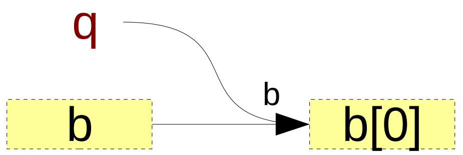
the 1st dimension can be accessed by incrementing (n-1)-d array pointer

***n*-d array name and (*n-1*)-d array pointer**

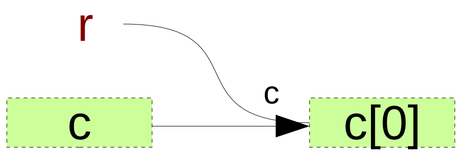
```
int a[4] ;                  p = &a[0];
int (*p) ;                  p = a;
```



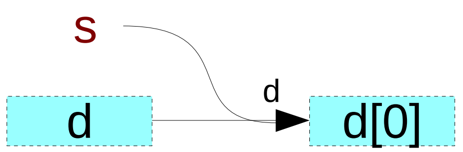
```
int b[4]  [2];              q = &b[0];
int (*q)  [2];              q = b;
```



```
int c[4]  [2][3];           r = &c[0];
int (*r)  [2][3];           r = c;
```



```
int d[4]  [2][3][4];        s = &d[0];
int (*s)  [2][3][4];        s = d;
```



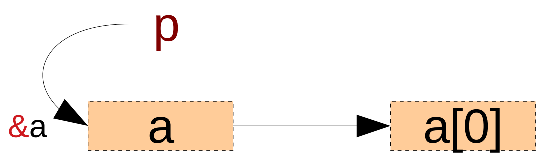
the 1st dimension can be accessed by incrementing (n-1)-d array pointer

***n*-d array pointer to a *n*-d array**

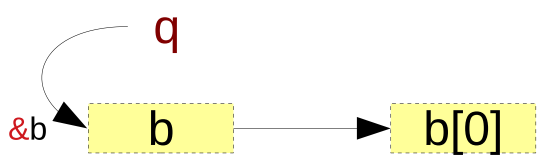
int a [4] ;	**1-d** array	
int (*p) [4];	**1-d** array pointer	(p = &a)
int b [4][2];	**2-d** array	
int (*q) [4][2];	**2-d** array pointer	(q = &b)
int c [4][2][3];	**3-d** array	
int (*r) [4][2][3];	**3-d** array pointer	(r = &c)
int d [4][2][3][4];	**4-d** array	
int (*s) [4][2][3][4];	**4-d** array pointer	(s = &d)

***n*-d** array name and ***n*-d** array pointer

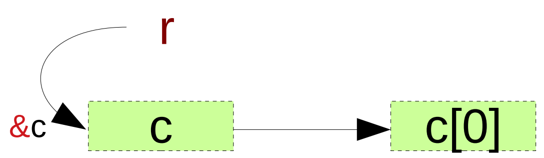
```
int  a    [4] ;          p = &a;
int (*p)  [4];
```



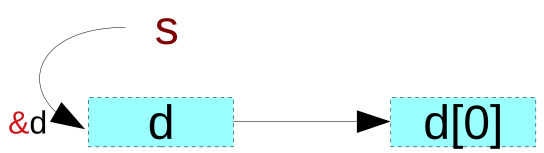
```
int  b    [4][2];        q = &b;
int (*q)  [4][2];
```



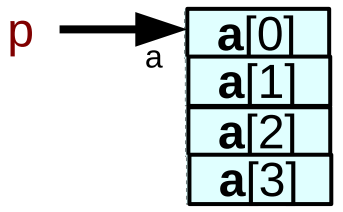
```
int  c    [4][2][3];     r = &c;
int (*r)  [4][2][3];
```

