Arrays (1A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Young Won Lim
12/4/17

Calculating the Mean of n Numbers

The mean of N numbers

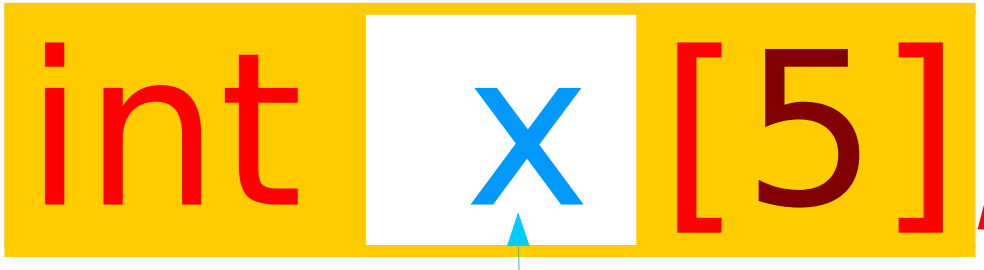
$$m = \frac{1}{N}\sum_{i=0}^{N-1} x_i$$

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

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]
```

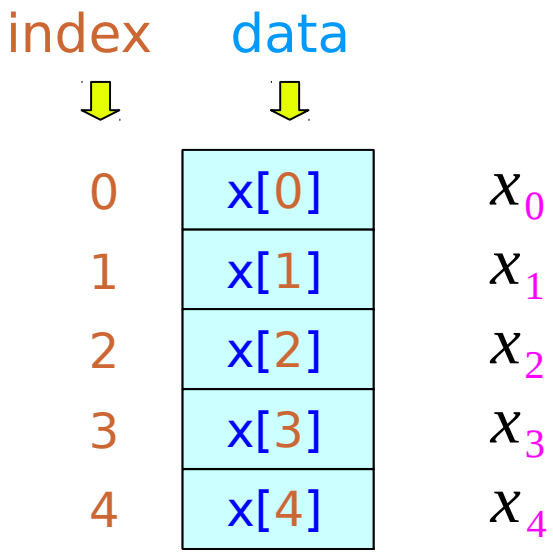
i = 0, ... , 4

Accessing array elements – using an index

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

x is an array
with 5 integer elements

5 int variables



Accessing array elements – using an address

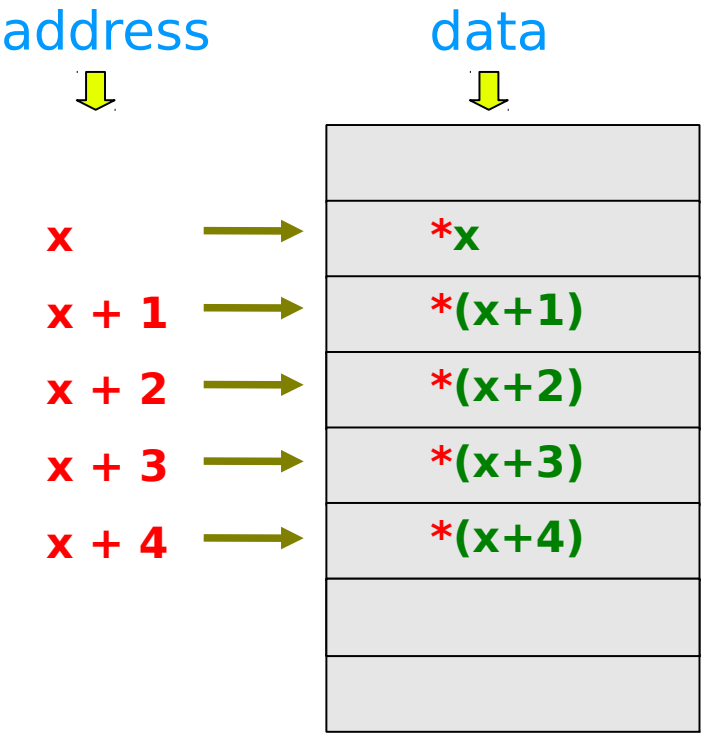
```
int     x[5];
```

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

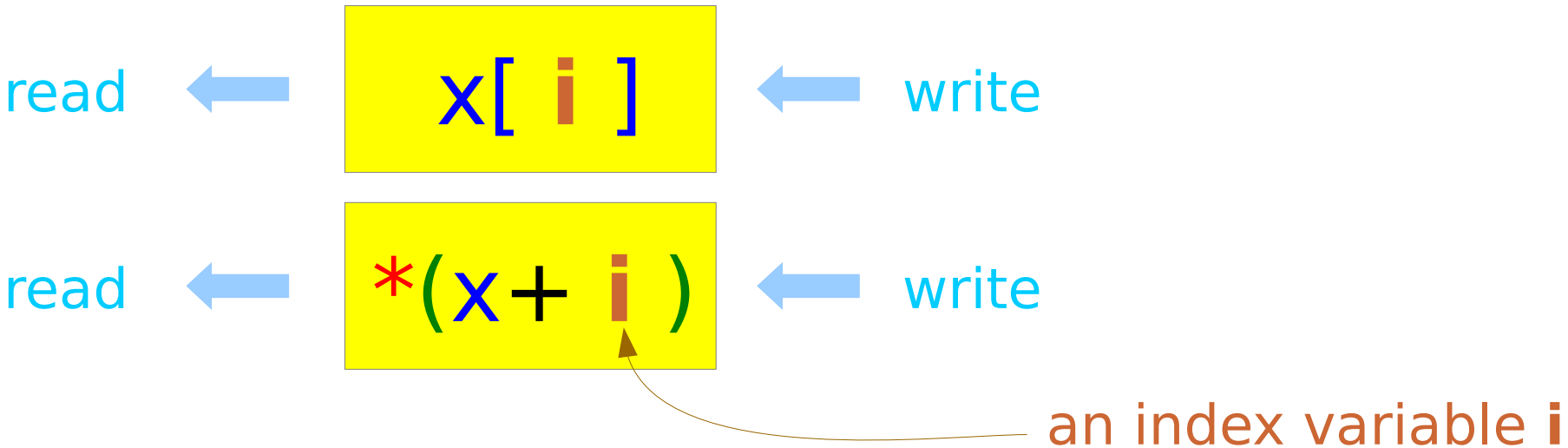
x : the starting address
x+i : the (i+1)-th element's address
***(x+i)** : the (i+1)-th element value

A variable expressed by another variable

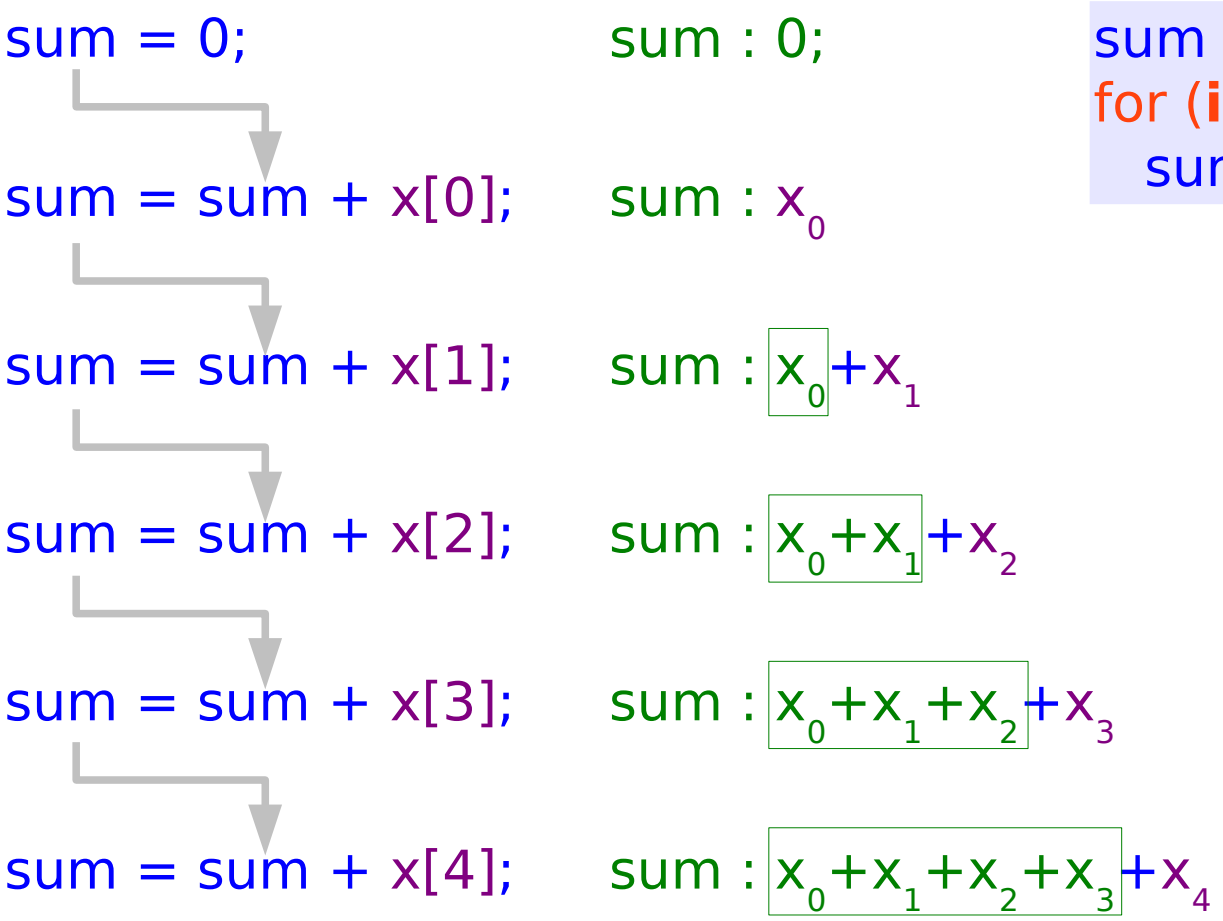
```
int     x[5];
```

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

treated as a variable

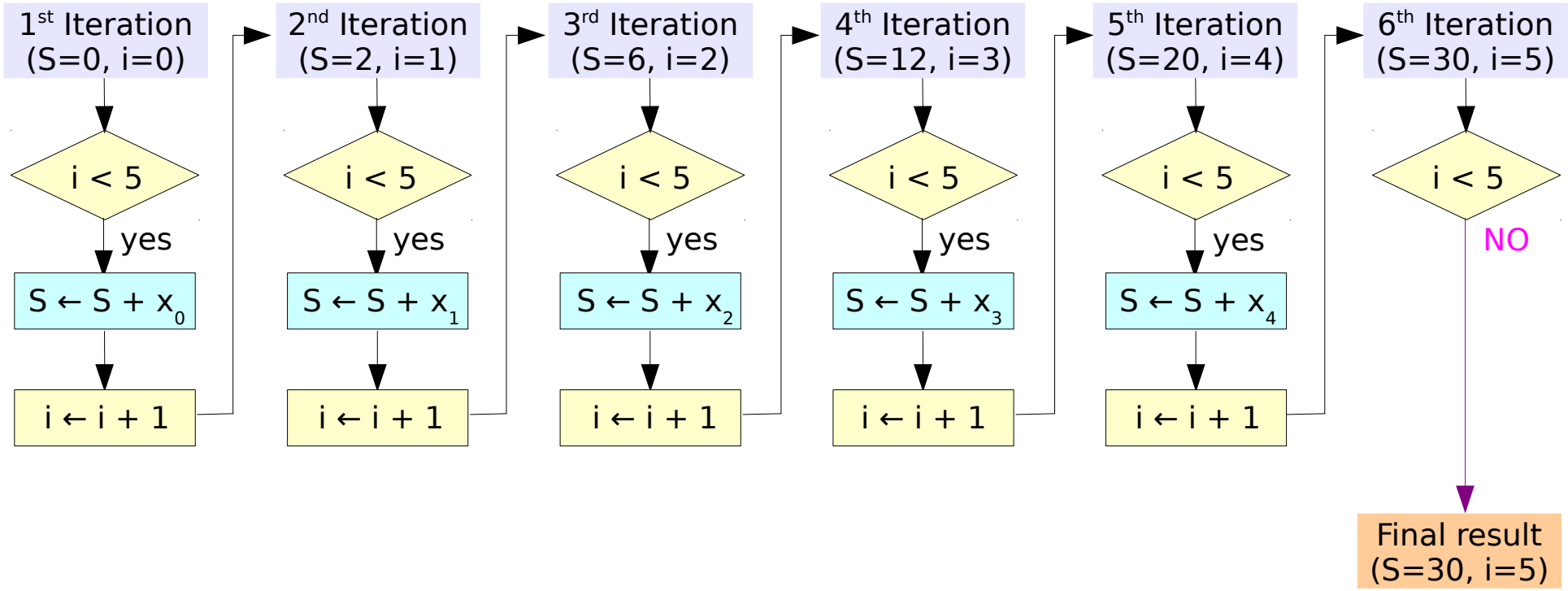


Computing the sum of n numbers (1)




