Slide 1

Administrivia

- Reminder: Homework 6 due next week.

Slide 2

Minute Essay From Last Lecture

- No clear consensus; some people compiled using `gcc` with no flags; others made one of the other choices.
- Some said they started with just `gcc` and then added `-Wall` if they ran into problems; others that they liked the extra warnings all the time. Be advised that it's a quirk of C (or `gcc`?) that sometimes warnings indicate problems you'd only observe if you compile with optimization (`-O`).
- A few people commented that it was nice not to have all programs compile to `a.out`.
- If you're interested in using `make` but don't know how, it's in the slides for 9/05.

Slide 3

Homework 4 Essays

- No clear consensus. Many people found one or both problems difficult. The second problem seemed to give the most trouble. I keep trying to explain the problem in a way that everyone finds clear, but without success (yet?).
- One person commented about it seeming odd to need `-lm` to compile (and link!) a program using `sqrt()`. Why is that . . . Next slide.
- One person commented on how it seems strange to talk about generating "random" numbers with a deterministic system. Agreed!

Slide 4

Compiling/Linking — Review/Clarification

- When you type `gcc hello.c`, `gcc` is actually doing two steps:
- In the first step (compiling), it converts your code into "object code" (binary machine instructions plus some extra info about, e.g., function names). At this point it doesn't need access to code for library functions; it just needs to know their names and signatures, typically via a `.h` file.
- In the second step (linking), it combines your code with library code. At this point it needs the actual library code (as object code). For some reason, probably historical, most of the standard C library functions are in a place where the linker looks be default. The math functions aren't, and `-lm` tells it where to look.

Slide 5

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, added 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 6

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 is allowed) and have it stick around until we give it back to the system.

  (Most implementations allocate this memory "on the heap", which is (usually?) limited only by how much total memory the program is allowed to use.)
- 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. Makes for easier programming but possibly-unpredictable performance.

Slide 7

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 8

- (Example — "improved" sort program.)

Slide 9

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 10

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 void *, const void *);
```

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

Slide 11

More `vim` Tips

- To edit multiple files at once, `vim` followed by their names. `:next` takes you to the next file, `:rew` back to the first one. `:q` exits only from the current file; `:qall` to exit from all.

  Or use "split the screen" (`:split`) to show two files (or two parts of the same file) at once; control-W twice switches between them. `:split` followed by filename splits the screen and puts the other file in the new "window".
- You (probably? maybe?) know about `diff` to compare contents of two files. What you might not know about is `vimdiff`, which shows files side by side (or one above the other with `-o`) using colors to highlight differences.
- If you don't like the colors, there are options: Type `:colorscheme` and a space and press "tab" repeatedly to cycle through choices, enter to try one. If you find one you like, put command in `.vimrc` file.

Slide 12

Minute Essay

- Questions? otherwise just "sign in".