Last Time

• Signed Integers
• Overflow
• Casting and Combining Types
Plan For Today

• Casting and Combining Types (cont’d.)
• Bitwise Operators
• Bitmasks
• Bit Shift Operators

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli's Stanford CS107 class
Good news, everyone!

• Labs start this week. Check your section assignment.

• Assg1 will be out on Oct 16 (due Oct 26)

• From now on, I will be having my office hour every Thursday from 2pm to 3pm.
New: COMP201 Slack Workspace

https://join.slack.com/t/comp201-winter-2020/signup
Slack Developer Community Code of Conduct

This code of conduct governs Slack Platform's Community events and discussions.

Introduction

• Diversity and inclusion make our community strong. We encourage participation from the most varied and diverse backgrounds possible and want to be very clear about where we stand.
• Our goal is to maintain a safe, helpful and friendly community for everyone, regardless of experience, gender identity and expression, sexual orientation, disability, personal appearance, body size, race, ethnicity, age, religion, nationality, or other defining characteristic.
• This code and related procedures apply to unacceptable behavior occurring in all community venues, including behavior outside the scope of community activities — online and in-person — as well as in all one-on-one communications, and anywhere such behavior has the potential to adversely affect the safety and well-being of community members.

Expected Behavior

• Be welcoming.
• Be kind.
• Look out for each other.

Unacceptable Behavior

• Conduct or speech which might be considered sexist, racist, homophobic, transphobic, ableist or otherwise discriminatory or offensive in nature.
  • Do not use unwelcome, suggestive, derogatory or inappropriate nicknames or terms.
  • Do not show disrespect towards others. (Jokes, innuendo, dismissive attitudes.)
• Intimidation or harassment (online or in-person). Please read the Citizen Code of Conduct for how we interpret harassment.
• Disrespect towards differences of opinion.
• Inappropriate attention or contact. Be aware of how your actions affect others. If it makes someone uncomfortable or feels unsafe, it’s wrong.

https://join.slack.com/t/comp201-winter-2020/signup
Lecture Plan

- Casting and Combining Types (cont’d.)
- Bitwise Operators
- Bitmasks
- 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.
Expanding Bit Representation

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
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
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
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
```
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
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
```
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

```c
long sizeof(type);
```

// Example
```c
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.
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

```
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.
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

• Casting and Combining Types (cont’d.)
• Bitwise Operators
• Bitmasks
• Bit Shift Operators
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 (&)

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 (|)**

OR is a binary operator. The OR of 2 bits is 1 if either (or both) bits is 1.

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

<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 (\sim)

NOT is a unary operator. The NOT of a bit is 1 if the bit is 0, or 1 otherwise.

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

<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 (^)

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.

```
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>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><strong>0110</strong> &amp; <strong>1100</strong></td>
<td><strong>0110</strong></td>
<td><strong>0110</strong> ^ <strong>1100</strong></td>
<td><strong>0110</strong> ^ <strong>1100</strong></td>
<td>~ <strong>1100</strong></td>
</tr>
<tr>
<td></td>
<td><strong>0100</strong></td>
<td><strong>1110</strong></td>
<td><strong>1010</strong></td>
<td><strong>0011</strong></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:

  - **AND**
    
    \[
    \begin{array}{c}
    0110 \\
    \& 1100 \\
    \hline
    0100
    \end{array}
    \]

  - **OR**
    
    \[
    \begin{array}{c}
    0110 \\
    \mid 1100 \\
    \hline
    1110
    \end{array}
    \]

  - **XOR**
    
    \[
    \begin{array}{c}
    0110 \\
    \^ 1100 \\
    \hline
    1010
    \end{array}
    \]

  - **NOT**
    
    \[
    \begin{array}{c}
    ~ 1100 \\
    \hline
    0011
    \end{array}
    \]

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</td>
<td>0110</td>
<td>0110 ^ 1100</td>
<td>~ 1100</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>0100</td>
<td>1110</td>
<td>1010</td>
<td>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>
</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

- Casting and Combining Types (cont’d.)
- Bitwise Operators
- Bitmasks
- Bit Shift Operators
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></th>
<th>COMP100</th>
<th>COMP106</th>
<th>COMP132</th>
<th>COMP201</th>
<th>COMP202</th>
<th>COMP291</th>
<th>COMP301</th>
<th>COMP302</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>
<td>1</td>
</tr>
</tbody>
</table>
Bit Vectors and Sets

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

\[
\begin{array}{cccccccc}
0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
\end{array}
\]

\begin{array}{cccccccc}
COMP100 & COMP106 & COMP132 & COMP201 & COMP202 & COMP291 & COMP301 & COMP302 \\
\end{array}

\[
\begin{array}{c}
00100011 \\
| 01100001 \\
\end{array}
\]

\[
\begin{array}{c}
-------- \\
01100011 \\
\end{array}
\]
• How do we find the intersection of two sets of courses taken? Use AND:

\[
\begin{array}{cccccccc}
0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
\end{array}
\]

COMP100   COMP106   COMP132   COMP201   COMP202   COMP291   COMP301   COMP302

\[
\begin{array}{c}
00100011 \\
\& 01100001 \\
\hline
00100001
\end{array}
\]
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 COMP202?

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMP100</td>
<td>COMP106</td>
<td>COMP132</td>
<td>COMP201</td>
<td>COMP202</td>
<td>COMP291</td>
<td>COMP301</td>
<td>COMP302</td>
<td></td>
</tr>
</tbody>
</table>

00100011  
|           | 00001000     |
|           | -------------|

00101011
Bit Masking

#define COMP100 0x1 /* 0000 0001 */
#define COMP106 0x2 /* 0000 0010 */
#define COMP132 0x4 /* 0000 0100 */
#define COMP201 0x8 /* 0000 1000 */
#define COMP202 0x10 /* 0001 0000 */
#define COMP291 0x20 /* 0010 0000 */
#define COMP301 0x40 /* 0100 0000 */
#define COMP302 0x80 /* 1000 0000 */

char myClasses = ...;
myClasses = myClasses | COMP201; // Add COMP201
Bit Masking

#define COMP100 0x1 /* 0000 0001 */
#define COMP106 0x2 /* 0000 0010 */
#define COMP132 0x4 /* 0000 0100 */
#define COMP201 0x8 /* 0000 1000 */
#define COMP202 0x10 /* 0001 0000 */
#define COMP291 0x20 /* 0010 0000 */
#define COMP301 0x40 /* 0100 0000 */
#define COMP302 0x80 /* 1000 0000 */

char myClasses = ...;
myClasses |= COMP201; // Add COMP201
Bit Masking

• **Example:** how do we update our bit vector to indicate we’ve *not* taken COMP132?

