CS107, Lecture 3
Bits and Bytes; Bitwise Operators

reading:
Bryant & O’Hallaron, Ch. 2.1
Plan For Today

• **Recap**: Integer Representations
• Truncating and Expanding
• Bitwise Boolean Operators and Masks
• **Demo 1**: Courses
• **Break**: Announcements
• **Demo 2**: Powers of 2
• Bit Shift Operators
Plan For Today

• **Recap**: Integer Representations
  • Truncating and Expanding
  • Bitwise Boolean Operators and Masks
• **Demo 1**: Courses
• **Break**: Announcements
• **Demo 2**: Powers of 2
  • Bit Shift Operators
Base 2

\[ 1 \, 0 \, 1 \, 1 \]

\[ 2^3 \, 2^2 \, 2^1 \, 2^0 \]
## 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>
## 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>
Unsigned Integers

4-bit unsigned integer representation

0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
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!
Signed Integers: 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!
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.

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

<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>800000000 (INT_MIN)</td>
<td>7FFFFFFFF (INT_MAX)</td>
</tr>
<tr>
<td>unsigned int</td>
<td>4</td>
<td>32</td>
<td>0</td>
<td>FFFFFFFFF (UINT_MAX)</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
<td>64</td>
<td>8000000000000000000000000 (LONG_MIN)</td>
<td>7FFFFFFFFFFFFFFFFFFFFFFFF (LONG_MAX)</td>
</tr>
<tr>
<td>unsigned long</td>
<td>8</td>
<td>64</td>
<td>0</td>
<td>FFFFFFFFFFFFFFFFFFFFFFFFF (ULONG_MAX)</td>
</tr>
</tbody>
</table>
Aside: ASCII

• ASCII is an encoding from common characters (letters, symbols, etc.) to bit representations (chars).
  • E.g. 'A' is 0x41

• Neat property: all uppercase letters, and all lowercase letters, are sequentially represented!
  • E.g. 'B' is 0x42
Unsigned Integers

≈+4billion 0

Discontinuity means overflow possible here

Increasing positive numbers

More increasing positive numbers
Signed Numbers

Discontinuity means overflow possible here

Increasing positive numbers

≈+2billion

≈-2billion

Negative numbers becoming less negative (i.e. increasing)
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 \texttt{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>
<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; -</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; (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>
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>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)*

\[ s_3 > u_3 \]
\[ u_2 > u_4 \]
\[ s_2 > s_4 \]
\[ s_1 > s_2 \]
\[ u_1 > u_2 \]
\[ s_1 > u_3 \]
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)

s3 > u3
u2 > u4
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
- $s_1 > s_2$ - true
- $u_1 > u_2$ - true
- $s_1 > u_3$ - true
Plan For Today

- Recap: Integer Representations
- Truncating and Expanding
- Bitwise Boolean Operators and Masks
- Demo 1: Courses
- Break: Announcements
- Demo 2: Powers of 2
- Bit Shift Operators
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
s = 0000 0000 0000 0100b

unsigned int i = s;
// conversion to 32-bit int, so
i = 0000 0000 0000 0000 0000 0000 0000 0100b
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 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 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 an 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 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 1101  // still -3
```

---

**Truncating Bit Representation**
Truncating Bit Representation

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**

- `sizeof` takes a variable type as a parameter and returns the number of bytes that type uses.

```c
printf("sizeof(char): %d\n", (int) sizeof(char));
printf("sizeof(short): %d\n", (int) sizeof(short));
printf("sizeof(int): %d\n", (int) sizeof(int));
printf("sizeof(unsigned int): %d\n", (int) sizeof(unsigned int));
printf("sizeof(long): %d\n", (int) sizeof(long));
printf("sizeof(long long): %d\n", (int) sizeof(long long));
printf("sizeof(size_t): %d\n", (int) sizeof(size_t));
printf("sizeof(void *): %d\n", (int) sizeof(void *));
```

