Arrays (1A)

Young Won Lim
1/28/17

Copyright (c) 2009 - 2017 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

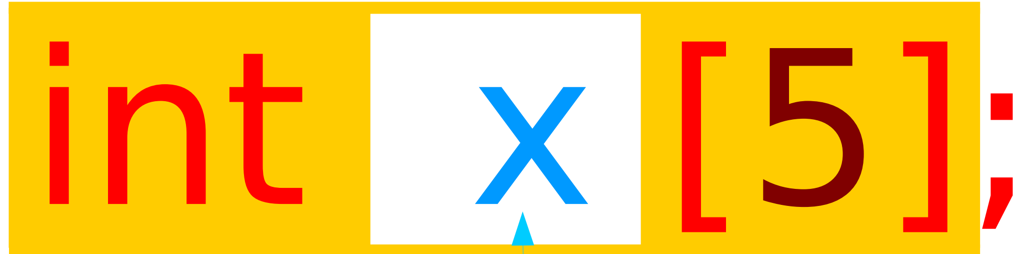
$$m = \frac{1}{N}\sum_{i=0}^{N-1} x_i$$

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

5 variables

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

Definition of an Array



Array Type

Array Name

A constant
value: the starting address of
5 consecutive int variables

Element Type

int x [5] ;

Array Type

Array Name

A constant
Value: the starting address of
5 consecutive int variables

int x [5] ;

i = 0, … , 4

Element Type

Using an Array

int x [5] ;   Array Name

int x [5] ;   Element Type : int

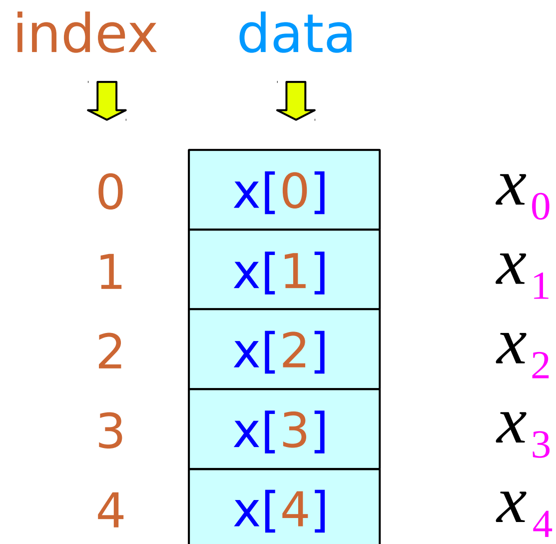
int variables   x [*i*]   i = 0, … , 4

Accessing array elements – using an index

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

x is an array
with 5 integer elements

5 int variables



x_0

x_1

x_2

x_3

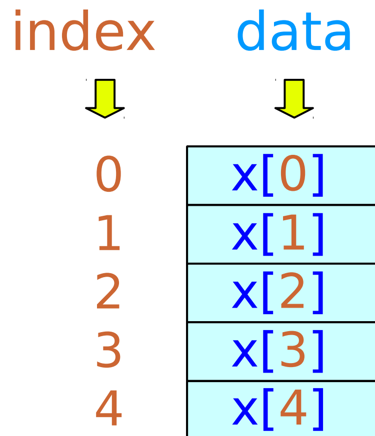
x_4

Accessing array elements – using an address

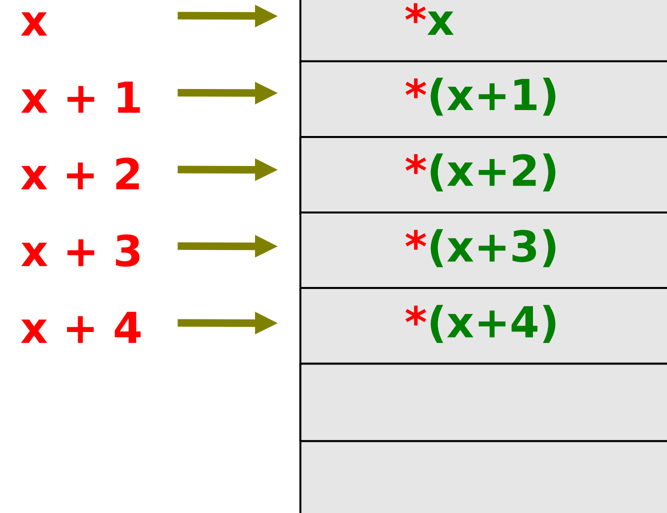
```
int        x[5];
```

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**)

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*

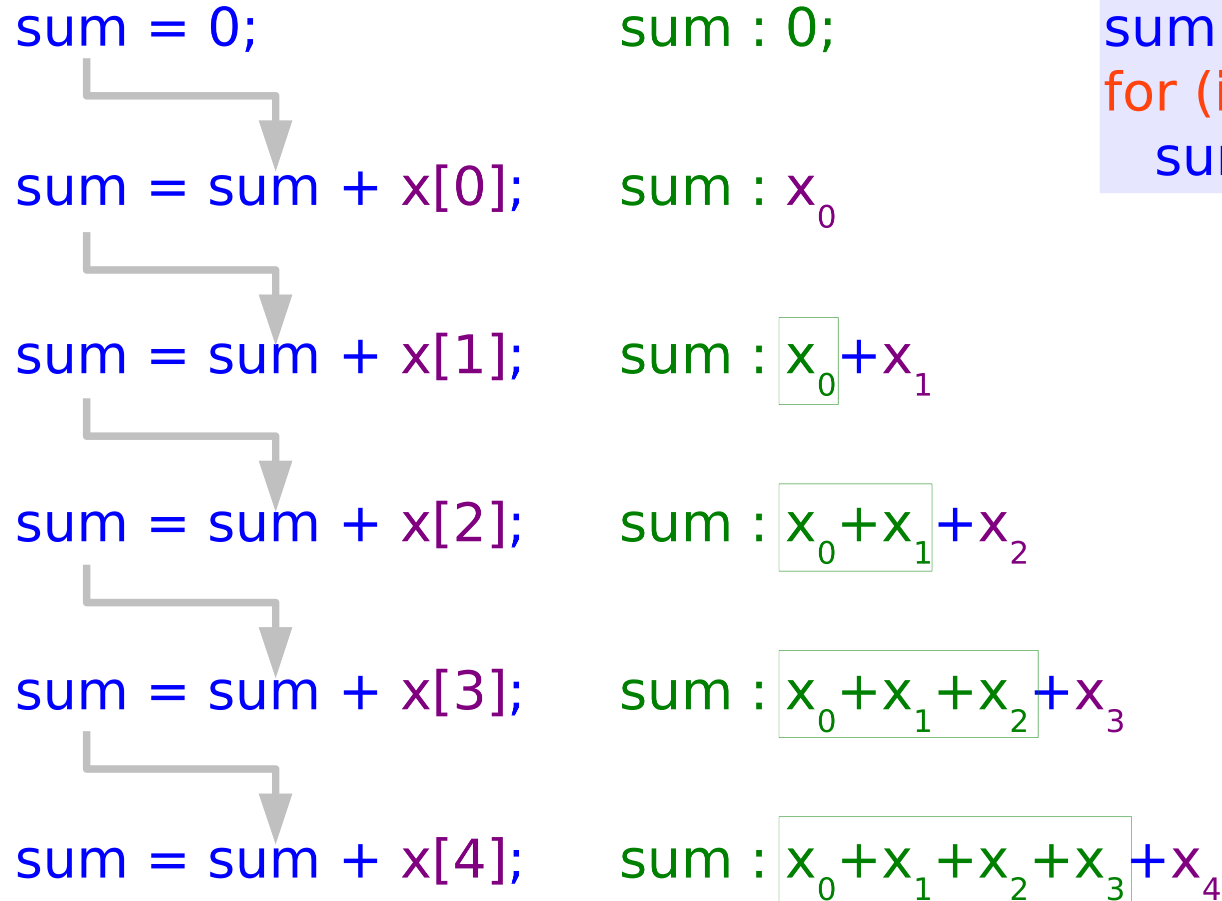
treated as a variable

read x[i] write

read *(x+ i) write

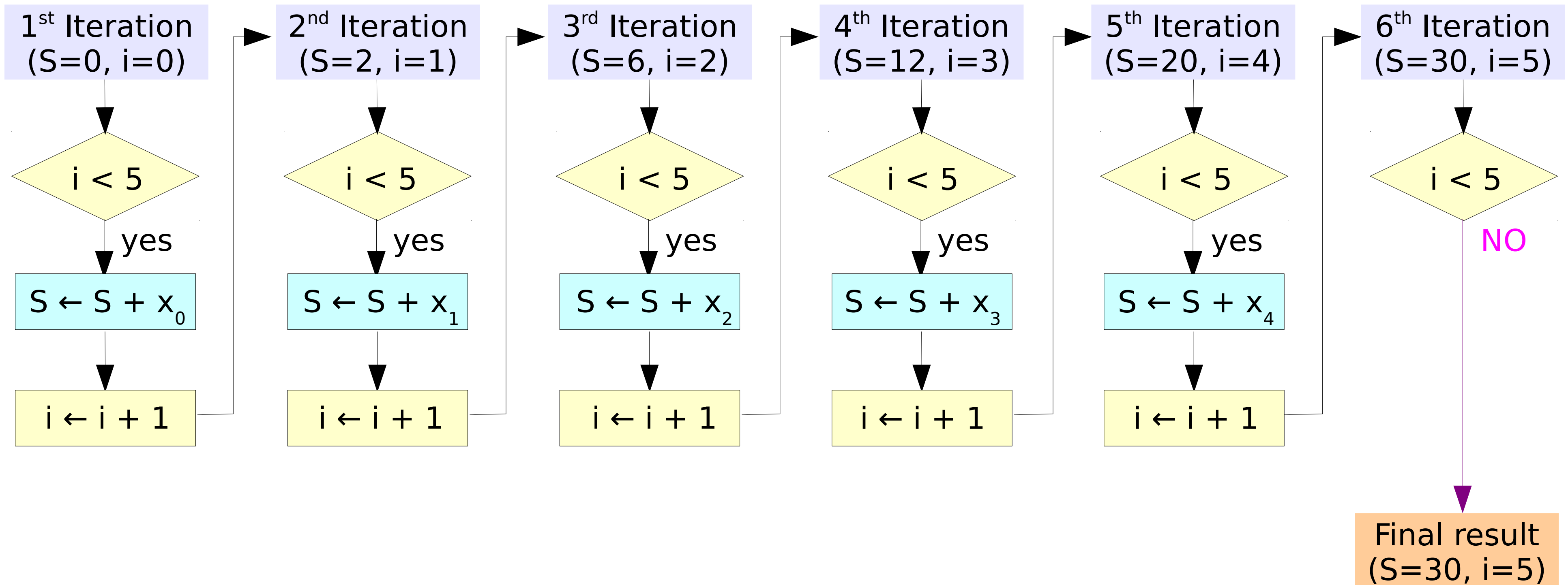
an index variable **i**

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) = 400;
```

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 [ ] ;
```

