Administrivia

Slide 1

- Grades for Homework 2 mailed.

- 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!

- 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.

- Sample solutions for Homeworks 1 and 2 posted (from the end of "Lecture topics and assignments"). Also a collection of `vim` tips (from "Useful links").

Minute Essay From Last Lecture

Slide 2

- Nothing really stood out in comments about the homework, but:

- Several mentioned difficulty adapting to a new syntax, especially trying to switch back and forth between Scala and C. Agreed, though practice helps some.

- Most people found the concepts simple, though a few had to think about the logic.

Slide 3

Minute Essay From Last Lecture, Continued

- Several people mentioned wanting to use conditionals and/or loops but not knowing how yet.

- One person mentioned difficulties caused by including a newline ($\backslash$n) in the format string for scanf. I've never been quite sure myself exactly what this does, but the effect is not good.

- One person mentioned not initially understanding that code has to go into a function (unlike in a Scala script).

- One person mentioned source files being turned into what looked like random garbage on every compilation. Not sure if this is the problem, but

  gcc -o inputfile inputfile

  would do that!

Slide 4

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, and 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. (And if you do anything else on homework, it breaks my semi-automated testing.)

Slide 5

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 6

while Loops

- Probably the simplest and most general 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?

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.)

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 */
```

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

Slide 11

- 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 };
  ```

Arrays in C, Continued

Slide 12

- 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" (VLAs) — 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.

- Note also that while C++ is almost a strict superset of C, VLAs are one C99
  feature that isn't in C++. But there are good alternatives in the standard C++
  library.

Slide 13

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 14

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") . . .

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.)

Arrays — Examples

- (Very silly example illustrating defining and using an array, including what happens when you reference an out-of-bounds index.)

Slide 17

Minute Essay

- If your solution to Homework 2 made use of constants such as 3600 for seconds per hour, did you get them from computing them (60 times 60) or from your favorite search engine?

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