Review: Data Representation and Boolean operators in C

Based on slides © O'Hallaron
Additional material © 2018 Narahari

Next…a little bit of “reality”

- look at how some of the concepts we have studied take shape in ‘real life’
  - C programming and O/S

Byte-Oriented Memory Organization

- Programs Refer to Virtual Addresses
  - Conceptually very large array of bytes
  - Actually implemented with hierarchy of different memory types
    - SRAM, DRAM, disk
    - Only allocate for regions actually used by program
  - In Unix and Windows, address space private to particular “process”
    - Program being executed
    - Program can clobber its own data, but not that of others

- Compiler + Run-Time System Control Allocation
  - Where different program objects should be stored
  - Multiple mechanisms: static, stack, and heap
  - In any case, all allocation within single virtual address space

Encoding Byte Values

- Byte = 8 bits
  - Binary 00000000₂ to 11111111₂
  - Decimal: 0₁₀ to 255₁₀
  - Hexadecimal 0₀₁₆ to FF₁₆
    - Base 16 number representation
    - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write FA1D37B₁₆ in C as 0xFA1D37B
      - Or 0xfald37b

Hex | Decimal | Binary
---|---------|-------
0   | 0       | 0000
1   | 1       | 0001
2   | 2       | 0010
3   | 3       | 0011
4   | 4       | 0100
5   | 5       | 0101
6   | 6       | 0110
7   | 7       | 0111
8   | 8       | 1000
9   | 9       | 1001
A   | 10      | 1010
B   | 11      | 1011
C   | 12      | 1100
D   | 13      | 1101
E   | 14      | 1110
F   | 15      | 1111
Machine Words

- Machine Has “Word Size”
  - Nominal size of integer-valued data
    - Including addresses
  - Some current machines are 32 bits (4 bytes)
    - Limits addresses to 4GB
    - Becoming too small for memory-intensive applications
  - Higher-end systems are 64 bits (8 bytes)
    - Potentially address \( \approx 1.8 \times 10^{19} \) bytes
  - Machines support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes

Word-Oriented Memory Organization

- Addresses Specify Byte Locations
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)

<table>
<thead>
<tr>
<th>32-bit Words</th>
<th>64-bit Words</th>
<th>Bytes</th>
<th>Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr = 0000</td>
<td>Addr = 0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>Addr = 0004</td>
<td>Addr = 0008</td>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0008</td>
<td>0002</td>
<td>0002</td>
</tr>
<tr>
<td>Addr = 0012</td>
<td>Addr = 0012</td>
<td>0003</td>
<td>0003</td>
</tr>
<tr>
<td>Addr = 0016</td>
<td>Addr = 0016</td>
<td>0004</td>
<td>0004</td>
</tr>
<tr>
<td>Addr = 0020</td>
<td>Addr = 0020</td>
<td>0005</td>
<td>0005</td>
</tr>
<tr>
<td>Addr = 0024</td>
<td>Addr = 0024</td>
<td>0006</td>
<td>0006</td>
</tr>
<tr>
<td>Addr = 0028</td>
<td>Addr = 0028</td>
<td>0007</td>
<td>0007</td>
</tr>
<tr>
<td>Addr = 0032</td>
<td>Addr = 0032</td>
<td>0008</td>
<td>0008</td>
</tr>
<tr>
<td>Addr = 0036</td>
<td>Addr = 0036</td>
<td>0009</td>
<td>0009</td>
</tr>
<tr>
<td>Addr = 0040</td>
<td>Addr = 0040</td>
<td>0010</td>
<td>0010</td>
</tr>
<tr>
<td>Addr = 0044</td>
<td>Addr = 0044</td>
<td>0011</td>
<td>0011</td>
</tr>
<tr>
<td>Addr = 0048</td>
<td>Addr = 0048</td>
<td>0012</td>
<td>0012</td>
</tr>
<tr>
<td>Addr = 0052</td>
<td>Addr = 0052</td>
<td>0013</td>
<td>0013</td>
</tr>
<tr>
<td>Addr = 0056</td>
<td>Addr = 0056</td>
<td>0014</td>
<td>0014</td>
</tr>
<tr>
<td>Addr = 0060</td>
<td>Addr = 0060</td>
<td>0015</td>
<td>0015</td>
</tr>
</tbody>
</table>