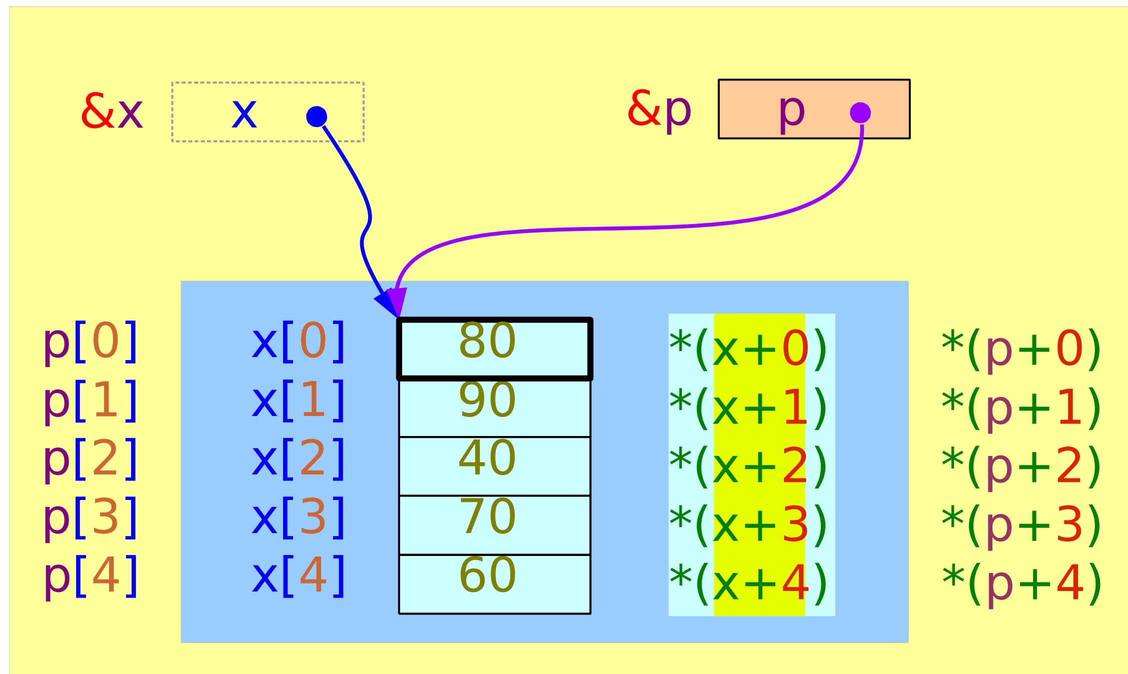
(crossed out)

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

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

Accessing an Array with a Pointer Variable

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

int  *p = x;
```



x is a constant symbol
cannot be changed

p is a variable
can point to other address

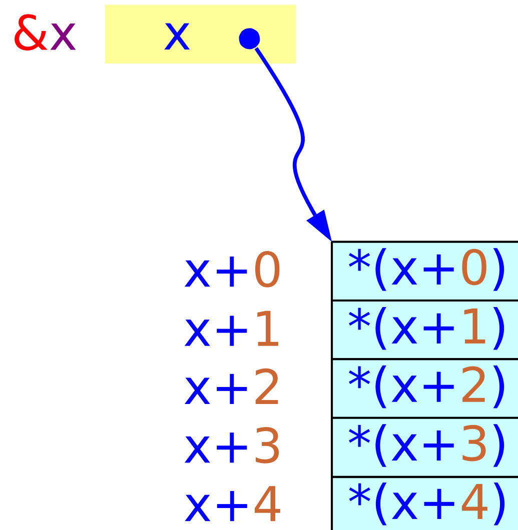
An Array Name and a Pointer Variable

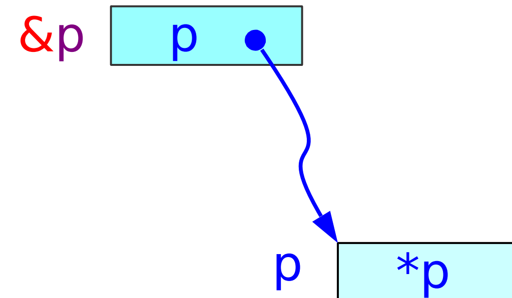
int x [5] ;

int * p ;

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

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

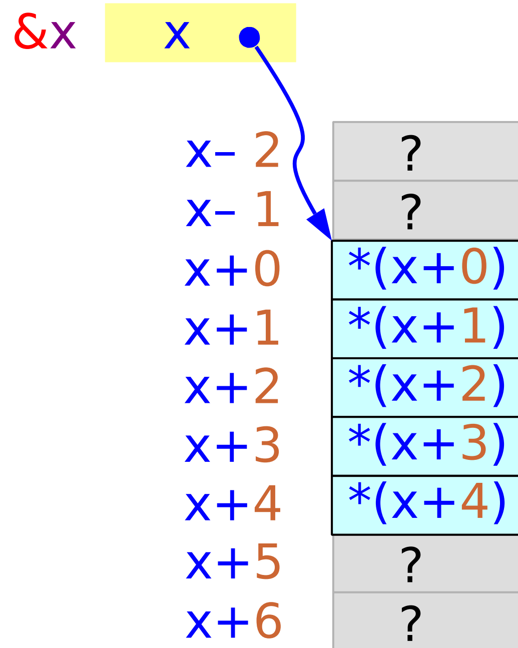


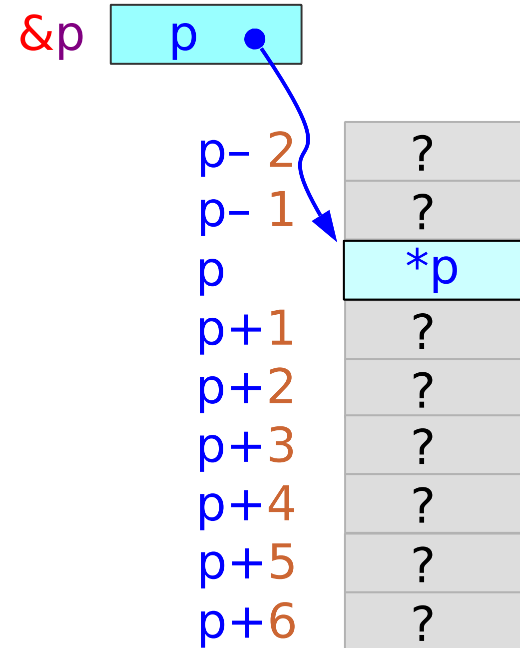


Out of range index

int x [5] ;

int * p ;



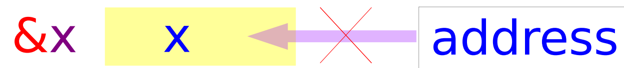


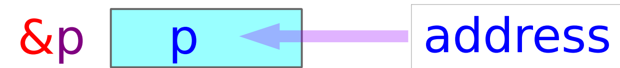
A programmer's responsibility

Assignment of an address

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

