Arrays (1A)

Young Won Lim
8/9/22

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

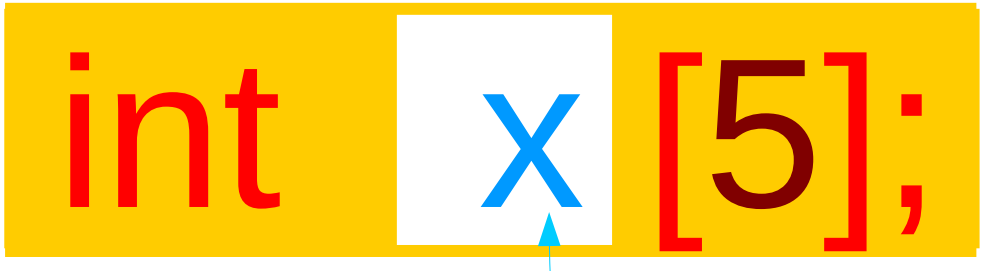
$$m = \frac{1}{N}\sum_{i=0}^{N-1} x_i$$

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

5 variables

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

Definition of an Array



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 ]
```

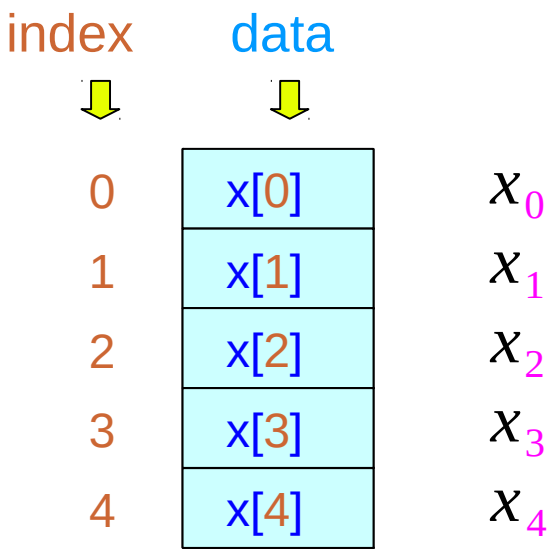
i = 0, … , 4

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

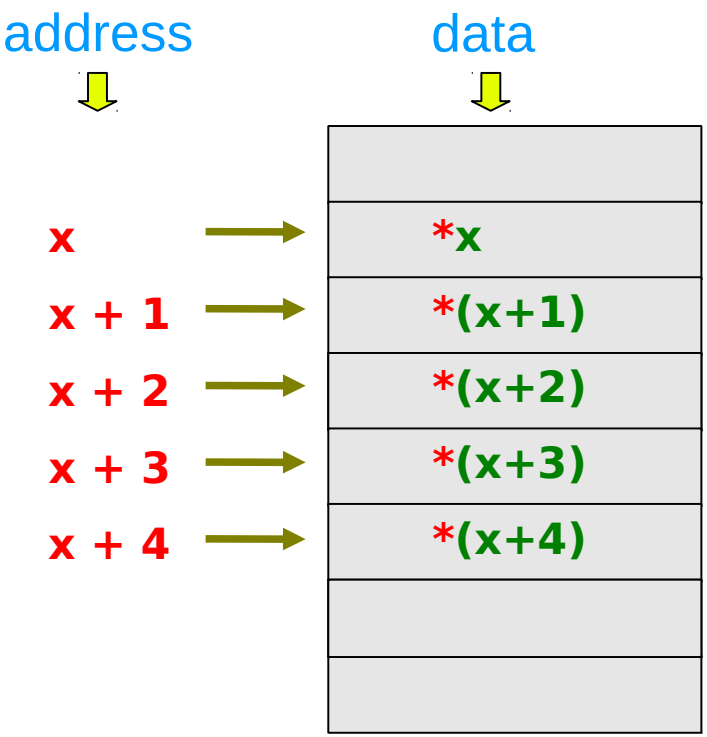
```
int     x[5];
```

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

5 int variables

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

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)**

i : an index variable [0..4]
x[i] : the (i+1)th element value

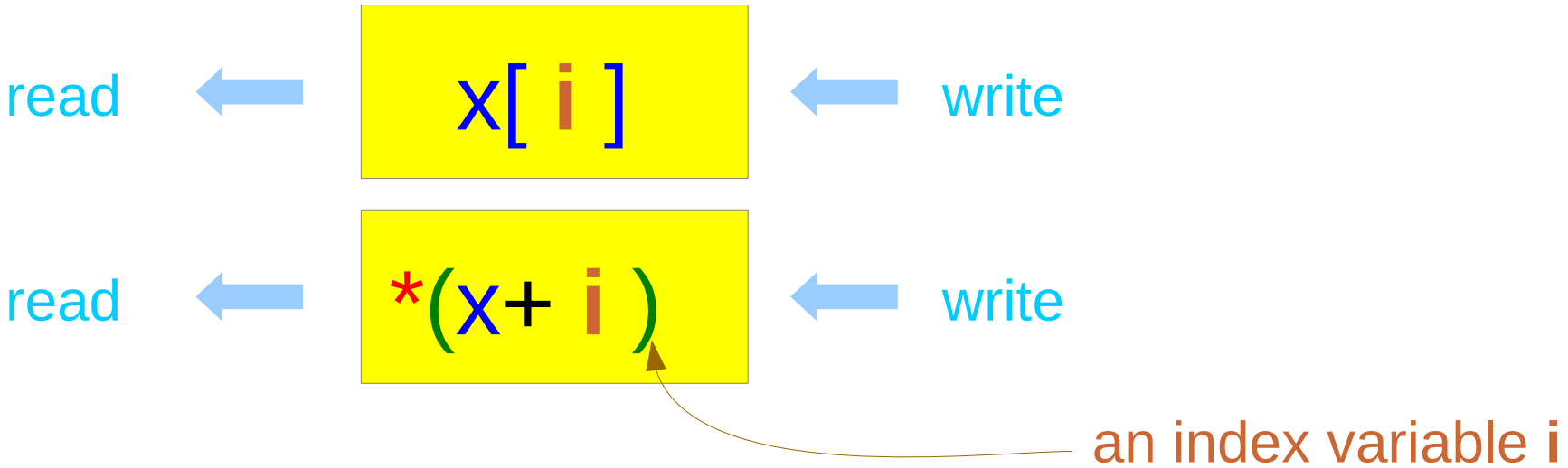
x : the starting address
x+i : the (i+1)th element's address
***(x+i)** : the (i+1)th element value

A variable expressed by another variable

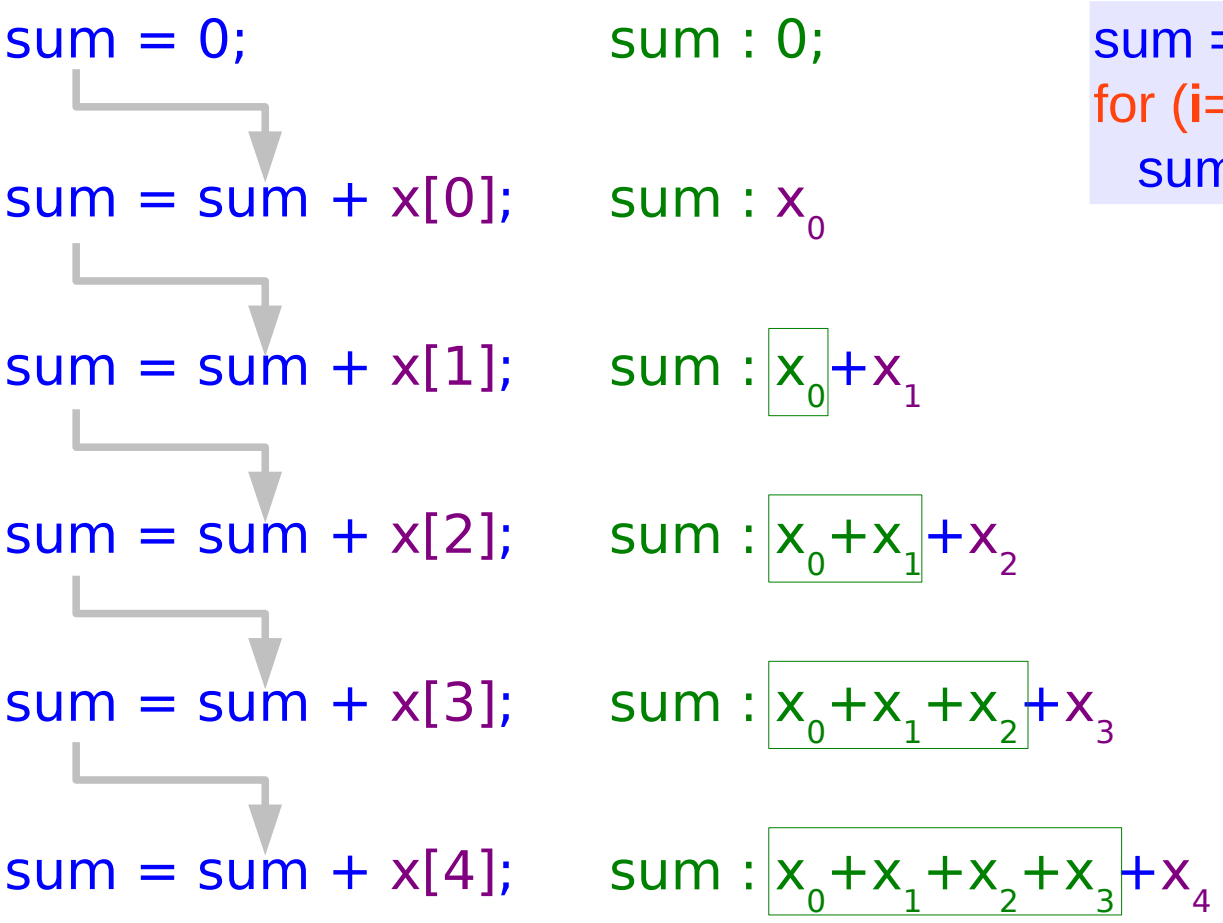
```
int     x[5];
```

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

treated as a variable

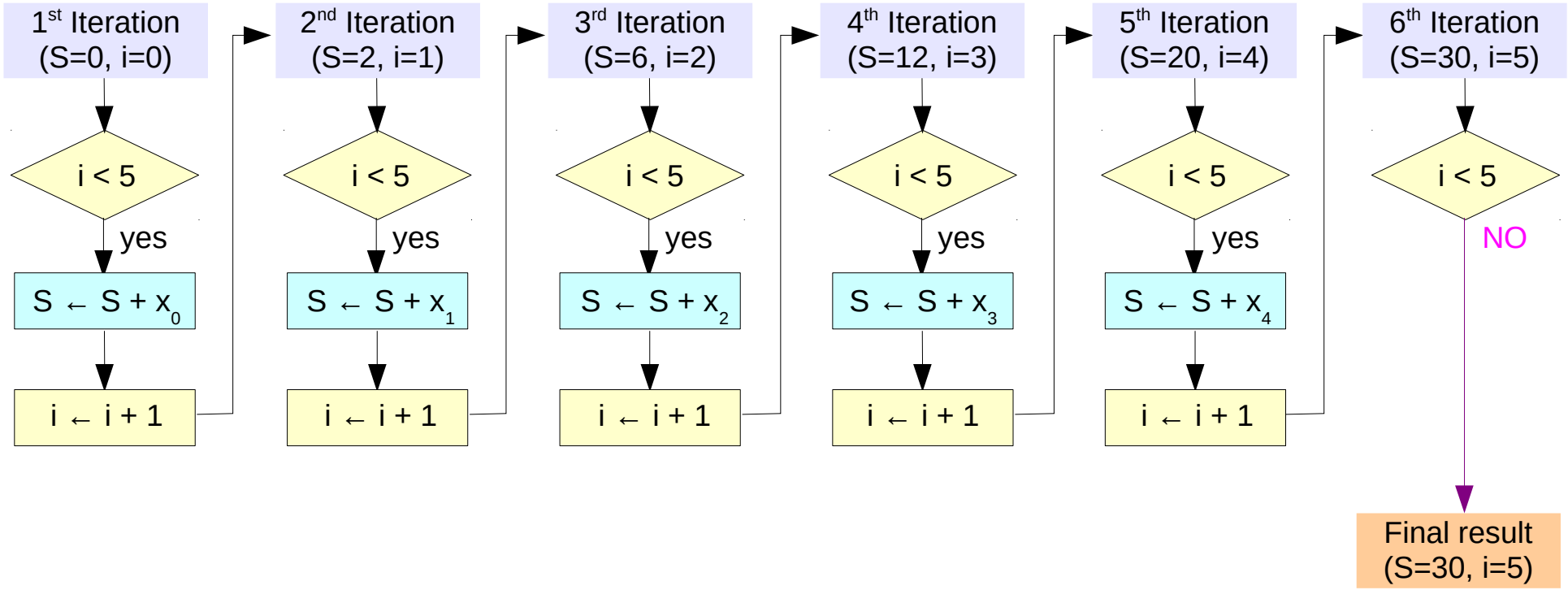


Computing the sum of n numbers (1)




```
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 [ ] ;`~~

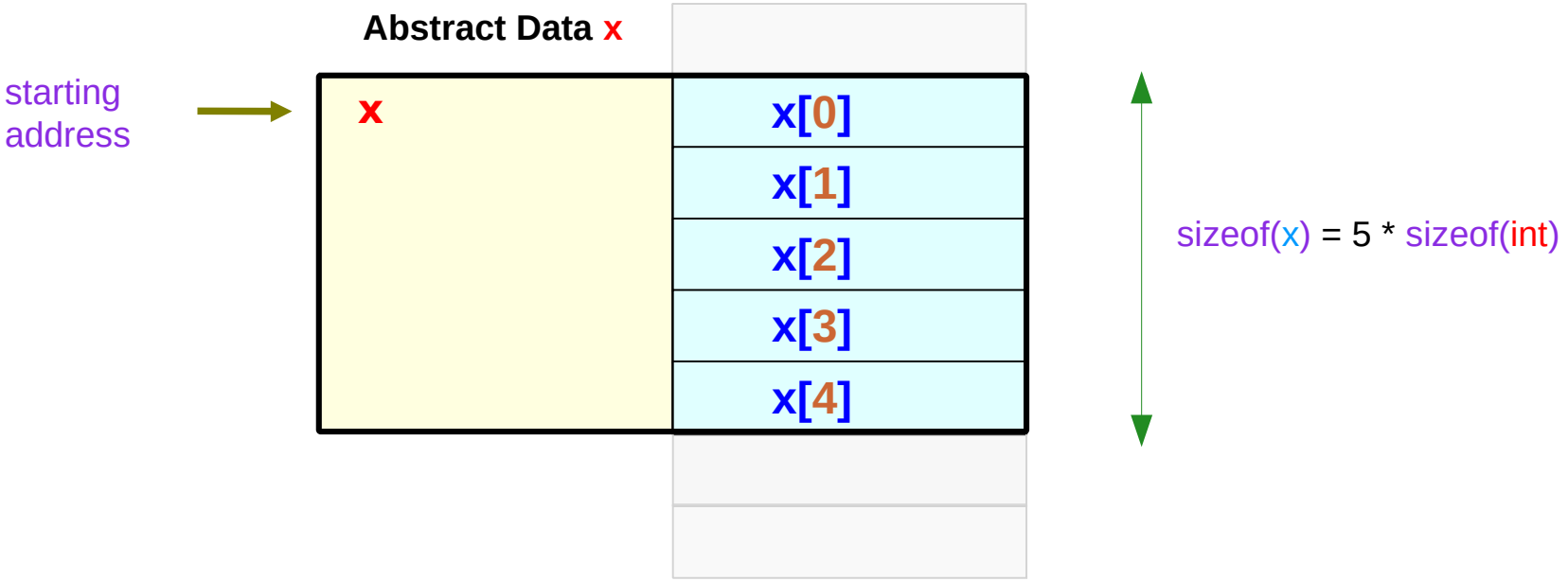
```
int c [3][4] = { { 1, 2, 3, 4},
                 { 5, 6, 7, 8},
                 {9,10,11,12} };
```

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

Abstract data **x**

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

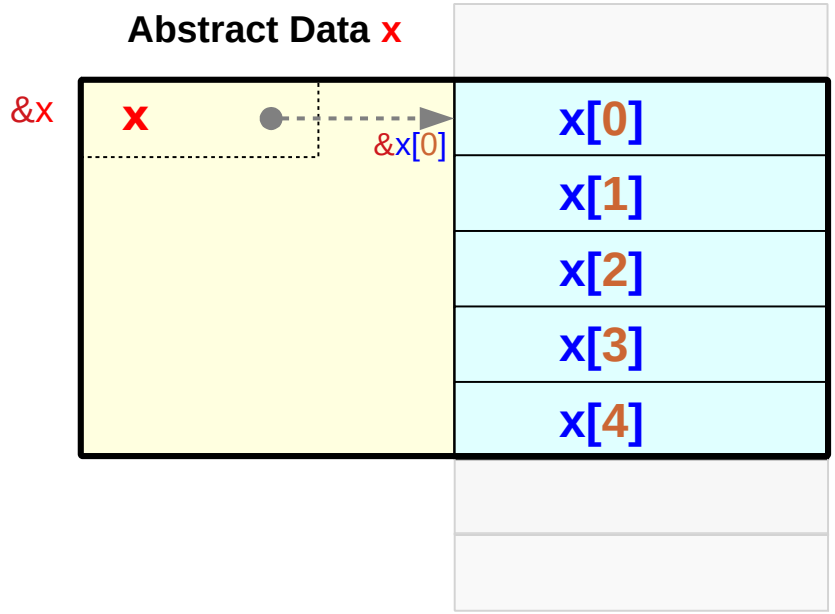
x holds the *starting address*
*of **5** consecutive **int** variables*



Abstract data **x** as a pointer

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

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



pointer relation

x ≡ &x[0]

