

# Applications of Array Pointers (1A)

---

Copyright (c) 2010 - 2018 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](mailto:youngwlim@hotmail.com).  
This document was produced by using LibreOffice.

---

# Pointer to Multi-dimensional Arrays

# Integer pointer types

`(int **)`

a pointer to a **integer pointer**  
size = 8 bytes

`(int *)`

a pointer to an **int**  
size = 8 bytes

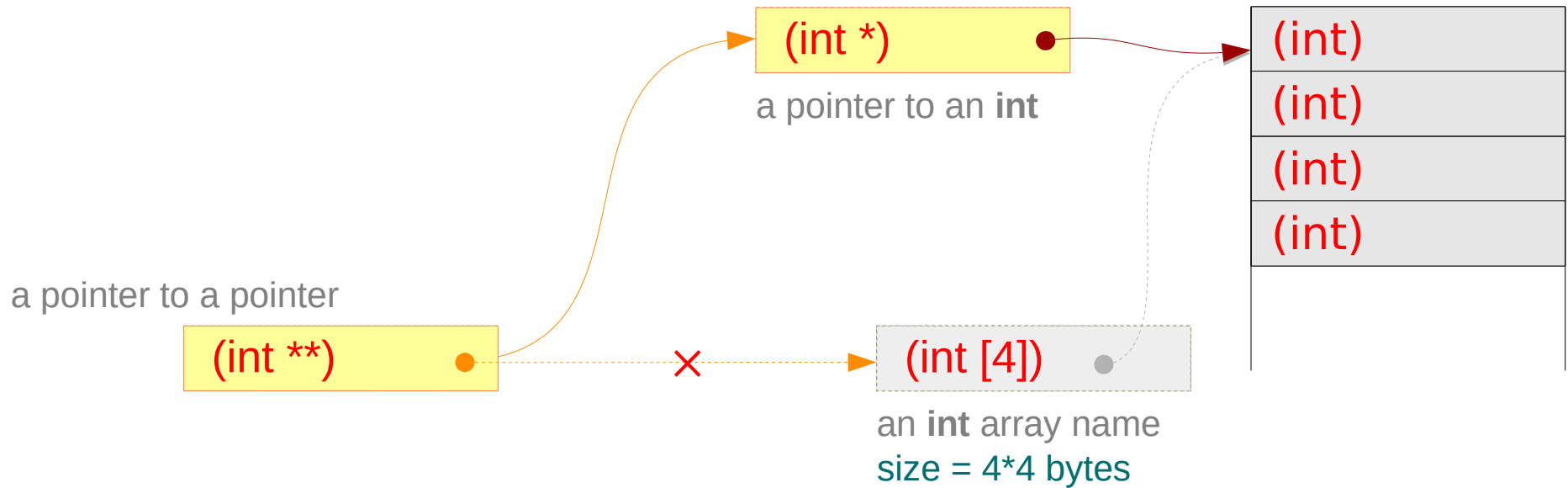
`(int (*)[4])`

a pointer to a **1-d array**  
size = 8 bytes

`(int [4])`

an **int array name**  
size = 4\*4 bytes

# Integer pointer type : (int \*\*)

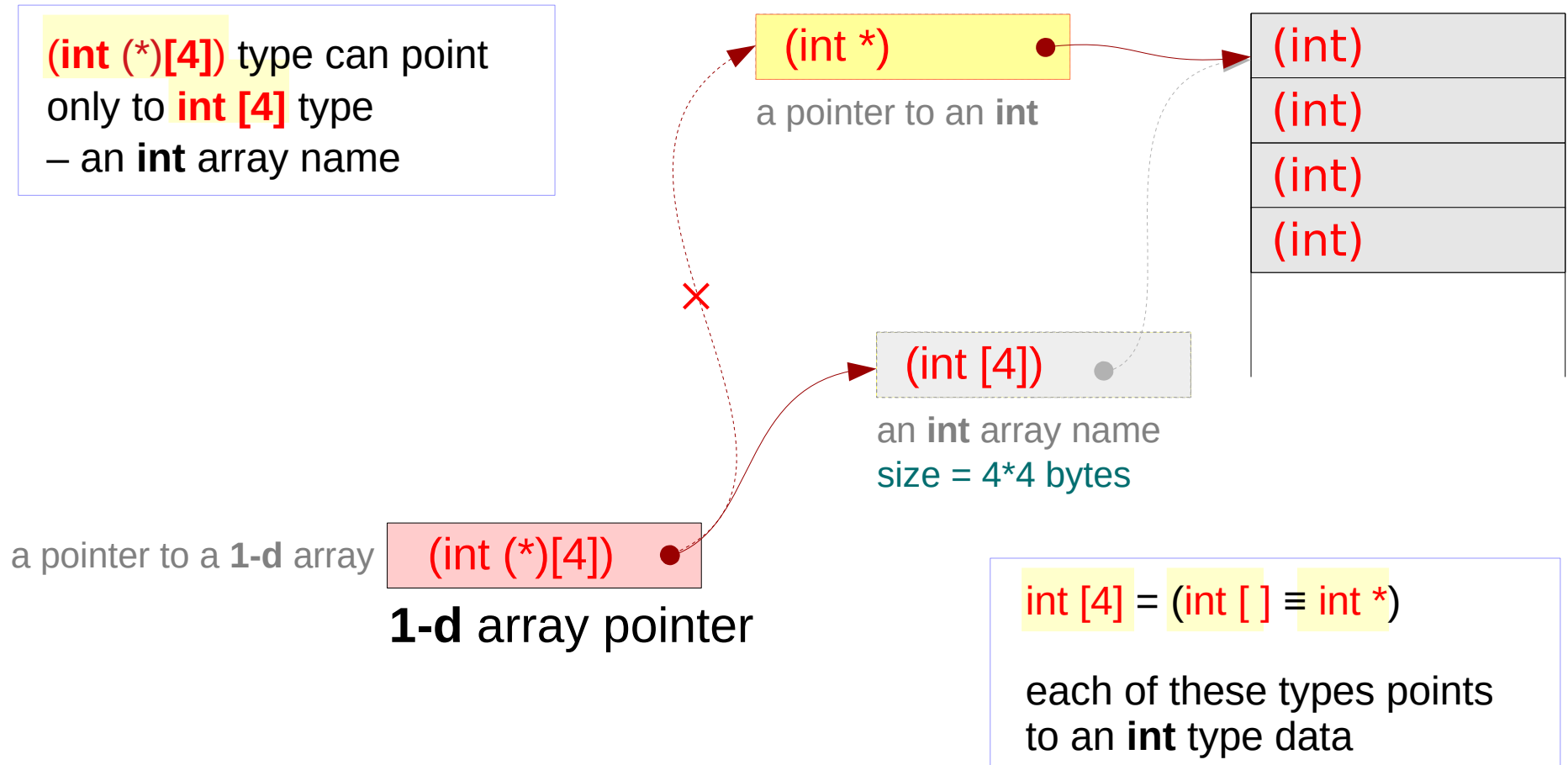


(int \*\*) type can point  
only to (int \*) type  
– an int array name

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

each of these types points  
to an int type data

# Integer pointer type : (int (\*)(4))



# 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
```

# Series of **1-d** arrays

---

series of 1-d array pointers

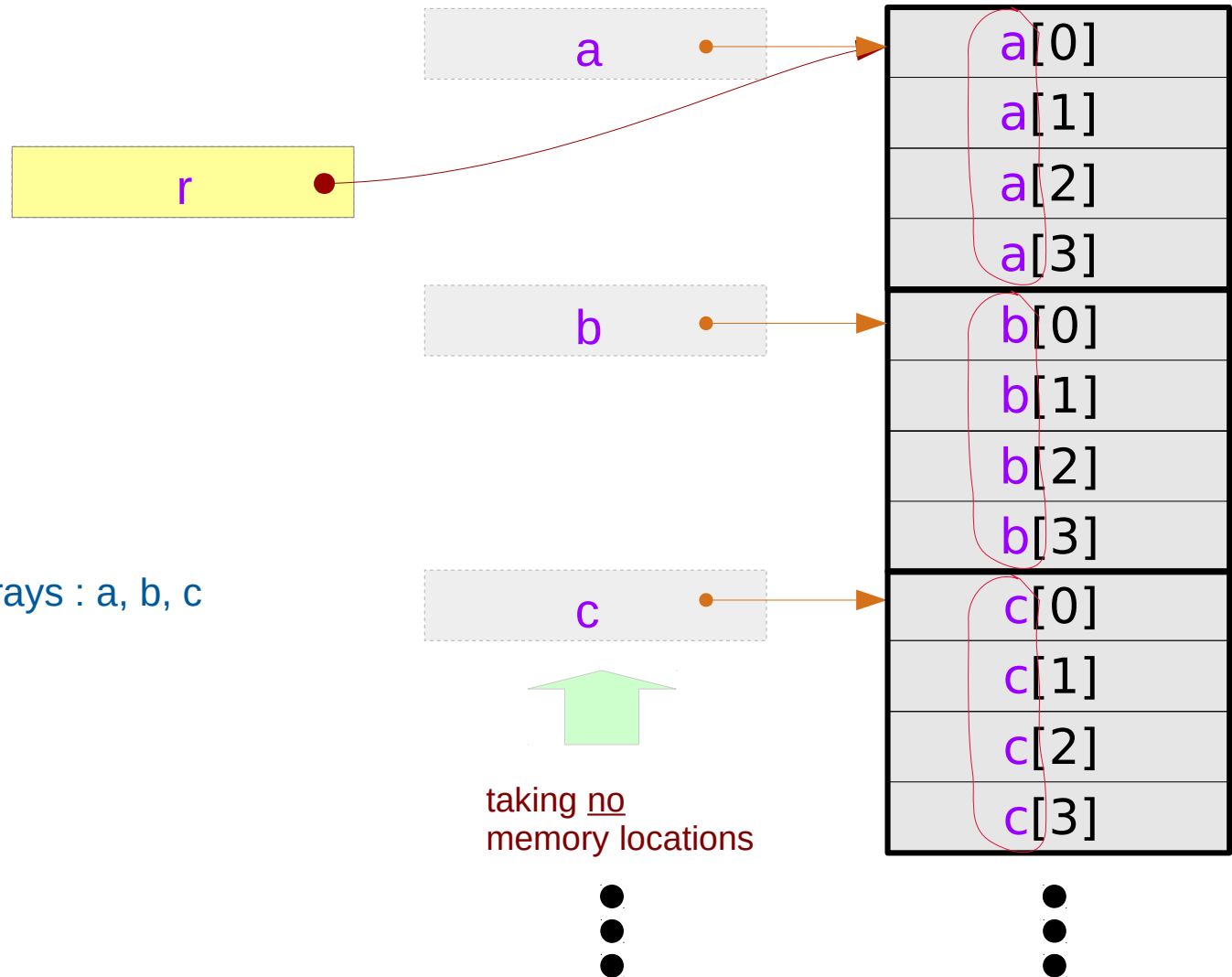
- extending a dimension
- enabling a 2-d access of 1-d arrays



# Contiguous 1-d arrays a, b, c are assumed

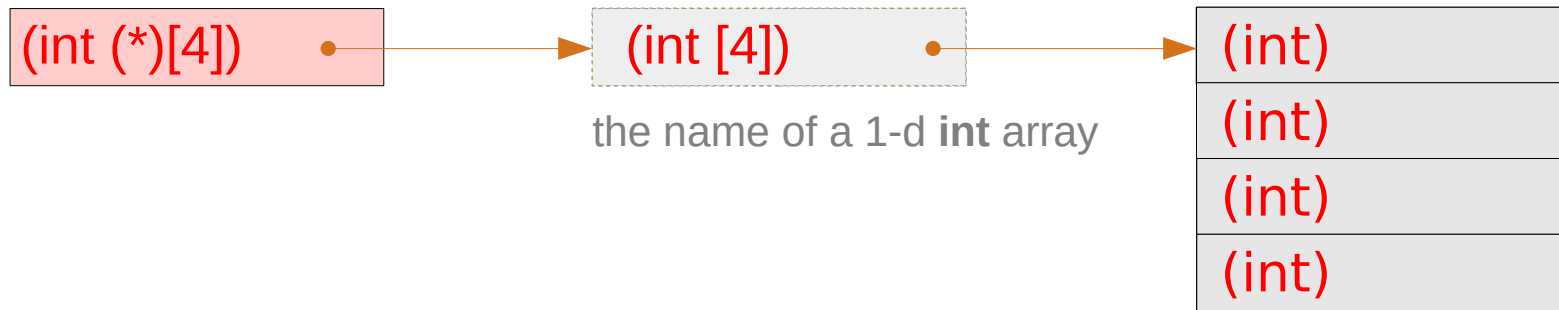
```
int a[4]; int (*r);  
int b[4];  
int c[4];
```

assume contiguous 1-d arrays : a, b, c



# a 1-d array pointer – a type view

a pointer to a 1-d array



the array name  
has a size of 0

does not take  
any memory location

# assigning series of array pointers **p1**, **p2**, **p3**

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

assignment

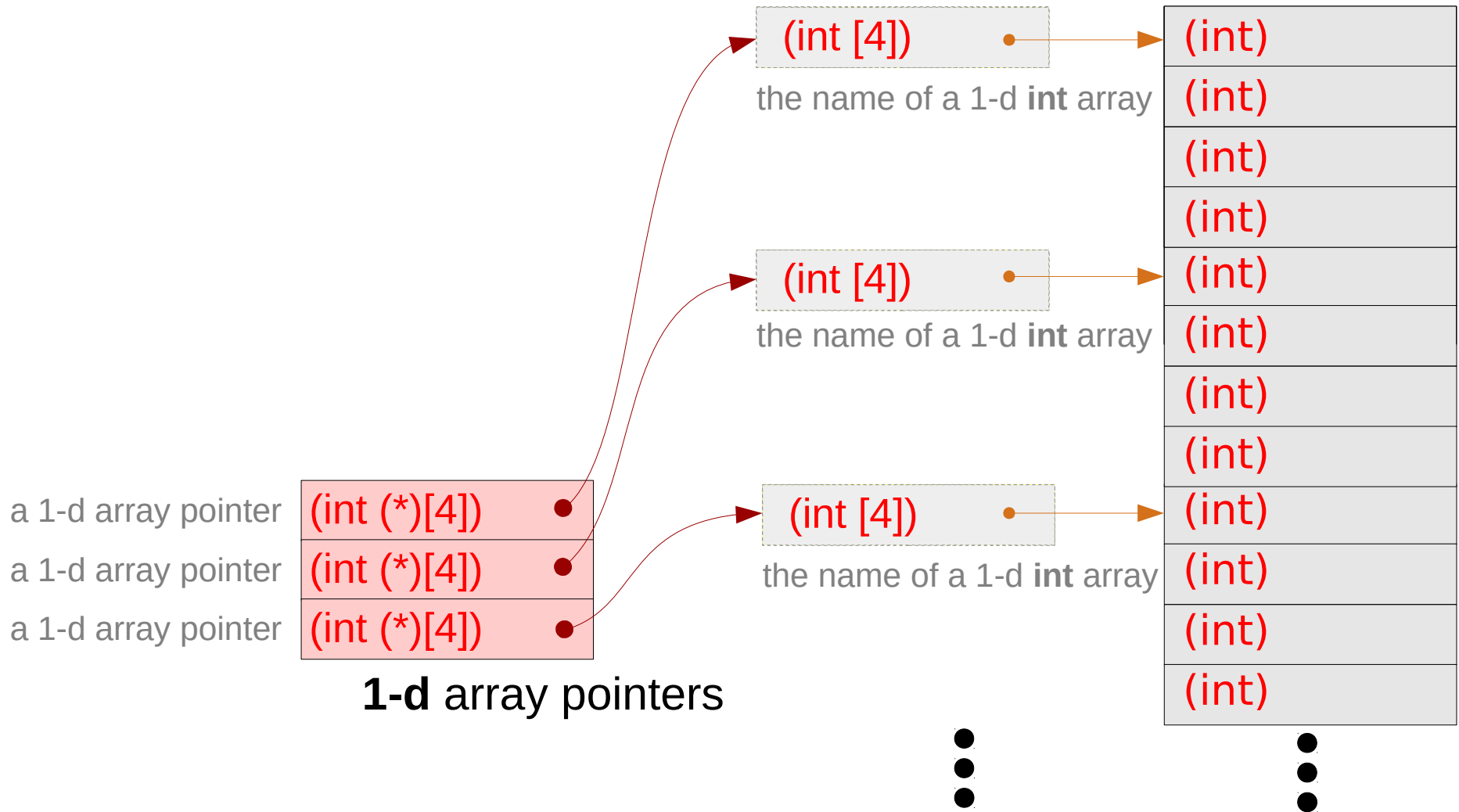
```
p1 = &a  
p2 = &b  
p3 = &c
```



equivalence

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

# type view of array pointers



# 1-d array pointers p1, p2, p3

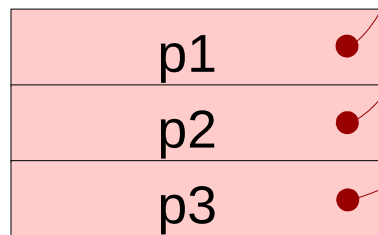
```
int (*p1)[4];  
int (*p2)[4];  
int (*p3)[4];
```

assignment

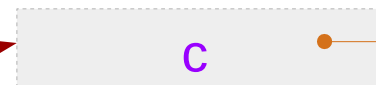
```
p1 = &a  
p2 = &b  
p3 = &c
```

## 1-d array pointers

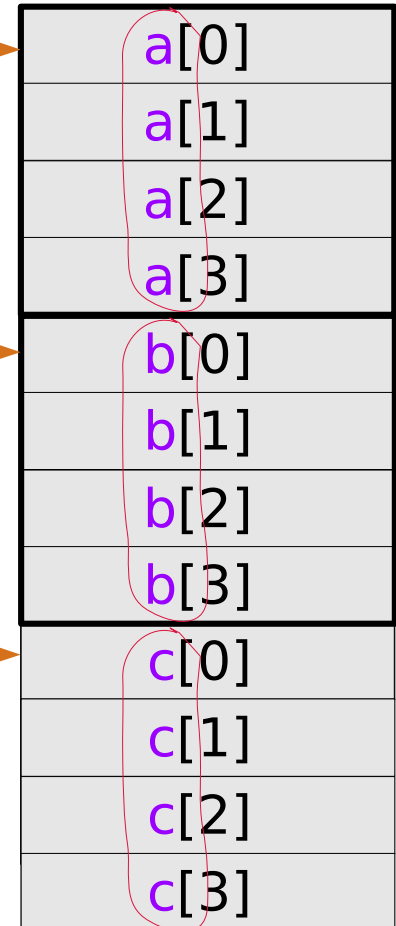
a 1-d array pointer  
a 1-d array pointer  
a 1-d array pointer



assume that array  
p1, p2, and p3 are  
contiguous



taking no  
memory locations



assume that array  
a, b, and c are  
contiguous

# 1-d arrays via p1, p2, p3

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

assignment

equivalence

p1 = &a

(\*p1) ≡ p1[0] ≡ a

p2 = &b

(\*p2) ≡ p2[0] ≡ b

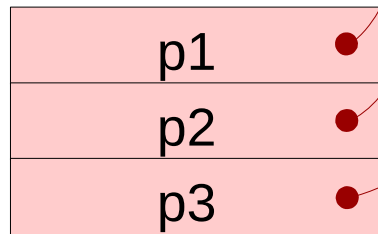
p3 = &c

(\*p3) ≡ p3[0] ≡ c

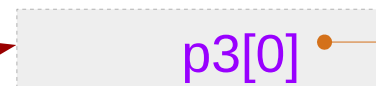
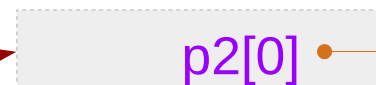
a 1-d array pointer

a 1-d array pointer

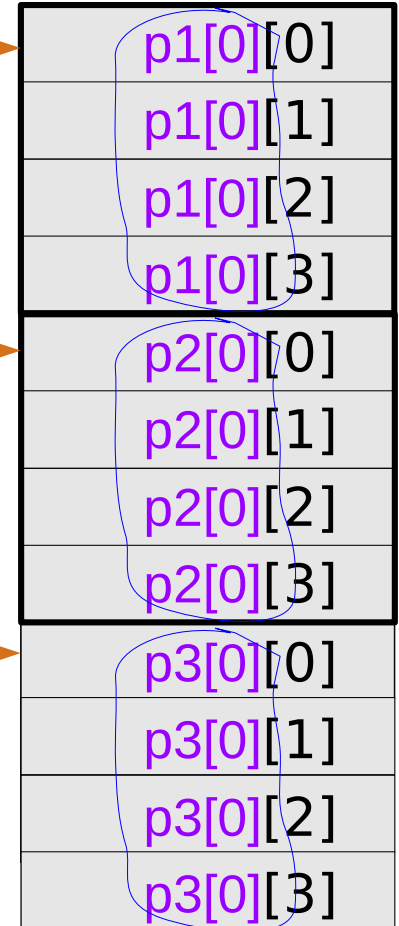
a 1-d array pointer



**1-d array pointers**



taking no  
memory locations



assume that array  
a, b, and c are  
contiguous

# 1-d array pointer p

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

assignment

