Arrays (1A)

Young Won Lim
2/10/24

Please send corrections (or suggestions) to 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

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

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

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: **int [5]**

Array Variable (constant): **x**

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

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

Integer Variables

```
x [ i ]
```

Element Type: **int**

Element Variable: **x[i]**

Index Variable: **i**

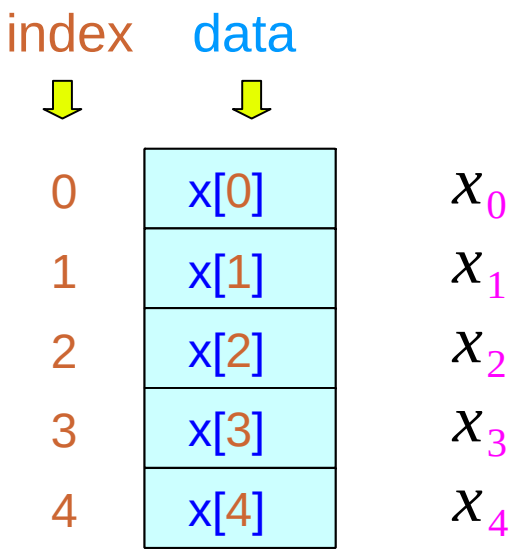
meaningful only for
i = 0, … , 4

Accessing array elements – using an index

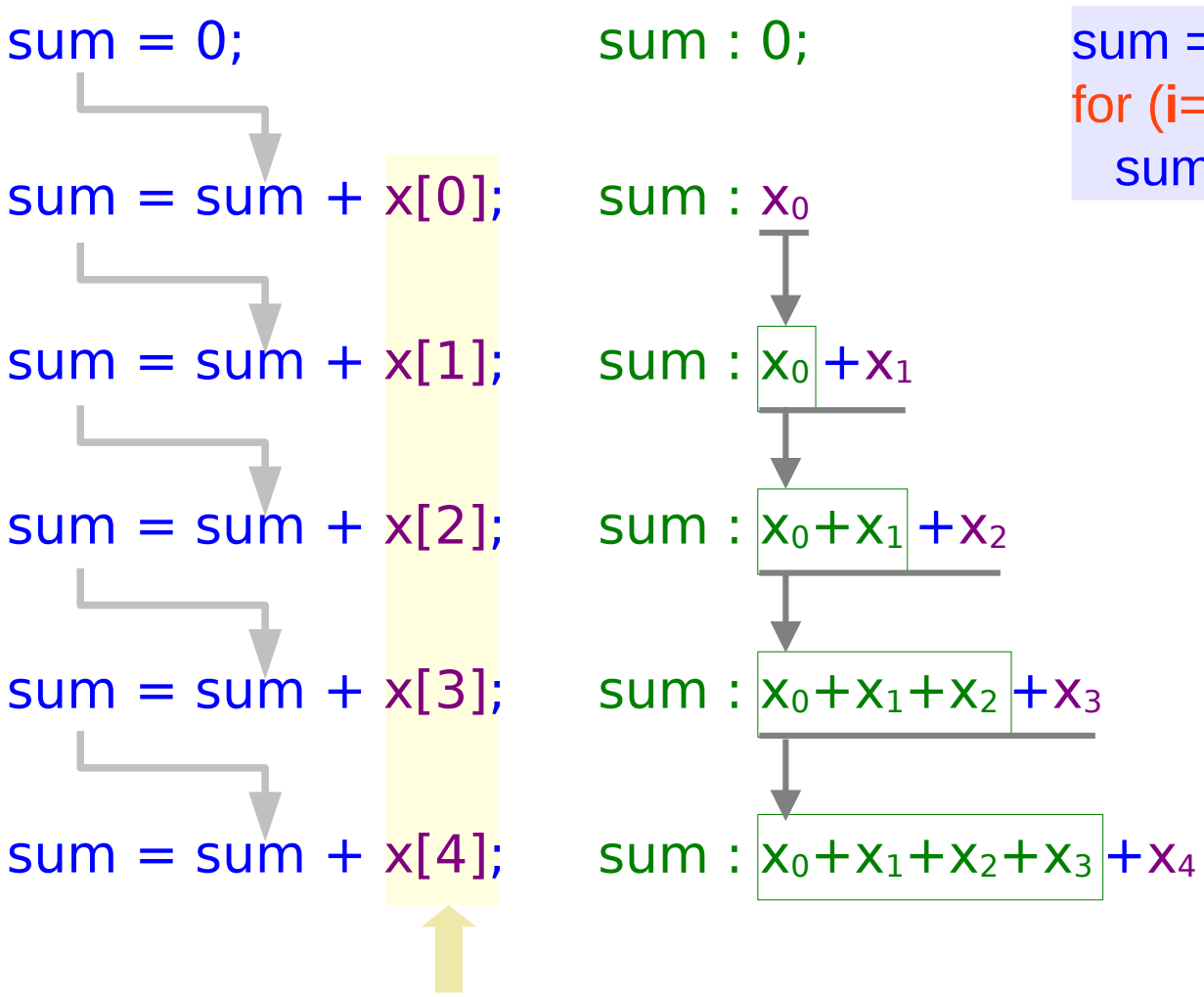
```
int     x[5];
```

x is an array of 5 integer elements

5 **int** variables

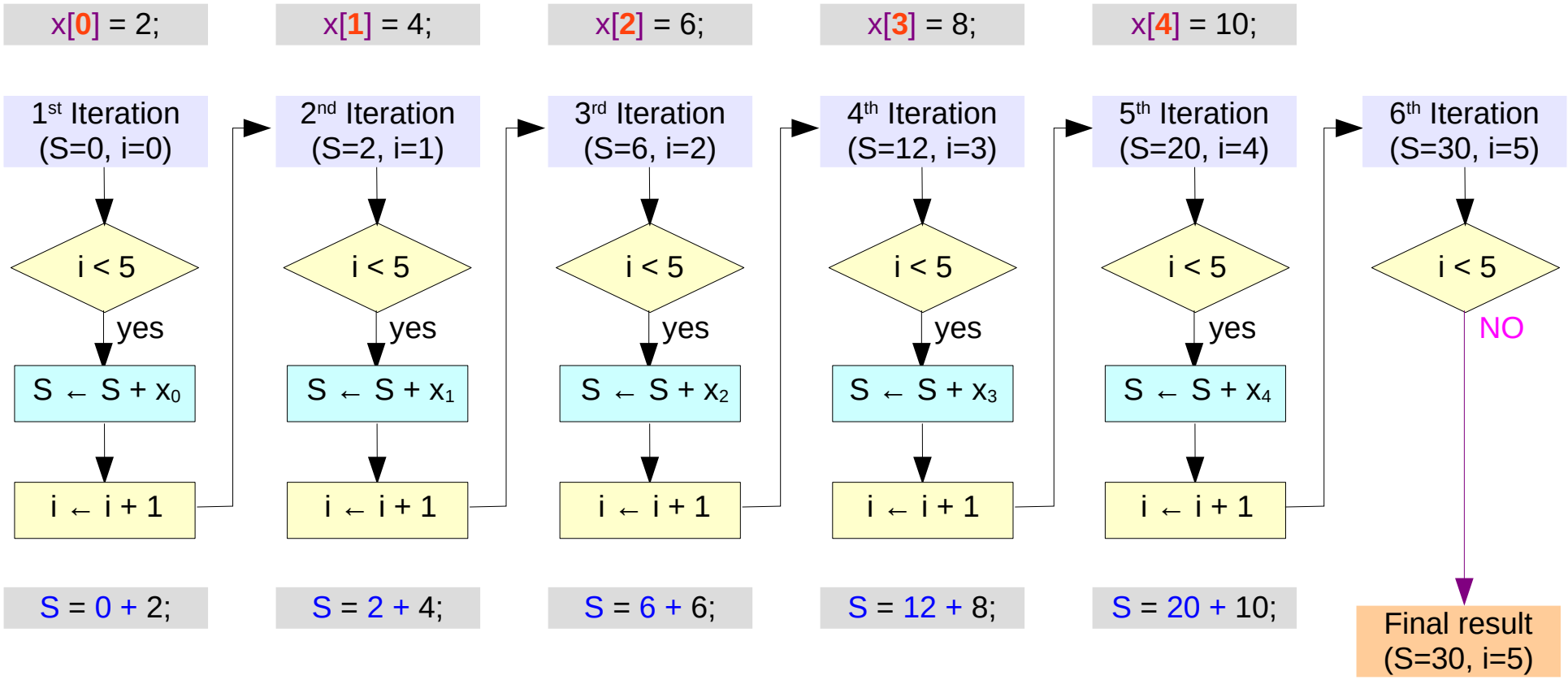


Computing the sum of ***n*** numbers (1)




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

Computing the sum of ***n*** numbers (2)



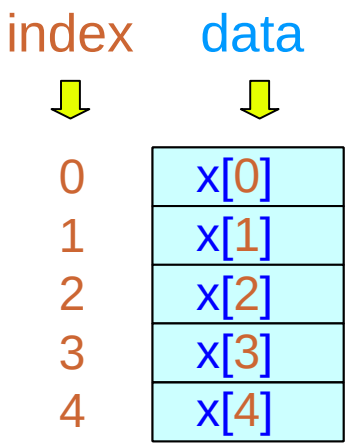

```
S = 0;
for (i=0; i<5; ++i)
  S = S + x[i];
```

Accessing array elements – using an address

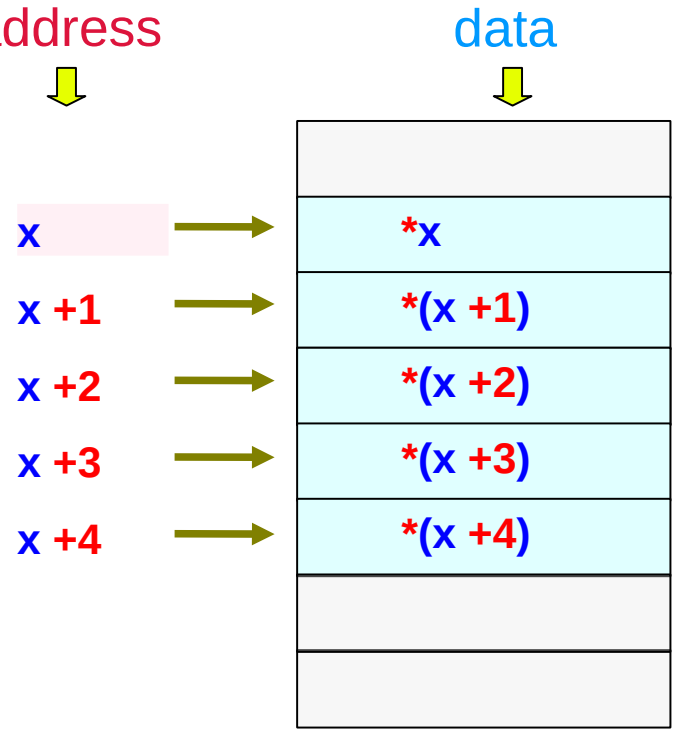
```
int     x[5];
```

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)th element variable

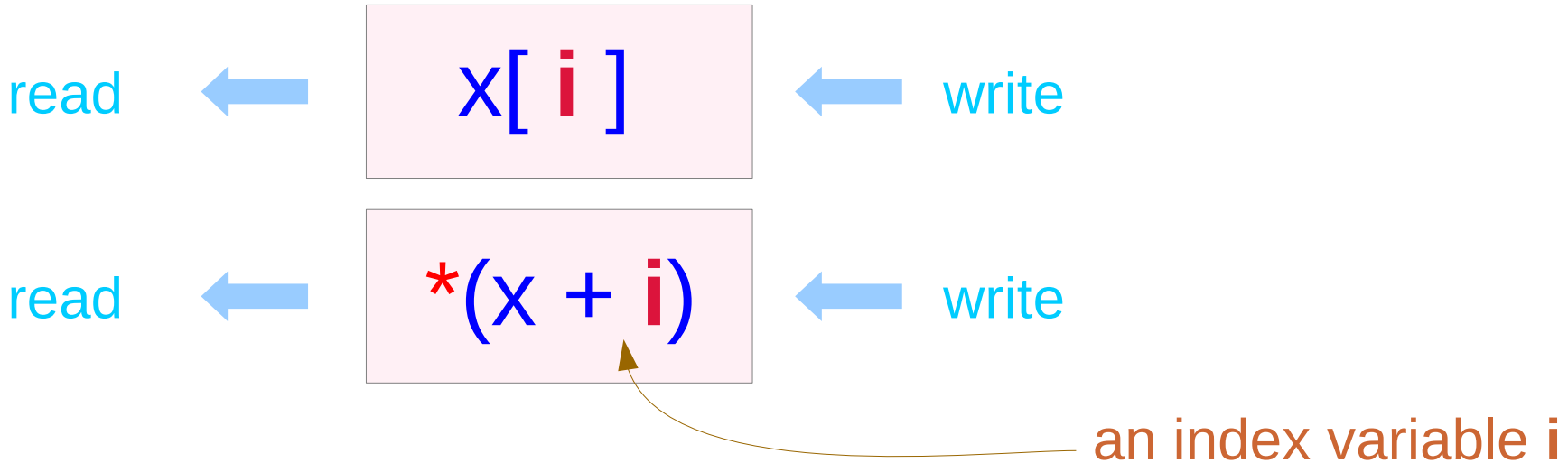
x	: the starting address
x+i	: the address of the (i+1)th element
***(x+i)**	: the (i+1)th element variable

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**

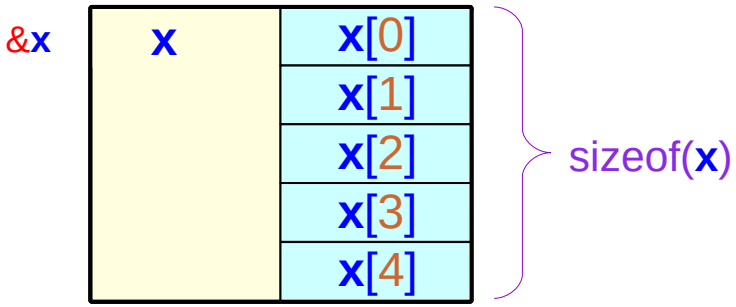
Value and size of an array variable

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

x : an array variable name (constant)

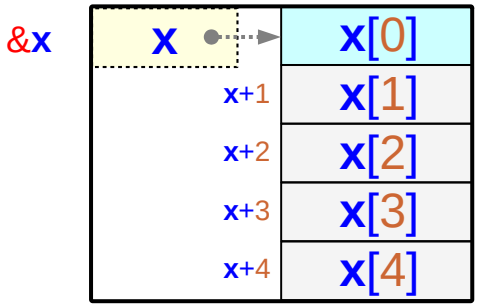
sizeof(**x**) : the *total size* of
5 consecutive **int** variables

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



sizeof(**x**) = 5 * sizeof(int)

subarray partitioning



value(&**x**) = value(**x**)

address replication

An array variable as a virtual pointer

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

x : an array variable name (constant)

equivalence relations

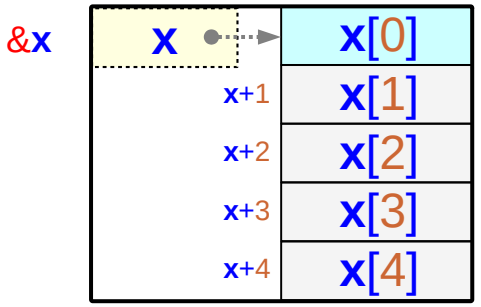
*(**x**+0) ≡ **x**[0]

(**x**+0) ≡ &**x**[0]

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

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

x can be viewed as a pointer because **x** holds the address of the 1st array element **x[0]**

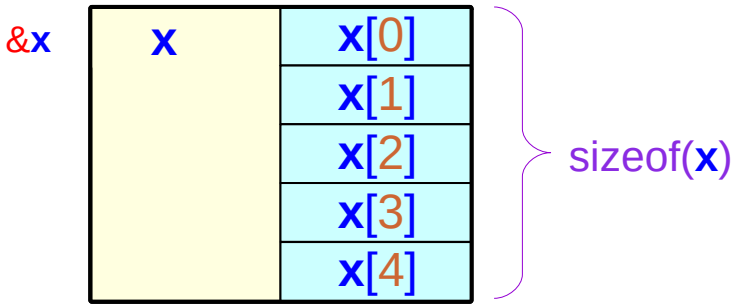


The address of an array variable

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

x : an array variable name (constant)

sizeof(**x**) : the *total size* of
5 consecutive **int** variables



when an array **x** is referenced,
the address &**x** of the array **x**
is the same as the address
of the 1st element &**x**[0] **= x**
: address replication

value(&**x**) = value(**x**)

Outside and inside array types

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

x : an array variable name (constant)

outside of an array x

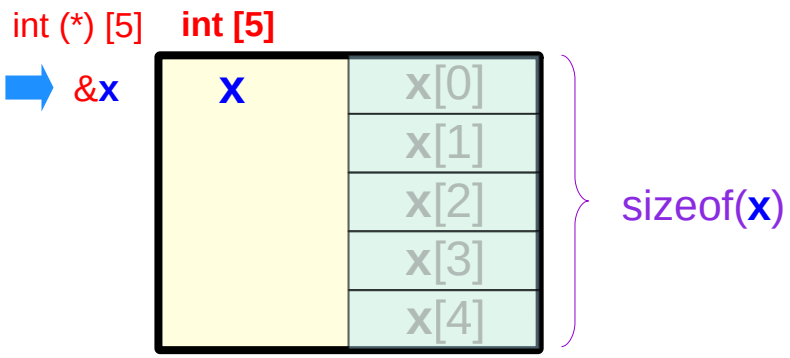
when an array is referenced

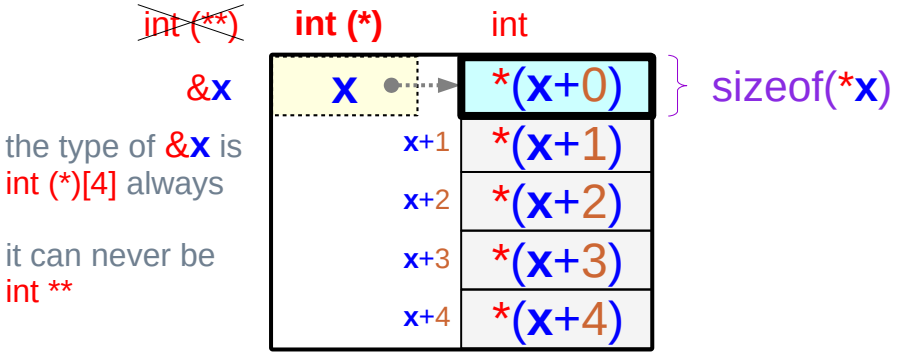
x has an array type int [5]

inside of an array x

when an element of an array is referenced

x can be viewed as a pointer type int (*)





