# Data Representations

## Sizes of C Objects (in Bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Compaq Alpha</th>
<th>Typical 32-bit</th>
<th>Intel IA32</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
<td>8</td>
<td>10/12</td>
</tr>
<tr>
<td>char *</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

» Or any other pointer
Byte Ordering

How should bytes within multi-byte word be ordered in memory?

Conventions

- Sun’s, Mac’s are “Big Endian” machines
  - Least significant byte has highest address
- Alphas, PC’s are “Little Endian” machines
  - Least significant byte has lowest address
Byte Ordering Example

Big Endian
- Least significant byte has highest address

Little Endian
- Least significant byte has lowest address

Example
- Variable \texttt{x} has 4-byte representation \texttt{0x01234567}
- Address given by \&\texttt{x} is \texttt{0x100}

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

<table>
<thead>
<tr>
<th>Little Endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>67</td>
<td>45</td>
<td>23</td>
<td>01</td>
</tr>
</tbody>
</table>
Reading Byte-Reversed Listings

Disassembly
- Text representation of binary machine code
- Generated by program that reads the machine code

Example Fragment

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048365:</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048366:</td>
<td>81 c3 ab 12 00 00</td>
<td>add $0x12ab, %ebx</td>
</tr>
<tr>
<td>804836c:</td>
<td>83 bb 28 00 00 00 00</td>
<td>cmpl $0x0, 0x28(%ebx)</td>
</tr>
</tbody>
</table>

Deciphering Numbers
- Value: 0x12ab
- Pad to 4 bytes: 0x000012ab
- Split into bytes: 00 00 12 ab
- Reverse: ab 12 00 00
Examining Data Representations

Code to Print Byte Representation of Data

- Casting pointer to unsigned char * creates byte array

```c
typedef unsigned char *pointer;

void show_bytes(pointer start, int len)
{
    int i;
    for (i = 0; i < len; i++)
        printf("0x%p\t0x%.2x\n", start+i, start[i]);
    printf("\n");
}
```

Printf directives:

- %p: Print pointer
- %x: Print Hexadecimal
**show_bytes Execution Example**

```c
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

**Result (Linux):**

```c
int a = 15213;
0x11ffffffcb8 0x6d
0x11ffffffcb9 0x3b
0x11ffffffcba 0x00
0x11ffffffcbb 0x00
```
Float F = 15213.0;

IEEE Single Precision Floating Point Representation

<table>
<thead>
<tr>
<th>Hex</th>
<th>4</th>
<th>6</th>
<th>6</th>
<th>D</th>
<th>B</th>
<th>4</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary:</td>
<td>0100 0110 0110 1101 1011 0100 0000 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

15213: \[1110 \ 1101 \ 1011 \ 01\]

Not same as integer representation, but consistent across machines
Can see some relation to integer representation, but not obvious
Boolean Algebra

Developed by George Boole in 19th Century

- Algebraic representation of logic
  - Encode “True” as 1 and “False” as 0

And
- \( A \& B = 1 \) when both \( A=1 \) and \( B=1 \)

\[
\begin{array}{c|cc}
& 0 & 1 \\
\hline
0 & 0 & 0 \\
1 & 0 & 1 \\
\end{array}
\]

Not
- \( \sim A = 1 \) when \( A=0 \)

\[
\begin{array}{c|c}
\sim & 1 \\
\hline
0 & 1 \\
1 & 0 \\
\end{array}
\]

Or
- \( A \mid B = 1 \) when either \( A=1 \) or \( B=1 \)

\[
\begin{array}{c|cc}
& 0 & 1 \\
\hline
0 & 0 & 1 \\
1 & 1 & 1 \\
\end{array}
\]

Exclusive-Or (Xor)
- \( A \wedge B = 1 \) when either \( A=1 \) or \( B=1 \), but not both

\[
\begin{array}{c|cc}
\wedge & 0 & 1 \\
\hline
0 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]
Cool Stuff with Xor

- Bitwise Xor is form of addition
- With extra property that every value is its own additive inverse
  \[ A \oplus A = 0 \]

