CS107 Lecture 3
Bits and Bytes; Bitwise Operators

reading:
Bryant & O'Hallaron, Ch. 2.1

<u>CS107 Topic 1</u>: How can a computer represent integer numbers?

CS107 Topic 1

How can a computer represent integer numbers?

Why is answering this question important?

- Helps us understand the limitations of computer arithmetic (last time)
- Shows us how to more efficiently perform arithmetic (today)
- Shows us how we can encode data more compactly and efficiently (today)

assign1: implement 3 programs that manipulate binary representations to (1) work around the limitations of arithmetic with addition, (2) simulate an evolving colony of cells, and (3) print Unicode text to the terminal.

Learning Goals

- Learn about the bitwise C operators and how to use them to manipulate bits
- Understand when to use one bitwise operator vs. another in your program
- Get practice with writing programs that manipulate binary representations

Lecture Plan

- **Recap**: Integer Representations
- Casting, Truncation and Expansion
- Bitwise Operators
- Bitmasks
- **Demo 1:** Courses
- **Demo 2:** Practice and Powers of 2
- Bit Shift Operators

Lecture Plan

- **Recap: Integer Representations**
- Casting, Truncation and Expansion
- Bitwise Operators
- Bitmasks
- **Demo 1:** Courses
- **Demo 2:** Practice and Powers of 2
- Bit Shift Operators

Base 10 vs. Binary vs. Hex

- Let's take a byte (8 bits):

`165`

Base-10: Human-readable,
but cannot easily interpret on/off bits

`0b10100101`

Base-2: Yes, computers use this,
but not human-readable

`0xa5`

Base-16: Easy to convert to Base-2,
More "portable" as a human-readable format
(fun fact: a half-byte is called a nibble or nybble)