Abstract Data x
an array

Primitive Data *x = x[0]
an array element

Abstract data **x** and virtual pointer **x**

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

x : an array variable name (constant)

outside of an array x

when an array is referenced

Abstract Data x – an array

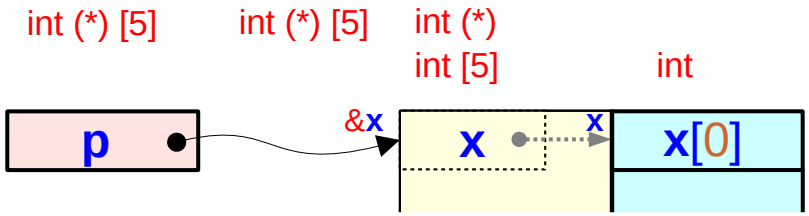
Address	value(&**x**) = value(**x**)	*the same*	
Size	sizeof(**x**) = 5 * sizeof(int)	*the same*	
Type	int [5]	*different*	

inside of an array x

when an element of an array is referenced

Virtual Pointer x – pointer to the 1st element

Address	value(**x**) = value(&**x**[0])
Size	sizeof(**x**) = 5 * sizeof(int)
Type	int (*)



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

address replication

*(**x**+0) ≡ **x**[0]

(**x**+0) ≡ &**x**[0]

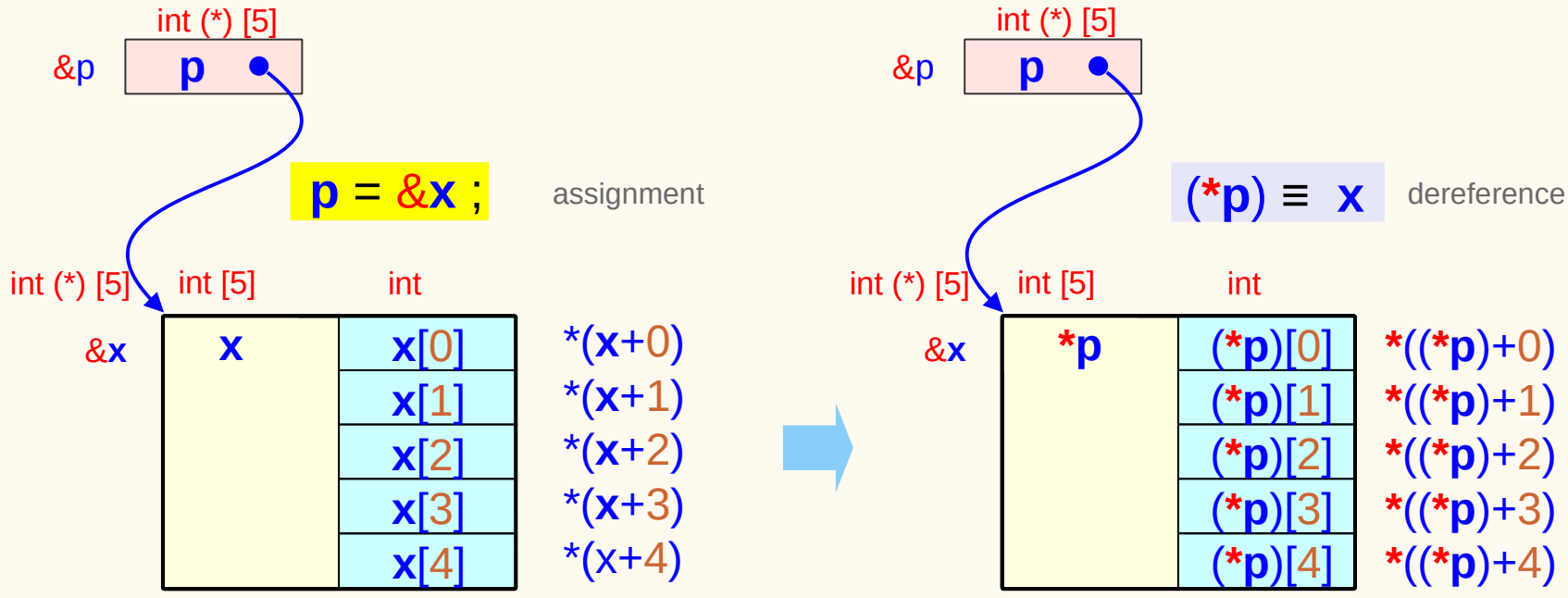
equivalence relations

Using a **1-d** array pointer **p**

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

```
int  (*p) [5]  = &x ;
```

referencing the outside of an array x when an array is referenced

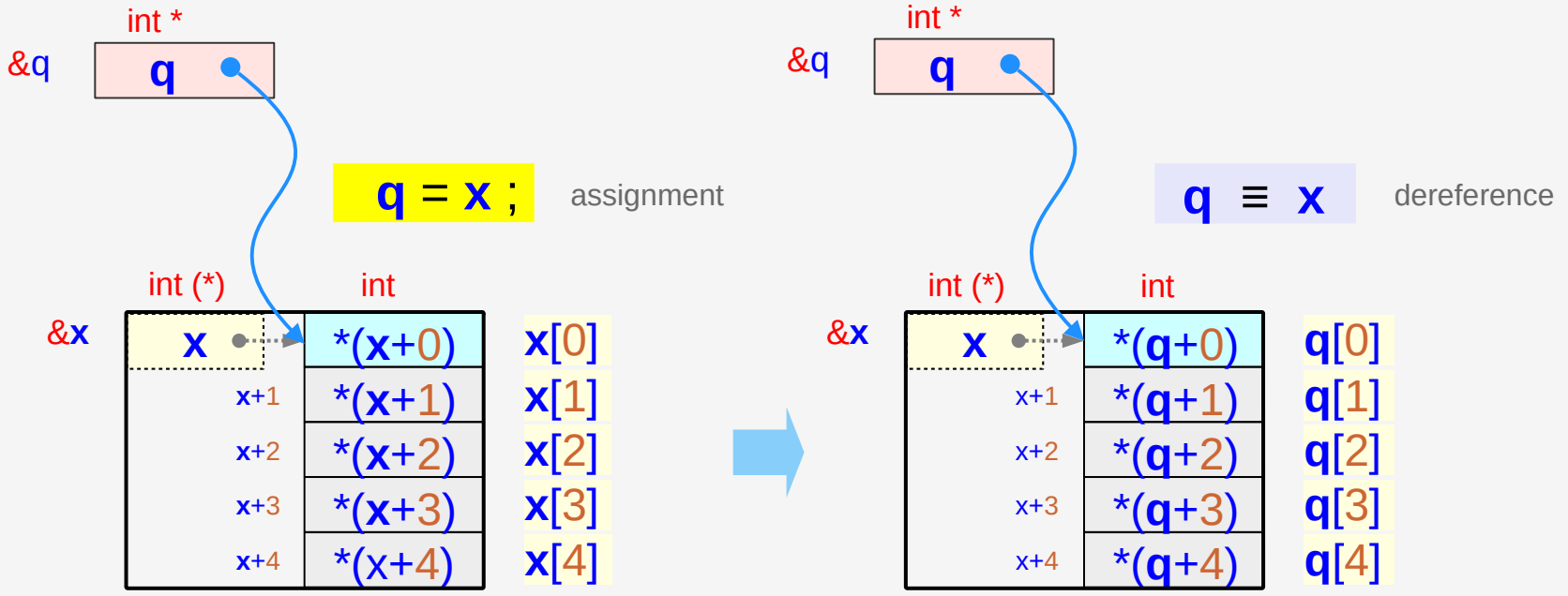


Using a **0-d** array pointer **q**

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

```
int (* q) = x ;
```

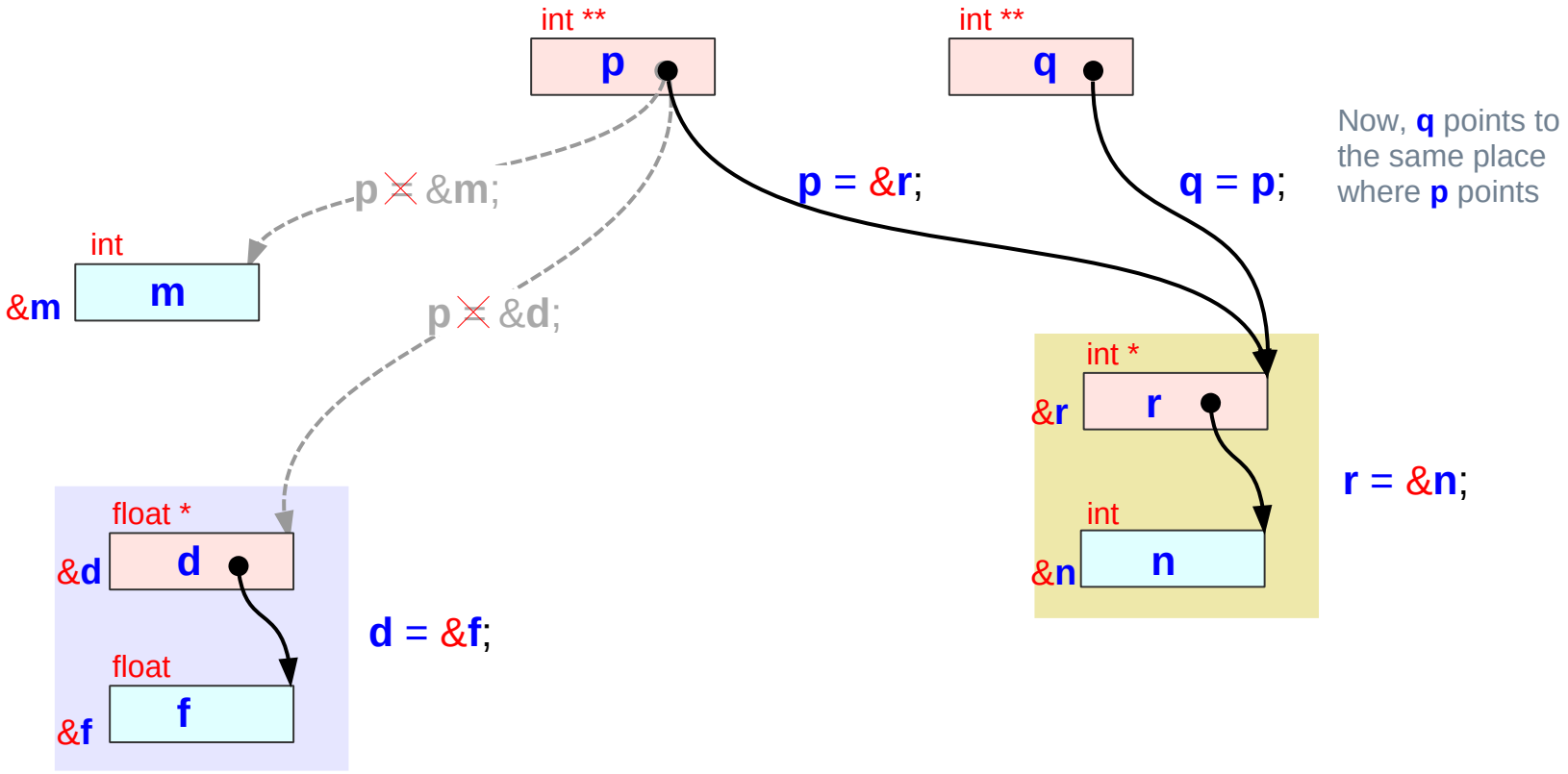
referencing the inside of an array x when an element of an array is referenced



Double pointer variable assignments

```
int  **p, **q, *r, m, n ;
```

```
float  *d, f ;
```

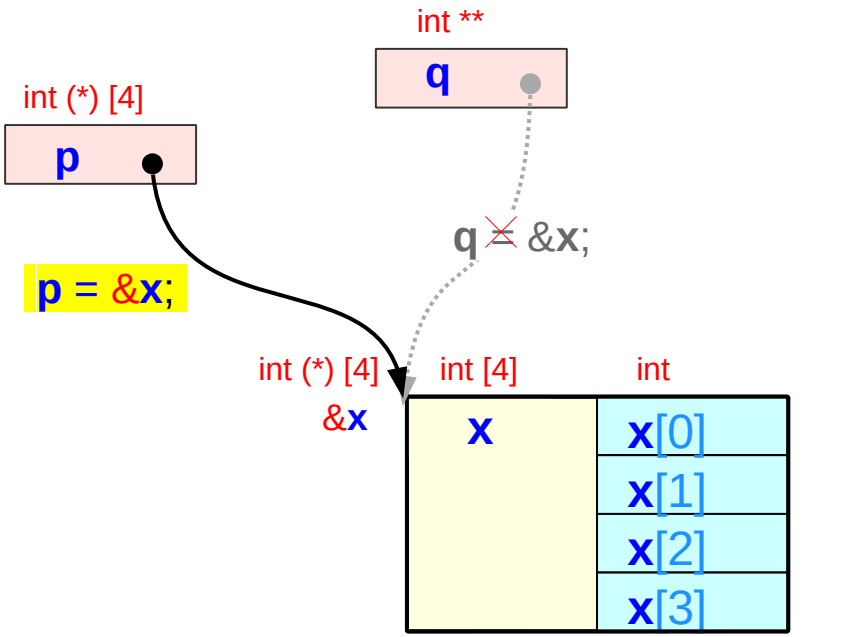


Double pointer variable assignments

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

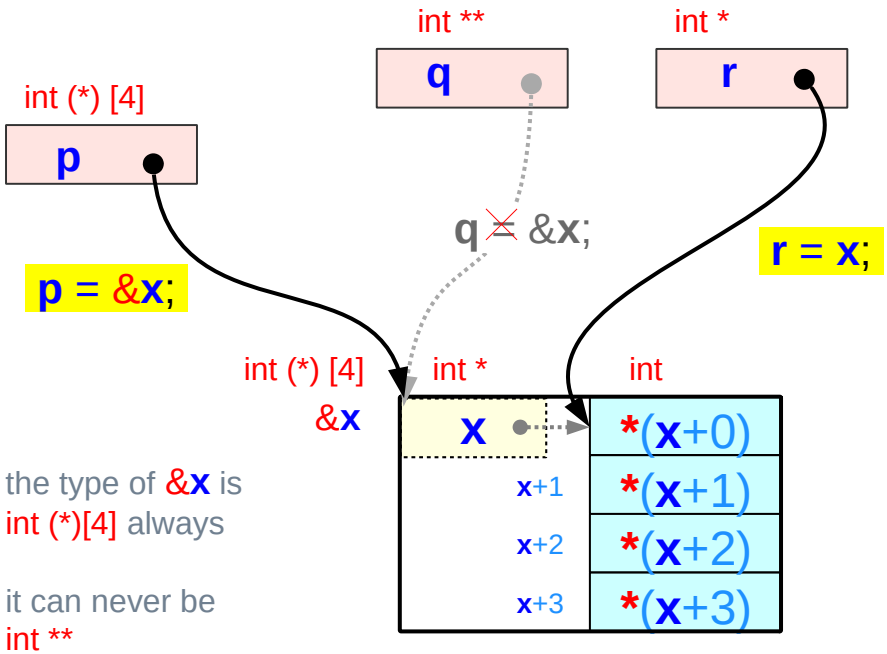
```
int  ** q ;
```

```
int  *r ;
```



when an array **x** is referenced,
x has **outside array type**
int [4]

thus, a pointer to **x** has
int (*) [4] type



the type of **&x** is
int (*)[4] always

it can never be
int **

when an element of an array **x**
is referenced, **x** has **inside**
array type int (*)

within an array **x**, the array type
int [4] can be relaxed to int (*)

.

- **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;