Byte Ordering

- How should bytes within multi-byte word be ordered in memory?
- Conventions
  - Sun’s, PowerPC (old Mac’s) are “Big Endian” machines
    - Least significant byte has highest address
    - Big end first
  - Intel x86, Alphas, PC’s are “Little Endian” machines
    - Least significant byte has lowest address
    - Little end first
  - Most network protocols use Big Endian
- The terms big-endian and little-endian come from Jonathan Swift’s eighteenth-century satire Gulliver’s Travels. The subjects of the empire of Blefuscu were divided into two factions: those who ate eggs starting from the big end and those who ate eggs starting from the little end.

Byte Ordering Example

- Big Endian
  - Least significant byte has highest address
- Little Endian
  - Least significant byte has lowest address
- Example
  - Variable \( x \) has 4-byte representation \( 0x01234567 \)
  - Address given by \( &x \) is \( 0x100 \)

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>Little Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100 0x101 0x102 0x103</td>
<td>0x100 0x101 0x102 0x103</td>
</tr>
<tr>
<td>01 23 45 67</td>
<td>67 45 23 01</td>
</tr>
</tbody>
</table>
Representing Integers

- int A = 15213;
- int B = -15213;
- long int C = 15213;

<table>
<thead>
<tr>
<th>Decimal: 15213</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary: 0011 1011 0110 1101</td>
</tr>
<tr>
<td>Hex: 0003 B 6 D</td>
</tr>
</tbody>
</table>

- Decimal: -15213 |
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary: 1100 0100 1001 0010</td>
</tr>
<tr>
<td>Hex: FFFF C 4 9 3</td>
</tr>
</tbody>
</table>

- Little endian layout for A:
  - For B
  - For C

- Big endian layout for A:
  - For B
  - For C

Binary Representation: Summary

- Every storage location stores a finite sequence of bits
  - 8-bit, 16-bit, 32-bit etc.
- The same bit string can mean different things depending on how the program wants to look at it.

<table>
<thead>
<tr>
<th>Address</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>36</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>37</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>38</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Unsigned: +129
2C: -127
2C: 109
ASCII: 'm'

Basic Logic Operations

- Equivalent Notations
  - not A = A’ = A
  - A and B = A.B = A & B = A intersection B
  - A or B = A+B = A ∨ B = A union B
- Other common logic operations:
  - NAND = NOT AND
    - Find AND and then Complement it (invert bit)
  - NOR = NOT OR
    - Find OR and then Complement it
  - XNOR = NOT XOR
Bitwise Logical Operations

- View n-bit field as a collection of n logical values
  - Apply operation to each bit independently

  Bitwise AND: useful for clearing bits
  - AND with zero = 0
  - AND with one = no change
  \[
  \begin{array}{c}
  11000101 \\
  \text{AND} \\
  00001111 \\
  = 00000101
  \end{array}
  \]

  Bitwise OR: useful for setting bits
  - OR with zero = no change
  - OR with one = 1
  \[
  \begin{array}{c}
  11000101 \\
  \text{OR} \\
  00001111 \\
  = 11001111
  \end{array}
  \]

  Computers don’t support individual bits as a data type
  - Just use least significant of n-bit integer
  - Integers are generally more useful

Data Representations

- Sizes of C Objects (in Bytes)
  \[
  \begin{array}{c|c|c|c|c}
  \text{C Data Type} & \text{Compaq Alpha} & \text{Typical 32-bit} & \text{Intel IA32} \\
  \hline
  \text{int} & 4 & 4 & 4 \\
  \text{long int} & 8 & 4 & 4 \\
  \text{char} & 1 & 1 & 1 \\
  \text{short} & 2 & 2 & 2 \\
  \text{float} & 4 & 4 & 4 \\
  \text{double} & 8 & 8 & 8 \\
  \text{long double} & 8 & 8 & 10/12 \\
  \text{char *} & 8 & 4 & 4 \\
  \end{array}
  \]

