

Multi-dimensional Arrays (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);
}
```

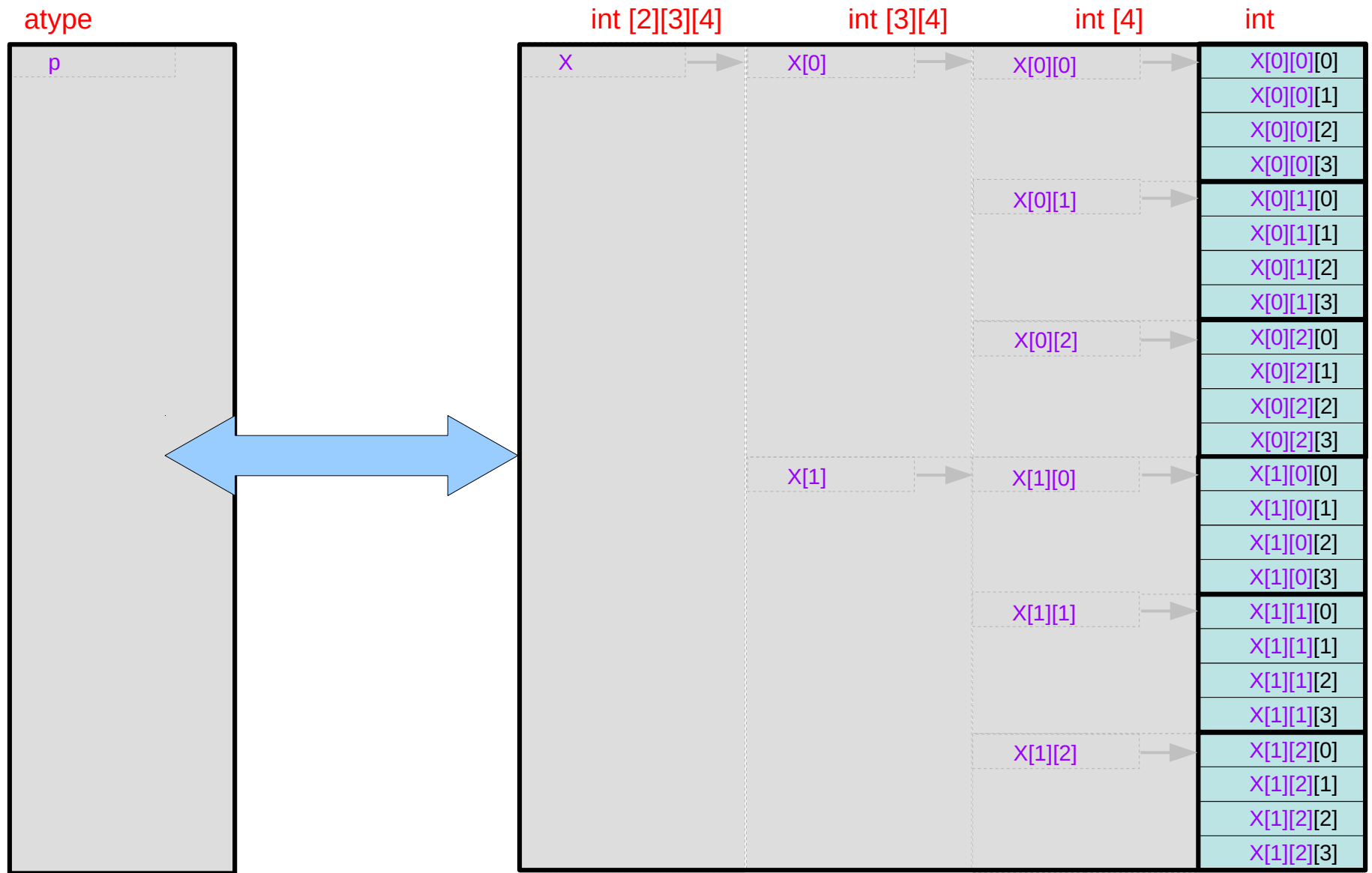
t.c: In function 'main':
t.c:5:16: warning: format '%d' expects argument of type 'int',
but argument 2 has type 'int (*)[3]' [-Wformat=]
printf ("c= %d \n", &c);

Abstract data type examples in `int p[2][3][4];`

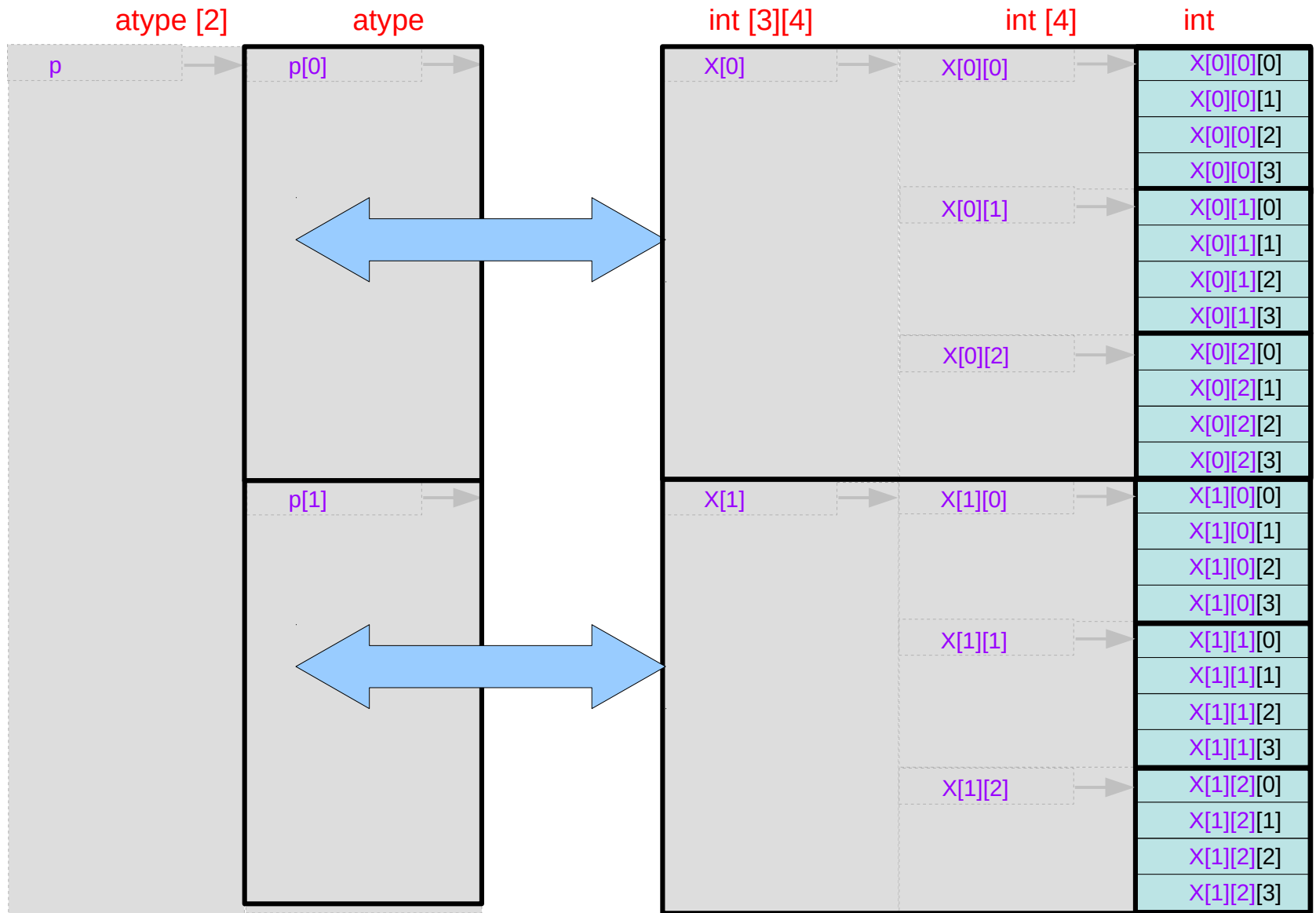
```
int p[2][3][4];
```

- (1) `atype p;` where `atype = int [2][3][4]`
- (2) `atype p[2];` where `atype = int [3][4]`
- (3) `atype p[2][3];` where `atype = int [4]`
- (4) `atype p[2][3][4];` where `atype = int`

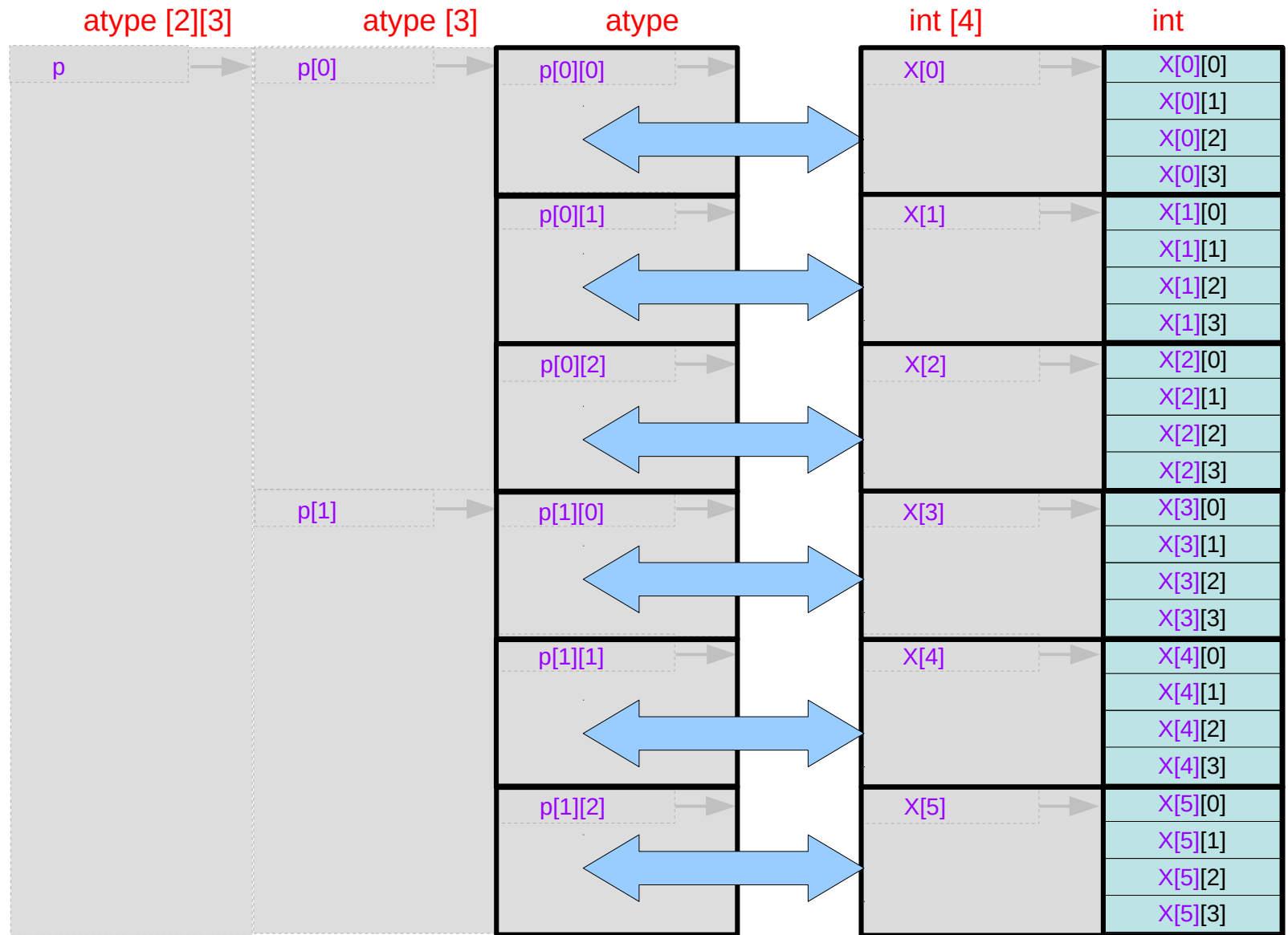
(1) `atype p;` where `atype = int [2][3][4]`



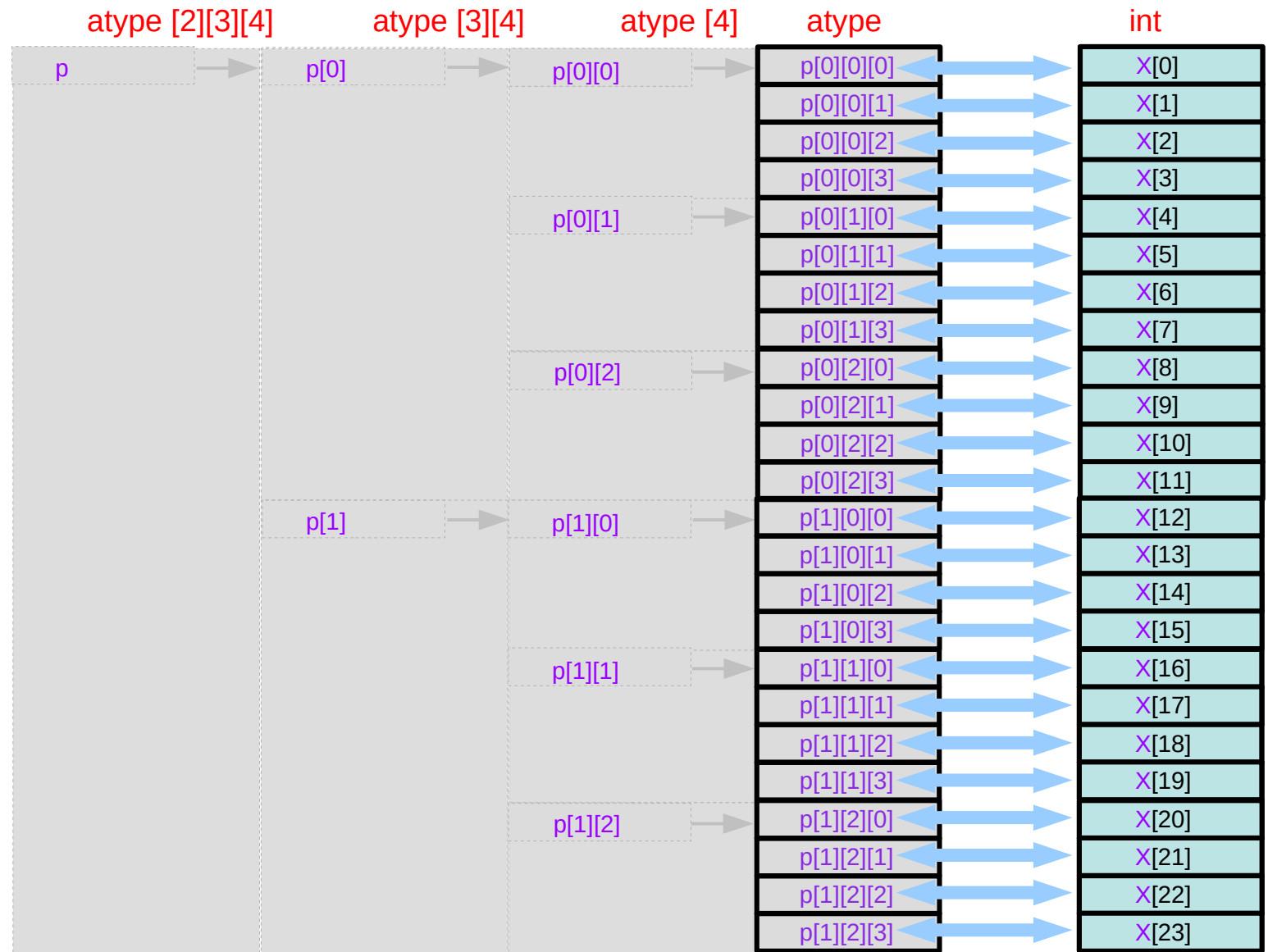
(2) `atype p[2];` where `atype = int [3][4]`



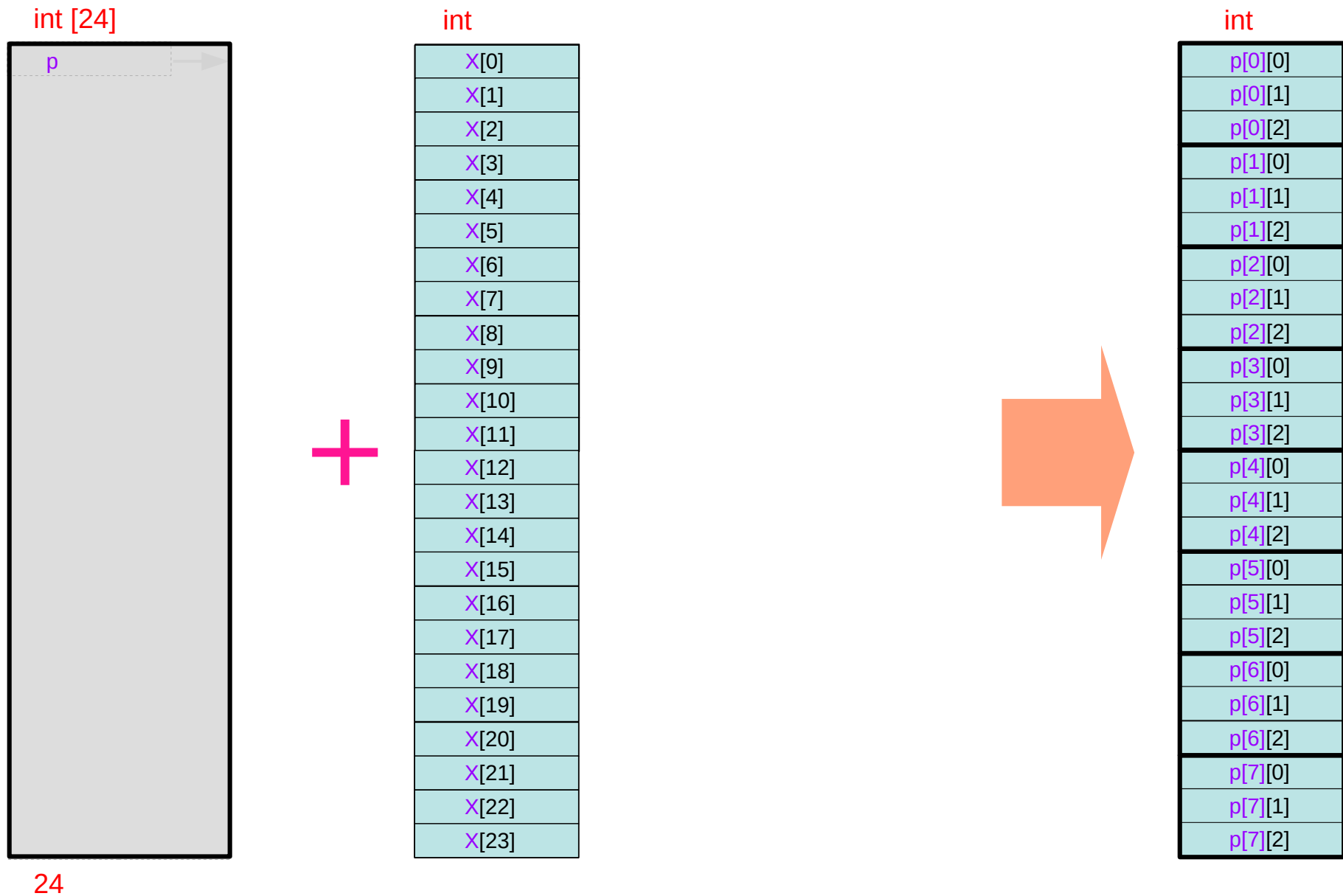
(3) `atype p[2][3];` where `atype = int [4]`



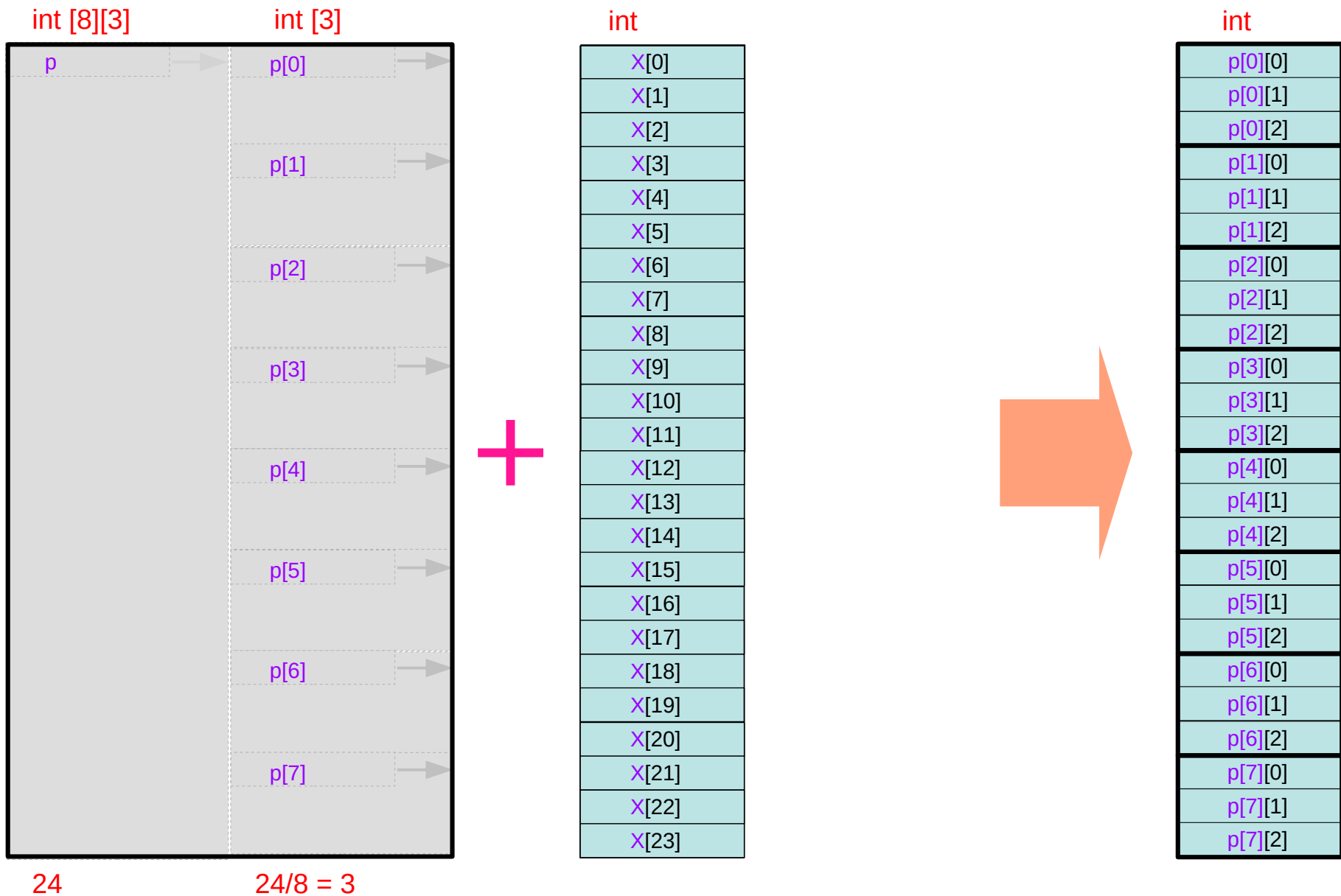
(4) `atype p[2][3][4];` where `atype = int`



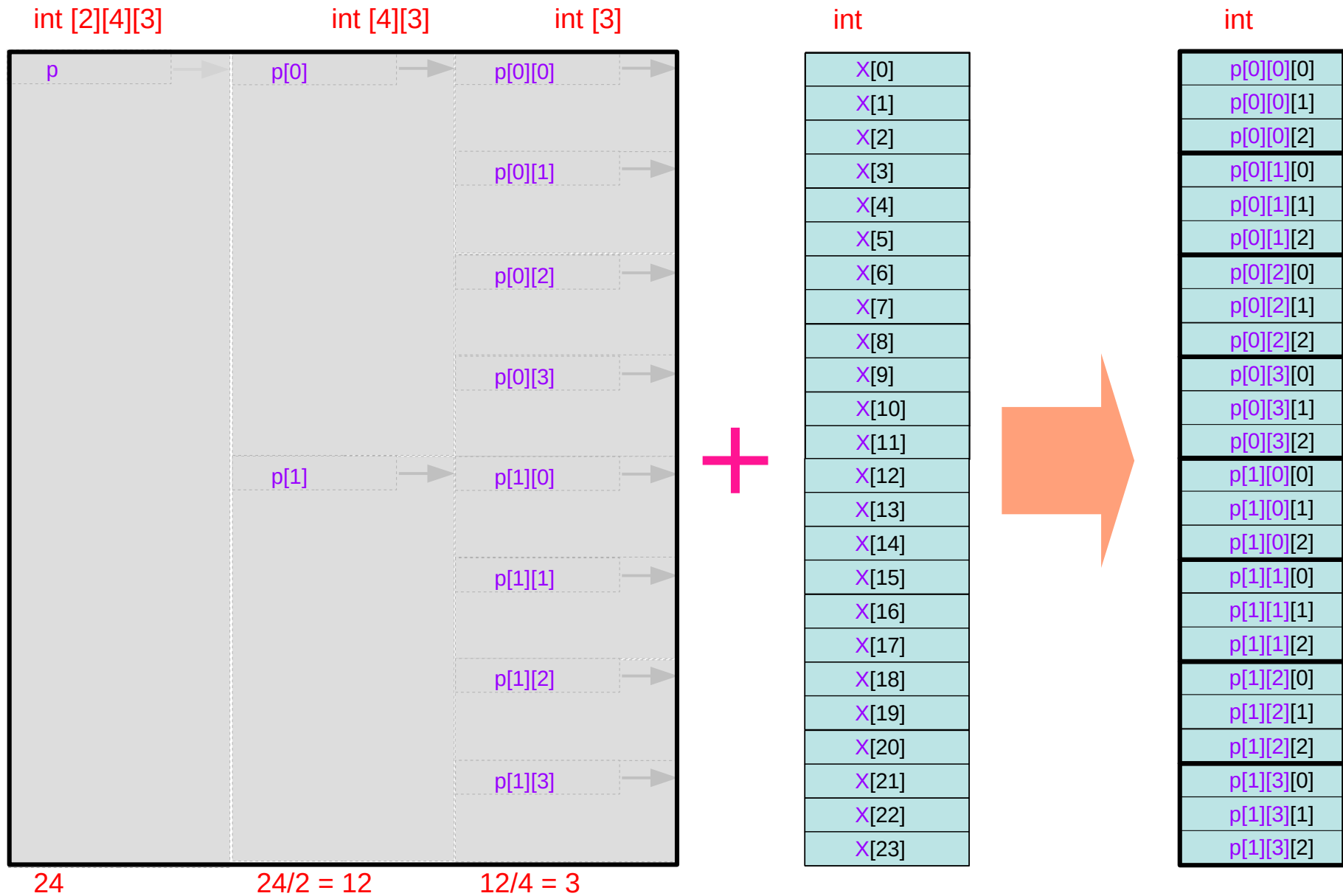
Indexing in int p[24]



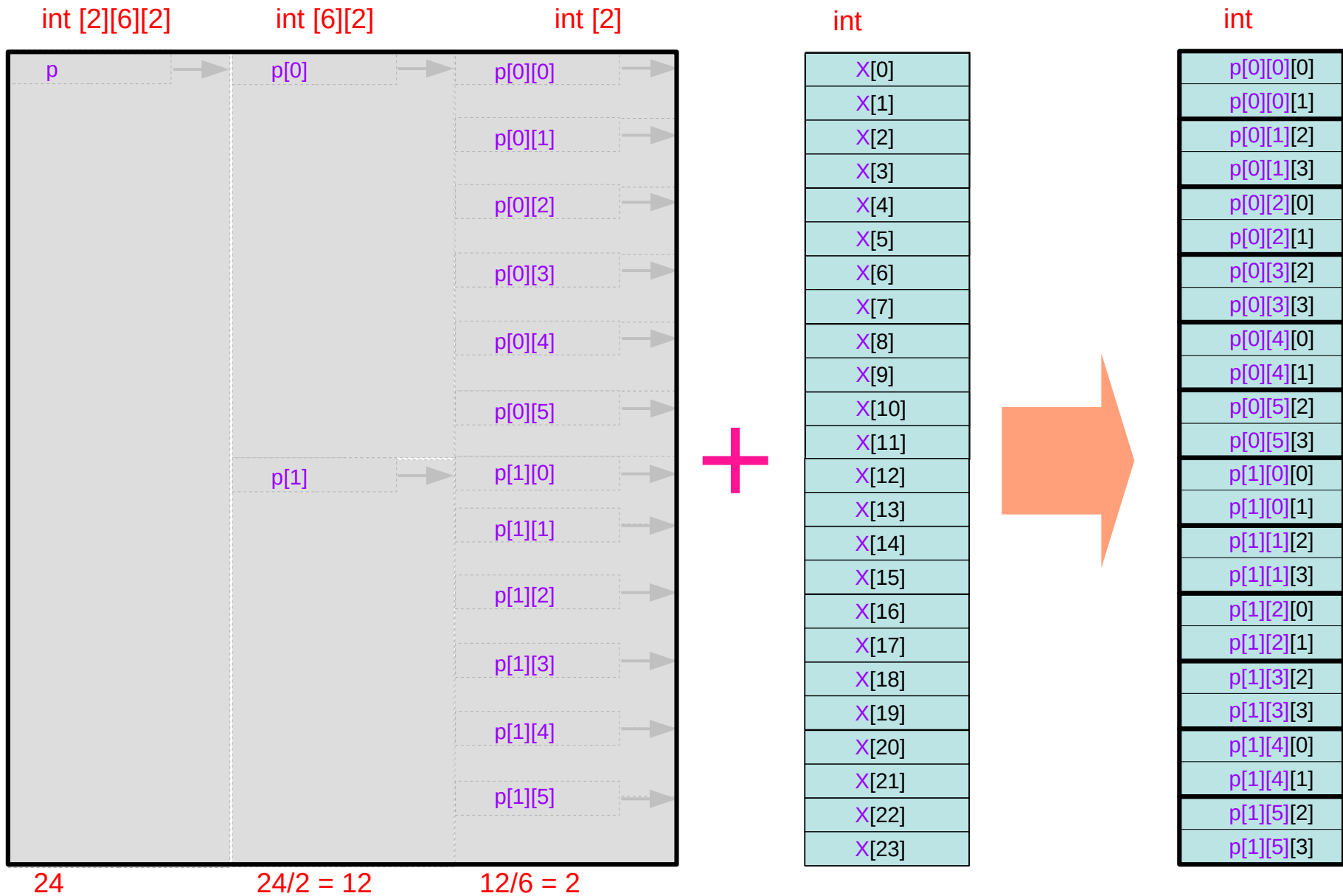
Indexing in int p[8][3]



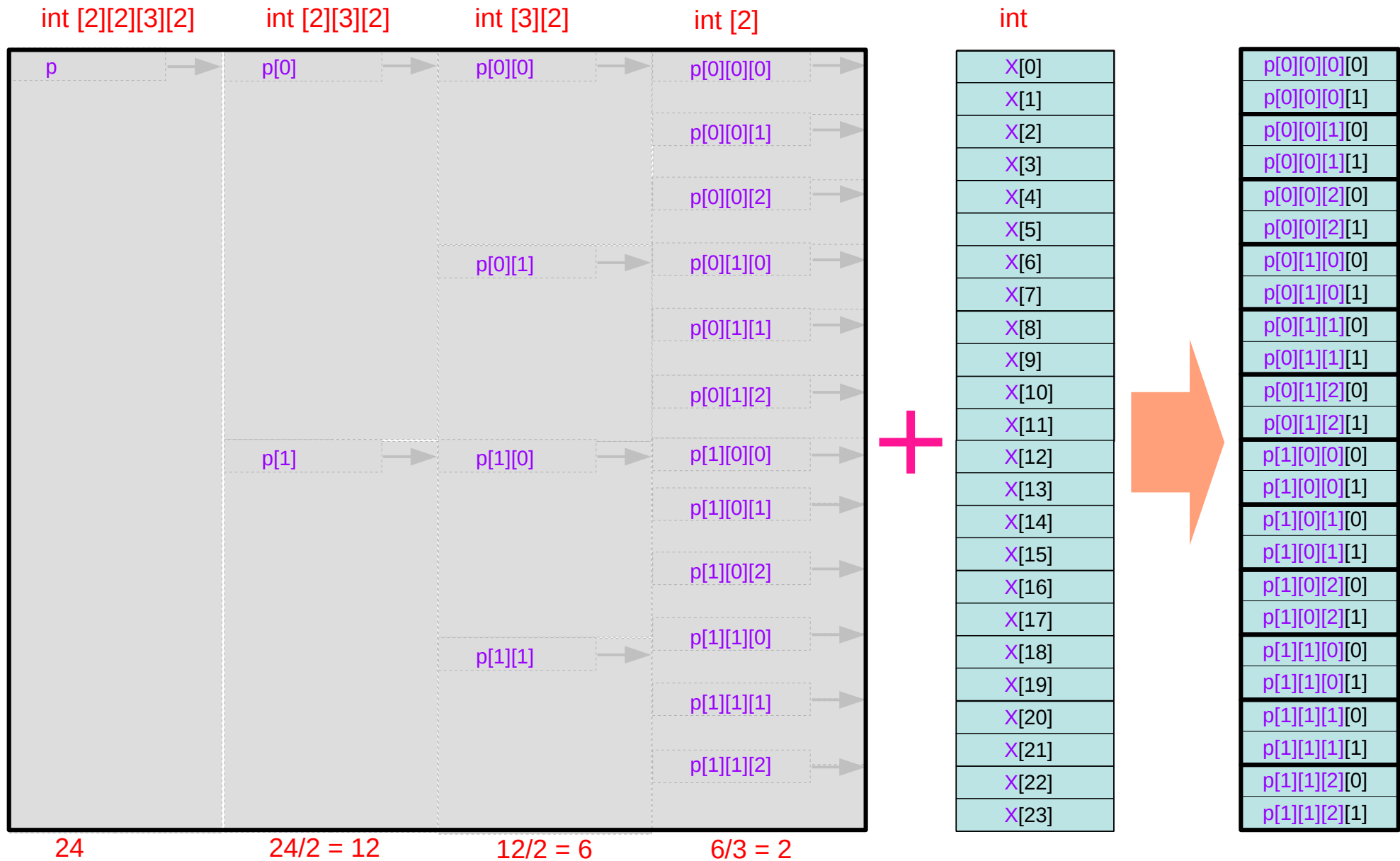
Indexing in int p[2][4][3]



Indexing in int p[2][6][2]



Indexing in int p[2][2][3][2]



Indexing in multi-dimension arrays

int [24]

X[0]
X[1]
X[2]
X[3]
X[4]
X[5]
X[6]
X[7]
X[8]
X[9]
X[10]
X[11]
X[12]
X[13]
X[14]
X[15]
X[16]
X[17]
X[18]
X[19]
X[20]
X[21]
X[22]
X[23]

1-d array

int p[8][3]

p[0][0]
p[0][1]
p[0][2]
p[1][0]
p[1][1]
p[1][2]
p[2][0]
p[2][1]
p[2][2]
p[3][0]
p[3][1]
p[3][2]
p[4][0]
p[4][1]
p[4][2]
p[5][0]
p[5][1]
p[5][2]
p[6][0]
p[6][1]
p[6][2]
p[7][0]
p[7][1]
p[7][2]

2-d array

int p[2][4][3]

p[0][0][0]
p[0][0][1]
p[0][0][2]
p[0][1][0]
p[0][1][1]
p[0][1][2]
p[0][2][0]
p[0][2][1]
p[0][2][2]
p[0][3][0]
p[0][3][1]
p[0][3][2]
p[1][0][0]
p[1][0][1]
p[1][0][2]
p[1][1][0]
p[1][1][1]
p[1][1][2]
p[1][2][0]
p[1][2][1]
p[1][2][2]
p[1][3][0]
p[1][3][1]
p[1][3][2]

3-d array

int p[2][6][2]

p[0][0][0]
p[0][0][1]
p[0][1][2]
p[0][1][3]
p[0][2][0]
p[0][2][1]
p[0][3][2]
p[0][3][3]
p[0][4][0]
p[0][4][1]
p[0][5][2]
p[0][5][3]
p[1][0][0]
p[1][0][1]
p[1][1][2]
p[1][1][3]
p[1][2][0]
p[1][2][1]
p[1][3][2]
p[1][3][3]
p[1][4][0]
p[1][4][1]
p[1][5][2]
p[1][5][3]

3-d array

int p[2][2][3][2]

p[0][0][0][0]
p[0][0][0][1]
p[0][0][1][0]
p[0][0][1][1]
p[0][0][2][0]
p[0][0][2][1]
p[0][1][0][0]
p[0][1][0][1]
p[0][1][1][0]
p[0][1][1][1]
p[0][1][2][0]
p[0][1][2][1]
p[1][0][0][0]
p[1][0][0][1]
p[1][0][1][0]
p[1][0][1][1]
p[1][0][2][0]
p[1][0][2][1]
p[1][1][1][0]
p[1][1][1][1]
p[1][1][2][0]
p[1][1][2][1]

4-d array

Subarray sizes of 24 element multi-dimensional arrays (1)

`int p[2][2][3][2]`

`p`
`sizeof(p)` `int [2][2][3][2]`
= 2*2*3*2 =24
integers

`p[0]`
`sizeof(p[0])` `int [2][3][2]`
= 2*3*2 =12
integers

`p[0][0]`
`sizeof(p[0][0])` `int [3][2]`
= 3*2 =6
integers

`p[0][0][0]`
`sizeof(p[0][0][0])` `int [2]`
= 2
integers

`p[0][0][0][0]`
`sizeof(p[0][0][0][0])` `int`
= 1
integers

`int p[2][6][2]`

`p`
`sizeof(p)` `int [2][6][2]`
= 2*6*2 =24
integers

`p[0]`
`sizeof(p[0])` `int [6][2]`
= 6*2 =12
integers

`p[0][0]`
`sizeof(p[0][0])` `int [2]`
= 2
integers

`p[0][0][0]`
`sizeof(p[0][0][0])` `int`
= 1
integers

`int p[2][4][3]`

`p`
`sizeof(p)` `int [2][4][3]`
= 2*4*3 =24
integers

`p[0]`
`sizeof(p[0])` `int [4][3]`
= 4*3 =12
integers

`p[0][0]`
`sizeof(p[0][0])` `int [3]`
= 3
integers

`p[0][0][0]`
`sizeof(p[0][0][0])` `int`
= 1
integers

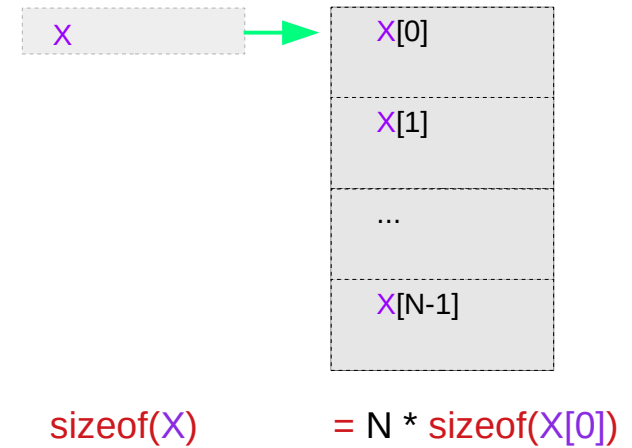
Subarray sizes of 24 element multi-dimensional arrays (1)