*x ≡ x[0]

value(&x) = value(x) = value(&x[0])

the starting address of 5 consecutive int variables

not a real pointer x

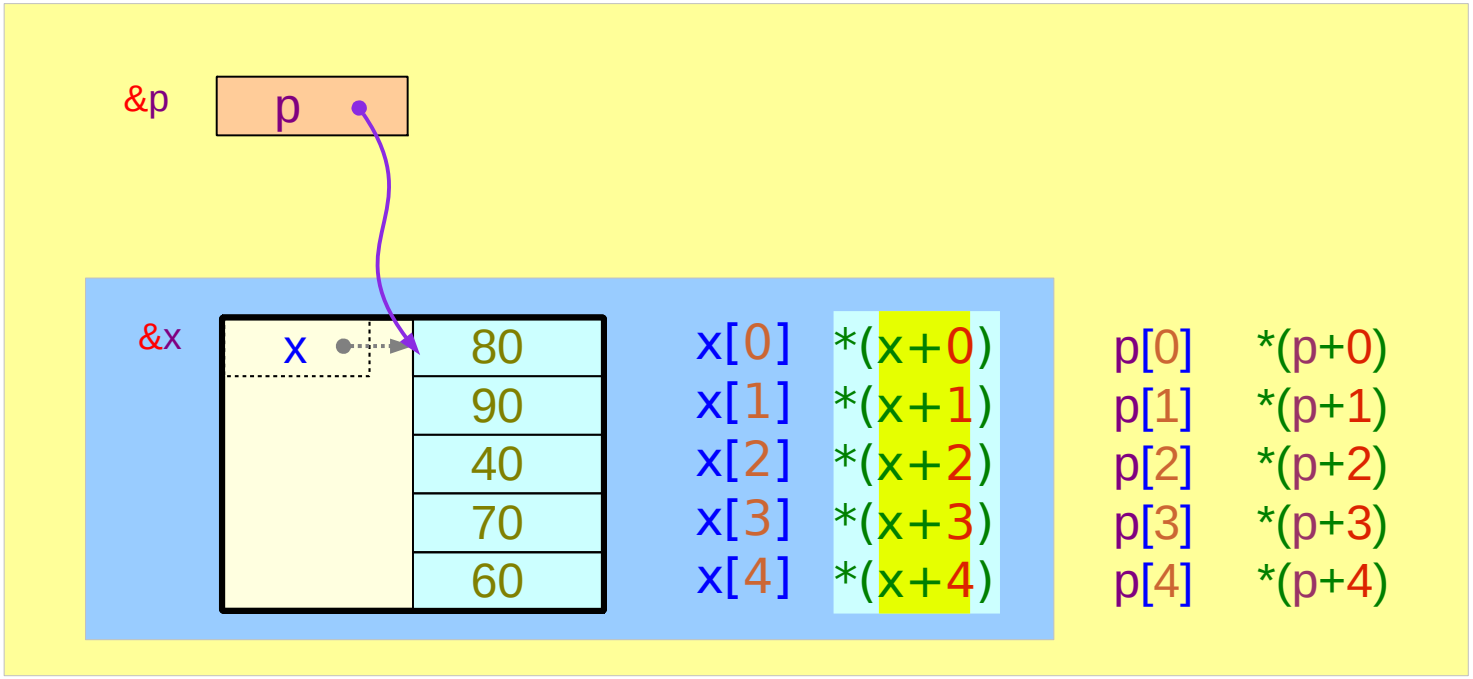
Accessing an Array with a Pointer Variable

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

int  *p = x;
```

x is a constant symbol
cannot be changed

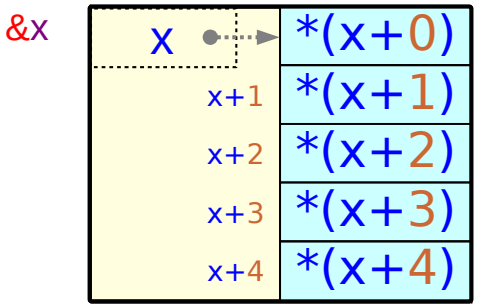
p is a variable
can point to other addresses



An Array Name and a Pointer Variable

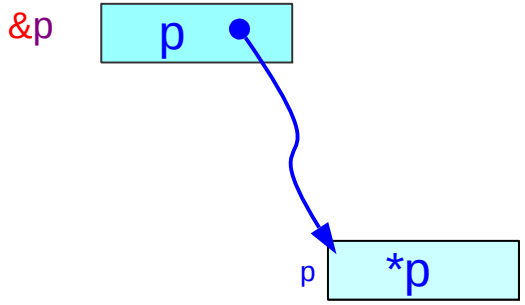
```
int x [5] ;
```

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




```
int  * p ;
```

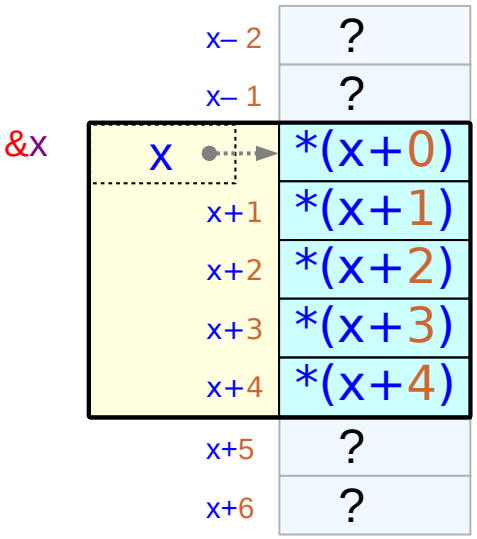
p: an variable name
Value: the address
of an int variable

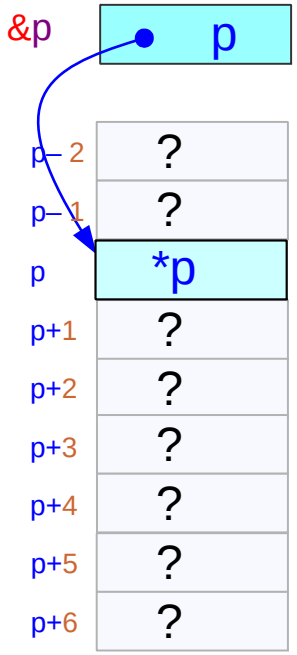


Out of range index

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

```
int  * p ;
```

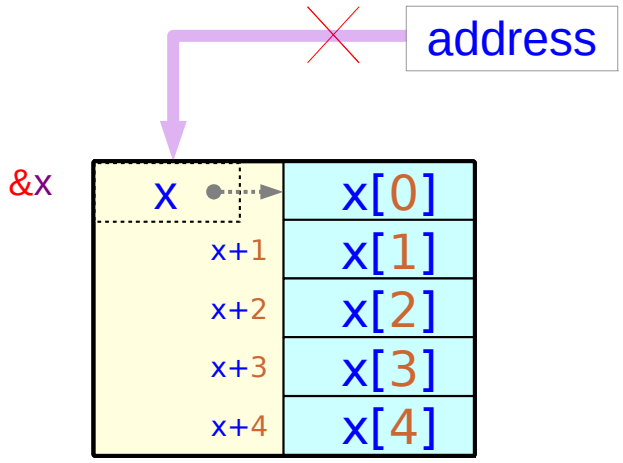




A programmer's responsibility

Assignment of an address

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

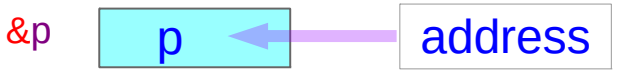


x is not a variable
but a constant symbol

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 ;
```



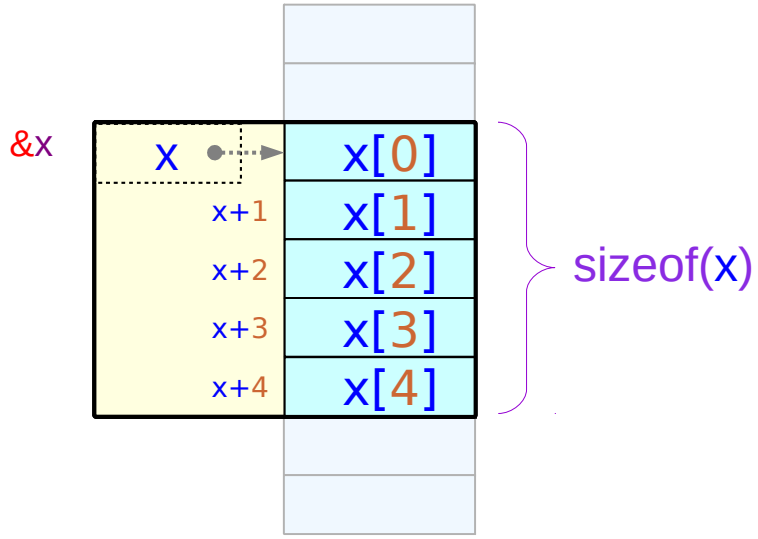
p is a variable

has an allocated memory location

its value can be changed

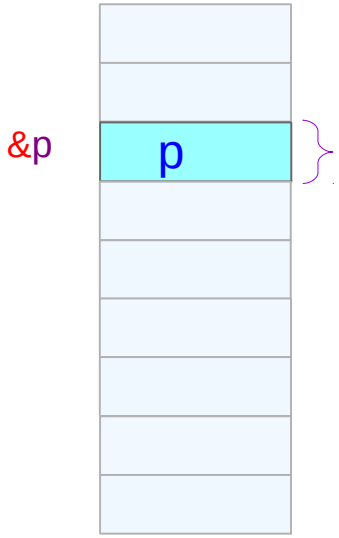
Size

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



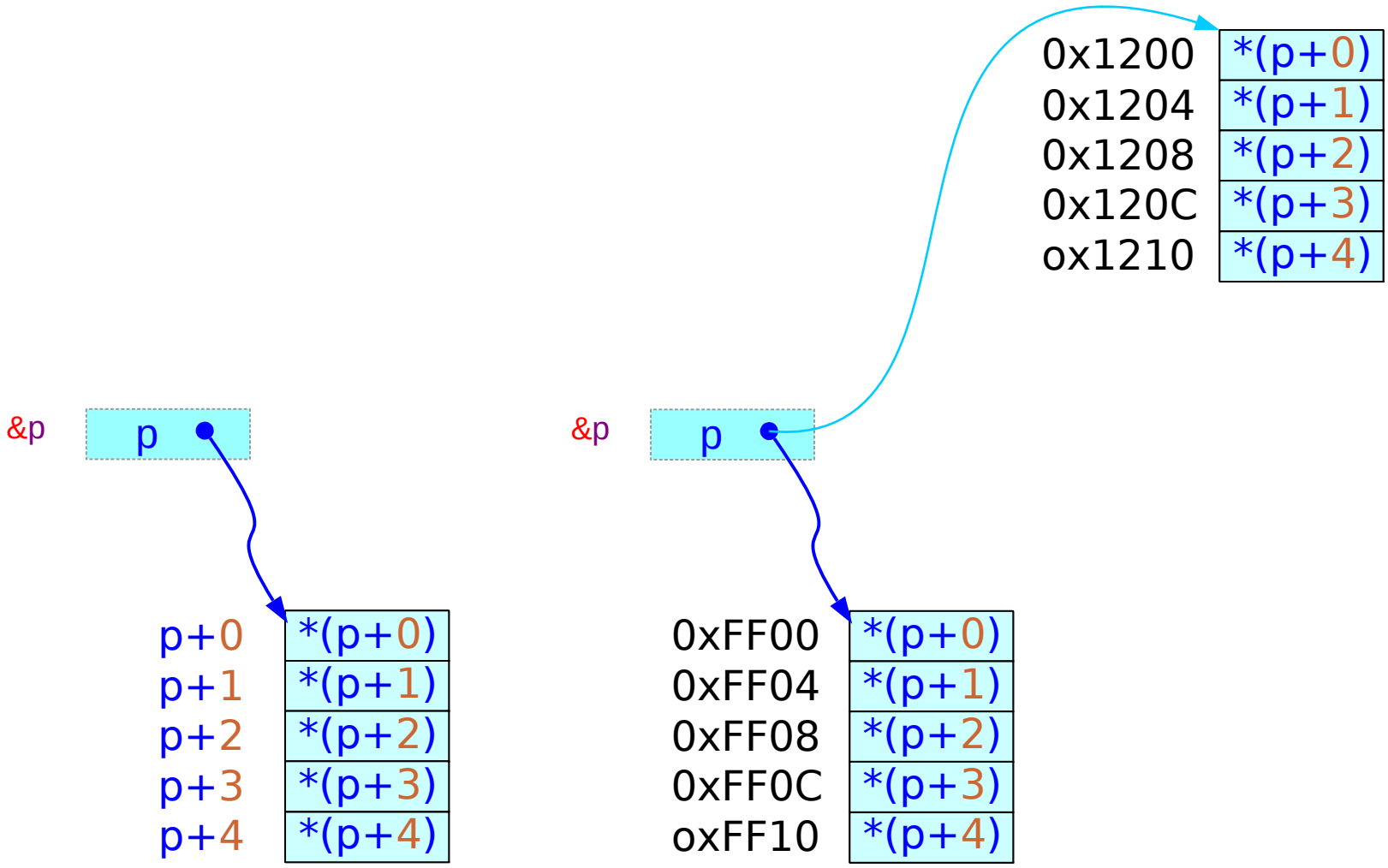
sizeof(x) = 5 * sizeof(int)

```
int  * p ;
```



sizeof(p) = size of a pointer
= 4 / 8 bytes

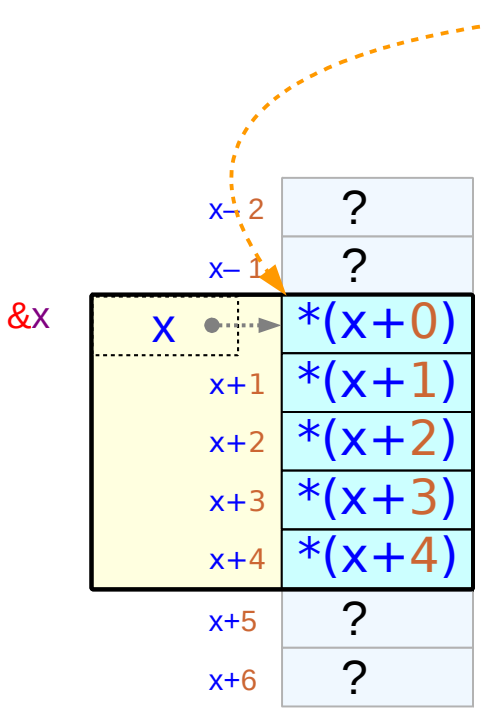
Pointer variable can point different locations

