

# Arrays (1A)

---

---

Copyright (c) 2023 - 2009 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.

- 
- **Array declaration**
  - **Accessing array elements**

# Computing the mean of $N$ numbers

**The mean of  $N$  numbers**

$$m = \frac{1}{N} \sum_{i=0}^{N-1} x_i$$

$$m = \frac{1}{5} \sum_{i=0}^4 x_i = \frac{1}{5} (x_0 + x_1 + x_2 + x_3 + x_4)$$

5 integer variables

x[0]

x[1]

x[2]

x[3]

x[4]

5 indices

0

1

2

3

4

# Definition of an Array

```
int x[5];
```

Array Type

**int [5]**

Array Name

**x**

5 integer variables

x[0]

x[1]

x[2]

x[3]

x[4]

5 indices

0

1

2

3

4

# Element Type

```
int x[5];
```

Array Type

`int [5]`

Array Variable (constant)

`x`

Value: the starting address of 5 consecutive int variables

```
int x[5];
```

Element Type

`int`

Element Variable

`x[i]`

Index Variable

`i`

# Using an Array

```
int x[5];
```

Array Type  
Array Variable  
(constant)

**int [5]**  
**x**

```
int x[5];
```

[5] is declared and  
[0], [1], [2], [3], [4] are used

Integer  
Variables

```
x[i]
```

Element Type  
Element Variable  
Index Variable

**int**  
**x[i]**  
**i**

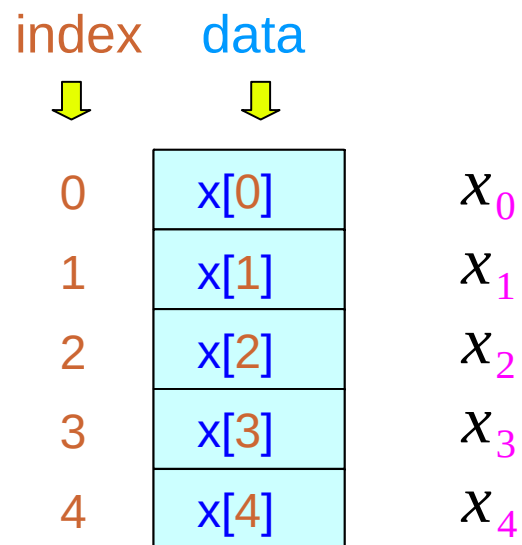
meaningful only for  
 $i = 0, \dots, 4$

# Accessing array elements – using an index

```
int    x[5];
```

`x` is an array of 5 integer elements

5 `int` variables





# Computing the sum of $n$ numbers (1)

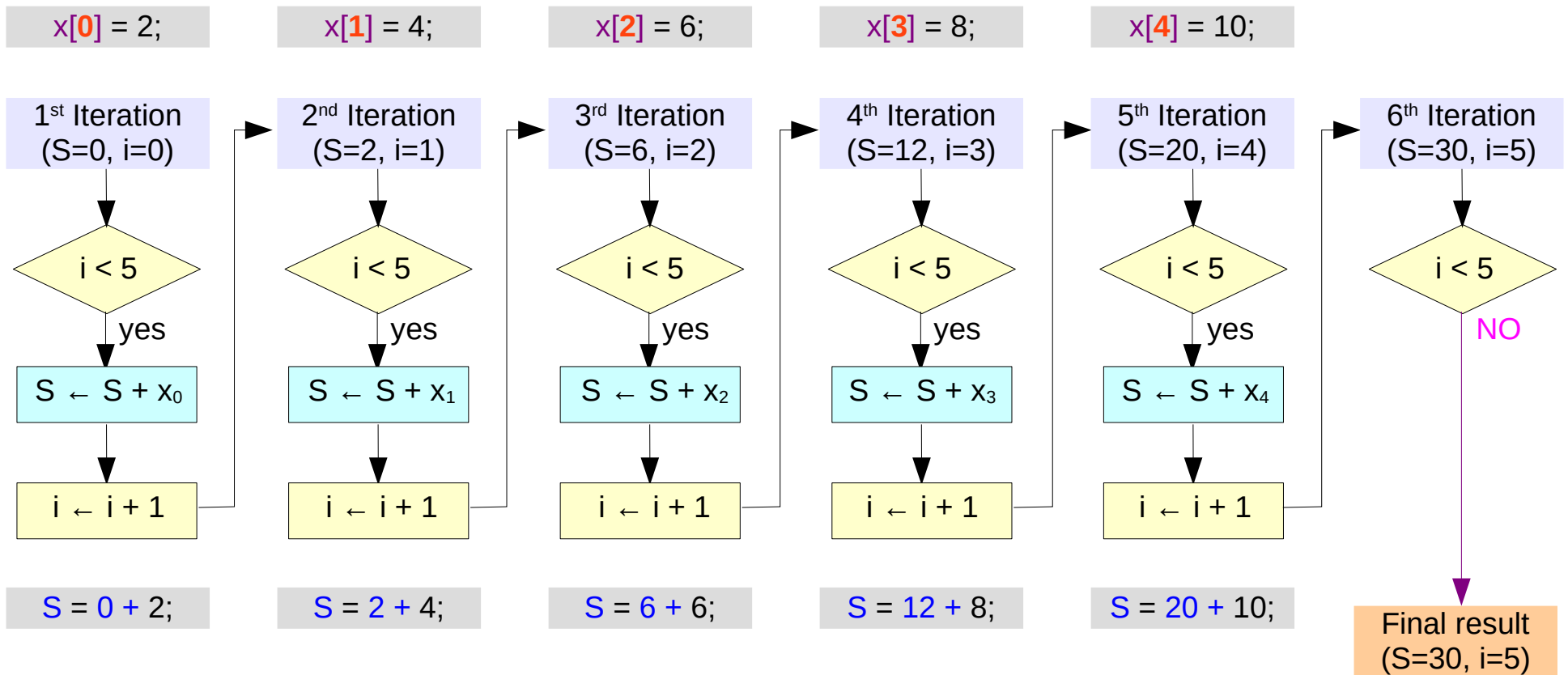
```
sum = 0;  
sum = sum + x[0];  
sum = sum + x[1];  
sum = sum + x[2];  
sum = sum + x[3];  
sum = sum + x[4];
```

treated as an **int** variable

```
sum : 0;  
sum : x0  
sum : x0 + x1  
sum : x0 + x1 + x2  
sum : x0 + x1 + x2 + x3  
sum : x0 + x1 + x2 + x3 + x4
```

```
sum = 0;  
for (i=0; i<5; ++i)  
    sum = sum + x[i];
```

# Computing the sum of $n$ numbers (2)



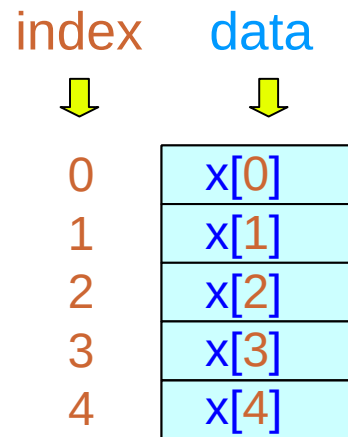
```
S = 0;  
for (i=0; i<5; ++i)  
    S = S + x[i];
```

# Accessing array elements – using an address

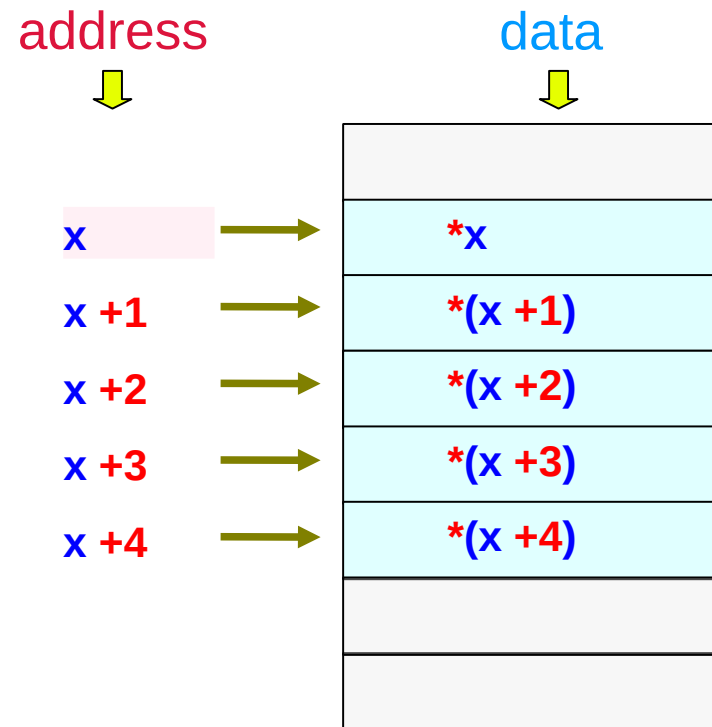
```
int    x[5];
```

**x** holds the starting address  
of 5 consecutive **int** variables

5 int variables



cannot change  
address **x**  
(constant)



# Index and address notations

```
int    x[5];
```

**x** holds the starting address  
of **5** consecutive **int** variables

**x[i]** or **\*(x+i)**

cannot change address **x** (constant)  
assigned by the **gcc** compiler

**i** : an index variable [0 .. 4]  
**x[i]** : the **(i+1)**<sup>th</sup> element variable

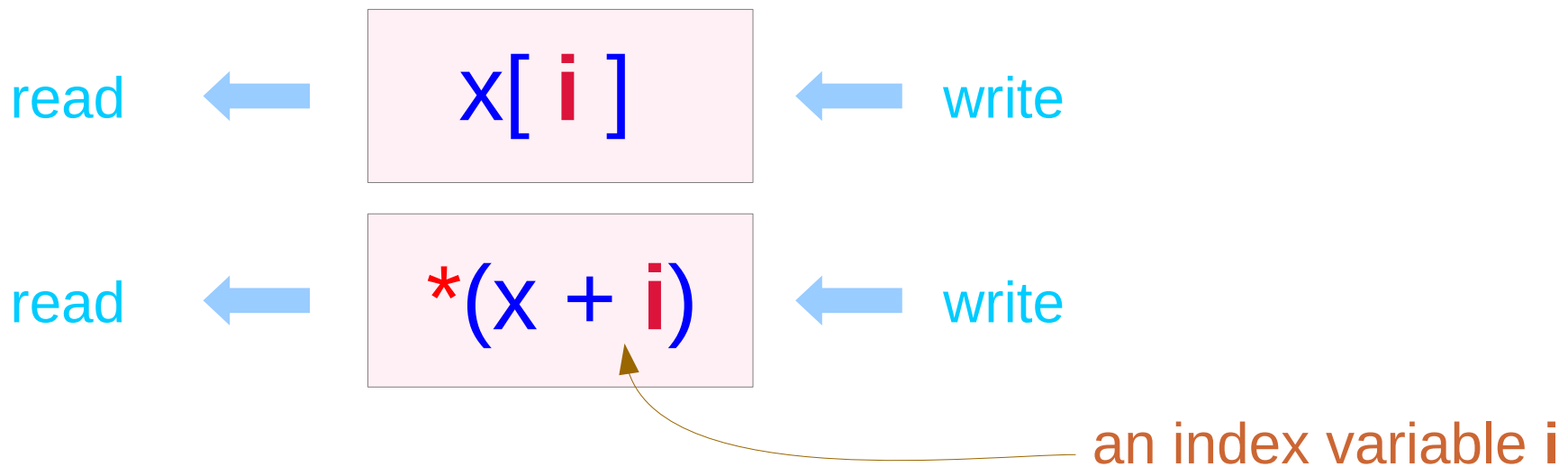
**x** : the starting address  
**x+i** : the address of the **(i+1)**<sup>th</sup> element  
**\*(x+i)** : the **(i+1)**<sup>th</sup> element variable

# A variable expressed by another variable

```
int    x[5];
```

**x** holds the starting address  
of 5 consecutive **int** variables

treated as an **int** variable



- 
- **Two aspects of a 1-d array variable**

## Two aspects of **x**

```
int    x[5];
```

**x** is an array of 5 integer elements

**Abstract data **x** ..... size**

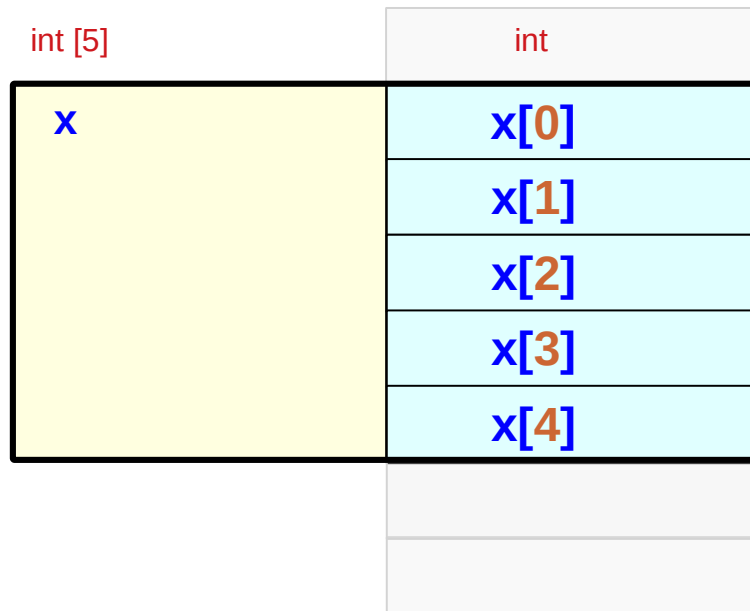
**Pointer **x** ..... starting address**

# Abstract data **x**

```
int    x[5];
```

the size of **x** is the total size of 5 consecutive `int` variables

**Abstract Data **x****

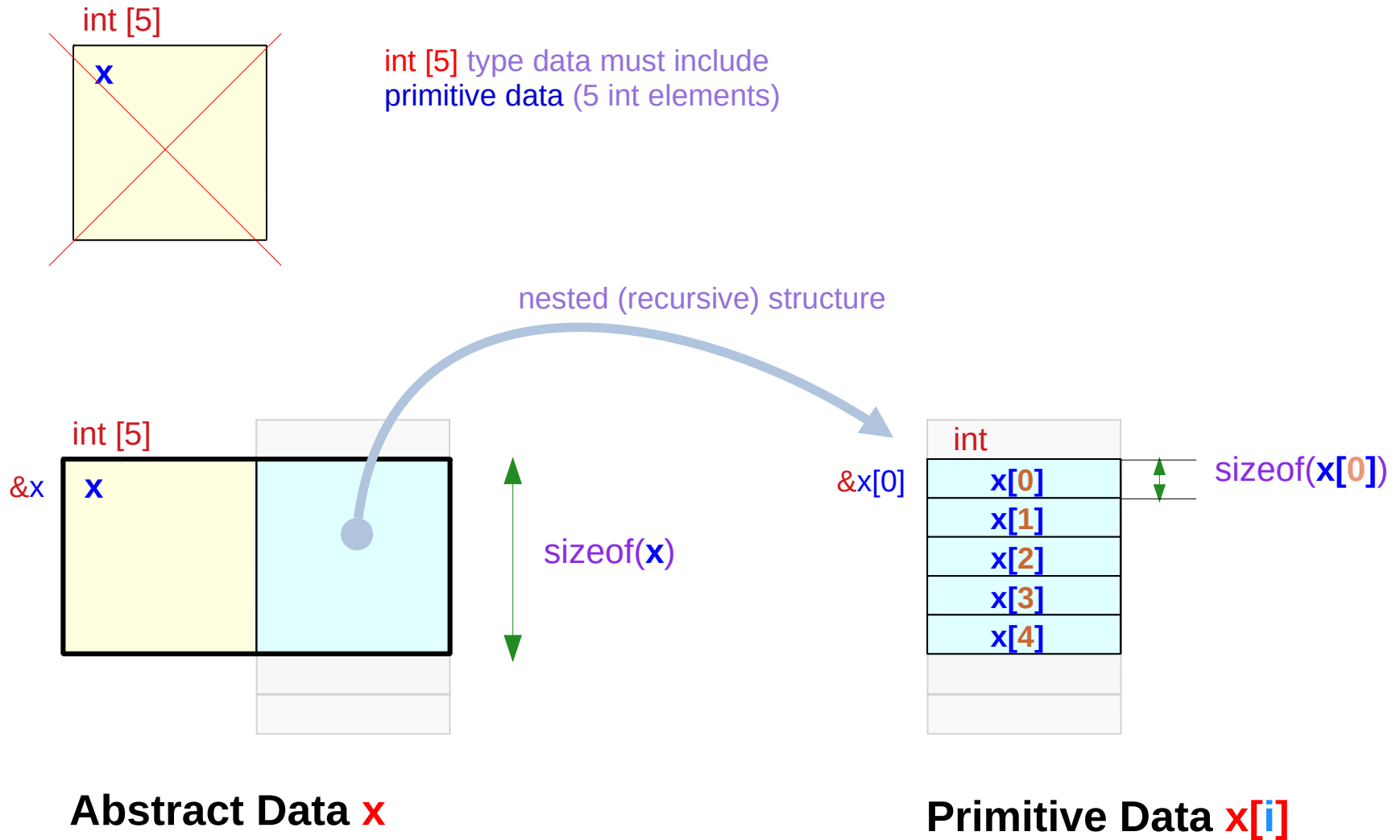


**size**

$$\text{sizeof}(\mathbf{x}) = 5 * \text{sizeof}(\text{int})$$



# Recursive structure



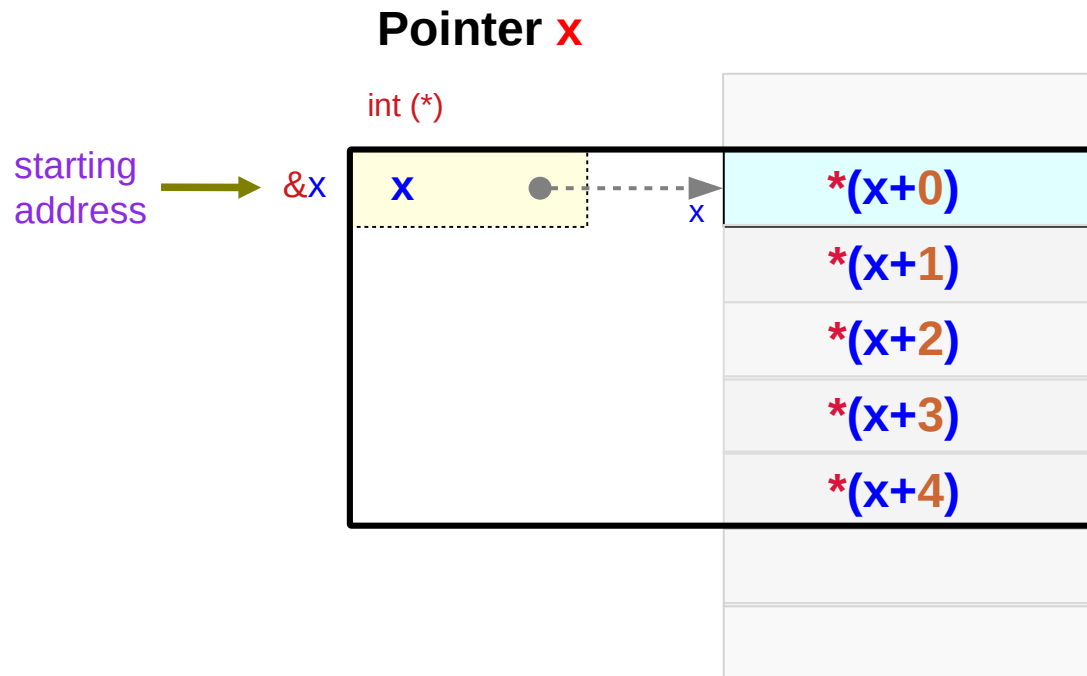
# Pointer x

```
int x[5];
```

x holds the starting address of 5 consecutive int variables

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

only for  $i = 0, \dots, 4$



**starting address**

pointer relation

$$x[0] \equiv *x$$
$$\&x[0] \equiv x$$

the starting address of 5 consecutive int variables

$$\underline{\underline{\text{value}(\&x) = \text{value}(x) = \text{value}(\&x[0])}}$$

$\rightarrow$  not a real pointer `x`

# Aggregate Data Type – 1-d Array

```
int    x[5];
```

**x** holds the starting address of 5 consecutive `int` variables

## Aggregate data with 5 elements

*the start address of each element :*

$\text{value}(\mathbf{x+0}) = \text{value}(\mathbf{x}) + 0 * \text{sizeof}(*\mathbf{x})$

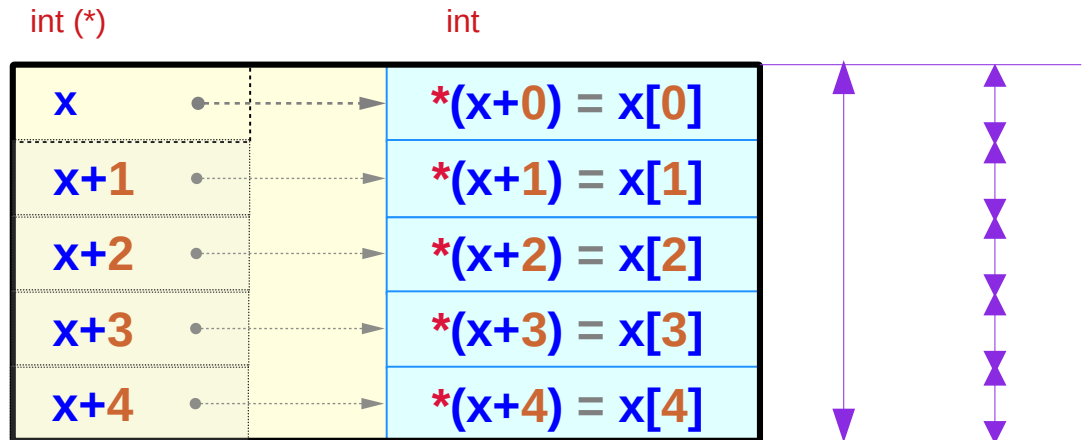
$\text{value}(\mathbf{x+1}) = \text{value}(\mathbf{x}) + 1 * \text{sizeof}(*\mathbf{x})$

$\text{value}(\mathbf{x+2}) = \text{value}(\mathbf{x}) + 2 * \text{sizeof}(*\mathbf{x})$

$\text{value}(\mathbf{x+3}) = \text{value}(\mathbf{x}) + 3 * \text{sizeof}(*\mathbf{x})$

$\text{value}(\mathbf{x+4}) = \text{value}(\mathbf{x}) + 4 * \text{sizeof}(*\mathbf{x})$

$\text{value}(\mathbf{x}) =$  *the start address of the array*



$\text{sizeof}(\mathbf{x}) = \text{sizeof}(*\mathbf{x}) * 5$

- 
- **An array  $x$  and a pointer  $p$**

# Address values of an array **x** and a pointer **p**

```
int x [5] ;
```

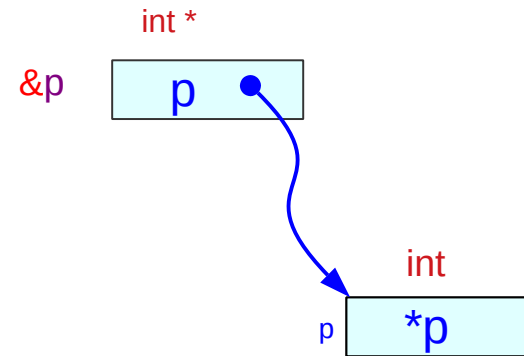
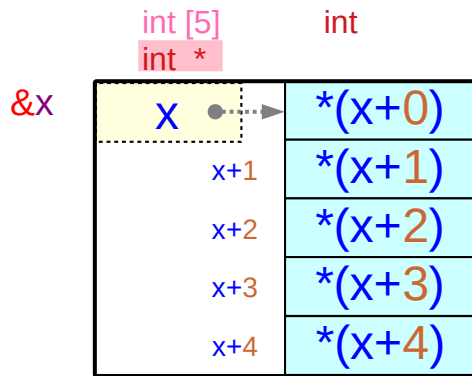
```
int * p ;
```

**x** : an array variable name (constant)

**p** : an variable name

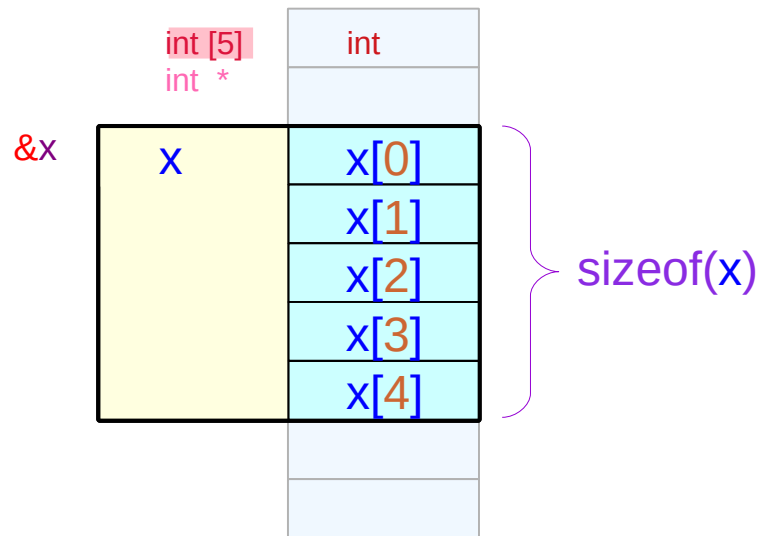
**value(x)** : the starting address of 5 consecutive **int** variables

