

Applications of Array Pointers (1A)

Copyright (c) 2021 - 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);
}
```

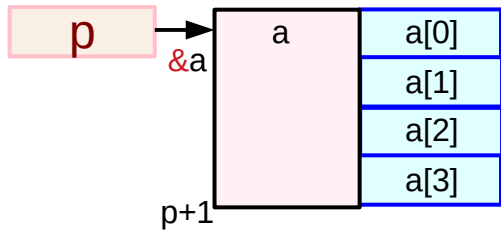
`c= 0x7fffd923487c`

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

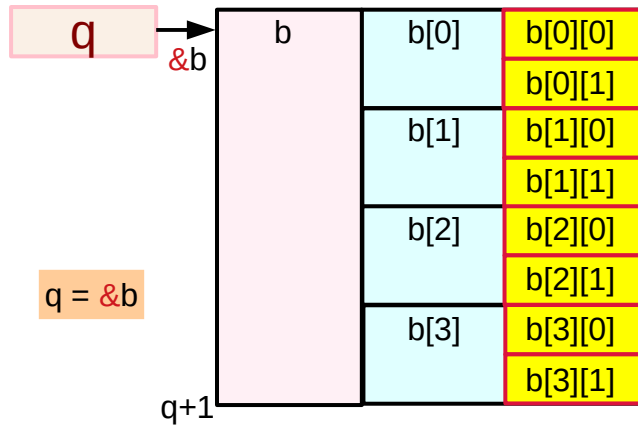
t.c: In function 'main':
t.c:5:16: warning: format '%d' expects argument of type 'int',
but argument 2 has type 'int (*)[3]' [-Wformat=]
printf ("c= %d \n", &c);

Multi-dimensional Array Pointers

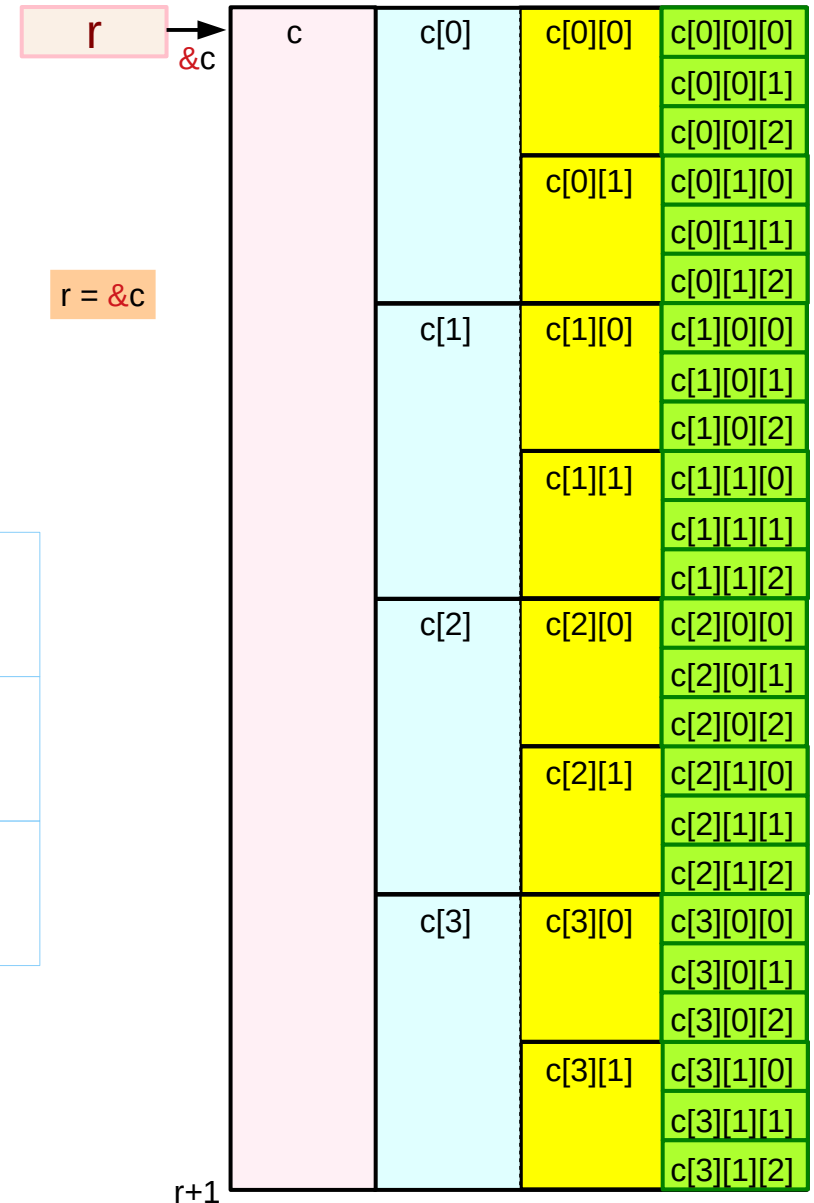
Arrays a, b, c and array pointers p, q, r



`p = &a`



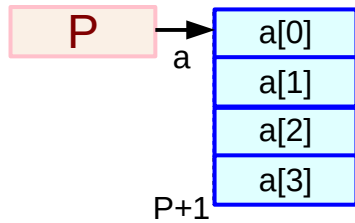
`q = &b`



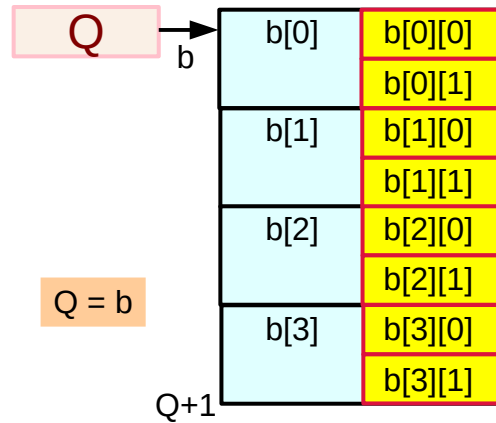
`r = &c`

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

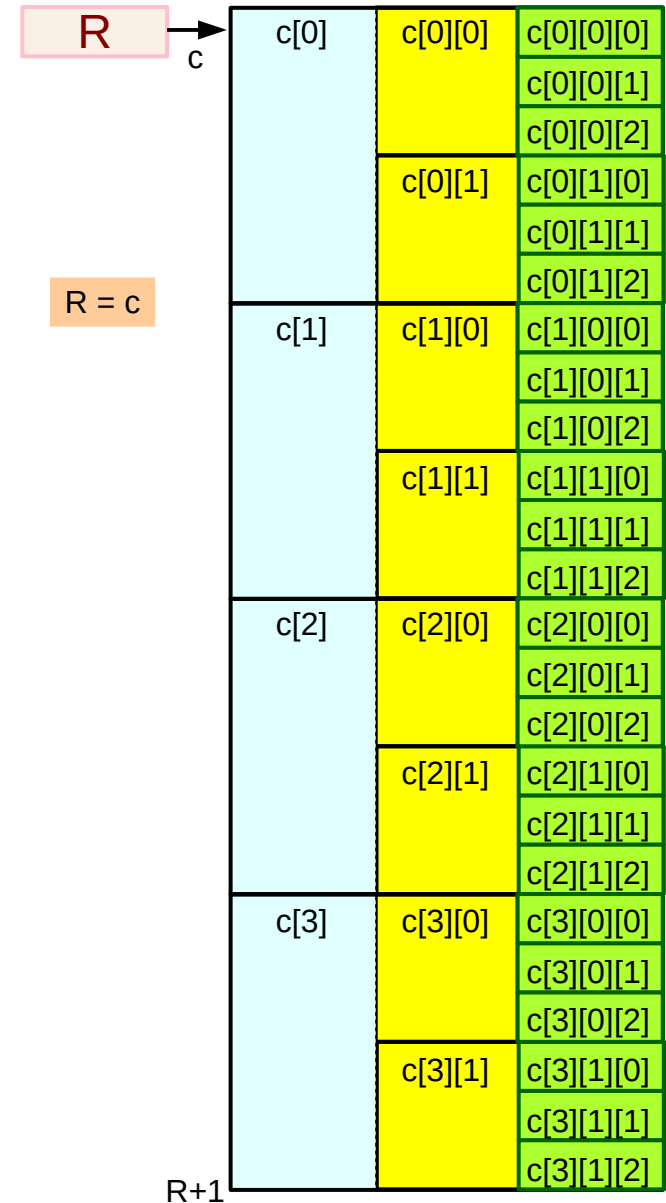
Arrays a, b, c and array pointers A, B, C



`P = a`



`Q = b`



`R = c`

<code>int a[4];</code>	1-d array	
<code>int (*P);</code>	0-d array pointer	<code>P = a</code>
<code>int b[4][2];</code>	2-d array	
<code>int (*Q)[2];</code>	1-d array pointer	<code>Q = b</code>
<code>int c[4][2][3];</code>	3-d array	
<code>int (*R)[2][3];</code>	2-d array pointer	<code>R = c</code>

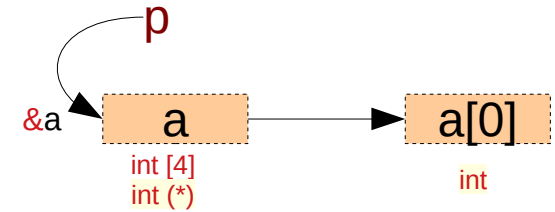
n-d array pointer to a *n*-d array

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

1-d array a

1-d array pointer p

(p = &a)

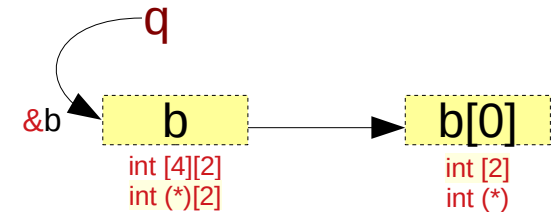


```
int b [4][2];  
int (*q) [4][2];
```

2-d array b

2-d array pointer q

(q = &b)

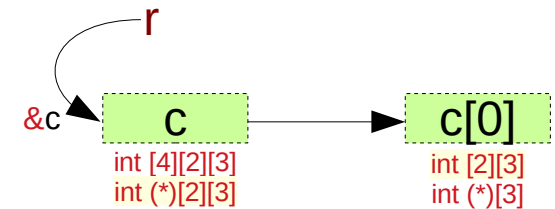


```
int c [4][2][3];  
int (*r) [4][2][3];
```

3-d array c

3-d array pointer r

(r = &c)

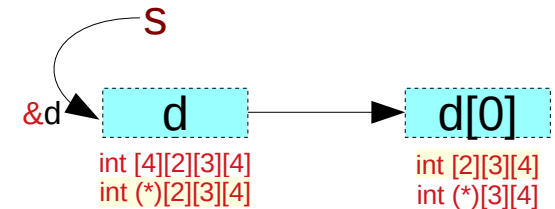


```
int d [4][2][3][4];  
int (*s) [4][2][3][4];
```

4-d array d

4-d array pointer s

(s = &d)



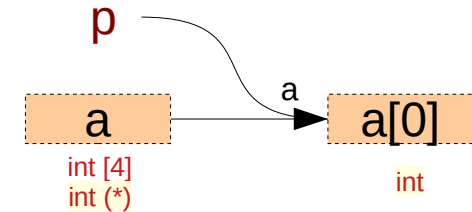
$(n-1)$ -d array pointer to a n -d array

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

1-d array a

0-d array pointer p

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

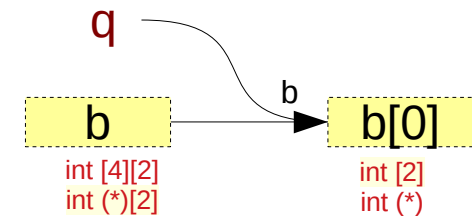


```
int b[4][2];  
int (*q)[2];
```

2-d array b

1-d array pointer q

```
q = &b[0];  
(q = b)
```

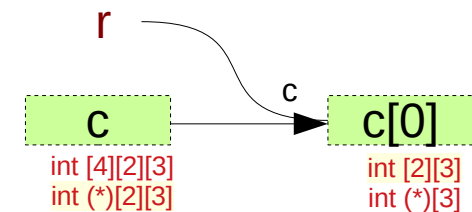


```
int c[4][2][3];  
int (*r)[2][3];
```

3-d array c

2-d array pointer r

```
r = &c[0];  
(r = c)
```

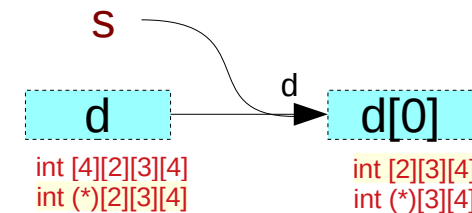


```
int d[4][2][3][4];  
int (*s)[2][3][4];
```

4-d array d

3-d array pointer s

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



Array and array pointer types

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

1-d array **a**



```
int (*)
```

a as a 0-d array pointer

```
int (*A);
```

0-d array pointer **A**



```
int [N]
```

A as a 1-d array

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

2-d array **b**



