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

```
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	5
• Bitmasks	16
• Demo 1: Courses	29
• Demo 2: Practice and Powers of 2	30
Bit Shift Operators	36
• Demo 3: Color Wheel	47

• Live session

49

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	5
• Bitmasks	16
• Demo 1: Courses	29
• Demo 2: Practice and Powers of 2	30
Bit Shift Operators	36
• Demo 3: Color Wheel	47
• Live session	49

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.

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.

outp	ut = a	ן ט;
а	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.

out	tpu	t	=	~a;
O O .	- P -			ر

a	output
0	1
1	0

Exclusive Or (^)

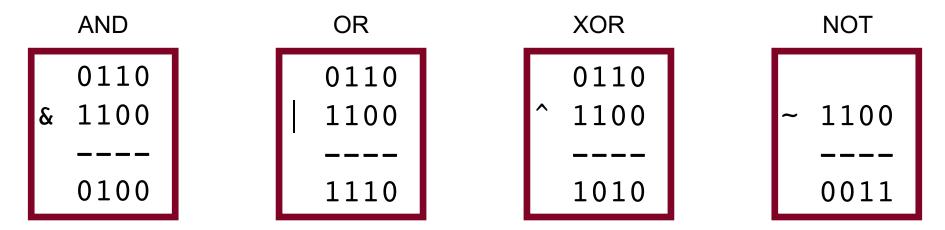
Exclusive Or (XOR) is a binary operator. The XOR of 2 bits is 1 if *exactly* one of the bits is 1, or 0 otherwise.

output	=	a	^	b;
--------	---	---	---	----

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

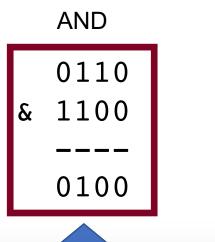
[^] with 1 to flip a bit, ^ with 0 to let a bit go through

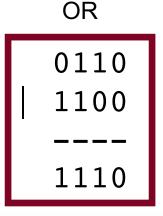
• When these operators are applied to numbers (multiple bits), the operator is applied to the corresponding bits in each number. For example:

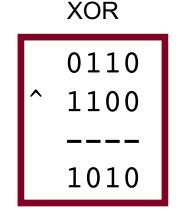


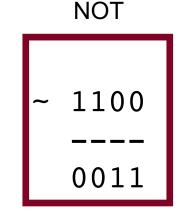
Note: these are different from the logical operators AND (&&), OR (||) and NOT (!).

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





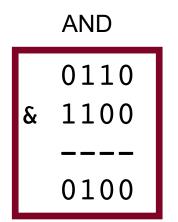


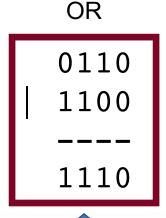