**value(x)** : the address of an **int** variable



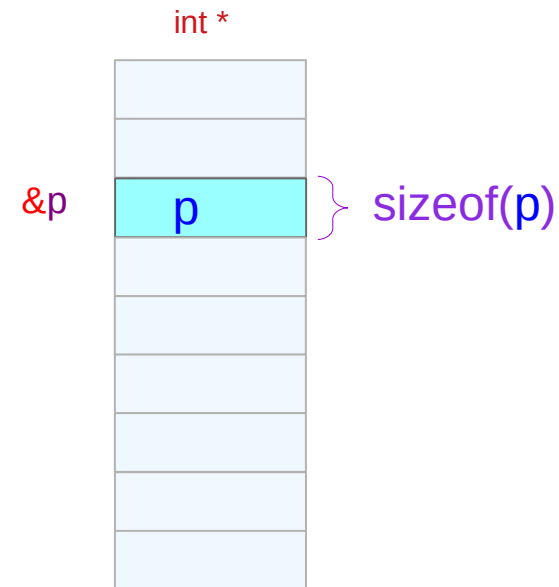
# Sizes of an array **x** and a pointer **p**

```
int x [5] ;
```



$\text{sizeof}(x) = 5 * \text{sizeof}(\text{int})$

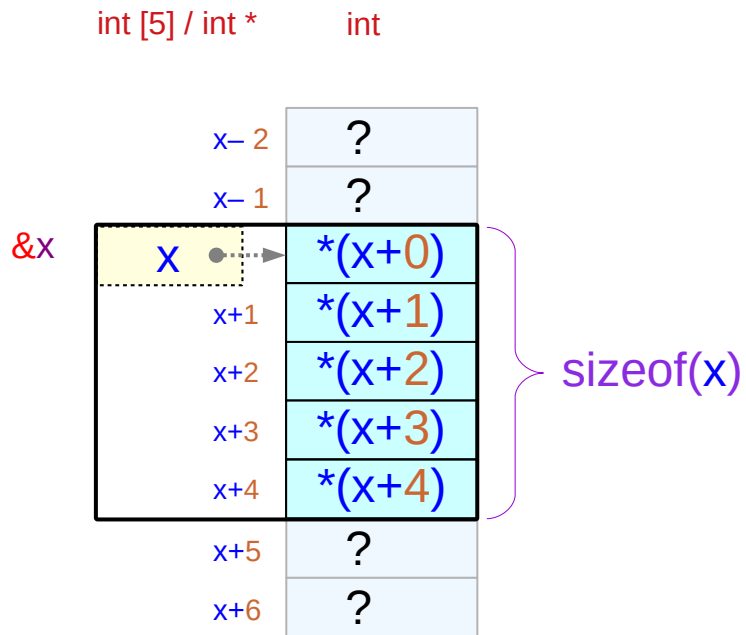
```
int * p ;
```



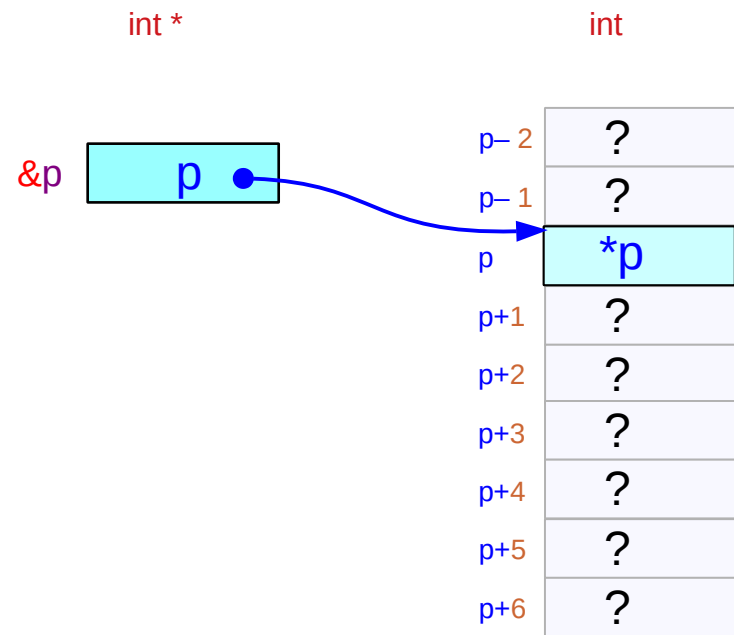
$\text{sizeof}(p) = \text{size of a pointer}$   
= 4 bytes (32-bit system)  
8 bytes (64-bit system)

# Out of range index

```
int x [5] ;
```



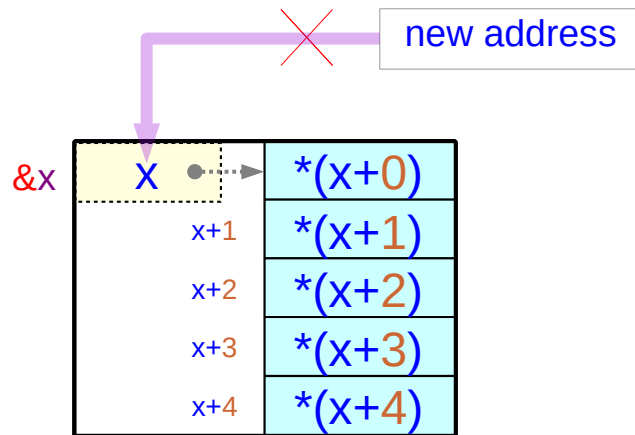
```
int * p ;
```



A programmer's responsibility

# Assignment of a new address

```
int x [5] ;
```



`x` is a constant variable

`x` and `&x` give the same value of `&x[0]`

This address is assigned by the compiler and cannot be changed

```
int * p ;
```



`p` is a pointer variable

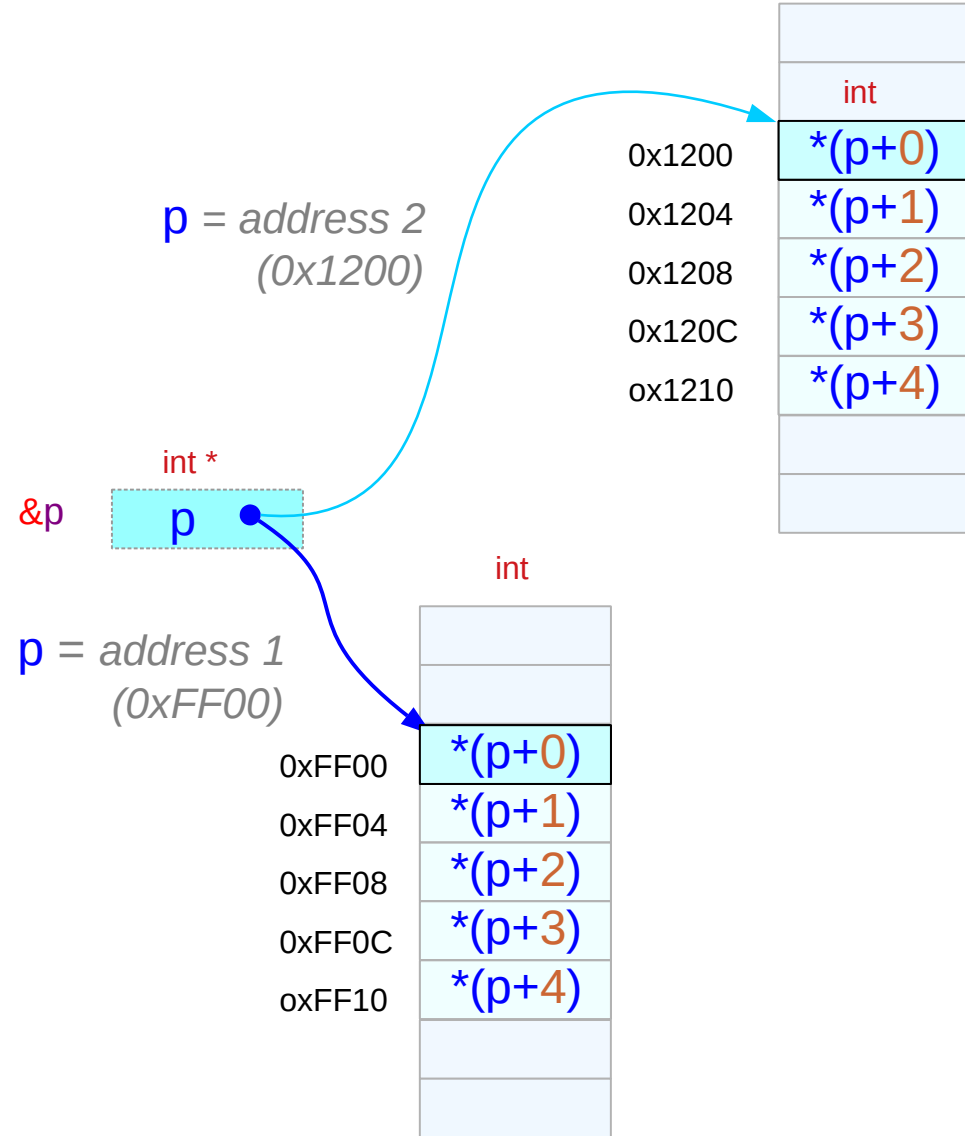
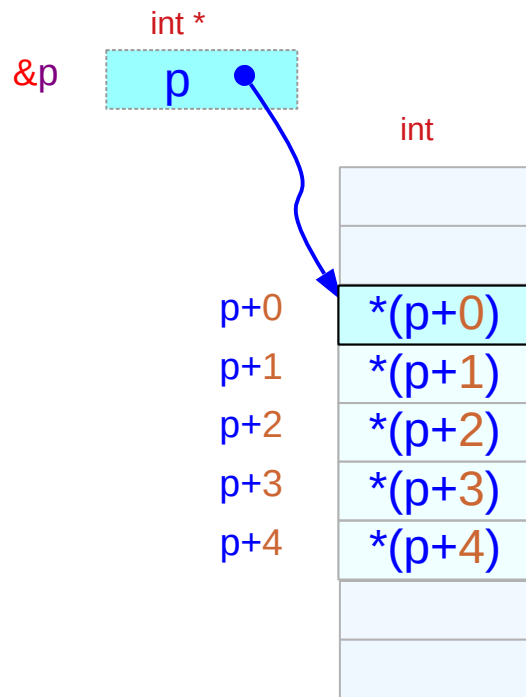
a memory location is allocated for a variable `p`

its value can be changed



# Pointer variable can point different locations

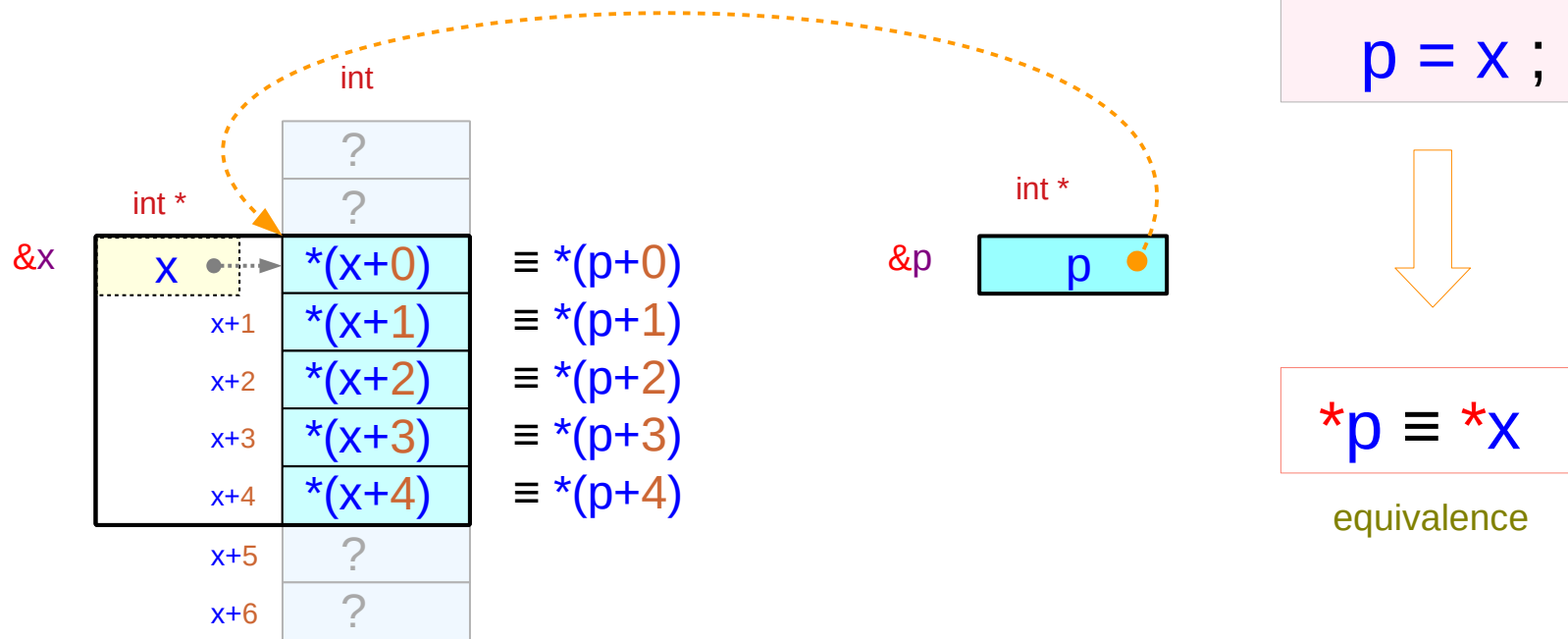
```
int * p ;
```



# Pointer to an array element

```
int x [5] ;
```

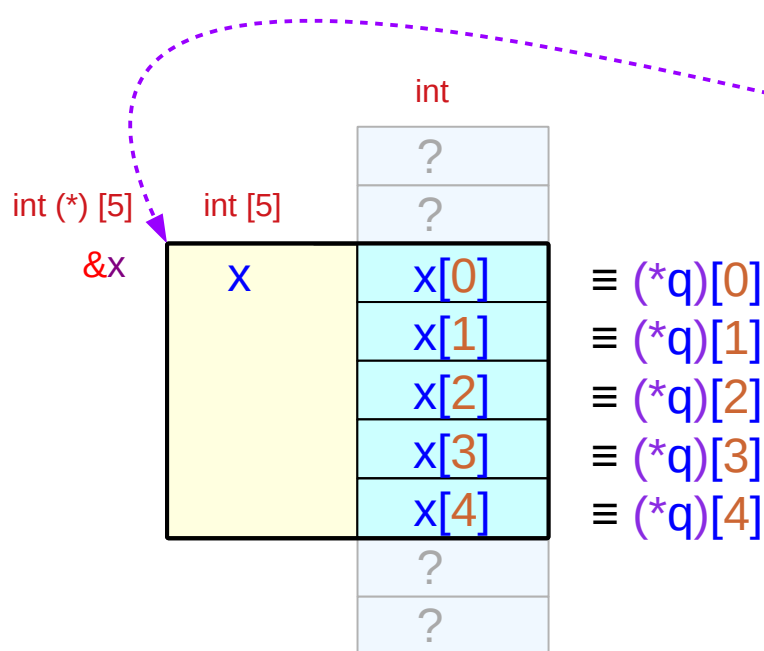
```
int * p ;
```



# Pointer to an array

```
int x [5] ;
```

```
int (*q) [5] ;
```



`q` is a pointer variable that points to an array with 5 integer variables

assignment

```
q = &x ;
```



```
(*q) ≡ x
```

equivalence

- 
- **Initializing an array**
  - **Copying and comparing arrays**

# Using arrays

initialization

```
int a [3] = { 1, 2, 3 };
```

≡

```
int a [] = { 1, 2, 3 };
```

accessing elements

```
a [0] = 100;
```

```
a [1] = 200;
```

```
a [2] = 300;
```

```
a [m] = 100 * m;
```

$m = 0, 1, 2$

a function argument

```
func( a );
```

```
func( int x [] ) { ... }
```



# Array initialization (1)

```
int a [5] ;
```

uninitialized values (garbage)

$a[0] = ?$ ,  $a[1] = ?$ ,  $a[2] = ?$ ,  $a[3] = ?$ ,  $a[4] = ?$ ;

```
int a [5] = { 1, 2, 3 } ;
```

$= \{ 1, 2, 3, 0, 0 \}$

$a[0] = 1$ ,  $a[1] = 2$ ,  $a[2] = 3$ ,  $a[3] = 0$ ,  $a[4] = 0$ ;

```
int a [5] = { 0 } ;
```

$= \{ 0, 0, 0, 0, 0 \}$

$a[0] = 0$ ,  $a[1] = 0$ ,  $a[2] = 0$ ,  $a[3] = 0$ ,  $a[4] = 0$ ;

all elements with zero

## Array initialization (2)

```
int a [5] = { 1, 2, 3, 4, 5 } ;
```

sizeof(a) = 5\*4 = 20 bytes

```
int b [ ] = { 1, 2, 3, 4 } ;
```

sizeof(b) = 5\*4 = 20 bytes



4

```
int b [ ] ;
```

must have initialization data

# Array initialization (3)

```
int c [3][4] = { { 1, 2, 3, 4},  
                { 5, 6, 7, 8},  
                {9,10,11,12} };
```

sizeof(c) = 3\*4\*4 = 48 bytes

```
c[0][0] = 1, c[0][1] = 2, c[0][2] = 3, c[0][3] = 4,  
c[1][0] = 5, c[1][1] = 6, c[1][2] = 7, c[1][3] = 8,  
c[2][0] = 9, c[2][1] = 10, c[2][2] = 11, c[2][3] = 12;
```

```
int d [ ][4] = { { 1, 2, 3, 4},  
                { 5, 6, 7, 8},  
                {9,10,11,12} };
```

sizeof(c) = 3\*4\*4 = 48 bytes

```
d[0][0] = 1, d[0][1] = 2, d[0][2] = 3, d[0][3] = 4,  
d[1][0] = 5, d[1][1] = 6, d[1][2] = 7, d[1][3] = 8,  
d[2][0] = 9, d[2][1] = 10, d[2][2] = 11, d[2][3] = 12;
```

```
int d [ ][ ] = { { 1, 2, 3, 4},  
                { 5, 6, 7, 8},  
                {9,10,11,12} };
```

Only the first dimension  
can be unsized



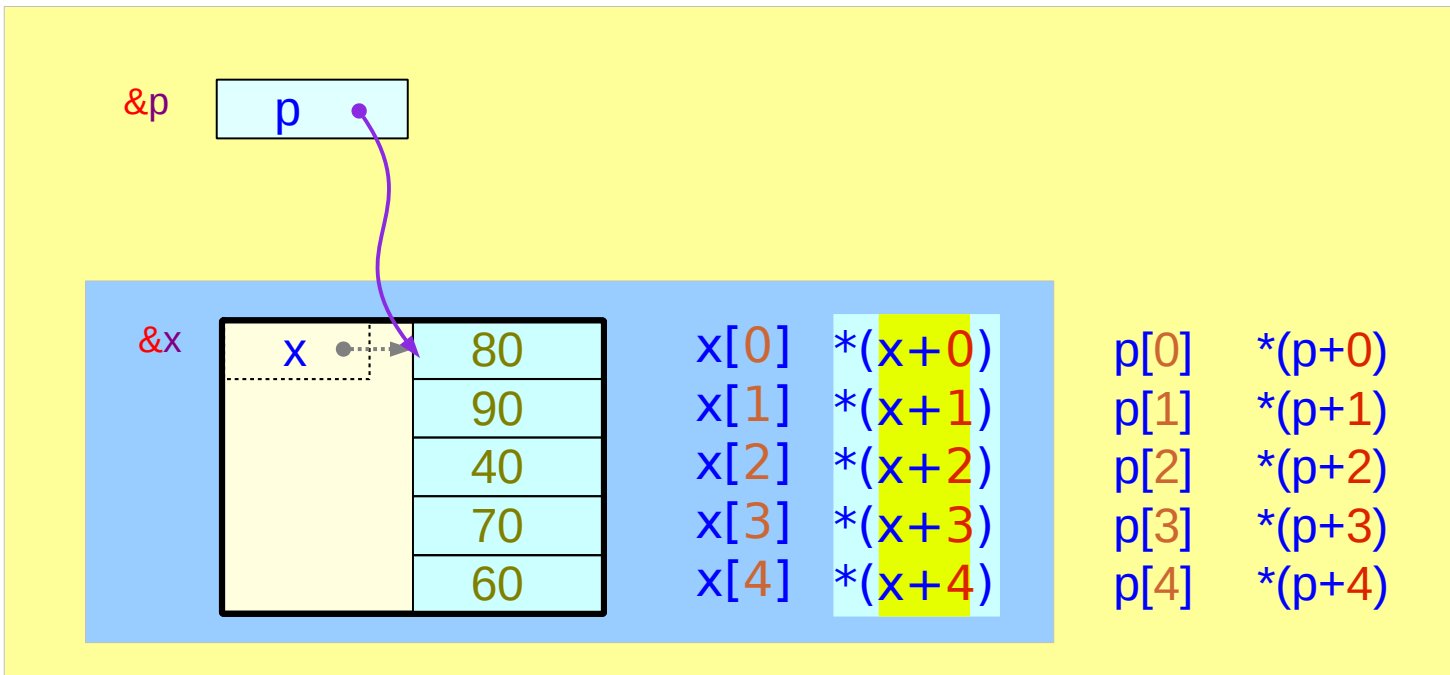
# Accessing an array with a pointer variable

```
int x [5] = { 80, 90, 40, 70, 60 } ;
```

```
int *p = x;
```

**x** is a constant variable and cannot be changed

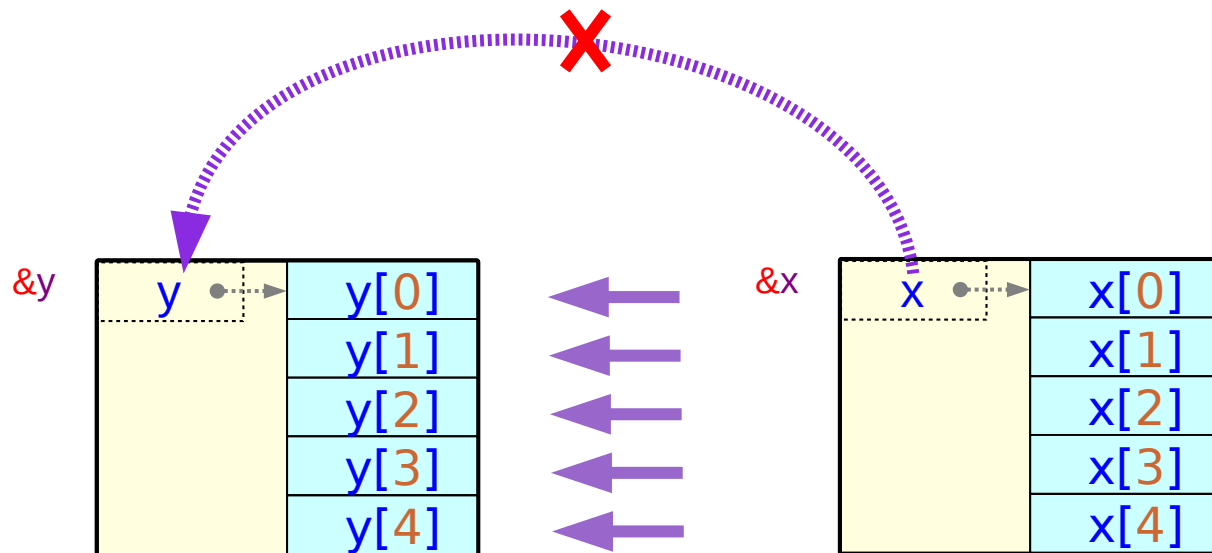
**p** is a variable can point to other addresses



# Copying an array to another array

```
int x [5] = { 1, 2, 3, 4, 5 };  
int y [5] ;  
y = x;
```

*y* is a constant variable and cannot be assigned (changed)



```
for (i=0; i<5; ++i)  
    y[i] = x[i];
```

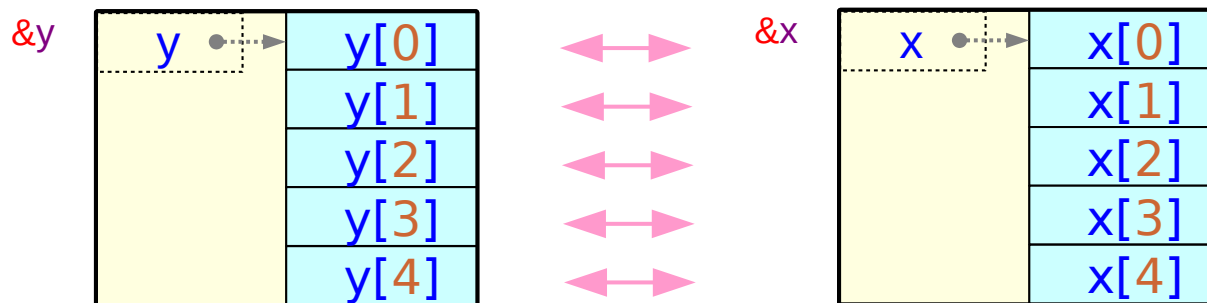
*must copy each element*

# Comparing an Array with another Array

```
int x [5] = { 1, 2, 3, 4, 5 };  
int y [5] = { 1, 2, 3, 4, 5 };  
x == y
```

```
EQ &= (y[i] == x[i]);
```

```
EQ = EQ & (y[i] == x[i]);
```



```
EQ=1;  
for (i=0; i<5; ++i)  
    EQ &= (y[i] == x[i]);
```

