Computer Systems

CS107

Cynthia Lee

Today's Topics

- More pointers and arrays
  - Review from CS106B/X, but digging deeper
- More strings
- Quickly for assign1:
  - files in C
  - error() function

NEXT LECTURE:

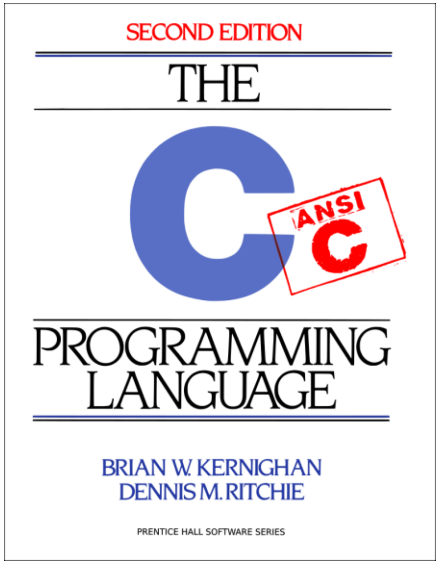
- Dynamic memory with malloc and free
- Memory layout and segments

- **For today (optional):**
  - You may want to connect to myth and follow along:
  - `cp -r /afs/ir/class/cs107/samples/lect3/ ~`

Pointers!

CULTURE FACT: IN CODE, IT'S NOT CONSIDERED RUDE TO POINT.

Stanford University



SECOND EDITION

THE C PROGRAMMING LANGUAGE

ANSI C

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

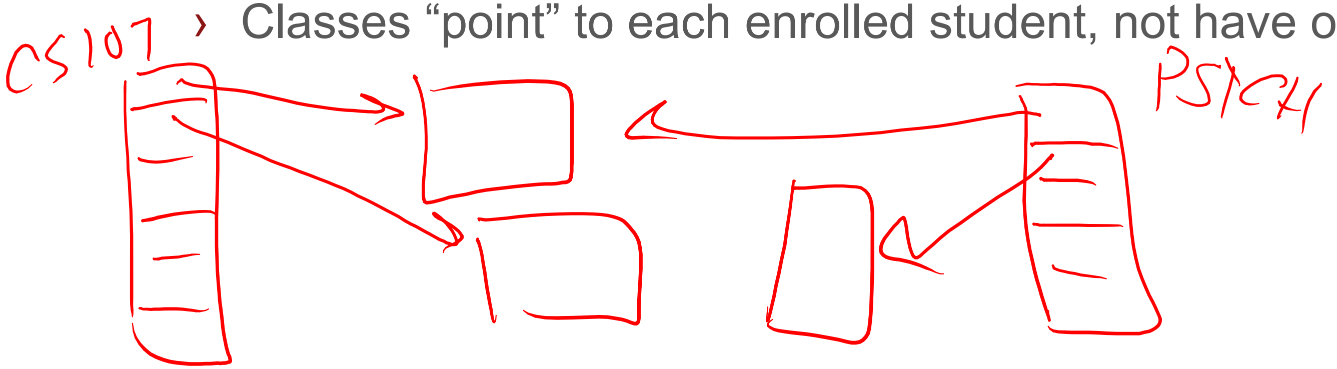
PRENTICE HALL SOFTWARE SERIES

Pointers in C

- Pointers are fundamental to almost everything in C
  - We’ll spend the quarter understanding why
  - Partly has to do with memory addresses being so fundamental to hardware and assembly, and C being a very thin layer between you and hardware/assembly

Why do we need pointers? What are they good for?

- Pointers shine in times when you want to share data structures between different components of your code
  - Imagine looking inside a program for managing student and class information at Stanford
    - *But not Axess because //shudder//*
  - Each student has a bulky record including vital stats, history, etc.
  - Classes "point" to each enrolled student, not have own bulky copy



Stanford University

Why do we need pointers? What are they good for?