```
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) = 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

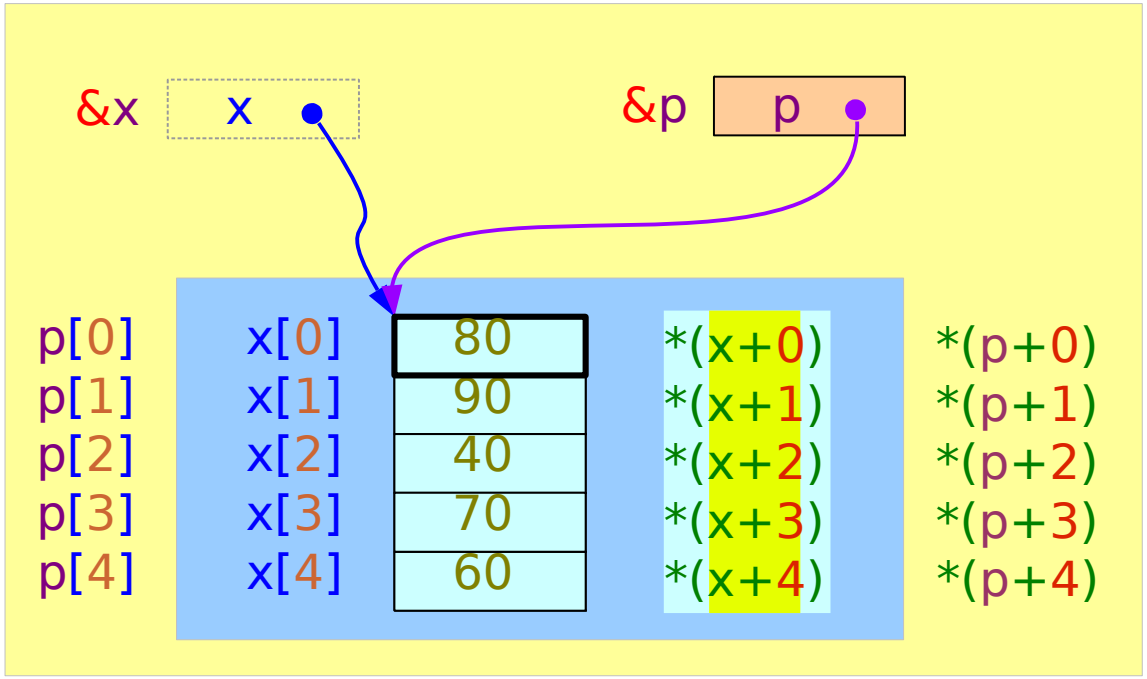
~~`int b [ ] ;`~~

```
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;
```