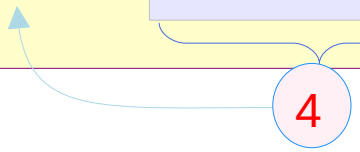
all elements with zero

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

~~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} };
```

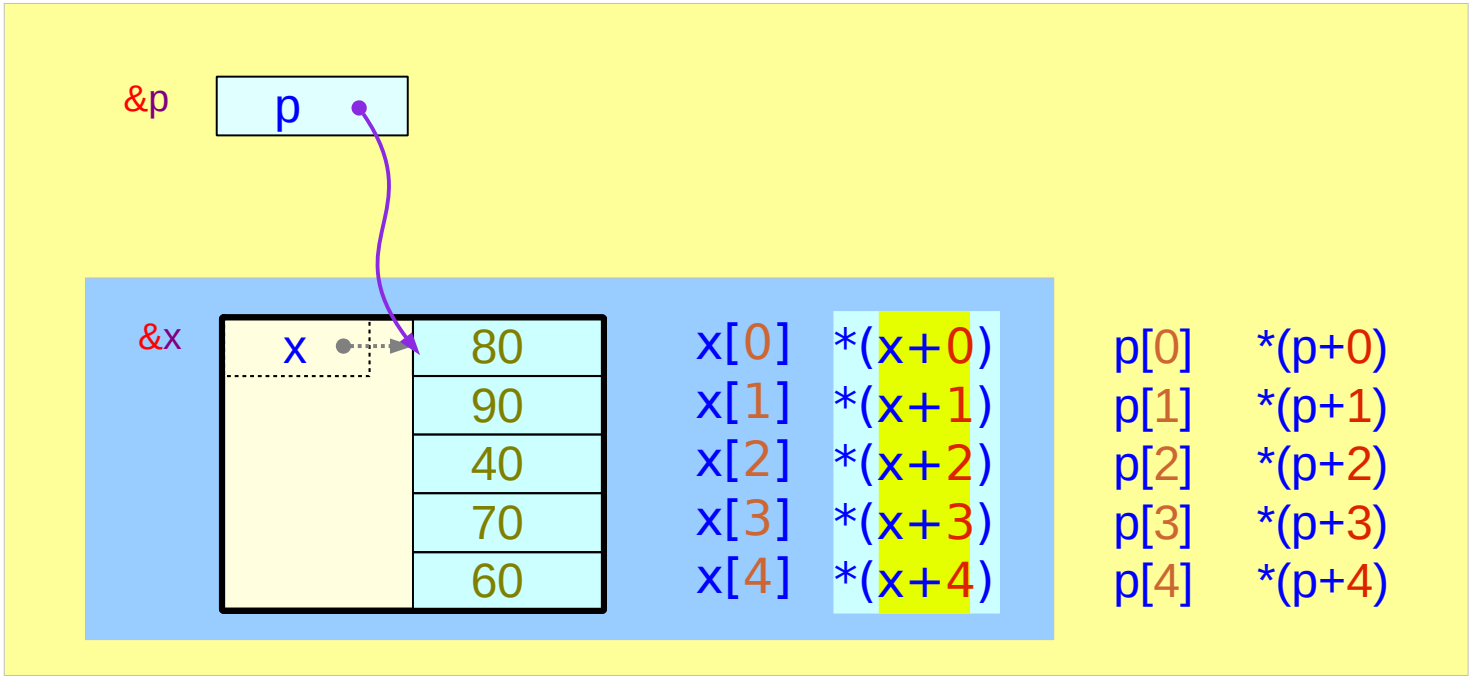
Only the first dimension
can be unsized

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

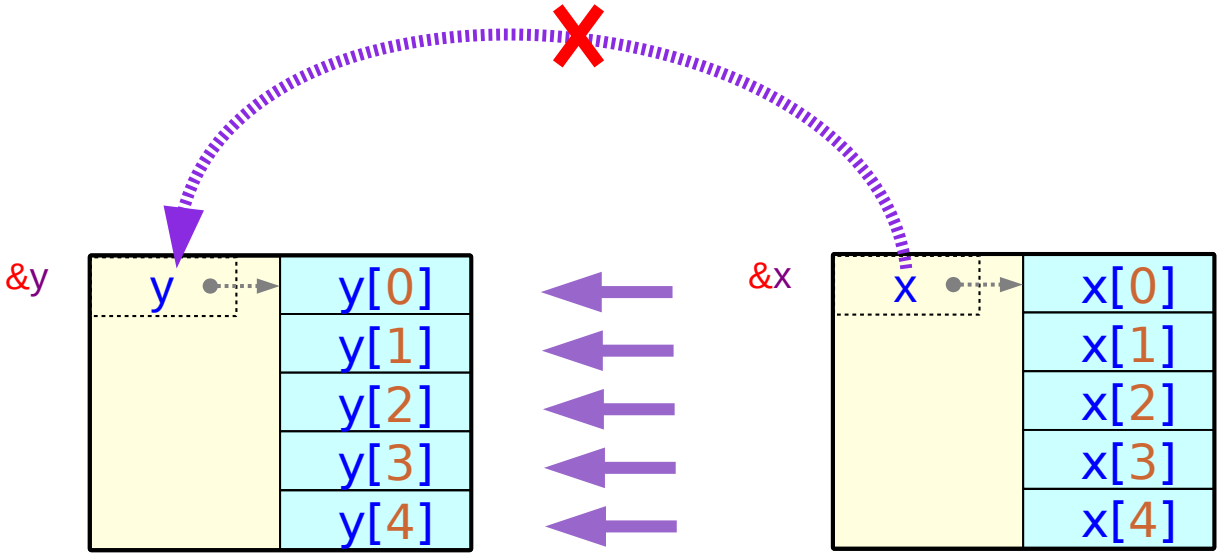
p is a variable
can point to other addresses



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)



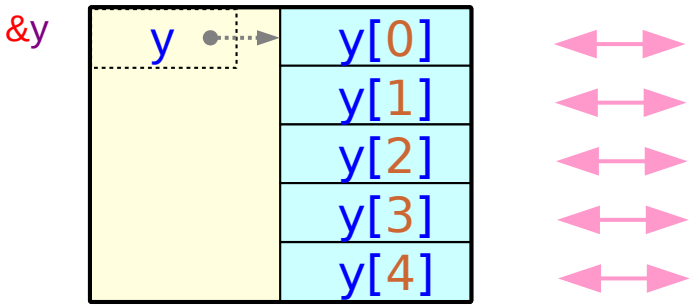

```
for (i=0; i<5; ++i)
    y[i] = x[i];
```

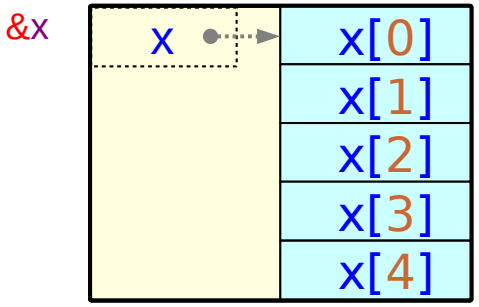
must copy each element

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]);
```





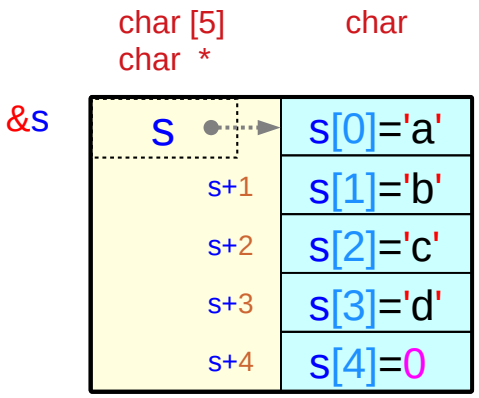

```
EQ=1;
for (i=0; i<5; ++i)
      EQ &= (y[i] == x[i]);
```

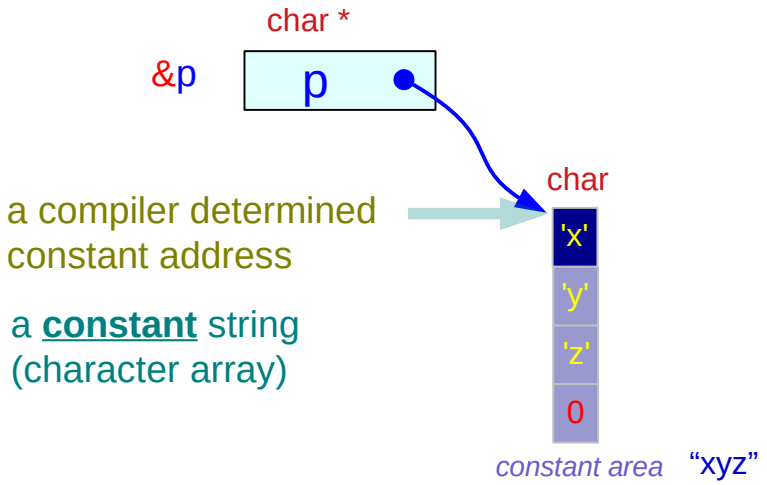
must compare each element

- **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

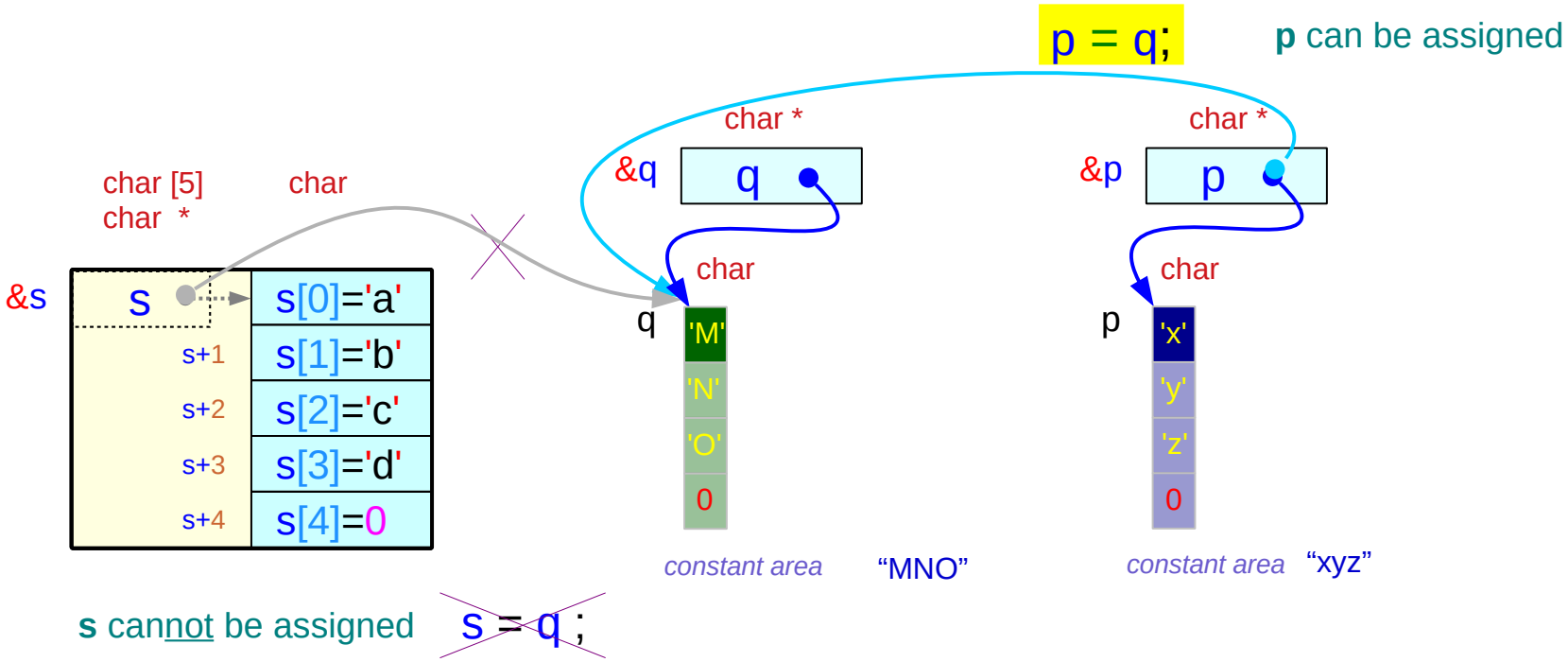
```
*s = 'm' ;
s[0] = 'm' ;
```

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

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

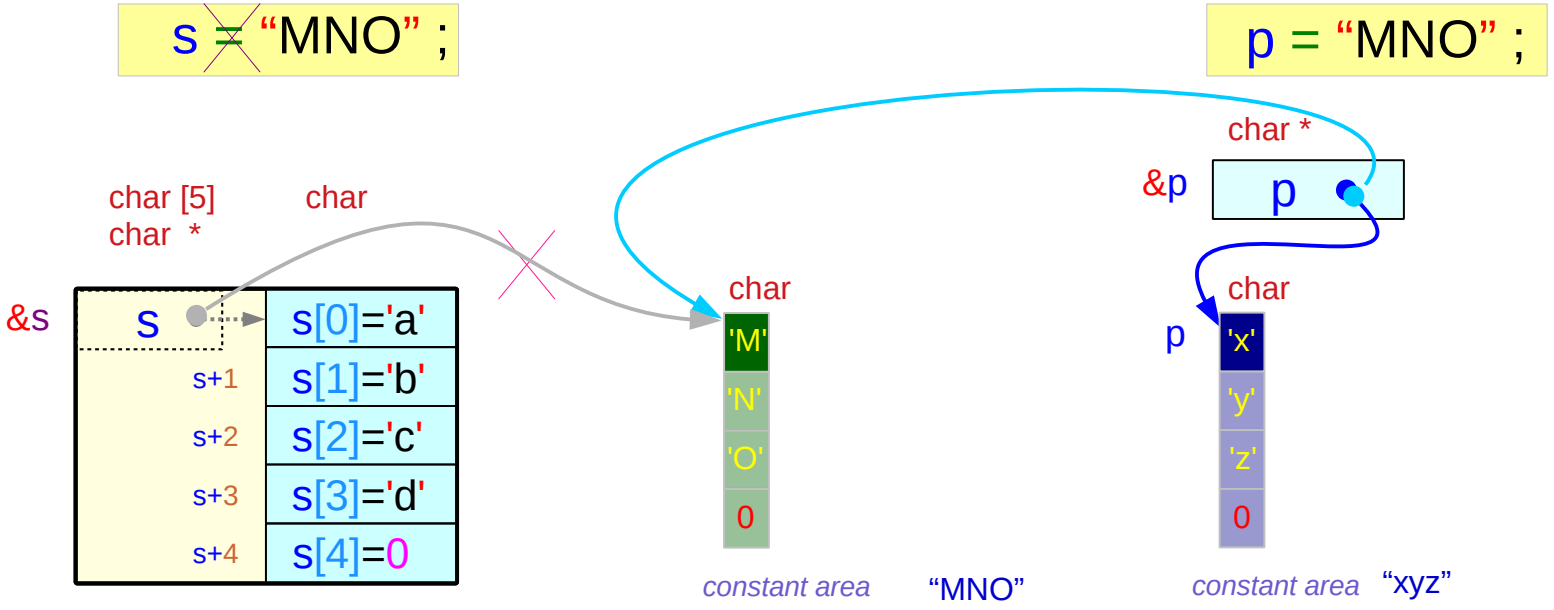
Initialized character arrays and pointers (2)

```
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” ;
```

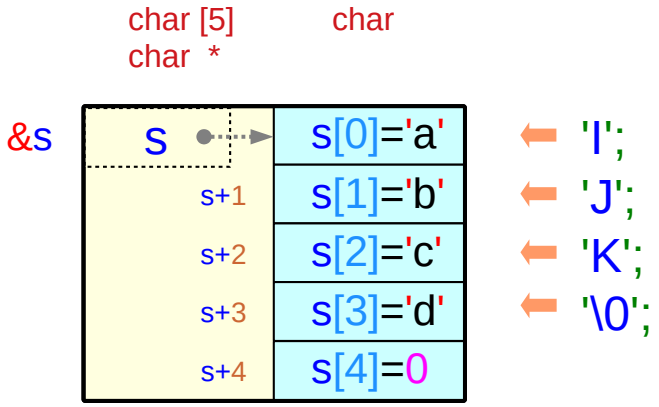


Copying a string

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

char  *p = "xyz" ;
```

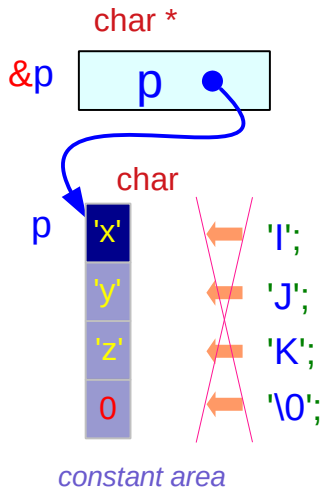
strcpy (**s**, "IJK");

s points to non-constant string



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

p: points to a constant string



Character arrays and a string pointer

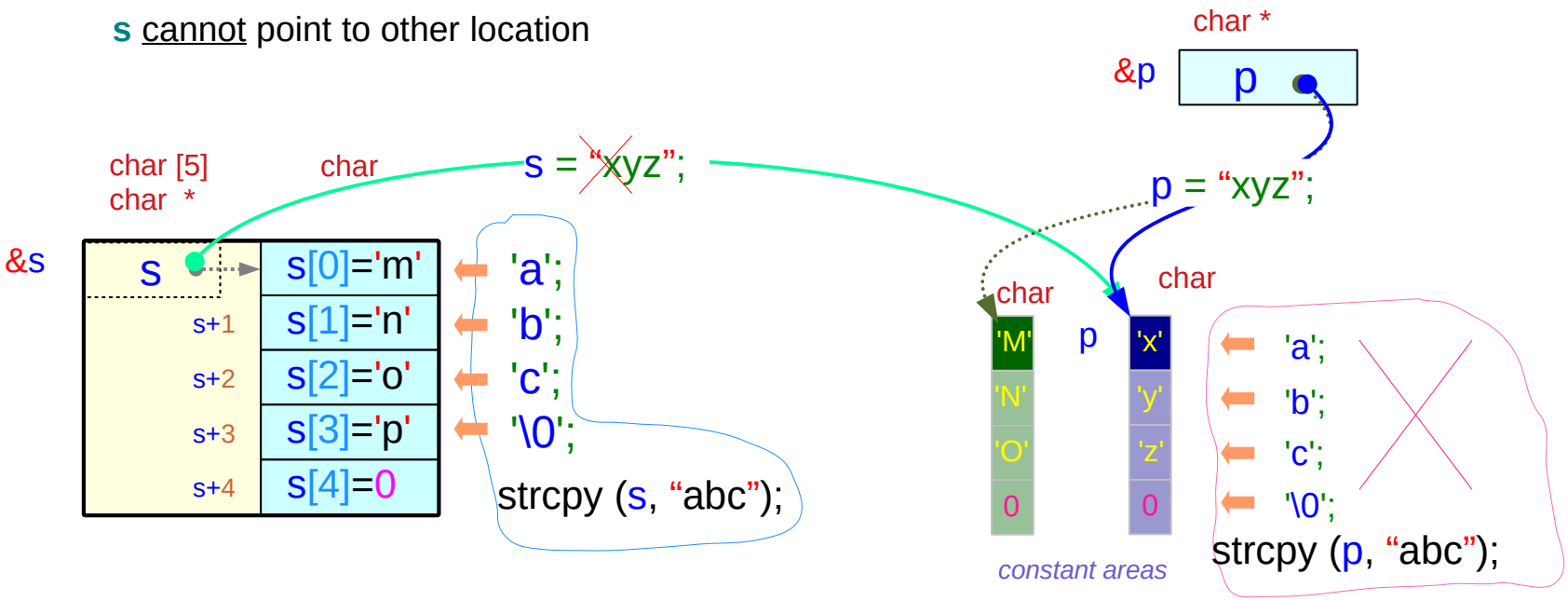
```
char s [5] = “mnop” ;
const char  *p = “MNO” ;
```