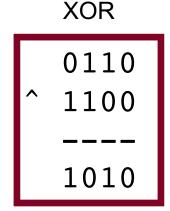


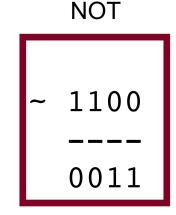
This is different from logical AND (&&). The logical AND returns true if both are nonzero, or false otherwise. With &&, this would be 6 && 12, which would evaluate to **true** (1).

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



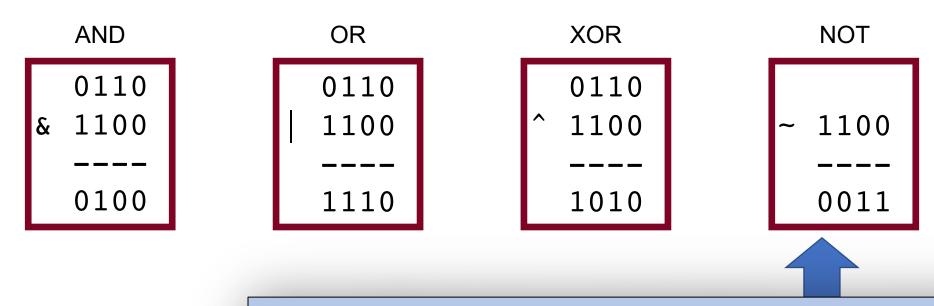






This is different from logical OR (||). The logical OR returns true if either are nonzero, or false otherwise. With ||, this would be 6 || 12, which would evaluate to **true** (1).

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



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

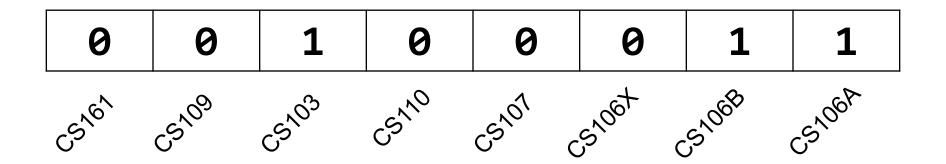
Bitwise Operators	<u></u>
• Bitmasks	16
• Demo 1: Courses	29
• Demo 2: Practice and Powers of 2	30
Bit Shift Operators	36
• Demo 3: Color Wheel	47

• Live session

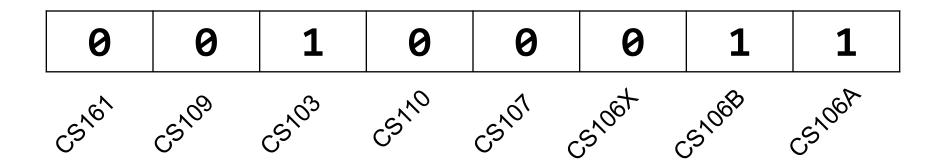
49

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.

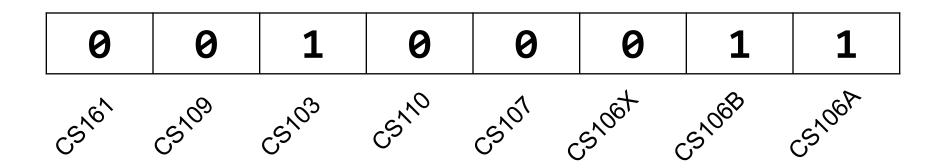


Bit Vectors and Sets



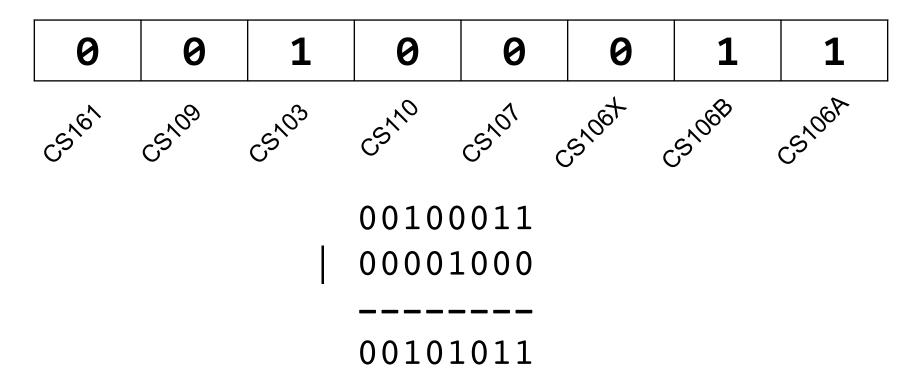
• How do we find the union of two sets of courses taken? Use OR:

Bit Vectors and Sets



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

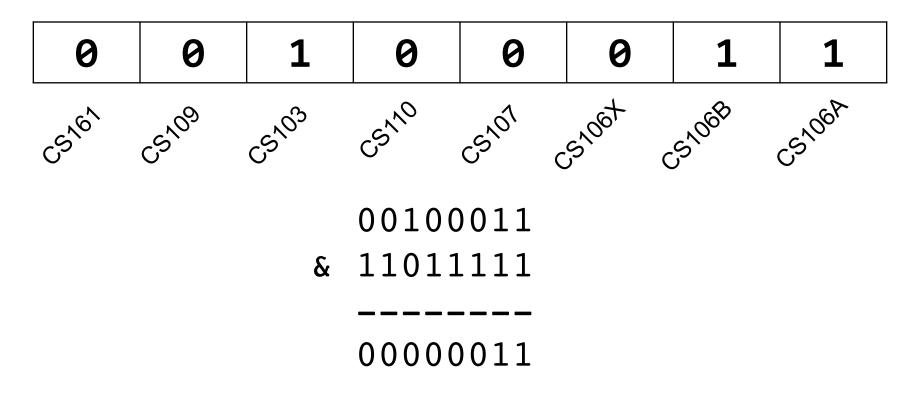
- We will frequently want to manipulate or isolate out specific bits in a larger collection of bits. A **bitmask** is a constructed bit pattern that we can use, along with bit operators, to do this.
- Example: how do we update our bit vector to indicate we've taken CS107?