```
p = &a
```

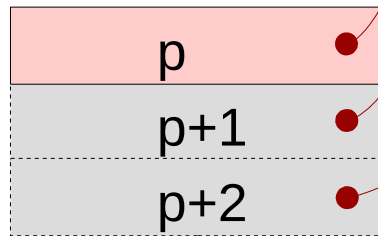
equivalence

```
(*p) ≡ p[0] ≡ a
```

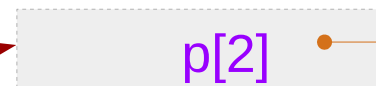
a 1-d array pointer

a 1-d array pointer

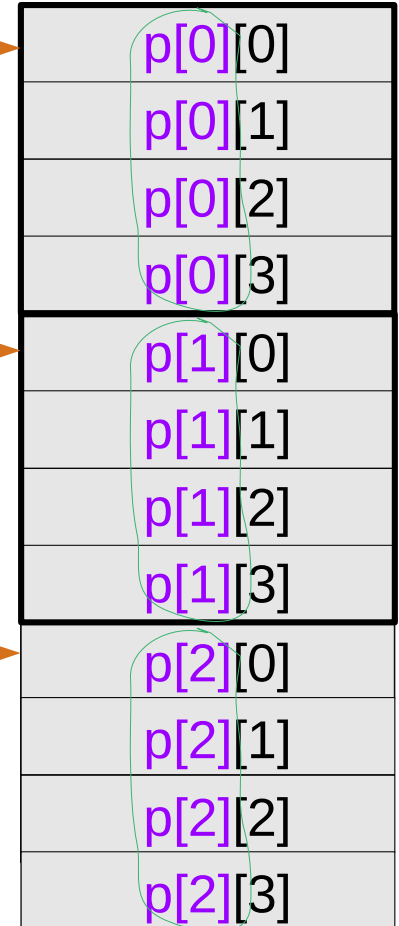
a 1-d array pointer



**1-d array pointers**



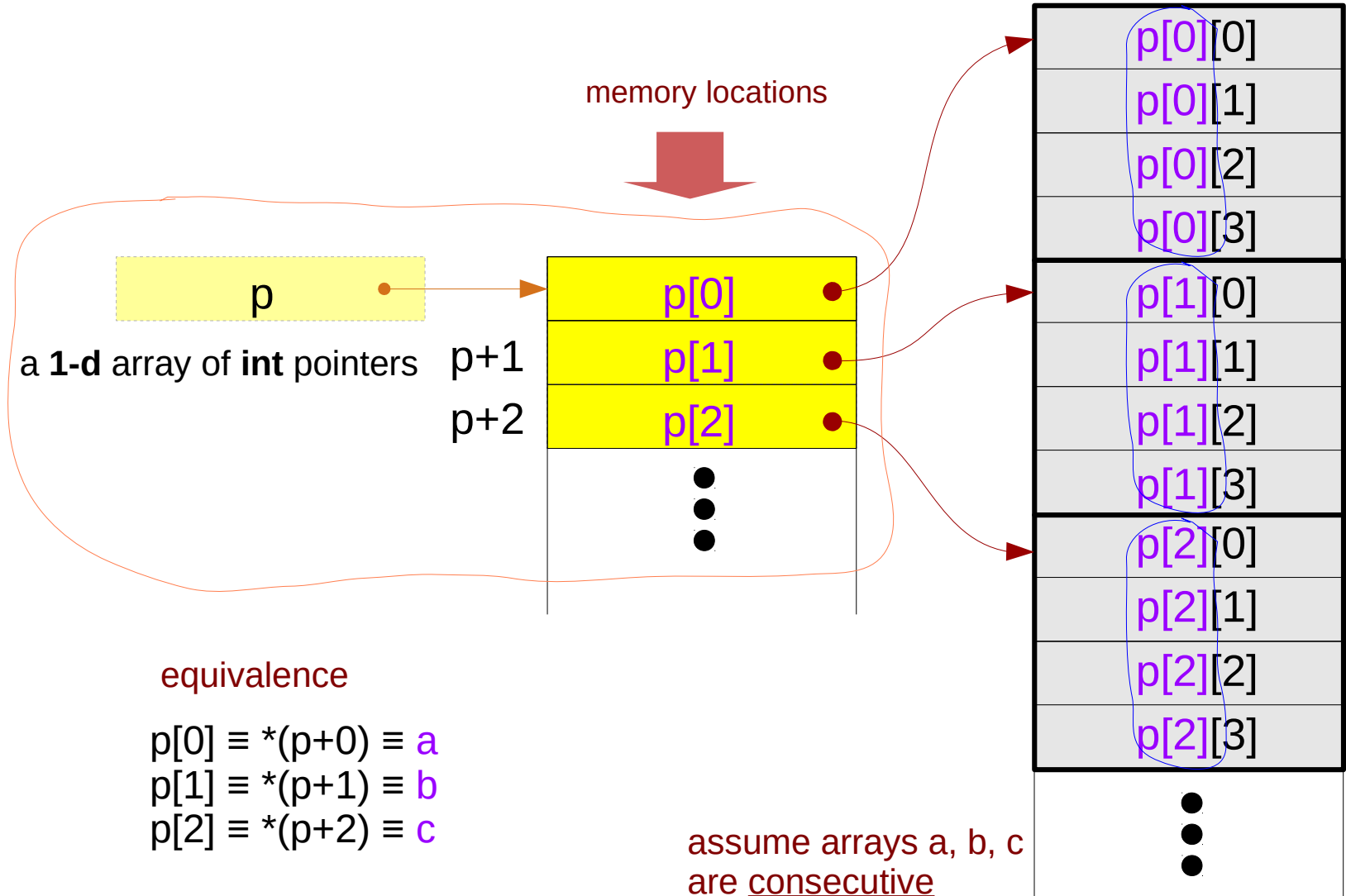
taking no  
memory locations



assume that array  
a, b, and c are  
contiguous

# 1-d array p of integer pointers

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





# an array **p** of array pointers

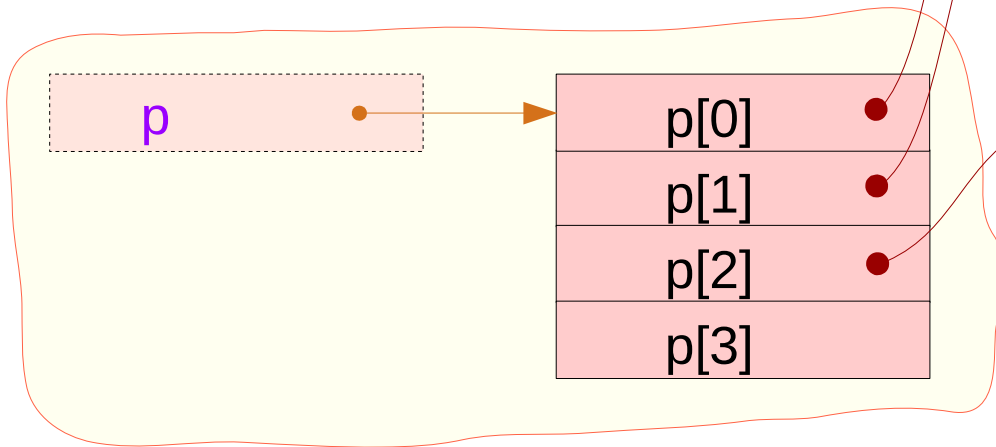
```
int (*p[4])[4];
```

assignment

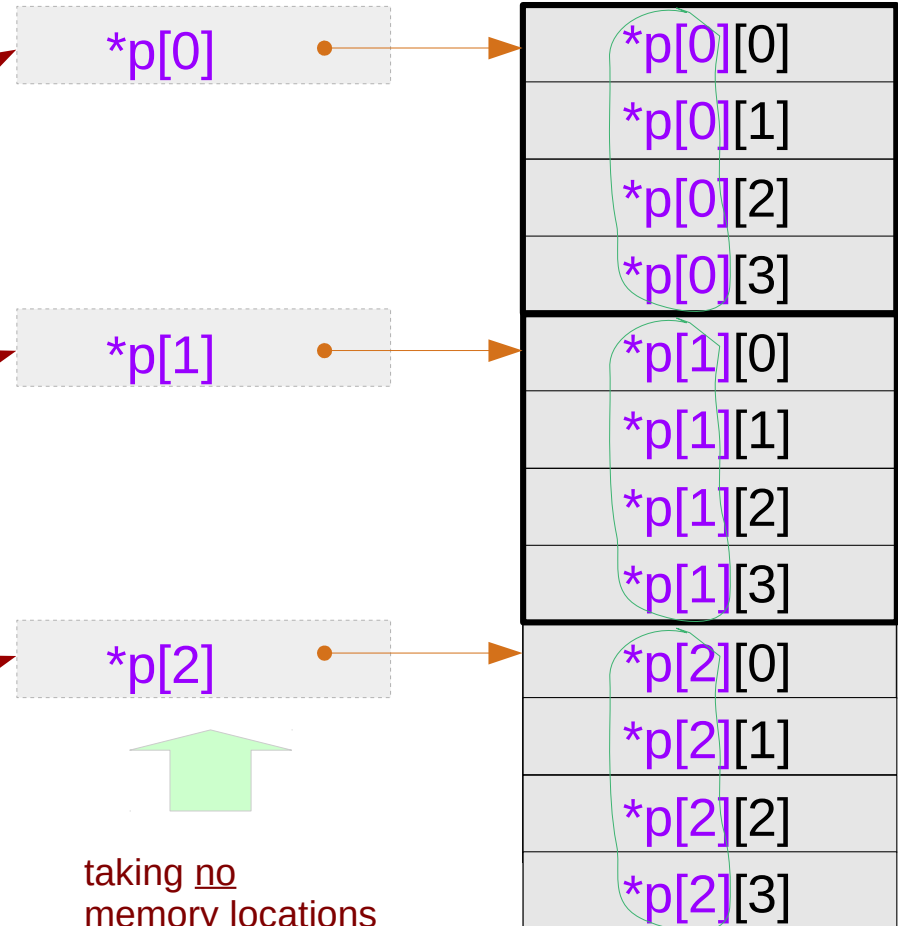
equivalence

$p[0] = \&a$	$*p[0] \equiv a$
$p[1] = \&b$	$*p[1] \equiv b$
$p[2] = \&c$	$*p[2] \equiv c$
$p[3] = \&d$	$*p[3] \equiv d$

a 1-d array pointer

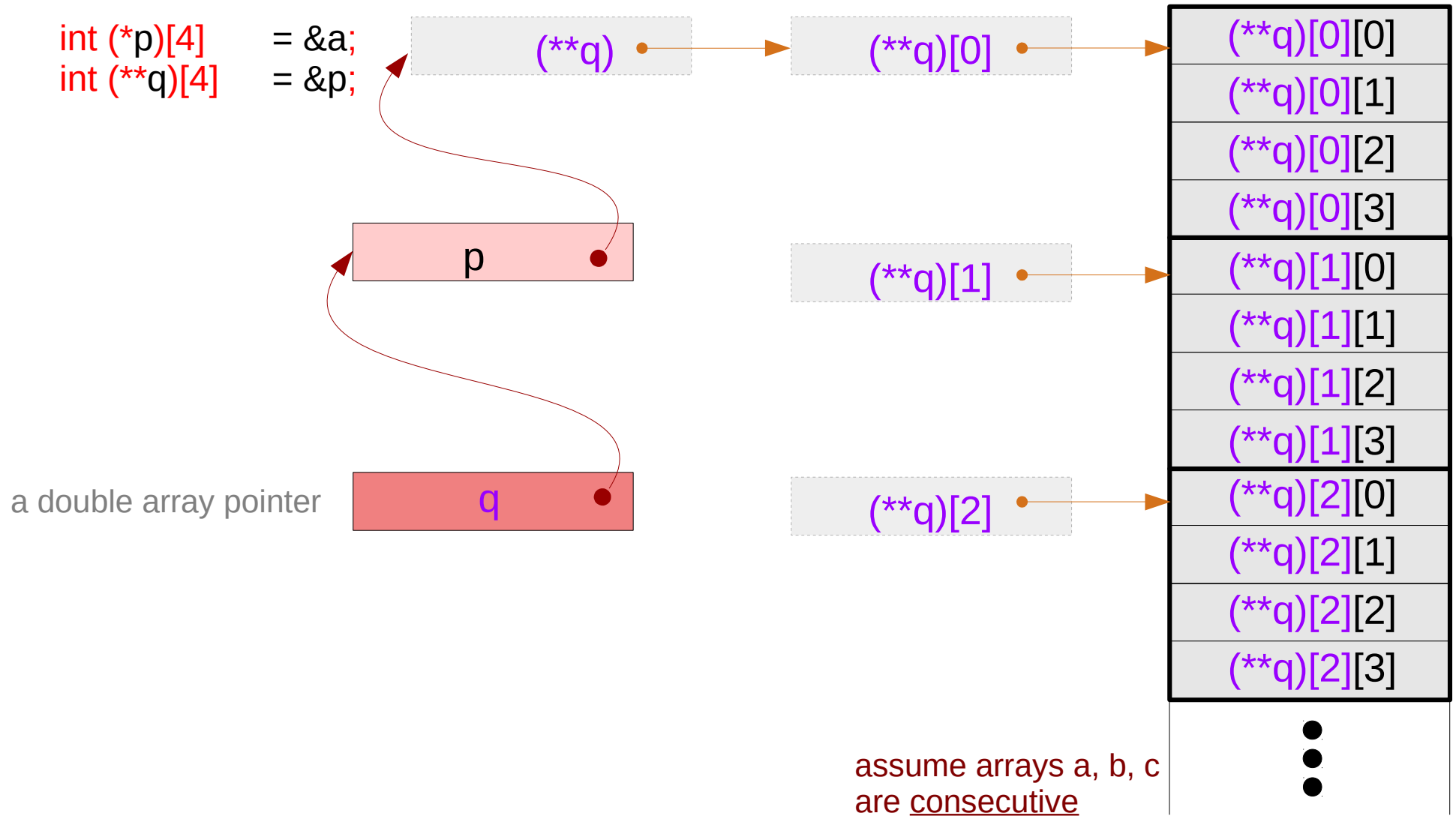


An array of **1-d** array pointers



assume that array  
a, b, and c  
are contiguous

# a double array pointer q



# 1-d array pointer to consecutive 1-d arrays

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

a pointer to a pointer array



**1-d** array pointer

assignment

