

Arrays (1A)

Copyright (c) 2022 - 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.

This document was produced by using LibreOffice.

Calculating the Mean of n Numbers

The mean of **N** numbers

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

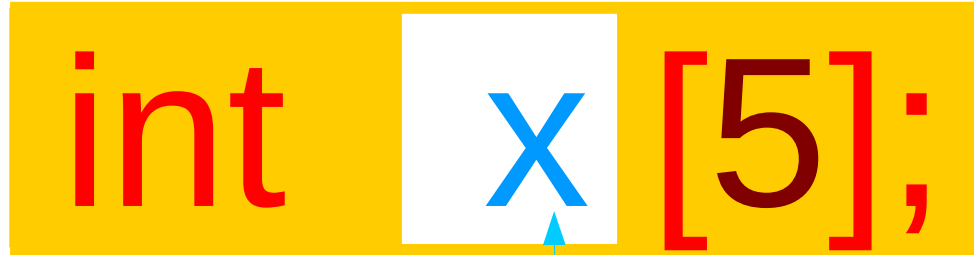
$$m = \frac{1}{5} \sum_{i=0}^4 x_i = \frac{1}{5} (x_0 + x_1 + x_2 + x_3 + x_4)$$

5 variables

x[0] x[1] x[2] x[3] x[4]

Definition of an Array

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

A diagram illustrating the components of the array declaration 'int x[5];'. The text is displayed on a yellow background. The word 'int' is in red, 'x' is in blue, and '[5];' is in red. A white box highlights the 'x', with a blue arrow pointing from the label 'Array Name' below to it. A blue arrow points from the label 'Array Type' below to the 'int'.

Array Type

int [5]

Array Name

x

Element Type

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

Array Type

int [5]

Array Name

x

a constant

Value: the starting address of
5 consecutive int variables

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

Element Type

int

Element

x[i]

i = 0, ... , 4

Using an Array

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

Array Name

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

Element Type : int

int variables

```
x[i]
```

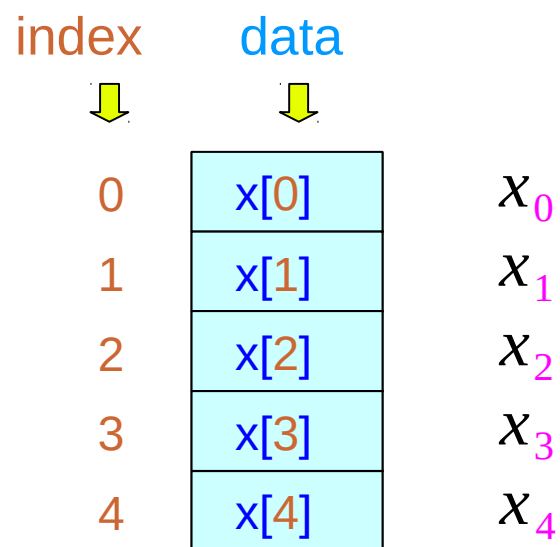
$i = 0, \dots, 4$

Accessing array elements – using an index

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

`x` is an array
with 5 integer elements

5 int variables

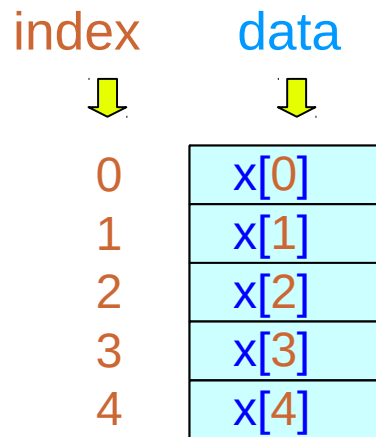


Accessing array elements – using an address

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

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

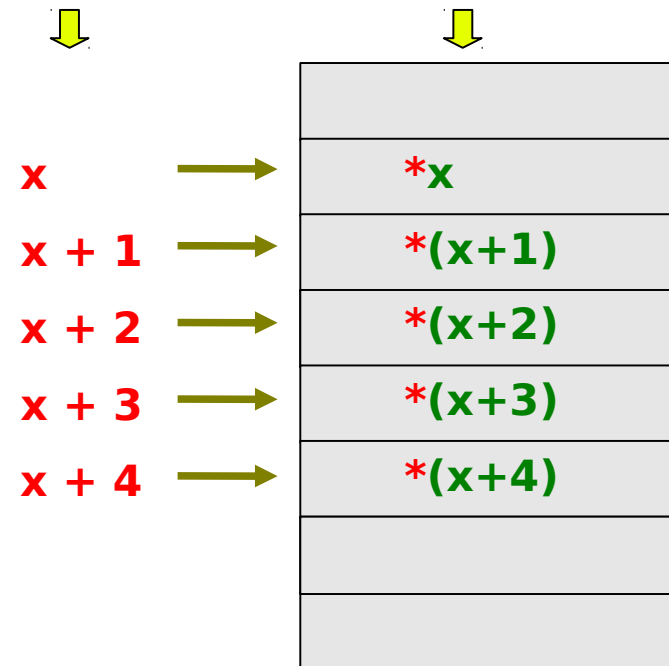
5 int variables



cannot change
address x
(constant)

address

data



Index and Address Notations

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

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

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

i : an index variable [0..4]
 $x[i]$: the $(i+1)^{\text{th}}$ element value

x : the starting address
 $x+i$: the $(i+1)^{\text{th}}$ element's address
 $*(x+i)$: the $(i+1)^{\text{th}}$ element value

A variable expressed by another variable

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

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

treated as a variable

read



$x[i]$



write

read



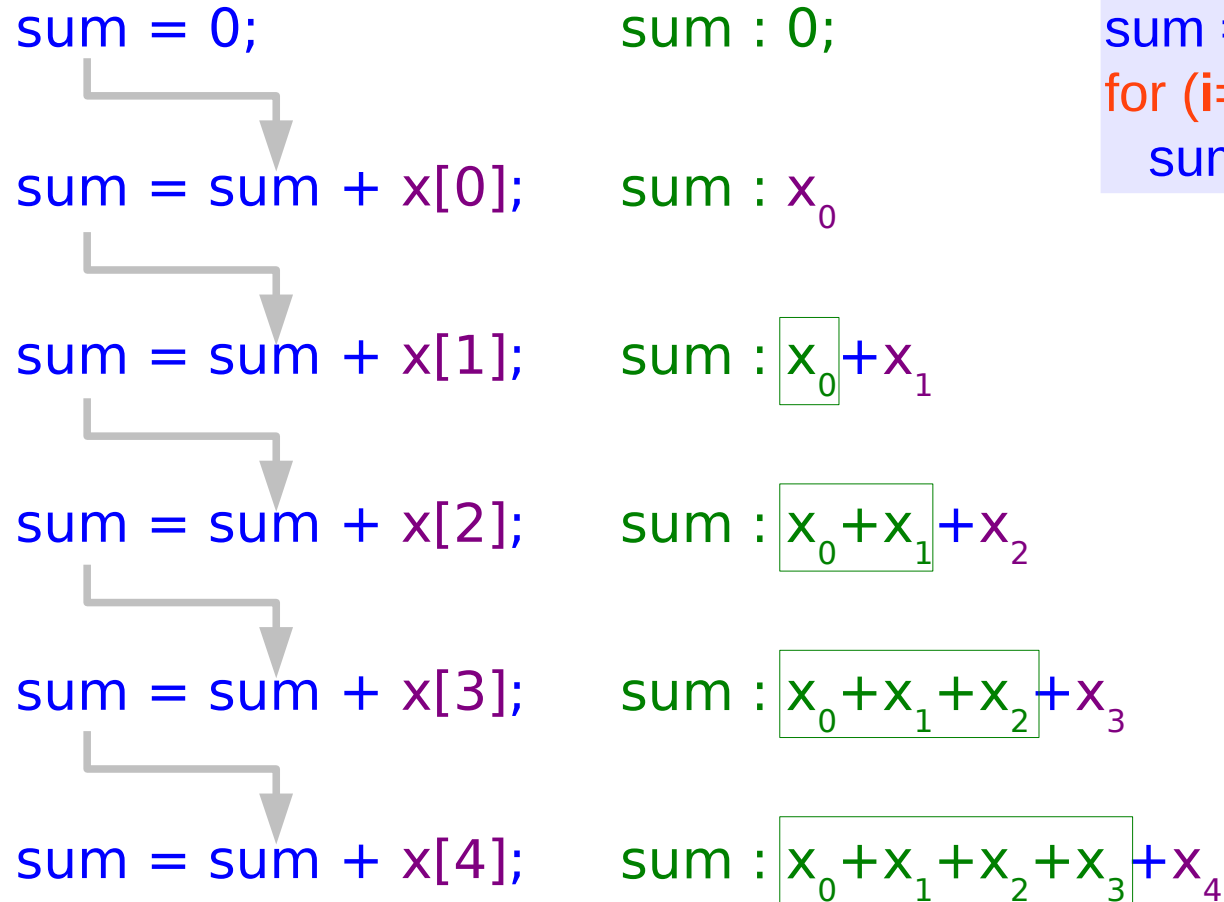
$*(x+i)$



write

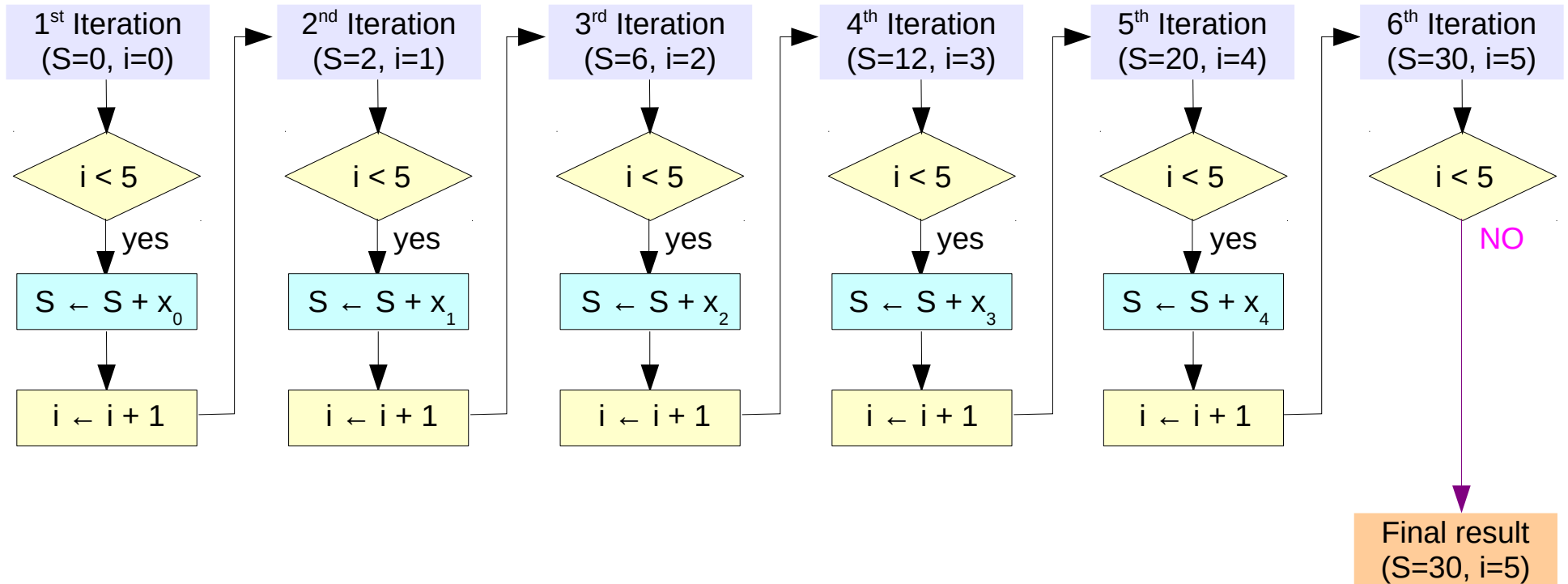
an index variable i

Computing the sum of n numbers (1)



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

Computing the sum of n numbers (2)



```

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

$x_0=2,$
 $x_1=4,$
 $x_2=6,$
 $x_3=8,$
 $x_4=10$

	A	B				
i	1	0	1	2	3	4
x_i		2	4	6	8	10
S	0	2	6	12	20	30

Using Array Names

declaration

```
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)

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

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

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

= { 0, 0, 0, 0, 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, 5 };
```

sizeof(b) = 5*4 = 20 bytes