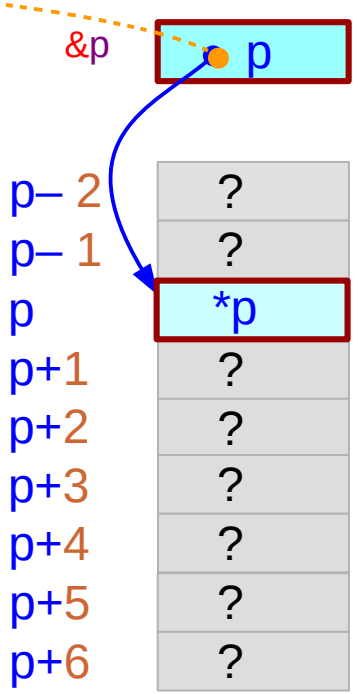


Pointer to an integer

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

```
int  * p ;
```





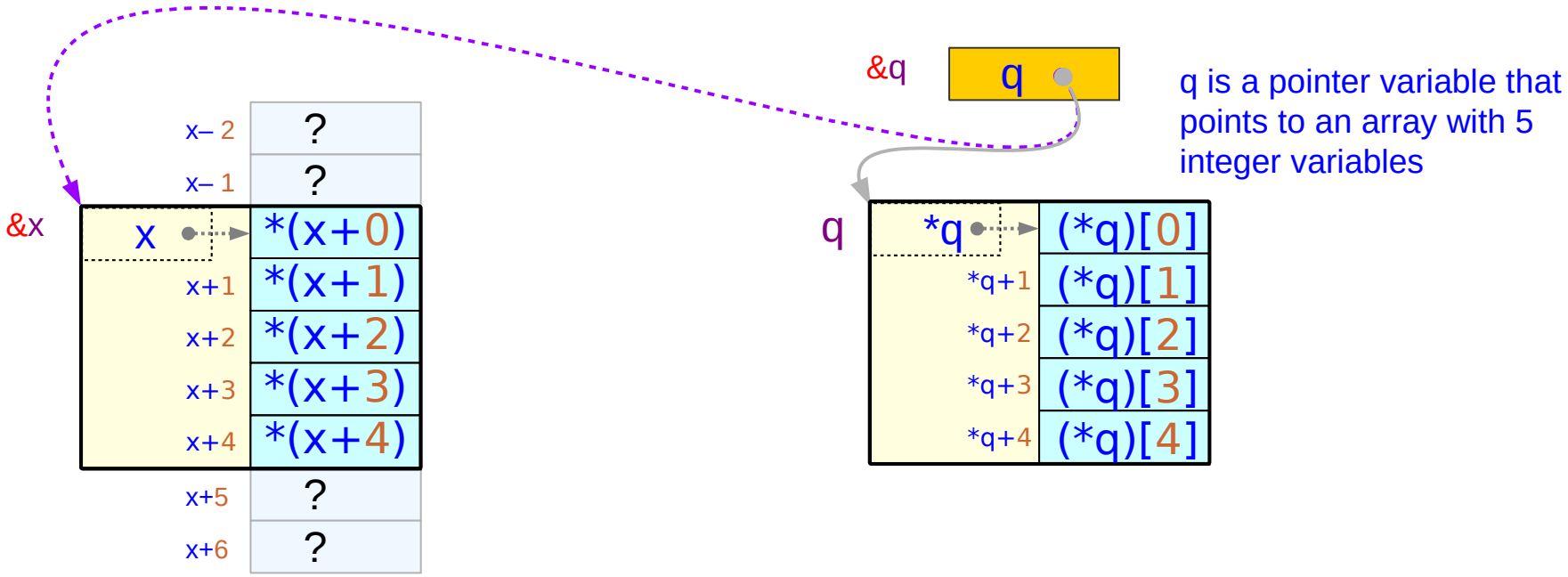

```
*p = *x ;
```

```
p = x ;
```

Pointer to an array

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

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



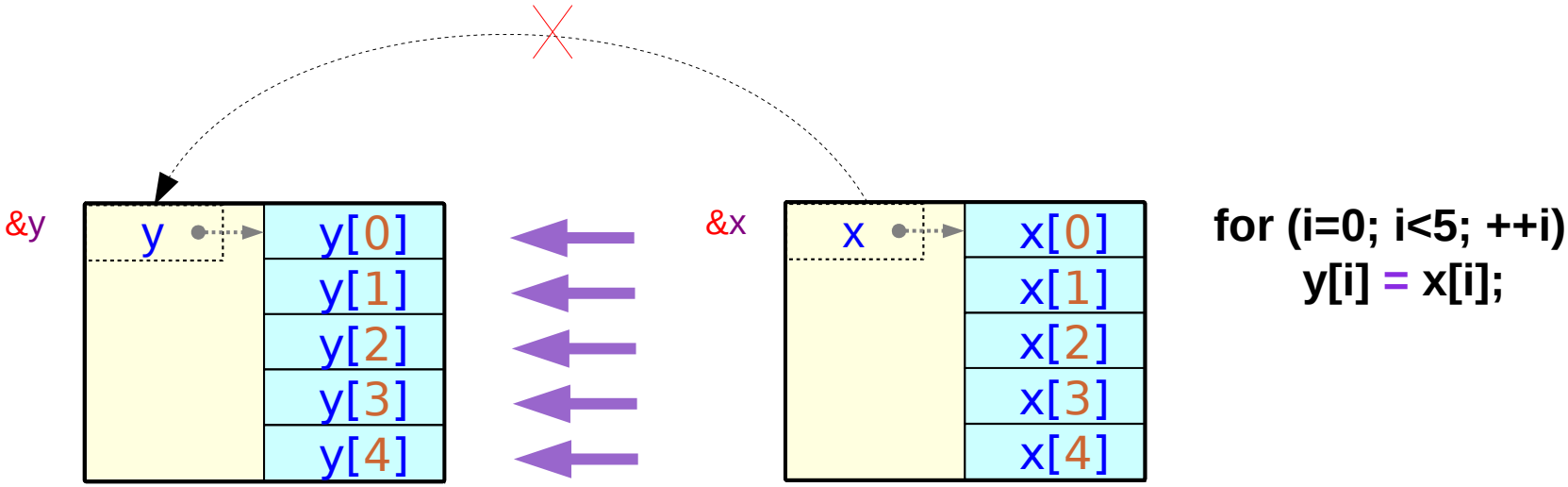

```
*q = x ;
```

```
q = &x ;
```

* not frequently used feature

Copying an Array to another Array

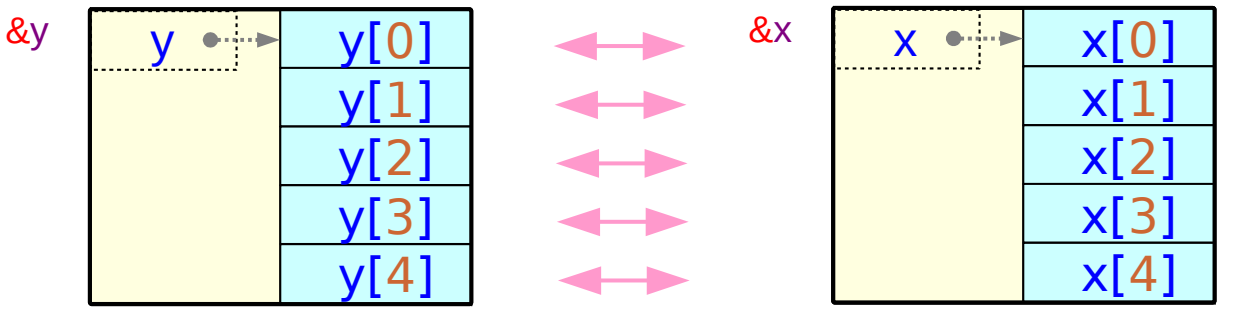
```
int x [5] = { 1, 2, 3, 4, 5 };
int y [5] ;
y = x;
```




```
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
```

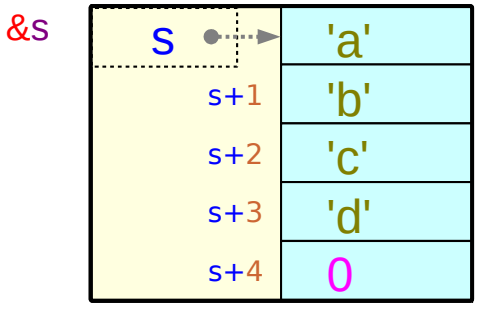



```
EQ=1;
for (i=0; i<5; ++i)
      EQ &= (y[i] == x[i]);
```

Initialized Character Arrays and Pointers (1)

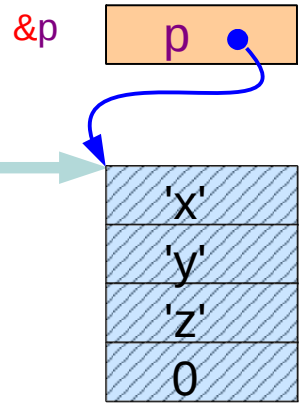
```
char s [5] = { 'a', 'b', 'c', 'd', 0 };
char s [5] = "abcd";

char  *p = "xyz";
```



a compiler determined constant address

a **constant** character string (array)



can change the value of any element

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

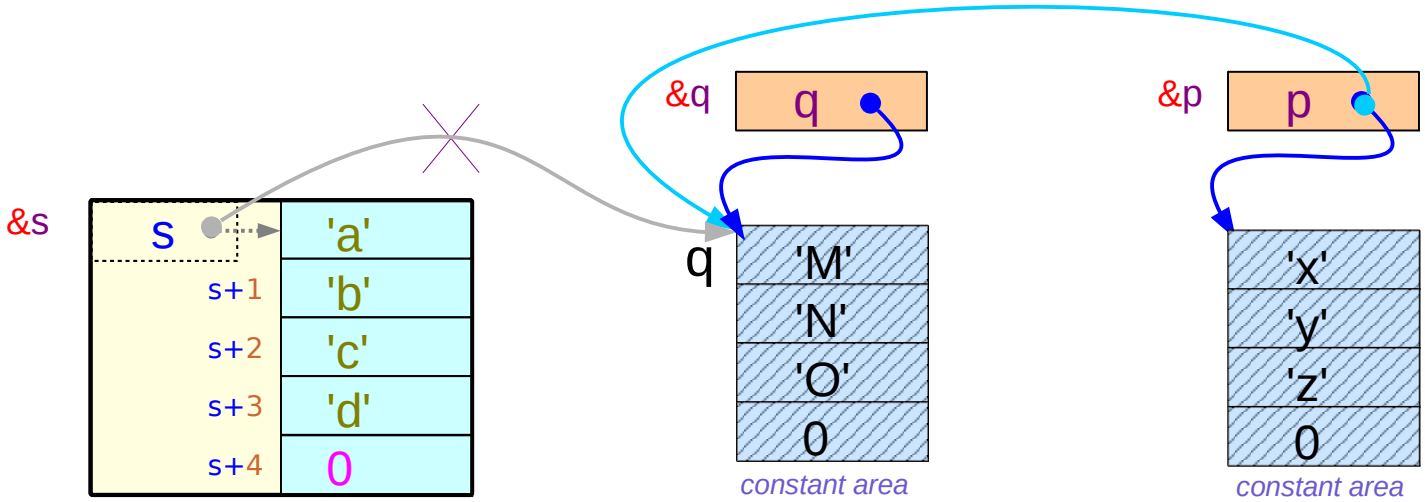
cannot change the value of any element of a ***constant*** array

```
*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" ;
```

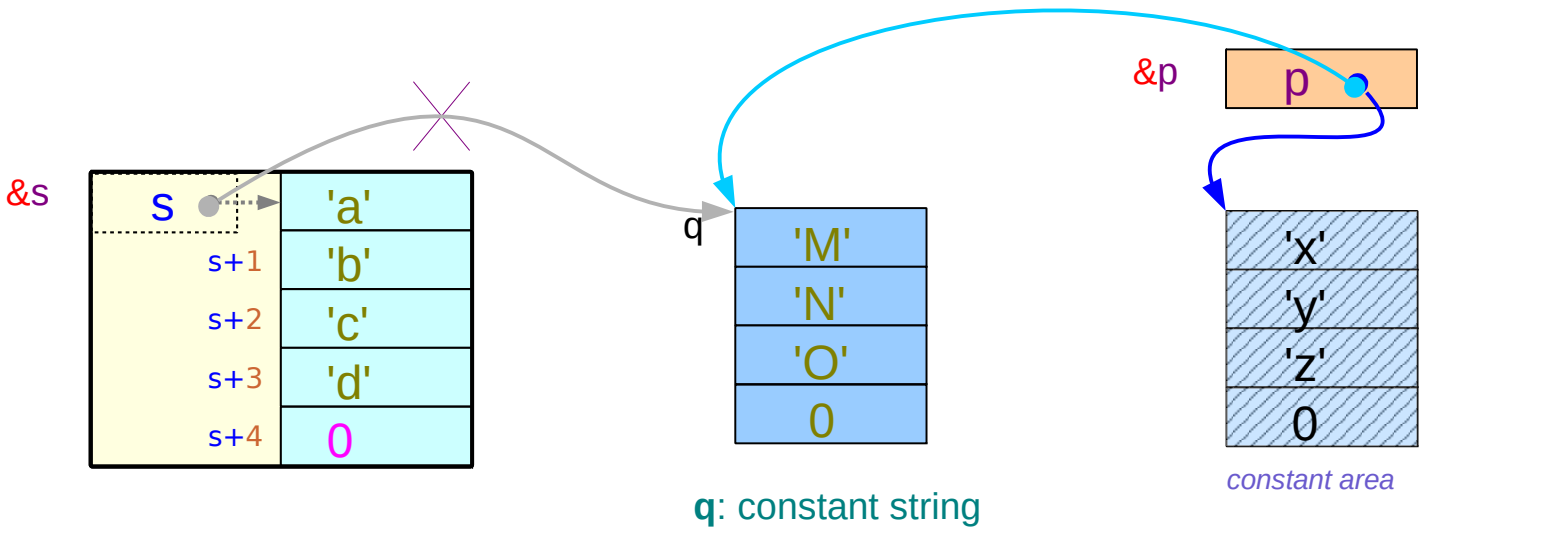


s cannot be assigned ~~s = q;~~

p can be assigned p = q;

Assigning a constant character string

```
char s [5] = { 'a', 'b', 'c', 'd', 0 };
char s [5] = “abcd”;
char  *p = “xyz”;
```

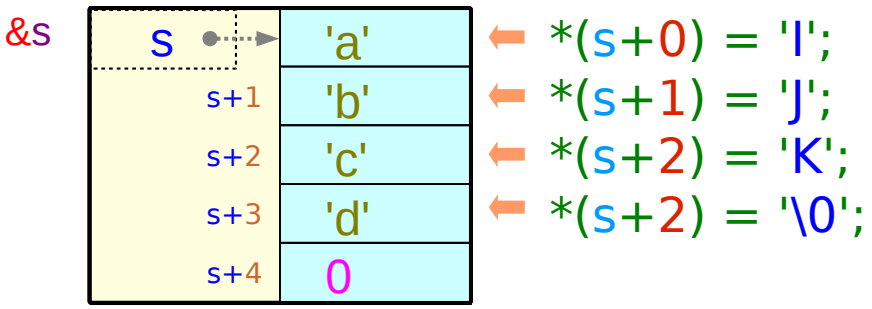


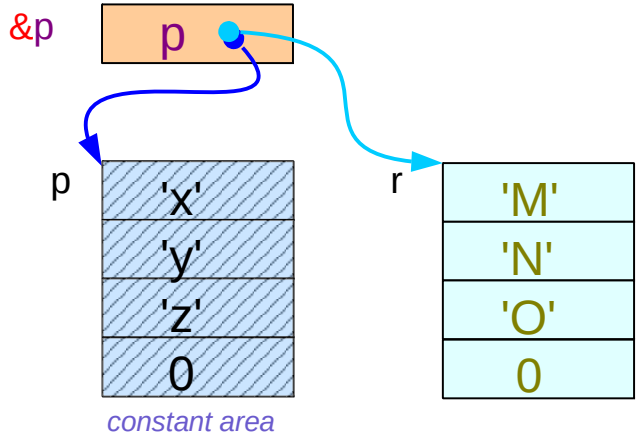
s = “MNO”

p = “MNO”

Copying a string

```
char s [5] = { 'a', 'b', 'c', 'd', 0 };
char s [5] = “abcd”;
char  *p = “xyz”;
```





strcpy (**s**, “IJK”);

p: constant string

strcpy (**p**, “IJK”);

r: non-constant string

strcpy (**r**, “IJK”);

