**Slide 1**

## Administrivia

- Reminder: Homework 5 and Homework 6 due next week.

**Slide 2**

## User-Defined Types in C — Review

- Discussed briefly last time — `typedef`, `enum`. One more to discuss briefly today — `union`.
- Discussed in a bit more detail — `struct`.

Slide 3

## Example — Singly-Linked List

- As an example, consider code for a singly-linked list of integers, using two `struct`s, one for list nodes and one for the list itself.
- To make things more interesting, we could write all the functions to use recursion . . .

Slide 4

## User-Defined Types in C — `union`

- For completeness, we should mention that C also provides a way of defining a structure that can contain one of several alternatives (“this OR that”, as opposed to the “this AND that” of `struct`) — `union`.
- See the discussion in the book for more about this; it can be useful, but can also make code more difficult to understand.

Slide 5

## A Little More About `gcc`

- Many, many compiler options for `gcc`. One of the most useful is `-Wall`.
- To automate using them every time, you can use the UNIX utility `make` . . .

Slide 6

## A Little About `make`

- Motivation: Most programming languages allow you to compile programs in pieces (“separate compilation”). This makes sense when working on a large program — when you change something, just recompile parts that are affected.
- Idea behind `make` — have computer figure out what needs to be recompiled and issue right commands to recompile it.

Slide 7

## Makefiles

- First step in using `make` is to set up "makefile" describing how files that make up your program (source, object, executable, etc.) depend on each other and how to update the ones that are generated from others. Normally call this file `Makefile` or `makefile`.

  Simple example (assuming `main.c` `#includes` `defs.h` and `foo.h`):

```
main:   main.o foo.o
        gcc -o main main.o foo.o
main.o: main.c defs.h foo.h
        gcc -c main.c
foo.o:  foo.c
        gcc -c foo.c
```

- When you type `make`, `make` figures out (based on files' timestamps) which files need to be recreated and how to recreate them.

Slide 8

## Predefined Implicit Rules

- `make` already knows how to "make" some things — e.g., `foo` or `foo.o` from `foo.c`.
- In applying these rules, it makes use of some variables, which you can override.
- A simple but useful makefile might just contain:

```
CFLAGS = -Wall -pedantic -O
```

- Or you could use

```
CFLAGS = -Wall -pedantic $(OPT)
OPT = -O
```

  and then optionally override the `-O` by saying, e.g., `make OPT=-g foo`.

Slide 9

## Some Interesting Things You Can Do In C

- Most UNIX/Linux “system calls” (requests to operating system) have a C library function to call them. Example — `fork` to create a new process. Most of them probably have a Java equivalent, but calling them directly from C may be interesting in being lower-level.
- Some functionality available in command shells is accessible via library functions — e.g., `readline` (tab completion, command history).
- The `ncurses` library provides functions to do fancier I/O (colors, cursor positioning, etc.).
- C “bindings” for OpenMP provide simple ways to do multithreading for performance.

Slide 10

## Minute Essay

- None — sign in.