CS107 Lecture 5
Bitwise Operators, Take II

Reading: Bryant & O'Hallaron, Ch. 2.1

Bitwise Manipulation Etudes

Bit masks needn't be applied to only **`char`**s (i.e., 8-bit values). They can be applied **`short`**s, **`int`**s, and **`long`**s as well.

Assuming that **`j`** and **`k`** are (32-bit) **`int`**s, **`m`** and **`n`** are **`unsigned long`**s:

Present a C statement that assigns **`k`** to be just the lowest byte of **`j`**.

Present an C statement that sets **`n`** to be the full bitwise inversion of **`m`**.

Present a C statement that sets **`k`** to **`j`**, where the first and last bytes of **`j`** have been bitwise-inverted.

Present an expression that evaluates to **`true`** iff **`n`** has two neighboring bits that are both 1.

Bitwise Manipulation Etudes

Bit masks needn't be applied to only **`char`**s (i.e., 8-bit values). They can be applied **`short`**s, **`int`**s, and **`long`**s as well.

Assuming that **`j`** and **`k`** are (32-bit) **`int`**s, **`m`** and **`n`** are **`unsigned long`**s:

Present a C statement that assigns **`k`** to be just the lowest byte of **`j`**.

```
k = j & 0xFF
```

Present an C statement that sets **`n`** to be the full bitwise inversion of **`m`**.

```
n = ~m
n = m ^ (~0L)
```

Present a C statement that sets **`k`** to **`j`**, where the first and last bytes of **`j`** have been bitwise-inverted.

```
k = j ^ 0xFF0000FF
```

Present an expression that evaluates to **`true`** iff **`n`** has two neighboring bits that are both 1.

```
(n & (n >> 1)) != 0
```

Bitwise Manipulation and is_power_of_2

Without using loops, how can we verify that an **`unsigned long`** called **`value`** is a power of 2? What's special about its binary footprint, and how can we take advantage of that?

```
bool is_power_of_2(unsigned long value) {
   return value != 0 && (value & (value - 1)) == 0;
} // why the value != 0? Because 0 isn't a power of 2.
```

A **power of 2** has a distinct binary structure. We can exploit its structure knowing powers of 2 have precisely one 1, with all other bits being 0.

All other numbers have two or more 1 bits (except for 0, which has none).

Subtracting 1 from a power of 2 has a very specific impact.

If **`value`** is, for instance, $1\ 000000_2$, then value – 1 is $0\ 111111_2$. Restated, when subtracting 1 from a power of 2, its only 1 becomes a 0 and all lower bits go from 0 to 1.

The bitwise & of **`value`** and **`value - 1`** in this case produces all zeroes.

If **`value`** has more than one 1 bit, at least one 1 bit survives the bitwise & of **`value`** and **`value - 1`** .

Left Shift (<<)

The **<<** operator shifts a bit pattern a certain number of positions to the left. Low-order bits are filled with 0's, and bits shifted off the left are discarded.

```
x << k;  // evaluates to x shifted to the left by k bits
x <<= k; // in-place shifts x to the left by k bits
```

Assuming 8-bit figures:

00110111 << 2 results in **11011100**

01100011 << 4 results in **00110000**

10010101 << 5 results in **10100000**

I'm showing you the **bit-pattern** effect. In real C code, you should only rely on this behavior for **unsigned** values.

Oh, and shifting by more than the bit width (e.g., 9) is technically undefined.

Right Shift (>>)

The **>>** operator shifts a bit pattern a certain number of positions to the right. High bits are **typically** filled with the sign bit, and bits shifted off are discarded.

```
x >> k;  // evaluates to x shifted to the right by k bits
x >>= k; // in-place shifts x to the right by k bits
```

Assuming **signed** 8-bit figures:

01011101 >> 1 results in **00101110**

01111110 >> 4 results in **00000111**

11111110 >> 4 results in **11111111**

11011011 >> 7 results in **11111111**

Had we been dealing with **unsigned** figures instead, the sign bit isn't explicit—it's implied, and therefore 0.

Right shifts are called **arithmetic** when the sign bit is replicated, and **logical** when zeros are shifted in.

Bitwise Manipulation and `absolute_value`

Implementing an **`absolute_value`** function in CS106 terms is trivial.

```
unsigned int absolute_value(int value) {
   return value < 0 ? -value : value;
}
```

Can we implement the same function without using relational operators (e.g., <) or runtime multiplication, but instead just using bitwise operators?

❤️ yes ❤️

```
unsigned int absolute_value_bitwise(int value) {
   int mask = value >> (sizeof(value) * CHAR_BIT - 1);
   return (value ^ mask) - mask;
}
```

Bitwise Manipulation and `absolute_value`

The implementation below assumes **two's-complement integers**—perfectly reasonable—and **arithmetic right shif**t, which is true on every machine you'll use here on campus.

When **`value`** is nonnegative, **`mask`** evaluates to 0, so the return statement essentially returns **`value`** unmodified. That's because exclusive-or'ing **`value`** with 0 is essentially a no-op, and subtracting zero from **`value`** is as well.

When **`value`** is negative, **`mask`** evaluates to **`~0`**, i.e., -1. Exclusive-or'ing **`value`** with **`mask`** generates its bitwise inversion by flipping every bit and **subtracting** -1 from that is the same as **adding** 1. In essence, the return statement is synthesizing the two's complement of a negative number to produce a positive one.

```
unsigned int absolute_value_bitwise(int value) {
   int mask = value >> (sizeof(value) * CHAR_BIT - 1);
   return (value ^ mask) - mask;
}
```

Thought question: What does **`absolute_value_bitwise(INT_MIN)`** return?

Bitwise Manipulation

What do each of the following functions ultimately compute? How?

```
size_t mystery(unsigned long ul) {
   size_t count = 0;
   for (size_t i = 0; i < sizeof(ul) * CHAR_BIT; i++) {
      if ((ul & (1UL << i)) != 0) count++;
   }
   return count;
}
```

```
size_t enigma(unsigned long ul) {
   size_t count = 0;
   while (ul != 0) {
      count++;
      ul &= ul - 1;
   }
   return count;
}
```

mystery brute-force counts the number of bits in **ul** that are 1.

- The **1UL << i** mask is used to identify whether the **i**th bit is 1.
- The **UL** is crucial, since 1 standalone is a 32-bit constant, and left-shifting by more than the bit width—e.g., by 33 here—is undefined behavior. (This is relevant to **assign1**.)

enigma also counts the number of 1 bits in **ul**, but more efficiently!

- Each iteration clears the lowest-set bit using **ul &= ul - 1**, so the loop runs once per 1 bit rather than once per bit position. That makes it faster for sparse values.

Introducing gdb

Is there a way to step through the execution of a program and print out values as it's running?

Why yes, yes there is.

The gdb Debugger

gdb is a **command-line debugger** with functionality just like those that ship with Qt Creator or PyCharm

- 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 inspect variable values in various ways (including binary)
- It lets you track down where and why your program crashed
- And much, much more!

gdb is an essential systems programming tool, and you'll learn more and more of it 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 evaluate 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!