Serialization and Bit Operations

CS 113: Introduction to C

Instructor: Saikat Guha

Cornell University

Fall 2006, Lecture 9
Sending data between programs
  - Disk
  - Network
  - Pipes

Between programs on multiple hosts
  - Different endianness
  - Different architectures
Binary vs. Text

Binary...

- Compact
- Easy to encode/decode
- Faster

e.g. IP, TCP, AIM, ...

Text...

- Easily debugged
- (Can be) self-documenting
- Arch/Endian independent

e.g. HTTP, SMTP, MSN
Handling Endianness

Decimal: 3735928559
Binary: 11011110101011011111011101111
Hex: 0xdeadbeef
Big Endian: 0xde 0xad 0xbe 0xef
Little Endian: 0xef 0xbe 0xad 0xde

Always in big-endian form when loaded into the CPU
### AND-Mask (clear bits)

<table>
<thead>
<tr>
<th>a &amp; b</th>
<th>110111101011011011101111</th>
<th>0xdeadbeef</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>1011111011101111</td>
<td>&amp;</td>
</tr>
<tr>
<td>00000000000000001111111000000000</td>
<td>0x0000FF00</td>
<td>=</td>
</tr>
<tr>
<td>=</td>
<td>00000000000000001011111000000000</td>
<td>=</td>
</tr>
<tr>
<td>00000000000000001011111000000000</td>
<td>0x0000be00</td>
<td></td>
</tr>
</tbody>
</table>
### OR-Mask (sets bits)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>11011110101011011011101111</td>
<td>0xdeadbeef</td>
<td></td>
</tr>
<tr>
<td>00000000000000000101010100000000</td>
<td>0x00005500</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>110111101010110111111111101111</td>
<td>0xdeadFFef</td>
<td></td>
</tr>
</tbody>
</table>
# Bit-Operations

## Left-Shift

The expression `a << b` performs a left shift operation on the binary number `a` by `b` positions. The result is the binary number shifted to the left by `b` positions.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1101 1110</td>
<td>101 0110</td>
<td>0xdeadbeef</td>
</tr>
<tr>
<td>&lt;&lt; 8</td>
<td>&lt;&lt; 8</td>
<td>=</td>
</tr>
<tr>
<td>1010 1101</td>
<td>0000 0000</td>
<td>0xadbeef00</td>
</tr>
</tbody>
</table>

Example:

- `1101 1110` shifted left by 8 positions equals `1010 1101 0000 0000`.

This operation is useful in serialization and other applications where data needs to be shifted for processing or storage.
Bit-Operations

Right-Shift

\[
a \ll b = \begin{array}{c} 11011110101101111101110111110 \ll 8 \\
0x\text{deadbeef} \ll 8 = \end{array}
\]

\[
00000000110111101010110110111110 = 0x00\text{adbeef}^1
\]

\[\text{1}^1\text{for unsigned ints only. For signed ints, the instead of zero-padding, the top-most bit is repeated}\]
int htonl(int x) {
    int b1, b2, b3, b4, y;

    b1 = (x _____) ___;
    b2 = (x _____) ___;
    b3 = (x _____) ___;
    b4 = (x _____) ___;

    y = (b1 _____) __ (b2 _____)
        __ (b3 _____) __ (b4 _____);

    return y;
}
int htonl(int x) {
    int y;
    char *xs = &x, *ys = &y;

    __ = __;
    __ = __;
    __ = __;
    __ = __;
    __ = __;
    __ = __;

    return y;
}
Use \texttt{uint8\_t}, \texttt{uint16\_t}, \texttt{uint32\_t}
#pragma pack(push)
struct ip {
  #ifdef LITTLE_ENDIAN
    uint8_t ihl:4, ver:4;
  #else
    uint8_t ver:4, ihl:4;
  #endif
  uint8_t tos;
  uint16_t len;
  uint16_t iid;
  uint16_t off;
  uint8_t ttl;
  uint8_t prt;
  uint16_t csm;
  uint32_t src;
  uint32_t dst;
  char opt[40];
};
#pragma pack(pop)
Serialization

```c
void foo(void) {
    struct ip *ip1, *ip2;

    ip1 = (struct ip *)malloc(sizeof(struct ip));
    ip2 = (struct ip *)malloc(sizeof(struct ip));

    ip1->ver = 4;
    ip1->protocol = 6;
    ip1->len = htonl(40);

    memcpy(ip2, ip1, sizeof(struct ip));

    printf("%d %d", ip2->ver, ntohl(ip2->len));

    ...
}
```
Use structures for data-types
Copy data in one-go
\texttt{memcpy}(dst, src, numbytes)
Use standard (big) endianness for multi-byte variables
NEVER serialize pointer values. Why?
### Compliment (flips bits)

<table>
<thead>
<tr>
<th>( \sim a )</th>
<th>( \sim 0x\text{deadbeef} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1101111010110110111111111101111</td>
<td>0x21524110</td>
</tr>
<tr>
<td>001000010101001001000000100010000</td>
<td></td>
</tr>
</tbody>
</table>

2’s compliment representation for negative numbers: 
\[-x = \sim x + 1\]