Introduction to C, C++, and Unix/Linux

CS 60

Lecture 11: Pointers

Today

→ Pointers!!!

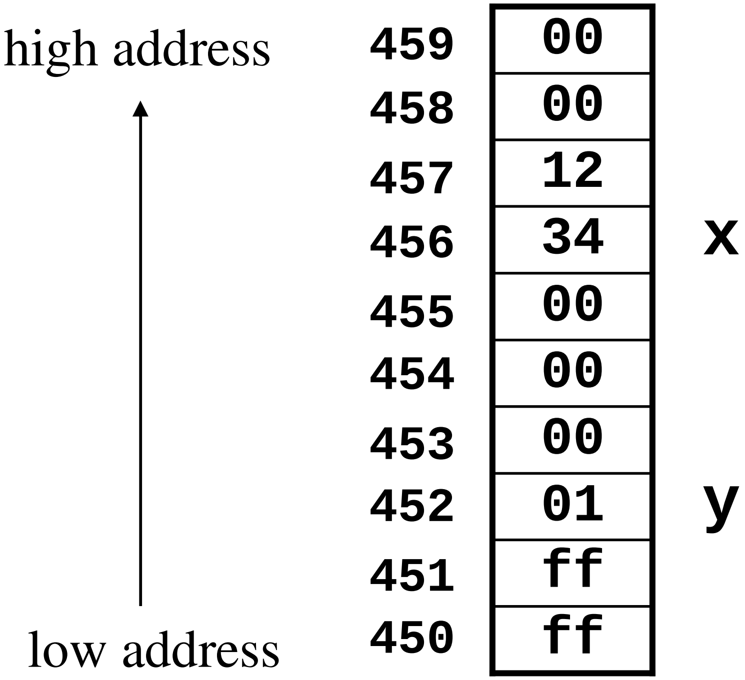
→ Read [KR] Chapters 1-7

Pointers

- A pointer is a variable that contains the address of a variable
  - Pointers are closely related to arrays
- Memory addresses (on our machines) are 4 bytes long (32 bits, 4G addresses)
  - So a pointer must be 4 bytes

`char *x;`	**`char`** is 1 byte	**`x`** is 4 bytes
`int *y;`	**`int`** is 4 bytes	**`y`** is 4 bytes
`double *z;`	**`double`** is 8 bytes	**`z`** is 4 bytes
`FILE *fp;`	**`FILE`** is 148 bytes	**`fp`** is 4 bytes

Memory reminder – What are the values of the integers **`x`** and **`y`**?



459	00	
458	00	
457	12	
456	34	x
455	00	
454	00	
453	00	
452	01	y
451	ff	
450	ff	

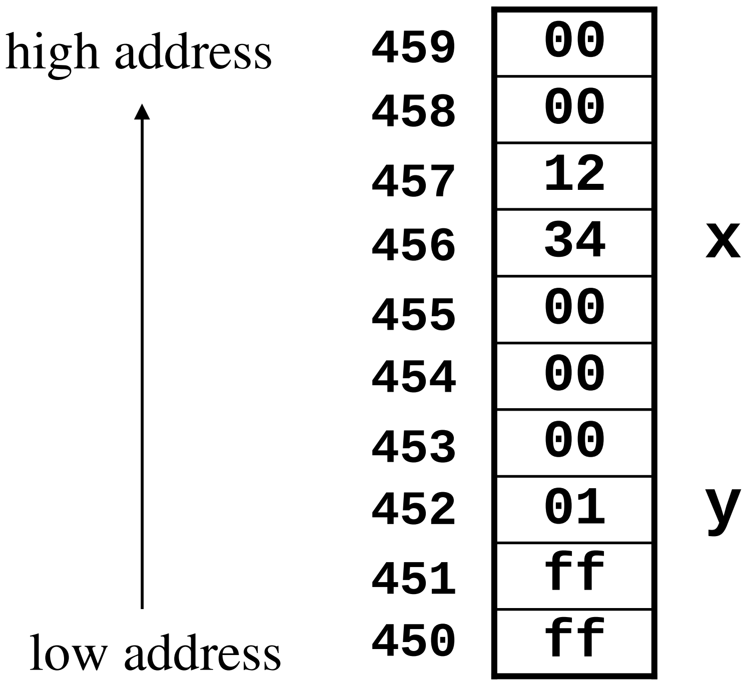
```
int x;
int y;
```

`x` is **`0x1234`**

`y` is **`1`**

Little-endian – The least significant byte has the lowest address ("little end first")

What is the address of the integers **`x`** and **`y`**?