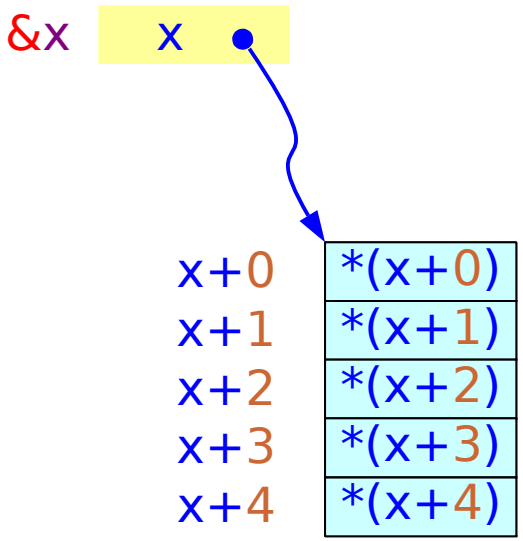
x is a constant symbol
cannot be changed

p is a variable
can point to other addresses

An Array Name and a Pointer Variable

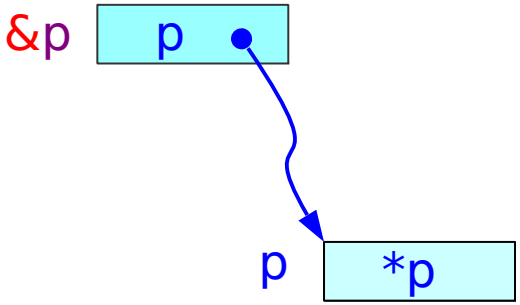
```
int x [5] ;
```

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



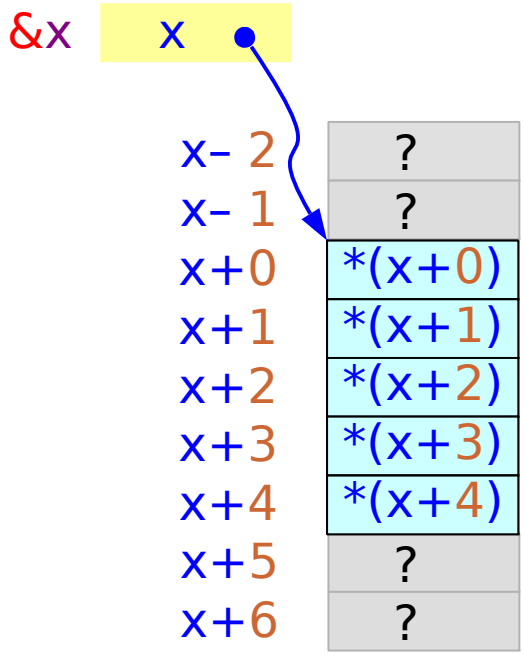

```
int  * p ;
```

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

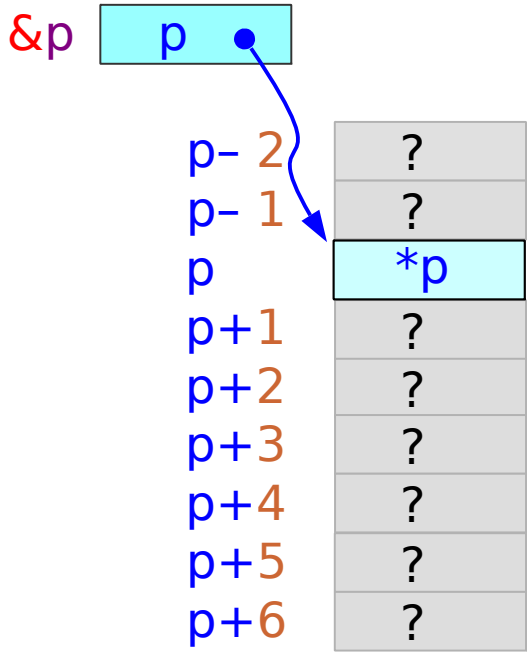


Out of range index

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



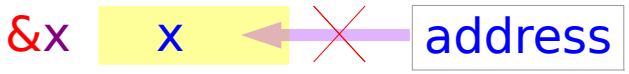

```
int  * p ;
```



A programmer’s responsibility

Assignment of an address

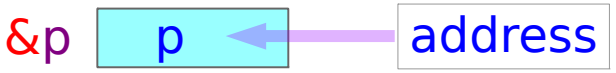
```
int x [5] ;
```



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

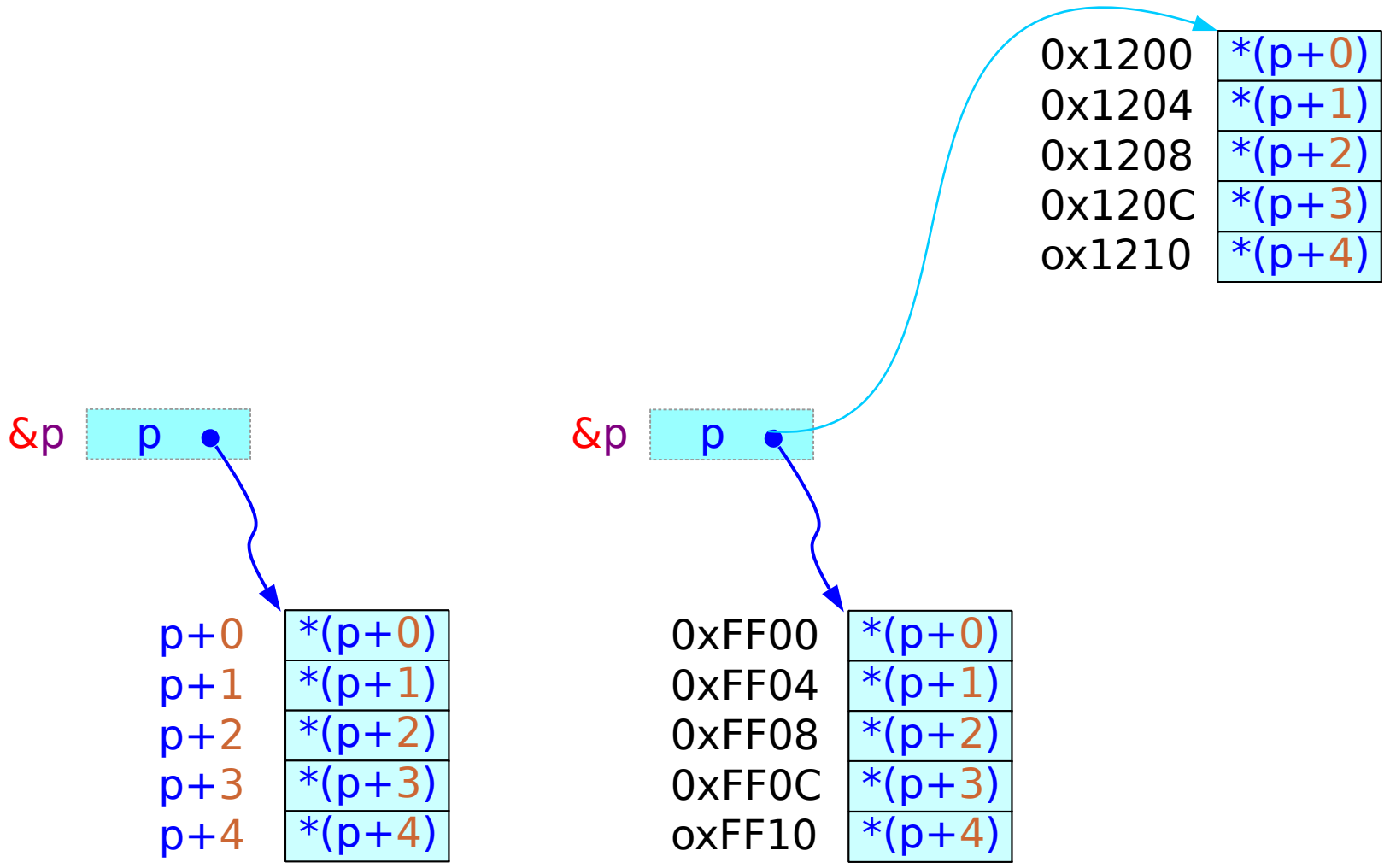
This address is embedded as a constant in the executable file and cannot be changed

```
int  * p ;
```