*must compare each element*

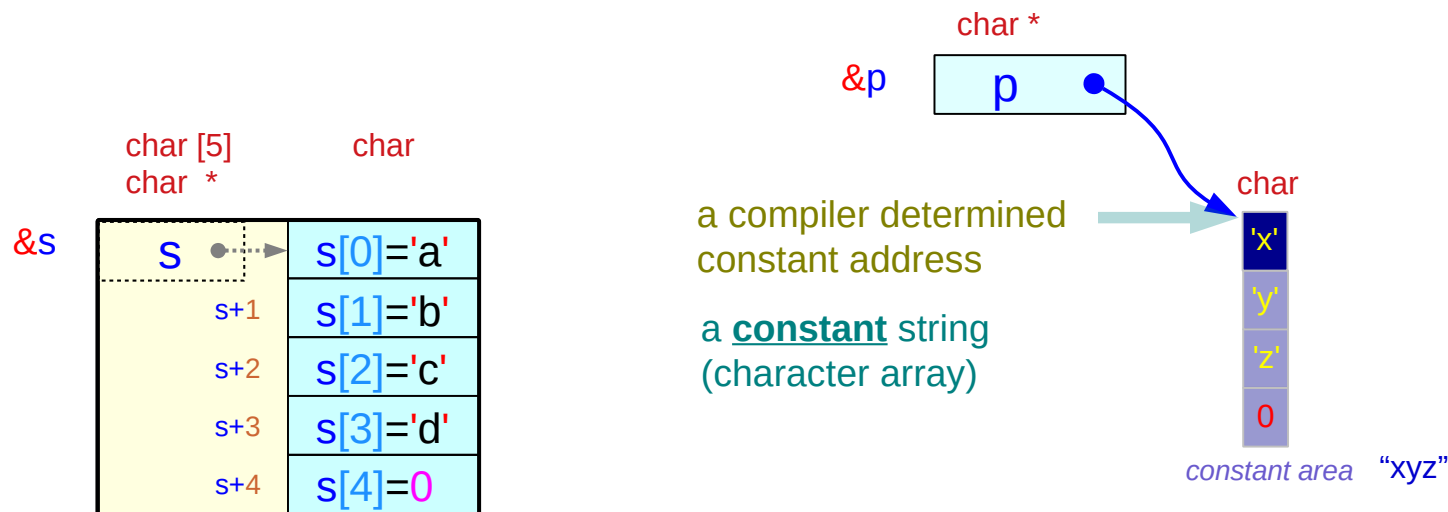
- 
- **A string and a character array**

# Initialized character arrays and pointers (1)

```
char s [5] = { 'a', 'b', 'c', 'd', 0 } ;
```

```
char s [5] = "abcd" ;
```

```
char *p = "xyz" ;
```



can change the value of any element

```
*s = 'm' ;  
s[0] = 'm' ;
```

cannot change the value of any element of a **constant** array

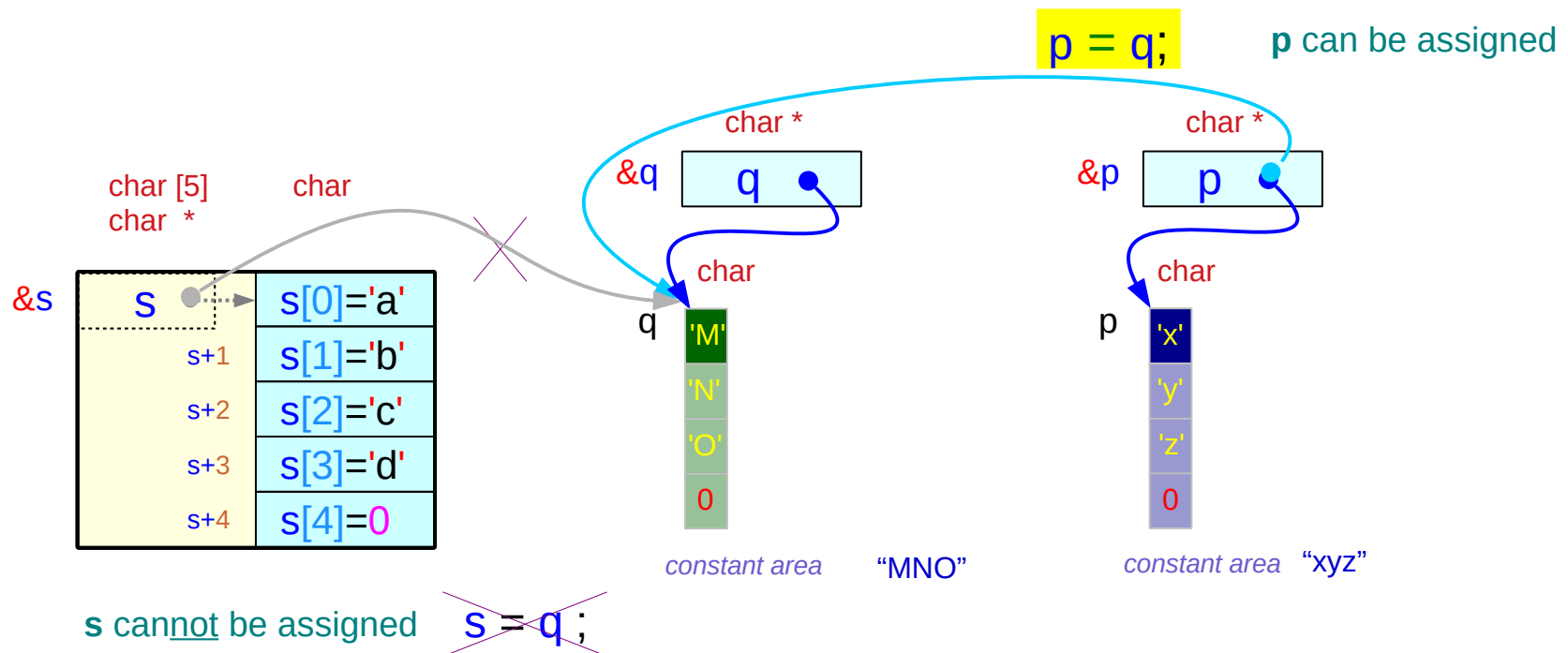
```
*p = 'm' ;  
p[0] = 'm' ;
```

# Initialized character arrays and pointers (2)

```
char s [5] = { 'a', 'b', 'c', 'd', 0 } ;
```

```
char s [5] = "abcd" ;
```

```
char *p = "xyz", *q = "MNO" ;
```

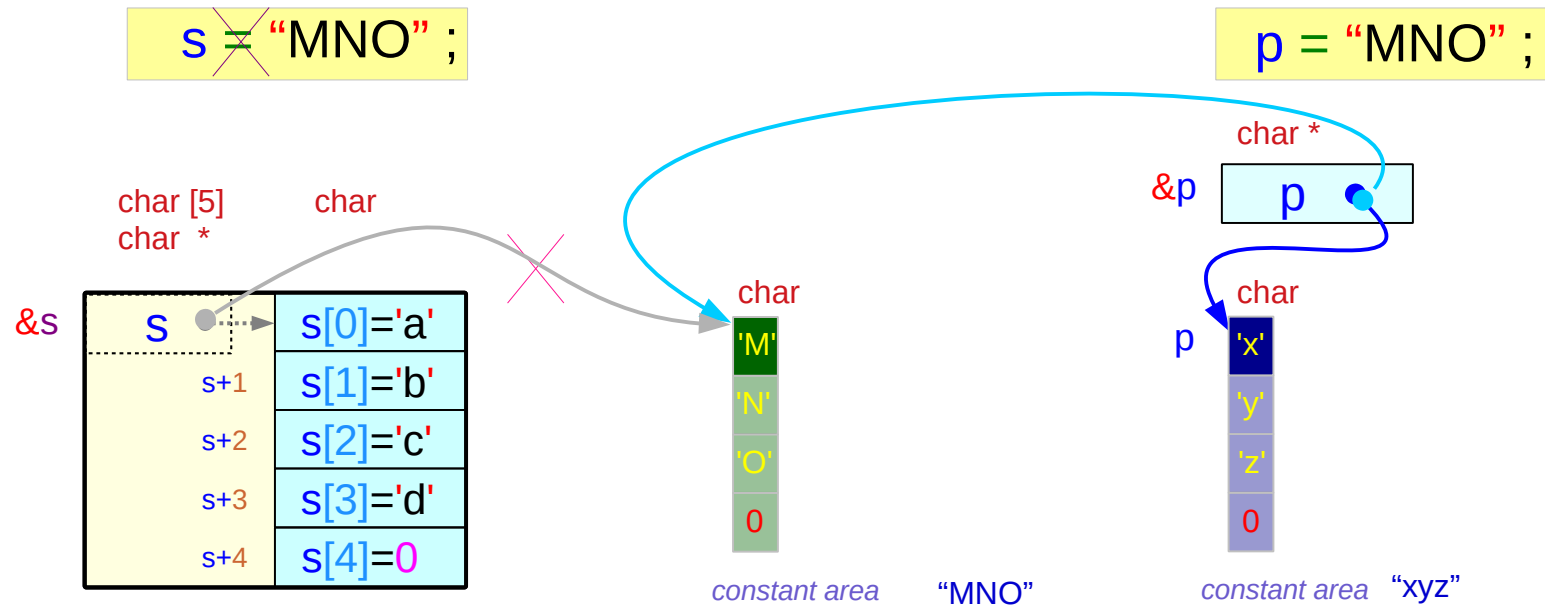


# Assigning a constant character string

```
char s [5] = { 'a', 'b', 'c', 'd', 0 } ;
```

```
char s [5] = "abcd" ;
```

```
char *p = "xyz" ;
```



# Copying a string

```
char s [5] = { 'a', 'b', 'c', 'd', 0 } ;
```

```
char s [5] = "abcd" ;
```



```
char *p = "xyz" ;
```

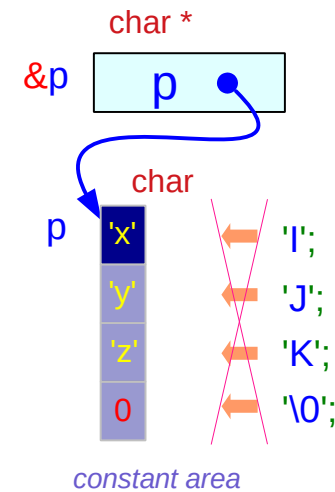
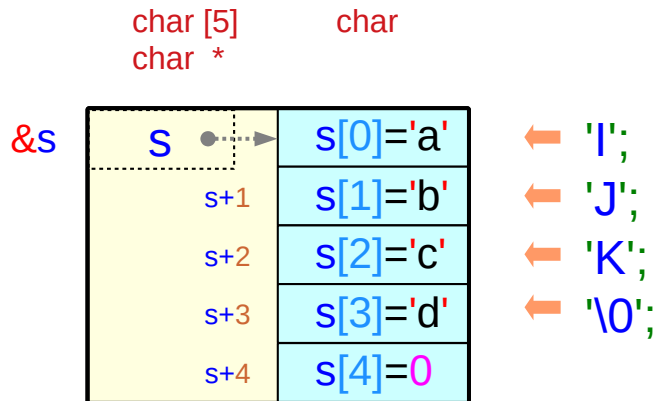
```
strcpy (s, "IJK") ;
```

s points to non-constant string

```
strcpy (p, "IJK") ;
```



p: points to a constant string





# Character arrays and a string pointer

```
char s [5] = "mnop" ;  
const char *p = "MNO" ;
```

```
s = "xyz" ;
```

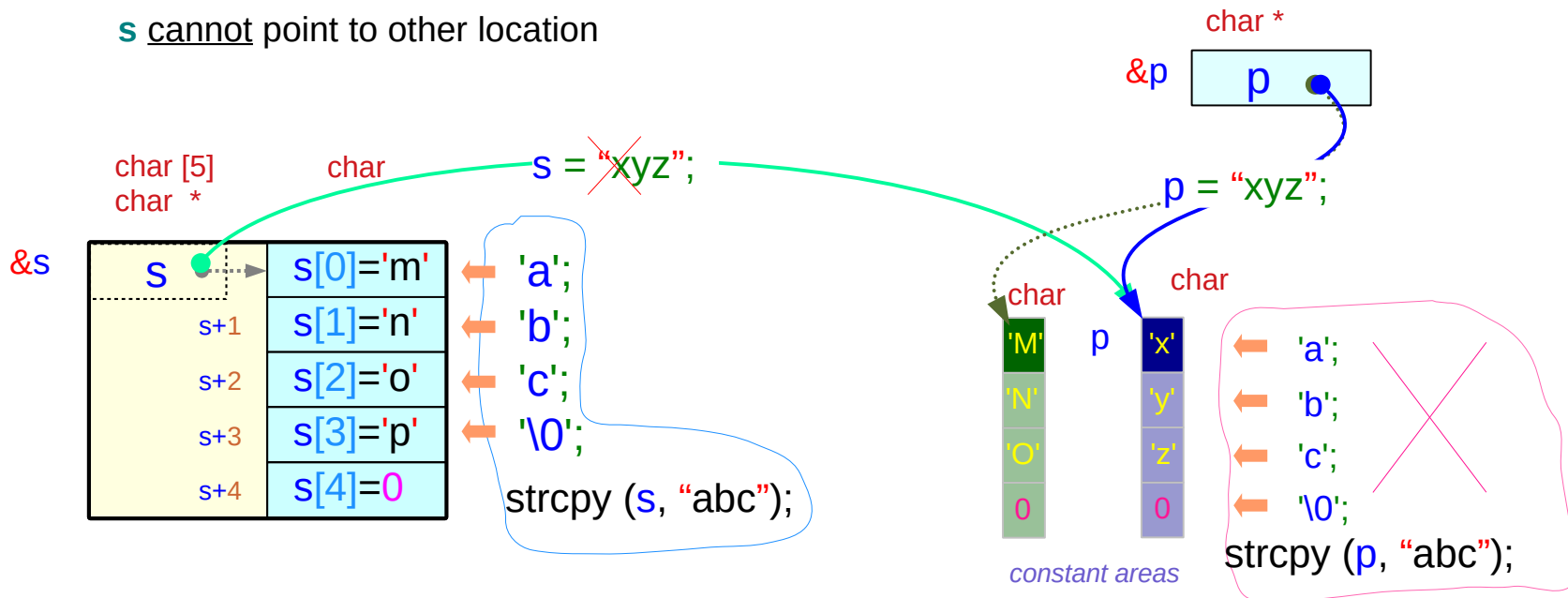
```
p = "xyz" ;
```

```
strcpy (s, "abc") ;
```

```
strcpy (p, "abc") ;
```

`char * const s`

`s` cannot point to other location

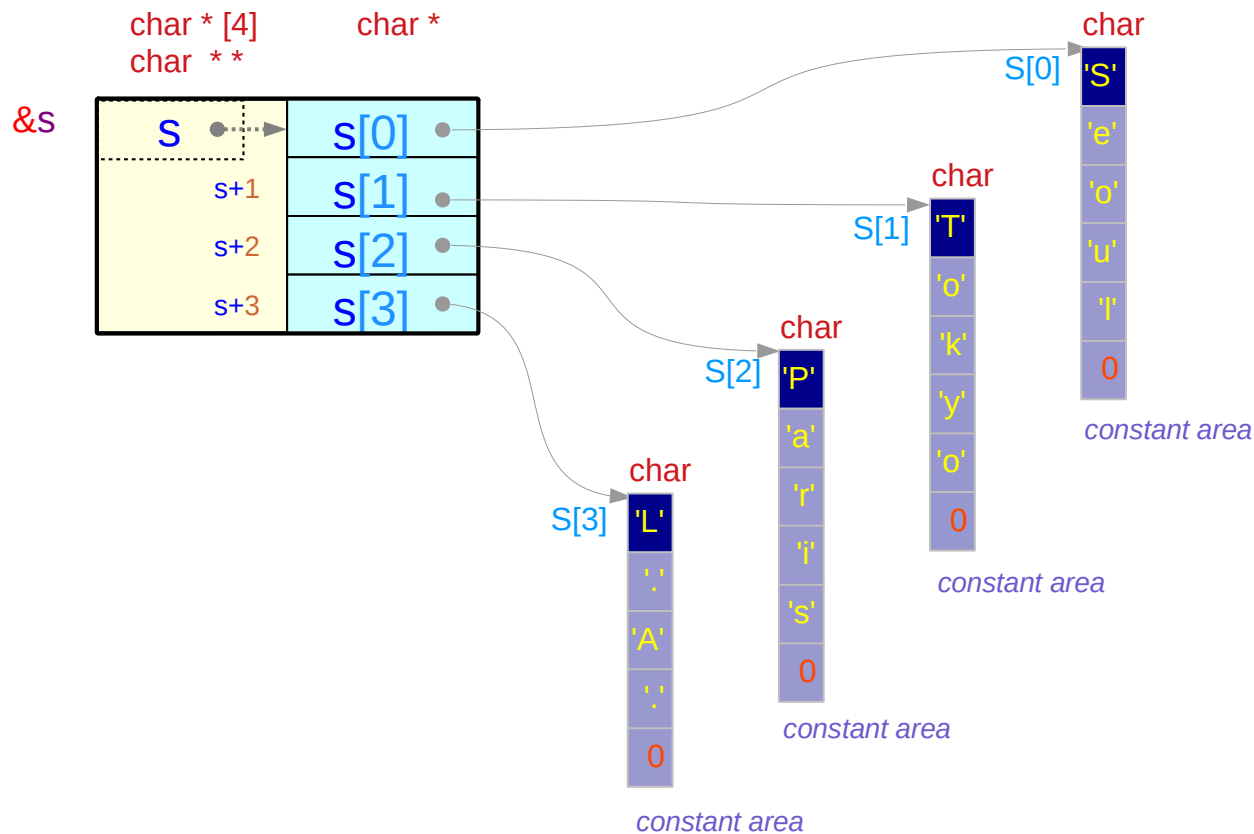


`const char * p`

`p` points to a string constant  
which cannot be changed

# Arrays of Pointers

```
char * S [4] = { "Seoul", "Tokyo", "Paris", "LA" } ;
```



A possible memory layout  
(little endian system)

MSB			LSB
'u'	'o'	'e'	'S'
'o'	'T'	'o'	'l'
'o'	'o'	'y'	'k'
'i'	'r'	'a'	'P'
'l'	'L'	'o'	's'
	'o'	'A'	

- 
- **Unsize array notations for 1-d arrays**

# Unsize array notation `x[ ]`

## 1. An array definition with **initializers**

```
int x [ ] = { 1, 2, 3 } ;
```



```
int x [3] ;
```

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

## 2. A formal **parameter** definition in a function

```
func( int x [ ] ) { ... }
```



```
int (*x)
```

compatible



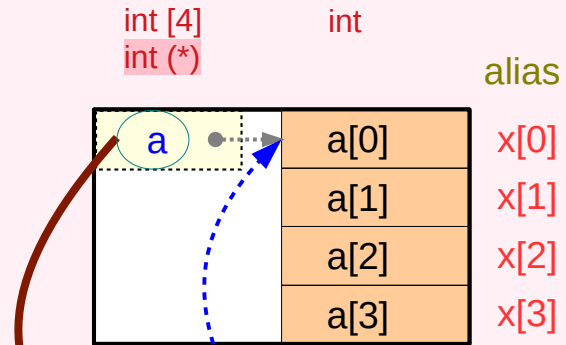
```
int * x
```

# Passing 1-d Arrays – using 0-d array pointer

```
int a[4] = { 1, 2, 3, 4 };
```

```
func( a );
```

```
func( int x [ ] ) {  
    ...  
}  
or  
func( int (*x) ) {  
    ...  
}
```



can change the original array **a** elements by the alias **x**

# Passing 1-d Arrays – using 1-d array pointer

```
int a[4] = { 1, 2, 3, 4 };
```

```
func( &a );
```

```
func( int x [ ][4] ) {
```

```
    ...
```

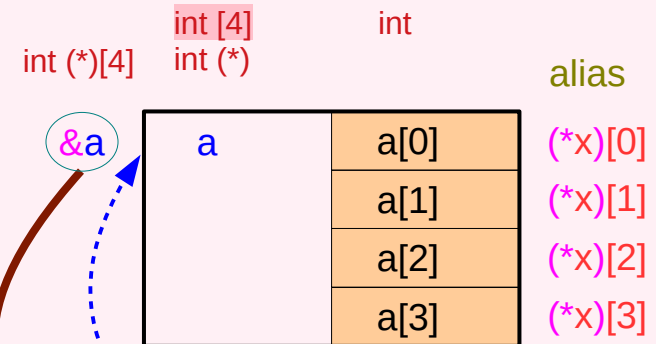
```
}
```

or

```
func( int (*x) [4] ) {
```

```
    ...
```

```
}
```




```
&x x = &a
```

can change the original array `a` elements by using the alias `(*x)`

# Passing an individual element by value

```
int a[4] = { 1, 2, 3, 4 };  
  
func(a[3]);
```




```
func(int x) {  
    ...  
}
```

int  
&x    x = a[3]

cannot change the original array element a[3]

```
int a[4] = { 1, 2, 3, 4 };  
  
func(&a[3]);
```



```
func(int *x) {  
    ...  
}
```

int \*  
&x    x = &a[3]


can change the original array element a[3] by the alias \*x

- 
- **Unsize array notations for 2-d arrays**





# Unsize array notation `x[ ][N]`

## 1. An array definition with **initializers**

```
int x[ ][3] = { {1, 2, 3}, {4, 5, 6} };  int x[2][3] ;  
x[0][0] = 1, x[0][1] = 2, x[0][2] = 3,  
x[1][0] = 4, x[1][1] = 5, x[1][2] = 6 ;
```

## 2. A formal **parameter** definition in a function

```
func( int x[ ][3] ) { ... }  int (*x)[3]  
  
not compatible  int ** p
```

# Passing 2-d Arrays – using 1-d array pointer

```
int b[2][3] = { {1, 2, 3}, {4, 5, 6} };
```

```
func( b );
```

```
func( int y [ ][3] ) {
```

```
    ...
```

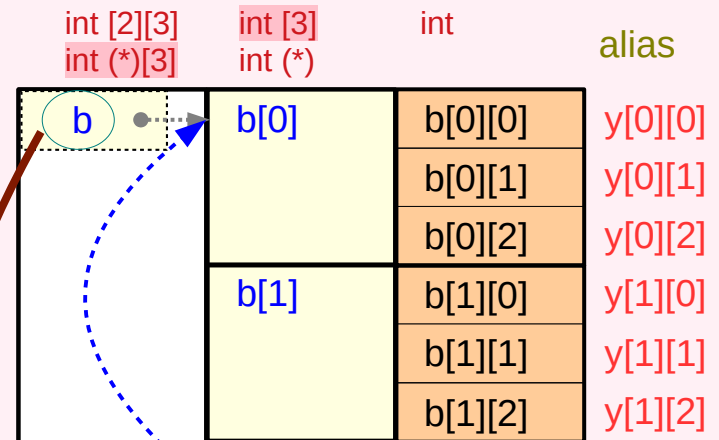
```
}
```

```
or
```

```
func( int (*y) [3] ) {
```

```
    ...
```

```
}
```



```
&y y = b
```

can change the original array **b** elements by using the alias **y**

# Passing 2-d Arrays – using 2-d array pointer

```
int b[2][3] = { {1, 2, 3}, {4, 5, 6} };
```

```
func( &b );
```

```
func( int y [ ][2][3] ) {
```

```
    ...
```

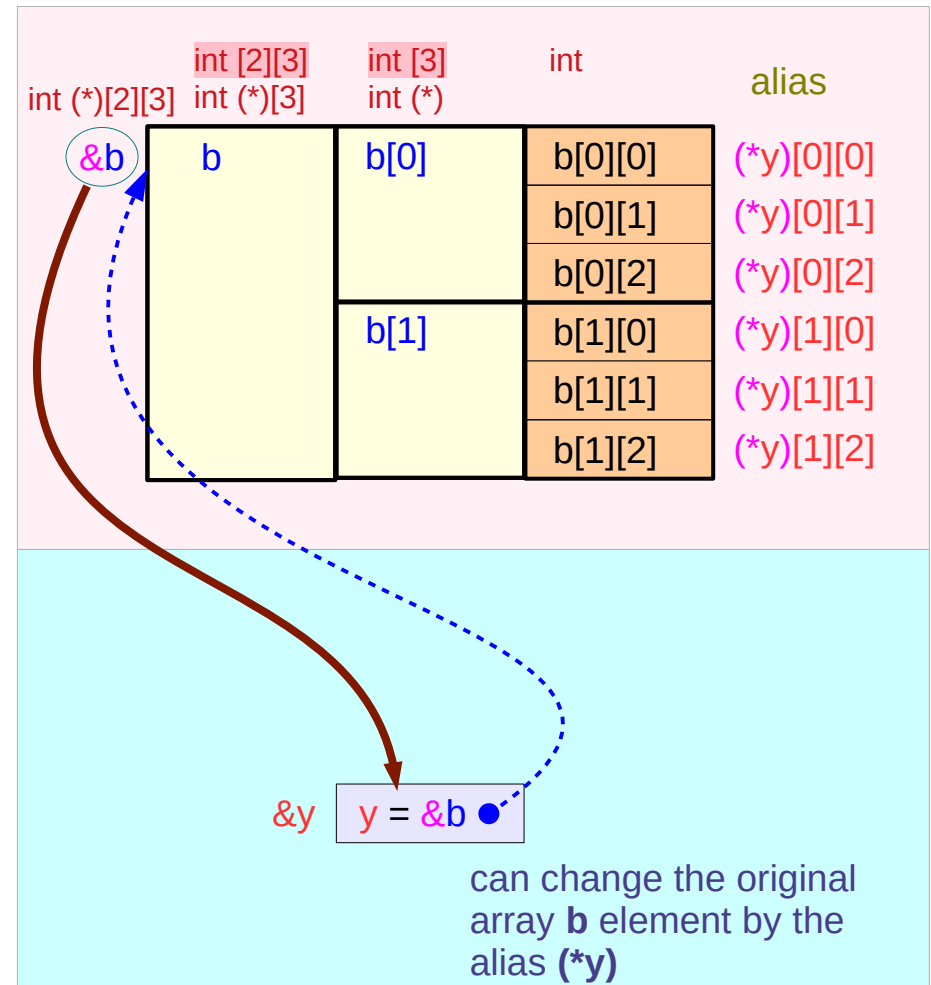
```
}
```

```
or
```

```
func( int (*y) [2][3] ) {
```

```
    ...
```

```
}
```