  – Or any other pointer

Unsigned & Signed Numeric Values

<table>
<thead>
<tr>
<th>X</th>
<th>B2U(X)</th>
<th>B2T(X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>–8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>–7</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>–6</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>–5</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>–4</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>–3</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>–2</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>–1</td>
</tr>
</tbody>
</table>

- Equivalence
  - Same encodings for nonnegative values

- Uniqueness
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding

  \[ \Rightarrow \text{Can Invert Mappings} \]

  - \( U2B(x) = B2U^{-1}(x) \)
    - Bit pattern for unsigned integer
  - \( T2B(x) = B2T^{-1}(x) \)
    - Bit pattern for two’s comp integer

Signed vs. Unsigned in C

- Constants
  - By default are considered to be signed integers
  - Unsigned if have “U” as suffix
    
    \[ 0U, 4294967259U \]

- Casting – nasty stuff!! Or is it fun ?
  - Explicit casting between signed & unsigned same as U2T and T2U
    
    \[
    \begin{array}{ll}
    \text{int } tx, ty; \\
    \text{unsigned ux, uy;} \\
    \text{tx = (int) } ux; \\
    \text{uy = (unsigned) ty;} \\
    \end{array}
    \]

  - Implicit casting also occurs via assignments and procedure calls
    
    \[
    \begin{array}{ll}
    \text{tx = ux;} \\
    \text{uy = ty;} \\
    \end{array}
    \]
Casting Signed to Unsigned

- C Allows Conversions from Signed to Unsigned
  ```
  short int  x = 15213;
  unsigned short int ux = (unsigned short) x;
  short int  y = -15213;
  unsigned short int uy = (unsigned short) y;
  ```

- Resulting Value
  - No change in bit representation – only in interpretation
  - What is value of ux?
  - What is value of uy?

Relation between Signed & Unsigned

- Two’s Complement
- Maintain Same Bit Pattern
- \( w-1 \)
- \( 0 \)
- \(-x\)
- \(+2^{w-1} - 2^{w-1} = 2^w \)
- \( \lfloor x \rfloor = \begin{cases} x & x \geq 0 \\ x + 2^w & x < 0 \end{cases} \)

Casting Surprises

- Expression Evaluation
  - If mix unsigned and signed in single expression, signed values implicitly cast to unsigned
  - Including comparison operations \(<, >, ==, <=, >=\)
  - Examples for \( W = 32 \)
- Constant\(_1\) Constant\(_2\) Relation Evaluation
  | \( 0 \) | \( 0 \) | \( == \) | unsigned |
  | \(-1\) | \( 0 \) | \(<\) | signed |
  | \(-1\) | \( 0 \) | \(>\) | unsigned |
  | \(2147483647\) | \(-2147483648\) | \(<\) | unsigned |
  | \(2147483647\) | \(-2147483648\) | \(>\) | signed |
  | \(-1\) | \(-2\) | \(>\) | unsigned |
**Why Should I Use Unsigned?**

- *Don’t Use Just Because Number Nonzero*
  - Easy to make mistakes
    ```c
    for (i = cnt-2; i >= 0; i--)
      a[i] += a[i+1];
    ```
- *Do Use When Performing Modular Arithmetic*
  - Multiprecision arithmetic
  - Other esoteric stuff
- *Do Use When Need Extra Bit’s Worth of Range*
  - Working right up to limit of word size

**Logical Operations in C**

- C supports both bitwise and boolean logic operations
  - `x & y` bitwise logic operation
  - `x && y` boolean operation: output is boolean value
- What’s going on here?
  - In boolean operation the result has to be TRUE (1) or FALSE (0)
  - Treats any non-zero argument as TRUE and returns only TRUE (1) or FALSE (0)
- In C: logical operators do not evaluate their second argument if result can be obtained from first
  - `a && 5/a` can we get divide by zero error?