- Pointers shine in times when you want to have flexibility for on-the-fly changes to the data you're storing
  - Grow and shrink collection size dynamically
  - Insert and remove elements into an ordered structure without needing a ton of reshuffling (think link list or tree structures)

- Pointers shine in times when you want to have data lifespan not so closely tied to function call/return
  - Allocate something in a function, but keep it when the function returns

(this last reason will make more sense when we talk more about stack vs heap next time)

Pointers and memory addresses

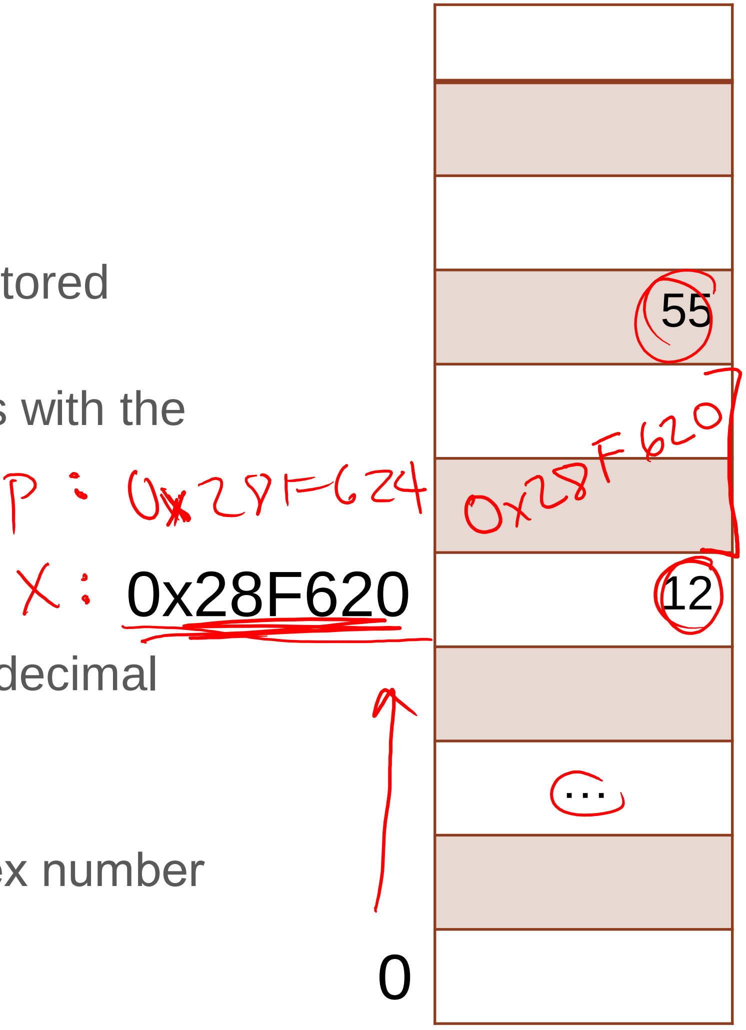
SOME EXAMPLES

Memory addresses: **106B/X review**

- When you declare a variable, it is necessarily stored *somewhere* in memory
- You can ask for any variable's memory address with the **& operator ("address-of")**

```
int x = 12;
int *p = &x;
```

- Memory addresses are usually written as hexadecimal (base-16 or "hex") numbers
  - Ex: 0x28F620
    - Prefix "0x" is a visual cue that this is a hex number (not really part of the number)



