



# CS107 Lecture 5

## Bitwise Operators

Reading: Bryant & O'Hallaron, Ch. 2.1

[Ed Discussion](#)

# 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.

$j \mid 0xff$

- **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.

$j \wedge \sim 0xff$

# Powers of 2

Without using loops, how can we detect if a number **num** is a power of 2? What's special about its binary representation and how can we take advantage of that?

# Demo: Powers of 2



# 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 right end of the number 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 right end of the number 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 right end of the number 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 right end of the number 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 with **Logical Right Shift**.

*Signed numbers* are right-shifted with **Arithmetic Right Shift**.

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

# Bit Operator Pitfall

- 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**. (This will come up in **assign1**.)

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

# Demo: Absolute Value



# Introducing GDB

Is there a way to step through the execution of a program and print out 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 arguments
- **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**
- **info locals**

**Important:** gdb does not run the current line until you execute "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 strategically placed **printf** statements.

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

- A fast "C interpreter": `p + <expression>`
  - Sandbox/try out ideas with bit shift operations, signed/unsigned types, etc.
  - Can print values out in binary!
  - Once you're happy, incorporate 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 huge!