```
s = “xyz”;
p = “xyz”;
```

```
strcpy (s, “abc”);
strcpy (p, “abc”);
```

char * const s

s cannot point to other location

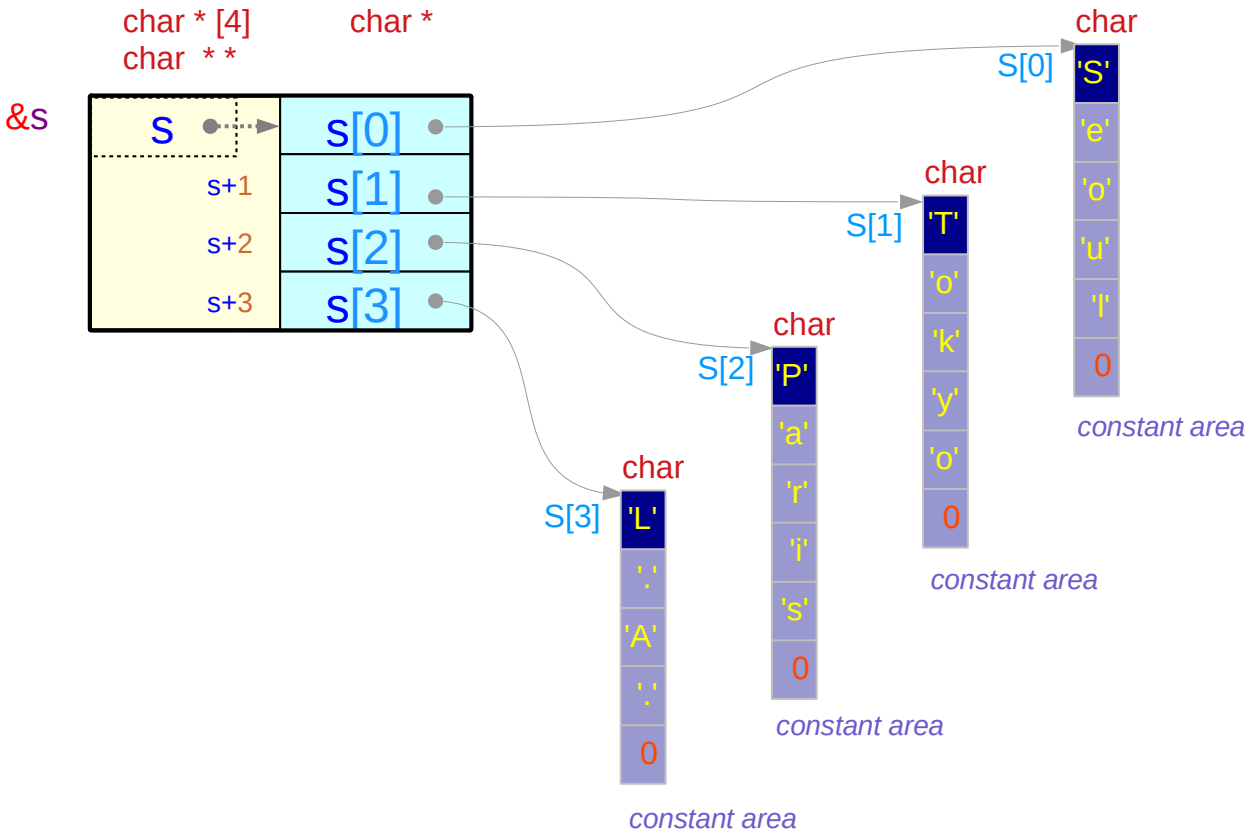


const char * p

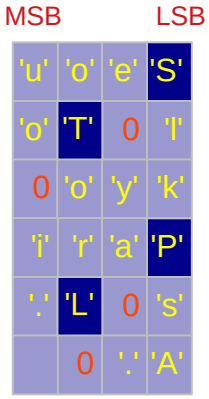
p points to a string constant
which cannot be changed

Arrays of Pointers

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



A possible memory layout
(little endian system)

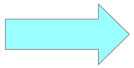


- **Unsized array notations for 1-d arrays**

Unsized array notation **x[ ]**

1. An array definition with initializers

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

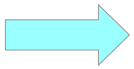


```
int x [3] ;
```

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

2. A formal parameter definition in a function

```
func( int x [ ] ) { … }
```



```
int (*x)
```

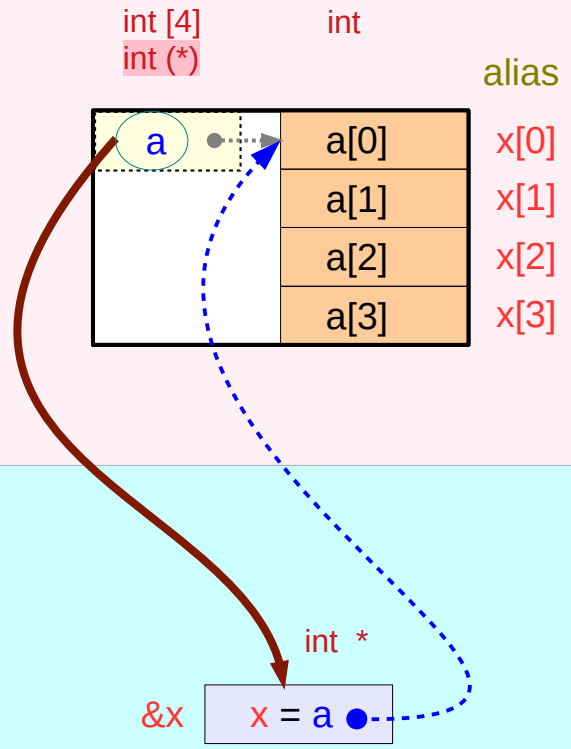
compatible

```
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) ) {
        …
}
```

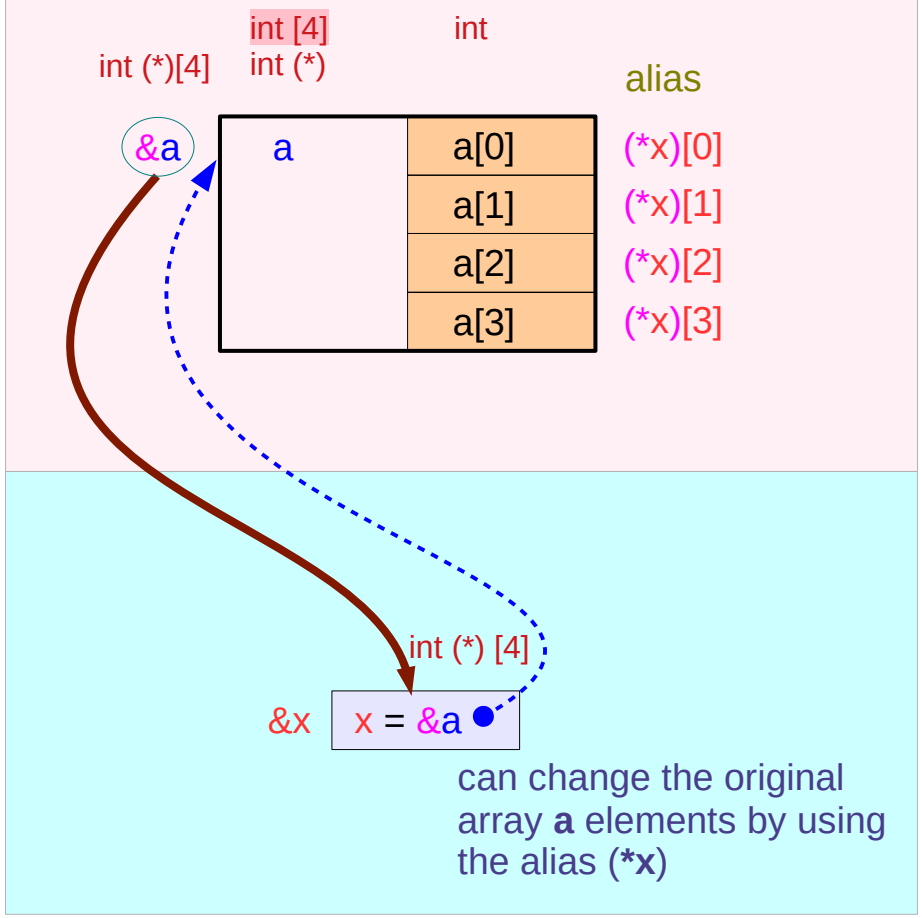


can change the original array **a** elements by the alias **x**

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] ) {
        …
}
```



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**

Passing an individual element by value

```
int a[4] ;

func( a );

func( int x [4] ) {  … }

func( int x [ ] ) {  … }

func( int (*x) ) {  … }
```

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

func( c );

func( int c [3][4] ) {  … }

func( int x [ ][4] ) {  … }

func( int (*x)[4] ) {  … }
```

Passing an individual element by value

```
int a[4] ;

func( &a );

func( int (*x) [4] ) {  … }

func( int x[ ] [4] ) {  … }
```

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

func( &c );

func( int (*x) [3][4] ) {  … }

func( int x[ ] [3][4] ) {  … }
```

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

Unsized 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

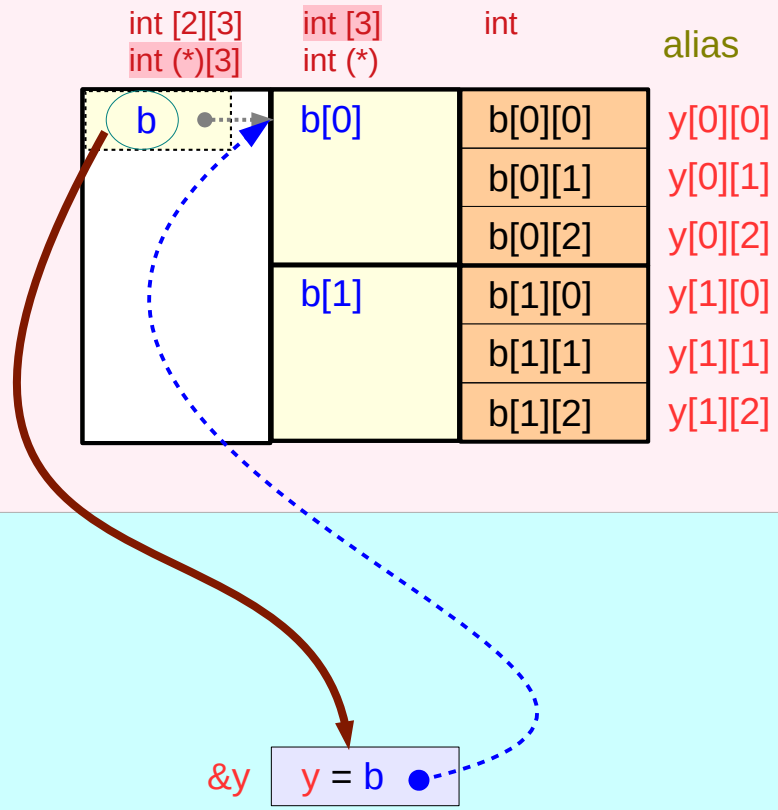
func(int x[ ][3]) { … } → int (*x)[3]

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] ) {
        …
}
```

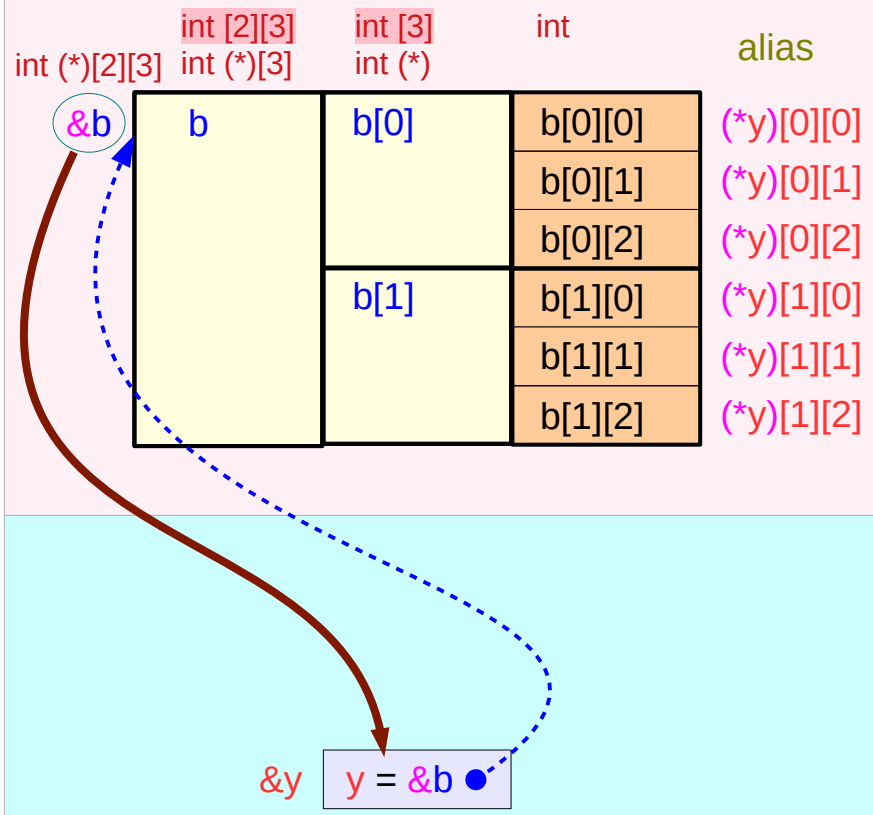


can change the original array **b** elements by using the alias **y**

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

type alias int_type variable i

```
typedef  int  int_type ;
```

```
int_type  i;
```

⟺

```
int  i ;
```

type alias iptr_type variable ip

```
typedef  int *  iptr_type ;
```

```
iptr_type  ip;
```

⟺

```
int *  ip ;
```

type alias arr_type variable a

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

```
arr_type  a;
```

⟺

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

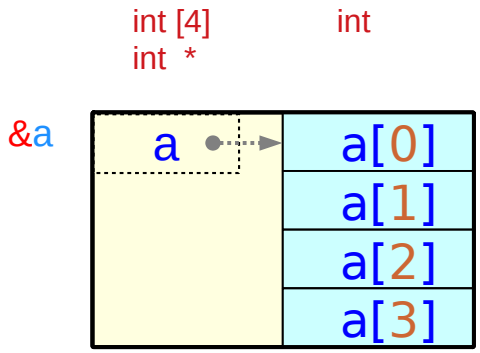
Array Type Definition

```
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;
```

≡≡

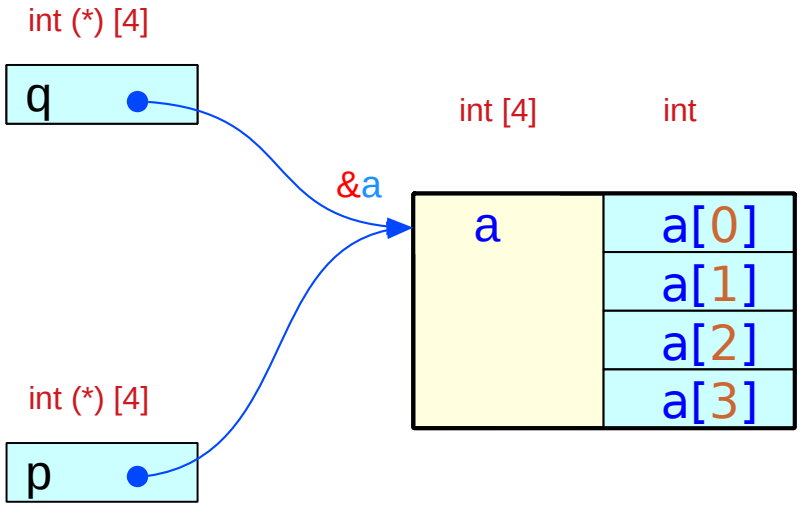
```
int (* p )[4] ;
```

Pointer to Array Type Assignment

```
typedef int arr_type [4] ;

arr_type    a, *q ;
```

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

arr_ptr     p;
```

```
q = &a ;
```



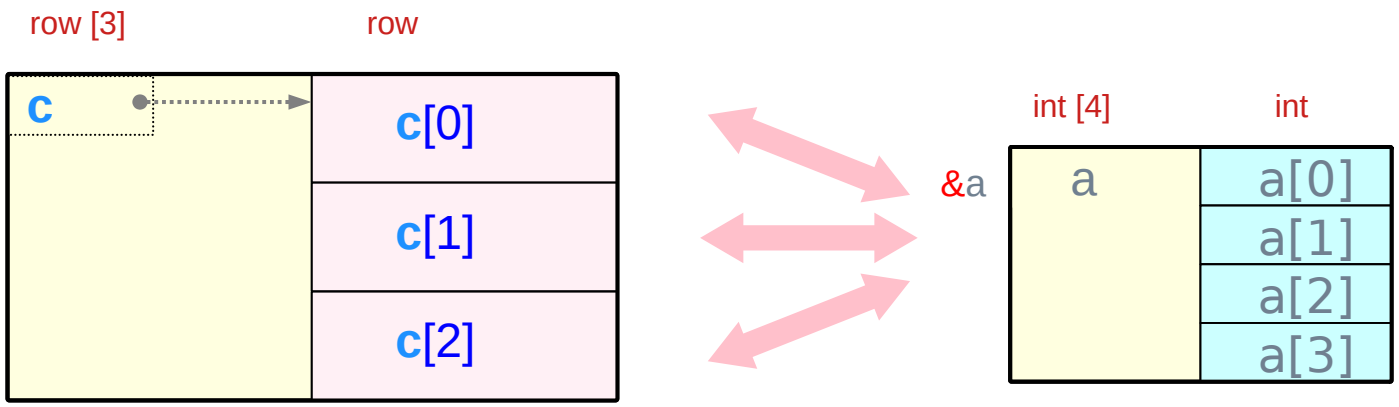

```
p = &a ;  or
p = q ;
```

Nested array declared explicitly

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

≡

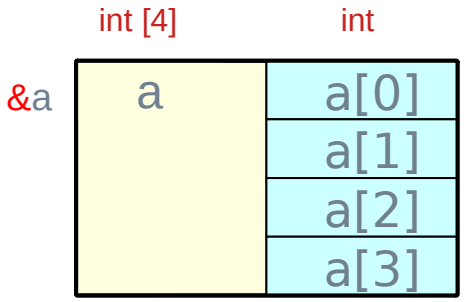
```
int c [3] [4] ;
```



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

Nested array declared explicitly

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



a ➡ c[0]

a ➡ c[1]

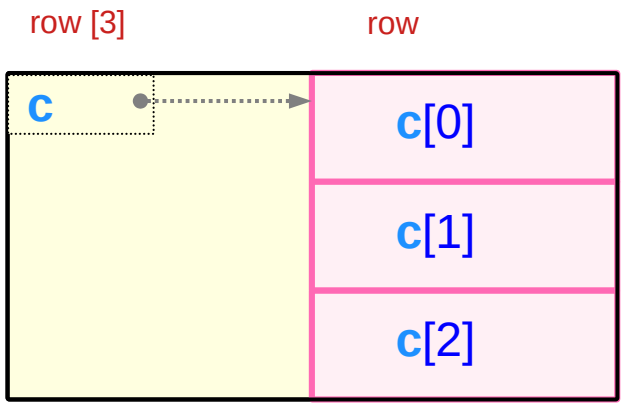
a ➡ c[0]

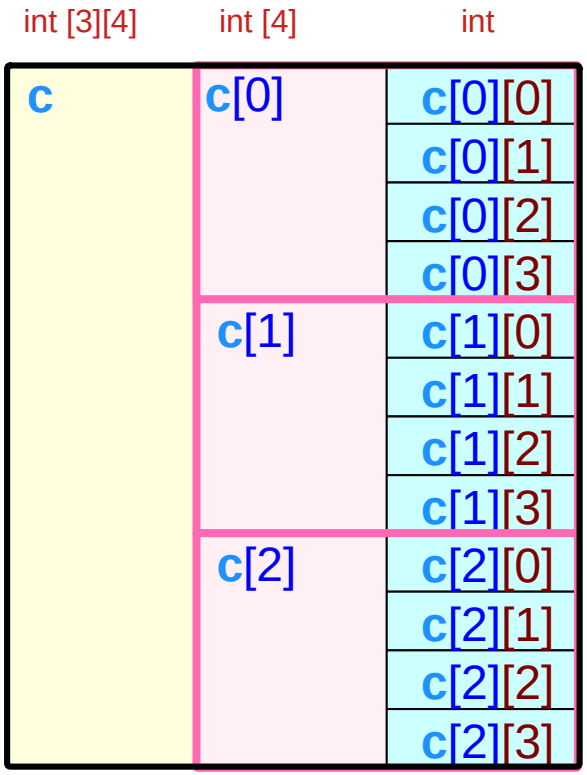
int [4]	int
c[0]	c[0][0]
	c[0][1]
	c[0][2]
	c[0][3]
c[1]	c[1][0]
	c[1][1]
	c[1][2]
	c[1][3]
c[2]	c[2][0]
	c[2][1]
	c[2][2]
	c[2][3]

