CS107 Winter 2020, Lecture 2
Bits and Bytes; Integer Representations

reading:
Bryant & O’Hallaron, Ch. 2.2-2.3
CS107 Topic 1: How can a computer represent integer numbers?
Demo: Unexpected Behavior

```
cp -r /afs/ir/class/cs107/samples/lectures/lect2 .
```
Plan For Today

• Bits and Bytes
• Hexadecimal
• Integer Representations
• Unsigned Integers
• **Break**: Announcements
• Signed Integers
• Overflow
Plan For Today

• Bits and Bytes
• Hexadecimal
• Integer Representations
• Unsigned Integers
• Break: Announcements
• Signed Integers
• Overflow
Introducing 0
Introducing 0’s Sidekick: 1
• Computers are built around the idea of two states: on and off. Transistors represent this in hardware, and bits represent this in software!
One Bit At A Time

• We can combine bits, as with base-10 numbers, to represent more data. **8 bits = 1 byte.**

• Computer memory is just a large array of bytes. It is **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!
  
  • Text
  • Images
  • Audio
  • Video
  • And more...
Base 10

5 9 3 4

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

\[5934 = 5 \times 1000 + 9 \times 100 + 3 \times 10 + 4 \times 1\]
Base 10

5 9 3 4

\[ \begin{array}{cccc}
10^3 & 10^2 & 10^1 & 10^0 \\
5 & 9 & 3 & 4 \\
\end{array} \]
Base 10

\[ 5 9 3 4 \]

\[ 10^x: \quad 3 \quad 2 \quad 1 \quad 0 \]
Base 2

\[ 1011 \]

\[ 2^x: \quad 3 \quad 2 \quad 1 \quad 0 \]

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

1 0 1 1

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

- **Strategy:**
  - What is the largest power of 2 ≤ 6? $2^2=4$
**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? \(2^1 = 2\)

\[
\begin{array}{cccc}
0 & 1 & 1 & \\
2^3 & 2^2 & 2^1 & 2^0
\end{array}
\]
**Base 10 to Base 2**

