Slide 1

Administrivia

- (None?)

Slide 2

User-Defined Types

- So far we've only talked about representing very simple types — numbers, characters, text strings, arrays, and pointers. A lot can be done with just these types, but for a lot of applications it's really useful to be able to represent more-complex kinds of data, and in general it's useful to be able to define your own types.
- (Aside: A standard computer-science definition of "type" is "a set of values plus some operations on them". So for example C's `int` represents a set of integer values in a fixed range, with operations including arithmetic and relational operators.)

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 — `struct`

- It's also useful to have ways of representing more-complex "types".

  Simple examples include rational numbers (with integer numerator and denominator) and points in 2D or 3D space (with two or three coordinates).

  More-complex examples . . . Well, it's almost "the sky's the limit", but as one example think about what might be involved in writing a program with a GUI: It would seem to make sense to have some way of representing "windows" and "buttons" and so forth.

- "Object-oriented" languages provide a nice way to do this. C doesn't provide a really nice way, but it does provide *a* way, via `struct`s.

Slide 5

`struct`s in C

- An array in C (and in most if not all other programming languages) is a collection of data items all of the same type, and you reference individual items via indices. Arrays can have any number of elements. You define an array by giving its type and its dimensions.
- In contrast, a `struct` is a collection of data items of possibly different different types. Individual items are called “fields” and have names by which you can refer to them. For a given `struct` type, all objects of that type have the same fields. You define a `struct` by giving names and types of its fields. Two syntaxes for doing that.

Slide 6

Defining a `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 7

Accessing Fields in a `struct`

- 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 8

`struct`s, This and That

- Can initialize a `struct` by giving values for all its fields in curly braces, e.g.,

  `point2D = { 10.1, 20.2 };`

  (Observe in passing that a similar syntax works for arrays. Textbook mentions it though we haven't used it in class.)

- Can assign one `struct` to another with the usual assignment operator.
- Can pass `struct`s as parameters to functions and return them from functions, with the usual(?) pass-by-value semantics, meaning that the whole `struct` is copied. Can also pass pointers to `struct`s, and for large `struct`s that's likely more efficient.

Sidebar: Files and Program Structure

- All the programs we've written have been single files. Fine for small programs but seems like it might be unwieldy for large ones, no?
- So usually code for large programs is split into multiple files, which can be separately compiled and then "linked" together to form an executable. This also allows reuse of the same function(s) in multiple programs. We'll do examples of both, later.
- "Library" functions build on this idea: Someone writes code for them, typically splitting it into a `.h` file with just declarations of functions (e.g., `stdio.h`) and a `.c` file with code for functions. The code is compiled and turned into something that can be linked into user programs. What's distributed might be just those `.h` files and the compiled code.

`struct`s – Example(s)

- As an example we might define a `struct` to represent three-dimensional vectors and write some functions for it . . .
- We might even usefully put the `struct` definition in a `.h` file, so we could use it in more than one program (with `#include "3d-vec.h"`).

Minute Essay

- Last time we talked about Conway's Game of Life. Had you seen or heard of it before?
- Do you plan to be here next Monday?