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

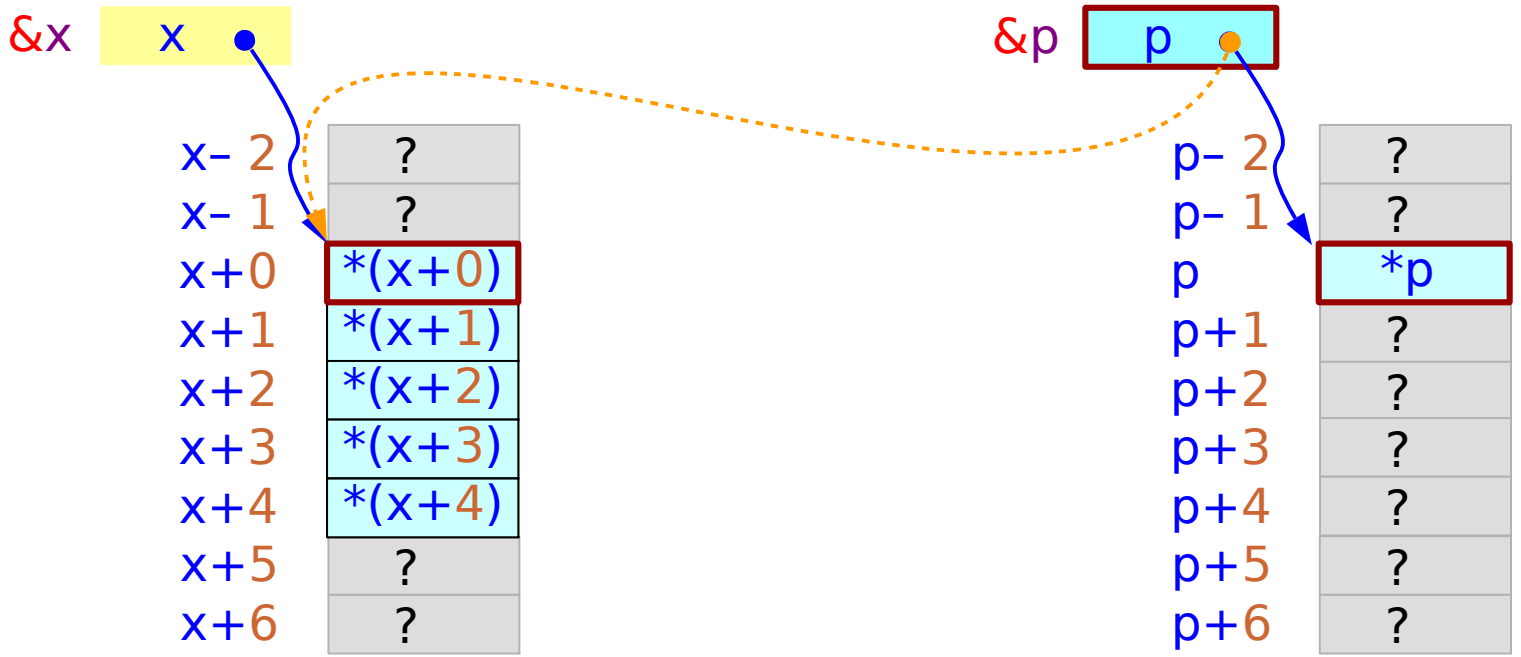
Pointer variable can point different locations



