CMSC 216
Introduction to Computer Systems
Lecture 24
Data Representation and Libraries

Jan Plane & Alan Sussman
{jplane, als}@cs.umd.edu

Administrivia

- Project 6 due next Friday, 12/10
  - public tests posted
- Project 4 grades visible later today
  - and email with style grades
- Last quiz tomorrow in discussion
- Final exam Monday, Dec. 13, 4-6PM, BRB 1101
  - practice exam posted by early next week
- Read Sections 2.2-2.4 and 7.6-7.13 of Bryant and O'Hallaron

Parts of Sections 2.1-2.4, Bryant and O'Hallaron

DATA REPRESENTATION

Representing characters

- We need:
  - to be able to represent common characters
  - to have standards so computers can interoperate
- Common formats
  - ASCII
    - is the most commonly used character code
    - uses 7 bits for characters (stored in 8 bits normally)
  - EBCDIC
    - an 8-bit code, used now only by some IBM mainframes
  - UNICODE
    - a family of encodings - 8, 16, and 32 bits per character
    - allows a greater variety of characters
    - is able to represent virtually any character in use today in any language, and some no longer in use

ASCII

- Represents normal characters on US keyboards
  - `A-Z` (the characters numbered 65-90)
  - `a-z` (97-122)
  - `0-9` (48-57)
  - space (32)
  - control characters (0-31, 127)
    - the first 26 (after 0) of the 33 ASCII control characters have names Ctrl-A - Ctrl-Z
    - for example, ASCII character 13, Ctrl-M, is CR (carriage return) (`\r` in C); ASCII char. 9, Ctrl-I, is HT (horizontal tab) (`\t` in C)
  - punctuation: `!@#$%^&*()_+-=[]{}|\;:"'<>?,./` (the remaining characters)
- The UNIX command "**`man ascii`**" shows the ASCII character set

UNICODE

- Different representations
- UTF-32: a 32-bit representation of all characters
  - all characters are the same size
  - uses lots of space (twice as much as UTF-16 for most things, four times as much as ASCII for many things)
- UTF-16: a 16-bit representation of characters
  - some characters are stored in two-character forms
  - is popular since most things can be represented in 16 bits
- UTF-8: an 8-bit representation of characters
  - provides backwards compatibility with ASCII
    - the low 7 bits are exactly ASCII
    - if the high bit is on it indicates part of UNICODE extensions
  - popular for web and other applications

Representing unsigned integers

- **All data** is stored in binary
  - all digits are 0 or 1
- In an unsigned number every bit position *i* represents the value 2*i*, where *i* is 0 for the rightmost bit. The value of a number is the sum of the values of the bit positions containing a 1.
- Example bit position values for an 8-bit number:

128	64	32	16	8	4	2	1

- The range of values that can be stored is 0 to $2^n - 1$, where *n* is the number of bits used
  - for example, using 16 bits, the numbers 0 to 65,535 can be represented

Representing signed integers

- Signed integers are usually stored using two's complement
  - the leftmost bit indicates if a number is positive (0) or negative (1)
- To get the two's complement representation of a negative value, flip all the bits of the positive value and add 1
- Two's complement allows easy addition of positive and negative numbers (just ignore overflow)
- The range of values that can be stored is -2^{n-1} to $2^{n-1}-1$ for *n* bits
  - for example, using 16 bits, the numbers -32,768 to 32,767 can be represented
- Two's complement isn't the same thing as an unsigned number with a sign bit
  - -5 is ~(5) + 1 = 11111010 + 1 = 11111011, not 10000101

Floating point representation

- Computers normally use a radix of 2 – binary (people often prefer a radix of 10 – decimal)
- Examples of floating point numbers

  $10.5_{10} = 1010.1_2 = 1.0101 \times 2^3$

  $7.4375_{10} = 111.0111_2 = 1.110111 \times 2^2$
- Decimal/binary points:

10^3	10^2	10^1	10^0	.	10^{-1}	10^{-2}	10^{-3}	10^{-4}
2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}

How are floats/doubles represented internally?

