Slide 1

Administrivia

- Readings column on “Lecture topics and assignments” page should be complete now. Notice that I added section 9.3 to the other readings from chapter 9 — shouldn’t have left it out.

  (About the reading — I say best to use it as a supplement to class, so okay to skim, and to not read all examples carefully.)
- Sample solutions to all quizzes online. Scores for Quiz 4 low overall.
- Sample solutions for Homeworks 1 through 5 posted. Grading of Homework 5 in progress.
- Reminder: Homework 6 due . . . by popular demand, Monday.
- Final will cover up through pointers but not beyond. Maybe one more homework, to be due *after* the final. Possibly optional / extra credit.

Slide 2

Multidimensional Arrays Revisited

- Multidimensional arrays are easy to declare:

  ```
  int matrix[100][200];
  ```
- The messy part comes when you try to pass one of these to a function, though as with 1D arrays, VLAs do help. (Without them, there’s really no way to specify *at runtime* all dimensions. The old-C way is to fix and specify all but the first dimension — e.g., for a 2D array, fix the number of columns.)
- Also as with 1D arrays, though, fixed-size arrays and VLAs have limitations, so may need to explicitly allocate at runtime using `malloc`.

Slide 3

Dynamically-Allocated Multidimensional Arrays

- One way — “arrays of arrays”, i.e., i.e., arrays of pointers. Could do this as what textbook calls “ragged arrays” or by building list of pointers into one big 1D array.
- Another way — store data in a 1D array and write functions/macros to convert multiple indices into a single index.
- (Examples to be on Web soon.)

Slide 4

User-Defined Data Types

- Can do a lot just with single variables and arrays (as I know from a long-ago job — software company, complex financial-analysis program, written in old-style FORTRAN with only arrays — !).
- But many things are easier and/or more readable if you can define additional types.
- More-modern languages often provide extensive libraries of data types.
  C doesn’t, but provides tools with which users can write their own (libraries.)

Slide 5

User-Defined Data Types in C — Constructs

- `typedef` — give an existing type a new name. A little more today.
- `enum` — “enumerated data type”. Can make code more readable, but really a thin veneer over integers, and language-level support is limited. Read textbook discussion if interested.
- `struct` — provide a way to define something that groups data of possibly different types. A little more today.
- `union` — provide a way to define different view of the *same* data. Useful in some circumstances but to be used with caution. Read textbook discussion if interested.

Slide 6

Defining New Types with `typedef`

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

```
typedef DATA_VALUE double;
```

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

Slide 7

“Structures” — `struct`

- More complex (interesting?) types can be defined with `struct`. Lets you define a new type as a collection of other types. (If you later learn an object-oriented language, the “classes” it lets you define are similar but with many more features.)

  Simple example — 2D point consisting of (x,y) coordinates. Yes you could use an array of size 2 but this gives a way to reference element by name rather than index.
- Two versions of syntax (next slide) . . .
- (Examples to be on Web soon.)

Slide 8

Defining `struct`s

- One syntax 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 9

Working with `structs`

- Can initialize by putting initial values in curly braces.
- Can assign one struct to another of the same type with assignment operator.
- Can access individual “fields” with their names:

  `.` 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 10

“Linked” Data Structures — Executive-Level Summary

- Many situations in which it’s helpful to be able to represent data using some structure that consists of individual elements linked in some way — as a list, a tree, a more-general graph (in the math sense), etc.
- Many/most programming languages support this. Executive-level summary — you define something that represents one element, with pointers/references to other elements. In C, you would do this by defining a `struct` and writing functions to build the desired structure from these elements.

Slide 11

Example — “Linked Lists”

- Arrays are good for working with lists of elements, but have some significant limitations, namely that it’s not easy to insert/remove elements, or to change size.
- “Linked lists” avoid such problems, but with some costs — code is more complex, memory requirements per element are larger, access to elements by index is much less efficient.
- To get a sense of how they work, review illustrations in textbook.

Slide 12

Minute Essay

- What did we not talk about, or not talk about enough, that you can imagine needing in order to write code for a problem you actually want to solve by programming?