CS107, Lecture 2
Bits and Bytes; Integer Representations

reading:
Bryant & O’Hallaron, Ch. 2.2-2.3
Plan For Today

• Bits and Bytes
• Hexadecimal
• Integer Representations
• Unsigned Integers
• Signed Integers
• Casting and Combining Types
Demo: Unexpected Behavior
Plan For Today

• Bits and Bytes
• Hexadecimal
• Integer Representations
• Unsigned Integers
• Signed Integers
• Break: Announcements
• Casting and Combining Types
• Computers are built around the idea of two states: “on” and “off”. Transistors represent this in hardware, and bits represent this in software!
• We can combine bits, like with base-10 numbers, to represent more data. \(8\) \textbf{bits} = \textbf{1 byte}.

• Computer memory is just a large array of bytes! It is \textit{byte-addressable}; you can’t address (store location of) a bit; only a byte.

• Computers still fundamentally operate on bits; we have just gotten more creative about how to represent different data as bits!
  
  • Images
  • Audio
  • Video
  • Text
  • And more...
Base 10

5 9 3 4

Digits 0-9 (0 to base-1)
Base 10

5 9 3 4

thousands  hundreds  tens  ones
Base 10

5 9 3 4

= 5*1000 + 9*100 + 3*10 + 4*1
Base 10

5 \times 10^3 + 9 \times 10^2 + 3 \times 10^1 + 4 \times 10^0
Base 10

5 9 3 4

10^x: 3 2 1 0
Base 2

Digits 0-1 (0 to base-1)
Base 2

1 0 1 1

\[2^3 \quad 2^2 \quad 2^1 \quad 2^0\]
Base 2

1 0 1 1

eights fours twos ones

= 1*8 + 0*4 + 1*2 + 1*1 = 11_{10}
Base 2

Most significant bit (MSB)  
Least significant bit (LSB)

1 0 1 1

Eights  Fours  Twos  Ones

= 1*8 + 0*4 + 1*2 + 1*1 = 11_{10}
Question: What is 6 in base 2?

• Strategy:
  • What is the largest power of 2 ≤ 6?
**Base 10 to Base 2**

**Question:** What is 6 in base 2?

- **Strategy:**
  - What is the largest power of 2 ≤ 6? \(2^2 = 4\)

```
  0 1
  2^3 2^2 2^1 2^0
```
Question: What is 6 in base 2?

• Strategy:
  • What is the largest power of 2 ≤ 6? $2^2 = 4$
  • Now, what is the largest power of 2 ≤ 6 – $2^2$?
Question: What is 6 in base 2?

• Strategy:
  • What is the largest power of 2 ≤ 6? $2^2 = 4$
  • Now, what is the largest power of 2 ≤ 6 − 2^2? $2^1 = 2$
**Question:** What is 6 in base 2?

**Strategy:**
- What is the largest power of 2 ≤ 6? $2^2 = 4$
- Now, what is the largest power of 2 ≤ $6 - 2^2$? $2^1 = 2$
- $6 - 2^2 - 2^1 = 0!$
Question: What is 6 in base 2?

• Strategy:
  • What is the largest power of 2 ≤ 6? $2^2 = 4$
  • Now, what is the largest power of 2 ≤ 6 $– 2^2$? $2^1 = 2$
  • $6 – 2^2 – 2^1 = 0!$
**Question:** What is 6 in base 2?

- **Strategy:**
  - What is the largest power of 2 ≤ 6? \(2^2 = 4\)
  - Now, what is the largest power of 2 ≤ 6 – 2\(^2\)? \(2^1 = 2\)
  - \(6 - 2^2 - 2^1 = 0!\)

0 \(\underline{2^3}\) 1 \(\underline{2^2}\) 1 \(\underline{2^1}\) 0 \(\underline{2^0}\)

\[= 0 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1 = 6\]
What is the base-10 representation of $1010_2$?

a) 20
b) 101
c) 10
d) 5
e) Other
What is the base-2 representation of 14?

a) $1111_2$

b) $1110_2$

c) $1010_2$

d) Other
Practice: Byte Values

• What is the minimum and maximum base-10 value a single byte (8 bits) can store?
Practice: Byte Values

• What is the minimum and maximum base-10 value a single byte (8 bits) can store?  
  minimum = 0  maximum = ?
Practice: Byte Values

• What is the minimum and maximum base-10 value a single byte (8 bits) can store?  
  \[ \text{minimum} = 0 \quad \text{maximum} = ? \]

\[ \begin{array}{ccccccc}
  1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array} \]

\[ 2^x: \begin{array}{ccccccc}
  7 & 6 & 5 & 4 & 3 & 2 & 1 \end{array} \]
• What is the minimum and maximum base-10 value a single byte (8 bits) can store?  
  minimum = 0  
  maximum = ?