```
int b [] ;
```

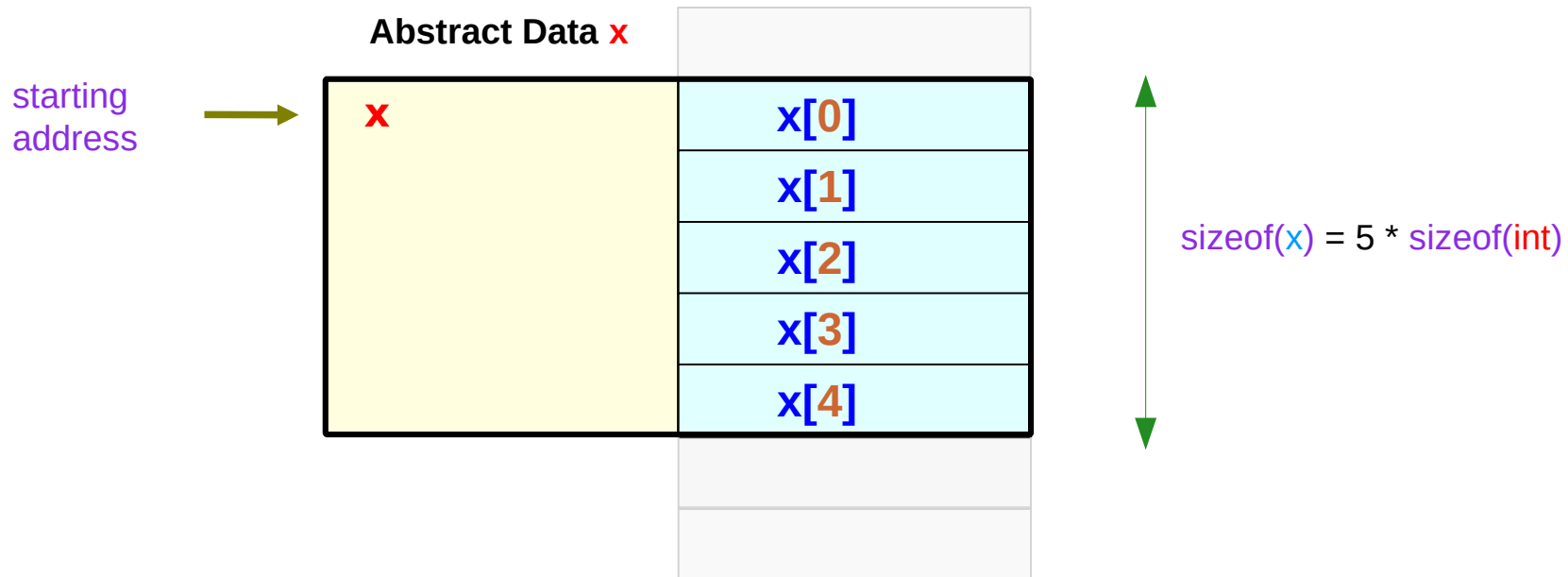
```
int c [3][4] = { { 1, 2, 3, 4},  
                 { 5, 6, 7, 8},  
                 {9,10,11,12} };
```

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

Abstract data **x**

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

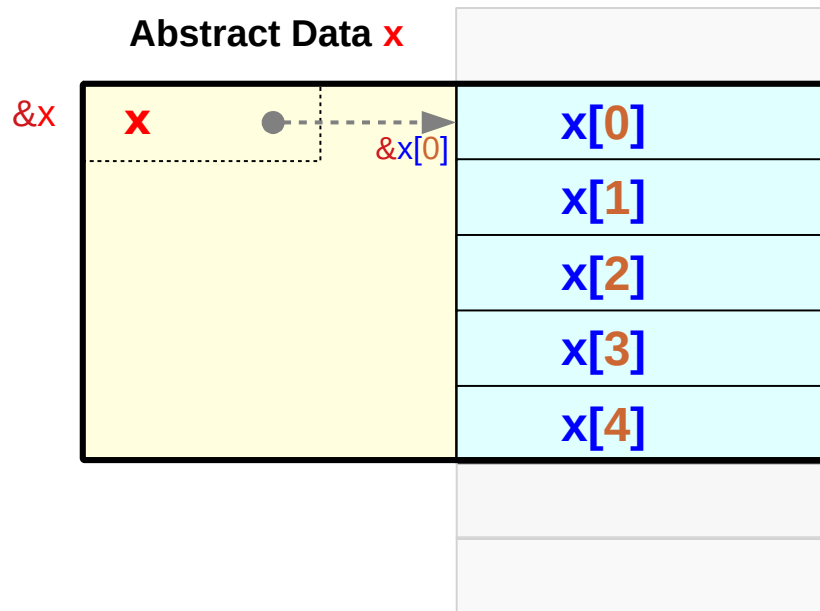
x holds the *starting address*
of **5** consecutive *int* variables



Abstract data x as a pointer

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

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



pointer relation

$x \equiv \&x[0]$

$*x \equiv x[0]$

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

the starting address of 5 consecutive int variables

not a real pointer x

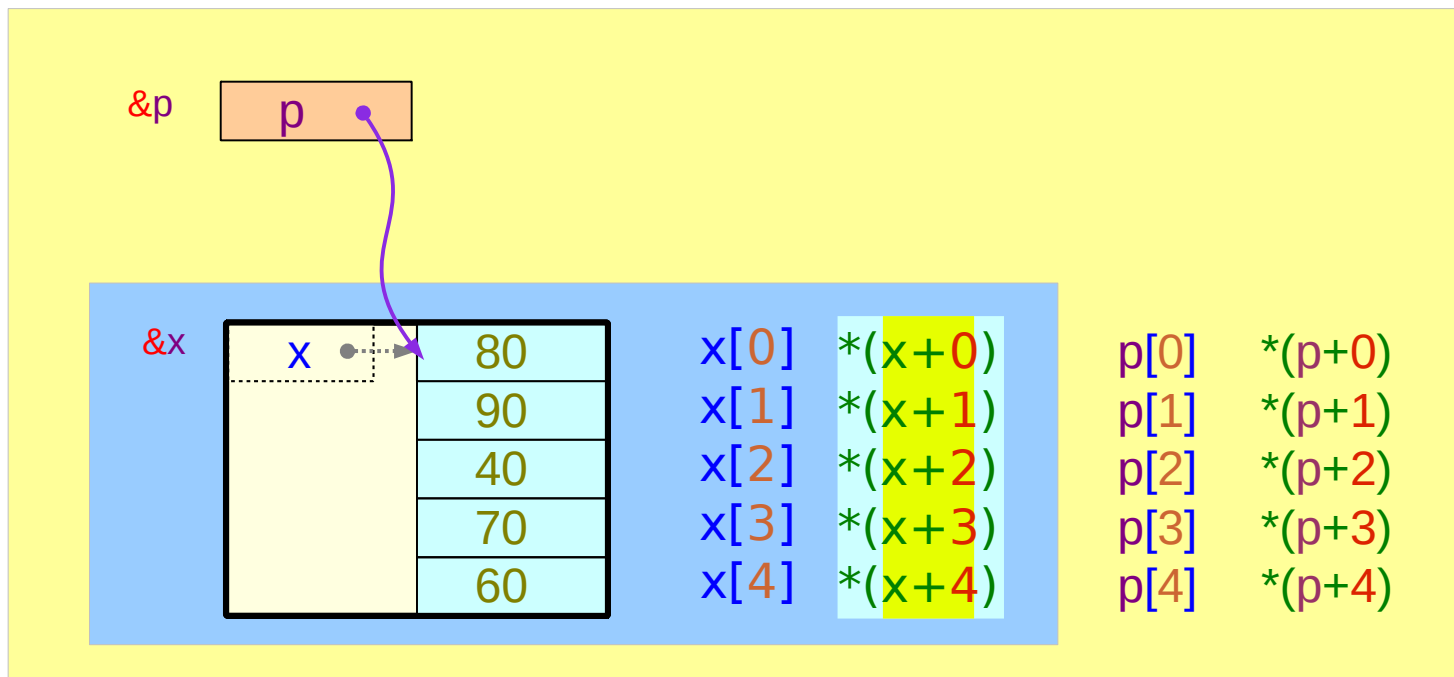
Accessing an Array with a Pointer Variable

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

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

x is a constant symbol
cannot be changed

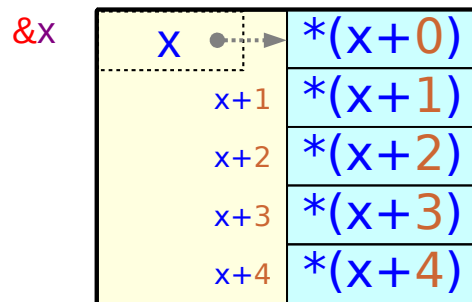
p is a variable
can point to other addresses



An Array Name and a Pointer Variable

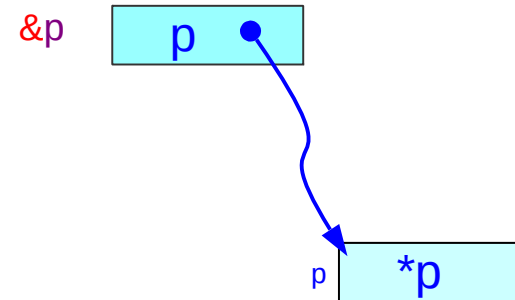
```
int x [5] ;
```

x : an array name (constant)
Value: the starting address of
5 consecutive int variables



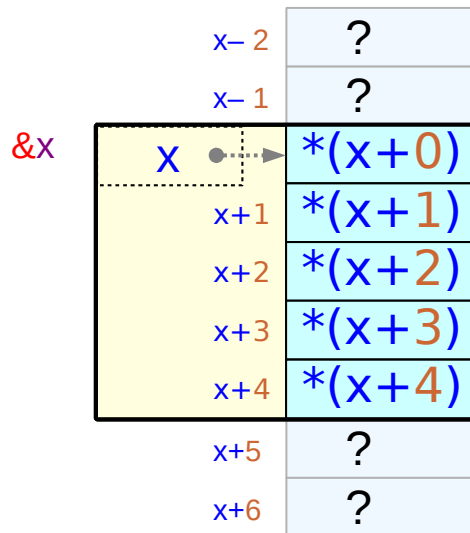
```
int * p ;
```

p : an variable name
Value: the address
of an int variable

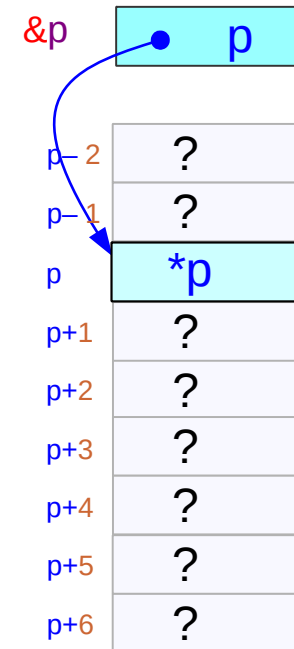


Out of range index

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



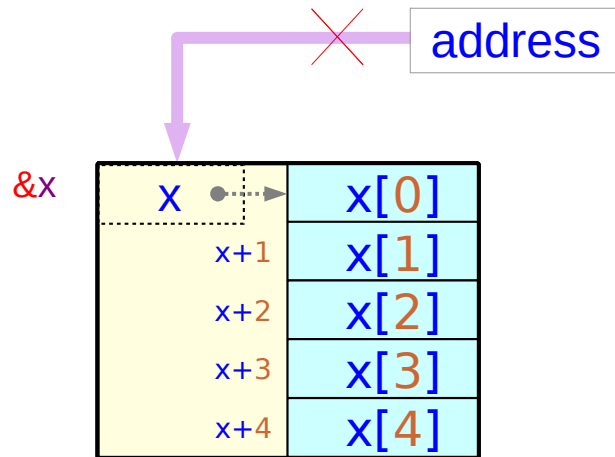
```
int * p ;
```



A programmer's responsibility

Assignment of an address

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



x is not a variable
but a constant symbol

x and $\&x$ give the same
value of $\&x[0]$

This address is embedded
as a constant in the executable file
and cannot be changed

```
int * p ;
```



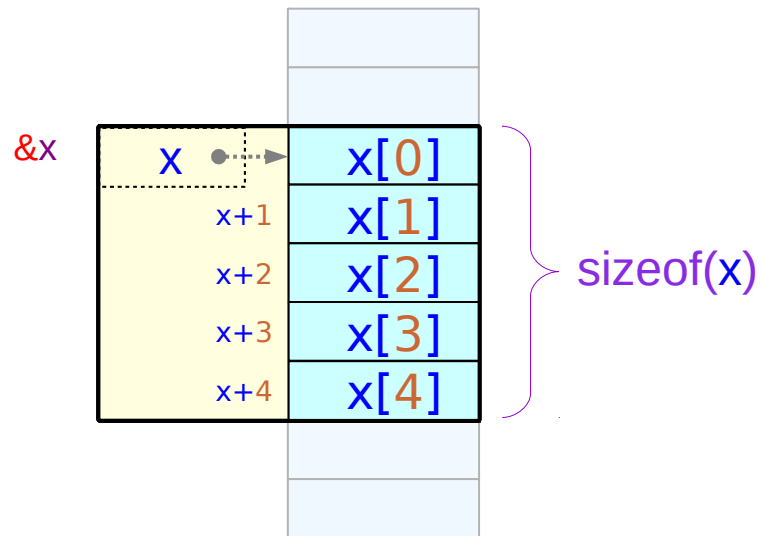
p is a variable

has an allocated memory location

its value can be changed

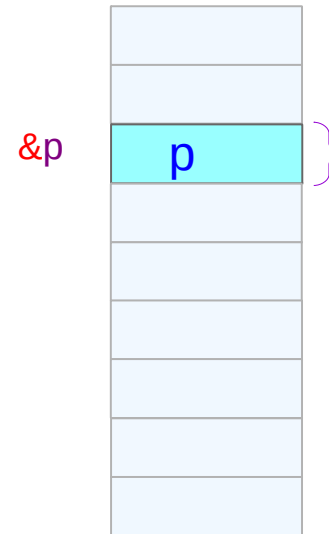
Size

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



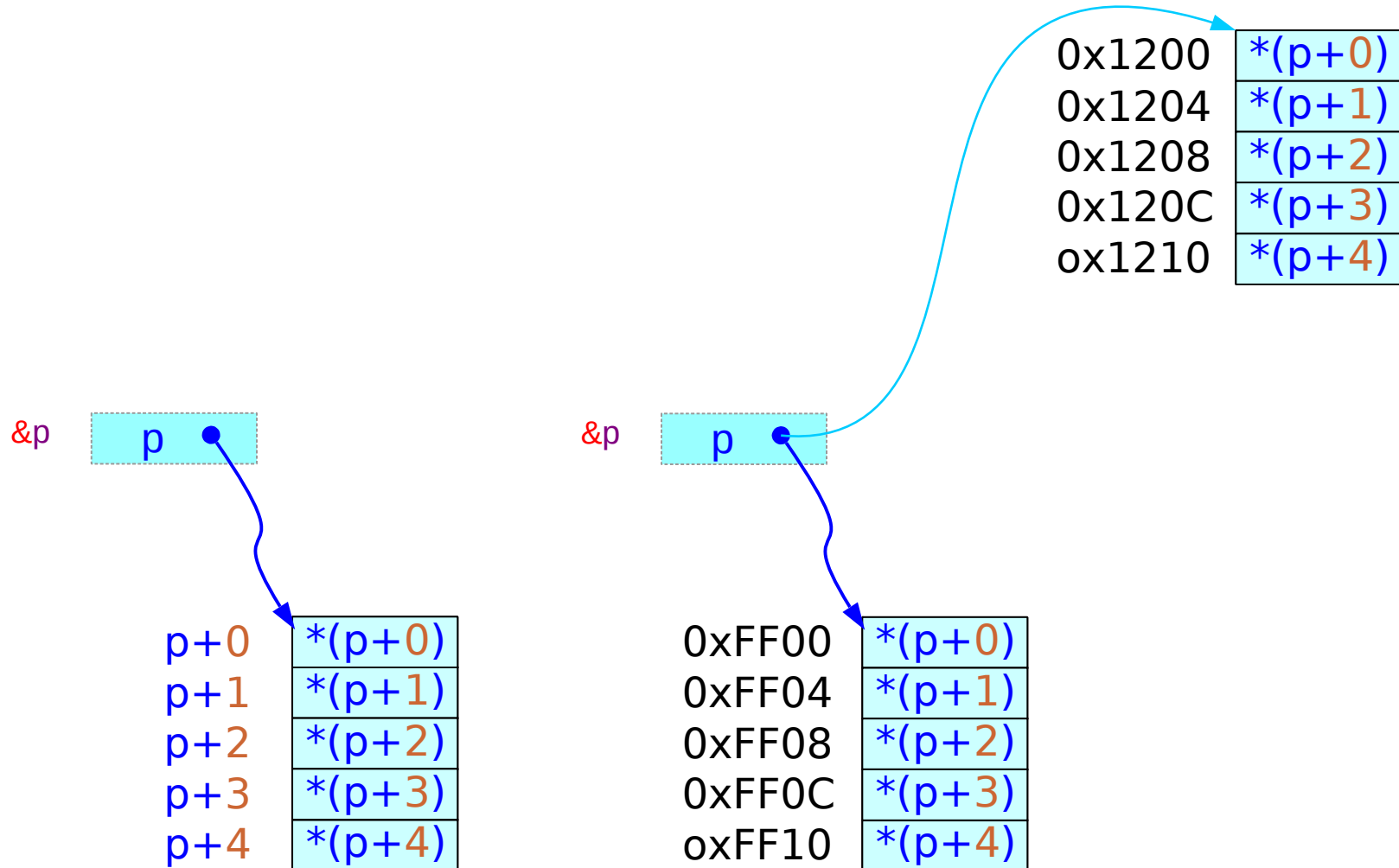
$\text{sizeof}(x) = 5 * \text{sizeof}(\text{int})$

```
int * p ;
```



$\text{sizeof}(p) = \text{size of a pointer}$
 $= 4 / 8 \text{ bytes}$

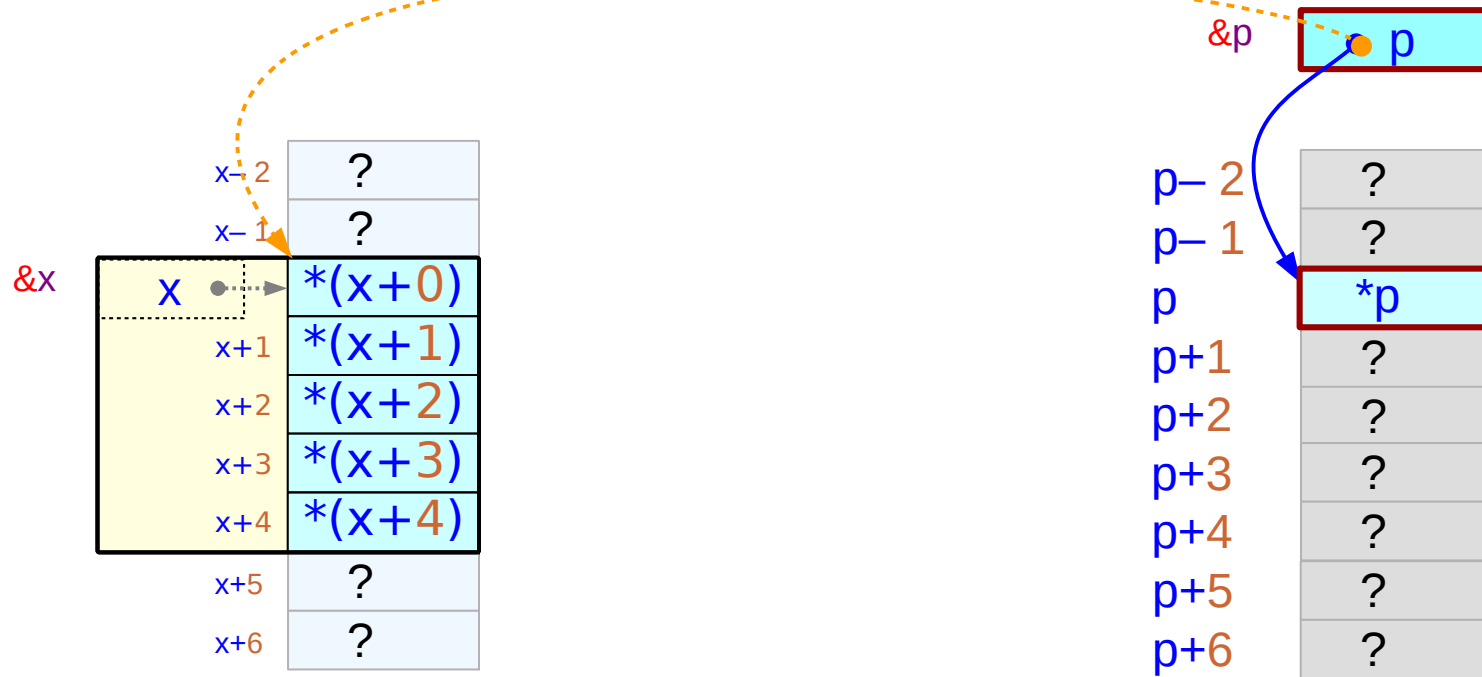
Pointer variable can point different locations



Pointer to an integer

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