# Passing an individual element by reference

```
int b[2][3] = { {1, 2, 3}, {4, 5, 6} };  
  
func( b[0][1] );  
  
func( int y ) {  
    ...  
}
```

$\&y$  `y = b[0][1]`

cannot change the original array element `b[0][1]`

```
int b[2][3] = { {1, 2, 3}, {4, 5, 6} };  
  
func( &b[0][1] );  
  
func( int *y ) {  
    ...  
}
```

$\&y$  `y = &b[0][1]`

can change the original array element `b[0][1]` the alias `*y`

- 
- **Type definitions and 2-d arrays**

# Type definitions

```
typedef int int_type ;
```

type alias **int\_type**      variable **i**

```
int_type i;
```



```
int i ;
```

```
typedef int * iptr_type ;
```

type alias **iptr\_type**      variable **ip**

```
iptr_type ip;
```



```
int * ip ;
```

```
typedef int arr_type [4] ;
```

type alias **arr\_type**      variable **a**

```
arr_type a;
```



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

# Array Type Definition

```
typedef int arr_type [4];
```

```
arr_type a;
```

≡

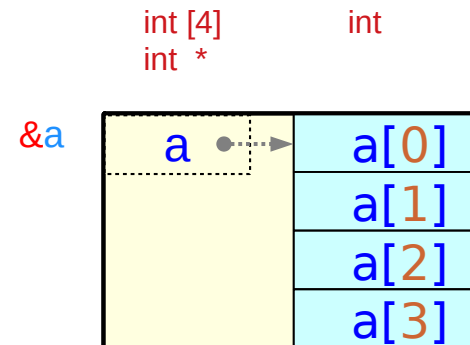
```
int a [4];
```

```
a [0] = 100;
```

```
a [1] = 200;
```

```
a [2] = 300;
```

```
a [3] = 400;
```



# Pointer to Array Type Definition

```
typedef int arr_type [4];  
arr_type a, *q;
```

≡

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

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

≡

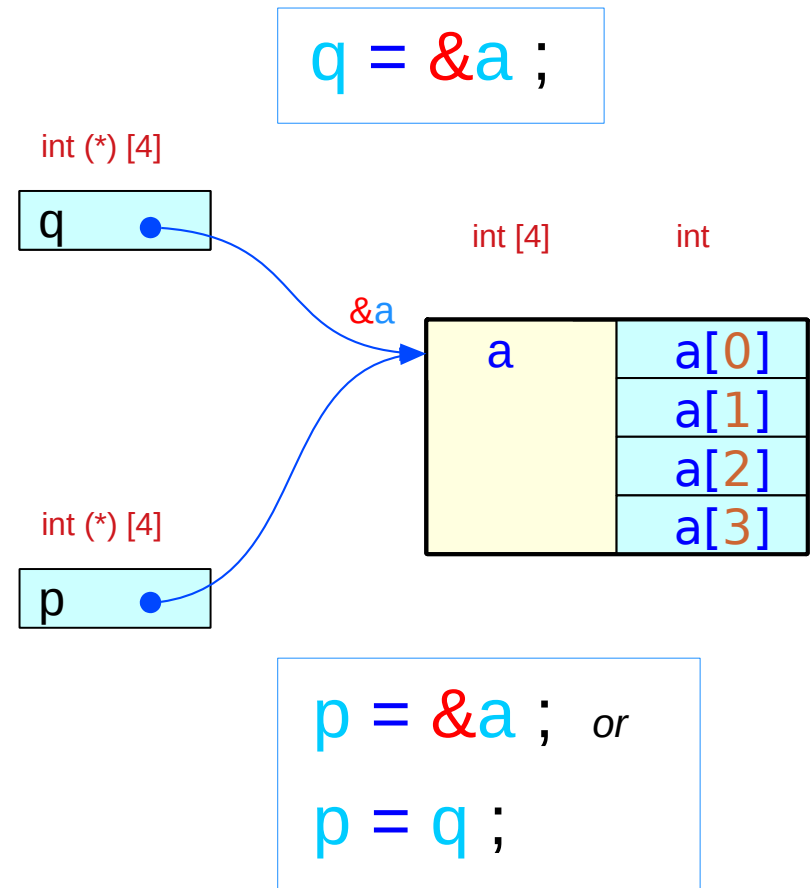
```
int (*p) [4];
```



# Pointer to Array Type Assignment

```
typedef int arr_type [4];  
arr_type a, *q;
```

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

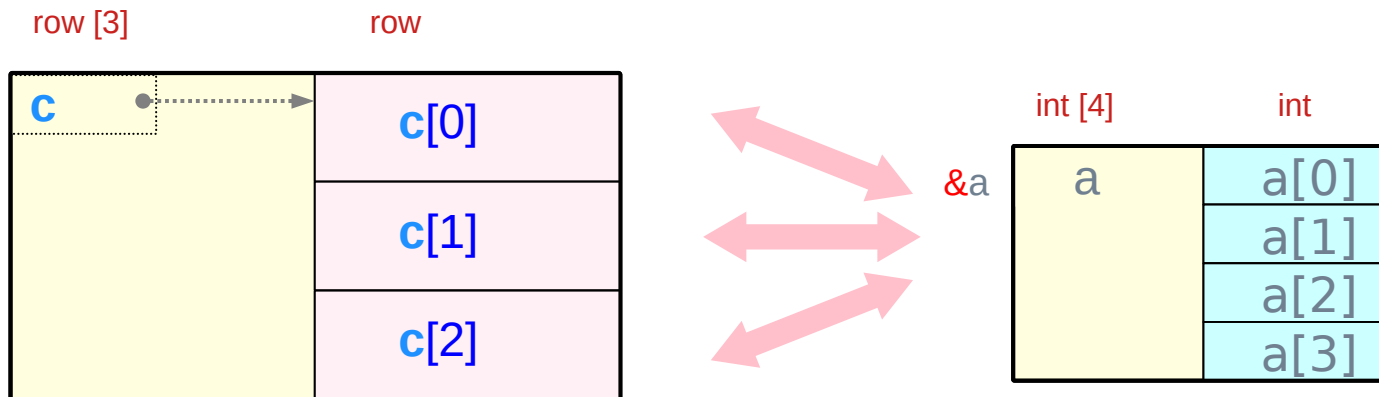


# Nested array declared explicitly

```
typedef int row [4] ;  
row c [3] ;
```

≡

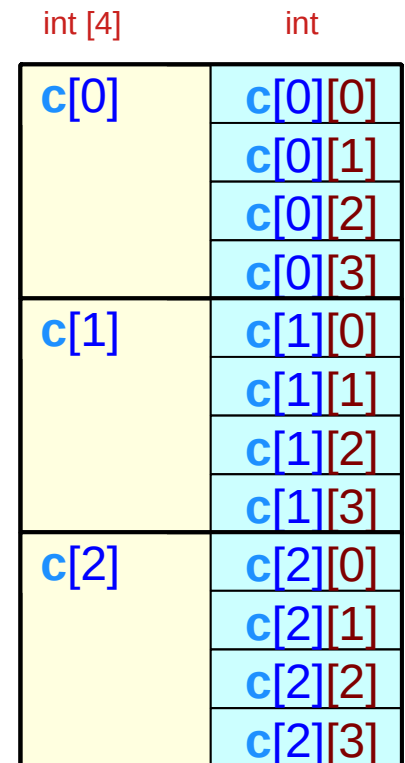
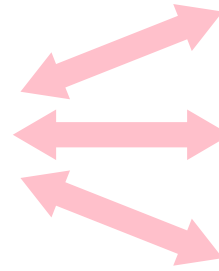
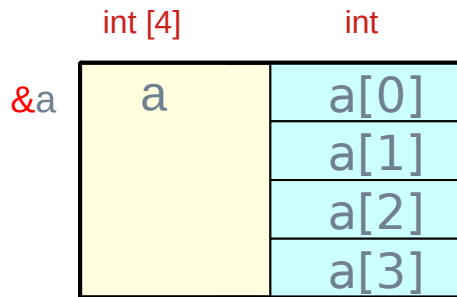
```
int c [3] [4] ;
```



each element **c[i]** has the type of **row (int [4])**

# Nested array declared explicitly

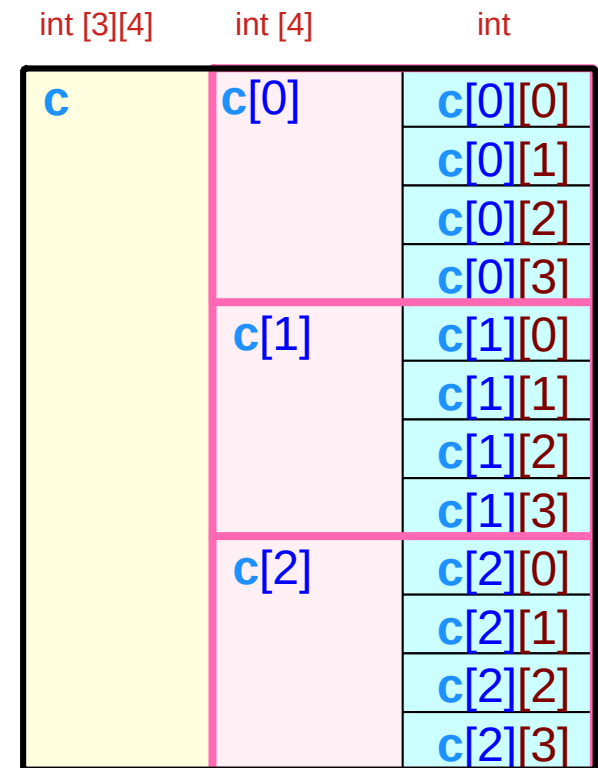
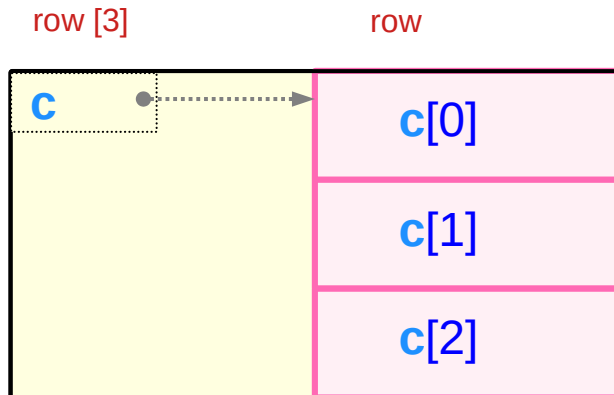
```
typedef int row [4] ;  
row c [3] ;
```



each element  $c[i]$  has the type of row (**int [4]**)

# Nested array declared explicitly

```
typedef int row [4] ;  
row c [3] ;
```



- 
- **2-d arrays and element addresses**

## 2-d array definition

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

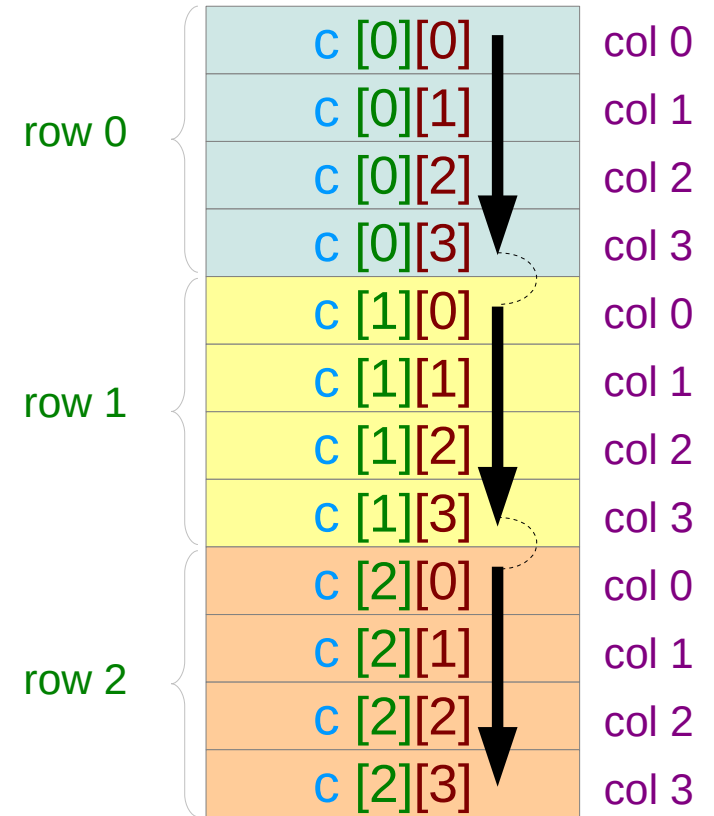
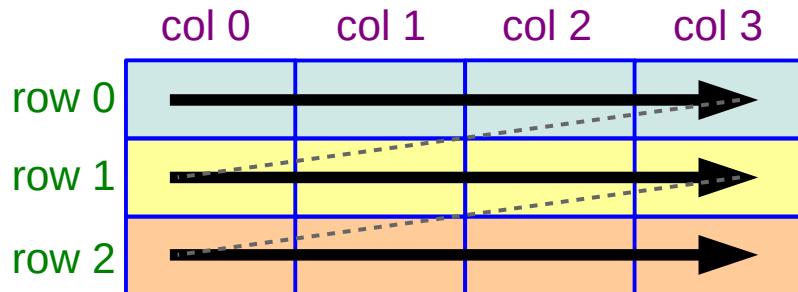
A matrix view

	col 0	col 1	col 2	col 3
row 0	c [0][0]	c [0][1]	c [0][2]	c [0][3]
row 1	c [1][0]	c [1][1]	c [1][2]	c [1][3]
row 2	c [2][0]	c [2][1]	c [2][2]	c [2][3]

# 2-d array stored as a linear array

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

row major order



# Element address $c[i] + j$

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

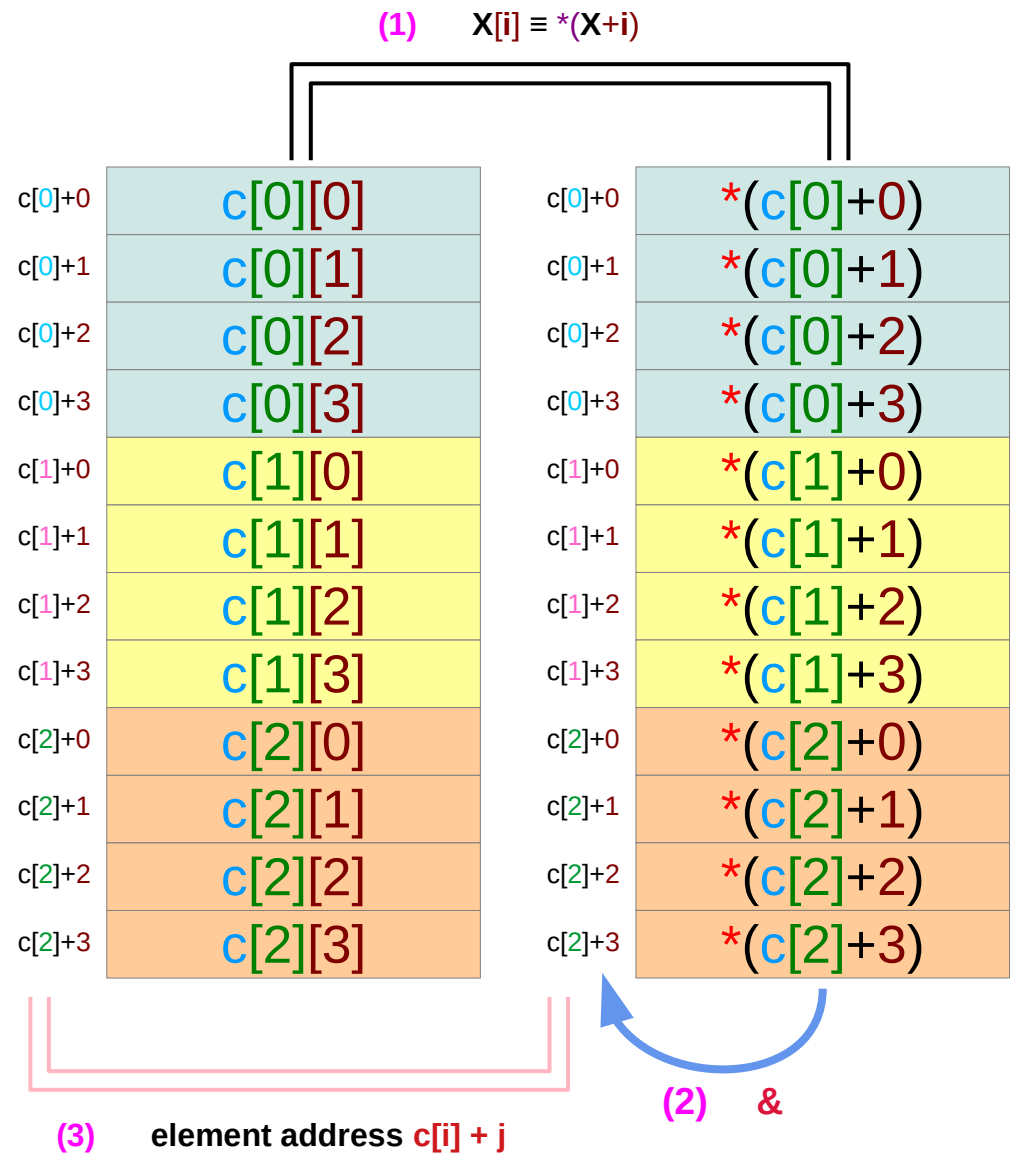
$$X[j] \equiv *(X+j)$$

$$\text{let } X \equiv c[i] \quad (1)$$

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

$$\&c[i][j] \equiv c[i]+j \quad (3)$$

the address of  $c[i][j]$  is  $c[i]+j$





# Row address $c[i]$

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

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

the address of  $c[i][j]$  is  $c[i]+j$

↓ let  $j = 0$

$$c[i][0] \equiv *c[i]$$

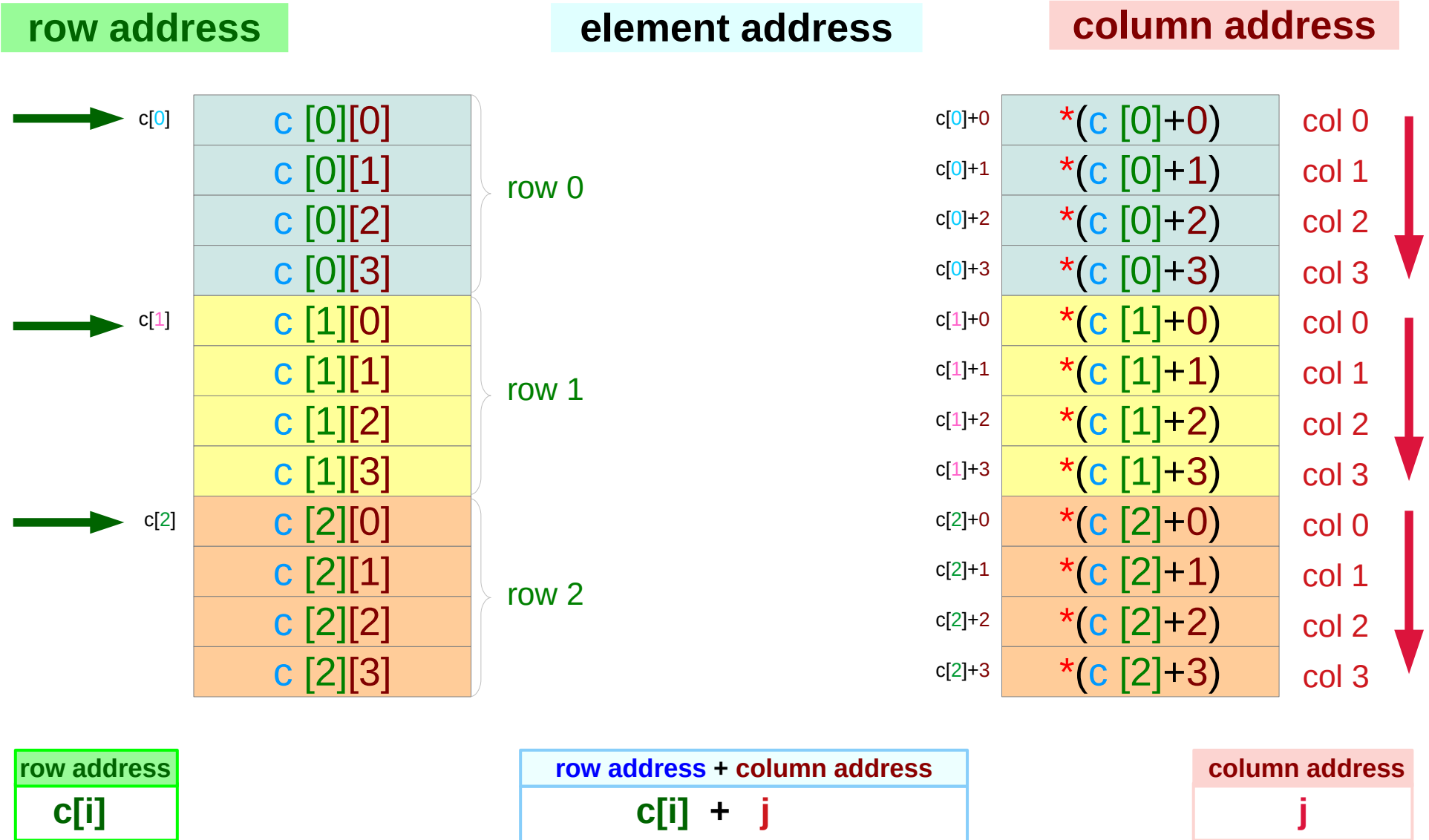
$$\&c[i][0] \equiv c[i]$$

row address : the address of the 1<sup>st</sup> element of each row

$c[0]$	$c[0][0]$
	$c[0][1]$
	$c[0][2]$
	$c[0][3]$
$c[1]$	$c[1][0]$
	$c[1][1]$
	$c[1][2]$
	$c[1][3]$
$c[2]$	$c[2][0]$
	$c[2][1]$
	$c[2][2]$
	$c[2][3]$

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

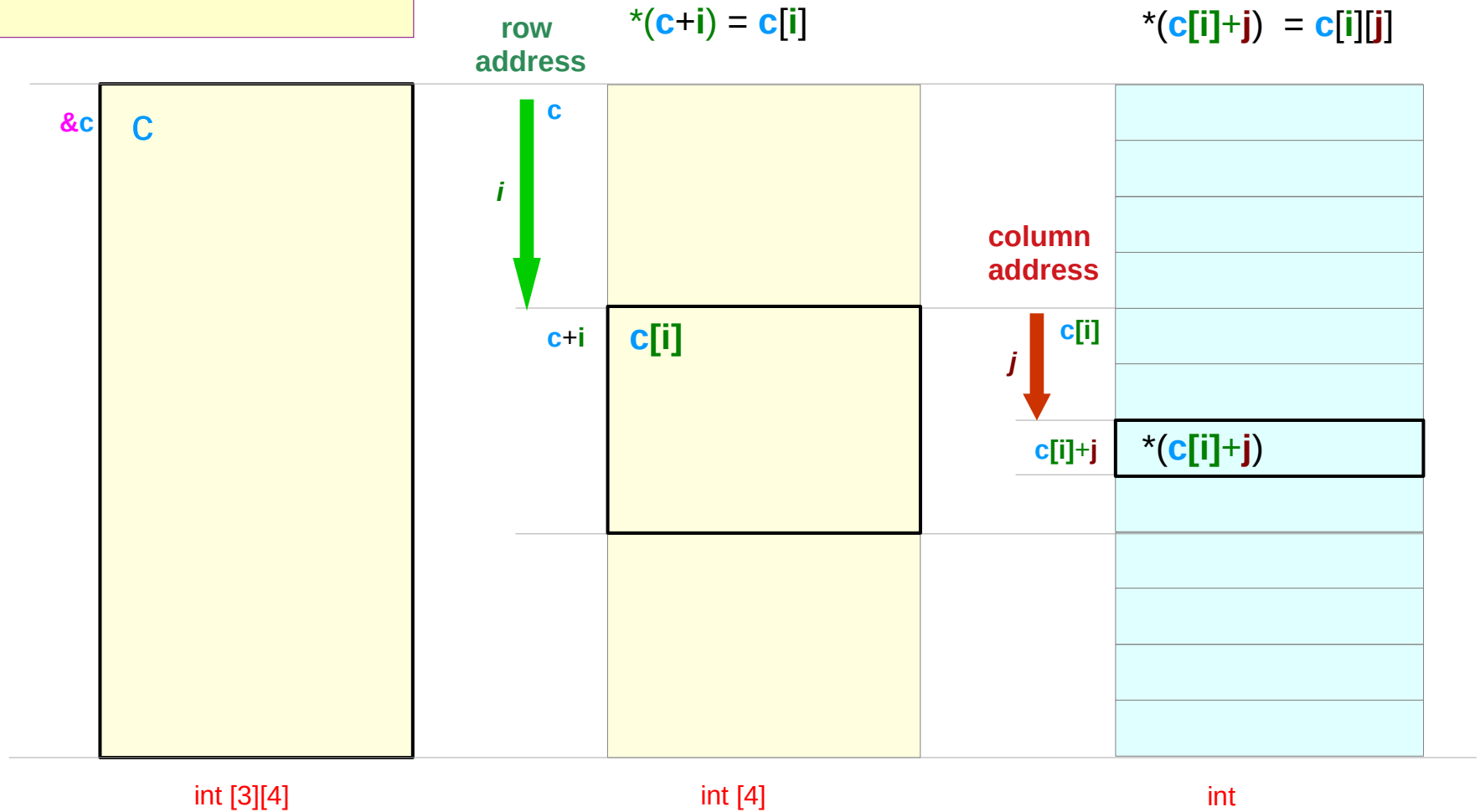
# Row Address $c[i]$ and Column Address $j$