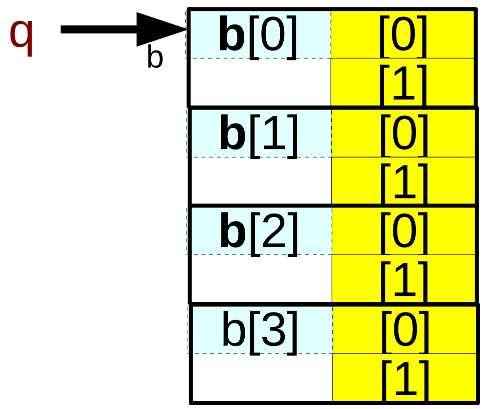


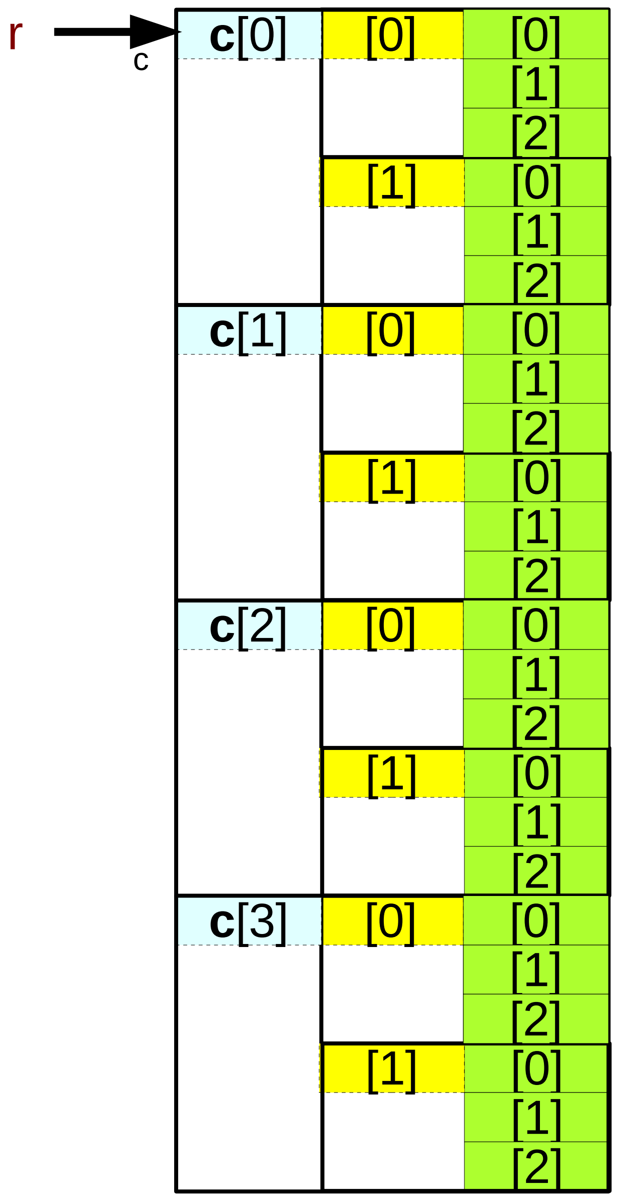
```
int  d    [4][2][3][4];  s = &d;
int (*s)  [4][2][3][4];
```



