Fill in the check-in form on cs107a.stanford.edu!
Announcements

- Until the end of Week 3, I’m willing to help anyone, enrolled or accepted or not, in Office Hours and over Slack
  - I’ve gotten 1 person / hour so far
- Enrollment codes were sent out last night.
  - Enrolling only gets you office hours and 1:1 time with me. You can still attend section and review sessions otherwise.
  - Once you can, please enroll right away, you can always drop later
- Attendance starts to count for credit today
- assign1 walkthrough posted
- First 1:1s coming soon-ish (end of Week 2 - Week 3?)
  - 1) Get to know you
  - 2) Assignment setup and debugging
- Lab assignments are out - you attend these separately and in addition to 107A section :)

Unix Tip Spotlight

- We'll highlight a tip on how to use Unix effectively each week until I run out of tips
- Use two terminals at once, side-by-side, to avoid having to exit emacs!
- How do you rerun previous commands?
  - `<UP>`: Brings back the previous run command
    - can be repeated to go further back in history
  - `<DOWN>`: Goes forward in history after using `<UP>`
  - `<CTRL+P>`: same as `<UP>`, but works in more places
  - `<CTRL+N>`: same as `<DOWN>`, but works in more places
Thoughts on assign1

- [https://web.stanford.edu/class/archive/cs/cs107/cs107.1226/assign1/](https://web.stanford.edu/class/archive/cs/cs107/cs107.1226/assign1/)
Agenda

- Integer Bases
- Integer Representations
- Integer Casting

A page of notes about integer bases and conversions are posted on the website.

We won’t have time to discuss everything about integers today.
Integer Bases
# Integer Bases for Computers

<table>
<thead>
<tr>
<th>Value</th>
<th>Decimal</th>
<th>Binary</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0b0000_0000</td>
<td>0x00</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0b0000_0001</td>
<td>0x01</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0b0000_0010</td>
<td>0x02</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0b0000_0011</td>
<td>0x03</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0b0000_0100</td>
<td>0x04</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0b0000_0101</td>
<td>0x05</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0b0000_0110</td>
<td>0x06</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>0b0000_0111</td>
<td>0x07</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0b0000_1000</td>
<td>0x08</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>0b0000_1001</td>
<td>0x09</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0b0000_1010</td>
<td>0x0a</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>0b0000_1011</td>
<td>0x0b</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>0b0000_1100</td>
<td>0x0c</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>0b0000_1101</td>
<td>0x0d</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>0b0000_1110</td>
<td>0x0e</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>0b0000_1111</td>
<td>0x0f</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0b0001_0000</td>
<td>0x10</td>
</tr>
</tbody>
</table>
In addition, there are always tricks you can perform for certain numbers.

- Binary -> Decimal: Sum up the powers of 2 each 1 corresponds to
- Decimal -> Binary: Break up the number into powers of 2, and put a 1 for each power, and 0 for the rest
- Binary -> Hexadecimal: Break up into groups of 4 bits, padding if necessary. Translate each group into a hex digit
- Hexadecimal: Translate each hex digit into 4 binary bits
- Hexadecimal -> Decimal: Sum up powers of 16
- Decimal -> Hexadecimal: go through binary, or break up into powers of 16
Integer Conversion Practice

- In pairs, work on p1 of https://web.stanford.edu/class/cs107a/exercises/integer-bases/
  - If you’re both stuck on one, do the ones you can do, then ask another pair (if time)
  - Check your answers with `gdb`, if you know/remember how
- We’ll go over them together afterwards
C always uses binary

- C allows you to write integer constants as decimal or hex, whatever you deem most appropriate for the situation.
- But C will represent every integer in binary for its own purposes.
- It doesn’t matter! Arithmetic still works the same way since arithmetic operates on the value, not how we write it down.
- C allows you to print out in whatever format (decimal or hex) you prefer.
- [https://cplayground.com/?p=mongoose-spider-axolotl](https://cplayground.com/?p=mongoose-spider-axolotl)
More Integer Conversion Practice

- p2 of https://web.stanford.edu/class/cs107a/exercises/integer-bases/
Why do we care about different formats?