high address

low address

459	00	
458	00	
457	12	
456	34	x
455	00	
454	00	
453	00	
452	01	y
451	ff	
450	ff	

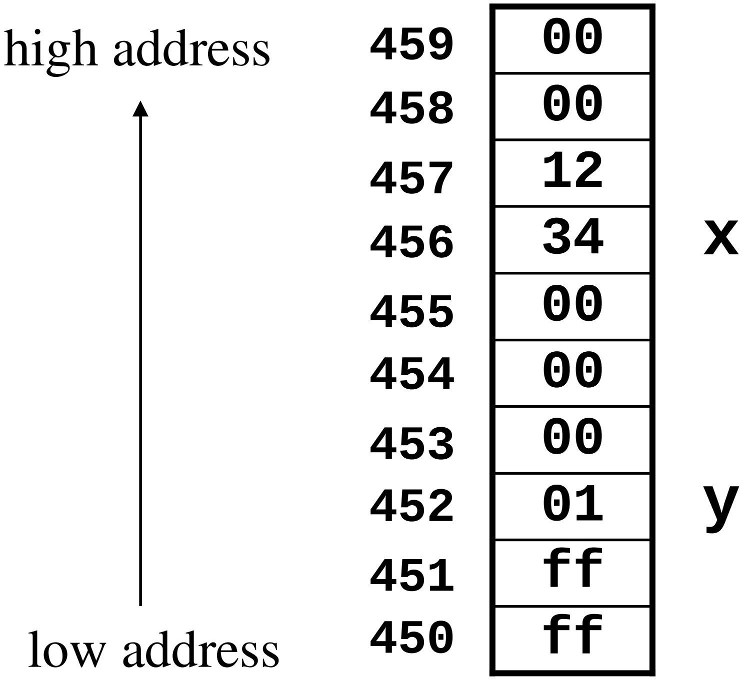
```
int x;
int y;
```

`x` is at address 456
`y` is at address 452

Same as saying

`x` “starts at” address 456
`y` “starts at” address 452

In other words



high address

459	00	
458	00	
457	12	
456	34	x
455	00	
454	00	
453	00	
452	01	y
451	ff	
450	ff	

low address

```
int x;
int y;
```

x is **0x1234**
y is **1**

&x is **456**
&y is **452**

& – “address” operator

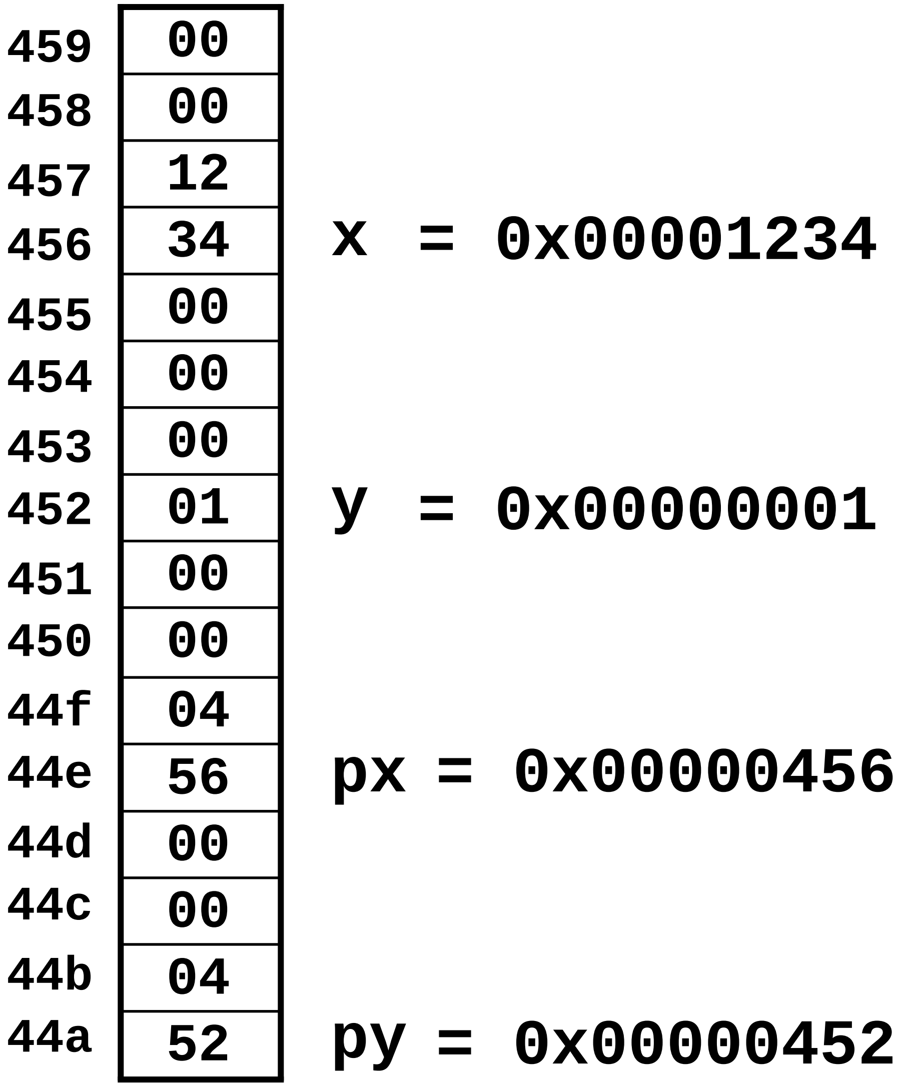
&x – “address of **x**”

int – 4 bytes   pointer – 4 bytes

```
int x;
int y;
int *px;
int *py;

x = 0x1234;
y = 1;

px = &x;
py = &y;
```