```
p = &a
```

equivalence

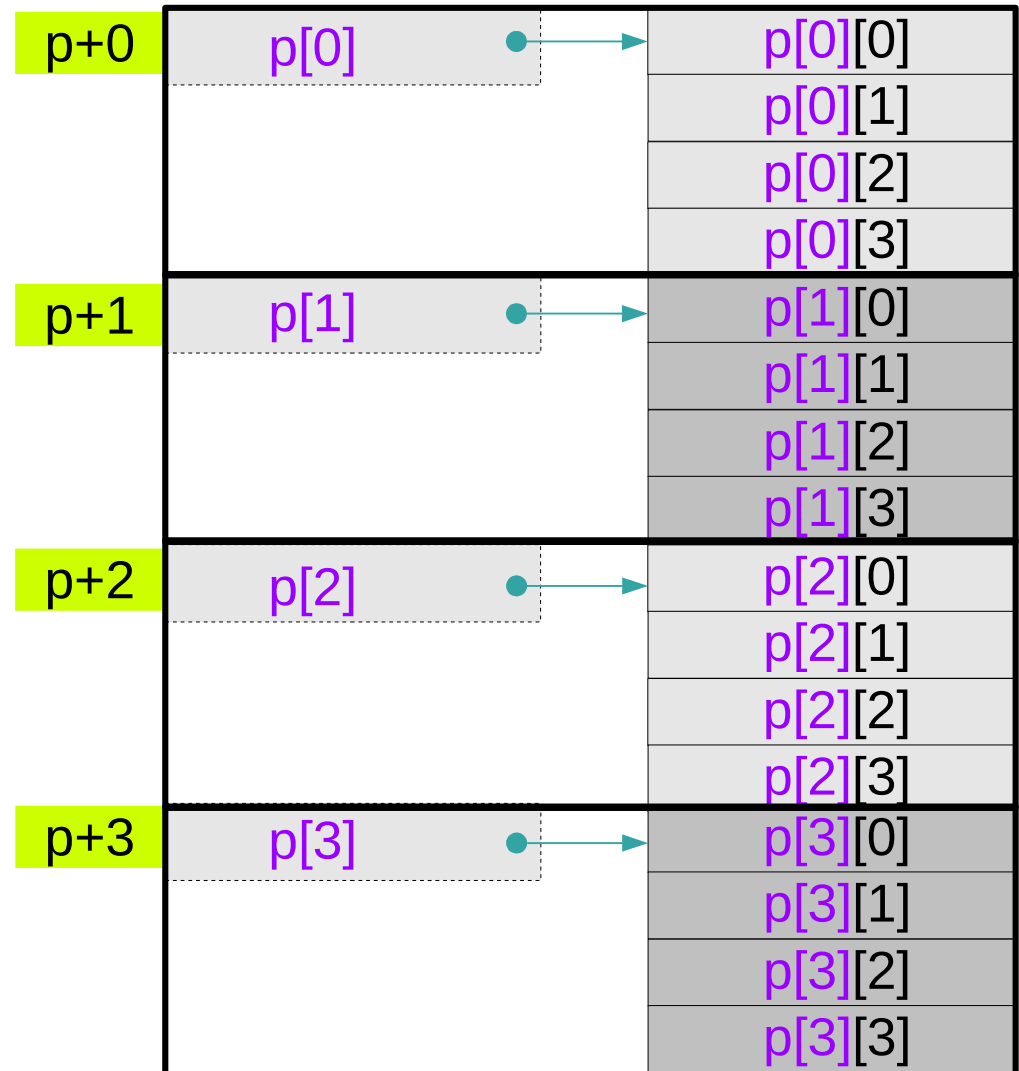
```
*(p+0) ≡ p[0] ≡ a
```