```
int (*) [2]
```

b as a 1-d array pointer

```
int (*B)[2];
```

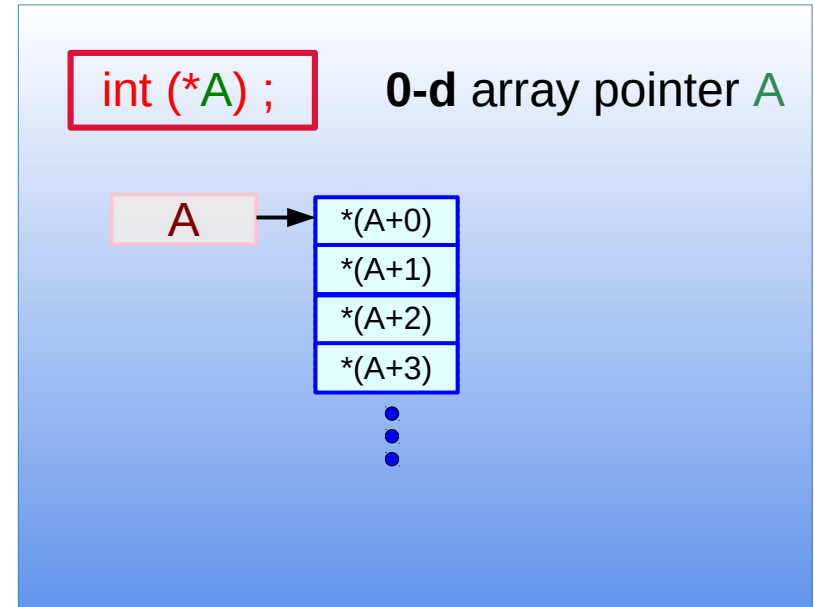
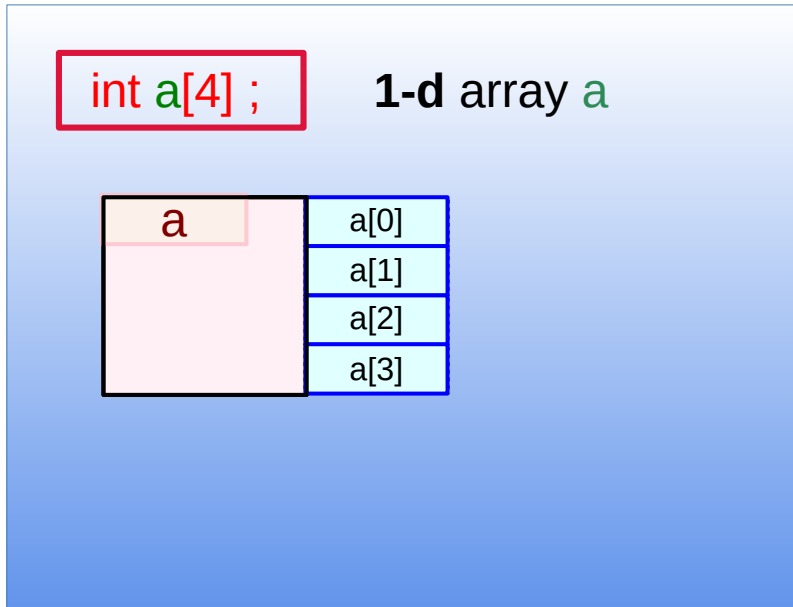
1-d array pointer **B**



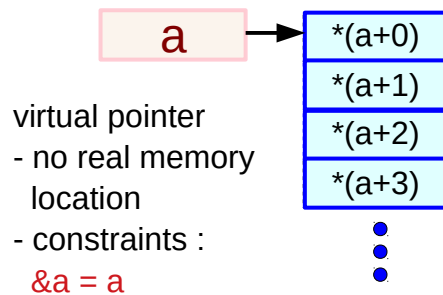
```
int [N][2]
```

B as a 2-d array

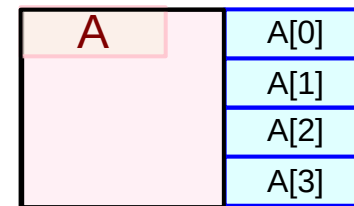
Array **a** vs array pointer **A**



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



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



N is not fixed to 4

`sizeof(A)` is not the size of the array but the size of a pointer variable

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

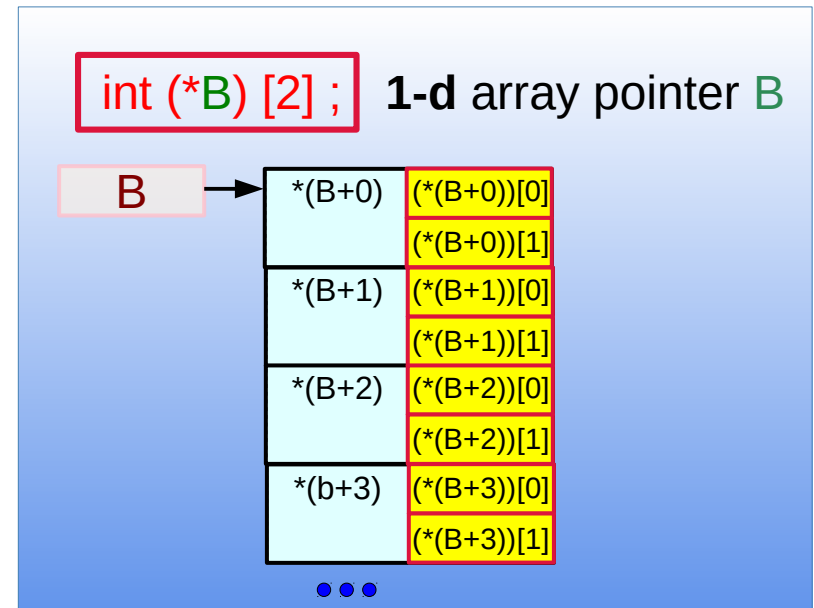
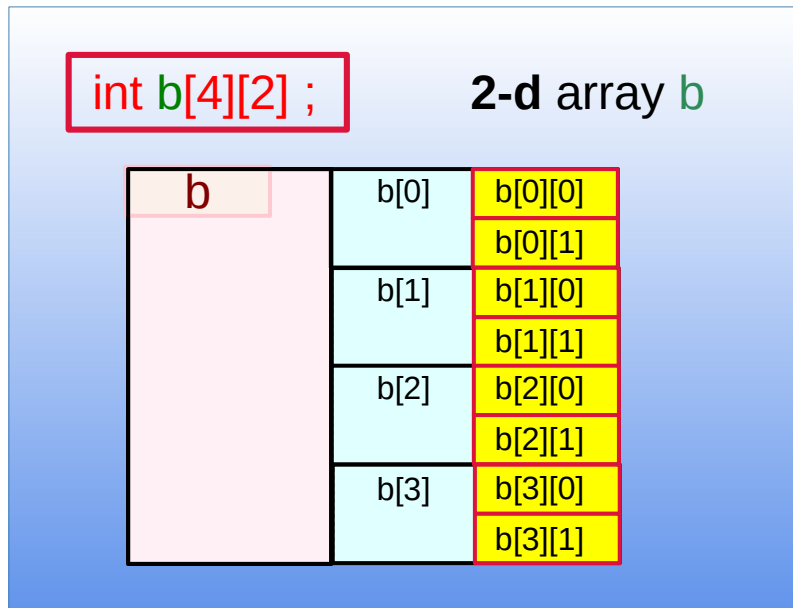
- `sizeof(a)` = an array size
- **a** = `&a`

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

A is not a real array

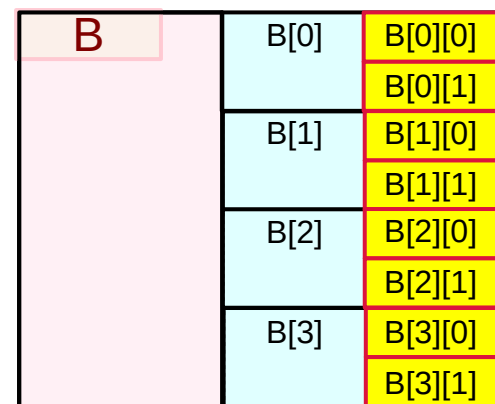
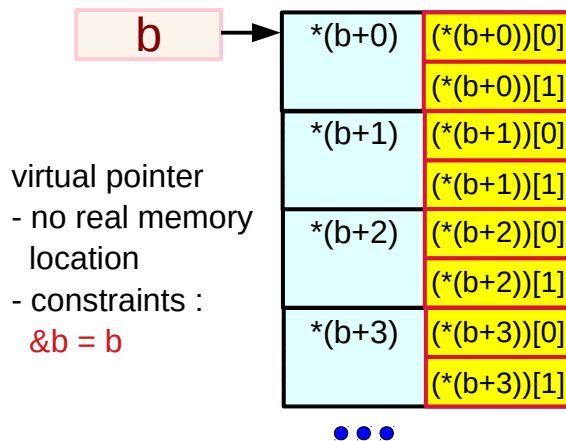
- `sizeof(A)` = a pointer size
- **A** ≠ `&A`

Array **b** and array pointers **B**



`int (*) [2]` **b** as a 1-d array pointer

`int [N][2]` **B** as a 2-d array



N is not fixed to 4

`sizeof(B)` is not the size of the array but the size of a pointer variable

Array **b** and array pointers **B**

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

2-d array **b**

- `sizeof(b)` = an array size
= 4 * 2 * 4 bytes
- # of 1-d arrays = fixed
= 4

```
int (*) [2]
```

b as a 1-d array pointer

b is not a real pointer

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

```
int (*B) [2];
```

1-d array pointer **B**

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

```
int [N][2]
```

B as a 2-d array

B is not a real array

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

Dual type - relaxing the 1st dimension of an array

`int a[4];` *array declaration*

`int [4]` **1-d array**
more constrained type
Like a **subtype**

`int (*)` **0-d array pointer**
more general type
Like a **supertype**

`int (*A);` *pointer declaration*

`int (*)` **0-d array pointer**
more general type
Like a **supertype**

`int [N]` **1-d array**
more constrained type
Like a **subtype**

`int b[4][2];` *array declaration*

`int [4][2]` **2-d array**
more constrained type
Like a **subtype**

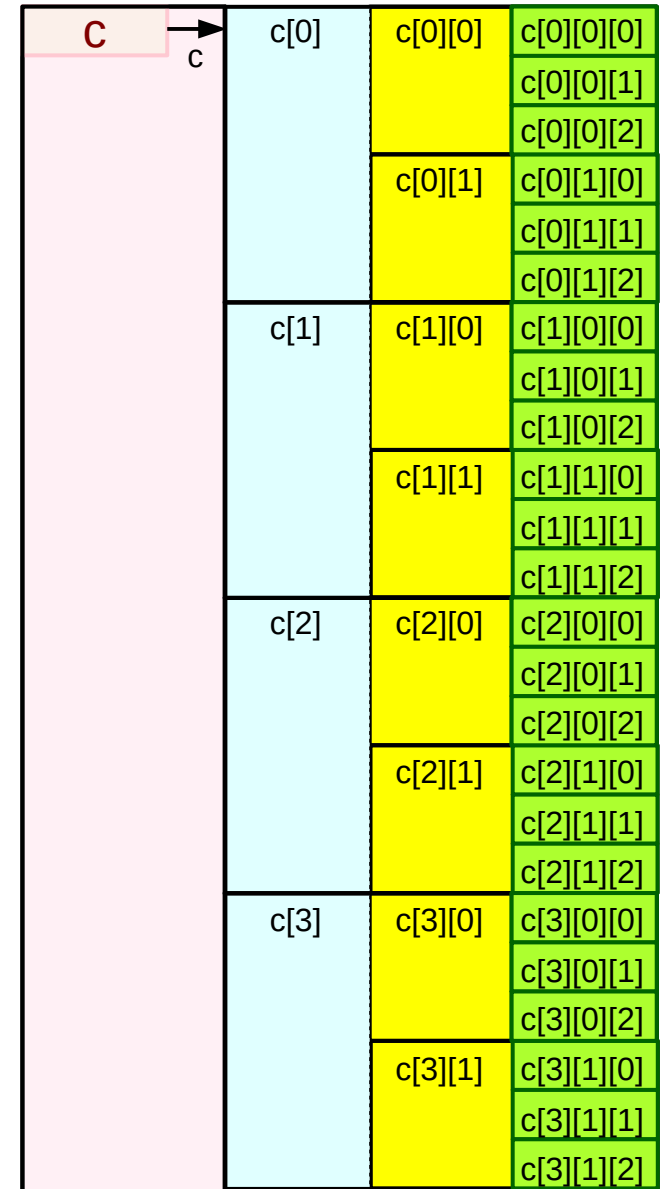
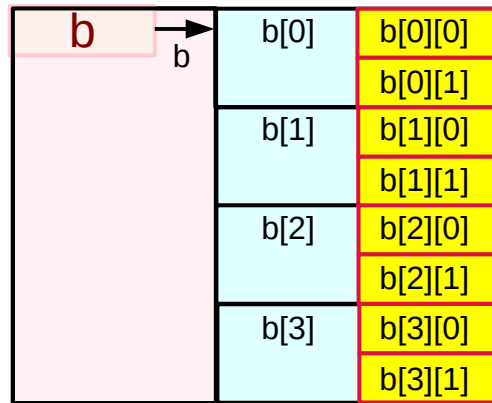
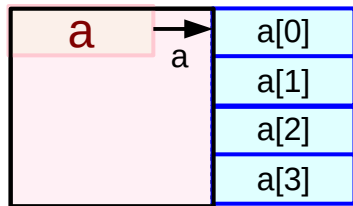
`int (*)[2]` **1-d array pointer**
more general type
Like a **supertype**

`int (*B)[2];` *pointer declaration*

`int (*)[2]` **1-d array pointer**
more general type
Like a **supertype**

`int [N][2]` **2-d array**
more constrained type
Like a **subtype**

Array names as array pointers: a, b, c



dual types

<code>int [4]</code>	1-d array a
<code>int (*)</code>	0-d array pointer a (virtual)
<code>int [4][2];</code>	2-d array b
<code>int (*)[2];</code>	1-d array pointer b (virtual)
<code>int [4][2][3];</code>	3-d array c
<code>int (*)[2][3];</code>	2-d array pointer c (virtual)

Dual types of subarrays

```
int d[4][2][3][4];
```

types

d	consider d [4][2][3][4] relax the 1 st dimension	→ int [4][2][3][4] → int (*)[2][3][4]	⇒ 4-d array ⇒ 3-d array pointer (virtual)
d[i]	consider d [i][2][3][4] relax the 1 st dimension	→ int [2][3][4] → int (*)[3][4]	⇒ 3-d array ⇒ 2-d array pointer (virtual)
d[i][j]	consider d [i][j][3][4] relax the 1 st dimension	→ int [3][4] → int (*)[4]	⇒ 2-d array ⇒ 1-d array pointer (virtual)
d[i][j][k]	consider d [i][j][k][4] relax the 1 st dimension	→ int [4] → int (*)	⇒ 1-d array ⇒ 0-d array pointer (virtual)

i,j,k are specific index values i=[0..3], j=[0..1], k=[0..2]

Initializing *n-d* array pointers with *n-d* subarrays

```
int d[4][2][3][4];
```

<code>d</code>	4-d array	<code>d[4][2][3][4]</code>	<code>p = &d</code> (<code>*p=d</code>)
<code>p</code>	4-d array pointer	<code>(*p)[4][2][3][4]</code>	<code>int (*p)[4][2][3][4] = &d;</code> <code>(*p)[i][j][k][l] ≡ d[i][j][k][l]</code>
<code>d[i]</code>	3-d array	<code>d[i][2][3][4]</code>	<code>q = &d[i]</code> (<code>*q=d[i]</code>)
<code>q</code>	3-d array pointer	<code>(*q)[2][3][4]</code>	<code>int (*q)[3][4] = &d[i];</code> <code>(*q)[j][k][l] ≡ d[i][j][k][l]</code> given i
<code>d[i][j]</code>	2-d array	<code>d[i][j][3][4]</code>	<code>r = &d[i][j]</code> (<code>*r=d[i][j]</code>)
<code>r</code>	2-d array pointer	<code>(*r)[3][4]</code>	<code>int (*r)[4] = &d[i][j];</code> <code>(*r)[k][l] ≡ d[i][j][k][l]</code> given i, j
<code>d[i][j][k]</code>	1-d array	<code>d[i][j][k][4]</code>	<code>s = &d[i][j][k]</code> (<code>*s=d[i][j][k]</code>)
<code>s</code>	1-d array pointer	<code>(*s)[4]</code>	<code>int (*s) = &d[i][j][k];</code> <code>(*s)[l] ≡ d[i][j][k][l]</code> given i, j, k

Initializing ($n-1$)-d array pointers with n -d subarrays

```
int d[4][2][3][4];
```

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

Passing a *n-d* array pointer

1-d array

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



call

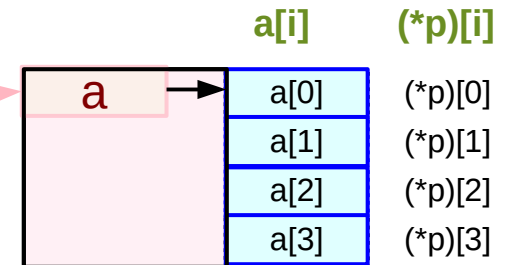
```
a :: int [4]  
&a :: int (*) [4]  
funa(&a, ...);
```

prototype