```c
void funny(int *x, int *y)
{
    *x = *x ^ *y;    /* #1 */
    *y = *x ^ *y;    /* #2 */
    *x = *x ^ *y;    /* #3 */
}
```

<table>
<thead>
<tr>
<th>Begin</th>
<th>( *x )</th>
<th>( *y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>( A \oplus B )</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>( A \oplus B )</td>
<td>( (A \oplus B) \oplus B = A )</td>
</tr>
<tr>
<td>3</td>
<td>( (A \oplus B) \oplus A = B )</td>
<td>A</td>
</tr>
</tbody>
</table>

| End     | B        | A        |
Bit-Level Operations in C

Operations &, |, ~, ^ Available in C

- Apply to any “integral” data type
  - long, int, short, char
- View arguments as bit vectors
- Arguments applied bit-wise

Examples (Char data type)

- ~0x41 --> 0xBE
  ~01000001₂ --> 10111110₂
- ~0x00 --> 0xFF
  ~00000000₂ --> 11111111₂
- 0x69 & 0x55 --> 0x41
  01101001₂ & 01010101₂ --> 01000001₂
- 0x69 | 0x55 --> 0x7D
  01101001₂ | 01010101₂ --> 01111101₂
Contrast: Logic Operations in C

Contrast to Logical Operators

- &&, ||, !
  - View 0 as “False”
  - Anything nonzero as “True”
  - Always return 0 or 1
  - Early termination

Examples (char data type)

- !0x41 --> 0x00
- !0x00 --> 0x01
- !!0x41 --> 0x01

- 0x69 && 0x55 --> 0x01
- 0x69 || 0x55 --> 0x01
- p && *p (avoids null pointer access)
Shift Operations

Left Shift: \( x << y \)
- Shift bit-vector \( x \) left \( y \) positions
  - Throw away extra bits on left
  - Fill with 0’s on right

Right Shift: \( x >> y \)
- Shift bit-vector \( x \) right \( y \) positions
  - Throw away extra bits on right
- Logical shift
  - Fill with 0’s on left
- Arithmetic shift
  - Replicate most significant bit on right
  - Useful with two’s complement integer representation

<table>
<thead>
<tr>
<th>Argument</th>
<th>01100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt; 3</td>
<td>00010000</td>
</tr>
<tr>
<td>Log. &gt;&gt; 2</td>
<td>00011000</td>
</tr>
<tr>
<td>Arith. &gt;&gt; 2</td>
<td>00011000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument</th>
<th>10100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt; 3</td>
<td>00010000</td>
</tr>
<tr>
<td>Log. &gt;&gt; 2</td>
<td>00101000</td>
</tr>
<tr>
<td>Arith. &gt;&gt; 2</td>
<td>11101000</td>
</tr>
</tbody>
</table>
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
- Nonnegative values unchanged
  - \( ux = 15213 \)
- Negative values change into (large) positive values
  - \( uy = 50323 \)
Signed vs. Unsigned in C

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

**Casting**
- Explicit casting between signed & unsigned same as U2T and T2U
  
  ```c
  int tx, ty;
  unsigned ux, uy;
  tx = (int) ux;
  uy = (unsigned) ty;
  ```
- Implicit casting also occurs via assignments and procedure calls
  
  ```c
  tx = ux;
  uy = ty;
  ```
Casting Surprises

Expression Evaluation

- If mix unsigned and signed in single expression, signed values implicitly cast to unsigned
- Including comparison operations `<`, `>`, `==`, `<=`, `>=`
- Examples for $W = 32$

<table>
<thead>
<tr>
<th>$\text{Constant}_1$</th>
<th>$\text{Constant}_2$</th>
<th>Relation</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 $\text{U}$</td>
<td><code>==</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td><code>&lt;</code></td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>0 $\text{U}$</td>
<td><code>&gt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>-2147483648</td>