```
$ ./sizeof
sizeof(char): 1
sizeof(short): 2
sizeof(int): 4
sizeof(unsigned int): 4
sizeof(long): 8
sizeof(long long): 8
sizeof(size_t): 8
sizeof(void *): 8
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Width in bytes</th>
<th>Width in bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>void *</td>
<td>8</td>
<td>64</td>
</tr>
</tbody>
</table>
Plan For Today

• **Recap**: Integer Representations
• Truncating and Expanding
• **Bitwise Boolean Operators and Masks**
• **Demo 1**: Courses
• **Break**: Announcements
• **Demo 2**: Powers of 2
• Bit Shift Operators
• You’re already familiar with many operators in C:
  • Arithmetic operators: +, -, *, /, %
  • Comparison operators: ==, !=, <, >, <=, >=
  • Logical Operators: &&, ||, !

• Today, we’re introducing a new category of operators: bitwise operators:
  • &, |, ~, ^, <<, >>
AND is a binary operator. The AND of 2 bits is 1 if both bits are 1, and 0 otherwise. **Note:** this is different from Boolean AND (&&)!

\[
\text{output} = a \& b;
\]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
OR is a binary operator. The OR of 2 bits is 1 if either (or both) bits is 1. **Note:** this is different from Boolean OR (||)!

output = a | b;

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
NOT is a unary operator. The NOT of a bit is 1 if the bit is 0, or 1 otherwise. 

**Note:** this is different from Boolean NOT (!)!

\[
output = \sim a;
\]

<table>
<thead>
<tr>
<th>a</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Exclusive Or (XOR) is a binary operator. The XOR of 2 bits is 1 if \textit{exactly} one of the bits is 1, or 0 otherwise.

\[
\text{output} = a \ ^\wedge\ b;
\]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
An Aside: Boolean Algebra

• These operators are not unique to computers; they are part of a general area called **Boolean Algebra**, and are called **Boolean Operators**. These are applicable in math, hardware, computers, and more!
Operators on Multiple Bits

- When these Boolean operators are applied to numbers (multiple bits), the operator is applied to the corresponding bits in each number. For example:

<table>
<thead>
<tr>
<th>AND</th>
<th>OR</th>
<th>XOR</th>
<th>NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110 &amp; 1100 = 0100</td>
<td>0110</td>
<td>0110 ^ 1100 = 1010</td>
<td>~ 1100 = 0011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1110</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td>1010</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1110</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0011</td>
<td></td>
</tr>
</tbody>
</table>
Bit Vectors and Sets

- We can use bit vectors (ordered collections of bits) to represent finite sets, and perform functions such as union, intersection, and complement.

- **Example:** we can represent current courses taken using a `char`.

<table>
<thead>
<tr>
<th>Course</th>
<th>CS161</th>
<th>CS109</th>
<th>CS103</th>
<th>CS110</th>
<th>CS107</th>
<th>CS106X</th>
<th>CS106B</th>
<th>CS106A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

38
• How do we find the union of two sets of courses taken? Use OR:

```
00100011
| 01100001
-----
01100011
```
Bit Vectors and Sets

• How do we find the intersection of two sets of courses taken? Use AND:

\[
\begin{array}{cccccccc}
& 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
\text{CS161} & \text{CS109} & \text{CS103} & \text{CS110} & \text{CS107} & \text{CS106X} & \text{CS106B} & \text{CS106A} \\
\end{array}
\]

\[
\begin{align*}
00100011 \\
\& 01100001 \\
\hline \\
00100001
\end{align*}
\]
Bit Masking

• We will frequently want to manipulate or isolate out specific bits in a larger collection of bits. A bitmask is a constructed bit pattern that we can use, along with bit operators, to do this.