459	00	
458	00	
457	12	
456	34	x = 0x00001234
455	00	
454	00	
453	00	
452	01	y = 0x00000001
451	00	
450	00	
44f	04	
44e	56	px = 0x00000456
44d	00	
44c	00	
44b	04	
44a	52	py = 0x00000452

Keep going!

```
int x;
int *px;
int **ppx;
int ***pppx;

x = 0x1234;
px = &x;
ppx = &px;
pppx = &ppx;
```

459	00
458	00
457	12
456	34
455	00
454	00
453	04
452	56
451	00
450	00
44f	04
44e	52
44d	00
44c	00
44b	04
44a	4e

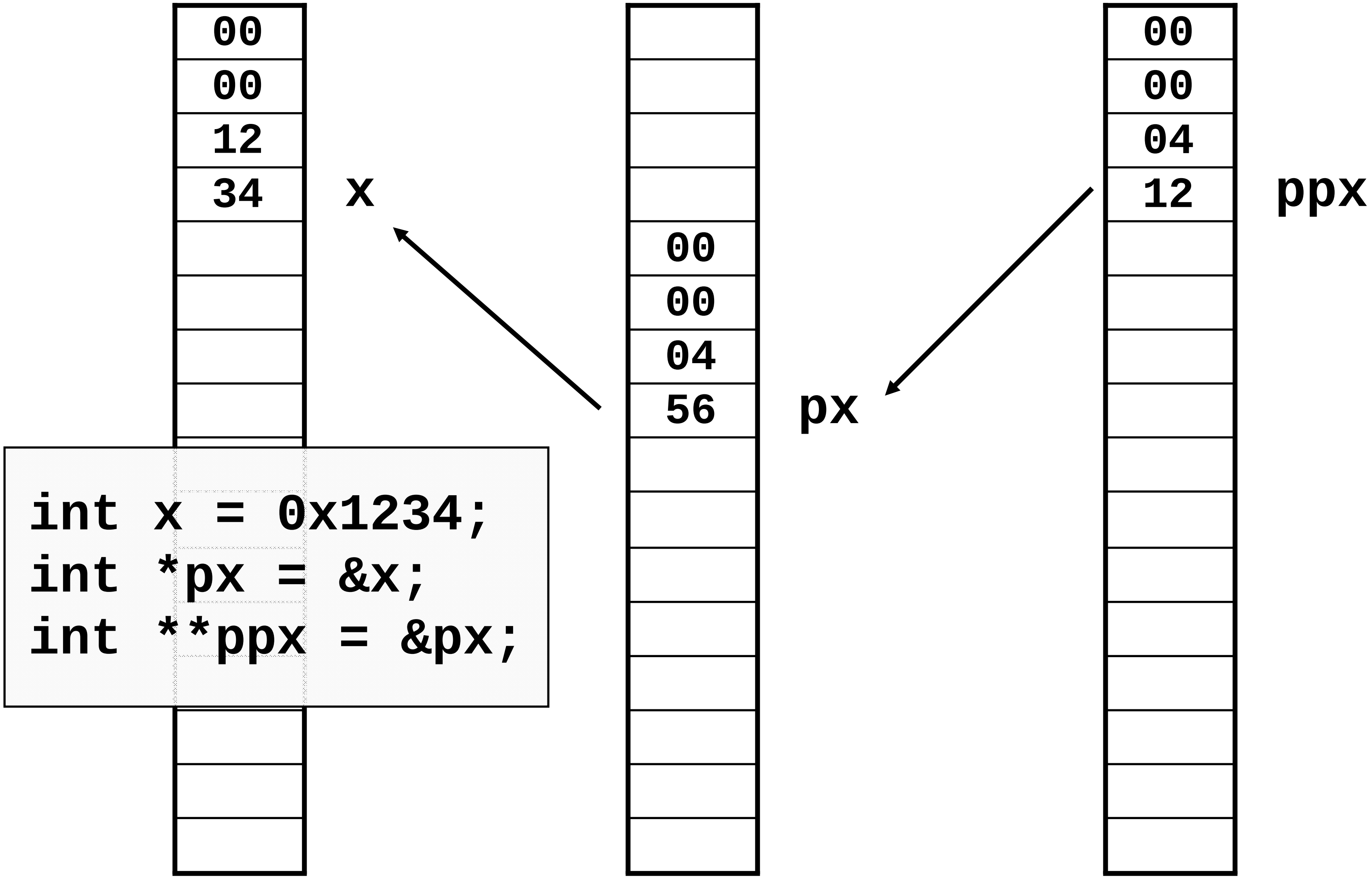
`x = 0x00001234`

`px = 0x00000456`

`ppx = 0x00000452`

`pppx = 0x0000044e`

It helps to visualize pointers like this:



```
int x = 0x1234;
int *px = &x;
int **ppx = &px;
```