# Element address $c[i] + j$

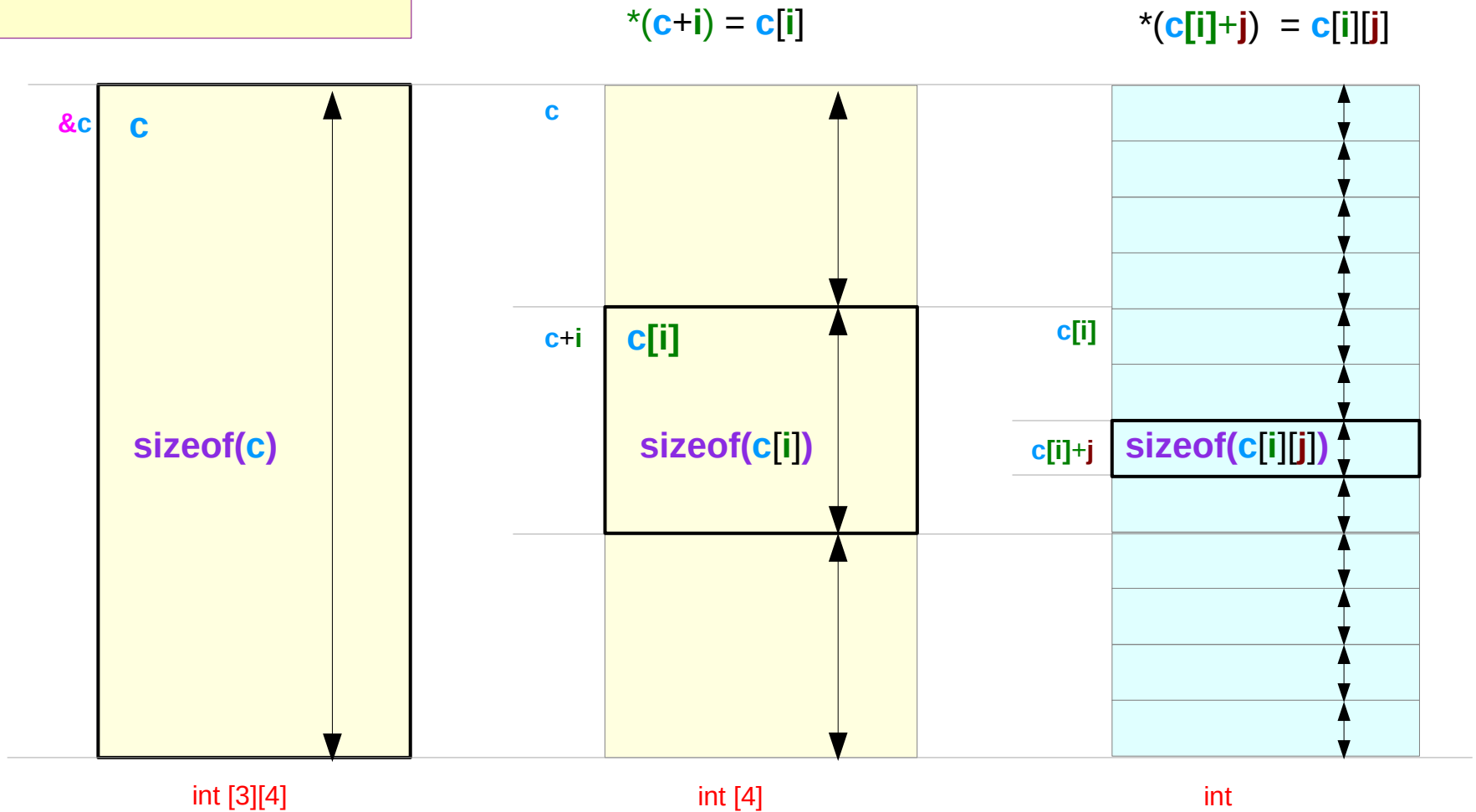
```
int c [3] [4];
```

row address + column address



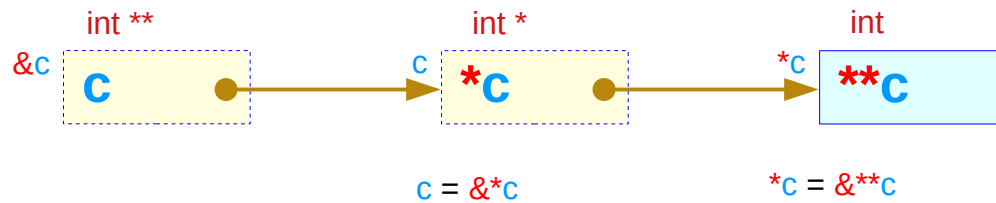
# Sizes of `c`, `c[i]`, `c[i][j]`

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

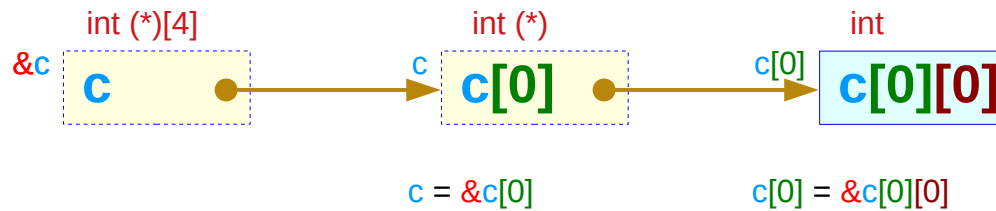


- 
- **Two types of a 2-d array  $c$**
  - **Two types of 1-d sub-arrays  $c[i]$**

# Chains of dereferences

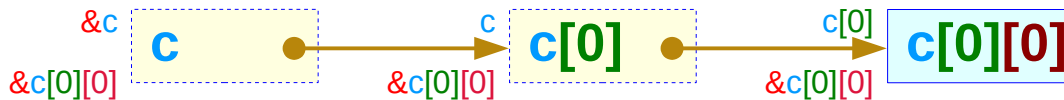


a chain of dereferences



using array notation [ ]

with incorrect types



Transferring the physical address

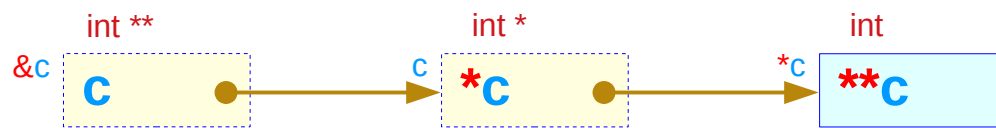
value( $\&c$ )  
 = value(**c**)

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

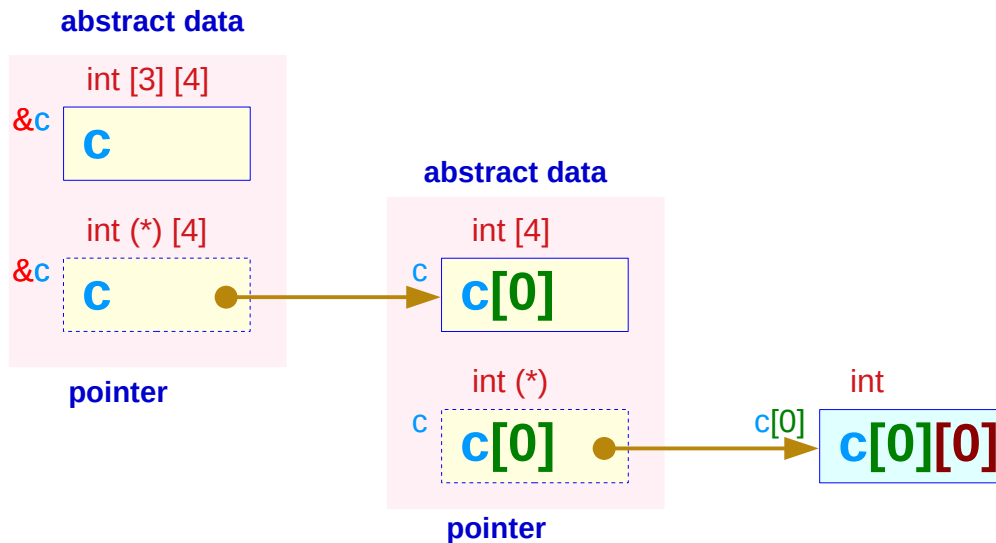
Conditions for **c** and **c[0]**  
 to start at the same address  $\&c[0][0]$

address(pointer) = value(pointer)

# Chains of dereferences with correct types



a chain of dereferences



using array notation [ ]  
with correct types

# Two types of an array – **c**, **c[i]**, **c[i][j]**

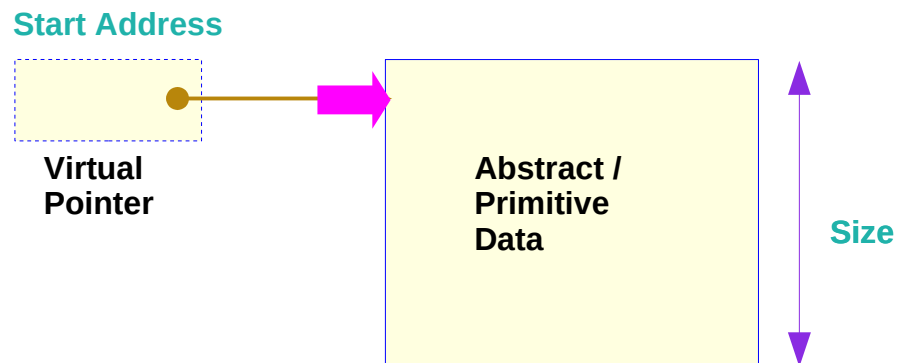
```
int c[3] [4] ;
```

<b>c</b>	<b>c[i]</b>	<b>c[i][j]</b>		
<b>2-d</b> array int [3][4]	<b>1-d</b> array int [4]	integer int	<b>Abstract / Primitive Data</b>	<b>Size</b>
<b>1-d</b> array pointer int (*)[4]	<b>0-d</b> array pointer int (*)		<b>Virtual Pointer</b>	<b>Start Address</b>



# Two types of an array – pointer to abstract data

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

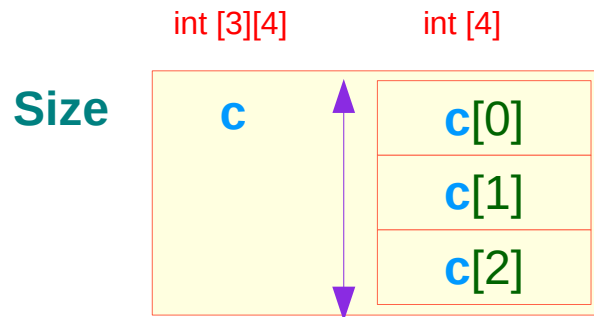


Virtual Pointer		Abstract / Primitive Data
<code>int (*)[4]</code>		<code>int [4]</code>
<code>c</code>	● →	<code>c[0]</code>
<code>int (*)</code>		<code>int</code>
<code>c[0]</code>	● →	<code>c[0][0]</code>
<code>c[1]</code>	● →	<code>c[1][0]</code>
<code>c[2]</code>	● →	<code>c[2][0]</code>
Start Address		Size

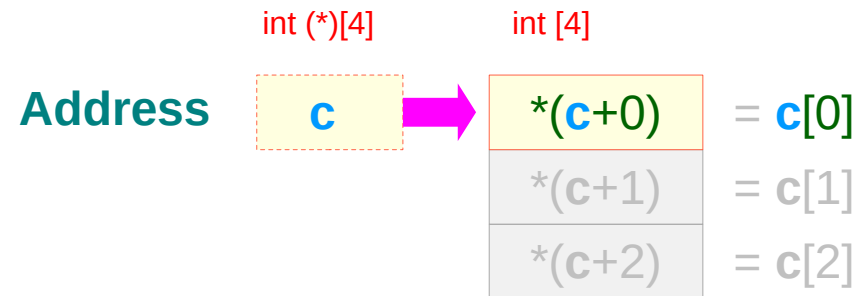
# Two types in a 2-d array `c`

`int c[3][4];`

## Abstract data `c`



## Pointer `c`



### `c` 2-d array

type : `int [3][4]`  
size : `3 * 4 * 4`

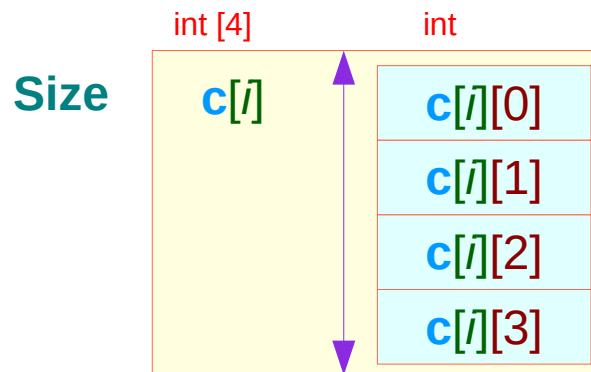
### `c` 1-d array pointer

type : `int (*)[4]`  
value : `&c[0][0]`

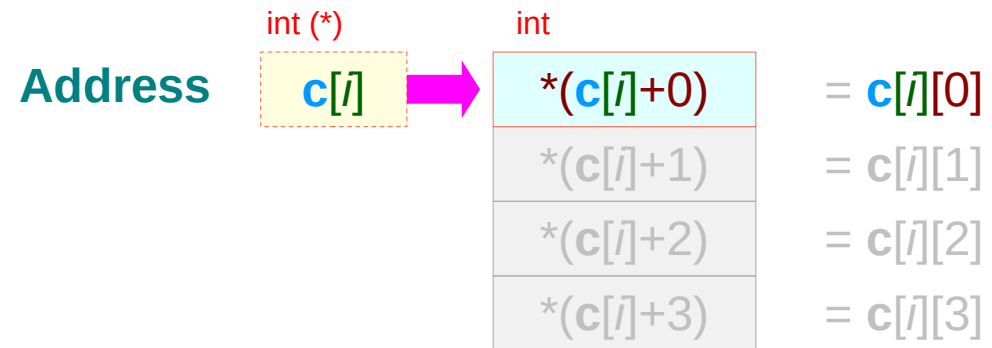
# Two types in a 1-d array $c[i]$

`int c[3][4];`

## Abstract data $c[i]$



## Pointer $c[i]$



`c[i]` 1-d array

type : `int [4]`

size : `4 * 4`

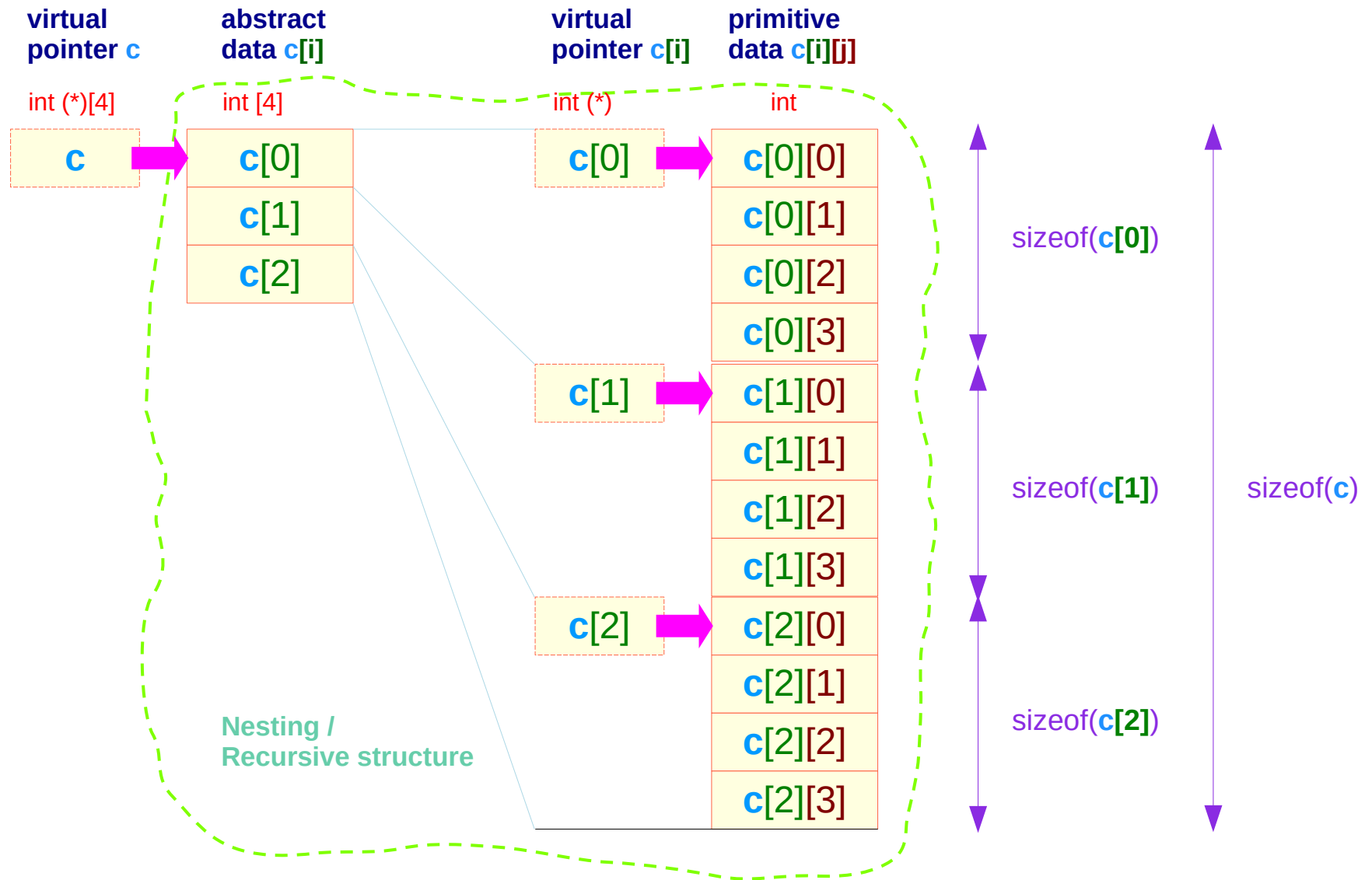
`c[i]` 0-d array pointer

type : `int (*)`

value : `&c[i][0]`

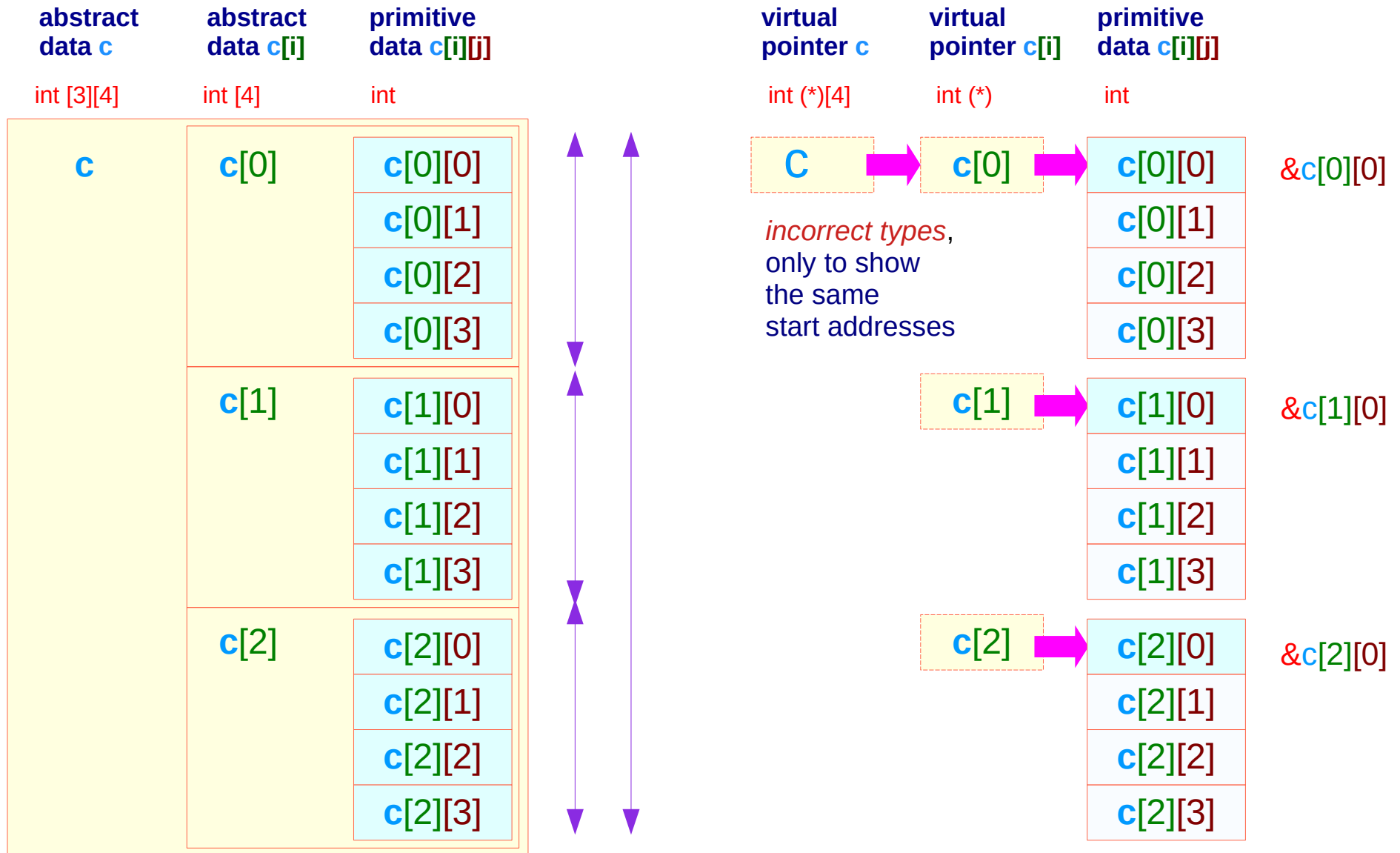
# Combining two types with sizes

`int c[3][4];`



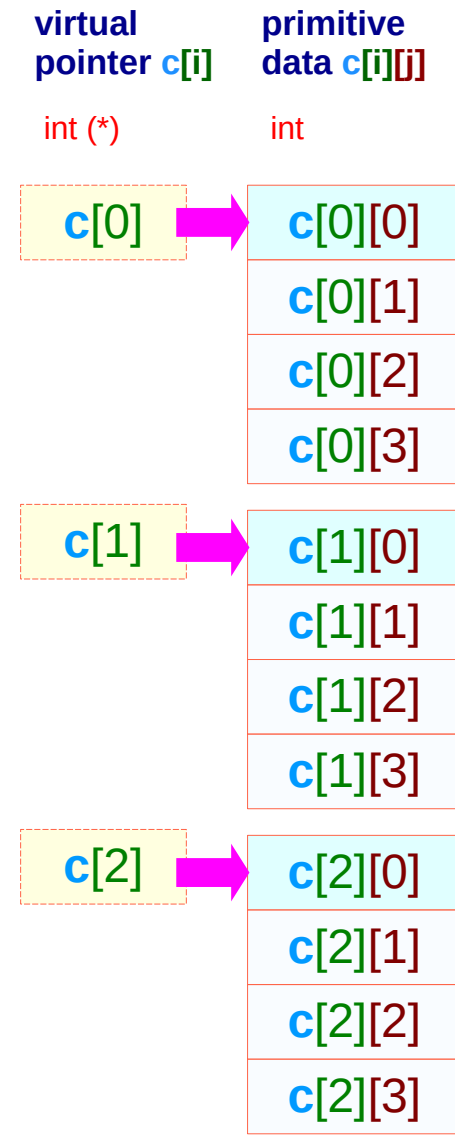
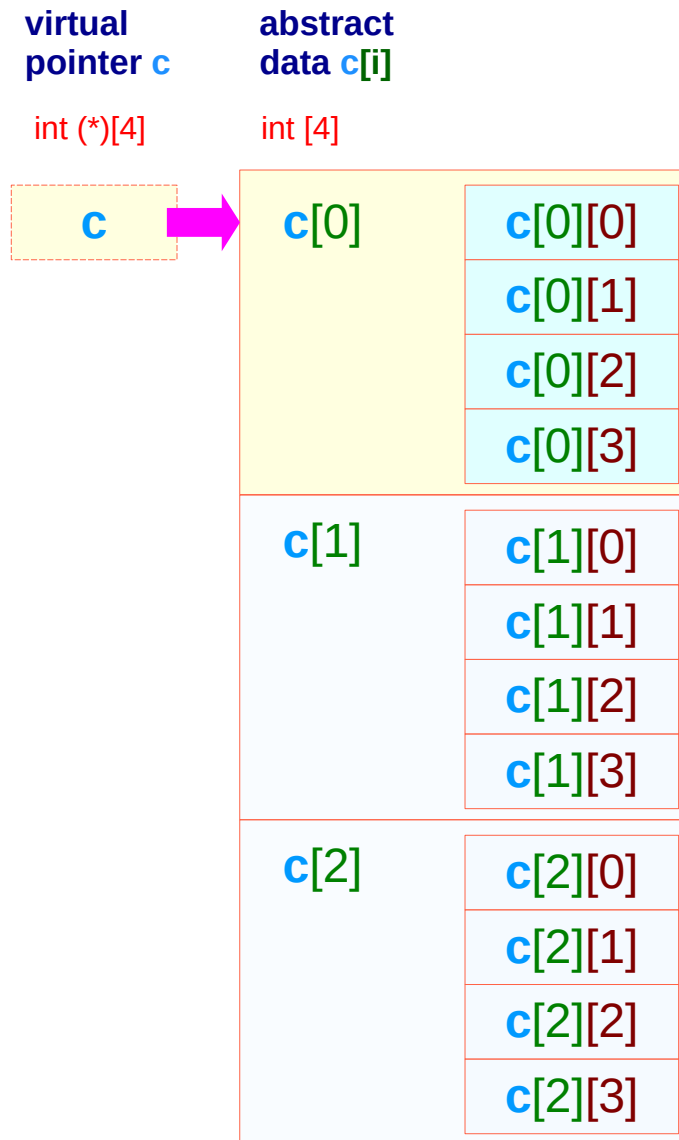
# Recursive data view and pointer view