```
int  * p ;
```



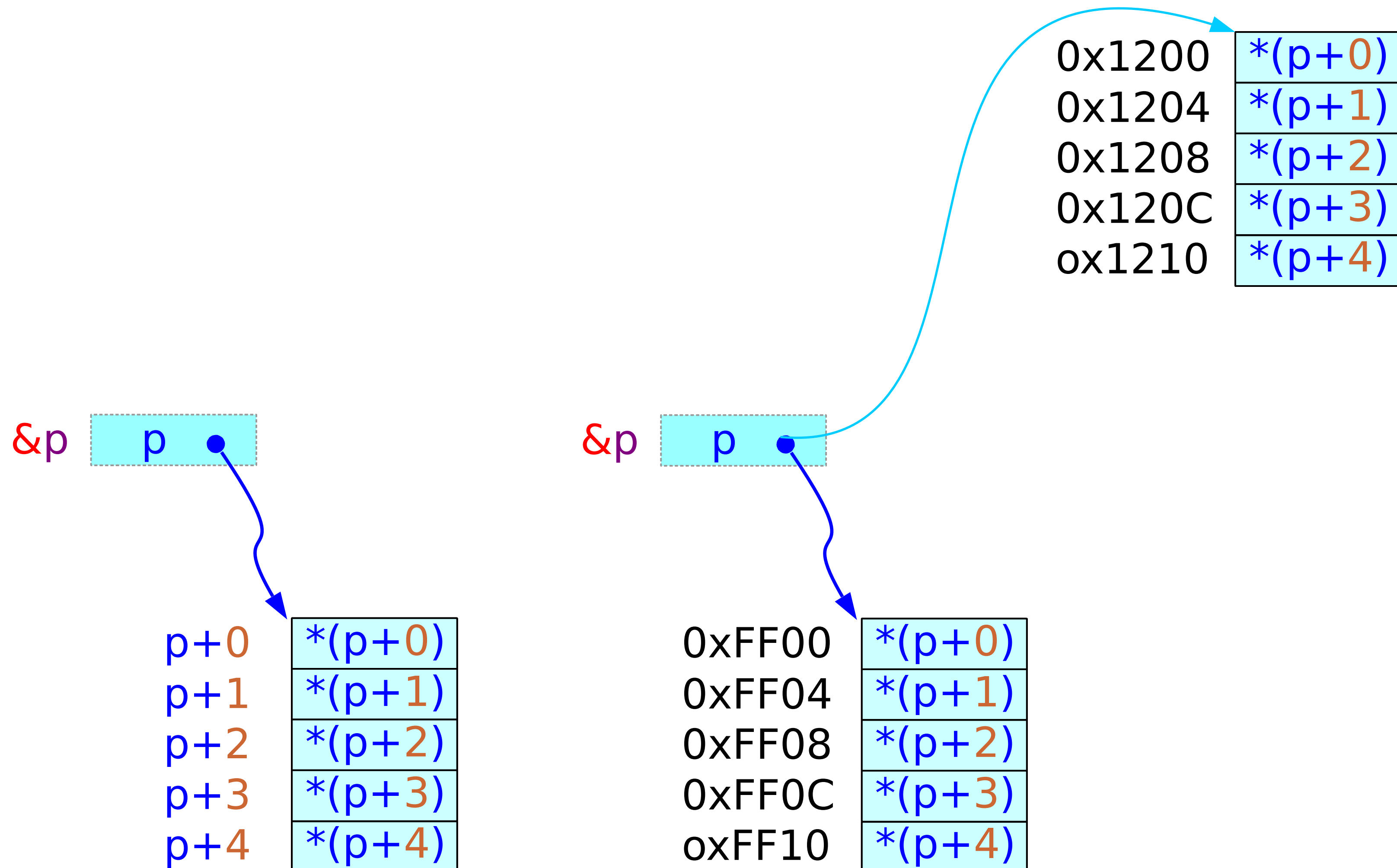


x is not a variable but a constant symbol (x and &x give the same address)

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

P is a variable with allocated memory and its value can be changed

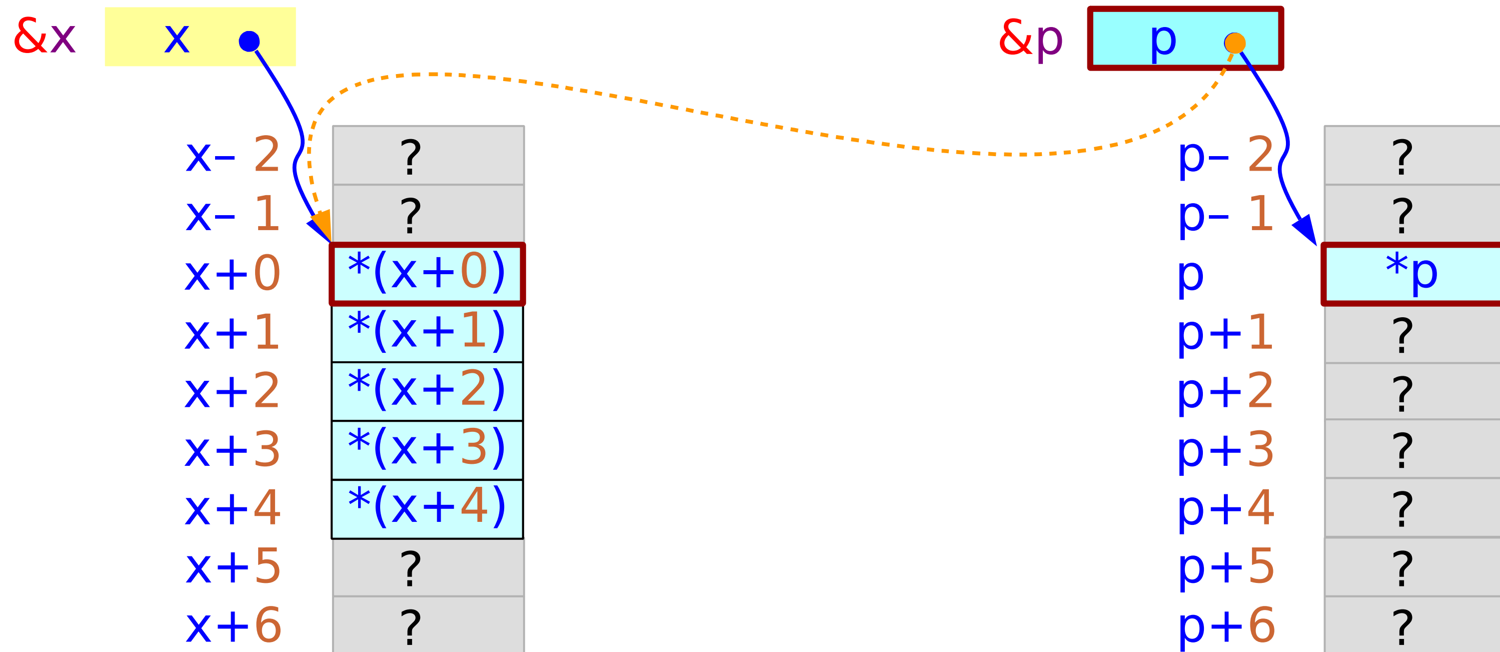
Pointer variable can point different locations



Pointer to an integer

int x [5] ;

int * p ;



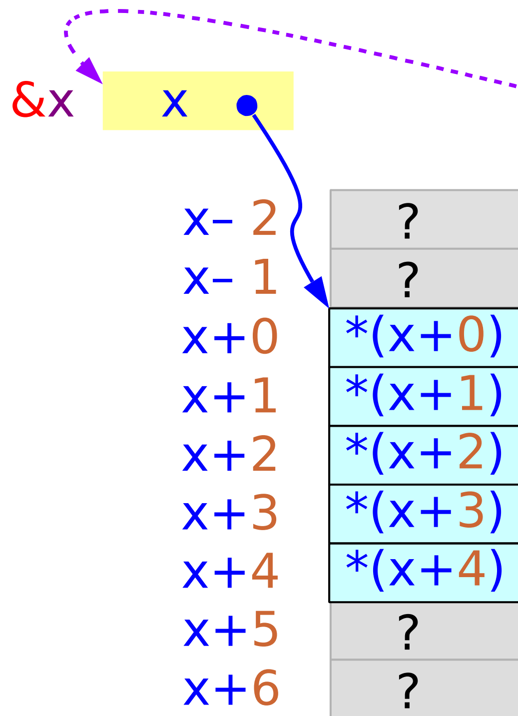
*p = *x ;

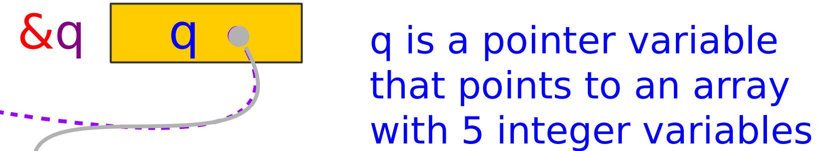
p = x ;

Pointer to an array

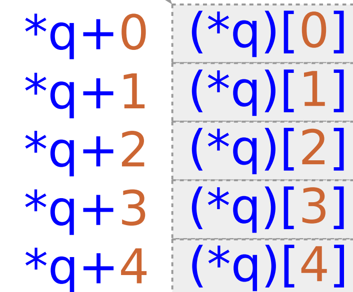
int x [5] ;

int (*q) [5] ;





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



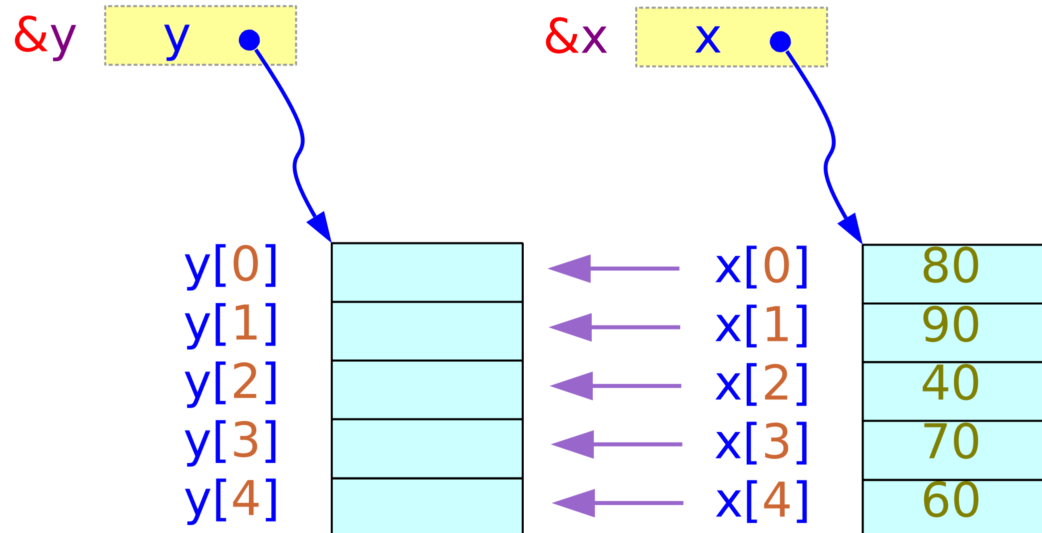
*q = x ;

q = &x ;

* not frequently used feature

Copying an Array to another Array

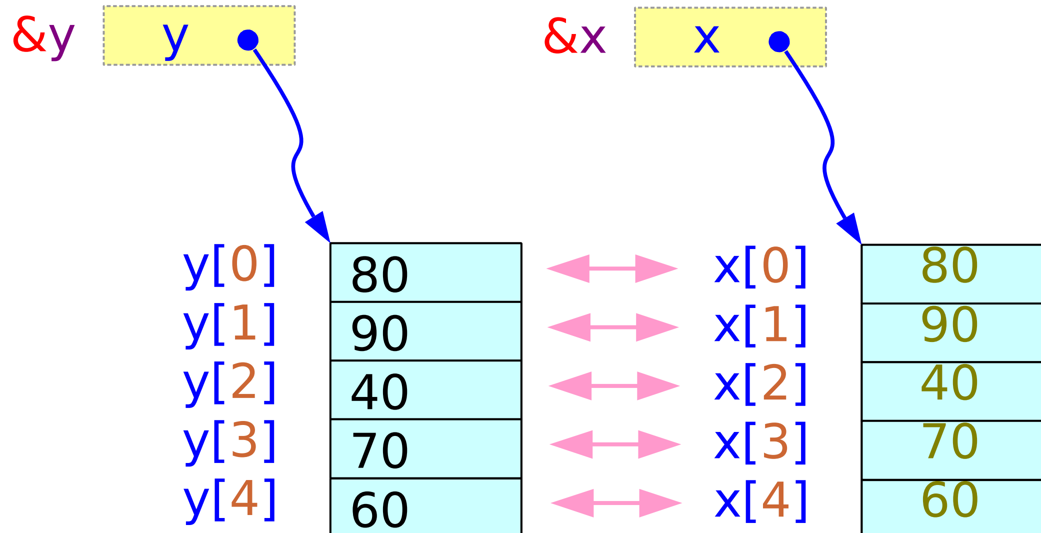
```
int x [5] = { 1, 2, 3, 4, 5 };
int y [5] ;
x = y;
```



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

Comparing an Array with another Array

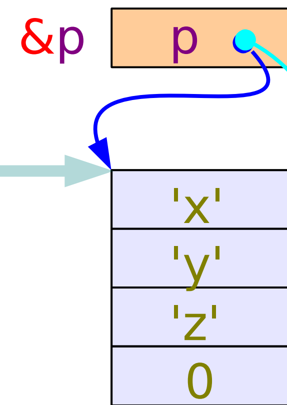
```
int x [5] = { 1, 2, 3, 4, 5 };
int y [5] = { 1, 2, 3, 4, 5 };

x == y
```




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

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

char  *p = “xyz”;
```

s+0	'a'
s+1	'b'
s+2	'c'
s+3	'd'
s+4	0



a compiler determined constant address

a constant character string (array)

~~*p = 'P';~~

s+0	'M'
s+1	'N'
s+2	0
s+3	'd'
s+4	0

```
*(s+0) = 'M';
*(s+1) = 'N';
*(s+2) = '\0';
```

```
strcpy (s, “MN”);
```

a different address

'I'
'J'
'K'
0

```
p = “IJK”
```

Character Arrays

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

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

(s = "xyz"; is crossed out)

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

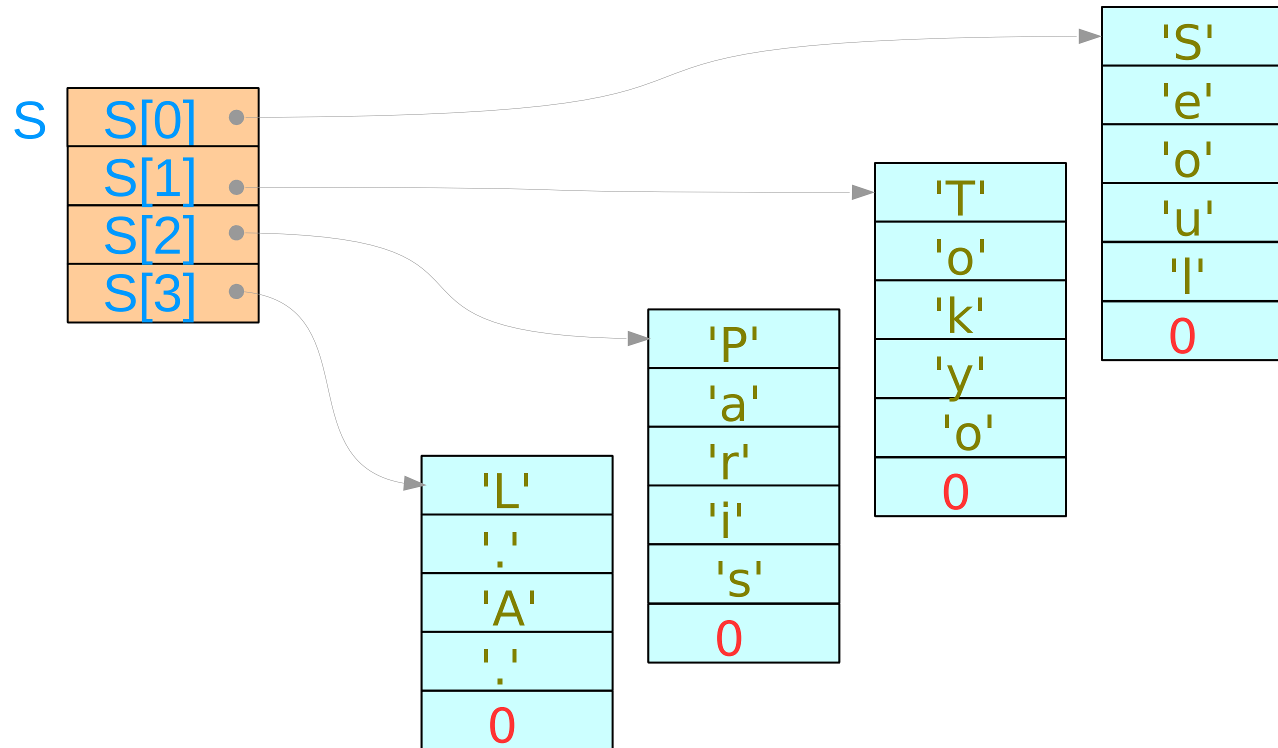
const

s+0	'a'
s+1	'b'
s+2	'c'
s+3	'd'
s+4	0

&p p

'x'
'y'
'z'
0

Arrays of Pointers



A[] Notation

1. An array definition with initializers

2. A formal parameter definition in a function

A[ ][n] Notation

1. An array definition with initializers

2. A formal parameter definition in a function

Passing 1-d Arrays

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

func( a );

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

address	value	alias
a	a[0]	x
a+1	a[1]	x+1
a+2	a[2]	x+2
a+3	a[3]	x+3

&x x=a

Passing 2-d Arrays

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

func( b );
```

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

address	value	alias
b[0]	b[0][0]	y[0]
b[0]+1	b[0][1]	y[0]+1
b[0]+2	b[0][2]	y[0]+2
b[1]	b[1][0]	y[1]
b[1]+1	b[1][1]	y[1]+1
b[2]+2	b[1][2]	y[1]+2

&y y=b

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 ;
```