<td><code>&lt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647 $\text{U}$</td>
<td>-2147483648</td>
<td><code>&gt;</code></td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>-2 $\text{U}$</td>
<td><code>&gt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>(unsigned) -1</td>
<td>-2</td>
<td><code>&lt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>2147483648 $\text{U}$</td>
<td><code>&lt;</code></td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 2147483648 $\text{U}$</td>
<td><code>&gt;</code></td>
<td>signed</td>
</tr>
</tbody>
</table>
Explanation of Casting Surprises

2’s Comp. → Unsigned

- Ordering Inversion
- Negative → Big Positive
Power-of-2 Multiply with Shift

Operation

- \( u \ll k \) gives \( u \times 2^k \)
- Both signed and unsigned

Examples

- \( u \ll 3 \) == \( u \times 8 \)
- \( u \ll 5 \) - \( u \ll 3 \) == \( u \times 24 \)
- Most machines shift and add much faster than multiply
  - Compiler generates this code automatically
Unsigned Power-of-2 Divide with Shift

Quotient of Unsigned by Power of 2

- $u >> k$ gives $\lfloor u / 2^k \rfloor$
- Uses logical shift

Operands:

<table>
<thead>
<tr>
<th>Division</th>
<th>Computed</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>15213</td>
<td>3B 6D</td>
<td>00111011 01101101</td>
</tr>
<tr>
<td>$x &gt;&gt; 1$</td>
<td>7606.5</td>
<td>1D B6</td>
<td>00011101 10110110</td>
</tr>
<tr>
<td>$x &gt;&gt; 4$</td>
<td>950.8125</td>
<td>03 B6</td>
<td>00000011 10110110</td>
</tr>
<tr>
<td>$x &gt;&gt; 8$</td>
<td>59.4257813</td>
<td>00 3B</td>
<td>00000000 00111011</td>
</tr>
</tbody>
</table>

Division:

\[
\begin{array}{c}
\underbrace{u}^{k} \\
/ \quad 2^k \\
\overline{\underline{u / 2^k}} \\
\end{array}
\]
Fractional Binary Numbers

Representation
- Bits to right of “binary point” represent fractional powers of 2
- Represents rational number:

\[ \sum_{k = -j}^{i} b_k \cdot 2^k \]
Representable Numbers

Limitation

- Can only exactly represent numbers of the form $x/2^k$
- Other numbers have repeating bit representations

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/3</td>
<td>0.0101010101[01]…₂</td>
</tr>
<tr>
<td>1/5</td>
<td>0.001100110011[0011]…₂</td>
</tr>
<tr>
<td>1/10</td>
<td>0.0001100110011[0011]…₂</td>
</tr>
</tbody>
</table>
Floating Point Representation

Numerical Form

- $-1^s M \cdot 2^E$
  - Sign bit $s$ determines whether number is negative or positive
  - Significand $M$ normally a fractional value in range $[1.0, 2.0)$
  - Exponent $E$ weights value by power of two

Encoding

- MSB is sign bit
- $exp$ field encodes $E$
- $frac$ field encodes $M$
Floating Point Precisions

Encoding

- MSB is sign bit
- \text{exp} field encodes $E$
- \text{frac} field encodes $M$

Sizes

- Single precision: 8 \exp\ bits, 23 \frac\ bits
  - 32 bits total
- Double precision: 11 \exp\ bits, 52 \frac\ bits
  - 64 bits total
- Extended precision: 15 \exp\ bits, 63 \frac\ bits
  - Only found in Intel-compatible machines
  - Stored in 80 bits
    - 1 bit wasted
Special Values

Condition

- $\exp = 111...1$

Cases

- $\exp = 111...1, \frac{}{\neq} 000...0$
  - Represents value $\infty$ (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., $1.0/0.0 = -1.0/-0.0 = +\infty, \quad 1.0/-0.0 = -\infty$

- $\exp = 111...1, \frac{}{\neq} 000...0$
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., $\sqrt{-1}, \infty - \infty$