multi-dimensional array pointers

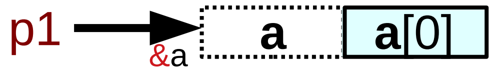




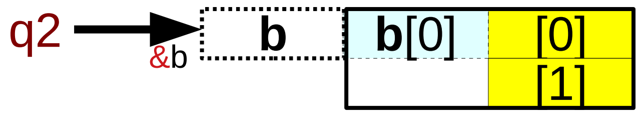


```
int a[4] ;                  1-d array
int (*p) ;                  0-d array pointer

int b[4]     [2];           2-d array
int (*q)     [2];           1-d array pointer

int c[4]     [2][3];        3-d array
int (*r)     [2][3];        2-d array pointer

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

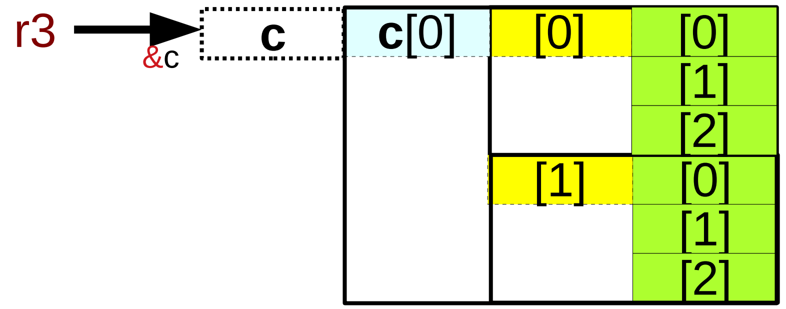
Initializing ***n-d*** array pointers



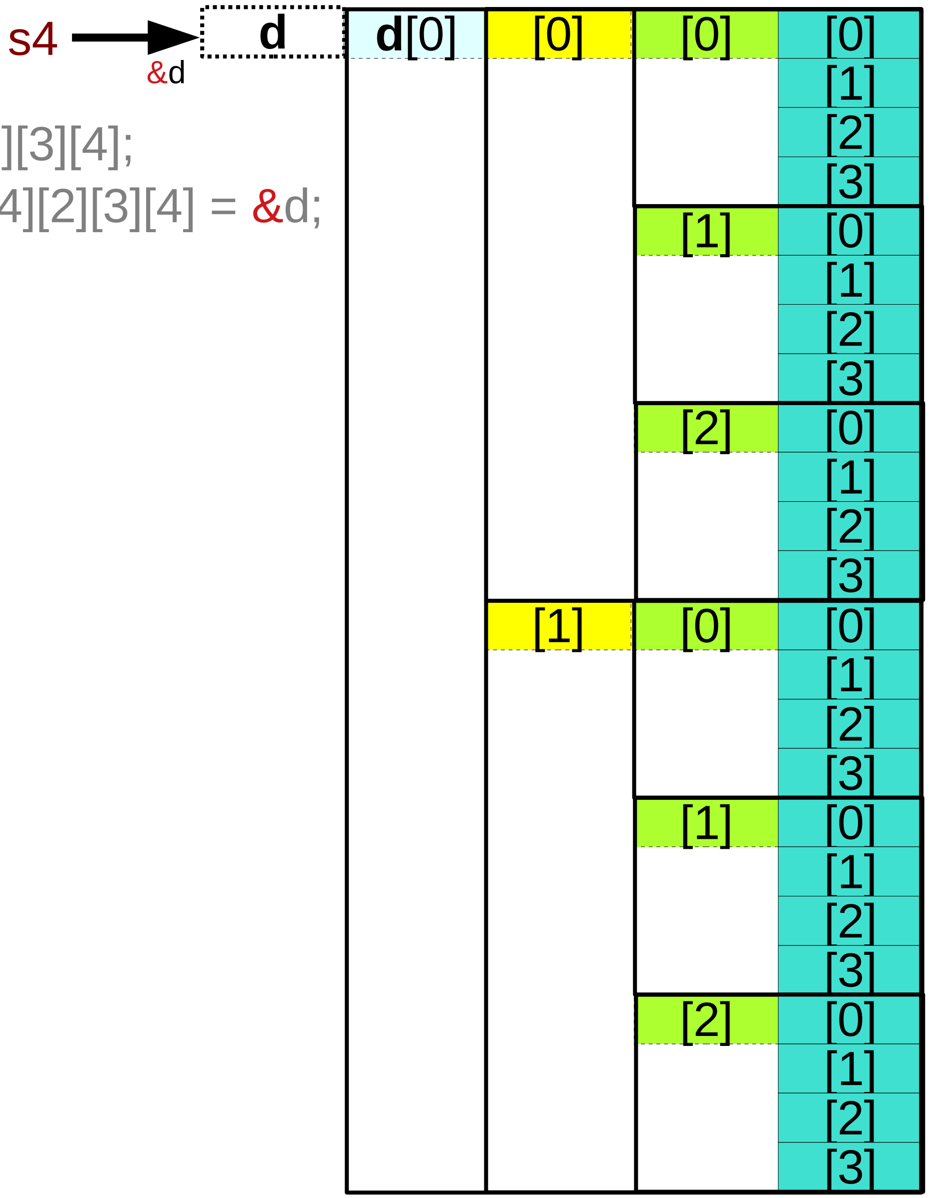
```
int a[4] ;
int (*p1)[4] = &a ;
```