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

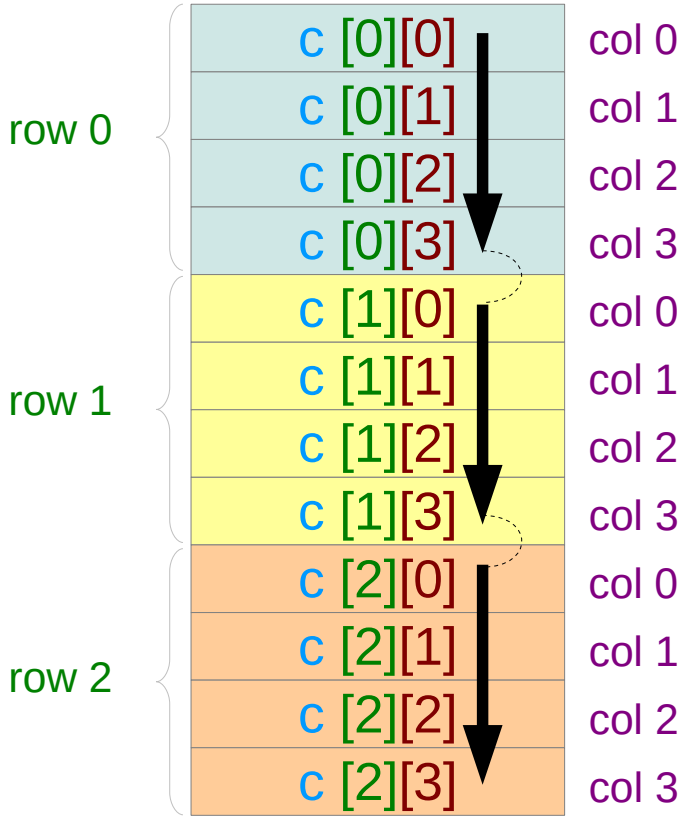
```
int c [3][4];
```

A matrix view

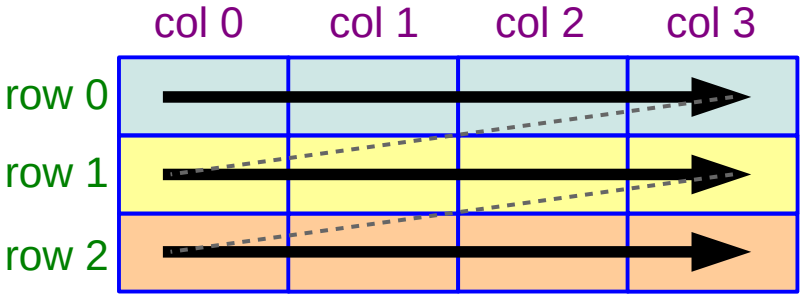
	col 0	col 1	col 2	col 3
row 0	c [0][0]	c [0][1]	c [0][2]	c [0][3]
row 1	c [1][0]	c [1][1]	c [1][2]	c [1][3]
row 2	c [2][0]	c [2][1]	c [2][2]	c [2][3]

2-d array stored as a linear array

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



row major order



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

X[i] ≡ *(X+i)

X[j] ≡ *(X+j)

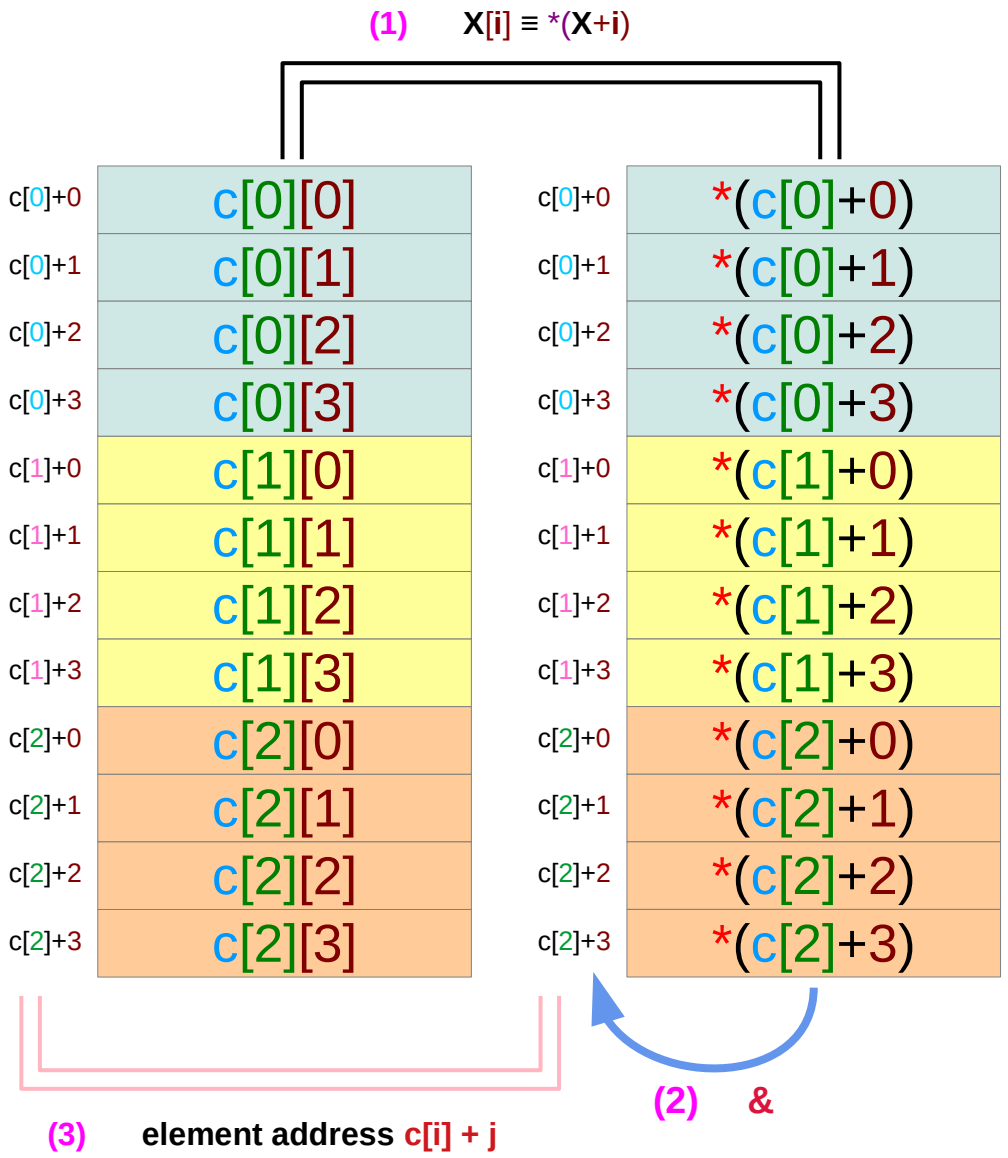
let **X ≡ c[i]** (1)

c[i][j] ≡ *(c[i]+j)

(2)

&c[i][j] ≡ c[i]+j (3)

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



Row address **c[i]**

c[i][j] ≡ *(c[i]+j)

&c[i][j] ≡ c[i]+j

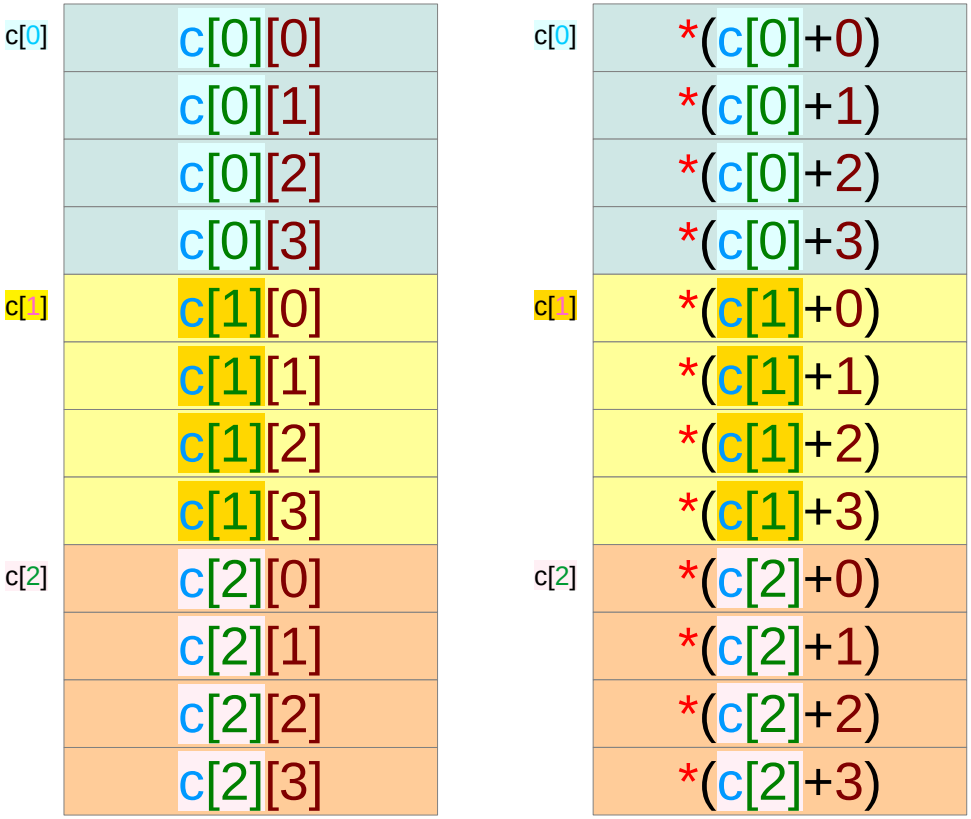
the address of **c[i][j]** is **c[i]+j**

let **j = 0**

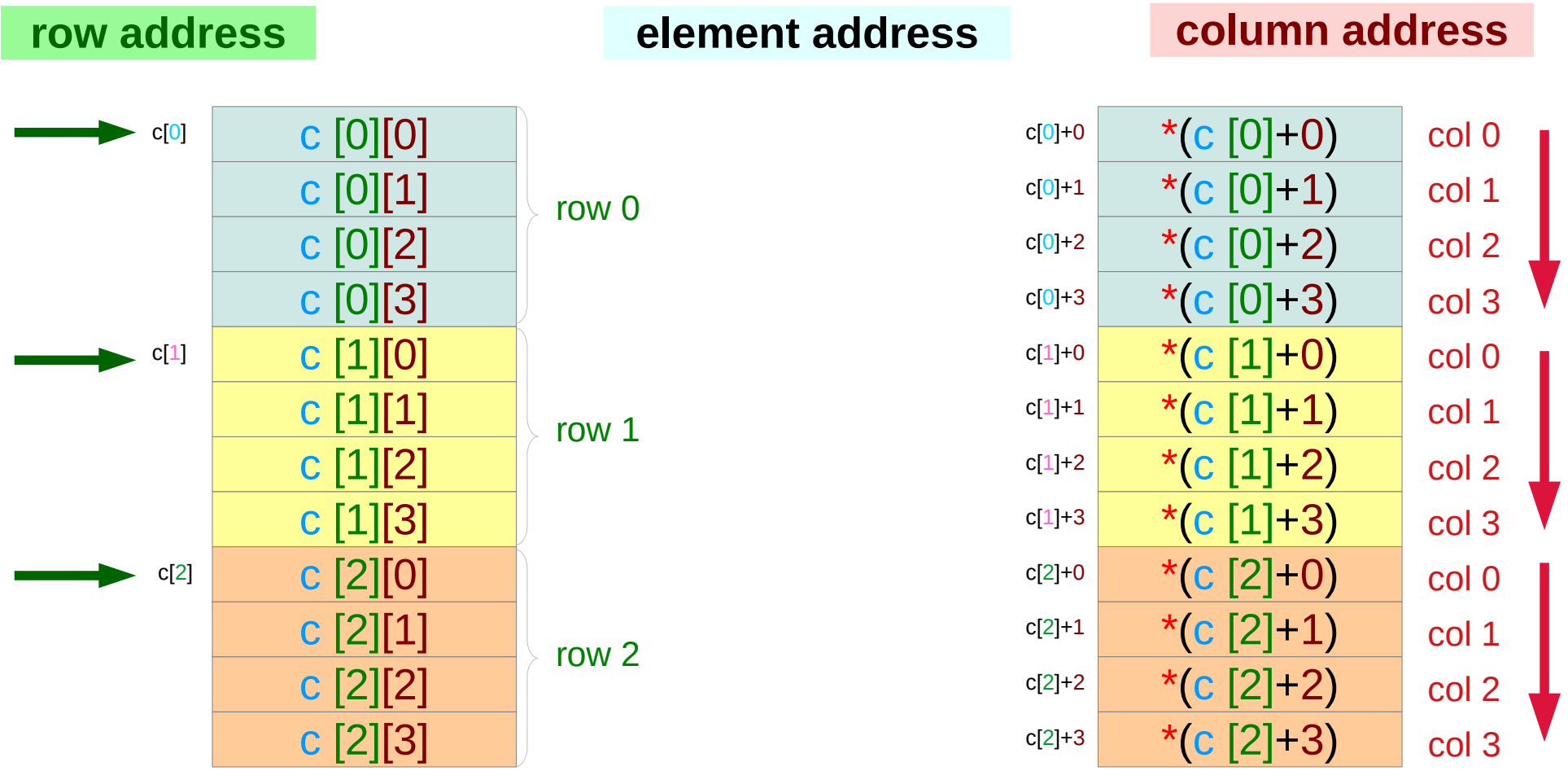
c[i][0] ≡ *c[i]

&c[i][0] ≡ c[i]

row address : the address of the 1st element of each row



Row Address **c[i]** and Column Address **j**



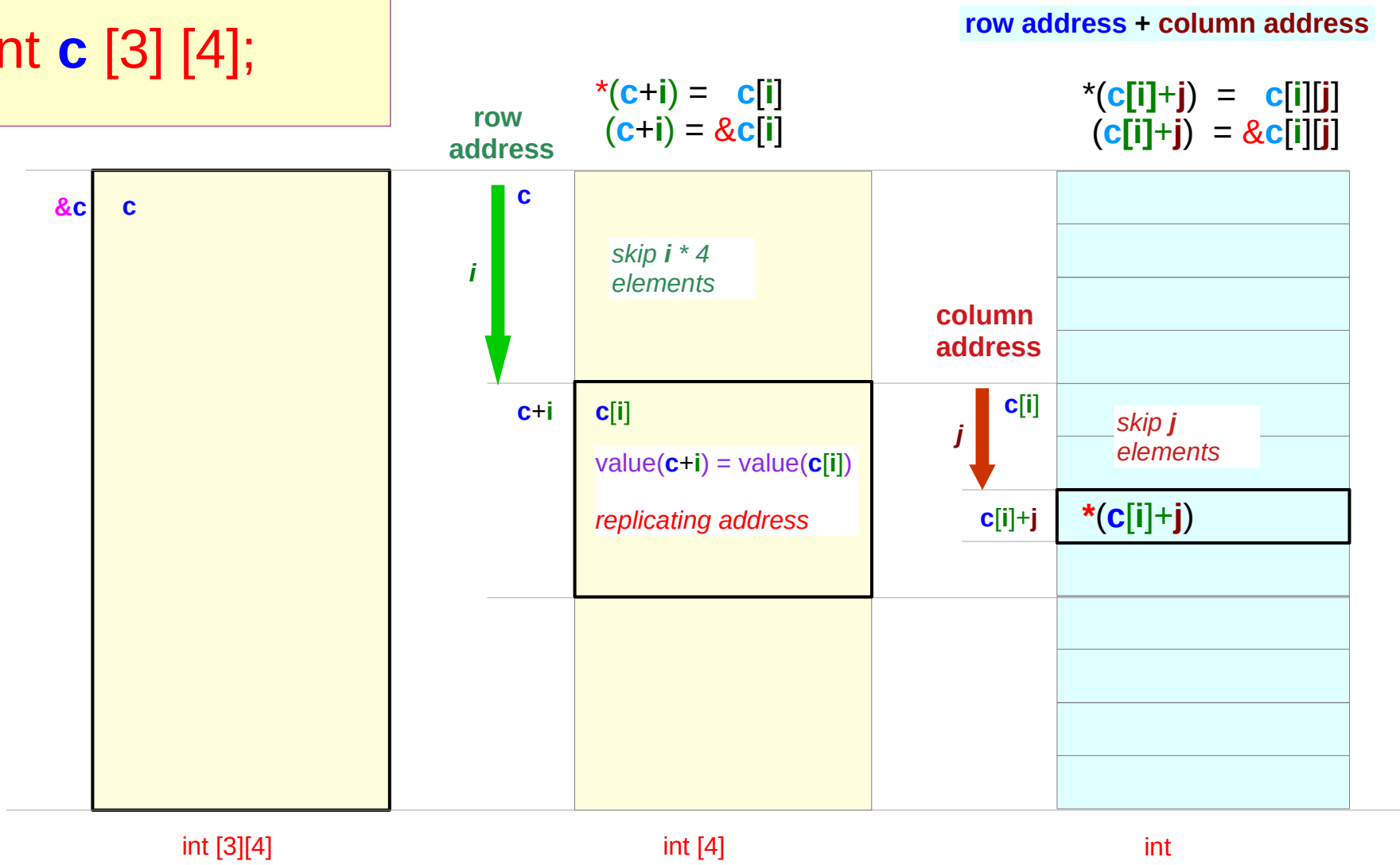
row address
c[i]