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!
```

- Unsigned numbers store the direct binary representation of its value
- Signed numbers use **two's complement** to store its positive/negative/0 value
- Overflow occurs when we exceed the the minimum or maximum value of the bit representation – it can cause some funky (and funny) bugs!

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

- **Recap**: Integer Representations
- **Casting, Truncation and Expansion**
- Bitwise Operators
- Bitmasks
- **Demo 1:** Courses
- **Demo 2:** Practice and Powers of 2
- Bit Shift Operators

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.**

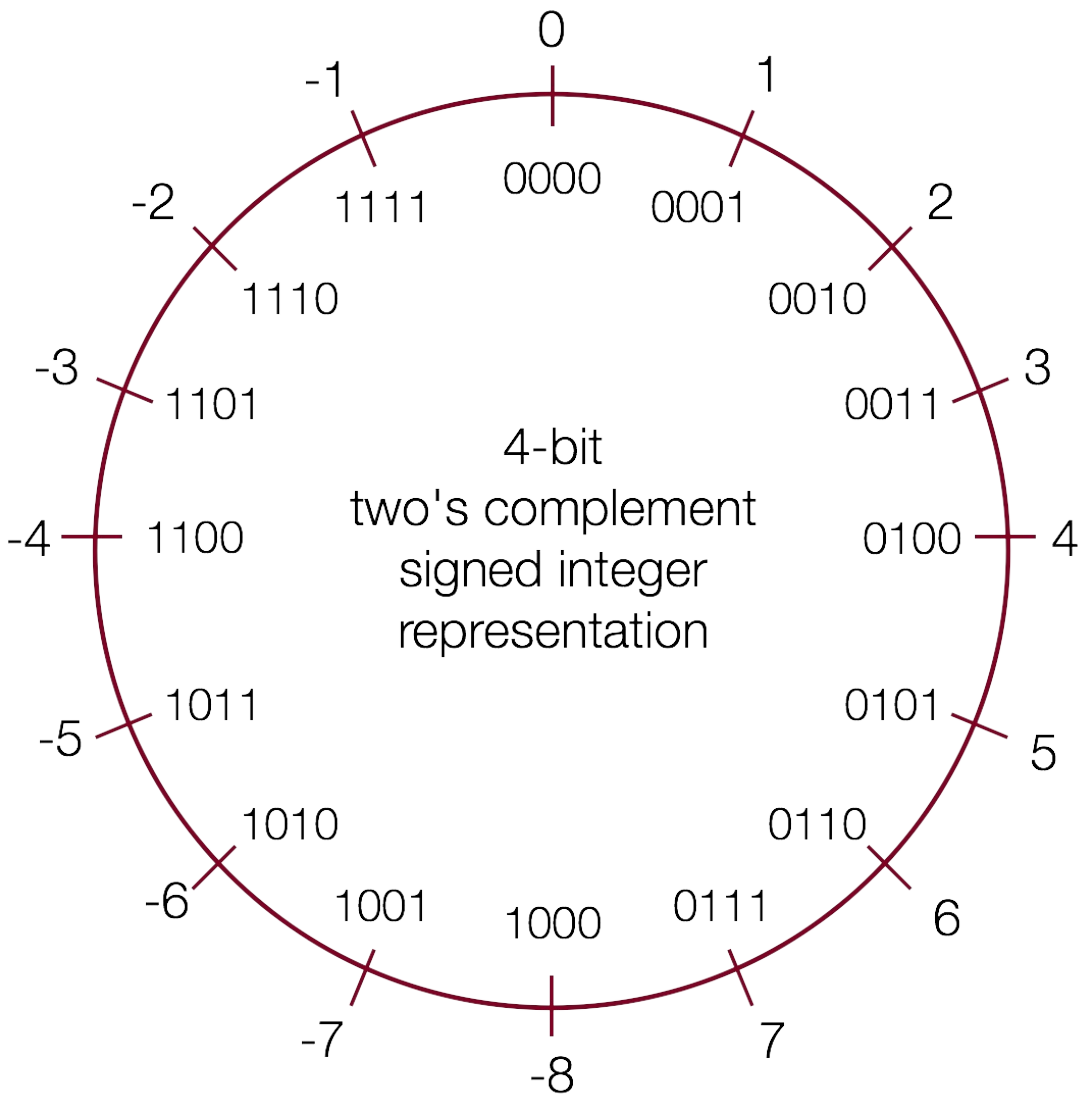
```
int v = -12345;
unsigned int uv = v;
