Slide 1

Administrivia

- Reminder: Homework 6 due today. If you're still struggling, turn in what you have and try to finish and turn in an improved version later; by end of the day Friday is fine.

  I say in the writeup that you can use the `sort` command to check your result, but in fact that won't always work — it does for me, but because I have some non-default options set so it just sorts in the same way `strcmp()` does. I mean for you to just sort using `strcmp())`. Sorry if I misled you!

Slide 2

Minute Essay From Last Lecture

- Some people had not seen `make` previously, though others mentioned that Dr. Fogarty is using it in Functional Languages this year.
- One person mentioned using it to install things on Linux. Yes, it's part of the traditional "install from source" that can still be useful, especially if you want to install something without being root (administrator).

Slide 3

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 4

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 5

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 6

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 7

User-Defined Types in C — `struct`

- One way to define uses `typedef`:

```
typedef struct {
    double x;
    double y;
} point2D;
point2D some_point;
```

- Another way doesn't:

```
struct point2D {
    double x;
    double y;
};
struct point2D some_point;
```

Slide 8

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 point2D some_point;
some_point.x = 10.1;
some_point.y = 20.1;
```

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

```
struct point2D * some_point_ptr = &some_point;
some_point_ptr->x = 10.1;
some_point_ptr->y = 20.1;
```

Slide 9

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 10

User-Defined Types and Library Code

- Library code often makes use of "opaque" types (e.g., `FILE`).
- Implementing this often involves separating functionality into interface (`.h` file containing type definitions, function declarations) and implementation (`.c` file containing function definitions.
- This leads into . . .

Slide 11

Separate Compilation and `make` — Review

- C (like many languages) lets you split large programs into multiple source-code files. Typical to put function declarations (headers), constants, etc., in file ending `.h`, function definitions (code) in file 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.
- `make` can help manage compilation process. (Can also be useful as a convenient way to always compile with preferred options.)

Slide 12

Example — Sorted Linked List

- As an example, consider writing code for a sorted linked list.
- (Start writing code in class.)

Slide 13

Minute Essay

- Anything noteworthy about Homework 6? did it meet my goal of helping you understand pointers better?