```
*(p+1) ≡ p[1] ≡ b
```

```
*(p+2) ≡ p[2] ≡ c
```

```
*(p+2) ≡ p[2] ≡ d
```

if arrays a, b, c, d  
are consecutive



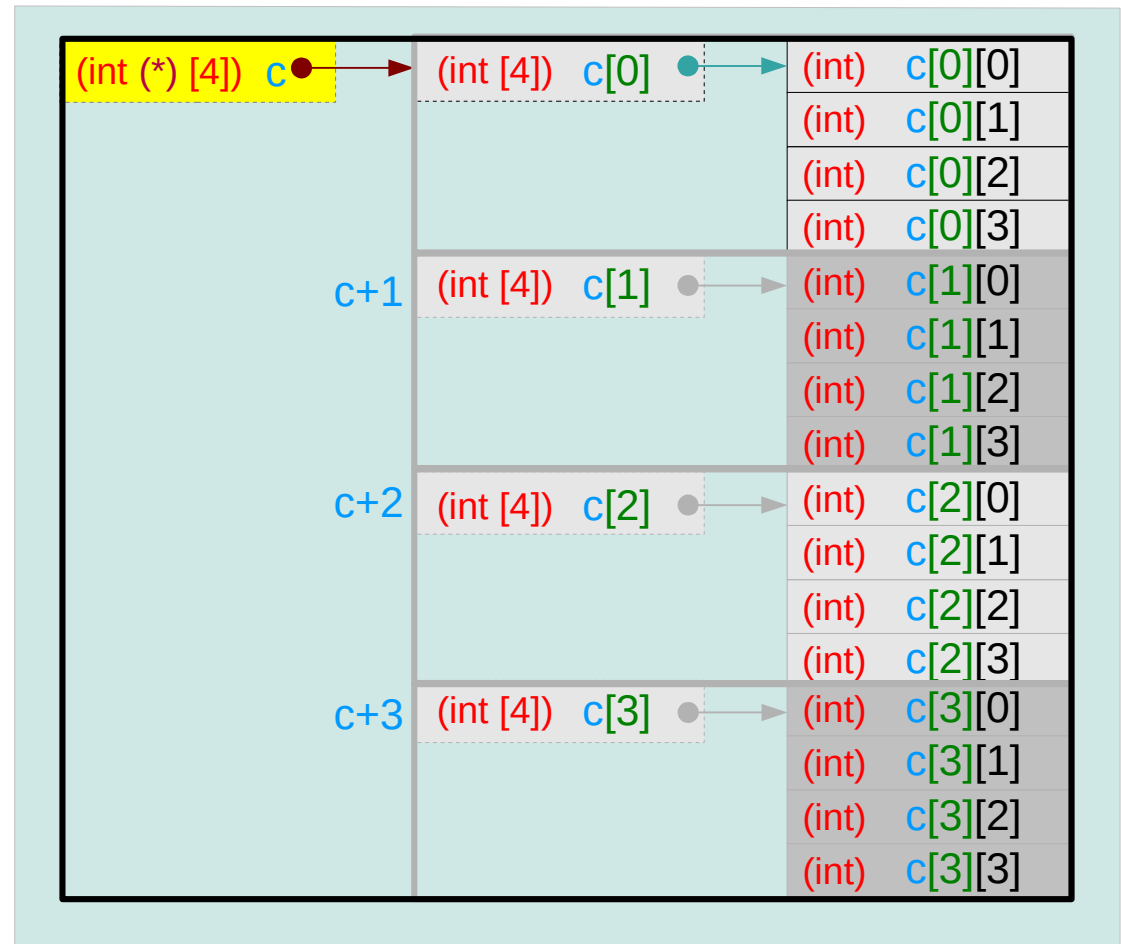
# A 2-d array and its sub-arrays – array name

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

**c** :

- the **2-d array name**
- the **2-d array starting address**
- the **1-d array pointer**  
points to its **1<sup>st</sup> 1-d sub-array**

compilers do not allocate  
**c**'s memory location



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

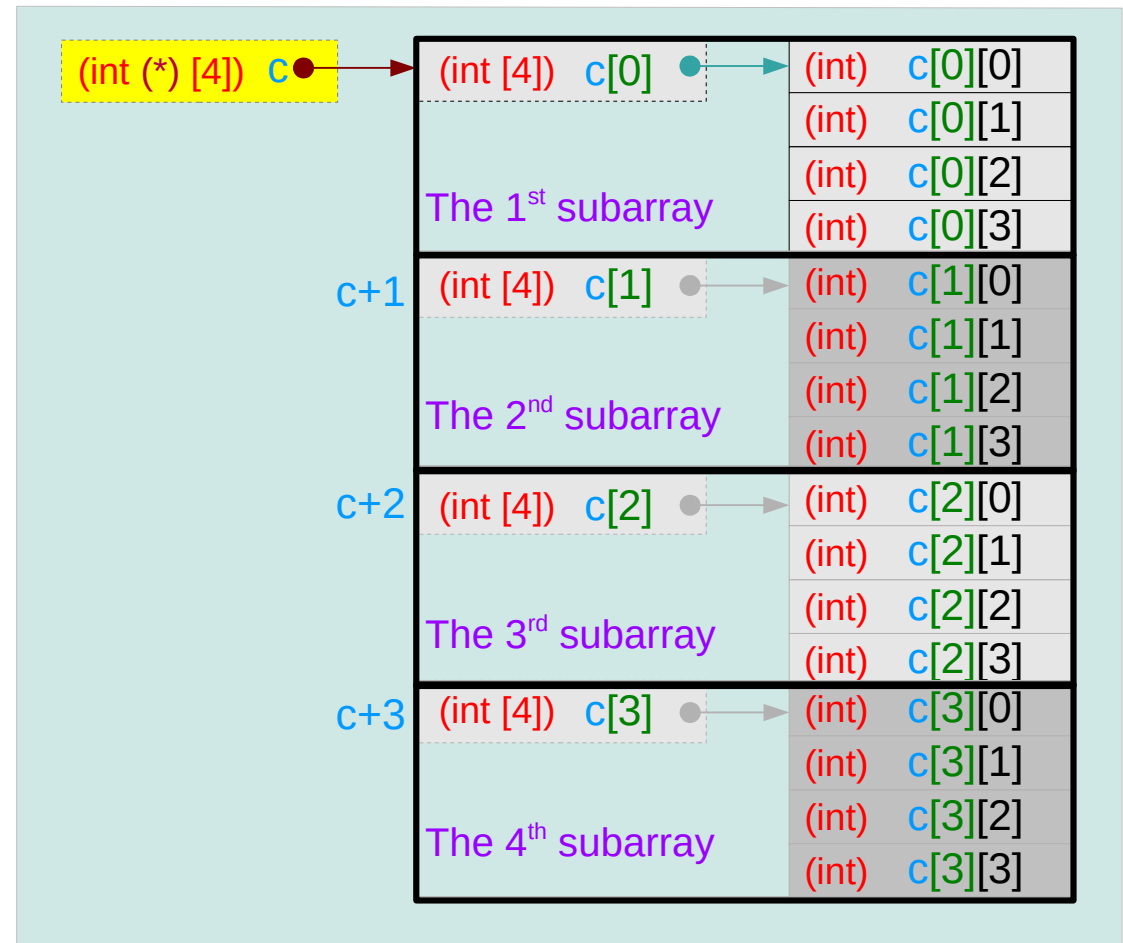
```
int c[4][4];
```

**c[i]**

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

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

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



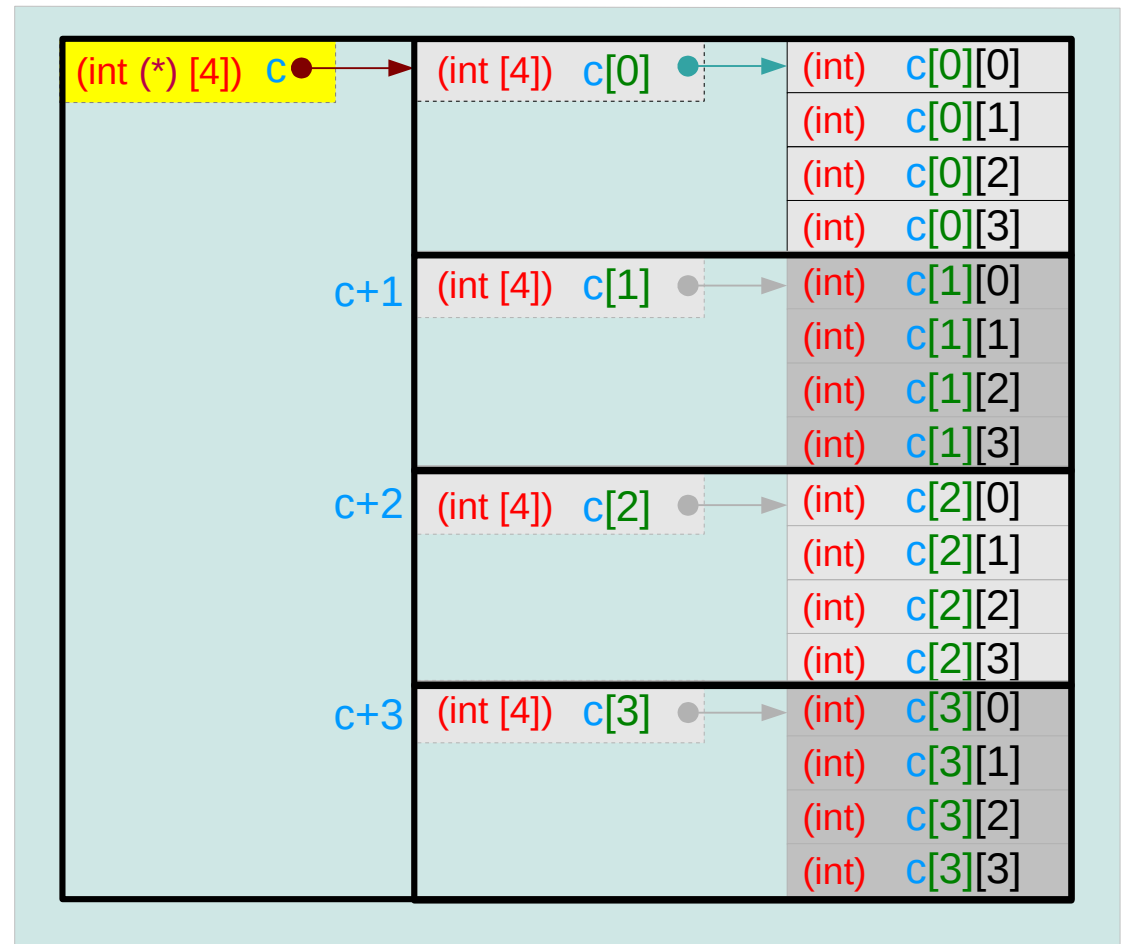
# A 2-d array and its sub-arrays – type sizes

**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)



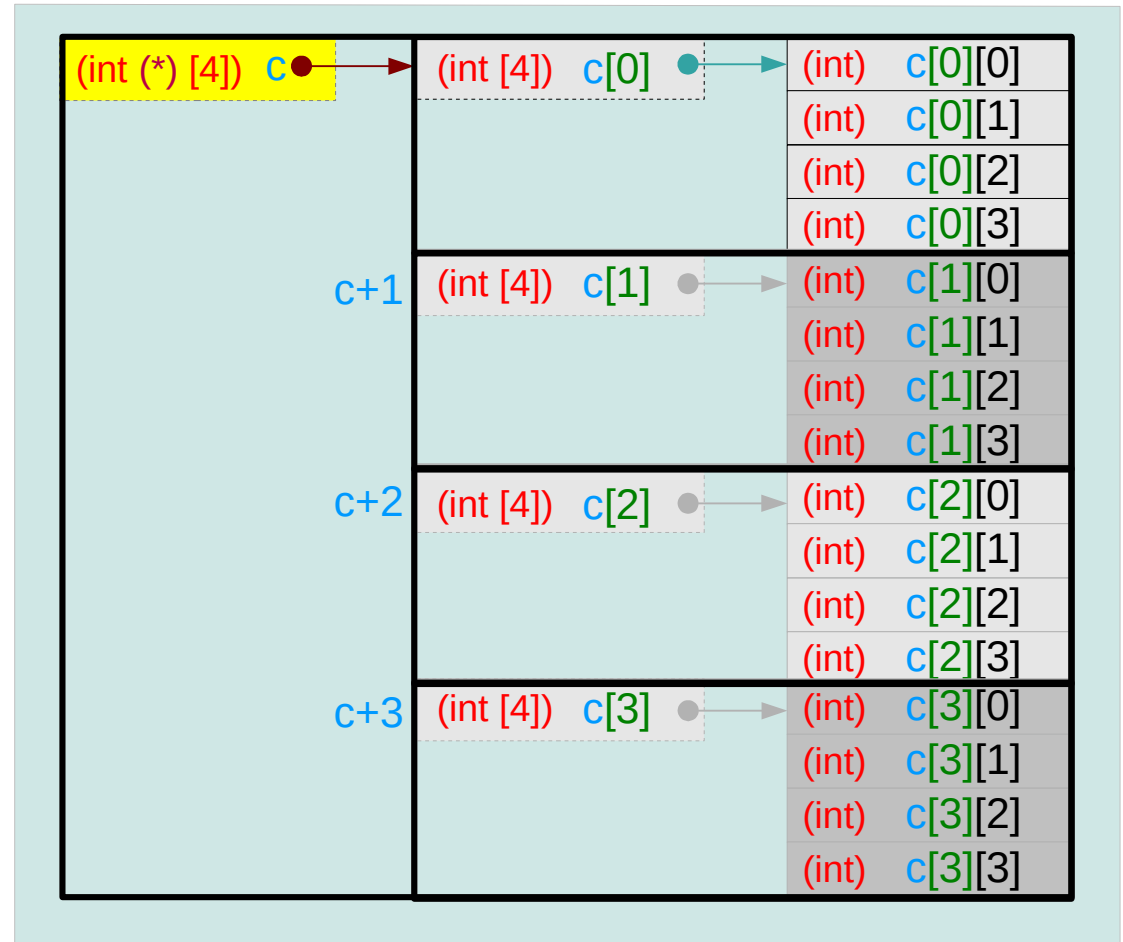
# A 2-d array and its 1-d sub-arrays – a type view

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



# 1-d subarray aggregated data type

The 1<sup>st</sup> subarray **c[0]** (=subarray name)

sizeof(**c[0]**) = 4\*4 bytes

(**c+0**) : start address

The 2<sup>nd</sup> subarray **c[1]** (=subarray name)

sizeof(**c[1]**) = 4\*4 bytes

(**c+1**) : start address

The 3<sup>rd</sup> subarray **c[2]** (=subarray name)

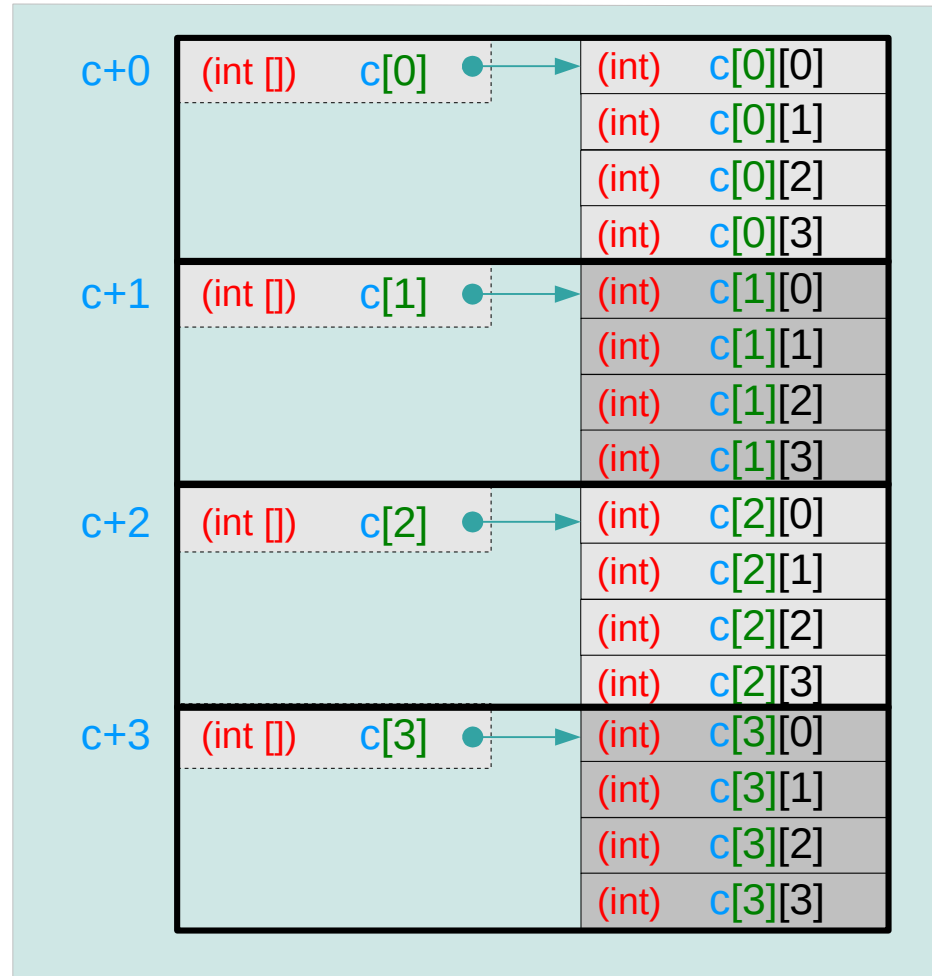
sizeof(**c[2]**) = 4\*4 bytes

(**c+2**) : start address

The 4<sup>th</sup> subarray **c[3]** (=subarray name)

sizeof(**c[3]**) = 4\*4 bytes

(**c+3**) : start address





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

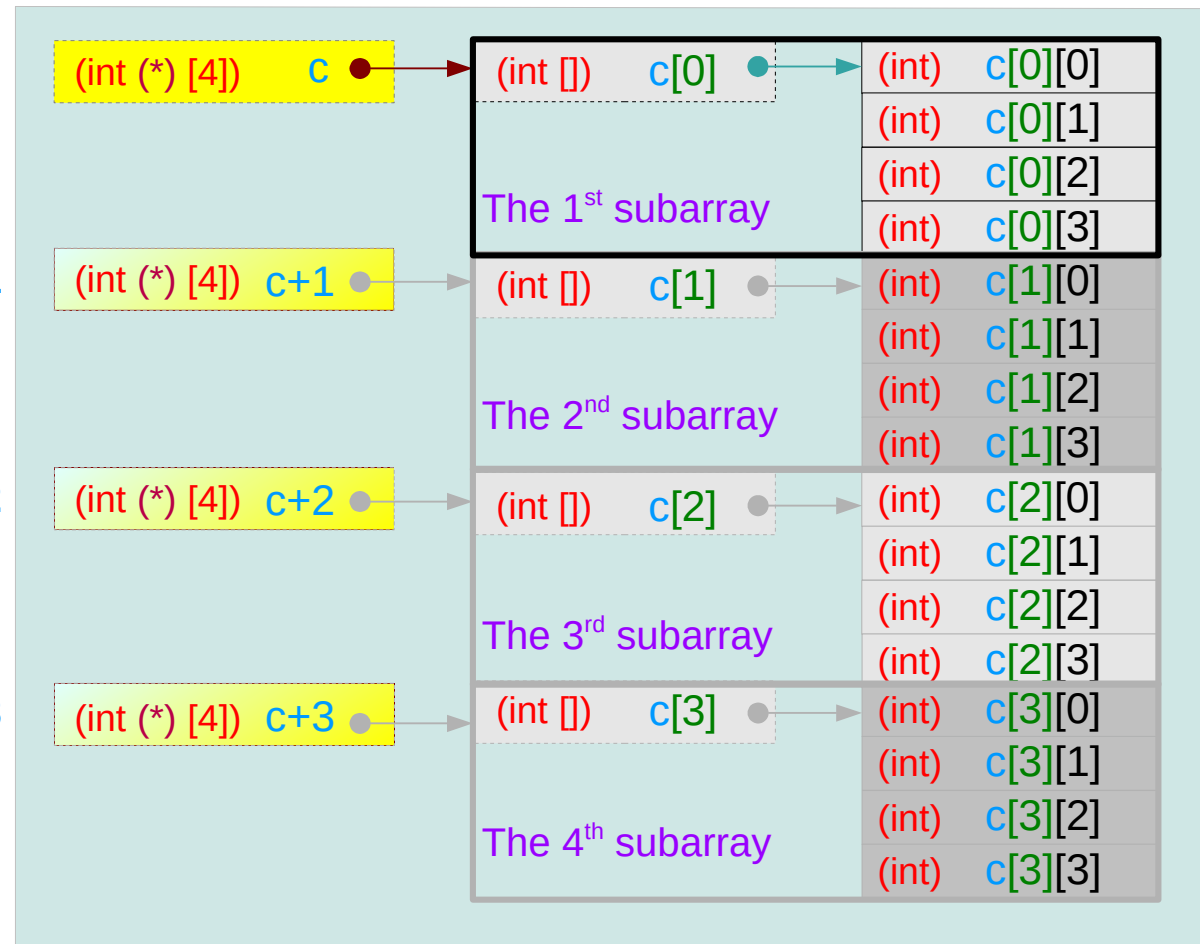
2-d array name **c**

1-d array pointer **c**

1-d array pointer **c+1**

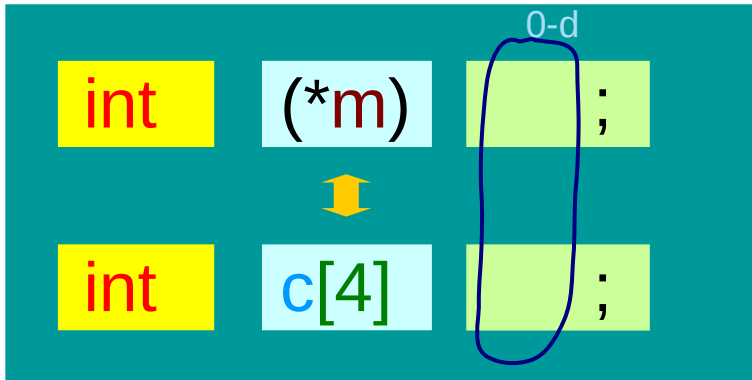
1-d array pointer **c+2**

1-d array pointer **c+3**



# 1-d array and 0-d and 1-d array pointers

0-d array pointer : int pointer



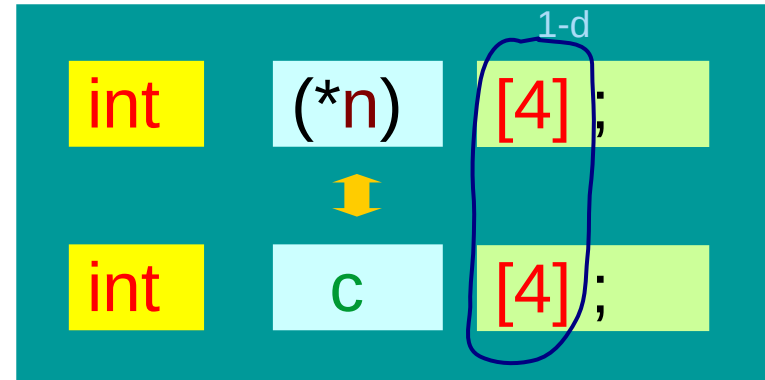
(int (\*))