printf("v = %d, uv = %u\n", v, uv);
```

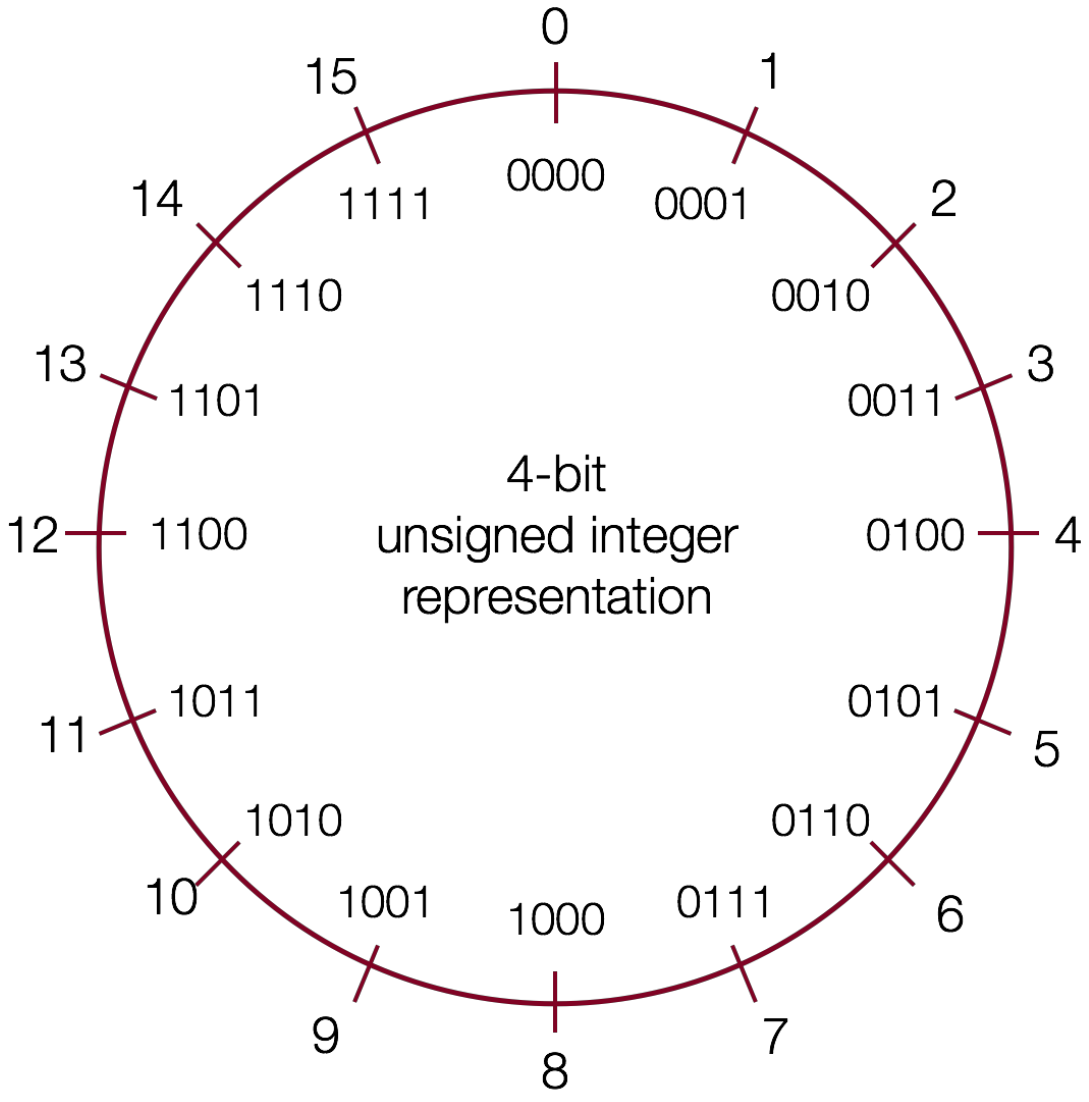
This prints out: `"v = -12345, uv = 4294954951"`. **Why?**

The bit representation for -12345 is 0b**11111111111111111001111111000111.**

If we treat this binary representation as a positive number, it's *huge*!

Casting


0
0000
1
0001
2
0010
3
0011
4
0100
5
0101
6
0110
7
0111
-8
1000
-7
1001
-6
1010
-5
1011
-4
1100
-3
1101
-2
1110
-1
1111
4-bit
two's complement
signed integer
representation


0
0000
1
0001
2
0010
3
0011
4
0100
5
0101
6
0110
7
0111
8
1000
9
1001
10
1010
11
1011
12
1100
13
1101
14
1110
15
1111
4-bit
unsigned integer
representation

Casting

You can cast something to another type by putting that type in parentheses in front of the value:

```
int v = -12345;
...(unsigned int)v...
```

You can also use the **U** suffix after a number literal to treat it as unsigned:

```
-12345U
```

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.

`Expression`	Comparison Type?	Evaluates To?	Mathematically correct?
`0 == 0U`			

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.

`Expression`	Comparison Type?	Evaluates To?	Mathematically correct?
`0 == 0U`	Unsigned	true	yes

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.

`Expression`	Comparison Type?	Evaluates To?	Mathematically correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`			

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.

`Expression`	Comparison Type?	Evaluates To?	Mathematically correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes

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.

`Expression`	Comparison Type?	Evaluates To?	Mathematically correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes
`-1 < 0U`			

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.

`Expression`	Comparison Type?	Evaluates To?	Mathematically correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes
`-1 < 0U`	Unsigned	false	No!

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.

Expression	Comparison Type?	Evaluates To?	Mathematically correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes
`-1 < 0U`	Unsigned	false	No!
`2147483647 > -2147483648`			

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.

`Expression`	Comparison Type?	Evaluates To?	Mathematically correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes
`-1 < 0U`	Unsigned	false	No!
`2147483647 > -2147483648`	Signed	true	yes

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.

Expression	Comparison Type?	Evaluates To?	Mathematically correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes
`-1 < 0U`	Unsigned	false	No!
`2147483647 > -2147483648`	Signed	true	yes
`2147483647U > -2147483648`			

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.

Expression	Comparison Type?	Evaluates To?	Mathematically correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes
`-1 < 0U`	Unsigned	false	No!
`2147483647 > -2147483648`	Signed	true	yes
`2147483647U > -2147483648`	Unsigned	false	No!

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.

Expression	Comparison Type?	Evaluates To?	Mathematically correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes
`-1 < 0U`	Unsigned	false	No!
`2147483647 > -2147483648`	Signed	true	yes
`2147483647U > -2147483648`	Unsigned	false	No!
`-1 > -2`			
`(unsigned)-1 > -2`			

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.

Expression	Comparison Type?	Evaluates To?	Mathematically correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes
`-1 < 0U`	Unsigned	false	No!
`2147483647 > -2147483648`	Signed	true	yes
`2147483647U > -2147483648`	Unsigned	false	No!
`-1 > -2`	Signed	true	yes
`(unsigned)-1 > -2`	Unsigned	true	yes

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*.

```
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*.

```
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*.

```
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 (or a variable itself) as a parameter and returns the size of that type, in bytes.

Lecture Plan

- **Recap**: Integer Representations
- Casting, Truncation and Expansion
- **Bitwise Operators**
- Bitmasks
- **Demo 1:** Courses
- **Demo 2:** Practice and Powers of 2
- Bit Shift Operators

Now that we understand values are really stored in binary, 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.

```
output = a & b;
```

a	b	output
0	0	0
0	1	0
1	0	0
1	1	1

& 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.

```
output = a | b;
```

a	b	output
0	0	0
0	1	1
1	0	1
1	1	1

| with 1 to turn on a bit, | with 0 to let a bit go through

Not (~)

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

```
output = ~a;
```

a	output
0	1
1	0

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;
```

a	b	output
0	0	0
0	1	1
1	0	1
1	1	0

^ 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:

AND

```
  0110
& 1100
  ----
  0100
```

OR

```
  0110
| 1100
  ----
  1110
```

XOR

```
  0110
^ 1100
  ----
  1010
```

NOT

```
~ 1100
  ----
  0011
```

Demo: Bits Playground

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

```
  0110
& 1100
  ----
  0100
```

OR

```
  0110
| 1100
  ----
  1110
```

XOR

```
  0110
^ 1100
  ----
  1010
```

NOT

```
~ 1100
  ----
  0011
```

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

```
  0110
& 1100
  ----
  0100
```

OR

```
  0110
| 1100
  ----
  1110
```

XOR

```
  0110
^ 1100
  ----
  1010
```

NOT

```
~ 1100
  ----
  0011
```

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:

AND

```
  0110
& 1100
  ----
  0100
```

OR

```
  0110
| 1100
  ----
  1110
```

XOR

```
  0110
^ 1100
  ----
  1010
```

NOT

```
~ 1100
  ----
  0011
```

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:

AND

```
  0110
& 1100
  ----
  0100
```

OR

```
  0110
| 1100
  ----
  1110
```

XOR

```
  0110
^ 1100
  ----
  1010
```

NOT

```
~ 1100
  ----
  0011
```

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

- **Recap**: Integer Representations
- Casting, Truncation and Expansion
- Bitwise Operators
- **Bitmasks**
- **Demo 1:** Courses
- **Demo 2:** Practice and Powers of 2
- Bit Shift Operators
- **Demo 3:** Color Wheel

Bitmasks

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.

Motivating Example: Bit vectors

Bit Vectors and Sets

Instead of using arrays of e.g., Booleans in our programs, sometimes it's beneficial to store that information in bits instead – more compact.

- **Example:** we can represent current courses taken using a **`char`** and manipulate its contents using bit operators.

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS110	CS107	CS106X	CS106B	CS106A

Bit Vectors and Sets

CS161	CS109	CS103	CS110	CS107	CS106X	CS106B	CS106A
0	0	1	0	0	0	1	1

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