```
void funa(int (*p)[4], ...);
```

1-d array pointer

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



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

1-d array

`int a[4];`



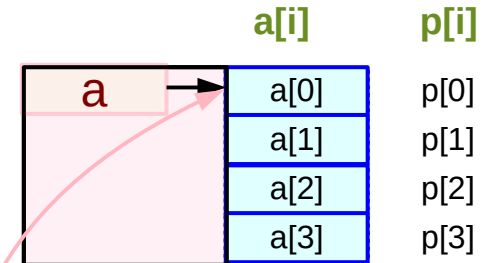
call `a :: int[4]`
`funa(a, ...);`

prototype

`void funa(int (*p), ...)` {

0-d array pointer

`p :: int(*)`



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

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

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

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

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

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

Passing *n*-d array pointers

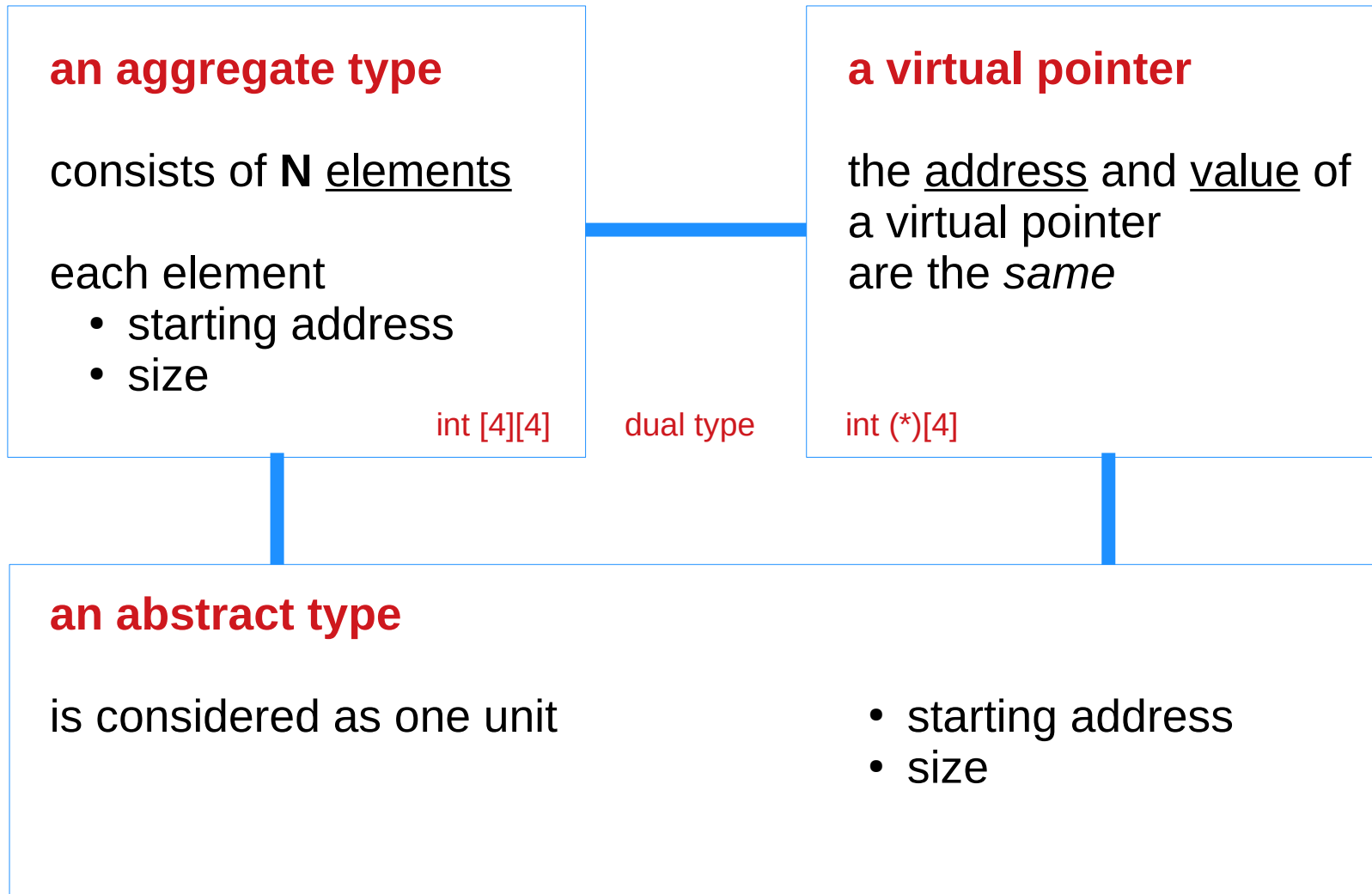
1-d array <code>int a [4] ;</code> <code>a[i]</code>	call <code>funa(&a, ...);</code>	prototype 1-d array pointer <code>void funa(int (*p)[4], ...);</code> <code>(*p)[i]</code>
2-d array <code>int b [4][2];</code> <code>b[i][j]</code>	call <code>funb(&b, ...);</code>	prototype 2-d array pointer <code>void funb(int (*q)[4][2], ...);</code> <code>(*q)[i][j]</code>
3-d array <code>int c [4][2][3];</code> <code>c[i][j][k]</code>	call <code>func(&c, ...);</code>	prototype 3-d array pointer <code>void func(int (*r)[4][2][3], ...);</code> <code>(*r)[i][j][k]</code>
4-d array <code>int d [4][2][3][4];</code> <code>d[i][j][k][l]</code>	call <code>fund(&d, ...);</code>	prototype 4-d array pointer <code>void fund(int (*s)[4][2][3][4], ...);</code> <code>(*s)[i][j][k][l]</code>

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

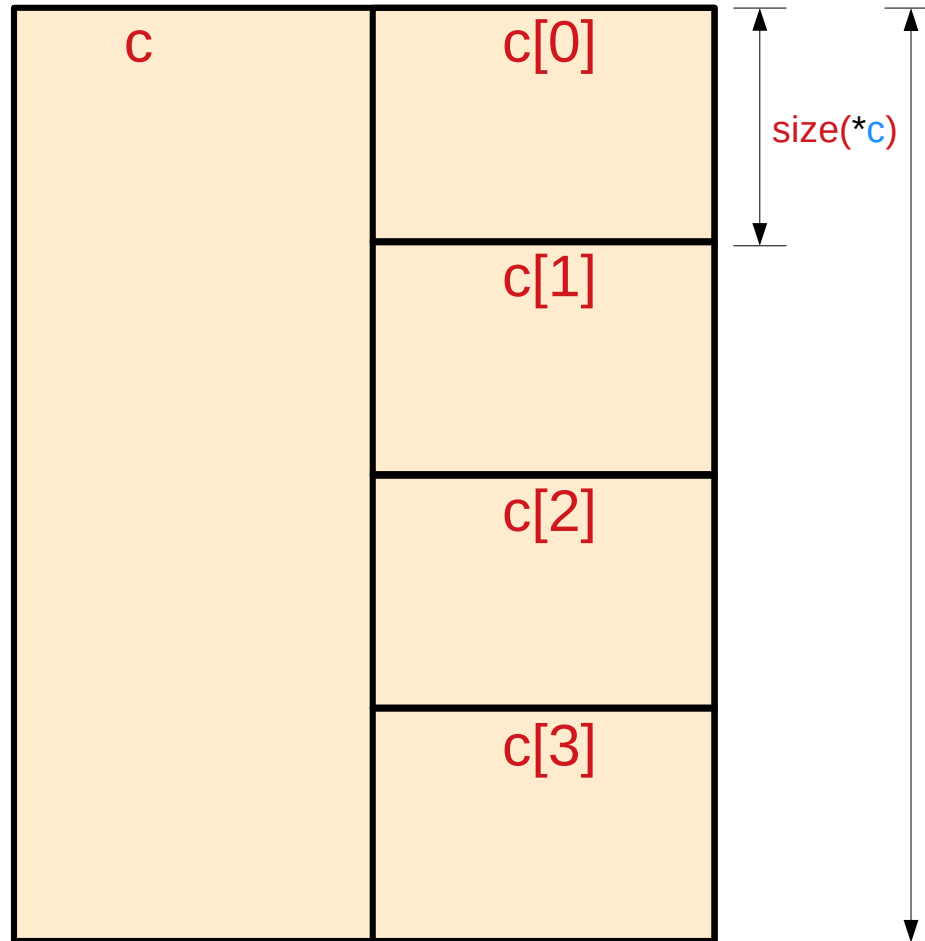
1-d array <code>int a[4];</code> <code>a[i]</code>	call <code>funa(a, ...);</code>	prototype 0-d array pointer <code>void funa(int (*p), ...);</code> <code>p[i]</code>
2-d array <code>int b[4][2];</code> <code>b[i][j]</code>	call <code>funb(b, ...);</code>	prototype 1-d array pointer <code>void funb(int (*q)[2], ...);</code> <code>q[i][j]</code>
3-d array <code>int c[4][2][3];</code> <code>c[i][j][k]</code>	call <code>func(c, ...);</code>	prototype 2-d array pointer <code>void func(int (*r)[2][3], ...);</code> <code>r[i][j][k]</code>
4-d array <code>int d[4][2][3][4];</code> <code>d[i][j][k][l]</code>	call <code>fund(d, ...);</code>	prototype 3-d array pointer <code>void fund(int (*s)[2][3][4], ...);</code> <code>s[i][j][k][l]</code>