```
int * p ;
```



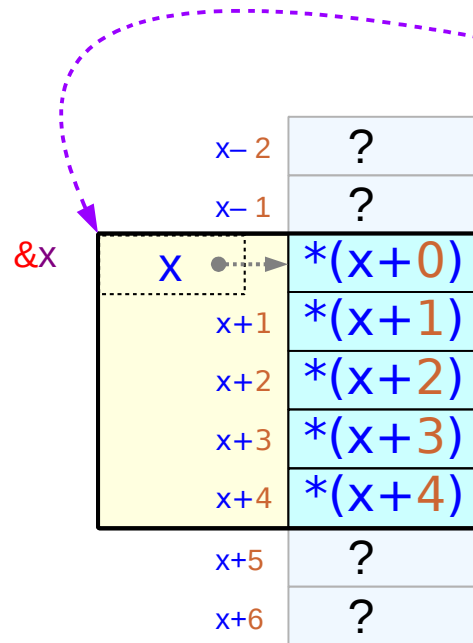
```
*p = *x ;
```



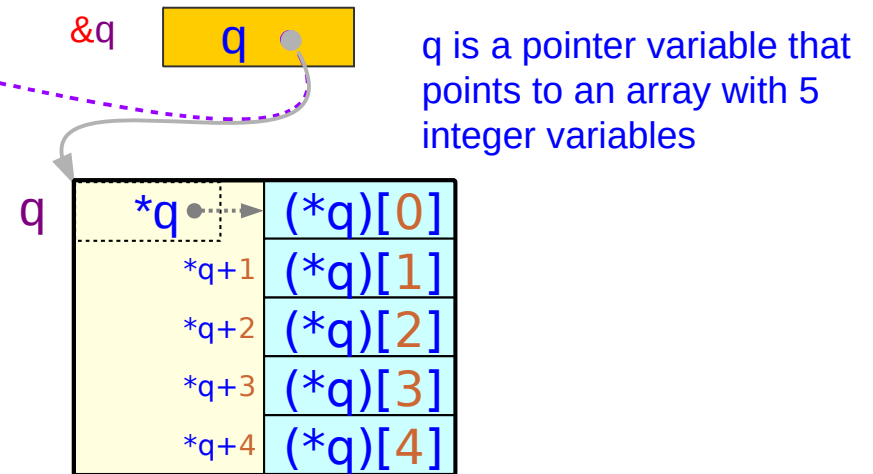
```
p = x ;
```


Pointer to an array

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



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



```
*q = x ;
```

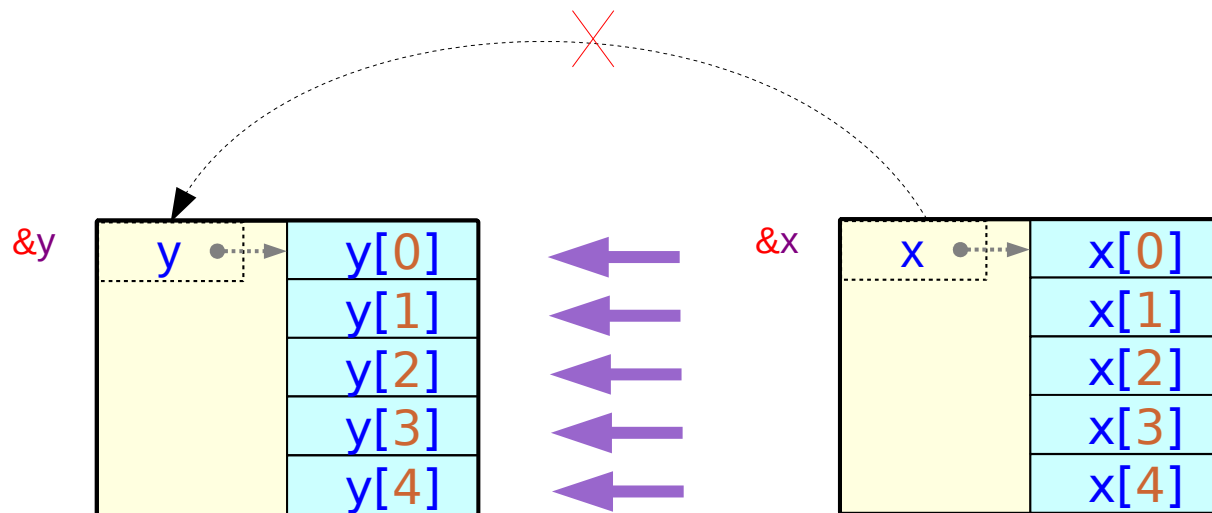


```
q = &x ;
```

* not frequently used feature

Copying an Array to another Array

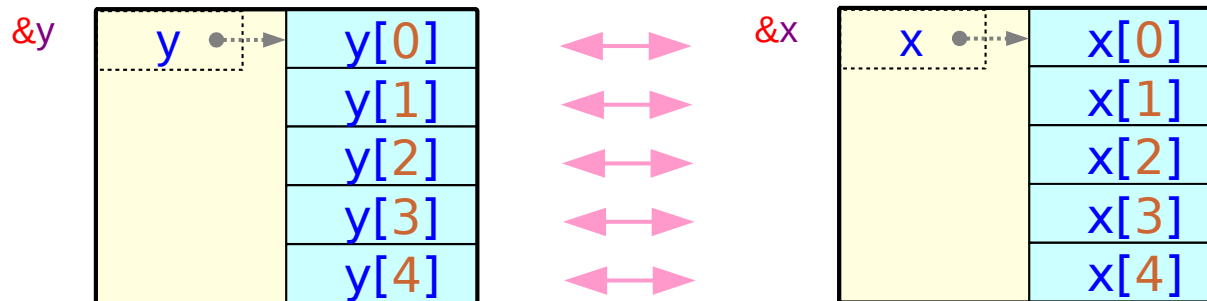
```
int x [5] = { 1, 2, 3, 4, 5 };  
int y [5] ;  
y = x;
```



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

Comparing an Array with another Array

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



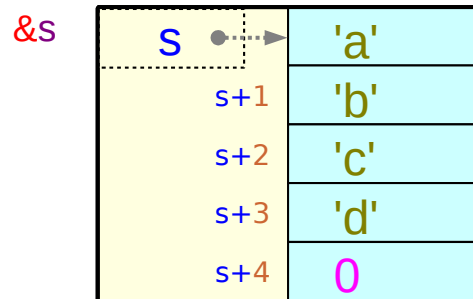
```
EQ=1;  
for (i=0; i<5; ++i)  
    EQ &= (y[i] == x[i]);
```

Initialized Character Arrays and Pointers (1)

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

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

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

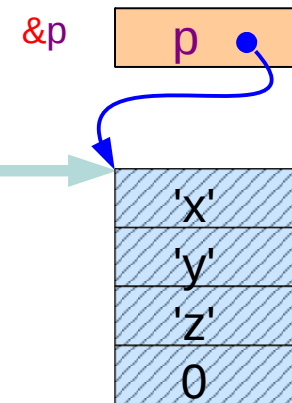


can change the value of any element

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

a compiler determined constant address

a **constant** character string (array)



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

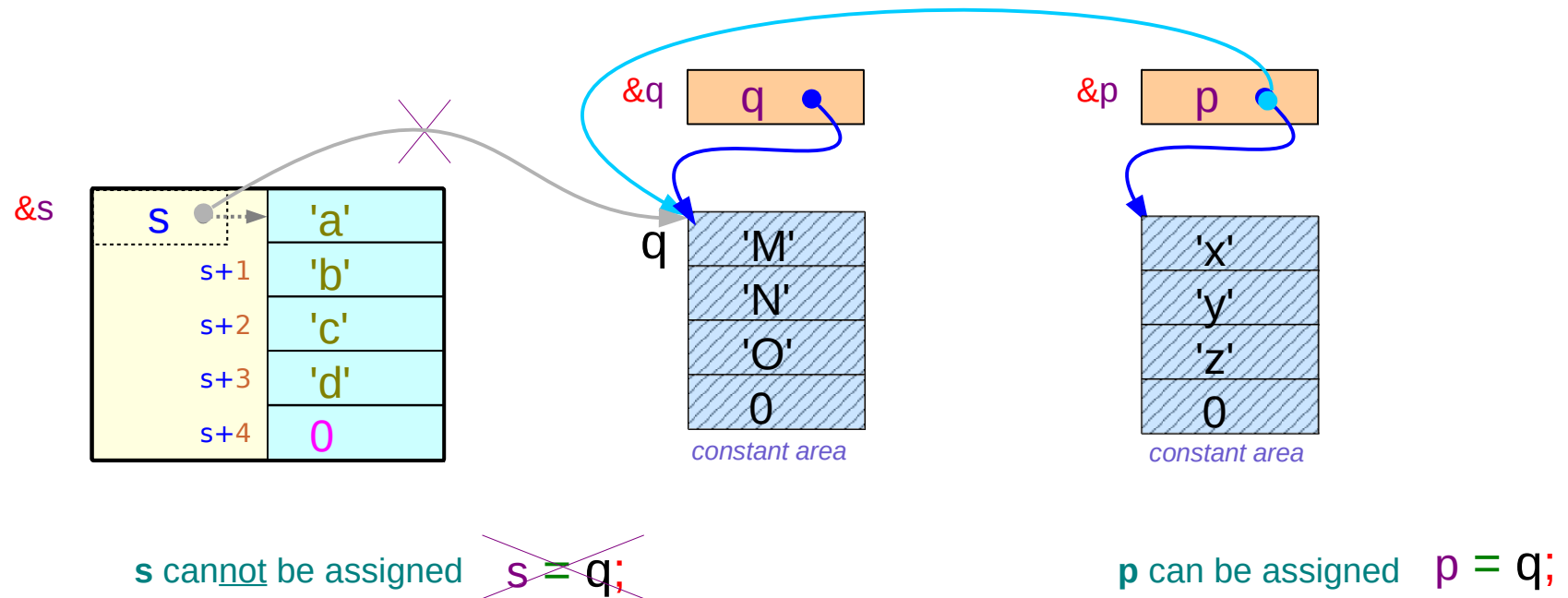
```
*p = 'm';  
p[0] = 'm';
```

Initialized Character Arrays and Pointers (2)

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

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

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

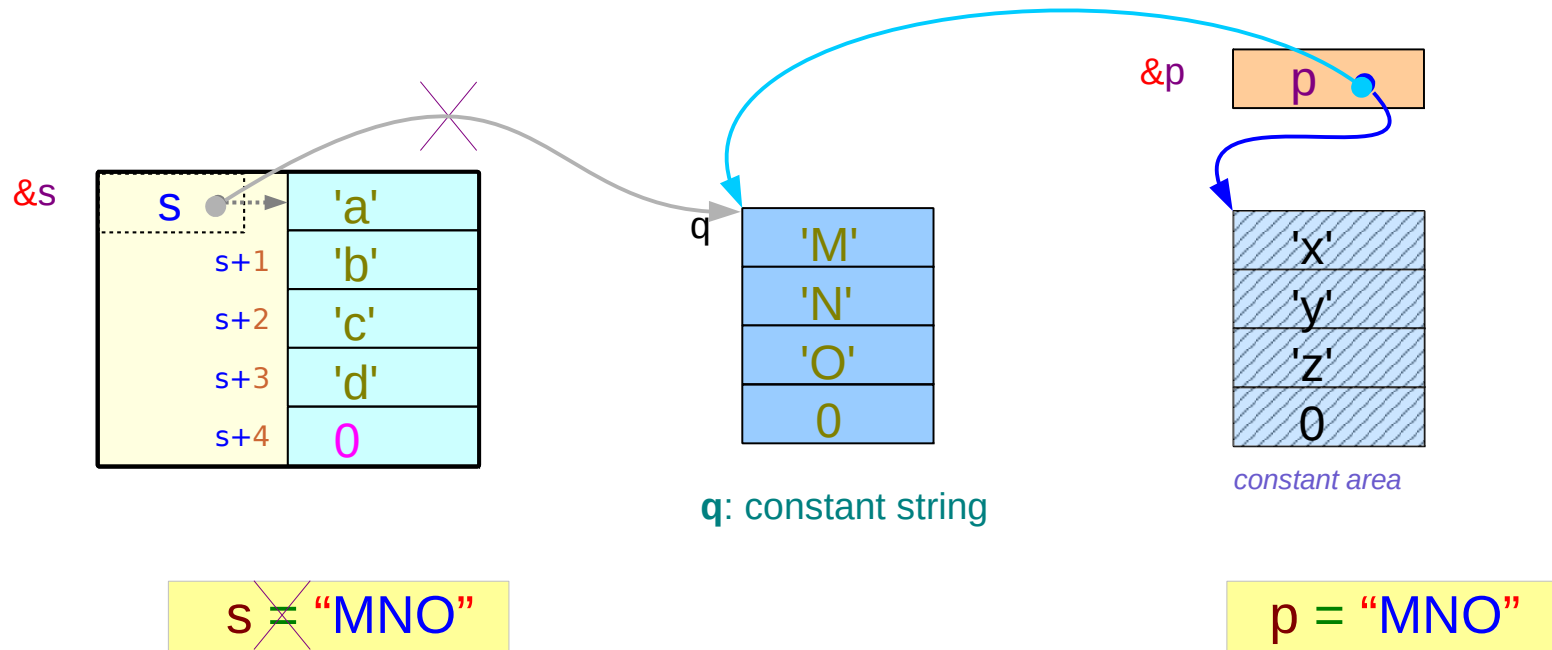


Assigning a constant character string

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

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

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

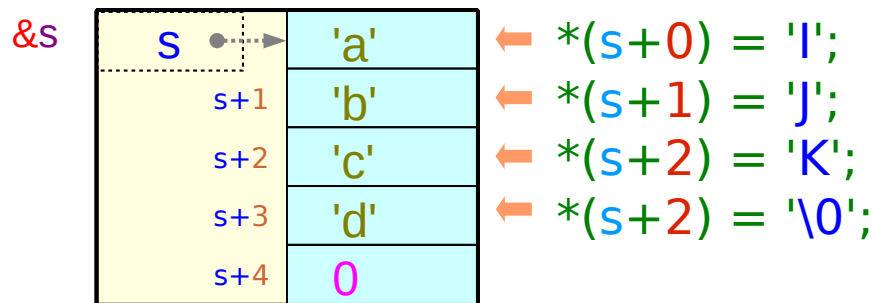


Copying a string

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

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

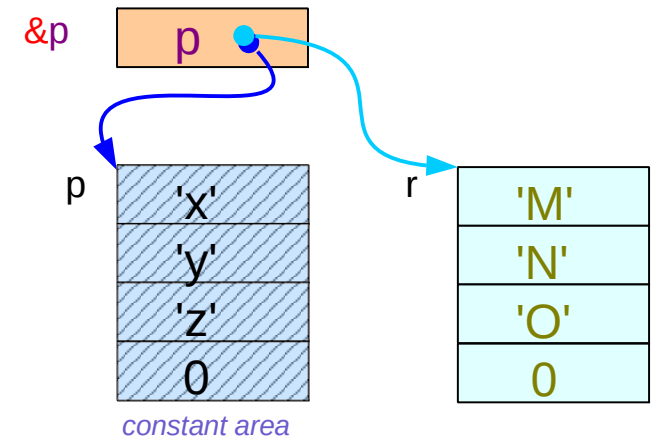
```
char *p = "xyz";
```



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

p: constant string

r: non-constant string



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

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

Uninitialized Character Arrays and Pointers

