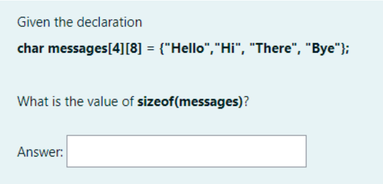
YORK U UNIVERSITÉ UNIVERSITY

EECS2031 Software Tools

F 2021

Oct 27/28 Lecture 13 29/1 Lecture 14

SMQ2

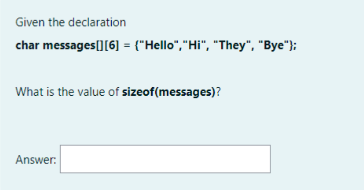


Given the declaration

char messages[4][8] = {"Hello","Hi", "There", "Bye"};

What is the value of **sizeof(messages)**?

Answer:

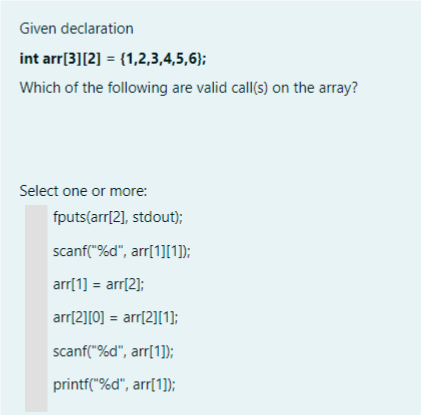


Given the declaration

char messages[][6] = {"Hello","Hi", "They", "Bye"};

What is the value of **sizeof(messages)**?

Answer:



Given declaration

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

Which of the following are valid call(s) on the array?

Select one or more:

fputs(arr[2], stdout);

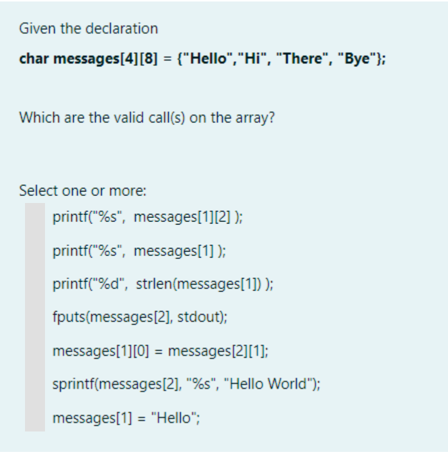
scanf("%d", arr[1][1]);

arr[1] = arr[2];

arr[2][0] = arr[2][1];

scanf("%d", arr[1]);

printf("%d", arr[1]);



Given the declaration

char messages[4][8] = {"Hello","Hi", "There", "Bye"};

Which are the valid call(s) on the array?

Select one or more:

printf("%s", messages[1][2]);

printf("%s", messages[1]);

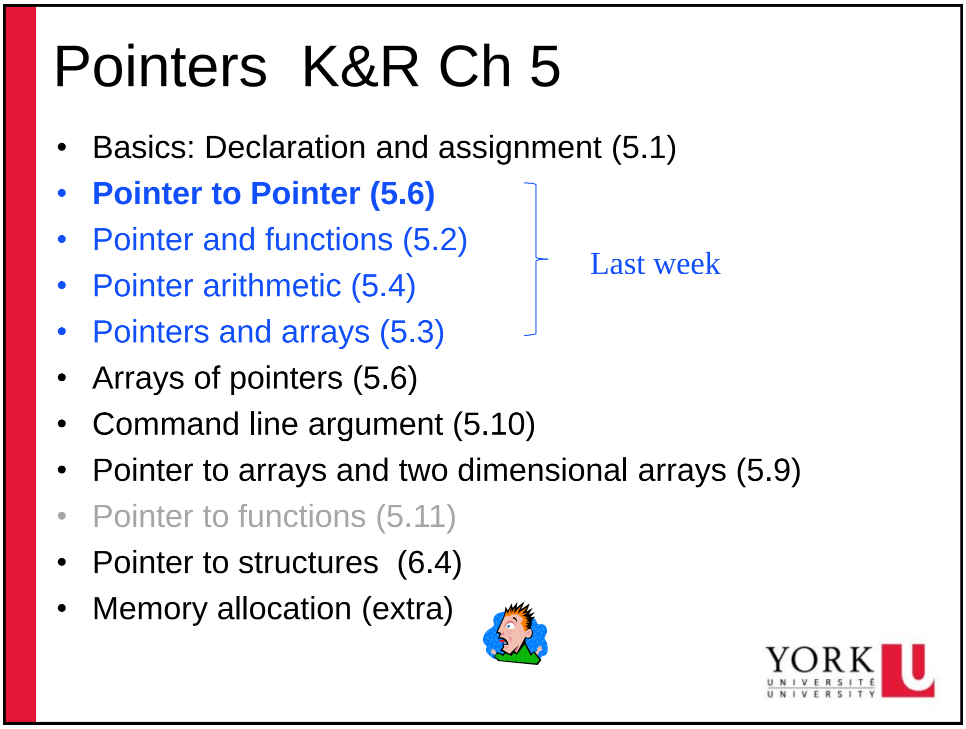
printf("%d", strlen(messages[1]));

fputs(messages[2], stdout);

messages[1][0] = messages[2][1];

sprintf(messages[2], "%s", "Hello World");

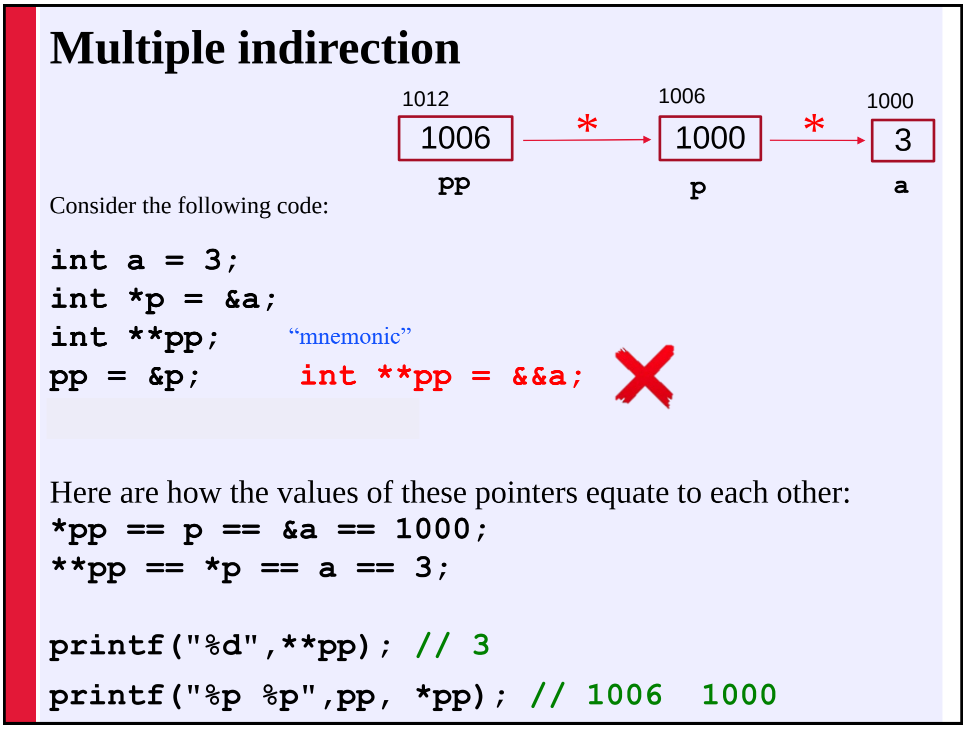
messages[1] = "Hello";



Pointers K&R Ch 5

- Basics: Declaration and assignment (5.1)
- **Pointer to Pointer (5.6)**
- Pointer and functions (5.2)
- Pointer arithmetic (5.4)
- Pointers and arrays (5.3)
- Arrays of pointers (5.6)
- Command line argument (5.10)
- Pointer to arrays and two dimensional arrays (5.9)
- Pointer to functions (5.11)
- Pointer to structures (6.4)
- Memory allocation (extra)

Last week



Multiple indirection

1012
1006
pp
*
1006
1000
p
*
1000
3
a

Consider the following code:

```
int a = 3;
int *p = &a;
int **pp;
pp = &p;
```

“mnemonic”

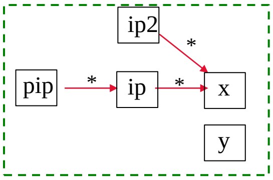
```
int **pp = &&a;
```

✗

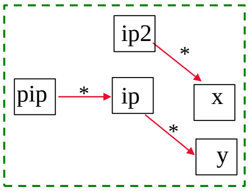
Here are how the values of these pointers equate to each other:

```
*pp == p == &a == 1000;
**pp == *p == a == 3;

printf("%d",**pp); // 3
printf("%p %p",pp, *pp); // 1006  1000
```

More Examples



```
int x = 1, y = 2;
int *ip, *ip2;

ip = &x;

int **pip      // I am a pointer to pointer
pip = &ip;    // pip points to ip

y = **pip;    // y=x      y is now 1
(**pip)--;   //  x--;  x is 0
```



```
ip = &y;
(**pip)--;  //  y--     y is 0 */
```

```
ip2 = pip; ???  Not valid!
pip = ip2; ???  Not valid!
```

Pointers K&R Ch 5

- Basics: Declaration and assignment (5.1)
- Pointer to Pointer (5.6)
- **Pointer and functions (5.2)**
- Pointer arithmetic (5.4)
- Pointers and arrays (5.3)
- Arrays of pointers (5.6)
- Command line argument (5.10)
- Pointer to arrays and two dimensional arrays (5.9)
- Pointer to functions (5.11)
- Pointer to structures (6.4)
- Memory allocation (extra)

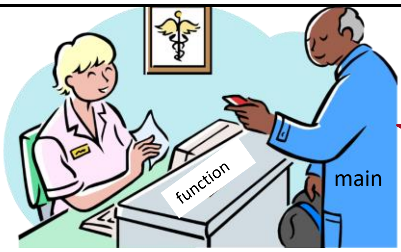
Last week

Pointers and function arguments

- In C, all functions are **called by value**
  - Value of the arguments are passed to functions, but not the arguments themselves (i.e., not ~~call by reference~~)
  - How to modify the arguments? `increment()` `swap()`
  - How to pass a structure such as array?

- Modify an actual argument by passing its address/pointer
  - Possibly modify passed arguments via their address
  - Efficient.

Send your friend a link to your file, instead of attachment, for editing



Mr. Main

Hi, function, I have some manuscripts, stored in lockers (memory), and I want you to repaint them so their color changed.

Ms function

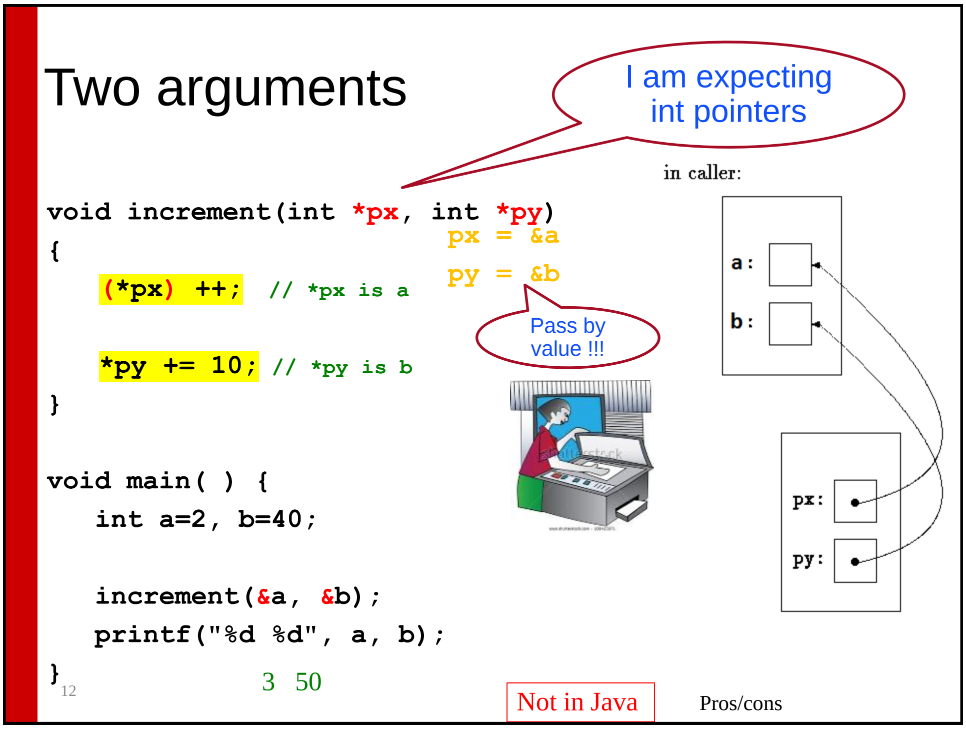
Hi, Mr Main, here is how we function work:
We don't take your original manuscript over the counter (not pass by reference). We always photocopy things (pass by value) passed here, and work on copies.

Mr. Main

Then, is there a way to have my original manuscripts' color changed?

Ms function

**Write down the locker number (address) on a paper, bring that paper to us (pass pointer/address) .
We photocopy the paper (still pass by value),
Then, based on the locker number on the copy, we go to your locker, fetch your original manuscripts there and work on them!**



Two arguments

I am expecting int pointers

```
void increment(int *px, int *py)
{
    (*px) ++;  // *px is a

    *py += 10; // *py is b
}

void main( ) {
   int a=2, b=40;

   increment(&a, &b);
   printf("%d %d", a, b);
}
```

px = &a

py = &b

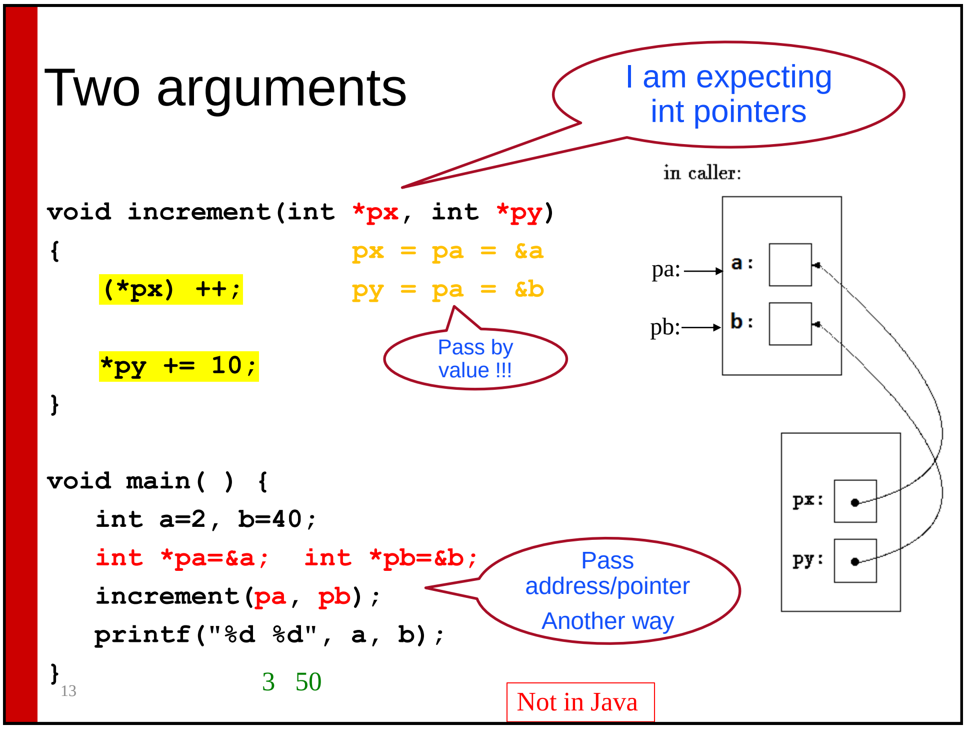
Pass by value !!!

3 50

Not in Java

Pros/cons

12



Two arguments

I am expecting int pointers

```
void increment(int *px, int *py)
{
    (*px) ++;

    *py += 10;
}

void main( ) {
   int a=2, b=40;
   int *pa=&a;  int *pb=&b;
   increment(pa, pb);
   printf("%d %d", a, b);
}
```

px = pa = &a

py = pa = &b

Pass by value !!!

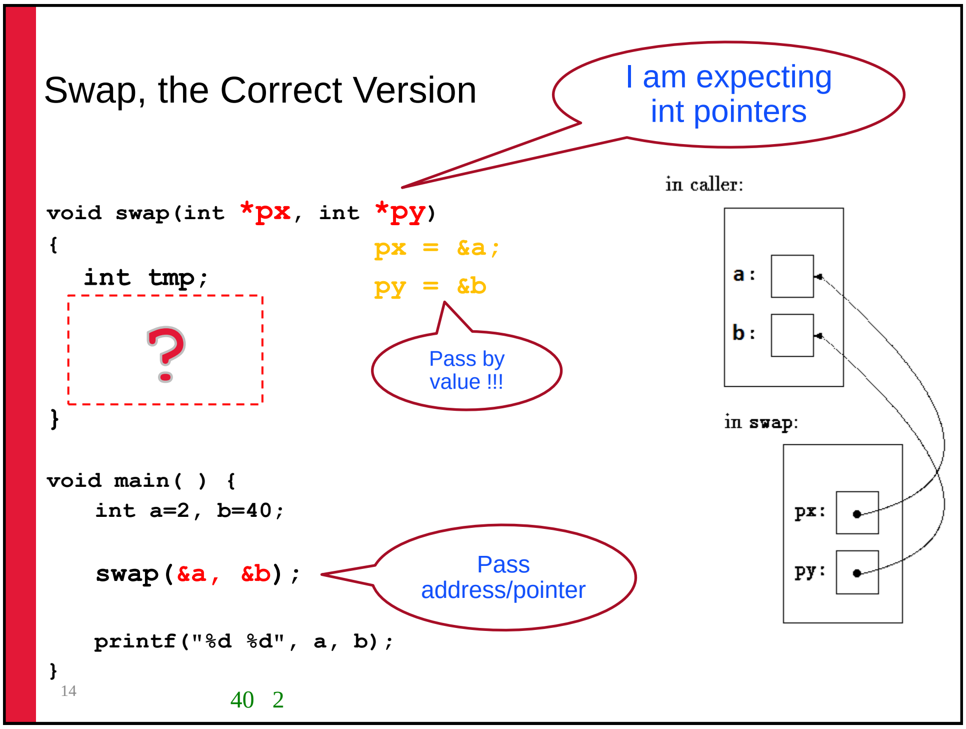
Pass address/pointer

Another way

3 50

Not in Java

13



Swap, the Correct Version

I am expecting int pointers

```
void swap(int *px, int *py)
{
   int tmp;
      ?
}

void main( ) {
    int a=2, b=40;

    swap(&a, &b);

    printf("%d %d", a, b);
}
```

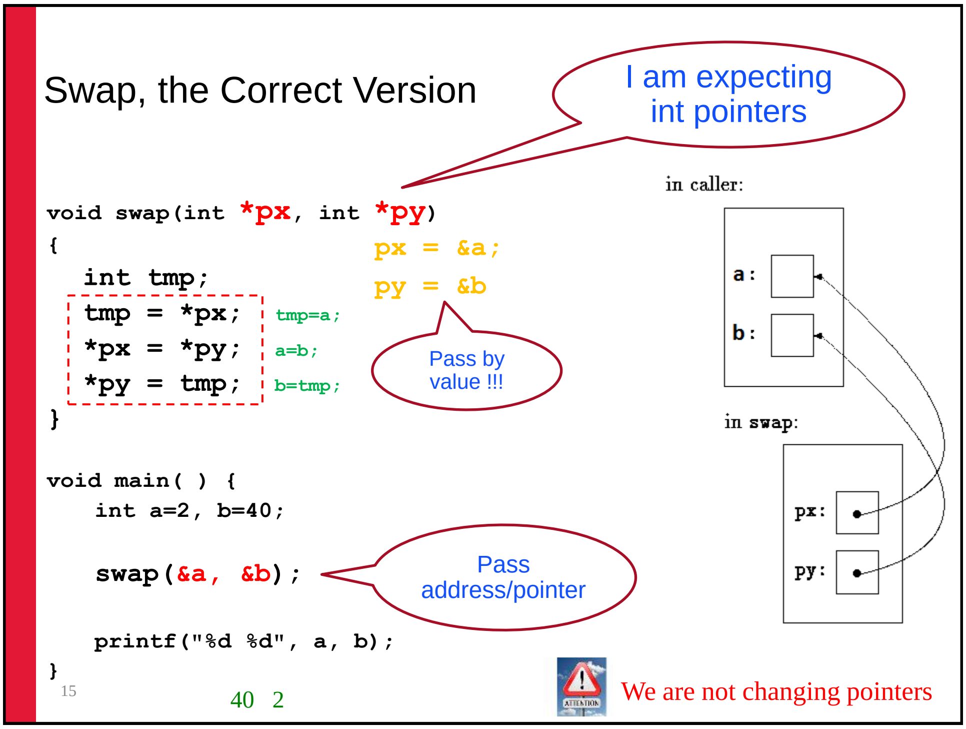
px = &a;
py = &b

Pass by value !!!