**Logical Operators in the C Language**

- Bitwise operators
  - What if variables are n bits long
- Logical operators
  - Conditional statements

**Bitwise Review**

- Can only be applied to integral operands
  - *that is, char, short, int and long*
- *(signed or unsigned)*
  - `&` Bitwise AND
  - `|` Bitwise OR
  - `^` Bitwise XOR
  - `<<` Shift Left
  - `>>` Shift Right
  - `~` 1’s Complement (Inversion)
Bitwise Logical Operators
- View \( n \)-bit number as a collection of \( n \) logical values
  - operation applied to each bit independently
- Number operated on is an \( n \)-bit number
- Operation being performed is logical operation on each bit

Bitwise AND
\[
0101 \text{ AND } 0111 \quad \text{in C:}(5 \ & \ 6)
\]
\[
\begin{array}{c}
0101 \\
0110
\end{array}
\]

Why use bitwise operators?
- Masking operations
  - If we are only interested in last 8 bits of a 16 bit number \( X \), how to extract this?
  - \( X \ & \ x00FF \)
    - Zero out the most significant 8 bits; value of least significant 8 bits is same as the value of these in \( X \)
    - \( x27A4 \ & \ x00FF = x00A4 \)

Bitwise OR
\[
0101 \ OR \ 0111 \quad \text{in C:}(5 \ | \ 6)
\]
\[
\begin{array}{c}
0101 \\
0110
\end{array}
\]
Bitwise NOT (Complement)

\[
\text{NOT 0101 } \quad \text{in C: } \sim 5
\]

\[
0101
\]

Bitwise XOR

\[
0101 \text{ XOR } 0111 \quad \text{in C: } 5^6
\]

\[
0101
\quad 0110
\]

Bitwise NAND

\[
0101 \text{ NAND } 0111 \quad \text{No C Operator}
\quad \sim (5 \& 6)
\]

\[
0101
\quad 0110
\]

Bitwise NOR

\[
0101 \text{ NOR } 0111 \quad \text{No C Operator}
\quad \sim (5 \mid 6)
\]

\[
0101
\quad 0110
\]
Shift Operations
- \( x >> y \)
  - \( x \) right shifted \( y \) bit positions, sign extended/arithmetic shift
    o Sign bit shifted into positions vacated by shifted bits
  - \( x = 011000 \)
  - \( y = 2 \)
  - \( x >> y \)?
  - \( z = 101000 \)
  - \( y = 2 \)
  - \( z >> y \)?
- \( x << y \)
  - \( x \) left shifted \( y \) bit positions, zero placed in positions vacated by shifted bits
  - \( x = ? \)
  - \( y = ? \)

Boolean Relational Operators?
- What is the semantics of:
  - If \( x = 0 \) then ……
  - How many outcomes for \( x = 0 \)?
- Concept of boolean operators
  - Apply logic operators, but treat input and output as boolean variables
    o Only 1 or 0 (True or False) values for entire variable
  - But input strings can be n-bits long?
    o Treat entire string as ONE boolean variable
    o How?

Logical Operations in C
- \( != \) Logical NOT
  - \( !x \)
    o \( !x = 0 \) if \( x \) is non-zero, \( !x = 1 \) if value of \( x \) is zero
- \( && \) Logical AND
  - \( x && y \)
    o \( x && y = 1 \) if value of \( x \) is not zero and value of \( y \) is not zero
    o \( x && y = 0 \) if both \( x \) and \( y \) are zero
- \( || \) logical OR
  - \( x || y \)
    o \( x || y = 1 \) if at least one of \( x,y \) are not zero
    o \( x || y = 0 \) if both \( x,y \) are zero

Examples
- 8 bit numbers, \( f=7, g=8 \)
  - \( f = 00000111 \)
  - \( g = 00001000 \)
  - \( h = f \& g \) (bitwise AND)…
    - \( h = ? \)
  - \( h = (f && g) \) (logical AND)…
    - \( h = ? \)
  - \( h = (f || g) \) (logical OR)…
    - \( h = ? \)
  - \( h = (!f && !g) \)…
    - \( h = ? \)
Why this discussion of Bit manipulation operations in C.....Project 2!

- Project 2: Given a set of functions, each of which does not use conditional statements and implements some bit manipulation function, determine the function being implemented.
  - Rewrite the code to provide an equivalent more readable code using any C operators including conditional statements.