Uninitialized Character Arrays and Pointers

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

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

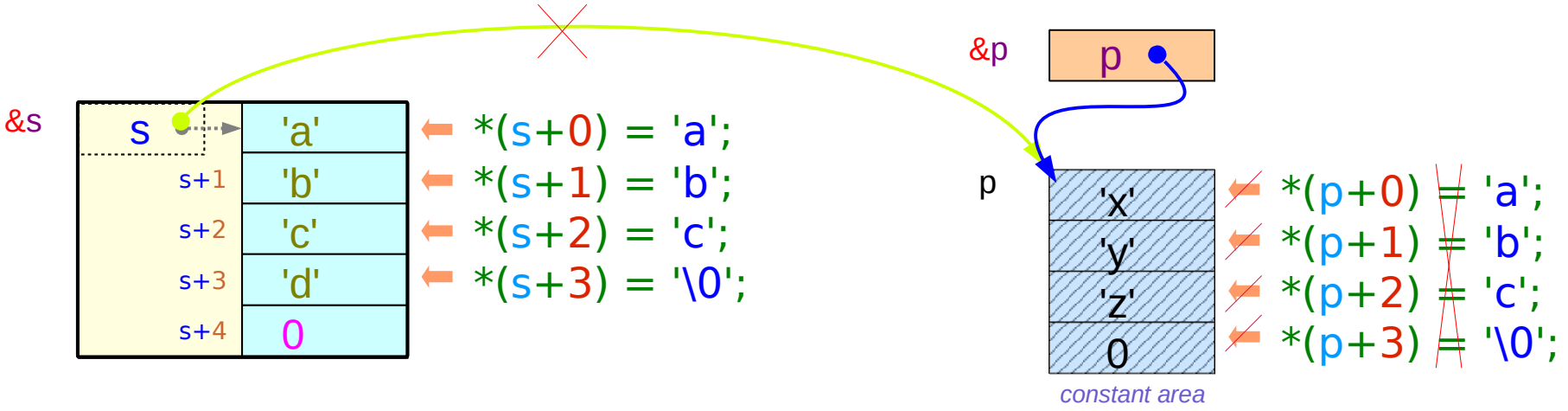
char * const **s**

const char * **p**

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

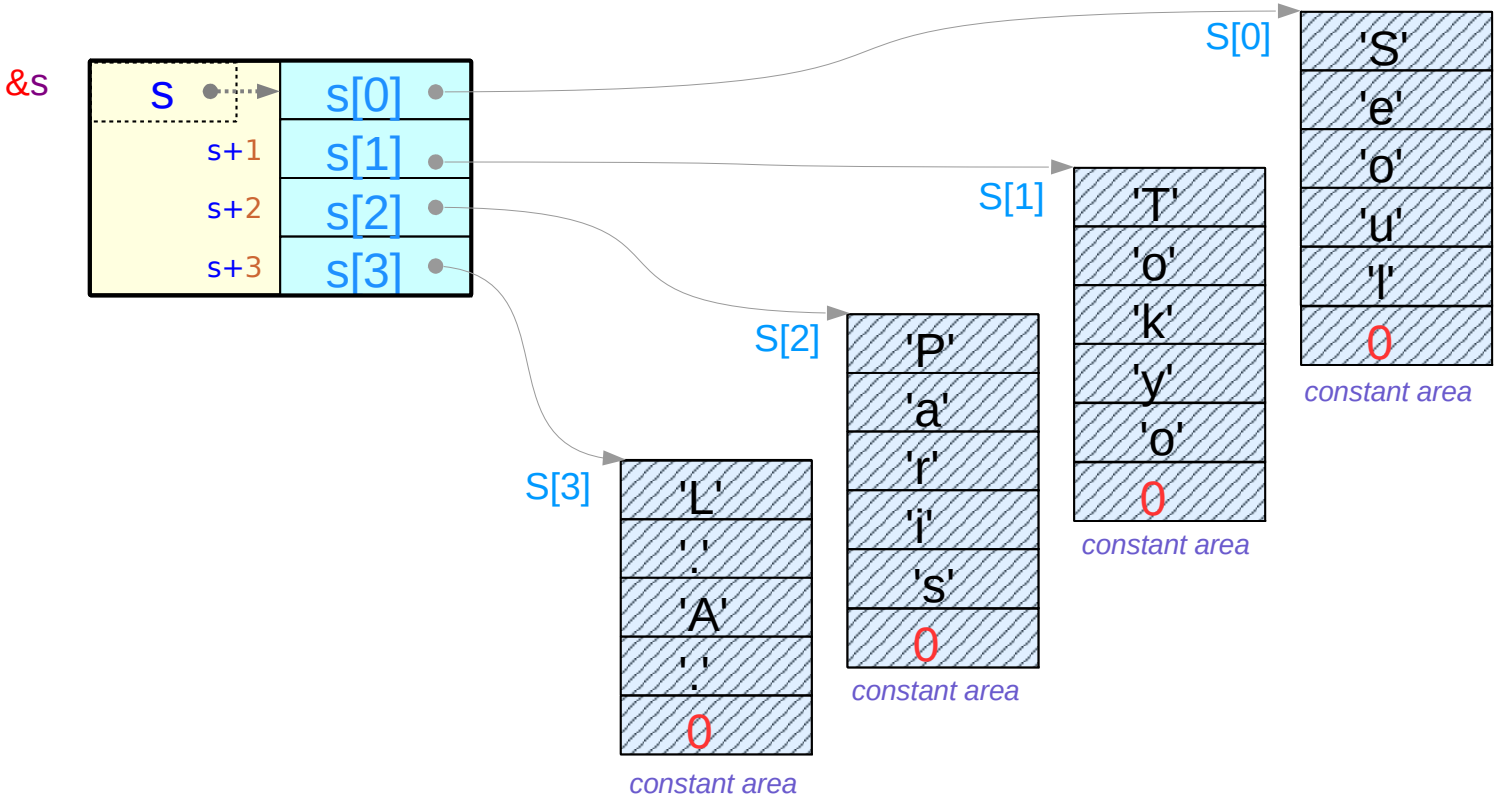
s cannot point to other location

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)
```

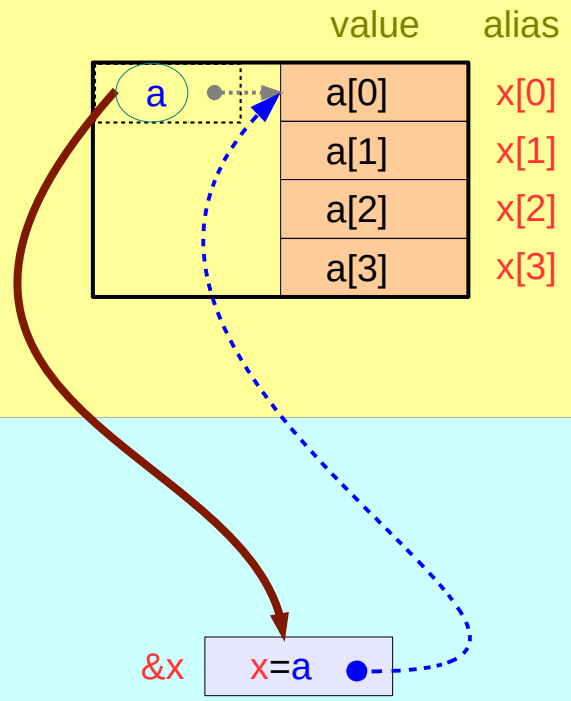
Passing **1-d** Arrays – using **0-d** array pointer

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

func( a );

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

or `int x [ ]`



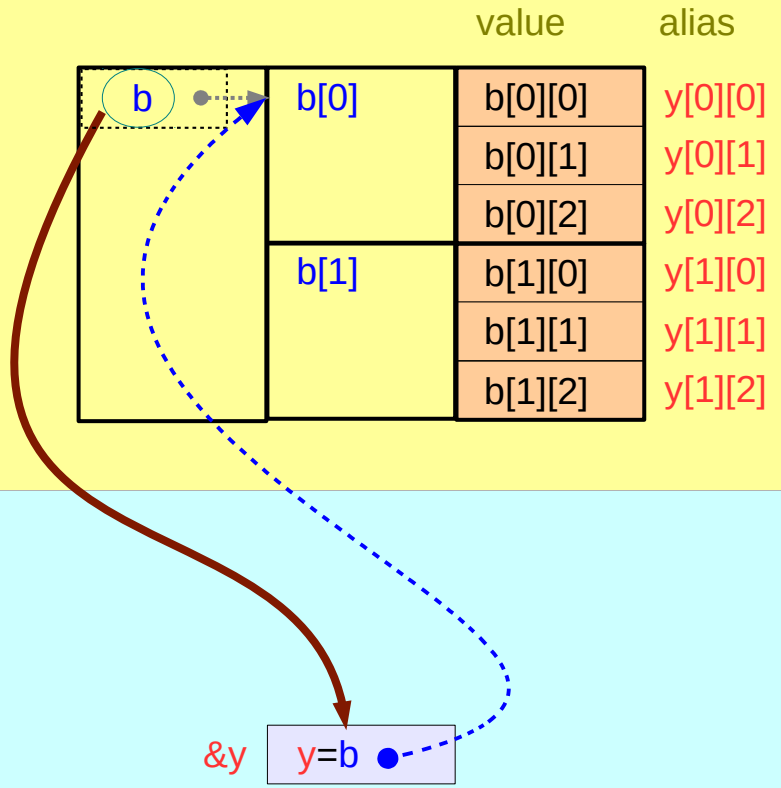
Passing **2-d** Arrays – using **1-d** array pointer

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

func( b );
```

```
func( int (*y) [3]) {
    …
}
```

or int y [ ][3]



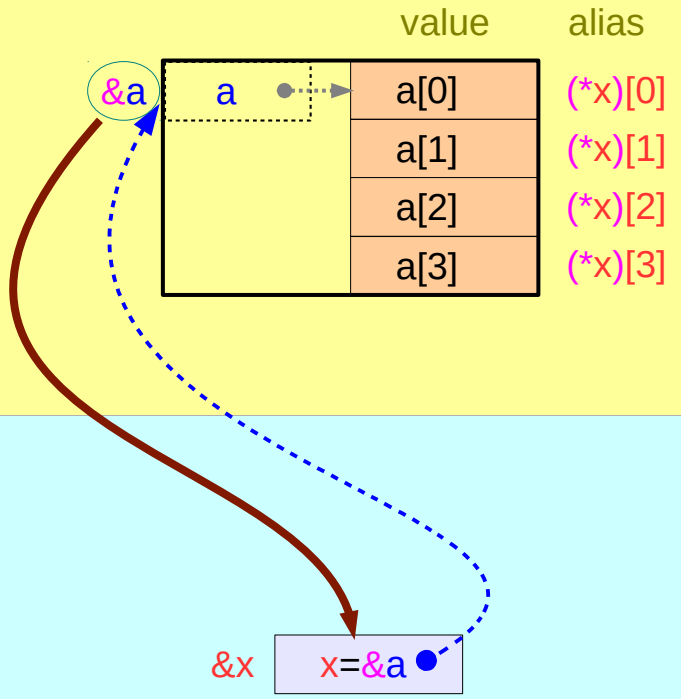
Passing **1-d** Arrays – using **1-d** array pointer

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

func( &a );
```

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

or int **x** [ ][4]



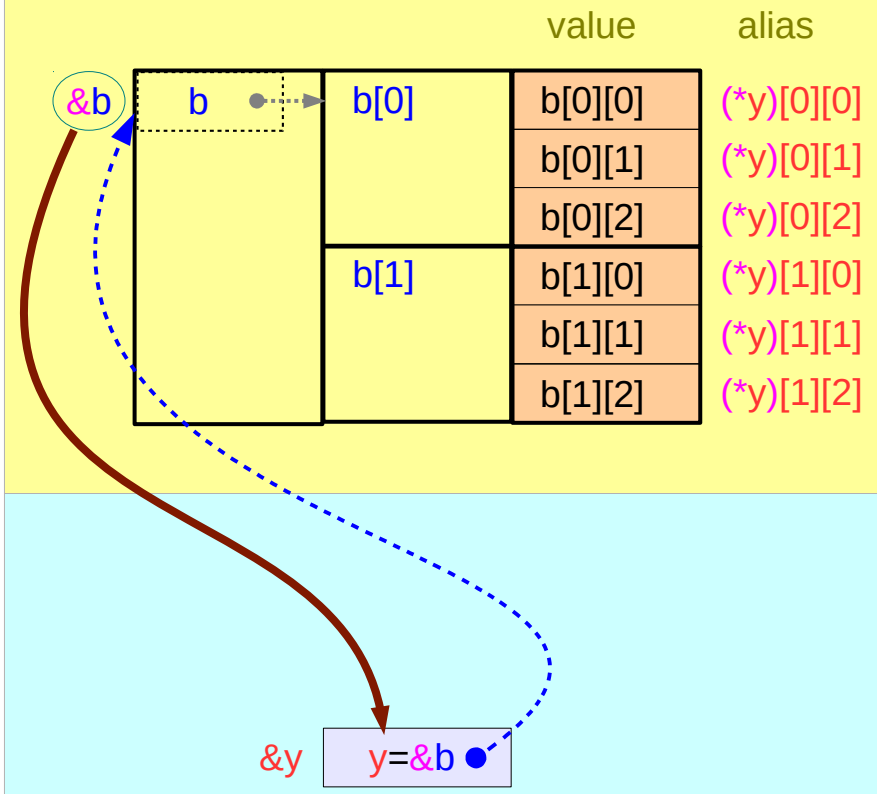
Passing **2-d** Arrays – using **2-d** array pointer

```
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

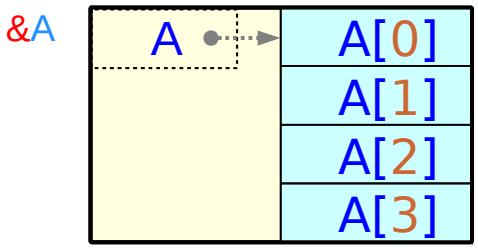
```
typedef int AType [4] ;

AType A;
```

≡

```
int A [4] ;
```

```
A [0] = 100;
A [1] = 200;
A [2] = 300;
A [3] = 400;
```



Pointer to Array Type Definition

```
typedef int AType [4] ;
AType A,  *q;
```

≡

```
int A[10] , int (*q)[4] ;
```

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

≡

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

Pointer to Array Type Assignment

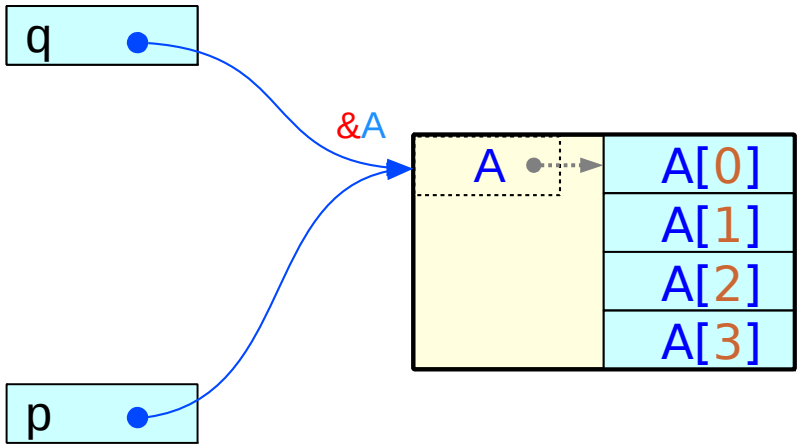
```
typedef int AType [4] ;

AType A,  *q;
```

```
typedef int (* PType) [4] ;

PType p;
```

q = &A ;



p = &A ; *or*

p = q ;

