Slide 1

Administrivia

- Reminder: Homework 4 due Thursday.
- Reminder: Quiz 3 Thursday. Likely topic is functions.

Slide 2

Functions and Recursion

- As mentioned last time: Something else we want to be able to do is repeat something some fixed number of times, or until some condition is true — for example, in the converter program, prompt again if we get invalid input.
- Chapter 6 introduces some new constructs for this — our next topic — but we can also do it with tools we had before, using *recursion* — having a function call itself.

  Obviously to make this work we need a way to stop recursing — a *base case* — otherwise we have something akin to the in-joke definition of GNU ("GNU is Not Unix").

  Also we need to be sure that every recursive call brings us closer to a base case.

Slide 3

Recursion, Continued

- How it works: When you call any function, the current "state" (values of variables) is preserved ("pushed onto a stack"), and space is reserved for the called function's local variables (including parameters). When the function returns, this space is freed up again. So if we stack up recursive calls to the same function, each has its own copy of all local variables.
- Simple examples — Fibonacci numbers, counting.

Slide 4

Repetition

- So we have one way to repeat something. But it's 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

- Looks very similar to `while` loop, but 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

Loop Examples

- Examples as time permits . . .

Slide 10

Minute Essay

- Write a `for` loop that prints the numbers 1 through 10 in reverse order.

Slide 11

Minute Essay Answer

- One way:

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