Following the pointer trail

- The address operator **&** returns the address of the variable
  - **&x** is the address of the variable **x**
  - **&** creates an address from a variable
- The indirection or dereferencing operator ***** returns the value that is stored in the memory address **x**
  - ***x** is the value of the variable at memory location **x**
  - ***** follows an address to create a variable

```
int x = 7;
int *px = &x;
int **ppx = &px;
int ***pppx = &ppx;
int ****ppppx = &pppx;
int *****pppppx = &ppppx;
```

How would we change the value of “x” using the pointer declared last from 7 to 35?

```
*****pppppx = 35;
```

The declaration syntax is supposed to be a mnemonic.

```
int *px;  // This could be interpreted as “*px is an integer”
int *****pppppx;  // And this could be interpreted as “*****pppppx is an integer”
```

So what's so special about arrays?

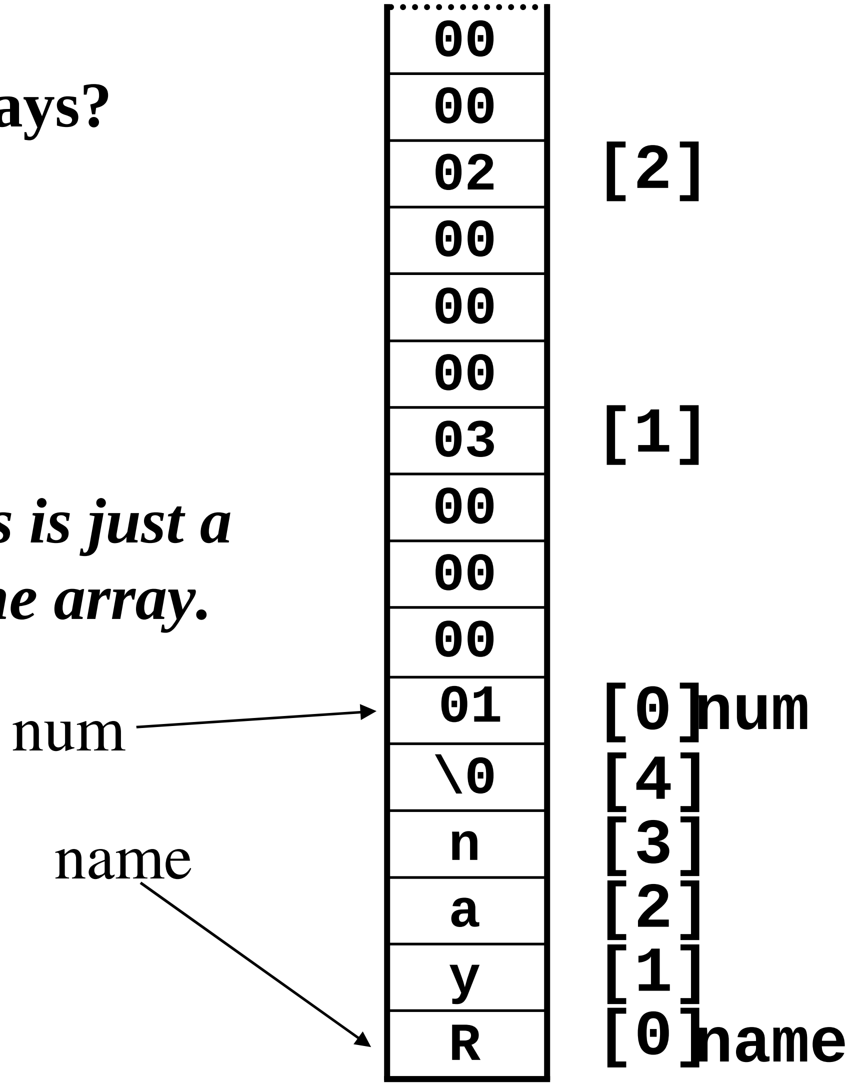
char name[5] = "Ryan";

int num[4] = { 1, 3, 2, 7 };

The array name without indices is just a pointer to the first element of the array.

That means these two assignments are the same!