Nested array declared explicitly

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

≡

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

row c[4] allocates 4 array pointers in the memory

```
int         a [4];

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

- Types of array names
- Values of array names

2-d array definition

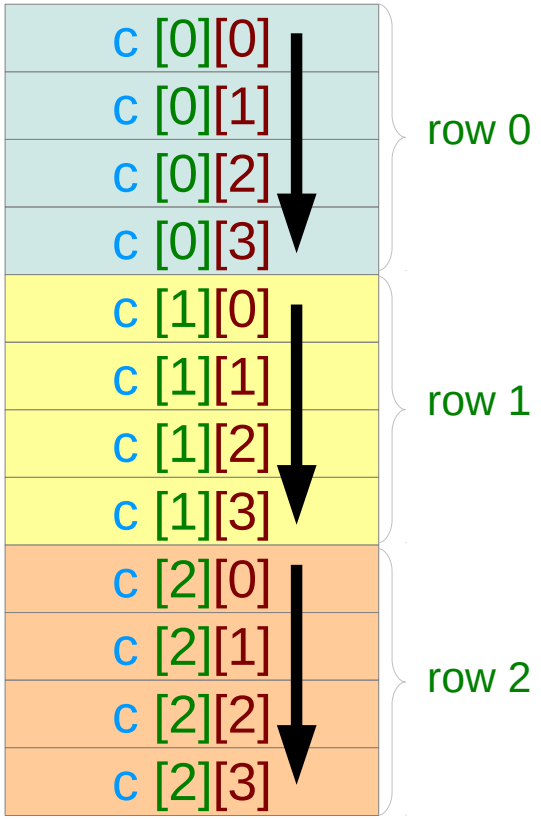
```
int c [3][4];
```

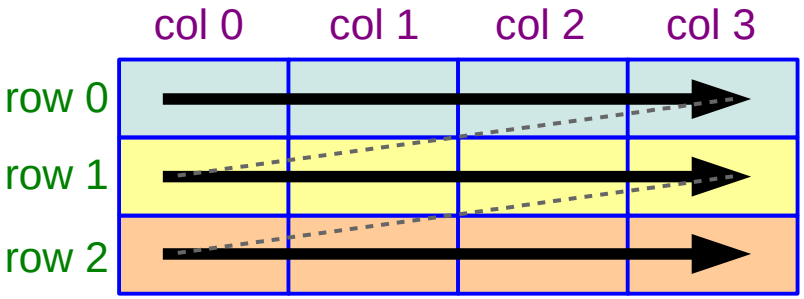
A matrix view

	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 stored as a linear array

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





2-d array element address

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

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

$$X \equiv c[i]$$

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

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

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]

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

Element address = Row address + Column address

Row address **c[i]**

c[i][0] ≡ *c[i]

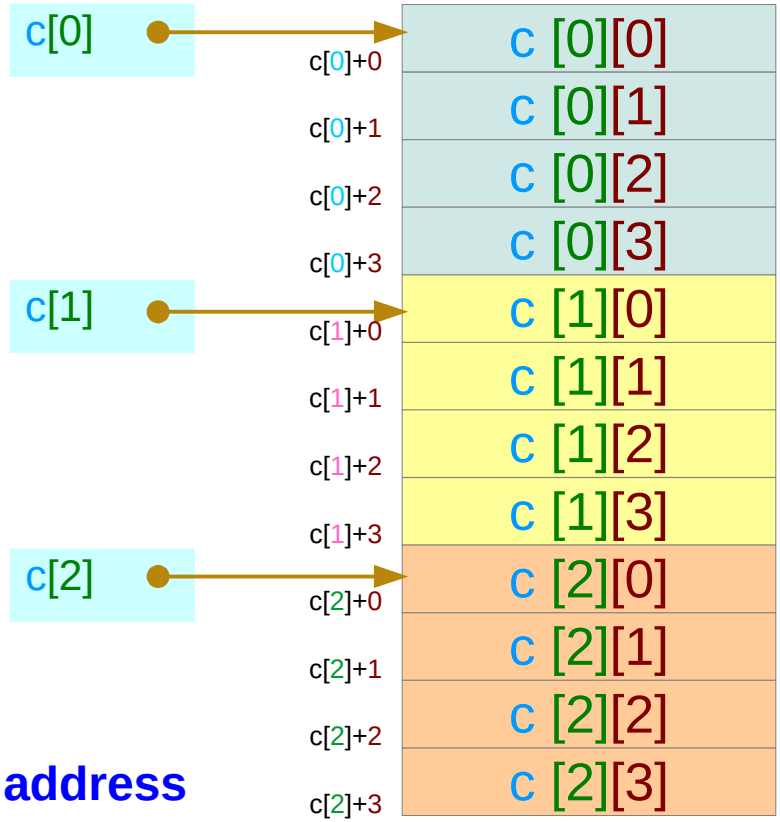
&c[i][0] ≡ c[i]

row address :
to the 1st element of each row

c[i][j] ≡ *(c[i]+j)

&c[i][j] ≡ c[i]+j

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



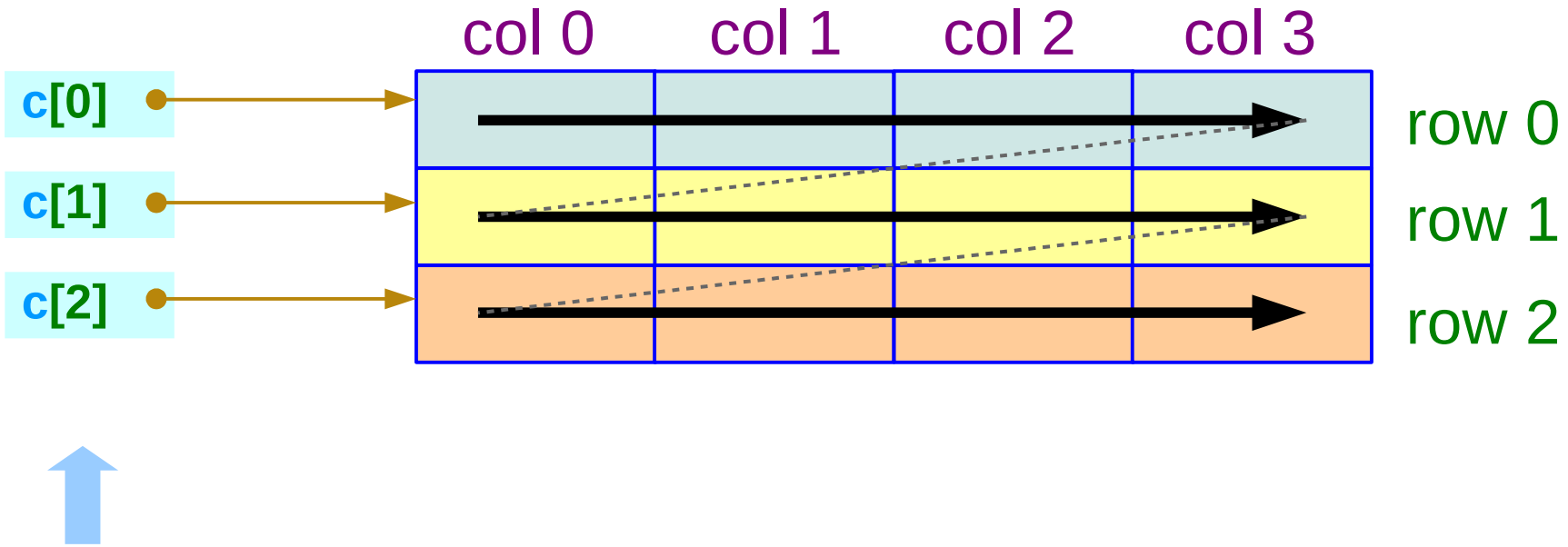
Row address

virtual pointer c[i]

c[i] as a pointer

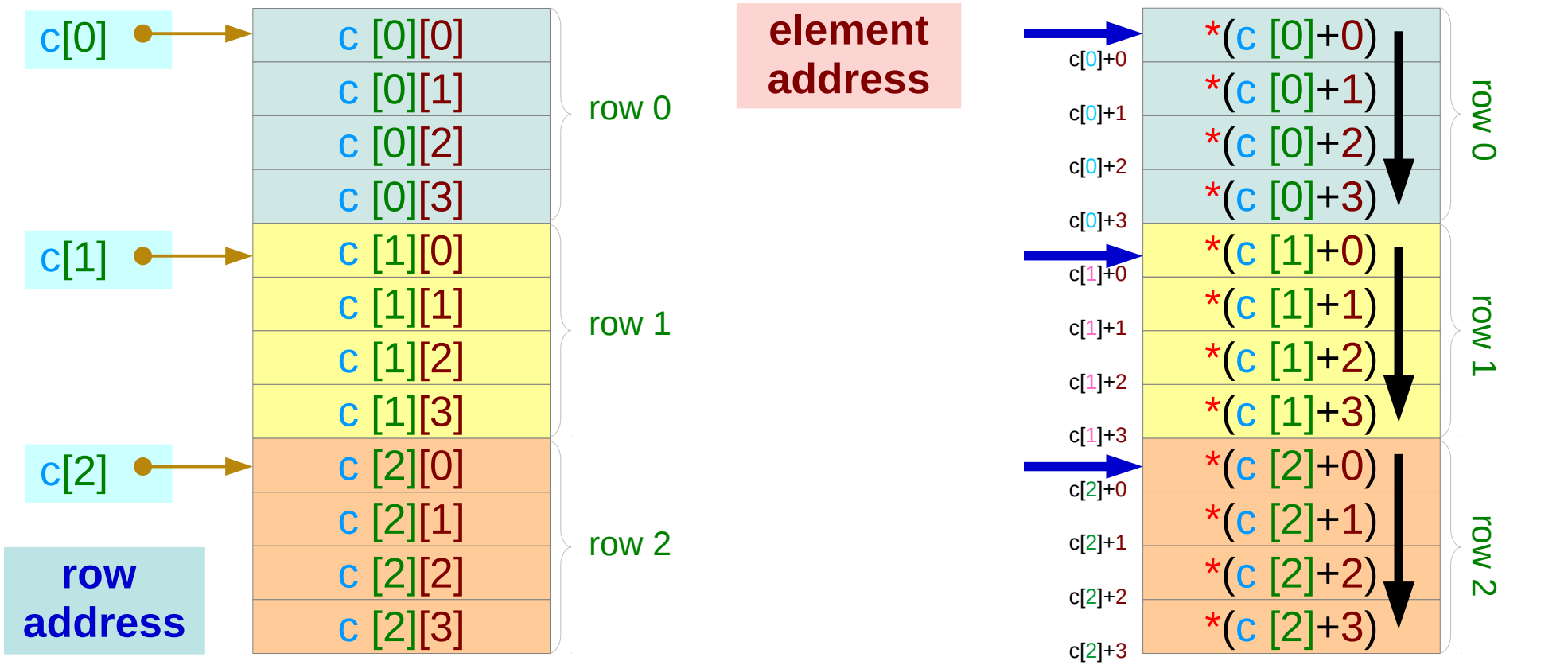
```
int c[3] [4];
```

row major ordering



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

Row Address and Element Address



c[i]

row address + column address

