Slide 1

Administrivia

- Reminder: Homework 4 due today.
- Quiz 3 moved to next Wednesday.
- Quiz 2 solution online. Sorry about the mixup with the name of the function in the second problem (`barfoo` in the code, `foo` in the text).
- When I ask you in a quiz or exam question what a function seems intended to accomplish, I'm looking not for a description in English of what the code is doing but what the purpose seems to be, as in the minute essay question from Friday and the extra-credit quiz question. Clearer?

Slide 2

More Administrivia

- I ask you to turn in programs by e-mail. Relatively easy if logged in from console of a classroom/lab machine. If not?
- Another way is the "mail files" script on the "sample programs" page.

Slide 3

Programming Tip

- If you're testing multiple conditions, only one of which is meant to be true, probably best to do so with a chain of

  `if .... else if .... else if .... else`

  rather than a lot of `if`s.
- In such a chain, notice what you already know when you get to an `else`. Example — I say there's something redundant in the following code:

```
if (a < b) { .... }
else if (a >= b) { .... }
```

  (Spot it?)

Slide 4

Repetition Via Loops

- Recursion provides one way to repeat something. Often not efficient (every call to a function requires space for local variables, and at some point you can run out of room), nor is it always convenient (writing a function every time you want to repeat something).
- Hence C, like most procedural languages, offers constructs called *loops*. All have four basic elements (sometimes implicit).

Slide 5

Loop Elements

- Initializer — something that sets initial values for variables involved in the repetition (iteration).
- Condition — something that determines whether repetition continues. Can be tested at the start of each iteration (*pre-test* loop) or at the end (*post-test* loop).
- Body — the code to repeat.
- Iterator — something that moves on to the next iteration.

Slide 6

`while` Loops

- Probably the simplest kind of loop. You decide where to put initializer and iterator. Test happens at start of each iteration.
- Example — print numbers from 1 to 10:

```
int n = 1;                /* initializer */
while (n <= 10) {         /* condition */
    printf("%d\n", n);    /* body */
    n = n + 1;            /* iterator */
}
```

- Various short ways to write `n = n + 1`:

```
n += 1;
n++;
++n;
```

What do you think happens if we leave out this line?

Slide 7

`for` Loops

- Probably the most common type of loop. Particularly useful for anything involving counting, but can be more general. Syntax has explicit places for initializer, condition, iterator (so it's less likely you'll forget one of them).
- Example — print numbers from 1 to 10:

```
int n;
for (n = 1; n <= 10; ++n) {
    printf("%d\n", n);
}
```

- Initializer happens once (at start); condition is evaluated at the start of each iteration; iterator is executed at the end of each iteration.

Slide 8

`do while` Loops

- Very similar to `while` loop, except that test happens at end of each iteration.
- Example — print numbers from 1 to 10:

```
int n = 1;                          /* initializer */
do {
    printf("%d\n", n);              /* body */
    n = n + 1;                      /* iterator */
} while (n <= 10);                  /* condition */
```

Slide 9

Loops — Simple Examples

- We could do loop versions of the factorial and Fibonacci programs, as examples of using `for` loops.
- We could do a loop version of the program to sum integers from stdin, as an example of using a `while` loop.
- Notice that we could even have a loop within a loop ("nested loops"). Silly example — printing a rectangle of `x`'s.

Slide 10

Minute Essay

- Anything noteworthy about Homework 4?
- Any thoughts about interesting or useful problems that would involve repetition (via loops or recursion)?