```
#define CS106A 0x1
                 /* 0000 0001 */
                     /* 0000 0010 */
#define CS106B 0x2
#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 */
                     /* 1000 0000 */
#define CS161
             0x80
char myClasses = ...;
myClasses = myClasses | CS107; // Add CS107
```

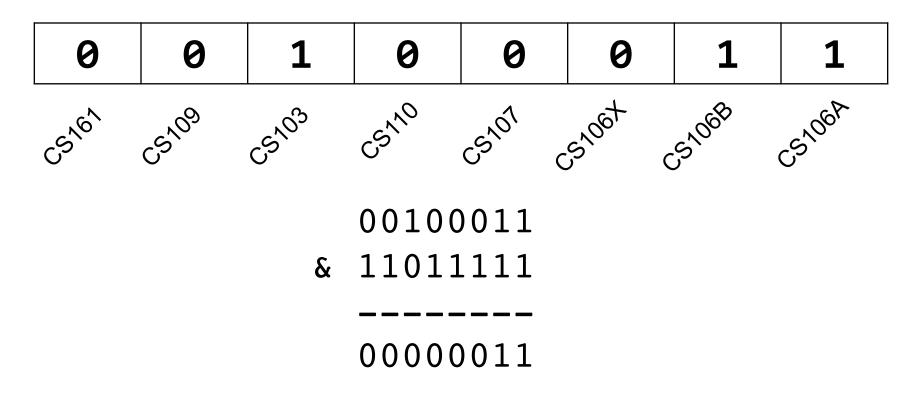
```
#define CS106A 0x1
                     /* 0000 0001 */
#define CS106B 0x2
                     /* 0000 0010 */
#define CS106X 0x4
                     /* 0000 0100 */
                     /* 0000 1000 */
#define CS107
             0x8
#define CS110 0x10
                     /* 0001 0000 */
                     /* 0010 0000 */
#define CS103
             0x20
#define CS109
                   /* 0100 0000 */
             0x40
#define CS161
             0x80
                     /* 1000 0000 */
char myClasses = ...;
myClasses |= CS107; // Add CS107
```

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



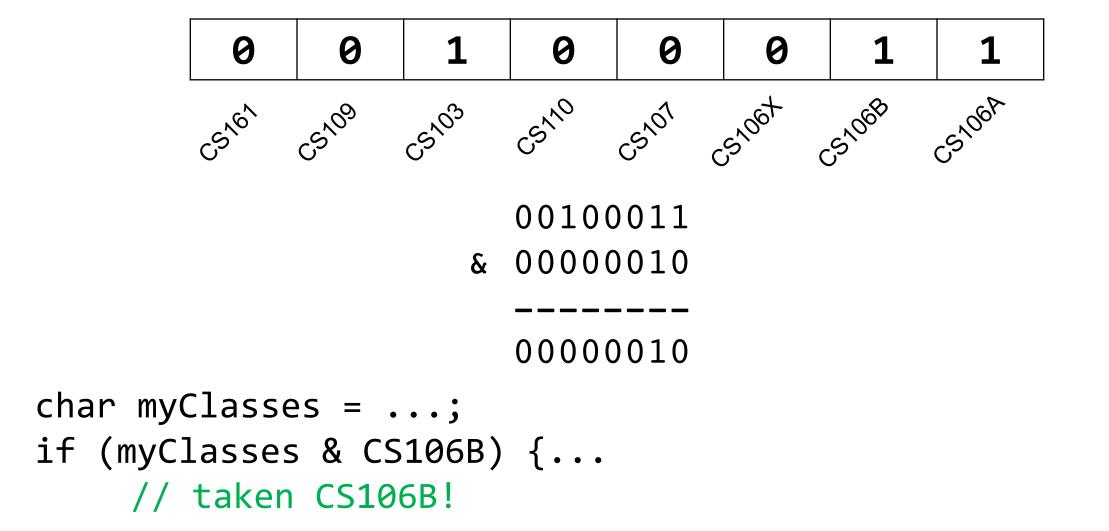
```
char myClasses = ...;
myClasses = myClasses & ~CS103; // Remove CS103
```

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

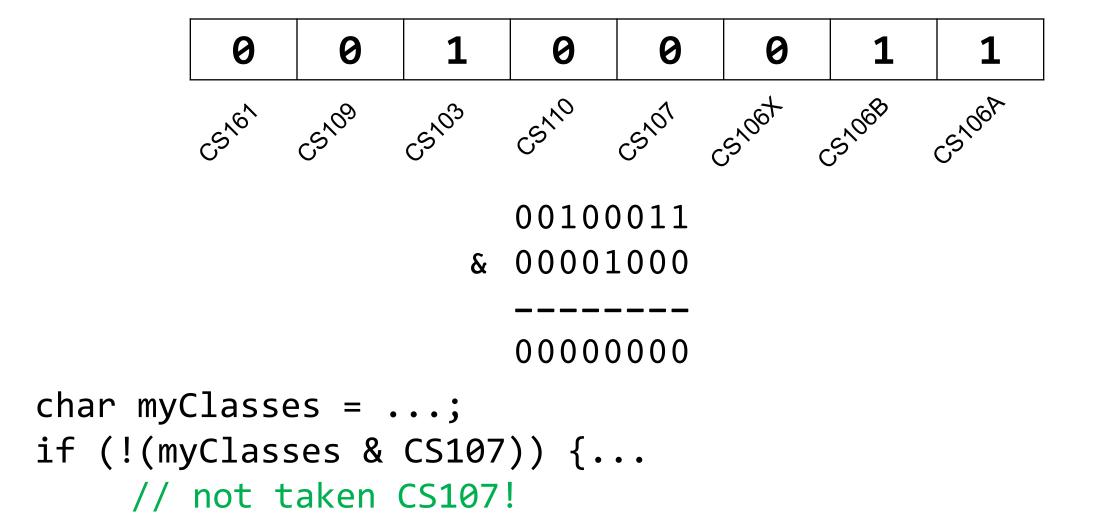