```
int b[4][2];
int (*q2)[4][2] = &b;
```

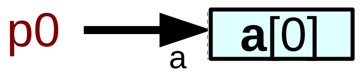


```
int c[4][2][3];
int (*r3)[4][2][3] = &c;
```

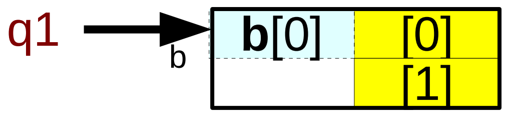


```
int d[4][2][3][4];
int (*s4)[4][2][3][4] = &d;
```

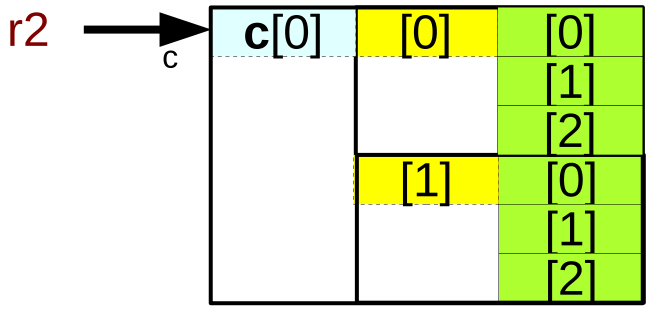
Initializing ***(n-1)-d*** array pointers