Aggregate Data Types
Abstract Data Types
Virtual Array Pointers

Aggregate data type



Aggregate data **c**

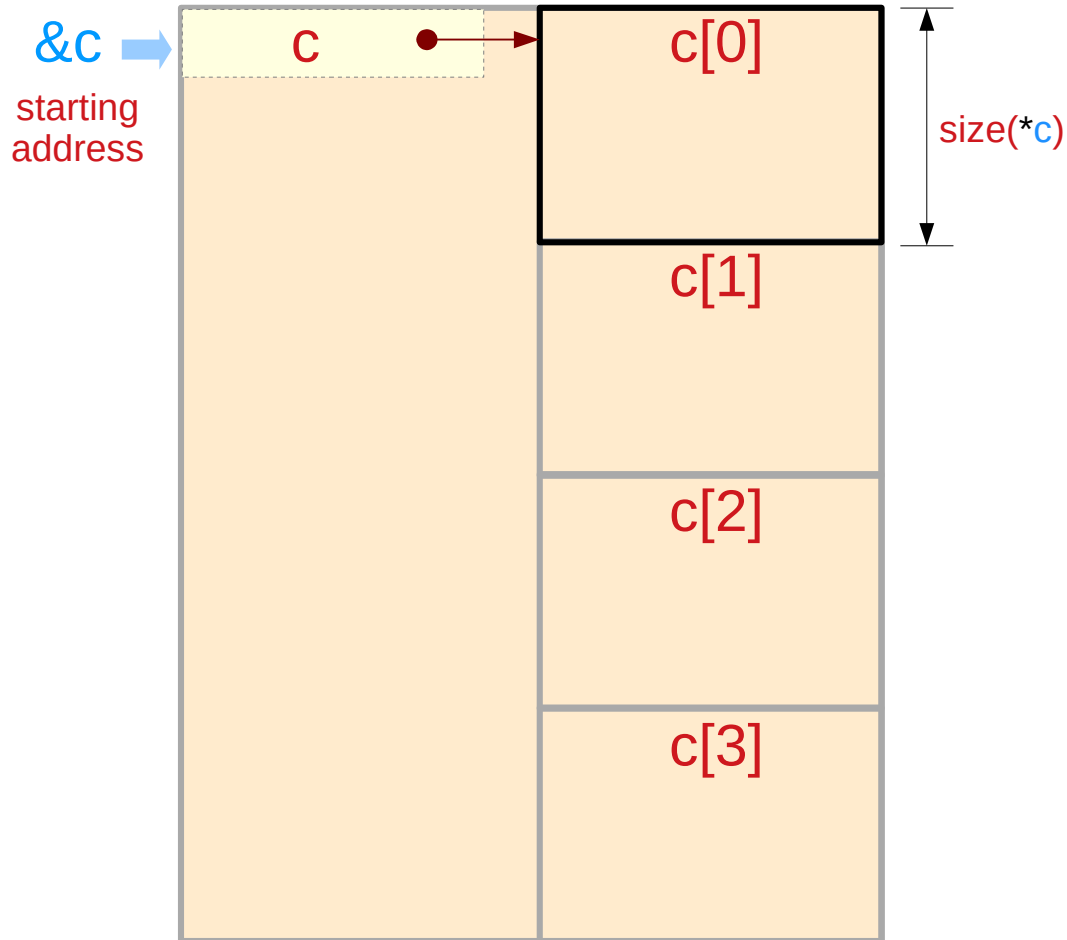


an aggregate type data **c**

- 1st element **c[0]**
- 2nd element **c[1]**
- 3rd element **c[2]**
- 4th element **c[3]**

Virtual pointer **c**

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



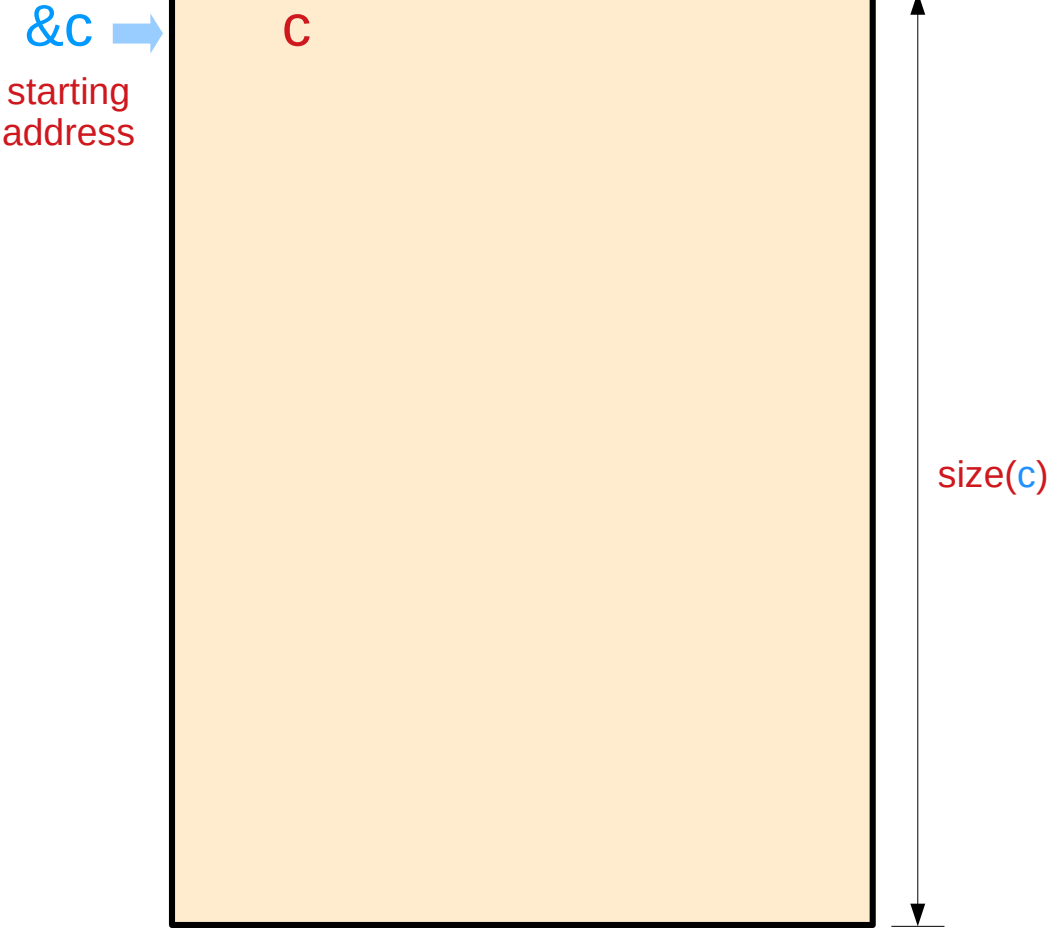
a virtual pointer **c**
- pointer address $\&c$
- pointer value $c = \&c[0]$

with the constraint
 $c = \&c$

an abstract data **c[i]**
- start address $\&c[i]$
- size $\text{sizeof}(c[i])$

relationship between **c** and **c[i]**
virtual pointer **c**
points to abstract data **c[i]**

Abstract data **c**

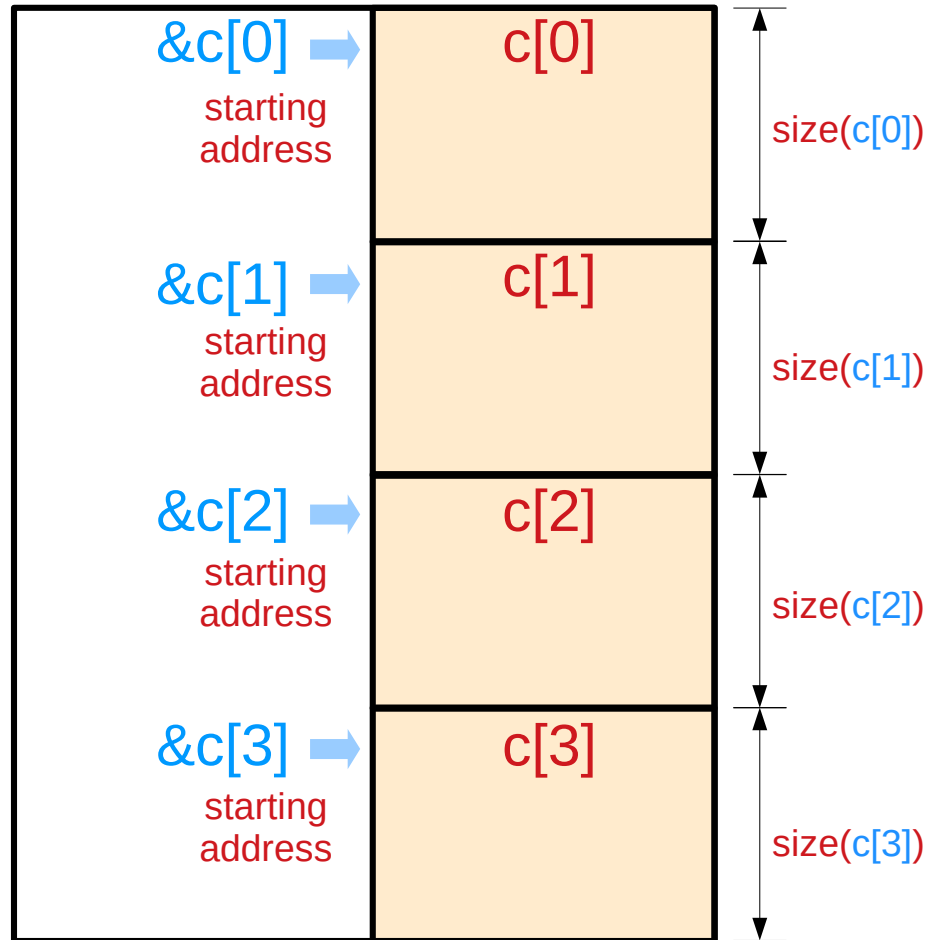


an abstract data

- start address
- size

c
&c
sizeof(c)

Abstract data $c[i]$



- an abstract data
 - start address $c[0]$
 - size $\&c[0]$
 - $\text{sizeof}(c[0])$
- an abstract data
 - start address $c[1]$
 - size $\&c[1]$
 - $\text{sizeof}(c[1])$
- an abstract data
 - start address $c[2]$
 - size $\&c[2]$
 - $\text{sizeof}(c[2])$
- an abstract data
 - start address $c[3]$
 - size $\&c[3]$
 - $\text{sizeof}(c[3])$

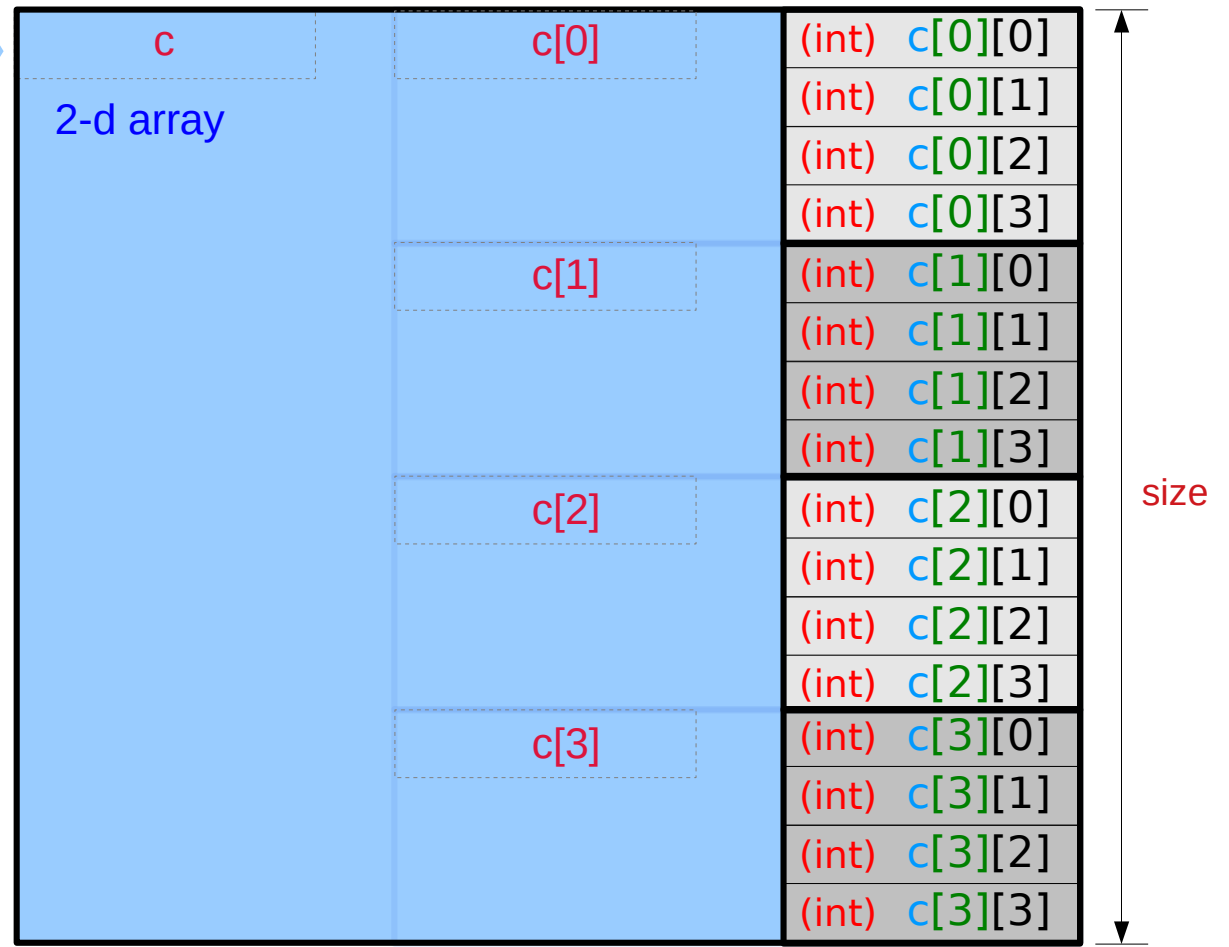
Abstract data **c**

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

An aggregate data **c**
- starting address **&c**
- size **sizeof(c)**

&c[0][0]

&c →
starting
address



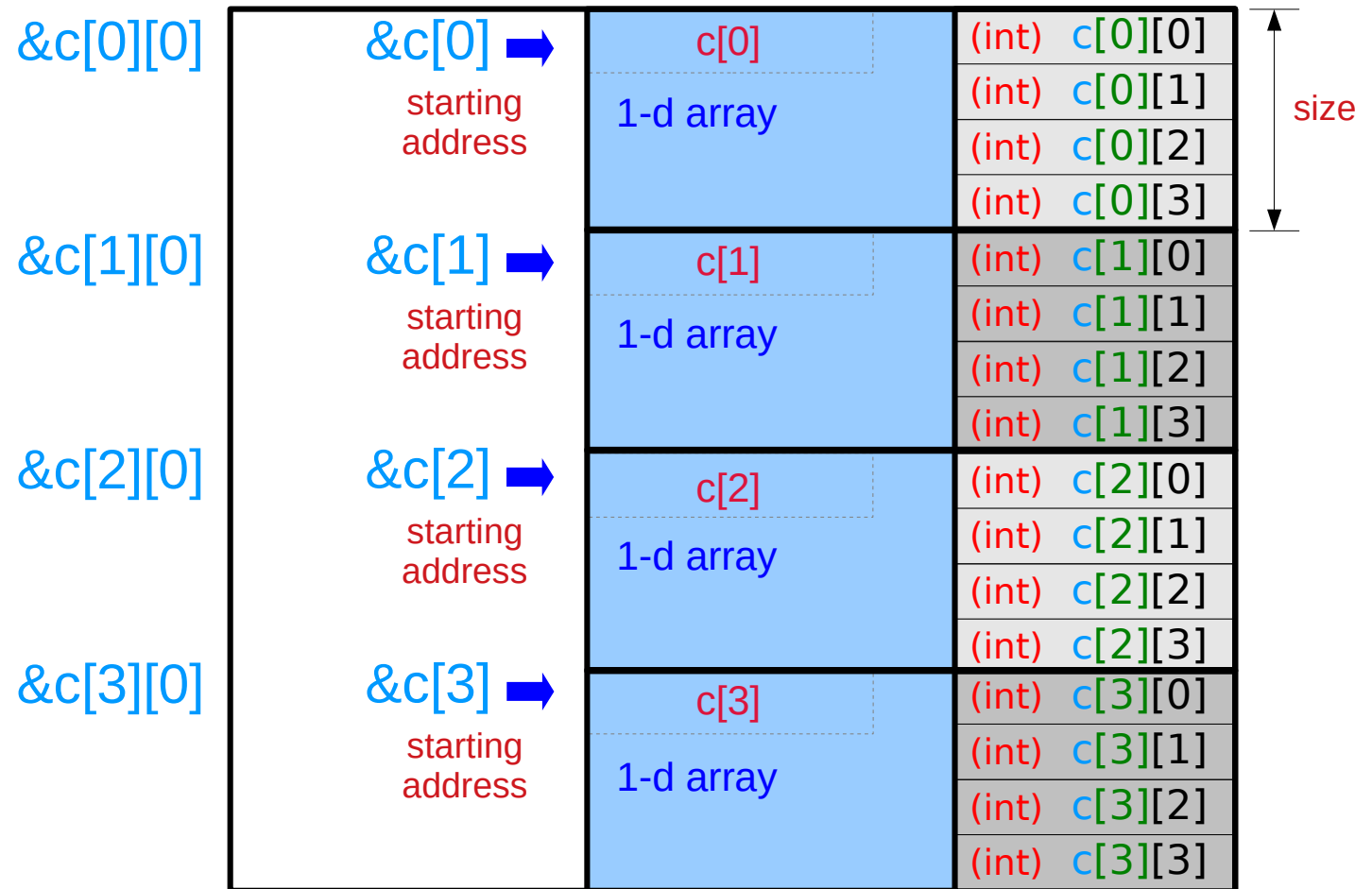
c and **c[0]**

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

Abstract data `c[i]`

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

An aggregate data `c[i]`
- starting address `&c[i]`
- size `sizeof(*c[i])`



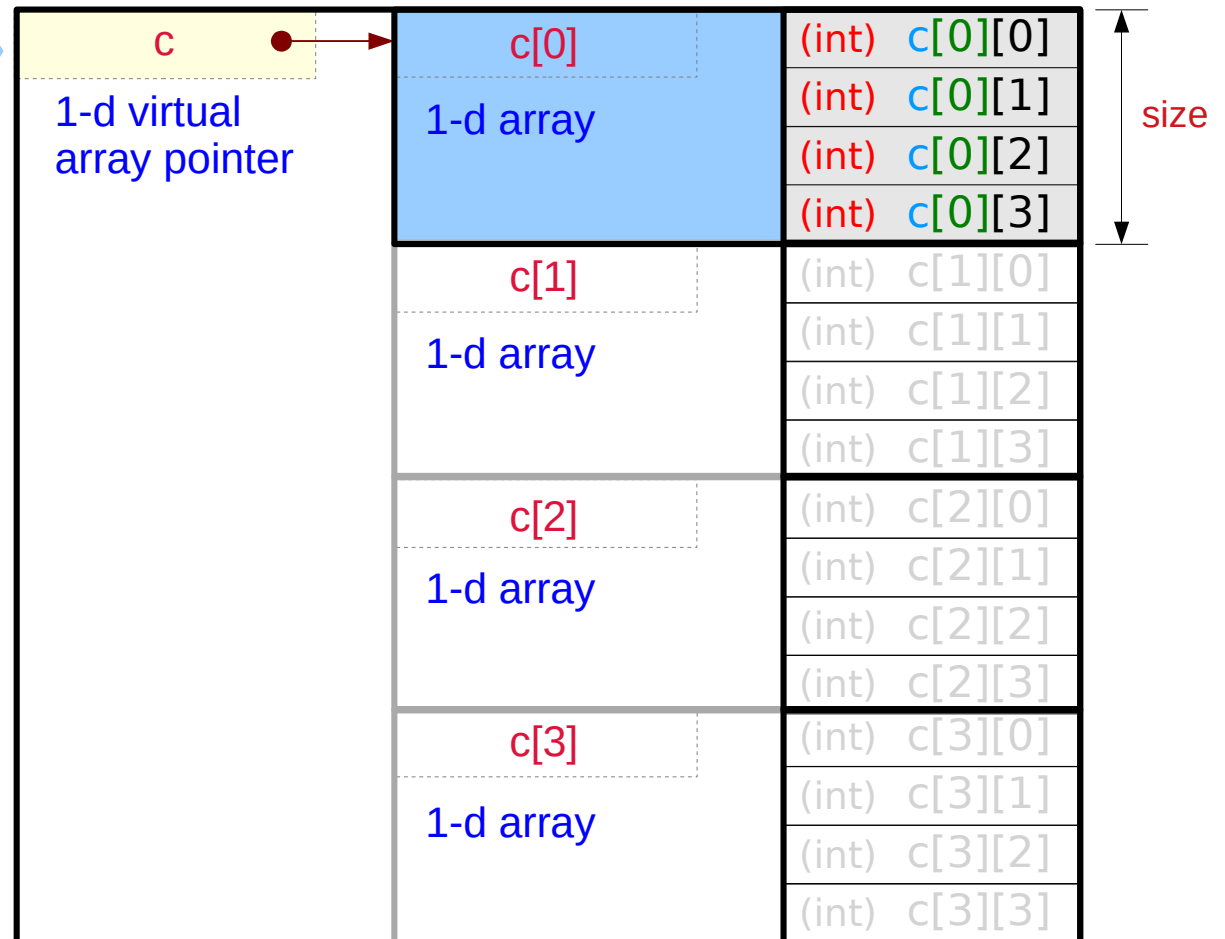
Virtual pointer **c**

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

A virtual pointer **c**
- pointer address $\&c$
- pointer value $\&c[0] = c$

$\&c[0][0]$

$\&c$ →
starting
address



virtual pointers

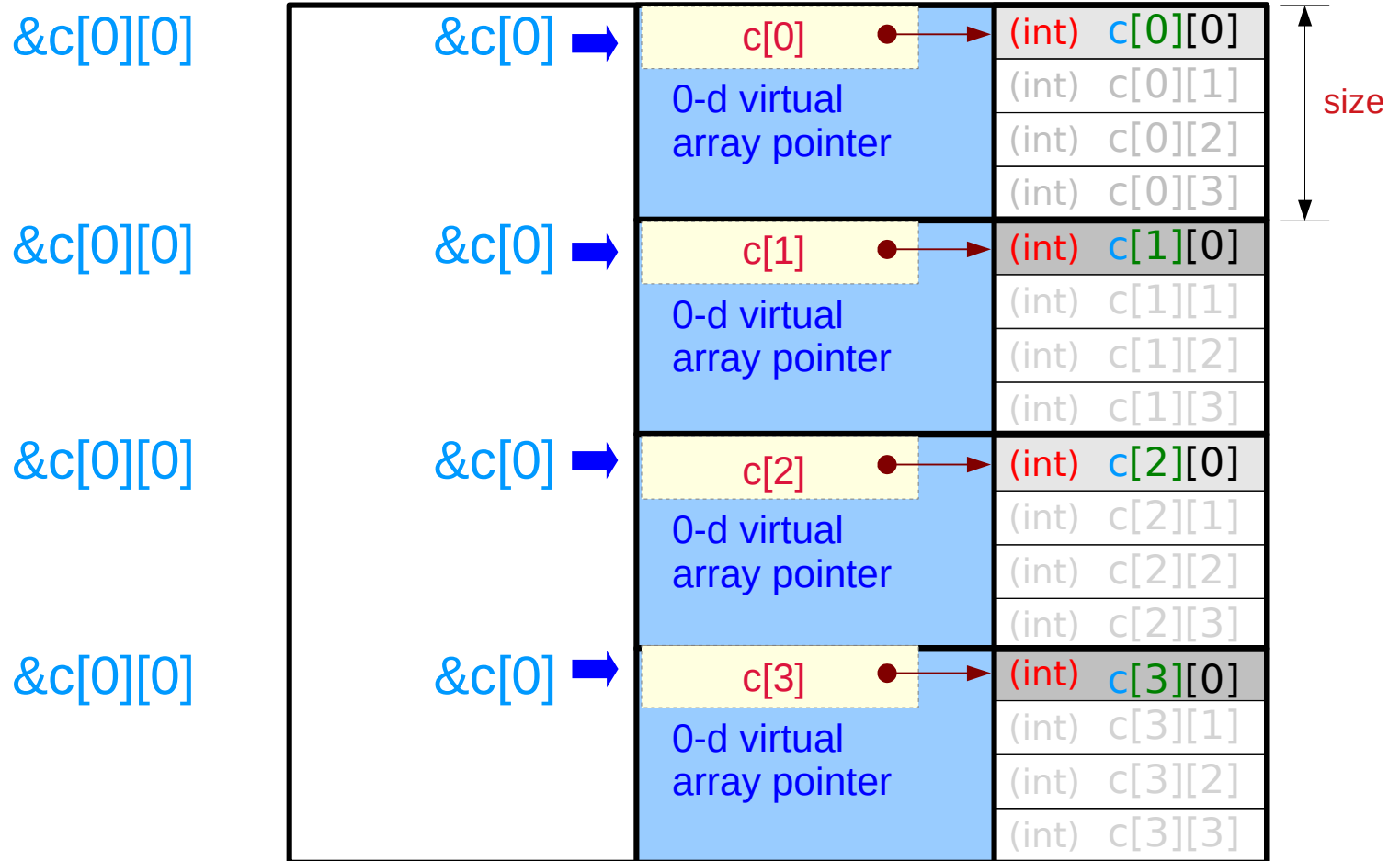
- no physical memory locations are allocated
- address and data have the same value

Virtual pointer $c[i]$

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

A virtual pointer $c[i]$

- pointer address $\&c[i]$
- pointer value $\&c[i][0]$