```
  00100011
| 01100001
--------
  01100011
```

Bit Vectors and Sets

CS161	CS109	CS103	CS110	CS107	CS106X	CS106B	CS106A
0	0	1	0	0	0	1	1

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

```
  00100011
& 01100001
  --------
  00100001
```

Bit Masking

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

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS110	CS107	CS106X	CS106B	CS106A

```
  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 */

char myClasses = ...;
myClasses = myClasses | CS107;   // Add CS107
```

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 */

char myClasses = ...;
myClasses |= CS107;     // Add CS107
```

Bit Masking

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

CS161	CS109	CS103	CS110	CS107	CS106X	CS106B	CS106A
0	0	1	0	0	0	1	1

```
  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?

CS161	CS109	CS103	CS110	CS107	CS106X	CS106B	CS106A
0	0	1	0	0	0	1	1

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

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

Bit Masking

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

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS110	CS107	CS106X	CS106B	CS106A

```
  00100011
& 00000010
---------
  00000010
```

```
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	1	1
CS161	CS109	CS103	CS110	CS107	CS106X	CS106B	CS106A

```
  00100011
& 00001000
  --------
  00000000
```

```
char myClasses = ...;
if (!(myClasses & 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

Introducing GDB

Is there a way to step through the execution of a program and print out its values as it's running? E.g., to view binary representations? **Yes!**

The GDB Debugger

- GDB is a **command-line debugger**, a text-based debugger with similar functionality to other debuggers you may have used, such as in Qt Creator
- It lets you put **breakpoints** at specific places in your program to pause there
- It lets you step through execution line by line
- It lets you print out values of variables in various ways (including binary)
- It lets you track down where your program crashed
- And much, much more!

GDB is essential to your success in CS107 this quarter! We'll be building our familiarity with GDB over the course of the quarter.

gdb on a program

- `gdb myprogram` run gdb on executable
- `b` Set breakpoint on a function (e.g., `b main`) or line (`b 42`)
- `r 82` Run with provided args
- `n, s, continue` control forward execution (next, step into, continue)
- `p` print variable (`p varname`) or evaluated expression (`p 3L << 10`)
  - `p/t, p/x` binary and hex formats.
  - `p/d, p/u, p/c`
- `info` `args, locals`

Important: gdb does not run the current line until you hit “next”

Demo: Bitmasks and GDB

gdb: highly recommended

At this point, setting breakpoints/stepping in gdb may seem like overkill for what could otherwise be achieved by copious **printf** statements.

However, gdb is incredibly useful for assign1 (and all assignments):

- A fast "C interpreter": p + <expression>
  - Sandbox/try out ideas around bitshift operators, signed/unsigned types, etc.
  - Can print values out in binary!
  - Once you're happy, then make changes to your C file
- **Tip**: Open two terminal windows and SSH into myth in both
  - Keep one for emacs, the other for gdb/command-line
  - Easily reference C file line numbers and variables while accessing gdb
- **Tip**: Every time you update your C file, **make** and then rerun gdb.

Gdb takes practice! But the payoff is tremendous! ☺

Pre-Lab 1 Video Slides

Lecture Plan

- **Recap**: Integer Representations
- Casting, Truncation and Expansion
- Bitwise Operators
- Bitmasks
- **Demo 1:** Courses
- **Demo 2: Practice and Powers of 2**
- Bit Shift Operators

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
```

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

- **Recap**: Integer Representations
- Casting, Truncation and Expansion
- Bitwise Operators
- Bitmasks
- **Demo 1:** Courses
- **Demo 2:** Practice and Powers of 2
- **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 << 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 (>>)

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 bit
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 the end are lost.

```
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 >> 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!