• Example: how do we update our bit vector to indicate we’ve taken CS107?

```
0 0 1 0 0 0 0 1 1
0 0 100011
| 00001000
------
00101011
```
enum Classes {
    CS106A = 0x1, /* 0000 0001 */
    CS106B = 0x2, /* 0000 0010 */
    CS106X = 0x4, /* 0000 0100 */
    CS107 = 0x8, /* 0000 1000 */
    CS110 = 0x10, /* 0001 0000 */
    CS103 = 0x20, /* 0010 0000 */
    CS109 = 0x40, /* 0100 0000 */
    CS161 = 0x80, /* 1000 0000 */
};

char myClasses = ...;
myClasses = myClasses | CS107; // Add CS107
Bit Masking

```c
enum Classes {
    CS106A = 0x1,  /* 0000 0001 */
    CS106B = 0x2,  /* 0000 0010 */
    CS106X = 0x4,  /* 0000 0100 */
    CS107 = 0x8,   /* 0000 1000 */
    CS110 = 0x10,  /* 0001 0000 */
    CS103 = 0x20,  /* 0010 0000 */
    CS109 = 0x40,  /* 0100 0000 */
    CS161 = 0x80,  /* 1000 0000 */
};

char myClasses = ...;
myClasses |= CS107;  // Add CS107
```
**Bit Masking**

- **Example:** how do we update our bit vector to indicate we’ve *not* taken CS103?

```plaintext
00000011
& 11011111
----
00000011
```

```
char myClasses = ...;
myClasses = myClasses & ~CS103; // Remove CS103
```
Bit Masking

- **Example:** how do we update our bit vector to indicate we’ve *not* taken CS103?

```
0 0 1 0 0 0 0 1 1
```

00100011

& 11011111

--------

00000011

```c
char myClasses = ...;
myClasses &= ~CS103;  // Remove CS103
```
Bit Masking

**Example:** how do we check if we’ve taken CS106B?

```java
char myClasses = ...;
if (myClasses & CS106B) {
    // taken CS106B!
```
Bit Masking

• **Example:** how do we check if we’ve *not* taken CS107?

```
0 0 1 0 0 0 0 1 1
```

| CS161 | CS109 | CS103 | CS110 | CS107 | CS106X | CS106B | CS106A |

```cpp
char myClasses = ...;
if (((myClasses & CS107) ^ CS107) {...
   // not taken CS107!
```
**Bit Masking**

**Example:** how do we check if we’ve *not* taken CS107?

```
0 0 1 0 0 0 1 1
```

<table>
<thead>
<tr>
<th>CS161</th>
<th>CS109</th>
<th>CS103</th>
<th>CS110</th>
<th>CS107</th>
<th>CS106X</th>
<th>CS106B</th>
<th>CS106A</th>
</tr>
</thead>
</table>

```char myClasses = ...;
if (!(myClasses & CS107)) {...
    // not taken CS107!
```
Demo: Bitmasks and GDB
Bit Masking

• Bit masking is also useful for integer representations as well. For instance, we might want to check the value of the most-significant bit, or just one of the middle bytes.

• **Example:** If I have a 32-bit integer \(j\), what operation should I perform if I want to get *just the lowest byte* in \(j\)?

```
int j = ...;
int k = j & 0xff;  // mask to get just lowest byte
```
Practice: Bit Masking

• **Practice 1:** write an expression that, given a 32-bit integer $j$, sets its least-significant byte to all 1s, but preserves all other bytes.

• **Practice 2:** write an expression that, given a 32-bit integer $j$, flips ("complements") all but the least-significant byte, and preserves all other bytes.
Practice: Bit Masking

• **Practice 1:** write an expression that, given a 32-bit integer j, sets its least-significant byte to all 1s, but preserves all other bytes.
  
  ```
  j | 0xff
  ```

• **Practice 2:** write an expression that, given a 32-bit integer j, flips (“complements”) all but the least-significant byte, and preserves all other bytes.
  
  ```
  j ^ ~0xff
  ```
Plan For Today

• **Recap:** Integer Representations
• Truncating and Expanding
• Bitwise Boolean Operators and Masks
• **Demo 1:** Courses

• **Break:** Announcements
• **Demo 2:** Powers of 2
• Bit Shift Operators
Announcements

• Office Hours Updates – Nick’s OH location, Fri time, and future 1-time adjustments. *See office hours calendar!*

• Please send us any OAE letters or athletics conflicts as soon as possible. Thanks!