• Strategy 1: \(1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 255\)
Practice: Byte Values

- What is the minimum and maximum base-10 value a single byte (8 bits) can store?  
  - minimum = 0  
  - maximum = 255

- **Strategy 1:**  
  \[ 1\times2^7 + 1\times2^6 + 1\times2^5 + 1\times2^4 + 1\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0 = 255 \]

- **Strategy 2:**  
  \[ 2^8 - 1 = 255 \]
Multiplying by Base

1453 \times 10 = 14530

1101_2 \times 2 = 11010

*Key Idea*: inserting 0 at the end multiplies by the base!
Plan For Today

• Bits and Bytes
• Hexadecimal
• Integer Representations
• Unsigned Integers
• Signed Integers
• Break: Announcements
• Casting and Combining Types
• When working with bits, oftentimes we have large numbers with 32 or 64 bits.
• Instead, we’ll represent bits in base-16 instead; this is called hexadecimal.

0110 1010 0011
Hexadecimal

- When working with bits, oftentimes we have large numbers with 32 or 64 bits.
- Instead, we’ll represent bits in base-16 instead; this is called hexadecimal.

0110 1010 0011

0-15 0-15 0-15
Hexadecimal

• When working with bits, oftentimes we have large numbers with 32 or 64 bits.
• Instead, we’ll represent bits in *base-16 instead*; this is called *hexadecimal*.

This is a base-16 number!
Hexadecimal

- Hexadecimal is base-16, so we need digits for 1-15. How do we do this?

0 1 2 3 4 5 6 7 8 9 a b c d e f
10 11 12 13 14 15
## Hexadecimal

<table>
<thead>
<tr>
<th>Hex digit</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal value</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Binary value</td>
<td>0000</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
<td>0100</td>
<td>0101</td>
<td>0110</td>
<td>0111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hex digit</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal value</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>Binary value</td>
<td>1000</td>
<td>1001</td>
<td>1010</td>
<td>1011</td>
<td>1100</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
</tr>
</tbody>
</table>
Hexadecimal

• In C, we commonly distinguish hexadecimal numbers by prefixing them with \texttt{0x}, and binary numbers by prefixing them with \texttt{0b}.
• E.g. \texttt{0xf5} is \texttt{0b11110101}

\begin{center}
\begin{tabular}{c}
\texttt{0x f 5} \\
1111 0101
\end{tabular}
\end{center}
Practice: Hexadecimal to Binary

What is $0x173A$ in binary?
Practice: Hexadecimal to Binary

What is \textbf{0x173A} in binary?

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>1</th>
<th>7</th>
<th>3</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>0001</td>
<td>0111</td>
<td>0011</td>
<td>1010</td>
</tr>
</tbody>
</table>
What is 0b1111001010110110110011 in hexadecimal? *(Hint: start from the right)*
What is \texttt{0b1111001010110110110011} in hexadecimal? (\textit{Hint: start from the right})

<table>
<thead>
<tr>
<th>Binary</th>
<th>11</th>
<th>1100</th>
<th>1010</th>
<th>1101</th>
<th>1011</th>
<th>0011</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hexadecimal</td>
<td>3</td>
<td>C</td>
<td>A</td>
<td>D</td>
<td>B</td>
<td>3</td>
</tr>
</tbody>
</table>
Plan For Today

- Bits and Bytes
- Hexadecimal
- **Integer Representations**
- Unsigned Integers
- Signed Integers
- **Break**: Announcements
- Casting and Combining Types
Number Representations

- **Unsigned Integers**: positive and 0 integers. (e.g. 0, 1, 2, ... 99999...)
- **Signed Integers**: negative, positive and 0 integers. (e.g. ...-2, -1, 0, 1,... 9999...)
- **Floating Point Numbers**: real numbers. (e.g. 0.1, -12.2, 1.5x10^{12})
Number Representations

• **Unsigned Integers**: positive and 0 integers. (e.g. 0, 1, 2, ... 99999...

