Slide 1

Administrivia

- Reminder: Homework 3 due today.

  If you can't finish completely by the due date/time, but you have something that represents at least a good start, *send me what you have* and submit a revised/improved version as soon as you can. You lose fewer points that way, I think you learn more, and I'd rather grade code that works than code that doesn't!

- Grades for Homework 2 mailed earlier today.. Sample solution to Homework 2 posted (also Homework 1, though it's kind of trivial), linked from bottom of "lecture topics and assignments" page.

- Homework 4 on the Web; due in two weeks. Both problems should be doable with material covered through today, but I plan to talk more about library function `rand()`, used in the second problem, next time.

Slide 2

Minute Essay From Last Lecture

- Nothing really stood out in comments about the homework, not like some semesters where many people mentioned forgetting tsemicolons.

- `scanf` doesn't seem to flag inputs thatt are too big. Right. If you think about what it's doing under the hood, and know that detecting integer-arithmetic overflow in C is basically not feasible, it may make sense?

- `#define` was interesting or unfamiliar. Last semester someone mentioned that syntax is so different that it must be different in some basic way? Yes — "preprocessor directive" as opposed to regular code.

Slide 3

Simple I/O, Revisited

- Doing a really good job with interactive input is surprisingly tricky — what constitutes an error, how do you prompt user to try again.
- So for this class we'll focus on some simple safety checks: if input should be numeric it is, values make sense for the program (e.g., inputs to GCD program are not both 0). I like to always print input values so users can at least confirm that what they thought that typed in is what the program read.
- Some online sources discourage use of `scanf`. There are reasons for getting input other ways, but I say they have their problems too. It *is* annoying that it doesn't detect overflow, but oh well.
- For this class it's usually best to just bail out on bad input, rather than retrying.

Slide 4

Repetition — Loops

- C, like most/many procedural languages, offers several syntaxes for repetition. Recursion (discussed already) is one, but often not the most straightforward.
- All have some way of expressing common elements (explicitly, rather than the "do for all" syntax allowed by some languages):
  - *Initializer* (as its name suggests).
  - *Condition* (determines whether repetition continues).
  - *Body* (code to repeat).
  - *Iterator* (something that moves on to next iteration).
- Worth noting that C, being fairly minimalist, doesn't offer some of the nice features for repetition Scala does.

Slide 5

`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 6

`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:

```
for (int 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. (Note that C89 standard required that `n` be declared outside the loop.)

Slide 7

`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 8

Loops — Example

- Simple example — loop to read integers and compute their sum. (Don't we need a place to store them all? No!)
- (Variant of example in book.)

Slide 9

Arrays — from CS1

- Previously we've talked about how to reserve space for a single number/character and give it a name.
- Arrays extend that by allowing you to reserve space for many numbers/characters and give a common name to all. You can then reference an individual element via its *index* (similar to subscripts in math).

Slide 10

Arrays in C

- Declaring an array — give its type, name, and how many elements. Examples:

```
int nums[10];
double stuff[N];
```

(The second example assumes `N` is declared and given a value previously. In C89, it had to be a constant. In C99, it can be a variable — "variable-length array".)

- Alternatively, give "initializer" (list of values) and let compiler figure out size:

```
int nums[] = { 2, 4, 6, 8 };
```

Slide 11

Arrays in C, Continued

- Arrays whose size isn't known at runtime — in C89, only with dynamic memory allocation (to be discussed later).

  C99 also allows "variable-length arrays" — arrays declared as usual but with dimensions specified at runtime.

- These are nice for arrays of reasonable size but not so great for large arrays, as we'll discuss later.

Slide 12

Arrays in C, Continued

- Referencing an array element — give the array name and an index (ranging from 0 to array size minus 1). Index can be a constant or a variable. Then use as you would any other variable. Examples:

```
nums[0] = 20;
printf("%d\n", nums[0]);
```

  (Notice that the second example passes an array element to a function. AOK!)

- So far nothing new, just different syntax. But . . .

Slide 13

Arrays in C, Continued

- C's support for arrays is — no surprise? — minimalist, sort of a thin veneer over the implementation (in which you get a contiguous chunk of memory and a name you can use to reference it).
- One aspect — they're not "first-class objects" and don't "know" their length (!).
- Also . . . We said if you declare an array to be of size n you can reference elements with indices 0 through $n - 1$. What happens if you reference element -1? n? $2n$?
- Well, the compiler won't complain. At runtime, the computer will happily compute a memory address based on the starting point of the array and the index. If the index is "in range", all is well. If it's not (i.e., it's "out of bounds) . . .

Slide 14

Arrays in C, Continued

- (What happens if you try to access an array with an index that's out of bounds?)
- "Results are unpredictable" ("undefined behavior" in C-speak). Maybe it's outside the memory your program can access, in which case you may get the infamous "Segmentation fault" error message (or with newer compilers you may get a screenful of equally cryptic messages).

  Almost worse is if it's not — then what's at the computed memory address might be some other variable in your program, which will then be accessed/changed. This is the essence of the *buffer overflows* you hear mentioned in connection with security problems.
- Why this behavior? Well, C was designed to compile to efficient code, and checking indices "costs". If you want it, put it in! (And very often you should.)

Slide 15

Arrays — Examples

- (First a very silly example showing what happens when you reference an out-of-bounds index.)
- A familiar(?) thing to do with a collection of data — sort it. We could sketch a a program to sort an array . . . Next time?

Slide 16

Minute Essay

- What did you find interesting, difficult, or otherwise noteworthy about Homework 3?