`int c[3][4];`



# Recursive pointer-to-data views

`int c[3][4];`

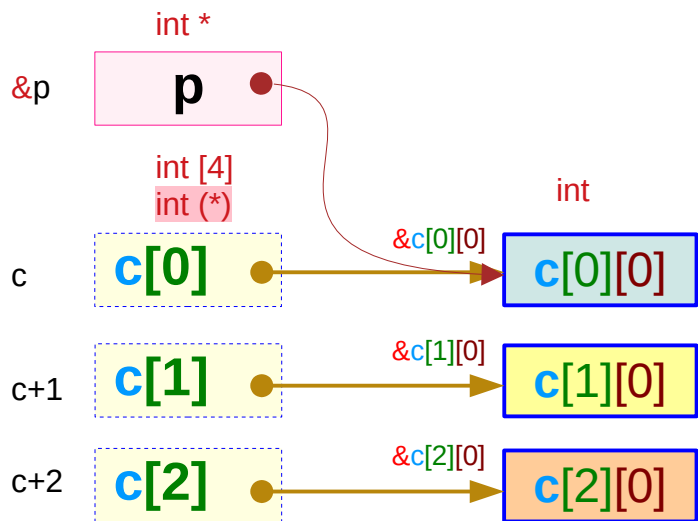


# Type, address, and value of $c[i]$

`int c[3][4];`

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

real pointer  
real memory location



non-real pointer  
no memory locations

primitive data

row addresses

$\text{type}(c[i]) = \text{int}[4]$  abstract data type  
 $\text{int} (*)$  virtual pointer type

address	variable	value
$\text{value}(c)$	$= \text{value}(c[0])$	$= \text{value}(\&c[0][0])$
$\text{value}(c+1)$	$= \text{value}(c[1])$	$= \text{value}(\&c[1][0])$
$\text{value}(c+2)$	$= \text{value}(c[2])$	$= \text{value}(\&c[2][0])$

the same address values

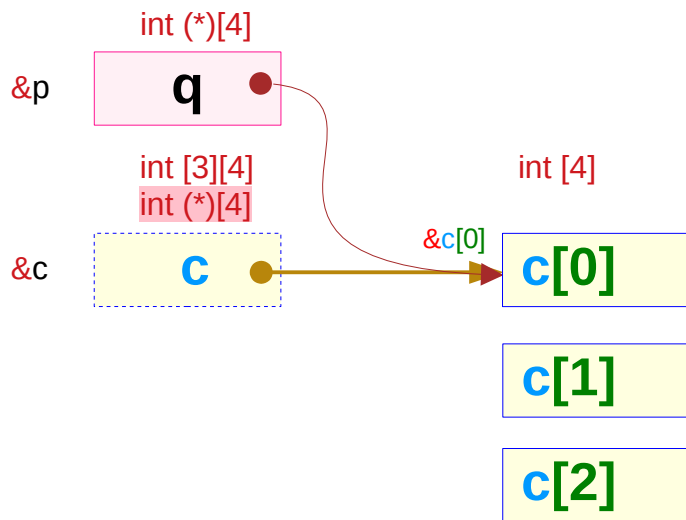
$\text{value}(\&p) \neq \text{value}(p) = \text{value}(\&c[0][0])$

# Type, address, and value of **c**

`int c [3] [4];`

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

real pointer  
real memory location



non-real pointer  
no memory locations

row addresses

abstract data

$\text{type}(\mathbf{c}) = \begin{array}{l} \text{int [3][4]} \text{ abstract data type} \\ \text{int (*)[4]} \text{ virtual pointer type} \end{array}$

address	variable	value
$\text{value}(\&\mathbf{c})$	$= \text{value}(\mathbf{c})$	$= \text{value}(\&\mathbf{c}[0][0])$

the same address values

$\text{value}(\&\mathbf{q})$	$\neq \text{value}(\mathbf{q})$	$= \text{value}(\&\mathbf{c}[0][0])$
------------------------------	---------------------------------	--------------------------------------



- 
- **Identifying nested arrays  
in a 2-d array declaration**

# Nested arrays in a 2-d array declaration

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

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

**c** : a 3 element array  
**c[i]** : each element

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

**c[i]**'s type 1 : **int [4]**  
an array of 4 integers

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

relaxed dimension

**c[i]**'s type 2: **int (\*)**  
a pointer to an integer

# Nested arrays

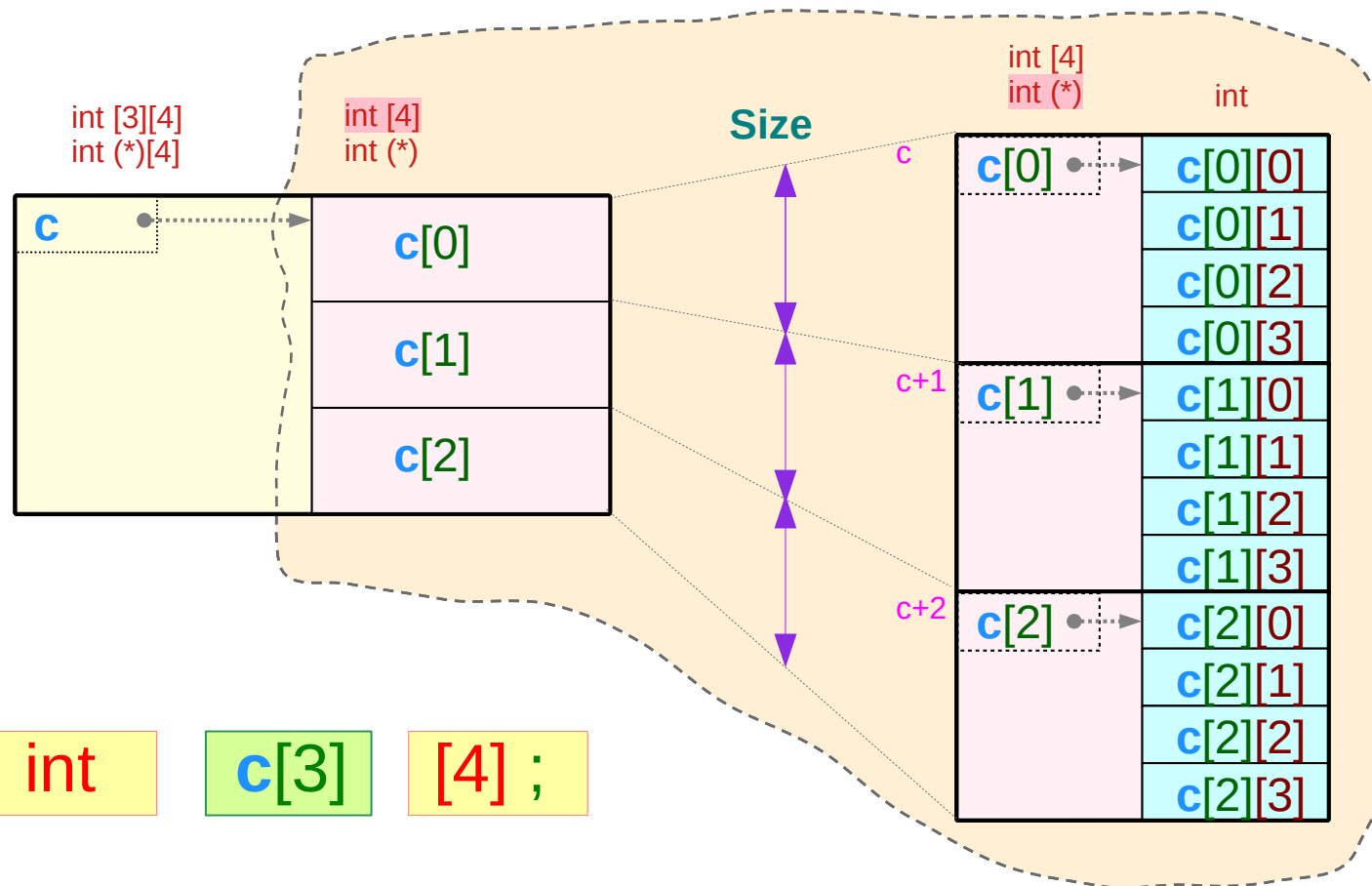
`c[3]`

`c` : a 3 element array  
`c[i]` : each element

`int`

`[4];`

`c[i]`'s type 1 : `int [4]`  
`c[i]`'s type 2 : `int (*)`



Address

`&c[0][0]`  $\rightarrow$  `c[0]`  $\rightarrow$  `c`

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

`&c[2][0]`  $\rightarrow$  `c[2]`

`int`

`c[3]`

`[4];`

# c : 3-element array

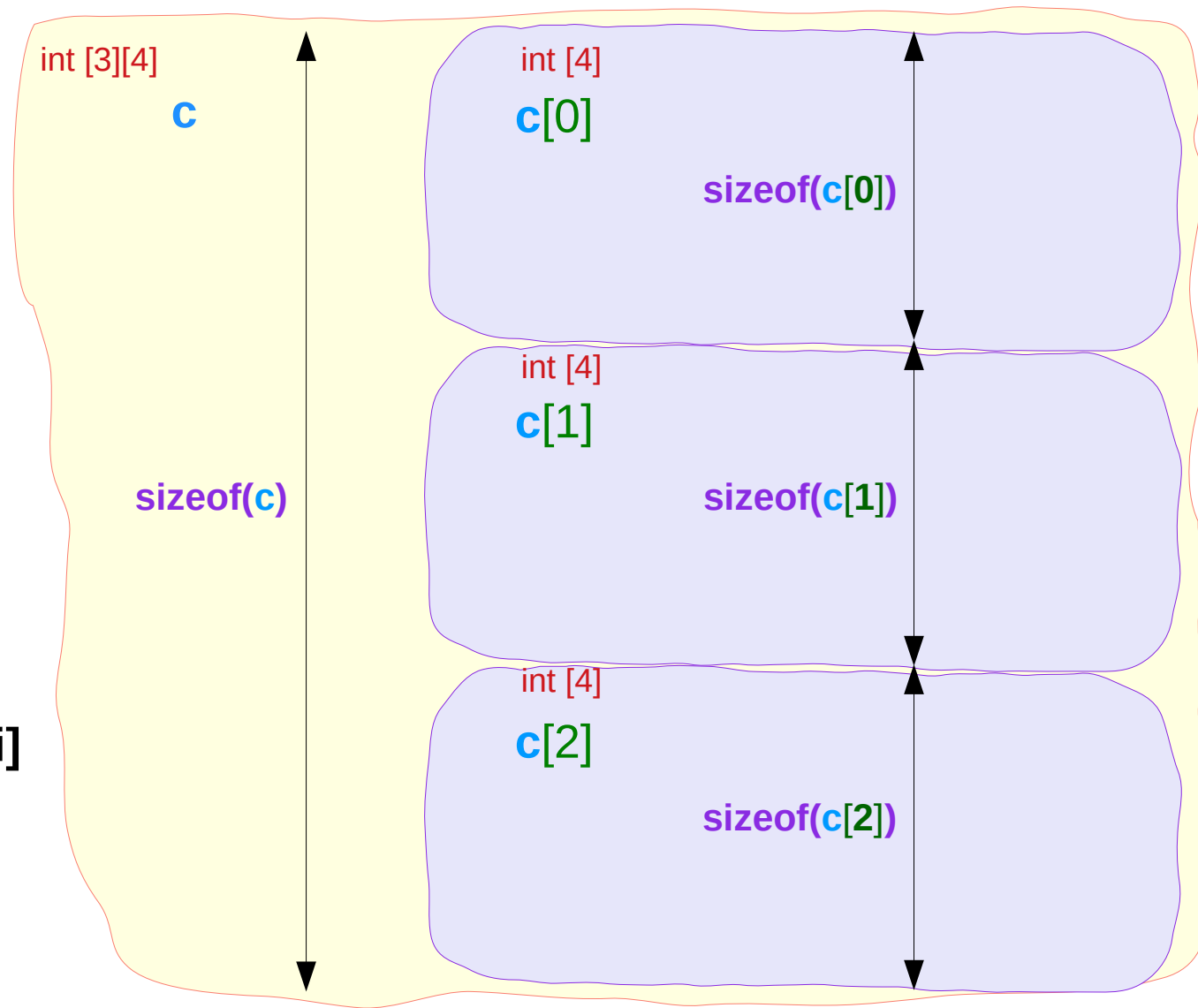
<b>c</b>	2-d array	int [3][4]
<b>c[i]</b>	1-d array	int [4]

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

## 3-element array c

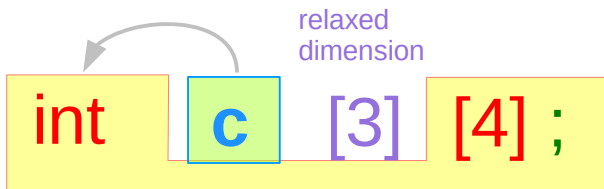
abstract data element **c[i]**

each element **c[i]** has the 1-d array type **int [4]**



# c : pointer to a 4-element array

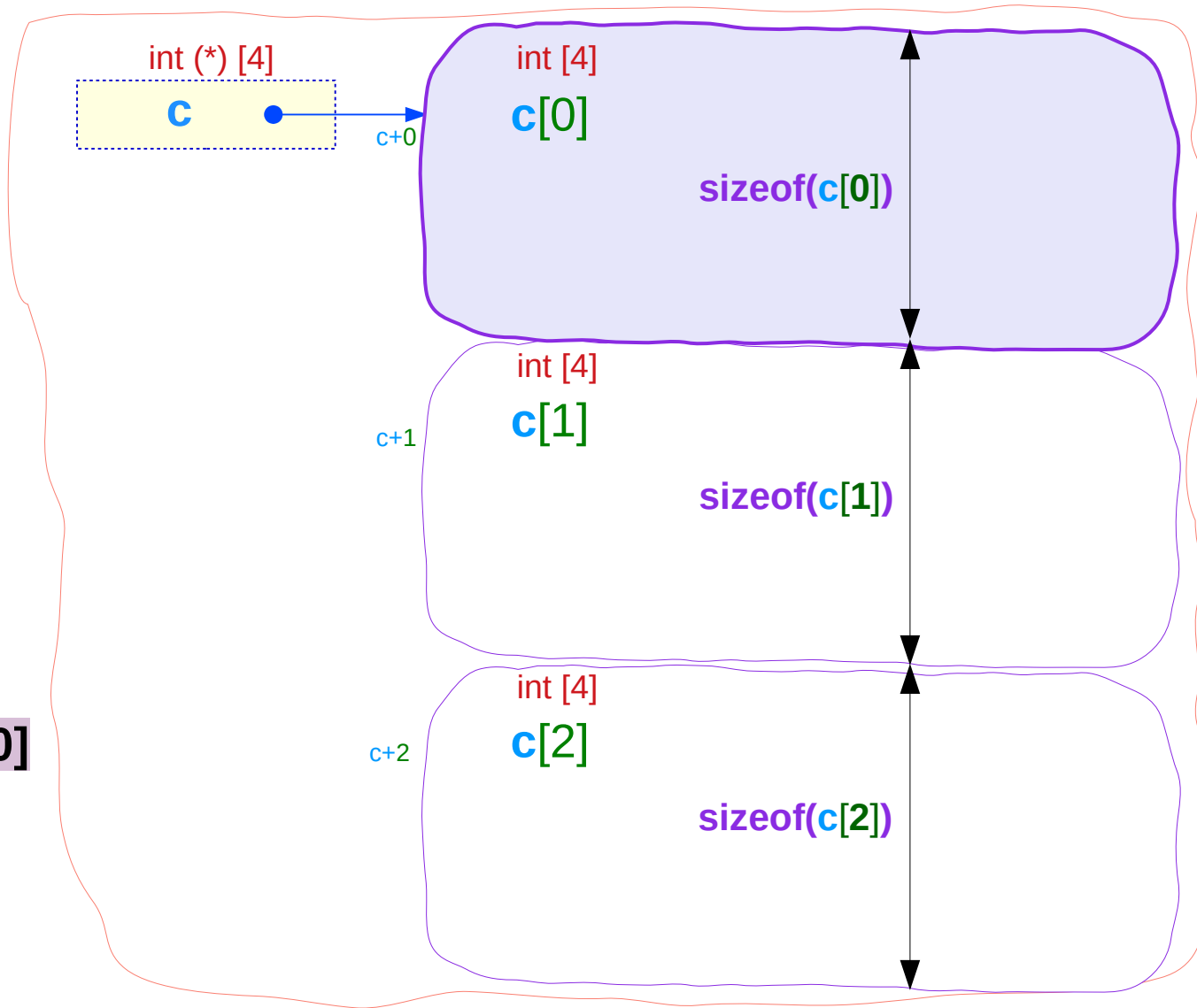
<b>c</b>	1-d array pointer	int (*)[4]
<b>c[i]</b>	1-d array	int [4]



## pointer c

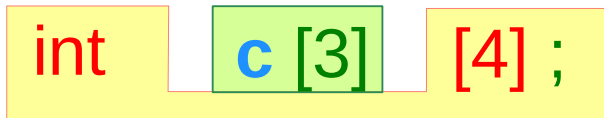
abstract data element **c[0]**

each element **c[i]** has the 1-d array type **int [4]**



# c[i] : 4-element array

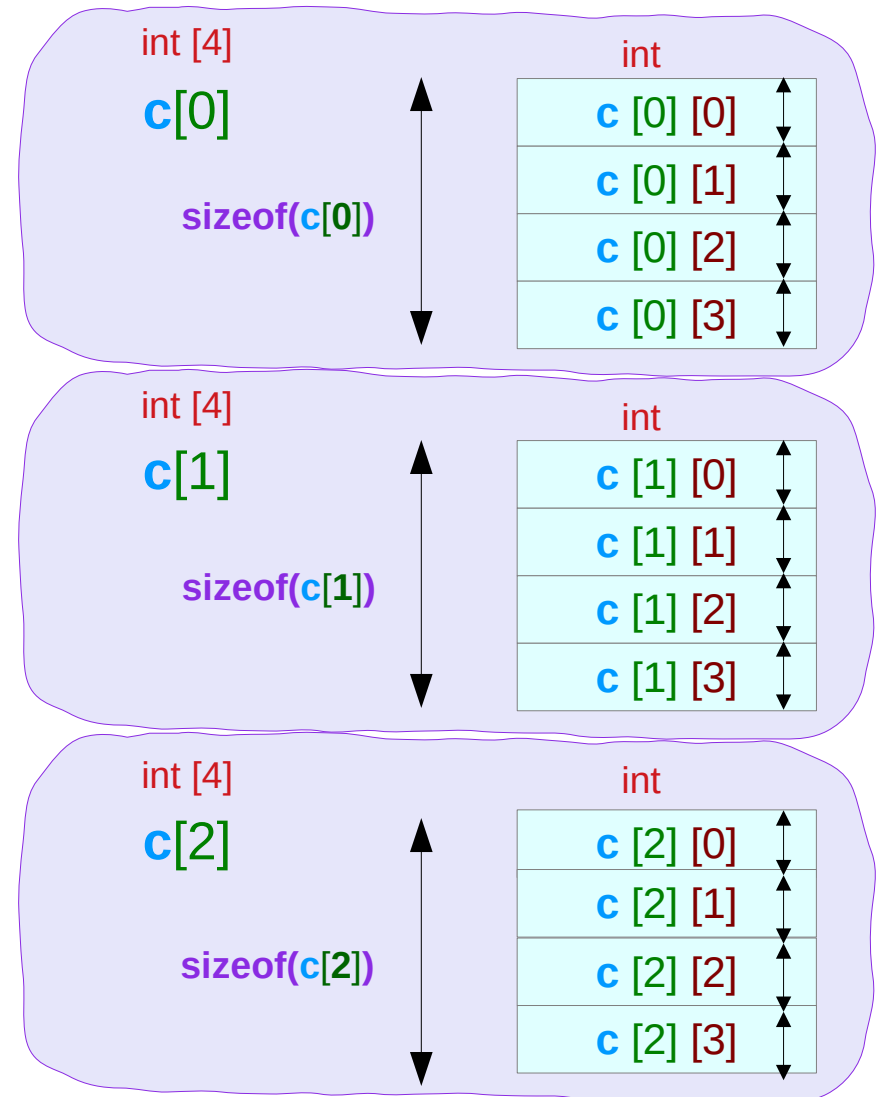
c[i]	1-d array	int [4]
c[i][j]	0-d array	int



## 4-element array c[i]

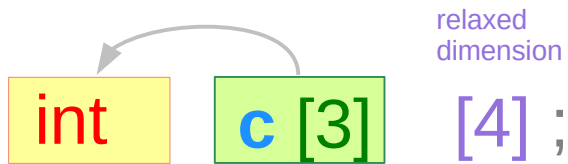
primitive data element `c[i][j]`

each element `c[i][j]` has the primitive type `int`



# c[i] : pointer to a primitive data

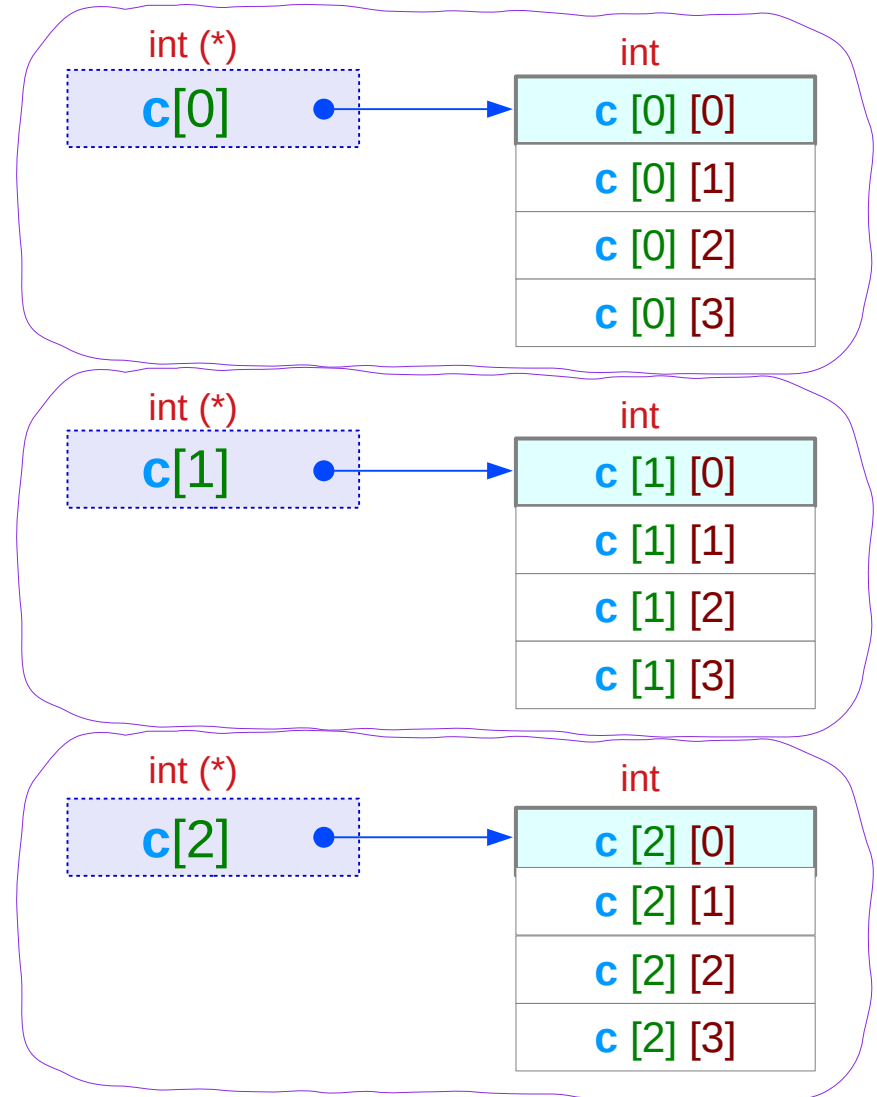
c[i]	0-d array pointer	int (*)
c[i][j]	0-d array	int