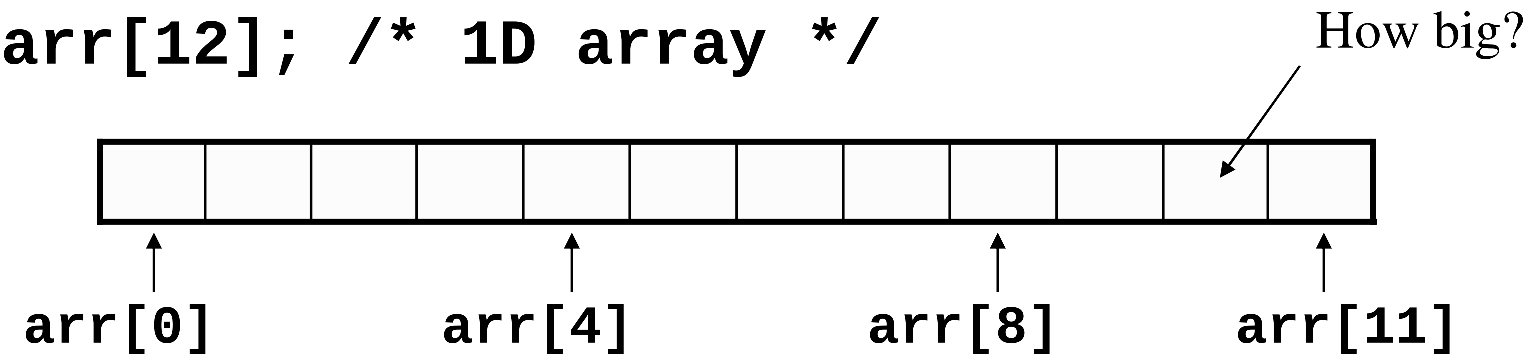
char *me = &name[0];
char *me = name;



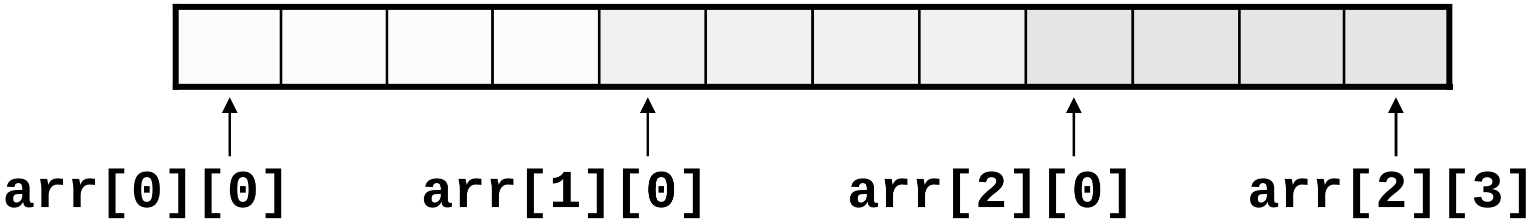
Pointers and arrays

- We’ve seen how 1D and 2D arrays are stored in memory

```
int arr[12]; /* 1D array */
```



```
int arr[3][4]; /* 2D array */
```



1D array – via pointer

```
int *par = (int *)malloc(12*sizeof(int));
```

`*par` **`*(par+4)`** **`*(par+8)`** **`*(par+11)`**

`par+4` is the address

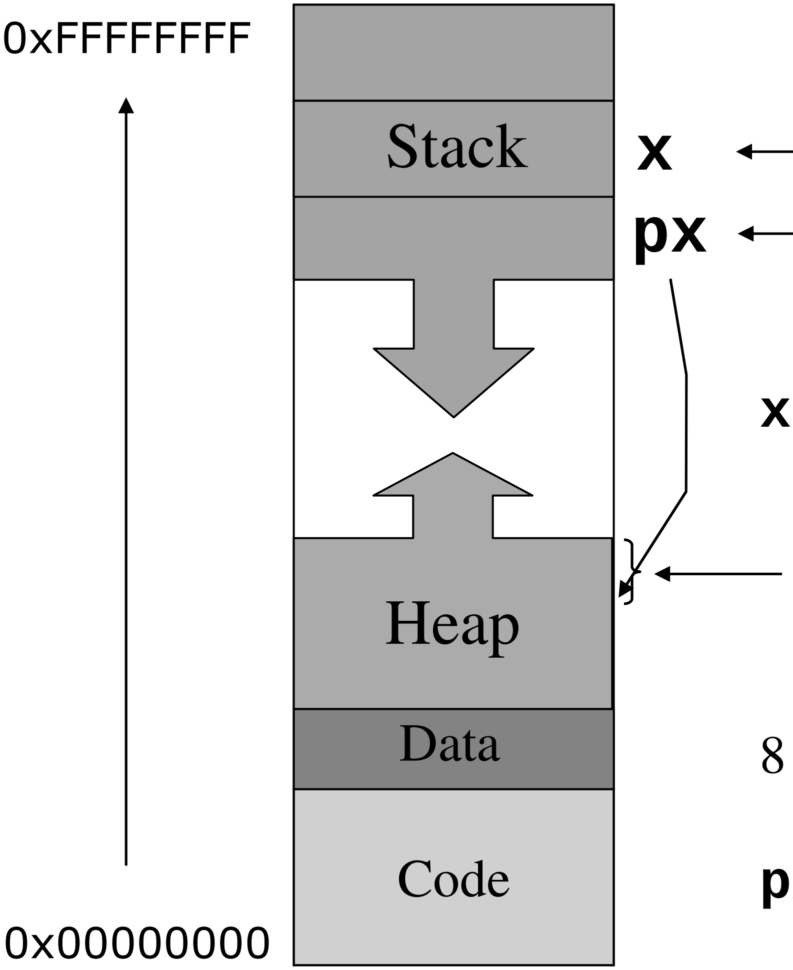
`*(par+4)` is the value of the integer at **`par+4`**