```
int a[4] ;
int (*p0) = a ;
```

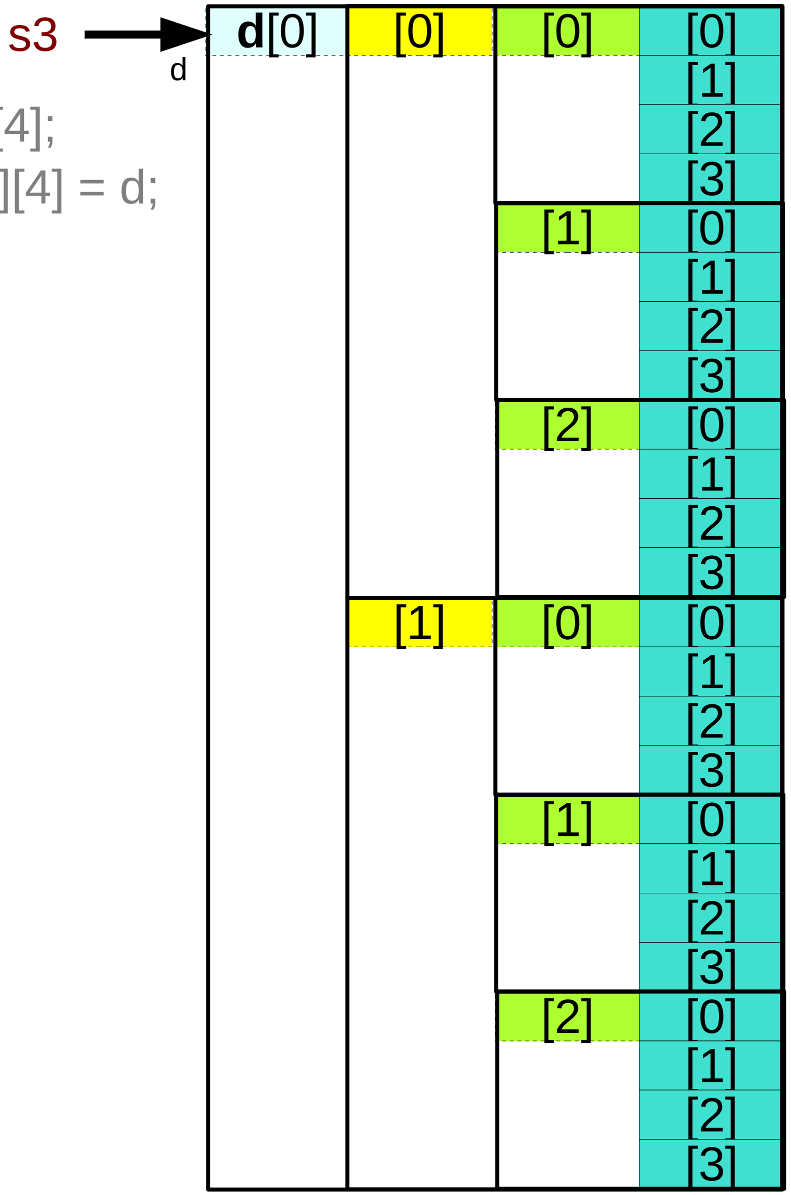


```
int b[4][2];
int (*q1)[2] = b;
```



```
int c[4][2][3];
int (*r2)[2][3] = c;
```

```
int d[4][2][3][4];
int (*s3)[2][3][4] = d;
```



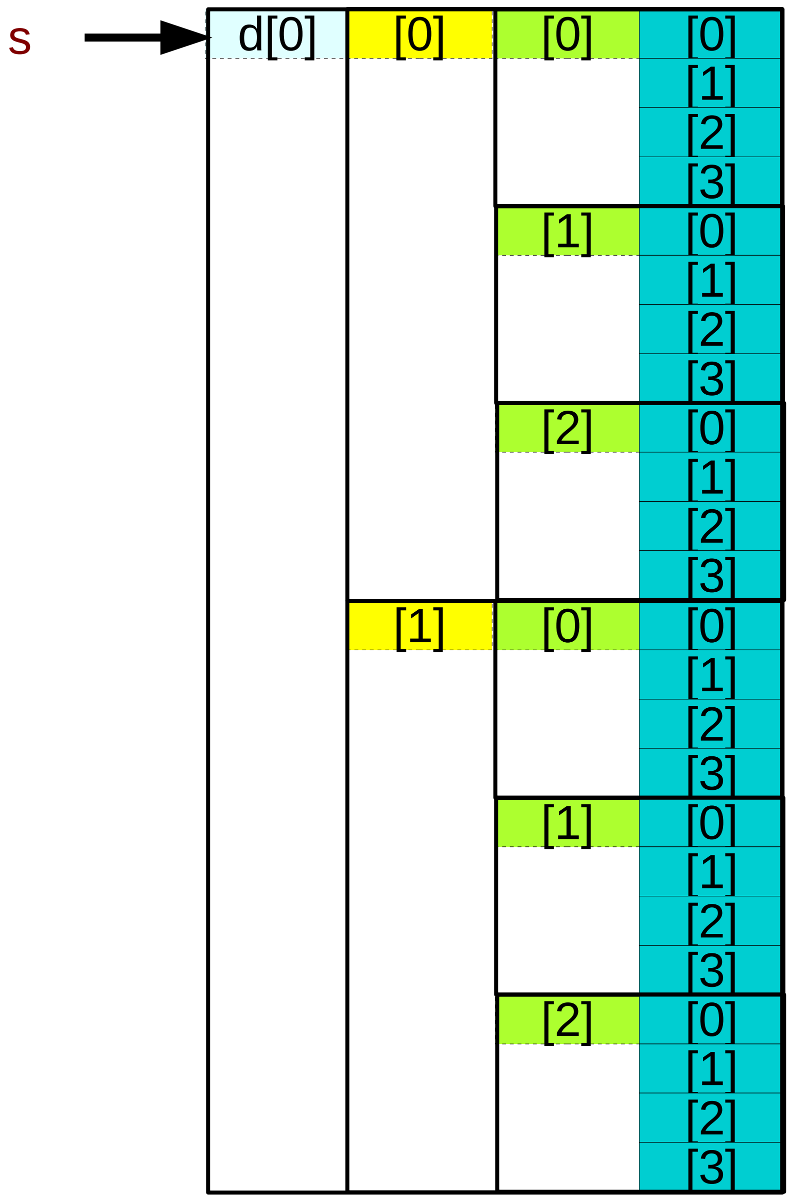
array pointers to multi-dimensional subarrays

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

d	4-d array name	d[4][2][3][4]
	3-d array pointer	(*p)[2][3][4]
d[i]	3-d array name	d[i][2][3][4]
	2-d array pointer	(*q)[3][4]
d[i][j]	2-d array name	d[i][j][3][4]
	1-d array pointer	(*r)[4]