Pass address/pointer

40 2

14



Swap, the Correct Version

I am expecting int pointers

```
void swap(int *px, int *py)
{
   int tmp;
   tmp = *px;     tmp=a;
   *px = *py;     a=b;
   *py = tmp;     b=tmp;
}

void main( ) {
    int a=2, b=40;

    swap(&a, &b);

    printf("%d %d", a, b);
}
```

px = &a;
py = &b

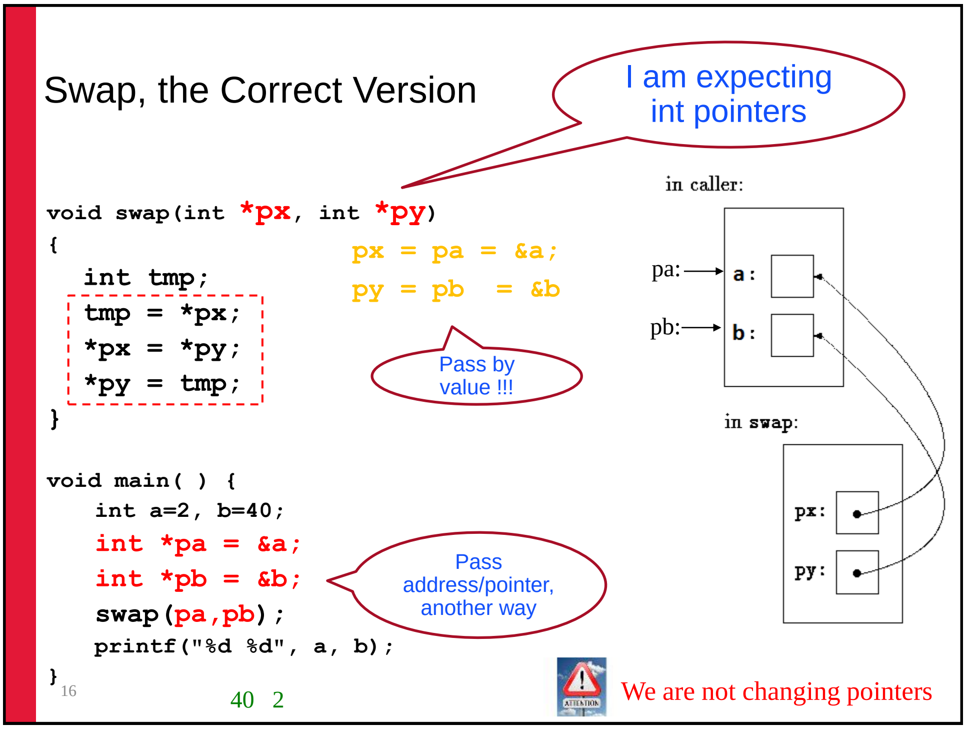
Pass by value !!!

Pass address/pointer

40 2

We are not changing pointers

15



Swap, the Correct Version

I am expecting int pointers

```
void swap(int *px, int *py)
{
   int tmp;
   tmp = *px;
   *px = *py;
   *py = tmp;
}

void main( ) {
    int a=2, b=40;
    int *pa = &a;
    int *pb = &b;
    swap(pa,pb);
    printf("%d %d", a, b);
}
```

px = pa = &a;

py = pb = &b

Pass by value !!!

Pass address/pointer, another way

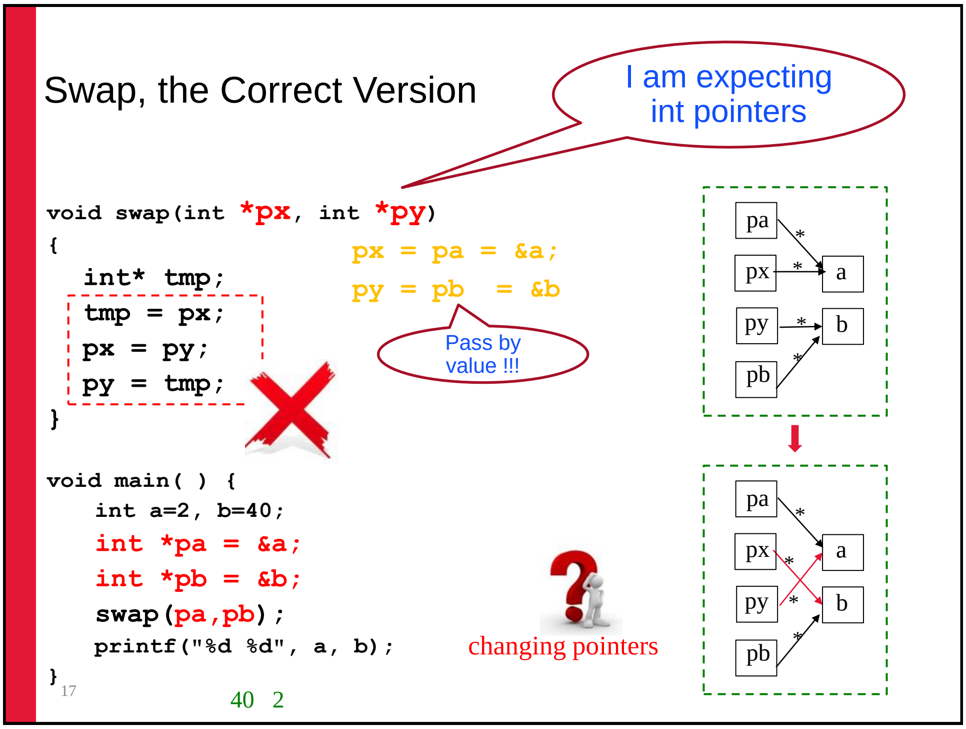
40   2

We are not changing pointers

in caller:
pa:
a :
pb:
b :
in swap:
px:
py:

16

16



Swap, the Correct Version

I am expecting int pointers

```
void swap(int *px, int *py)
{
   int* tmp;
   tmp = px;
   px = py;
   py = tmp;
}

void main( ) {
    int a=2, b=40;
    int *pa = &a;
    int *pb = &b;
    swap(pa,pb);
    printf("%d %d", a, b);
}
```

px = pa = &a;

py = pb = &b

Pass by value !!!

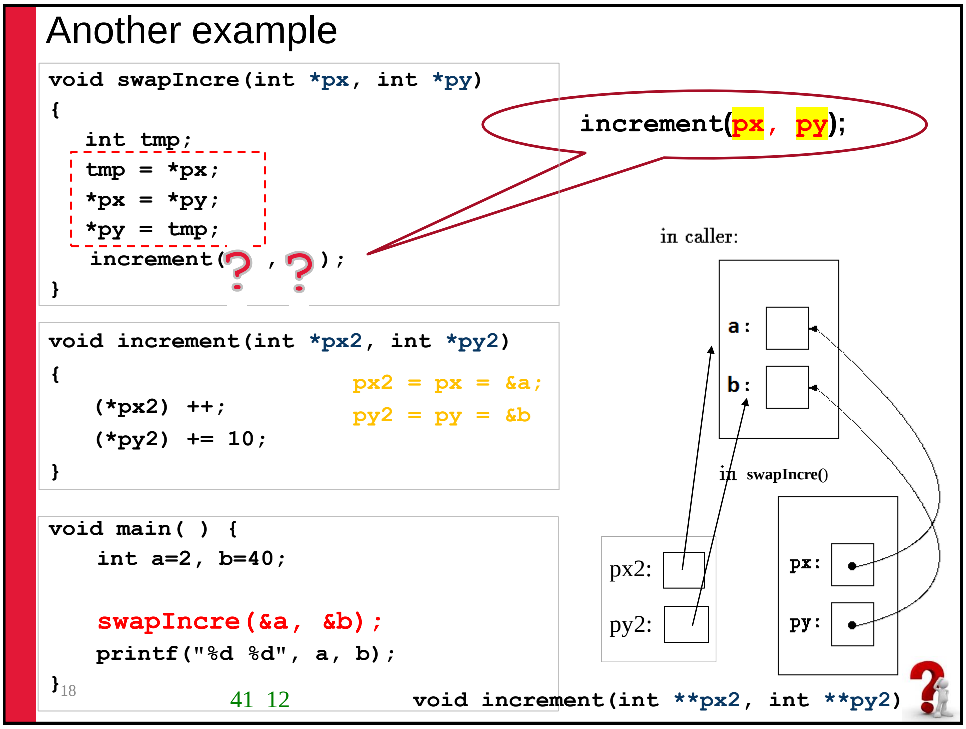
changing pointers

40   2

pa
px
py
pb
a
b
*
*
*
*
pa
px
py
pb
a
b
*
*
*
*

17

17



Another example

```
void swapIncre(int *px, int *py)
{
   int tmp;
   tmp = *px;
   *px = *py;
   *py = tmp;
    increment(? , ?);
}
```

increment(px, py);

```
void increment(int *px2, int *py2)
{
    (*px2) ++;
    (*py2) += 10;
}
```

px2 = px = &a;
py2 = py = &b

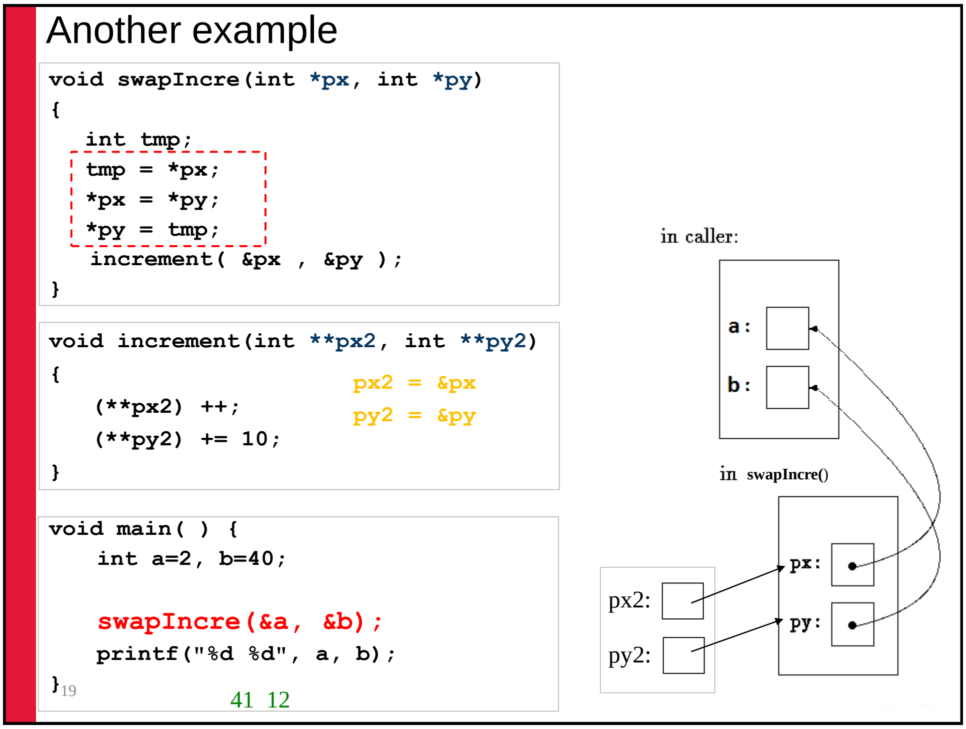
```
void main( ) {
    int a=2, b=40;

    swapIncre(&a, &b);
    printf("%d %d", a, b);
}
```

18

41 12

void increment(int **px2, int **py2)

18



Another example

```
void swapIncre(int *px, int *py)
{
   int tmp;
   tmp = *px;
   *px = *py;
   *py = tmp;
    increment( &px , &py );
}
```

```
void increment(int **px2, int **py2)
{
    (**px2) ++;
    (**py2) += 10;
}
```

px2 = &px
py2 = &py

```
void main( ) {
    int a=2, b=40;

    swapIncre(&a, &b);
    printf("%d %d", a, b);
}
```

19

41 12

19

Now understand scanf() -- more or less

```
int x=1;  int y = 2;
swap(&x,&y);  increment(&x,&y);

int x; char c;
scanf ("%d %c", &x, &c);
printf("%d %c", x, c);

int x;
int *px = &x;
scanf("%d", px);
printf("%d",*px);
```

explain shortly

But why array name is used directly

```
scanf ("%s %d", name, &age);
fgets (input, 5,stdin);
```

lab4

```
sscanf(table[curr_row],"%s %d %f", name, &age, &rate);
```

20

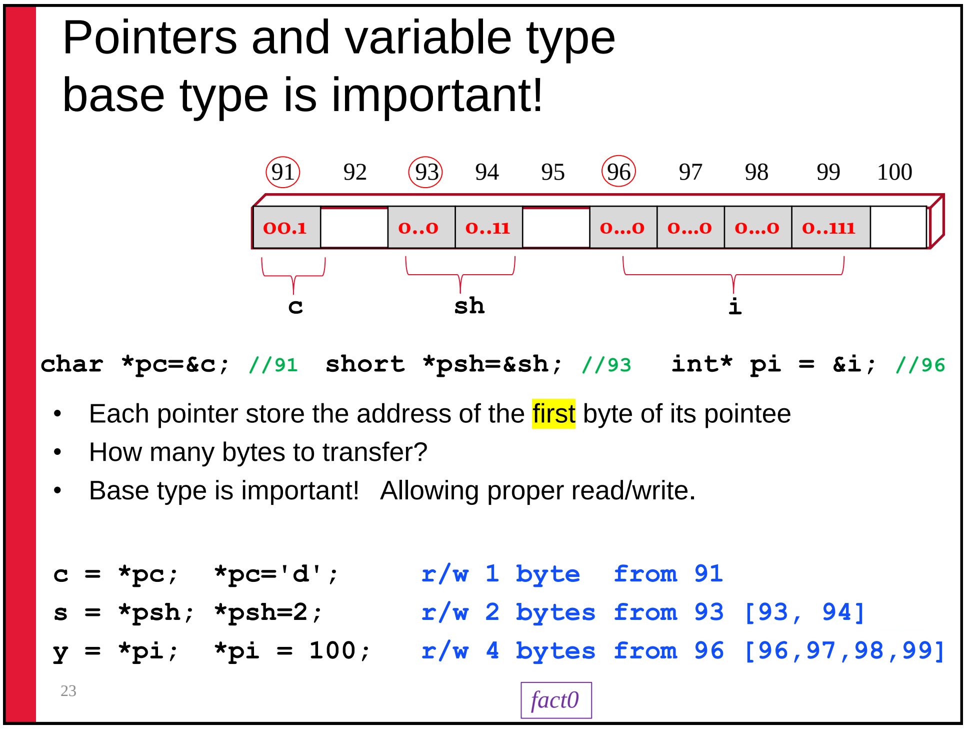
Pointers K&R Ch 5

- Basics: Declaration and assignment (5.1)
- Pointer to Pointer (5.6)
- Pointer and functions (5.2)
- **Pointer arithmetic (5.4)**
- Pointers and arrays (5.3)
- Arrays of pointers (5.6)
- Command line argument (5.10)
- Pointer to arrays and two dimensional arrays (5.9)
- Pointer to functions (5.11)
- Pointer to structures (6.4)
- Memory allocation (extra)

Last lecture

YORK U
UNIVERSITÉ
UNIVERSITY

22



Pointers and variable type base type is important!

