Slide 1

Administrivia

- Reminder: Homework 3 due today at 5pm.
- Reminder: Midterm a week from today. There will be a short review sheet on the Web soon, and we can spend part of Thursday reviewing.

Slide 2

Conditional Execution, Continued

- Last time we looked at examples of conditional execution, with at most two cases. What if more than two? We could "nest" `if`/`else` constructs, e.g.,

```
if (x < 0) {
    printf("less than\n");
}
else {
    if (x > 0) {
        printf("greater than\n");
    }
    else {
        printf("equal\n");
    }
}
```

- But this gets ugly fairly quickly. So . . .

Slide 3

Conditional Execution, Continued

- Better:

```
if (x < 0) {
    printf("less than\n");
}
else if (x > 0) {
    printf("greater than\n");
}
else {
    printf("equal\n");
}
```

- Can have as many cases as we need; can omit `else` if not needed.

Slide 4

Conditional Execution, Continued

- Sometimes we can go further, though. If all of the conditions are of the form

  integer_expression == *value*

  then we can use the `switch` construct. Notice that characters (`char`) count as integers in this context.

- Example (similar to calculator example in book) on next slide.

Slide 5

Conditional Execution, Continued

```
char menu_pick;  /* should be one of '+', '-' */
/* .... */
switch (menu_pick) {
    case '+':
        result = input1 + input2;
        break;
    case '-':
        result = input1 + input2;
        break;
    default:
        result = 0;
        printf("operator not recognized\n");
}
```

Slide 6

Conditional Expressions

- C also provides a short way to express things of the form

  `if (`*condition*`)`

  variable = *value1*

  `else`

  variable = *value2*

  namely the ternary (three operands) operator `?`.

- Example:

  ```
  sign = (x >= 0) ? 1 : -1;
  ```

  assigns 1 to `sign` if `x` is non-negative, -1 otherwise.

- (Use with caution — compact, but can easily lead to code that's difficult for humans to understand.)

Slide 7

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 can't, really 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*. Using this feature to good effect is something of an art, but may teach you something about problem decomposition in general, which is a useful skill.

Slide 8

Functions in C

- 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).
- We will talk a little now, and more next time, about how to define our own functions. Notice for now that every program you / we have written so far defines a function called `main`, and most of them use system library functions `scanf` and `printf`.

Slide 9

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, as discussed previously, and its value is the value returned by the function.

Slide 10

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 (a type of variable) called `a` and `b`. When we call `add` from `main`, the 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 11

The `main` Function

- As noted, 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.
- `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).

Slide 12

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, quit, etc.)

Slide 13

Minute Essay

- None — quiz.