```
char s [5];  
char *p;
```

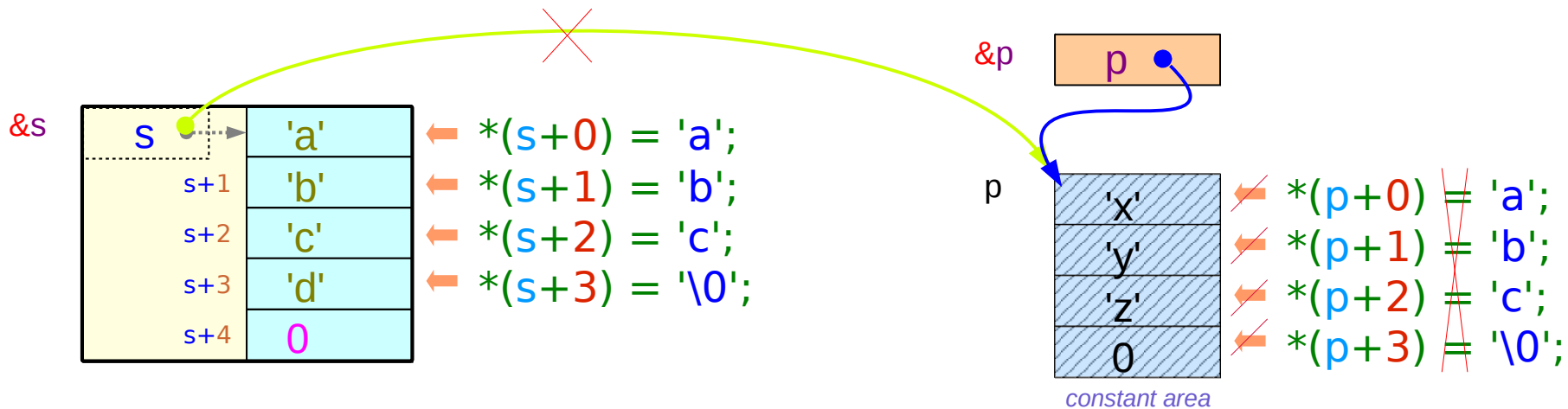
```
s = "xyz";      char * const s  
p = "xyz";      const char * p
```

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

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

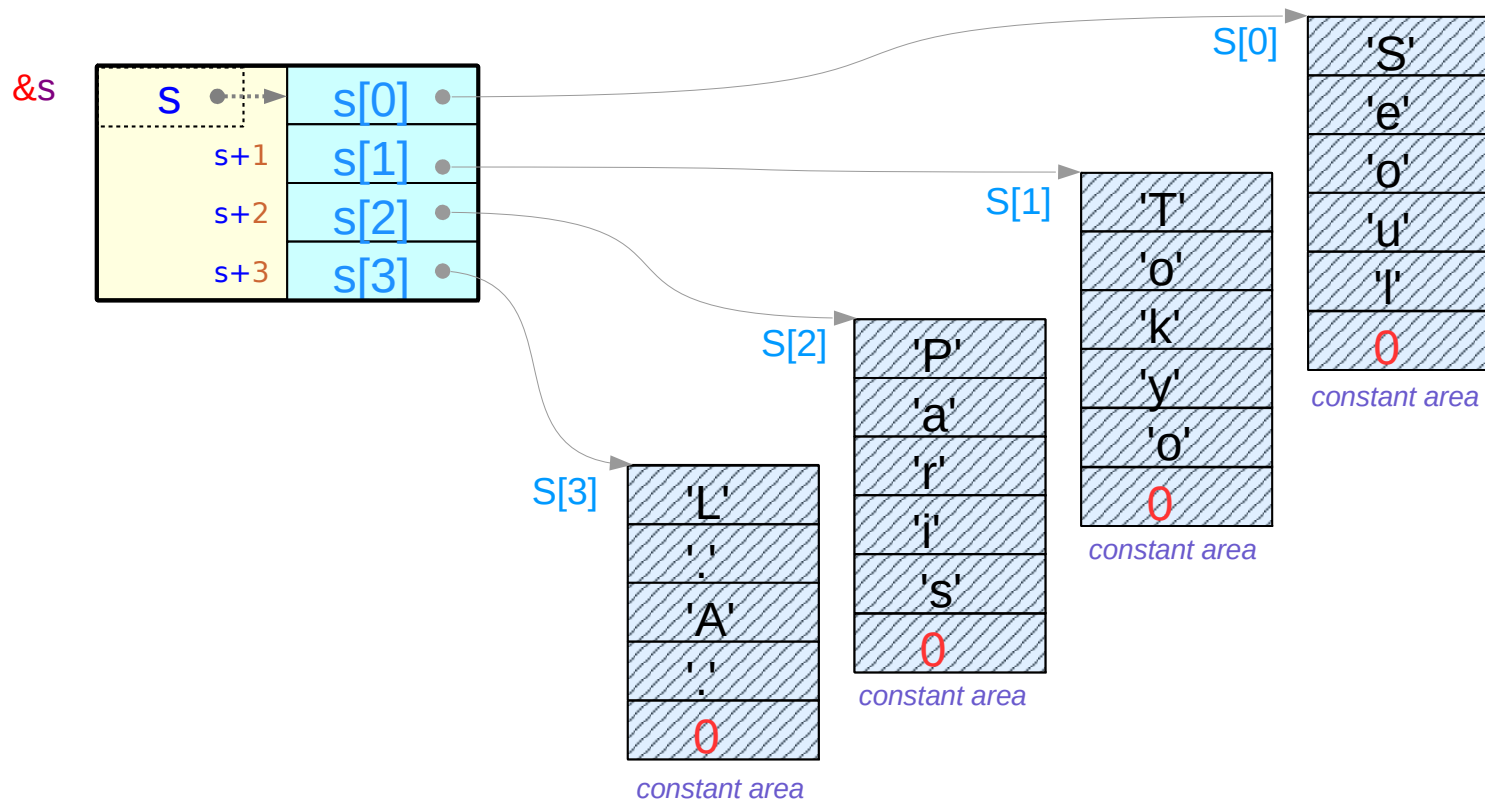
s cannot point to other location

p points to a string constant
which cannot be changed



Arrays of Pointers

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



A[] Notation

1. An array definition with **initializers**

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

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

`func(int x []) { ... }  int * const x (x : a constant)`

`compatible  int * p (p : a variable)`

A[][n] Notation

1. An array definition with **initializers**

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

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

`func(int x [][3]) { ... }`  `int (* const x)[3]` (constant)

not compatible  `int ** p` (variable)

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

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

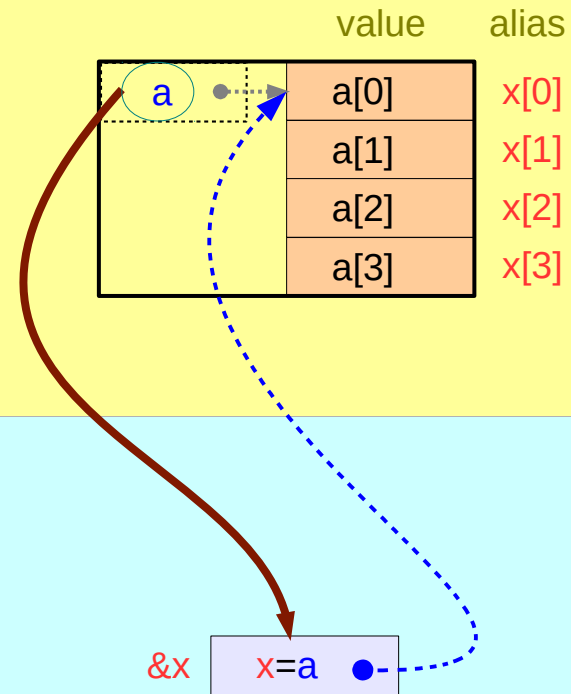
```
func( a );
```

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

```
    ...
```

```
}
```

or `int x []`



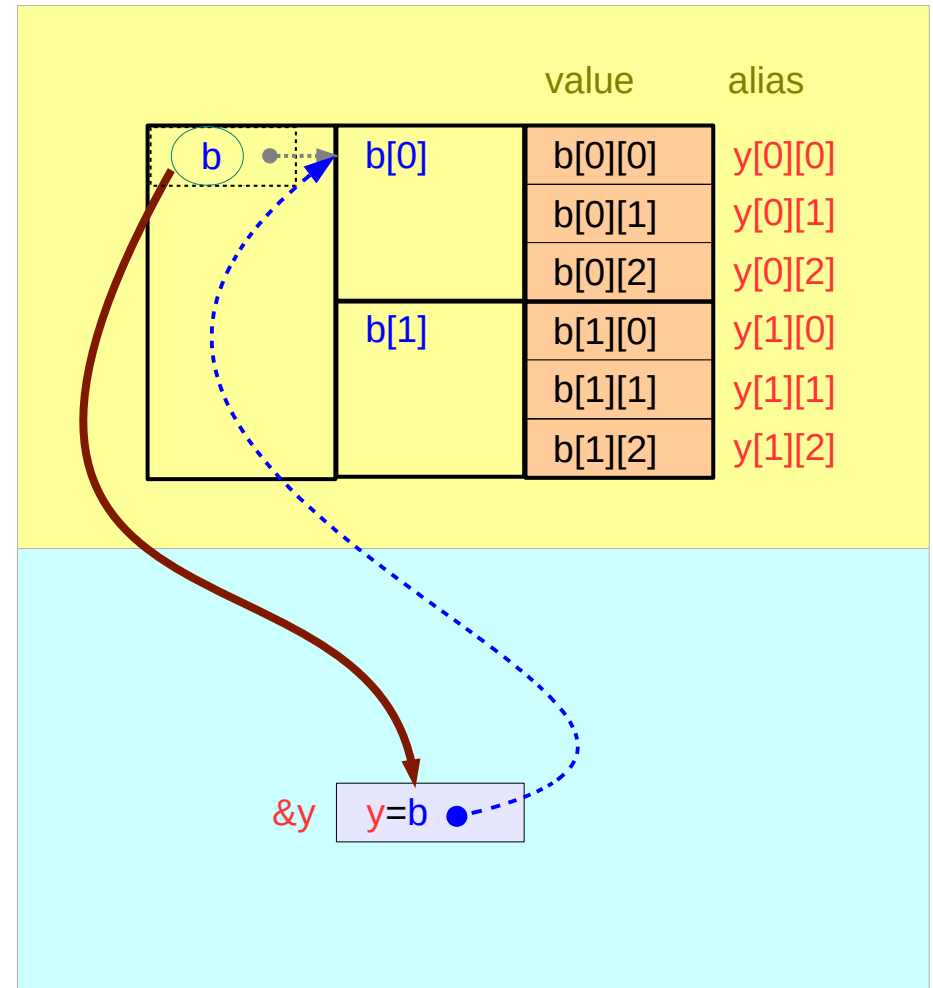
Passing 2-d Arrays – using 1-d array pointer

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

```
func( b );
```

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

or `int y [][3]`



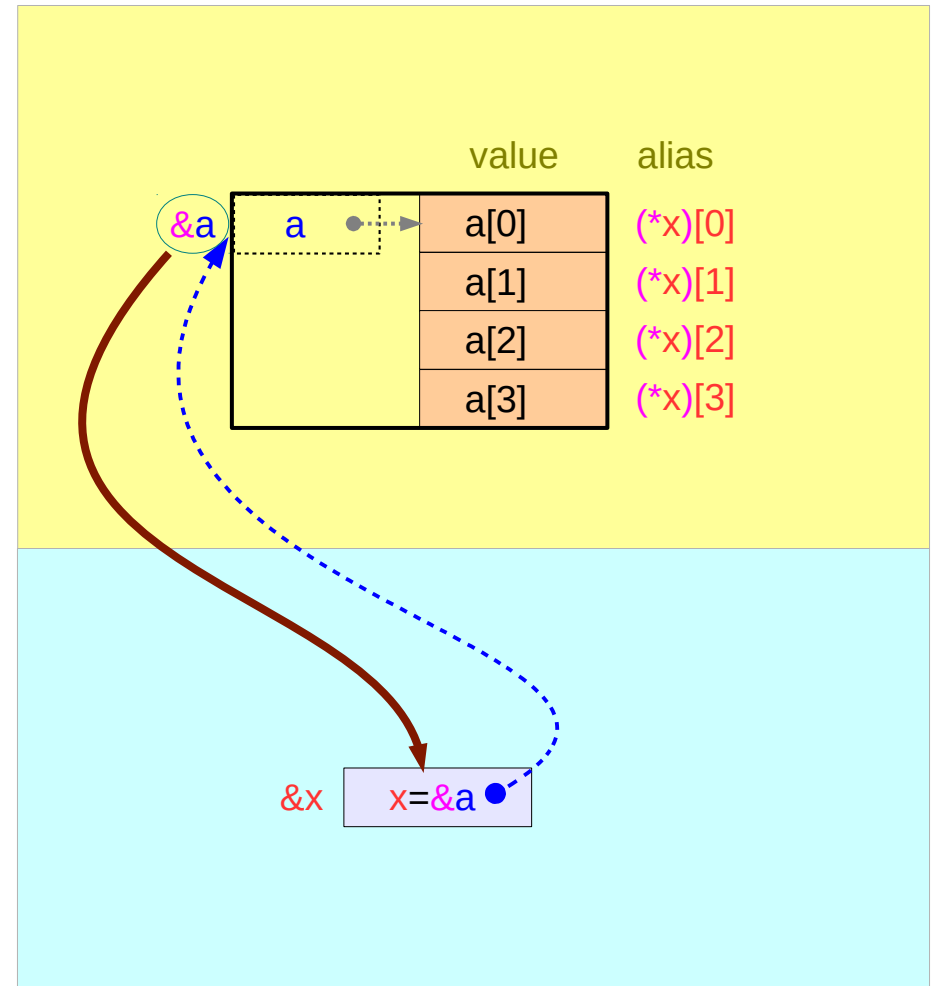
Passing 1-d Arrays – using 1-d array pointer

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

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

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

or `int x [][4]`



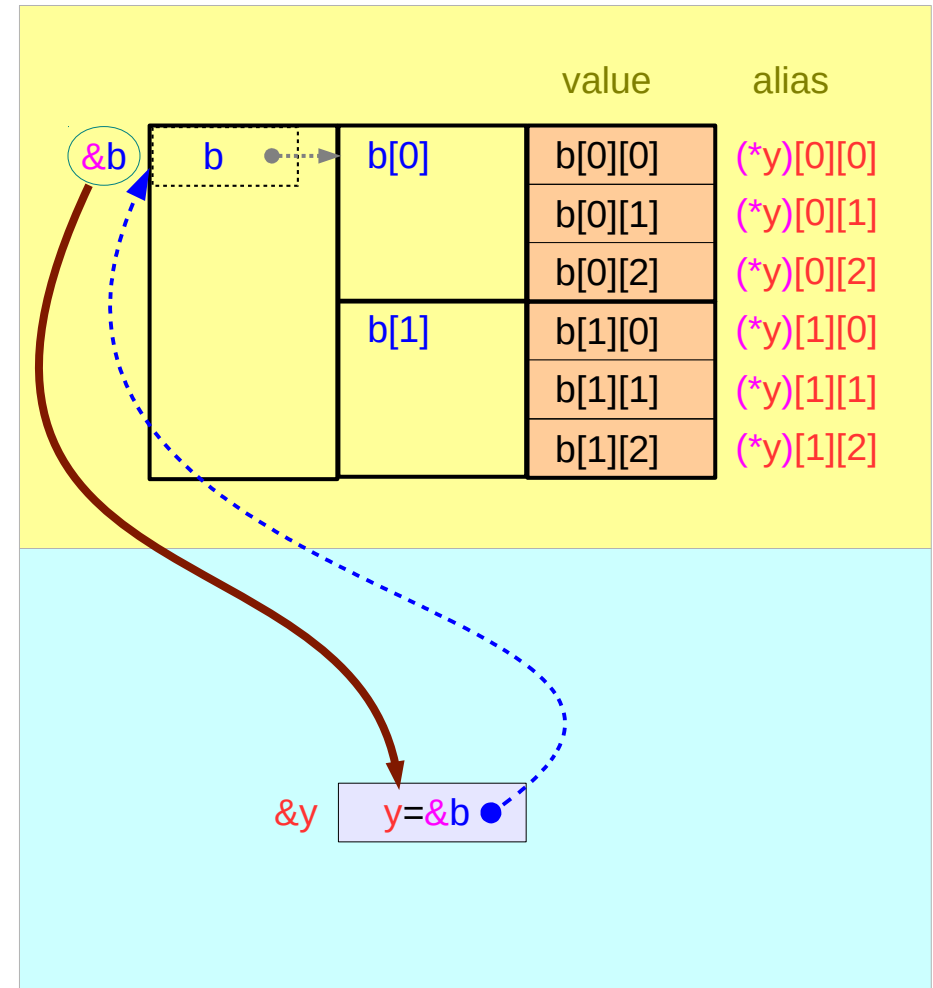
Passing 2-d Arrays – using 2-d array pointer

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

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

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

or `int y [][2][3]`



Passing an individual element by value

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

```
func( a[3] );
```

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

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

```
func( b[0][1] );
```

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


Passing an individual element by reference

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

```
func( &a[3] );
```

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

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

```
func( &b[0][1] );
```

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

Array Type definition

