Representing Data with Bits

bits, bytes, numbers, and notation
### Positional Number Representation

Base = 10 (decimal)

<table>
<thead>
<tr>
<th>Position</th>
<th>Weight</th>
<th>Digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>10^2</td>
<td>10^1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>10^2</td>
<td>10^1</td>
<td>10^0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Base determines:

- Maximum digit (base – 1). Minimum digit is 0.
- Weight of each position.
- Each position holds a digit.

Represented value = sum of all position values

\[
\text{position value} = \text{digit value} \times \text{base}^{\text{position}}
\]

\[
2 \times 10^2 + 4 \times 10^1 + 0 \times 10^0 = 240
\]
Binary digits are called **bits**: 0, 1

When ambiguous, subscript with base:

101$_{10}$ Dalmatians (movie)

101$_2$-Second Rule (folk wisdom for food safety)
### Powers of 2:
memorize up to $2^{10}$ (in base ten)

<table>
<thead>
<tr>
<th>Power: $2^?$</th>
<th>Decimal value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>
conversion from binary to decimal

1 0 1 1 0 1 \_2 = \_10

Interpret the positional representation according to the base: sum the place weights where 1 appears (in either direction).
conversion from decimal to binary

\[ 19_{10} = ?_2 \]

<table>
<thead>
<tr>
<th>Divide-by-2 Approach (Right to Left)</th>
<th>Powers-of-2 Approach (Left to Right)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quotient</td>
<td>Value</td>
</tr>
<tr>
<td>Remainder?</td>
<td>Power that fits?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
binary arithmetic

110_2 + 1011_2 = ?_2  

1101_2 − 1011_2 = ?_2

1001011_2 × 2_{10} = ?_2
conversion and arithmetic

$19_{10} = ?_2$

$1001_2 = ?_{10}$

$240_{10} = ?_2$

$11010011_2 = ?_{10}$

$101_2 + 1011_2 = ?_2$

$101011_2 \times 2_{10} = ?_2$
**byte = 8 bits**

a.k.a. octet

**Smallest unit of data**
_used by a typical modern computer_

**Binary** 00000000\(_2\) -- 11111111\(_2\)

**Decimal** 000\(_{10}\) -- 255\(_{10}\)

**Hexadecimal** 00\(_{16}\) -- FF\(_{16}\)

**Byte = 2 hex digits!**

Programmer’s hex notation (C, etc.):

0xB4\(=\) B4\(_{16}\)

Octal (base 8) also useful.
A C-style string is represented by a series of bytes (**chars**).

- One-byte **ASCII** codes for each character.
- **ASCII** = American Standard Code for Information Interchange

<table>
<thead>
<tr>
<th>char</th>
<th>decimal</th>
<th>ASCII code</th>
</tr>
</thead>
<tbody>
<tr>
<td>space</td>
<td>32</td>
<td>01000010</td>
</tr>
<tr>
<td>!</td>
<td>33</td>
<td>01000011</td>
</tr>
<tr>
<td>&quot;</td>
<td>34</td>
<td>01000100</td>
</tr>
<tr>
<td>#</td>
<td>35</td>
<td>01000101</td>
</tr>
<tr>
<td>$</td>
<td>36</td>
<td>01000110</td>
</tr>
<tr>
<td>%</td>
<td>37</td>
<td>01000111</td>
</tr>
<tr>
<td>&amp;</td>
<td>38</td>
<td>01001000</td>
</tr>
<tr>
<td>(</td>
<td>39</td>
<td>01001001</td>
</tr>
<tr>
<td>)</td>
<td>40</td>
<td>01001010</td>
</tr>
<tr>
<td>*</td>
<td>41</td>
<td>01001011</td>
</tr>
<tr>
<td>+</td>
<td>42</td>
<td>01001100</td>
</tr>
<tr>
<td>,</td>
<td>43</td>
<td>01001101</td>
</tr>
<tr>
<td>&lt;</td>
<td>44</td>
<td>01001110</td>
</tr>
<tr>
<td>=</td>
<td>45</td>
<td>01001111</td>
</tr>
<tr>
<td>&gt;</td>
<td>46</td>
<td>01000000</td>
</tr>
<tr>
<td>/</td>
<td>47</td>
<td>01000001</td>
</tr>
</tbody>
</table>
**word** /wərd/, n.

