Slide 1

Administrivia

- Reminder/extension: Homework 5 due next week (yes, a holiday, so turn in before you leave).

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 — 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.
- We could write some code, using two types of `structs`, one for the nodes in the list and one for the list itself. (Example on "sample programs" page. Textbook also has somewhat simpler code for list — single data structure, iterative rather than recursive.)

Slide 10

Separate Compilation and Makefiles — 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.

Slide 11

Minute Essay

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