# Introduction to Computer Science and Programming in C

Session 12: October 9, 2008

Columbia University

# Announcements

- Homework 2 is out. Due 10/14 before class
- Midterm Review on 10/16, exam on 10/21
- Bert's office hours on 10/14 moved to Wednesday 10/15, 1-3 PM (or by appointment)

# Review

- Leftovers from FILE I/O:
  ferror(), feof(), rewind()
- C preprocessor:
  - #include
  - #define
  - #ifdef, #else, #endif, #ifndef

# Today

- Bit Operations and Boolean Logic

# Bit Operations

- Sometimes we want to manipulate the actual bits of variables
- This is useful when we want extremely efficient and compact programs
  - Specialized devices like cellphones, watches
  - Graphics programming

# Hexadecimal

- It's often useful to represent numbers in hexadecimal, which is base-16, instead of binary (base-2) or decimal (base-10).
- This is because hexadecimal (base-16) means each symbol corresponds to four bits in binary (base-2)
- printf() can convert to hex. using the placeholder %x

# Hexadecimal

- Consider a four-bit “word”.
- The word can have 2^4 possible settings: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111
- In binary, these settings represent 0-15
- In hex., we encode these settings as the symbols: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

# Hexadecimal

| Hexadecimal | Binary | Hexadecimal | Binary |
|---|---|---|---|
| 0 | 0000 | 8 | 1000 |
| 1 | 0001 | 9 | 1001 |
| 2 | 0010 | A | 1010 |
| 3 | 0011 | B | 1011 |
| 4 | 0100 | C | 1100 |
| 5 | 0101 | D | 1101 |
| 6 | 0110 | E | 1110 |
| 7 | 0111 | F | 1111 |

# Hexadecimal

| Hexadecimal | Binary | Hexadecimal | Binary |
|---|---|---|---|
| 0x0 | 0000 | 0x8 | 1000 |
| 0x1 | 0001 | 0x9 | 1001 |
| 0x2 | 0010 | 0xA | 1010 |
| 0x3 | 0011 | 0xC | 1011 |
| 0x4 | 0100 | 0xB | 1100 |
| 0x5 | 0101 | 0xD | 1101 |
| 0x6 | 0110 | 0xE | 1110 |
| 0x7 | 0111 | 0xF | 1111 |

# Bit Operators

- C has a few operators that manipulate bits:

  - and &
  - or |
  - xor (exclusive or) ^
  - not ~

  Boolean Logic

  - Shift, << and >>

# Bitwise “and”

- For each bit independently, if both **operands** are 1, the corresponding bit in the result is 1.
- 1 & 1 = 1  1 & 0 = 0
  0 & 1 = 0  0 & 0 = 0
- ```
  char c1 = 0x45, c2 = 0x71, c3;
  c3 = c1 & c2;
  ```
- ```
  c1: 0100 0101
  c2: 0111 0001
  c3: 0100 0001   ------> 0x41
  ```

# Bitwise “or”

- For each bit independently, if either **operand** is 1, the corresponding bit in the result is 1.
- $1 \mid 1 = 1 \qquad 1 \mid 0 = 1$
  $0 \mid 1 = 1 \qquad 0 \mid 0 = 0$
- ```
  char c1 = 0x45, c2 = 0x71, c3;
  c3 = c1 | c2;
  ```
- ```
  c1: 0100 0101
  c2: 0111 0001
  c3: 0111 0101   ------> 0x75
  ```

# Bitwise "xor"

- For each bit independently, if **exactly one** operand is 1, the corresponding result bit is 1.
- $1 \text{ ^ } 1 = 0 \qquad 1 \text{ ^ } 0 = 1$
  $0 \text{ ^ } 1 = 1 \qquad 0 \text{ ^ } 0 = 0$
- ```
  char c1 = 0x45, c2 = 0x71, c3;
  c3 = c1 ^ c2;
  ```
- ```
  c1: 0100 0101
  c2: 0111 0001
  c3: 0011 0100   ------> 0x34
  ```

# Bitwise “not”

- AKA complement, flip. Set all 1’s to 0, and all 0’s to 1.
- $\sim 1 = 0 \quad \sim 0 = 1$
- ```
  char c1 = 0x45;
  c2 = ~c2;
  ```
- ```
  c1: 0100 0101
  c2: 1011 1010   ------> 0xBA
  ```

# Left Bit Shift <<

- Moves bits to the left
  Bits shifted outside the variable are erased
  Bits shifted into the variable are 0's
- 00100 << 1 = 01000  00100 << 2 = 10000
- ```
  char c1 = 0x45, c2;
  c2 = c1 << 2;
  ```
- ```
  c1: 0100 0101
  c2: 0001 0100   ------> 0x14
  ```

# Right Bit Shift >>

- Moves bits to the right
  Bits shifted in from the right take the value of the sign bit (unless variable is unsigned)
- On signed variables, the first bit is reserved to indicate sign **and** the meanings of 1 and 0 are reversed
- 00100 >> 1 = 00010  00100 >> 2 = 00001
  11011 >> 1 = 11101  11011 >> 2 = 11110

# Shifting to multiply

- **x << 1** is equal to **x*2**
  **x >> 1** is equal to **x/2**
- The flipped representation of negative numbers preserves this.
- Just like shifting in decimal is equivalent to multiplying or dividing by 10: 123 x 10 = 1230

# Shifting to multiply

- Shifting is faster than multiplying on modern computer hardware
- But compilers know this, so **gcc** will automatically convert code to take advantage
- So no need for programmers to use shifting to multiply.

# Set, clear, test

- Now that we can operate on lots of bits at a time, how do we work with one bit at a time?
- We want to do three things to each bit:
  - **Set** a bit: set bit to 1
  - **Clear** a bit: set bit to 0
  - **Test** a bit: get the value of bit

# Testing a bit

- Say we want to get the value of the 4th bit of:
  `char c1;`
- We can "**and**" it with 0000 1000 (0x08),
  which we can also write as:
  `1 << 3`
- `(c1 & (1 << 3))!=0`
- `c1: 0100 0101`
  `&  0000 1000`
- `(1 << 3)` is sometimes called a "**mask**"

# Setting a bit

- Again using a mask, we can set the 4th bit by "or"-ing with the mask:

- ```
  c1 = c1 | (1 << 3);
  c1 |= (1<<3);
  ```

- ```
  c1: 0100 0101
   |  0000 1000
  ---------------
      0100 1101
  ```

# Clearing a bit

- Clearing is slightly trickier. But we can just flip all the bits then set the bit we want and flip it back.

- `c1 = ~( (~c1) | (1 << 3));`

- c1:  0100 0101

  ~c1: 1011 1010
  | 0000 1000
  1011 1010

~1011 1010
0100 0101

# Wrap up

- Bit operations are very fast and space-efficient
- Useful when writing time sensitive code
- (Unix uses bit operations for job management)

# Reading

- Practical C Programming Ch. 11