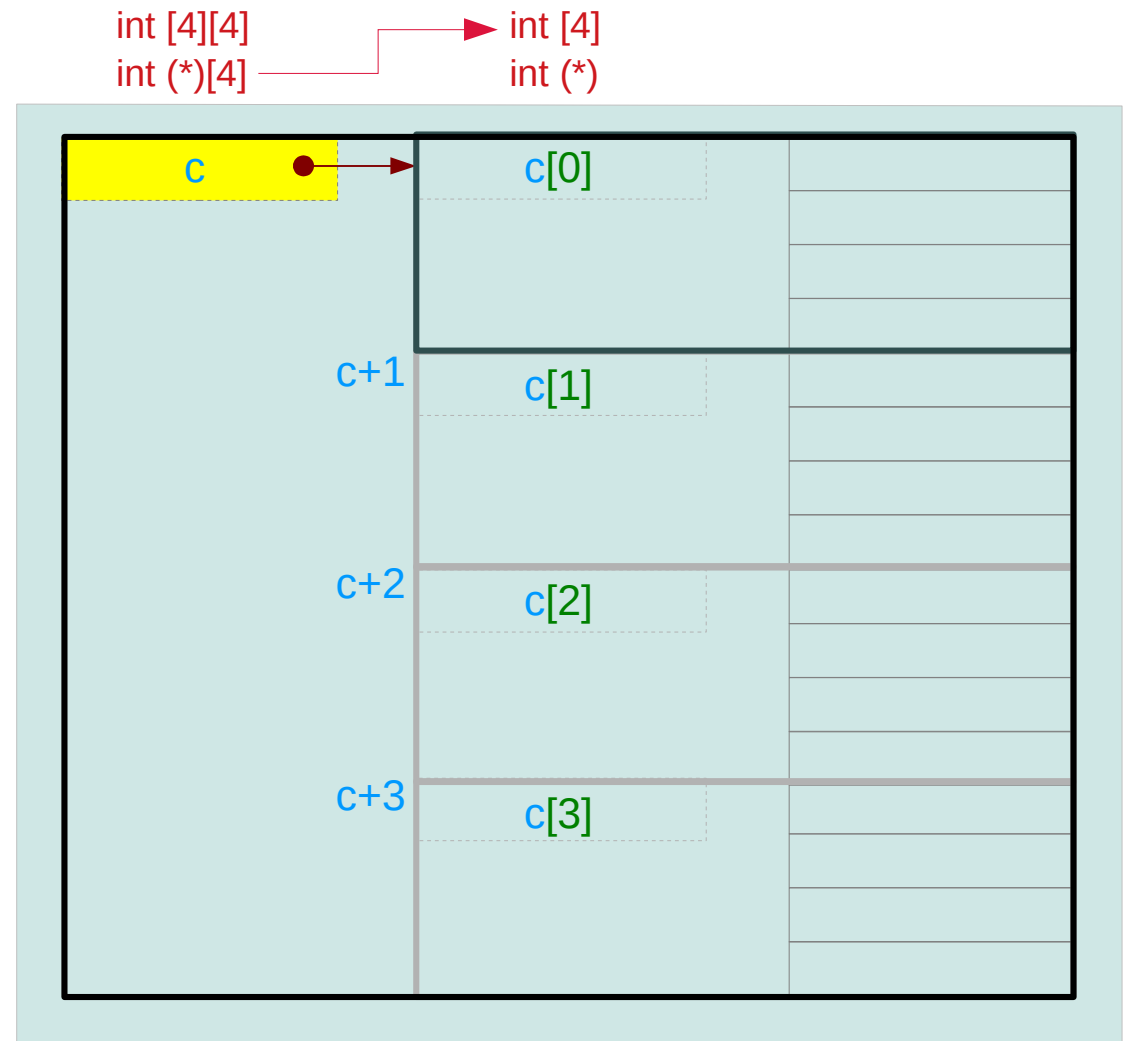
A 2-d array **c**

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

c : dual type

- aggregate data `int [4][4]`
a **2-d** array name
- virtual pointer `int (*)[4]`
an **1-d** array pointer
 - value `&c[0]`
starting address of
 - abstract data `c[0]`
a **1-d** array `int [4]`

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



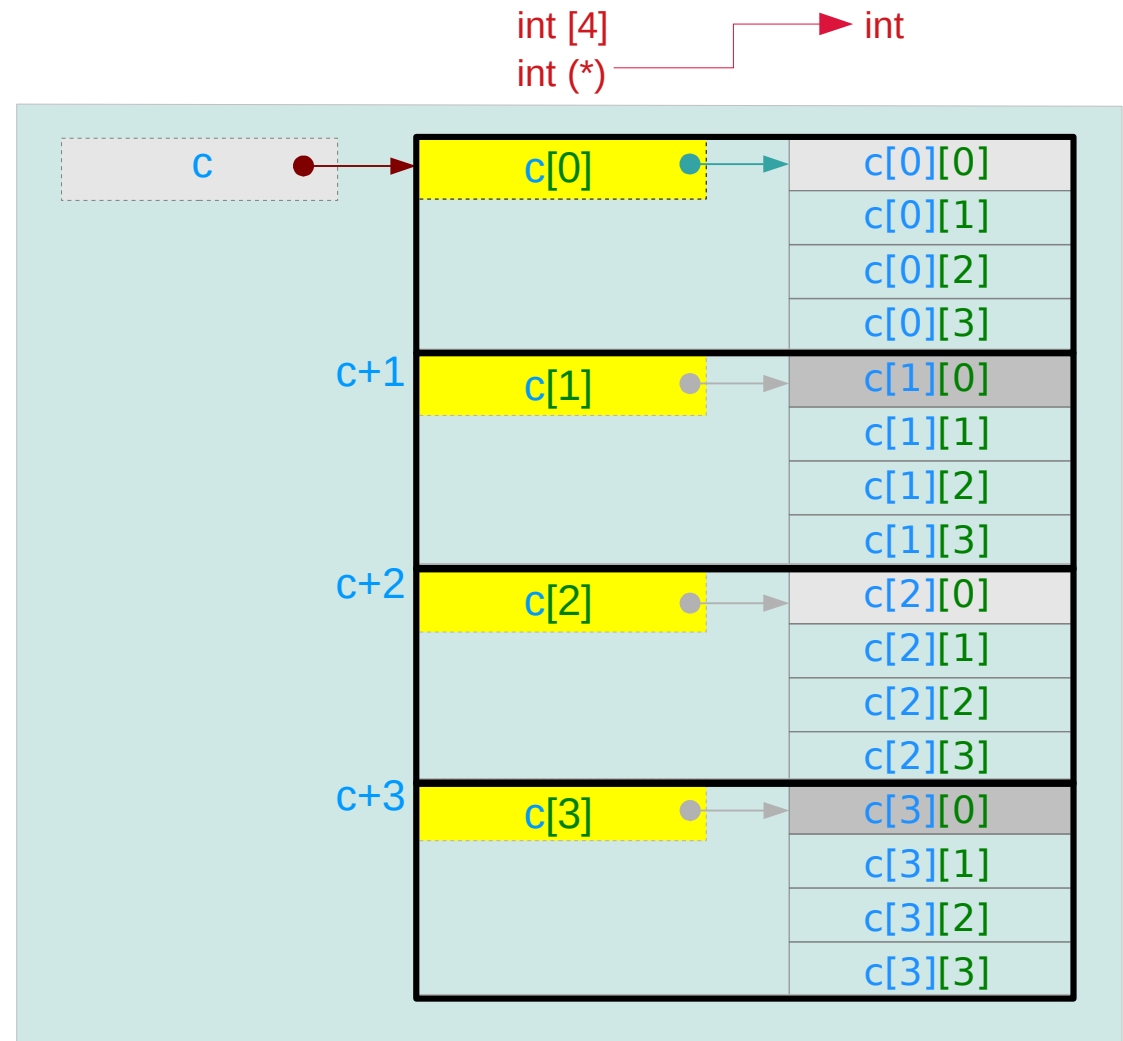
A 1-d subarray $c[i]$

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

$c[i]$: dual type

- aggregate data `int [4]`
a 1-d array name
- virtual pointer `int (*)`
a 0-d array pointer
 - value `&c[i][0]`
starting address
 - integer data `c[i][0]`
a 0-d array `int`

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



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

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

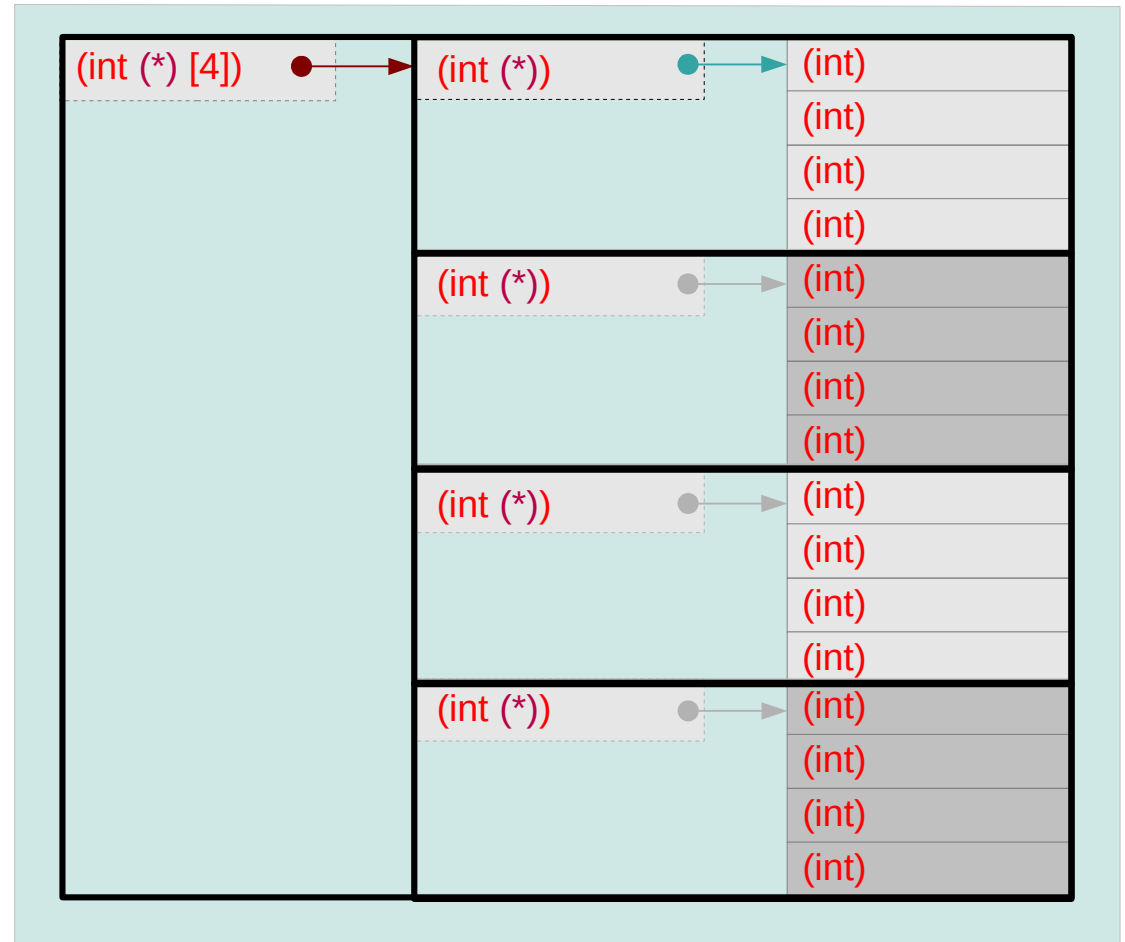
0-d array pointer `c[0]` `int *`

0-d array pointer `c[1]` `int *`

0-d array pointer `c[2]` `int *`

0-d array pointer `c[3]` `int *`

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



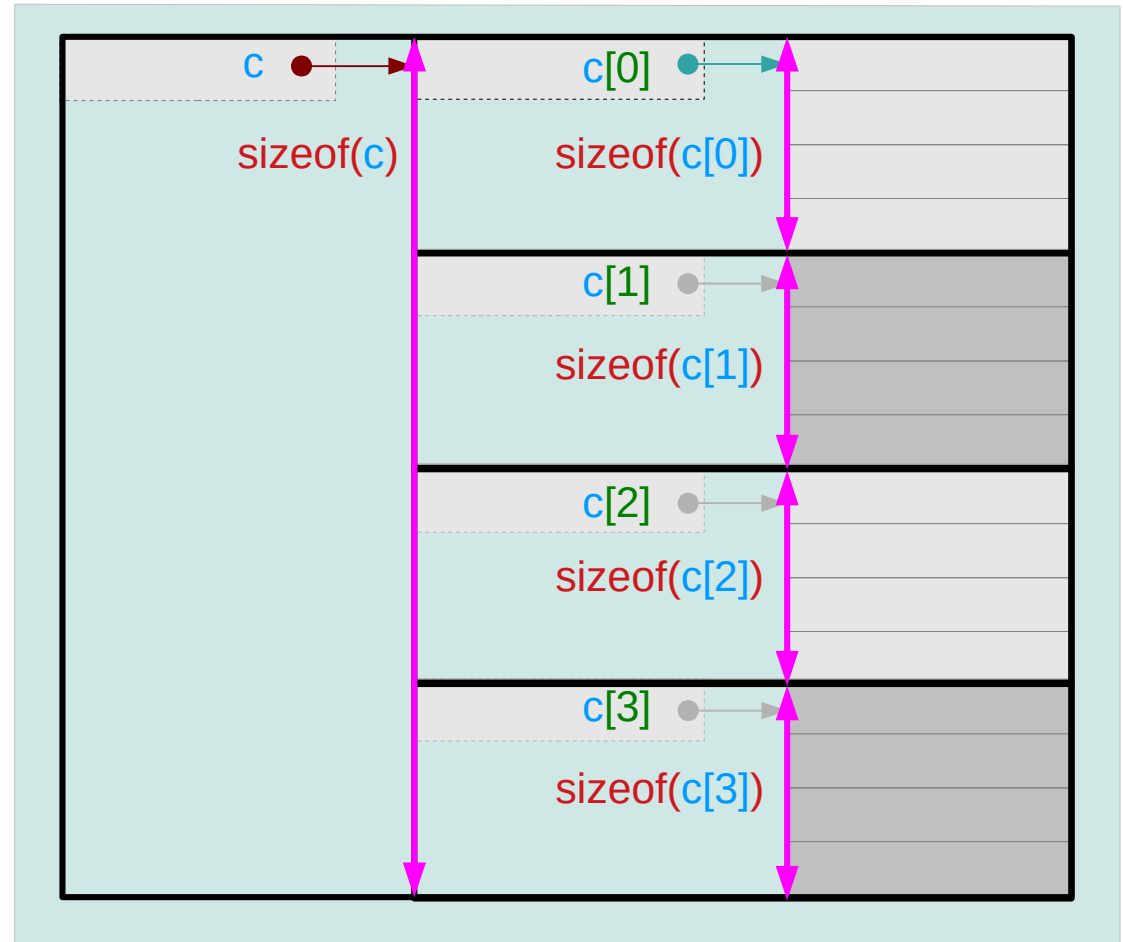
A 2-d array and its sub-arrays – type sizes

sizeof(c) = 4*4*4 bytes

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

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

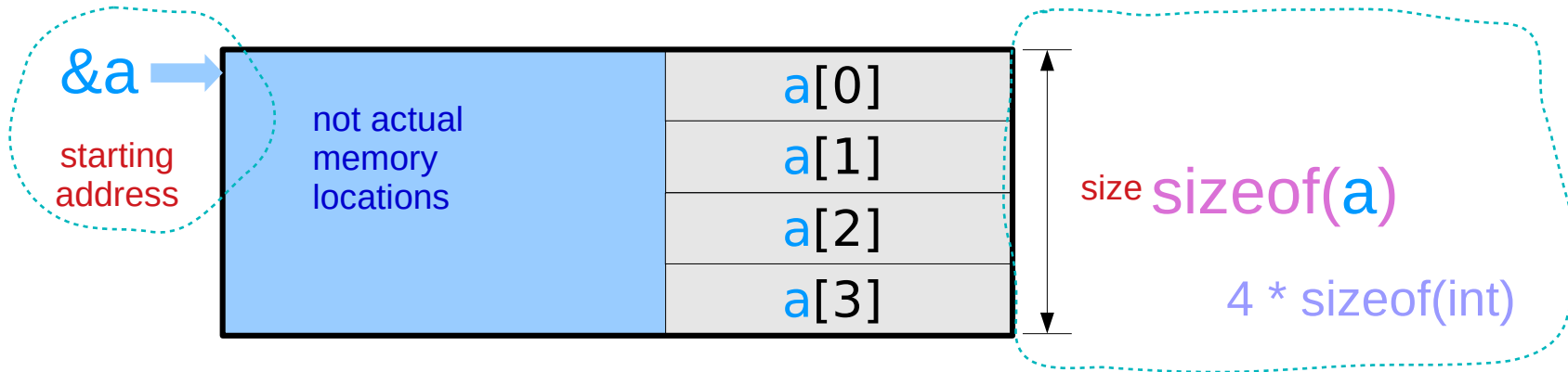
c : the **2-d** array name
c[i] : the **1-d** array name
c[i][j] : the **0-d** array name
(a scalar integer)



1-d array – an abstract type view

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

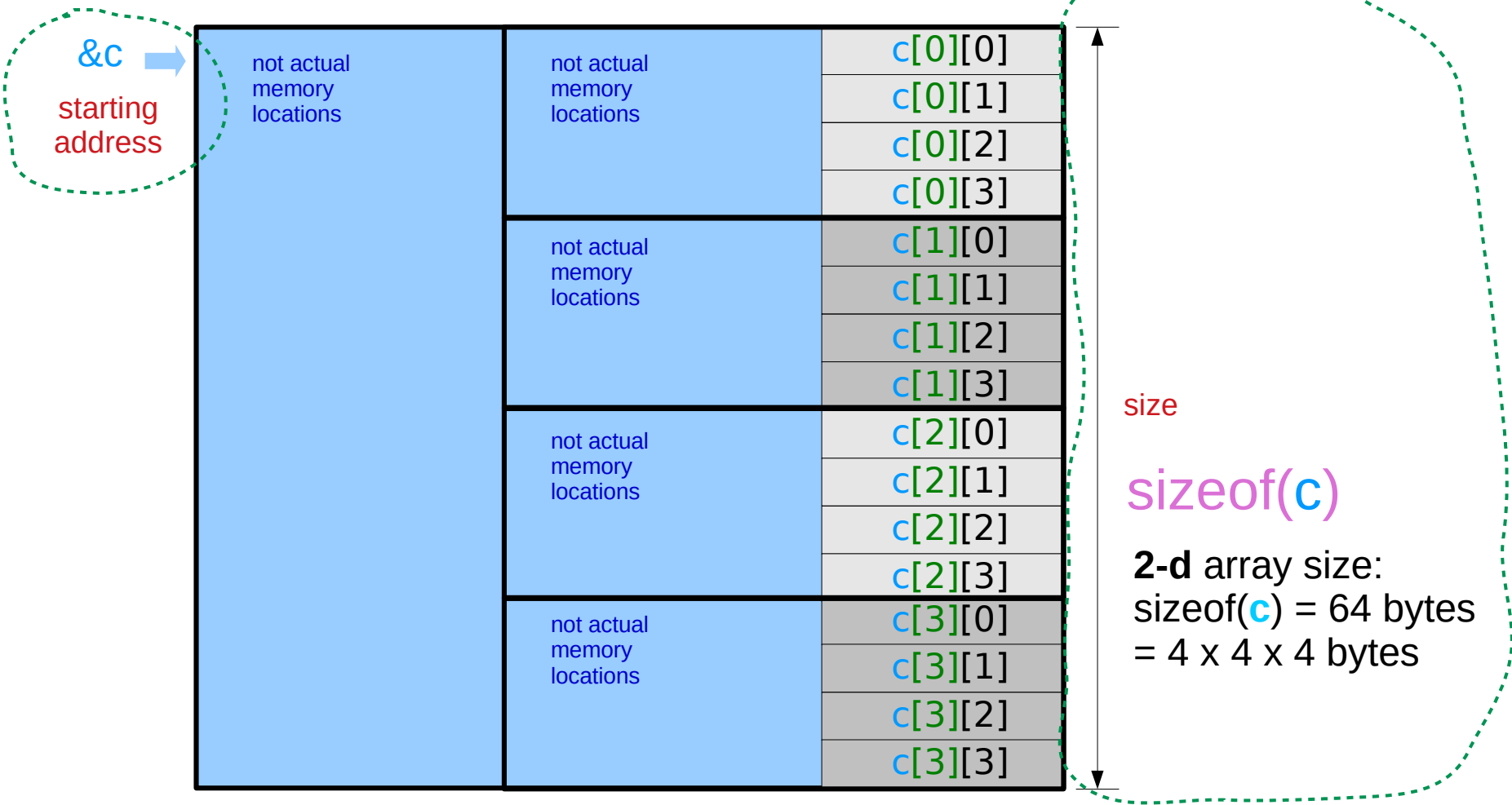
An aggregate type `a`
- starting address `&a`
- size `sizeof(a)`



2-d array – an abstract type view

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

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

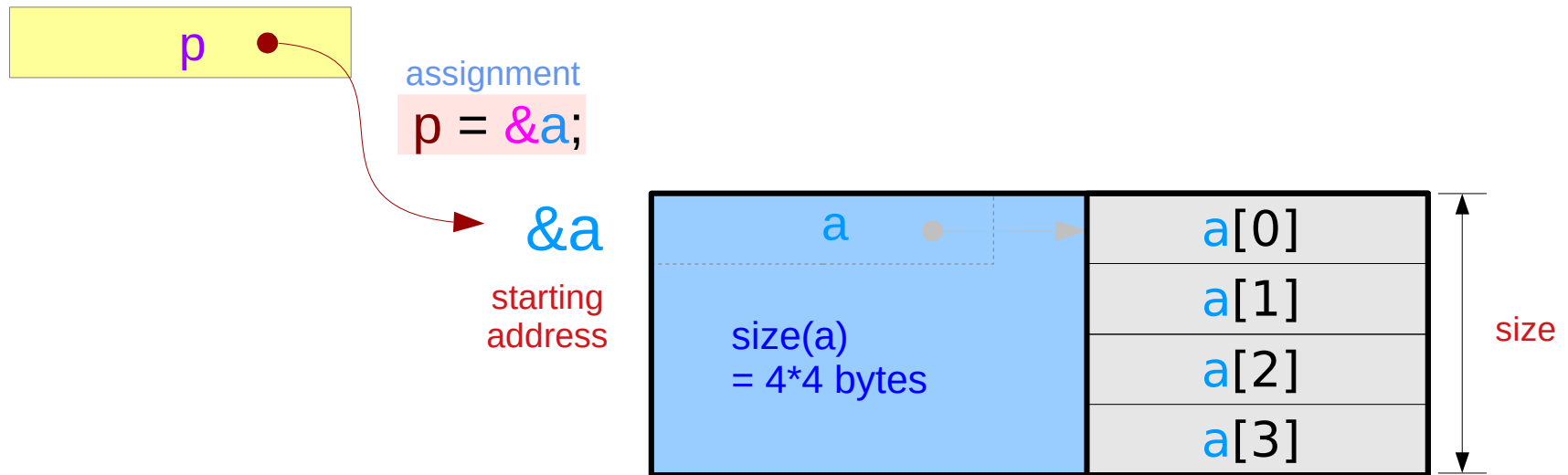


1-d array pointer to a 1-d array

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

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

1-d array pointer



`sizeof(p) = 4/8 bytes`
`sizeof(*p) = sizeof(a)`

2-d array pointer to a 2-d array

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

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

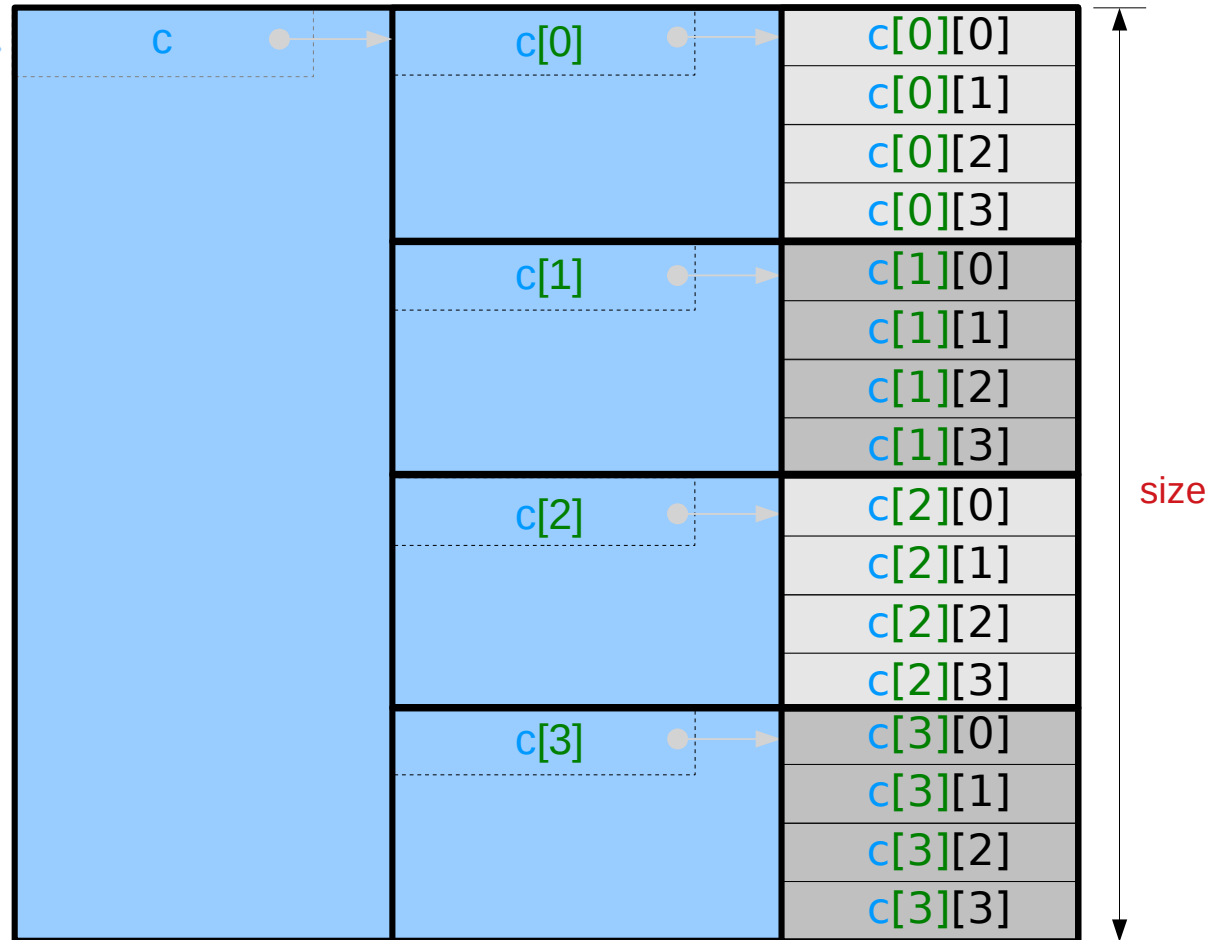
assignment