c[i] + j

column address

j

Base address **c**

c[0] ≡ *(c+0) ≡ *c

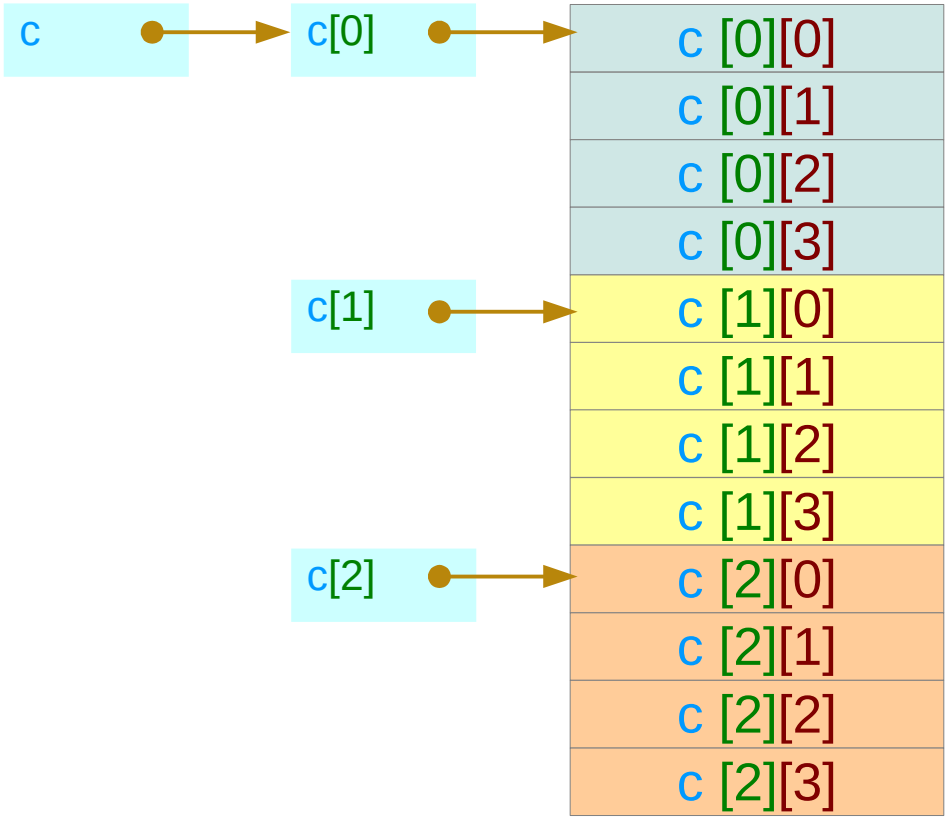
&c[0] ≡ c+0 ≡ c

Base address
to the 1st row virtual pointer

c[i][0] ≡ *c[i]

&c[i][0] ≡ c[i]

virtual pointers
to the 1st element of each row



virtual pointers **c**, **c[i]**

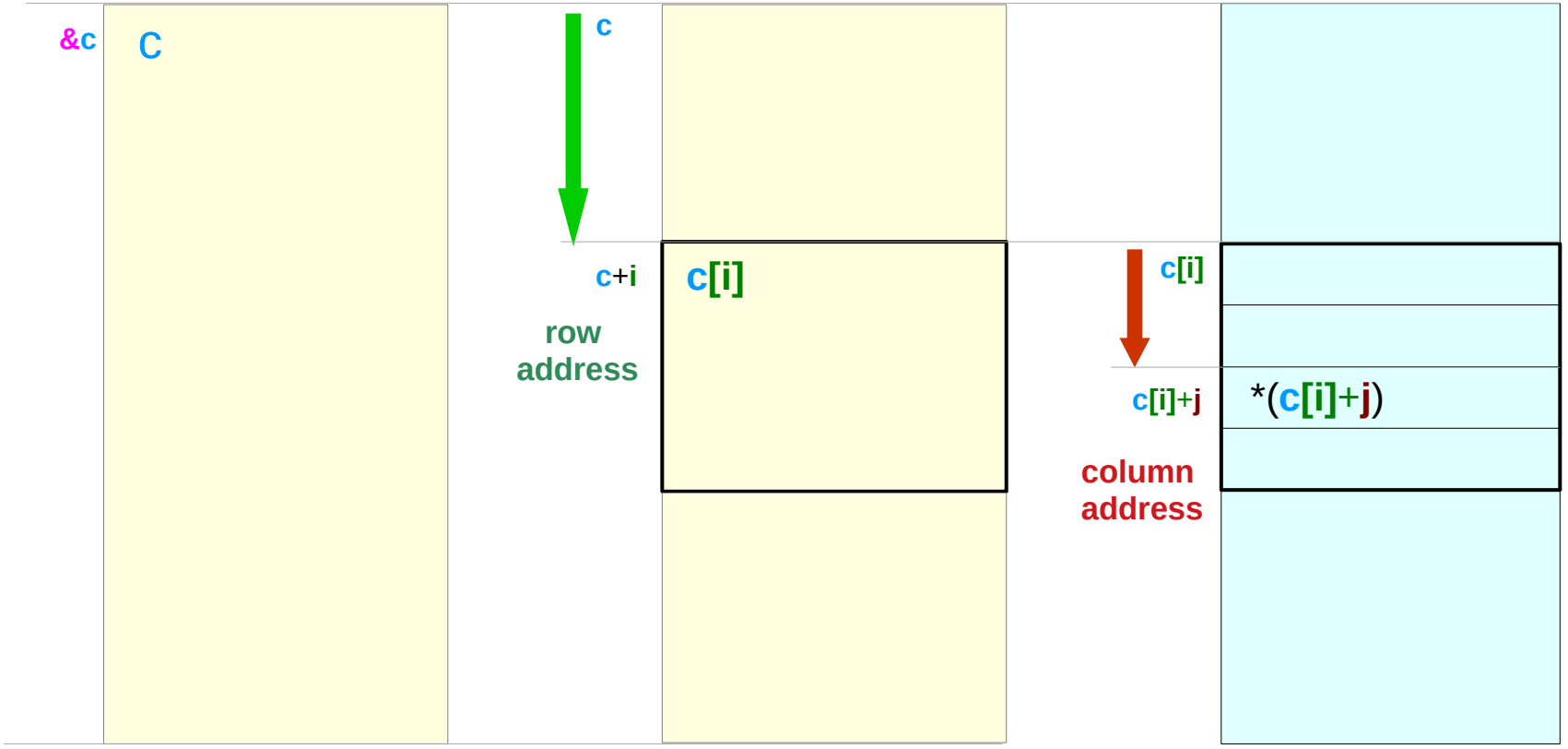
Element address **c[i] + j**

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

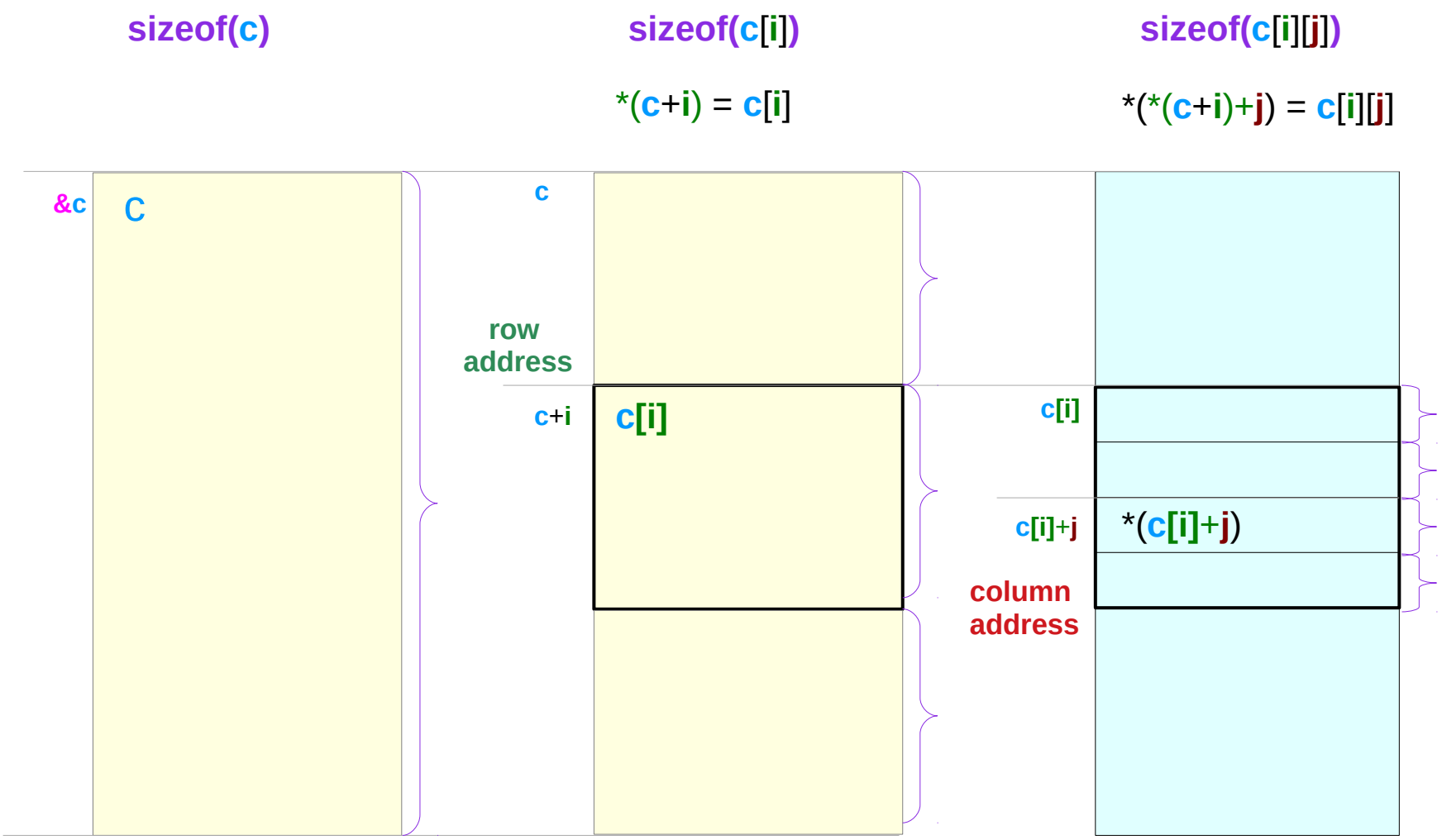
row address + column address

*(c+i) = c[i]

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

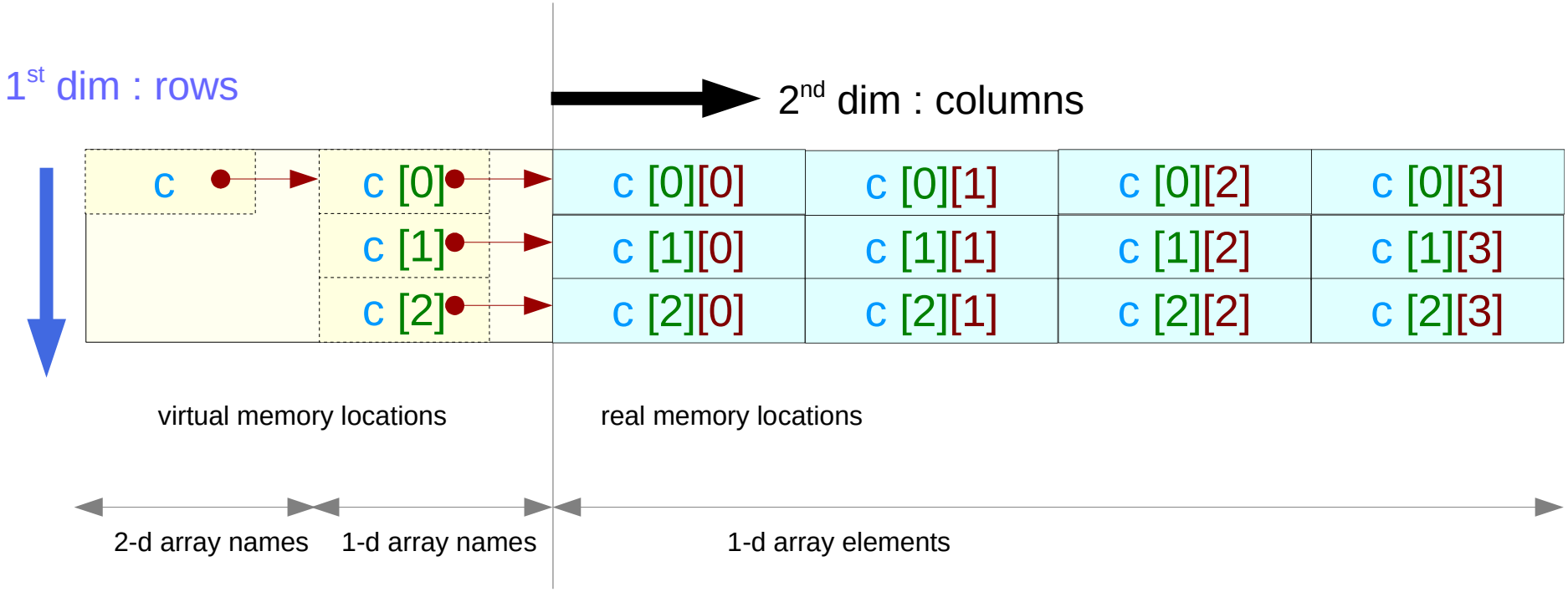


Sizes of **c**, **c[i]**, **c[i][j]**



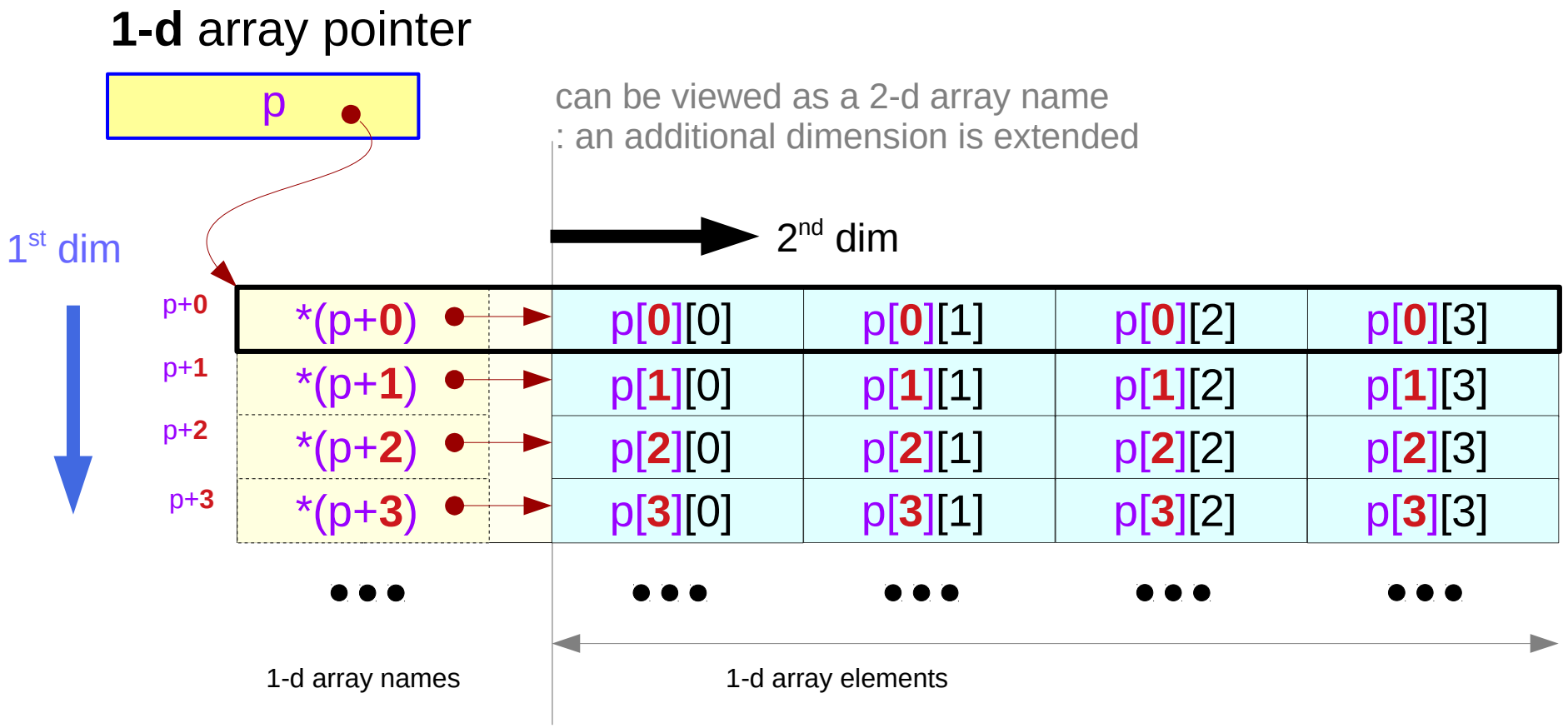
Virtual pointers **c**, **c[i]**

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



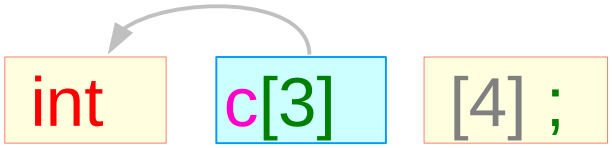
A **1-d** array pointer – extending a dimension

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

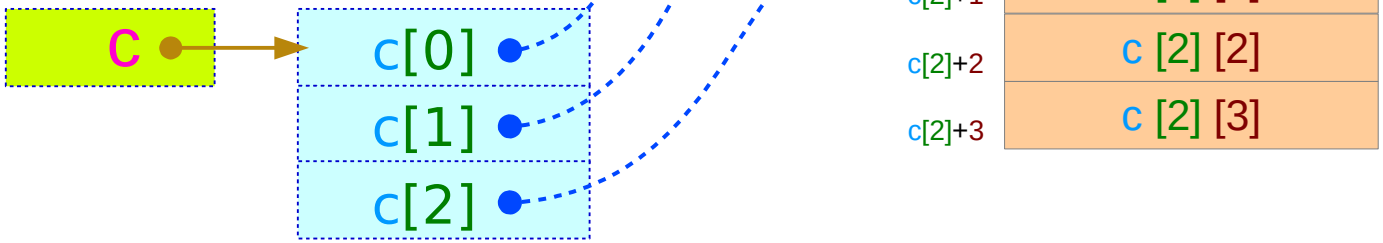


Nested arrays

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



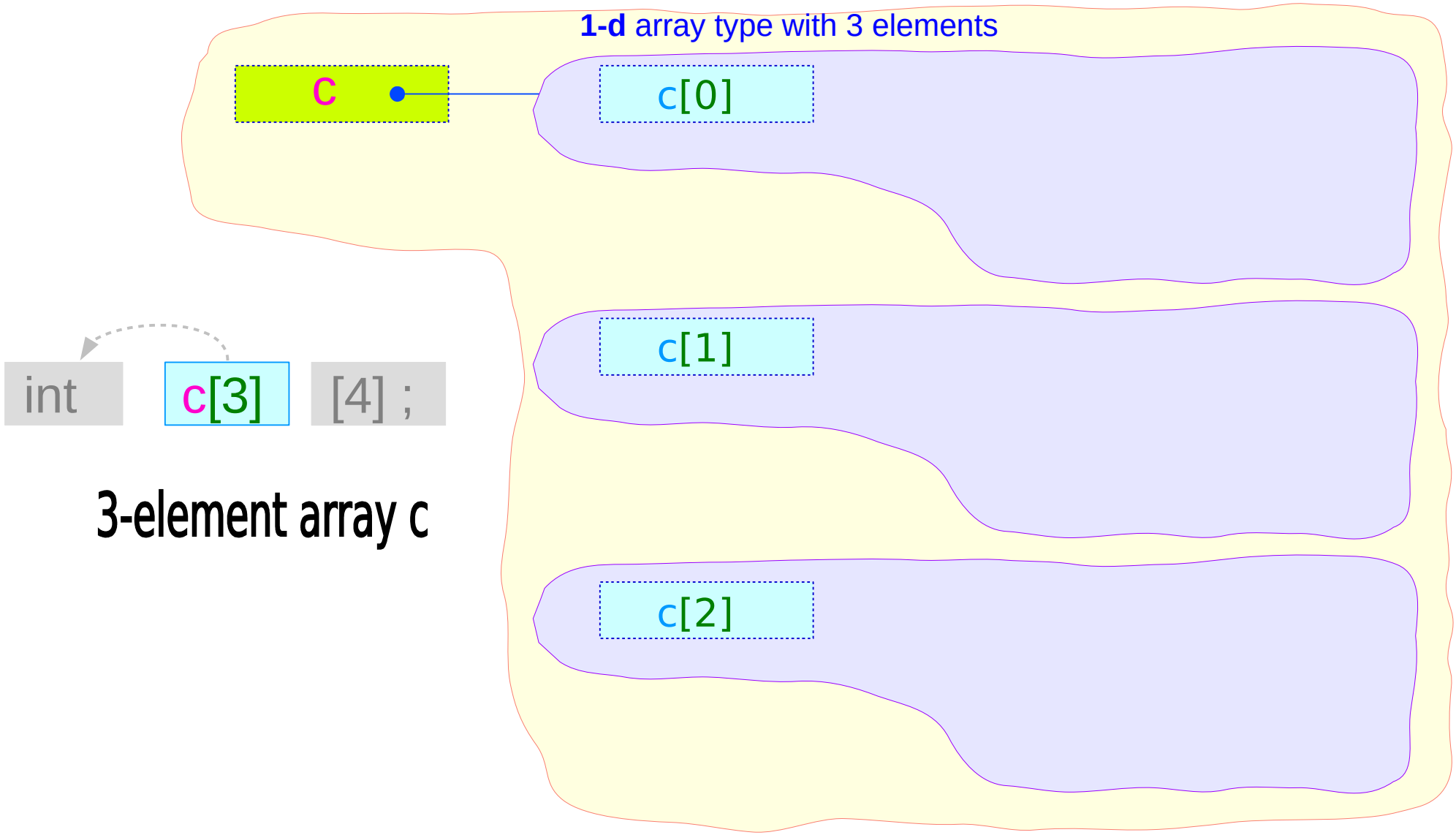
c: an array of
1-d pointers



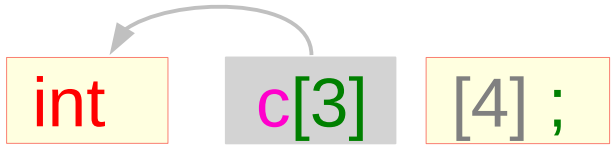
an imaginary array

c, and c[0], c[1], c[2], c[3] take no actual memory locations

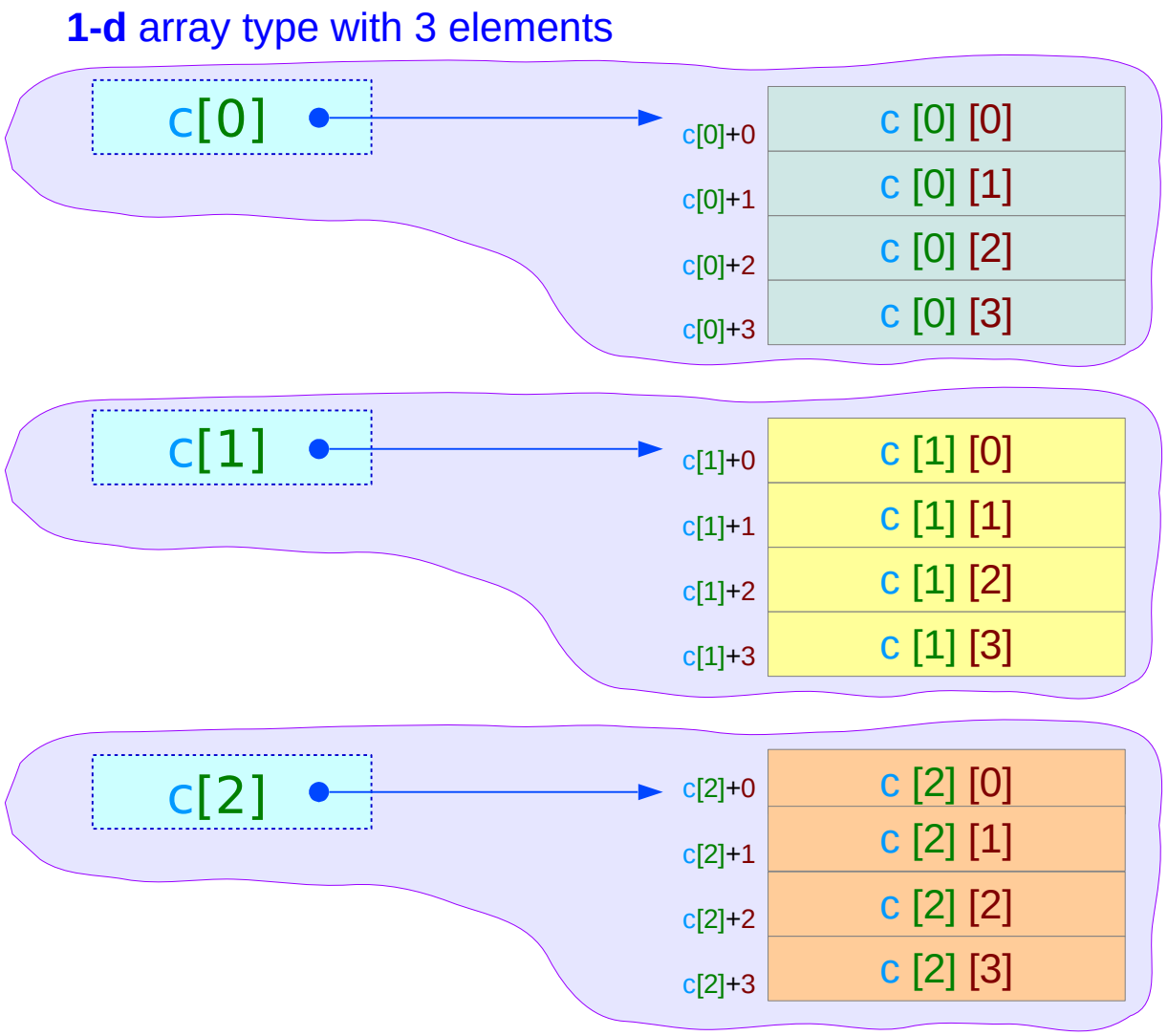
c : 4-element array



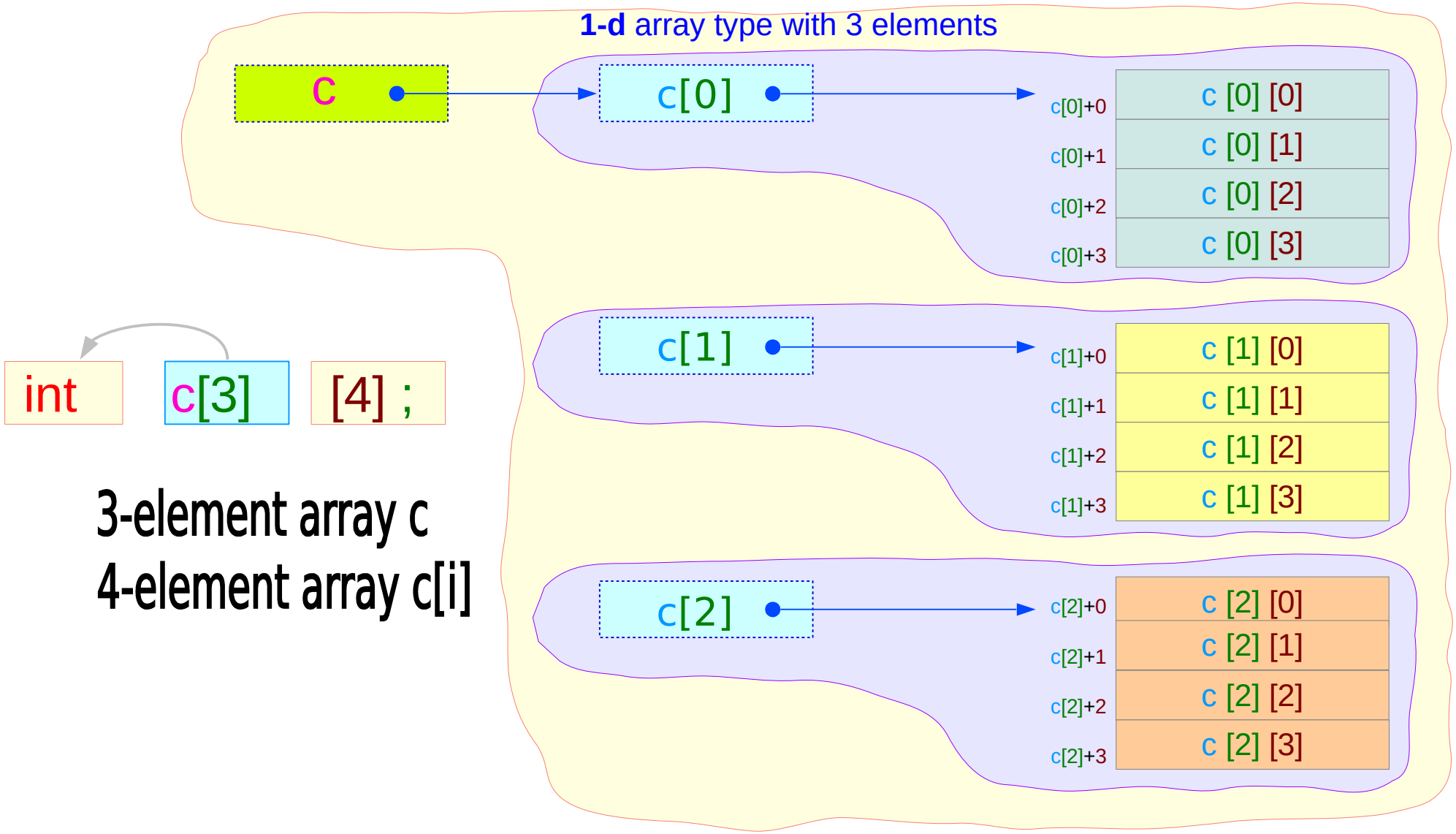
c[i] : 4-element array



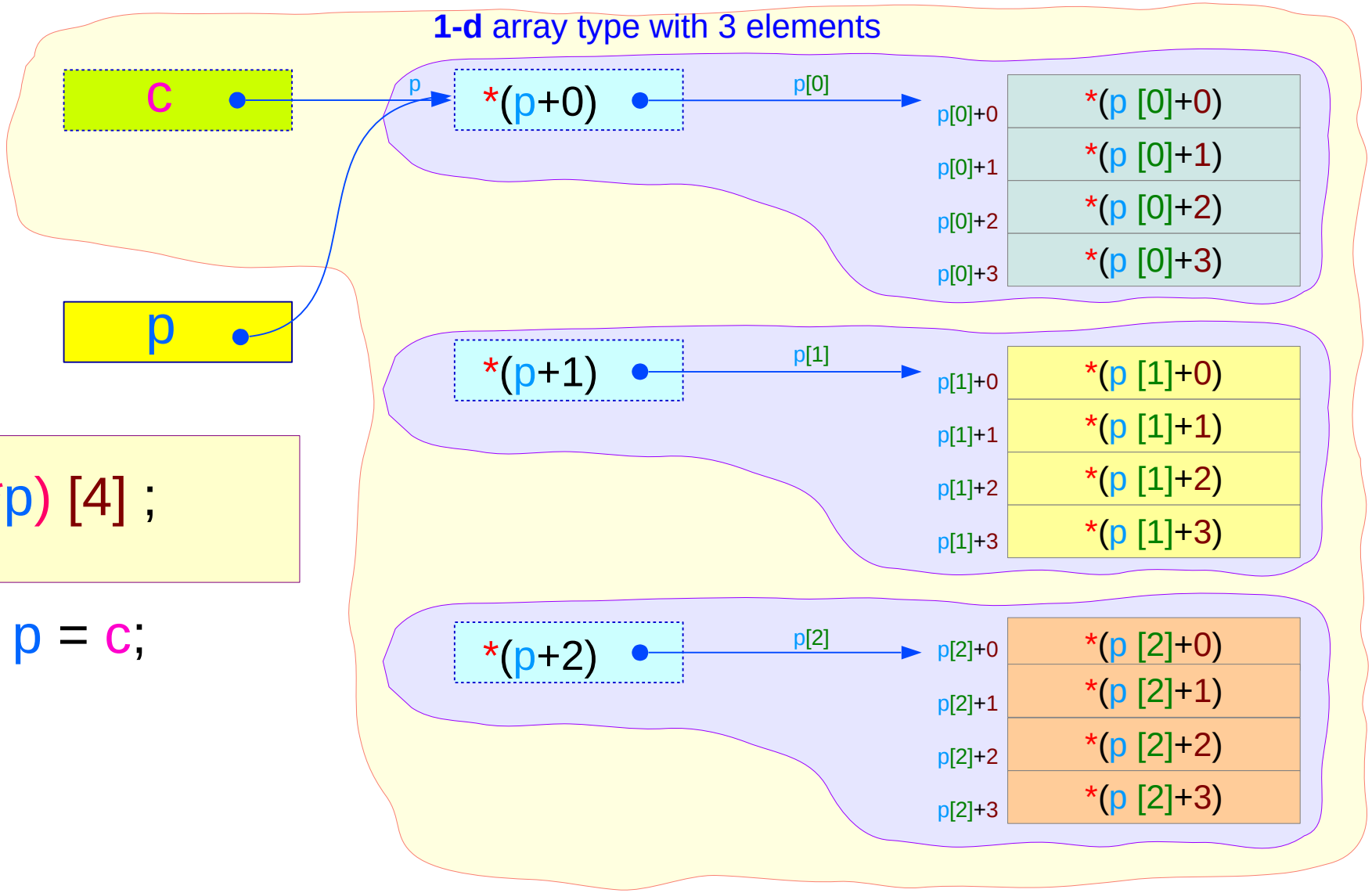
4-element array c[i]



2-d Array : rows of **1-d** arrays



2-d Array : rows of **1-d** arrays




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