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

• **Strategy:**
  • What is the largest power of $2 \leq 6$? $2^2=4$
  • Now, what is the largest power of $2 \leq 6 - 2^2$? $2^1=2$
  • $6 - 2^2 - 2^1 = 0!$

```
0 1 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 – \(2^2\)? \(2^1 = 2\)
  - \(6 – 2^2 – 2^1 = 0!\)

\[
\begin{array}{cccc}
0 & 1 & 1 & 0 \\
2^3 & 2^2 & 2^1 & 2^0 \\
\end{array}
\]
**Base 10 to Base 2**

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

- **Strategy:**
  1. What is the largest power of 2 ≤ 6? \(2^2 = 4\)
  2. Now, what is the largest power of 2 ≤ 6 – 2^2? \(2^1 = 2\)
  3. \(6 – 2^2 – 2^1 = 0!\)

\[
\begin{array}{cccc}
0 & 1 & 1 & 0 \\
\hline
2^3 & 2^2 & 2^1 & 2^0 \\
\end{array}
\]

= \(0 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1 = 6\)
What is the base-2 value 1010 in base-10?

a) 20
b) 101
c) 10
d) 5
e) Other
Practice: Base 10 to Base 2

What is the base-10 value 14 in base 2?

a) 1111  
b) 1110  
c) 1010  
d) Other
Byte Values

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

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

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

• Strategy 1: \[1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 255\]
Byte Values

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

• Strategy 1:  
  \[1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 255\]

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

1450 \times 10 = 14500

1100_{2} \times 2 = 11000_{2}

Key Idea: inserting 0 at the end multiplies by the base!
Dividing by Base

1450 / 10 = 145
1100₂ / 2 = 110

*Key Idea*: removing 0 at the end divides by the base!
Plan For Today

• Bits and Bytes
• **Hexadecimal**
• Integer Representations
• Unsigned Integers
• **Break:** Announcements
• Signed Integers
• Overflow
Hexadecimal

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

Each is a base-16 digit!
Hexadecimal

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

• We distinguish hexadecimal numbers by prefixing them with \(0x\), and binary numbers with \(0b\).

• E.g. \(0xf5\) is \(0b11110101\)
Practice: Hexadecimal to Binary

What is \(0\times173A\) 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>
### Practice: Hexadecimal to Binary

What is `0b1111001010` in hexadecimal? *(Hint: start from the right)*

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

• Bits and Bytes
• Hexadecimal
• **Integer Representations**
• Unsigned Integers
• **Break:** Announcements
• Signed Integers
• Overflow
Number Representations

- **Unsigned Integers**: positive integers, and 0. (e.g. 0, 1, 2, ... 99999…)
- **Signed Integers**: negative, positive and 0. (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 integers, and 0. (e.g. 0, 1, 2, … 99999…)
• **Signed Integers**: negative, positive and 0. (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!
## Number Representations

<table>
<thead>
<tr>
<th>C Declaration</th>
<th>Size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>char *</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
</tr>
</tbody>
</table>
In The Days Of Yore...

<table>
<thead>
<tr>
<th>C Declaration</th>
<th>Size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
</tr>
</tbody>
</table>
Transitioning To Larger Datatypes

• Early 2000s: most computers were 32-bit. This means that pointers were 4 bytes (32 bits).

• 32-bit pointers store a memory address from 0 to $2^{32}-1$, equaling $2^{32}$ 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 some fundamental datatypes got bigger; pointers, for instance, needed to be 64 bits.

• 64-bit pointers store a memory address from 0 to $2^{64}-1$, equaling $2^{64}$ bytes of addressable memory. This equals 16 Exabytes, meaning that 64-bit computers could have up to 1024*1024*1024 GB of memory (RAM)!
Plan For Today

• Bits and Bytes
• Hexadecimal
• Integer Representations
• **Unsigned Integers**
• **Break:** Announcements
• Signed Integers
• Overflow
Unsigned Integers

- An **unsigned** integer is 0 or a positive integer (no negatives).
- We have already discussed converting between decimal and binary. 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
• **Break:** Announcements
• Signed Integers
• Overflow
Announcements

• Sign up for Piazza on the Help page if you haven’t already!

• Lab signups opened this morning at 10:30am, and they start next week.
  • Lab materials posted on the course website at the start of each week

• Helper Hours started earlier this week

• Assignment 0 due Monday evening.
  • No grace period for this one. We want to grade it immediately!

• Assignment 1 goes out this coming Monday, will exercise material from today and this coming Monday.
Plan For Today

• Bits and Bytes
• Hexadecimal
• Integer Representations
• Unsigned Integers
• **Break:** Announcements
• **Signed Integers**
• Overflow
Signed Integers

• A **signed** integer is a negative integer, 0, or a positive integer.

• *Problem:* How can we represent negative *and* positive numbers in binary?
Signed Integers

- A **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 complicates the hardware support for something as fundamental as addition.

Can we do better?
A Better Idea

• Ideally, binary addition should just work regardless of whether the numbers are positive or negative.

\[
\begin{array}{c}
0101 \\
+ \text{ ???} \\
\hline
0000
\end{array}
\]
• Ideally, binary addition should just work regardless of whether the numbers are positive or negative.

```
0101
+1011
------
00000
```
A Better Idea

• Ideally, binary addition should just work regardless of whether the numbers are positive or negative.

\[
\begin{array}{c}
0011 \\
+ ???? \\
0000
\end{array}
\]
A Better Idea

- Ideally, binary addition should just work regardless of whether the numbers are positive or negative.

\[
\begin{align*}
0011 & \quad + \quad 1101 \\
\hline
0000 &
\end{align*}
\]
A Better Idea

• Ideally, binary addition should just work regardless of whether the numbers are positive or negative.

\[
\begin{array}{c}
0000 \\
+ \makecell{????}
\end{array}
\rightleftharpoons 0000
\]
A Better Idea

• Ideally, binary addition should just work regardless of whether the numbers are positive or negative.