```
char myClasses = ...;
myClasses &= ~CS103; // Remove CS103
```

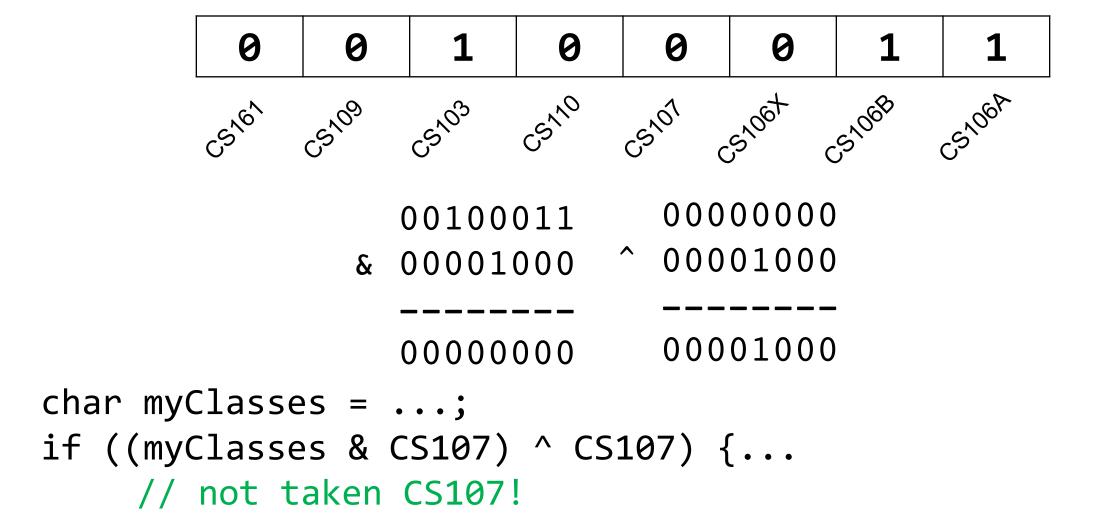
• Example: how do we check if we've taken CS106B?



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



• Example: how do we check if we've 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	5
• Bitmasks	16
• Demo 1: Courses	29
• Demo 2: Practice and Powers of 2	30
Bit Shift Operators	36
• Demo 3: Color Wheel	47

• Live session

49

• 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.

• **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.

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

Bitwise Operators	5
• Bitmasks	16
• Demo 1: Courses	29
• Demo 2: Practice and Powers of 2	30
• Bit Shift Operators	36
• Demo 3: Color Wheel	47
• Live session	49

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</pre>
```