d[i][j][k]	1-d array name	d[i][j][k][4]
	0-d array pointer	(*s)

i,j,k are specific index values

i =[0..3], j = [0..1], k= [0..2]



Initializing array pointers to multi-dimensional subarrays

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

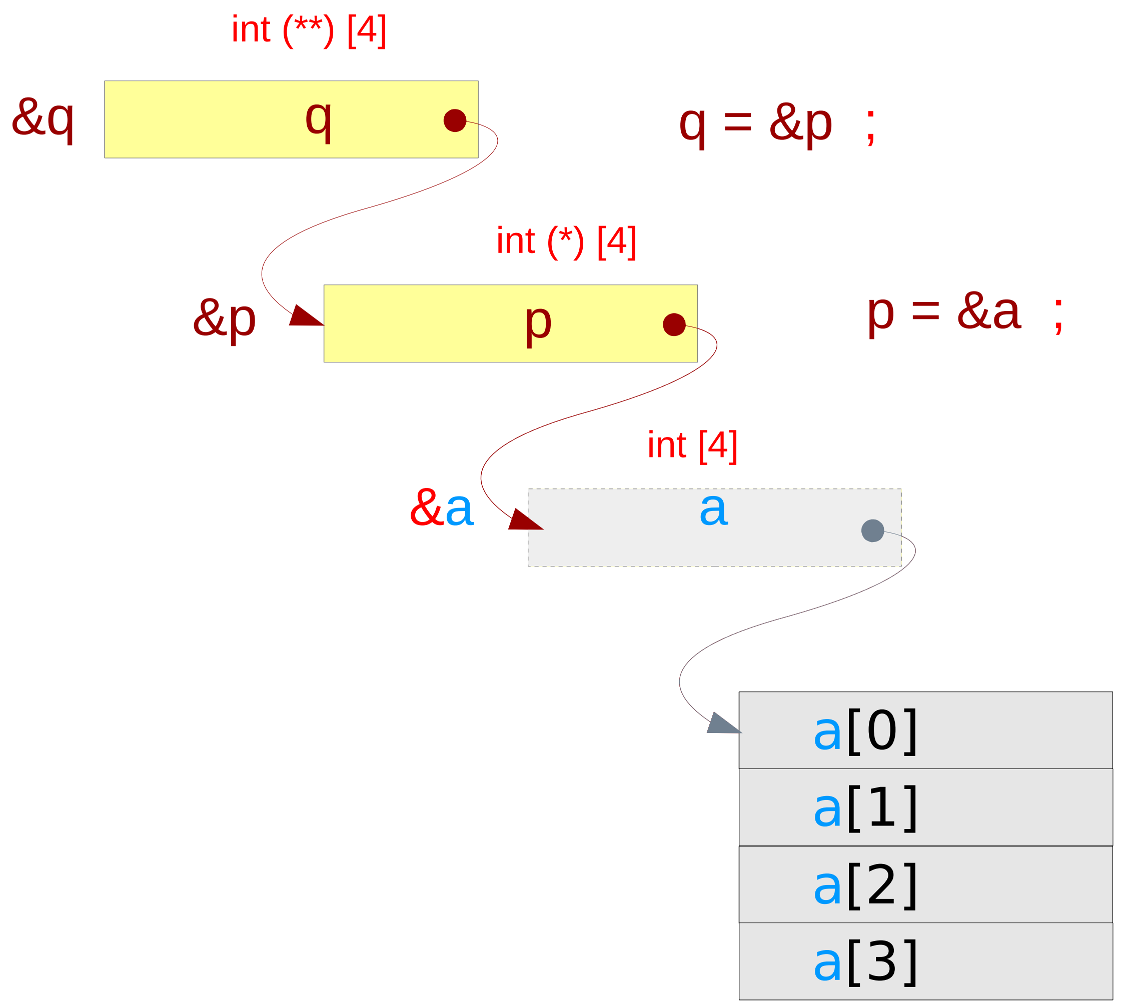
d	4-d array name	d[4][2][3][4]	p[i][j][k][l]
	3-d array pointer	(*p)[2][3][4]	int (*p)[2][3][4] = d;
d[i]	3-d array name	d[i][2][3][4]	q[j][k][l]
	2-d array pointer	(*q)[3][4]	int (*q)[3][4] = d[i];
d[i][j]	2-d array name	d[i][j][3][4]	r[k][l]
	1-d array pointer	(*r)[4]	int (*r)[4] = d[i][j];