```
m = c;
```

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

$m[i] \equiv c[i]$

1-d array pointer



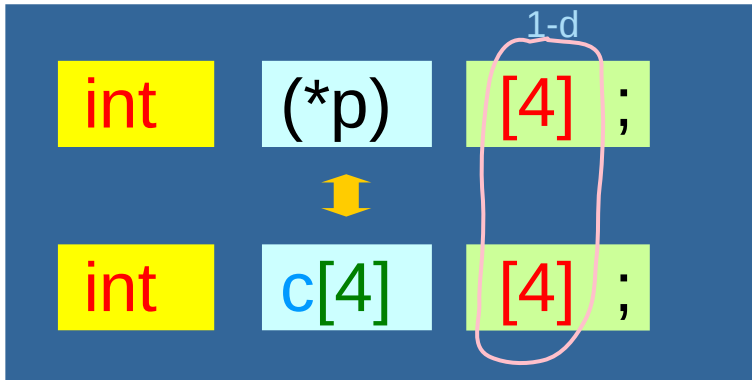
(int(\*)[4])

```
n = &c;
```

$(*n)[i] \equiv n[0][i] \equiv c[i]$

# 2-d array and 1-d and 2-d array pointers

## 1-d array pointer



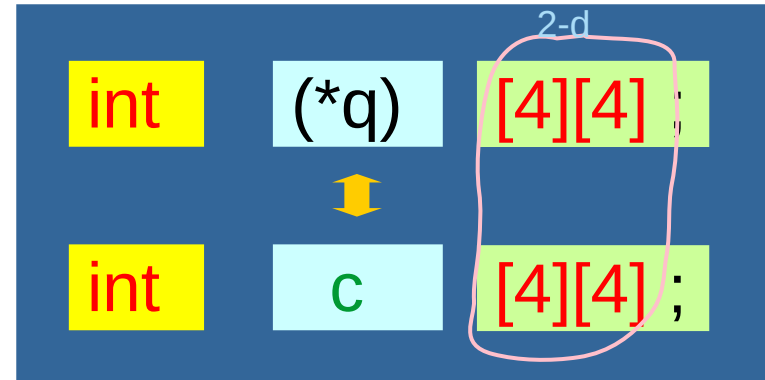
(int (\*) [4])

```
p = c;
```

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

$p[i] \equiv c[i]$

## 2-d array pointer

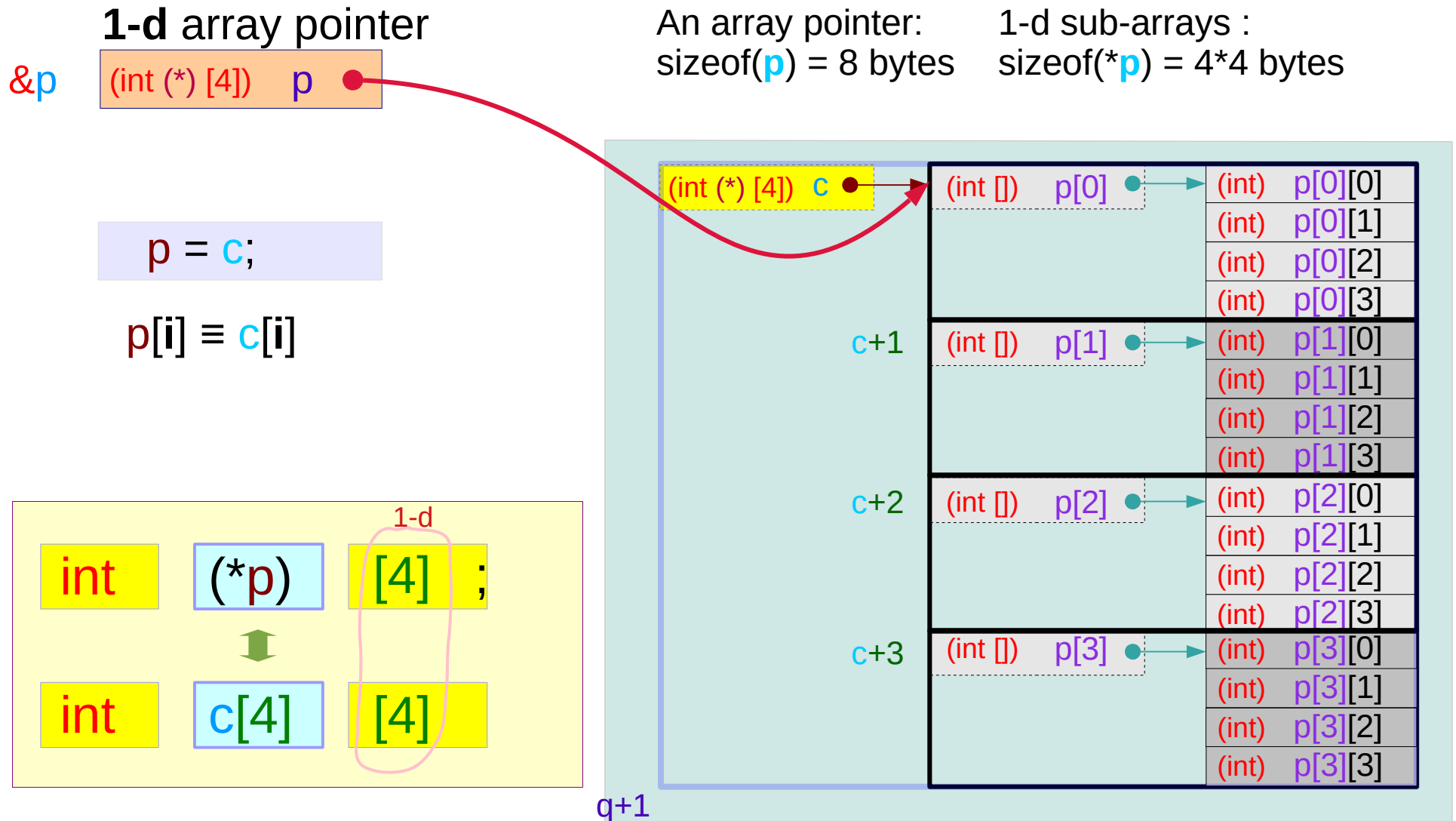


(int(\*)[4][4])

```
q = &c;
```

$(*q)[i][j] \equiv q[0][i][j] \equiv c[i][j]$

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

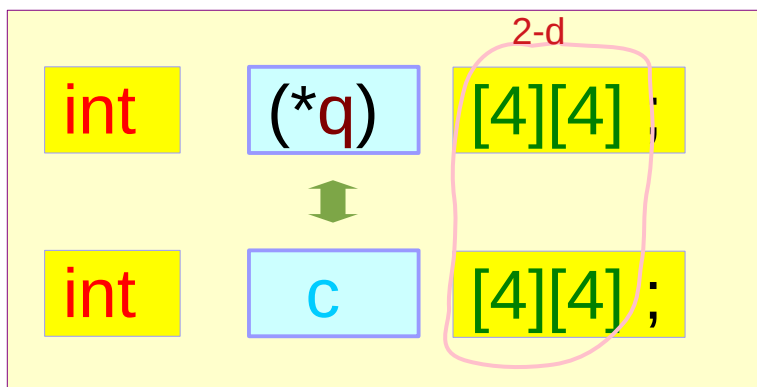


# Using a 2-d array pointer to a 2-d array

**2-d array pointer**  
&p (int(\*)[4][4] q ●

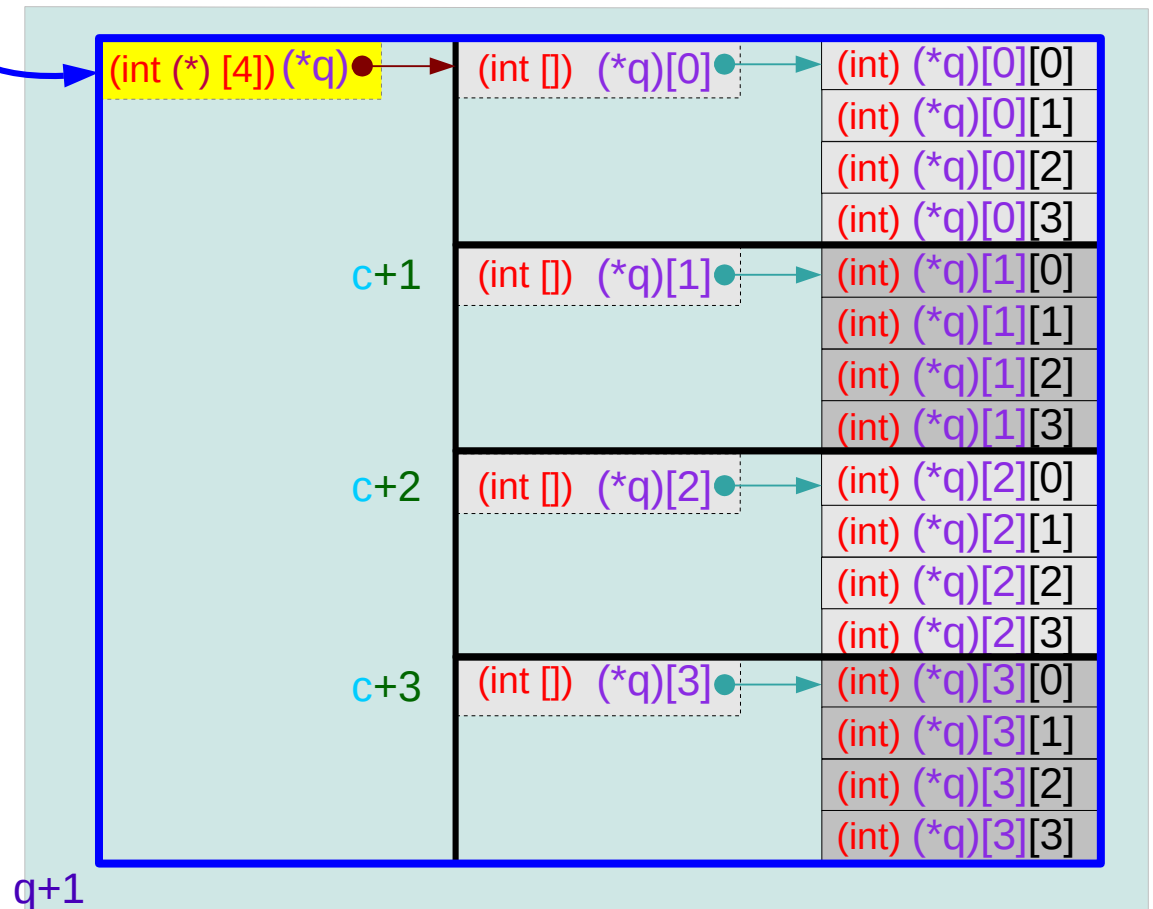
q = &c;

(\*q)[i] ≡ c[i]



An array pointer:  
sizeof(q) = 8 bytes

1-d sub-arrays :  
sizeof(\*q) = 4\*4\*4 bytes



## 2-d array access using array pointers

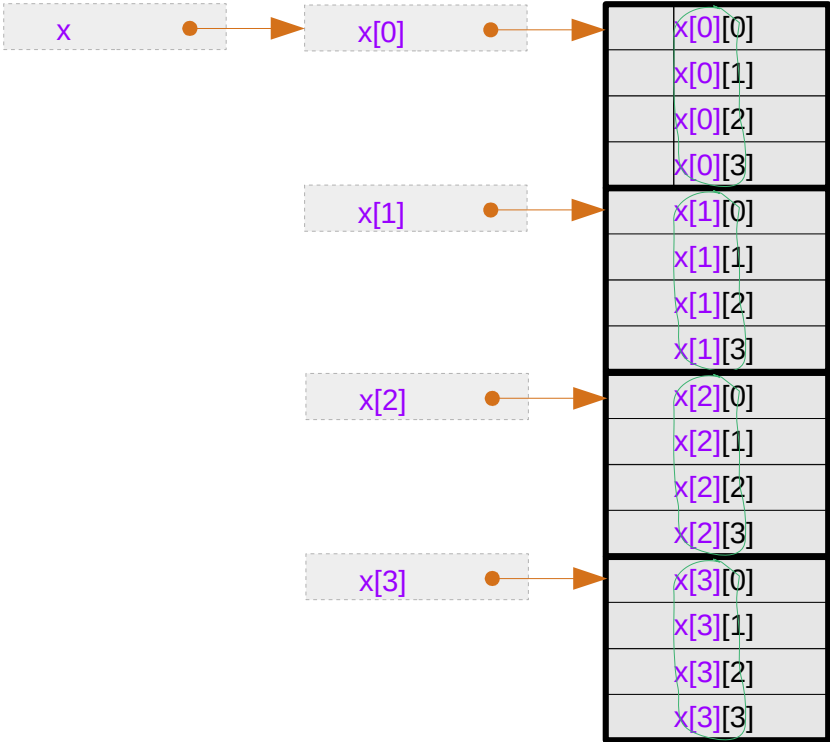
---

- **1-d** array pointer
- **2-d** array pointer
- array of **1-d** array pointers

# Accessing a 2-d array

```
int x[4][4];
```

A 2-d array



# Type definitions

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

A **1-d** array **p** of integer pointers

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

A **1-d** array pointer **p**

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

A **2-d** array pointer **p**

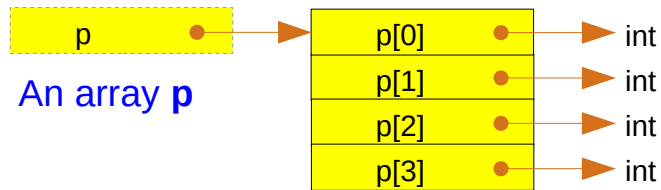
```
int (*p[4])[4];
```

An array **p** of **1-d** array pointers

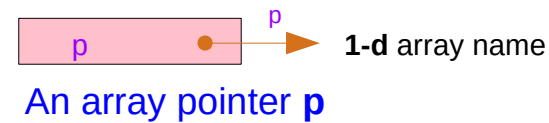


# Arrays and Array Pointers **p**

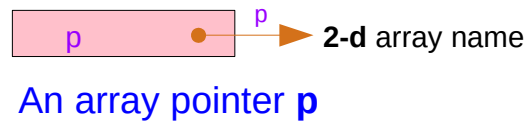
A **1-d** array **p** of integer pointers  
`int *p[4];`



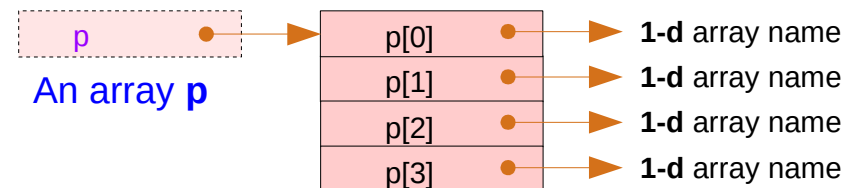
A **1-d** array pointer **p**  
`int (*p)[4];`



A **2-d** array pointer **p**  
`int (*p)[4][4];`



An array **p** of **1-d** array pointers  
`int (*p[4])[4];`



# Sizes of `p`, `p[0]`, `(*p)`, `(*p)[0]`, `*p[0]`

A **1-d** array `p` of integer pointers

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

`sizeof(p)=32 (4*8)` an array of int pointers  
`sizeof(p[0])=8` an int pointer  
`sizeof(p[0][0])=4` an integer

An array `p`

A **1-d** array pointer `p`

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

`sizeof(p)=8` a 1-d array pointer  
`sizeof(*p)=16 (4*4)` a 1-d array  
`sizeof((*p)[0])=4` an integer

An array pointer `p`

A **2-d** array pointer `p`

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

`sizeof(p)=8` a 2-d array pointer  
`sizeof(*p)=64` a 2-d array (4\*4\*4)  
`sizeof((*p)[0])=16` a 1-d array (4\*4)

An array pointer `p`

An array `p` of **1-d** array pointers

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

`sizeof(p)=32 (4*8)` a 1-d array of 1-d array pointers  
`sizeof(p[0])=8` a 1-d array pointer  
`sizeof(*p[0])=16 (4*4)` a 1-d array

An array `p`

# Initialization

A **1-d** array **p** of integer pointers

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