```
typedef int AType [10] ;
```

```
AType A;
```

≡

```
int A [10] ;
```

```
A [0] = 100;
```

```
A [1] = 200;
```

```
A [2] = 300;
```

```
A [m] = 400;
```

Pointer to Array Type definition

```
typedef int AType [10] ;
```

```
AType A, *q;
```

```
q = &A ;
```

```
typedef int (* PType) [10] ;
```

```
PType p;
```

```
p = &A ;
```

```
p = q ;
```

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

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

- Types of array names
- Values of array names

2-d array definition

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

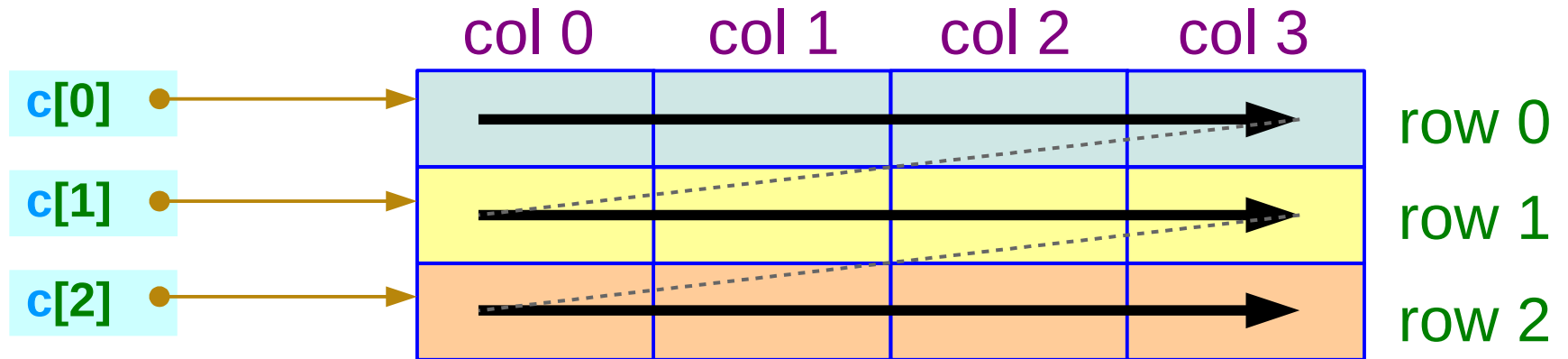
A matrix view

	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 as a matrix

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

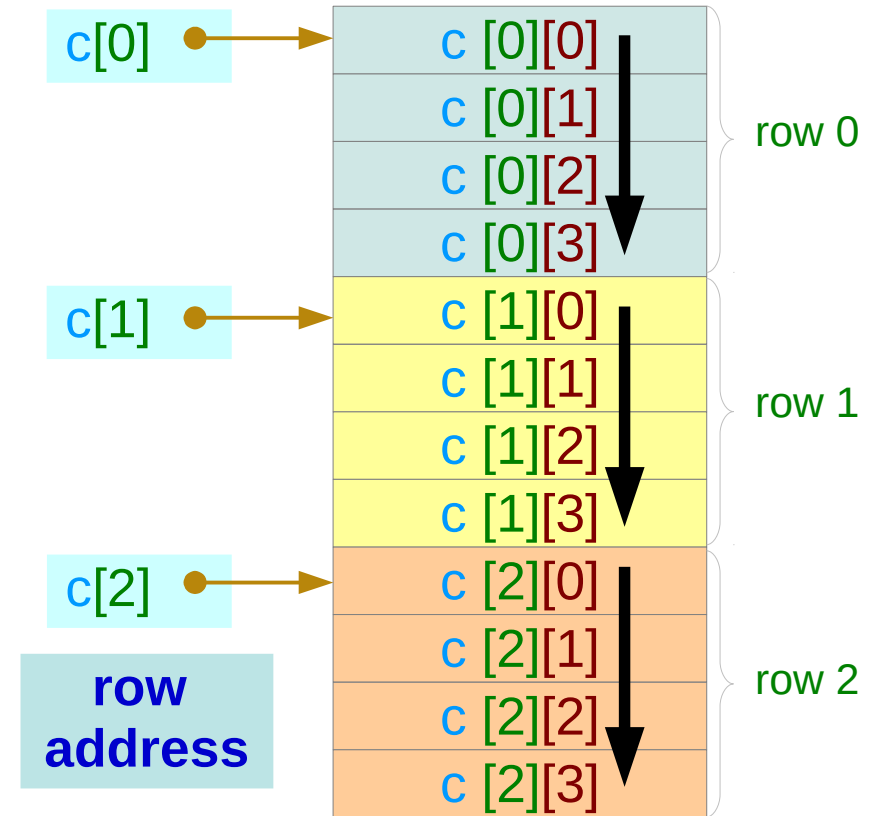
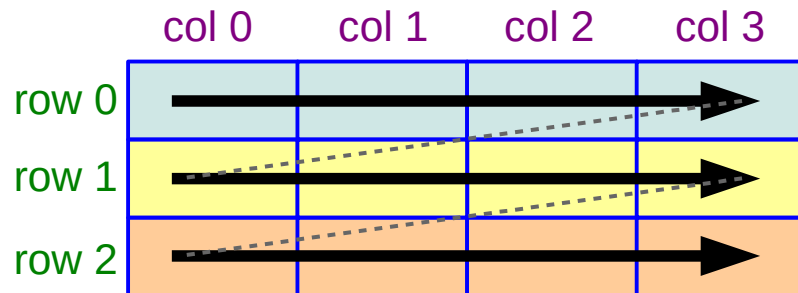
row major ordering



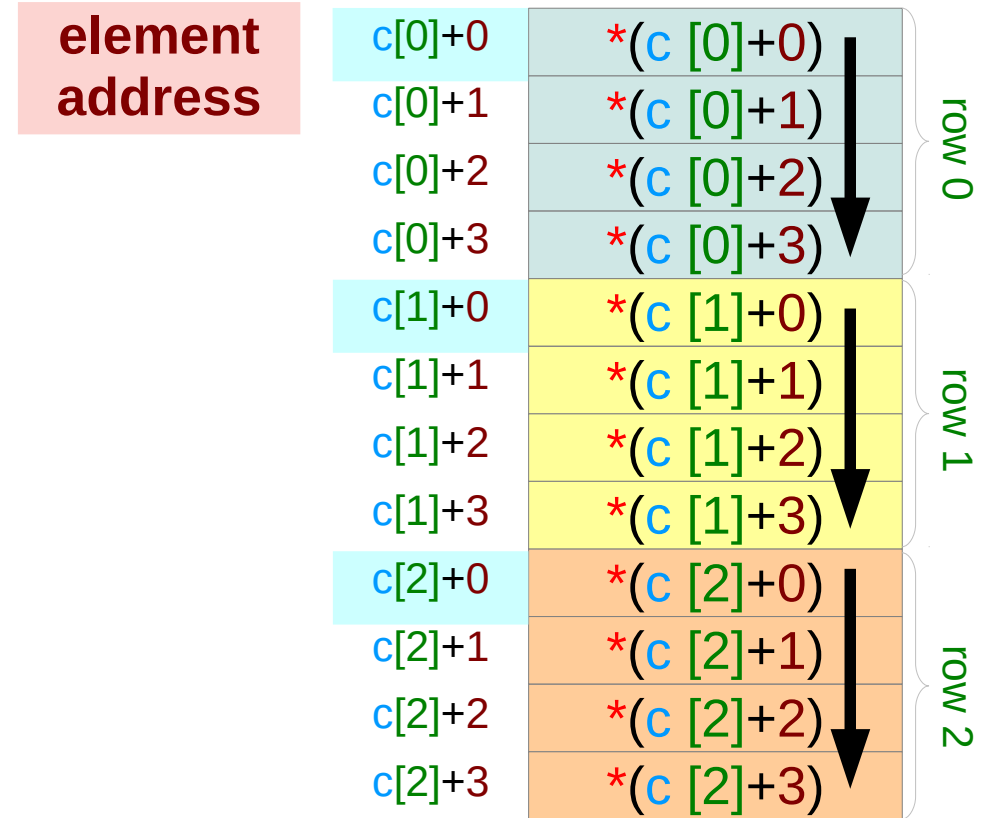
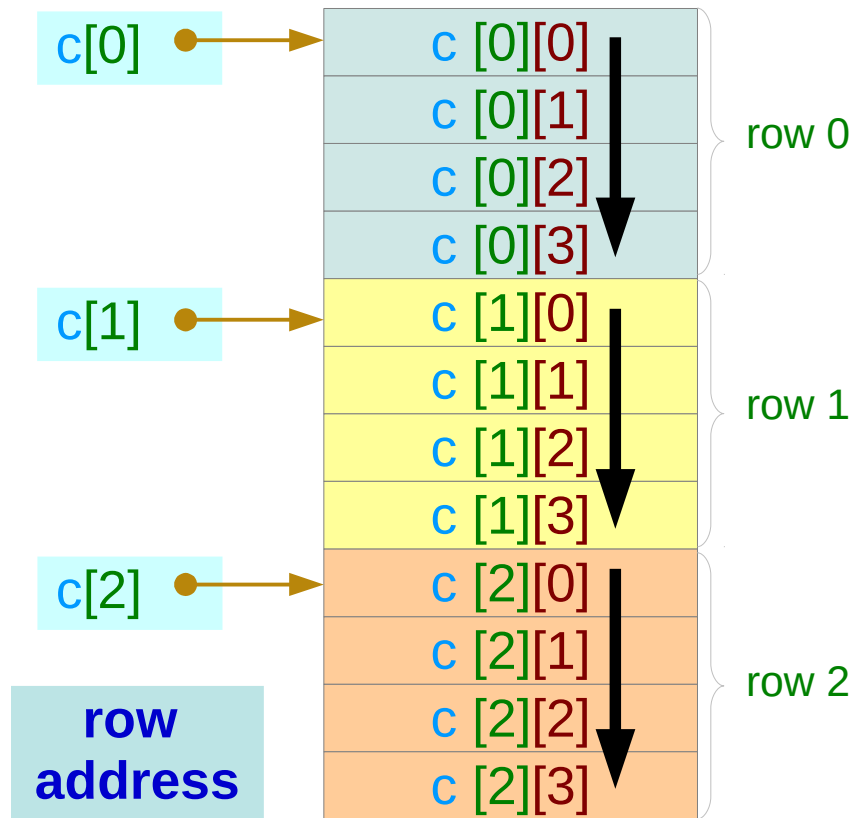
consider each $c[i]$ as a pointer to the first element of each 4 element array

2-d array stored as a linear array

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



Row Address and Element Address

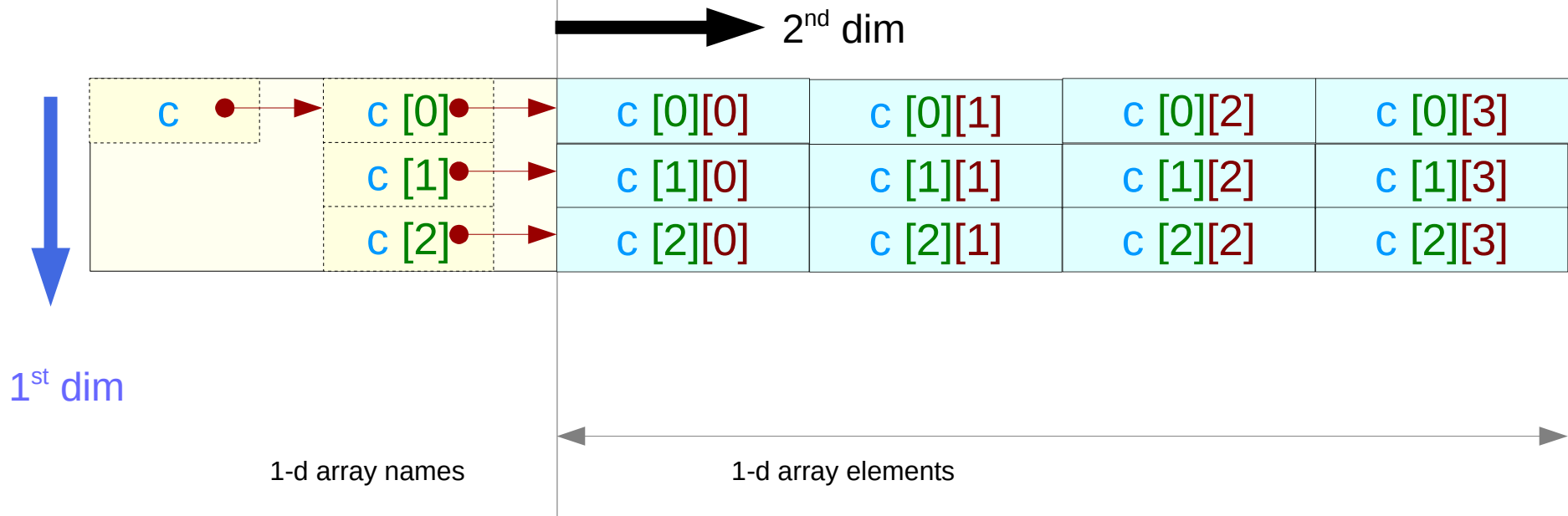


row address + column address

A 1-d array pointer – extending a dimension

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

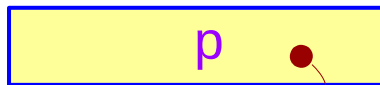
can be viewed as a 2-d array name
: an additional dimension is extended



A 1-d array pointer – extending a dimension

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

1-d array pointer



can be viewed as a 2-d array name
: an additional dimension is extended

→ 2nd dim



1st dim

p+0	*(p+0)	•	→	p[0][0]	p[0][1]	p[0][2]	p[0][3]
p+1	*(p+1)	•	→	p[1][0]	p[1][1]	p[1][2]	p[1][3]
p+2	*(p+2)	•	→	p[2][0]	p[2][1]	p[2][2]	p[2][3]
p+3	*(p+3)	•	→	p[3][0]	p[3][1]	p[3][2]	p[3][3]

• • •

• • •

• • •

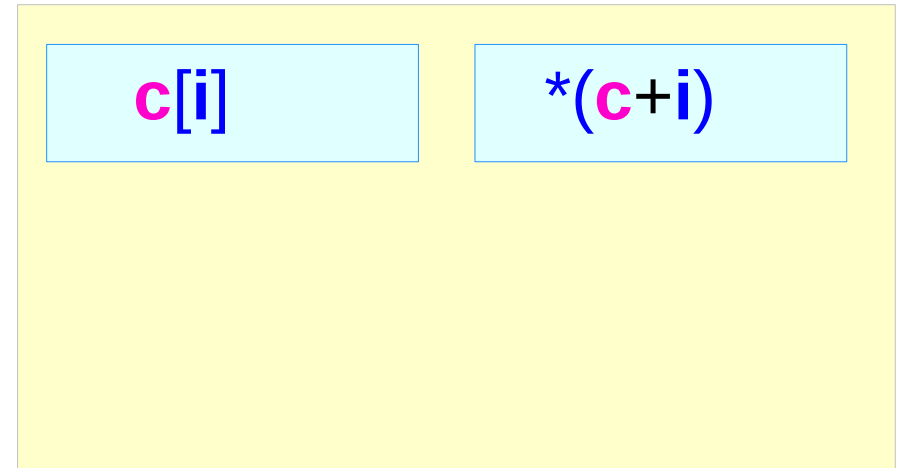
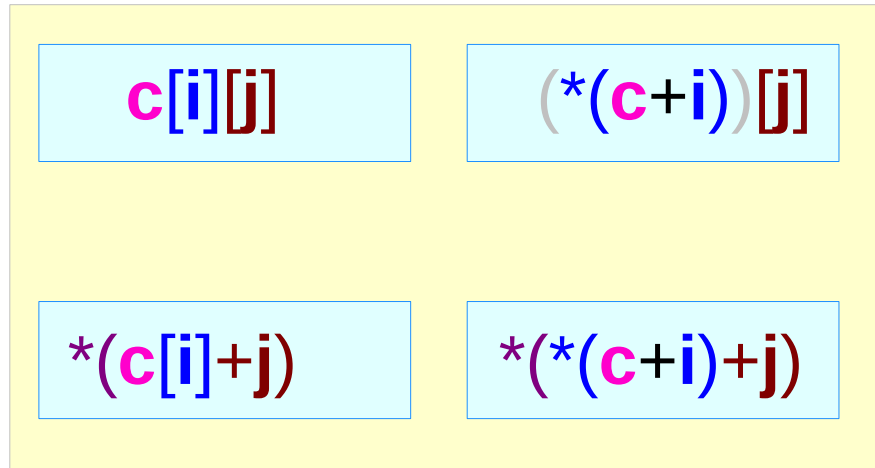
• • •

• • •

1-d array names

1-d array elements

A 2-D array element address



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

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

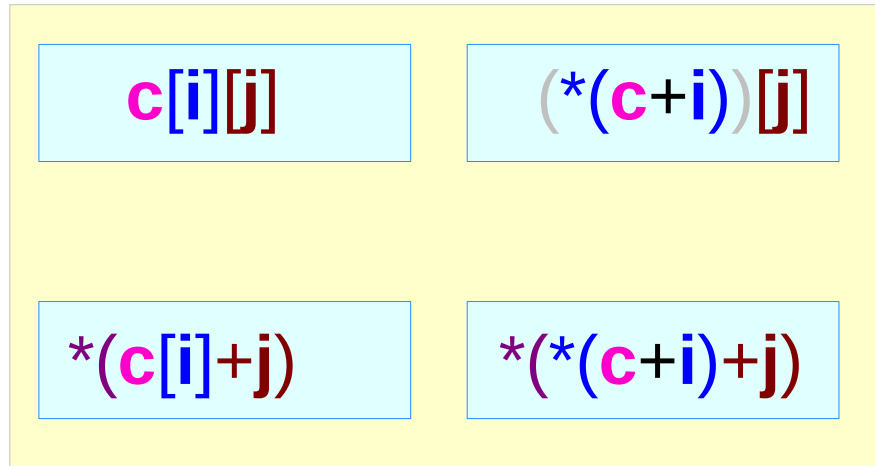