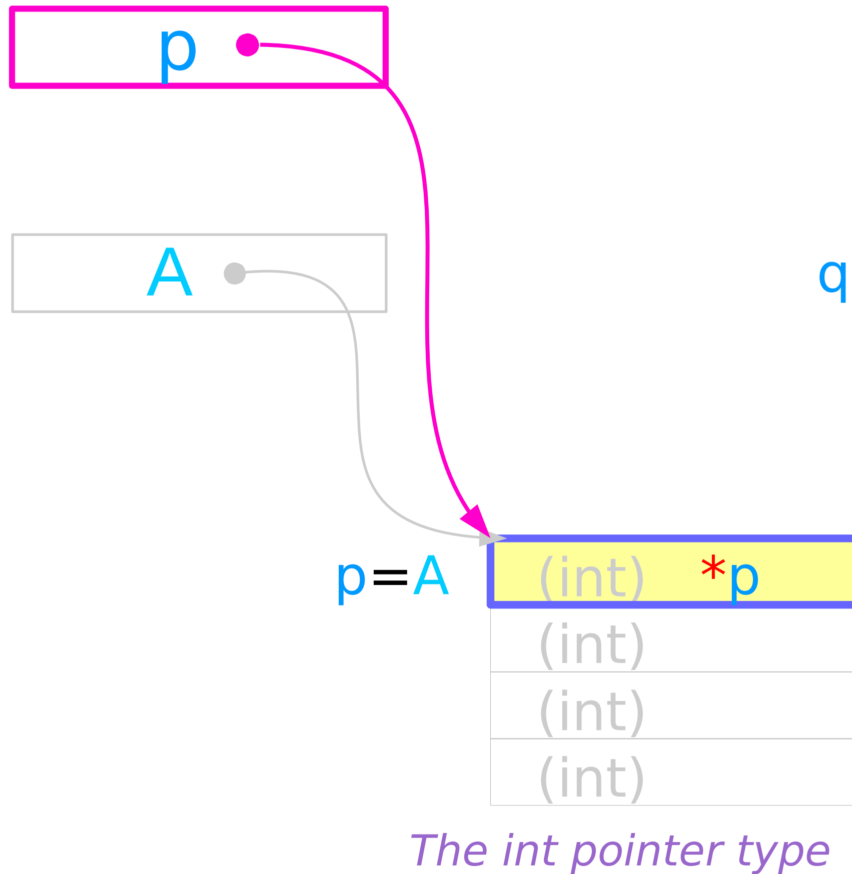
```
typedef int (* PType) [10] ;

PType p;
```

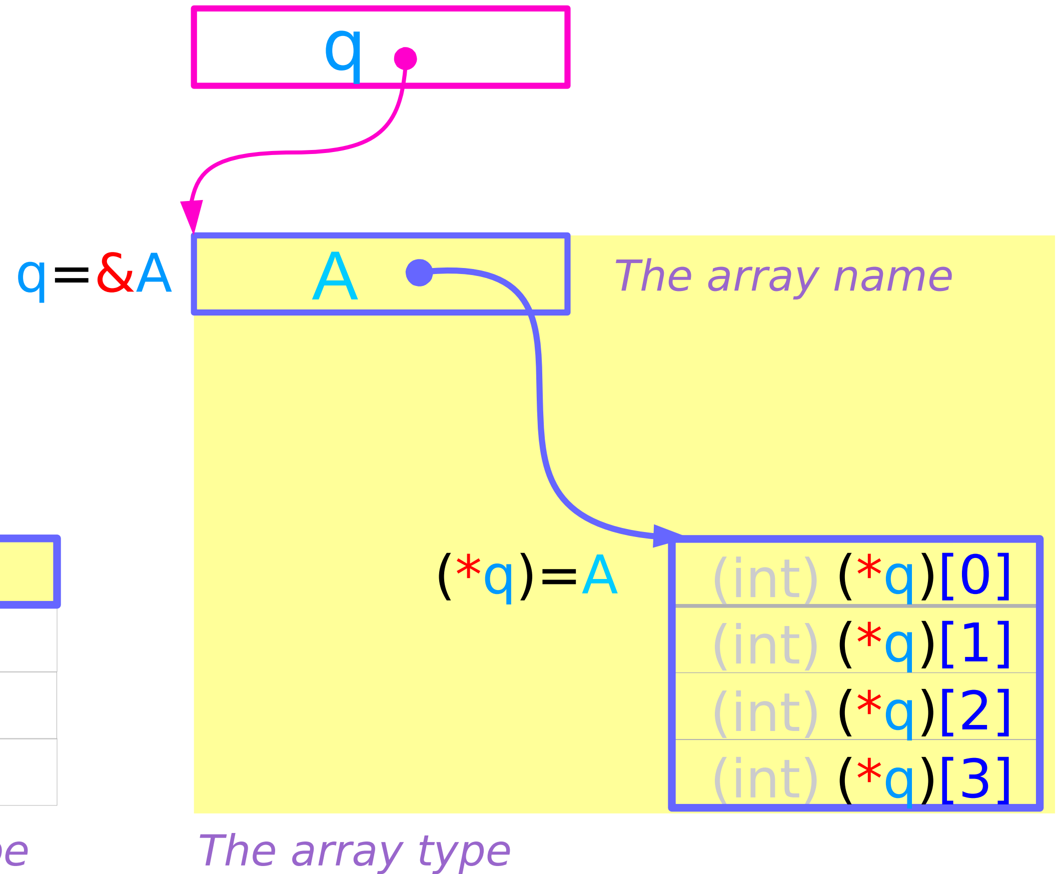
```
p = &A ;

p = q ;
```

Pointer to Integer v.s. Pointer to Array

```
int    *p ;        p = A ;
```

```
int    (*q) [4];   q = &A ;
```