Natural unit of data used by processor.

**Fixed size** (e.g. 32 bits, 64 bits)

Defined by ISA: Instruction Set Architecture

machine instruction operands

word size = register size = address size

Java/C int = 4 bytes: 11,501,584

MSB: most significant bit

LSB: least significant bit
### fixed-size data representations

(size in bytes)

<table>
<thead>
<tr>
<th>Java Data Type</th>
<th>C Data Type</th>
<th>[word = 32 bits]</th>
<th>[word = 64 bits]</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>char</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>short</td>
<td>short int</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>long int</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long</td>
<td>long long</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>long double</td>
<td>8</td>
<td>16</td>
</tr>
</tbody>
</table>

Depends on word size!
**bitwise operators**

**Bitwise operators** on fixed-width **bit vectors**.

- **AND &**
- **OR |**
- **XOR ^**
- **NOT ~**

Laws of Boolean algebra apply bitwise.

*e.g.*, DeMorgan’s Law: \( \sim(A \lor B) = \sim A \land \sim B \)
**bitwise operators in C**

\&  |  ^  ~  

apply to any *integral* data type

long, int, short, char, unsigned

Examples (char)

\~0x41 =

\~0x00 =

0x69 \& 0x55 =

0x69  |  0x55 =

Many bit-twiddling puzzles in upcoming assignment
Preparation (videos) understanding check 1: what is the value `0b0110` converted to decimal?

2

3

5

6

None of the above
Preparation (videos) understanding check 2: what is result of the bitwise not operation on the binary value `0b0110`: `~0b0110`?

- 0b0110
- 0b1001
- 0b0111
- 0b1111
- None of the above
Preparation (videos) understanding check 3: what is the binary number `0b1010` in hexadecimal?

- 0x1010
- 0x10
- 0xA
- 0xF
- None of the above
# basics of C (vs. Java)

## Similar

<table>
<thead>
<tr>
<th>Interaction</th>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable declaration;</td>
<td>compile, then run</td>
<td>int a = 5;</td>
</tr>
<tr>
<td>arithmetic</td>
<td></td>
<td>a = a * 5;</td>
</tr>
<tr>
<td>Loops</td>
<td>for (int i = 0; i &lt; N; i++) {...}</td>
<td></td>
</tr>
</tbody>
</table>

## Different

<table>
<thead>
<tr>
<th>Data Management</th>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Management</td>
<td>Objects/references</td>
<td>Pointers/structs/allocated memory</td>
</tr>
<tr>
<td></td>
<td>Garbage collection</td>
<td>Manual memory management</td>
</tr>
<tr>
<td>Functions</td>
<td>public static int max(int a, int b)</td>
<td>int max(int a, int b)</td>
</tr>
<tr>
<td>Main, printing</td>
<td></td>
<td>#include&lt;stdio.h&gt;</td>
</tr>
<tr>
<td></td>
<td>public class HelloWorld {</td>
<td>int main(void) {</td>
</tr>
<tr>
<td></td>
<td>public static void main(String[] args) {</td>
<td></td>
</tr>
<tr>
<td></td>
<td>System.out.println(&quot;Hello&quot;);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>printf(&quot;Hello\n&quot;);</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>return 0;</td>
</tr>
</tbody>
</table>
Representation Example 1:
Sets as Bit Vectors

**Representation:** \(n\)-bit vector gives subset of \(\{0, ..., n-1\}\).

\[ a_i = 1 \equiv i \in A \]

\[ a = 0b01101001 \quad A = \{0, 3, 5, 6\} \]

\[ b = 0b01010101 \quad B = \{0, 2, 4, 6\} \]