- Decimal is useful since it’s how humans generally think about real-life quantities, so you’d want to be able to easily express them in code.
- Binary is useful because it’s what the computer uses natively, and bitwise operations (next section) operate directly on the bit representation, so you have to know it.
- Hex is useful since binary can be very verbose, whereas hex is 4 times shorter, and with practice it’s very easy to convert between them (especially since you can’t really write binary in your code).
Integer Representations
Bits and Bytes

Try to use the right terminology here! Students often mix these up.

- bit: a single 1 or 0
- byte: a group of 8 bits

Note: A hex digit represents 4 bits.

A 8-bit number takes up 1 byte. In hex, this is two digits: 0xab = 0b1010_1011.

A 32-bit number takes up 4 bytes. In hex, this requires 8 digits.

A 64-bit number takes up 8 bytes. In hex, this requires 16 digits.
Some C Integer Types you may see in CS 107
(Note: sizes are as given on the Myth machines)

<table>
<thead>
<tr>
<th>Size</th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 byte / 8 bits</td>
<td>signed char</td>
<td>unsigned char</td>
</tr>
<tr>
<td>2 bytes / 16 bits</td>
<td>[signed] short</td>
<td>unsigned short</td>
</tr>
<tr>
<td>4 bytes / 32 bits</td>
<td>[signed] int signed</td>
<td>unsigned int unsigned</td>
</tr>
<tr>
<td>8 bytes / 64 bits</td>
<td>[signed] long</td>
<td>unsigned long size_t</td>
</tr>
</tbody>
</table>
What should I use by default?

- In general, we just use `int` unless we have a good reason to use a different type
  - ex) Use `long` if you’re working with huge (more than a billion) numbers
  - ex) Use `unsigned` types if you’re working with non-negative quantities
    - But even then, not always a good idea. [Google C++ Style Guide](https://Extras/Google Style Guide) says to avoid them
  - I raise an eyebrow whenever someone uses a non-`int` type
- Using a smaller type is NOT necessarily (and probably isn’t) more efficient.
  - We’ll see more on why when we study assembly
Size and sign

- The difference between integer types is in their representable range.
- An unsigned $n$-bit type can represent values in $[0, 2^n - 1]$.  
- A signed $n$-bit type can represent values in $[-2^{n-1}, 2^{n-1} - 1]$.  

<table>
<thead>
<tr>
<th>Byte Size</th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 byte / 8 bytes</td>
<td>signed char: ([-2^7, 2^7-1]), ([-128, 127])</td>
<td>unsigned char: ([0, 2^8-1]), ([0, 255])</td>
</tr>
<tr>
<td>2 bytes / 16 bits</td>
<td>[signed] short: ([-2^{15}, 2^{15}-1])</td>
<td>unsigned short: ([0, 2^{16}-1])</td>
</tr>
<tr>
<td>4 bytes / 32 bits</td>
<td>[signed] int</td>
<td>unsigned int</td>
</tr>
<tr>
<td></td>
<td>signed: ([-2^{31}, 2^{31}-1])</td>
<td>unsigned: ([0, 2^{32}-1])</td>
</tr>
<tr>
<td>8 bytes / 64 bits</td>
<td>[signed] long</td>
<td>unsigned long size_t</td>
</tr>
<tr>
<td></td>
<td>signed: ([-2^{63}, 2^{63}-1])</td>
<td>([0, 2^{64}-1])</td>
</tr>
</tbody>
</table>
Unsigned representation

- C uses regular binary to represent unsigned integer types (but with a limited number of bits)
- Assume a 8-bit system
  - 0 is 0b0000_0000
  - 1 is 0b0000_0001
  - ...
  - 254 is 0b1111_1110
  - 255 is 0b1111_1111
  - 256 is unrepresentable
Signed representation

- C uses a special kind of binary called “Two’s Complement”
- It’s just regular binary except we negate the contribution of the MSB (Most Significant Bit - the leftmost)
- This means the MSB also determines the sign: 1 = negative, 0 = 0 or positive

\[
\text{Ob}1111\_1111
\]

\[
-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 =
-128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 =
-1
\]