- Each number has three parts:
  - its sign (s), which is 0 for positive numbers, and 1 for negative numbers
  - a mantissa (m), which represents a number between 0 and 1
    - it's represented as a binary number, i.e., ½ = 0.1
    - it's normalized into [1,2) (the exponent is adjusted as needed)
  - an exponent (e), which designates the position of the binary point
- A number is $(-1)^s \times m \times r^e$, where r is the radix
  - the number $6132.789_{10} = 1 \times 6.132789 \times 10^3$ (the radix is 10 for this example)
  - the number $0.05_{10} = 1 \times 5.0 \times 10^{-2}$ (radix is also 10)
  - the number $-1001.1110_2 = -1 \times 1.001110 \times 2^3$ (here the radix is 2)
- This is much like scientific notation, with the addition of the sign as a factor, and the ability to use a base other than 10

The IEEE 754 floating point standard

- The IEEE 754 floating point standard has different sizes for values:
  - 32 bit floating point (C float):
    - 1 sign bit, 8 bits exponent, 23 bits mantissa
    - the range of representable values is approximately $2^{-126} .. 2^{128}$, which is approximately $1.2 \times 10^{-38} .. 3.4 \times 10^{38}$
  - 64 bit floating point (C double):
    - 1 sign bit, 11 bits exponent, 52 bits mantissa
    - this is the precision most commonly used for real applications
    - the range of representable values is approximately $2^{-1022} .. 2^{1024}$, which is approximately $2.2 \times 10^{-308} .. 1.8 \times 10^{308}$
  - 128 bit floating point (quad):
    - 1 sign bit, 15 bits exponent, 112 bits of mantissa
    - this is not commonly used

More about IEEE 754 floating-point numbers

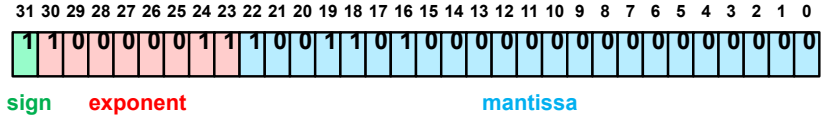
- The leading 1 of the mantissa isn't stored:
  - the binary point (like a decimal point) is moved just to the right of the leftmost nonzero digit
  - but in binary, the leftmost nonzero digit must be a 1, so there's no need to actually store it, giving one more bit of precision in the mantissa for free
- The exponent:
  - uses a *bias*, rather than two's complement, for storing negative as well as positive exponents. The bias is added to the exponent's value.
  - the bias is 127 for single-precision IEEE numbers (C `float`'s), and 1023 for double-precision numbers (C `double`'s)
- The use of a bias allows the representation of the number zero to be all zeros; in fact, an exponent of all 1s or all 0s represents a special number
  - 0, infinities, NaN, denormalized numbers

Example IEEE floating-point number

- Here's how the example number -25.625 is represented in IEEE floating point (single precision):
  - The sign bit (one bit) is 1, since the number is negative; we compute the absolute value of the number below
  - To compute the mantissa (23 bits):
    - write the number in binary, with a binary point:
      25_{10} is 11001_2
      $.625_{10}$ is 1/2 + 1/8, which is $.101_2$
      so 25.625_{10} is 11001.101_2
    - move the binary point right after the first nonzero digit, giving 1.1001101 (moved 4 places to the left)
    - drop the leading 1 (and the binary point), giving 1001101
    - add zeros to the right to get 23 bits (here 16 zeros are needed)
    - so the mantissa is **1001101**0000000000000000

Example IEEE floating-point number, cont.

- Recall the example number is -25.625
  - To determine the exponent (8 bits):
    - in the previous step, we moved the binary point 4 places to the left to place it to the right of the first nonzero digit, so the exponent value is 4
    - to bias the exponent, we add 127; 127 + 4 = 131, so the value of the exponent field is 131
    - 131 in binary is 10000011
- Putting it all together, the number is represented as
  $(-1)^1 \times 1.1001101 \times 2^4 = -1.6015625 \times 16 = -25.625$
- And the number is stored in memory as



Imprecision with real numbers

- The real numbers are dense (unlike the integers), but anything in computer memory has to be stored in a finite bit representation; this causes imprecision
- First consider an analogy with decimal numbers:
  - There are some numbers that can't be represented exactly in a finite number of digits - they require an infinite number of repeating digits
  - Example: 1/3 = .3333333333333...
  - Suppose we have only a fixed number of decimal digits in which to express 1/3, say for example 8 digits. The closest we can get is .33333333. But notice this is .0000000033333... away from the actual number 1/3
  - The next representable number (if we only have 8 digits) is .33333334, and any number between these two can only be approximated as one or the other of these two values- there are no values between them

Imprecision with real numbers, cont.

