CS107 Lecture 3
Bits and Bytes; Bitwise Operators

reading:
Bryant & O’Hallaron, Ch. 2.1
Bits and Bytes So Far

• all data is ultimately stored in memory in binary
• When we declare an integer variable, under the hood it is stored in binary

```java
int x = 5; // really 0b0...0101 in memory!
```

• Until now, we only manipulate our integer variables in base 10 (e.g. increment, decrement, set, etc.)
• Today, we will learn about how to manipulate the underlying binary representation!
• This is useful for: more efficient arithmetic, more efficient storing of data, etc.
Lecture Plan

- Bitwise Operators
- Bitmasks
- **Demo 1**: Courses
- **Demo 2**: Practice and Powers of 2
- Bit Shift Operators
- **Demo 3**: Color Wheel
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
Lecture Plan

• Bitwise Operators
• Bitmasks
• Demo 1: Courses
• Demo 2: Practice and Powers of 2
• Bit Shift Operators
• Demo 3: Color Wheel
Now that we understand binary representations, how can we manipulate them at the bit level?
Bitwise 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.

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

<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>

& with 1 to let a bit through, & with 0 to zero out a bit
OR is a binary operator. The OR of 2 bits is 1 if either (or both) bits is 1.

```
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>

| with 1 to turn on a bit, | with 0 to let a bit go through |
NOT is a unary operator. The NOT of a bit is 1 if the bit is 0, or 1 otherwise.

\[ \text{output} = \neg a; \]

<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 *exactly* one of the bits is 1, or 0 otherwise.

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

<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>

^ with 1 to flip a bit, ^ with 0 to let a bit go through
Operators on Multiple Bits

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

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

**Note:** these are different from the logical operators AND (&&), OR (||) and NOT (!).
Operators on Multiple Bits

• When these 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>
</tbody>
</table>

This is different from logical AND (&&). The logical AND returns true if both are nonzero, or false otherwise. With &&, this would be 6 && 12, which would evaluate to true (1).
Operators on Multiple Bits

- When these 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>
</tbody>
</table>

This is different from logical OR (||). The logical OR returns true if either are nonzero, or false otherwise. With ||, this would be 6 || 12, which would evaluate to true (1).
Operators on Multiple Bits

- When these 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 ---- 1110</td>
<td>~ 1100 ---- 0011</td>
</tr>
<tr>
<td></td>
<td>1100</td>
<td>1110</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1110</td>
<td>1010</td>
<td></td>
</tr>
</tbody>
</table>

This is different from logical NOT (!). The logical NOT returns true if this is zero, and false otherwise. With !, this would be !12, which would evaluate to **false** (0).
Lecture Plan

• Bitwise Operators
• **Bitmasks**
• **Demo 1:** Courses
• **Demo 2:** Practice and Powers of 2
• Bit Shift Operators
• **Demo 3:** Color Wheel
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>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS161</td>
<td>CS109</td>
<td>CS103</td>
<td>CS110</td>
<td>CS107</td>
<td>CS106X</td>
<td>CS106B</td>
<td>CS106A</td>
</tr>
</tbody>
</table>
Bit Vectors and Sets

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

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

\[
\begin{align*}
00100011 \lor 01100001 &= 01100011
\end{align*}
\]
• How do we find the intersection of two sets of courses taken? Use AND:

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

\[
00100011 \\
\& \quad 01100001 \\
\hline
00100001
\]
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?

```plaintext
00100011 | 00001000
--------  
00101011
```

<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>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

00100011

00001000

--------

00101011
Bit Masking

#define CS106A 0x1 /* 0000 0001 */
#define CS106B 0x2 /* 0000 0010 */
#define CS106X 0x4 /* 0000 0100 */
#define CS107 0x8 /* 0000 1000 */
#define CS110 0x10 /* 0001 0000 */
#define CS103 0x20 /* 0010 0000 */
#define CS109 0x40 /* 0100 0000 */
#define CS161 0x80 /* 1000 0000 */

cchar myClasses = ...;
myClasses = myClasses | CS107; // Add CS107
#define CS106A 0x1 /* 0000 0001 */
#define CS106B 0x2 /* 0000 0010 */
#define CS106X 0x4 /* 0000 0100 */
#define CS107  0x8 /* 0000 1000 */
#define CS110  0x10 /* 0001 0000 */
#define CS103  0x20 /* 0010 0000 */
#define CS109  0x40 /* 0100 0000 */
#define 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
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1</th>
<th></th>
<th></th>
<th></th>
<th>1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CS161</td>
<td>CS109</td>
<td>CS103</td>
<td>CS110</td>
<td>CS107</td>
<td>CS106X</td>
<td>CS106B</td>
<td>CS106A</td>
<td></td>
</tr>
</tbody>
</table>

00100011
\& 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?