## pointer c[i]

primitive data element c[i][0]

each element c[i][j] has the primitive type int

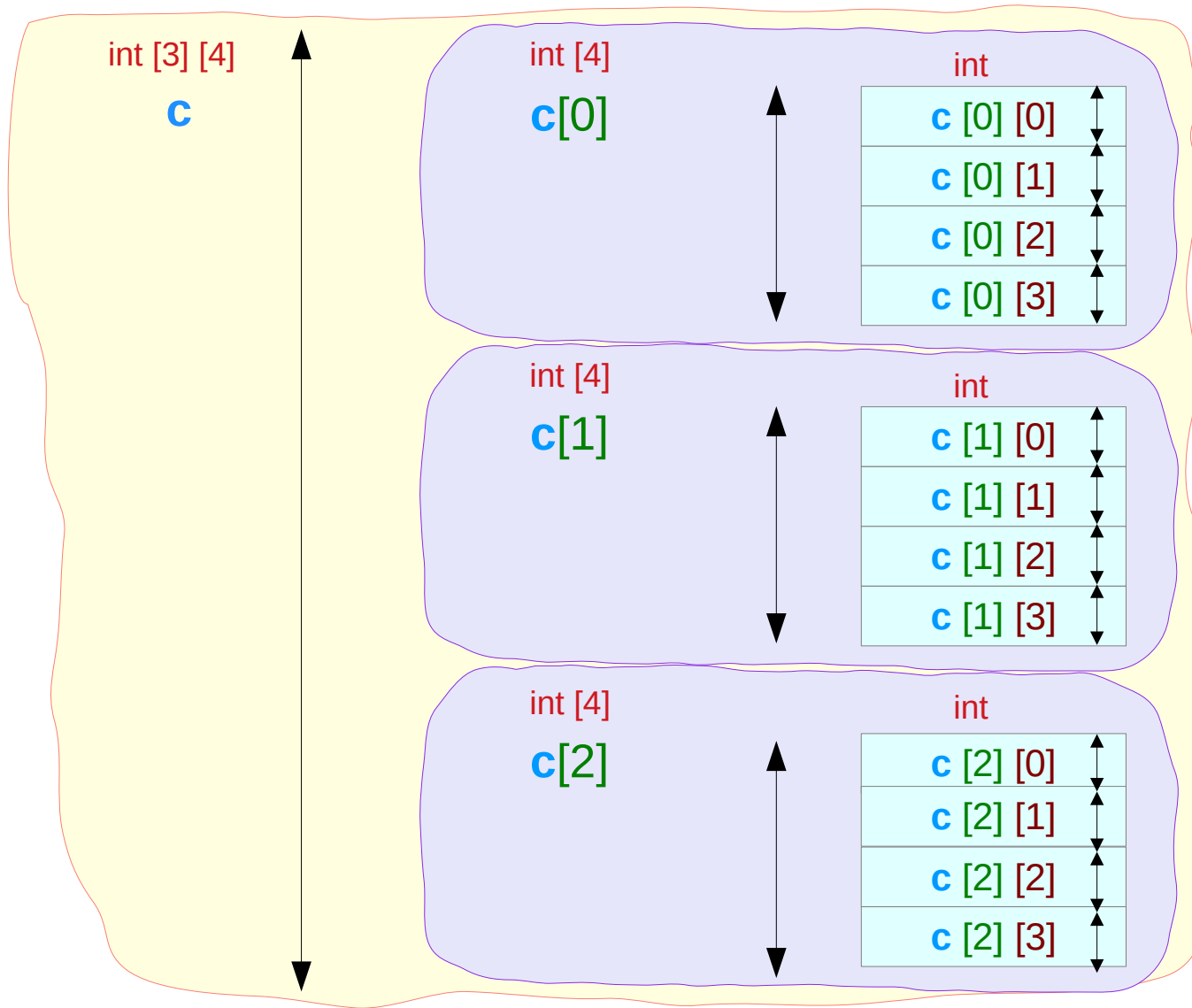


# Recursive data view

<code>c</code>	2-d array	<code>int [3][4]</code>
<code>c</code>	1-d array pointer	<code>int (*)[4]</code>
<code>c[i]</code>	1-d array	<code>int [4]</code>
<code>c[i]</code>	0-d array pointer	<code>int (*)</code>
<code>c[i][j]</code>	0-d array	<code>int</code>

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

3-element array `c`  
4-element array `c[i]`





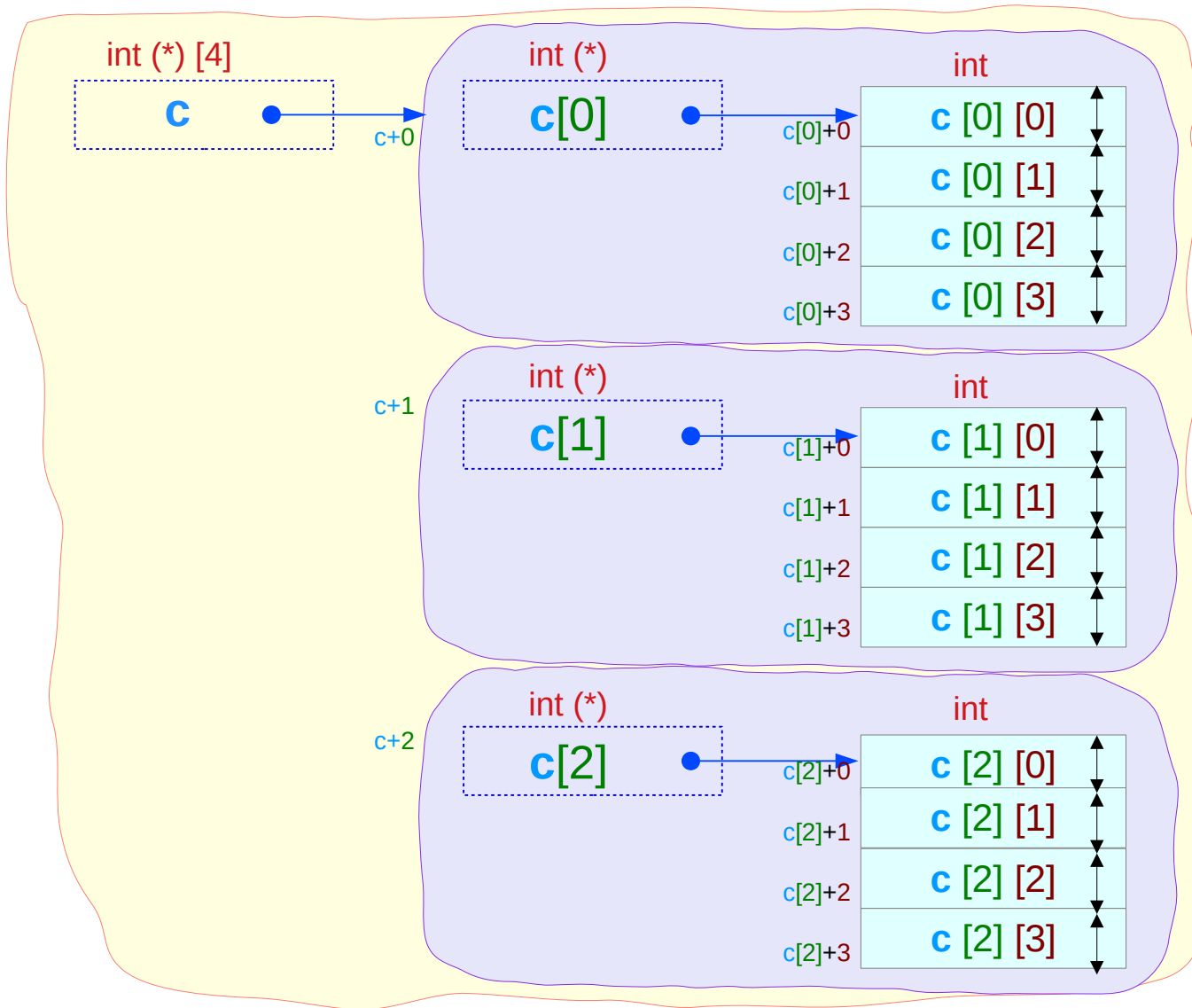
# Pointer view

<b>c</b>	<b>2-d array</b>	<b>int [3][4]</b>
<b>c</b>	<b>1-d array pointer</b>	<b>int (*)[4]</b>
<b>c[i]</b>	<b>1-d array</b>	<b>int [4]</b>
<b>c[i]</b>	<b>0-d array pointer</b>	<b>int (*)</b>
<b>c[i][j]</b>	<b>0-d array</b>	<b>int</b>

**int**   **c[3]**   **[4] ;**

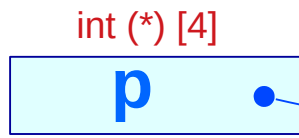
$$\begin{aligned}
 v(\mathbf{c}) &= v(\mathbf{c}[0]) = v(\&\mathbf{c}[0][0]) \\
 v(\mathbf{c}[1]) &= v(\&\mathbf{c}[1][0]) \\
 v(\mathbf{c}[2]) &= v(\&\mathbf{c}[2][0])
 \end{aligned}$$

$v \equiv \text{value}$



# 1-d array pointer

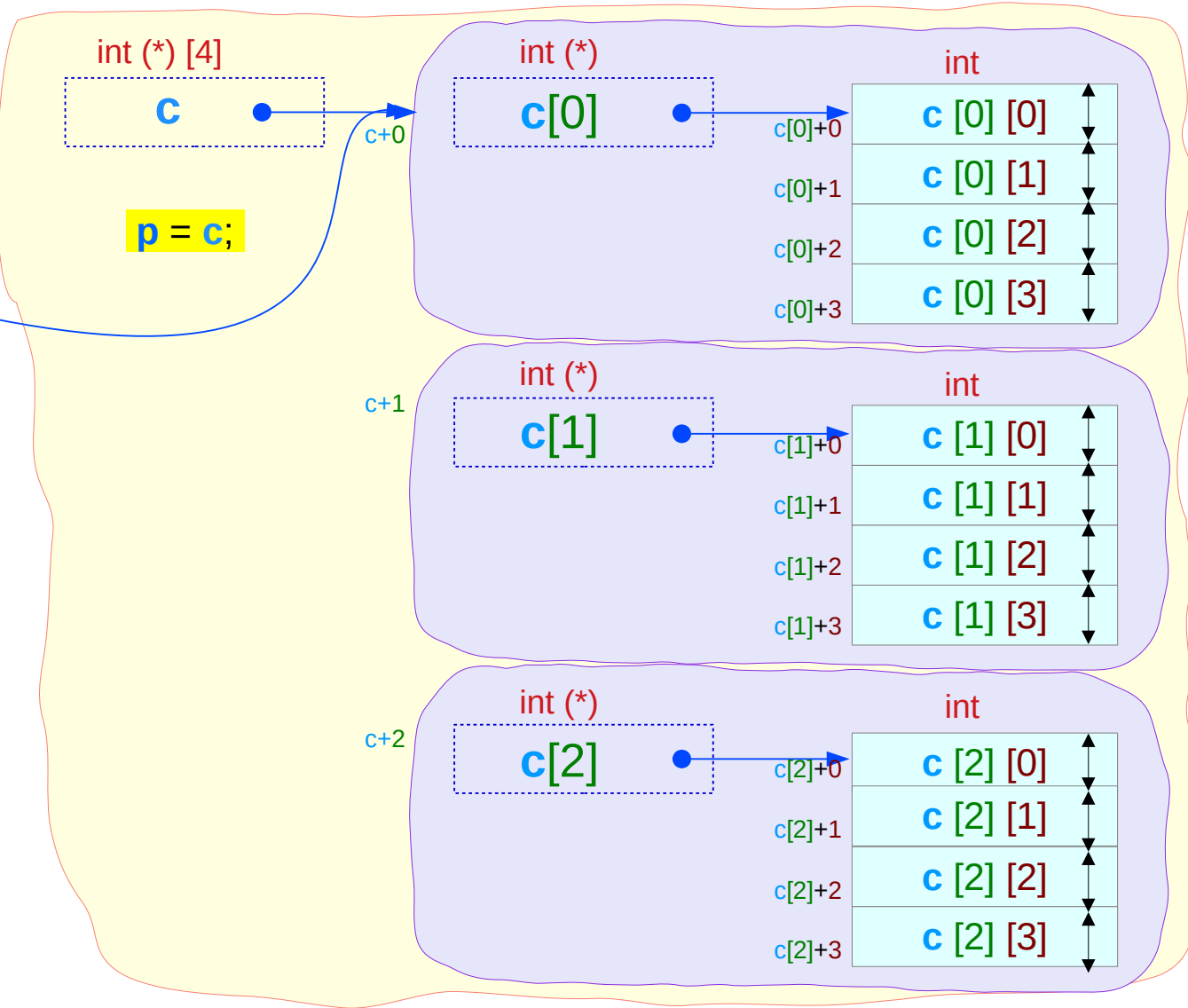
```
int (*p) [4];
```



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

$$\begin{aligned} v(\mathbf{c}) &= v(\mathbf{c}[0]) = v(\&\mathbf{c}[0][0]) \\ v(\mathbf{c}[1]) &= v(\&\mathbf{c}[1][0]) \\ v(\mathbf{c}[2]) &= v(\&\mathbf{c}[2][0]) \end{aligned}$$

$v \equiv$  value



- 
- **Two aspects of an array variable**

## Two aspects of **c**

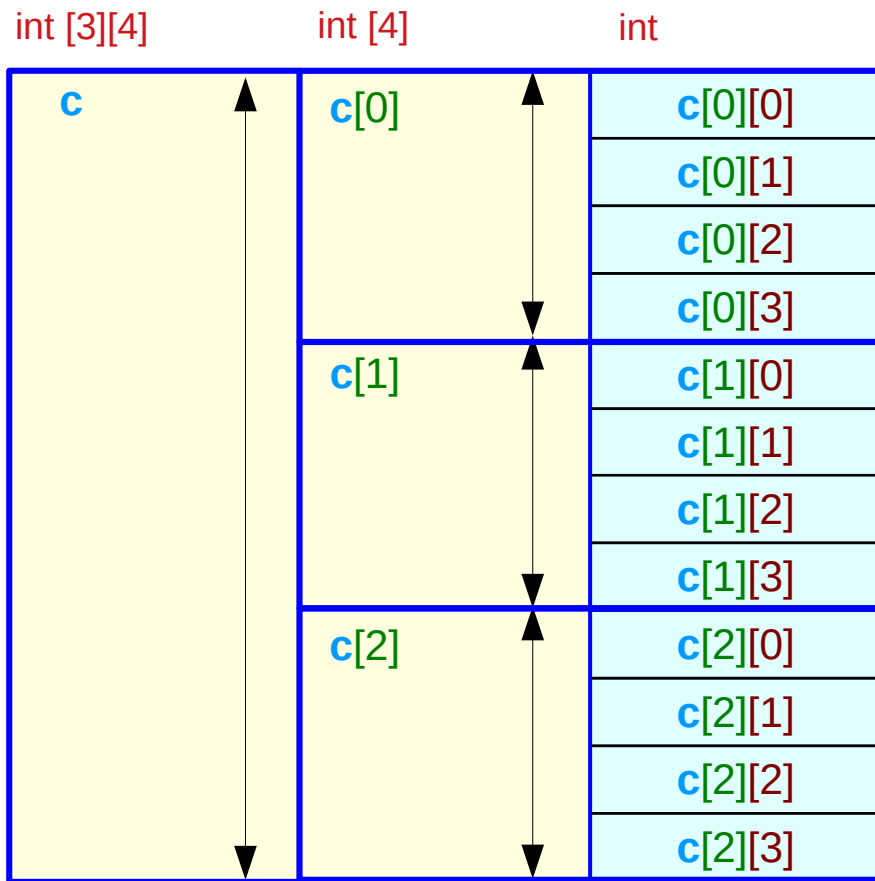
```
int      c[3][4];
```

**c** is an array of 3\*4 integer elements

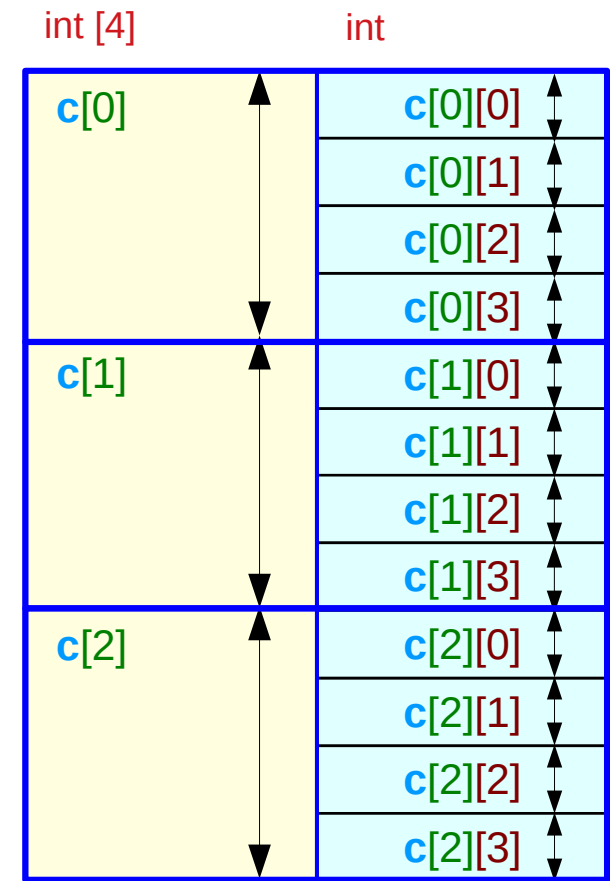
**Abstract data **c**, **c**[i] ..... size**

**Pointer **c**, **c**[i] ..... starting address**

# Abstract data $c$ , $c[i]$

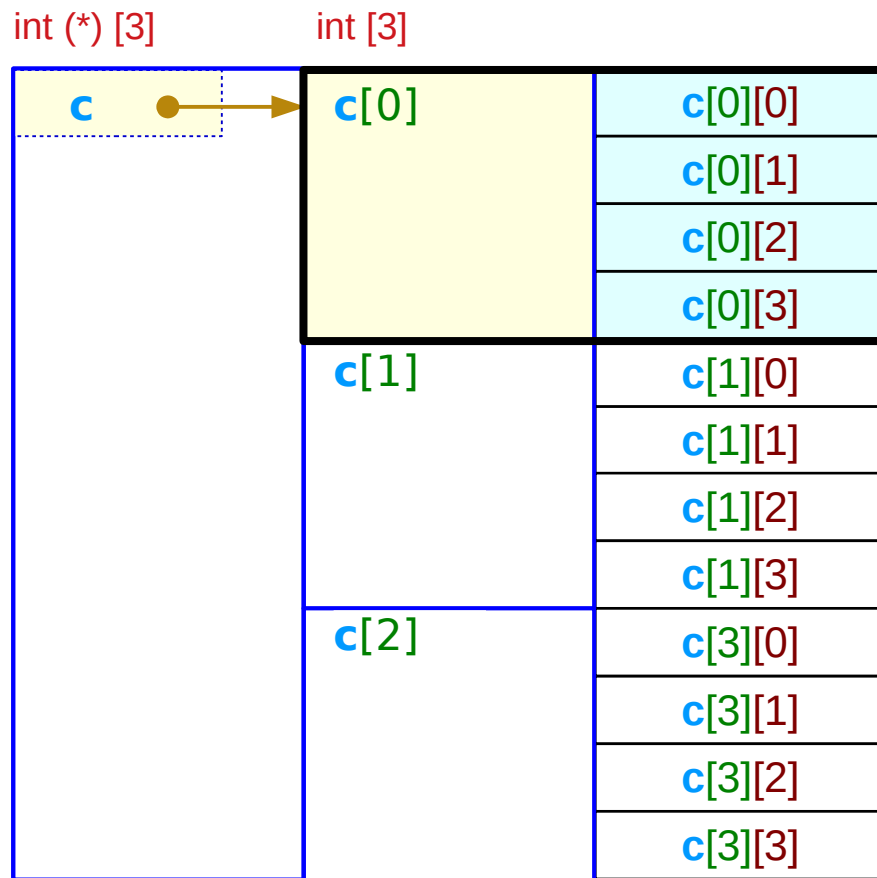


Abstract data  $c$

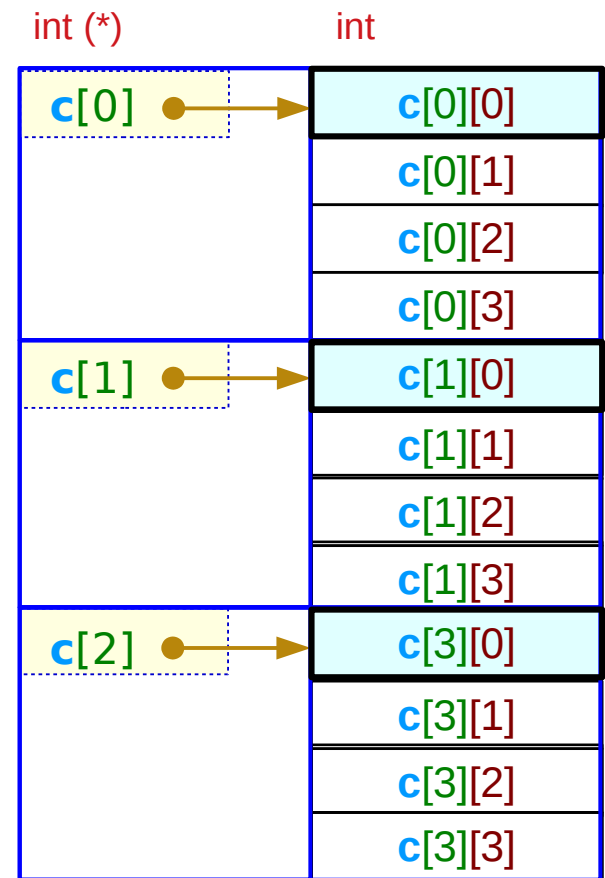


Abstract data  $c[i]$

# Pointer `c`, `c[i]`

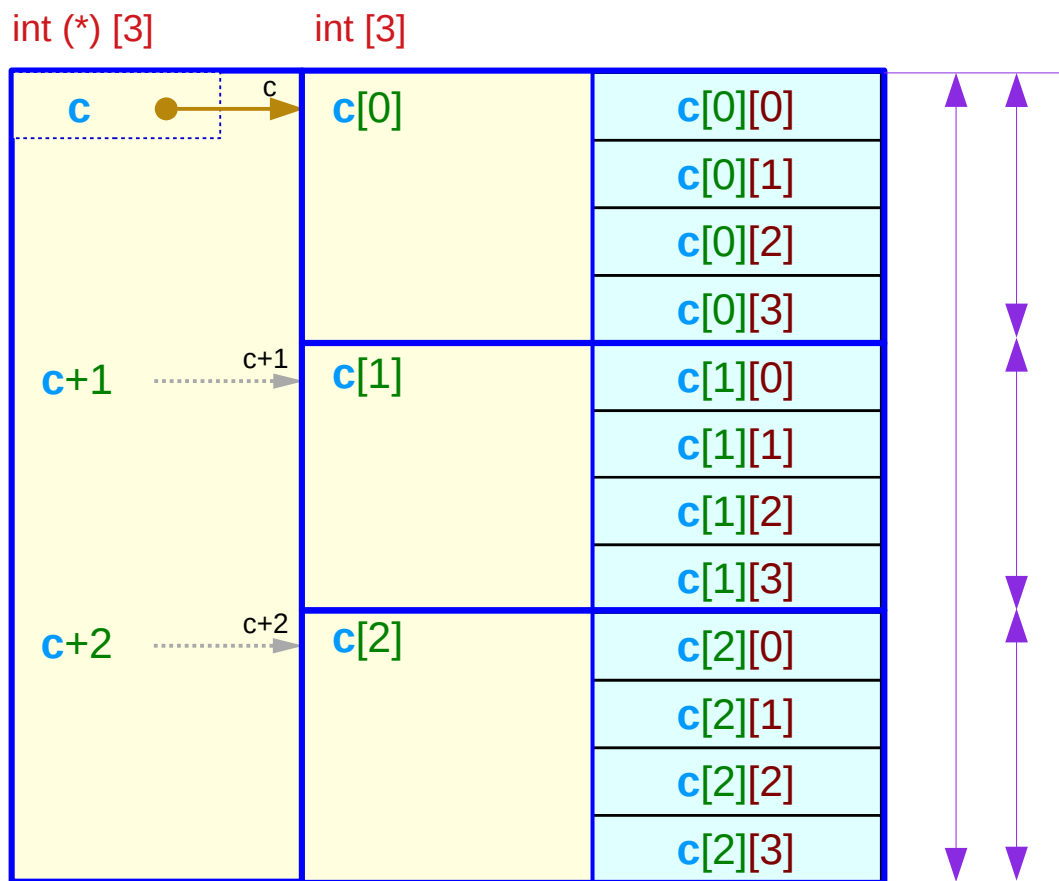


Pointer `c`



Pointer `c[i]`

# Aggregate Data Type – 2-d array **c**



Aggregate data with **3** elements

$\text{value}(c)$  = the start address of the 1<sup>st</sup> element

the start address of each element :

$$\text{value}(c+0) = \text{value}(c) + 0 * \text{sizeof}(*c)$$

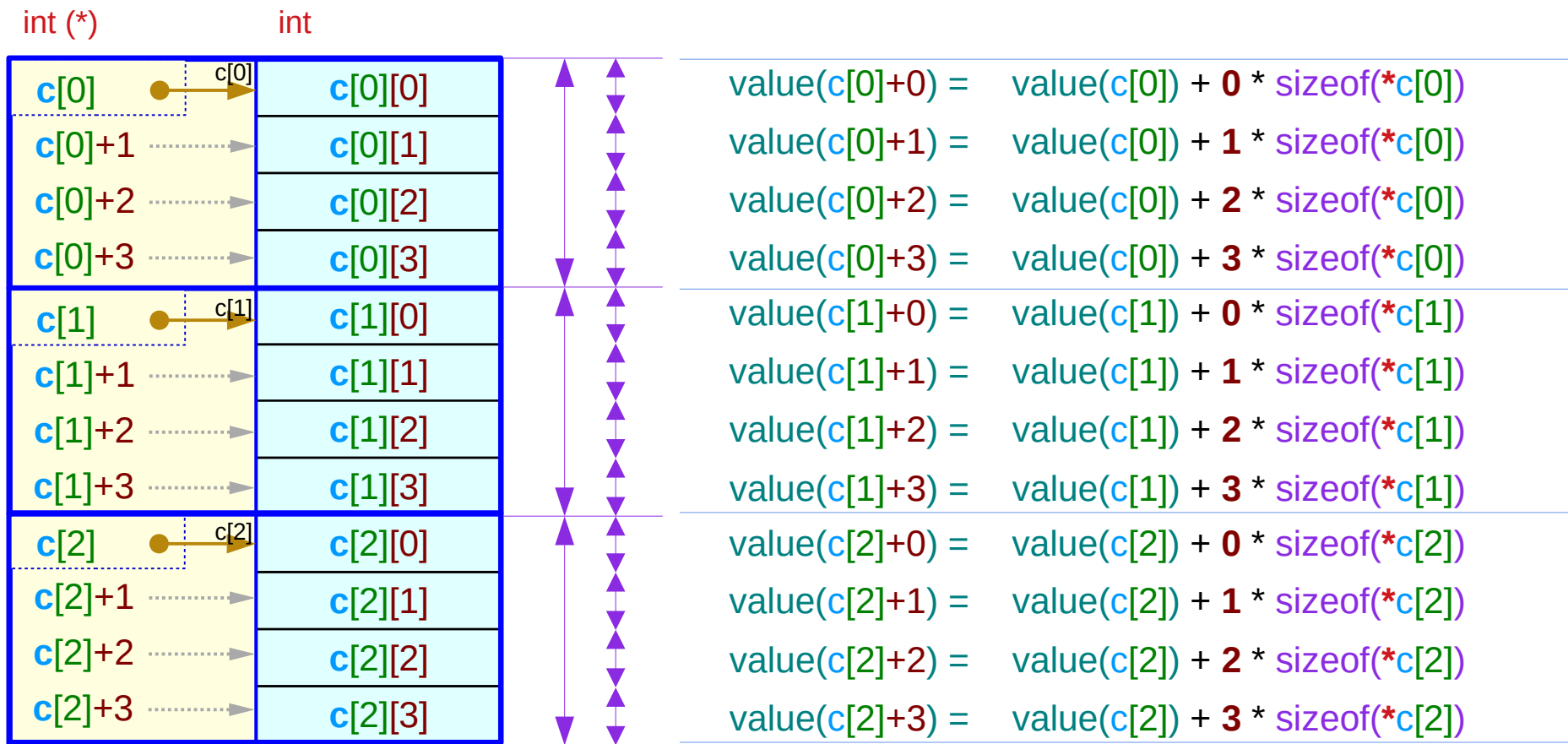
$$\text{value}(c+1) = \text{value}(c) + 1 * \text{sizeof}(*c)$$