the address of $c[i][j]$ is $*(c+i)+j$

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

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

the address of $c[i]$ is $(c+i)$

A 2-D array element address



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

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

the address of $c[i][j]$ is $*(c+i)+j$

Element address = Row address + Column address

$$c[0]+0 = *(c+0)+0$$

$$c[0]+1 = *(c+0)+1$$

$$c[0]+2 = *(c+0)+2$$

$$c[0]+3 = *(c+0)+3$$

$$c[1]+0 = *(c+1)+0$$

$$c[1]+1 = *(c+1)+1$$

$$c[1]+2 = *(c+1)+2$$

$$c[1]+3 = *(c+1)+3$$

$$c[2]+0 = *(c+2)+0$$

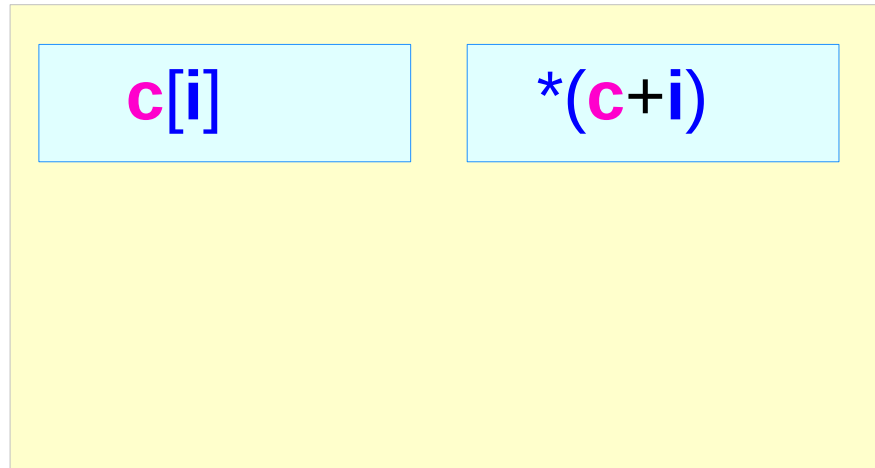
$$c[2]+1 = *(c+2)+1$$

$$c[2]+2 = *(c+2)+2$$

$$c[2]+3 = *(c+2)+3$$

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

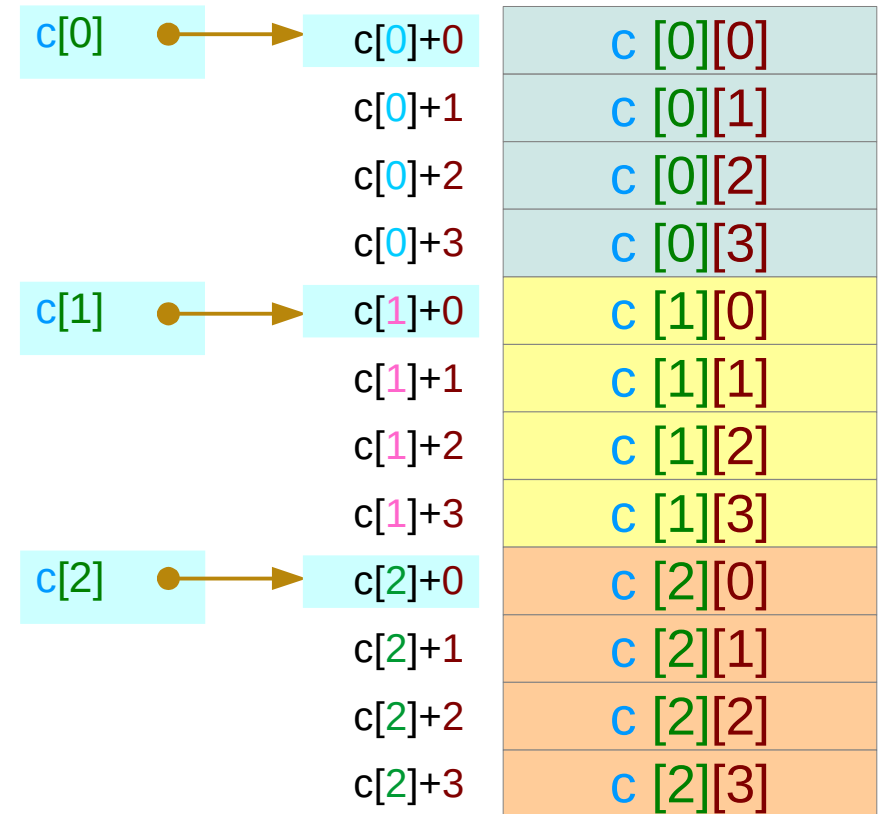
A 2-D array element address



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

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

Row address

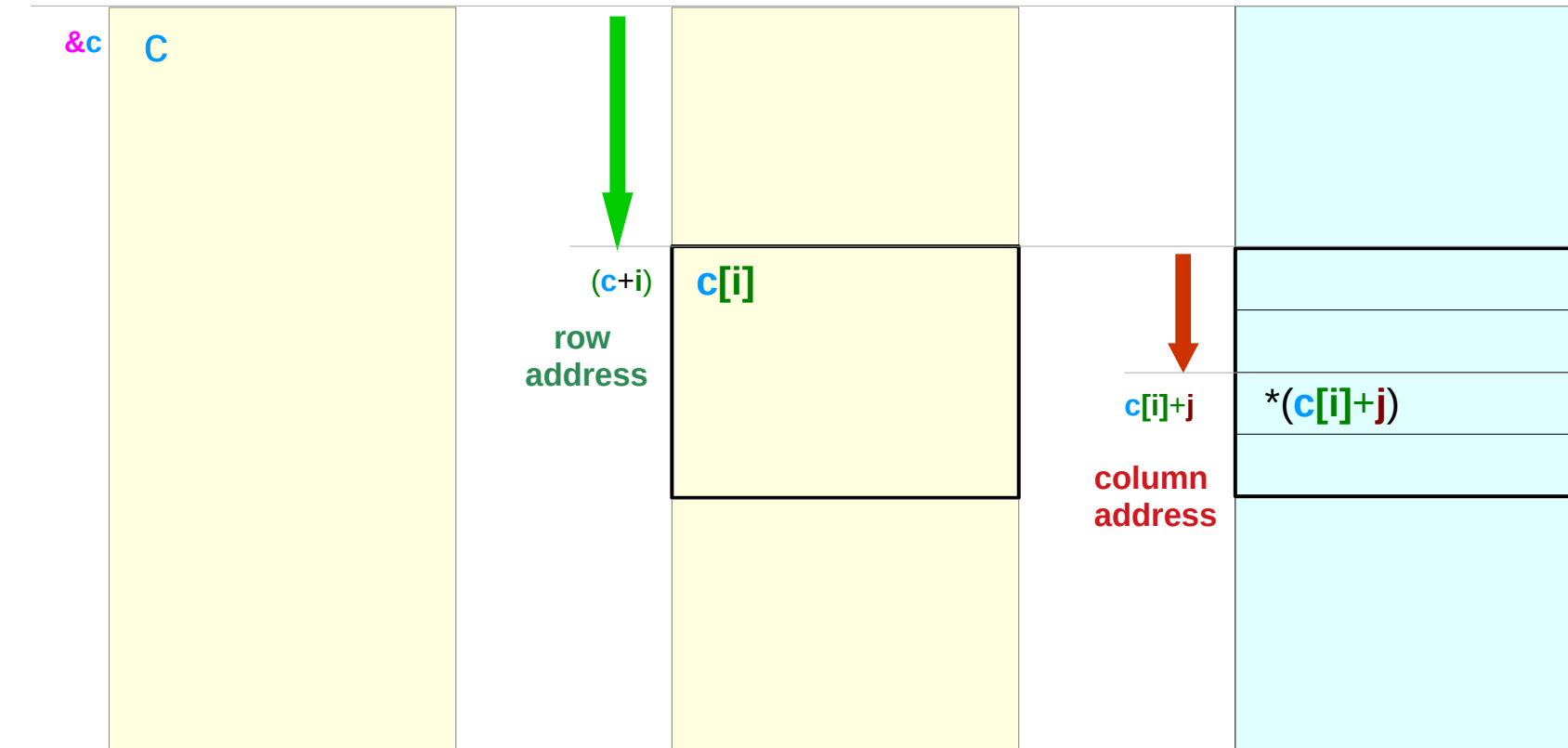


A 2-d Array – an index view

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

$$*(c+i) = c[i]$$

$$*(*(c+i)+j) = c[i][j]$$

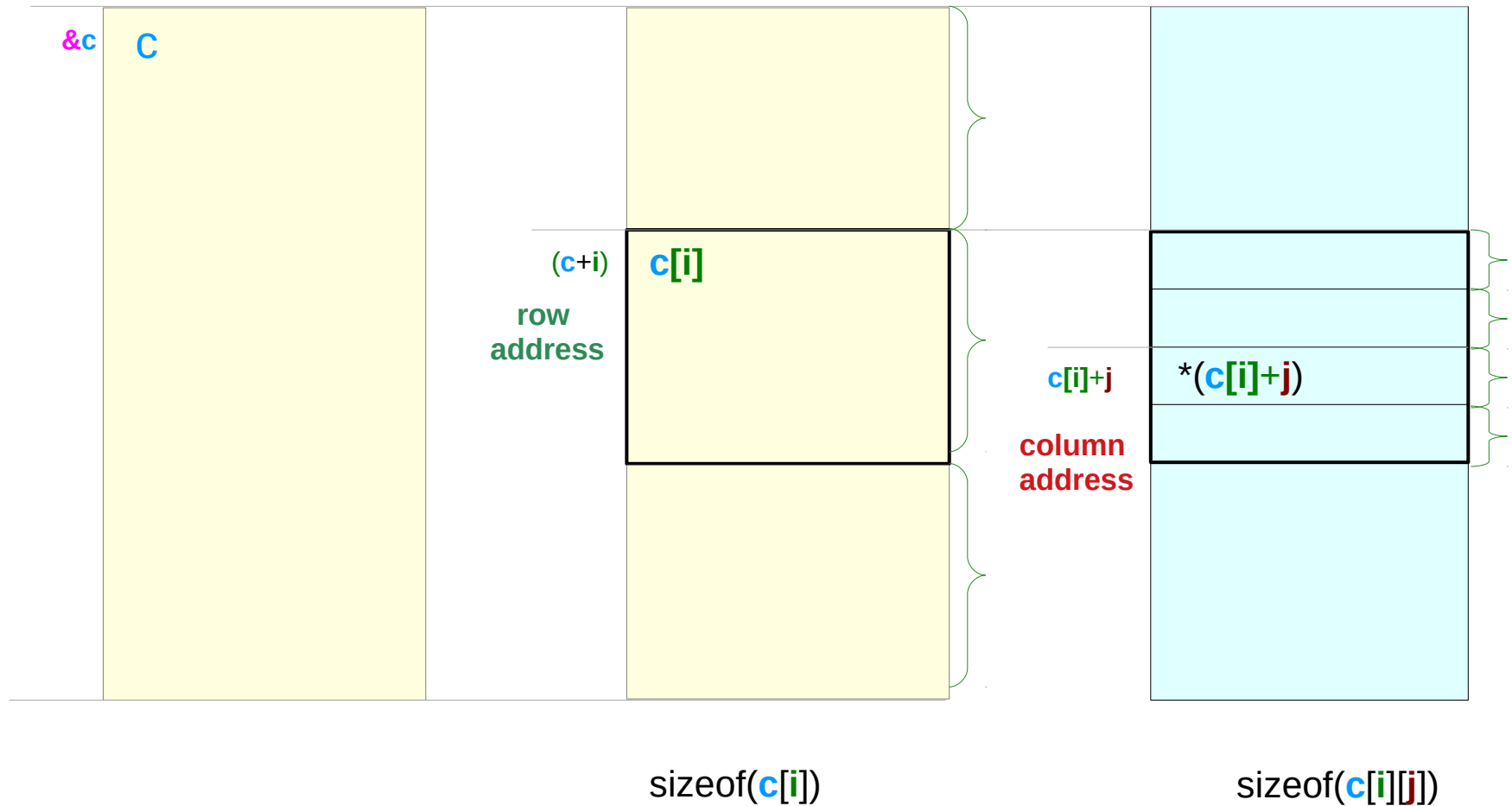


A 2-d Array – an index view

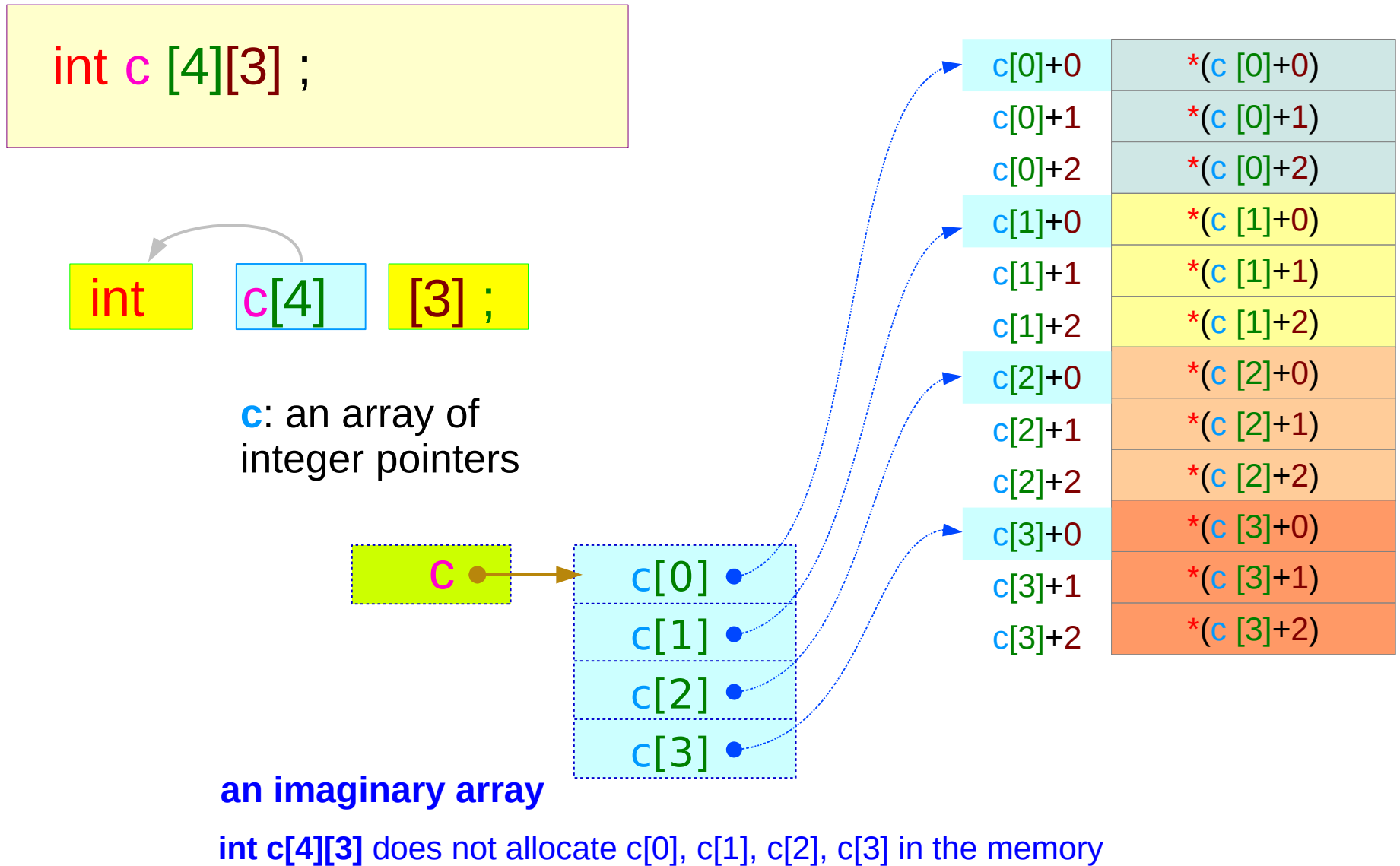
```
int c [3] [4];
```

$$*(c+i) = c[i]$$

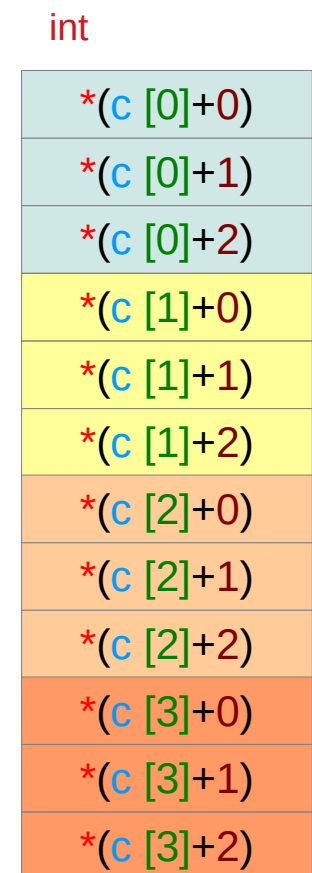
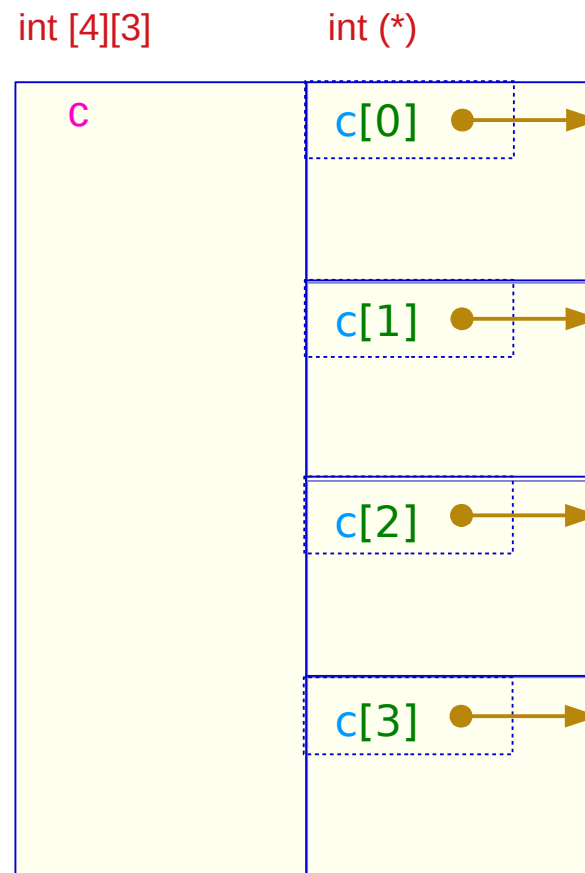
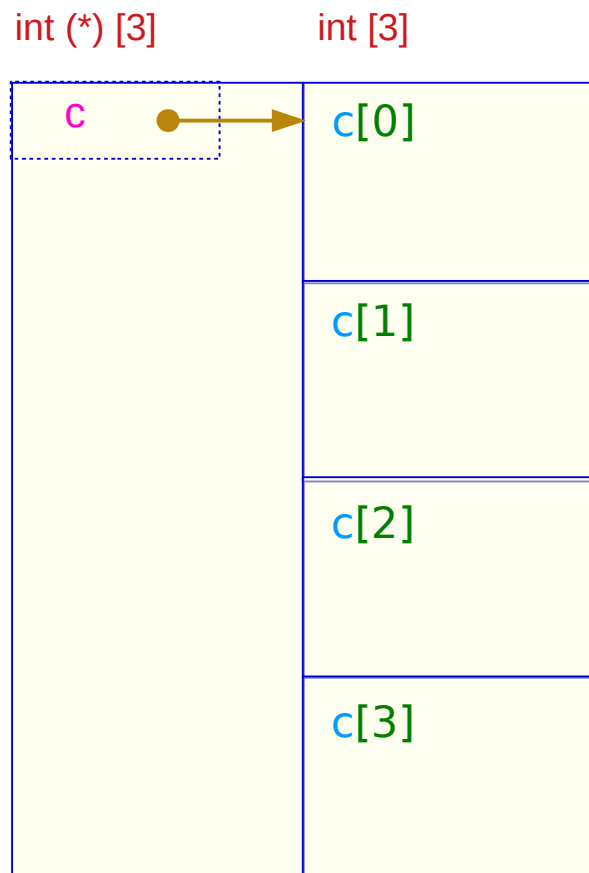
$$*(*(c+i)+j) = c[i][j]$$



Nested arrays



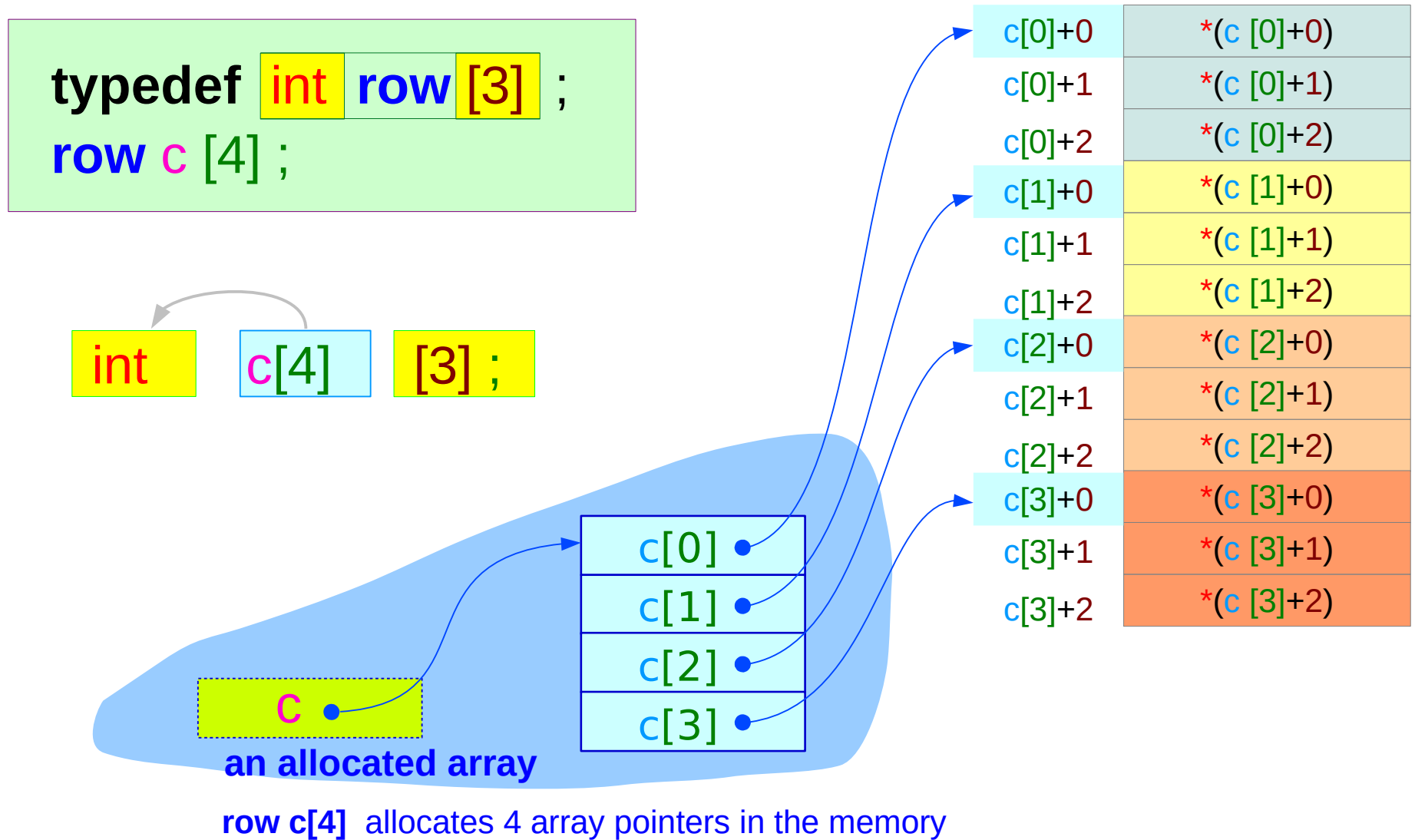
Nested arrays



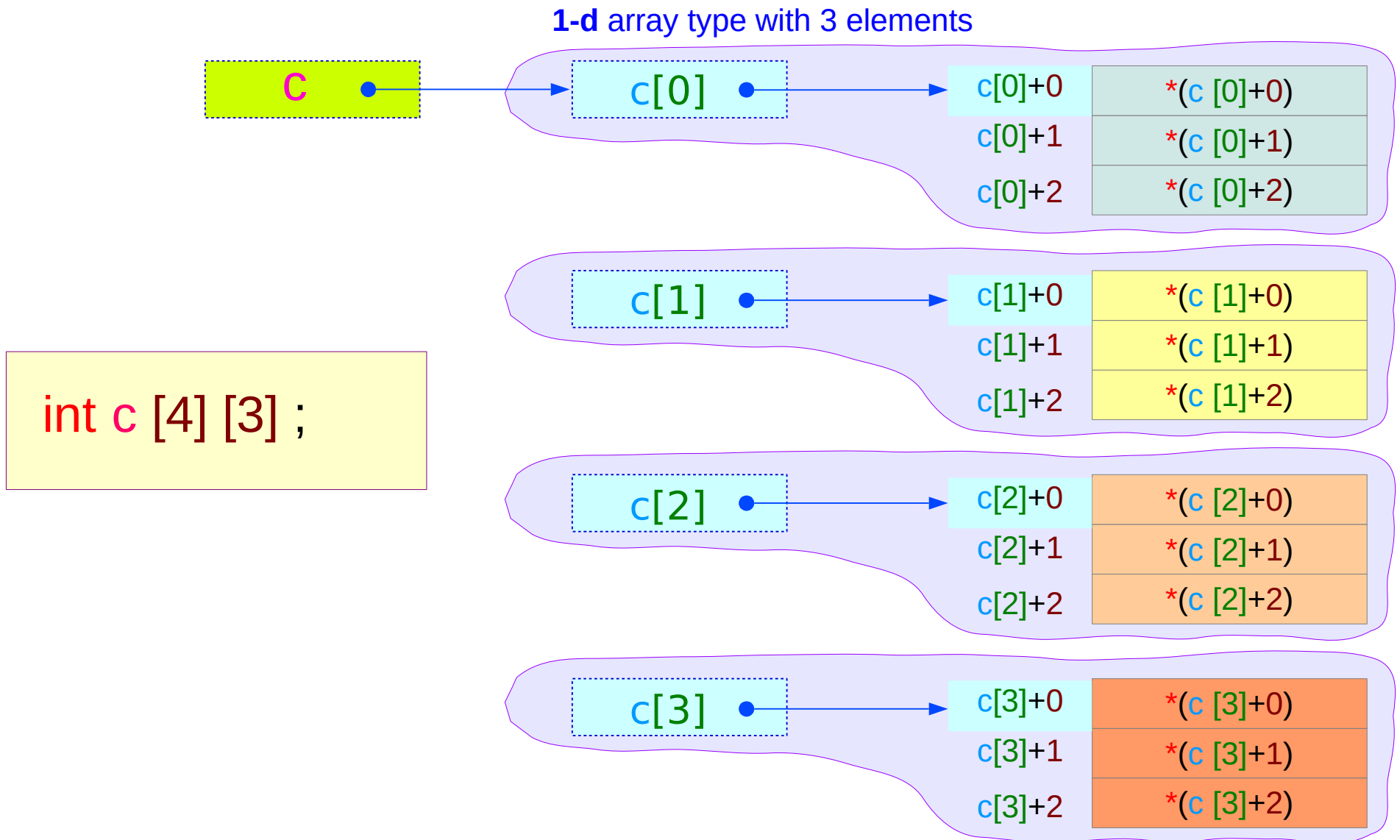
an imaginary array

`int c[4][3]` does not allocate `c[0]`, `c[1]`, `c[2]`, `c[3]` in the memory

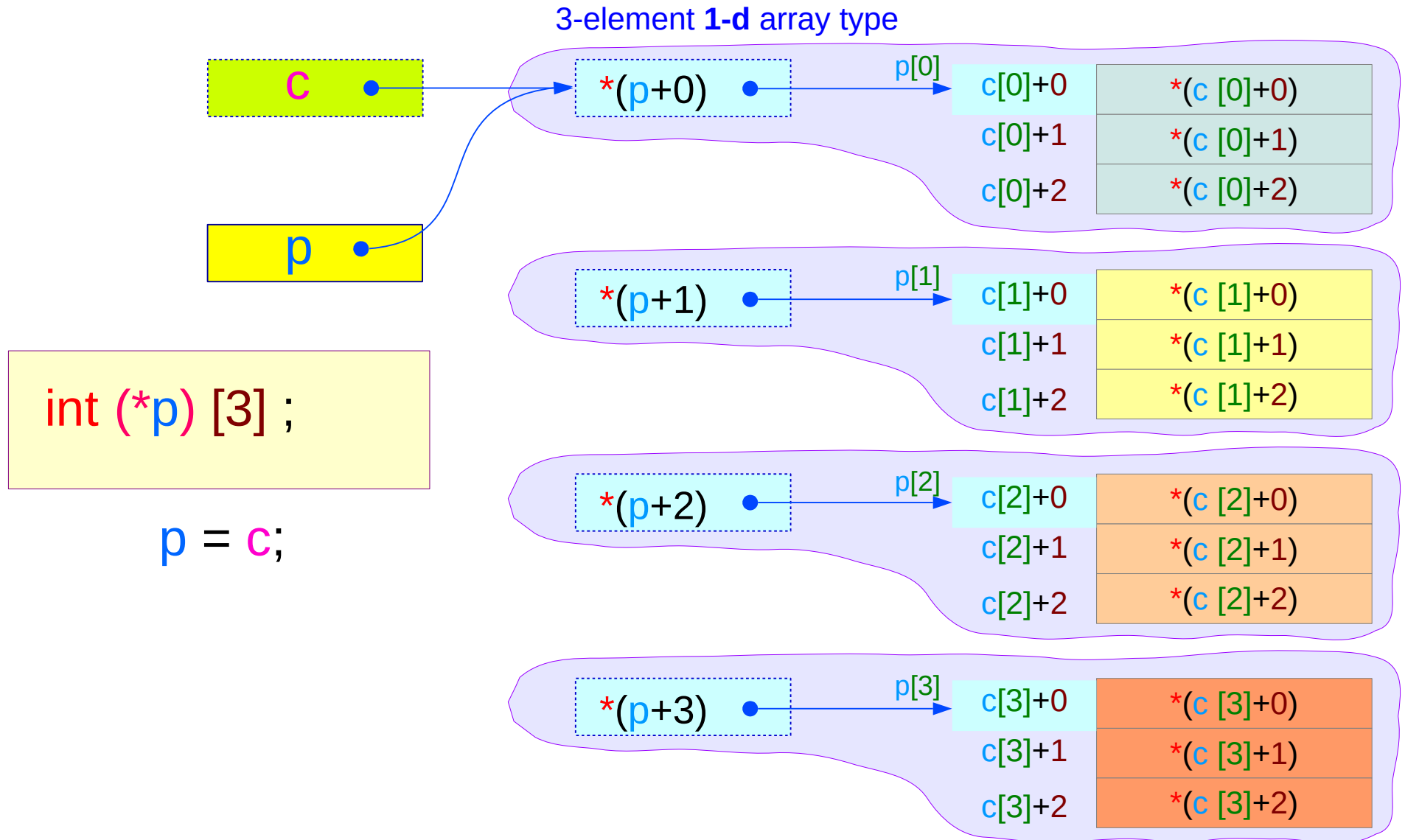
Nested array declared explicitly



2-d Array : rows of 1-d arrays



Pointer to 1-d arrays



Types of array names

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

`a` is the name of the 1-d array

`int [4]`

start address = `&a[0]`

`sizeof(a)` = $4 * 4$

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

`c[i]` is the name of the 1-d subarray

`int [4]`

start address = `&c[i][0]`

`sizeof(c[i])` = $4 * 4$

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