```
char myClasses = ...;
myClasses = myClasses & ~COMP132;  // Remove COMP132
```
Bit Masking

- **Example:** how do we update our bit vector to indicate we’ve *not* taken COMP132?

```c
char myClasses = ...;
myClasses &= ~COMP132;  // Remove COMP132
```
Bit Masking

• Example: how do we check if we've taken COMP301?

```
char myClasses = ...;
if (myClasses & COMP301) {
    // taken COMP301!
```
Bit Masking

- Example: how do we check if we've *not* taken COMP201?

<table>
<thead>
<tr>
<th></th>
<th>COMP100</th>
<th>COMP106</th>
<th>COMP132</th>
<th>COMP201</th>
<th>COMP202</th>
<th>COMP291</th>
<th>COMP301</th>
<th>COMP302</th>
</tr>
</thead>
<tbody>
<tr>
<td></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>

\[
\begin{array}{c}
00100011 \\
& 00010000 \\
\hline \\
00000000
\end{array}
\]

```c
char myClasses = ...;
if (!(myClasses & COMP201)) {...
    // not taken COMP201!
```
Bit Masking

- **Example:** how do we check if we've *not* taken COMP201?

```
char myClasses = ...;
if ((myClasses & COMP201) ^ COMP201) {...
  // not taken COMP201!
```
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
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.
  
  $$j ^ ~0xff$$
Powers of 2

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
Lecture Plan

- Casting and Combining Types (cont’d.)
- Bitwise Operators
- Bitmasks
- Bit Shift Operators
Left Shift (<<)

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.

\[
x \ll k; \quad // \text{evaluates to } x \text{ shifted to the left by } k \text{ bits}
\]
\[
x \ll= k; \quad // \text{shifts } x \text{ to the left by } k \text{ bits}
\]

8-bit examples:

\[
00110111 \ll 2 \text{ results in } 11011100
\]
\[
01100011 \ll 4 \text{ results in } 00110000
\]
\[
10010101 \ll 4 \text{ results in } 01010000
\]
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.

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

**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
```
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?

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

```c
short x = -2; \quad \text{// 1111 \hspace{1em} 1111 \hspace{1em} 1111 \hspace{1em} 1110}
x \gg= 1; \quad \text{// 0111 \hspace{1em} 1111 \hspace{1em} 1111 \hspace{1em} 1111}
printf("%d\n", x); \quad \text{// 32767!}
```
Right Shift (\texttt{\textgreater\textgreater\textgreater})

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

**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 ($\gg$)

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?

**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
```
Right Shift (\texttt{\textgreater\textgreater\textgreater})

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

\begin{verbatim}
  x >> k;    // evaluates to x shifted to the right by k bit
  x >>= k;  // shifts x to the right by k bits
\end{verbatim}

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

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

\begin{verbatim}
short x = -2;  // 1111 1111 1111 1110
x >>= 1;       // 1111 1111 1111 1111
printf("%d\n", x); // -1!
\end{verbatim}
Right Shift (>>)

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!
Shift Operation Pitfalls

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:

   \[(1 \ll 2) + (3 \ll 4)\]
Bit Operator Pitfalls

• 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:

```cpp
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.

```cpp
long num = 1L << 32;
```
Recap

• Bitwise Operators
• Bitmasks
• Bit Shift Operators

Next time: How can a computer represent floating point numbers?