Slide 1

Administrivia

- Reminder: Homework 1 due today (11:59pm). Meant to be straightforward practice involving only assignments, conditionals, and simple I/O.
- Homework 2 on the Web. Due next week. Meant to also be fairly straightforward practice using loops and arrays.
- (Review minute essay from last time.)

Slide 2

Review(?): Input/Output Redirection

- Normally programs run from the command line write output to the terminal window. Can instead "redirect" output to a file:

  $>$ `outfile` (overwrite)

  $>>$ `outfile` (append)

- Normally programs get input from the keyboard, but can also make them get input from a file with $<$.

  (How could this help you in checking your programs?)

- Finally, can use "pipes" (vertical-bar |) to have output from one program become input to another. Example:

  `ruptime | grep xena` (show status of HAS 340 machines)

  Very powerful idea! this and some other ways of connecting simple programs makes for a very powerful and flexible environment.

Slide 3

Conditional Execution — One More Thing

- Last time we looked at if/else syntax.
- One other conditional-execution construct you may encounter — `switch`. Basically a short form of if/elseif/else. Somewhat like `match` in Scala but nowhere near as powerful. Example:

```
char c; /* code to set value omitted */
switch (c) {
    case 'a': printf("first case\n"); break;
    case 'b': printf("second case\n"); break;
    default:  printf("default\n");
}
```

Slide 4

Repetition — Loops

- C, like most/many procedural languages, offers several syntaxes for repetition. (One of them, recursion, we'll talk about later.)
- 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).

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

- 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.)

- 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!)

Slide 11

Arrays in C, Continued

- 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 12

Arrays in C, Continued

- (What happens if you try to access an array with an index that's out of bounds?)
- "Results are unpredictable." 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.
- What to do? *Be careful.* (Probably worth noting here that more-recent languages are apt to check for such errors.)

Slide 13

Arrays — Example

- Back story: Conventional wisdom says that many library functions for generating sequences of random numbers aren't very random in their least-significant bits, so mapping their output to a small range using the mod operator isn't a good idea.
- We could write a short program to check, in a crude way, whether that's true, or at least how well the results are distributed over the range: Prompt for how many "random" numbers to generate and for a divisor, then generate the sequence, divide each by the divisor, and count how many have remainder 0, remainder 1, etc.

Slide 14

Minute Essay

- What did you find interesting about Homework 1? What did you find difficult? (Optional question — okay to just sign in.)