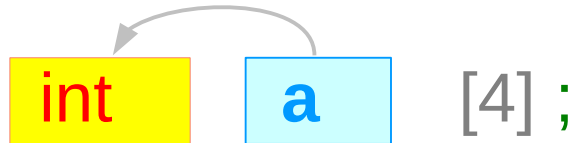
`c` is the name of the 2-d array

`int [3][4]`

start address = `&c[0][0]`

`sizeof(c[i])` = $3 * 4 * 4$

Values of array names

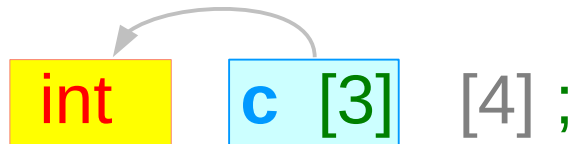


the value of **a** is the starting address of a 4 element array of **int** type

int (*)

a: pointer to the first element

&a[0] = a



each value of **c[i]** is the starting address of a 4 element array of **int** type

int (*)

c[i]: pointer to the first element

&c[i][0] = c[i]



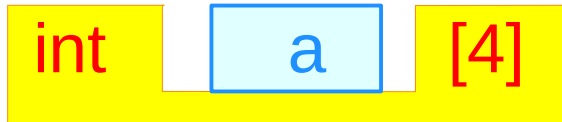
the value of **c** is the starting address of a 3 element array of **int [4]** type

int (*) [4]

c: pointer to the first element

&c[0] = c

Array and pointer types in a 1-d array



a 1-d array

type : int [4]

size : 4 * 4



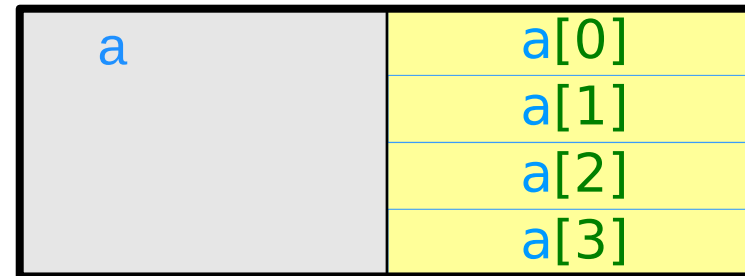
a 0-d array pointer (virtual)

type : int (*)

size : 4 * 4

a points to the 1st int element
there are 4 int elements

int [4]

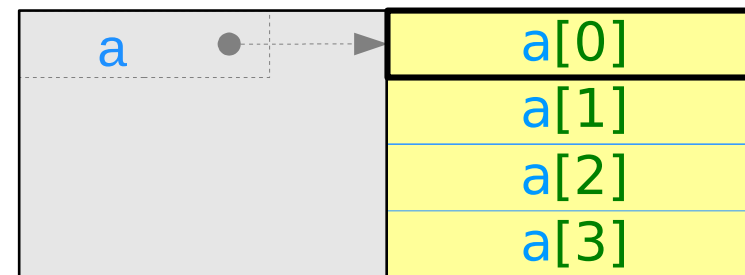


no physical
memory locations

real consecutive
memory locations

int (*)

int



no physical
memory locations

real consecutive
memory locations

2-d array type



C 2-d array

type : int [3][4]

size : 3 * 4 * 4

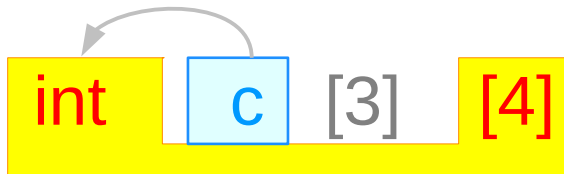
int [3][4]

c	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]

no physical memory locations

real consecutive
memory locations

1-d array pointer type

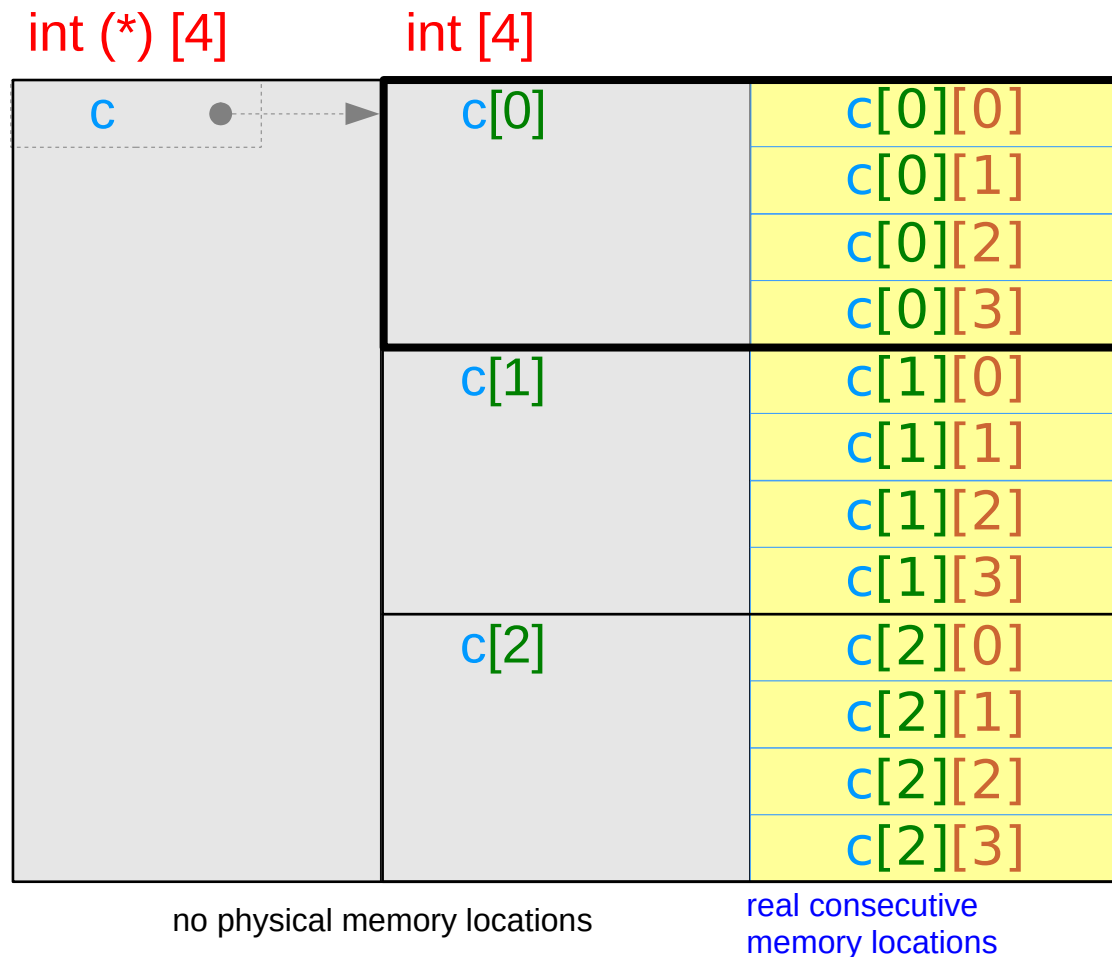


C 1-d array pointer (virtual)

type : `int (*) [4]`

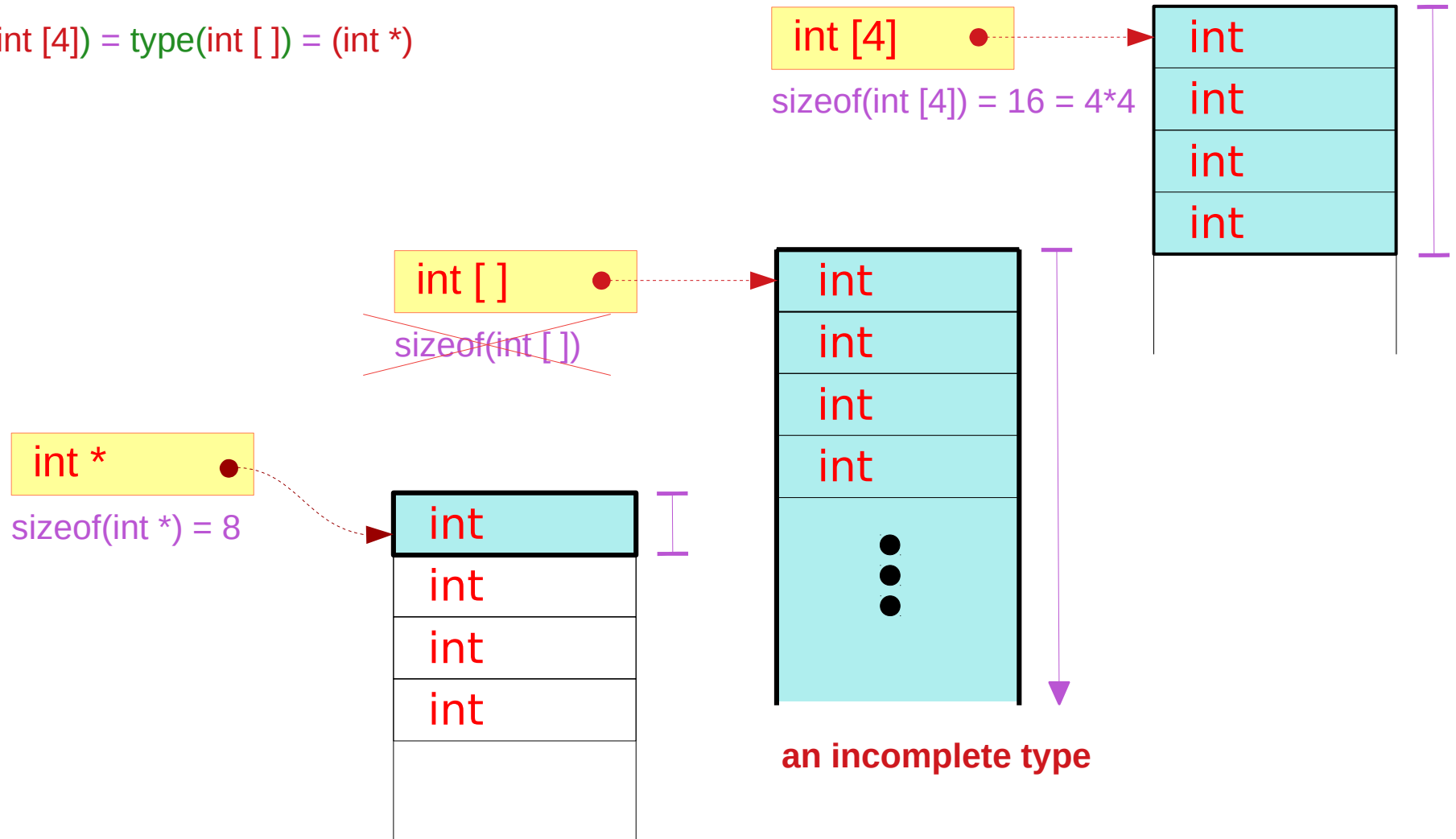
size : `3 * 4 * 4`

`c` points to the 1st `int [4]` element
There are 3 `int [4]` elements



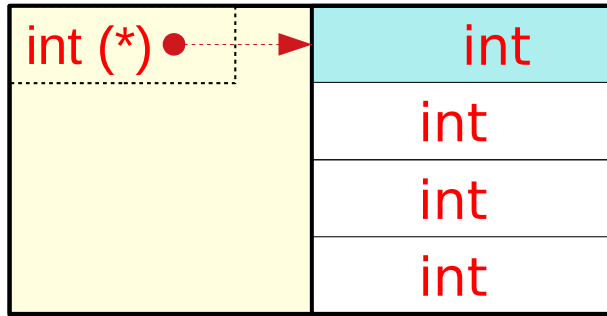
Differences in pointer types – `int [4]`, `int []`, `int *`

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

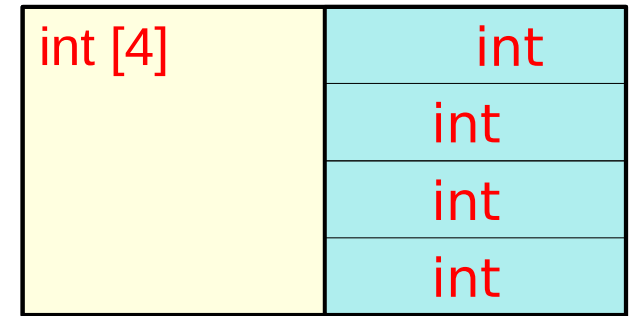


Differences in pointer types – `int [4]`, `int []`, `int *`

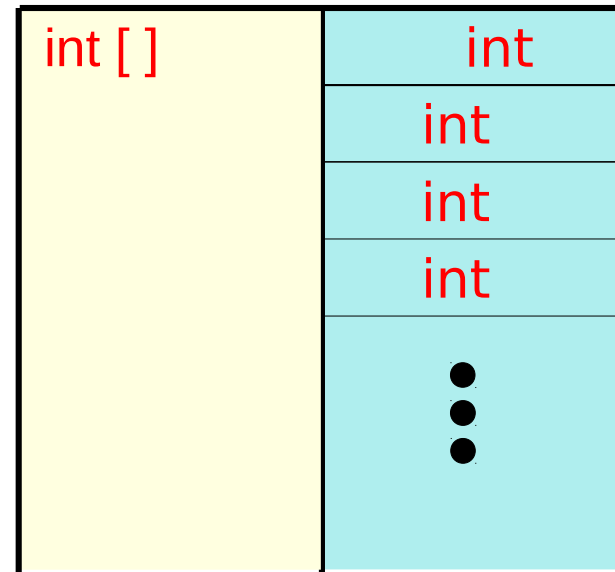
`sizeof(int (*)) = 16 = 4*4`



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



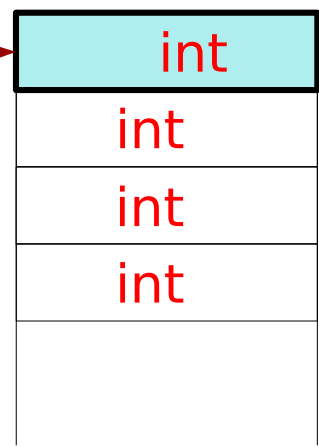
~~`sizeof(int [])`~~



an incomplete type

`int *`

`sizeof(int *) = 8`



Types of multi-dimension array names