row address + column address
c[i] + j

column address
j

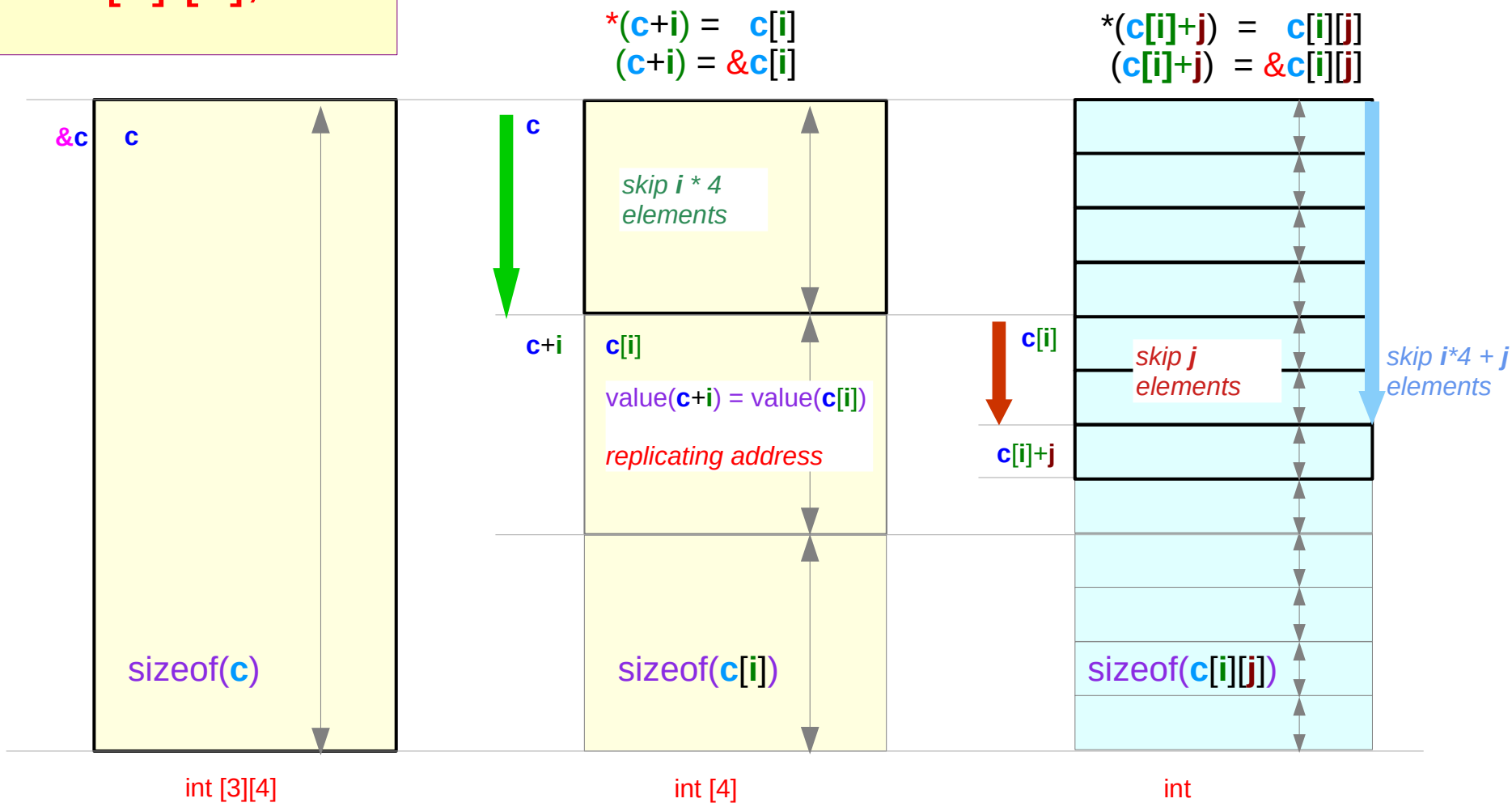
Element address **c[i] + j**

int **c** [3] [4];



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

int **c** [3] [4];



Address of **c[i][j]**

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

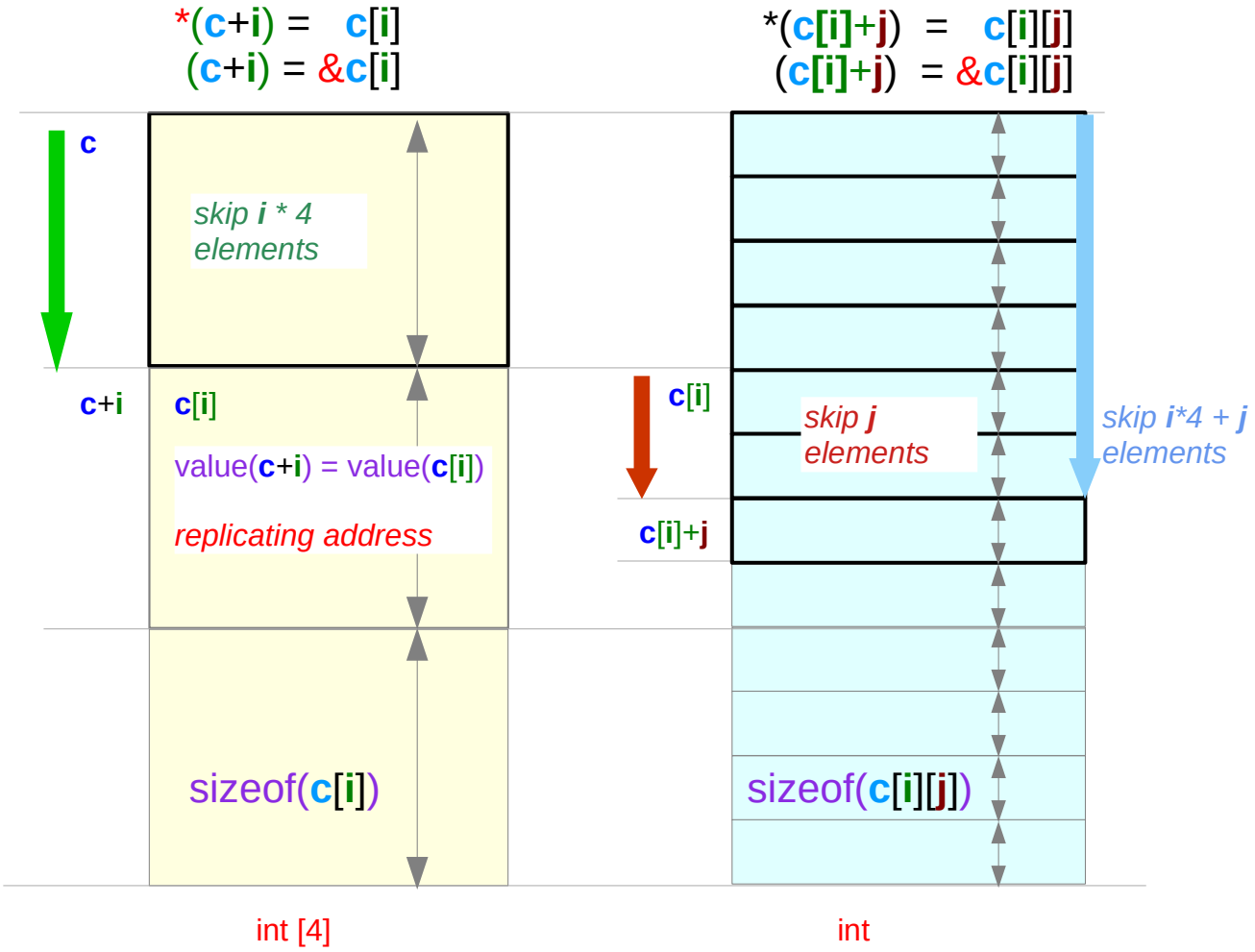
value(c + i) =
value(c) + i * sizeof(*c)

value(c[i] + j) =
value(c[i]) + j * sizeof(*c[i])

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

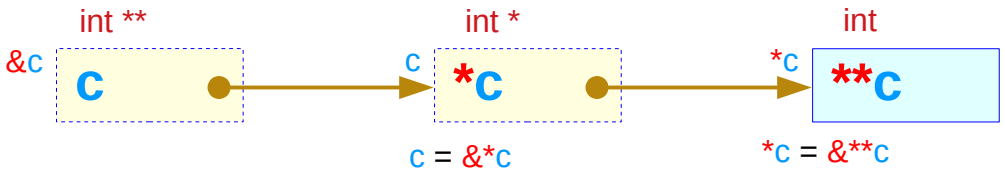
address replication

&c[i][j] = value(c[i] + j)
= value(c[i]) + j * sizeof(*c[i])
= value(c + i) + j * sizeof(*c[i])
= value(c) + i * sizeof(*c)
+ j * sizeof(*c[i])
= value(c) + i * 4 * 4 + j * 4

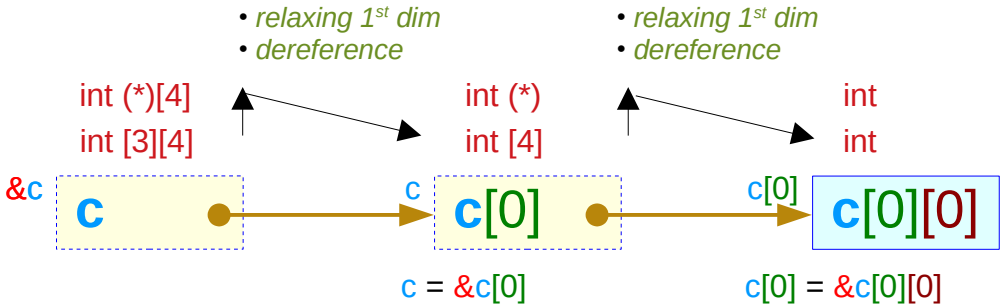


- **Two types of a 2-d array c**
- **Two types of 1-d sub-arrays c[i]**

Chains of dereferences



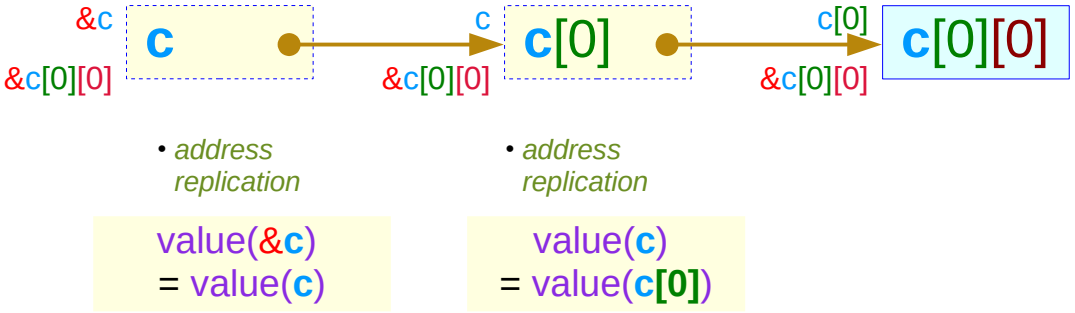
a chain of dereferences using *



virtual array pointer type

abstract data (array) type

a chain of dereferences using [ ]

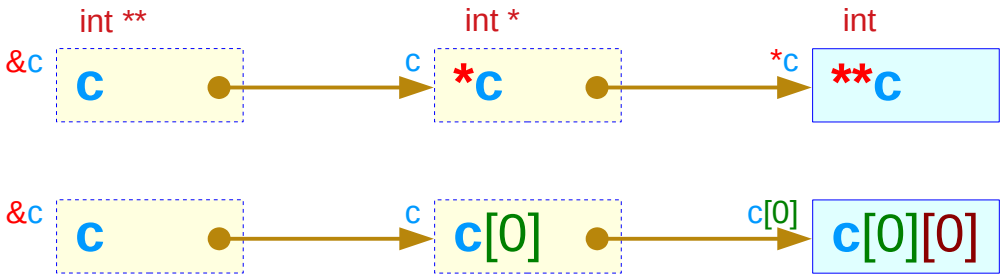


replicating a physical address

conditions for **c** and **c[0]**
to start at the same address **&c[0][0]**

address(pointer) = value(pointer)

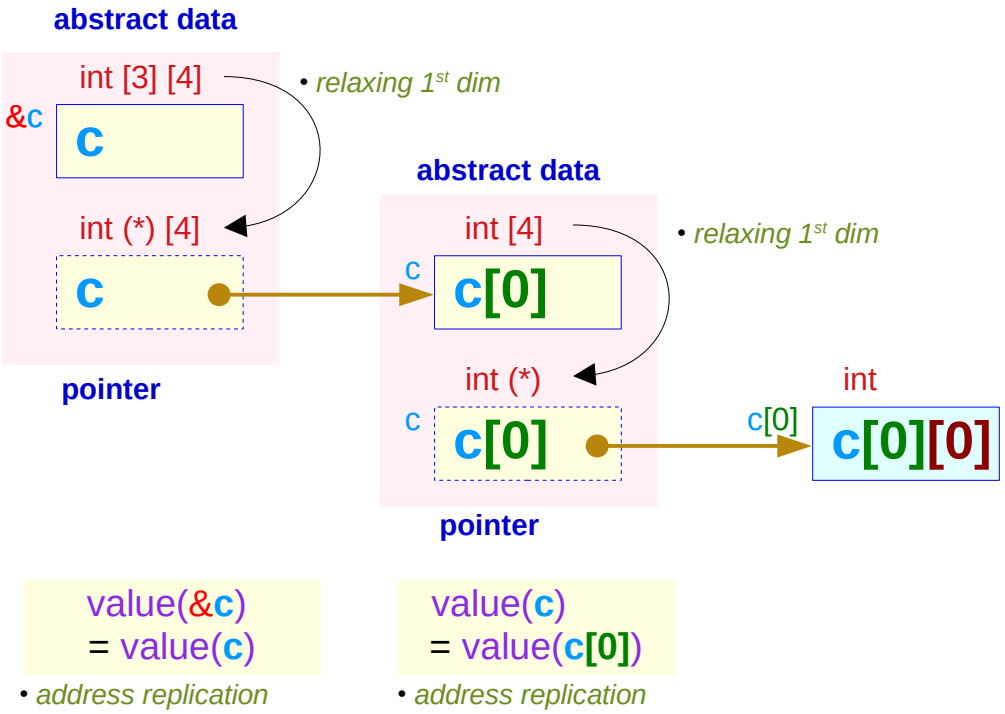
Chains of dereferences with correct types



a chain of dereferences using *

a chain of dereferences using [ ]

with correct types of referencing



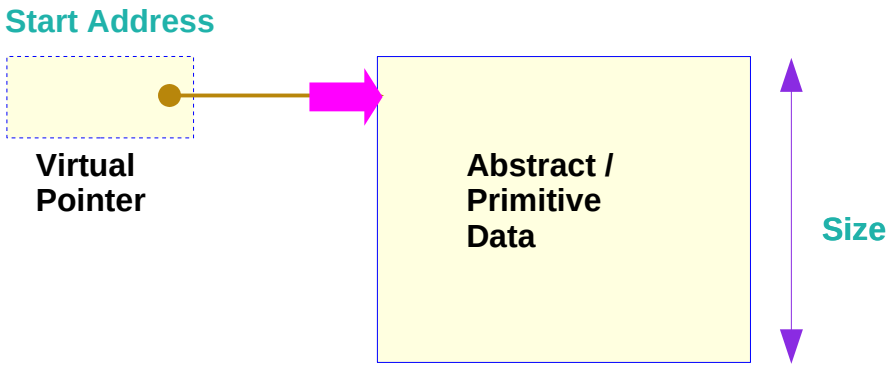
Two types of an array – **c**, **c[i]**, **c[i][j]**

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

		c	c[i]	c[i][j]
abstract / primitive Data	**Size**	**2-d** array int [3][4]	**1-d** array int [4]	integer int
virtual pointer	**Start Address**	**1-d** array pointer int (*)[4]	**0-d** array pointer int (*)	

Two types of an array – pointer to abstract data

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



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

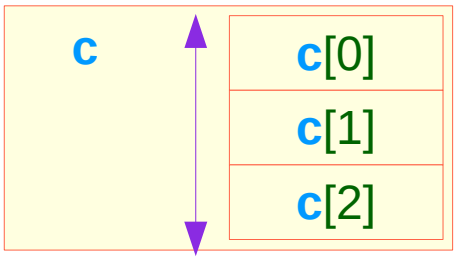
Two types in a **2-d** array **c**

int c[3] [4];

Abstract data c

int [3][4] int [4]

Size



int (c[3]) [4] ;

3 element array **c**

type c[3] ;

C **2-d array**

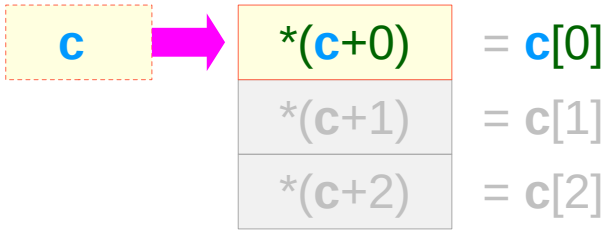
type : int [3][4]

size : 3 * 4 * 4

Pointer c

int (*)[4] int [4]

Address



int (c[3]) [4] ;

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

type c[3] ;

C **1-d array pointer**

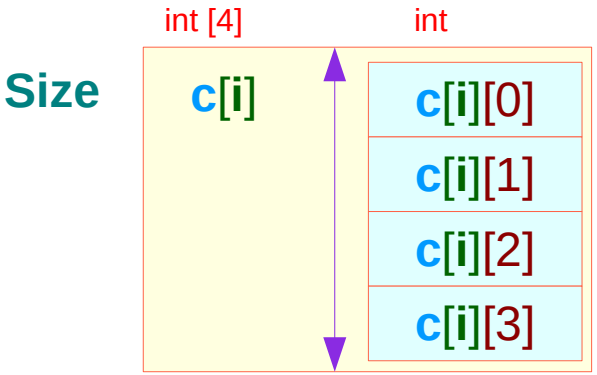
type : int (*)[4]

value : &c[0][0]

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

int c[3] [4];

Abstract data c[i]



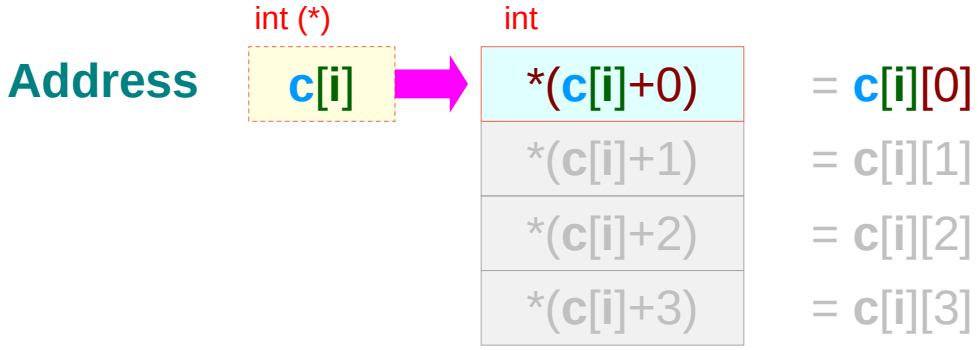
int (c[3]) [4] ;

4 element array **c[i]**

int ***c[i]*** [4] ;

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

Pointer c[i]



int (c[3]) [4] ;