```
int p[8][3]
```

<code>p</code>	<code>int [8][3]</code>
<code>sizeof(p)</code>	= 8*3 =24 integers
<code>p[0]</code>	<code>int [3]</code>
<code>sizeof(p[0])</code>	= 3 integers
<code>p[0][0]</code>	<code>int</code>
<code>sizeof(p[0][0])</code>	= 1 integers

```
int p[24]
```

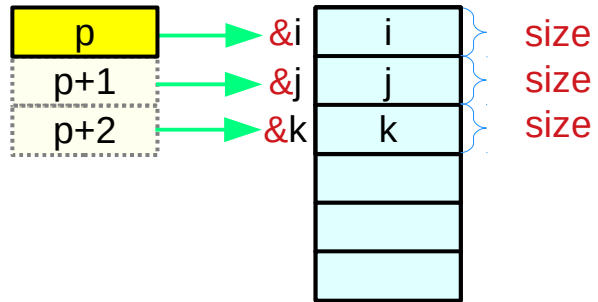
<code>p</code>	<code>int [24]</code>
<code>sizeof(p)</code>	= 24 integers
<code>p[0]</code>	<code>int</code>
<code>sizeof(p[0])</code>	= 1 integers



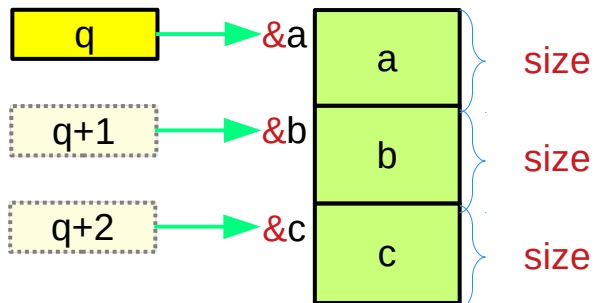
Virtual array pointers in a multi-dimensional array

Pointers to various data types

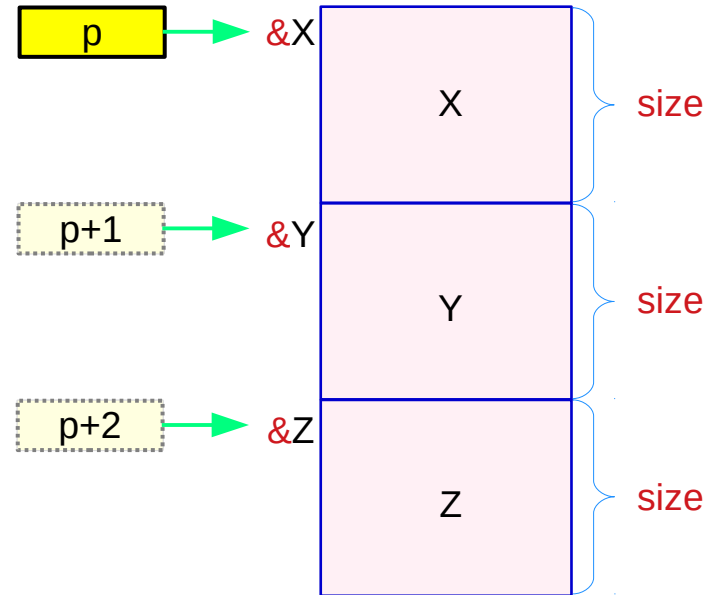
int *p; **int i, j, k;**



double *q; **double a, b, c;**



T *p; **T X, Y, Z;**



pointer

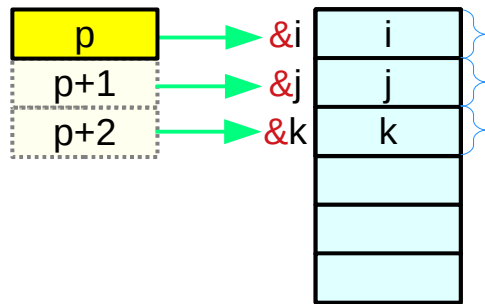
abstract data

Pointers to primitive data

int *p;

int i, j, k;

sizeof(int) = 4 bytes



size
size
size

$\text{size} = \text{sizeof}(i) = \text{sizeof}(*p)$
 $\text{size} = \text{sizeof}(j) = \text{sizeof}(*(p+1))$
 $\text{size} = \text{sizeof}(k) = \text{sizeof}(*(p+2))$

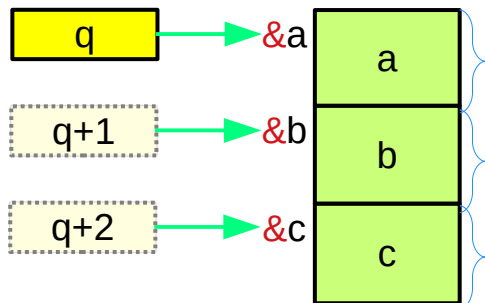
$\neq \text{sizeof}(p)$
 $\neq \text{sizeof}(p+1)$
 $\neq \text{sizeof}(p+2)$

pointer size
4 or 8 bytes

double *q;

double a, b, c;

sizeof(double) = 8 bytes



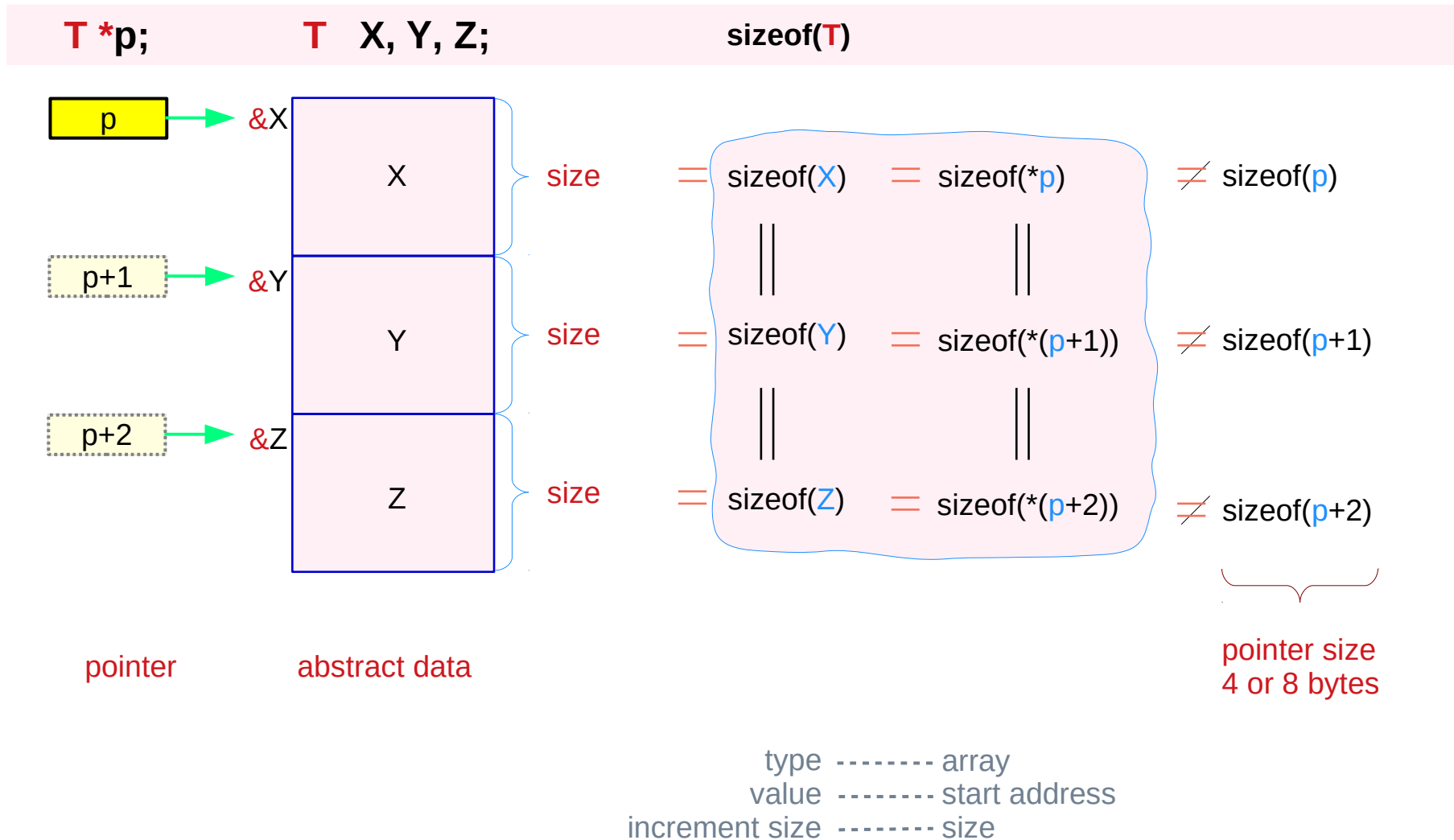
size
size
size

$\text{size} = \text{sizeof}(a) = \text{sizeof}(*q)$
 $\text{size} = \text{sizeof}(b) = \text{sizeof}(*(q+1))$
 $\text{size} = \text{sizeof}(c) = \text{sizeof}(*(q+2))$

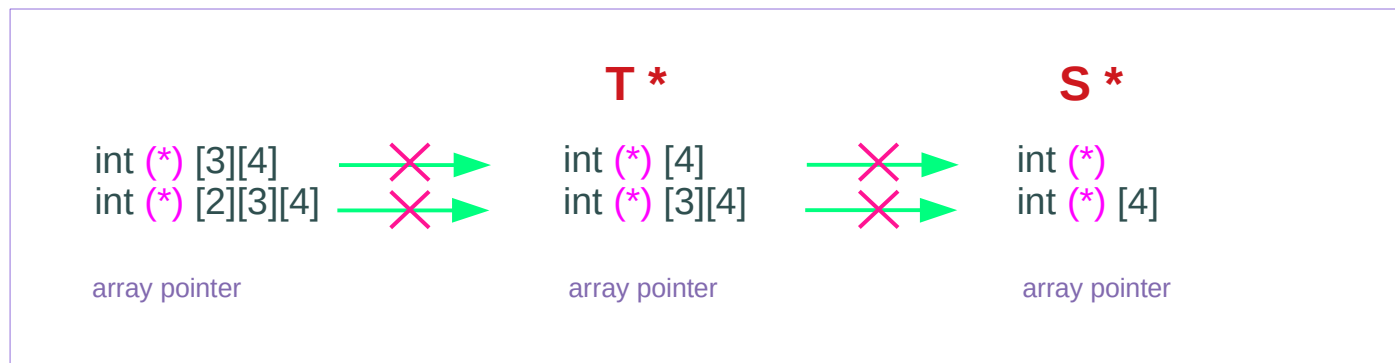
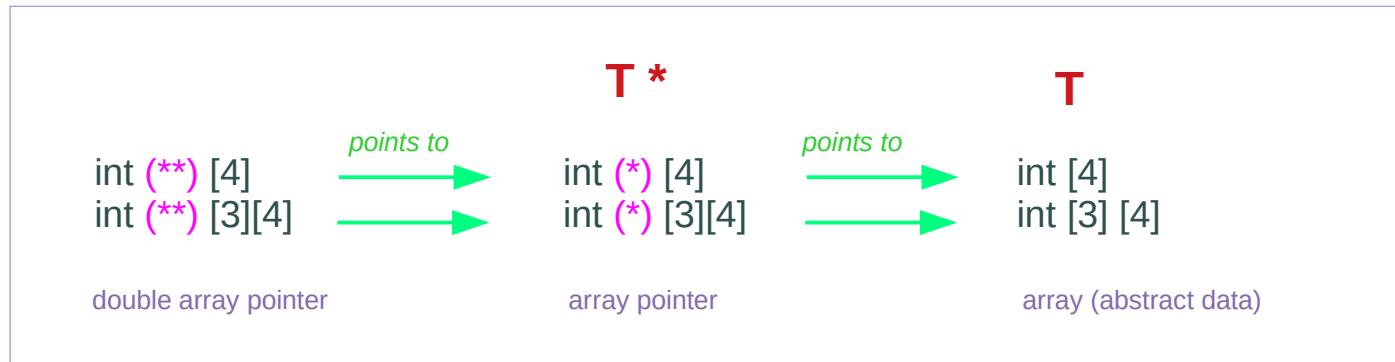
$\neq \text{sizeof}(q)$
 $\neq \text{sizeof}(q+1)$
 $\neq \text{sizeof}(q+2)$

pointer size
4 or 8 bytes

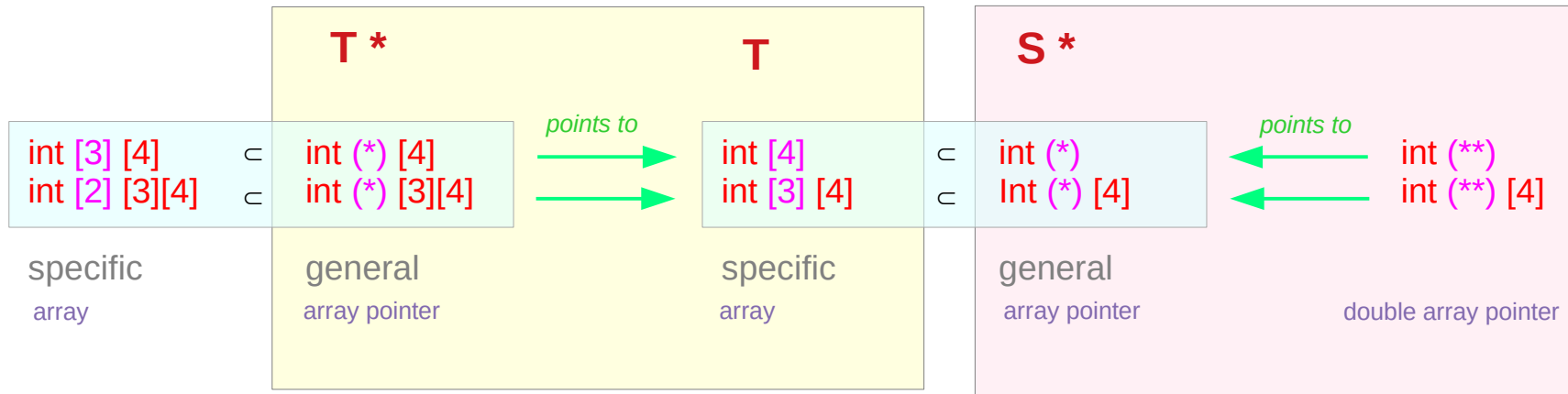
Pointers to abstract data



Array pointer types v.s. array types



General array pointer types v.s. specific array types



Array pointers have augmented dimensions

```
typedef int (*T1) [4];  
typedef int (*T1) [3][4];
```

int (*) [4]
int (*) [3][4]
general

```
typedef int T2[4];  
typedef int T2[3][4];
```

int [4]
int [3] [4]
specific

```
T1 a;  
T2 b;
```

T1 is a pointer type
T2 is an array type
T1 has one more dimension than T2

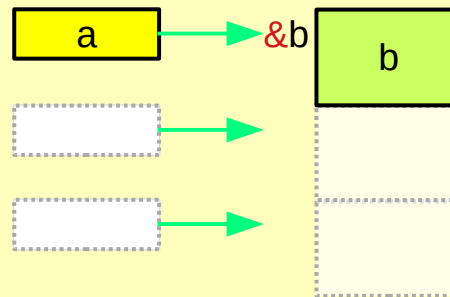
```
a = &b;  
*a = b;
```

a references b

b is the dereference of a

```
(a+1) = ?  
*(a+1) = ?
```

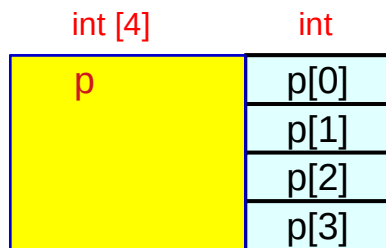
```
(a+2) = ?  
*(a+2) = ?
```



Dual types in an array of integers