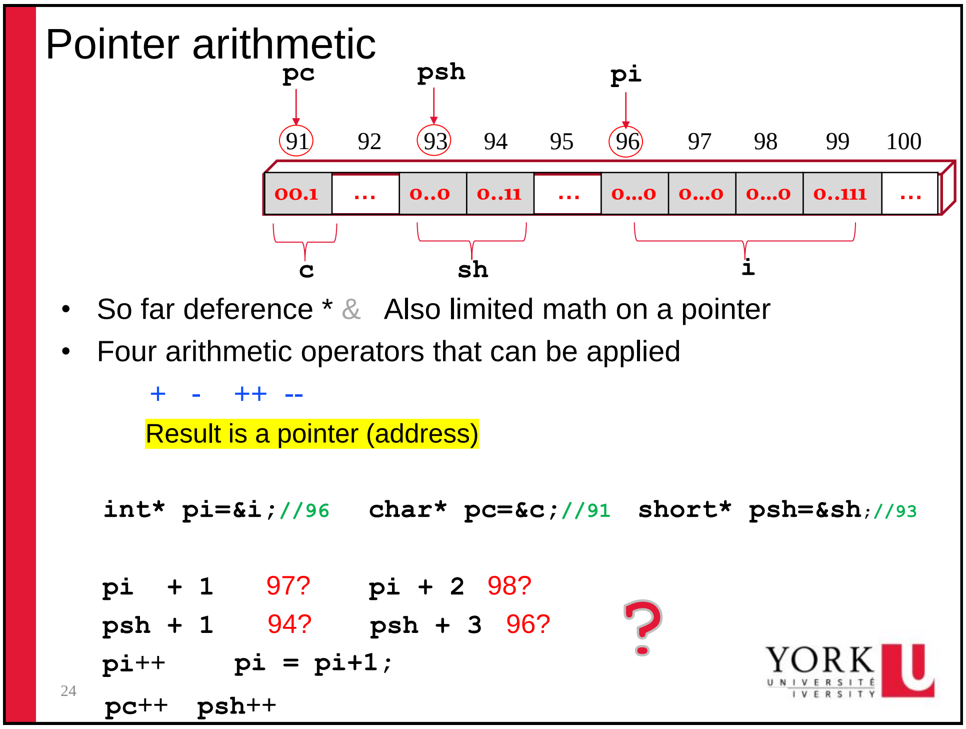
```
char *pc=&c; //91  short *psh=&sh; //93   int* pi = &i; //96
```

- Each pointer store the address of the first byte of its pointee
- How many bytes to transfer?
- Base type is important!   Allowing proper read/write.

```
c = *pc;   *pc='d';      r/w 1 byte  from 91
s = *psh; *psh=2;        r/w 2 bytes from 93 [93, 94]
y = *pi;   *pi = 100;    r/w 4 bytes from 96 [96,97,98,99]
```

23

fact0



Pointer arithmetic

- So far deference * &   Also limited math on a pointer
- Four arithmetic operators that can be applied

```
+  -  ++ --
```

Result is a pointer (address)

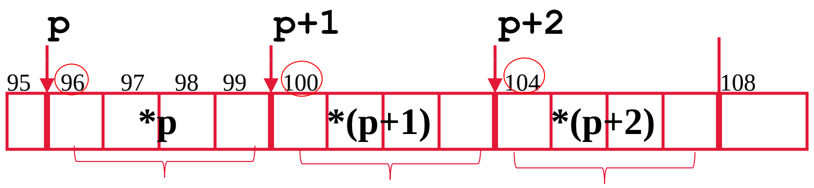
```
int* pi=&i;//96   char* pc=&c;//91  short* psh=&sh;//93

pi  + 1    97?   pi + 2  98?
psh + 1    94?   psh + 3  96?
pi++     pi = pi+1;
pc++  psh++
```

?

24

Pointer arithmetic – scaled

- Incrementing / decrementing a pointer by *n* moves it *n* units bytes
  p ± *n* ➔ p ± *n* × unit byte
  - value of a "unit" is based upon the size of the pointee type
    - fact1 ○ p ± *n* ➔ p ± *n* × pointee-type-size
  - If p points to an integer (4 bytes), value of unit is 4
    p + n advances by n*4 bytes:
    p + 1 = 96 + 1*4 = 100 p + 2 = 96 + 2*4 = 104

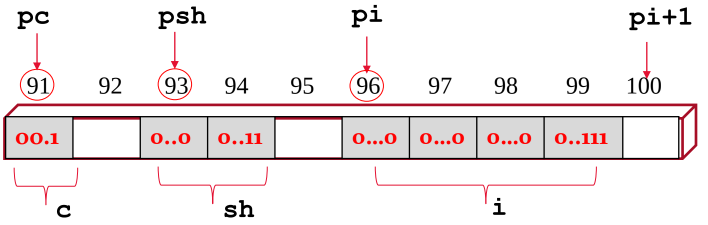


- Why would we need to move pointer? p+1; p++
- Why designed this way? *"p+1 is p+4"*

26

26

Pointer arithmetic – scaled. *"p+1 is p+4"*



```
int* pi=&i;//96     char *pc=&c;//91    short* psh=&sh;//93
```

assume
int 4 bytes
short 2 bytes
char 1 byte

pi + 1 address 96+ 1*4 =100

pi + 2 address 96+ 2*4 =104

? !!!

psh +1 address 93+ 1*2 =95

psh +2 address 93+ 2*2 =97

pc + 1 address 91+ 1*1 =92

pi++? **pc**++? **psh**++? **pi = 96 + 4** **pc = 91 + 1** **psh = 93 + 2**

27

27

Pointers K&R Ch 5

- Basics: Declaration and assignment (5.1)
- Pointer to Pointer (5.6)
- Pointer and functions (5.2)
- Pointer arithmetic (5.4)
- **Pointers and arrays (5.3)**
- Arrays of pointers (5.6)
- Command line argument (5.10)
- Pointer to arrays and two dimensional arrays (5.9)
- Pointer to functions (5.11)
- Pointer to structures (6.4)
- Memory allocation (extra)

Last lecture

29

Pointers K&R Ch 5

- Basics: Declaration and assignment (5.1)
- Pointer to Pointer (5.6)
- Pointer and functions (5.2) -- pass pointer by value
- Pointer arithmetic (5.4) +   - ++ --   "*p+1 is p+4*"

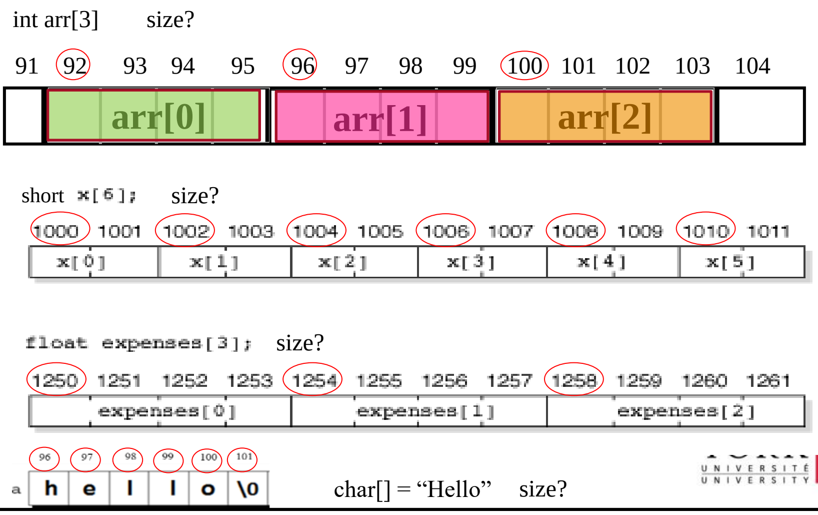
fact1

- **Pointers and arrays (5.3)**
  - Arrays are stored consecutively   *fact2*
  - Pointer to array elements `p + i = &a[i]`   `*(p+i) = a[i]`
  - Array name contains address of 1st element   `a = &a[0]`
  - Pointer arithmetic on array (extension)
  - Array as function argument – "decay"
  - Pass sub_array

30

Pointers and Arrays (5.3)

- Array members are <u>next to each other</u> in memory
  - fact2
  - **`arr[0]`** always occupies in the lowest address
  - **`&arr[i+1] == &arr[i] +`** size of type in byte**`;`**



int arr[3]   size?

91 92 93 94 95 96 97 98 99 100 101 102 103 104

arr[0] arr[1] arr[2]

short `x[6];`   size?

1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011

x[0] x[1] x[2] x[3] x[4] x[5]

`float expenses[3];`   size?

1250 1251 1252 1253 1254 1255 1256 1257 1258 1259 1260 1261

expenses[0] expenses[1] expenses[2]

96 97 98 99 100 101

a h e l l o \0

char[] = “Hello”   size?

Pointers K&R Ch 5

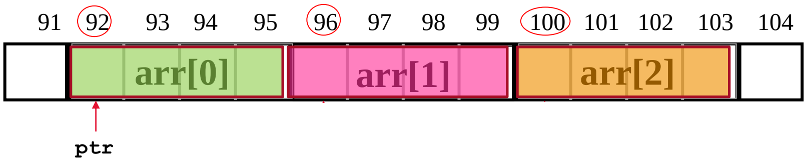
- Basics: Declaration and assignment (5.1)
- Pointer to Pointer (5.6)
- Pointer and functions (5.2) -- pass pointer by value
- Pointer arithmetic (5.4) + - ++ --   “*p+1 is p+4*”

fact1

- **Pointers and arrays (5.3)**
  - Arrays are stored consecutively   fact2
  - **Pointer to array elements**   `p+i = &a[i]`   `*(p+i) = a[i]`   fact3
  - Array name contains address of 1st element   `a = &a[0]`
  - Pointer arithmetic on array (extension)
  - Array as function argument – “decay”
  - Pass sub_array

Pointers and Arrays (5.3)

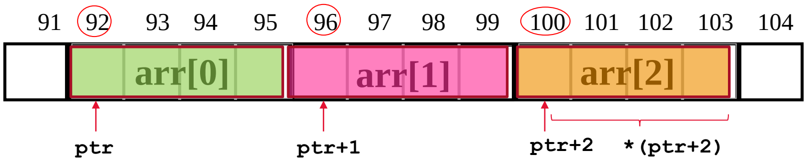
- *“p+1 is p+4”* *fact1*
- Array members are next to each other in memory *fact2*
  - **arr[0]** always occupies in the lowest address



```
int arr[3]; int *ptr;
ptr = &arr[0];  // 92

ptr + 1
ptr + 2
*(ptr+2)
```

Pointers and Arrays (5.3)

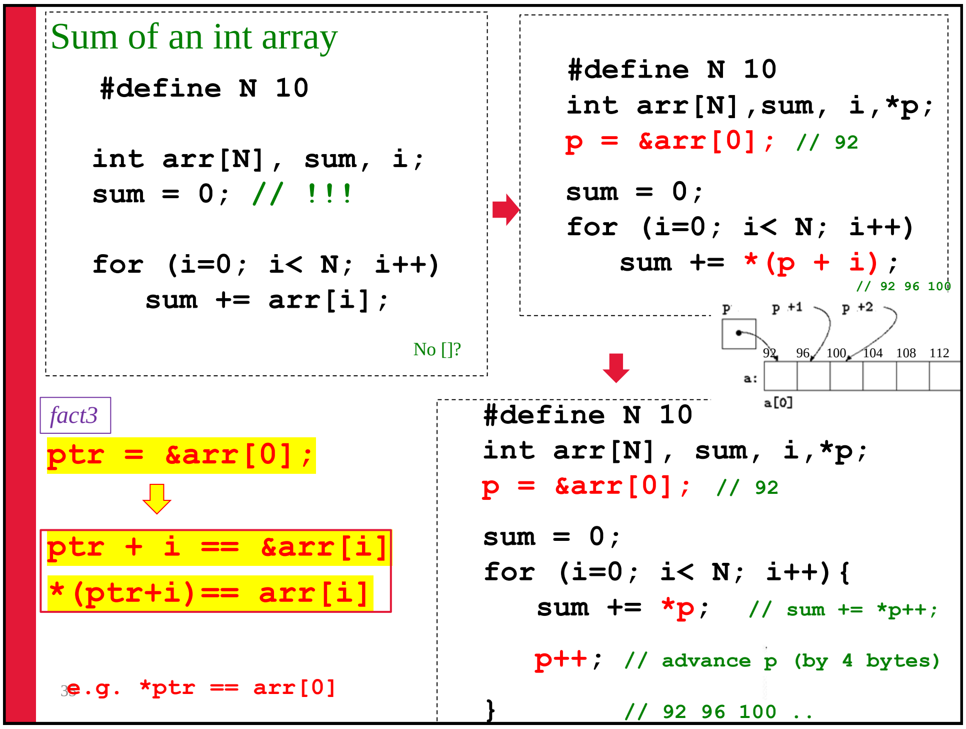
- *“p+1 is p+4”* *fact1*
- Array members are next to each other in memory *fact2*
  - **arr[0]** always occupies in the lowest address



```
int arr[3]; int *ptr;
ptr = &arr[0];  // 92

ptr + 1         // 92+1*4 = 96 == &arr[1]
ptr + 2         // 92+2*4 = 100 == &arr[2]
*(ptr+2)       // *&arr[2] → access arr[2]
```

```
ptr + i     == &arr[i]
*(ptr + i) == arr[i]
```

fact3



Sum of an int array

```
#define N 10

int arr[N], sum, i;
sum = 0; // !!!

for (i=0; i< N; i++)
   sum += arr[i];
```

No []?

```
#define N 10
int arr[N],sum, i,*p;
p = &arr[0]; // 92

sum = 0;
for (i=0; i< N; i++)
   sum += *(p + i);
                    // 92 96 100
```

p
p +1
p +2
92
96
100
104
108
112
a:
a[0]

fact3

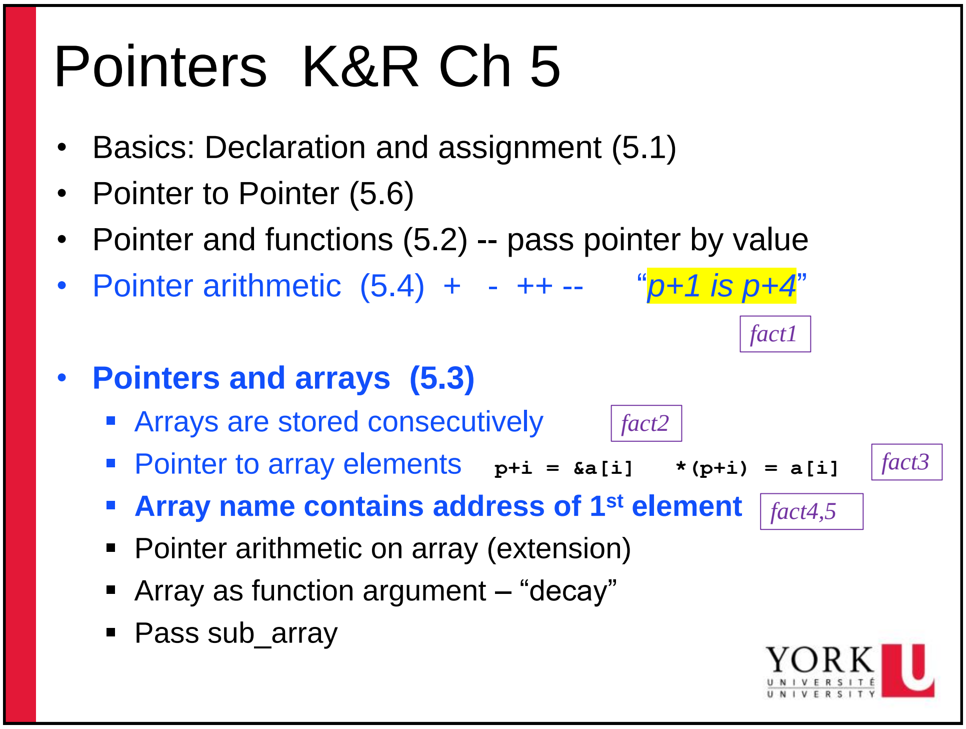
```
ptr = &arr[0];
```

```
ptr + i == &arr[i]
*(ptr+i)== arr[i]
```

```
e.g. *ptr == arr[0]
```

```
#define N 10
int arr[N], sum, i,*p;
p = &arr[0]; // 92

sum = 0;
for (i=0; i< N; i++){
   sum += *p;  // sum += *p++;

   p++; // advance p (by 4 bytes)

}          // 92 96 100 ..
```

35



Pointers K&R Ch 5

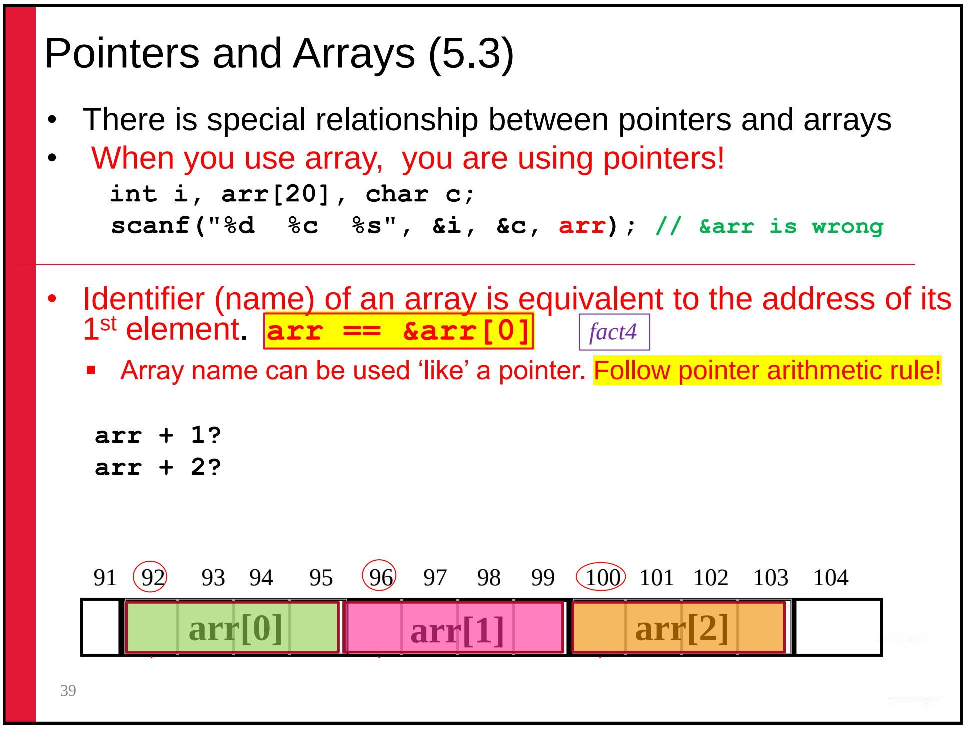
- Basics: Declaration and assignment (5.1)
- Pointer to Pointer (5.6)
- Pointer and functions (5.2) -- pass pointer by value
- Pointer arithmetic (5.4) +   - ++ --   "*p+1 is p+4*"

fact1

- **Pointers and arrays (5.3)**
  - Arrays are stored consecutively   fact2
  - Pointer to array elements   `p+i = &a[i]   *(p+i) = a[i]`   fact3
  - **Array name contains address of 1st element**   fact4,5
  - Pointer arithmetic on array (extension)
  - Array as function argument – "decay"
  - Pass sub_array

YORK UNIVERSITÉ UNIVERSITY

36



Pointers and Arrays (5.3)

- There is special relationship between pointers and arrays
- When you use array, you are using pointers!

```
int i, arr[20], char c;
scanf("%d  %c  %s", &i, &c, arr); // &arr is wrong
```

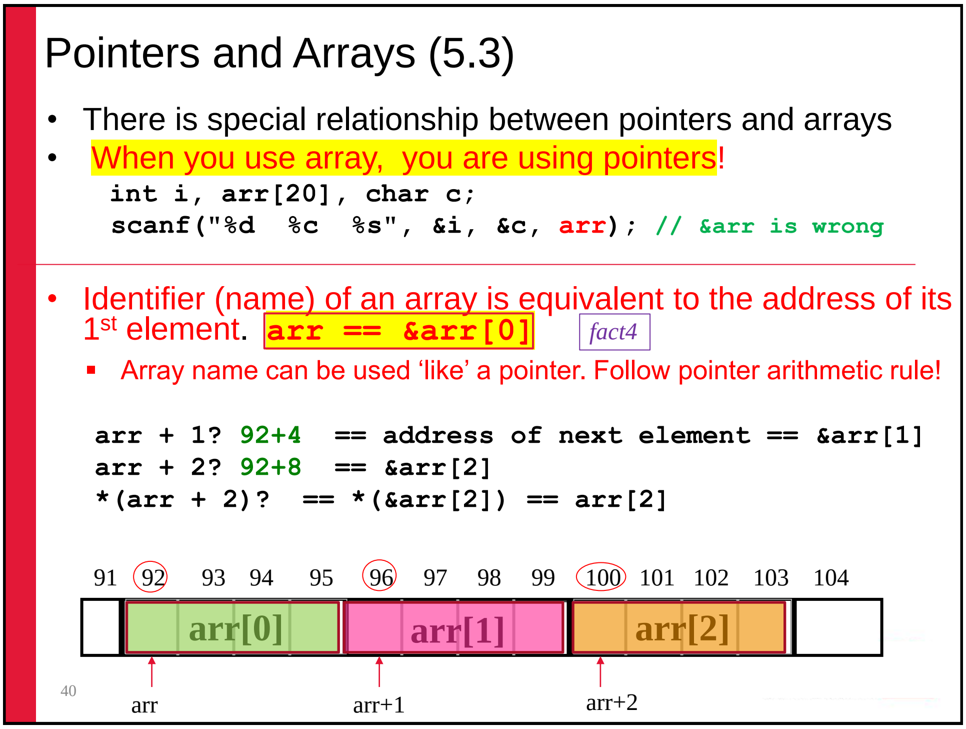
- Identifier (name) of an array is equivalent to the address of its 1st element. **arr == &arr[0]** *fact4*
  - Array name can be used 'like' a pointer. Follow pointer arithmetic rule!

```
arr + 1?
arr + 2?
```

91 92 93 94 95 96 97 98 99 100 101 102 103 104

	arr[0]	arr[1]	arr[2]	

39



Pointers and Arrays (5.3)

- There is special relationship between pointers and arrays
- When you use array, you are using pointers!

```
int i, arr[20], char c;
scanf("%d  %c  %s", &i, &c, arr); // &arr is wrong
```

- Identifier (name) of an array is equivalent to the address of its 1st element. **arr == &arr[0]** *fact4*
  - Array name can be used 'like' a pointer. Follow pointer arithmetic rule!

