Slide 1

Administrivia

- Reminder: Homework 4 due today.
- Homework 2 grades mailed Friday.
- Homework 5 to be on the Web by end of the day. Due in two weeks (because it will likely be more difficult).

Slide 2

Minute Essay From Last Lecture

- I asked about I/O redirection. Many people's responses were about general use of files. Interesting too. Or see "answer" slide for some ways I find it useful.

Slide 3

Dynamic Memory and C

- With the C89 standard, you had to decide when you compiled the program how big to make things, particularly arrays — a significant limitation.
- Variable-length arrays in C99 standard help with that, but don't solve all related problems:

  In many implementations, space is obtained for them on "the stack", an area of memory that's limited in size.

  You can return a pointer from a function, *but* not to one of the function's local variables (because these local variables cease to exist when you return from the function).

Slide 4

Dynamic Memory and C, Continued

- "Dynamic allocation" of memory gets around these limitations — allows us to request memory of whatever size we want (well, up to limitations on total memory the program can use) and have it stick around until we give it back to the system.

  (The trick here is that most implementations differentiate between two areas of memory, a "stack" used for local variables, and a "heap" used for dynamic memory allocation. Usually the former is more limited in size.)
- To request memory, use `malloc`. To return it to the system, use `free`. (For short simple programs you can skip this, but not good practice, since in "real" programs you may eventually run out of memory.)
- Python and Scala hide most of this from you — allocating space for objects is automatic/hidden, and space is reclaimed by automatic garbage collection.

Slide 5

Dynamic Memory and C, Continued

- Simple examples:

```
int * nums = malloc(sizeof(int) * 100);
char * some_text = malloc(sizeof(char) *
20);
free(nums);
```

though it's better style/practice to write

```
int * nums = malloc(sizeof(*nums) * 100);
char * some_text = malloc(sizeof(*some_text)
* 20);
free(nums);
```

- Some books/resources recommend "casting" value returned by `malloc`. Other references recommend the opposite! But you should check the value — if `NULL`, system was not able to get that much memory.

Slide 6

- (Example — improved sort program.)

Slide 7

Function Pointers

- You know from more-abstract languages that there are situations in which it's useful to have method parameters that are essentially code. Some languages make that easy (functions are "first-class objects") and others don't, but almost all of them provide some way to do it, since it's so useful — e.g., providing a "less-than" function for a generic sort.
- In C, you do this by explicitly passing a pointer to the function.

Slide 8

Function Pointers in C

- The type of a function pointer includes information about the number and types of parameters, plus the return type.
- Example — last parameter to library function `qsort` (in its `man` page). Call this by providing, in your code, a function with declaration

```
int my_compare(const char *, const char *);
```

  and using `my_compare` as the last parameter to `qsort`.
- (Example — improved sort program.)

Slide 9

Minute Essay

- Many current high-level languages manage memory for you, including garbage collection. What advantages do you think this has? What disadvantages? (Both as compared to doing it yourself, as you do in C.)
- Anything noteworthy about Homework 4?

Slide 10

Minute Essay Answer

- Advantages: easier, less error-prone.
- Disadvantages: less control, possibly unpredictable performance (which in some contexts matters).