- But doing this arithmetic can get tiresome - you can also use the Two’s Complement identity trick: \(-x = \sim x + 1\)
Integer Representation Practice

- In pairs, work on p4 of [https://web.stanford.edu/class/cs107a/exercises/integer-representations/](https://web.stanford.edu/class/cs107a/exercises/integer-representations/)
- We'll go over them together afterwards
- If time, also do p3
sizeof

- Gets you the number of *bytes* a value or type takes up
- [https://cplayground.com/?p=kudu-mammoth-okapi](https://cplayground.com/?p=kudu-mammoth-okapi)
Warning: default literal ints are signed ints

- while (i < 3) ← signed int
- while (i < 3U) ← unsigned int
- while (i < 3L) ← signed long
- while (i < 3UL) ← unsigned long

See what I did there
What if arithmetic causes you to go out of the representable range?

- **Unsigned types**: wrap-around overflow
  - 8 bit system: $255 + 1 = 0$. $255 + 2 = 1$. $0 - 1 = 255$.
  - $n$ bit system: $2^{n-1} + 1 = 0$. $2^{n-1} + 2 = 1$. $0 - 1 = 2^{n-1}$.
  - In general, forget about the ranges, do the arithmetic, take the remainder mod $2^n$
    - $255 + 356 \mod 256 = -1 + 100 \mod 256 = 99$

- **Signed types**: \_/\_\_\_ (but wrap-around overflow)
  - In general: forget about the ranges, do the arithmetic, and add or subtract multiples of $2^n$ until in range
    - $127 + 1 \rightarrow 128 \rightarrow subtract 256 \rightarrow -128$
#include this file gets you some nice constants, like `CHAR_BIT`

<table>
<thead>
<tr>
<th></th>
<th>MIN</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit</td>
<td><code>SCHAR_MIN</code></td>
<td><code>SCHAR_MAX, UCHAR_MAX</code></td>
</tr>
<tr>
<td>16-bit</td>
<td><code>SHRT_MIN</code></td>
<td><code>SHRT_MAX, USHRT_MAX</code></td>
</tr>
<tr>
<td>32-bit</td>
<td><code>INT_MIN</code></td>
<td><code>INT_MAX, UINT_MAX</code></td>
</tr>
<tr>
<td>64-bit</td>
<td><code>LONG_MIN</code></td>
<td><code>LONG_MAX, ULONG_MAX, SIZE_MAX</code> (not part of <code>&lt;limits.h&gt;</code>)</td>
</tr>
</tbody>
</table>
More Integer Representation Practice

- In pairs, work on p5 of https://web.stanford.edu/class/cs107a/exercises/integer-representations/
- We'll go over it together afterwards
Integer Casting
Implicit vs Explicit Casting

- Explicit casting is when you explicitly cast
  - `(int)x, (unsigned long)3U`

- Implicit casting occurs in several situations, and happens automatically. Some examples:
  - When you assign to a variable of a different type: `int x = 3U;`
  - When you pass a value to a function as a parameter of a different type:
    `printf("%d\n", 4UL);`
  - When you do arithmetic or compare integers of different type: `2 < 4U, 5L + 6`
    - If different sign, the signed operand is casted to unsigned
      - `1U < -1 → 1U < 4294967295U → true`
    - Values are always casted to the larger size or at least `int` ("Integer Promotion")
Casting Rules (with some handwaving)

- If casting to a different sign: preserve the bit representation, and reinterpret them using the new type's rules
  - (unsigned char) -1 → (unsigned char) 0xFF → 255
- If casting to a larger size: add leading bits that preserve the value (leading zeroes for unsigned, leading MSBs for signed)
  - (signed long) -1 is still -1L because leading 1s (the MSB) were added!
- If casting to a smaller size:, truncation of the higher-ordered bytes occurs
  - (unsigned char) 259 → (unsigned char) 0x0103 → 3
Integer Casting Practice

- In pairs, work on p2-4 (feel free to skip around, I’d like you to do p4) of https://web.stanford.edu/class/cs107a/exercises/integer-casting/
- We’ll go over them together afterwards