CS107 Lecture 5
Bitwise Operators, Continued

reading:

Bryant & O'Hallaron, Ch. 2.1

Updates / Announcements

- Labs start this week!
  - Learning goals: practice with bit ops, exploring binary representations, GDB practice!
- In-class Polls and location check-in:
  - If you encounter any check-in issues, please see us after lecture, we're happy to help! We'll also display a list of any responses that didn't have a check-in at the end of lecture.
  - Please make sure to come see us (or come let us know you can't stick around, so we can make a note for later) – need to verify attendance to update any poll scores.
- Assign1 released
  - Post-assignment goals: familiarity with bitwise manipulation, working around limitations of computer arithmetic, understanding real-world software impacts of integer representations, practicing debugging strategies with GDB.
  - We are eager to help in helper hours! Focusing on helping you get unstuck to continue making progress, and helping you drive your own debugging. Building familiarity with debugging will pay off tremendously throughout the quarter!

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 week)

- Shows us how to more efficiently perform arithmetic (today)

- Shows us how we can encode data more compactly and efficiently (last time)

---

assign1: implement 3 programs that manipulate binary representations to (1) work around the limitations of arithmetic with addition, (2) simulate a chamber of gas particles, and (3) print Unicode text to the terminal.

Learning Goals

- Learn about the bit shift operators

- Understand when to use one bitwise operator vs. another in your program

- Get practice with writing programs that manipulate binary representations

Lecture Plan

- **Recap**: Bit Operators so far

- Bit Shift Operators

- **Example:** Powers of 2

- **Demo:** GDB

```
cp -r /afs/ir/class/cs107/lecture-code/lect5 .
```

Lecture Plan

- **Recap: Bit Operators so far**
- Bit Shift Operators
- **Example:** Powers of 2
- **Demo:** GDB

```
cp -r /afs/ir/class/cs107/lecture-code/lect5 .
```

Bits and Bytes So Far

```
// 1. Data is really stored in binary
int x = 5;   // really 0b00…0101 in memory!


// 2. We know what that binary representation is for integers
int y = -5;   // two's complement: 0b111...11011


// 3. We can use/manipulate a binary representation with bit ops
x |= 0x2;   // turn on the 2nd bit from the right: 0b00…0111


// 4. A variable and its binary representation are
// one and the same
printf("%d\n", x); // prints 7!
```

Bitwise OR (|)

| with 1 is useful for turning select bits on.

```
int x = 5; // 0b101

// Turn on the 2nd bit from the right
x |= 0x2; // 0b111
```

| is useful for taking the union of bits.

```
int x = 5;        // 0b00101
int y = 26;       // 0b11010
int z = x | y; // 0b11111
printf("%d\n", z); // 31
```

& with 0 is useful for turning select bits off.

```
int x = 5; // 0b101

// Turn off the 3rd bit from the right
x &= -5; // -5 is 0b111...1011
```

& is useful for taking the intersection of bits.

```
int x = 21;      // 0b10101
int y = 27;      // 0b11011
int z = x & y; // 0b10001
printf("%d\n", z); // 17
```

Bitwise XOR (^)

^ with 1 is useful for flipping select bits.

```
int x = 5; // 0b101

// Flip the 2nd bit from the right
x ^= 0x2; // 0b111
```

Bitwise NOT (~)

~ is useful for flipping all bits.

```
int x = 5; // 0b101

// Flip all bits
x = ~x;   // 0b11111...1010, which is -6

// Take two's complement (same as negating)
int y = ~x + 1; // same as -x
```

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	CS111	CS107	CS107E	CS106B	CS106A

Bit Vectors and Sets

```
#define CS106A 0x1     /* 0000 0001 */
#define CS106B 0x2     /* 0000 0010 */
#define CS107E 0x4     /* 0000 0100 */
#define CS107  0x8     /* 0000 1000 */
#define CS111  0x10    /* 0001 0000 */
#define CS103  0x20    /* 0010 0000 */
#define CS109  0x40    /* 0100 0000 */
#define CS161  0x80    /* 1000 0000 */


unsigned char myClasses = ...;
myClasses |= CS107;       // Add CS107
if (myClasses & CS106B) {...
      // taken CS106B!
```

Practice: Bit Masking

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

1. What operator is good for flipping certain bits?

2. What mask do we want?

3. How do we create that mask?

```
j ^ 0xff
```

Lecture Plan

- **Recap**: Bit Operators so far

- **Bit Shift Operators**

- **Example:** Powers of 2

- **Demo:** GDB

```
cp -r /afs/ir/class/cs107/lecture-code/lect5 .
```

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 does it fill in the new higher-order bits?