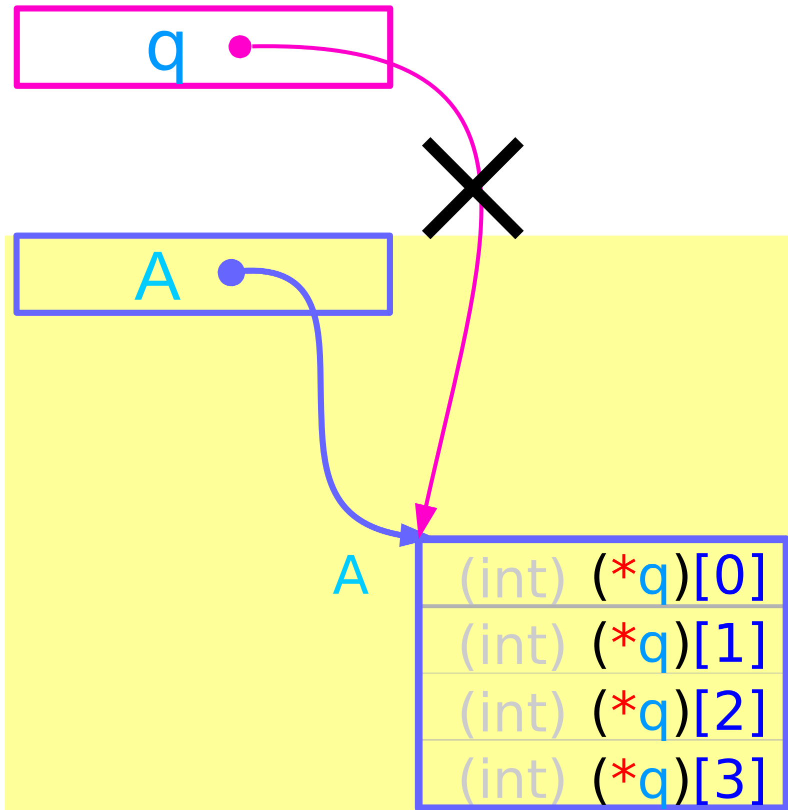


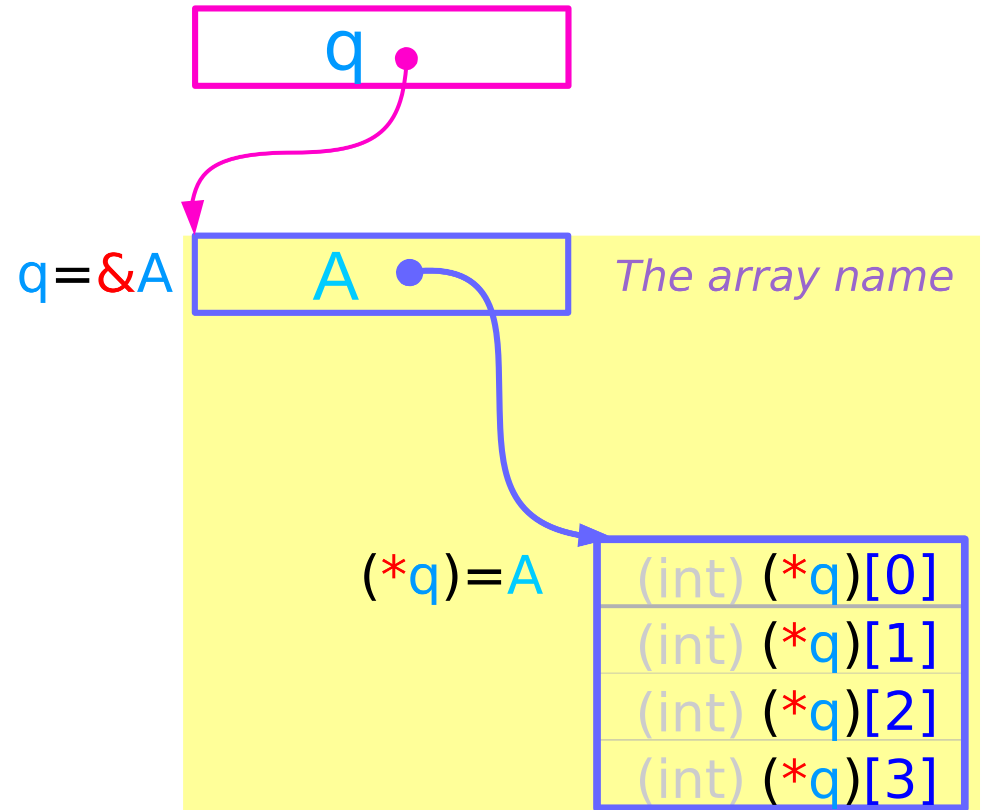


Must point to an array type (array name)

```
int    (*q) [4];   q ≠ A ;
```

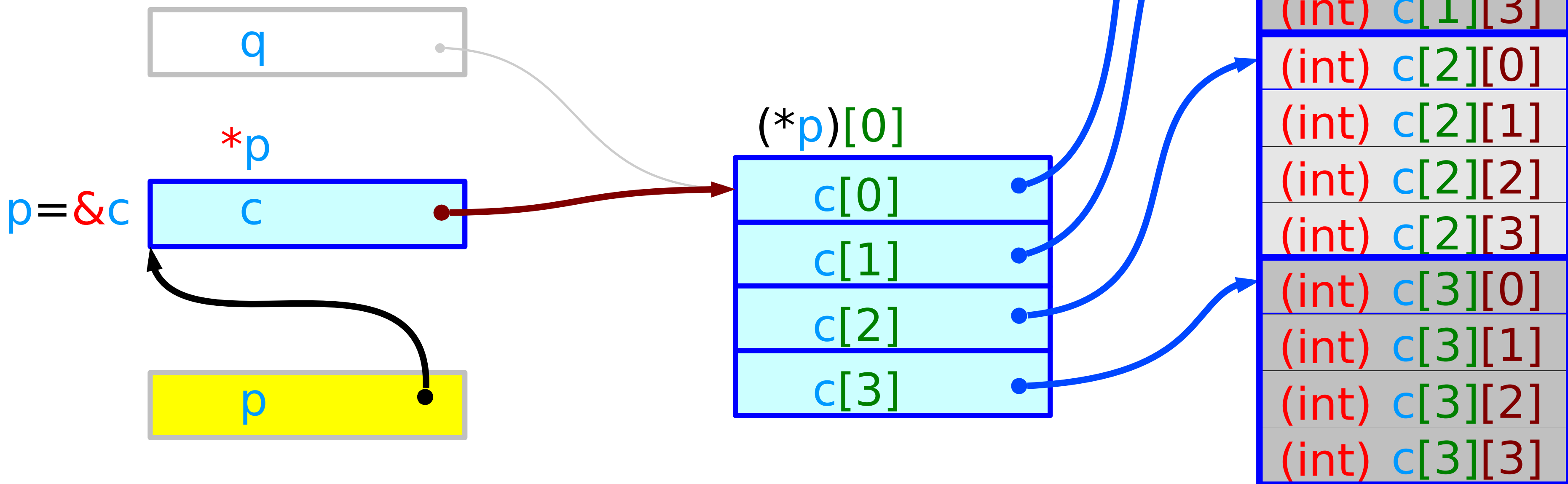
```
int    (*q) [4];   q = &A ;
```



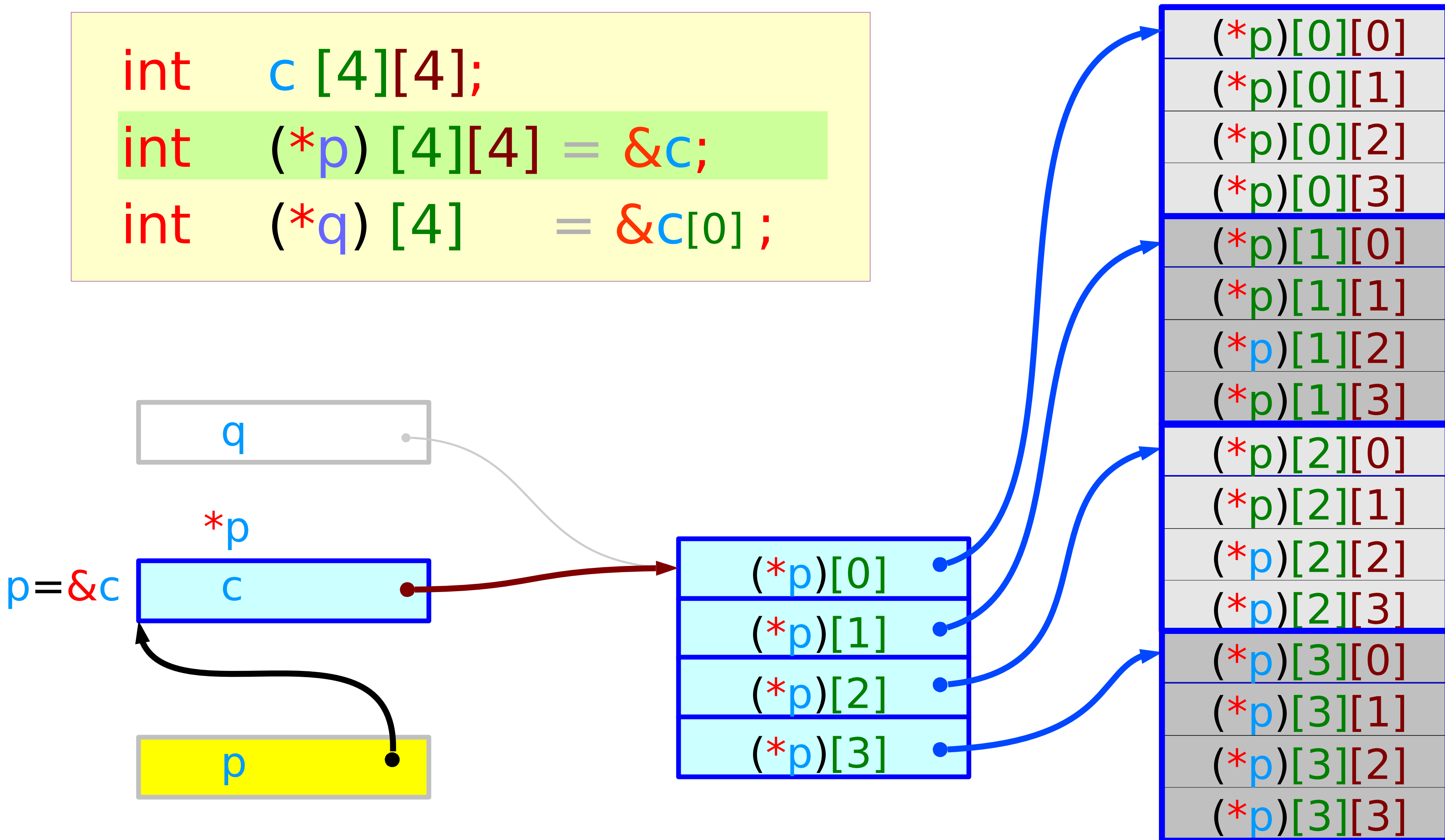


Pointer to 2-d Arrays

```
int     c [4][4];
int     (*p) [4][4] = &c;
int     (*q) [4]     = &c[0] ;
```

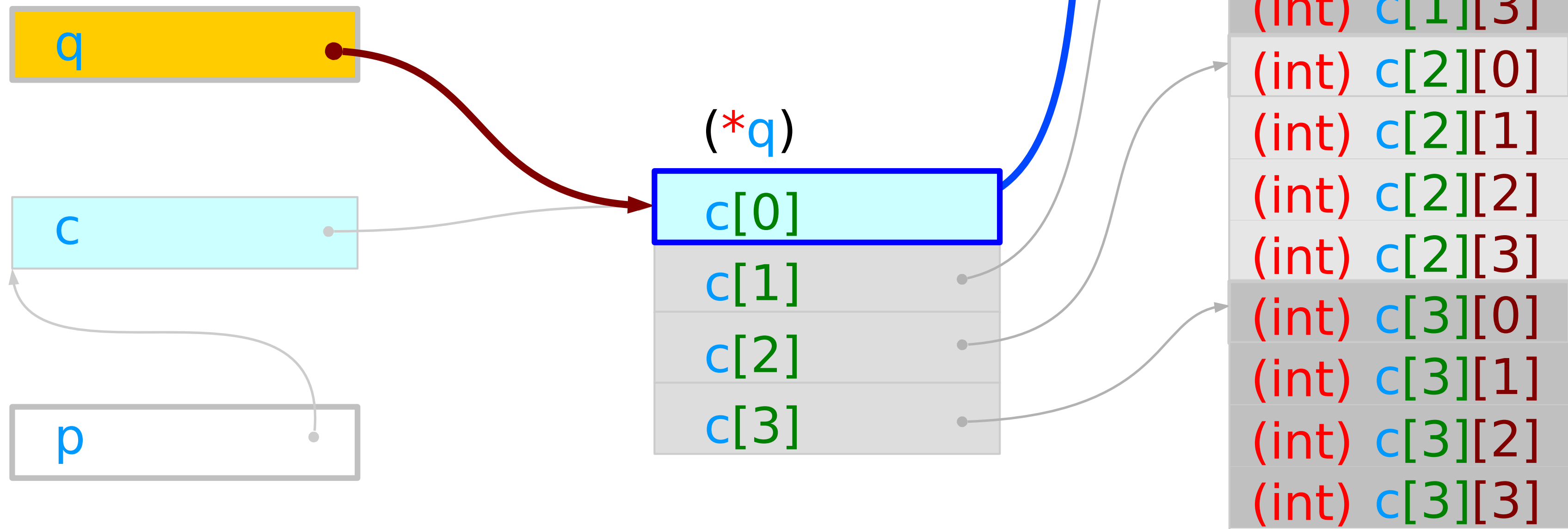


Pointer to 2-d Arrays – accessing elements



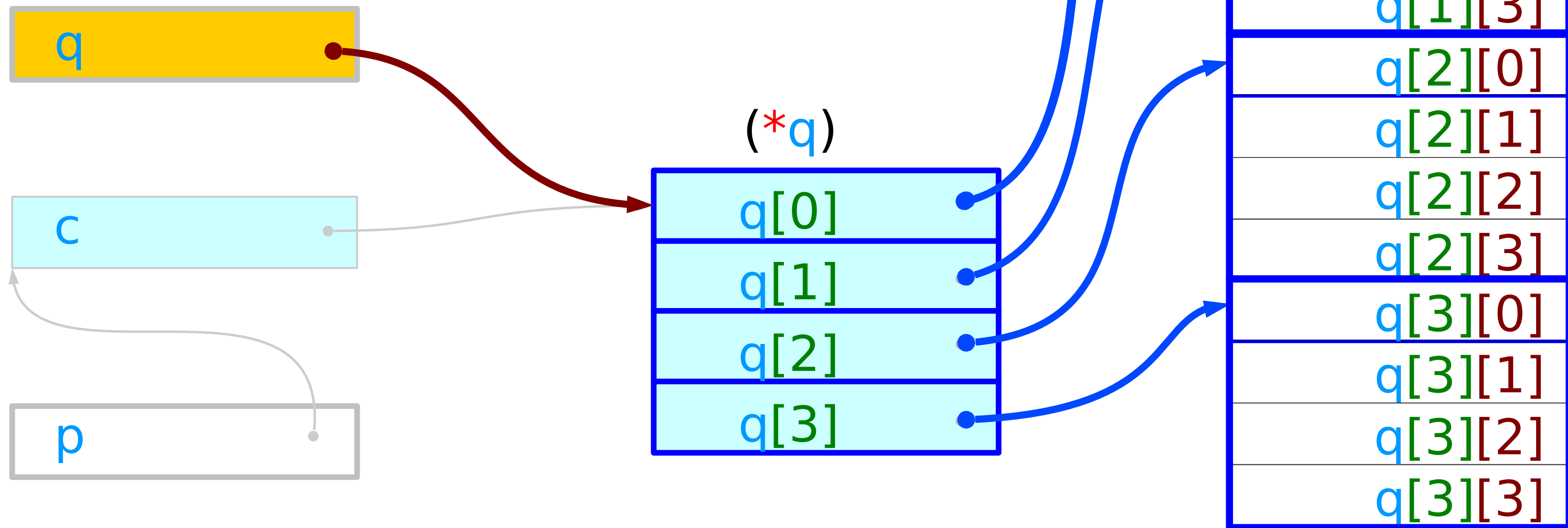
Pointer to 1-d Arrays

```
int    c [4][4];
int    (*p) [4][4] = &c;
int    (*q) [4]    = &c[0] ;
```



Pointer to 1-d Arrays – accessing 2-d elements

```
int    c [4][4];
int    (*p) [4][4] = &c;
int    (*q) [4]    = &c[0] ;
```



Pointer to 1-d Arrays – pointer notation

```
int    c [4][4];
int    (*p) [4][4] = &c;
int    (*q) [4]    = &c[0] ;
```

(*(q+0))	⟺	q[0]
(*(q+1))	⟺	q[1]
(*(q+2))	⟺	q[2]
(*(q+3))	⟺	q[3]

(*(q+0))[0]	⟺	q[0][0]
(*(q+0))[1]	⟺	q[0][1]
(*(q+0))[2]	⟺	q[0][2]
(*(q+0))[3]	⟺	q[0][3]
(*(q+1))[0]	⟺	q[1][0]
(*(q+1))[1]	⟺	q[1][1]
(*(q+1))[2]	⟺	q[1][2]
(*(q+1))[3]	⟺	q[1][3]
(*(q+2))[0]	⟺	q[2][0]
(*(q+2))[1]	⟺	q[2][1]
(*(q+2))[2]	⟺	q[2][2]
(*(q+2))[3]	⟺	q[2][3]
(*(q+3))[0]	⟺	q[3][0]
(*(q+3))[1]	⟺	q[3][1]
(*(q+3))[2]	⟺	q[3][2]
(*(q+3))[3]	⟺	q[3][3]