Pointer to an integer

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

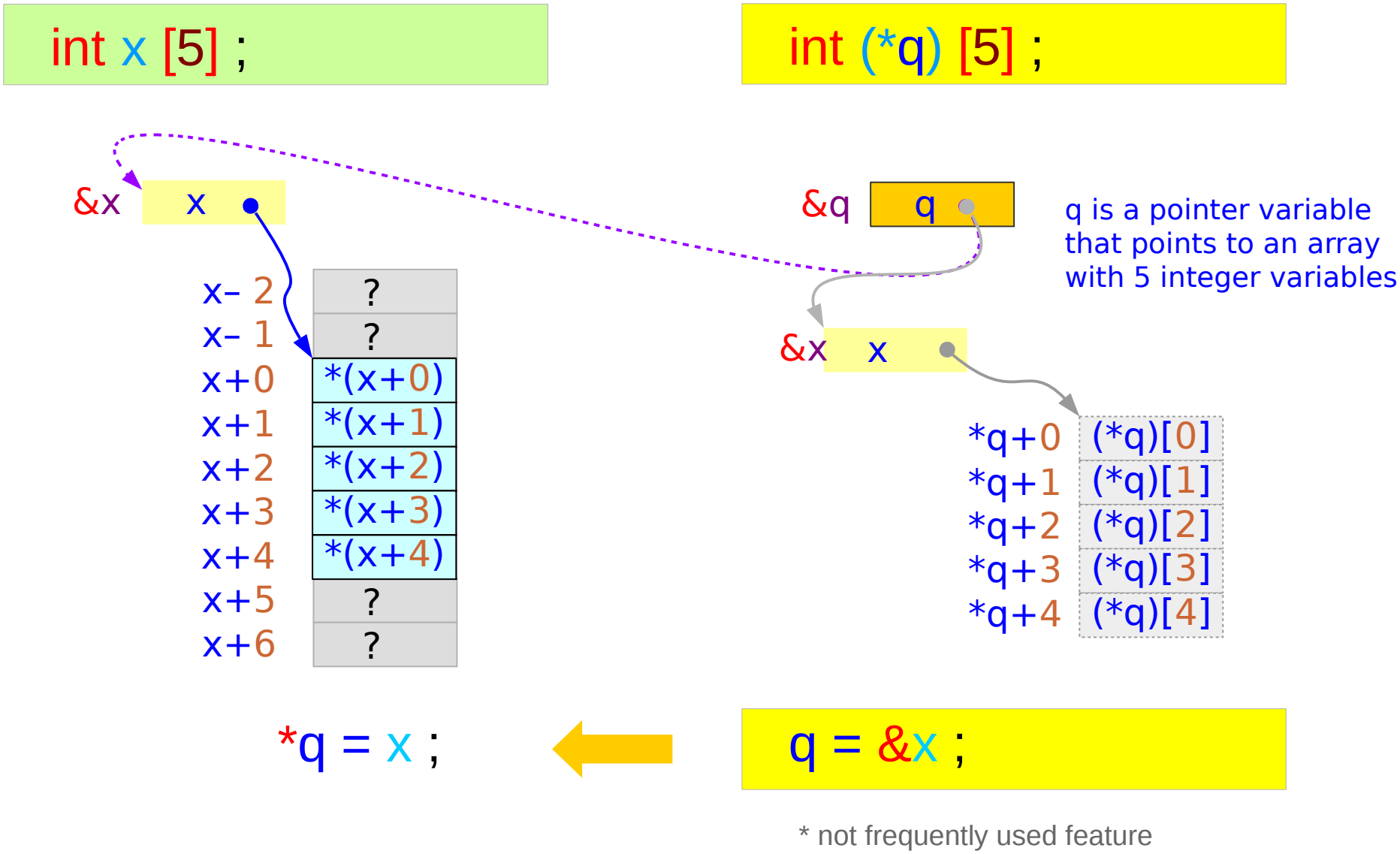
```
int  * p ;
```




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

```
p = x ;
```

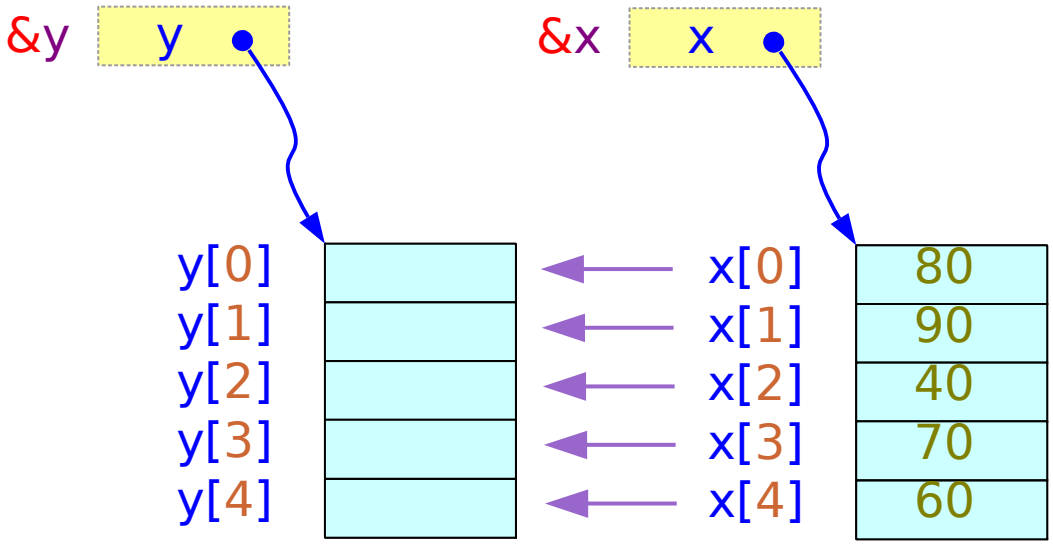
Pointer to an array



* not frequently used feature

Copying an Array to another Array

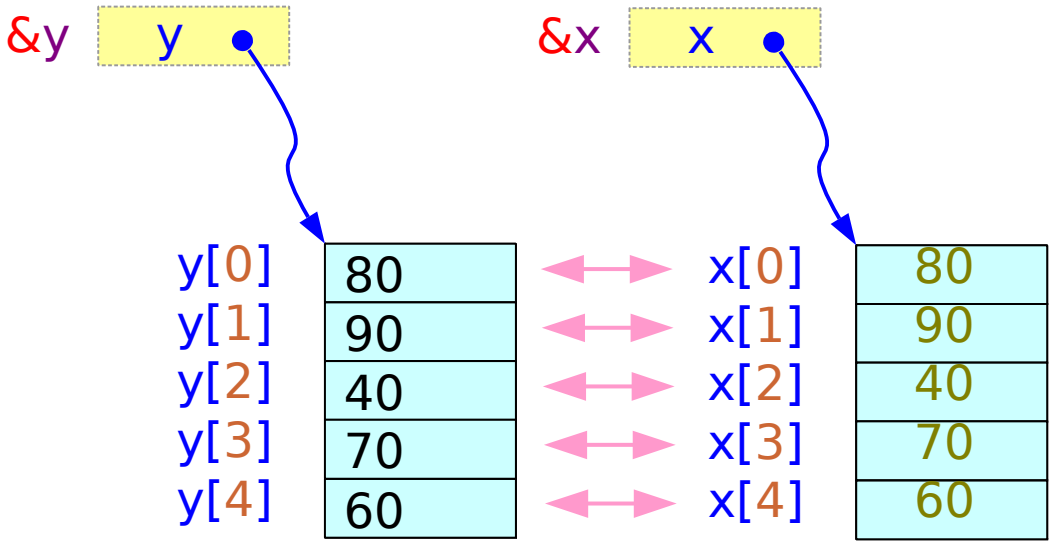
```
int x [5] = { 1, 2, 3, 4, 5 };
int y [5] ;
y = x;
```




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

Comparing an Array with another Array

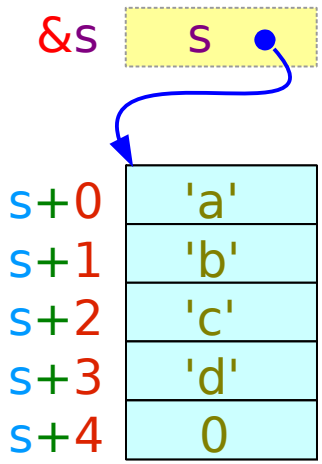
```
int x [5] = { 1, 2, 3, 4, 5 };
int y [5] = { 1, 2, 3, 4, 5 };
x == y
```




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

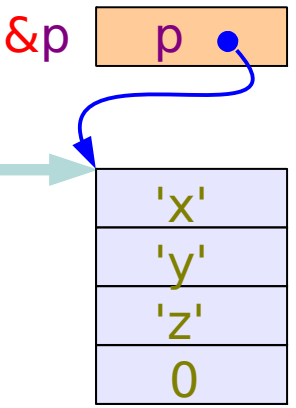
Initialized Character Arrays and Pointers (1)

```
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

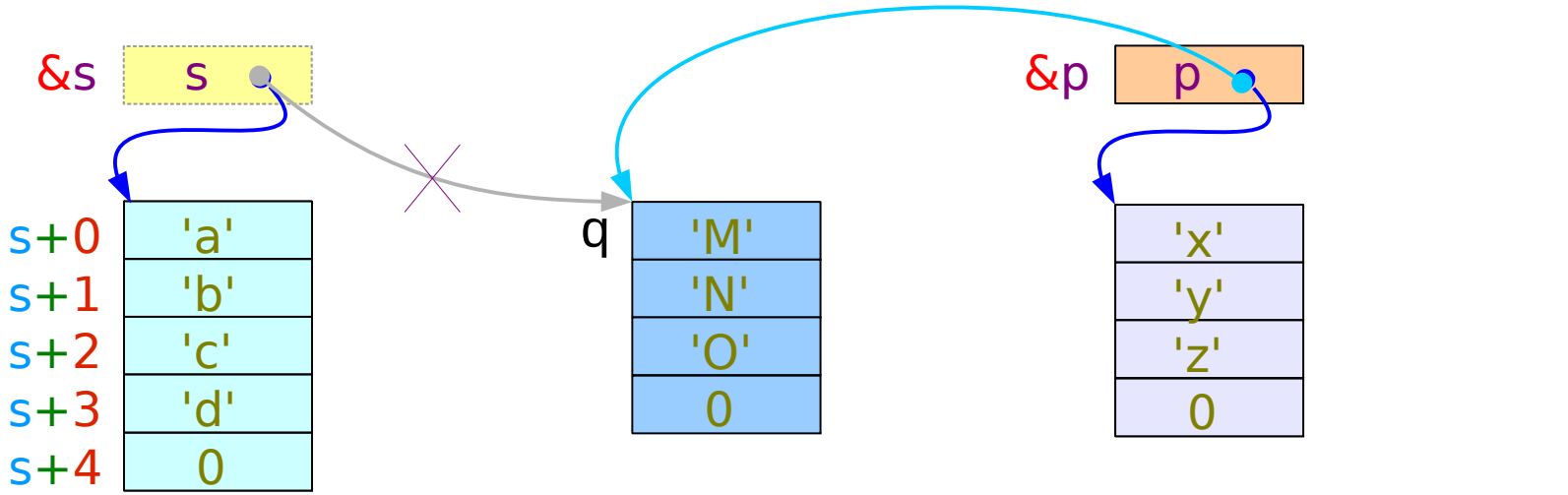
```
*s = 'm';
s[0] = 'm';
```

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”;
```

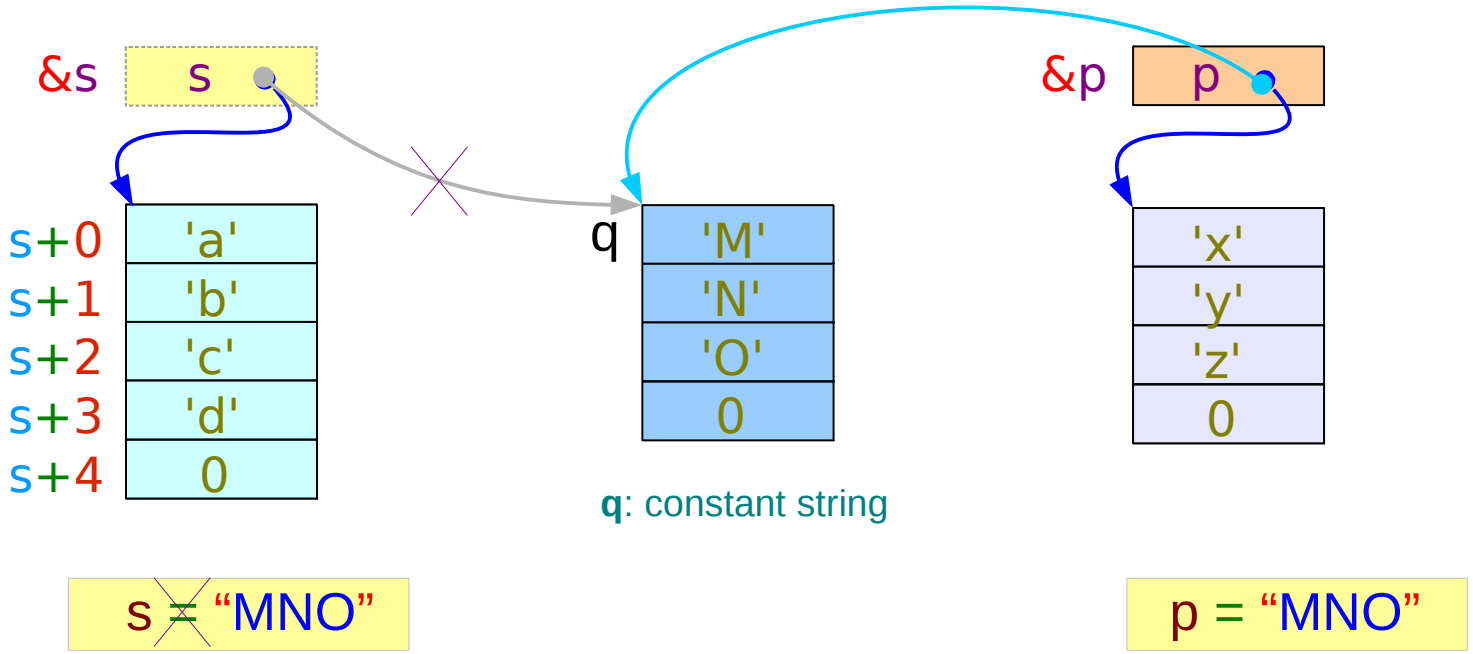


s cannot be assigned ~~s = q;~~

p can be assigned p = q;