```
p[0] = x[0];  
p[1] = x[1];  
p[2] = x[2];  
p[3] = x[3];
```

A **2-d** array pointer **p**

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

```
p = &x
```

A **1-d** array pointer **p**

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

```
p = &x[0]
```

An array **p** of **1-d** array pointers

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

```
p[0] = x[0];  
p[1] = x[1];  
p[2] = x[2];  
p[3] = x[3];
```

# Access methods

A **1-d** array **p** of integer pointers

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

$*(p[m]+n)$   
 $p[m][n]$

A **2-d** array pointer **p**

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

$(*p)[m][n]$

A **1-d** array pointer **p**

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

$*(p+m)[n]$   
 $p[m][n]$

An array **p** of **1-d** array pointers

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

$(*p[m])[n]$   
 $*p[m][n]$

# Initialization

<code>int *p[4]</code>	<code>= {x[0], x[1], x[2], x[3]};</code>	<code>p[m][n]</code>
<code>int (*p)[4]</code>	<code>= &amp;x[0];</code>	<code>p[m][n]</code>
<code>int (*p)[4][4]</code>	<code>= &amp;x;</code>	<code>(*p)[m][n]</code>
<code>int (*p[4])[4]</code>	<code>= {x[0], x[1], x[2], x[3]};</code>	<code>(*p[m])[n]</code>

# Access methods

int \*p[4];

\*p[m]

\*(p[m]+n) = p[m][n]

int (\*p)[4];

(\*p)[n]

\*(p+m)[n] = p[m][n]

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

(\*p)[m][n]

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

(\*p[m])[n]

# Using a 1-d array of integer pointer : `int *p[4]`

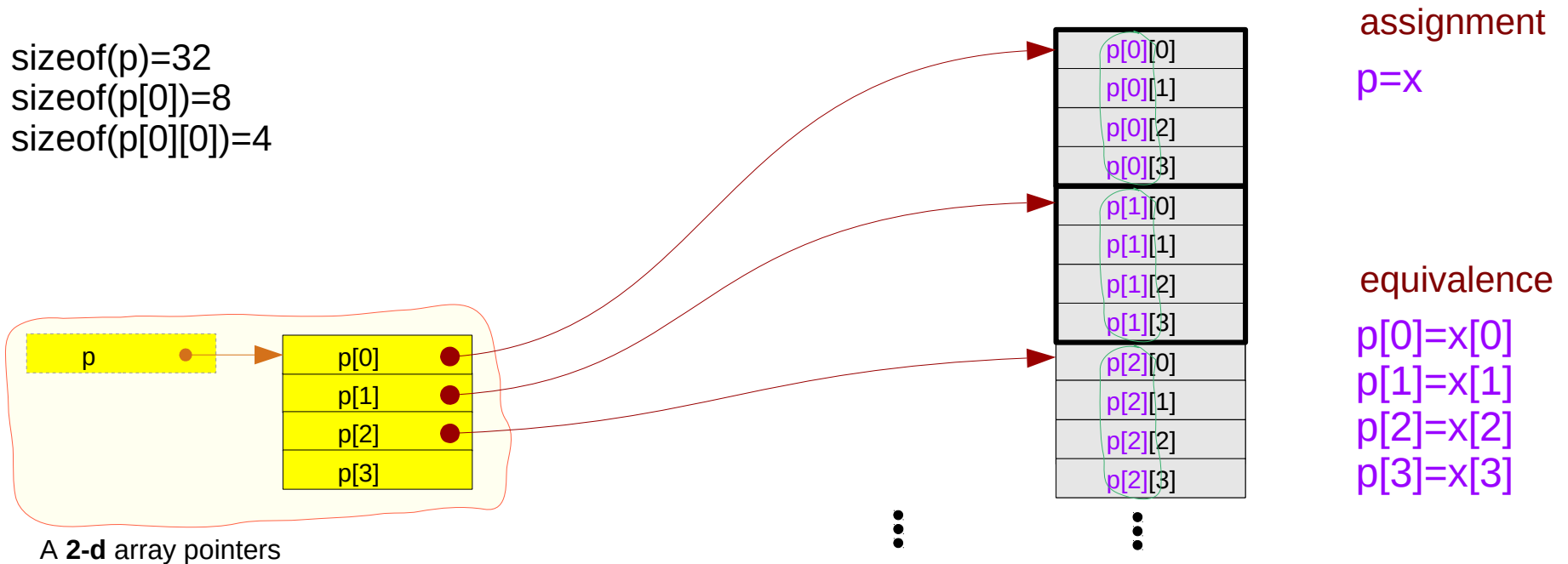
```
int *p[4];
```

Type Definition

```
*(p[m]+n) ≡ p[m][n]
```

Access Method

`sizeof(p)=32`  
`sizeof(p[0])=8`  
`sizeof(p[0][0])=4`



# Using a 1-d array pointer : `int (*p)[4]`

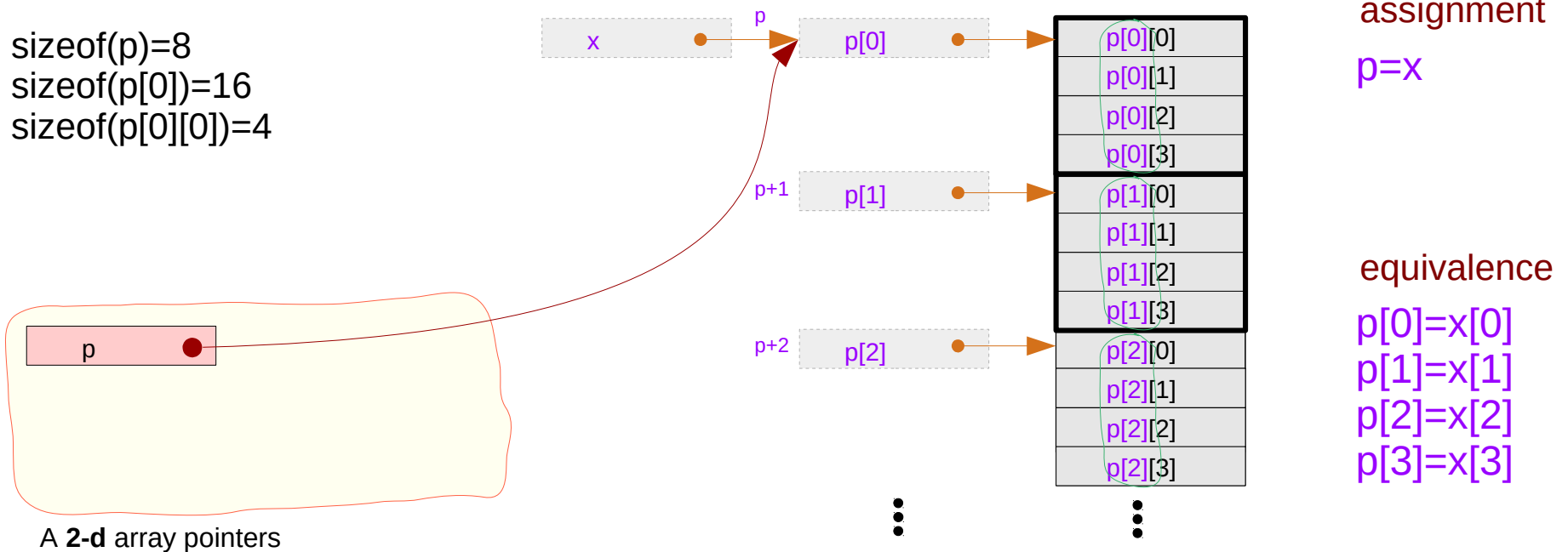
```
int (*p)[4];
```

Type Definition

```
(*p+i)[j];      ≡      p[i][j];
```

Access Method

`sizeof(p)=8`  
`sizeof(p[0])=16`  
`sizeof(p[0][0])=4`





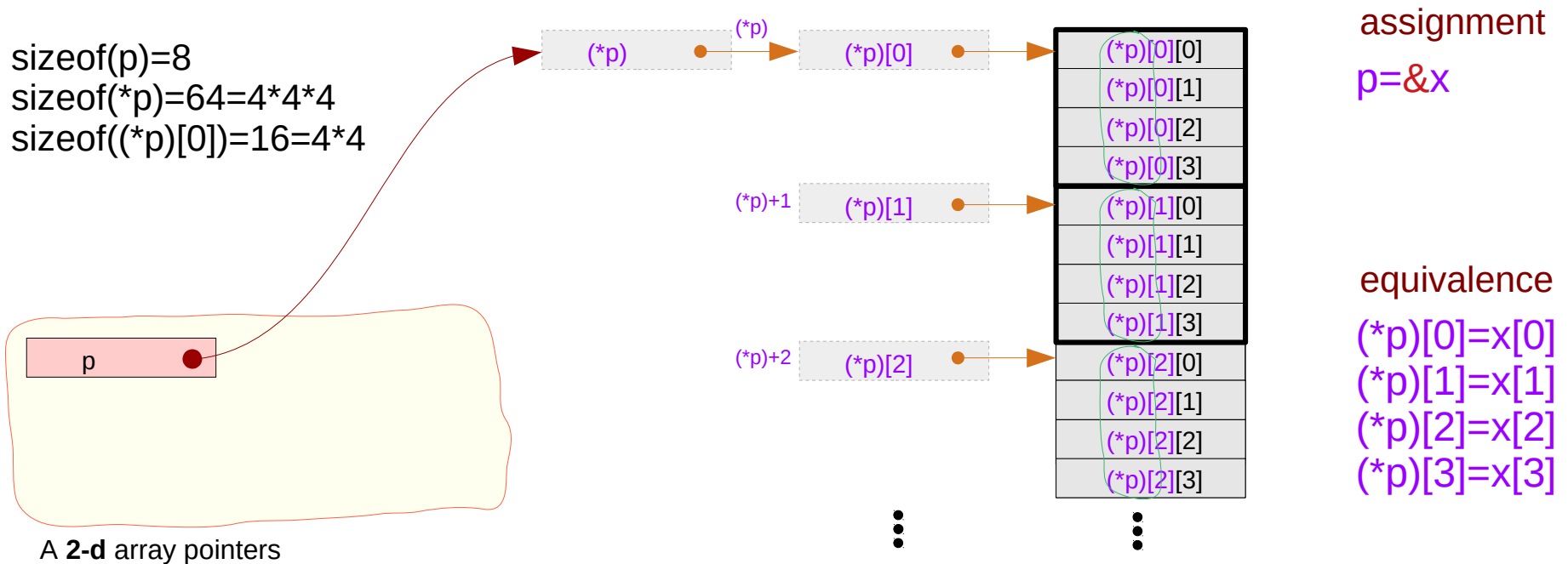
# Using a 2-d array pointer : `int (*p)[4][4]`

`int (*p)[4][4];`  $\equiv$  `int ((*p)[4])[4];`

Type Definition

`(*p)[i][j];`  $\equiv$  `((*p)[i])[j];`

Access Method



# Using an array of 1-d array pointers : `int (*p[4])[4]`

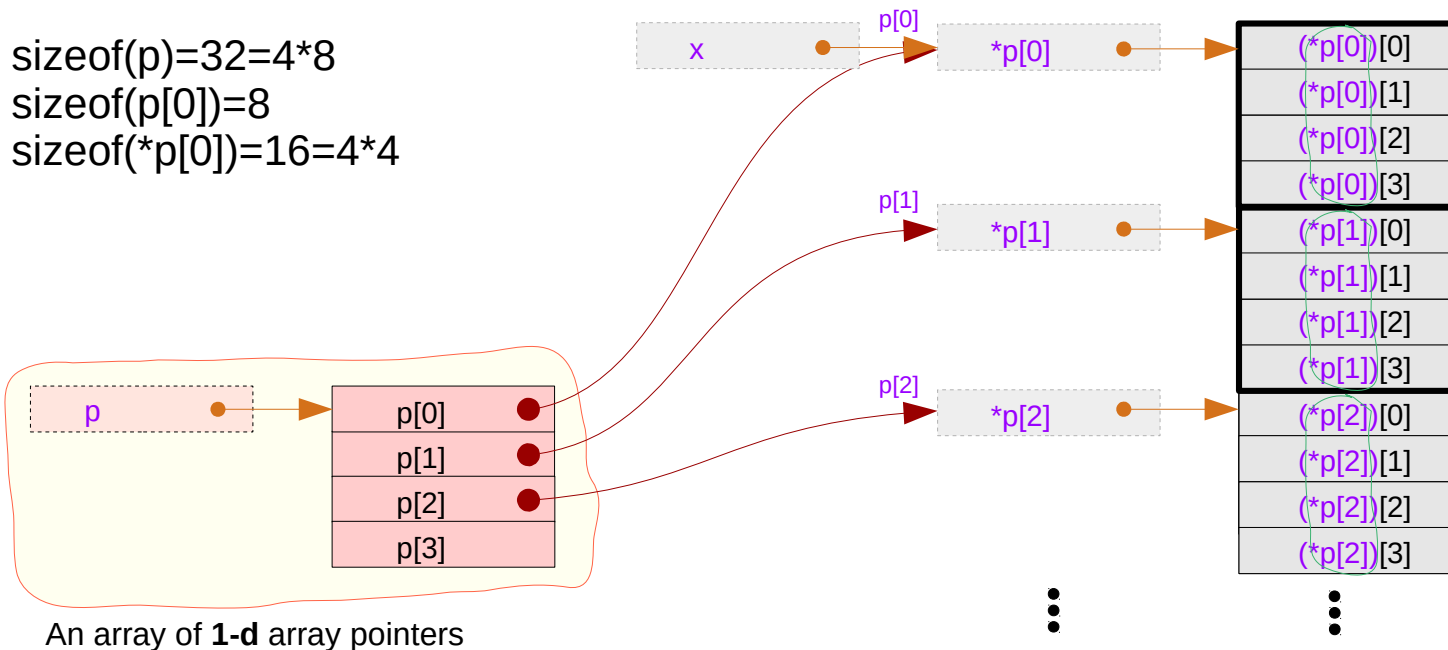
`int (*p[4])[4];`  $\equiv$  `int (*(p[4]))[4];`

Type Definition

`(*p[i])[j];`  $\equiv$  `*(p[i])[j];`

Access Method

`sizeof(p)=32=4*8`  
`sizeof(p[0])=8`  
`sizeof(*p[0])=16=4*4`



assignment

`p[0]=&x[0]`  
`p[1]=&x[1]`  
`p[2]=&x[2]`  
`p[3]=&x[3]`

equivalence

`*p[0]=x[0]`  
`*p[1]=x[1]`  
`*p[2]=x[2]`  
`*p[3]=x[3]`

# int (\*p[4])[4] and (\*p)[4][4] : OK

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

assignment