• Assignment 0 deadline tonight at 11:59PM PST

• Assignment 1 (Bit operations!) goes out tonight at Assignment 0 deadline
  • Saturated arithmetic
  • Game of Life
  • Unicode and UTF-8

• Lab 1 this week!
Demo: Powers of 2
Plan For Today

• Recap: Integer Representations
• Truncating and Expanding
• Bitwise Boolean Operators and Masks
• Demo 1: Courses
• Break: Announcements
• Demo 2: Powers of 2

• Bit Shift Operators
The LEFT SHIFT operator shifts a bit pattern a certain number of positions to the left. New lower order bits are filled in with 0s, and bits shifted off of the end are lost.

```c
x << k;    // shifts x to the left by k bits
```

8-bit examples:
- `00110111 << 2` results in `11011100`
- `01100011 << 4` results in `00110000`
- `10010101 << 4` results in `01010000`
The RIGHT SHIFT operator shifts a bit pattern a certain number of positions to the right. Bits shifted off of the end are lost.

\[ x >> k; \quad // \text{shifts } x \text{ to the right by } k \text{ bits} \]

**Question:** how should we fill in new higher-order bits?

**Idea:** let’s follow left-shift and fill with 0s.

```c
short x = 2;  // 0000 0000 0000 0010
x >> 1;       // 0000 0000 0000 0001
printf("%d\n", x); // 1
```
The RIGHT SHIFT operator shifts a bit pattern a certain number of positions to the right. Bits shifted off of the end are lost.

```
x >> k;       // shifts x to the right by k bits
```

**Question:** how should we fill in new higher-order bits?

**Idea:** let’s follow left-shift and fill with 0s.

```
short x = -2;  // 1111 1111 1111 1110
x >> 1;        // 0111 1111 1111 1111
printf("%d\n", x); // 32767!
```
Right Shift (>>)

The RIGHT SHIFT operator shifts a bit pattern a certain number of positions to the right. Bits shifted off of the end are lost.

```
x >> k;       // shifts x to the right by k bits
```

**Question:** how should we fill in new higher-order bits?

**Problem:** always filling with zeros means we may change the sign bit.

**Solution:** let’s fill with the sign bit!
The RIGHT SHIFT operator shifts a bit pattern a certain number of positions to the right. Bits shifted off of the end are lost.

\[ x \gg k; \quad // \text{shifts } x \text{ to the right by } k \text{ bits} \]

**Question:** how should we fill in new higher-order bits?

**Solution:** let’s fill with the sign bit!

```c
short x = 2;  // 0000 0000 0000 0010
x \gg 1;  // 0000 0000 0000 0001
printf("%d\n", x);  // 1
```
The RIGHT SHIFT operator shifts a bit pattern a certain number of positions to the right. Bits shifted off of the end are lost.

\[ x \gg k; \quad // \text{shifts} \ x \text{ to the right by} \ k \text{ bits} \]

**Question:** how should we fill in new higher-order bits?

**Solution:** let’s fill with the sign bit!

```c
short x = -2; // 1111 1111 1111 1110
x >>= 1; // 1111 1111 1111 1111
printf("%d\n", x); // -1!
```
There are two kinds of right shifts, depending on the value and type you are shifting:

- **Logical Right Shift:** fill new high-order bits with 0s.
- **Arithmetic Right Shift:** fill new high-order bits with the most-significant bit.

*Unsigned numbers* are right-shifted using **Logical Right Shift**.

*Signed numbers* are right-shifted using **Arithmetic Right Shift**.

This way, the sign of the number (if applicable) is preserved!
1. *Technically*, the C standard does not precisely define whether a right shift for signed integers is logical or arithmetic. However, *almost all* compilers/machines use arithmetic, and you can most likely assume this.

2. Operator precedence can be tricky! For example:

\[ 1 \ll 2 + 3 \ll 4 \text{ means } 1 \ll (2+3) \ll 4 \text{ because addition and subtraction have higher precedence than shifts! Always use parentheses to be sure:} \]

\[ (1 \ll 2) + (3 \ll 4) \]
Recap

- **Recap**: Integer Representations
- Truncating and Expanding
- Bitwise Boolean Operators and Masks
- **Demo 1**: Courses
- **Break**: Announcements
- **Demo 2**: Powers of 2
- Bit Shift Operators

**Next time**: C strings