Administrivia

Slide 1

- Reminder: First quiz Monday.

  As noted previously, you'll have access to the textbook, your notes (paper or electronic), and the course Web site, but the only allowed computer use is to access these (so no typing in code and trying it). Intended to take no more than 10 minutes.

  Likely questions include "what does this C program print?", "write some C code to do the following", and questions about the material on binary numbers.

Minute Essay From Last Lecture

Slide 2

- Pretty much everyone got it right (yay!), except for one person who I think forgot what `%` means in C.

Slide 3

Homework 2 Essays

- A few people commented that it was good to start writing code — interesting, helpful, even fun.
- The comment that stood out most was about how to divide 5 by 9 and get the desired result. (That was a key take-away message for the first problem.)
- A few people mentioned that they were able to adapt examples from class. Often that's my intent — that you use them as starting points or guides.

Slide 4

Functions and Problem Decomposition

- So far all our programs have been one big chunk of code. This is okay for simple programs, but quickly becomes difficult to understand as problems get bigger.
- Further, some things we don't want to, or really can't, write ourselves, such as the code for input/output.
- So C, like many/most other programming languages, gives you a way of decomposing problems into subproblems. C calls them *functions*.

  C functions are similar to functions in math, except that they can have side effects (similar to how evaluation of expressions can have side effects).

  Using this feature to good effect is something of an art, but experience helps.

Slide 5

Functions in C, Continued

- Every function has
  - A name (where rules for names are the same as those for variables).
  - Zero or more inputs (called *parameters*).
  - A return type (`void` to indicate that the function doesn't return anything).
  - Some code to be executed when the function is called.
- When you call (use) a function, you
  - Supply values for inputs (pass in values for parameters).
  - Optionally, use the value returned by the function. The function call is an expression, and its value is the value returned by the function.

Slide 6

Defining and Using Functions

- Simple example of defining and using a function to add two integers:

```
int add(int a, int b) {
    return a + b;
}
int main(void) {
    int result = add(1, 2);
    printf("%d\n", result);
    return 0;
}
```

- `add` has two parameters called `a` and `b`, which are basically variables local to the function. When we call `add` from `main`, values 1 and 2 are copied into these variables. The code in `add` executes until it reaches a `return`. At that point, we go back to the calling function, and the value of the function call is whatever is after the keyword `return`.

Slide 7

Functions in C — Declaration Versus Definition

- Many languages let you put function definitions in any order you want, and even split them up among files.
- But some of this requires the compiler to be somewhat smarter than C compilers are required to be. In C, functions must either be defined or *declared* before being used.
- Function declarations give function name, number and types of parameters, and return type. Syntax is just like that for function definitions, except no parameter names needed, and body is replaced with a semicolon.

Slide 8

Functions in C — Declaration Versus Definition, Continued

- For your own functions, you can either define them before using them, or define them in whatever order you like and put declarations at the top.
- For library functions? declarations are part of what's supplied by `#include` directives. ("Aha, so that's what that is"?)

Slide 9

The `main` Function, Revisited

- Every C program you/we have written so far includes a definition of a function called `main`. All complete C programs must have such a function.
- `main` is defined in your code:
  - It has no parameters. (Actually, it can — there's an alternative definition that allows it to accept command-line arguments, similar to the ones that follow commands such as `gcc`, `ls`, etc. Later!)
  - It returns an integer value.

Slide 10

The `main` Function, Continued

- `main` is called by some type of environment (the command shell for us, when you type `./a.out` after compiling). It gives your code the optional parameters (more about this later) and receives the value you return. Return value can be used to indicate success/failure (useful for shells that themselves support conditional execution).
- Almost all of this program, and other examples, should now more or less make sense! (Exceptions are representation of character strings, `&` syntax for parameters. Soon!)

Slide 11

C Library Functions

- Standard C comes with a number of *library functions* to do things many programs want to do.
- Examples we've seen so far: `scanf`, `printf`.
- UNIX/Linux systems normally have `man` pages for these functions, describing parameters and return values in full detail (hence, not always easy reading).

  (Tip: `man printf` gives the `man` page for a command rather than the C function. Use `man 3 printf` to get what we want.)

  (Tip: When reading a `man` page, `h` will bring up a summary of what keys do what — page up/down, search, quit, etc.)

Slide 12

Defining and Using Functions — Example

- As a somewhat contrived example, we could rearrange the "solve a quadratic equation" example from previous class.
- By putting the code to solve the equation and print results in a function, we can also easily have it print some examples/tests. Maybe do this before prompting for input?

Minute Essay

- Any questions?