```
p = c;
```

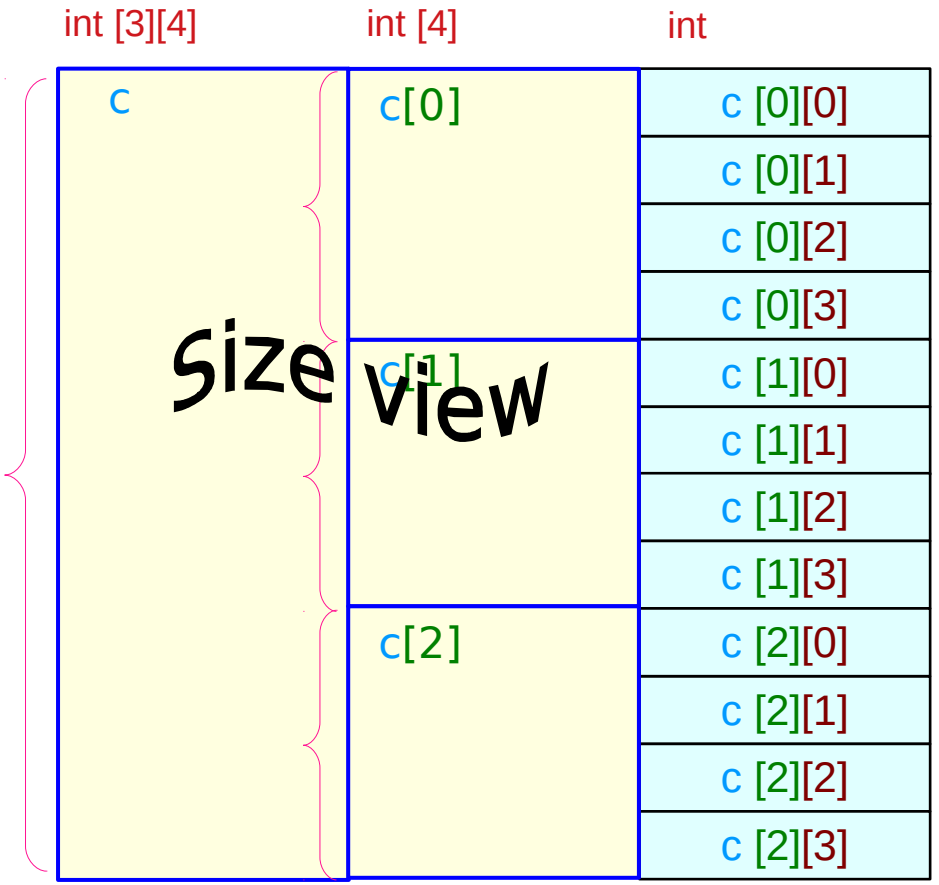
Size view of **c**, **c[i]**

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

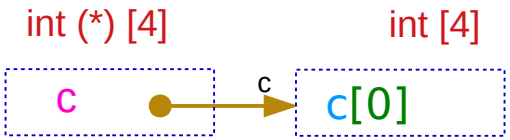
size of 4 elements
c[0], c[1], c[2]

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

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

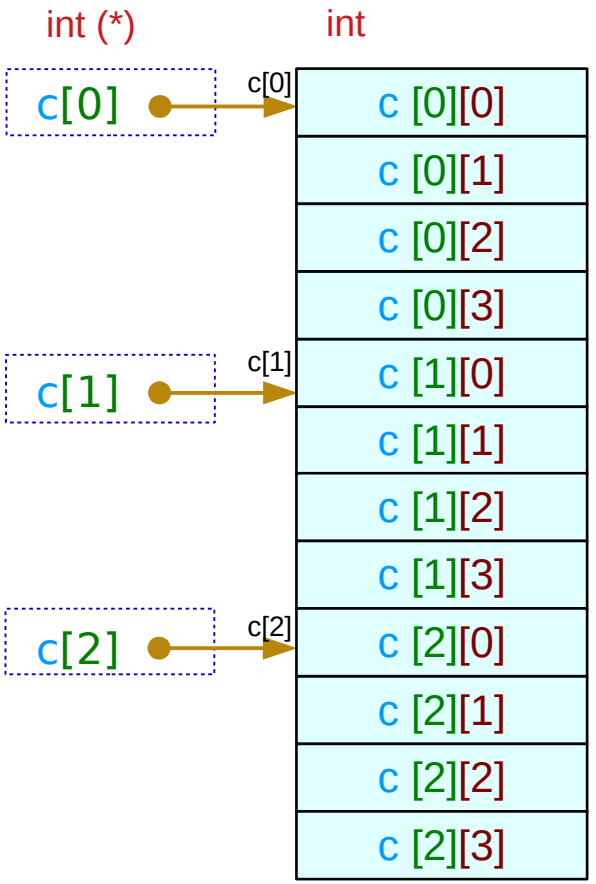


Addresses of **c[i]**, **c[i][0]**



c = &c[0]

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




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

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

Address views of **c**, **c[i]**

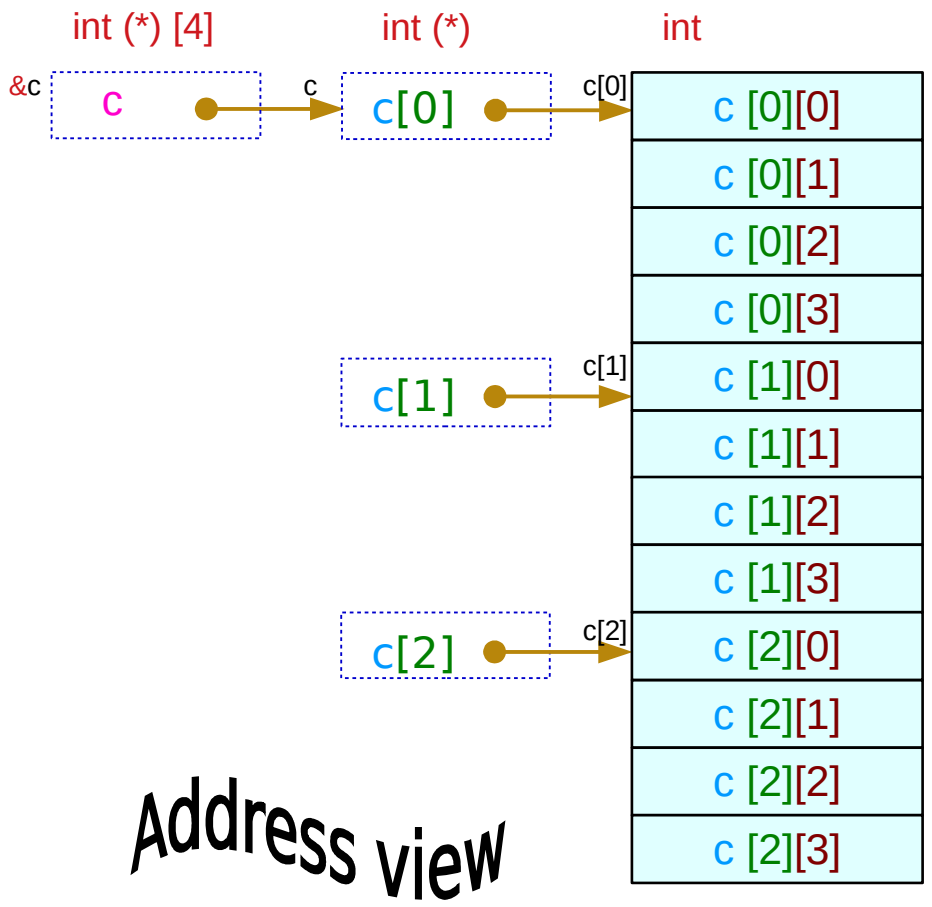
c = c[0] = &c[0][0]
c[1] = &c[1][0]
c[2] = &c[2][0]

&c = c

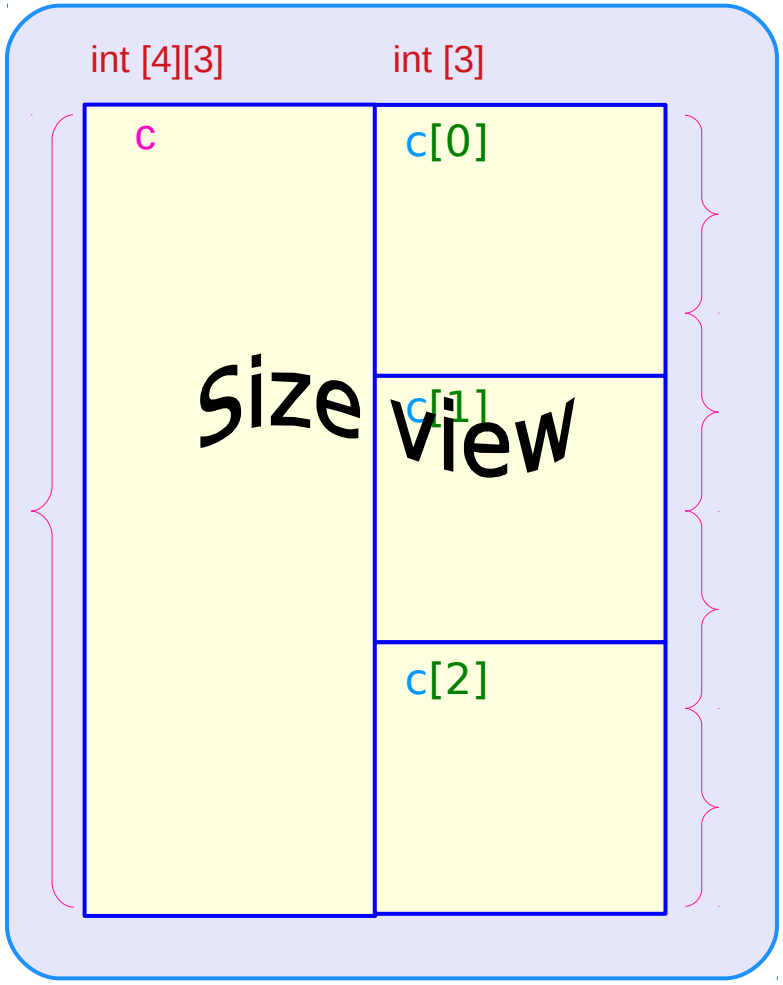
&c[0] = c[0]
&c[1] = c[1]
&c[2] = c[2]

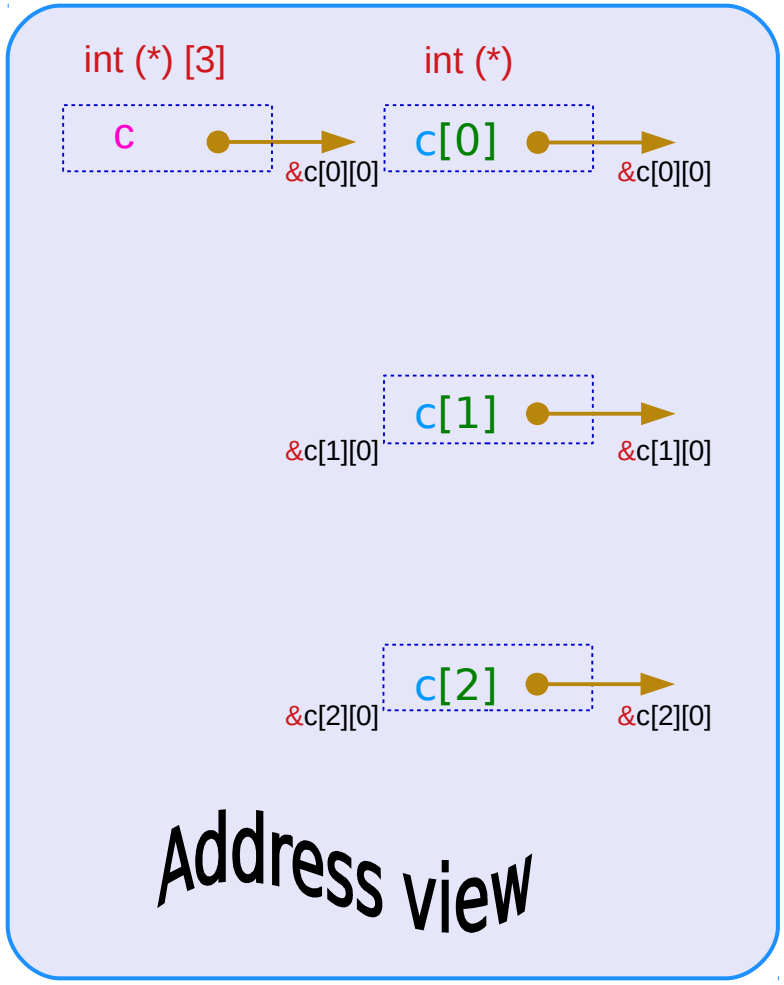
no real pointer can satisfy these conditions

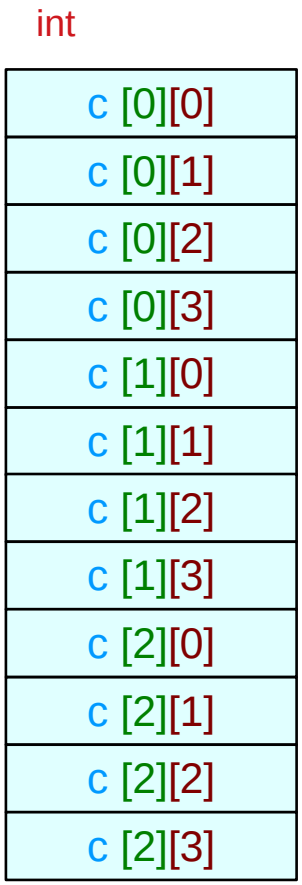
no physical memory location



Size and address views of **c**, **c[i]**



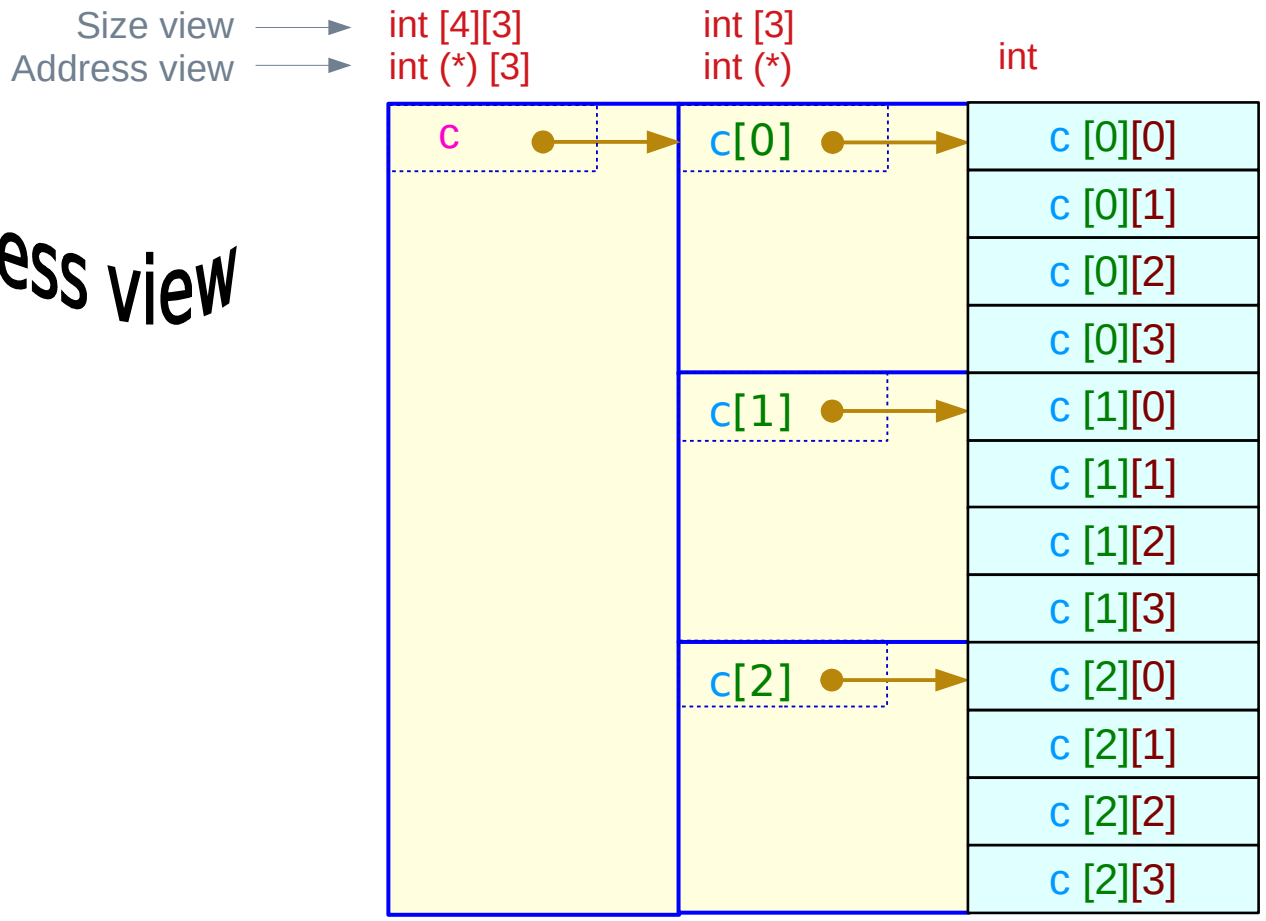




sizeof(c) = sizeof(c[i]) * 3

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

Combining size view and address view



Size view + Address view

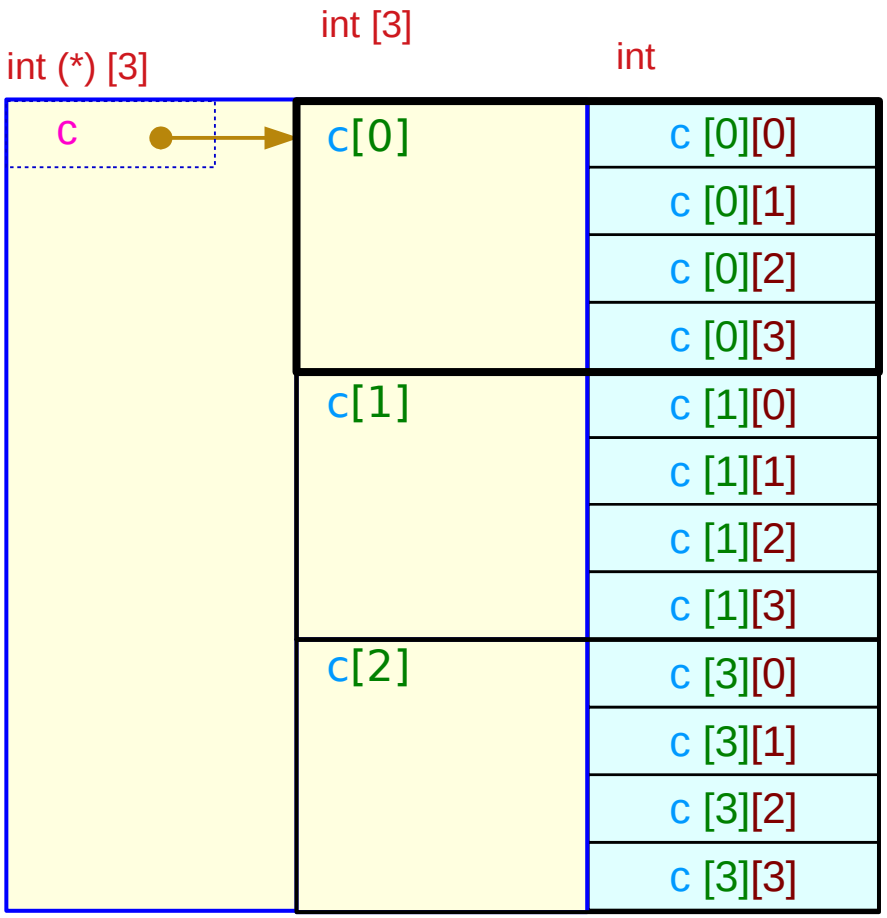
sizeof(c) = sizeof(c[i]) * 3

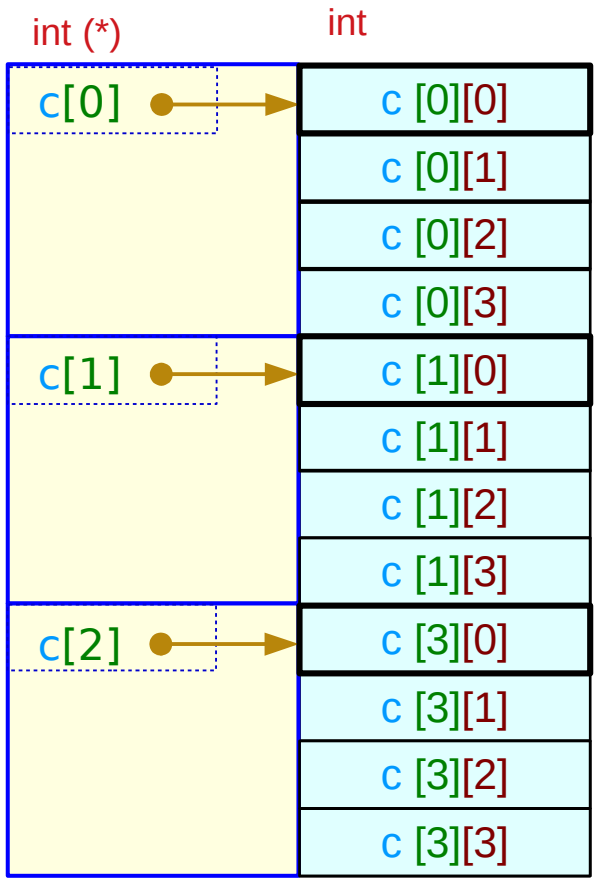
c = c[0] = &c[0][0]
c[1] = &c[1][0]
c[2] = &c[2][0]

an imaginary array

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

c points c[0], c[i] points c[i][0]





an imaginary array

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

