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:**
  - `&`, `|`, `~`, `^`, `<<`, `>>`

Practice: Bitwise Operations

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
----------
0b00001111
```

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

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

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

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 (>>)

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;
```

Real Numbers

Problem: every number base has un-representable real numbers.

Base 10: $1/6_{10} = 0.16666666\ldots\ldots_{10}$

Base 2: $1/10_{10} = 0.000110011001100110011\ldots_{2}$

Therefore, by representing in base 2, *we will not be able to represent all numbers*, even those we can exactly represent in base 10.

Fixed Point

- **Idea:** Like in base 10, let's add binary decimal places to our existing number representation.

1	0	1	1	.	0	1	1
8s	4s	2s	1s		1/2s	1/4s	1/8s

- **Pros:** arithmetic is easy! And we know exactly how much precision we have.

Fixed Point

- **Problem:** we have to fix where the decimal point is in our representation. What should we pick? This also fixes us to 1 place per bit.

.	0	1	1	0	0	1	1
	1/2s	1/4s	1/8s	...			

1	0	1	1	0	.	1	1
16s	8s	4s	2s	1s		1/2s	1/4s

Fixed Point

- **Problem:** we have to fix where the decimal point is in our representation. What should we pick? This also fixes us to 1 place per bit.

Base 10		Base 2
5.07E30	=	$10\underbrace{\cdots\cdots}_{\text{100 zeros}}0.1$
9.86E-32	=	$0.0\underbrace{\cdots\cdots}_{\text{100 zeros}}01$

> To be able to store both these numbers using the same fixed point representation, the bitwidth of the type would need to be at least 207 bits wide!

Let’s Get Real

What would be nice to have in a real number representation?

- Represent widest range of numbers possible
- Flexible “floating” decimal point
- Represent scientific notation numbers, e.g. 1.2×10^6
- Still be able to compare quickly
- Have more predictable over/under-flow behavior