2-D Array Definition

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

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

row major ordering

2-D Array Interpretation

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

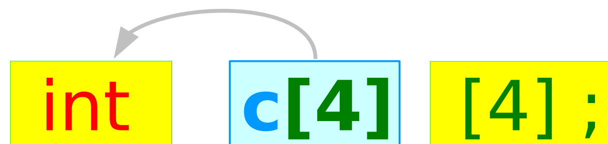
1. recursive interpretation

2. linear array interpretation

Pointer to the start of 1-d arrays

```
int        a [4];

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

a points to *a 4 integer element array*

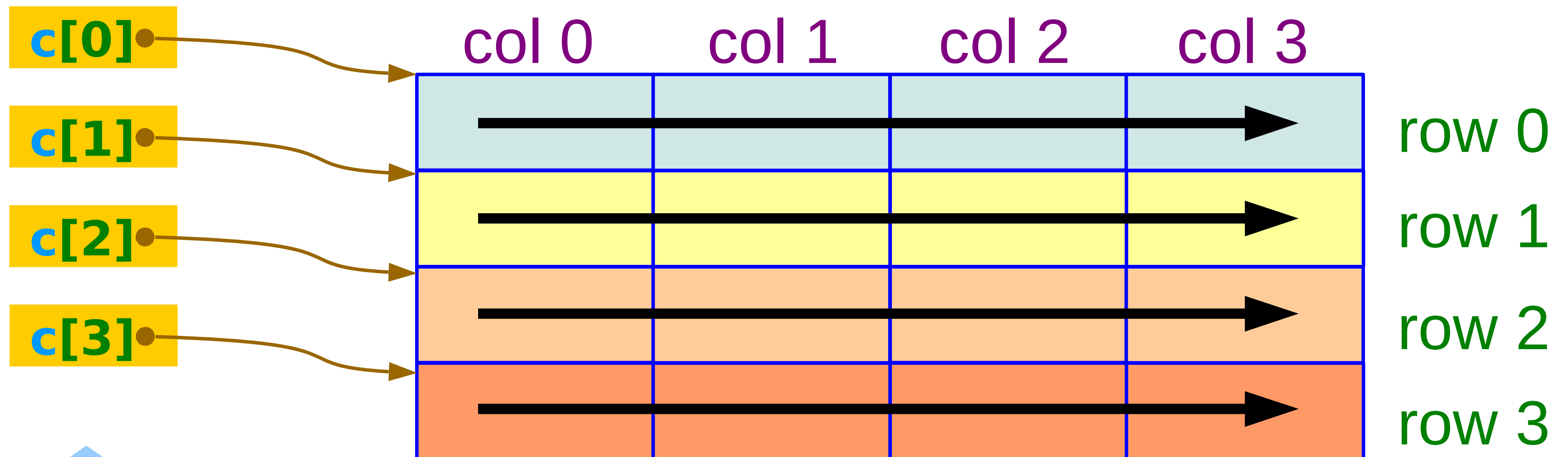


each of **c[0]**, **c[1]**, **c[2]**, **c[3]**
points to *a 4 integer element array*

Row Major Ordering

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

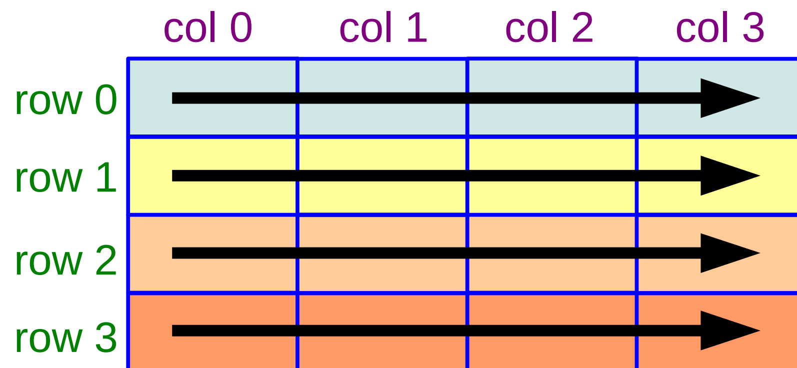
row major ordering

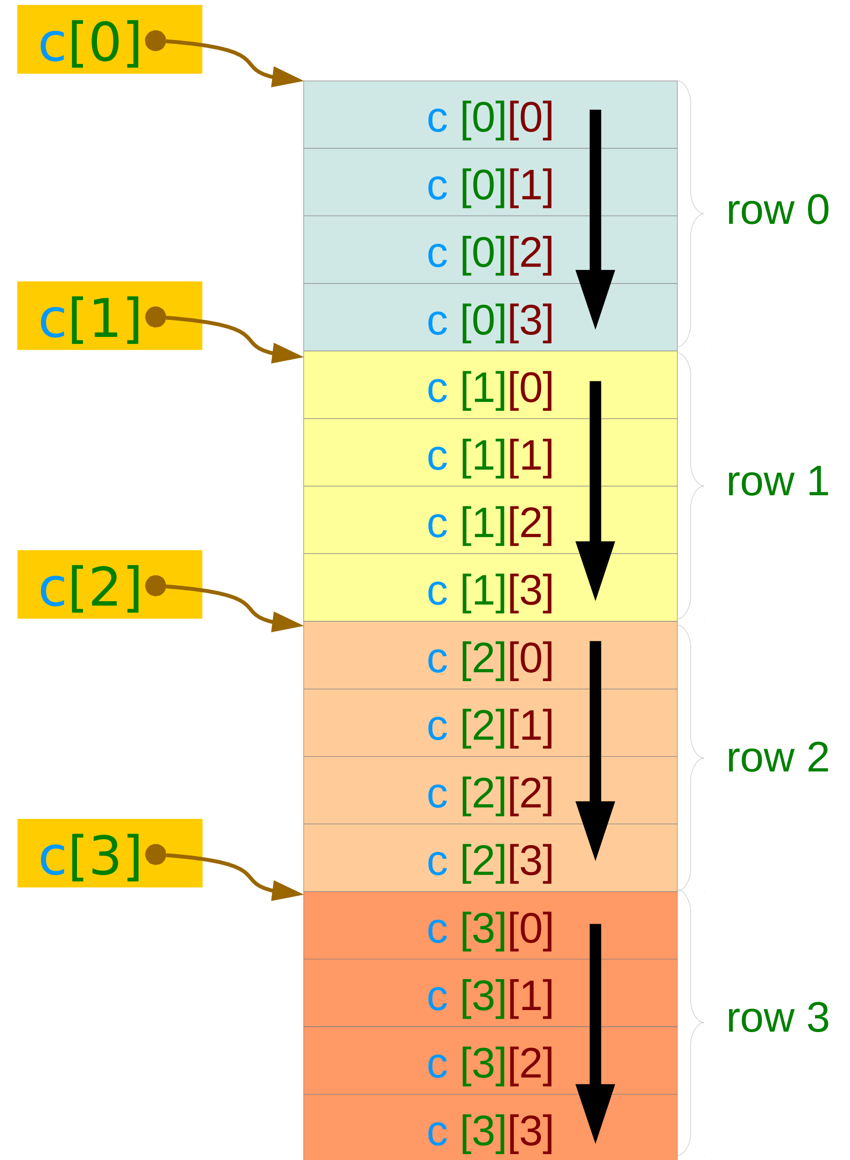


Consider each c[i] as the name of an array
that has 4 integer elements

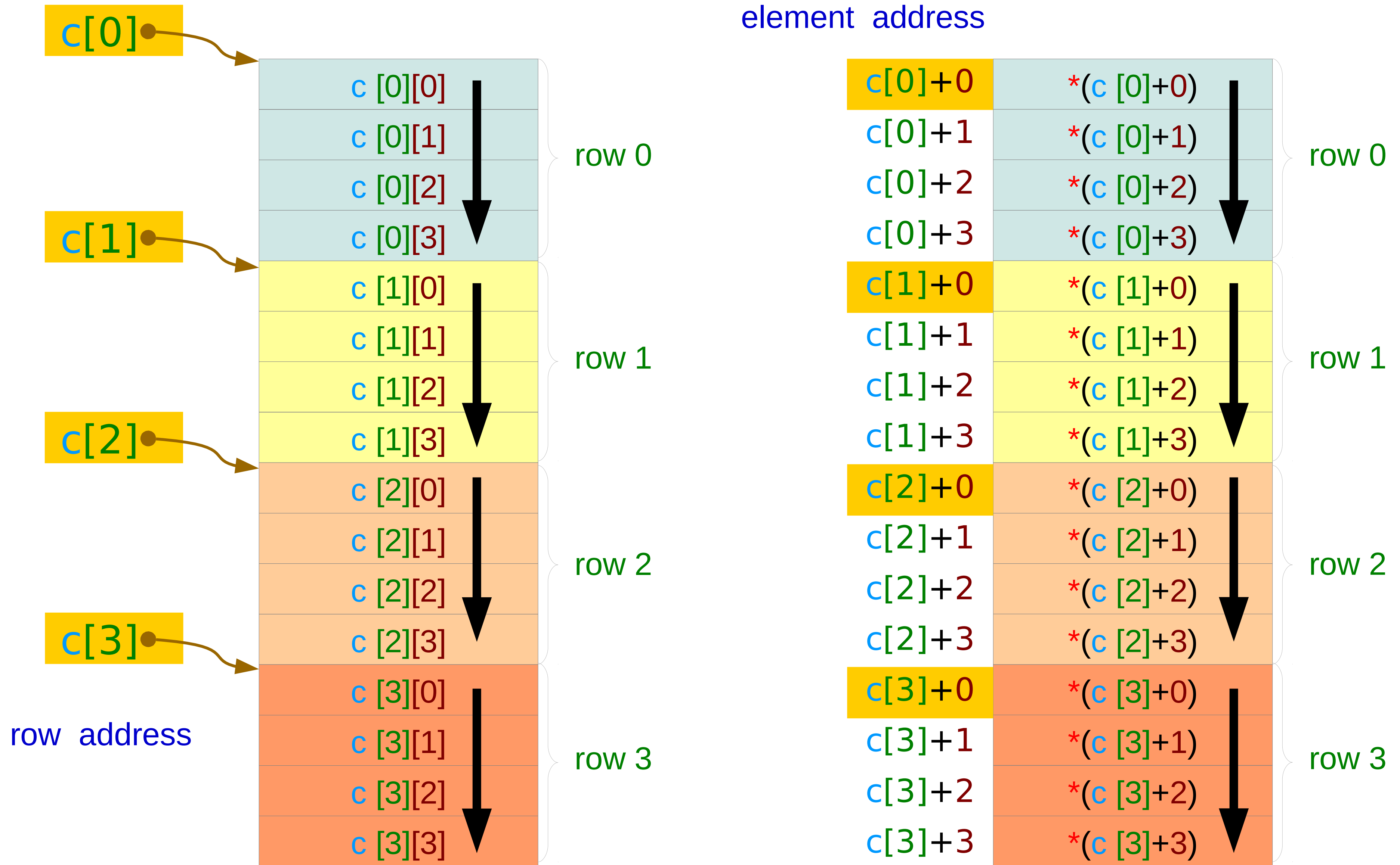
Linear Array Memory Layout

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





Row Address and Element Address



A 2-D array element address

c [i][j]

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

= *(*(c+i)+j)

*(c+i) : i-th row address

((c+i)+j)
: i-th row, j-th column
element address

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

2-d array access via recursive pointers

```
int      x [4];
```

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

```
x[ i ]
```

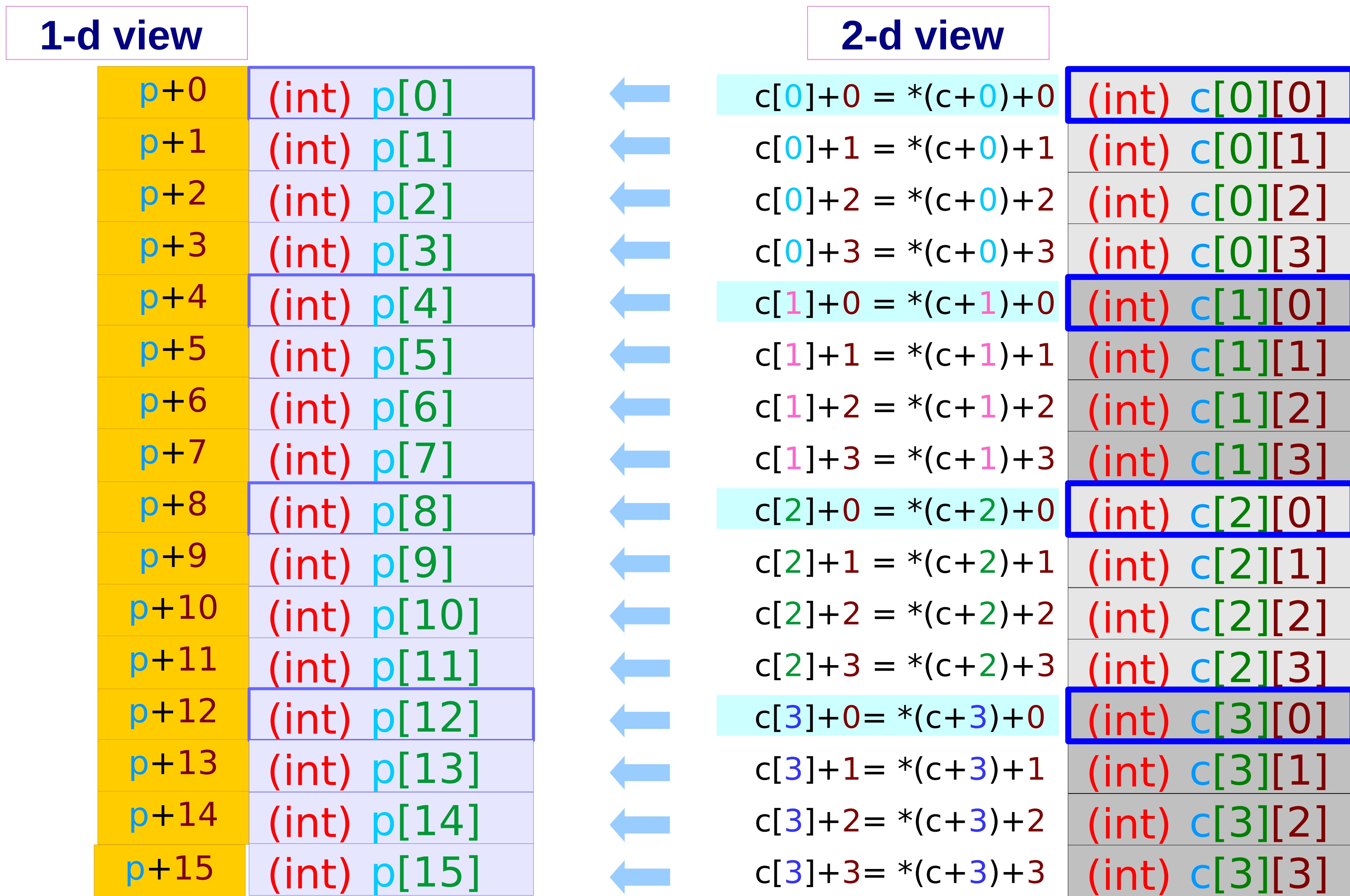
```
*(x+ i )
```

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