**Bitwise Operations**
- \(a \& b = 0b01000001 \{0, 6\}\)
- \(a \mid b = 0b01111101 \{0, 2, 3, 4, 5, 6\}\)
- \(a \ ^{\text{^}} b = 0b00111100 \{2, 3, 4, 5\}\)
- \(\sim b = 0b10101010 \{1, 3, 5, 7\}\)

**Set Operations**
- Intersection
- Union
- Symmetric difference
- Complement
Logical operations in C

&&  ||  ! apply to any "integral" data type
long, int, short, char, unsigned

0 is false  nonzero is true  result always 0 or 1

Early termination  a.k.a.  short-circuit evaluation

Examples (char)
!0x41 =
!0x00 =
!!0x41 =

0x69 && 0x55 =
0x69 || 0x55 =
Representation Example 2: Playing Cards

52 cards in 4 suits

How do we encode suits, face cards?

What operations should be easy to implement?

Get and compare rank
Get and compare suit
Two possible representations

52 cards – 52 bits with bit corresponding to card set to 1

“One-hot” encoding

Hard to compare values and suits independently
Not space efficient

4 bits for suit, 13 bits for card value – 17 bits with two set to 1

Pair of one-hot encoded values

Easier to compare suits and values independently
Smaller, but still not space efficient
Two better representations

Binary encoding of all 52 cards – only 6 bits needed
Number cards uniquely from 0
Smaller than one-hot encodings.
Hard to compare value and suit

Binary encoding of suit (2 bits) and value (4 bits) separately
Number each suit uniquely
Number each value uniquely
Still small
Easy suit, value comparisons
Compare Card **Suits**

**mask:** a bit vector that, when bitwise ANDed with another bit vector \( v \), turns all *but* the bits of interest in \( v \) to 0

```cpp
#define SUIT_MASK 0x30
```

```cpp
int sameSuit(char card1, char card2) {
    return !((card1 & SUIT_MASK) ^ (card2 & SUIT_MASK));

    // same as (card1 & SUIT_MASK) == (card2 & SUIT_MASK);
}
```

```cpp
char hand[5]; // represents a 5-card hand
...
if (sameSuit(hand[0], hand[1])) { ... }
```
Compare Card **Values**

**mask**: a bit vector that, when bitwise ANDed with another bit vector $v$, turns all *but* the bits of interest in $v$ to 0.

```c
#define VALUE_MASK

int greaterValue(char card1, char card2) {

}
```

```c
char hand[5];  // represents a 5-card hand
...
if (greaterValue(hand[0], hand[1])) { ... }
```
Bit shifting

**Logical shift left 2**

$x \ll 2$

$\begin{array}{c}
1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
\end{array}$

$\rightarrow$

$\begin{array}{c}
0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
\end{array}$

- Lose bits on left
- Fill with zeroes on right

**Logical shift right 2**

$x \gg 2$

$\begin{array}{c}
1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
\end{array}$

$\rightarrow$

$\begin{array}{c}
0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
\end{array}$

- Fill with zeroes on left
- Lose bits on right

**Arithmetic shift right 2**

$x \gg 2$

$\begin{array}{c}
1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
\end{array}$

$\rightarrow$

$\begin{array}{c}
1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
\end{array}$

- Fill with copies of MSB on left
- Lose bits on right
Shift gotchas

*Logical or arithmetic shift right: how do we tell?*

C: compiler chooses  
   Usually based on type: rain check!  
Java: >> is arithmetic, >>> is logical

Shift an \( n \)-bit type by at least 0 and no more than \((n-1)\).

C: other shift distances are undefined.  
   *anything* could happen  
Java: shift distance is used modulo number of bits in shifted type

*Given* \( \text{int } x; \quad x << 34 == x << 2 \)
Shift and mask: extract a bit field

Write a C function that extracts the 2\textsuperscript{nd} most significant byte from its 32-bit integer argument.

Example behavior:
argument: \texttt{0b 01100001 01100010 01100011 01100100}
expected result: \texttt{0b 00000000 00000000 00000000 01100010}

All other bits are zero. Desired bits in least significant byte.

```c
int get2ndMSB(int x) {
}
```