```
00100011 & 11011111 ---- ---- 00000011
```

<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>
<tbody>
<tr>
<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>

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

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

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

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

<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>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

00100011
& 00001000

00000000

char myClasses = ...;
if (!(myClasses & CS107)) {...
// not taken CS107!
• Example: how do we check if we’ve *not* taken CS107?

```
char myClasses = ...;
if (((myClasses & CS107) ^ CS107) {...
  // not taken CS107!
```
Bitwise Operator Tricks

• | with 1 is useful for turning select bits on
• & with 0 is useful for turning select bits off
• | is useful for taking the union of bits
• & is useful for taking the intersection of bits
• ^ is useful for flipping select bits
• ~ is useful for flipping all bits
Demo: Bitmasks and GDB
Lecture Plan

• Bitwise Operators
• Bitmasks
• **Demo 1**: Courses
• **Demo 2**: Practice and Powers of 2
• Bit Shift Operators
• **Demo 3**: Color Wheel
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 \( j \)?

```c
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.
  \[
  j \mid 0xff
  \]

- **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.
  \[
  j \^ \sim 0xff
  \]
Without using loops, how can we detect if a binary number is a power of 2? What is special about its binary representation and how can we leverage that?
Demo: Powers of 2
• Bitwise Operators
• Bitmasks
• **Demo 1**: Courses
• **Demo 2**: Practice and Powers of 2
• **Bit Shift Operators**
• **Demo 3**: Color Wheel
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 the end are lost.

```plaintext
x << k;  // evaluates to x shifted to the left by k bits
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`
Right Shift (>>) 

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

```
x >> k;       // evaluates to x shifted to the right by k bits
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;  // 0000 0000 0000 0010
x >>= 1;      // 0000 0000 0000 0001
printf("%d\n", x); // 1
```
Right Shift (\(\gg\))

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

\[
x \gg k; \quad // \text{evaluates to } x \text{ shifted to the right by } k \text{ bit}
x >>= k; \quad // \text{shifts } x \text{ to the right by } k \text{ bits}
\]

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

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

```c
short x = -2; // 1111 1111 1111 1110
x >>= 1;      // 0111 1111 1111 1111
printf("%d\n", x); // 32767!
```
The RIGHT SHIFT operator shifts a bit pattern a certain number of positions to the right. Bits shifted off the end are lost.

```c
x >> k; // evaluates to x shifted to the right by k bit
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!
Right Shift (>>>)

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

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

**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 >>= 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 the end are lost.

\[
\begin{align*}
\text{x >> k;} & \quad \text{// evaluates to x shifted to the right by k bit} \\
\text{x >>= k;} & \quad \text{// shifts x to the right by k bits}
\end{align*}
\]

**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:

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

\[(1 \ll 2) + (3 \ll 4)\]
The default type of a number literal in your code is an `int`.

Let’s say you want a long with the index-32 bit as 1:

```java
long num = 1 << 32;
```

This doesn’t work! 1 is by default an `int`, and you can’t shift an int by 32 because it only has 32 bits. You must specify that you want 1 to be a `long`.

```java
long num = 1L << 32;
```
Lecture Plan

• Bitwise Operators
• Bitmasks

• **Demo 1:** Courses

• **Demo 2:** Practice and Powers of 2

• Bit Shift Operators

• **Demo 3:** Color Wheel
Demo: Color Wheel
Recap

• Bitwise Operators
• Bitmasks
• **Demo 1:** Courses
• **Demo 2:** Practice and Powers of 2
• Bit Shift Operators
• **Demo 3:** Color Wheel

**Next time:** *How can a computer represent and manipulate more complex data like text?*