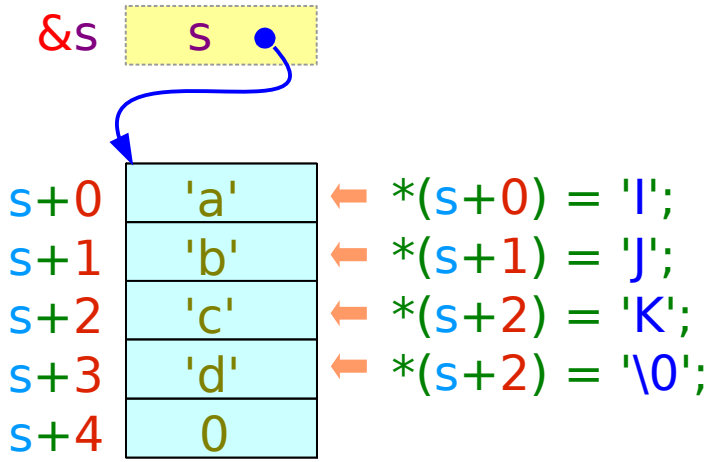
Assigning a constant character string

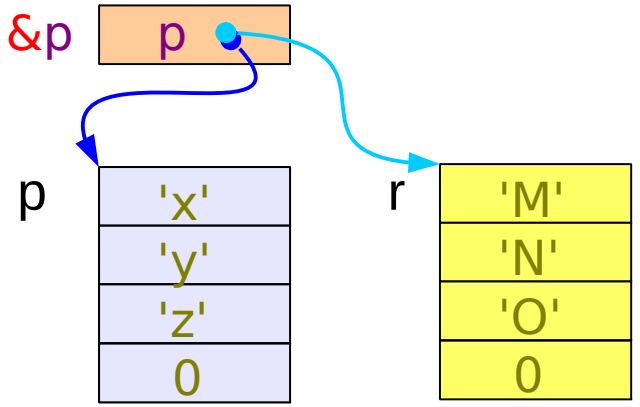
```
char s [5] = { 'a', 'b', 'c', 'd', 0 };
char s [5] = “abcd”;

char  *p = “xyz”;
```



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”);

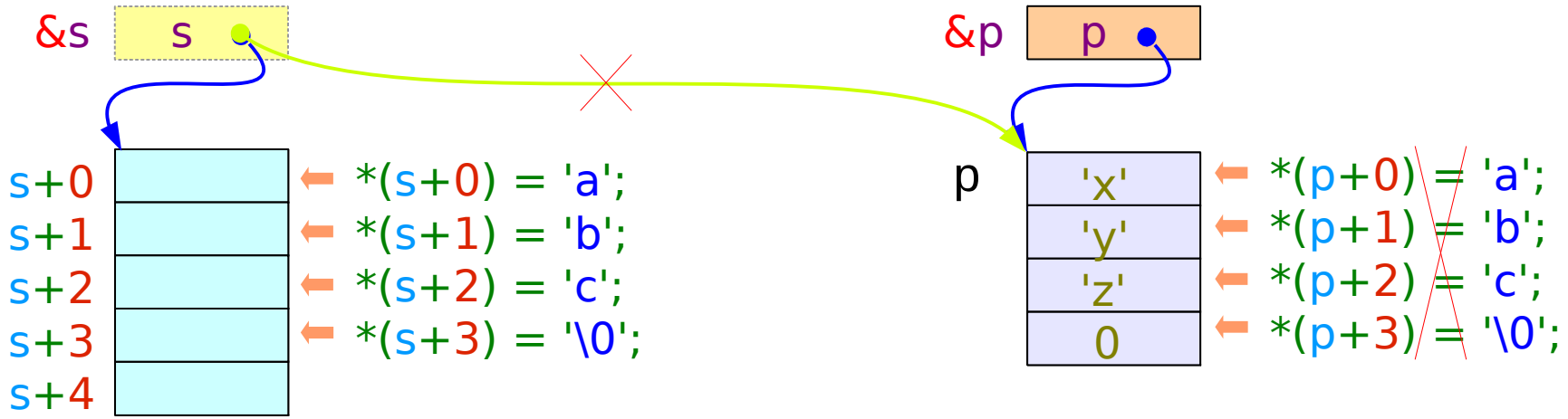
Uninitialized Character Arrays and Pointers

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

```
s = "xyz";          char * const s
p = "xyz";          const char * p
strcpy (s, "abc");
strcpy (p, "abc");
```

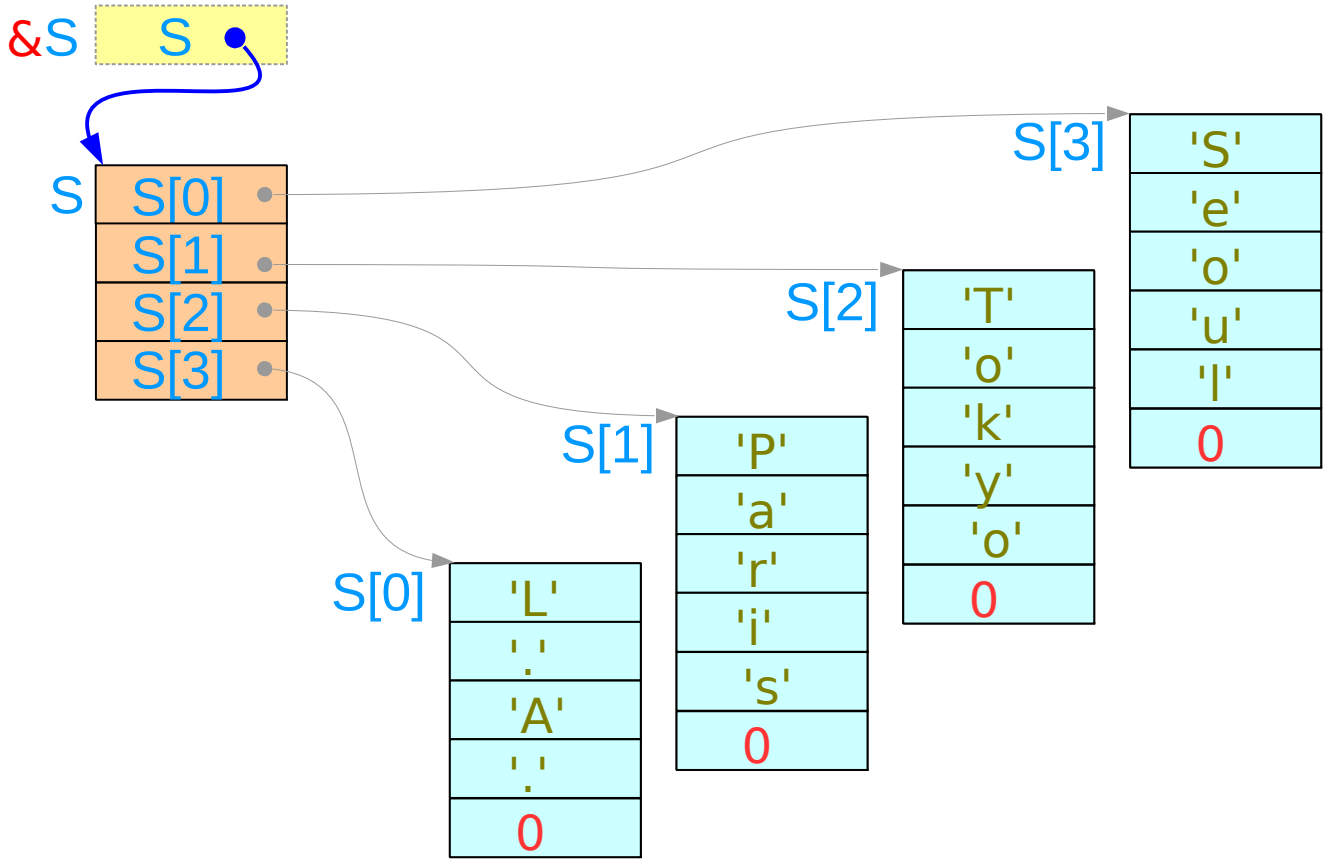
s cannot point to other location

p points to a string constant
which cannot be changed



Arrays of Pointers

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



A[] Notation

1. An array definition with initializers

int x [ ] = { 1, 2, 3 }; ⇒ int x [3]

2. A formal parameter definition in a function

func(int x [ ]) { … } ⇒ int * const x (x : a constant)

compatible ⇒ int * p (p : a variable)