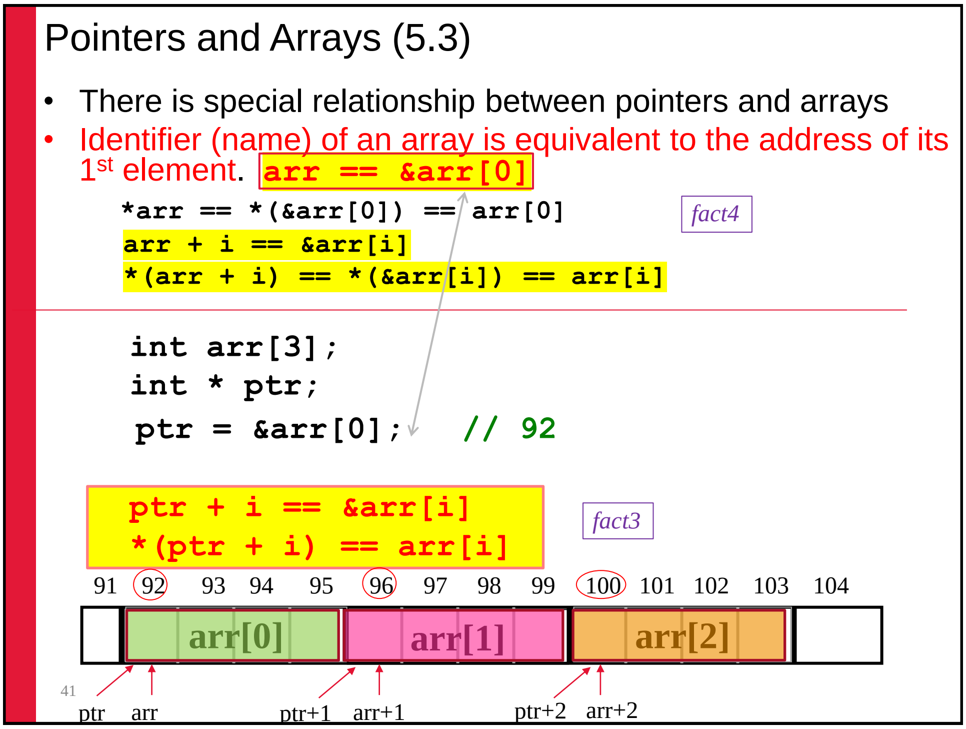
```
arr + 1? 92+4  == address of next element == &arr[1]
arr + 2? 92+8  == &arr[2]
*(arr + 2)?  == *(&arr[2]) == arr[2]
```

91 92 93 94 95 96 97 98 99 100 101 102 103 104

	arr[0]	arr[1]	arr[2]	

arr   arr+1   arr+2

40



Pointers and Arrays (5.3)

- There is special relationship between pointers and arrays
- Identifier (name) of an array is equivalent to the address of its 1st element. `arr == &arr[0]`

```
*arr == *(&arr[0]) == arr[0]
arr + i == &arr[i]
*(arr + i) == *(&arr[i]) == arr[i]
```

fact4

```
int arr[3];
int * ptr;
ptr = &arr[0];   // 92
```

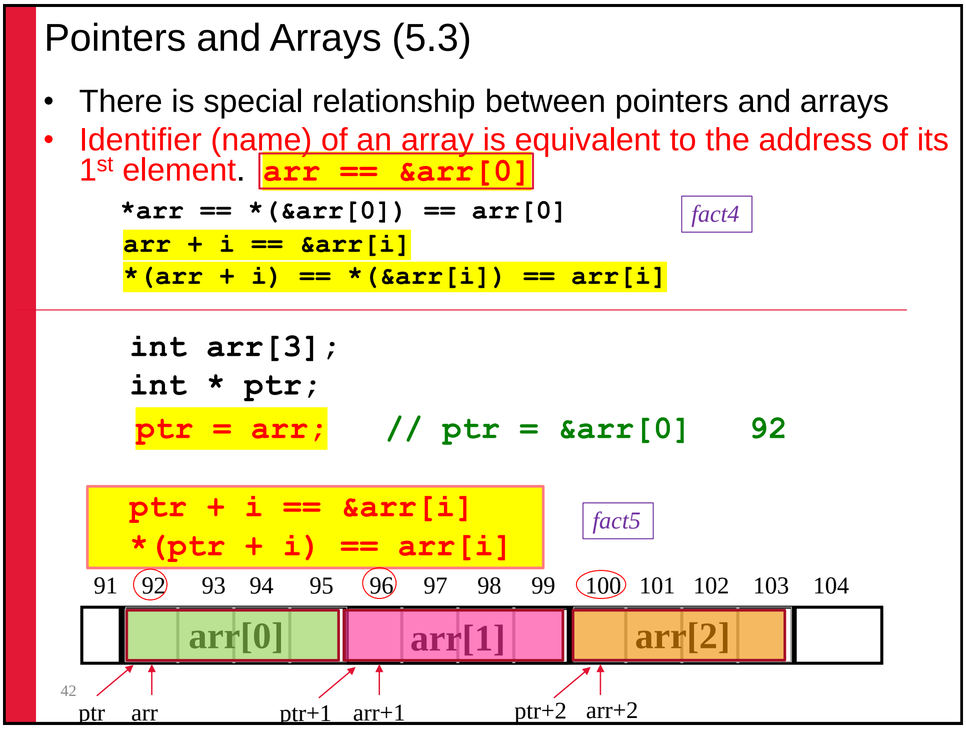
```
ptr + i == &arr[i]
*(ptr + i) == arr[i]
```

fact3

91 92 93 94 95 96 97 98 99 100 101 102 103 104
arr[0]
arr[1]
arr[2]
ptr
arr
ptr+1
arr+1
ptr+2
arr+2

41

41



Pointers and Arrays (5.3)

- There is special relationship between pointers and arrays
- Identifier (name) of an array is equivalent to the address of its 1st element. `arr == &arr[0]`

```
*arr == *(&arr[0]) == arr[0]
arr + i == &arr[i]
*(arr + i) == *(&arr[i]) == arr[i]
```

fact4

```
int arr[3];
int * ptr;
ptr = arr;    // ptr = &arr[0]   92
```

```
ptr + i == &arr[i]
*(ptr + i) == arr[i]
```

fact5

91 92 93 94 95 96 97 98 99 100 101 102 103 104
arr[0]
arr[1]
arr[2]
ptr
arr
ptr+1
arr+1
ptr+2
arr+2

42

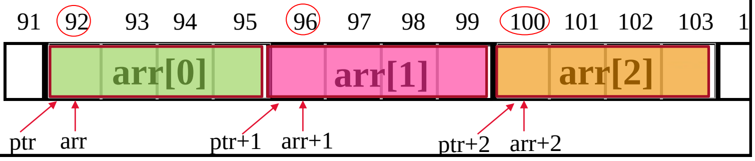
42

- There is special relationship between pointers and arrays
- Identifier (name) of an array is equivalent to the address of its 1st element. `arr == &arr[0]`

```
int arr[3]; int * ptr;
ptr = arr;   /*   ptr = &arr[0]   */
```

fact 4, 5

```
arr+i == &arr[i]
ptr+i == &arr[i]
```



- There is special relationship between pointers and arrays
- Identifier (name) of an array is equivalent to the address of its 1st element. `arr == &arr[0]`

```
int arr[3]; int * ptr;
ptr = arr;   /*   ptr = &arr[0]   */
```

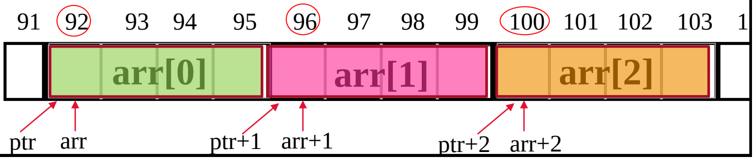
fact 4, 5

```
arr+i == &arr[i]
ptr+i == &arr[i]
```

```
arr[i]

*(ptr + i)

*(arr + i)

ptr[i];
```

Compiler converts arr[2] to *(arr+2)

equivalent

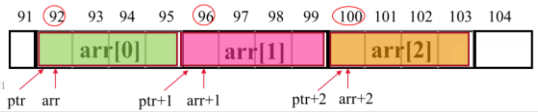


arrayAddressPPP.c

```
/* Demonstrates use of pointer arithmetic in array*/
main() {
 int arr[10] = {0,10,20,30,40,50,60,70,80,90}, i;
 int *ptr = arr;    /* = &arr[0] */

 printf("%p %p", arr, ptr); // print array name!

/* Print the addresses of each array element. */
 for (i = 0; i < 10; i++)
   printf("%p %p %p", &arr[i], arr+i, ptr+i);

/* Print the content of each array element. */
 for (i = 0; i < 10; i++)
   printf("%d %d %d", arr[i], *(arr+i), *(ptr+i));

}
```

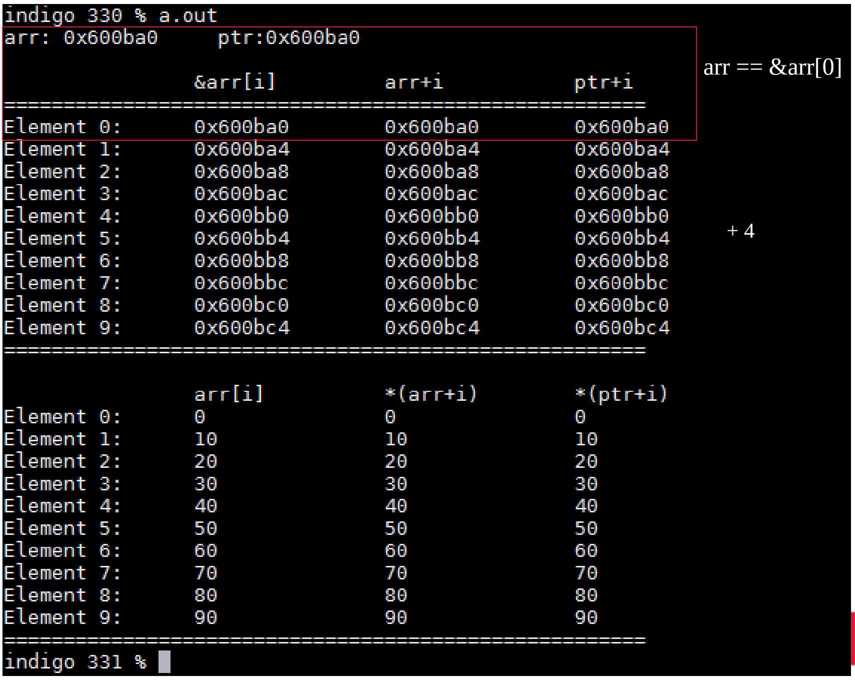
Different ways of accessing array element addresses

Different ways of accessing array elements



45

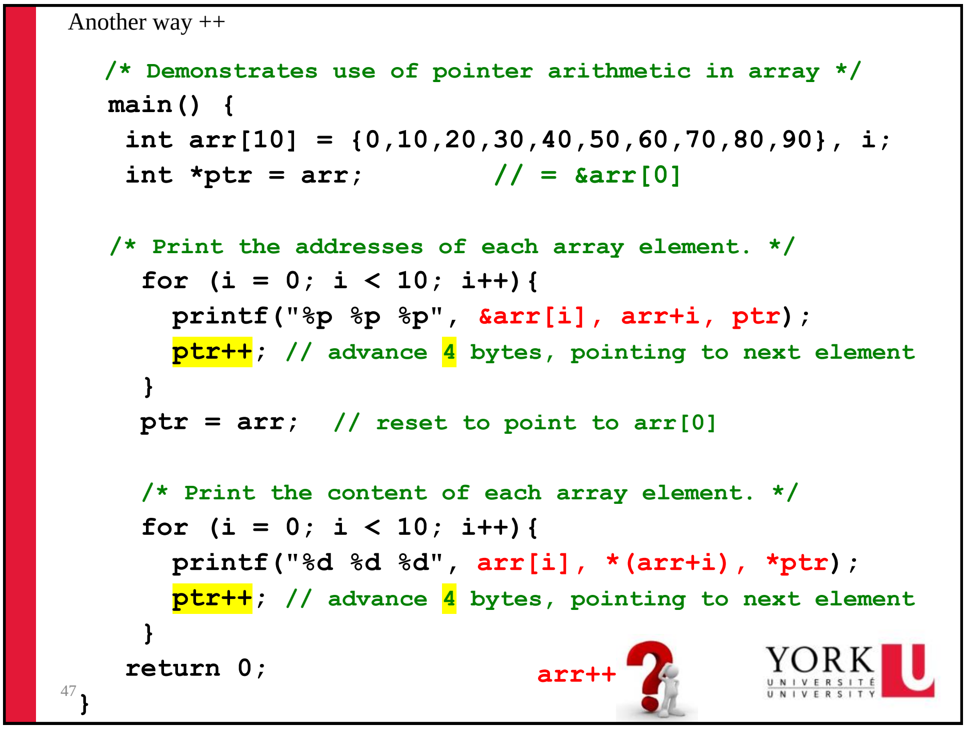
21



```
indigo 330 % a.out
arr: 0x600ba0     ptr:0x600ba0

                &arr[i]         arr+i           ptr+i
======================================================
Element 0:      0x600ba0        0x600ba0        0x600ba0
Element 1:      0x600ba4        0x600ba4        0x600ba4
Element 2:      0x600ba8        0x600ba8        0x600ba8
Element 3:      0x600bac        0x600bac        0x600bac
Element 4:      0x600bb0        0x600bb0        0x600bb0
Element 5:      0x600bb4        0x600bb4        0x600bb4
Element 6:      0x600bb8        0x600bb8        0x600bb8
Element 7:      0x600bbc        0x600bbc        0x600bbc
Element 8:      0x600bc0        0x600bc0        0x600bc0
Element 9:      0x600bc4        0x600bc4        0x600bc4
======================================================

                arr[i]          *(arr+i)        *(ptr+i)
Element 0:      0               0               0
Element 1:      10              10              10
Element 2:      20              20              20
Element 3:      30              30              30
Element 4:      40              40              40
Element 5:      50              50              50
Element 6:      60              60              60
Element 7:      70              70              70
Element 8:      80              80              80
Element 9:      90              90              90
======================================================
indigo 331 %
```

arr == &arr[0]

+ 4

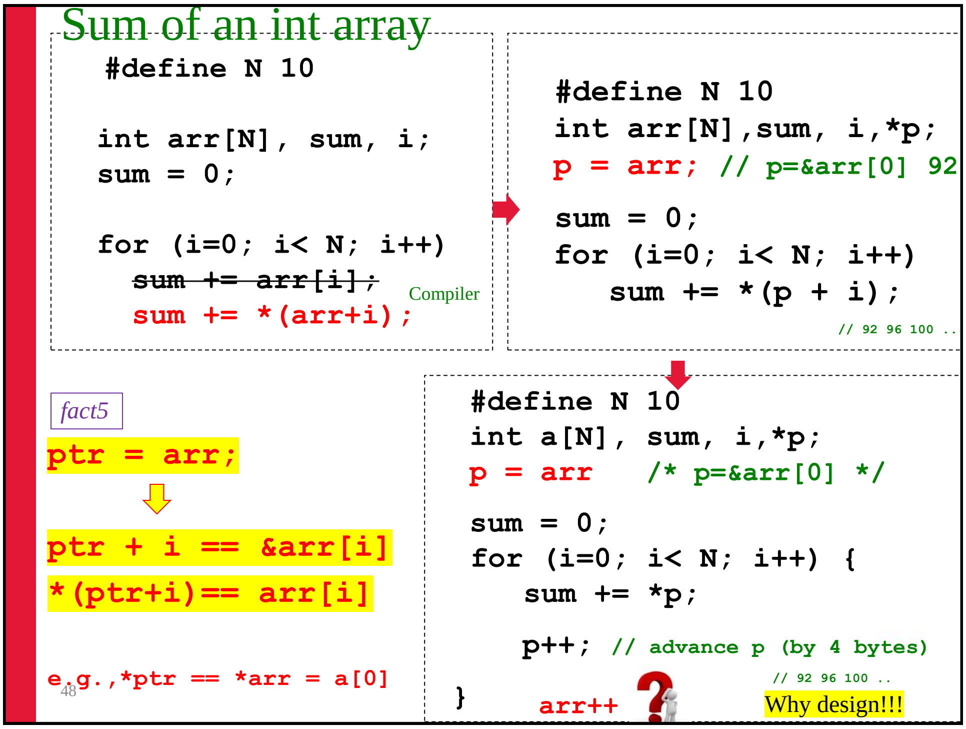


Another way ++

```
/* Demonstrates use of pointer arithmetic in array */
main() {
 int arr[10] = {0,10,20,30,40,50,60,70,80,90}, i;
 int *ptr = arr;         // = &arr[0]

/* Print the addresses of each array element. */
  for (i = 0; i < 10; i++){
    printf("%p %p %p", &arr[i], arr+i, ptr);
    ptr++; // advance 4 bytes, pointing to next element
  }
  ptr = arr;  // reset to point to arr[0]

  /* Print the content of each array element. */
  for (i = 0; i < 10; i++){
    printf("%d %d %d", arr[i], *(arr+i), *ptr);
    ptr++; // advance 4 bytes, pointing to next element
  }
 return 0;
}
```

arr++ ?