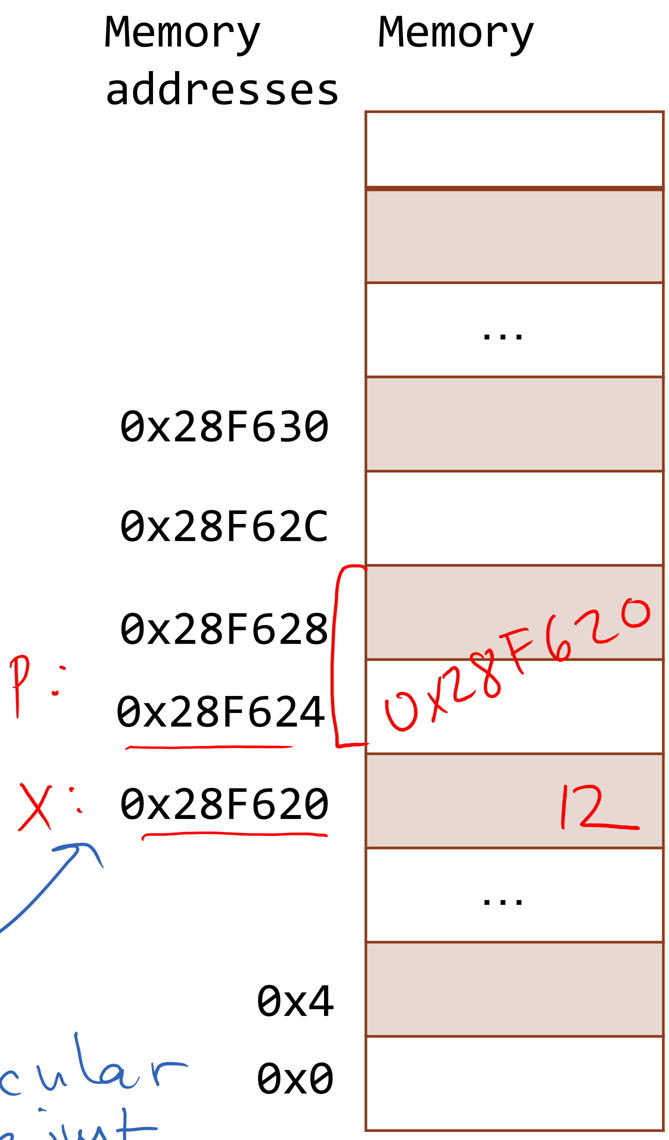
Memory addresses: some examples

```
int main(int argc, char *argv[]) {
    int x = 12;
    int *p = &x;
    printf("%d\n", x);
    printf("%p\n", p);
```

12

0x28F620

- What do the last two lines do?
  - *(answer separately for each line)*
    - Compiler warning or error
    - Runtime error or crash
    - Prints 12 twice
    - [✓ Prints 12 and something else]
    - Prints something else



	Memory addresses	Memory
		…
	0x28F630	
	0x28F62C	
	0x28F628	0x28F620
p:	0x28F624	
x:	0x28F620	12
		…
	0x4	
	0x0	

these particular addresses are just an example

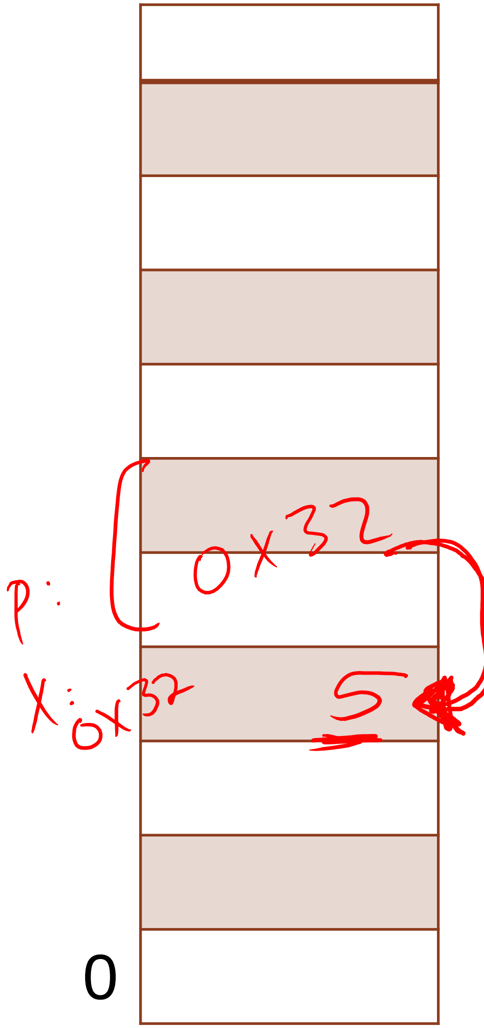
Dereference operator *

- Note this is a different use of * from the type declaration

```
int x = 5; ✓
int *p = &x;        // p has type "pointer to int"
printf("%d\n", *p); // dereferencing p   5
```