```
*(c[ i ]+ j)
```

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

A linearization of a 2-D array

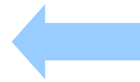


2-d array access via a single pointer

```
int     *p = (int *) c;
```

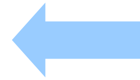
```
int c [4][4];
```

```
p[ i*4 + j ]
```



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

```
*(p+ i*4 + j )
```



```
*(c[ i ]+ j)
```

```
*(p +k)
```

```
i = k / 4;
j = k % 4;
```

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

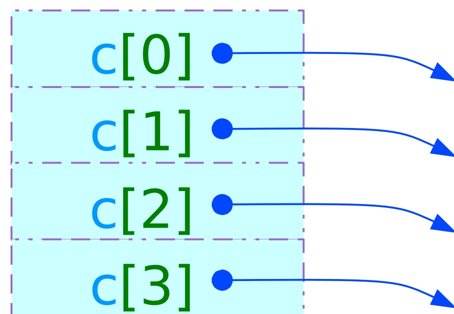
Static Allocation of a 2-d Array

int A [4][3];

A in %eax,
i in %edx,
j in %ecx

```
sall    $2, %ecx                ;; j * 4
leal    (%edx, %edx, 2), %edx   ;; i * 3
leal    (%ecx, %edx, 4), %edx   ;; j * 4 + i * 12
movl    (%eax, %edx), %eax      ;; read M[ X_A +4(3i +j) ]
```

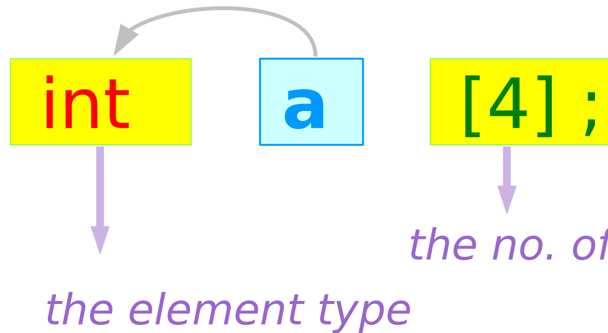
c[0]+0	*(c [0]+0)
c[0]+1	*(c [0]+1)
c[0]+2	*(c [0]+2)
c[1]+0	*(c [1]+0)
c[1]+1	*(c [1]+1)
c[1]+2	*(c [1]+2)
c[2]+0	*(c [2]+0)
c[2]+1	*(c [2]+1)
c[2]+2	*(c [2]+2)
c[3]+0	*(c [3]+0)
c[3]+1	*(c [3]+1)
c[3]+2	*(c [3]+2)



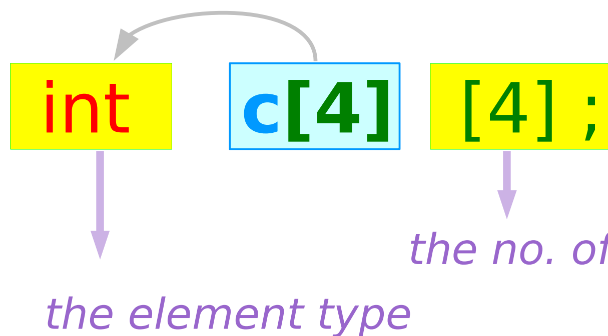
The intermediate array :
Not necessarily allocated
in memory

A recursive definition of a 2-d array

```
int      a [4];

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



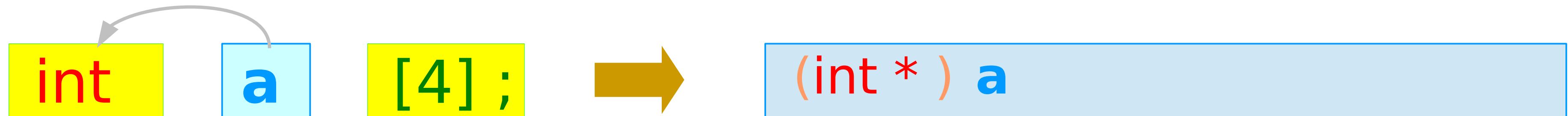
The array name **a** holds *the starting address of the 4 integer element array*



c[0], **c[1]**, **c[2]**, **c[3]** holds *the starting address of the 4 integer element array*

A 2-d array name as a double pointer

```
int        a [4];

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



a points to an integer data

c[i] points to an integer data
c points to an integer pointer

A 2-d array name as a pointer to an array

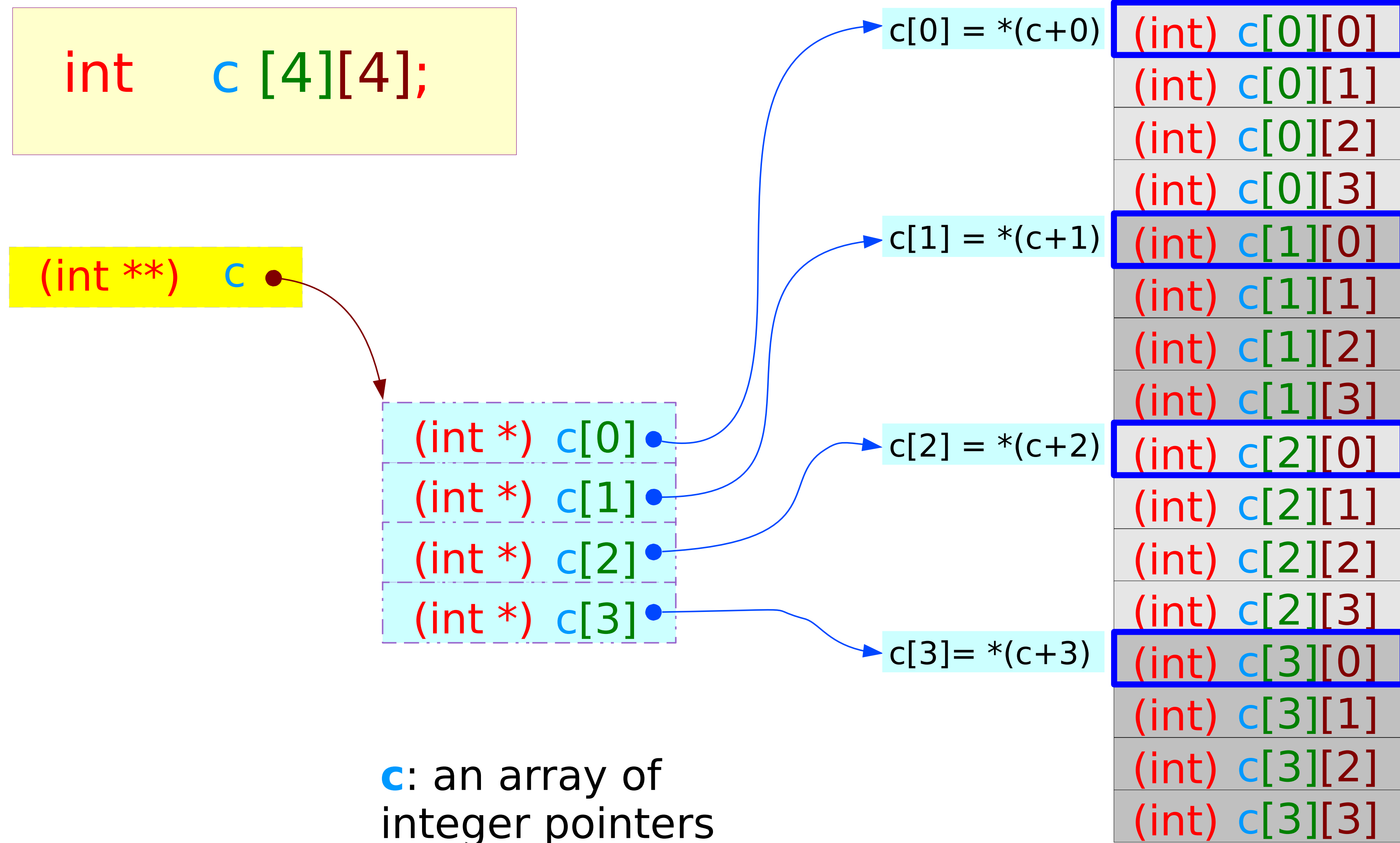
```
int        a [4];