YORK UNIVERSITÉ UNIVERSITY



Sum of an int array

```
 #define N 10

int arr[N], sum, i;
sum = 0;

for (i=0; i< N; i++)
  ~~sum += arr[i];~~
  sum += *(arr+i);
```

Compiler

```
#define N 10
int arr[N],sum, i,*p;
p = arr; // p=&arr[0] 92

sum = 0;
for (i=0; i< N; i++)
   sum += *(p + i);
                        // 92 96 100 ..
```

fact5

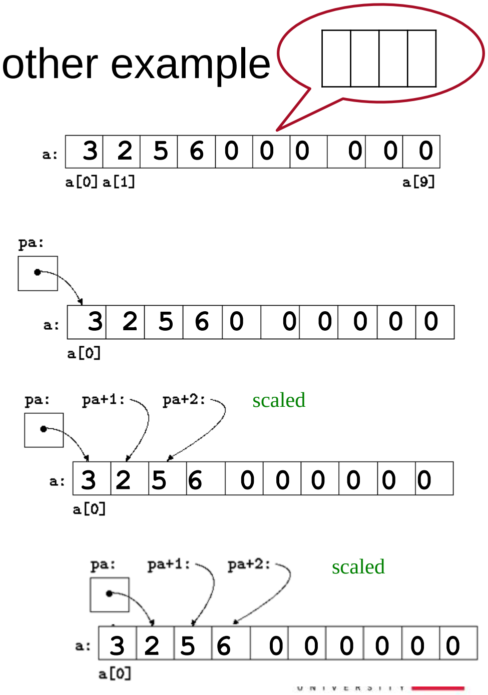
```
ptr = arr;

ptr + i == &arr[i]
*(ptr+i)== arr[i]

e.g.,*ptr == *arr = a[0]
```

```
#define N 10
int a[N], sum, i,*p;
p = arr   /* p=&arr[0] */

sum = 0;
for (i=0; i< N; i++) {
   sum += *p;

   p++; // advance p (by 4 bytes)
                       // 92 96 100 ..
}    arr++ ?
```

Why design!!!

Pointers and Arrays: another example

```
int a[10]={3,2,5,6};
int *pa;
pa = a; // pa=&a[0]

int x,y,z;

x = *pa;     // x = *a
     //same as x = a[0]

y = *(pa + 1);// pa+4 a[1]
z = *(a + 2); // a[2]

pa++;  // pa=&a[1]   = a+1
x = *(pa+2)    ?
y = *pa +2     ?

*(pa + 3) = 200;
```



x:6 y:4 z:5

a: 3 2 5 6 200 0 0 0 0 0

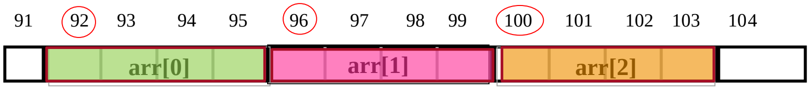
Summary

- Pointer arithmetic: If p points to an integer of 4 bytes, p + n advances by 4*n bytes: p + 1 = 96 + 1*4 = 100   p + 2 = 96 + 2*4 = 104

fact 1

- Array in memory:

fact 2



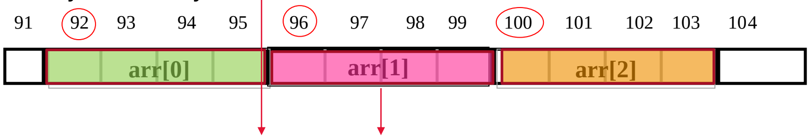
Summary

- Pointer arithmetic: If p points to an integer of 4 bytes, p + n advances by 4*n bytes: p + 1 = 96 + 1*4 = 100   p + 2 = 96 + 2*4 = 104

fact 1

- Array in memory:

fact 2



- Suppose `p` points to array element `k`, then `p+1` points to `k+1` (next) element. `p + i` points to `arr[k+i]`.
  - `p = &arr[k]:       p + i == &arr[k+i]   → *(p+i) == arr[k+i]`
  - `k=0: p=&arr[0]:  p + i == &arr[i]    → *(p+i) == arr[i]`

fact 3

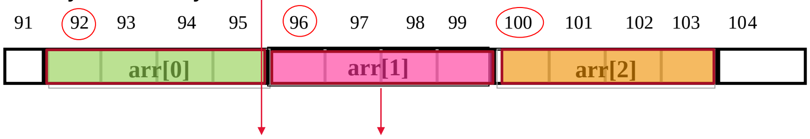
Summary

- Pointer arithmetic: If p points to an integer of 4 bytes, p + n advances by 4*n bytes: p + 1 = 96 + 1*4 = 100   p + 2 = 96 + 2*4 = 104

fact 1

- Array in memory:

fact 2



- Suppose `p` points to array element `k`, then `p+1` points to `k+1` (next) element. `p + i` points to `arr[k+i]`.
  - `p = &arr[k]:       p + i == &arr[k+i]   → *(p+i) == arr[k+i]`
  - `k=0: p=&arr[0]:  p + i == &arr[i]    → *(p+i) == arr[i]`

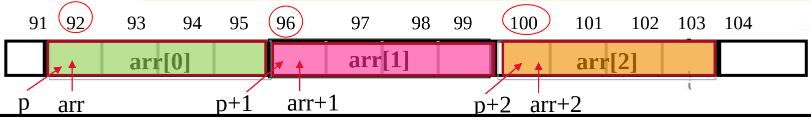
fact 3

- **Array name contains pointer to 1st element**   `arr==&arr[0]`
  - `arr==&arr[0]:     arr+i == &arr[i]   → *(arr+i) == arr[i]`

fact 4

`p = arr:    p + i == &arr[i]  →   *(p+i) == arr[i]`

fact 5



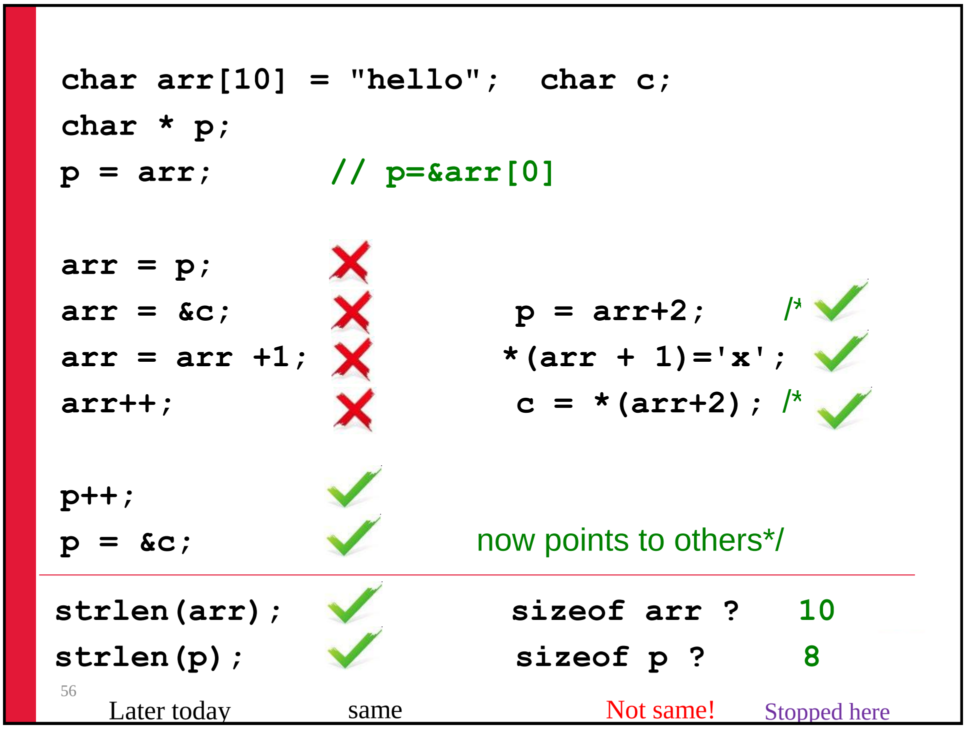
Attention: Array name can be used as a pointer, but is not a pointer variable!

```
int arr[20];
int * p = arr;
```

- **p** and **arr** are equivalent in that they have the same properties: **&arr[0]**
- Difference: **p** is a pointer variable, **arr** is a pointer constant
  - we could assign another value to **p**
  - **arr** will always point to the first of the 20 integer numbers of type int. Cannot change **arr** (point to somewhere else)

```
p = arr;   /*valid*/ ✓      arr = p;  /*invalid*/ ✗
p++;       /*valid*/ ✓      arr++;    /*invalid*/ ✗
```

```
char arr[10] = "hello";  char c;
char * p;
p = arr;      // p=&arr[0]

arr = p;
arr = &c;                     p = arr+2;     /*
arr = arr +1;                *(arr + 1)='x';
arr++;                        c = *(arr+2); /*

p++;
p = &c;
```

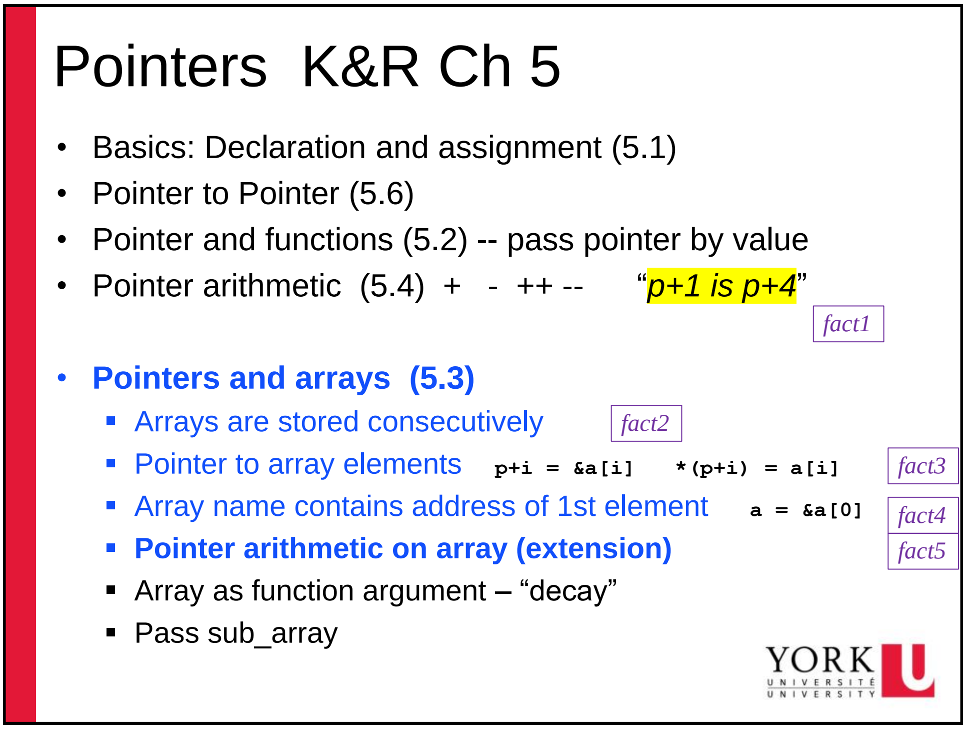


```
char arr[10] = "hello";   char c;
char * p;
p = arr;        // p=&arr[0]

arr = p;         ✗
arr = &c;        ✗           p = arr+2;      /* ✓
arr = arr +1;    ✗          *(arr + 1)='x';    ✓
arr++;           ✗           c = *(arr+2);  /* ✓

p++;             ✓
p = &c;          ✓          now points to others*/
--------------------------------------------------------
strlen(arr);     ✓           sizeof arr ?    10
strlen(p);       ✓           sizeof p ?       8
```

56

Later today   same   Not same!   Stopped here

56



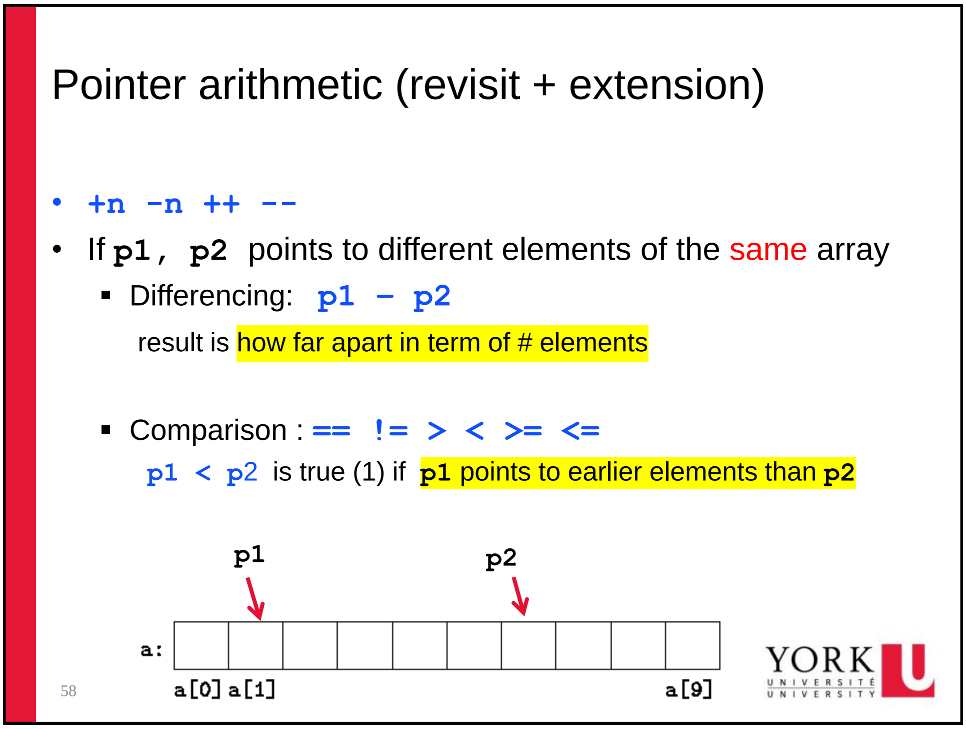
Pointers K&R Ch 5

- Basics: Declaration and assignment (5.1)
- Pointer to Pointer (5.6)
- Pointer and functions (5.2) -- pass pointer by value
- Pointer arithmetic (5.4) + - ++ --   “*p+1 is p+4*” *fact1*

- **Pointers and arrays (5.3)**
  - Arrays are stored consecutively *fact2*
  - Pointer to array elements `p+i = &a[i]   *(p+i) = a[i]` *fact3*
  - Array name contains address of 1st element `a = &a[0]` *fact4*
  - **Pointer arithmetic on array (extension)** *fact5*
  - Array as function argument – “decay”
  - Pass sub_array

YORK U UNIVERSITÉ UNIVERSITY

57



Pointer arithmetic (revisit + extension)

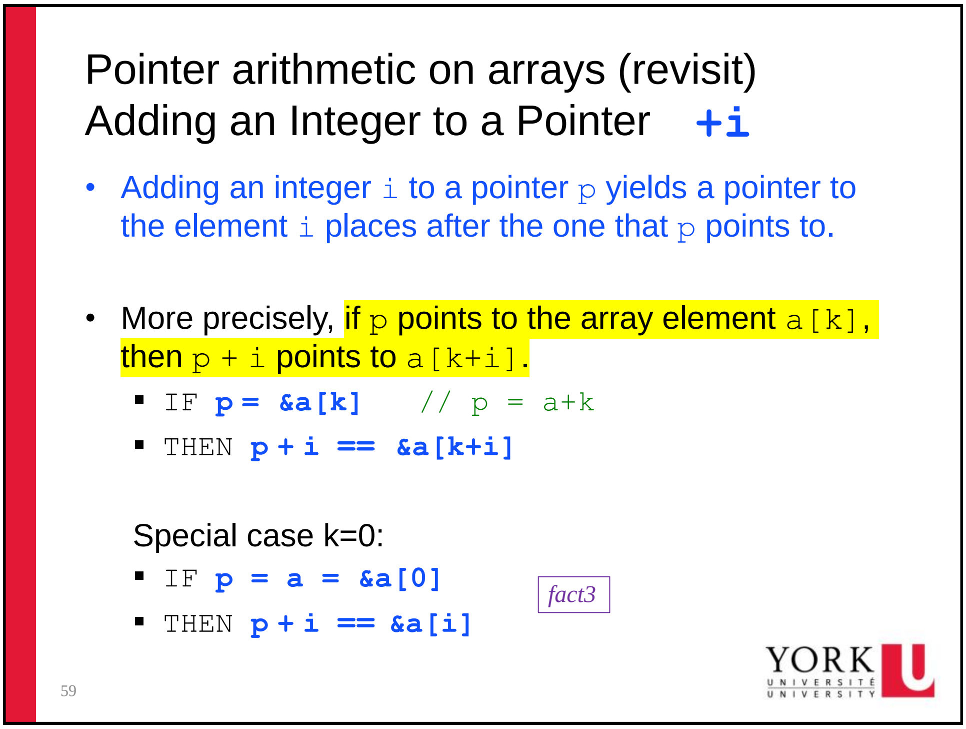
- **+n -n ++ --**
- If **p1, p2** points to different elements of the same array
  - Differencing: **p1 – p2**

    result is how far apart in term of # elements

  - Comparison : **== != > < >= <=**

    p1 < p2 is true (1) if **p1** points to earlier elements than **p2**

p1
p2
a:
a[0]
a[1]
a[9]



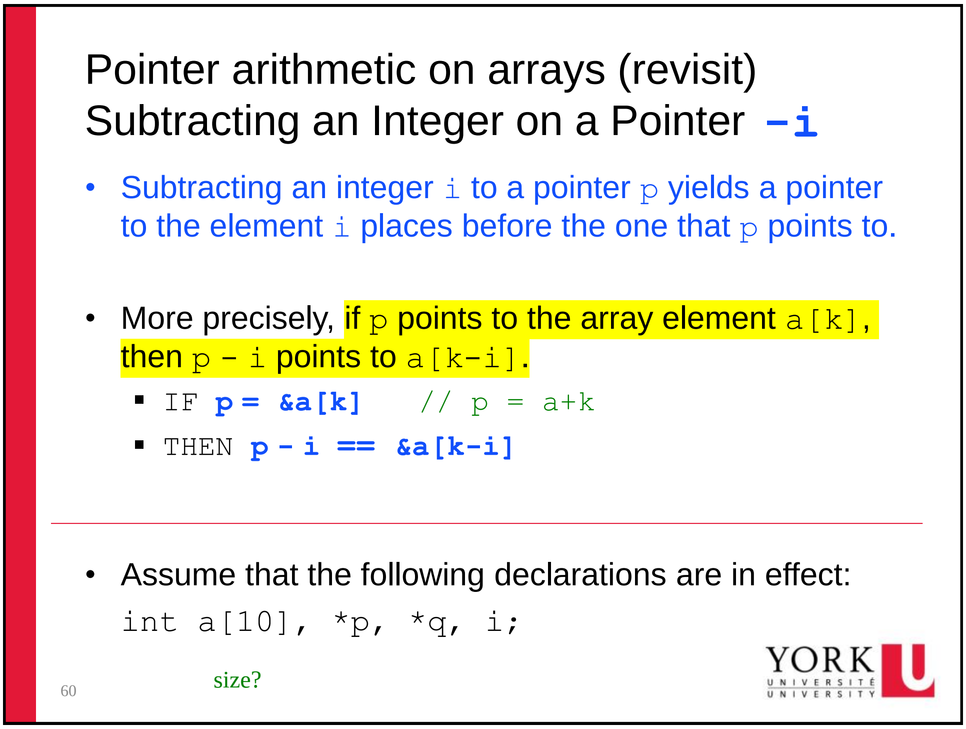
Pointer arithmetic on arrays (revisit)
Adding an Integer to a Pointer **+i**

- Adding an integer `i` to a pointer `p` yields a pointer to the element `i` places after the one that `p` points to.