```
  0000
+00000
  ----
  00000
```
There Seems Like a Pattern Here...

- The negative number is the positive number \textit{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 compute the negative 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.

\[100100\]
\[+???????\]
\[\underline{0000000}\]
Another Trick

• To compute the negative 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.

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

• To compute the negative 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.

\[ 100100 + 011100 = 0000000 \]
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:** the most significant bit still indicates the sign of a number.
- **Pro:** addition works for any combination of positive and negative!
Two’s Complement

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

\[
\begin{array}{c}
0010 \\
+1011 \\
\hline
1101
\end{array}
\]

2
-5
-3
Two’s Complement

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

\[
\begin{array}{c}
0100 \\
-0101
\end{array}
\quad 4
\quad 5
\quad \begin{array}{c}
0100 \\
+1011
\end{array}
\quad 4
\quad -5
\begin{array}{c}
\hline
1111
\end{array}
\quad -1
\]
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)
Plan For Today

• Bits and Bytes
• Hexadecimal
• Integer Representations
• Unsigned Integers
• Break: Announcements
• Signed Integers
• Overflow
Overflow

- If you exceed the **maximum** value of your bit representation, you *wrap around* or *overflow* back to the **smallest** bit representation. (Assume unsigned 4-bit numbers).

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

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

\[ 0b0000 - 0b1 = 0b1111 \]
# Min and Max Integer Values

<table>
<thead>
<tr>
<th>Type</th>
<th>Size (Bytes)</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>unsigned char</td>
<td>1</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>-32768</td>
<td>32767</td>
</tr>
<tr>
<td>unsigned short</td>
<td>2</td>
<td>0</td>
<td>65535</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>-2147483648</td>
<td>2147483647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>4</td>
<td>0</td>
<td>4294967295</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
<td>-9223372036854775808</td>
<td>9223372036854775807</td>
</tr>
<tr>
<td>unsigned long</td>
<td>8</td>
<td>0</td>
<td>18446744073709551615</td>
</tr>
</tbody>
</table>
Min and Max Integer Values

`INT_MIN`, `INT_MAX`, `UINT_MAX`, `LONG_MIN`, `LONG_MAX`, `ULONG_MAX`, ...
Overflow
At which points can overflow occur for signed and unsigned int?

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

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

Increasing positive numbers

More increasing positive numbers

Discontinuity means overflow possible here
Signed Numbers

Discontinuity means overflow possible here

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

Increasing positive numbers

≈-2 billion

≈+2 billion
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 January 13, 2038!
Overflow in Practice:

- Pacman Level 256
- Make sure to reboot Boeing Dreamliners every 248 days
- Comair/Delta airline had to cancel thousands of flights days before Christmas
- Reported vulnerability CVE-2019-3857 in libssh2 may allow a hacker to remotely execute code
- Donkey Kong Kill Screen
Recap

• Bits and Bytes
• Hexadecimal
• Integer Representations
• Unsigned Integers
• **Break:** Announcements
• Signed Integers
• Overflow
• **Surprise Bonus Topic If We Have Time:** Casting and Combining Types
printf and Integers

• 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

• The placeholder—not the expression filling in the placeholder—dictates what gets printed!
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);
```

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.

