Slide 1

Administrivia

- For Homework 6, as of my writing this only two people had turned in complete versions. Some of the others said they'd send improved versions soon; others didn't. I just sent e-mail to the latter group asking about intentions. I don't mind waiting for improved versions; I just want to know whether to grade what you turned in or wait.
- Reminder: Homework 7 due Friday.
- Reminder: Quiz 4 Friday. Likely topic is arrays.

Slide 2

Homework 7

- (Review problems / answer questions.)

Slide 3

Pointers — Recap/Review

- Pointers "point to" data of a particular type. Declare with type and `*`.
- Related operators include `&` ("address of") and `*` (dereference — find what pointer points to).
- Useful if we need for a function to modify one of its parameters. Many other uses too, which may become apparent as we continue.

Slide 4

Character Strings in C — Preview

- We'll talk more about text data soon, but for now a summary version:
- Text strings are represented as arrays of characters. Can vary in length; end of string indicated by a special character.
- Text in double quotes (e.g., in a call to `printf`) defines a string constant — so somewhere in memory there is an array of those characters.

Slide 5

Pointers and Arrays in C

- C treats pointers and arrays as interchangeable in most respects. (This is why it works that many functions whose parameters are supposed to be strings — arrays of characters — declare them as pointers. Look again at `man` page for `printf`, e.g.)
- About the only difference is behavior of `sizeof` operator — for locally-declared array you get size *in bytes*, for array parameter or pointer you get pointer size.

Slide 6

Pointer Arithmetic in C

- C also permits doing some arithmetic operations on pointers, though only the ones that are "sensible".
- Adding an integer n to a pointer that points to *type* advances it n times the size of *type*. Subtracting an integer from a pointer works similarly. (Strictly speaking, though, you should only do this within an array.)
- Subtracting one pointer from another gives an integer result. (This can be particularly useful in working with strings.)
- Comparing pointers with relational operators works, though strictly speaking you should probably only use less-than and greater-than operators on pointers into the same array.

Slide 7

Pointer Arithmetic in C, Continued

- Example: If `a` is an array of `int`s, `a[2]` and `*(a+2)` are equivalent.
- So we could write loops over arrays using pointers. Once upon a time that was sometimes more efficient. With current compilers, probably not so, so use whatever is most readable.
- (Example.)

Slide 8

Dynamic Memory and C

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

  In most 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 9

Dynamic Memory and C

- "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.

  (How this helps — 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.)

- Dynamic memory allocation also needed to build "ragged" arrays (arrays in which rows are of different sizes) and "linked" data structures (later).

Slide 10

Dynamic Memory and C, Continued

- To request memory, use `malloc`.
- To return it to the system, use `free`. (For short simple programs you can probably get away with skipping `free` since the operating system will probably clean up after you, but for longer and more complicated programs, you should clean up when you can, or eventually you may run out of memory.)

Slide 11

Dynamic Memory and C, Continued

- Examples:

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

  or better:

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

  and then

```
free(nums);
free(some_text);
```

- Book recommends "casting" value returned by `malloc`. Other references recommend the opposite! But you should check the value — if `NULL`, system

Slide 12

was not able to get that much memory.

- Example — program to generate and sort "random" numbers, two ways.

Slide 13

Minute Essay

- Many people seem to be falling behind with the homework. If that's you, can you say what's going wrong? Are you not getting help, is the help not enough, . . . ?