• **Signed Integers**: negative, positive and 0 integers. (e.g. ...-2, -1, 0, 1,... 9999...)

• **Floating Point Numbers**: real numbers. (e,g. 0.1, -12.2, 1.5x10^{12})

Stay tuned until week 5!
• In the early 2000’s, most computers were **32-bit**. This means that pointers in programs were **32 bits**.

• 32-bit pointers could store a memory address from 0 to $2^{32}-1$, for a total of $2^{32}$ **bytes of addressable memory**. This equals **4 Gigabytes**, meaning that 32-bit computers could have at most **4GB** of memory (RAM)!

• Because of this, computers transitioned to **64-bit**. This means that pointers in programs were **64 bits**.

• 64-bit pointers could store a memory address from 0 to $2^{64}-1$, for a total of $2^{64}$ **bytes of addressable memory**. This equals **16 Exabytes**, meaning that 64-bit computers could have at most **1024*1024*1024 GB** of memory (RAM)!
### Number Representations

<table>
<thead>
<tr>
<th>C declaration</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed</td>
<td>Signed char</td>
</tr>
<tr>
<td>[signed] char</td>
<td>unsigned char</td>
</tr>
<tr>
<td>short</td>
<td>unsigned short</td>
</tr>
<tr>
<td>int</td>
<td>unsigned</td>
</tr>
<tr>
<td>long</td>
<td>unsigned long</td>
</tr>
<tr>
<td>int32_t</td>
<td>uint32_t</td>
</tr>
<tr>
<td>int64_t</td>
<td>uint64_t</td>
</tr>
<tr>
<td>char *</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td></td>
</tr>
</tbody>
</table>
## Number Representations

<table>
<thead>
<tr>
<th>C declaration</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed</td>
<td>Unsigned</td>
</tr>
<tr>
<td>[signed] char</td>
<td>unsigned char</td>
</tr>
<tr>
<td>short</td>
<td>unsigned short</td>
</tr>
<tr>
<td>int</td>
<td>unsigned</td>
</tr>
<tr>
<td>long</td>
<td>unsigned long</td>
</tr>
<tr>
<td>int32_t</td>
<td>uint32_t</td>
</tr>
<tr>
<td>int64_t</td>
<td>uint64_t</td>
</tr>
<tr>
<td>char *</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td></td>
</tr>
</tbody>
</table>
Plan For Today

• Bits and Bytes
• Hexadecimal
• Integer Representations
• **Unsigned Integers**
• Signed Integers
• **Break**: Announcements
• Casting and Combining Types
Unsigned Integers

• An **unsigned** integer is 0 or a positive integer (no negatives).
• We have already discussed converting between decimal and binary, which is a nice 1:1 relationship. Examples:
  
  0b0001 = 1
  0b0101 = 5
  0b1011 = 11
  0b1111 = 15

• The range of an unsigned number is $0 \rightarrow 2^w - 1$, where $w$ is the number of bits. E.g. a 32-bit integer can represent 0 to $2^{32} - 1$ (4,294,967,295).
Unsigned Integers

4-bit unsigned integer representation
Plan For Today

• Bits and Bytes
• Hexadecimal
• Integer Representations
• Unsigned Integers
• Signed Integers
• Break: Announcements
• Casting and Combining Types
Signed Integers

• An **signed** integer is a negative integer, 0, or a positive integer.
• *Problem*: How can we represent negative *and* positive numbers in binary?
Signed Integers

• An **signed** integer is a negative integer, 0, or a positive integer.
• *Problem:* How can we represent negative *and* positive numbers in binary?