d[i][j][k]	1-d array name	d[i][j][k][4]	s[l]
	0-d array pointer	(*s)	int (*s) = d[i][j][k];

i =[0..3], j = [0..1], k= [0..2]

Passing multidimensional array names

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

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

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

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

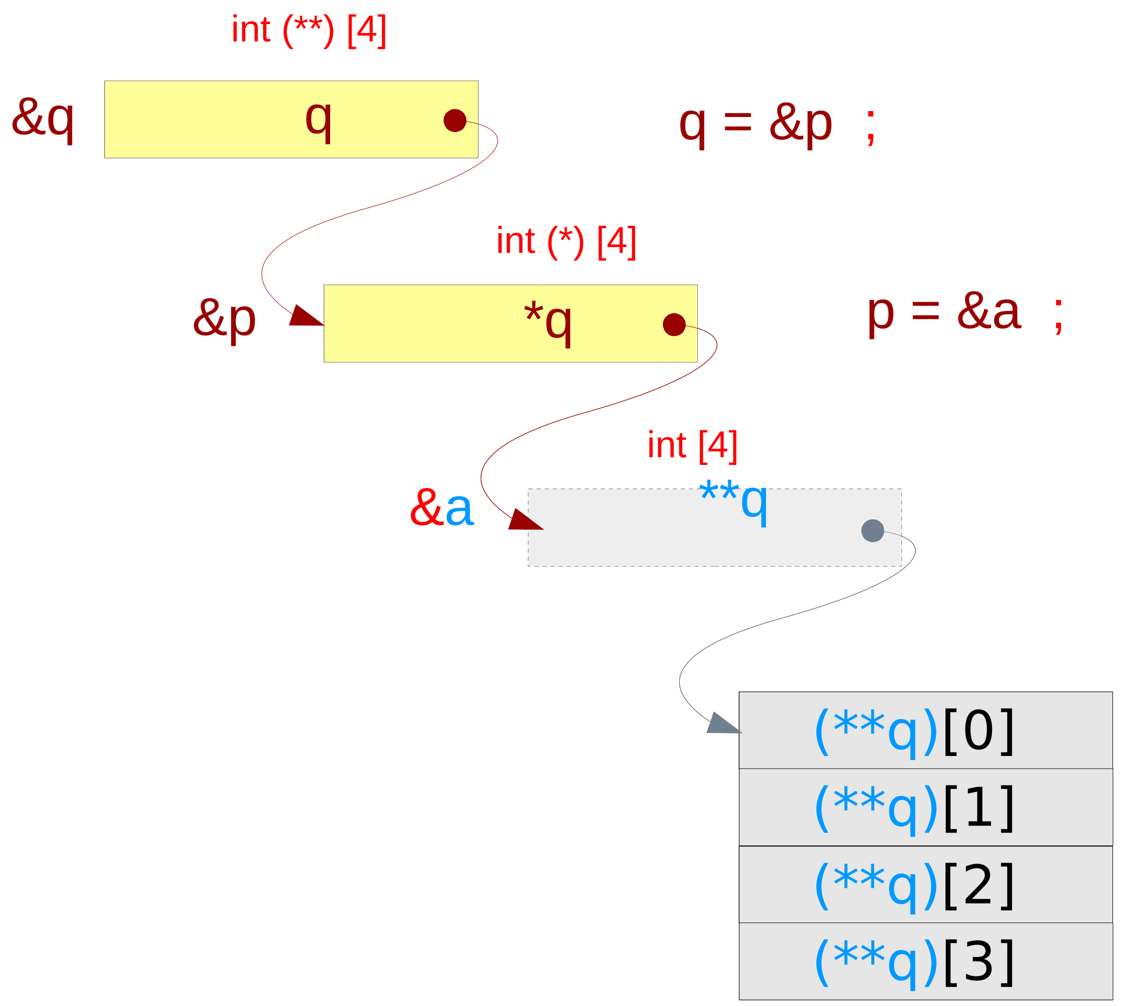
Double pointer to a **1-d** array – a variable view (p, q)



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

