Slide 1

Administrivia

- (Lecture notes for the lecture that didn't happen.)

Slide 2

User-Defined Types

- So far we've only talked about representing very simple types — numbers, characters, text strings, arrays, and pointers. You might ask whether there are ways to represent more complex objects, such as one can do with classes in object-oriented languages.
- The answer is "yes, sort of" — C doesn't provide nearly as much syntactic help with object-oriented programming, but you can get something of the same effect. But first, some simpler user-defined types . . .

Slide 3

User-Defined Types in C — `typedef`

- `typedef` just provides a way to give a new name to an existing type, e.g.:

```
typedef charptr char *;
```

- This can make your code more readable, or allow you to isolate things that might be different on different platforms (e.g., whether to use `float` or `double` in some application) in a single place.

Slide 4

User-Defined Types in C — `enum`

- In C (and in some other programming languages) an *enumeration* or an *enumerated type* is just a way of specifying a small range of values, e.g.

```
enum basic_color { red, green, blue, yellow };
enum basic_color color = red;
```

  This can make code more readable, and sometimes combines nicely with `switch` constructs.

- Under the hood, C enumerated types are really just integers, though, and they can be ugly to work with in some ways (e.g., no nice way to do I/O with them).

Slide 5

User-Defined Types in C — `struct`

- More complex (interesting?) types can be defined with `struct`, which lets you define a new type as a collection of other types — something like a class in an object-oriented language, but with no methods and no way to hide fields/variables.
- Two versions of syntax (next slide) . . .

Slide 6

User-Defined Types in C — `struct`

- One way to define uses `typedef`:

```
typedef struct {
    int dollars;
    int cents;
} money;
money bank_balance;
```

- Another way doesn't:

```
struct money {
    int dollars;
    int cents;
};
struct money bank_balance;
```

Slide 7

User-Defined Types in C — `struct`, Continued

- Either way you define a `struct`, how you access its fields is the same:

  `.` if what you have is a `struct` itself:

```
struct money bank_balance;
bank_balance.dollars = 100;
bank_balance.cents = 100;
```

  `->` if what you have is a pointer to a `struct`:

```
struct money * bank_balance_ptr = &bank_balance;
bank_balance_ptr->dollars = 100;
bank_balance_ptr->cents = 100;
```

Slide 8

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 discussion in textbook about this; it can be useful, but can also make code more difficult to understand.

Slide 9

Example — Sorted Singly-Linked List

- Now we have enough tools to do a low-level version of something probably familiar to you — linked list. Idea is the same as in higher-level languages, but must explicitly deal with many details.
- Textbook has code for singly-listed list; example on “sample programs” takes a somewhat different approach (recursion rather than iteration, and sorted).

Slide 10

Separate Compilation — Review

- C (like many languages) lets you split large programs into multiple source-code files. Typical to put function and other declarations in files ending `.h`, function definition in files ending `.c`. Compilation process can be separated into “compile” (convert source to object code) and “link” (combine object and library code to make executable) steps.
- UNIX utility `make` can help manage compilation process. Can also be useful as a convenient way to always compile with preferred options. Idea behind `make` — have computer figure out what needs to be recompiled and issue right commands to recompile it.

Slide 11

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 12

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 13

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

Slide 14

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 -std=c99
  ```
- Or you could use

  ```
  OPT = -O
  CFLAGS = -Wall -pedantic -std=c99 $(OPT)
  ```

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

Slide 15

One More Useful Tool — `valgrind`

- `valgrind` can check for a lot of potential errors — including errors in use of `malloc` and `free`.
- Compile with `-g` and `-O0` and `valgrind` executable-name

Slide 16

Minute Essay

- Anything about C that you'd like to hear more about next time?