**Idea:** let’s reserve the *most significant bit* to store the sign.
Sign Magnitude Representation

0110
positive  6

1011
negative  3
Sign Magnitude Representation

0000
positive 0

1000
negative 0
Sign Magnitude Representation

1 000 = -0 0 000 = 0
1 001 = -1 0 001 = 1
1 010 = -2 0 010 = 2
1 011 = -3 0 011 = 3
1 100 = -4 0 100 = 4
1 101 = -5 0 101 = 5
1 110 = -6 0 110 = 6
1 111 = -7 0 111 = 7

• We’ve only represented 15 of our 16 available numbers!
Sign Magnitude Representation

- **Pro:** easy to represent, and easy to convert to/from decimal.
- **Con:** +-0 is not intuitive
- **Con:** we lose a bit that could be used to store more numbers
- **Con:** arithmetic is tricky: we need to find the sign, then maybe subtract (borrow and carry, etc.), then maybe change the sign...this might get ugly!
A Better Idea

• Ideally, binary addition would *just work regardless* of whether the number is positive or negative.

```
  0101
+ ????
  0000
```
Ideally, binary addition would *just work regardless* of whether the number is positive or negative.

\[
\begin{array}{c}
0101 \\
+1011 \\
\hline
0000
\end{array}
\]
• Ideally, binary addition would *just work regardless* of whether the number is positive or negative.
A Better Idea

• Ideally, binary addition would *just work regardless* of whether the number is positive or negative.

\[
\begin{array}{c}
0011 \\
+1101 \\
\hline
0000
\end{array}
\]
• Ideally, binary addition would *just work regardless* of whether the number is positive or negative.

```
0000
+????
-----
00000
```
Ideally, binary addition would *just work regardless* of whether the number is positive or negative.
# A Better Idea

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Positive</th>
<th>Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1111</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>1110</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>1101</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>1100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>1011</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>1010</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>1001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Positive</th>
<th>Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>1001 (same as -7!)</td>
<td>NA</td>
</tr>
<tr>
<td>10</td>
<td>1010 (same as -6!)</td>
<td>NA</td>
</tr>
<tr>
<td>11</td>
<td>1011 (same as -5!)</td>
<td>NA</td>
</tr>
<tr>
<td>12</td>
<td>1100 (same as -4!)</td>
<td>NA</td>
</tr>
<tr>
<td>13</td>
<td>1101 (same as -3!)</td>
<td>NA</td>
</tr>
<tr>
<td>14</td>
<td>1110 (same as -2!)</td>
<td>NA</td>
</tr>
<tr>
<td>15</td>
<td>1111 (same as -1!)</td>
<td>NA</td>
</tr>
</tbody>
</table>
There Seems Like a Pattern Here...

- The negative number is the positive number inverted, plus one!
There Seems Like a Pattern Here…

A binary number plus its inverse is all 1s.

\[
\begin{array}{c}
0101 \\
+1010 \\
\hline
1111
\end{array}
\]

Add 1 to this to carry over all 1s and get 0!

\[
\begin{array}{c}
1111 \\
+0001 \\
\hline
0000
\end{array}
\]
Another Trick

• To find the negative equivalent of a number, work right-to-left and write down all digits *through* when you reach a 1. Then, invert the rest of the digits.

\[
\begin{align*}
100100 & \\
+?????? & \\
\hline
000000 & 
\end{align*}
\]
Another Trick

• To find the negative equivalent of a number, work right-to-left and write down all digits through when you reach a 1. Then, invert the rest of the digits.

\[
\begin{array}{c}
100100 \\
+ \text{????100} \\
\hline \\
0000000
\end{array}
\]
Another Trick

• To find the negative equivalent of a number, work right-to-left and write down all digits *through* when you reach a 1. Then, invert the rest of the digits.

\[ \begin{array}{c}
100100 \\
+ 011100 \\
\hline
0000000
\end{array} \]
Two’s Complement

4-bit two's complement signed integer representation
Two’s Complement

- In **two’s complement**, we represent a positive number as itself, and its negative equivalent as the **two’s complement of itself**.
- The **two’s complement** of a number is the binary digits inverted, plus 1.
- This works to convert from positive to negative, and back from negative to positive!
Two’s Complement

- **Con:** more difficult to represent, and difficult to convert to/from decimal and between positive and negative.
- **Pro:** only 1 representation for 0!
- **Pro:** all bits are used to represent as many numbers as possible
- **Pro:** it turns out that the most significant bit still indicates the sign of a number.
- **Pro:** arithmetic is easy: we just add!
Two’s Complement

- Adding two numbers is just...adding! There is no special case needed for negatives. E.g. what is 2 + -5?

\[
\begin{align*}
0010 & \quad 2 \\
+1011 & \quad -5 \\
\hline
1101 & \quad -3
\end{align*}
\]
Two’s Complement