```
int a ;
```

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

```
int c [4][5];
```

```
int d [4][5][6];
```

```
a :: int
```

→ int

```
b :: int [4]
```

→ int (*)

int *

```
c :: int [4][5]
```

→ int (*)[5]

```
d :: int [4][5][6]
```

→ int (*)[5][6]

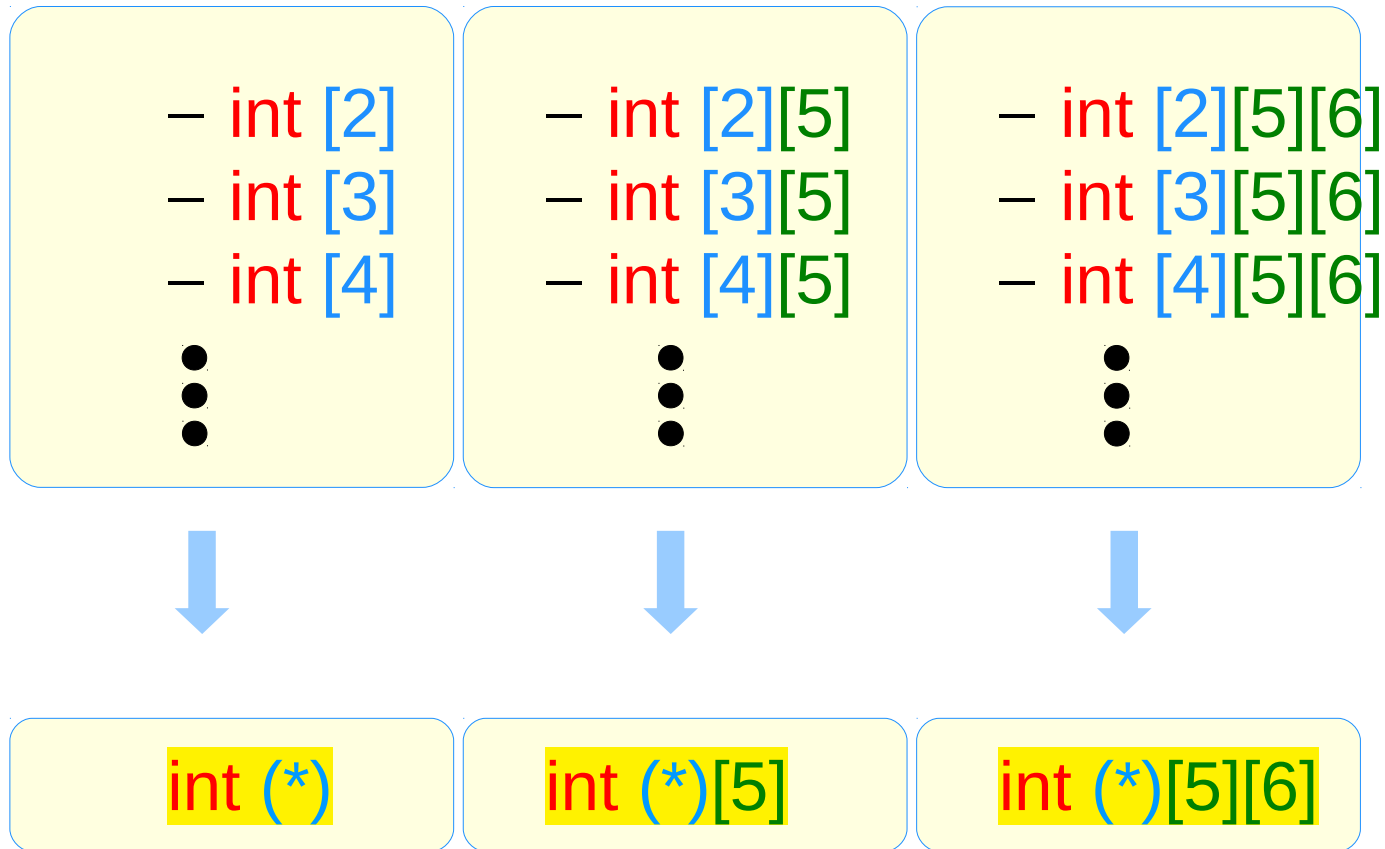
array types

specific types

array pointer types

general type

Array pointers v.s. Array



specific types

Relaxing the 1st dimension

general type

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>