```
int p[3];
```

p is an abstract data (array)

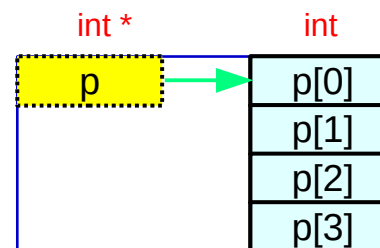


p is the name of an array

p has the size of the whole array

p has an array type (abstract data)

p can also be viewed as a pointer



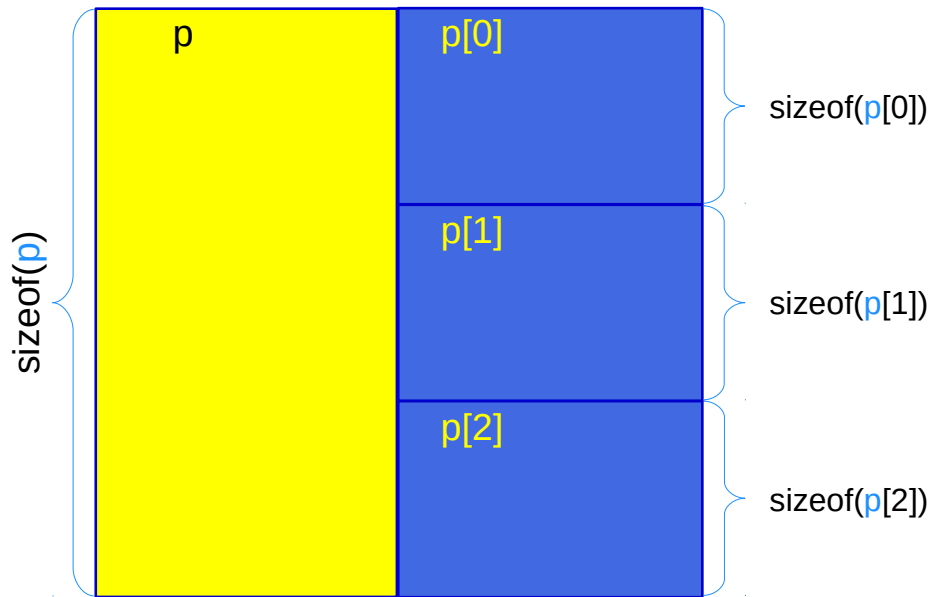
p also has pointer characteristics

p has the value of the starting address

p is a virtual pointer

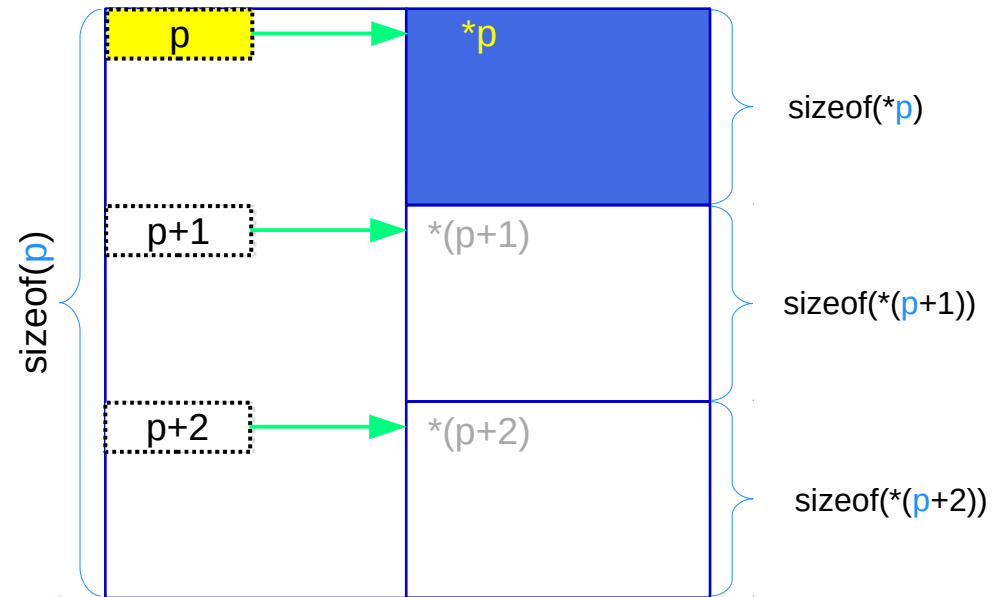
Dual types in an array of abstract data

Abstract data array p



- p has an array type (abstract data element)
- p is the name of an array
- p has the size of the whole array

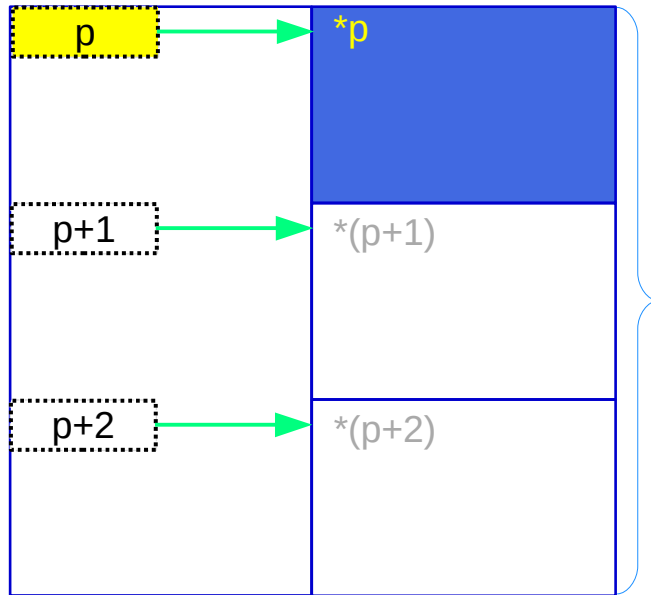
Virtual pointer p



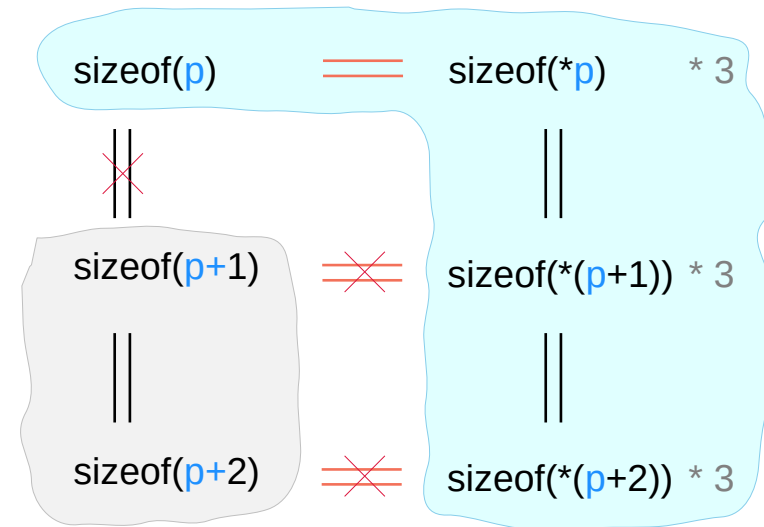
- p also has a pointer type
- p has the value of the starting address
- p is a virtual array pointer

Virtual pointer to abstract data

virtual pointer p abstract data $*p$



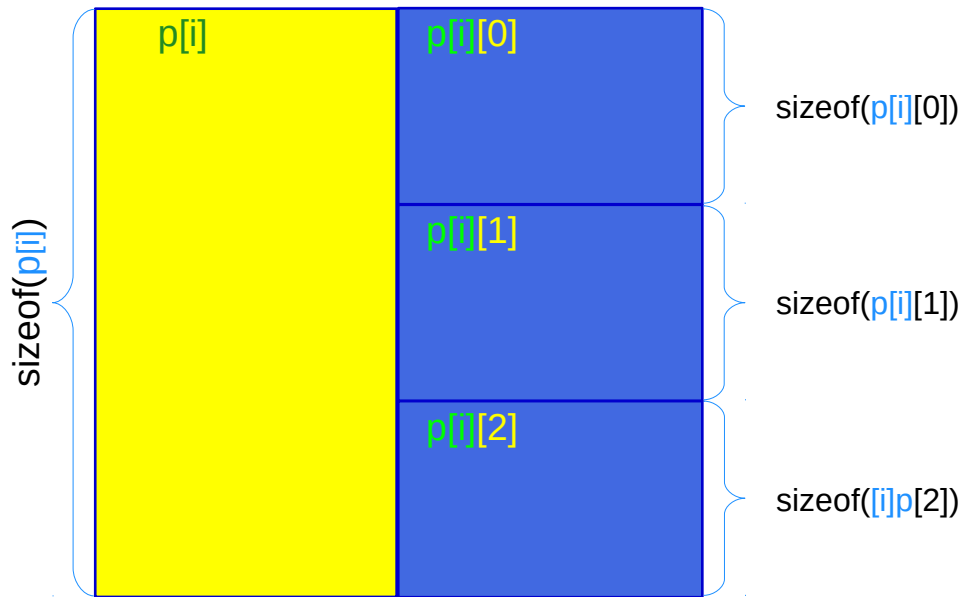
whole array size



pointer size
4 / 8 bytes

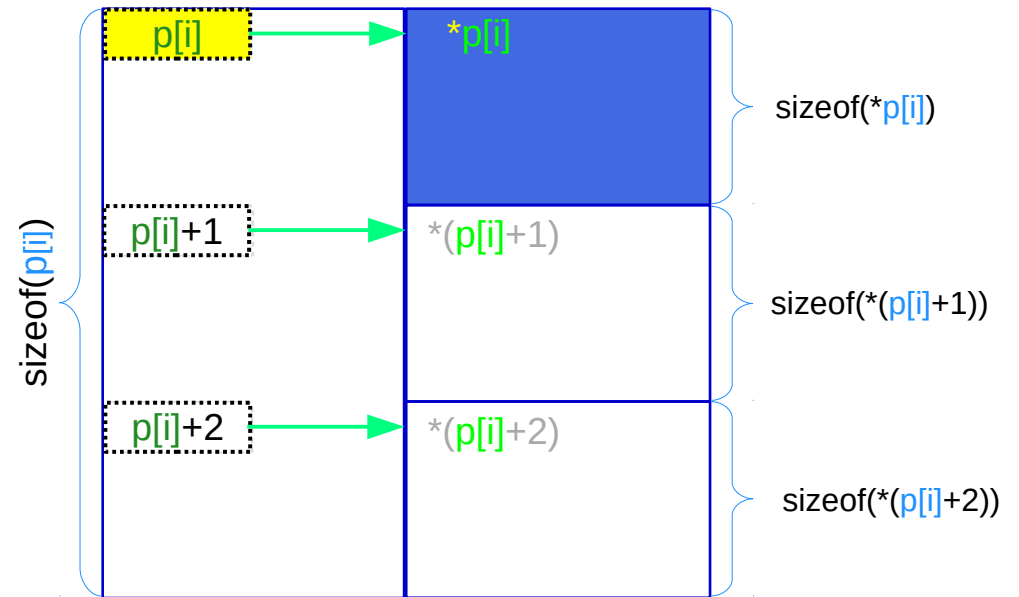
Dual types in a multi-dimensional array

Abstract data (array) $p[i]$



$p[i]$ has an array type (abstract data)
 $p[i]$ is the name of an array
 $p[i]$ has the size of the whole array

Virtual array pointer $p[i]$



$p[i]$ also has an array pointer type
 $p[i]$ has the value of the starting address
 $p[i]$ is a virtual array pointer

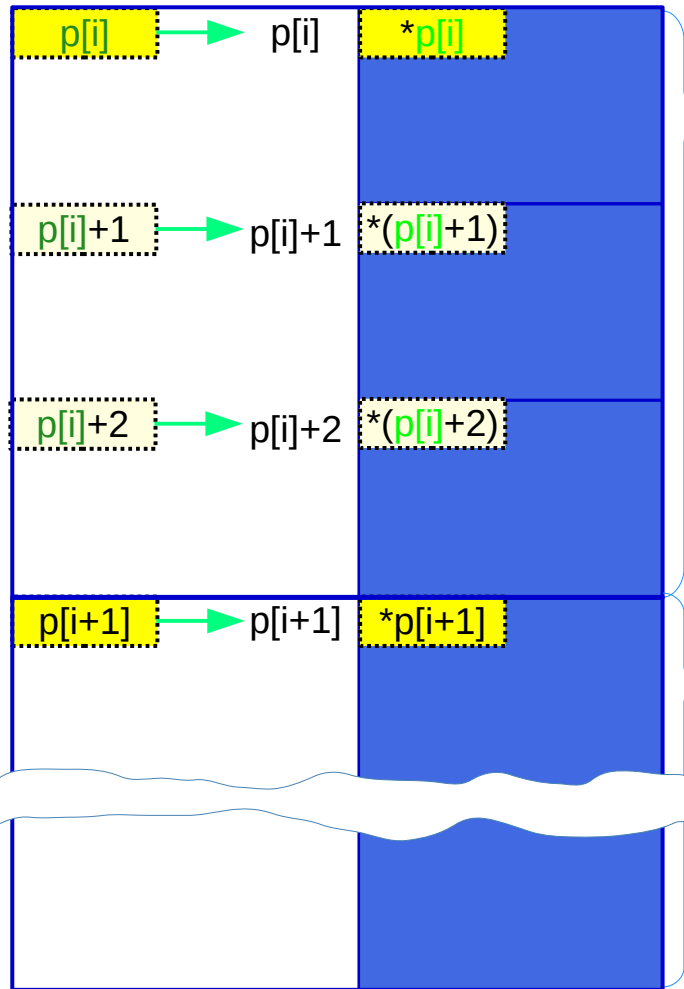
Virtual pointers to sub-arrays

$p[i] :: T^*$

array pointer type

$*p[i], *p[i+1] :: T$

array type



$\text{sizeof}(p[i])$

$$= \text{sizeof}(*p[i]) * N$$

$$= \text{sizeof}(p[i][0]) * N$$

size = $\text{sizeof}(*p[i]) = \text{sizeof}(p[i][0])$

size = $\text{sizeof}(*(p[i]+1)) = \text{sizeof}(p[i][1])$

size = $\text{sizeof}(*(p[i]+2)) = \text{sizeof}(p[i][2])$

$\text{sizeof}(p[i+1])$

$$= \text{sizeof}(*p[i+1]) * N$$

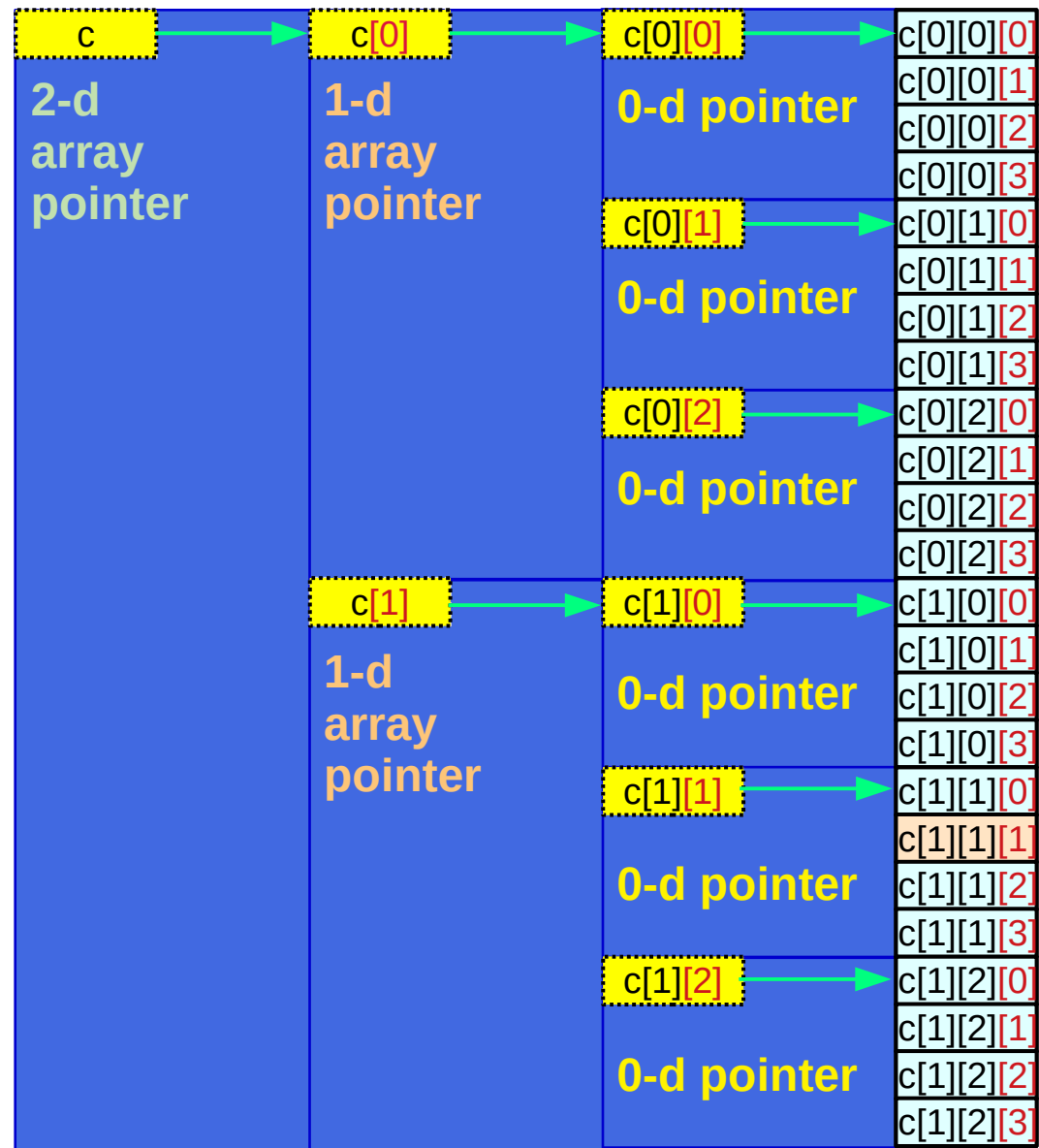
$$= \text{sizeof}(p[i+1][0]) * N$$

3-d array structure – virtual pointer representation

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

```
*(***(c +i) +j) +k)
```

- Hierarchical
- Nested Structure
- Virtual Array Pointers to abstract data (subarrays)
- Contiguous and Linear Data Layout
- Row Major Order



3-d array structure – abstract data representation

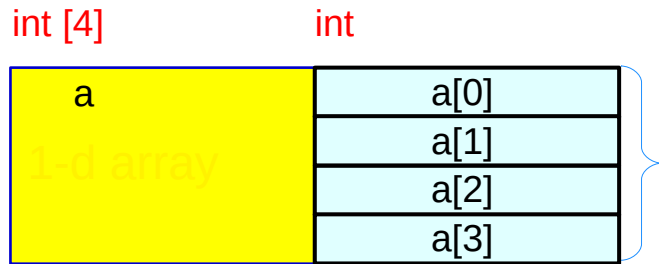
```
int c[2][3][4];
```