Dynamic memory allocation

- Variables are allocated memory space
  - Stack – local variables
  - Heap – dynamically allocated variables
- "Dynamic allocation" means that memory is actively managed by the programmer
- Functions used for this:
  - malloc, calloc, realloc, free
  - Defined in the C standard library
  - **`#include <stdlib.h>`**

```
px = (int *)malloc(2*sizeof(int));
```

Better



```
int x;
int *px;
```

x and **px** are local variables on the stack

```
px = (int *)malloc(8);
```

8 bytes (2 ints) of memory are allocated on the heap

px points to the beginning of this memory

`void *malloc(int size)`

`size_t`

Allocated ***`size`*** <u>bytes</u> of heap memory and returns a pointer to the beginning of the memory

Returns NULL (0) if there's an error (not enough available memory)

The memory is not initialized

Return value should be cast to pointer of the expected type

Good to write for size: **`N*sizeof(type)`**

space for N variables of the specified type

`void *calloc(int num, int size)`

Allocated ***`num`*** elements, ***`size`*** bytes each, of heap memory and returns a pointer to the beginning of the memory

Returns NULL (0) if there’s an error (not enough available memory)

The memory is initialized to zero

`void *realloc(void *ptr, int size)`

Changes the size of the block pointed to by ptr to ***`size`*** bytes and returns a pointer to the (possibly moved) block

The contents will be unchanged up to the lesser of the new and old sizes.

If **`ptr`** is NULL, realloc() behaves like malloc() for the specified size

If **`size`** is zero and **`ptr`** is not a null pointer, the object pointed to is freed

(Probably not used in this course)

`void free(void *ptr)`

Frees the memory (heap) space pointed to by **`ptr`** , which must have been previously allocated by malloc, calloc, or realloc – the memory is then available for further allocation

After memory is freed, it is illegal to access it

If **`ptr`** is NULL, nothing happens (legal)

It is an error if the memory has already been freed (undefined behavior)

`ptr` should (must?) point to the beginning of the memory block (as returned by malloc, etc.)

Allocating and freeing memory

- When a program or function is finished with the heap memory it has allocated, it should call “free” to deallocate the memory
  - This is the programmer’s RESPONSABILITY!
- Common problem – “memory leak”
  - Chunks of memory that is allocated but never freed
  - Builds up over the life of a program and causes it to fail

Example

```
char* myfunc(int num)
{
  char *str = (char *)malloc(32);

  sprintf(str, “Number is %d”, num);
  return(“lost str”);
}
```

Allocated memory for **`str`**, but never freed it!

32 bytes of memory leak

What if this function gets called millions of times?

Example – fixed

```
char* myfunc(int num)
{
  char *str = (char *)malloc(32);

  sprintf(str, “Number is %d”, num);
  free(str);
  return(str);
}
```

The function will return invalid memory!

Example – better way

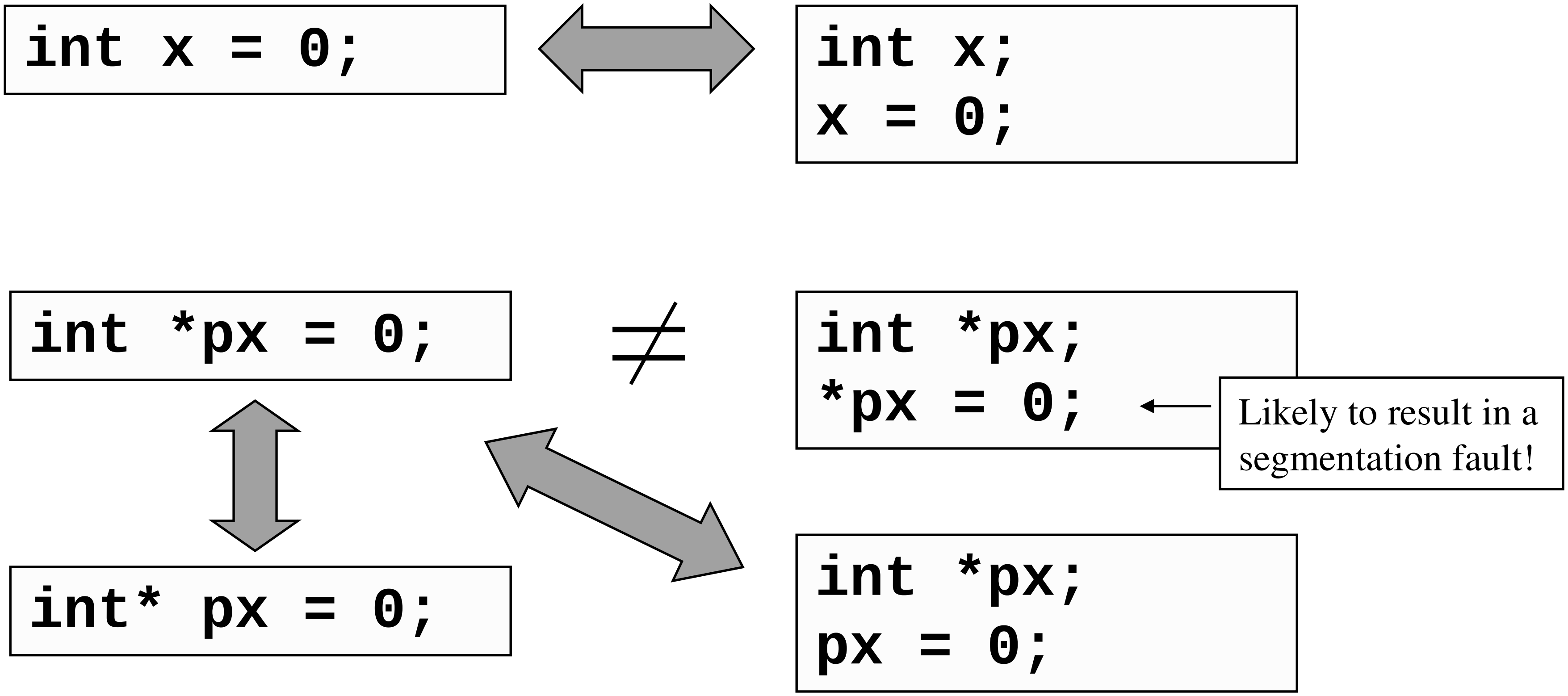
```
char *myfunc(char *str, int num)
{
  sprintf(str, “Number is %d”, num);
  return(str);
}
```