- More precisely, if `p` points to the array element `a[k]`, then `p + i` points to `a[k+i]`.
  - IF **p= &a[k]** // p = a+k
  - THEN **p+i == &a[k+i]**

  Special case k=0:
  - IF **p = a = &a[0]**
  - THEN **p+i == &a[i]**

fact3



Pointer arithmetic on arrays (revisit) Subtracting an Integer on a Pointer **–i**

- Subtracting an integer `i` to a pointer `p` yields a pointer to the element `i` places before the one that `p` points to.

- More precisely, if `p` points to the array element `a[k]`, then `p – i` points to `a[k-i]`.
  - IF **`p= &a[k]`**   `// p = a+k`
  - THEN **`p - i == &a[k-i]`**

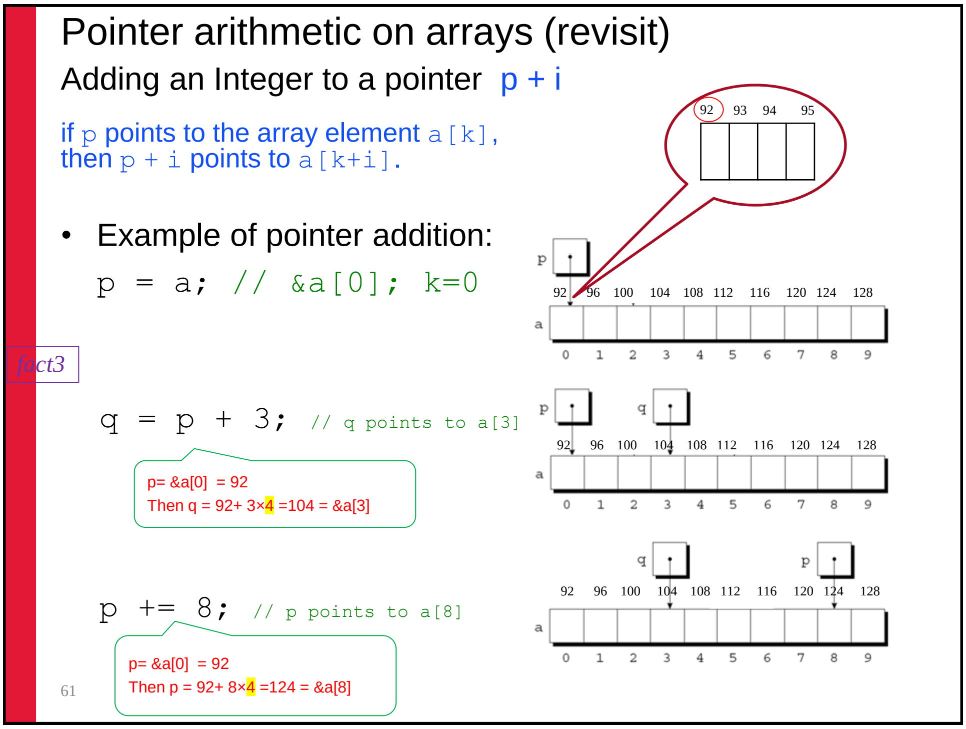
- Assume that the following declarations are in effect:

```
int a[10], *p, *q, i;
```

60   size?

YORK U UNIVERSITÉ UNIVERSITY

60



Pointer arithmetic on arrays (revisit)

Adding an Integer to a pointer p + i

if `p` points to the array element `a[k]`,
then `p + i` points to `a[k+i]`.

- Example of pointer addition:

```
p = a;  // &a[0]; k=0
```

fact3

```
q = p + 3;  // q points to a[3]
```

p= &a[0] = 92
Then q = 92+ 3×4 =104 = &a[3]

```
p += 8;  // p points to a[8]
```

p= &a[0] = 92
Then p = 92+ 8×4 =124 = &a[8]

61

61

Pointer arithmetic on arrays (revisit)

Adding an Integer to a pointer p + i

if `p` points to the array element `a[k]`,
then `p + i` points to `a[k+i]`.

- Example of pointer addition:

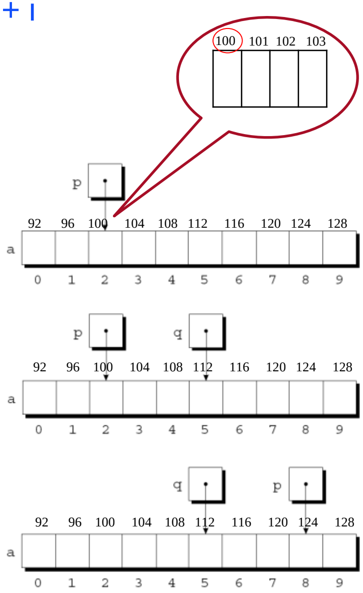
```
p = &a[2]; // k=2 p=a+2
```

```
q = p + 3;//q points to a[   ]
```

p= &a[2] = 100
Then q = 100+ 3×4 =112 = &a[5]

```
p += 6; // p points to a[  ;]
```

p= &a[2] = 100
Then p = 100+ 6×4 =124 = &a[8]



Pointer arithmetic on arrays (revisit)

Subtracting an Integer on a pointer p - i

if `p` points to the array element `a[k]`,
then `p – i` points to `a[k–i]`.

- Example:

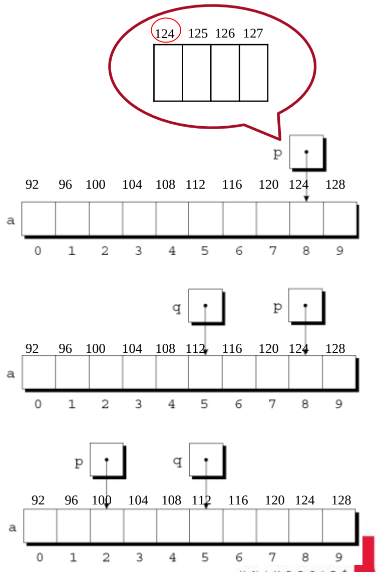
```
p = &a[8]; // k=8  p=a+8
```

```
q = p - 3; // q points to a[8-3]
```

p= &a[8] = 124
Then q = 124 - 3×4 =112 = &a[5]

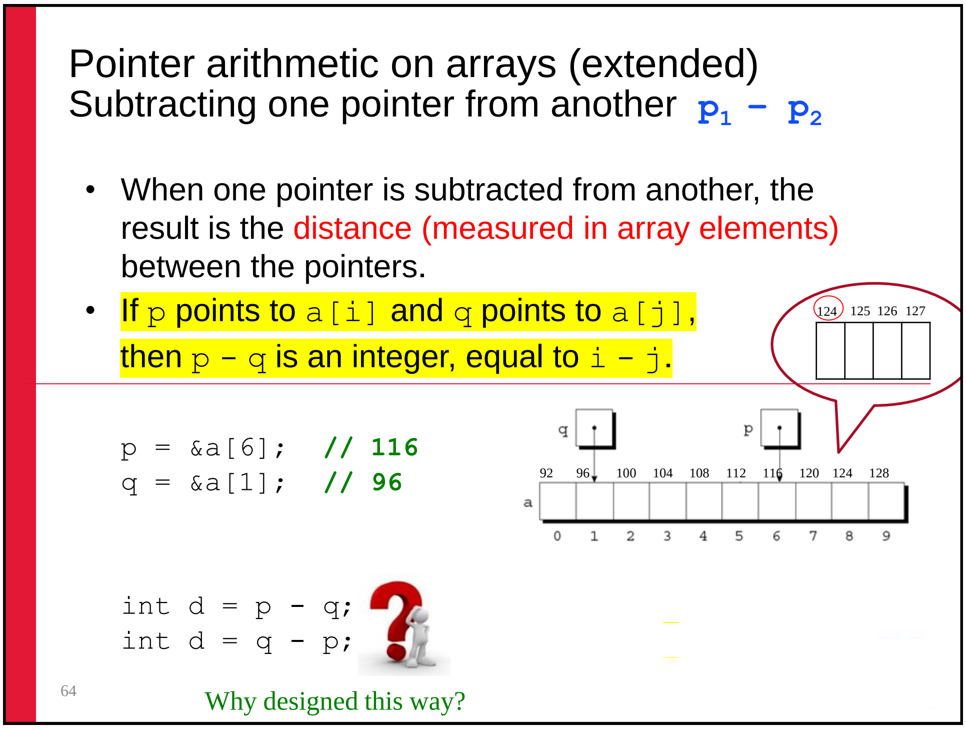
```
p -= 6; // p points to a[8-6]
```

p= &a[8] = 124
Then p = 124 - 6×4 =100 = &a[2]



p-20?

UNIVERSITÉ
UNIVERSITY



Pointer arithmetic on arrays (extended)
Subtracting one pointer from another $p_1 - p_2$

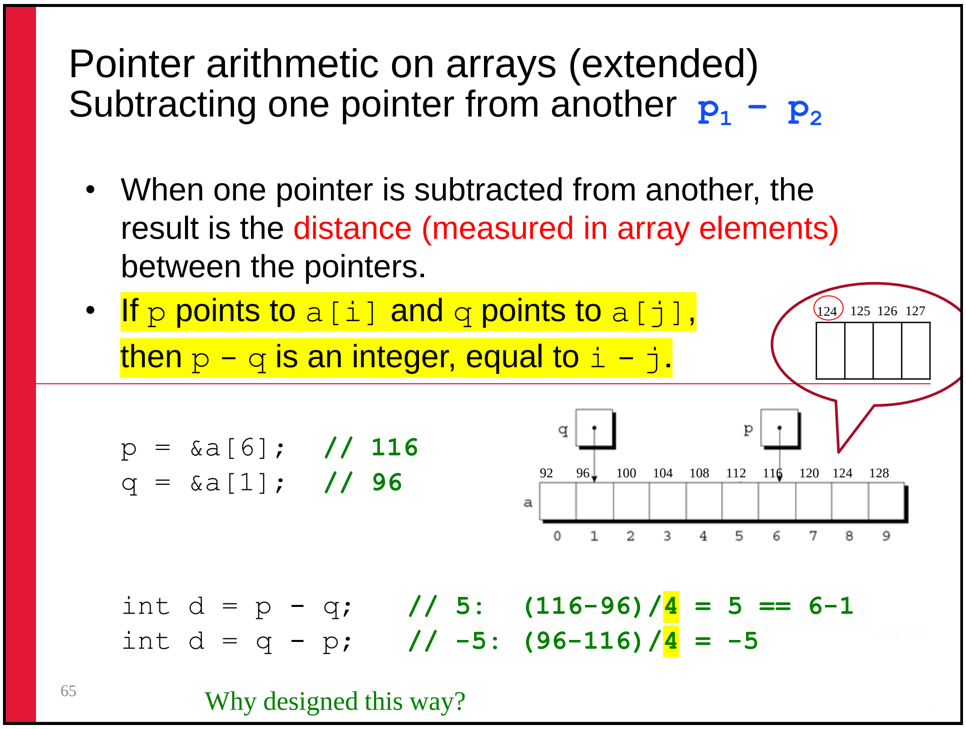
- When one pointer is subtracted from another, the result is the distance (measured in array elements) between the pointers.
- If `p` points to `a[i]` and `q` points to `a[j]`, then `p – q` is an integer, equal to `i – j`.

```
p = &a[6];   // 116
q = &a[1];   // 96
```

124 125 126 127
q
p
92 96 100 104 108 112 116 120 124 128
a
0 1 2 3 4 5 6 7 8 9

```
int d = p - q;
int d = q - p;
```

Why designed this way?



Pointer arithmetic on arrays (extended)
Subtracting one pointer from another $p_1 - p_2$

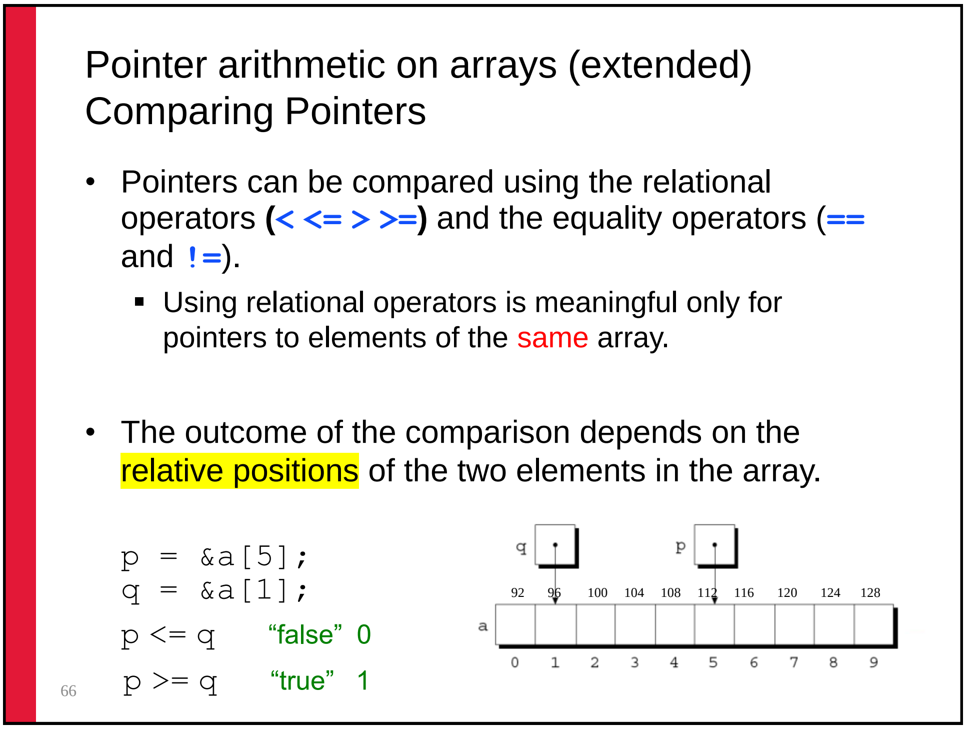
- When one pointer is subtracted from another, the result is the distance (measured in array elements) between the pointers.
- If `p` points to `a[i]` and `q` points to `a[j]`, then `p – q` is an integer, equal to `i – j`.

```
p = &a[6];   // 116
q = &a[1];   // 96
```

124 125 126 127
q
p
92 96 100 104 108 112 116 120 124 128
a
0 1 2 3 4 5 6 7 8 9

```
int d = p - q;    // 5:  (116-96)/4 = 5 == 6-1
int d = q - p;    // -5: (96-116)/4 = -5
```

Why designed this way?



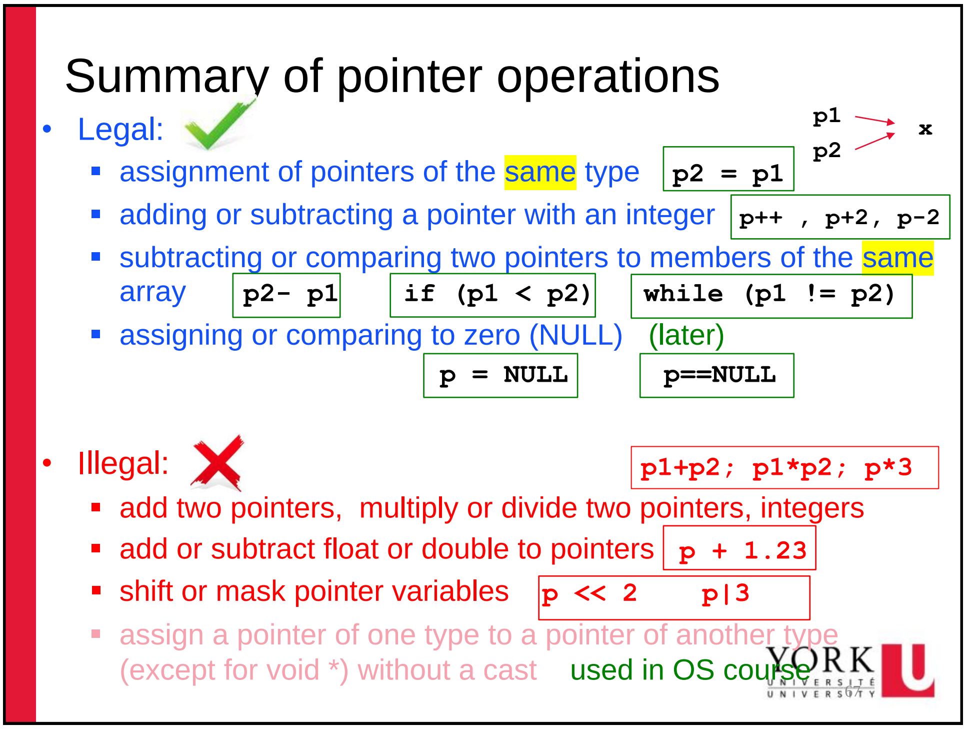
Pointer arithmetic on arrays (extended) Comparing Pointers

- Pointers can be compared using the relational operators **(< <= > >=)** and the equality operators (**==** and **!=**).
  - Using relational operators is meaningful only for pointers to elements of the same array.