```
((c [i])[j])[k]
```

- Hierarchical
- Nested Structure
- Virtual Array Pointers to abstract data (subarrays)
- Contiguous and Linear Data Layout
- Row Major Order

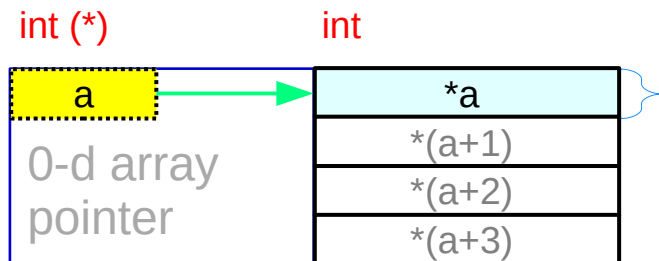
c 3-d array name	c[0] 2-d array name	c[0][0] 1-d array name	c[0][0][0] c[0][0][1] c[0][0][2] c[0][0][3]
		c[0][1] 1-d array name	c[0][1][0] c[0][1][1] c[0][1][2] c[0][1][3]
		c[0][2] 1-d array name	c[0][2][0] c[0][2][1] c[0][2][2] c[0][2][3]
	c[1] 2-d array name	c[1][0] 1-d array name	c[1][0][0] c[1][0][1] c[1][0][2] c[1][0][3]
		c[1][1] 1-d array name	c[1][1][0] c[1][1][1] c[1][1][2] c[1][1][3]
		c[1][2] 1-d array name	c[1][2][0] c[1][2][1] c[1][2][2] c[1][2][3]

Array **a** and virtual pointer **a**



1-d array **a** specific array type

$\text{sizeof}(a)$



pointer **a** general pointer type

$\text{sizeof}(a) = \text{sizeof}(*a) * 4$

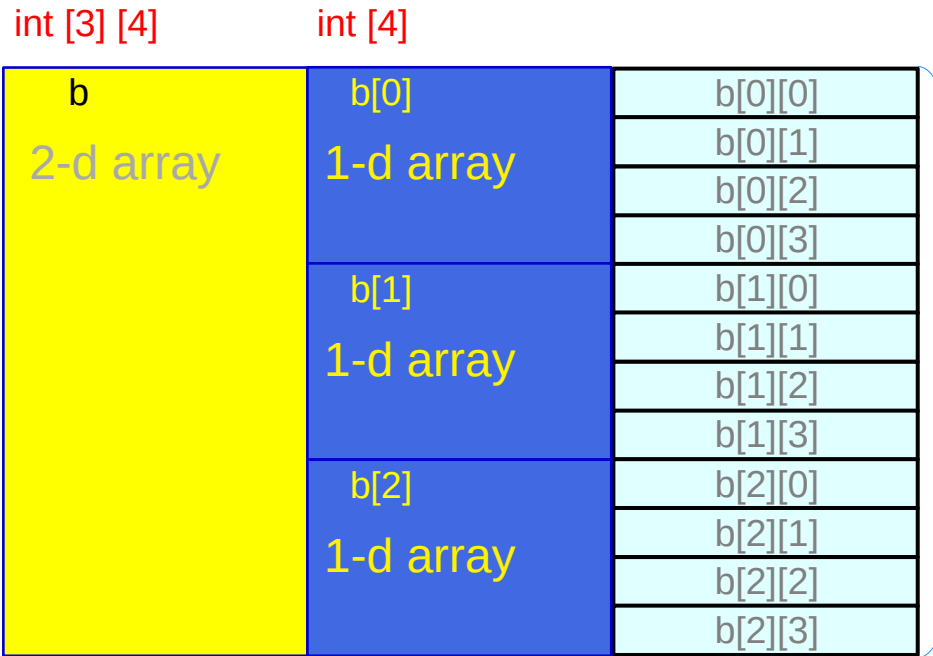
- a** is the name of a 1-d array
- a** also has a pointer type
- a** has the size of the array
- a** has the value of the starting address

a is a virtual array pointer

Array **b** and virtual pointer **b**

2-d array **b** specific array type

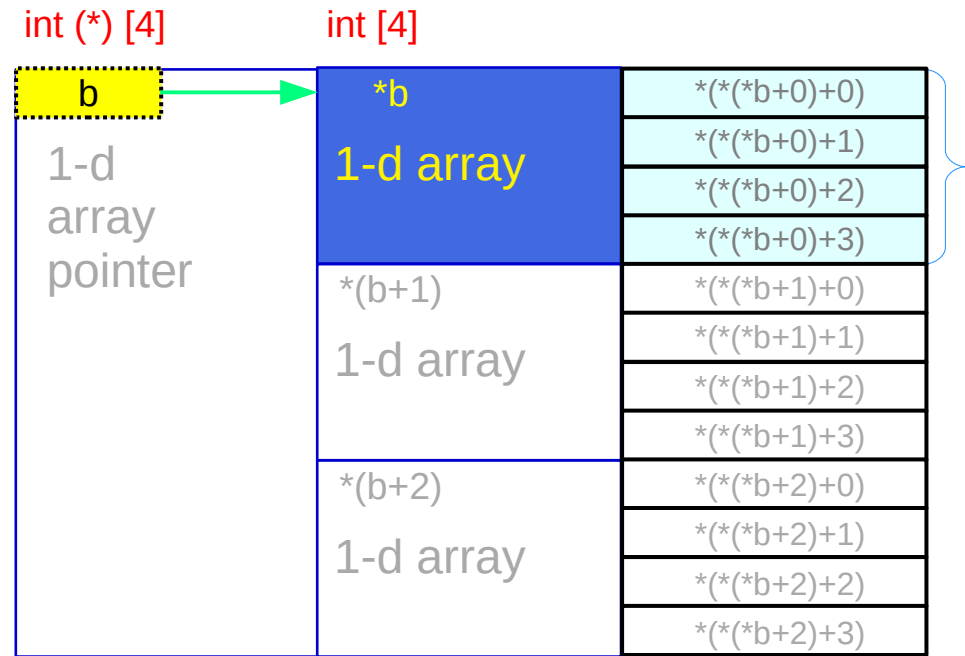
`sizeof(b)`



b is the name of a 2-d array
b has the size of the array

1-d array pointer **b** general pointer type

`sizeof(b) = sizeof(*b) * 3`



b also has a 1-d array pointer type
b has the value of the starting address

b is a virtual array pointer

Array c

3-d array **c**

specific array type

sizeof(c)

c is the name of a 3-d array
c has the size of the array

int [2][3][4]	int [3][4]	int [4]		
c 3-d array	c[0] 2-d array	c[0][0] 1-d array	c[0][0][0]	
			c[0][0][1]	
			c[0][0][2]	
				c[0][0][3]
	c[0][1] 1-d array		c[0][1][0]	
			c[0][1][1]	
			c[0][1][2]	
				c[0][1][3]
	c[0][2] 1-d array		c[0][2][0]	
			c[0][2][1]	
			c[0][2][2]	
				c[0][2][3]
c[1] 2-d array	c[1][0] 1-d array		c[1][0][0]	
			c[1][0][1]	
			c[1][0][2]	
			c[1][0][3]	
c[1][1] 1-d array		c[1][1][0]		
		c[1][1][1]		
		c[1][1][2]		
			c[1][1][3]	
c[1][2] 1-d array		c[1][2][0]		
		c[1][2][1]		
		c[1][2][2]		
			c[1][2][3]	

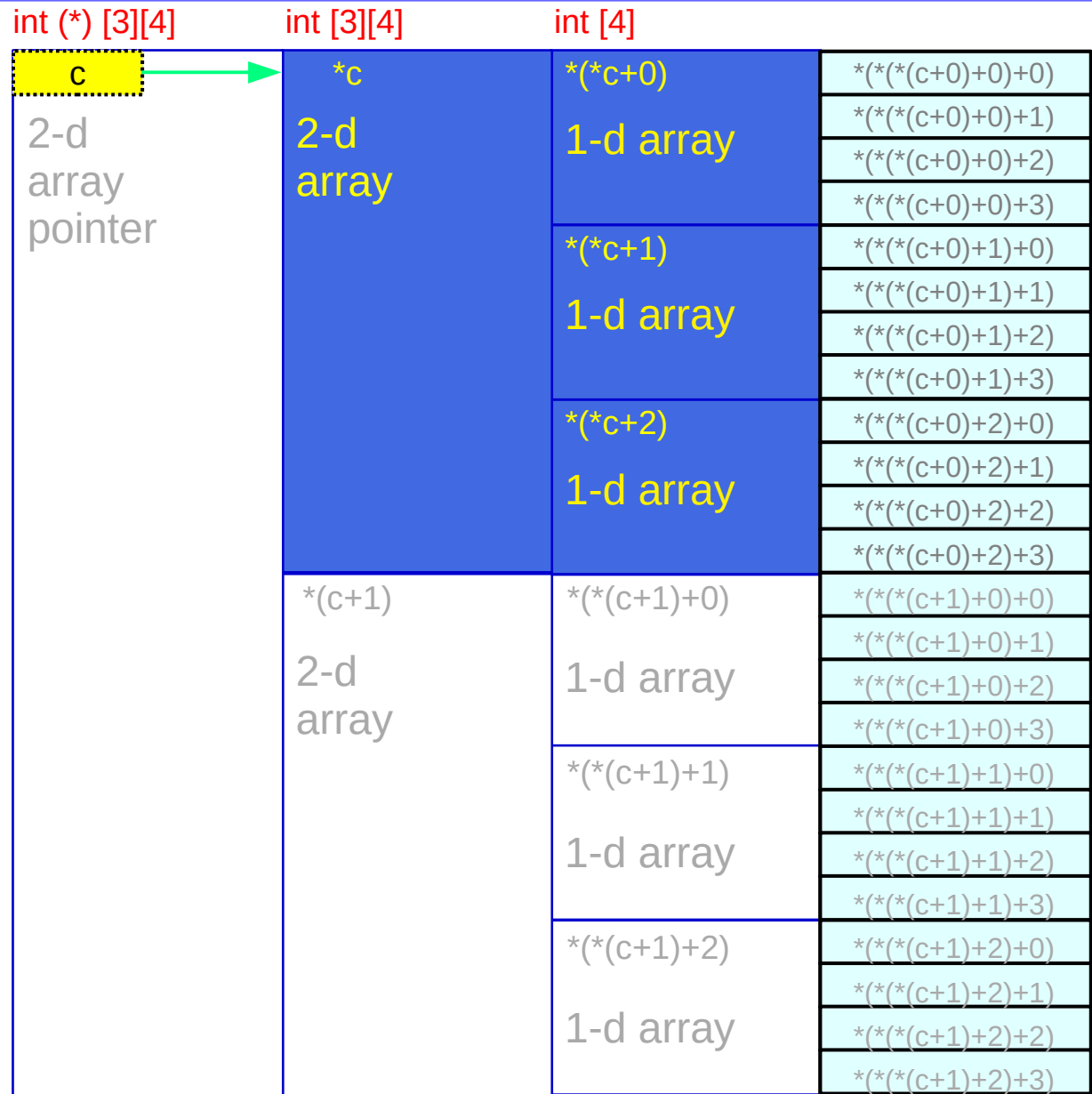
Virtual pointer **c**

2-d array pointer **c**

general pointer type

$\text{sizeof}(c) = \text{sizeof}(*c) * 2$

- c** also has a 2-d array pointer type
- c** has the value of the starting address
- c** is a virtual array pointer



Finding dual types in a 3-d array

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

The type of **c**

- dual types
- `int [2][3][4]` abstract data type
 - `int [][3][4]` relaxing the 1st dimension
 - `int (*)[3][4]` virtual pointer type - gcc displaying type

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

The type of **c[i]**

- dual types
- `int [3][4]` abstract data type
 - `int [][4]` relaxing the 1st dimension
 - `int (*)[4]` virtual pointer type - gcc displaying type

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

The type of **c[i][j]**

- dual types
- `int [4]` abstract data type
 - `int []` relaxing the 1st dimension
 - `int (*)` virtual pointer type - gcc displaying type

Types of virtual array pointers in a 3-d array

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

`c[i][j][k]`

`c[i][j]`
[k]

`c[i]`
[j] [k]

`c`
[i] [j] [k]

int

int [4]
[k]

int [3][4]
[j] [k]

int [2][3][4]
[i] [j] [k]

array type (name)

int

int (*)
[k]

int (*)[4]
[j] [k]

int (*)[3][4]
[i] [j] [k]

array pointer type

Sizes of virtual array pointers in a **3-d** array

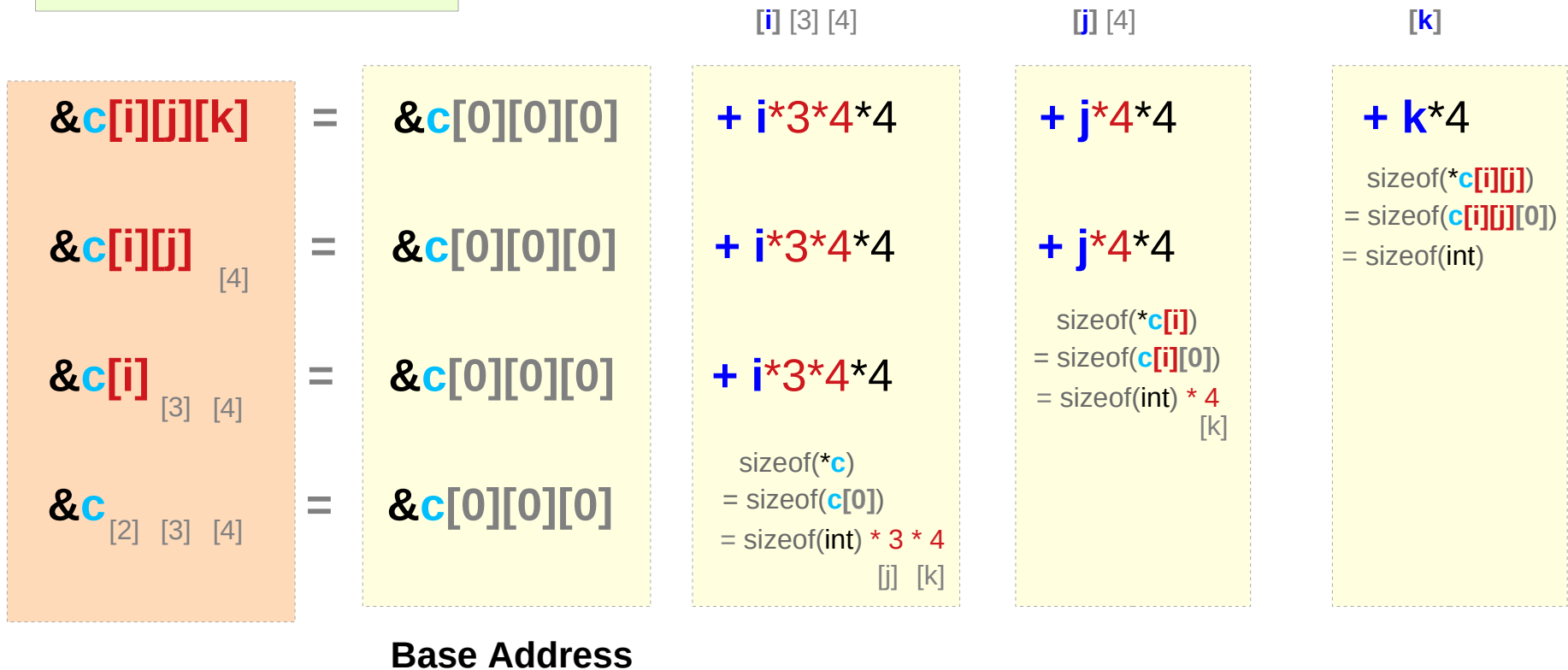
```
int c[2][3][4];
```

$$\begin{aligned} \text{sizeof}(c[i][j][k]) &= \text{sizeof}(\text{int}) \\ \text{sizeof}(c[i][j])_{[k]} &= \text{sizeof}(\text{int}) * 4_{[k]} \\ \text{sizeof}(c[i])_{[j] [k]} &= \text{sizeof}(\text{int}) * 3_{[j]} * 4_{[k]} \\ \text{sizeof}(c)_{[i] [j] [k]} &= \text{sizeof}(\text{int}) * 2_{[i]} * 3_{[j]} * 4_{[k]} \end{aligned}$$

Element Size

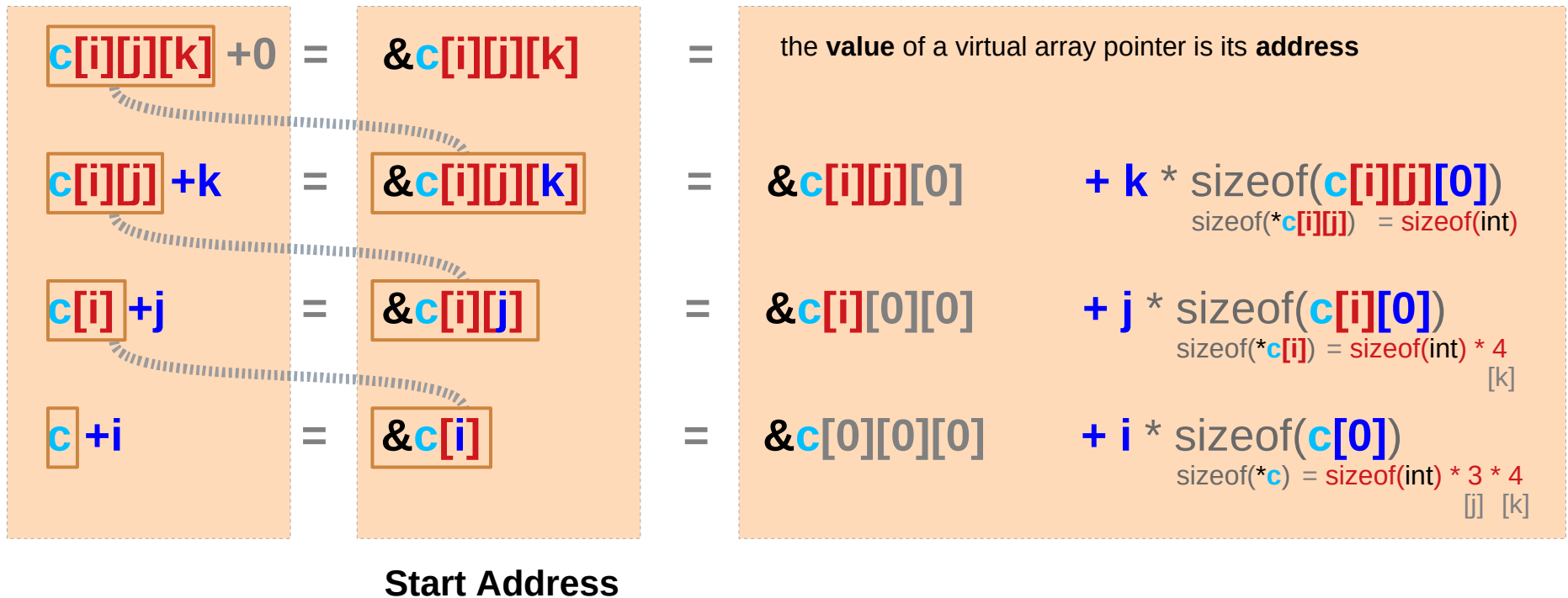
Addresses of virtual array pointers in a 3-d array

