CSC161 2010F *Imperative Problem Solving*

Class 12: Binary Representation and Bitwise Operators

Held: Wednesday, 15 September 2010

Summary: We begin to study binary representation, focusing on representations of integers.

Related Pages:

- EBoard.
- Lab: C's Bitwise Operations.
- Reading: K&R 2.9, 6.9 ; Wright: A Tutorial on Binary Numbers..
- Due: Assignment 3: Explaining Assignments and I/O.

Notes:

- EC for attending tomorrow's Thursday extra: Dr. Davis on Participatory Design.
- EC for attending tomorrow's Scholars' Convocation on Iran.
- Reading for Friday: IEEE Floating-Point Representation of Real Numbers.
- Although I have a lab listed for today, we are unlikely to do that lab.
- Are there questions on Assignment 3?
- One question: How do I use `math.h`?

Overview:

- Why study underlying representations?
- Basics of binary.
- Unsigned integers.
- Signed integers.
- Some of C's bitwise operations.

Why Study Representations

- As you'll note, we have a few classes devoted to underlying representations of a variety of types of numbers.
- Why do we study these issues in this course?
- As you've noted, C makes some assumptions that you understand the underlying representations.
  - Key types like `short`, `long`, and more.
  - Bitwise operations
- Successful programming in C requires you to understand these underlying representations.
- Some of the most important:
  - Unsigned integers
  - Signed integers
  - IEEE floating-point numbers.

    - Characters (ASCII and Unicode)

Binary

- On most computers, the smallest unit of information is the *bit*, which has only two possible values: off/on, 0/1, false/true, whatever.
- We combine bits into reasonable groups, such as the *byte* and *word*.
    - On most computers, a byte is 8 bits and a word is big enough to hold an address in memory.
- Clearly, we need ways to interpret sequences of bits.
- The interpretation is just that: An agreed-upon way to understand the meanings of the bits.
    - Common interpretations are encoded in most hardware.
- Generally, we have rules for interpreting bit sequences as integers, and then rules for interpreting other values in terms of integers.
    - E.g., characters
- For floating-point numbers, we have a different representation.

Unsigned Integers

- Base two numbers. Nothing more, and nothing less.
- Practice!

Signed Integers

- First problem: How to represent the sign.
- Typical solution: Use the leftmost bit to indicate sign.
    - 0 means "positive"
    - 1 means "negative"
- Next problem: How does one interpret the remaining bits?
- Many possible options. Here are four of the most common.
    - "Normally". The remaining N-1 bits are simply an unsigned integer.
        - Formal term: *Signed magnitude*
    - "Backwards". 0 represents a negative 1, 1 represents 0.
        - Formal term: *One’s complement*
    - "Encoded". To represent signed N in k bits, we write unsigned $N+2^{k-1}$.
        - Note that in this system, a leading 0 means "negative" and a leading 1 means "positive".
        - This system is called *Excess* 2^{m-1}
    - "Just plain weird": We think procedurally. To negate a number, we flip all the bits and add 1.
        - This system is called *Two’s complement*
- Exercise: Let’s try a few numbers in 5 bit notation.
- What criteria might one use to decide which one to use?
    - Ease of interpreting numbers.
    - Ease of adding numbers.
    - Ease of negating

  - Ease of subtracting
  - Range of numbers representable
  - Others ...
- We’ll try each of these

Bitwise Operations in C

Logical

- & - bitwise "and"
  - 0 and 0 is 0
  - 0 and 1 is 0
  - 1 and 0 is 0
  - 1 and 1 is 1
- | - bitwise "or"
  - 0 or 0 is 0
  - 0 or 1 is 1
  - 1 or 0 is 1
  - 1 or 1 is 1
- ~ - bitwise "not"
  - not 0 is 1
  - not 1 is 0
  - Why is this different than negate?
- We can use these to extract bits from an integer.
  - To access the kth bit of i, compute 2^k and and it with i
  - If the result is 0, the bit was 0. If the result is non-zero (true, in C), the bit was 1.
- We can use these to change bits in an integers
  - To change the kth bit of i, compute 2^k and or it with i.
- We often use integers to store a variety of flags (settings)
  - One bit per flag
  - If the bit is on, the flag is set
  - If the bit is off, the flag is not set

Shifting

- << - left shift
- >> - right shift
- Lots of variants.

Lab

- Lab.
- This is an optional lab. We probably won’t have time for it.