A[][n] Notation

1. An array definition with initializers

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

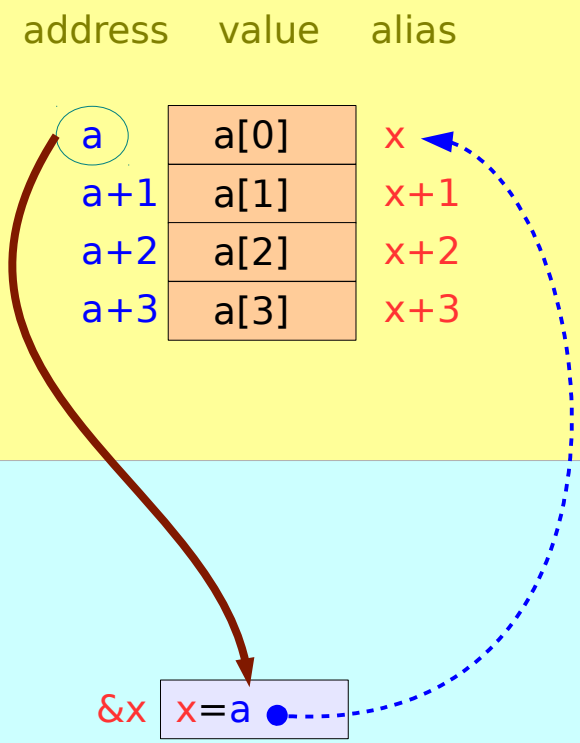
2. A formal parameter definition in a function

func(int x [ ][3]) { … } ⇒ int (* const x)[3] (constant)

not compatible ⇒ int ** p (variable)

Passing 1-d Arrays

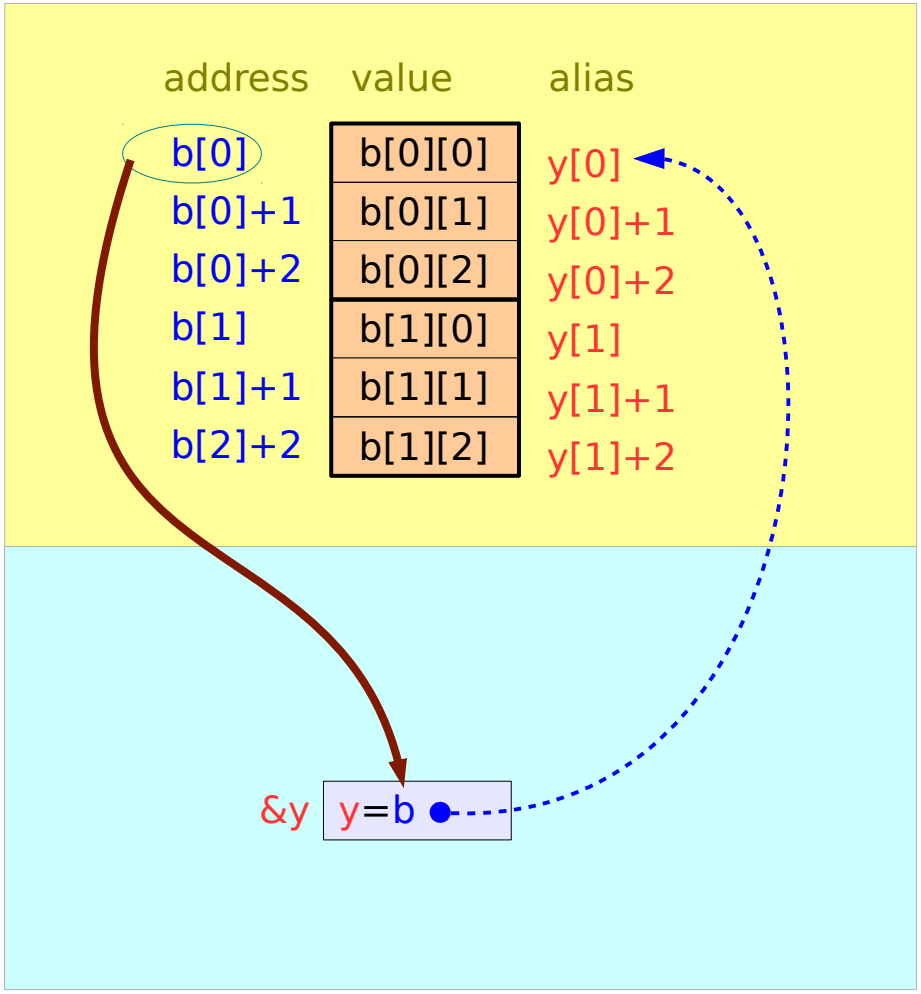
```
int a [ ] = { 1, 2, 3, 4 };

func( a );

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



Passing 2-d Arrays

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

func( b );
```

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



Passing an individual element by value

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

func( a[3] );
```

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

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

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

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

Passing an individual element by reference

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

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

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

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

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

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

Array Type definition

```
typedef int AType [10] ;

AType A;
```

≡

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

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

A [m] = 400;
```

Pointer to Array Type definition

```
typedef int AType [10] ;

AType A,  *q;
```

```
q = &A ;
```

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

PType p;
```

```
p = &A ;

p = q ;
```

2-D Array Definition

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

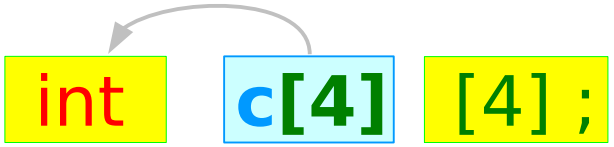
	col 0	col 1	col 2	col 3
row 0	c [0][0]	c [0][1]	c [0][2]	c [0][3]
row 1	c [1][0]	c [1][1]	c [1][2]	c [1][3]
row 2	c [2][0]	c [2][1]	c [2][2]	c [2][3]
row 3	c [3][0]	c [3][1]	c [3][2]	c [3][3]

row major ordering

Pointer to the start of 1-d arrays

```
int     a [4];

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

a points to *a 4 integer element array*

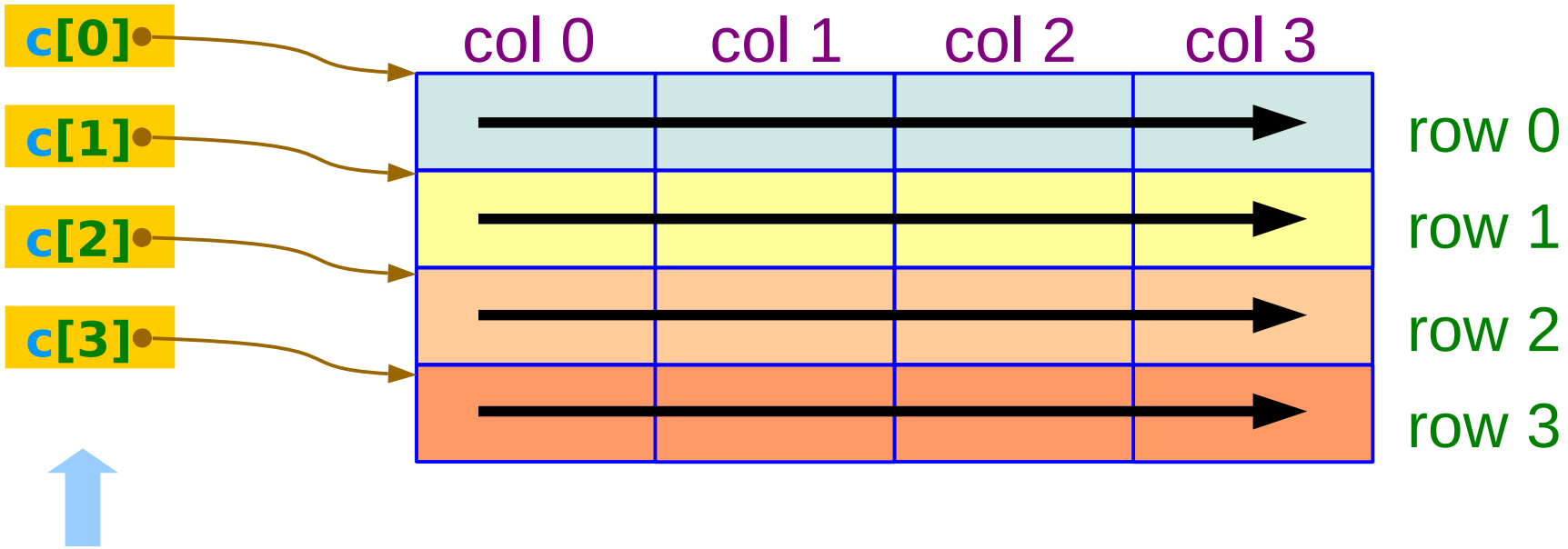


each of **c[0]**, **c[1]**, **c[2]**, **c[3]**
points to *a 4 integer element array*