- The outcome of the comparison depends on the relative positions of the two elements in the array.

```
p = &a[5];
q = &a[1];
p <= q      "false"  0
p >= q      "true"   1
```

q
p
92
96
100
104
108
112
116
120
124
128
a
0
1
2
3
4
5
6
7
8
9

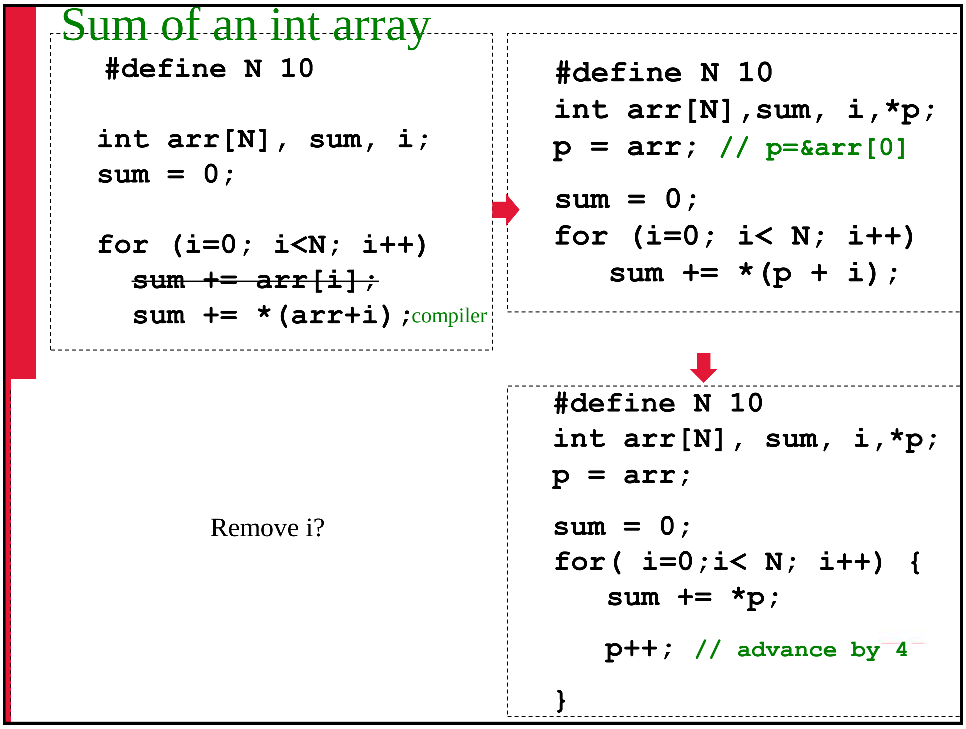


Summary of pointer operations

p1 → x
p2 → x

- Legal: ✓
  - assignment of pointers of the same type `p2 = p1`
  - adding or subtracting a pointer with an integer `p++ , p+2, p-2`
  - subtracting or comparing two pointers to members of the same array `p2- p1` `if (p1 < p2)` `while (p1 != p2)`
  - assigning or comparing to zero (NULL) (later) `p = NULL` `p==NULL`

- Illegal: ✗ `p1+p2; p1*p2; p*3`
  - add two pointers, multiply or divide two pointers, integers
  - add or subtract float or double to pointers `p + 1.23`
  - shift or mask pointer variables `p << 2` `p|3`
  - assign a pointer of one type to a pointer of another type (except for void *) without a cast used in OS course



Sum of an int array

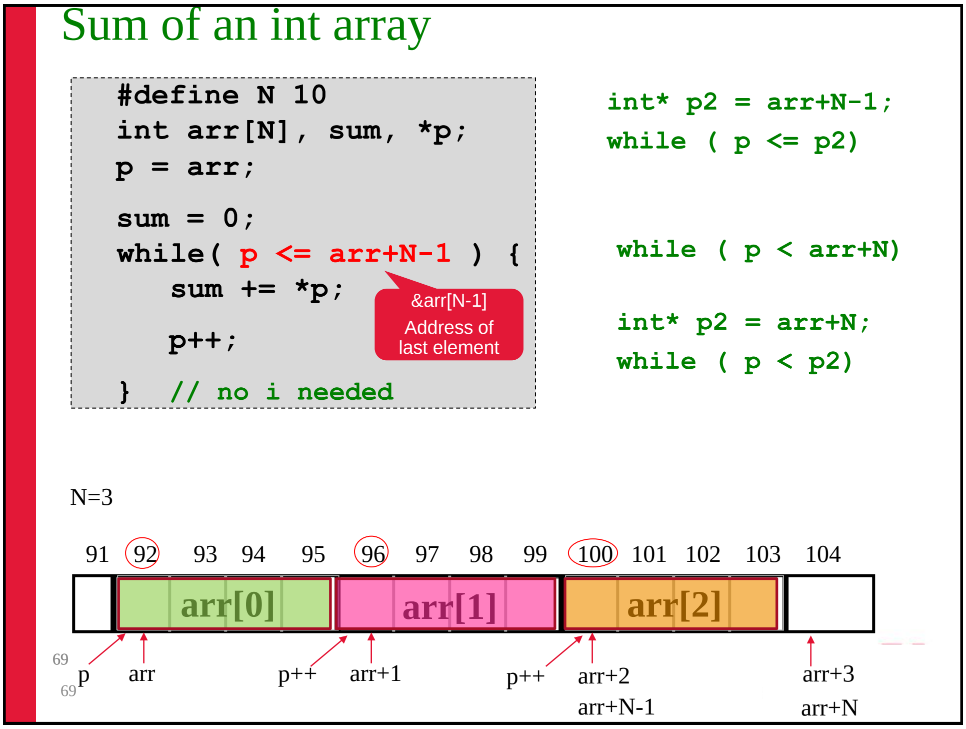
```
#define N 10

int arr[N], sum, i;
sum = 0;

for (i=0; i<N; i++)
  ~~sum += arr[i];~~
  sum += *(arr+i);compiler
```

```
#define N 10
int arr[N],sum, i,*p;
p = arr; // p=&arr[0]
sum = 0;
for (i=0; i< N; i++)
    sum += *(p + i);
```

```
#define N 10
int arr[N], sum, i,*p;
p = arr;
sum = 0;
for( i=0;i< N; i++) {
    sum += *p;

    p++; // advance by 4

}
```

Remove i?

68



Sum of an int array

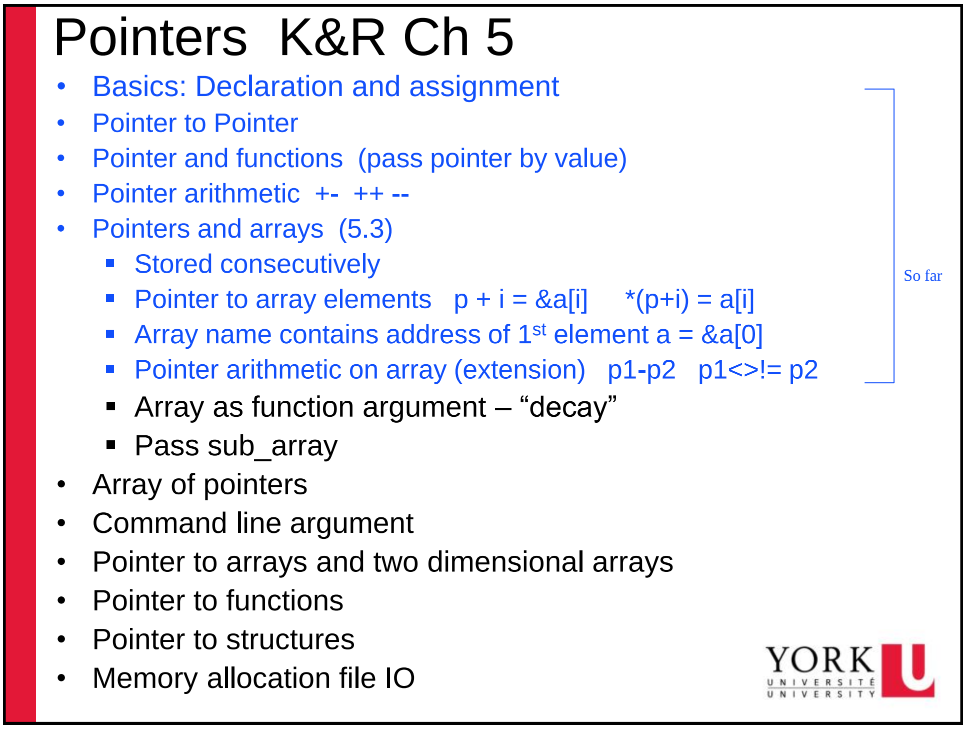
```
#define N 10
int arr[N], sum, *p;
p = arr;
sum = 0;
while( p <= arr+N-1 ) {
    sum += *p;

    p++;

}  // no i needed
```

&arr[N-1]
Address of
last element

```
int* p2 = arr+N-1;
while ( p <= p2)


while ( p < arr+N)

int* p2 = arr+N;
while ( p < p2)
```

N=3

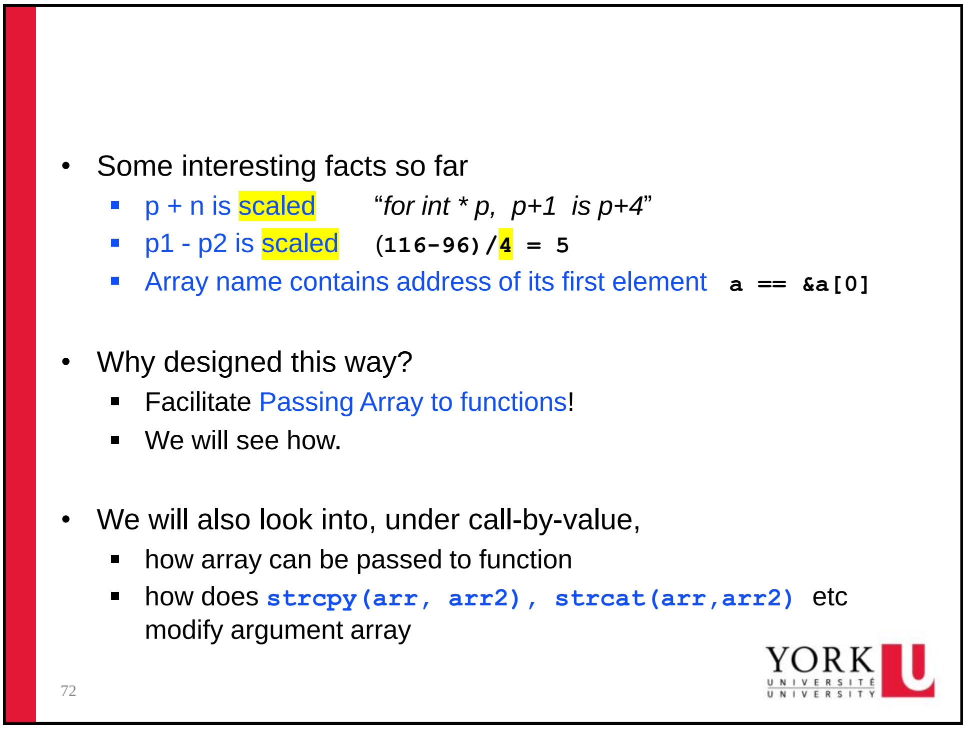
69



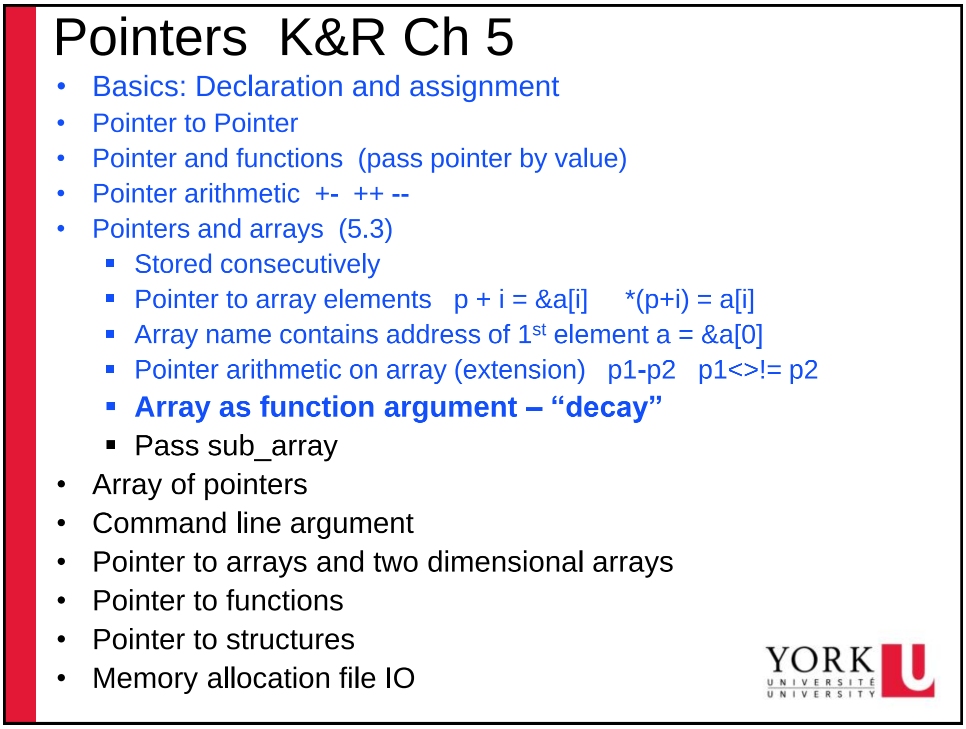
Pointers K&R Ch 5

- Basics: Declaration and assignment
- Pointer to Pointer
- Pointer and functions (pass pointer by value)
- Pointer arithmetic +- ++ --
- Pointers and arrays (5.3)
  - Stored consecutively
  - Pointer to array elements   p + i = &a[i]   *(p+i) = a[i]
  - Array name contains address of 1st element a = &a[0]
  - Pointer arithmetic on array (extension)   p1-p2   p1<>!= p2
  - Array as function argument – “decay”
  - Pass sub_array
- Array of pointers
- Command line argument
- Pointer to arrays and two dimensional arrays
- Pointer to functions
- Pointer to structures
- Memory allocation file IO

So far

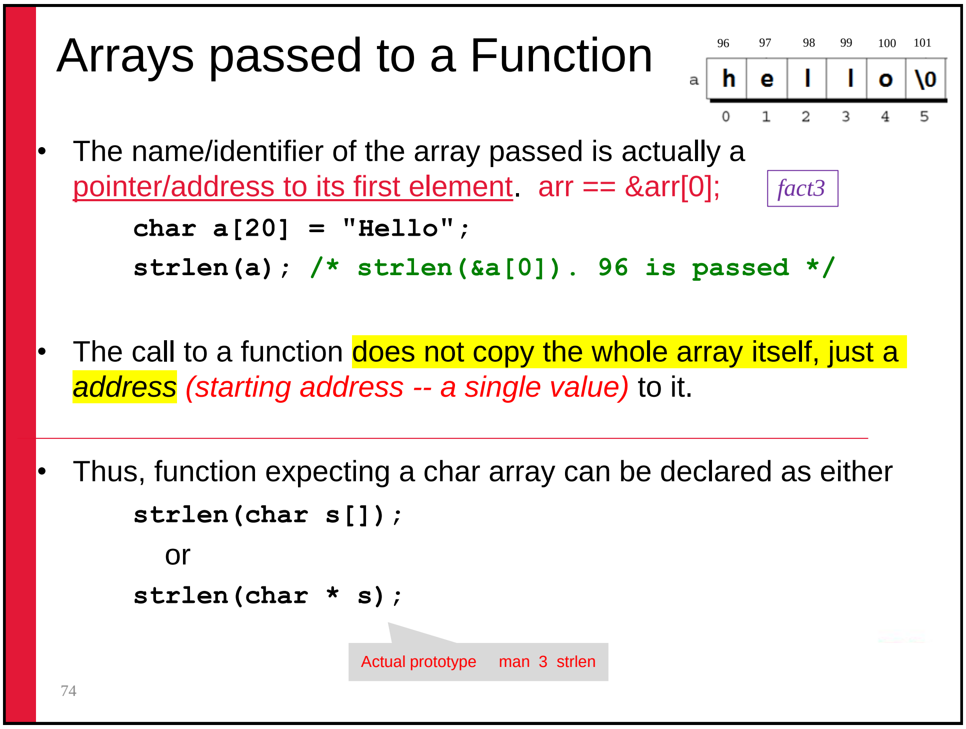


- Some interesting facts so far
  - p + n is scaled   *“for int * p, p+1 is p+4”*
  - p1 - p2 is scaled   `(116-96)/4 = 5`
  - Array name contains address of its first element `a == &a[0]`
- Why designed this way?
  - Facilitate Passing Array to functions!
  - We will see how.
- We will also look into, under call-by-value,
  - how array can be passed to function
  - how does `strcpy(arr, arr2)`, `strcat(arr,arr2)` etc modify argument array



Pointers K&R Ch 5

- Basics: Declaration and assignment
- Pointer to Pointer
- Pointer and functions (pass pointer by value)
- Pointer arithmetic +- ++ --
- Pointers and arrays (5.3)
  - Stored consecutively
  - Pointer to array elements   p + i = &a[i]   *(p+i) = a[i]
  - Array name contains address of 1st element a = &a[0]
  - Pointer arithmetic on array (extension)   p1-p2   p1<>!= p2
  - **Array as function argument – “decay”**
  - Pass sub_array
- Array of pointers
- Command line argument
- Pointer to arrays and two dimensional arrays
- Pointer to functions
- Pointer to structures
- Memory allocation file IO



Arrays passed to a Function

	96	97	98	99	100	101
a	h	e	l	l	o	\0
	0	1	2	3	4	5

- The name/identifier of the array passed is actually a pointer/address to its first element. arr == &arr[0];

fact3

```
char a[20] = "Hello";
strlen(a); /* strlen(&a[0]). 96 is passed */
```

- The call to a function does not copy the whole array itself, just a *address (starting address -- a single value)* to it.