```
p[0]=&x[0]
p[1]=&x[1]
p[2]=&x[2]
p[3]=&x[3]
```

equivalence

```
*p[0]=x[0]
*p[1]=x[1]
*p[2]=x[2]
*p[3]=x[3]
```

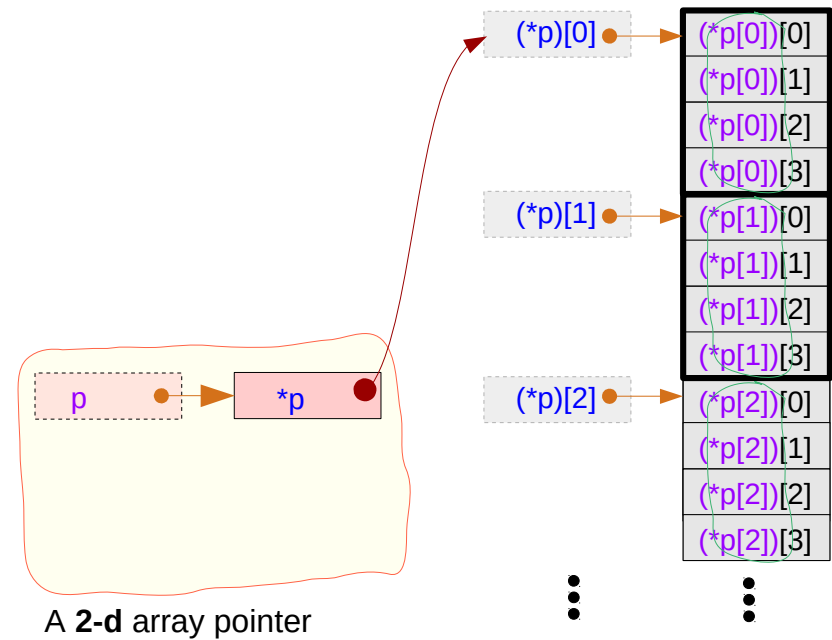
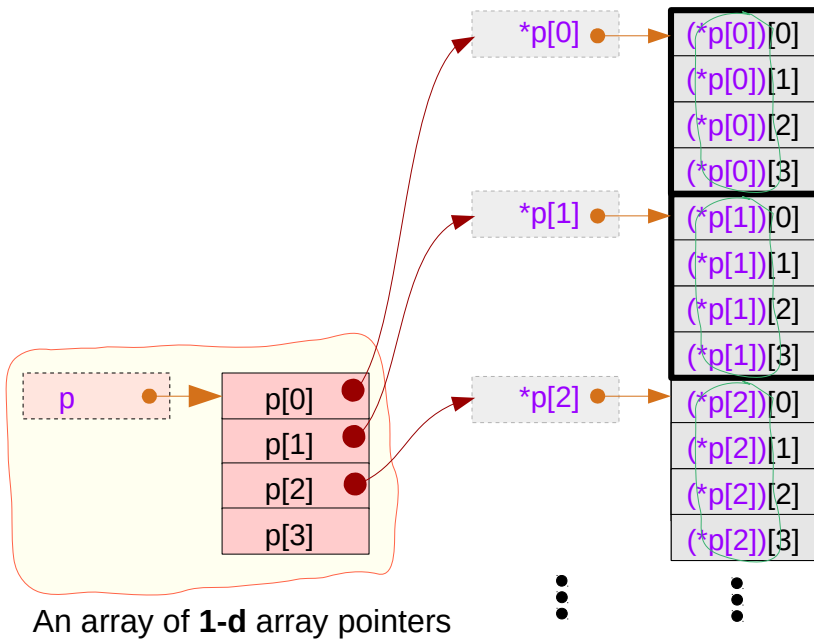
**int (\*p)[4][4];**

assignment

```
p=&x
```

equivalence

```
(*p)[0]=x[0]
(*p)[1]=x[1]
(*p)[2]=x[2]
(*p)[3]=x[3]
```



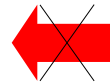
# int (\*p)[4][4] and (\*p[i])[j] : not OK

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

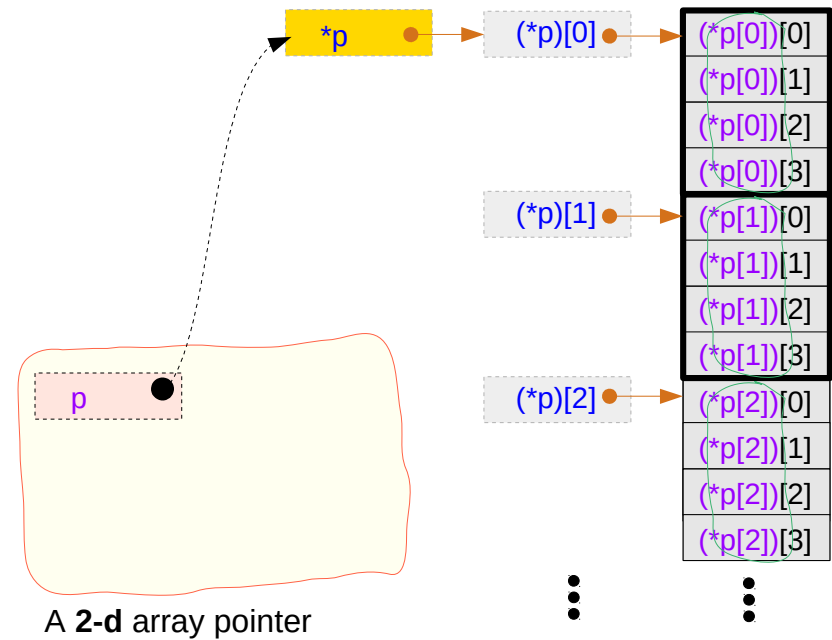
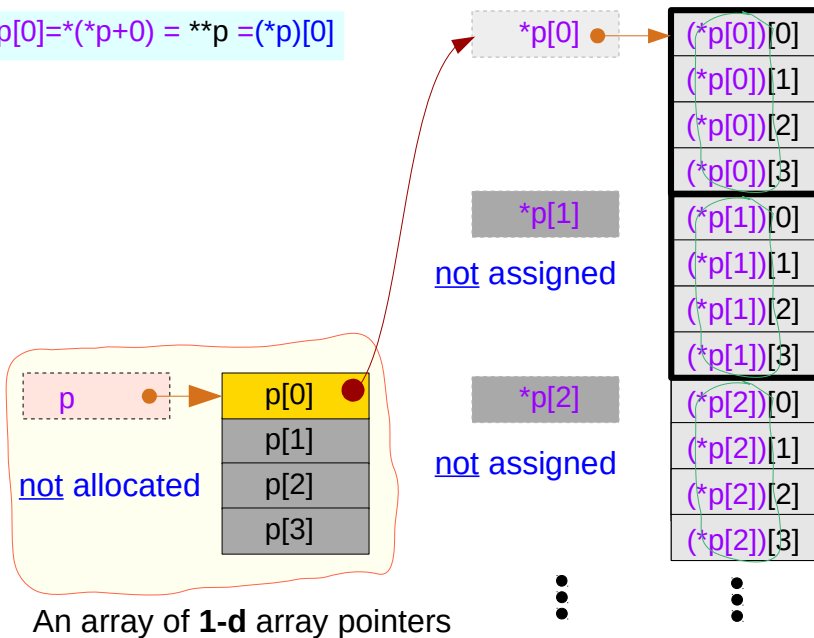
assignment	equivalence
<code>p[0]=&amp;x[0]</code>	<code>*p[0]=x[0]</code>
<code>p[1]=&amp;x[1]</code>	<code>*p[1]=x[1]</code>
<code>p[2]=&amp;x[2]</code>	<code>*p[2]=x[2]</code>
<code>p[3]=&amp;x[3]</code>	<code>*p[3]=x[3]</code>

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

assignment	equivalence
<code>p=&amp;x</code>	<code>(*p)[0]=x[0]</code>
	<code>(*p)[1]=x[1]</code>
	<code>(*p)[2]=x[2]</code>
	<code>(*p)[3]=x[3]</code>



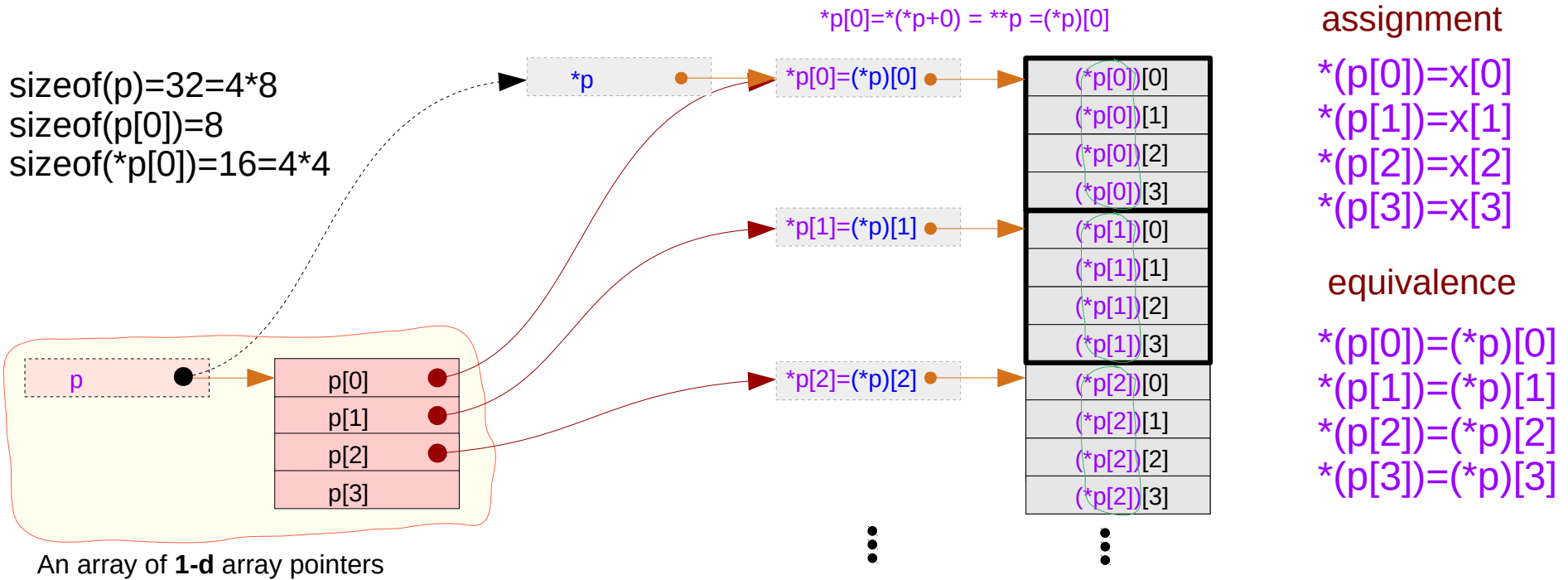
`*p[0]=*(p+0) = **p = (*p)[0]`



# int (\*p[4])[4] and accessing a 2-d array

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

`(*p[i])[j];` == `*(p[i])[j];` == `((*p)[i])[j];`

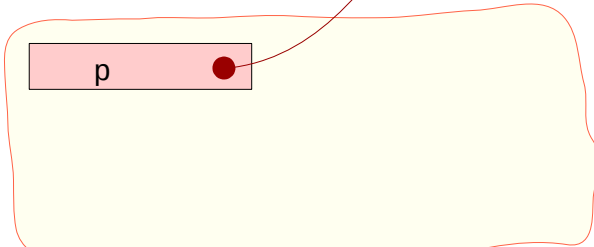


# int (\*p)[4][4] and accessing a 2-d array

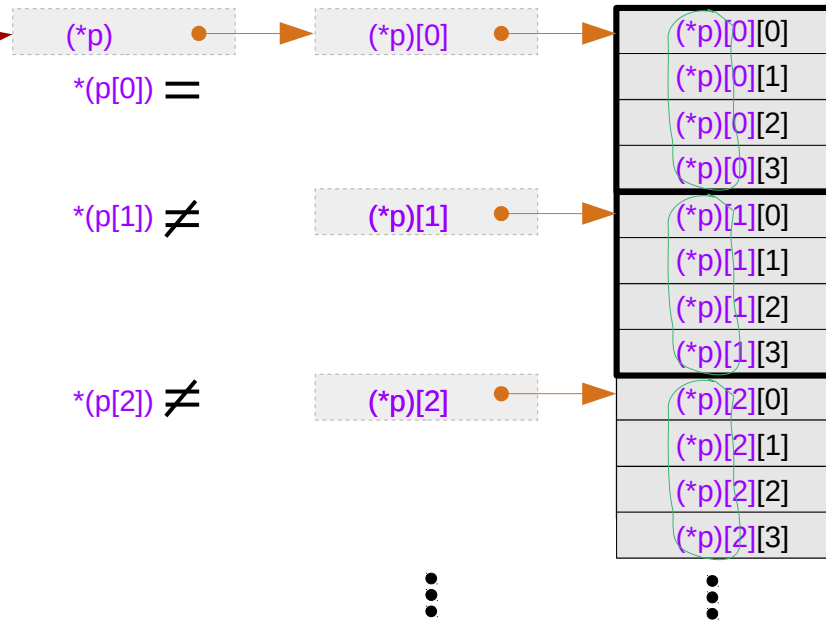
```
int (*p)[4][4]; == int ((*p)[4])[4];
```

```
(*p)[i][j]; == ((*p)[i])[j]; ≠ (*(p[i]))[j];
```

sizeof(p)=8  
sizeof(\*p)=64=4\*4\*4  
sizeof((\*p)[0])=16=4\*4



A 2-d array pointers



# int (\*p[4])[4] and equivalence relations

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

assignment

p[0]=&x[0]  
p[1]=&x[1]  
p[2]=&x[2]  
p[3]=&x[3]

equivalence

\*(p[0])=x[0]  
\*(p[1])=x[1]  
\*(p[2])=x[2]  
\*(p[3])=x[3]

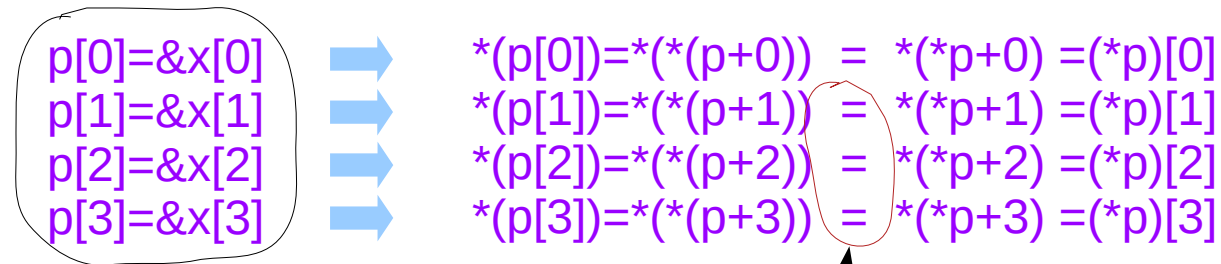
int (\*p)[4][4];

assignment

p=&x

equivalence

(\*p)[0]=x[0]  
(\*p)[1]=x[1]  
(\*p)[2]=x[2]  
(\*p)[3]=x[3]



these assignments make these equivalences

# int (\*p)[4][4] and equivalence relation

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

assignment

~~p[0]=&x[0]  
p[1]=&x[1]  
p[2]=&x[2]  
p[3]=&x[3]~~



equivalence

~~\*(p[0])=x[0]  
\*(p[1])=x[1]  
\*(p[2])=x[2]  
\*(p[3])=x[3]~~

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

assignment

p=&x



equivalence

(\*p)[0]=x[0]  
(\*p)[1]=x[1]  
(\*p)[2]=x[2]  
(\*p)[3]=x[3]

p=&x

$*(p[0])=*(*(p+0)) = *(*p+0) = (*p)[0]$   
 $*(p[1])=*(*(p+1)) \neq *(*p+1) = (*p)[1]$   
 $*(p[2])=*(*(p+2)) \neq *(*p+2) = (*p)[2]$   
 $*(p[3])=*(*(p+3)) \neq *(*p+3) = (*p)[3]$



# int (\*p[4])[4] and int (\*p)[4][4]

An array of **1-d** array pointers

int (\*p[4])[4]; = int (\*(p[4]))[4];  $\neq$  int ((\*p)[4])[4];

(\*p[i])[j]; = (\*(p[i]))[j]; = ((\*p)[i])[j];

A **2-d** array pointers

int (\*p)[4][4]; = int ((\*p)[4])[4];  $\neq$  int (\*(p[4]))[4];

(\*p)[i][j]; = ((\*p)[i])[j];  $\neq$  (\*(p[i]))[j];

# $(n-1)$ -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        ( $q = b$ )

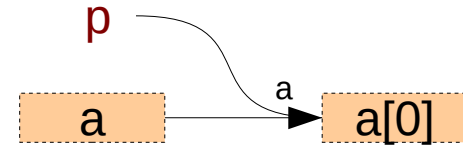
`int c[4][2][3];`                    **3-d** array  
`int (*r)[2][3];`                    **2-d** array pointer        ( $r = c$ )

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

# $n$ -d array name and $(n-1)$ -d array pointer

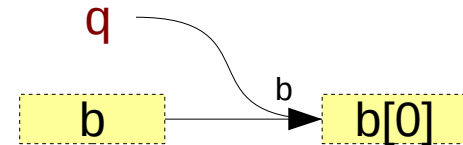
```
int a[4];  
int (*p);
```

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



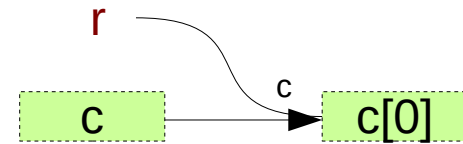
```
int b[4][2];  
int (*q)[2];
```

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



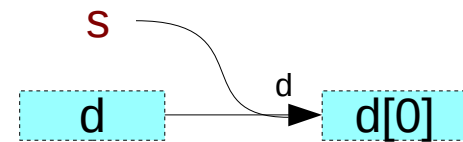
```
int c[4][2][3];  
int (*r)[2][3];
```

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



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

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



# *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`)

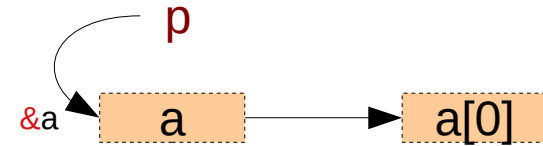
`int c [4][2][3];`            **3-d** array  
`int (*r) [4][2][3];`        **3-d** array pointer        (`r = &c`)

`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

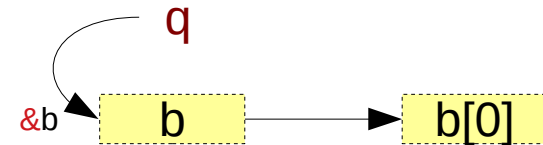
```
int a [4];  
int (*p) [4];
```