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 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
```

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!
```

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!

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
```

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!
```

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 =
$$1L << 32$$
;

Lecture Plan

Bitwise Operators	5
• Bitmasks	16
• Demo 1: Courses	29
• Demo 2: Practice and Powers of 2	30
Bit Shift Operators	36
• Demo 3: Color Wheel	47

• Live session

49

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?

Additional Live Session Slides

Plan For Today

First 5 minutes: post questions or comments on Ed for what we should discuss

Lecture 3 takeaway: 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!

Bitwise Warmup

How can we use bitmasks + bitwise operators to...

0b00001101

- 1. ...turn **on** a particular set of bits?
- 2. ...turn **off** a particular set of bits?

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

0b00001101

0b00001101

0b00001101

0b000011<u>1</u>1

0b00001<u>0</u>01

0b00001<u>01</u>1

Bitwise Warmup

How can we use bitmasks + bitwise operators to...

0b00001101

- 1. ...turn **on** a particular set of bits? **OR**
- 2. ...turn **off** a particular set of bits? AND

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

0b00001101 0b00000010 0b00001101
0b11111011 &

0b00001101 0b00000110 ^

0b000011<u>1</u>1

0b00001001

0b00001<u>01</u>1

GDB as an Interpreter

- gdb live_session run gdb on live_session executable
- 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
- <enter>
 Execute last command again
 - q Quit gdb

Important When first launching gdb:

- Gdb is not running any program and therefore can't print variables
- It can still process operators on constants

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 \mid (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)?

$$x \& (-1L << i)$$



gdb on a program

```
    gdb live_session 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)</li>
    p/t, p/x binary and hex formats.
```

• info args, locals

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

• p/d, p/u, p/c

On your own

- Print a variable
- Print (in binary, then in hex) result of left-shifting 14 and 32 by 4 bits.
- Print (in binary, then in hex) result of subtracting 1 from 128

- 1 << 32
- Why is this zero? Compare with 1 << 31.
- Print in hex to make it easier to count zeros.

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! ©