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]**

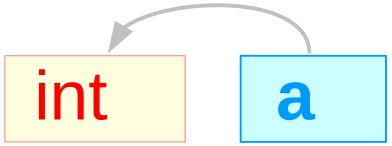
start address = &**c[i]**[0] sizeof(**c[i]**) = 4 * 4

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

c is the name of the **2-d** array **int [3][4]**

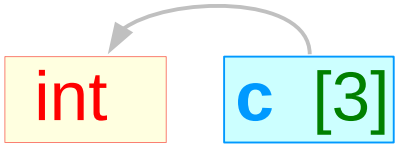
start address = &**c**[0][0] sizeof(**c[i]**) = 3 * 4 * 4

Values of array names



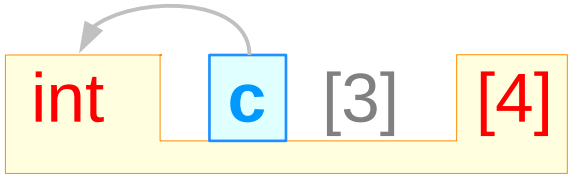
the value of **a** is the starting address of *a 4 element array* of **int** type — **int (*)**

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 (*)**

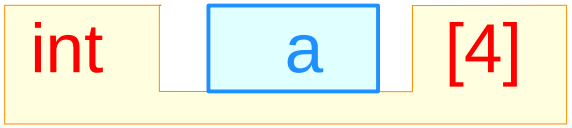
c[i]: pointer to the first element — **&c[i][0] = c[i]**



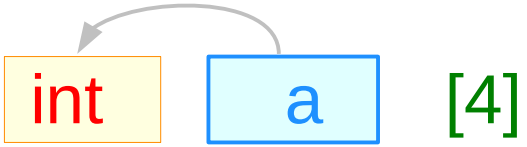
the value of **c** is the starting address of *a 3 element array* of **int [4]** type — **int (*) [4]**

c: pointer to the first element — **&c[0] = c**

Array and pointer types in a **1-d** array

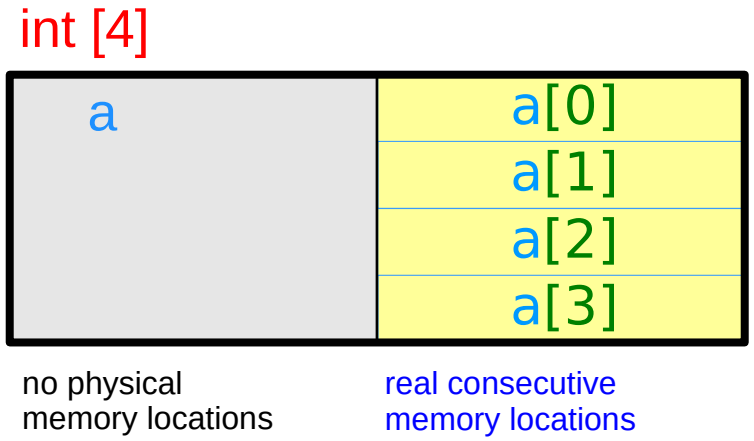


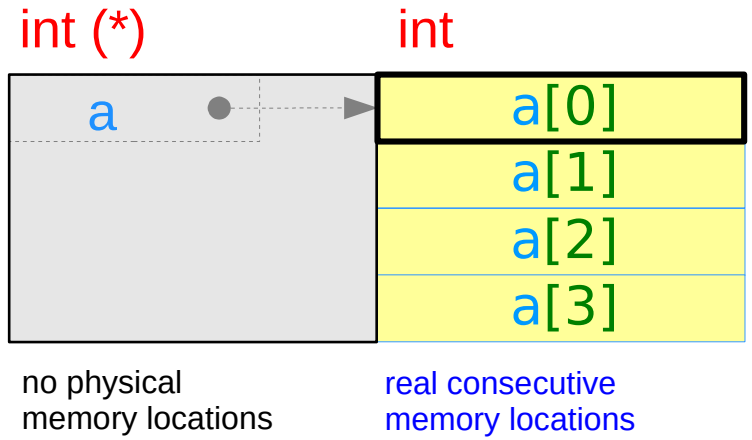
a **1-d array**
type : int [4]
size : 4 * 4



a **0-d array pointer** (virtual)
type : int (*)
size : 4 * 4

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





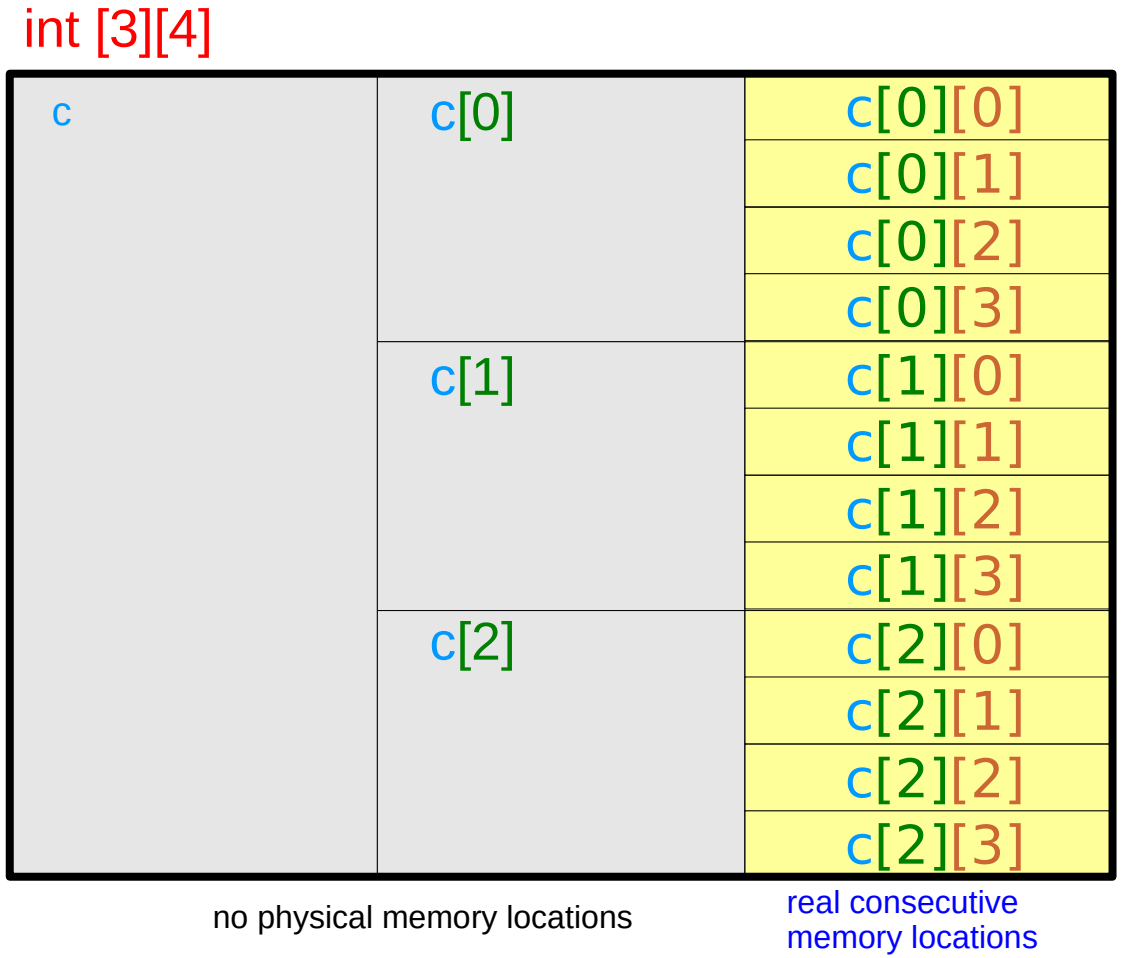
2-d array type

int c [3] [4]

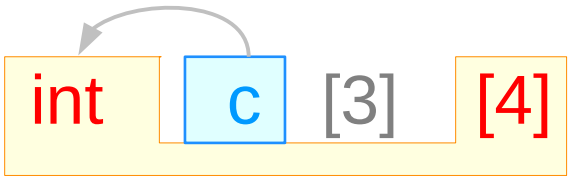
C 2-d array

type : int [3][4]

size : 3 * 4 * 4



1-d array pointer type

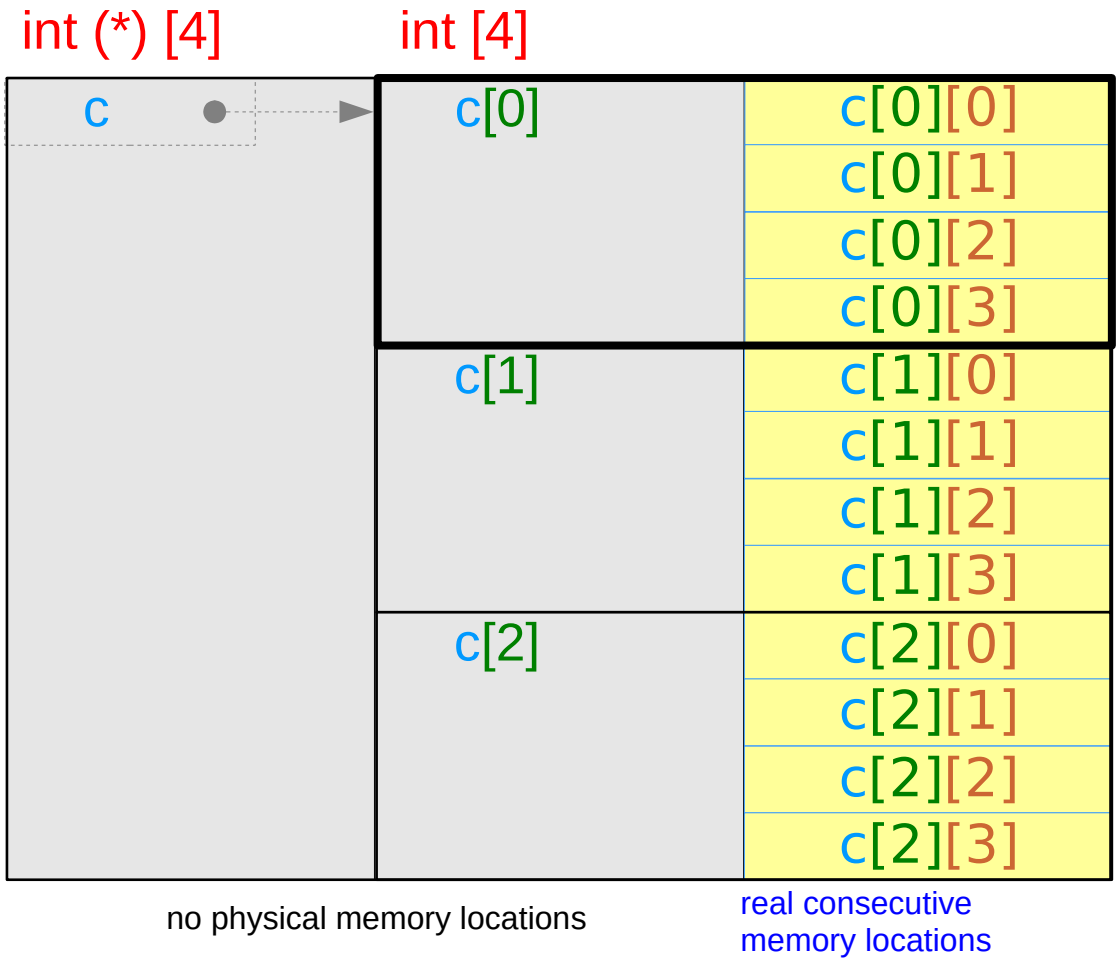


C **1-d array pointer** (virtual)

type : int (*) [4]

size : 3 * 4 * 4

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



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