- Subtracting two numbers is just performing the two’s complement on one of them and then adding. E.g. $4 - 5 = -1$. 

\[
\begin{array}{cccccc}
0100 & 4 & 0100 & 4 \\
-0101 & & +1011 & \\
\hline
1111 & & \text{-1} & \\
\end{array}
\]
Two’s Complement

• Multiplying two numbers is just multiplying, and discarding overflow digits. E.g. -2 x -3 = 6.

\[
\begin{array}{c}
1110 \text{ (-2)} \\
x1101 \text{ (-3)} \\
1110 \\
0000 \\
1110 \\
+1110 \\
10110110 \text{ (6)}
\end{array}
\]
Practice: Two’s Complement

What are the negative or positive equivalents of the numbers below?

a) -4 (1100)
b) 7 (0111)
c) 3 (0011)
d) -8 (1000)
Practice: Two’s Complement

What are the negative or positive equivalents of the numbers below?

a) -4 (1100)
b) 7 (0111)
c) 3 (0011)
d) -8 (1000)
Plan For Today

• Bits and Bytes
• Hexadecimal
• Integer Representations
• Unsigned Integers
• Signed Integers

• Break: Announcements
• Casting and Combining Types
Announcements

• Sign up for Piazza on the Help page if you haven’t already!
• Assign0 released earlier this week, due Mon.
• Lab signups opened earlier this week, start next week.
• Please send course staff OAE letters for accommodations!
Plan For Today

- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- Signed Integers
- Break: Announcements
- Casting and Combining Types
Overflow and Underflow

• If you exceed the **maximum** value of your bit representation, you *wrap around* or *overflow* back to the **smallest** bit representation.

\[ 0b1111 + 0b1 = 0b0000 \]

• If you go below the **minimum** value of your bit representation, you *wrap around* or *underflow* back to the **largest** bit representation.

\[ 0b0000 - 0b1 = 0b1111 \]
<table>
<thead>
<tr>
<th>Type</th>
<th>Width (bytes)</th>
<th>Width (bits)</th>
<th>Min in hex (name)</th>
<th>Max in hex (name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>8</td>
<td>80 (CHAR_MIN)</td>
<td>7F (CHAR_MAX)</td>
</tr>
<tr>
<td>unsigned char</td>
<td>1</td>
<td>8</td>
<td>0</td>
<td>FF (UCHAR_MAX)</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>16</td>
<td>8000 (SHRT_MIN)</td>
<td>7FFF (SHRT_MAX)</td>
</tr>
<tr>
<td>unsigned short</td>
<td>2</td>
<td>16</td>
<td>0</td>
<td>FFFF (USHRT_MAX)</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>32</td>
<td>80000000 (INT_MIN)</td>
<td>7FFFFFFFF (INT_MAX)</td>
</tr>
<tr>
<td>unsigned int</td>
<td>4</td>
<td>32</td>
<td>0</td>
<td>7FFFFFFFF (UINT_MAX)</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
<td>64</td>
<td>80000000000000000 (LONG_MIN)</td>
<td>7FFFFFFFFFFFFFFFF (LONG_MAX)</td>
</tr>
<tr>
<td>unsigned long</td>
<td>8</td>
<td>64</td>
<td>0</td>
<td>7FFFFFFFF (ULONG_MAX)</td>
</tr>
</tbody>
</table>
Overflow and Underflow
At which points can overflow occur for signed and unsigned int? (assume binary values shown are all 32 bits)

A. Signed and unsigned can both overflow at points X and Y
B. Signed can overflow only at X, unsigned only at Y
C. Signed can overflow only at Y, unsigned only at X
D. Signed can overflow at X and Y, unsigned only at X
E. Other
Unsigned Integers

Discontinuity means overflow possible here

More increasing positive numbers

Increasing positive numbers

\[ \approx +4 \text{billion} \]
Signed Numbers

Discontinuity means overflow possible here

Increasing positive numbers

Negative numbers becoming less negative (i.e. increasing)

≈-2billion

≈+2billion

000...000
000...001
000...010
000...011
...
111...101
111...100
111...110
111...111
...
011...111
011...110
011...101
011...100
111...110
111...111
111...101
111...100
...
000...010
000...011
000...001
000...000
-1
0
+1

≈-2billion

≈+2billion
Overflow In Practice: PSY

YouTube: “We never thought a video would be watched in numbers greater than a 32-bit integer (=2,147,483,647 views), but that was before we met PSY. "Gangnam Style" has been viewed so many times we had to upgrade to a 64-bit integer (9,223,372,036,854,775,808)!”
Overflow In Practice: Timestamps

• Many systems store timestamps as **the number of seconds since Jan. 1, 1970** in a **signed 32-bit integer**.