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

a : an array name

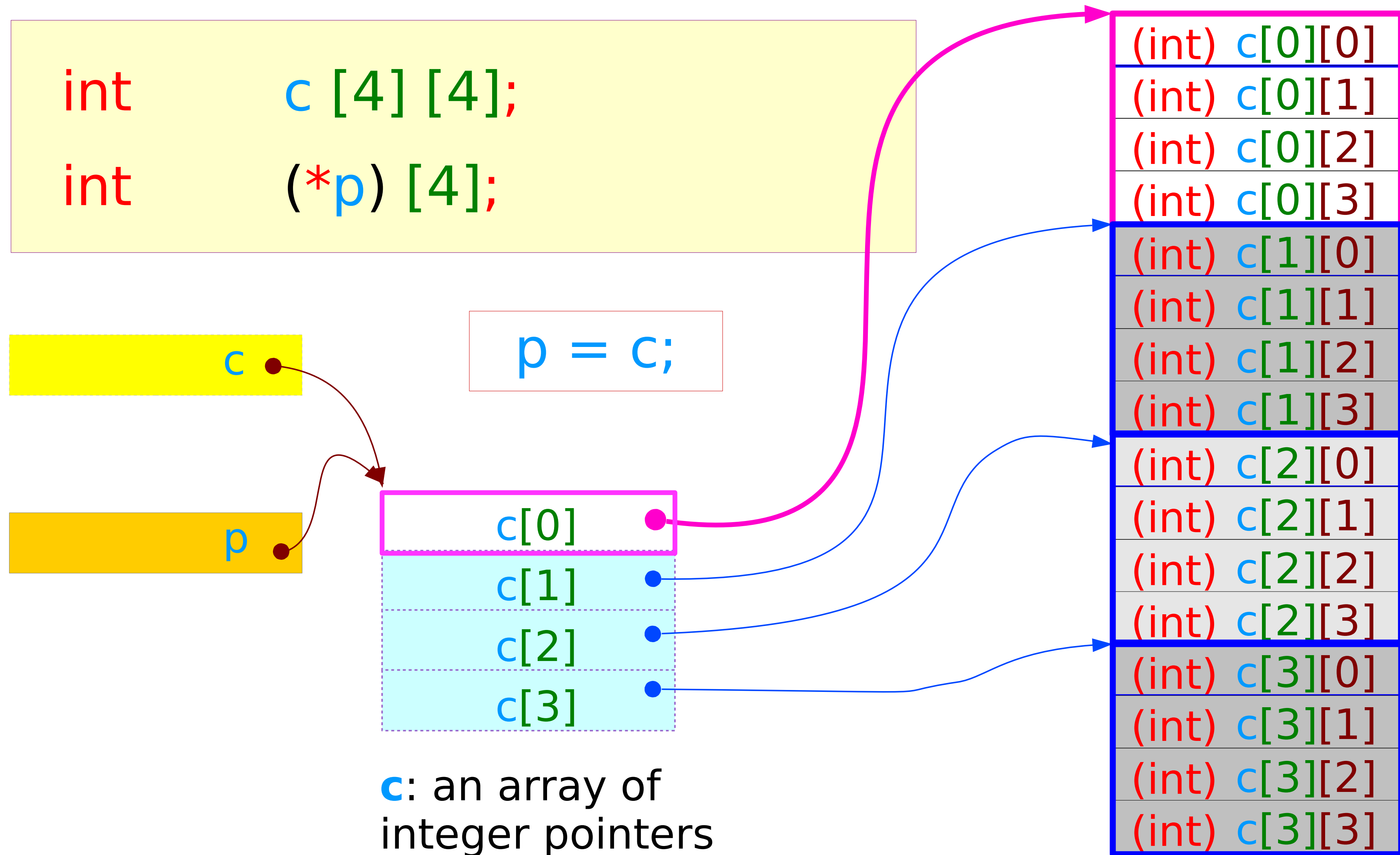
c[i] : array names

c : the intermediate array name

An intermediate array in a 2-d array

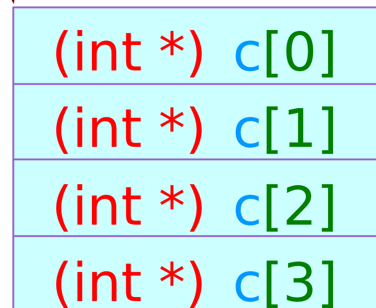


A 2-d array and a pointer to a 1-d array



2-d array dynamic allocation : method 1 (a)

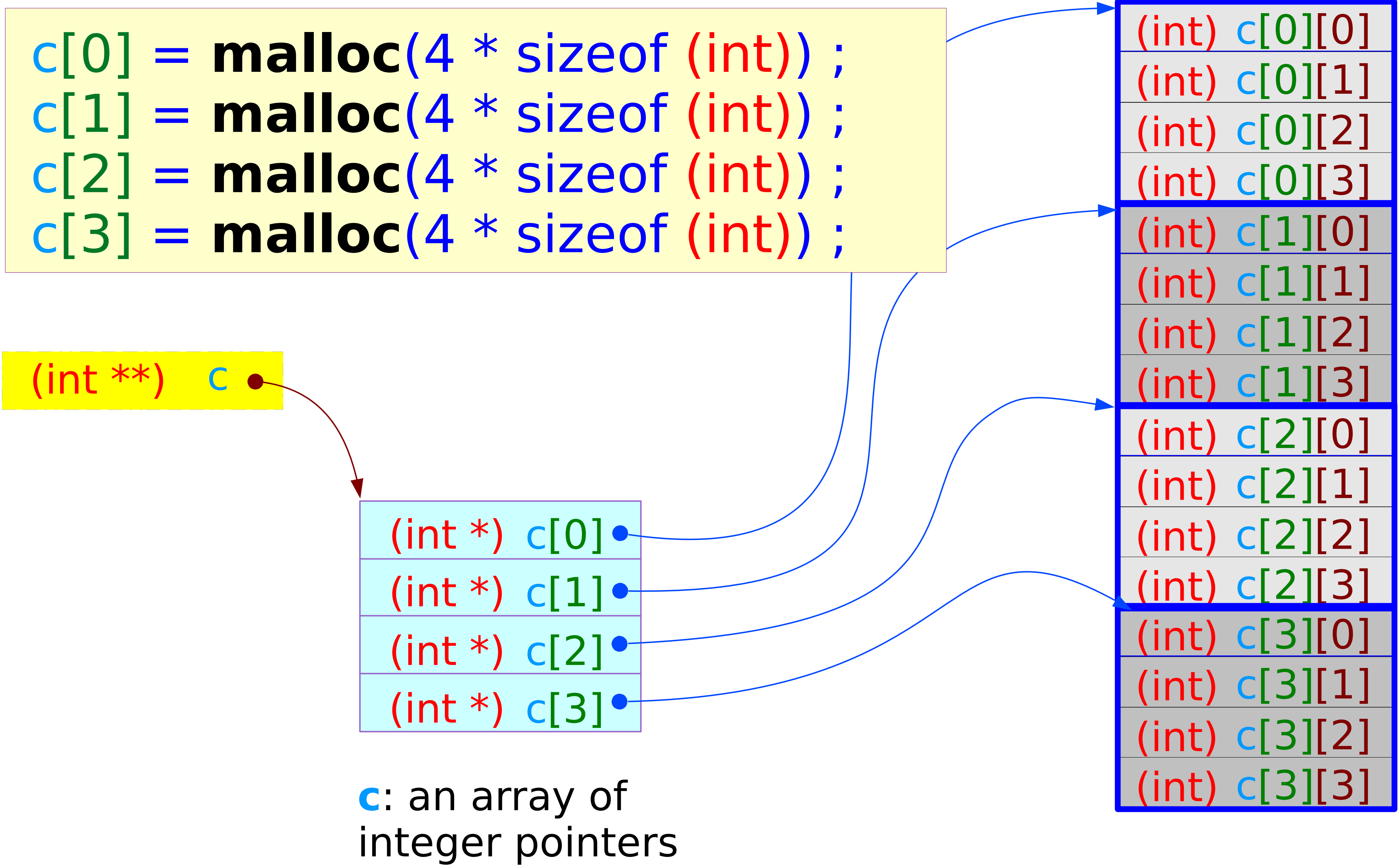
```
int    ** c ;

c = malloc(4 * sizeof (int *)) ;
```



The intermediate array :
Allocated physically
in memory

c: an array of
integer pointers

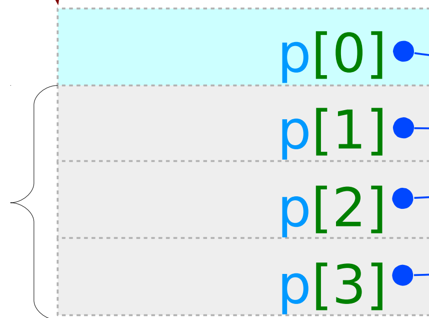
2-d array dynamic allocation : method 1 (b)

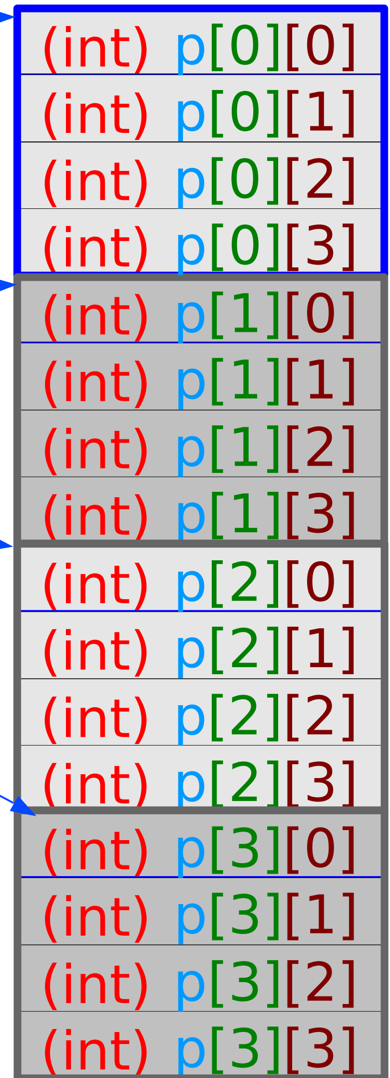


c: an array of integer pointers

2-d array dynamic allocation : method 2

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

p = malloc(4 * 4 * sizeof (int)) ;
```





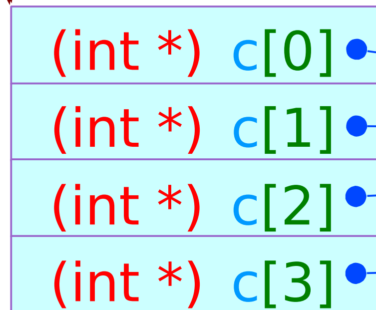
utilize pointer
addition property

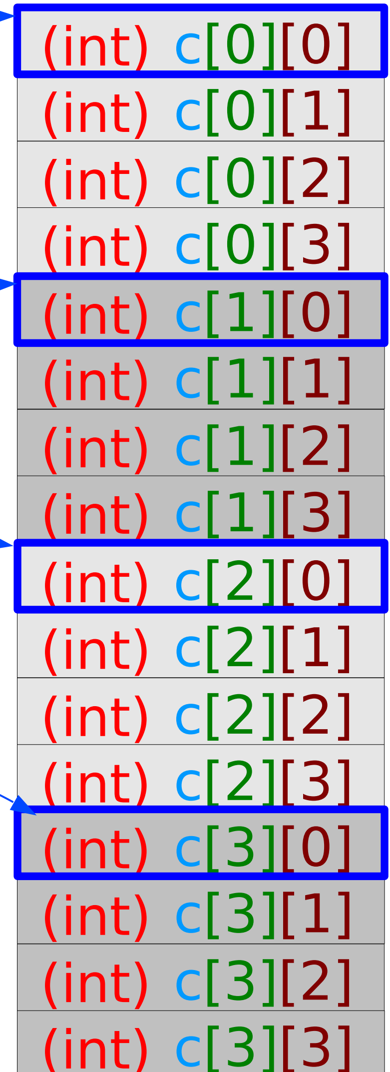
The intermediate array :
No physical allocation

2-d array dynamic allocation : method 3 (a)

```
int   ** c ;
int   *  p ;
c = malloc( 4 * sizeof(int *) ) ;
p = malloc( 4 * 4 * sizeof(int) ) ;
```

(int **) c



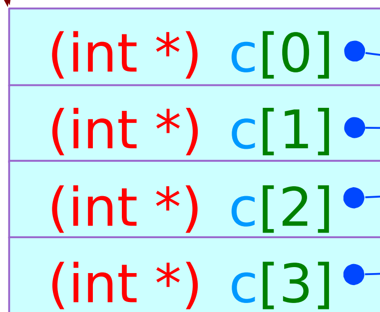


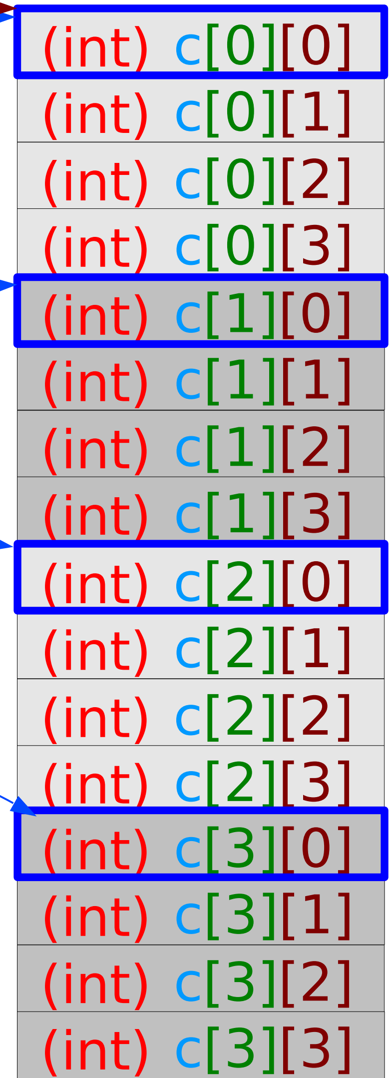
The intermediate array :
Allocated physically in memory

```
for (i=0; i<M; i++)
    c[i] = p + i*N;
```

(int *) p

(int **) c





c: an array of
integer pointers

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