Row Major Ordering

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

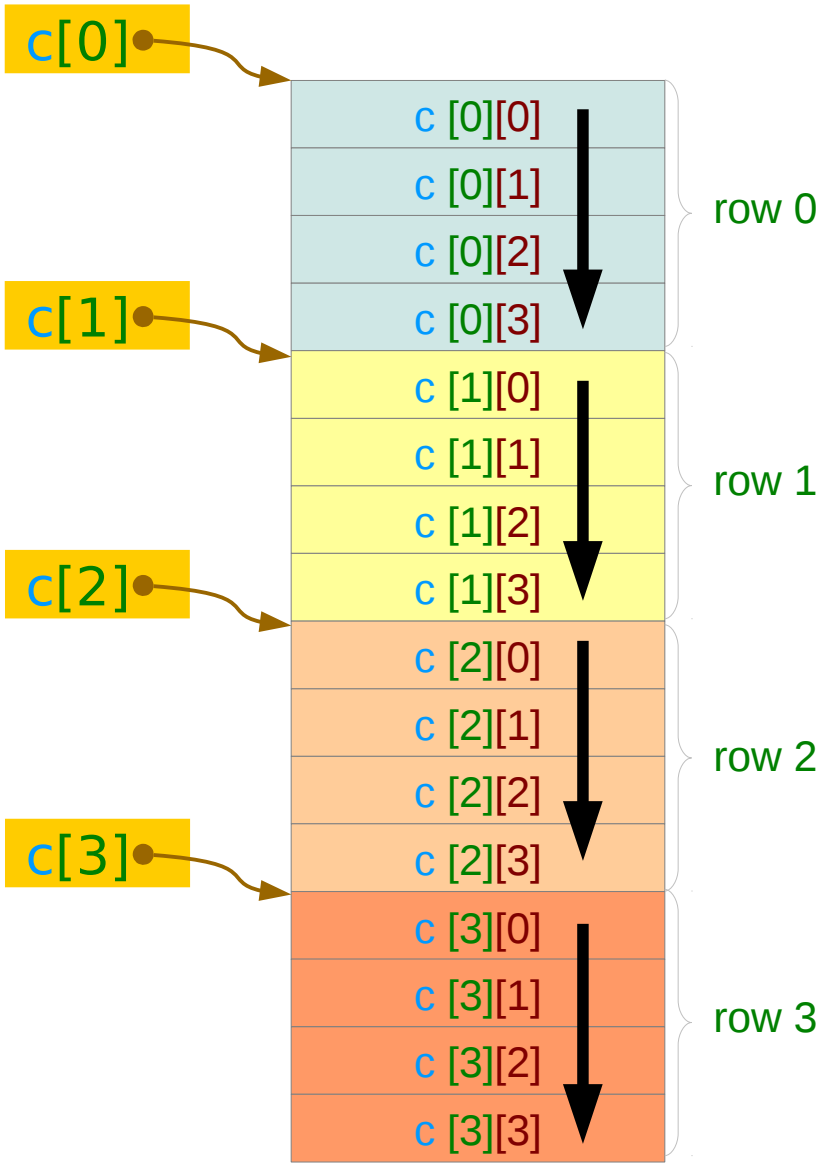
row major ordering

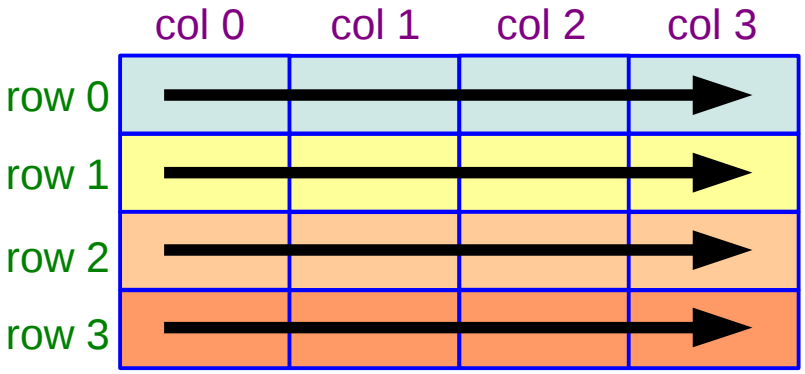


Consider each c[i] as the name of an array
that has 4 integer elements

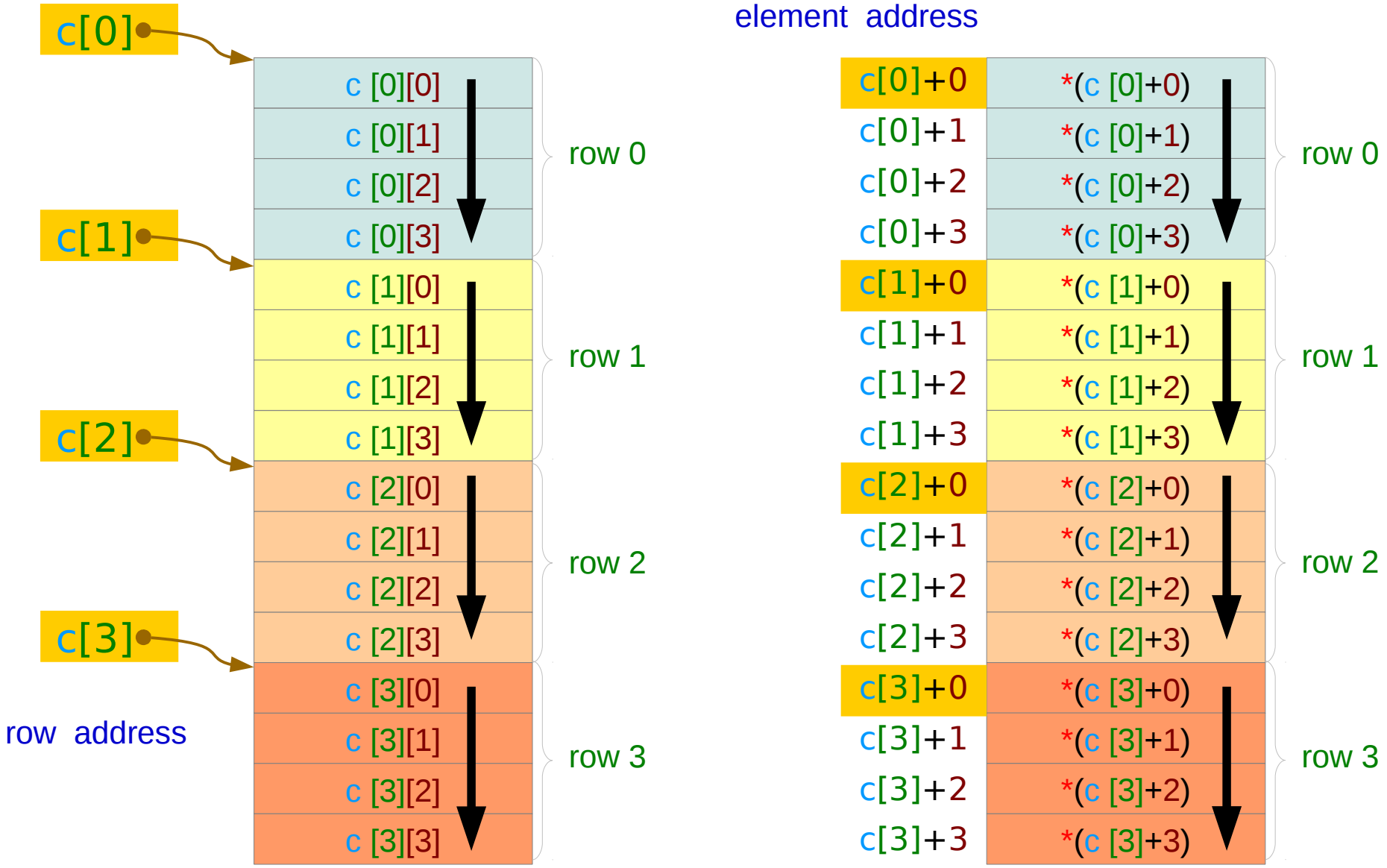
Linear Array Memory Layout

```
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) : row address

((c+i)+j) : element address

first select a row, then a column

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]

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

Young Won Lim
12/4/17