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



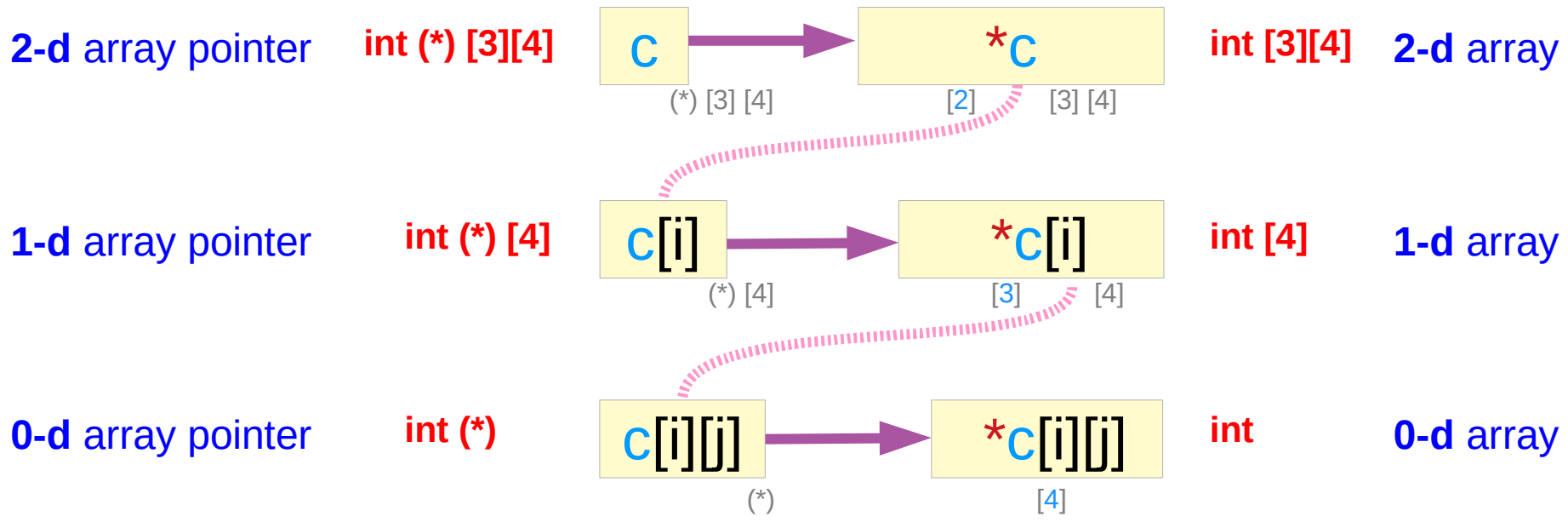
Values of virtual array pointers in a 3-d array

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



Virtual array pointers and abstract data in a 3-d array

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

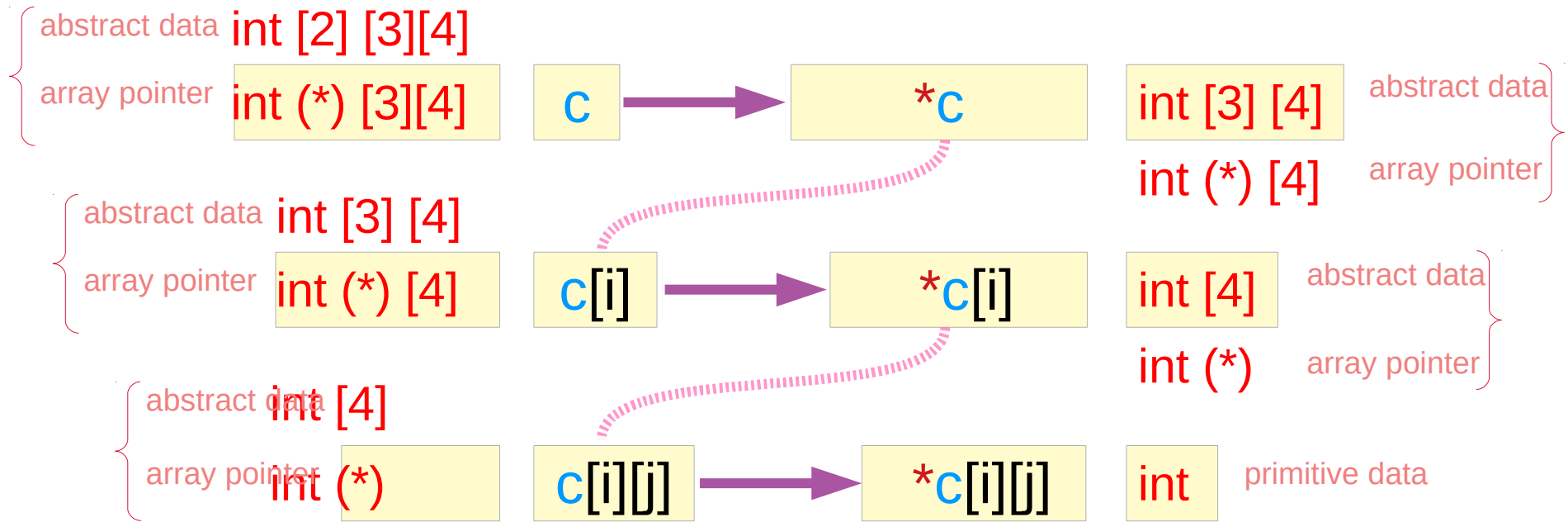


Virtual Array Pointers

Abstract Data (Array)

Dual types in a 3-d array

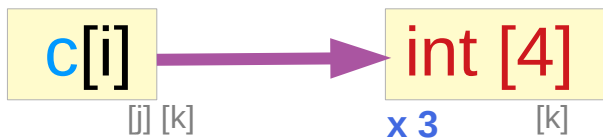
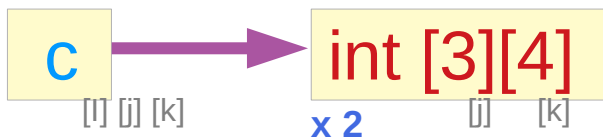
```
int c [2][3][4];
```



Pointed array sizes in a 3-d array

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

the size of a pointer type is fixed
Here, the sizes of virtual pointers are shown
i.e, the sizes of different abstract data types



sizeof(*c) = sizeof(int [3][4])

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

sizeof(*c[i][j]) = sizeof(int)

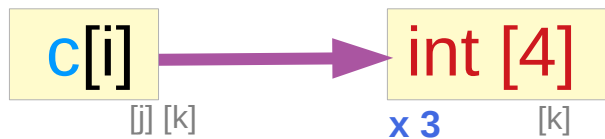
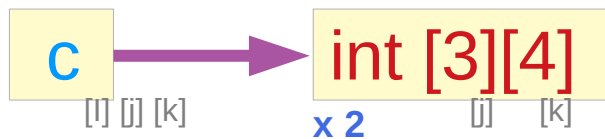
all are sizes of arrays

c, c[i], c[i][j] are virtual array pointers
and they are also abstract data (arrays)

when sizes are considered,
view them as abstract data (arrays)

Virtual array pointer sizes in a 3-d array

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



size of a virtual array pointer = size of the pointed abstract data type * the number of such data

$$\begin{aligned} \text{sizeof}(c) &= 2 * \text{sizeof}(*c) \\ \text{sizeof}(c[i]) &= 3 * \text{sizeof}(*c[i]) \\ \text{sizeof}(c[i][j]) &= 4 * \text{sizeof}(*c[i][j]) \end{aligned}$$

$\text{sizeof}(\text{Virtual Array Pointer}) = \text{sizeof}(\text{Array of the dual type})$

Virtual pointer sizes are subarray sizes

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

`c` \rightarrow `int [3][4]`
[i] [j] [k] [j] [k]

`c[i]` \rightarrow `int [4]`
[j] [k] [k]

`c[i][j]` \rightarrow `int`
[k]

not real array pointers
virtual array pointers

`sizeof(Virtual Array Pointer) =
sizeof(Array of the dual type)`



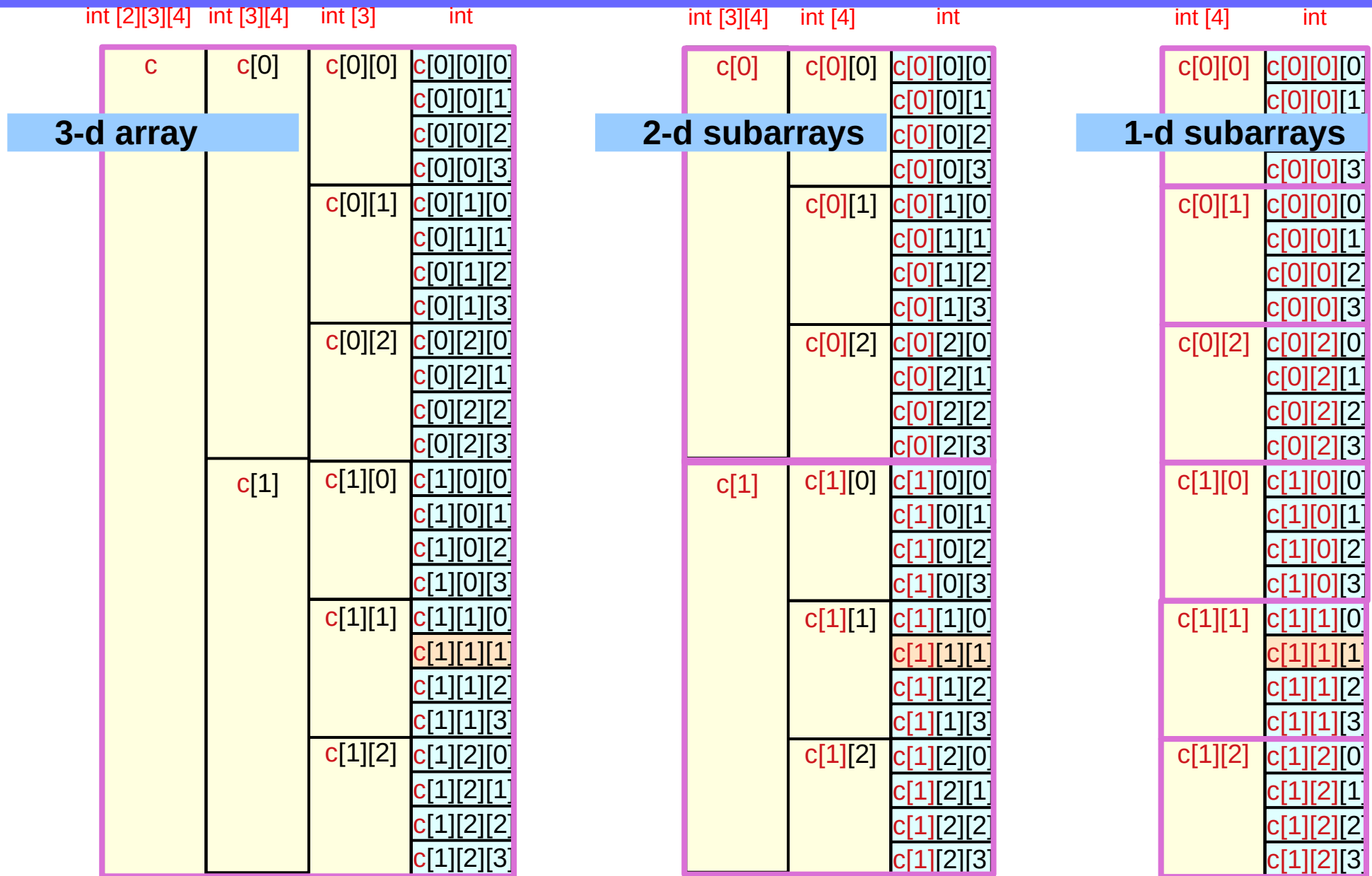
`sizeof(int [2] [3][4]) = sizeof(c) = 2*3*4 * 4`
`sizeof(int (*) [3][4]) = pointer size = 4 or 8`

`sizeof(int [3] [4]) = sizeof(c[i]) = 3*4 * 4`
`sizeof(int (*) [4]) = pointer size = 4 or 8`

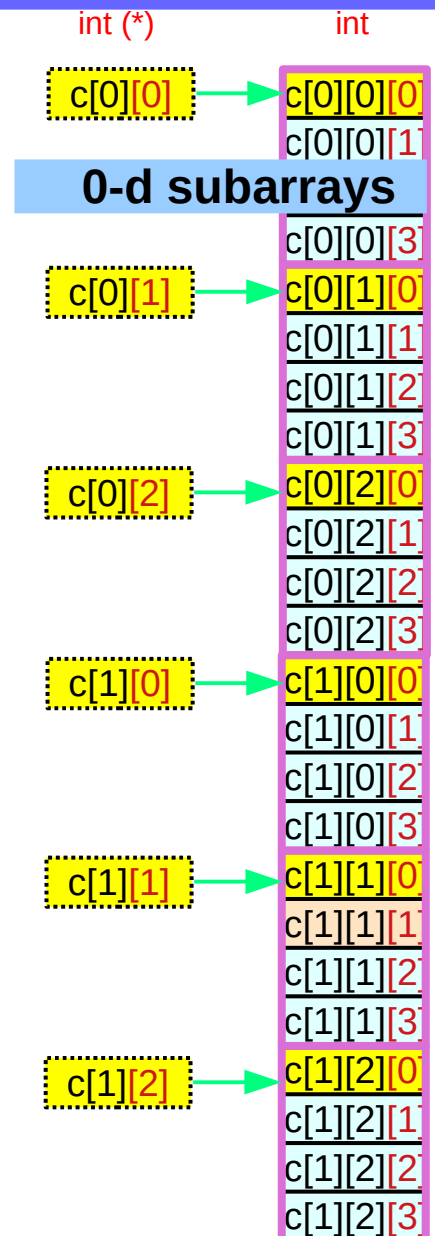
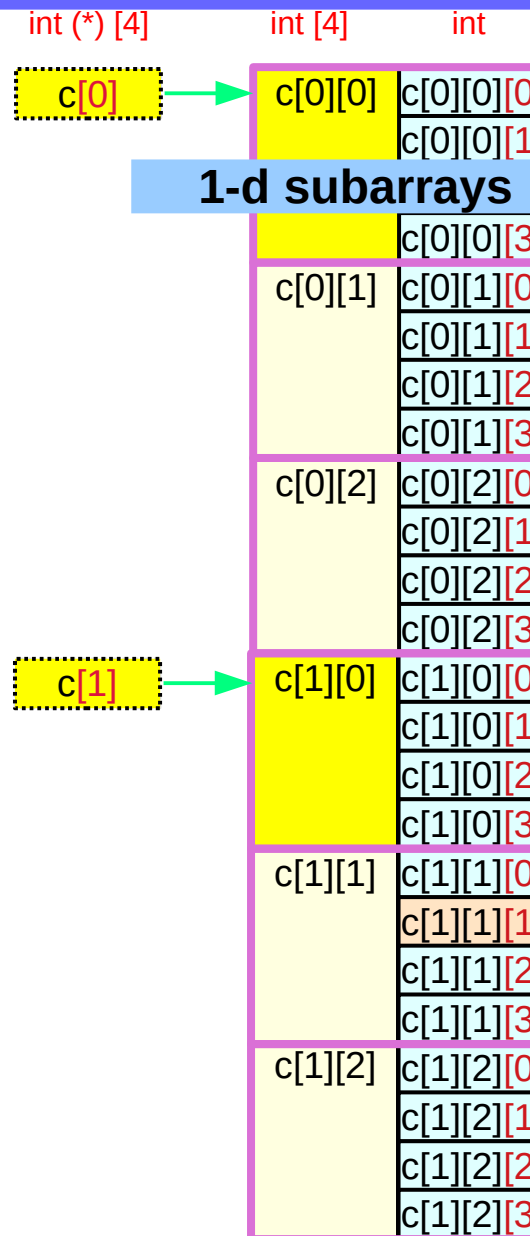
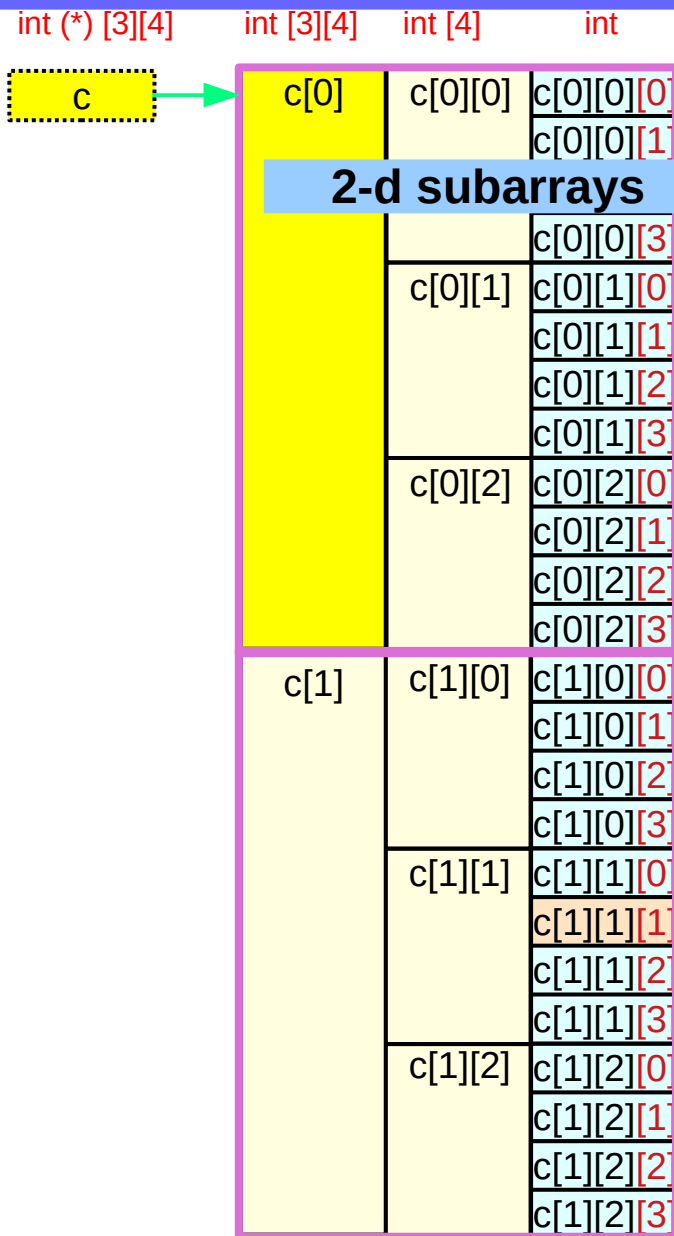
`sizeof(int [4]) = sizeof(c[i][j]) = 4 * 4`
`sizeof(int [4]) = pointer size = 4 or 8`

4 bytes for 32-bit machines
8 bytes for 64-bit machines

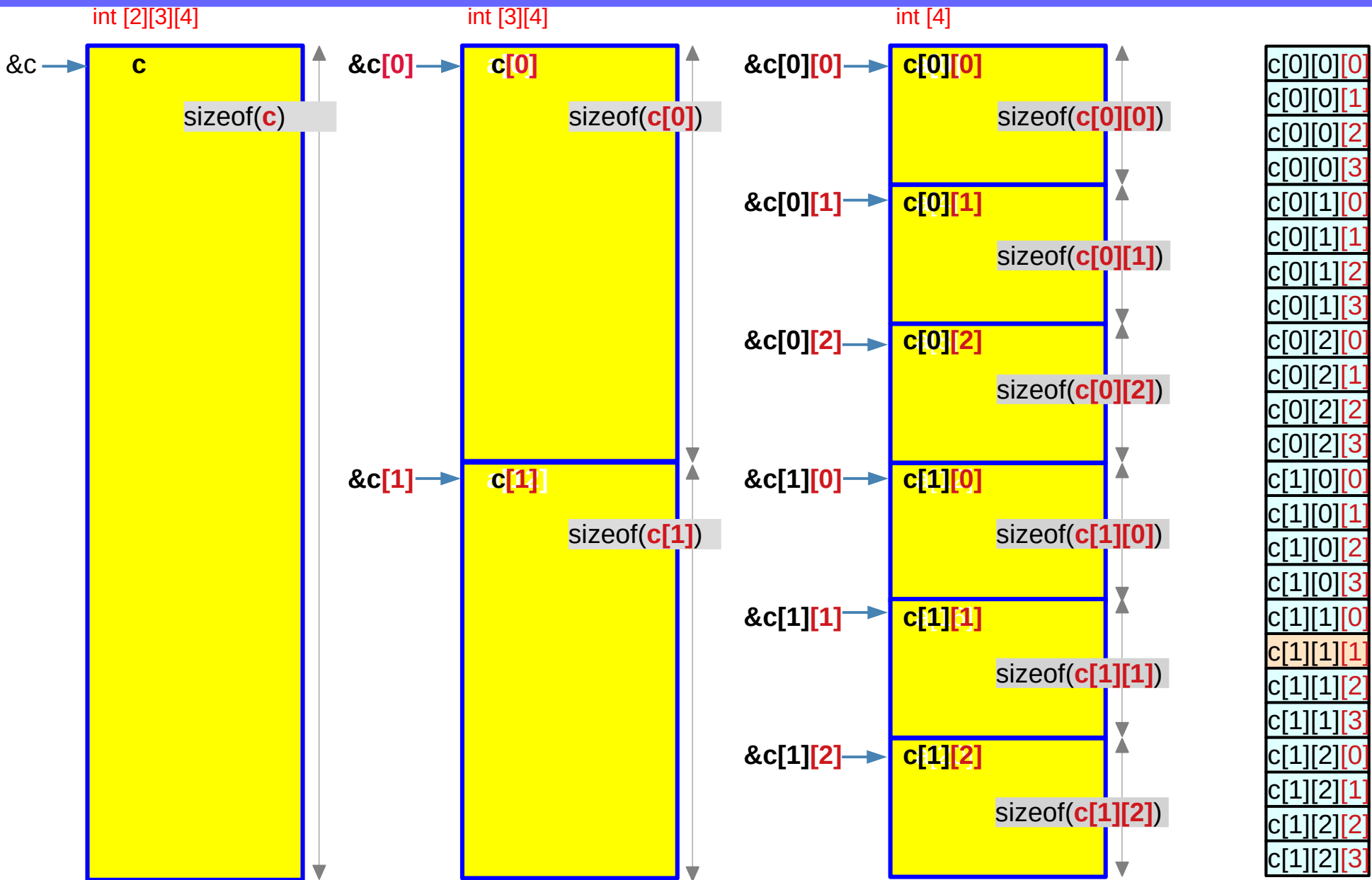
Subarrays c , $c[i]$, $c[i][j]$ in a 3-d array



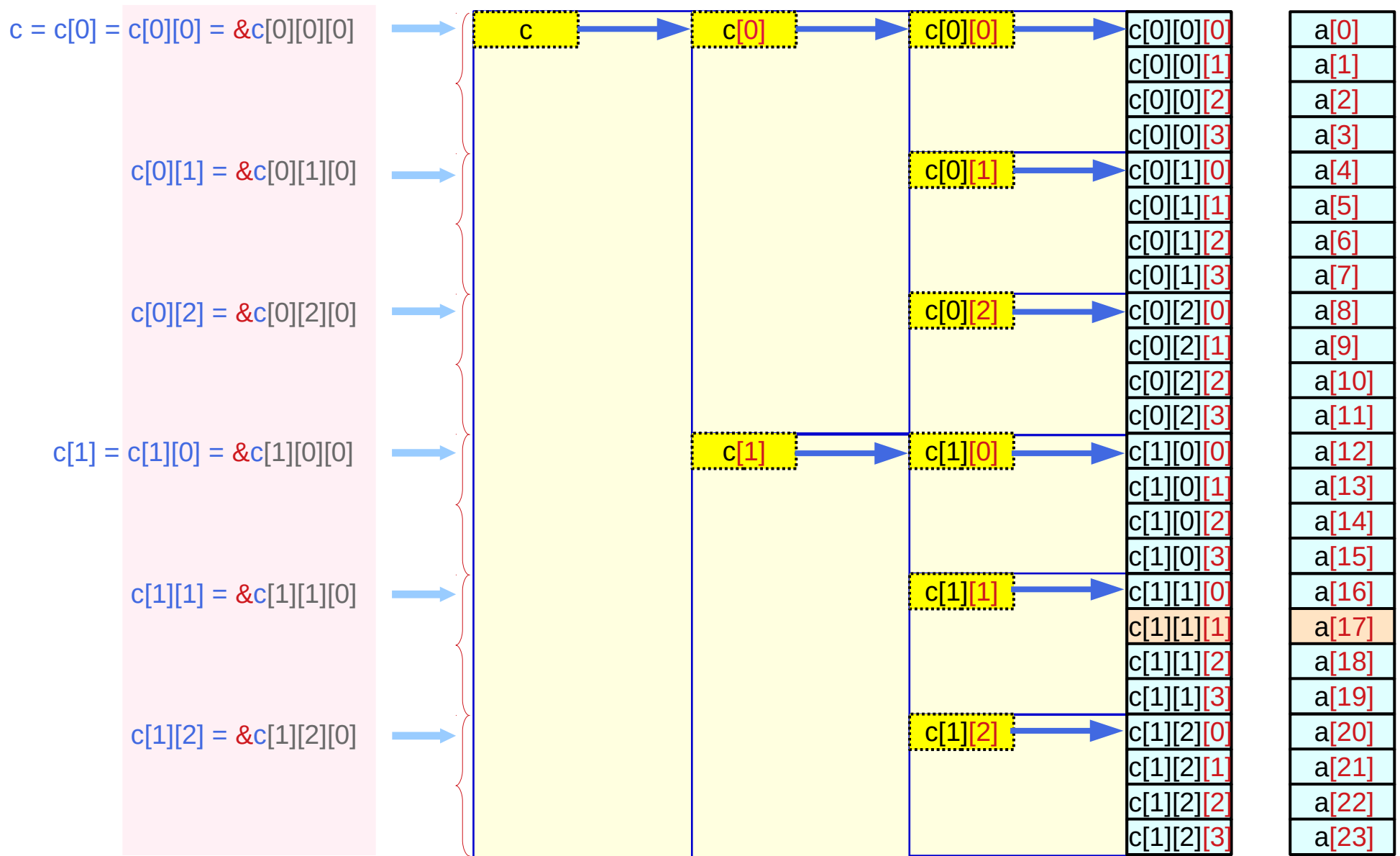
Virtual array pointers c , $c[i]$, $c[i][j]$ in a 3-d array



Abstract data \mathbf{c} , $\mathbf{c}[i]$, $\mathbf{c}[i][j]$ – start addresses and sizes



Virtual array pointer c , $c[i]$, $c[i][j]$ – values (addresses)



Virtual array pointer c , $c[i]$, $c[i][j]$ – values and types

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

$c[0][1] = \&c[0][1][0]$ means

$c[0][2] = \&c[0][2][0]$ means

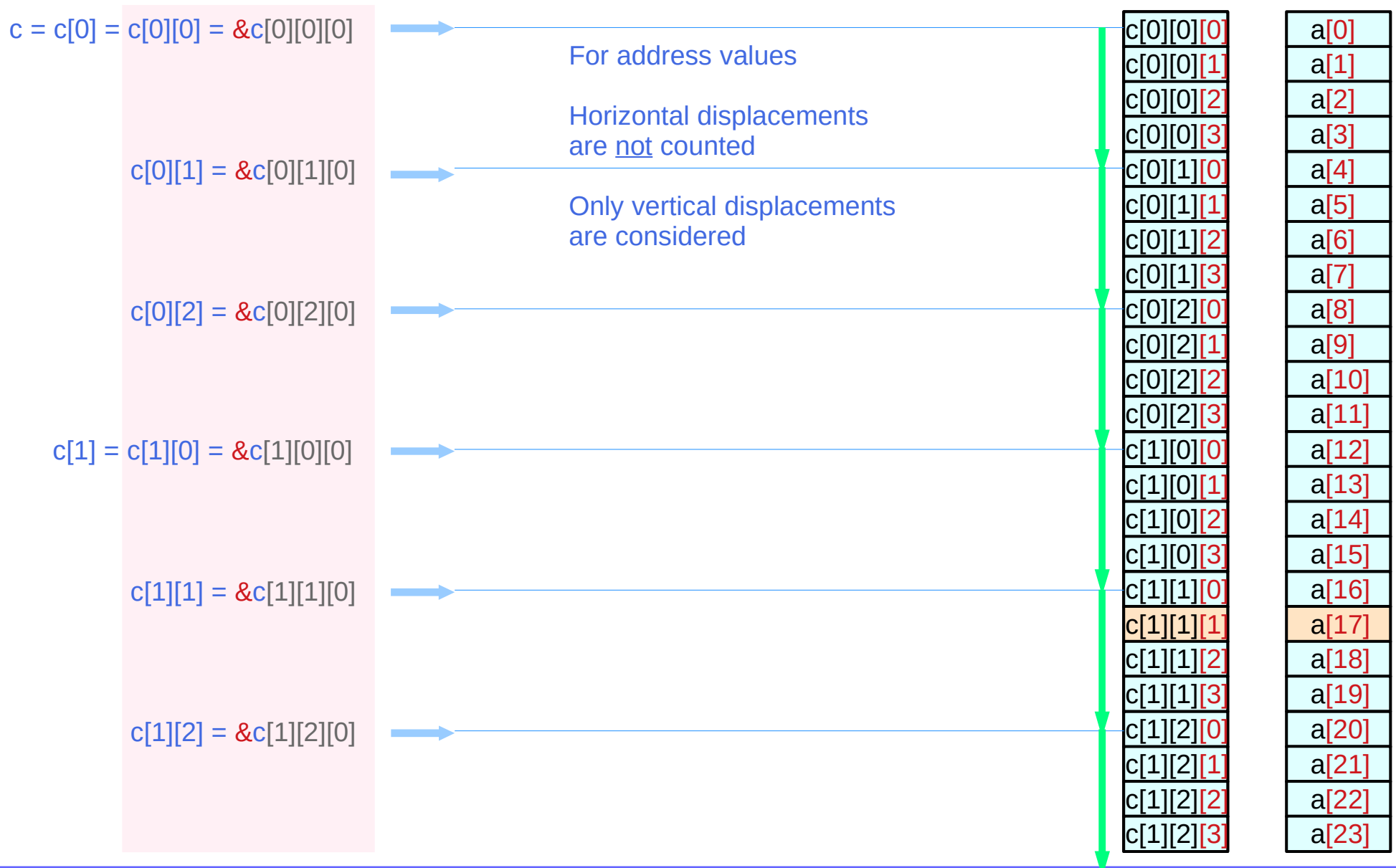
$c[1] = c[1][0] = \&c[1][0][0]$ means

$c[1][1] = \&c[1][1][0]$ means

$c[1][2] = \&c[1][2][0]$ means

$\text{value}(c) = \text{value}(c[0]) = \text{value}(c[0][0]) = \text{value}(\&c[0][0][0])$ $\text{type}(c) \neq \text{type}(c[0]) \neq \text{type}(c[0][0]) = \text{type}(\&c[0][0][0])$ $\text{int } (*) [3][4] \quad \text{int } (*) [4] \quad \text{int } * \quad \text{int } *$	$\text{value}(c[0][1]) = \text{value}(\&c[0][1][0])$ $\text{type}(c[0][1]) = \text{type}(\&c[0][1][0])$ $\text{int } * \quad \text{int } *$
$\text{value}(c[0][2]) = \text{value}(\&c[0][2][0])$ $\text{type}(c[0][2]) = \text{type}(\&c[0][2][0])$ $\text{int } * \quad \text{int } *$	$\text{value}(c[1]) = \text{value}(c[1][0]) = \text{value}(\&c[1][0][0])$ $\text{type}(c[1]) \neq \text{type}(c[1][0]) = \text{type}(\&c[1][0][0])$ $\text{int } (*) [4] \quad \text{int } * \quad \text{int } *$