Right Shift (>>)

There are *two kinds* of right shifts, depending on the value and type you are shifting:

- *Unsigned numbers* are right-shifted by filling new high-order bits with 0s ("**logical right shift**").

- *Signed numbers* are right-shifted by filling new high-order bits with the most significant bit ("**arithmetic right shift**").

This way, the sign of the number (if applicable) is preserved!

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
```

```
unsigned short x = 2;   // 0000 0000 0000 0010
x >>= 1;                // 0000 0000 0000 0001
printf("%u\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
```

```
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
```

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

Shifting and Masking

Suppose we have a 32-bit number.   `int x = 0b1010010;`

How can we use bit operators to design a mask that turns on the i-th bit of a number for any i (0, 1, 2, …, 31)? (indexes are from right to left, LSB to MSB)

1. **What operator is good for turning on certain bits?**

2. **What mask do we want?**

3. How do we create that mask?

Shifting and Masking

Suppose we have a 32-bit number.   `int x = 0b1010010;`

How can we use bit operators to design a mask that turns on the i-th bit of a number for any i (0, 1, 2, …, 31)? (indexes are from right to left, LSB to MSB)

1. **What operator is good for turning on certain bits?**

2. **What mask do we want?**

3. **How do we create that mask? How would we write this operation in code?**

Respond on PollEv for #3:
pollev.com/cs107

What mask would help us turn on the i-th bit of a number?

Nobody has responded yet.

Hang tight! Responses are coming in.

Shifting and Masking

Suppose we have a 32-bit number.   `int x = 0b1010010;`

How can we use bit operators to design a mask that turns on the i-th bit of a number for any i (0, 1, 2, …, 31)? (indexes are from right to left, LSB to MSB)

1.   **What operator is good for turning on certain bits?**

2.   **What mask do we want?**

3.   **How do we create that mask? How would we write this operation in code?**

x | (1 << i)

Shifting and Masking

Suppose we have a 32-bit number.   `int x = 0b1010010;`

How can we use bit operators to design a mask that turns on the i-th bit of a number for any i (0, 1, 2, …, 31)? (indexes are from right to left, LSB to MSB)

x | (1 << i)

What if x is a 64-bit number (e.g. long) and i could be 0-63? It turns out there's a problem with this expression…

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;
```

Shifting and Masking

Suppose we have a 64-bit number.   `long x = 0b1010010;`

How can we use bit operators to design a mask that turns on the i-th bit of a number for any i (0, 1, 2, …, 63)? (indexes are from right to left, LSB to MSB)

x | (1L << i)

Number Literal Suffixes

U makes a literal unsigned, and **L** makes a literal a long.

```
int w = -5 >> 1;  // 0b1111...1101, -3
int x = -5U >> 1; // 0b0111...1101, 2147483645

long y = 1 << 32;  // 0! (technically undefined)
long z = 1L << 32; // 4294967296
```

1L << i

What does **1L << i** represent numerically?

A power of 2! Specifically, 2^i.

Lecture Plan

- **Recap**: Bit Operators so far
- **Bit Shift Operators**
- **Example: Powers of 2**
- **Demo:** GDB

```
cp -r /afs/ir/class/cs107/lecture-code/lect5 .
```

Powers of 2

Challenge: without using loops or math library functions, how could we detect whether a number is a power of 2?

What is true about a power of 2 but not other numbers? 🤔

Powers of 2

Key idea: A power of 2 minus 1 will have all bits below the original bit be 1, and everything else be 0. E.g.

0b10000 – 1 = 0b01111

0b100 – 1 = 0b011

Not true for other non-power-of-2 numbers (excluding 0):

0b10010 – 1 = 0b10001

Cool idea: _no bits overlap_ between a power of 2 and a power of 2 minus 1. How is this handy?

Demo: Powers of 2

is_power_of_2.c

Lecture Plan

- **Recap**: Bit Operators so far
- Bit Shift Operators
- **Example:** Powers of 2
- **Demo: GDB**

```
cp -r /afs/ir/class/cs107/lecture-code/lect5 .
```

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 Guide: cs107.stanford.edu/resources/gdb.html

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

bits_playground.c

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! ☺

Recap

- **Recap**: Bit Operators so far
- **Bit Operators + GDB Demo:** Courses
- **Demo 2:** Practice and Powers of 2
- Bit Shift Operators

Lecture 5 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

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)

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**.

0x 42 53 01 44

Opacity   Red   Green   Blue

Demo: Color Wheel

`color_wheel.c`

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
  ```

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 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 zeros out (i.e., turns off) the bottom i bits (and keeps the rest of the bits the same)?

x & (-1L << i)