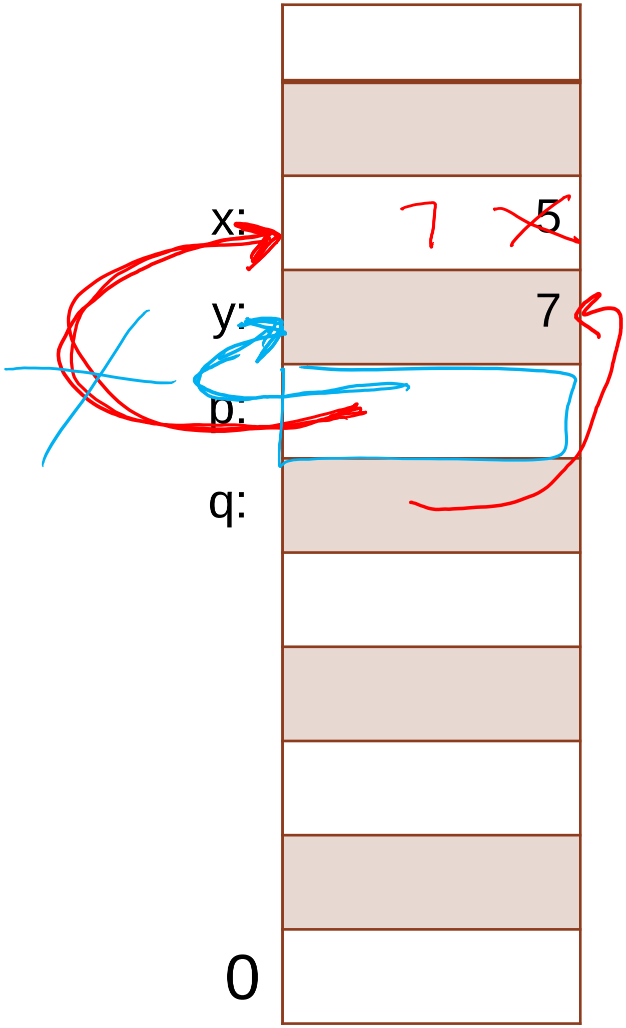
- * ONLY works on variables that are pointer types
  - › Unlike address-of &, which works on any variable
- “Follows” the pointer to the destination for a read or write of that value
  - › *(in this class we call that value the “pointee,” but that’s not an official term)*

- ***& - takes data and asks for its address*** ✓
- **** - takes an address and asks for its data*** ✓



Memory addresses: some examples

```
int main(int argc, char *argv[]) {
    int x = 5;
    int y = 7;
    int *p = &x
    int *q = &y;
```



- Do these lines of code do the same thing?

```
*p = y;
p = &y;
```

A. YES!

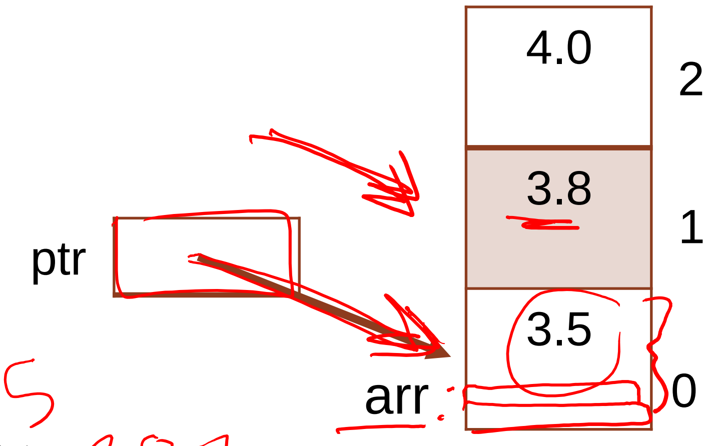
[✓ B. NO!]

C. I don't know!!

Arrays, Pointers, Strings, and Pointer Arithmetic

We can add one to a pointer to access the next element in the array

```
int main(int argc, char *argv[]) {
    double arr[3];
    double *ptr = arr;
    ptr[0] = 3.5;   arr[0] = 3.5;
    ptr[1] = 3.8;
    ptr[2] = 4.0;
    printf("%p => %g\n", ptr, *ptr); 3.5
    printf("%p => %g\n", ptr+1, *(ptr+1)); 3.8
    printf("%p => %g\n", ptr+1, ptr[1]);
```



- **Important note:** the last two lines are *completely equivalent.* C invented the array[index] notation as a shorthand version of *(array + index) notation, because it is so common to want to do that and the latter is clunky. ✓

Stanford University

Code example: print_args_ending_in_end

Prints all of its arguments that end in the suffix –“end"