Assumes that the calling function has already allocated memory for **`str`**

It is legal for functions to allocate memory for their local variables and not free the memory, but it is inadvisable!

In that case, the calling program must take responsibility for freeing the memory sometime after the function call

Pointer syntax



```
int x = 0;
```

```
int x;
x = 0;
```

```
int *px = 0;
```

≠

```
int *px;
*px = 0;
```

Likely to result in a segmentation fault!

```
int* px = 0;
```

```
int *px;
px = 0;
```

Know the difference!

```
int x = 0;
```

```
int x;
x = 0;
```

```
int y;
```

```
x = y;
```

```
int *x = 0;
```

```
int *x;
x = 0;
```

```
int y;
```

```
x = &y;
```

```
*x = y;
```

```
char *c = (char *)malloc(100);
int *i = (int *)malloc(100);
double *d = (double *)malloc(100);
uint x, tmp; /* unsigned int */

tmp = (uint)c;
c++;
x = (uint)c – tmp;

tmp = (uint)i;
i++;
x = (uint)i – tmp;
```

```
tmp = (uint)d;
d++;
x = (uint)d – tmp;
```

`x =` 1, 4, and 8

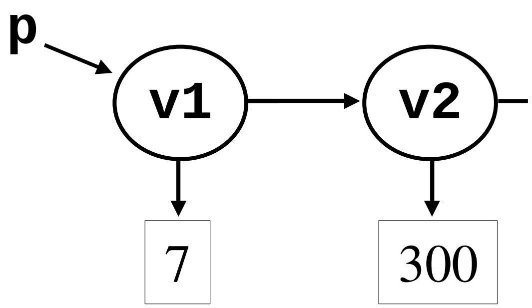
```
typedef union {
  char *pc;
  int *pi;
  double *pd;
  void *pv;
} Megapointer;

Megapointer P;
void *data = (void *)malloc(100);

P.pv = data;
printf("%p, %p, %p, %p\n", P.pc, P.pi, P.pd, P.pv);
// E.g., 0x9856008, 0x9856008, 0x9856008, 0x9856008
P.pi++;
printf("%p, %p, %p, %p\n", P.pc, P.pi, P.pd, P.pv);
// E.g., 0x985600c, 0x985600c, 0x985600c, 0x985600c
```

How large is **`P`**? … 4

```
struct List {
  short int val;
  struct List *next;
};
```



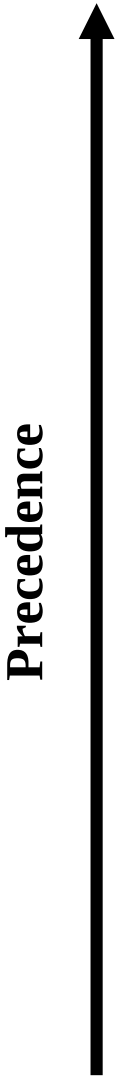
0

```
struct List v1 = { 1, NULL };
struct List v2 = { 2, NULL };
struct List *p = &v1;

v1.val = 7;
*p.val = 7;   No!
(*p).val = 7;
p->val = 7;
p->next = &v2;
```

```
v2.val = 300;
(*(v1.next)).val = 300;
v1.next->val = 300;
(*(p->next)).val = 300;
p->next->val = 300;
```