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

The bit representation for -12345 is 0b110011111000111. 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>
<th>Correct?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 == 0U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1 &lt; 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1 &lt; 0U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647 &gt; -2147483647 - 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647U &gt; -2147483647 - 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647 &gt; (int)2147483648U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1 &gt; -2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(unsigned)-1 &gt; -2</td>
<td></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>
<th>Correct?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 == 0U</td>
<td>Unsigned</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>-1 &lt; 0</td>
<td>Unsigned</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>-1 &lt; 0U</td>
<td>Unsigned</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>2147483647 &gt; -2147483647 - 1</td>
<td></td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>2147483647U &gt; -2147483647 - 1</td>
<td></td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>2147483647 &gt; (int)2147483648U</td>
<td></td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>-1 &gt; -2</td>
<td></td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>(unsigned)-1 &gt; -2</td>
<td></td>
<td>-1</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>
<th>Correct?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 == 0U</td>
<td>Unsigned</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>-1 &lt; 0</td>
<td>Signed</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>-1 &lt; 0U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647 &gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-2147483647 - 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-2147483647U &gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-2147483647 - 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647 &gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(int)2147483648U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1 &gt; -2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(unsigned)-1 &gt; -2</td>
<td></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>
<th>Correct?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>0 == 0U</code></td>
<td>Unsigned</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td><code>-1 &lt; 0</code></td>
<td>Signed</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td><code>-1 &lt; 0U</code></td>
<td>Unsigned</td>
<td>0</td>
<td>No!</td>
</tr>
<tr>
<td><code>2147483647 &gt;</code></td>
<td><code>2147483647 - 1</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>(-2147483647 &lt; 1)</code></td>
<td><code>2147483647U &gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>(-2147483647 - 1)</code></td>
<td><code>(int)2147483648U</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>-1 &gt; -2</code></td>
<td><code>(unsigned)</code>-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>
<th>Correct?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 == 0U</td>
<td>Signed</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>-1 &lt; 0</td>
<td>Signed</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>-1 &lt; 0U</td>
<td>Unsigned</td>
<td>0</td>
<td>No!</td>
</tr>
<tr>
<td>2147483647 &gt; 2147483647 - 1</td>
<td>Signed</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>2147483647U &gt; 2147483647 - 1</td>
<td>Signed</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>2147483647 &gt; (int)2147483648U</td>
<td>Signed</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>-1 &gt; -2</td>
<td>Unsigned</td>
<td>-1 &lt; 0</td>
<td>No!</td>
</tr>
<tr>
<td>(unsigned)-1 &gt; -2</td>
<td>Unsigned</td>
<td>-1 &lt; 0</td>
<td>No!</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>
<th>Correct?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 == 0U</td>
<td>Unsigned</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>-1 &lt; 0</td>
<td>Signed</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>-1 &lt; 0U</td>
<td>Unsigned</td>
<td>0</td>
<td>No!</td>
</tr>
<tr>
<td>2147483647 &gt; -2147483647 - 1</td>
<td>Signed</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>2147483647U &gt; -2147483647 - 1</td>
<td>Unsigned</td>
<td>0</td>
<td>No!</td>
</tr>
<tr>
<td>2147483647 &gt; (int)2147483648U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1 &gt; -2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(unsigned)-1 &gt; -2</td>
<td></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>
<th>Correct?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 == 0U</td>
<td>Unsigned</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>-1 &lt; 0</td>
<td>Signed</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>-1 &lt; 0U</td>
<td>Unsigned</td>
<td>0</td>
<td>No!</td>
</tr>
<tr>
<td>2147483647 &gt; -2147483647 - 1</td>
<td>Signed</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>2147483647U &gt; -2147483647 - 1</td>
<td>Unsigned</td>
<td>0</td>
<td>No!</td>
</tr>
<tr>
<td>2147483647 &gt; (int)2147483648U</td>
<td>Signed</td>
<td>1</td>
<td>No!</td>
</tr>
<tr>
<td>-1 &gt; -2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(unsigned)-1 &gt; -2</td>
<td></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>
<th>Correct?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 == 0U</td>
<td>Unsigned</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>-1 &lt; 0</td>
<td>Signed</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>-1 &lt; 0U</td>
<td>Unsigned</td>
<td>0</td>
<td>No!</td>
</tr>
<tr>
<td>2147483647 &gt; -2147483647 - 1</td>
<td>Signed</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>2147483647U &gt; -2147483647 - 1</td>
<td>Unsigned</td>
<td>0</td>
<td>No!</td>
</tr>
<tr>
<td>2147483647 &gt; (int)2147483648U</td>
<td>Signed</td>
<td>1</td>
<td>No!</td>
</tr>
<tr>
<td>-1 &gt; -2</td>
<td>Signed</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>(unsigned)-1 &gt; -2</td>
<td>Signed</td>
<td>1</td>
<td>yes</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>
<th>Correct?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 == 0U</td>
<td>Unsigned</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>-1 &lt; 0</td>
<td>Signed</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>-1 &lt; 0U</td>
<td>Unsigned</td>
<td>0</td>
<td>No!</td>
</tr>
<tr>
<td>2147483647 &gt; -1</td>
<td>Signed</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>2147483647U &gt; -1</td>
<td>Unsigned</td>
<td>0</td>
<td>No!</td>
</tr>
<tr>
<td>2147483647 &gt; (int)2147483648U</td>
<td>Signed</td>
<td>1</td>
<td>No!</td>
</tr>
<tr>
<td>-1 &gt; -2</td>
<td>Signed</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>(unsigned)-1 &gt; -2</td>
<td>Unsigned</td>
<td>1</td>
<td>yes</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)

s3 > u3
u2 > u4
s2 > s4
s1 > s2
u1 > u2
s1 > u3
Which many of the following statements are true? (assume that variables are set to values that place them in the spots shown)

- $s_3 > u_3$ - true
- $u_2 > u_4$
- $s_2 > s_4$
- $s_1 > s_2$
- $u_1 > u_2$
- $s_1 > u_3$
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)