```
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 >> 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!

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

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<<2 + 3<<4` means **`1 << (2+3) << 4`** because addition and subtraction have higher precedence than shifts! Always use parentheses to be sure:

   `(1<<2) + (3<<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:

```
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**.

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

Recap

- **Recap**: Integer Representations
- Casting, Truncation and Expansion
- Bitwise Operators
- Bitmasks
- **Demo 1:** Courses
- **Demo 2:** Practice and Powers of 2
- Bit Shift Operators

Lecture 3 takeaways: We can use bit operators like &, |, ~, <<, etc. to manipulate the binary representation of values. A number is a bit pattern that can be manipulated arithmetically or bitwise at your convenience!

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

Extra Practice

Color Wheel

- Another application for storing data efficiently in binary is representing **colors**.
- A color representation commonly consists of opacity (how transparent or opaque it is), and how much red/green/blue is in the color.
- **Key idea:** we can encode each of these in 1 byte, in a value from 0-255! Thus, an entire color can be represented in one 4-byte **integer**.

$$\mathbf{0x}\ \underbrace{\mathbf{42}}_{\text{Opacity}}\ \underbrace{\mathbf{53}}_{\text{Red}}\ \underbrace{\mathbf{01}}_{\text{Green}}\ \underbrace{\mathbf{44}}_{\text{Blue}}$$

Demo: Color Wheel

`color_wheel.c`

Hexadecimal and Truncation

For each initialization of x, what will be printed?

i. `x = 130;  // 0x82`

ii. `x = -132; // 0xff7c`

iii. `x = 25;  // 0x19`

```
short x = ______;
char cx = x;
printf("%d", cx);
```

Hexadecimal and Truncation

For each initialization of x, what will be printed?

-126 i. `x = 130;  // 0x82`

124 ii. `x = -132; // 0xff7c`

25 iii. `x = 25;  // 0x19`

```
short x = ______;
char cx = x;
printf("%d", cx);
```

Limits and Comparisons

2. Will the following `char` comparisons evaluate to `true` or `false`?

i. `-7 < 4` **`true`**

ii. `-7 < 4U` **`false`**

iii. `(char) 130 > 4` **`false`**

iv. `(char) -132 > 2` **`true`**

By default, numeric constants in C are signed ints, unless they are suffixed with u (unsigned) or L (long).

Bitwise Warmup

How can we use bitmasks + bitwise operators to…

`0b00001101`

1. …turn **on** a particular set of bits?

`0b00001101`

`0b00001111`

2. …turn **off** a particular set of bits?

`0b00001101`

`0b00001001`

3. …**flip** a particular set of bits?

`0b00001101`

`0b00001011`

Bitwise Warmup

How can we use bitmasks + bitwise operators to…

`0b00001101`

1. …turn **on** a particular set of bits? **OR**

```
  0b00001101
  0b00000010 |
-------------
  0b00001111
```

2. …turn **off** a particular set of bits? **AND**

```
  0b00001101
  0b11111011 &
-------------
  0b00001001
```

3. …**flip** a particular set of bits? **XOR**

```
  0b00001101
  0b00000110 ^
-------------
  0b00001011
```

More Exercises

Suppose we have a 64-bit number.

```
long x = 0b1010010;
```

How can we use bit operators, and the constant `1L` or `-1L` to…

- …design a mask that turns on the i-th bit of a number for any i (0, 1, 2, …, 63)?

- …design a mask that zeros out (i.e., turns off) the bottom i bits (and keeps the rest of the bits the same)?

More Exercises

Suppose we have a 64-bit number.

```
long x = 0b1010010;
```

How can we use bit operators, and the constant `1L` or `-1L` to…

- …design a mask that turns on the i-th bit of a number for any i (0, 1, 2, …, 63)?

x | (1L << i)

- …design a mask that zeros out (i.e., turns off) the bottom i bits (and keeps the rest of the bits the same)?

x & (-1L << i)