```
q = &c;
```

q

2-d array pointer

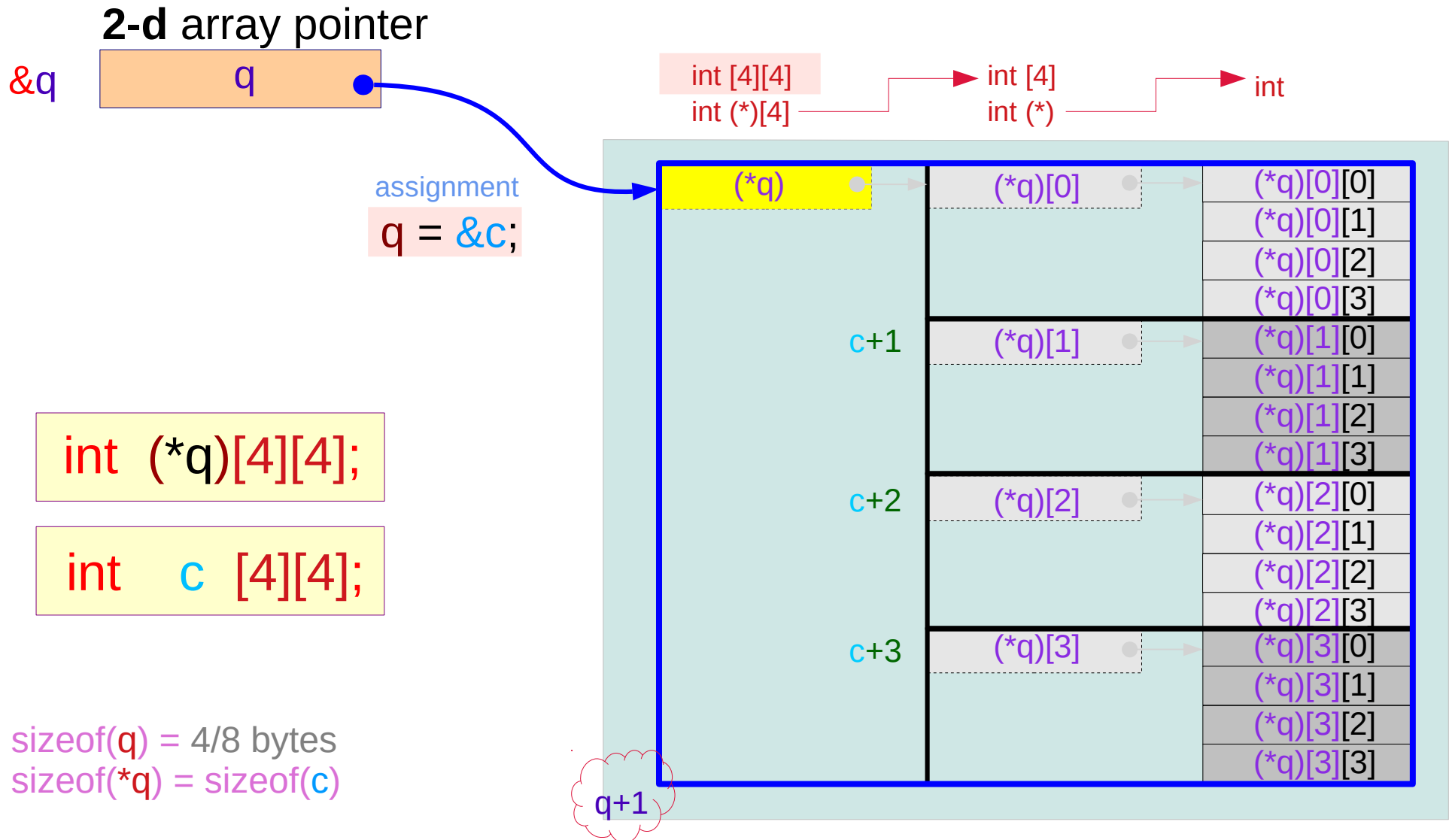
&c
starting address



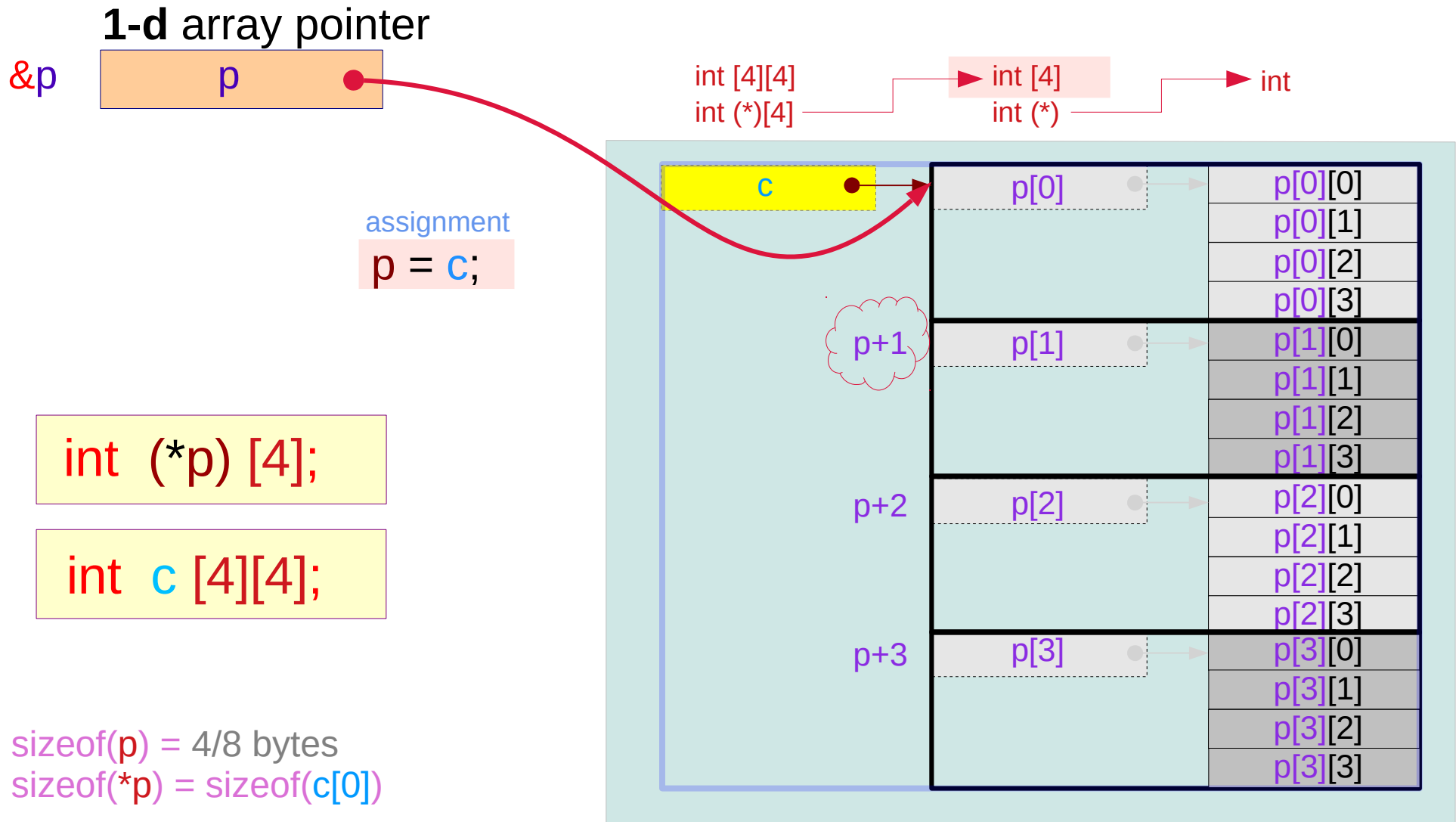
sizeof(q) = 4/8 bytes

sizeof(*q) = sizeof(c)

Accessing a 2-d array using a 2-d array pointer

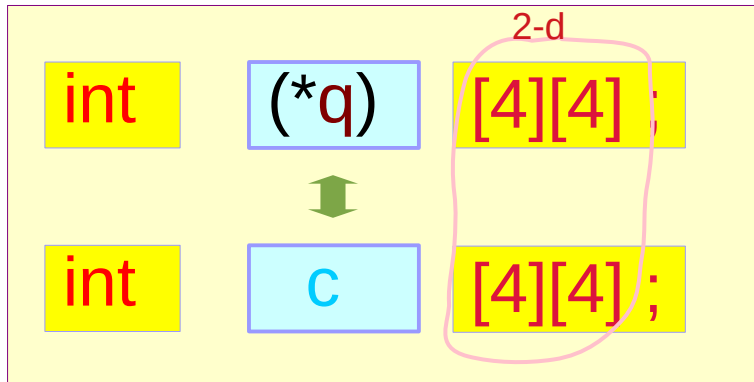


Accessing a 2-d array using a 1-d array pointer



2-d and 1-d array pointers

2-d array pointer q



`*q ≡ c;`

correspondence

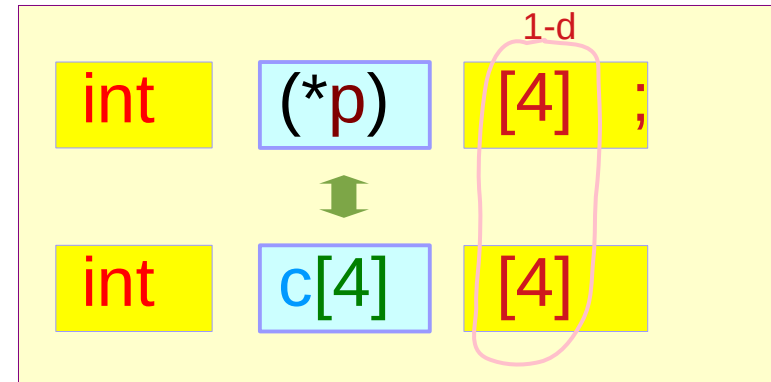
`q = &c;`

assignment

`(*q)[i][j] ≡ c[i][j]`

substitution

1-d array pointer p



`*p ≡ *c;`

correspondence

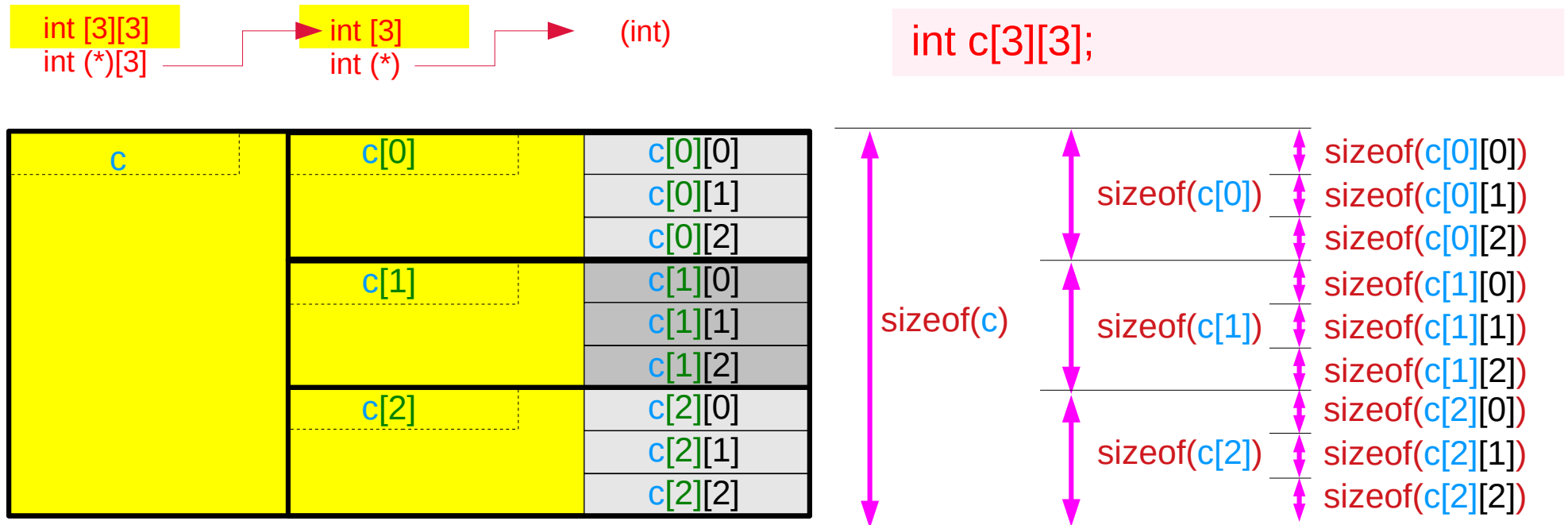
`p = c;`

assignment

`p[i][j] ≡ c[i][j]`

substitution

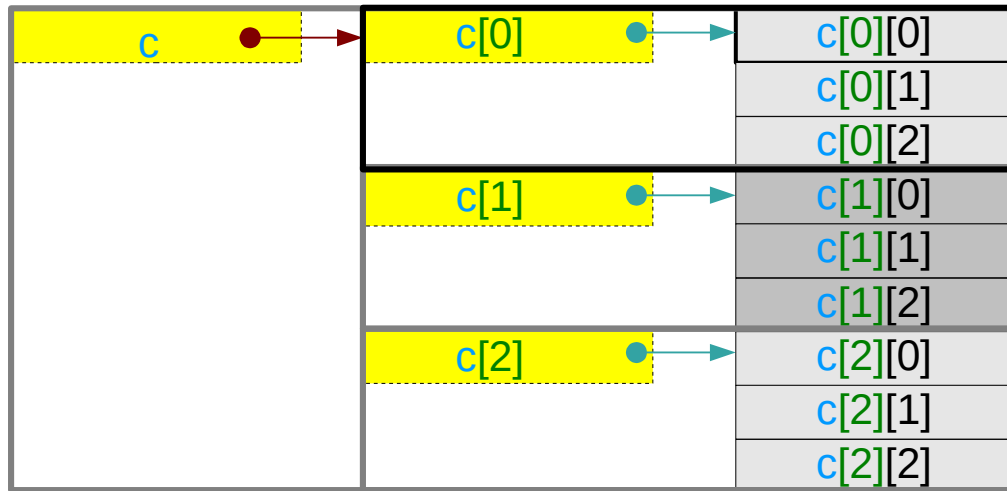
A 2-d array and its 1-d sub-arrays – a size view



A 2-d array and its 1-d sub-arrays – a virtual pointer view



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

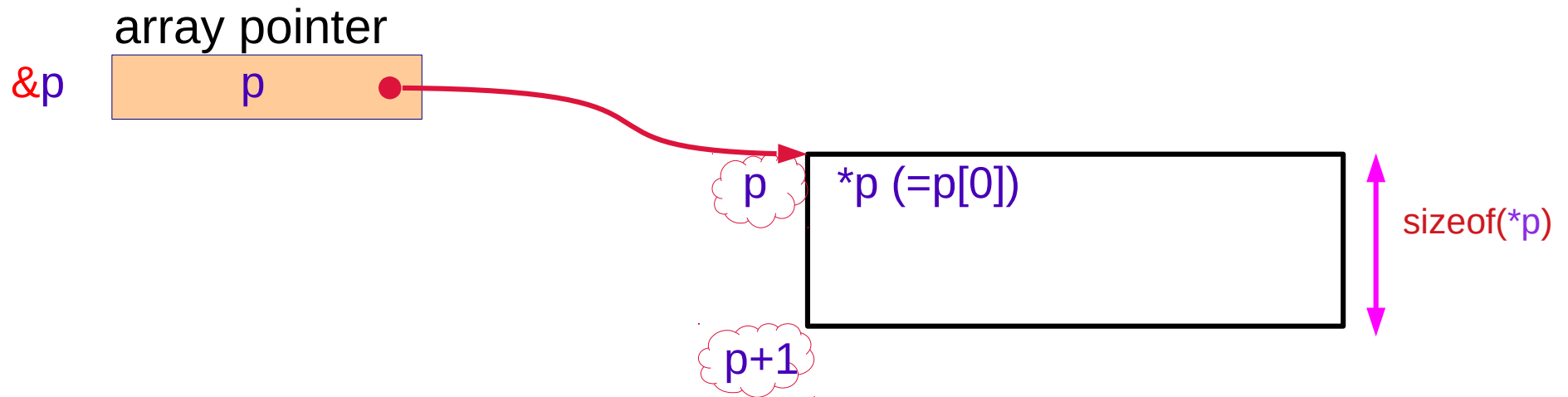


value(c) = value(c[0]) = value(&c[0][0])
 = value(c[1]) = value(&c[1][0])
 = value(c[2]) = value(&c[2][0])

value(&c) = value(&c[0]) = value(&c[0][0])
 = value(&c[1]) = value(&c[1][0])
 = value(&c[2]) = value(&c[2][0])

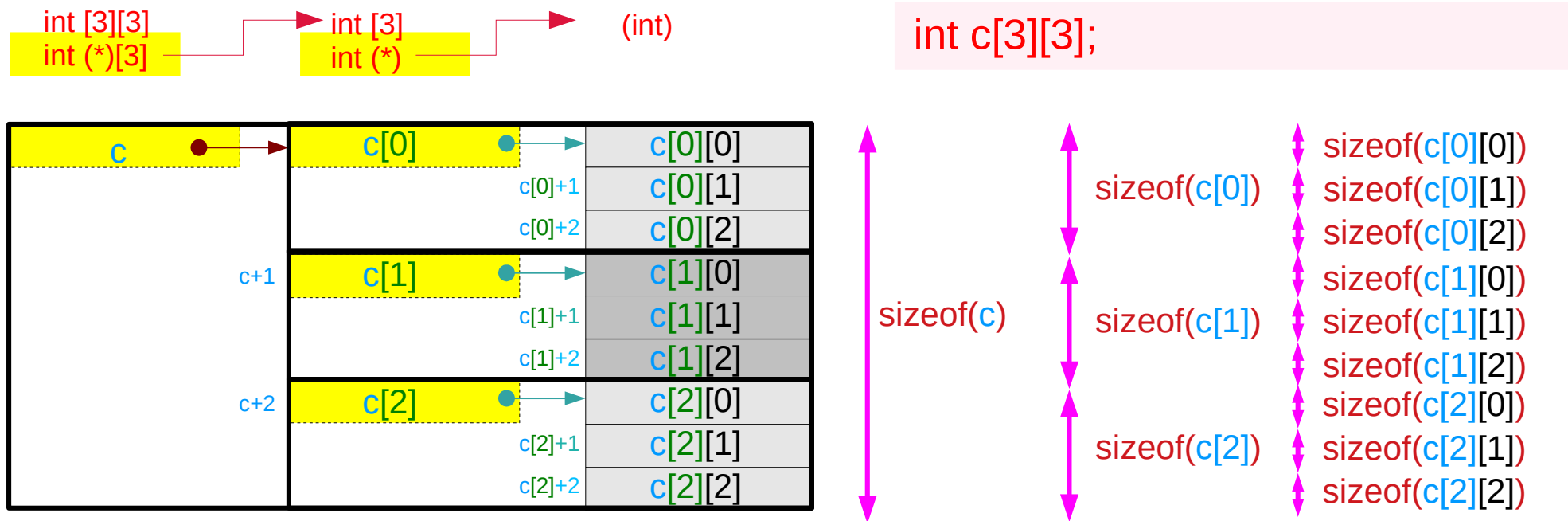
address(c) = address(c[0]) = address(c[0][0])
 = address(c[1]) = address(c[1][0])
 = address(c[2]) = address(c[2][0])

Array pointer **p** and **p+1** relationship



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

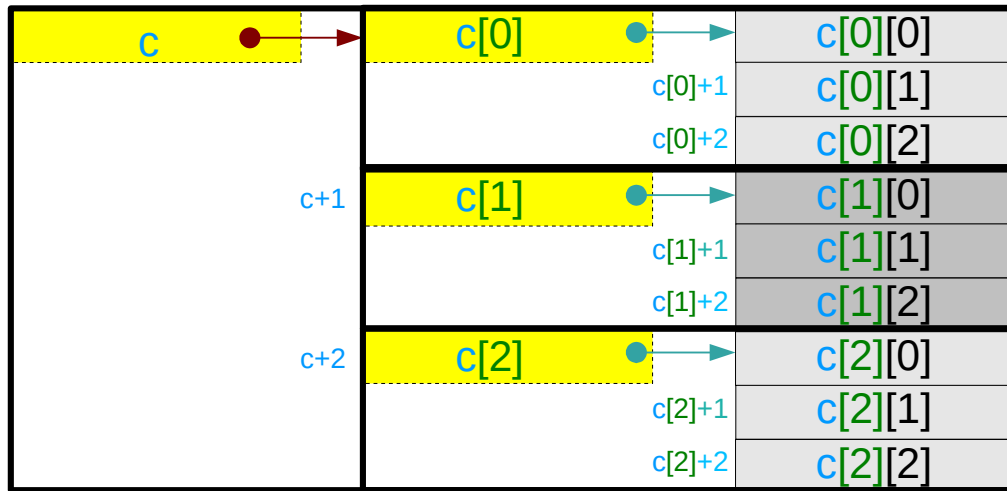
Virtual array pointer **c** and **c+1** relationship



A 2-d array and its 1-d sub-arrays – value relation



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



`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`

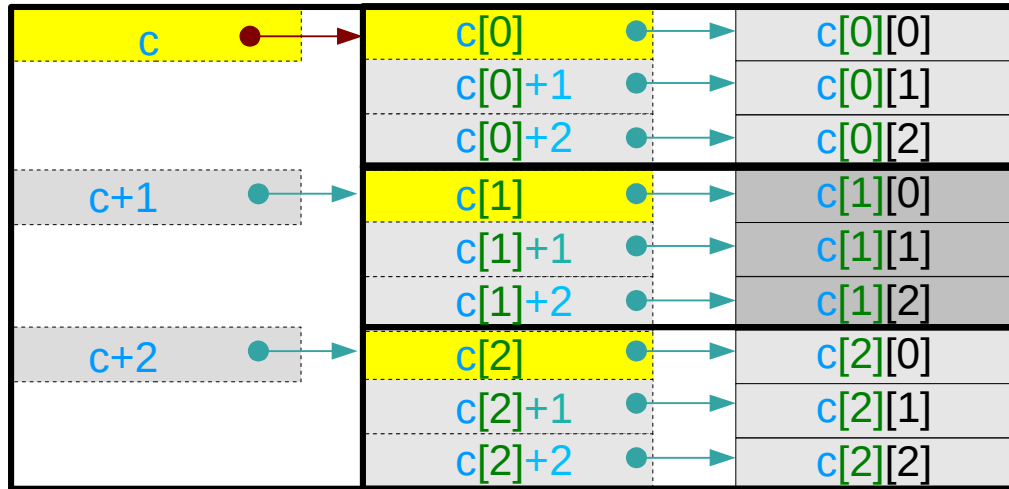
`value(c[2]) = value(c[2][0]) ... leading element`
`value(c[2]+1) = value(c[2][0]) + sizeof(c[2][0]) * 1`
`value(c[2]+2) = value(c[2][0]) + sizeof(c[2][0]) * 2`

`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`

A 2-d array and its 1-d sub-arrays – size relation



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



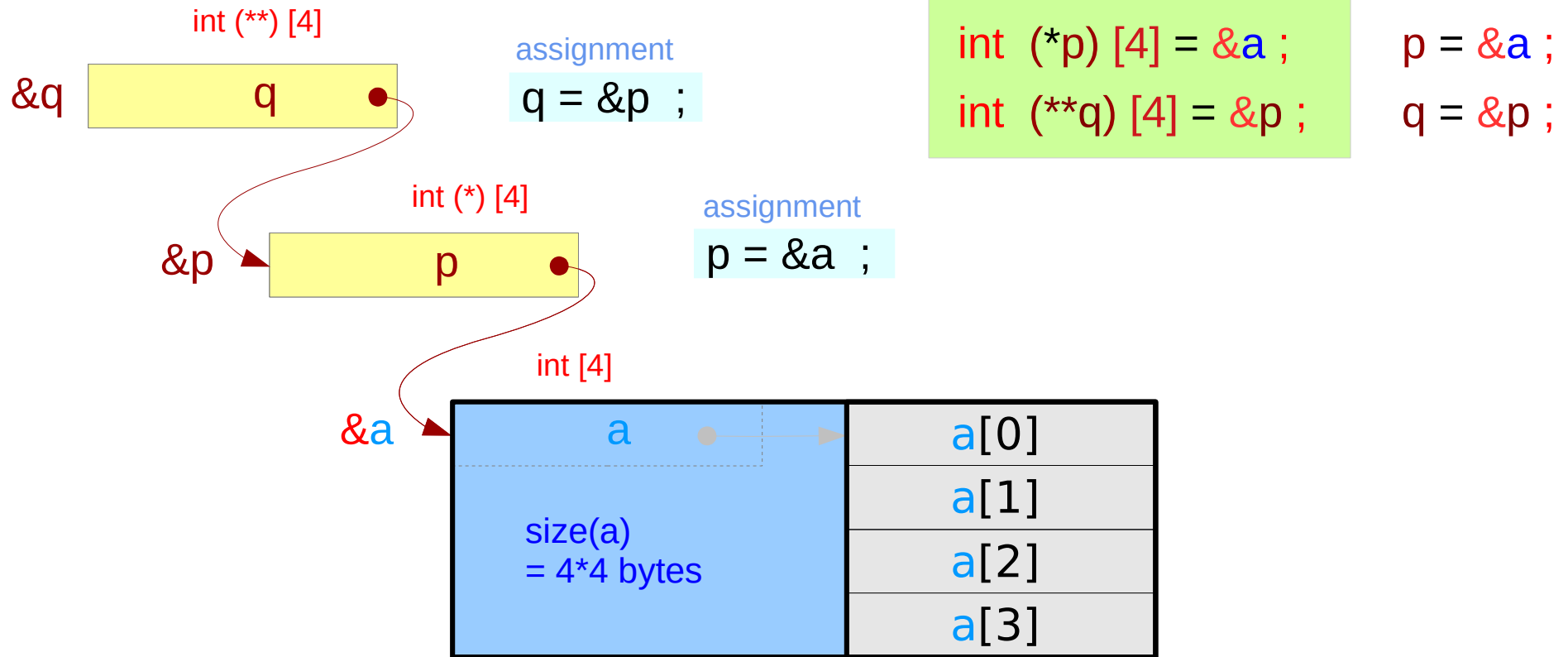
`sizeof(c[0]) = sizeof(c[0][0]) * 3 ... leading element`
`sizeof(c[0]+1) = pointer size (4/8 bytes)`
`sizeof(c[0]+2) = pointer size (4/8 bytes)`

`sizeof(c[1]) = sizeof(c[1][0]) * 3 ... leading element`
`sizeof(c[1]+1) = pointer size (4/8 bytes)`
`sizeof(c[1]+2) = pointer size (4/8 bytes)`

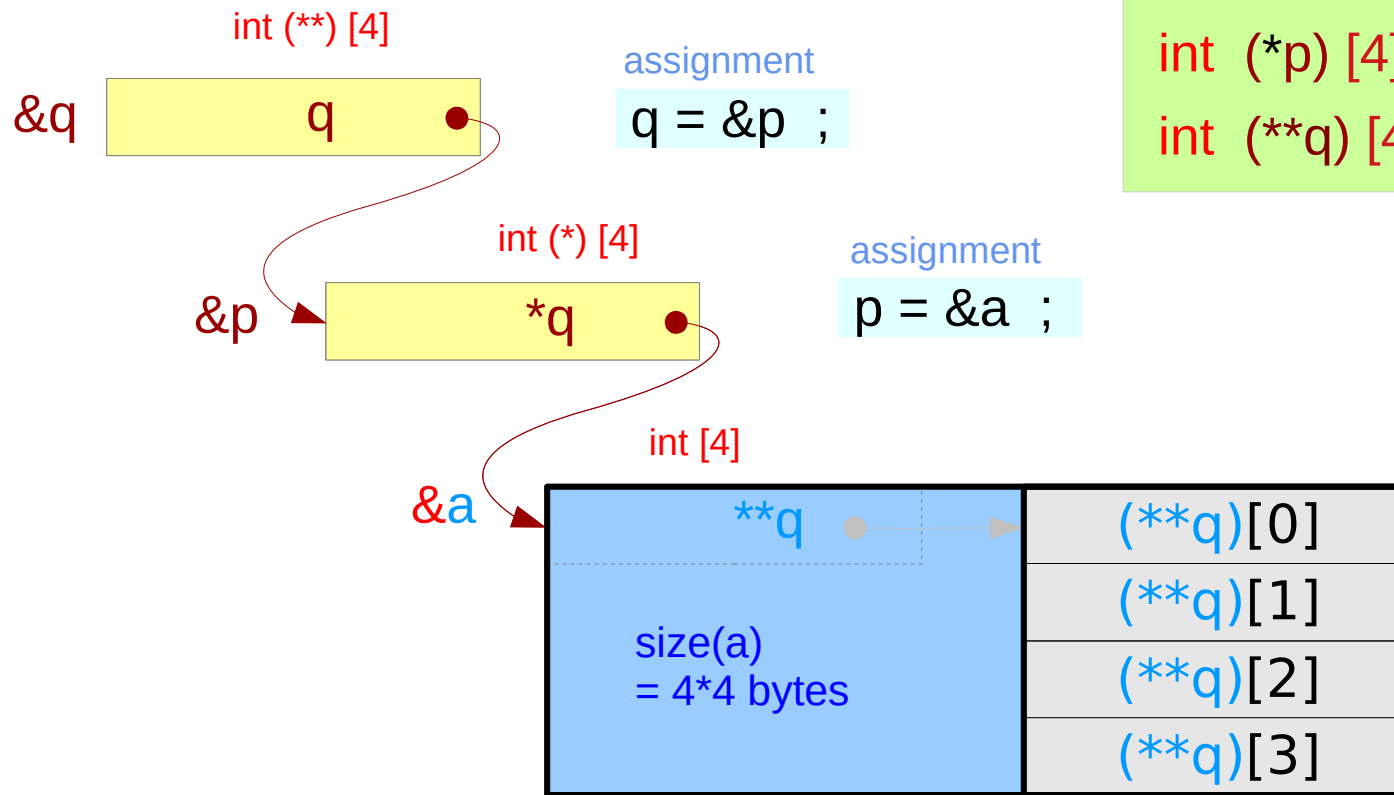
`sizeof(c[2]) = sizeof(c[2][0]) * 3 ... leading element`
`sizeof(c[2]+1) = pointer size (4/8 bytes)`
`sizeof(c[2]+2) = pointer size (4/8 bytes)`

`sizeof(c) = sizeof(c[0]) * 3 ... leading element`
`sizeof(c+1) = pointer size (4/8 bytes)`
`sizeof(c+2) = pointer size (4/8 bytes)`

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



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



assignment
`q = &p ;`

assignment
`p = &a ;`

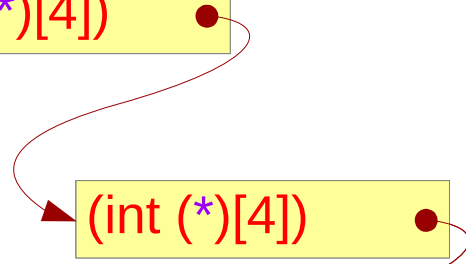
```
int a [4] ;  
int (*p) [4] = &a ;  
int (**q) [4] = &p ;
```

`p = &a ;`
`q = &p ;`

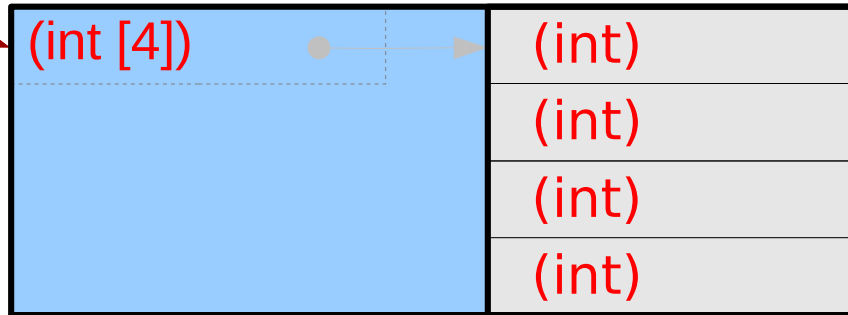
Double pointer to a 1-d array – a type view

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

(int (**)[4])



1-d array pointer

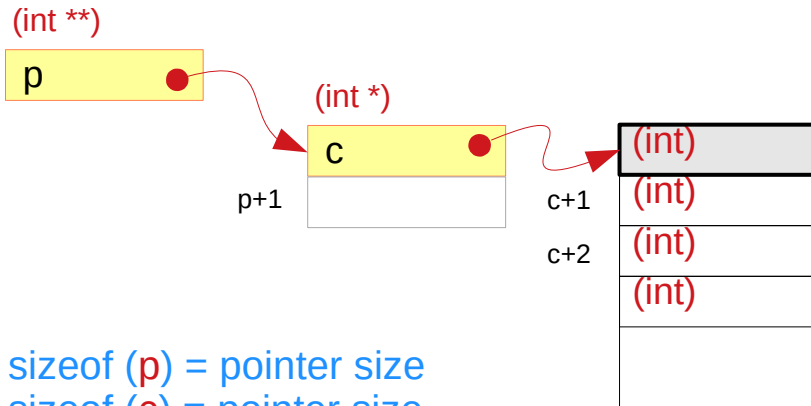