$\text{value}(c[1][1]) = \text{value}(\&c[1][1][0])$ $\text{type}(c[1][1]) = \text{type}(\&c[1][1][0])$ $\text{int } * \quad \text{int } *$	$\text{value}(c[1][2]) = \text{value}(\&c[1][2][0])$ $\text{type}(c[1][2]) = \text{type}(\&c[1][2][0])$ $\text{int } * \quad \text{int } *$

Virtual array pointer c , $c[i]$, $c[i][j]$ – vertical displacement



Virtual array pointers – types, sizes, and values

int c[2][3][4];	c[i][j]	c[i][j][0]	
type	int [4] int (*)	int int	<ul style="list-style-type: none"> abstract data type array pointer type
size	sizeof(c[i][j]) =	sizeof(c[i][j][0]) * 4	= sizeof(int) * 4
value (address)	c[i][j] =	&c[i][j][0]	
int c[2][3][4];	c[i]	c[i][0]	
type	int [3][4] int (*)[4]	int [4] int (*)	<ul style="list-style-type: none"> abstract data type array pointer type
size	sizeof(c[i]) =	sizeof(c[i][0]) * 3	= sizeof(int) * 4 * 3
value (address)	c[i] =	&c[i][0][0]	
int c[2][3][4];	c	c[0]	
type	int [2][3][4] int (*)[3][4]	int [3][4] int (*)[4]	<ul style="list-style-type: none"> abstract data type array pointer type
size	sizeof(c) =	sizeof(c[0]) * 2	= sizeof(int) * 4 * 3 * 2
value (address)	c =	&c[0][0][0]	

Summary of virtual array pointers in a 3-d array

$$c[i] \equiv *(c + i)$$

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

address value $c + i$

$\&c[0][0][0] + i * \text{sizeof}(*c)$
 $\&c[0][0][0] + i * \text{sizeof}(c[0])$
 $\&c[0][0][0] + i * 4 * 3 * 4$

leading elements

$c[0][0][0]$

$$c[i][j] \equiv *(c[i] + j)$$

int (*) [4] 1-d array pointers $c[i]$
Int [3] [4] 2-d array names $c[i]$

address value $c[i] + j$

$\&c[i][0][0] + j * \text{sizeof}(*c[i])$
 $\&c[i][0][0] + j * \text{sizeof}(c[i][0])$
 $\&c[i][0][0] + j * 4 * 4$

leading elements

$c[0][0][0]$

$c[1][0][0]$

$$c[i][j][k] \equiv *(c[i][j] + k)$$

int (*) 0-d array pointers $c[i][j]$
int [4] 1-d array names $c[i][j]$

address value $c[i][j] + k$

$\&c[i][j][0] + k * \text{sizeof}(*c[i][j])$
 $\&c[i][j][0] + k * \text{sizeof}(c[i][j][0])$
 $\&c[i][j][0] + k * 4$

leading elements

$c[0][0][0]$

$c[0][1][0]$

$c[0][2][0]$

$c[1][0][0]$

$c[1][1][0]$

$c[1][2][0]$

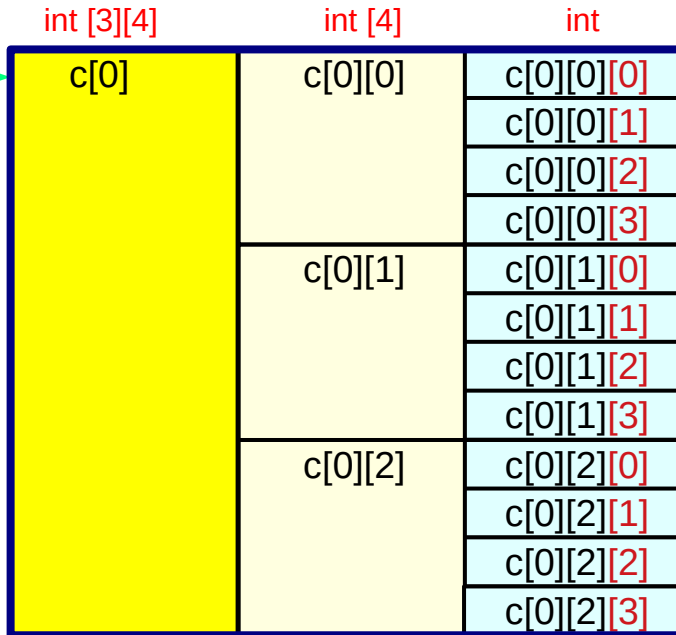
Dual type constraints in a multi-dimensional array

Virtual array pointers to subarrays in a 3-d array

virtual 2-d array pointer

sizeof(c) =
sizeof(c[0]) * 2

int (*) [3][4]
c



the first 2-d subarray

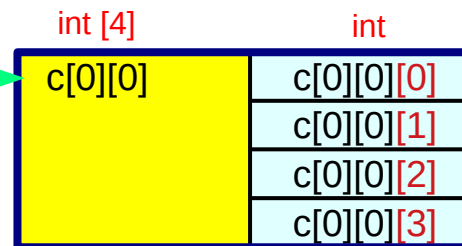
sizeof(c[0]) =
sizeof(int [3][4])

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

virtual 1-d array pointer

sizeof(c[0]) =
sizeof(c[0][0]) * 3

int (*) [4]
c[0]



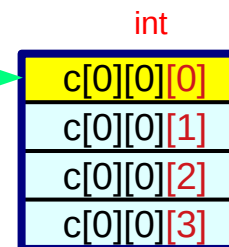
the first 1-d subarray

sizeof(c[0][0]) =
sizeof(int [4])

virtual 0-d array pointer

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

int (*)
c[0][0]



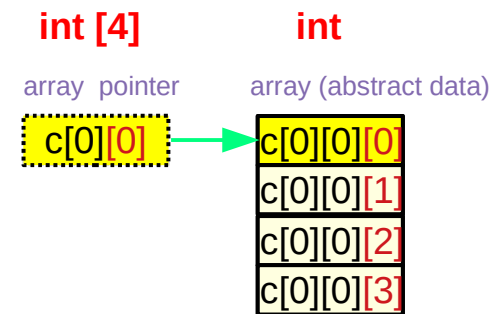
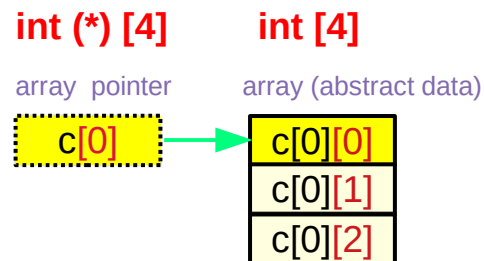
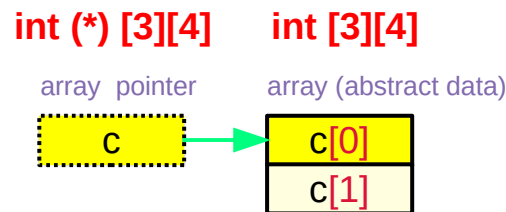
the first 0-d subarray

sizeof(c[0][0][0]) =
sizeof(int)

Virtual array pointer `c`, `c[0]`, `c[0][0]` – types and sizes

Types – array pointers

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



Sizes – abstract data

`sizeof(c)`
`sizeof(int [2][3][4])`
`sizeof(int) * 2 * 3 * 4`

`sizeof(int [2][3][4]) = 96`
`sizeof(int (*)[3][4]) = 4 / 8`

`sizeof(c[0])`
`sizeof(int [3][4])`
`sizeof(int) * 3 * 4`

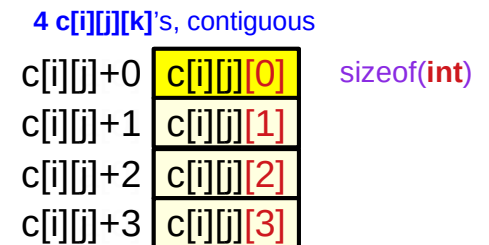
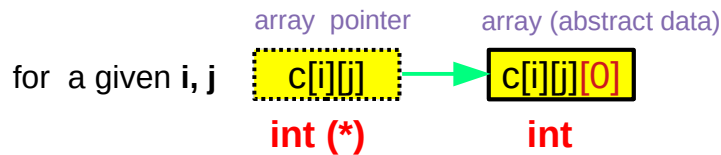
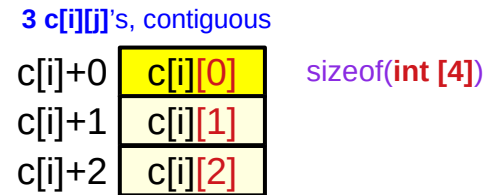
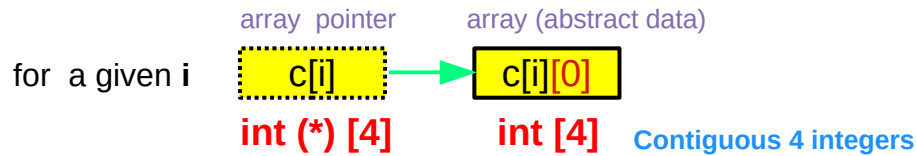
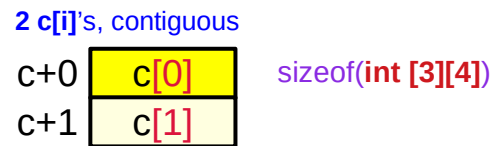
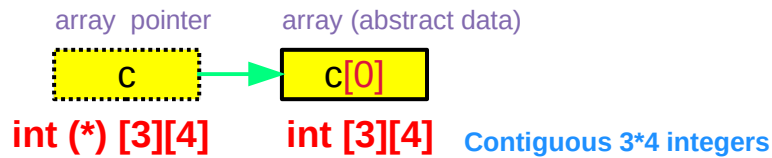
`sizeof(int [3][4]) = 48`
`sizeof(int (*)[4]) = 4 / 8`

`sizeof(c[0][0])`
`sizeof(int [4])`
`sizeof(int) * 4`

`sizeof(int [4]) = 16`
`sizeof(int *) = 4 / 8`

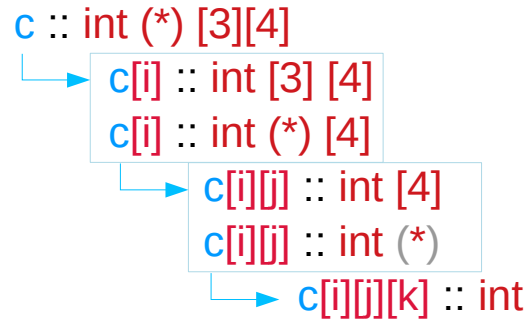
Contiguous subarrays $c[i]$, $c[i][j]$, $c[i][j][k]$

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

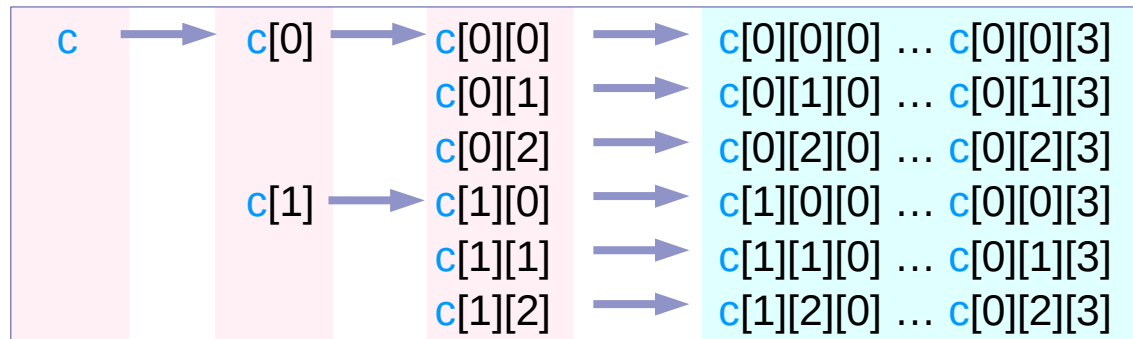


Dual types of `c`, `c[i]`, `c[i][j]`

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



- 2-d array pointers
- 2-d arrays
- 1-d array pointers
- 1-d arrays
- 0-d array pointers
- 0-d arrays (integers)