Table 2.1



Category	Operator	Associativity
Postfix	`()   []   ->   .   ++   --`	Left to right
Unary prefix	`+ - ! ~ ++ -- (type) * & sizeof`	Right to left
Multiplicative	`*   /   %`	Left to right
Additive	`+   -`	Left to right
Shift	`<<   >>`	Left to right
Relational	`<   <=   >   >=`	Left to right
Equality	`==   !=`	Left to right
Bitwise AND	`&`	Left to right
Bitwise XOR	`^`	Left to right
Bitwise OR	`|`	Left to right
Logical AND	`&&`	Left to right
Logical OR	`||`	Left to right
Conditional	`?:`	Right to left
Assignment	`=   +=   -=   *=   /=   %=` `>>=   <<=   &=   ^=   |=`	Right to left
Comma	`,`	Left to right

The `->` operator

- `->` is just a convenience (and used quite often!)
- Rather than `(*p).val`, we can write `p->val`

- So what does `p++->next` do?
- How about `p->next->val++` ?
- And `p->next++->val++` ??? (Try it!)

Pointers and arrays

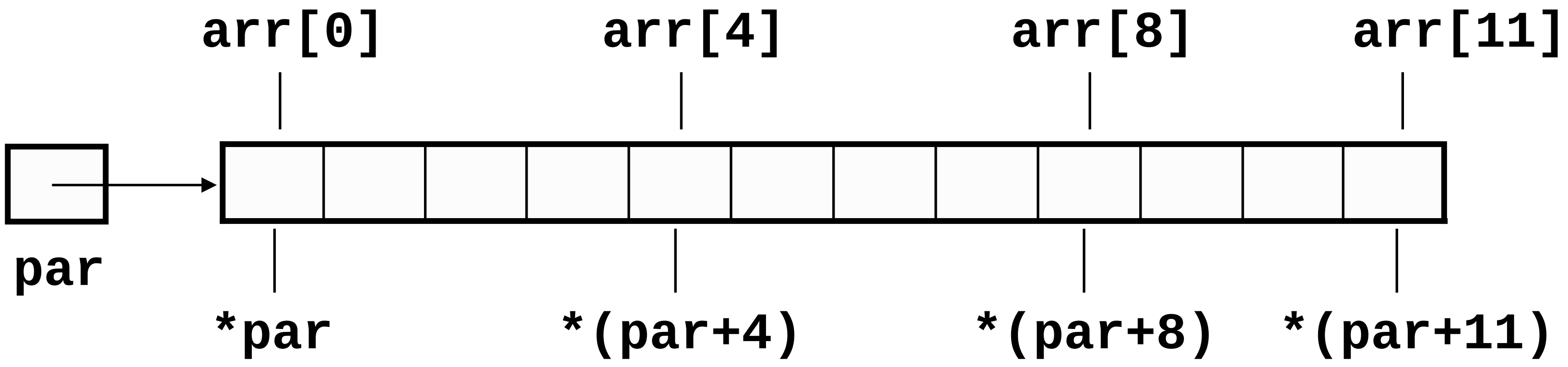
- Pointers and arrays are almost interchangeable
  - Anything you can do with arrays, you can do with pointers
    - ♦ Usually faster and more efficiently
    - ♦ But, more error-prone!

- There is one big difference:
  - You cannot modify the value of the array name
    - ♦ Think of it as a number, not a variable

Pointers and arrays

```
int arr[12];

int *par = (int *)malloc(12*sizeof(int));
```



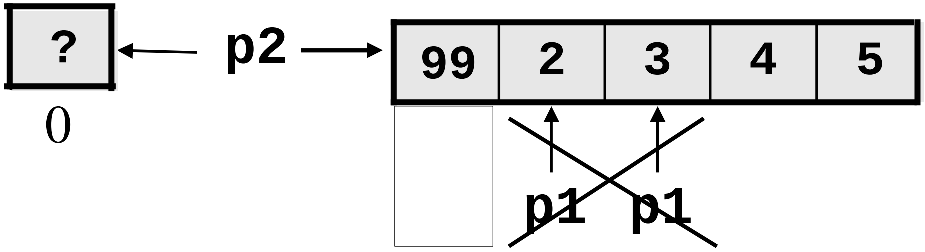
`par` is a variable – held somewhere in memory
`arr` is not – it’s just a label that the compiler understands

Example …

What is the value of **`x`** ?   **`x`** is 99

```
int x, p1[] = { 1, 2, 3, 4, 5 };
int *p2 = 0;

p2 = p1;
*p1 = 99;
p1++; p1++; // not allowed
x = *p2;
```



Pointers and arrays (cont.)

- Otherwise, array names can be treated just like pointers
  - And vice versa

- In C, arrays cannot be passed to, and returned from, functions as native types
  - You must pass/return pointers instead

Legal and fine:

```
int arr[] = {1, 2, 3};
int *pa = arr;

*arr = 100;
*(arr+1) = 200;
pa[2] = 300;
```

Because of operator precedence, this is the same as

```
pa = &(arr[0]);
```

Same thing:

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

```
arr[i];
*(arr+i);
```

```
a+i
&a[i]
```

```
void f(char s[])
void f(char *s)
```