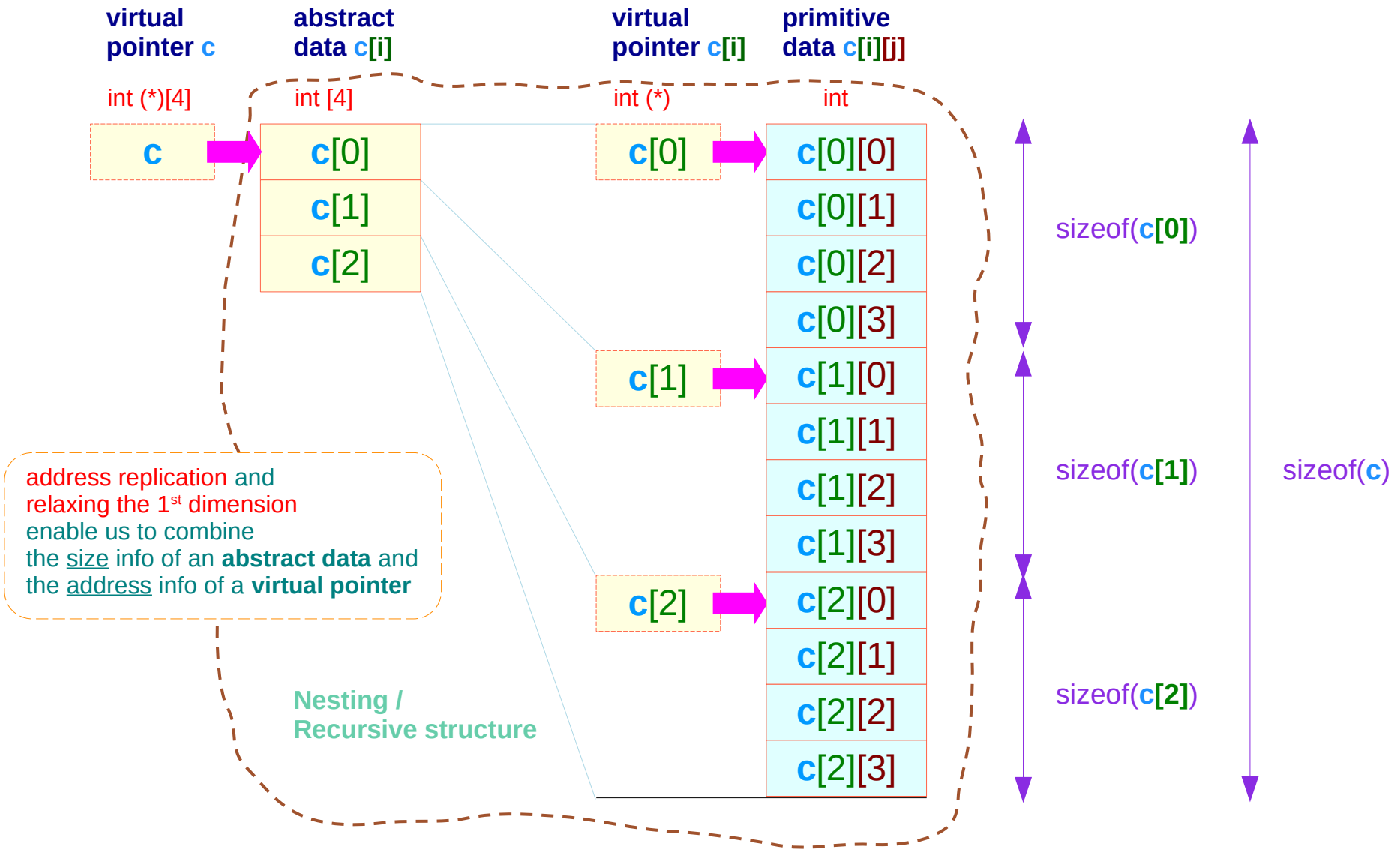
each element **c[i][j]** has
the array type int

int ***c[i]*** [4] ;

c[i] **0-d array pointer**
type : int (*)
value : &c[i][0]

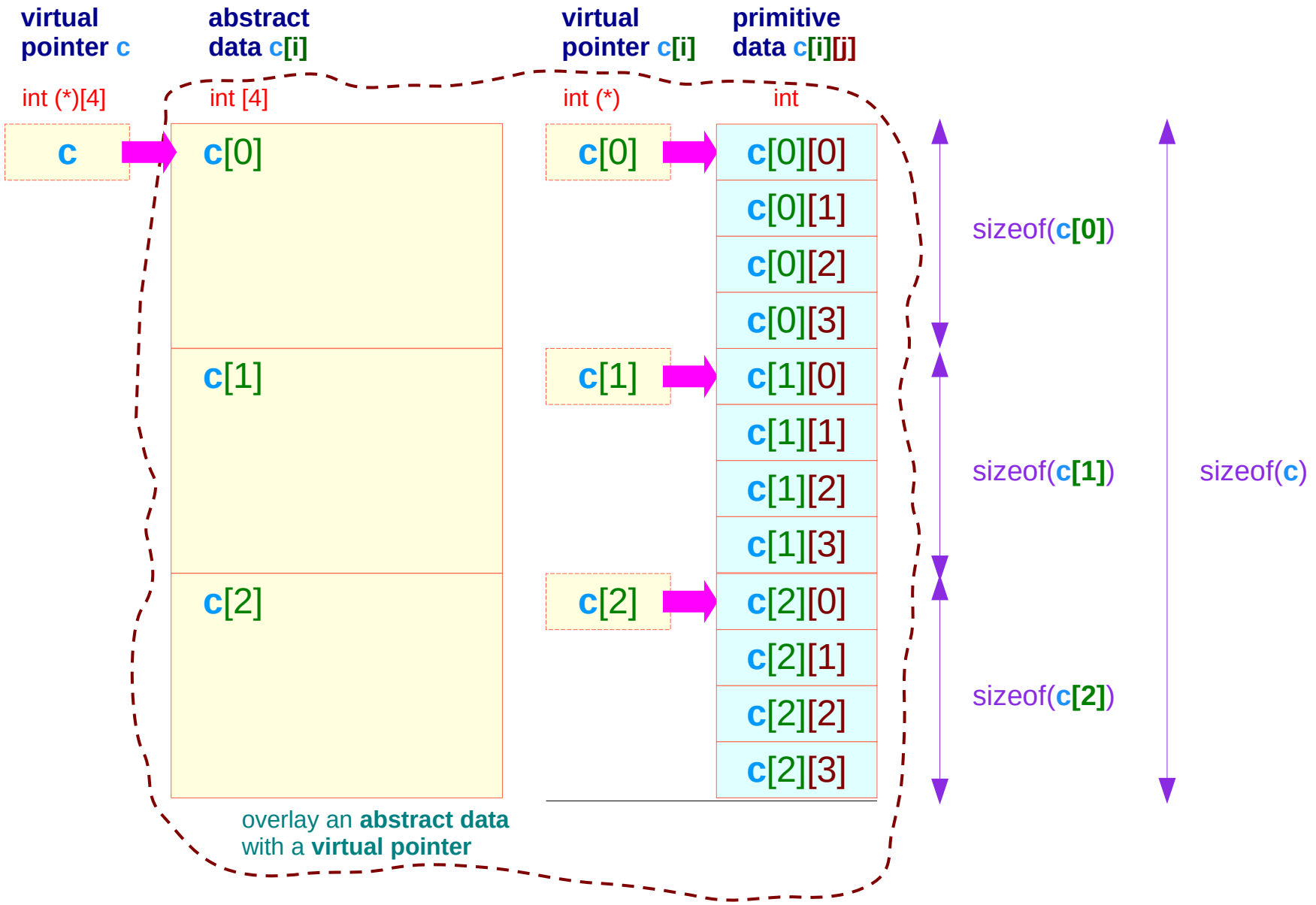
Combining size and address information (1)

int c[3] [4];



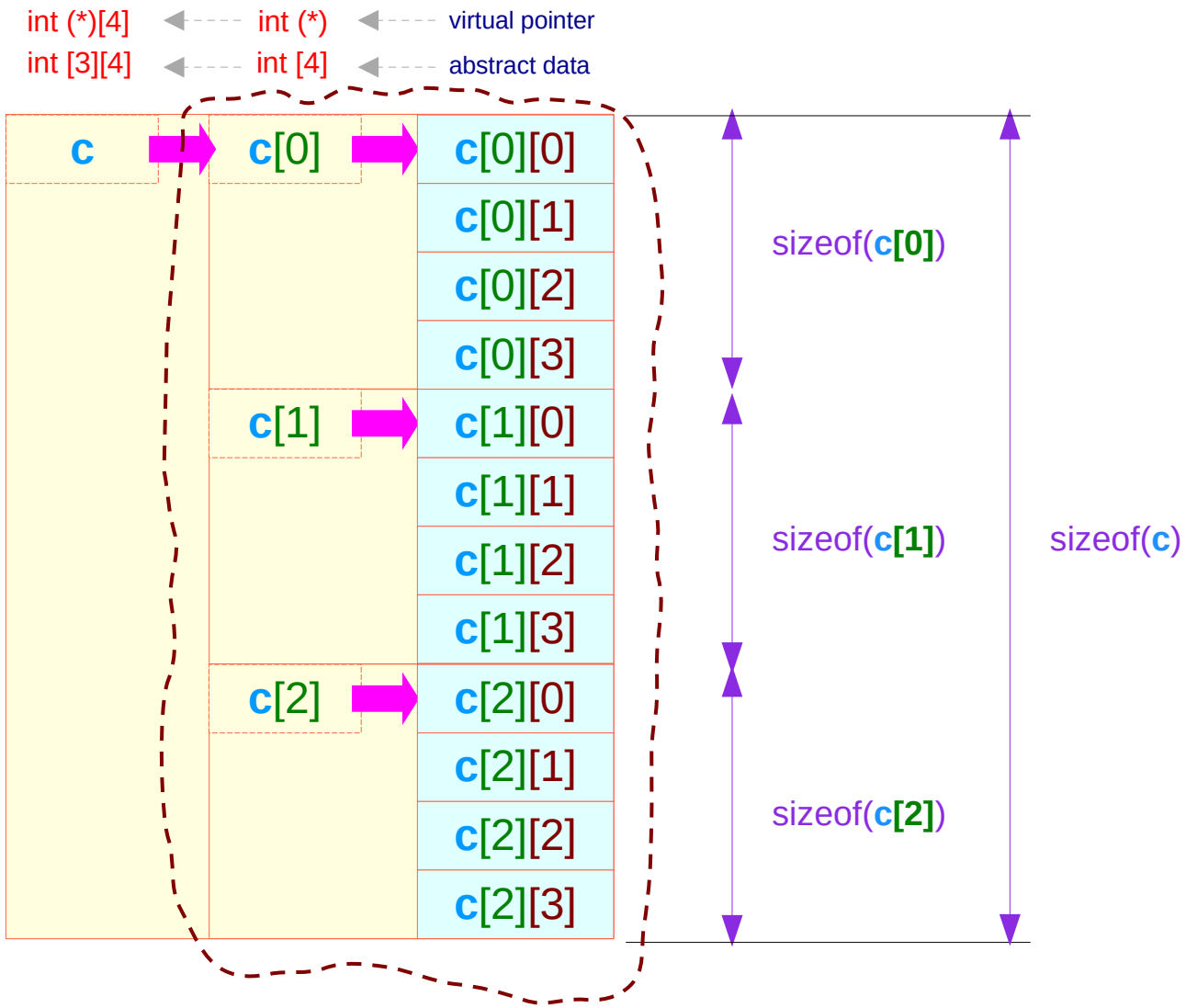
Combining size and address information (2)

int c[3] [4];



Combining size and address information (3)

int c[3] [4];

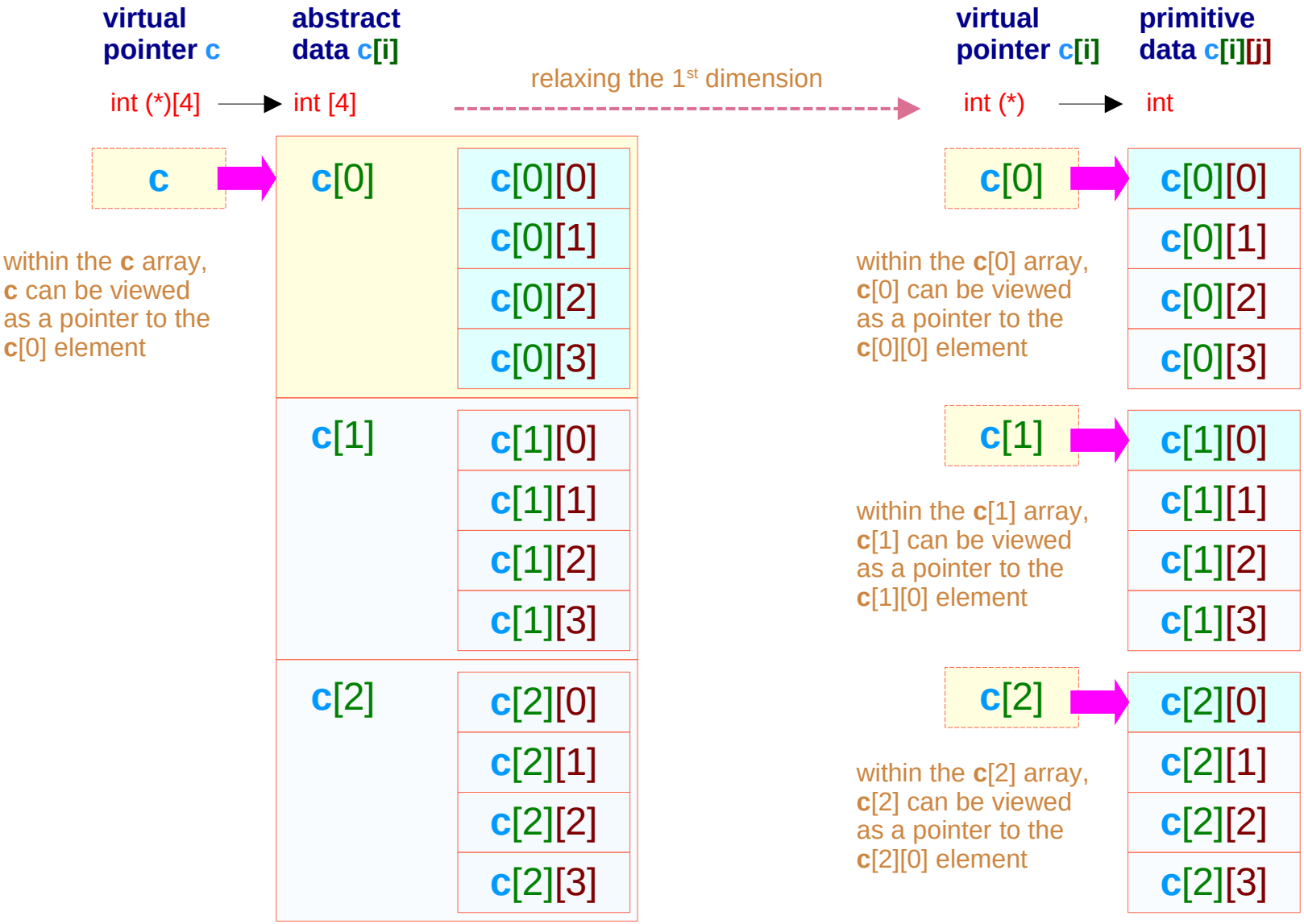


When **c[i]** is referenced outside, c[i] **has the array type int[4]**

When **c[i]** is referencing the 1st element inside, **c[i]** has the pointer type int (*)

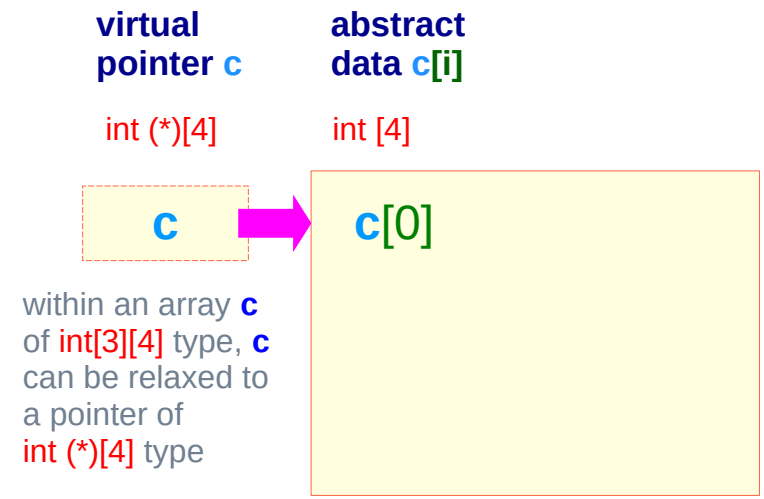
Recursive pointer-to-data views

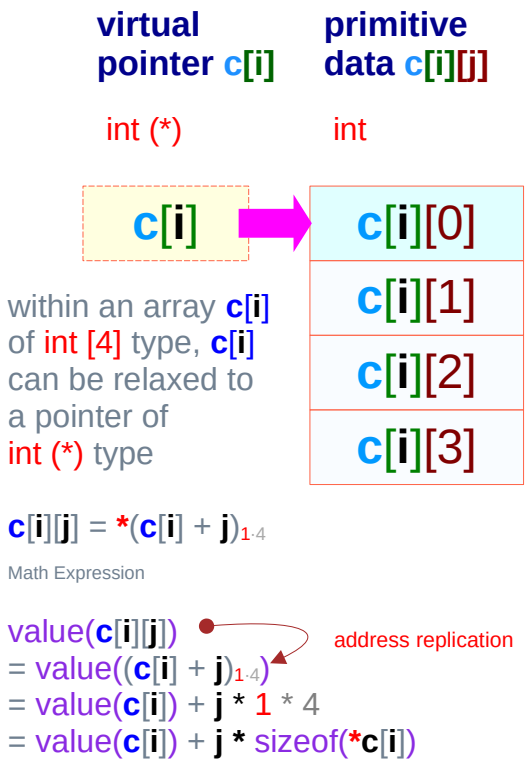
int c[3] [4];



