Slide 1

Administrivia

- Reminder: Homework 4 due today.

Slide 2

Minute Essay From Last Lecture

- (Question was about uses for I/O redirection.)
- "Glue" together programs in different languages? (Maybe! Depends on what you mean by "glue"?)
- Kill processes on machines where others are logged in. (Well, only superuser can kill others' processes.)
- Many answers seemed to be more about working with files in general than with I/O redirection. In my thinking redirection is more about being able to easily decide at runtime whether program input/output should be interactive or from/to files or other programs.

Slide 3

"It's All Ones and Zeros"

- At the hardware level, all data is represented in binary form — ones and zeros. (Why? hardware for that is simpler to build.)
- How then do we represent various kinds of data? First a short review of binary numbers . . .

Slide 4

Binary Numbers

- Humans usually use the decimal (base 10) number system, but other (positive integer) bases work too. (Well, maybe not base 1.)
- In base 10, there are ten possible digits, with values 0 through 9.

  In base 2, there are 2 possible digits (*bits*), with values 0 and 1.
- In base 10, 1010 means what? What about in base 2?

Slide 5

Converting Between Bases

- Converting from another base to base 10 is easy if tedious (just use definition).
- Converting from base 10 to another base? Two algorithms for that . . .

Slide 6

Decimal to Binary, Take 1

- One way is to first find the highest power of 2 smaller than or equal to the number, write that down, subtract it from the number, and continue.
- In somewhat sloppy pseudocode (letting n be the number we want to convert):

```
while $(n > 0)$
        find largest $p$ such that $2^p \leq n$
        write a 1 in the $p$-th output position
        subtract $2^p$ from n
end while
```

Slide 7

Decimal to Binary, Take 2

- Another way produces the answer from right to left rather than left to right, repeatedly dividing by 2 (again n will be the number we want to convert):

```
while $(n > 0)$
        divide $n$ by 2, giving quotient $q$ and remainder $r$
        write down $r$
        set $n$ equal to $q$
end while
```

(Again, this is a bit sloppy.)

Slide 8

Octal and Hexadecimal Numbers

- Binary numbers are convenient for computer hardware, but cumbersome for humans to write. Octal (base 8) and hexadecimal (base 16) are more compact, and conversions between these bases and binary are straightforward.
- To convert binary to octal, group bits in groups of three (right to left), and convert each group to one octal digit using the same rules as for converting to decimal (base 10).
- Converting binary to hexadecimal is similar, but with groups of four bits. What to do with values greater than 9? represent using letters A through F (upper or lower case).

Slide 9

Computer Representation of Integers

- So now you can probably guess how non-negative integers can be represented using ones and zeros — number in binary. Fixed size (so we can only represent a limited range).
- How about negative numbers, though? No way to directly represent plus/minus. Various schemes are possible. The one most used now is *two's complement*: Motivated by the idea that it would be nice if the way we add numbers doesn't depend on their sign. So first let's talk about addition ...

Slide 10

Machine Arithmetic — Integer Addition and Negative Numbers

- Adding binary numbers works just like adding base-10 numbers — work from right to left, carry as needed. (Example.)
- Two's complement representation of negative numbers is chosen so that we easily get 0 when we add $-n$ and n.

  Computing $-n$ is easy with a simple trick: If m is the number of bits we're using, addition is in effect modulo 2^m. So $-n$ is equivalent to $2^m - n$, which we can compute as $((2^m - 1) - n) + 1)$.
- So now we can easily (?) do subtraction too — to compute $a - b$, compute $-b$ and add.

Slide 11

Binary Fractions

- We talked about integer binary numbers. How would we represent fractions?
- With base-10 numbers, the digits after the decimal point represent negative powers of 10. Same idea works in binary.

Slide 12

Computer Representation of Real Numbers

- How are non-integer numbers represented? usually as *floating point*.
- Idea is similar to scientific notation — represent number as a binary fraction multiplied by a power of 2:

$$x = (-1)^{sign} \times (1 + frac) \times 2^{bias+exp}$$

and then store $sign$ $frac$, and exp. Sign is one bit; number of bits for the other two fields varies — e.g., for usual single-precision, 8 bits for exponent and 23 for fraction. Bias is chosen to allow roughly equal numbers of positive and negative exponents.

- Current most common format — “IEEE 754”.

Slide 13

Numbers in Math Versus Numbers in Programming

- The integers and real numbers of the idealized world of math have some properties not completely shared by their computer representations.
- Math integers can be any size; computer integers can't.
- Math real numbers can be any size and precision; floating-point numbers can't. Also, some quantities that can be represented easily in decimal can't be represented in binary.
- Math operations on integers and reals have properties such as associativity that don't necessarily hold for the computer representations. (Yes, really!)

Slide 14

Computer Representation of Text

- We talked already about how "text strings" are, in C, arrays of "characters". How are characters represented? Various encodings possible.
- One common one is ASCII — strictly speaking, 7 bits, so fits nicely in smallest addressable unit of storage on most current systems (8-bit byte).
- Another one is Unicode — originally 16 bits (Java's `char` type), now somewhat more complicated.
- Either encoding can be considered as "small integers".
- C's `char` type often ASCII but doesn't have to be. (Older systems use(d) EBCDIC, an encoding rooted in punched cards.) C also has `wchar_t`, which *could* be Unicode.

A Little About the C Preprocessor

- C logically divides the process of producing an executable into distinct phases. First phase is “preprocessing”.
- Preprocessing makes use of “preprocessor directives”, which start with a #.
- Examples you’ve seen — `#include` to include information about library functions, `#define` to define constants.
- Other functionality includes macros and “conditional compilation”. More in chapter 14, some beyond the scope of this course. Focus is on relatively simple text manipulation.

A Little More About `gcc`

- Many, many compiler options for `gcc`. One of the most useful is `-Wall`.
- To automate using them every time, you can use the UNIX utility `make` . . .

Slide 17

A Little About `make`

- Motivation: Most programming languages allow you to compile programs in pieces ("separate compilation"). This makes sense when working on a large program — when you change something, just recompile parts that are affected.
- Idea behind `make` — have computer figure out what needs to be recompiled and issue right commands to recompile it.
- (To be continued.)

Slide 18

Minute Essay

- None — sign in.