```
➡  p = &a ;
➡  q = &p ;
```

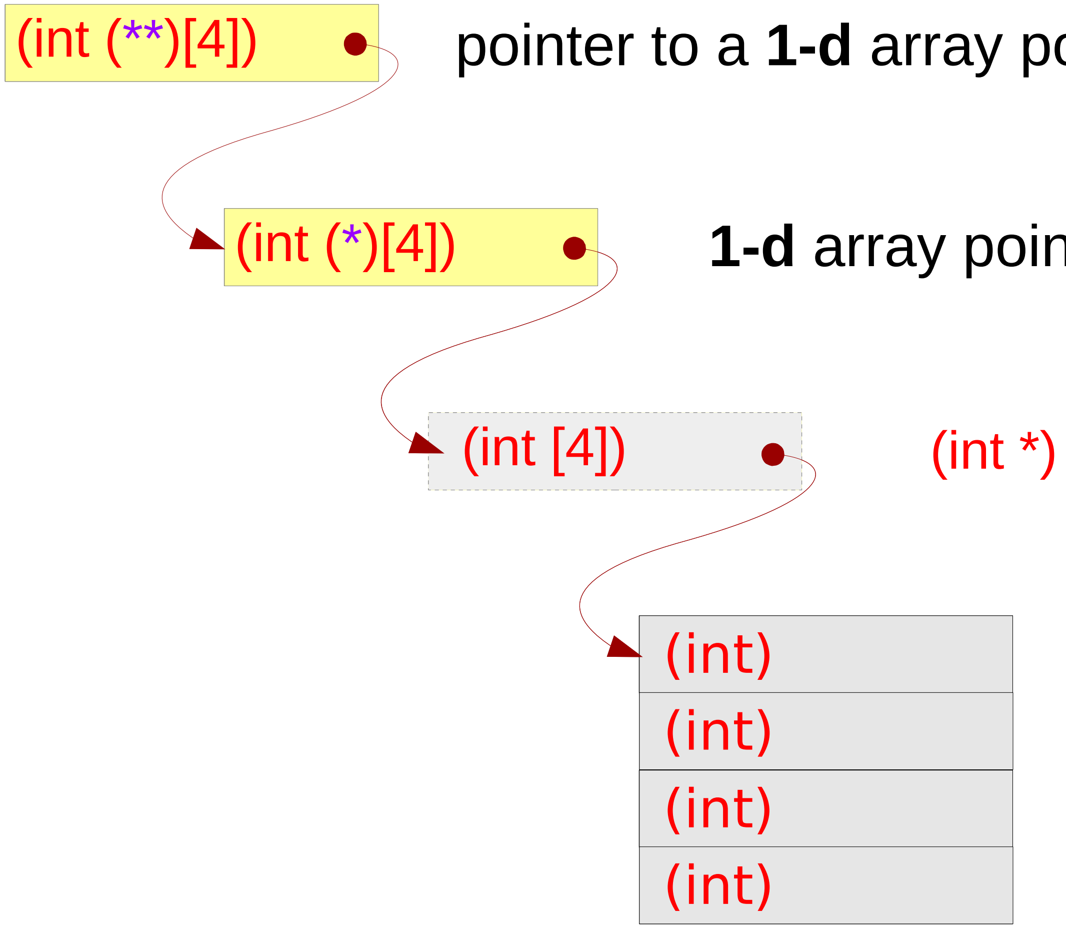
Double pointer to a **1-d** array – a variable view (q)



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

```
➡  p = &a ;
➡  q = &p ;
```

Double pointer to a **1-d** array – a type view



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

```
➡   p = &a ;
➡   q = &p ;
```

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