• **Problem:** the latest timestamp that can be represented this way is 3:14:07 UTC on Jan. 13 2038!
• In the game “Civilization”, each civilization leader had an “aggression” rating. Gandhi was meant to be peaceful, and had a score of 1.

• If you adopted “democracy”, all players’ aggression reduced by 2. Gandhi’s went from 1 to 255!

• Gandhi then became a big fan of nuclear weapons.

https://kotaku.com/why-gandhi-is-such-an-asshole-in-civilization-1653818245
• There are 3 placeholders for 32-bit integers that we can use:
  • %d: signed 32-bit int
  • %u: unsigned 32-bit int
  • %x: hex 32-bit int

• As long as the value is a 32-bit type, printf will treat it according to the placeholder!
What happens at the byte level when we cast between variable types? **The bytes remain the same!** This means they may be interpreted differently depending on the type.

```c
int v = -12345;
unsigned int uv = v;
printf("v = %d, uv = %u\n", v, uv);
```

This prints out: "v = -12345, uv = 4294954951". Why?
Casting

• What happens at the byte level when we cast between variable types? The bytes remain the same! This means they may be interpreted differently depending on the type.

```c
int v = -12345;
unsigned int uv = v;
printf("v = %d, uv = %u\n", v, uv);
```

The bit representation for -12345 is `0b11000000111001`. If we treat this binary representation as a positive number, it’s huge!
Casting

4-bit two's complement signed integer representation

4-bit unsigned integer representation
Comparisons Between Different Types

- **Be careful** when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned, and then performs the operation assuming both numbers are non-negative.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 == 0U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1 &lt; 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1 &lt; 0U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647 &gt; -2147483647 - 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647U &gt; -2147483647 - 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647 &gt; (int)2147483648U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1 &gt; -2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(unsigned)-1 &gt; -2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparisons Between Different Types

- **Be careful** when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned, and then performs the operation assuming both numbers are non-negative.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 == 0U</td>
<td>Unsigned</td>
<td>1</td>
</tr>
<tr>
<td>-1 &lt; 0</td>
<td>Signed</td>
<td>1</td>
</tr>
<tr>
<td>-1 &lt; 0U</td>
<td>Unsigned</td>
<td>0</td>
</tr>
<tr>
<td>2147483647 &gt; -2147483647 - 1</td>
<td>Signed</td>
<td>1</td>
</tr>
<tr>
<td>2147483647U &gt; -2147483647 - 1</td>
<td>Unsigned</td>
<td>0</td>
</tr>
<tr>
<td>2147483647 &gt; (int)2147483648U</td>
<td>Signed</td>
<td>1</td>
</tr>
<tr>
<td>-1 &gt; -2</td>
<td>Signed</td>
<td>1</td>
</tr>
<tr>
<td>(unsigned)-1 &gt; -2</td>
<td>Unsigned</td>
<td>1</td>
</tr>
</tbody>
</table>
Comparisons Between Different Types

Which many of the following statements are true? (assume that variables are set to values that place them in the spots shown)

- $s_3 > u_3$
- $u_2 > u_4$
- $s_2 > s_4$
- $s_1 > s_2$
- $u_1 > u_2$
- $s_1 > u_3$
Comparisons Between Different Types

Which many of the following statements are true? (assume that variables are set to values that place them in the spots shown)

\[
\begin{align*}
    s_3 & > u_3 \quad \text{- true} \\
    u_2 & > u_4 \quad \text{- true} \\
    s_2 & > s_4 \quad \text{- false} \\
    s_1 & > s_2 \quad \text{- true} \\
    u_1 & > u_2 \quad \text{- true} \\
    s_1 & > u_3 \quad \text{- true}
\end{align*}
\]
Recap

- Bits and Bytes
- Hexadecimal
- Integer Representations
- Unsigned Integers
- Signed Integers
- Break: Announcements
- Casting and Combining Types

Next time: Boolean logic and bit operations