Slide 1

Administrivia

- Reminder: Homework 5 was due last week. If you weren't able to get yours working, remember that you have the option of turning in something preliminary and then following up with a revised version. I haven't been explicit about how long you have for that, but for this homework I'll accept revised versions through next Wednesday at least.
- Homework 6 is on the Web. Due May 6. This is the last homework.

Slide 2

Minute Essay From Last Lecture

- Most people had not previously encountered `make`, except for some who had done so in working with Linux internals or the like (e.g., recompiling a kernel).
- Some people observed that IDEs seemed to offer similar functionality. True. Worth noting that `make` is not very good at dealing with circular dependencies such as one finds in Scala and Java.

Slide 3

make — Review/Recap

- (Review slides from previous lecture — probably no time to do this in class but I suggest you do.)
- One use for `make` is just to compile with your favorite options. But this can be somewhat annoying since you either have to put a copy of the `Makefile` in every directory or use `-f`. A workaround might be to use a script such as the one (now) on the "sample programs" page.

Slide 4

One More Useful Tool — valgrind

- `valgrind` can check for a lot of potential errors — including errors in use of `malloc` and `free`.
- Compile with `-g` and `-O0` and

  `valgrind` executable-name

Slide 5

Linked Data Structures in C

- We now have what we need to write "linked data structures" (e.g., linked lists) in C. Conceptually much the same as in more-abstract languages, but details are more verbose, and must explicitly allocate memory and free it when done. Support for generics is sort of possible using `void *`, just not as pretty or type-safe.
- Example of this and of use of `make` — sorted linked list of `int` on "sample programs" page. (Review very briefly now, more later.)

Slide 6

"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 7

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 8

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 9

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 10

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 11

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 12

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 13

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 14

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 15

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:

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

  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 16

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 17

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.

Slide 18

Minute Essay

- What if anything was interesting, difficult, or otherwise noteworthy about Homework 5?