- In binary there are also real numbers (not necessarily the same ones as in decimal) that can't be represented in a finite number of (binary) digits
- Example: $(1/3)_{10}$ = .01010101010101010101...
- Another example: $(1/5)_{10}$ = .00110011001100110011...
- If we have only four binary digits, the closest we can come to representing $(1/5)_{10}$ is .0011 (1/8 + 1/16 = .1875)
- If we have eight binary digits, we can come closer to representing $(1/5)_{10}$: .00110011 (1/8 + 1/16 + 1/128 + 1/256 = .19921875). The more digits we have, the closer we can come to representing it
- But we'll never get exactly to 0.2_{10}, if we only have a fixed number of binary digits in which to represent the number

Imprecision with real numbers, cont.

- The IEEE representation of 1/5, with a 23-digit mantissa, is 00111110010011001100110011001101, which works out to 0.20000000298023223876953125
- The next smaller bit pattern (only one bit different) is 00111110010011001100110011001100, which works out to 0.19999998807907104492187500
- **There is no (single-precision) IEEE 754 float between these two values** because, with a fixed 23 digits of mantissa, there is no bit pattern between them
- If you try to compute or store values between these, such as 0.19999998825, 0.19999998850, 0.19999998875, etc., they'll all be represented as 00111110010011001100110011001100, which is 0.19999998807907104492187500

Another example (a large number)

- The IEEE float 375207297024.0 is represented as 01010010101011101011100000110010
- The next bit pattern is 01010010101011101011100000110011, which is the float 375207329792.0
- These two numbers are 32,768 apart, yet there is no IEEE 754 float value between them

An example 32-bit number

- On a 32-bit machine, consider the bit pattern 11001101010101110101110000011001:
  - as an unsigned integer, this bit pattern represents the value 3445054489
  - as a two's complement signed integer, this bit pattern represents the value -849912807
  - and as a single-precision IEEE float, this bit pattern represents the value -225821072.0
- A pattern of bits can represent lots of different things - we need to know what kind of thing they're supposed to represent to make sense of them
- Given the information above, what does the following code print?

```
unsigned int num = 3445054489;
printf("%f\n", * (float *) &num);
```


Sections 7.6-7.13, Bryant and O'Hallaron

LIBRARIES

Motivation

- Suppose we wrote some really useful functions to do something, and want to distribute them to clients or to other programmers. How can we do this?
  - give out the source code
  - give out the object code
  - in the form of a library
- What's a library? Basically a collection of object files that provide compiled functions performing some related tasks (often utility functions)
- Libraries can be linked into programs
  - linking can happen prior to execution (at compilation)
  - linking can be done **during** program execution

Comparison

- Giving out source code is platform-independent, but it needs to be recompiled and relinked by every client or user. It exposes our intellectual property or trade secrets, which we may not want to do
  - it also makes details of the implementation visible, and clients may come to rely on that
- Giving out object code doesn't require recompilation of that object code, but it requires relinking of the application which is going to use it
- Giving out either object code or a library is platform-dependent, but all we have to provide besides the object code or library is the header file (at most), not the source code
- Giving out some types of libraries doesn't even require relinking of the application using it

Object code vs. library

- In UNIX systems the linker includes an entire object file in an executable, even if not all the functions in it are used. With libraries, the linker can include the code for only the functions from the library that are actually called by a program
- The linker has to search through an object file to find each function, but a library can be indexed for faster lookup by the linker, so compilation is slightly faster
- Some types of libraries allow different executables to share the same library code, saving disk and memory space
- The UNIX utility `nm` lists the symbols (functions and other names) in a library

Types of libraries

- Static libraries (extension `.a`, for "archive")
  - are linked into a program as part of the linking phase of compilation
  - require space in each executable that uses them, which uses disk space, and memory space during execution
  - updating a library requires recompiling (relinking) all applications using it
  - are easy to use
- Shared libraries (extension `.so`, for "shared object")
  - are linked into a program at program startup, or during execution
  - require only one copy for the entire system
  - libraries can be updated independent of applications
  - must have version numbers associated with them, to control which version works with which applications