Slide 1

Administrivia

- Reminder: Homework 4 due today. (And if you haven't turned in Homework 3 yet, please do!)
- Next homework to be on Web soon. More information by e-mail.

Slide 2

Computer Representation of Data — Recap

- Integers are usually represented using base 2 (binary). Negative integers are usually represented using "two's complement" notation.
- Fractional values are usually represented as "floating point" (base-2 version of scientific notation).
- Notice that any fixed-size representation leads to behavior not exhibited by the idealized numbers of mathematics.
- Character data can be represented as ASCII (7-bit) or Unicode (originally 16-bit, now more). Other encodings possible too.

Slide 3

Dynamic Memory and C

- With the C89 standard, you had to decide when you compiled the program how big to make things, particularly arrays — a significant limitation.
- Variable-length arrays in C99 standard help with that, but don't solve all related problems:

  In many implementations, space is obtained for them on "the stack", an area of memory that's limited in size.

  You can return a pointer from a function, *but* not to one of the function's local variables (because these local variables cease to exist when you return from the function).

Slide 4

Dynamic Memory and C, Continued

- "Dynamic allocation" of memory gets around these limitations — allows us to request memory of whatever size we want (well, up to limitations on total memory the program can use) and have it stick around until we give it back to the system.

  (The trick here is that most implementations differentiate between two areas of memory, a "stack" used for local variables, and a "heap" used for dynamic memory allocation. Usually the former is more limited in size.)
- To request memory, use `malloc`. To return it to the system, use `free`. (For short simple programs you can not bother with `free`, but for longer and more complicated programs, you should clean up when you can, or eventually you may run out of memory.)
- Compare/contrast with Java — allocate space for objects with `new`, no explicit deallocation, garbage collection. (And in Python and Scala . . . ?)

Slide 5

Dynamic Memory and C, Continued

- Examples:

```
int * nums = malloc(sizeof(int) * 100);
char * some_text = malloc(sizeof(char) *
20);
free(nums);
```

- Some books/resources recommend "casting" value returned by `malloc`. Other references recommend the opposite! But you should check the value — if `NULL`, system was not able to get that much memory.
- (Example — slightly improve sort program.)

Slide 6

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 Java.
- 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 7

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 8

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 9

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 Java class, but with no methods, and public fields only.
- Two versions of syntax (next slide) . . .

Slide 10

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 11

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 12

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 tutorial (or elsewhere) about this; it can be useful, but can also make code more difficult to understand.

  (Example: "show in hexadecimal" program.)

Slide 13

Minute Essay

- TBA