Type, address, and value of **c** and **c[i]**

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

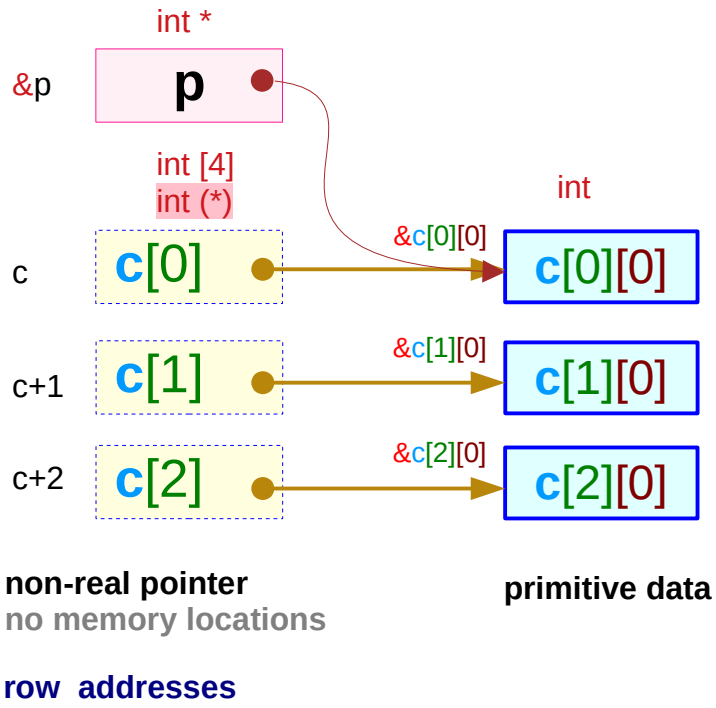




Type, address, and value of **c[i]**

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

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



type(**c[i]**) =	int [4]	abstract data type
	int (*)	virtual pointer type

address		variable		value
value(**c**)	=	value(**c**[0])	=	value(&c[0][0])
value(**c**+1)	=	value(**c**[1])	=	value(&c[1][0])
value(**c**+2)	=	value(**c**[2])	=	value(&c[2][0])

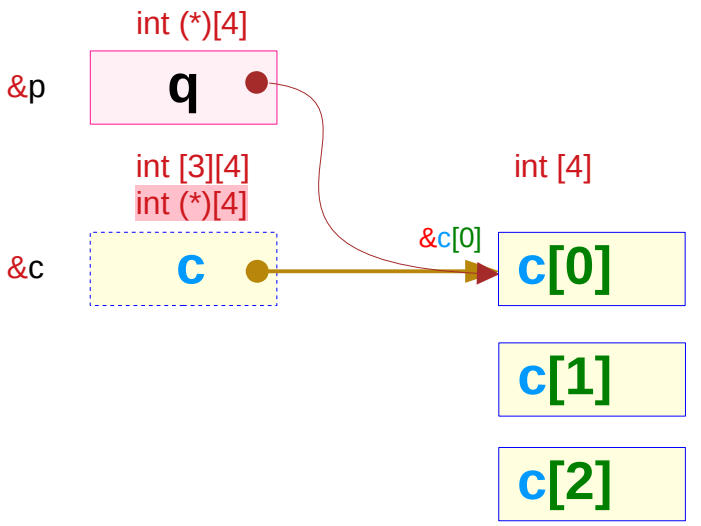
address replications

value(**&p**) ≠ value(**p**) = value(&c[0][0])

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

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

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



type(**c**) =	int [3][4]	abstract data type
	int (*)[4]	virtual pointer type

address	variable	value
value(**&c**) =	value(**c**)	= value(&c[0][0])

address replications

value(**&q**) ≠ value(**q**) = value(&c[0][0])

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

size of abstract data c

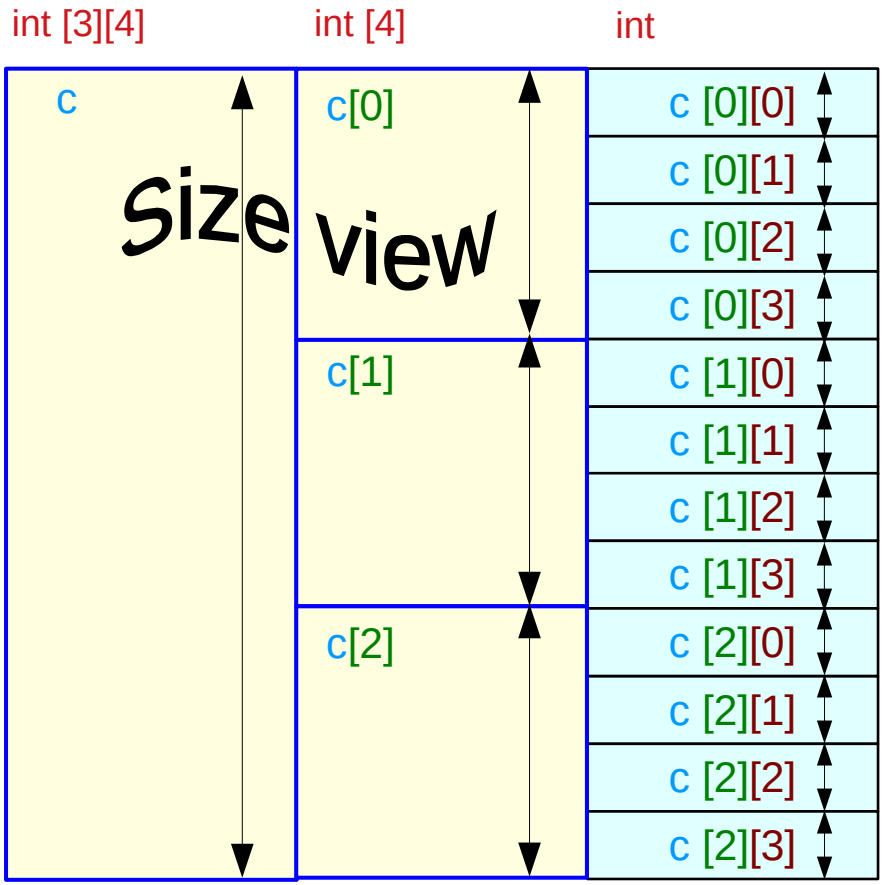
sizeof(c) = sizeof(*c) * 3
= sizeof(c[i]) * 3

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

size of abstract data c[i]

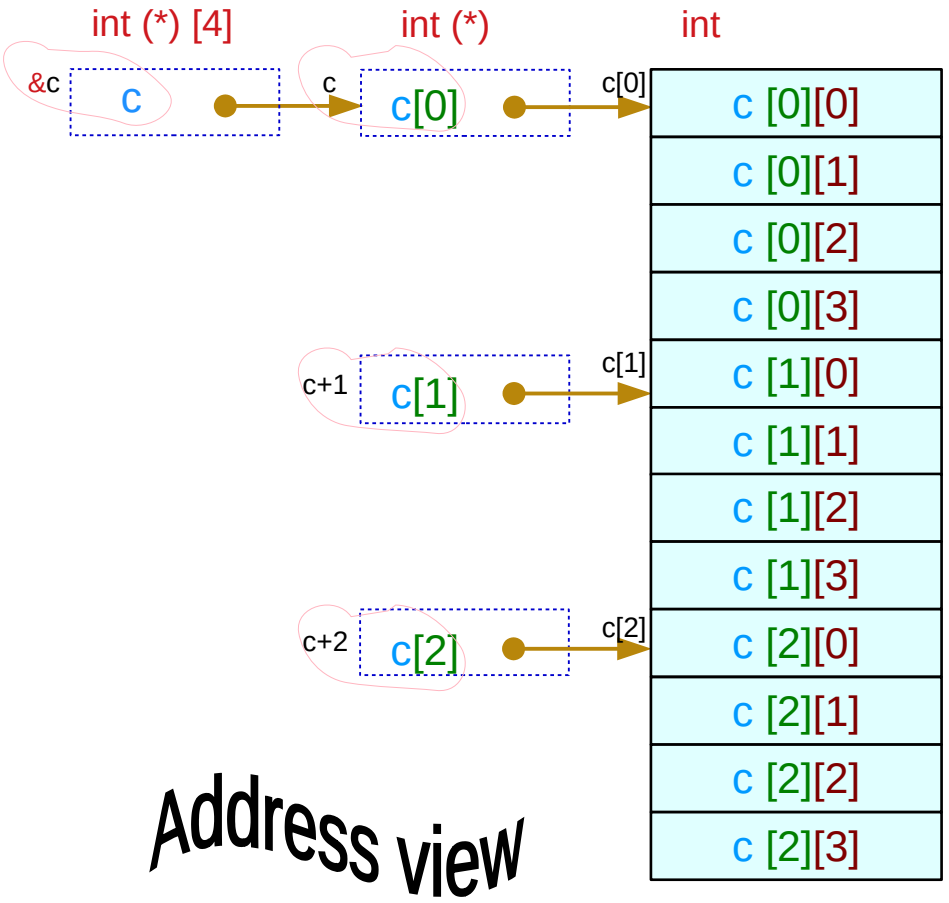
sizeof(c[i]) = sizeof(*c[i]) * 4
= sizeof(c[i][j]) * 4

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



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

value(**c**) = value(**c[0]**) = value(**&c[0][0]**)
value(**c[1]**) = value(**&c[1][0]**)
value(**c[2]**) = value(**&c[2][0]**)



address replications

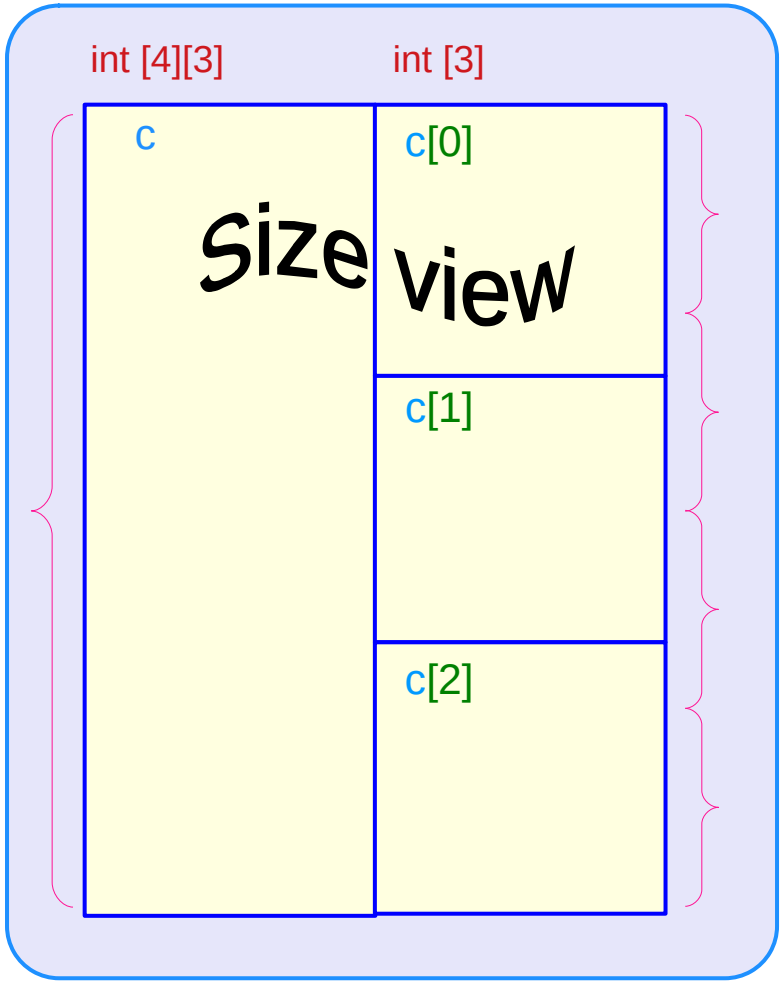
value(&**c**) = value(**c**)

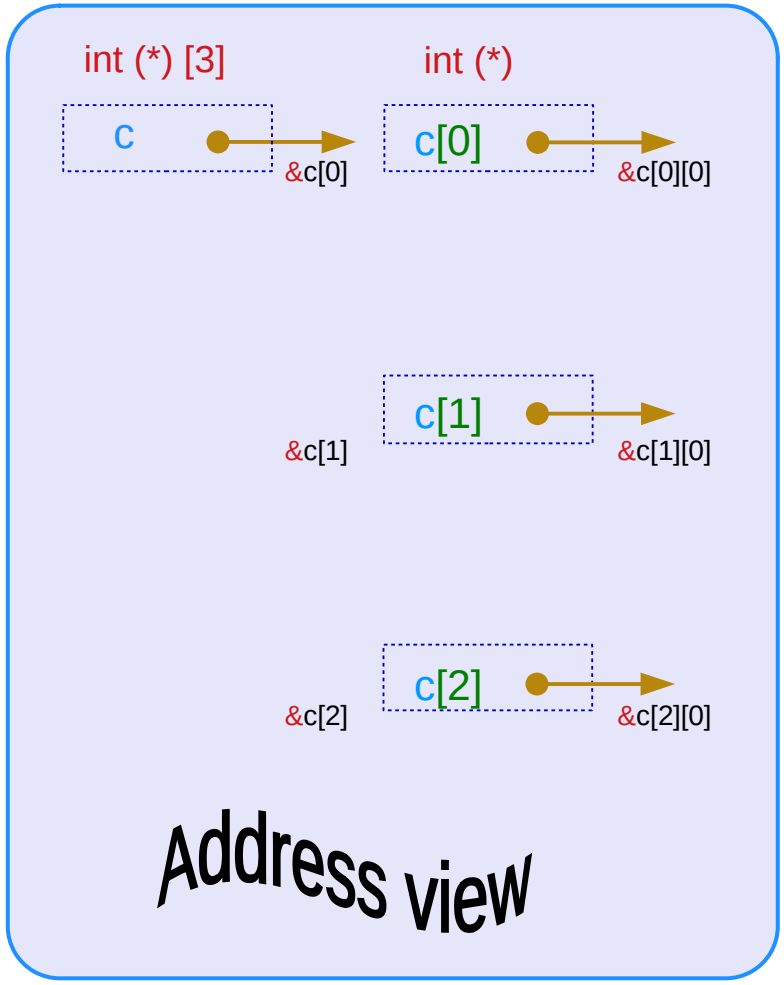
value(&**c**[0]) = value(**c**[0])
value(&**c**[1]) = value(**c**[1])
value(&**c**[2]) = value(**c**[2])

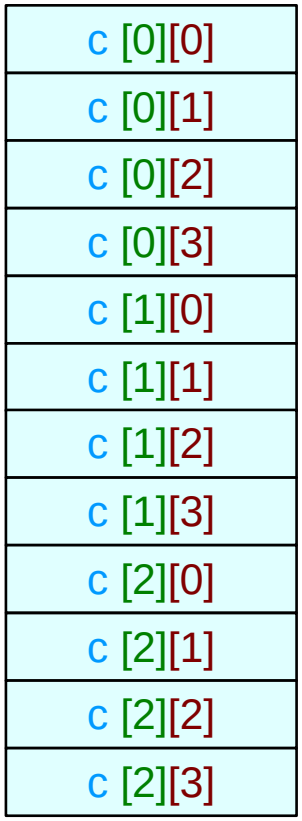
no real pointer can satisfy
these conditions

no physical memory location

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







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

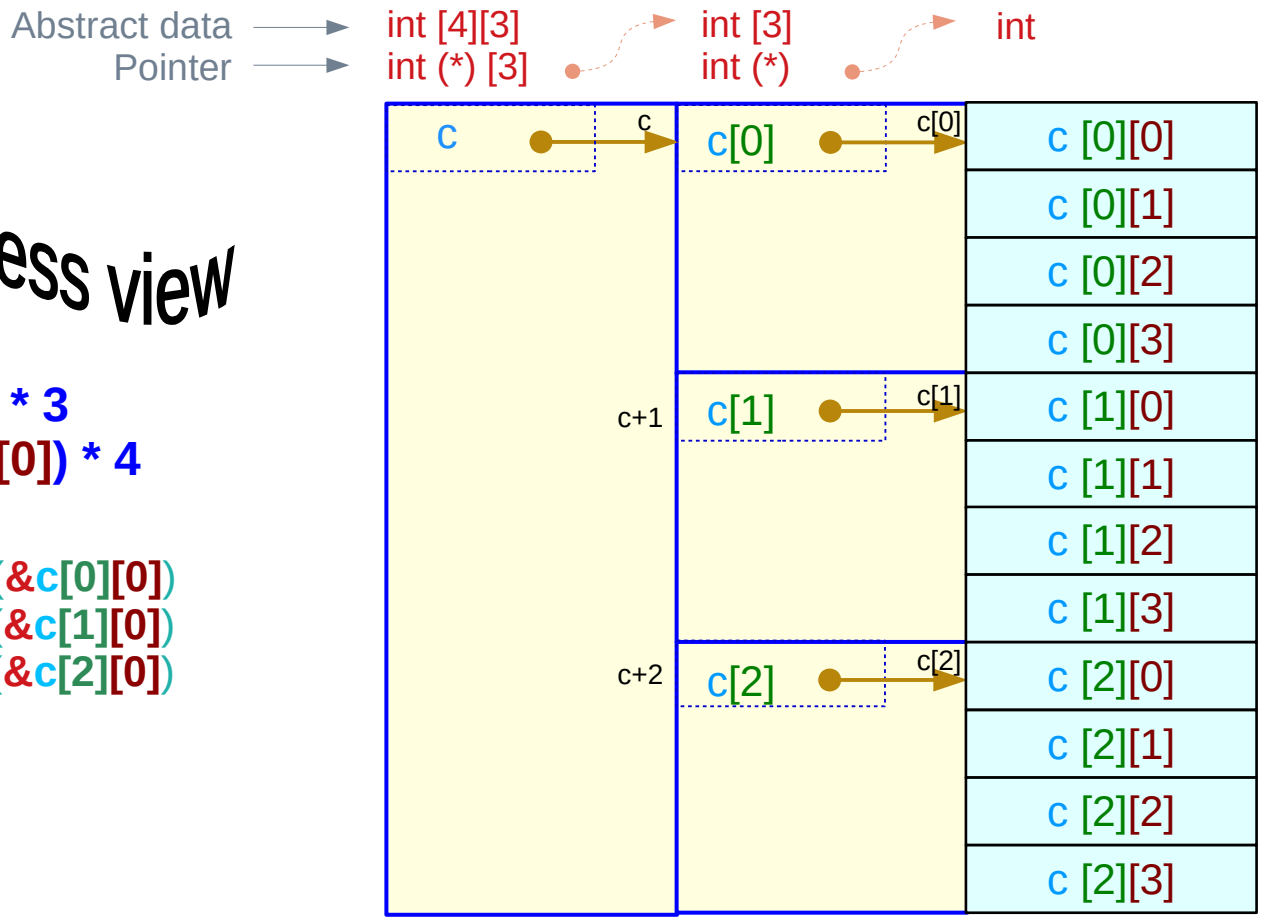
value(**c**) **=** value(**c[0]**) **=** value(**&c[0][0]**)
value(**c[1]**) **=** value(**&c[1][0]**)
value(**c[2]**) **=** value(**&c[2][0]**)

Combining size view and address view

Size view + Address view

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

value(**c**) = value(**c[0]**) = value(**&c[0][0]**)
value(**c[1]**) = value(**&c[1][0]**)
value(**c[2]**) = value(**&c[2][0]**)



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