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

$$\text{sizeof}(c) = \text{sizeof}(*c) * 3$$

**c aggregates** **c[0], c[1], c[2]**

# Aggregate Data Type – 1-d subarray $c[i]$



$$\text{sizeof}(c[i]) = \text{sizeof}(*c[i]) * 4$$

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

$c[i]$  aggregates  $c[i][0]$ ,  $c[i][1]$ ,  $c[i][2]$ ,  $c[i][3]$



# Size view of abstract data $c$ , $c[i]$

size of abstract data  $c$

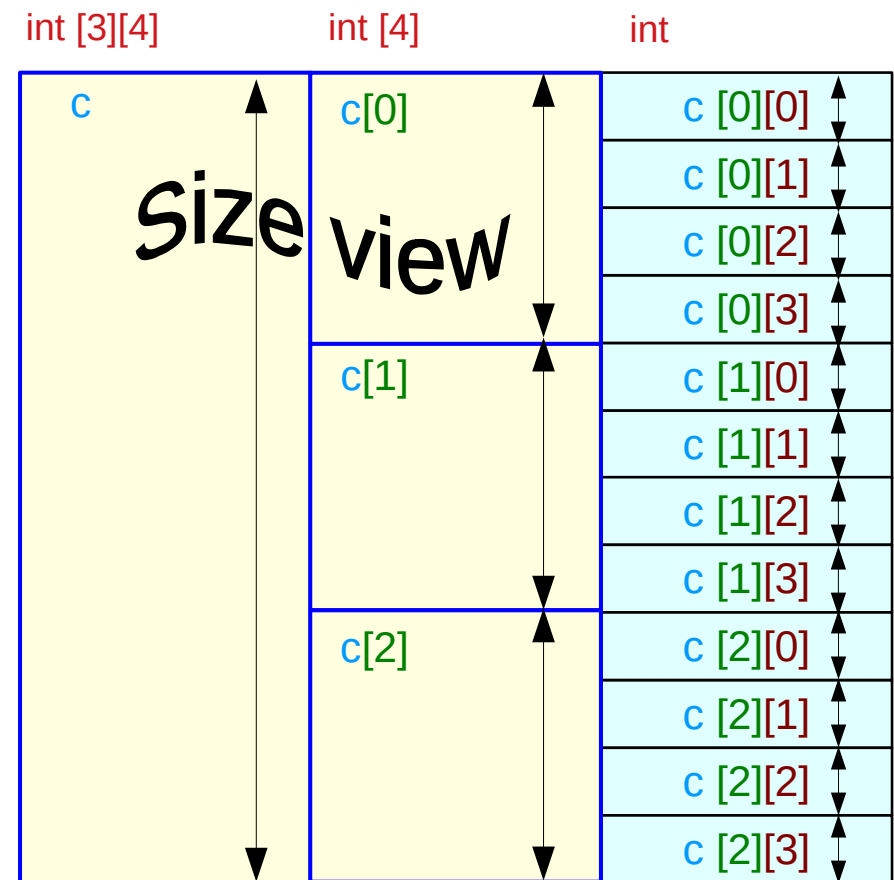
$$\begin{aligned}\text{sizeof}(c) &= \text{sizeof}(*c) * 3 \\ &= \text{sizeof}(c[i]) * 3\end{aligned}$$

size of 4 elements  
 $c[0]$ ,  $c[1]$ ,  $c[2]$

size of abstract data  $c[i]$

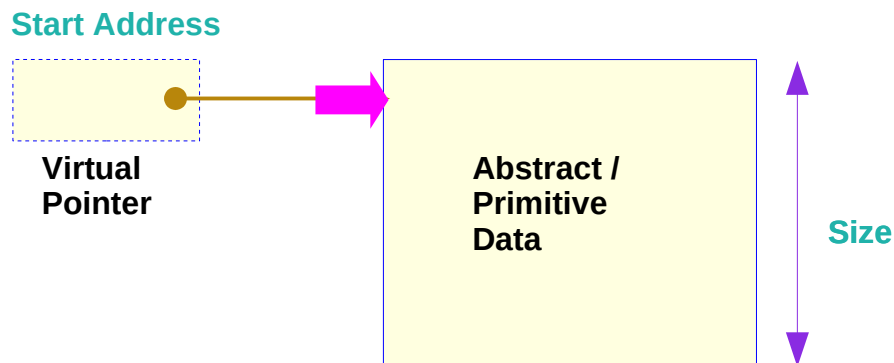
$$\begin{aligned}\text{sizeof}(c[i]) &= \text{sizeof}(*c[i]) * 4 \\ &= \text{sizeof}(c[i][j]) * 4\end{aligned}$$

size of 4 elements  
 $c[i][0]$ ,  $c[i][1]$ ,  $c[i][2]$ ,  $c[i][3]$



# Two types of an array – pointer to abstract data

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

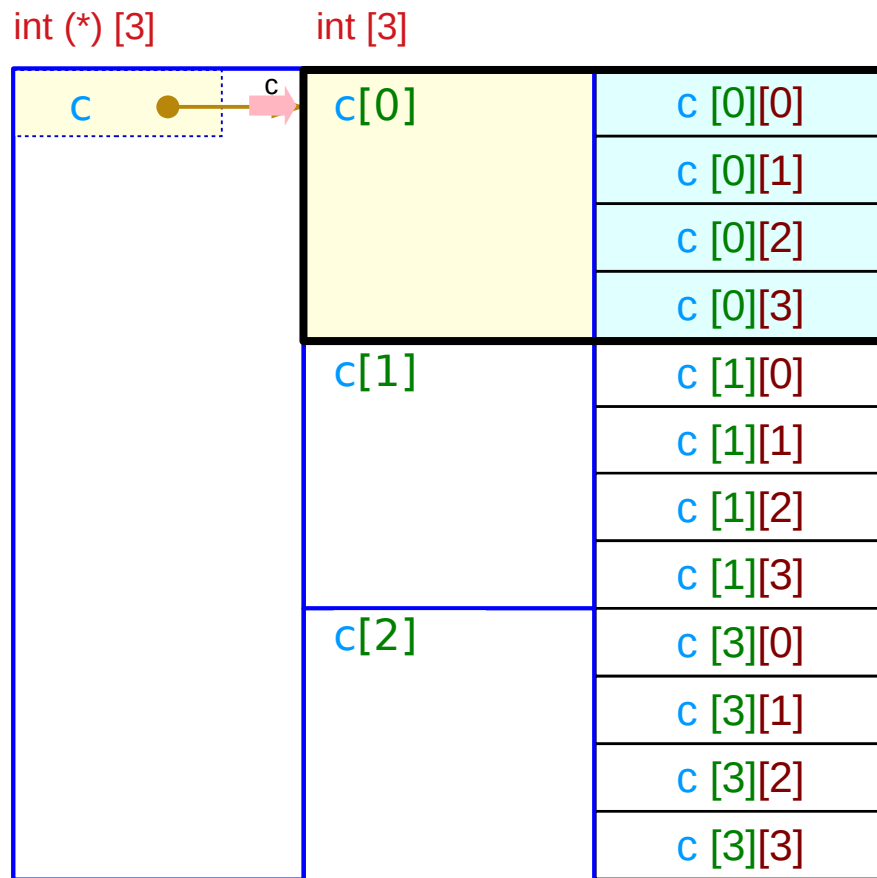


Virtual Pointer		Abstract / Primitive Data
<code>int (*)[4]</code>		<code>int [4]</code>
<code>c</code>	● →	<code>c[0]</code>
<code>int (*)</code>		<code>int</code>
<code>c[0]</code>	● →	<code>c[0][0]</code>
<code>c[1]</code>	● →	<code>c[1][0]</code>
<code>c[2]</code>	● →	<code>c[2][0]</code>
Start Address		Size

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

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

# Addresses of abstract data `c[0]`

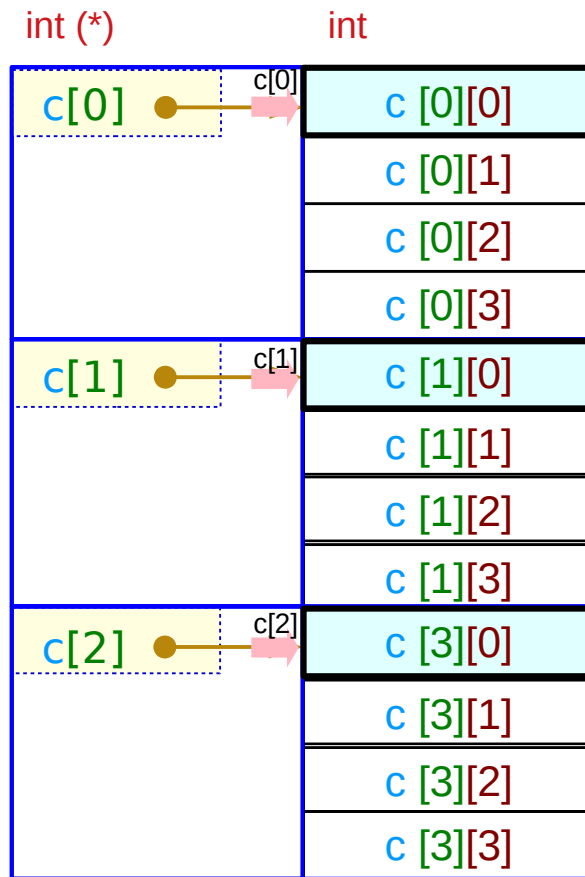


addresses of `c[0]`

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

`c[i] ≡ *(c+i)`  
`&c[i] ≡ c+i`  
`&c[0] ≡ c+0`

# Addresses of primitive data $c[i][j]$



## addresses of $c[i][0]$

$\text{value}(c[0]) = \text{value}(\&c[0][0])$

$\text{value}(c[1]) = \text{value}(\&c[1][0])$

$\text{value}(c[2]) = \text{value}(\&c[2][0])$

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

# Address views of virtual pointer $\mathbf{c}$ , $\mathbf{c}[i]$

$\text{value}(\mathbf{c}) = \text{value}(\mathbf{c}[0]) = \text{value}(\&\mathbf{c}[0][0])$   
 $\text{value}(\mathbf{c}[1]) = \text{value}(\&\mathbf{c}[1][0])$   
 $\text{value}(\mathbf{c}[2]) = \text{value}(\&\mathbf{c}[2][0])$

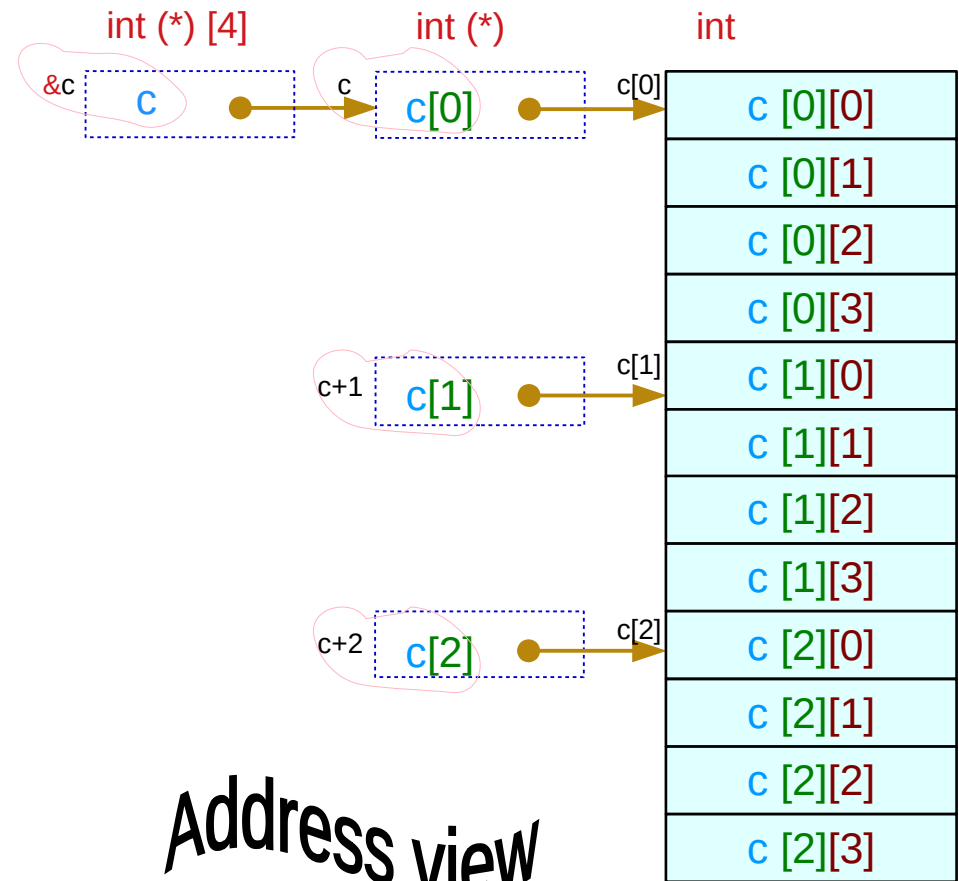


$\text{value}(\&\mathbf{c}) = \text{value}(\mathbf{c})$

$\text{value}(\&\mathbf{c}[0]) = \text{value}(\mathbf{c}[0])$   
 $\text{value}(\&\mathbf{c}[1]) = \text{value}(\mathbf{c}[1])$   
 $\text{value}(\&\mathbf{c}[2]) = \text{value}(\mathbf{c}[2])$

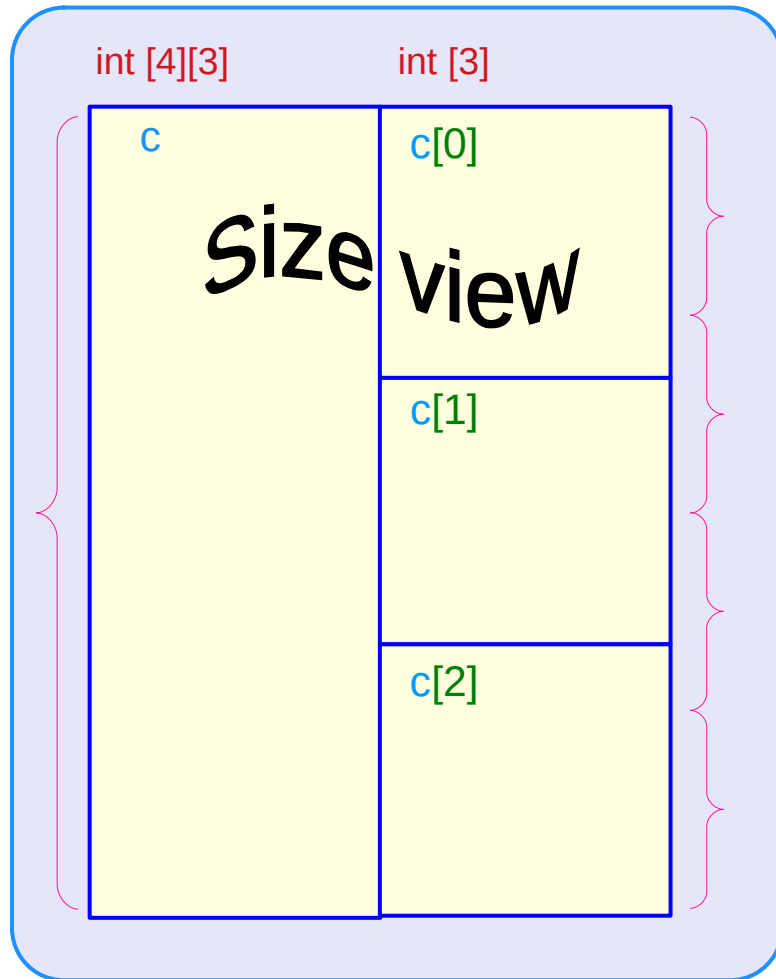
no real pointer can satisfy  
these conditions

no physical memory location

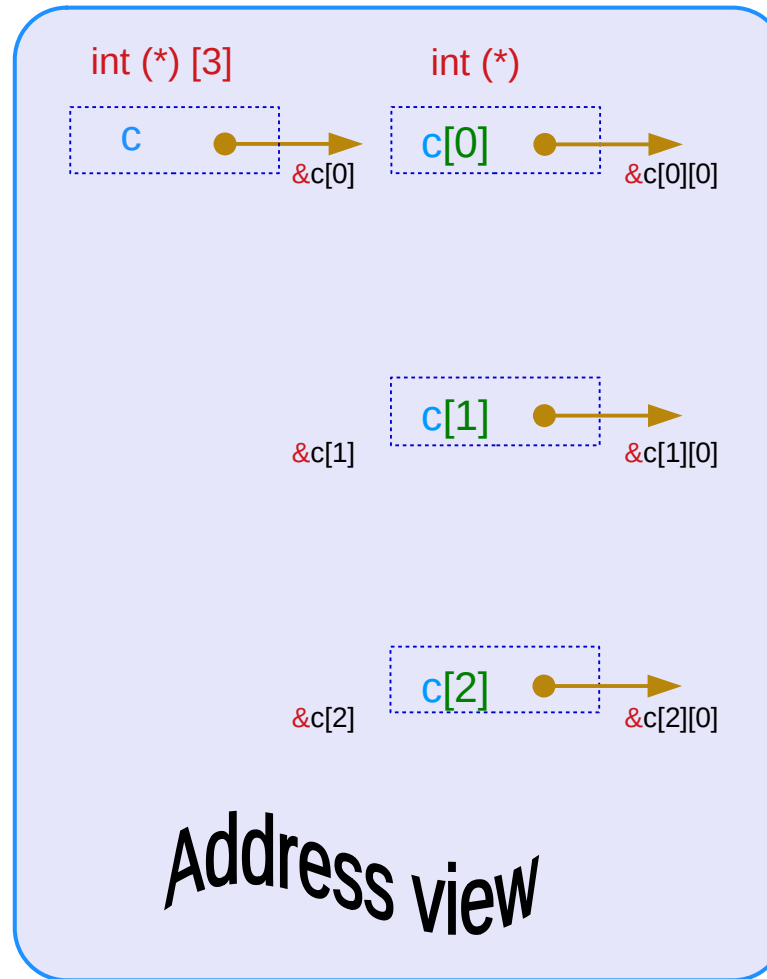


*Address view*

# Size and address views of `c`, `c[i]`



$$\text{sizeof}(c) = \text{sizeof}(c[i]) * 3$$



$$\begin{aligned} \text{value}(c) &= \text{value}(c[0]) = \text{value}(\&c[0][0]) \\ \text{value}(c[1]) &= \text{value}(\&c[1][0]) \\ \text{value}(c[2]) &= \text{value}(\&c[2][0]) \end{aligned}$$

int
<code>c [0][0]</code>
<code>c [0][1]</code>
<code>c [0][2]</code>
<code>c [0][3]</code>
<code>c [1][0]</code>
<code>c [1][1]</code>
<code>c [1][2]</code>
<code>c [1][3]</code>
<code>c [2][0]</code>
<code>c [2][1]</code>
<code>c [2][2]</code>
<code>c [2][3]</code>

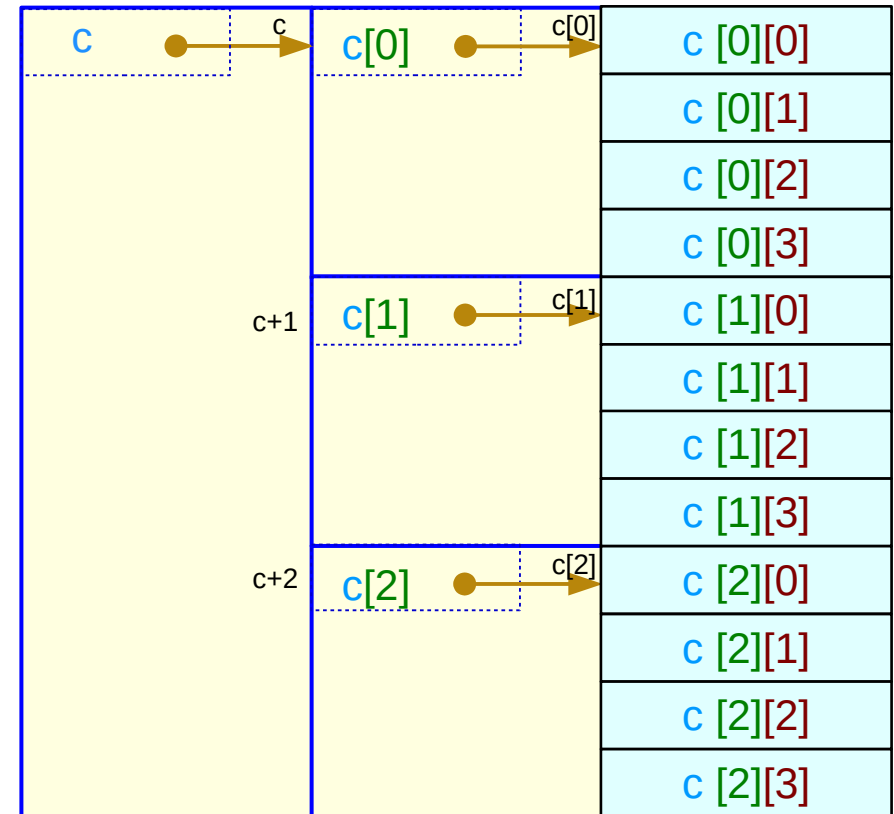
# Combining size view and address view

## Size view + Address view

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

$\text{value}(c) = \text{value}(c[0]) = \text{value}(\&c[0][0])$   
 $\text{value}(c[1]) = \text{value}(\&c[1][0])$   
 $\text{value}(c[2]) = \text{value}(\&c[2][0])$

Abstract data  $\longrightarrow$  int [4][3]  $\longrightarrow$  int [3]  $\longrightarrow$  int  
Pointer  $\longrightarrow$  int (\*) [3]  $\longrightarrow$  int (\*)  $\longrightarrow$  int



# Limitations

---

No index Range Checking

Array Size must be a constant expression

Variable Array Size

Arrays cannot be Copied or Compared

Aggregate Initialization and Global Arrays

Precedence Rule

Index Type Must be Integral



# 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
- [5] <https://pdos.csail.mit.edu/6.828/2008/readings/pointers.pdf>