```
int a [4] ;  
int (*p) [4] = &a ;  
int (**q) [4] = &p ;
```

p = &a ;
q = &p ;

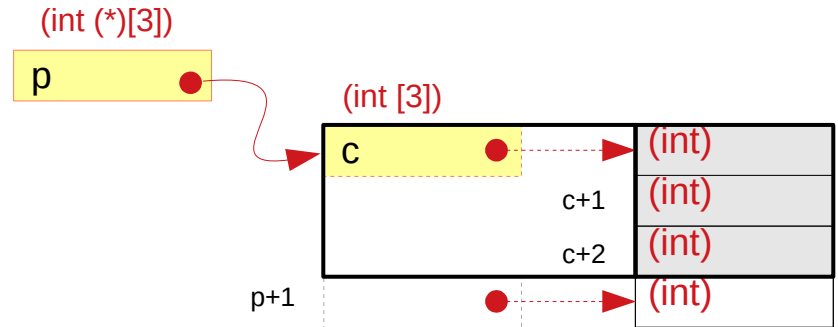
Integer pointer and array types – `int **`, `int (*)[3]`, `int[2][3]`

`int **p;` `int *c;` $v(\&c) \neq v(c)$



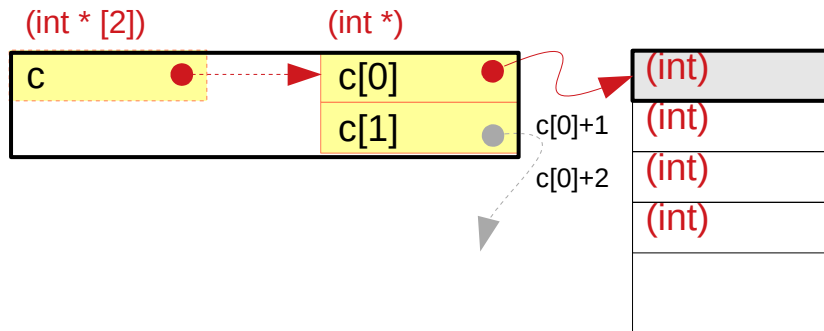
`sizeof (p)` = pointer size
`sizeof (c)` = pointer size

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



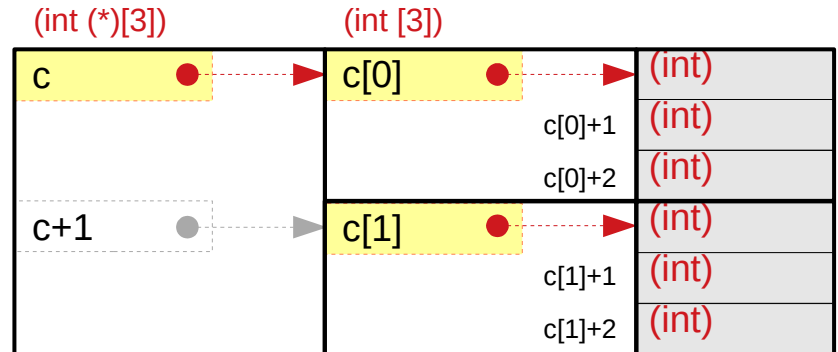
`sizeof (p)` = pointer size
`sizeof (c)` = `sizeof(int) * 3`

`int* c[2];` $v(\&c[0]) \neq v(c[0])$



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

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



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

Integer pointer types