```
p = &a;
```



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

```
q = &b;
```



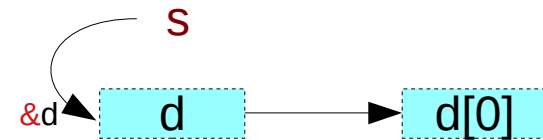
```
int c [4][2][3];  
int (*r) [4][2][3];
```

```
r = &c;
```

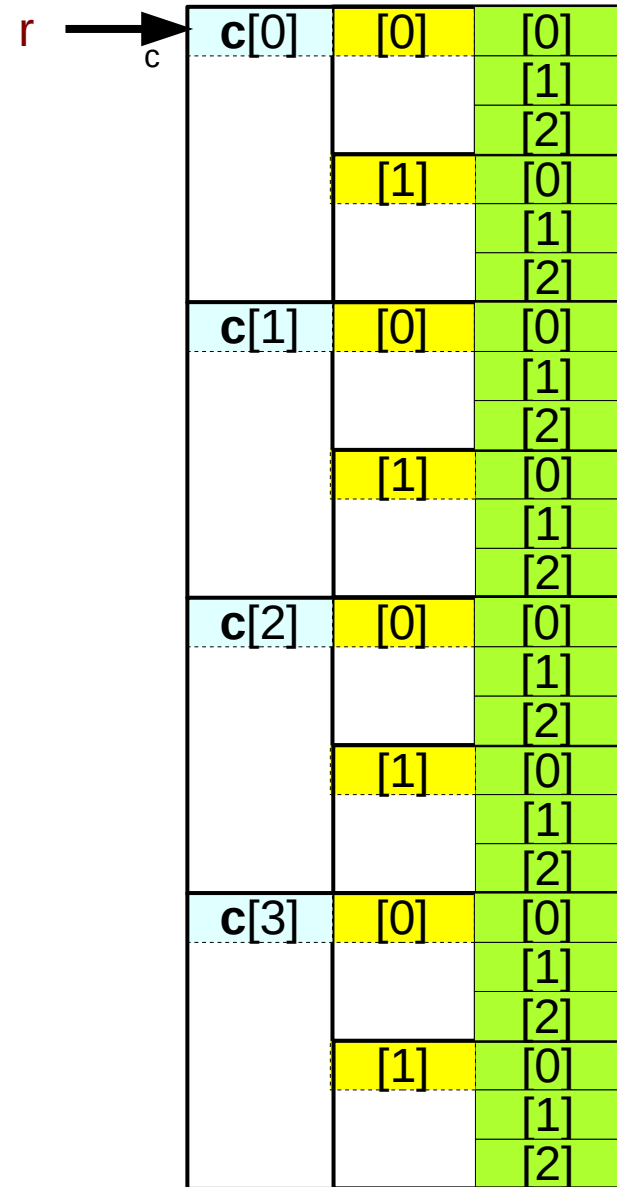
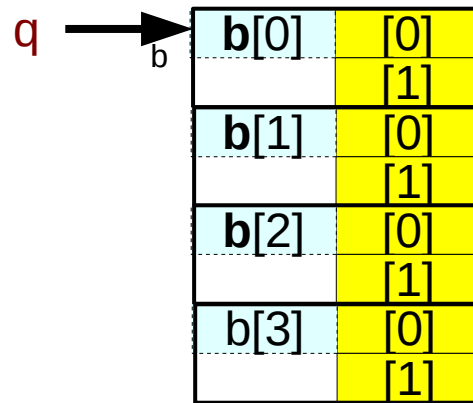
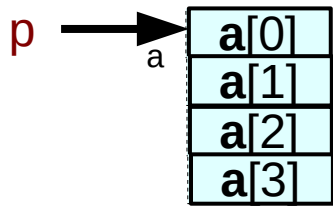


```
int d [4][2][3][4];  
int (*s) [4][2][3][4];
```

```
s = &d;
```

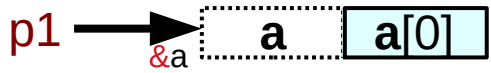


# multi-dimensional array pointers

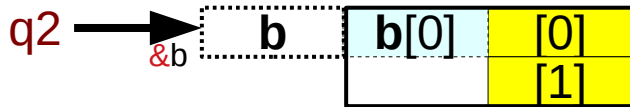


<code>int a[4];</code>	<b>1-d array</b>
<code>int (*p);</code>	<b>0-d array pointer</b>
<code>int b[4] [2];</code>	<b>2-d array</b>
<code>int (*q) [2];</code>	<b>1-d array pointer</b>
<code>int c[4] [2][3];</code>	<b>3-d array</b>
<code>int (*r) [2][3];</code>	<b>2-d array pointer</b>
<code>int d[4]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

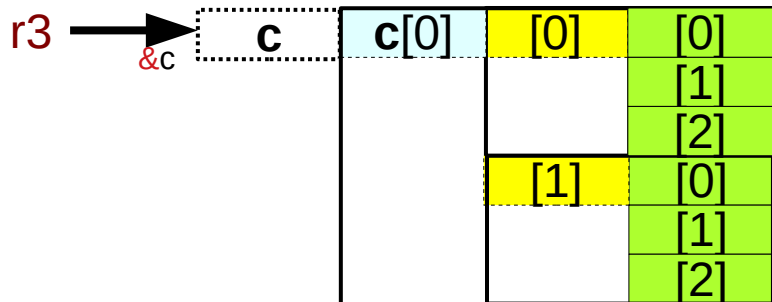
# Initializing *n-d* array pointers



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



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

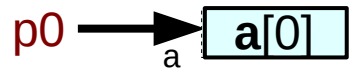


```
int c[4][2][3];
int (*r3)[4][2][3] = &c;
```

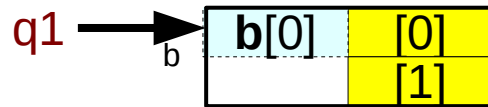


```
int d[4][2][3][4];
int (*s4)[4][2][3][4] = &d;
```

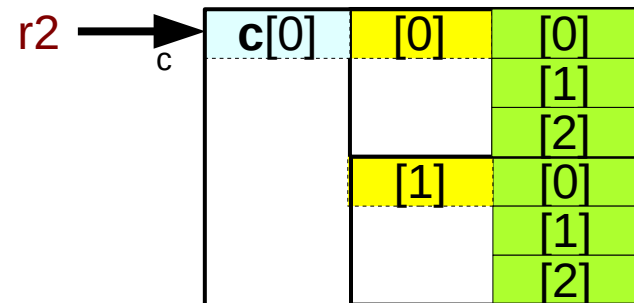
# Initializing $(n-1)$ -d array pointers



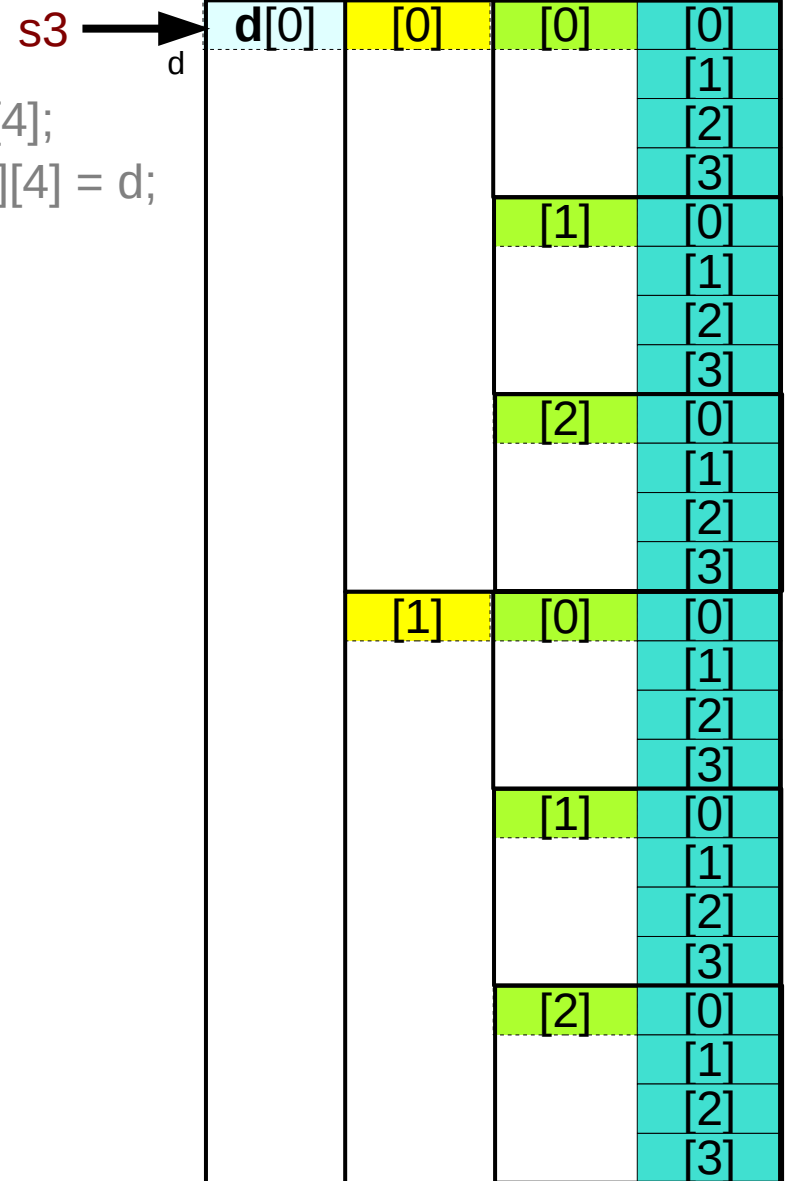
```
int a[4];
int (*p0) = a;
```



```
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;
```



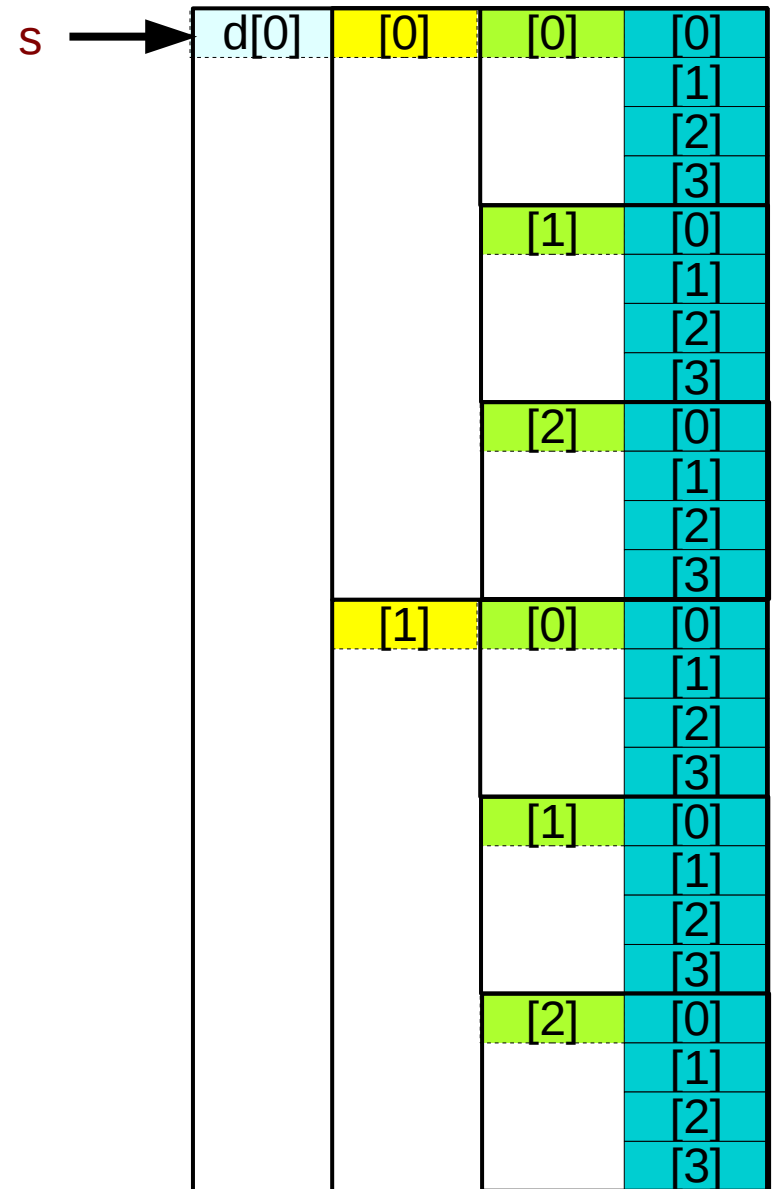
# array pointers to multi-dimensional subarrays

```
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];
```

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

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

# Passing multidimensional array names

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

call  
**fun**a(a, ...);

prototype  
void **fun**a(int (\*p), ...);

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

call  
**fun**b(b, ...);

prototype  
void **fun**b(int (\*q)[2], ...);

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

call  
**func**(c, ...);

prototype  
void **func**(int (\*r)[2][3], ...);

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

call  
**fund**(d, ...);

prototype  
void **fund**(int (\*s)[2][3][4], ...);

## References

- [1] Essential C, Nick Parlante
- [2] Efficient C Programming, Mark A. Weiss
- [3] C A Reference Manual, Samuel P. Harbison & Guy L. Steele Jr.
- [4] C Language Express, I. K. Chun