```

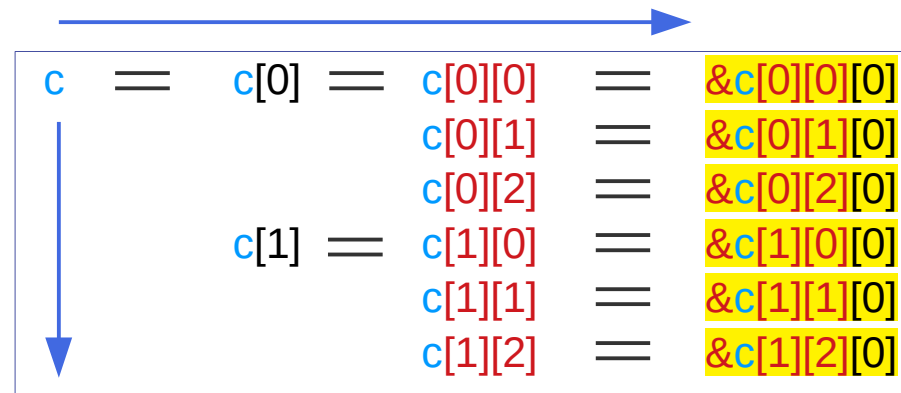
int [2] [3][4]  int [3] [4]      int [4]      int ... int
int (*) [3][4] int (*) [4]      int (*)     int ... int

```

Values of virtual array pointers c , $c[i]$, $c[i][j]$

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

virtual array pointers have address values in each row in the following figure have the same address value



Horizontal displacements are not counted only **vertical displacements** are considered for address values

virtual assignments