- s3 > u3 - true
- u2 > u4 - true
- s2 > s4
- s1 > s2
- u1 > u2
- s1 > u3
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)

- \( s_3 > u_3 \) - true
- \( u_2 > u_4 \) - true
- \( s_2 > s_4 \) - false
- \( s_1 > s_2 \)
- \( u_1 > u_2 \)
- \( s_1 > u_3 \)
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)

s3 > u3 - true
u2 > u4 - true
s2 > s4 - false
s1 > s2 - true
u1 > u2
s1 > u3
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)

- $s_3 > u_3$ - true
- $u_2 > u_4$ - true
- $s_2 > s_4$ - false
- $s_1 > s_2$ - true
- $u_1 > u_2$ - true
- $s_1 > u_3$
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)

- $s_3 > u_3$ - true
- $u_2 > u_4$ - true
- $s_2 > s_4$ - false
- $s_1 > s_2$ - true
- $u_1 > u_2$ - true
- $s_1 > u_3$ - true
Expanding Bit Representations

• Sometimes, we want to convert between two integers of different sizes (e.g. short to int, or int to long).

• We might not be able to convert from a bigger data type to a smaller data type, but we do want to always be able to convert from a smaller data type to a bigger data type.

• For unsigned values, we can add leading zeros to the representation (“zero extension”)

• For signed values, we can repeat the sign of the value for new digits (“sign extension”)

• Note: when doing <, >, <=, >= comparison between different size types, it will promote to the larger type.
unsigned short s = 4;
// short is a 16-bit format, so
unsigned int i = s;
// conversion to 32-bit int, so i = 0000 0000 0000 0000 0000 0000 0000 0100b
Expanding Bit Representation

short s = 4;
// short is a 16-bit format, so s = 0000 0000 0000 0100b

int i = s;
// conversion to 32-bit int, so i = 0000 0000 0000 0000 0000 0000 0000 0100b

— or —

short s = -4;
// short is a 16-bit format, so s = 1111 1111 1111 1100b

int i = s;
// conversion to 32-bit int, so i = 1111 1111 1111 1111 1111 1111 1111 1100b
If we want to **reduce** the bit size of a number, C **truncates** the representation and discards the more significant bits.

```c
int x = 53191;
short sx = x;
int y = sx;
```

What happens here? Let's look at the bits in \( x \) (a 32-bit int), 53191:

```
0000 0000 0000 0000 1100 1111 1100 0111
```

When we cast \( x \) to a short, it only has 16-bits, and C **truncates** the number:

```
1100 1111 1100 0111
```

This is -12345! And when we cast \( sx \) back to int, we sign-extend the number.

```
1111 1111 1111 1111 1100 1111 1100 0111 // still -12345
```
If we want to **reduce** the bit size of a number, C **truncates** the representation and discards the more significant bits.

```c
int x = -3;
short sx = x;
int y = sx;
```

What happens here? Let's look at the bits in `x` (a 32-bit int), -3:

```
1111 1111 1111 1111 1111 1111 1111 1101
```

When we cast `x` to a short, it only has 16-bits, and C **truncates** the number:

```
1111 1111 1111 1101
```

This is -3! **If the number does fit, it will convert fine.** `y` looks like this:

```
1111 1111 1111 1111 1111 1111 1111 1101    // still -3
```
If we want to **reduce** the bit size of a number, C **truncates** the representation and discards the more significant bits.

```c
unsigned int x = 128000;
unsigned short sx = x;
unsigned int y = sx;
```

What happens here? Let's look at the bits in x (a 32-bit unsigned int), 128000:

```
0000 0000 0000 0001 1111 0100 0000 0000
```

When we cast x to a short, it only has 16-bits, and C **truncates** the number:

```
1111 0100 0000 0000
```

This is 62464! **Unsigned numbers can lose info too.** Here is what y looks like:

```
0000 0000 0000 0000 1111 0100 0000 0000  // still 62464
```
The sizeof Operator

long sizeof(type);

// Example
long int_size_bytes = sizeof(int);  // 4
long short_size_bytes = sizeof(short);  // 2
long char_size_bytes = sizeof(char);  // 1

sizeof takes a variable type as a parameter and returns the size of that type, in bytes.