```
#include <stdio.h>
```

```
void func(int d[ ])
```

```
{  
}  
}
```

```
int main(void) {
```

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

```
    int *b;
```

```
    int **c;
```

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

```
    func(a);
```

```
}
```

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

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

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

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

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

C scalar data types

Arithmetic types and **pointer** types

- collectively called **scalar types**
- hold single data item
- size and format

Array and **structure** / **union** types

- collectively called **aggregate types**
- hold more than one data items

C scalar data types provide their size and format
The alignment of a scalar data types is equal to its size

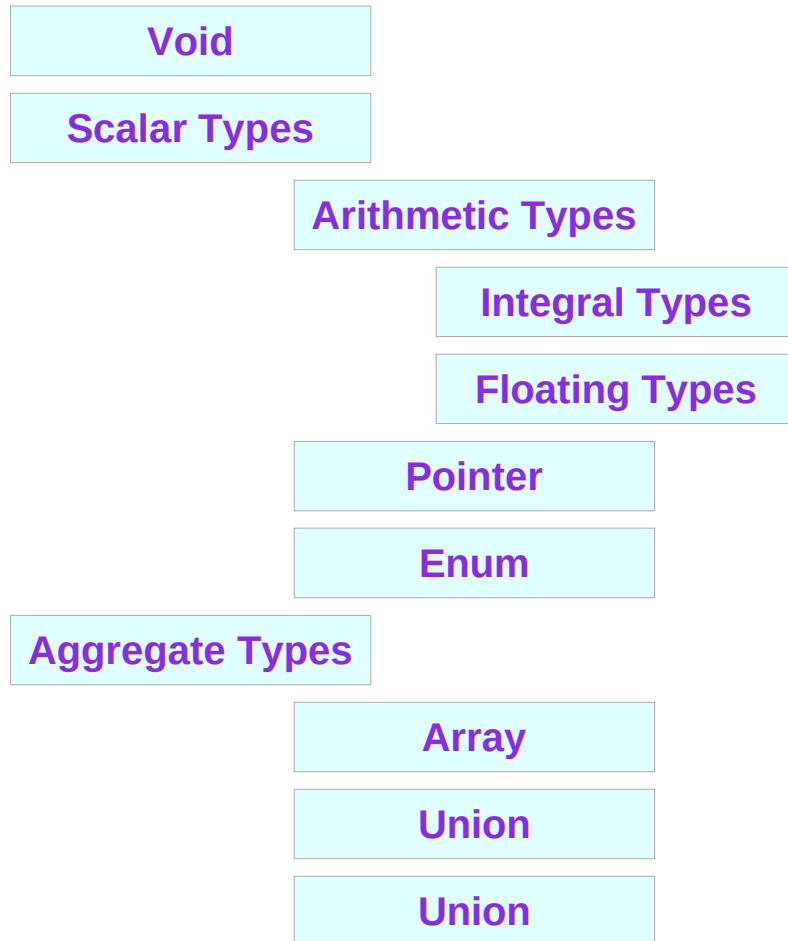
Scalar alignment shows scalar alignments that apply to individual scalars and to scalars that are elements of an array or members of a structure or union.

Wide characters are supported (character constants prefixed with an L)

The size of each wide character is 4 bytes

<https://stackoverflow.com/questions/35722514>

Data Types



<https://stackoverflow.com/questions/35722514>

C abstract data types

Array as an Abstract Data Type and as a Data Structure

Abstract Data Types, ADTs are a way of classifying data structures based on how they are used and the behaviors they provide.

They do not specify how the data structure must be implemented but simply provide a minimal expected interface and set of behaviors.

Data structure is a concrete implementation of a data type.

It's possible to analyze the time and memory complexity of a Data Structure but not from a data type.

The Data Structure can be implemented in several ways and its implementation may vary from language to language

[Lucasmagnum.edium.com/sidenotes-array-sbstract-data-type-data-structure...](https://lucasmagnum.edium.com/sidenotes-array-sbstract-data-type-data-structure...)

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