```
 $c[i][j] = \&c[i][j][0]$   
 $c[i] = \&c[i][0][0]$   
 $c = \&c[0][0][0]$ 
```

Finding values of virtual array pointers **c**, **c[i]**, **c[i][j]**

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

append **[0]** to the right

c	$\xrightarrow{+[0]}$	c[0]	$\xrightarrow{+[0]}$	c[0][0]	$\xrightarrow{+[0]}$	&c[0][0][0]
				c[0][1]	$\xrightarrow{+[0]}$	&c[0][1][0]
				c[0][2]	$\xrightarrow{+[0]}$	&c[0][2][0]
		c[1]	$\xrightarrow{+[0]}$	c[1][0]	$\xrightarrow{+[0]}$	&c[1][0][0]
				c[1][1]	$\xrightarrow{+[0]}$	&c[1][1][0]
				c[1][2]	$\xrightarrow{+[0]}$	&c[1][2][0]

int (*) [3][4] int (*) [4] int [4] int

c[0][0][0] :
leading
elements
of **c**

c[i][0][0] :
leading
elements
of **c[i]**

c[i][j][0] :
leading
elements
of **c[i][j]**

&c[0][0][0]

&c[0][0][0]

&c[1][0][0]

&c[0][0][0]
&c[0][1][0]
&c[0][2][0]
&c[1][0][0]
&c[1][1][0]
&c[1][2][0]

Finding sub-arrays with the address `&c[i][j][0]`

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

delete `[0]` from the right

<code>&c[0][0][0]</code>	<code>==^{-[0]}</code>	<code>c[0][0]</code>	<code>==^{-[0]}</code>	<code>c[0]</code>	<code>==^{-[0]}</code>	<code>c</code>
<code>&c[0][1][0]</code>	<code>==^{-[0]}</code>	<code>c[0][1]</code>				
<code>&c[0][2][0]</code>	<code>==^{-[0]}</code>	<code>c[0][2]</code>				
<code>&c[1][0][0]</code>	<code>==^{-[0]}</code>	<code>c[1][0]</code>	<code>==^{-[0]}</code>	<code>c[1]</code>		
<code>&c[1][1][0]</code>	<code>==^{-[0]}</code>	<code>c[1][1]</code>				
<code>&c[1][2][0]</code>	<code>==^{-[0]}</code>	<code>c[1][2]</code>				

int

int [4]

int (*) [4]

int (*) [3][4]

`c[0][0][0]` is the leading element of `c[0][0]`, `c[0]`, `c`

`c[0][1][0]` is the leading element of `c[0][1]`

`c[0][2][0]` is the leading element of `c[0][2]`

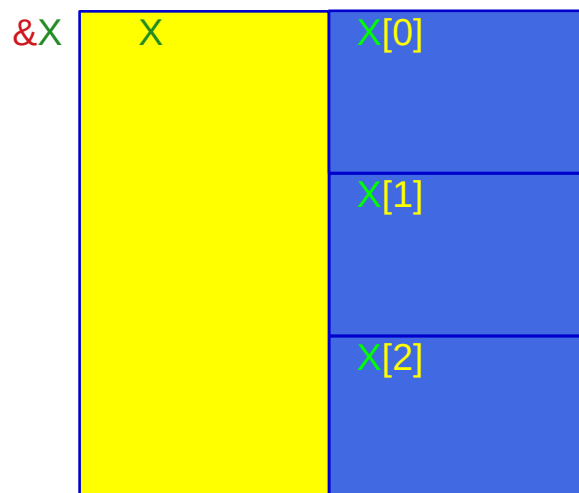
`c[1][0][0]` is the leading element of `c[1][0]`, `c[1]`

`c[1][1][0]` is the leading element of `c[1][1]`

`c[1][2][0]` is the leading element of `c[1][2]`

Dual types in a 3-d array

Abstract data (array) X



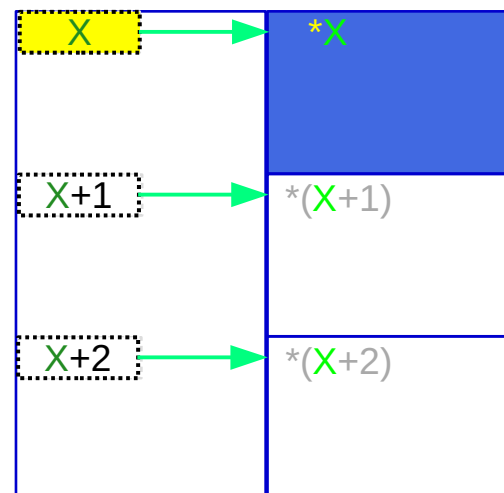
array (abstract data)

array (abstract data)

`c[i][j]` starts from `&c[i][j][0]`
`c[i]` starts from `&c[i][0]`
`c` starts from `&c[0]`

`&c[i][j]` = `&c[i][j][0]`
`&c[i]` = `&c[i][0]`
`&c` = `&c[0]`

Virtual array pointer X



array pointer

array (abstract data)

`c[i][j]` points to `c[i][j][0]`
`c[i]` points to `c[i][0]`
`c` points to `c[0]`

address value

`c[i][j]` = `&c[i][j][0]`
`c[i]` = `&c[i][0]`
`c` = `&c[0]`

Dual type constraints

```
&c[i][j][0] = c[i][j]
&c[i][0]     = c[i]
&c[0]       = c
```

**Virtual
array
pointer**

array pointer		array (abstract data)	
<code>c[i][j]</code>	points to	<code>c[i][j][0]</code>	
<code>c[i]</code>	points to	<code>c[i][0]</code>	
<code>c</code>	points to	<code>c[0]</code>	
address value			

X



```
&c[i][j][0] = &c[i][j]
&c[i][0]    = &c[i]
&c[0]      = &c
```

**Abstract
data
(array)**

array (abstract data)		array (abstract data)	
<code>c[i][j]</code>	starts from	<code>&c[i][j][0]</code>	
<code>c[i]</code>	starts from	<code>&c[i][0]</code>	
<code>c</code>	starts from	<code>&c[0]</code>	

&X X



```
c[i][j] = &c[i][j]
c[i]    = &c[i]
c       = &c
```

array (abstract data)		Address of an array pointer
<code>c[i][j]</code>	pointer value =	pointer address <code>&c[i][j]</code>
<code>c[i]</code>	pointer value =	pointer address <code>&c[i]</code>
<code>c</code>	pointer value =	pointer address <code>&c</code>
X		&X

c[0] = c[0][0] relation

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

```
c == c[0] == c[0][0] == &c[0][0][0]
```

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

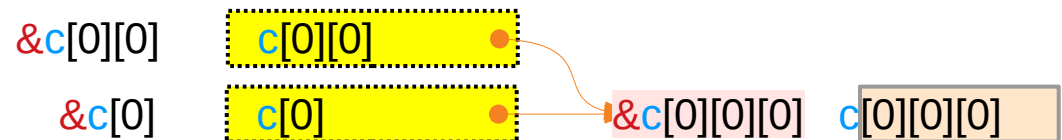
```
value(c[0][0]) = &c[0][0][0]
```

```
type(c[0]) = int (*)[4]
```

```
type(c[0][0]) = int [4]
```

```
c[0] = c[0][0] means  
value(c[0]) = value(c[0][0])
```

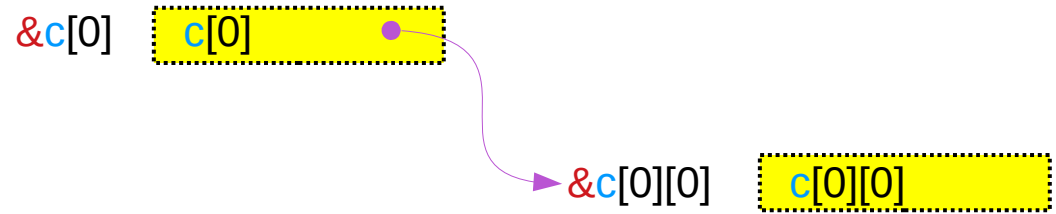
```
c[0] = c[0][0] does not mean  
type(c[0]) = type(c[0][0])
```



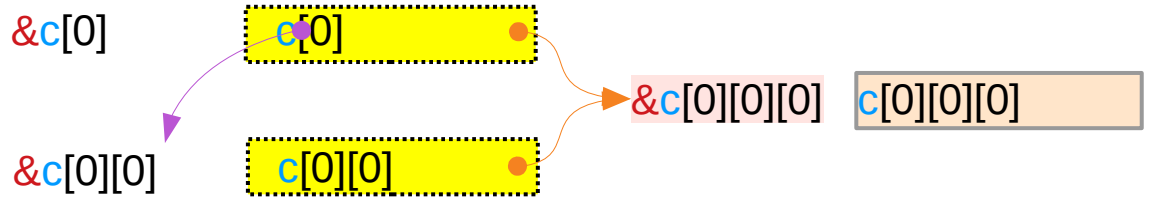
Addresses and Values of `c[0]` and `c[0][0]`

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

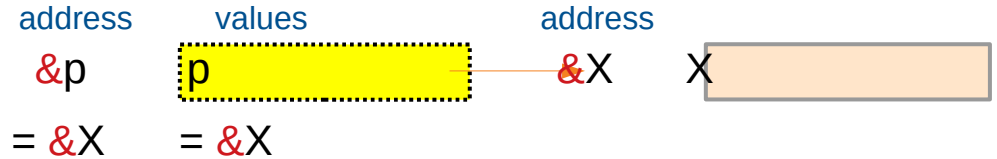
`c[0]` → `c[0][0]`



`c[0]` == `c[0][0]` == `&c[0][0][0]`



A virtual pointer's address and value are the same

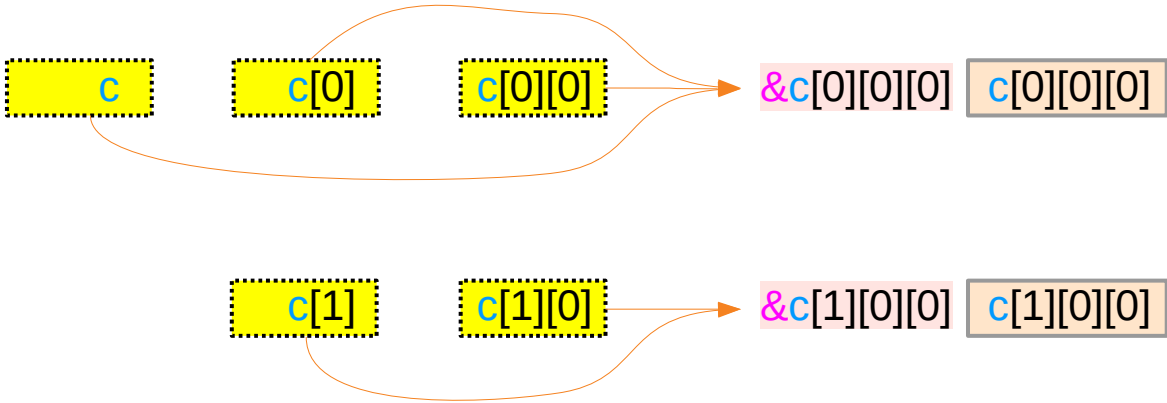


c[i] and c[i][0] point to the same c[i][0][0]

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

```
c = c[0] = c[0][0] = &c[0][0][0] ← value  
int(*)[3][4] int(*)[4] int(*) int ← type
```

```
c[1] = c[1][0] = &c[1][0][0] ← value  
int(*)[4] int(*) int ← type
```



These virtual pointers have different types but the same value (address)

&c[i][0] and &c[i][0][0] – equivalence relations

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

int(*)[3][4] int(*)[4] int(*) int *

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

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

equivalences

```
c ≡ &c[0],  
c[0] ≡ &c[0][0]  
c[0][0] ≡ &c[0][0][0]
```

$c[1] = c[1][0] = \&c[1][0][0]$

$\&c[1]$ $\&c[1][0]$

equivalences

```
c[1] ≡ &c[1][0]  
c[1][0] ≡ &c[1][0][0]
```

Horizontal displacements are not counted
only vertical displacements are considered
for address values

equivalences

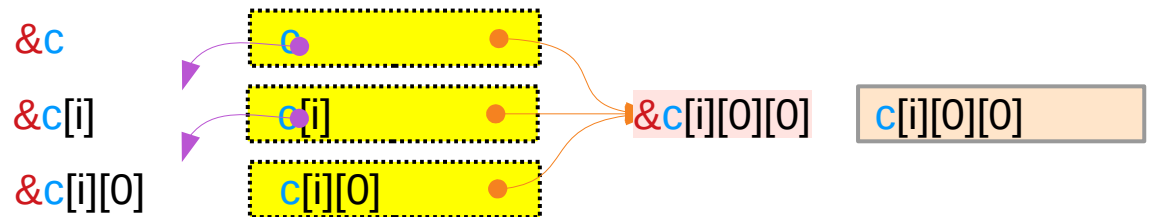
```
c ≡ &c[0],  
c[i] ≡ &c[i][0]  
c[i][0] ≡ &c[i][0][0]
```

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

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

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

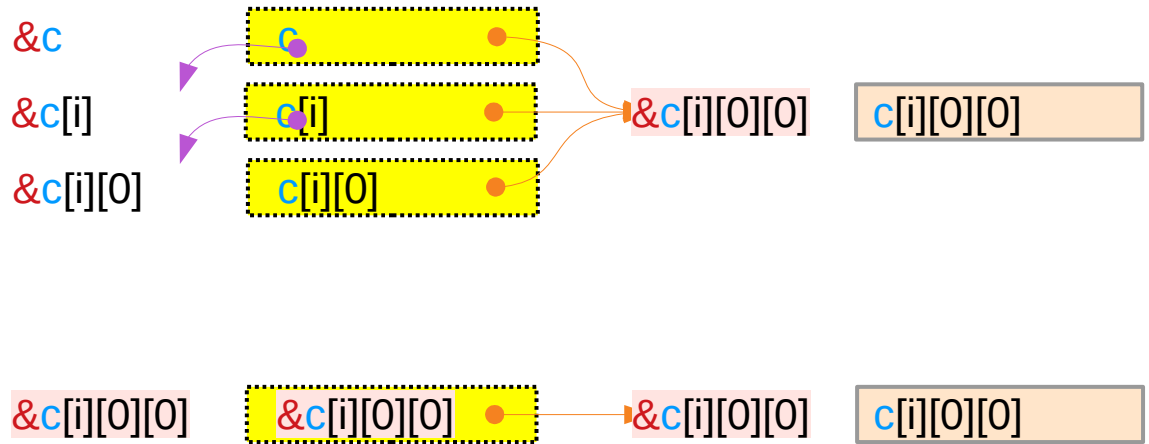
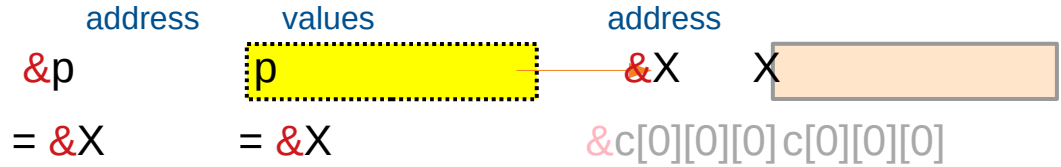
$c[1] = c[1][0] = \&c[1][0][0]$
||
 $\&c[1] = \&c[1][0]$



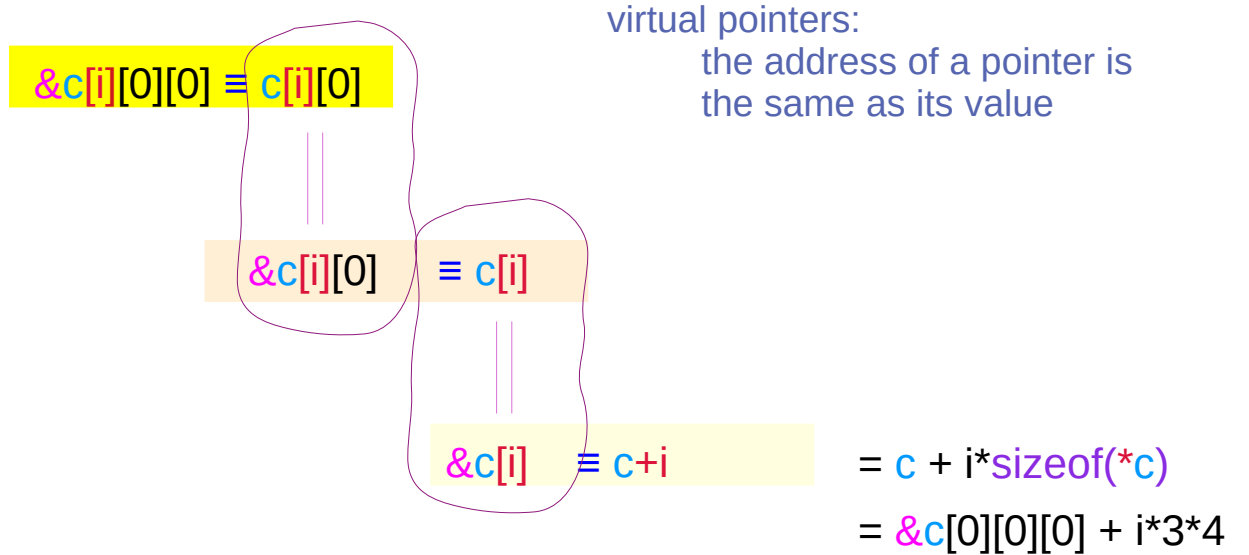
$c[i] = \&c[i]$ and $c[i][0] = \&c[i][0]$

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

A virtual pointer's address and value are the same



Array Pointers to $c[i][0][0]$



delete [0] from the right

$\&c[0][0][0]$	\equiv	$c[0][0]$	\equiv	$c[0]$	\equiv	c
$\&c[1][0][0]$	\equiv	$c[1][0]$	\equiv	$c[1]$	\equiv	

Array Pointers to $c[i][j][0]$

$$\&c[i][j][0] \equiv c[i][j]$$

$$\&c[i][j] \equiv c[i] + j$$

$$= c[i] + j * \text{sizeof}(*c[i])$$

$$= c + i * \text{sizeof}(*c) + j * 4$$

$$= \&c[0][0][0] + i * 3 * 4 + j * 4$$

delete [0] from the right

$\&c[0][0][0]$	$\overline{\overline{\overline{-[0]}}}$	$c[0][0]$	$\overline{\overline{-[0]}}$	$c[0]$	$\overline{\overline{-[0]}}$	c
$\&c[0][1][0]$	$\overline{\overline{\overline{-[0]}}}$	$c[0][1]$				
$\&c[0][2][0]$	$\overline{\overline{\overline{-[0]}}}$	$c[0][2]$	$\overline{\overline{-[0]}}$			
$\&c[1][0][0]$	$\overline{\overline{\overline{-[0]}}}$	$c[1][0]$		$c[1]$		
$\&c[1][1][0]$	$\overline{\overline{\overline{-[0]}}}$	$c[1][1]$				
$\&c[1][2][0]$	$\overline{\overline{\overline{-[0]}}}$	$c[1][2]$				

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