- Thus, function expecting a char array can be declared as either

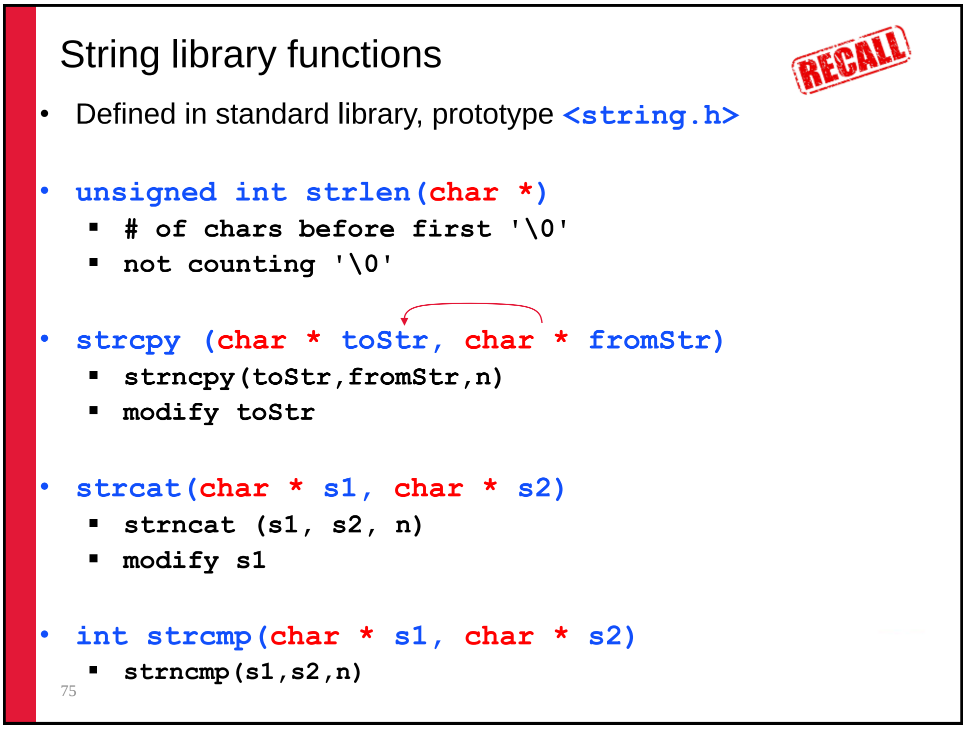
```
strlen(char s[]);
```

or

```
strlen(char * s);
```

Actual prototype   man 3 strlen

74

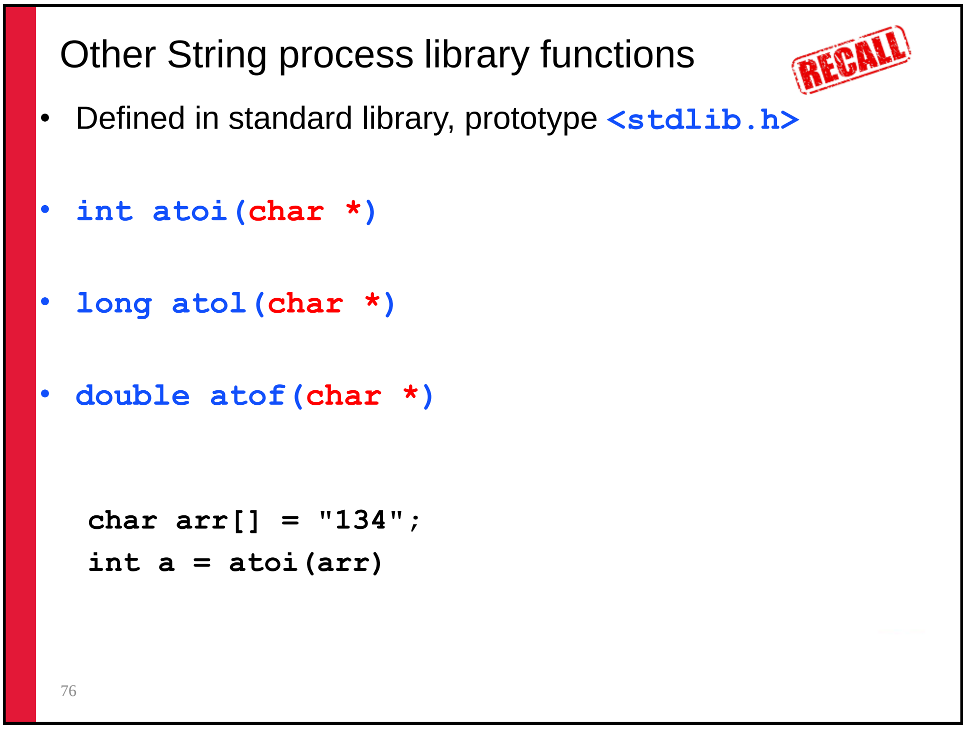


String library functions

RECALL

- Defined in standard library, prototype <string.h>
- unsigned int strlen(char *)
  - # of chars before first '\0'
  - not counting '\0'
- strcpy (char * toStr, char * fromStr)
  - strncpy(toStr,fromStr,n)
  - modify toStr
- strcat(char * s1, char * s2)
  - strncat (s1, s2, n)
  - modify s1
- int strcmp(char * s1, char * s2)
  - strncmp(s1,s2,n)

75



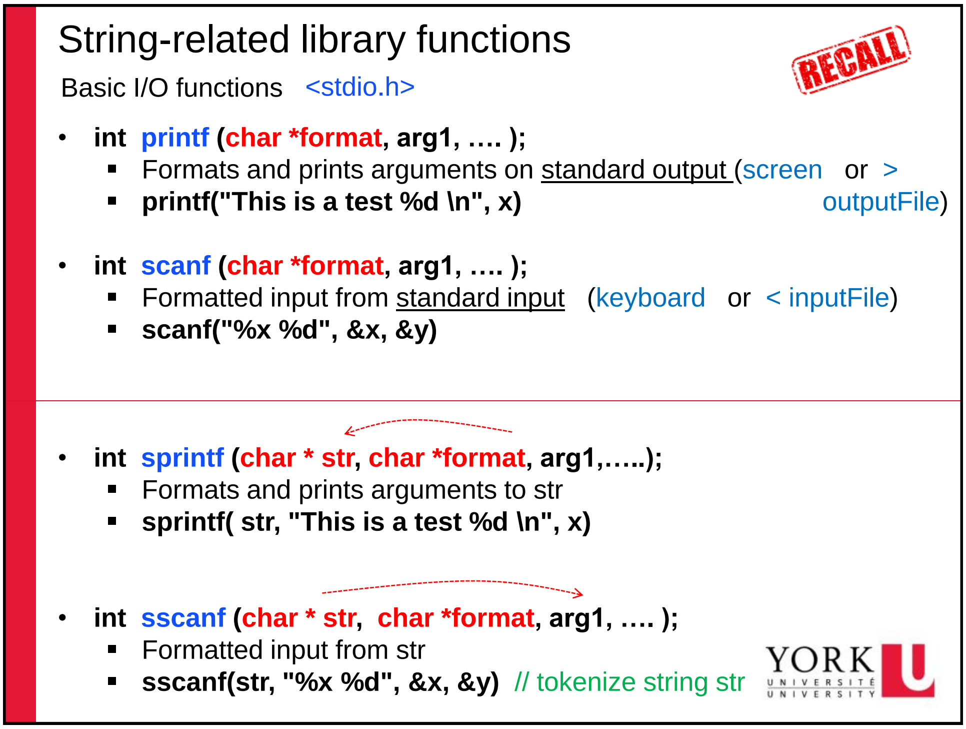
Other String process library functions

RECALL

- Defined in standard library, prototype <stdlib.h>
- int atoi(char *)
- long atol(char *)
- double atof(char *)

```
char arr[] = "134";
int a = atoi(arr)
```

76



String-related library functions

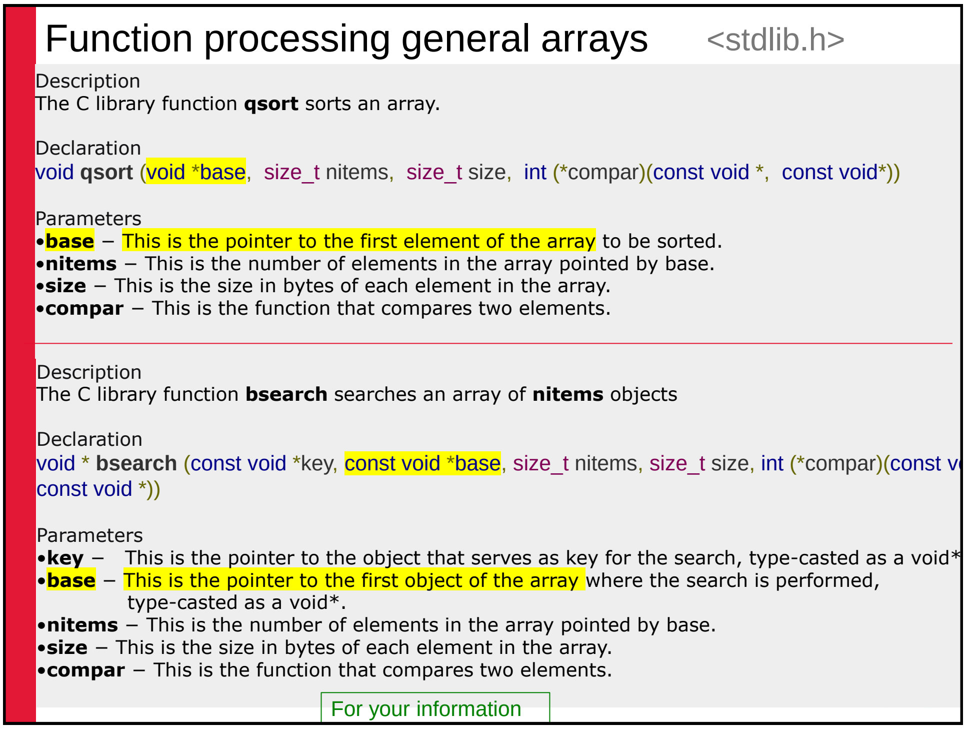
Basic I/O functions <stdio.h>

RECALL

- **int printf (char *format, arg1, ….);**
  - Formats and prints arguments on standard output (screen or > outputFile)
  - **printf("This is a test %d \n", x)**
- **int scanf (char *format, arg1, ….);**
  - Formatted input from standard input (keyboard or < inputFile)
  - **scanf("%x %d", &x, &y)**
- **int sprintf (char * str, char *format, arg1,…..);**
  - Formats and prints arguments to str
  - **sprintf(str, "This is a test %d \n", x)**
- **int sscanf (char * str, char *format, arg1, ….);**
  - Formatted input from str
  - **sscanf(str, "%x %d", &x, &y)** // tokenize string str

YORK UNIVERSITÉ UNIVERSITY

77



Function processing general arrays <stdlib.h>

Description
The C library function **qsort** sorts an array.

Declaration
void **qsort** (void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*))

Parameters
•**base** – This is the pointer to the first element of the array to be sorted.
•**nitems** – This is the number of elements in the array pointed by base.
•**size** – This is the size in bytes of each element in the array.
•**compar** – This is the function that compares two elements.

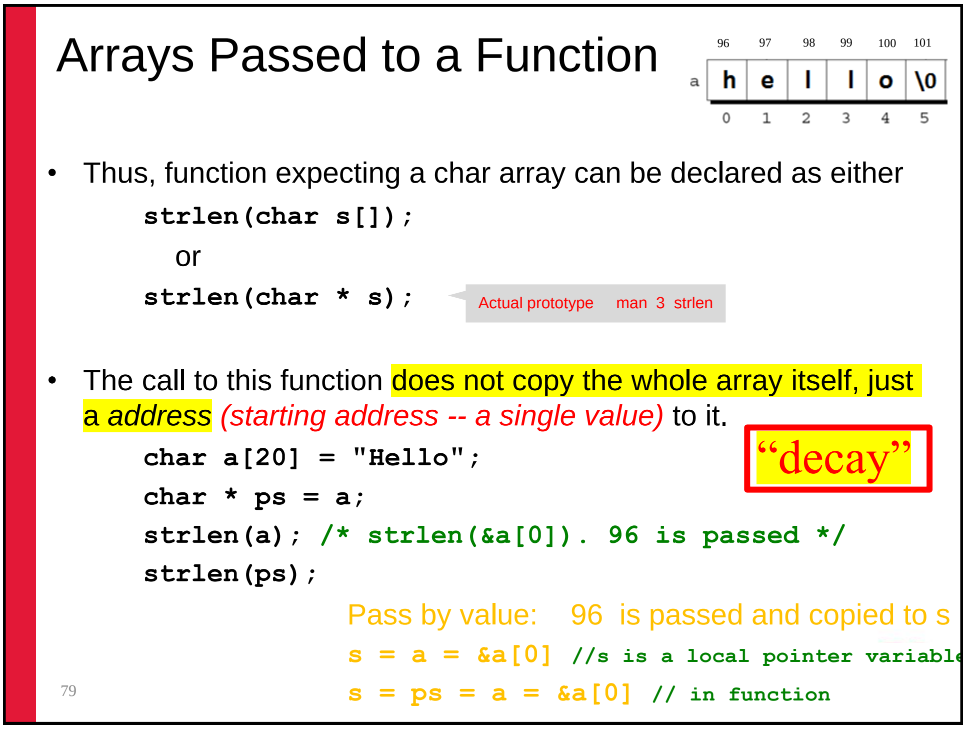
Description
The C library function **bsearch** searches an array of **nitems** objects

Declaration
void * **bsearch** (const void *key, const void *base, size_t nitems, size_t size, int (*compar)(const v
const void *))

Parameters
•**key** – This is the pointer to the object that serves as key for the search, type-casted as a void*
•**base** – This is the pointer to the first object of the array where the search is performed, type-casted as a void*.
•**nitems** – This is the number of elements in the array pointed by base.
•**size** – This is the size in bytes of each element in the array.
•**compar** – This is the function that compares two elements.

For your information

78



Arrays Passed to a Function

	96	97	98	99	100	101
a	h	e	l	l	o	\0
	0	1	2	3	4	5

- Thus, function expecting a char array can be declared as either

```
strlen(char s[]);
```

or

```
strlen(char * s);
```

Actual prototype   man 3 strlen

- The call to this function does not copy the whole array itself, just a *address* *(starting address -- a single value)* to it.

“decay”

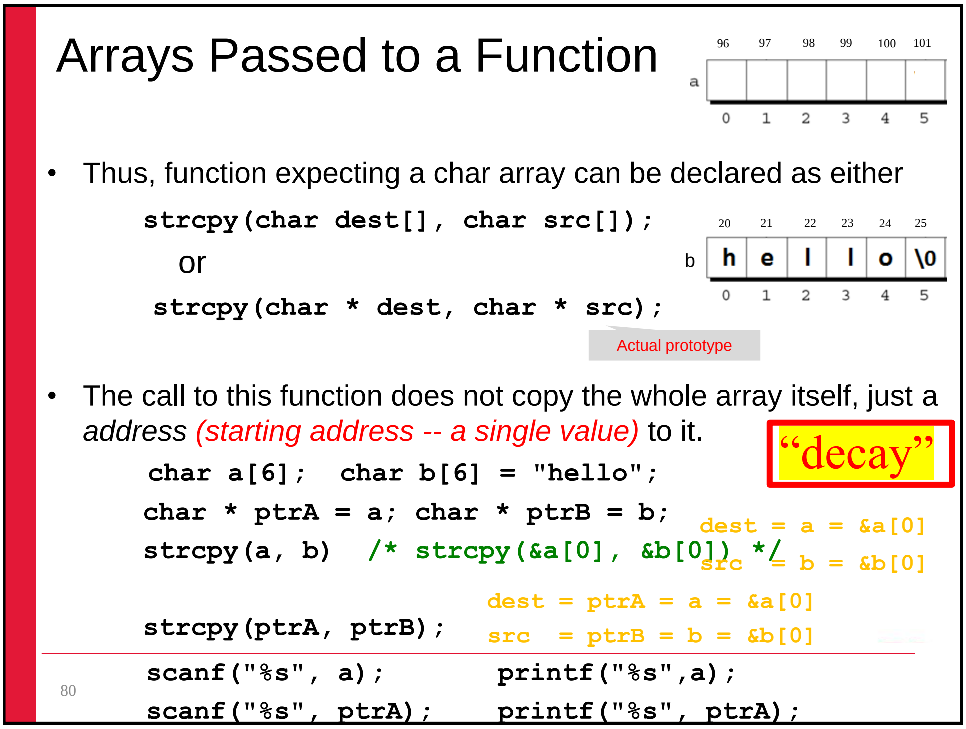
```
char a[20] = "Hello";
char * ps = a;
strlen(a); /* strlen(&a[0]). 96 is passed */
strlen(ps);
```

Pass by value: 96 is passed and copied to s

```
s = a = &a[0] //s is a local pointer variable
s = ps = a = &a[0] // in function
```

79

79



Arrays Passed to a Function

	96	97	98	99	100	101
a						
	0	1	2	3	4	5

- Thus, function expecting a char array can be declared as either

```
strcpy(char dest[], char src[]);
```

or

```
 strcpy(char * dest, char * src);
```

	20	21	22	23	24	25
b	h	e	l	l	o	\0
	0	1	2	3	4	5

Actual prototype

- The call to this function does not copy the whole array itself, just a *address* *(starting address -- a single value)* to it.

“decay”

```
 char a[6];  char b[6] = "hello";
char * ptrA = a; char * ptrB = b;
strcpy(a, b)   /* strcpy(&a[0], &b[0]) */
```

```
dest = a = &a[0]
src  = b = &b[0]
```

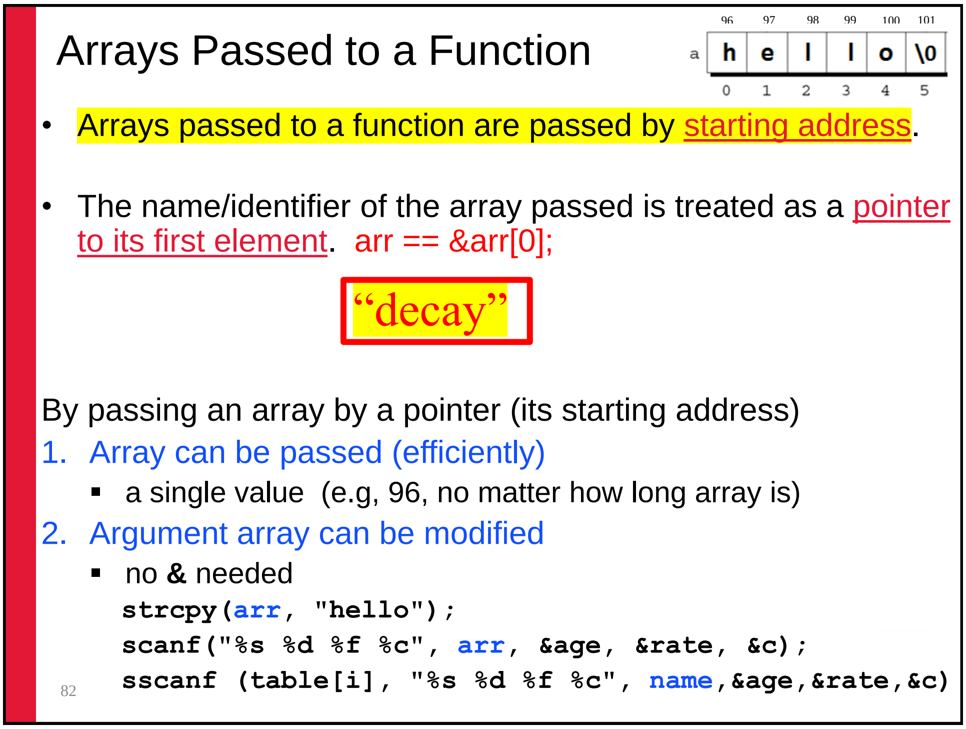
```
dest = ptrA = a = &a[0]
src  = ptrB = b = &b[0]
```

```
strcpy(ptrA, ptrB);
```

```
 scanf("%s", a);         printf("%s",a);
 scanf("%s", ptrA);      printf("%s", ptrA);
```

80

80



Arrays Passed to a Function

	96	97	98	99	100	101
a	h	e	l	l	o	\0
	0	1	2	3	4	5

- Arrays passed to a function are passed by starting address.
- The name/identifier of the array passed is treated as a pointer to its first element. arr == &arr[0];

“decay”

By passing an array by a pointer (its starting address)

1. Array can be passed (efficiently)
   - a single value (e.g, 96, no matter how long array is)
2. Argument array can be modified
   - no **&** needed

```
strcpy(arr, "hello");
scanf("%s %d %f %c", arr, &age, &rate, &c);
sscanf (table[i], "%s %d %f %c", name,&age,&rate,&c)
```