



CS107 Lecture 4

Bits and Bytes Wrap-up, Bitwise Operators

Reading: Bryant & O'Hallaron, Ch. 2.1

Now that we understand values are really stored in binary, 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: &&, ||, !

Here's a new set of operators: the **bitwise operators**:

- &: bitwise and
- |: bitwise or
- ~: bitwise inversion
- ^: bitwise exclusive or
- <<: bitwise left shift
- >>: bitwise right shift

and (&)

& is a binary operator
the & 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

& a bit with 1 to let that bit through, & a bit with 0 to zero it out

or (|)

| is a binary operator
the | of 2 bits is 0 if both bits are 0, and 1 otherwise

output = a | b;

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

| a bit with 1 to make that bit 1, | a bit with 0 leave it alone

not (~)

~ is a unary operator and it inverts a 1 to a 0 and a 0 to a 1

output = ~a;

a	output
0	1
1	0

exclusive or (^)

The ^ of 2 bits is 1 if and only if exactly one of two bits is 1.

$$\text{output} = a \wedge b;$$

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

^ a bit with 1 to flip it, ^ a bit with 0 to leave it alone

Operators on Multiple Bits

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

AND

```
0110
& 1100
----
0100
```

OR

```
0110
| 1100
----
1110
```

XOR

```
0110
^ 1100
----
1010
```

NOT

```
~ 1100
----
0011
```

Operators on Multiple Bits

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

AND	OR	XOR	NOT
<pre>0110 & 1100 ---- 0100</pre>	<pre>0110 1100 ---- 1110</pre>	<pre>0110 ^ 1100 ---- 1010</pre>	<pre>~ 1100 ---- 0011</pre>

This is different from logical `&&`. `&&` returns **true** if both operands are nonzero, and **false** otherwise. With `&&`, this would be `6 && 12`, which would evaluate to **true**.

Operators on Multiple Bits

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

AND	OR	XOR	NOT
<pre>0110 & 1100 ---- 0100</pre>	<pre>0110 1100 ---- 1110</pre>	<pre>0110 ^ 1100 ---- 1010</pre>	<pre>~ 1100 ---- 0011</pre>

This is different from `||`. `||` is **true** if either are nonzero, and **false** otherwise. With `||`, this would be `6 || 12`, which would evaluate to **true**.

Operators on Multiple Bits

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

AND	OR	XOR	NOT
$\begin{array}{r} 0110 \\ \& 1100 \\ \hline 0100 \end{array}$	$\begin{array}{r} 0110 \\ 1100 \\ \hline 1110 \end{array}$	$\begin{array}{r} 0110 \\ ^ 1100 \\ \hline 1010 \end{array}$	$\begin{array}{r} \sim 1100 \\ \hline 0011 \end{array}$

This is different from `!`. `!` produces a **true** if applied to a zero, and **false** otherwise. With `!`, this would be `!12`, which would evaluate to **false**.

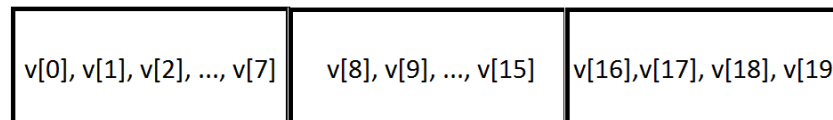
Bitwise Operators and Bitmasks

We will frequently want to **manipulate** or otherwise **isolate specific bits** in a larger collection of them. A **bitmask** is a constructed bit pattern that we can use, along with standard bit operators like **&**, **|**, **^**, **~**, **<<**, and **>>**, to do this.

Motivating Example: bit vectors

C++, for example, uses bit vectors to implement the **vector<bool>**

```
std::vector<bool> v(20);
```



The idea? Allocate three **chars** for its 24 bits and ignore the last four bits of the third **char**.

Bit Vectors and Sets

Instead of using an array of **bool**s, store Boolean information in bits instead.

Example: we can represent the set of core CS courses being taken this quarter using a single **char**.

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS111	CS107	CS106AX	CS106B	CS106A

Bit Vectors and Sets

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS111	CS107	CS106AX	CS106B	CS106A

How do we compute the union of two course schedules?

```
  00100011
| 01100001
-----
  01100011
```

We use **bitwise or** here, since the result is 1 if one or both operands are 1. Assuming 1 means an element is present—i.e., the course is part of someone's schedule—that's precisely what we want if we're to compute the **union of two sets**.

Bit Vectors and Sets

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS111	CS107	CS106AX	CS106B	CS106A

How do we compute the intersection of two course schedules?

```
00100011
& 01100001
-----
00100001
```

This time it's **bitwise and**, since the result is 1 if and only if both operands are 1. Assuming 1 means an element is present—i.e., the course is part of someone's schedule—this is precisely what we want here as well.

Bit Masking

How do we update the course schedule to indicate a student is **taking** CS107?

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS111	CS107	CS106AX	CS106B	CS106A

```
  00100011
| 00001000
|-----
  00101011
```

Bit Masking

```
#define CS106A 0x1    /* 0000 0001 */
#define CS106B 0x2    /* 0000 0010, or 1 << 1 */
#define CS106AX 0x4   /* 0000 0100, or 1 << 2 */
#define CS107 0x8     /* 0000 1000, or 1 << 3 */
#define CS111 0x10    /* 0001 0000, or 1 << 4 */
#define CS103 0x20    /* 0010 0000, or 1 << 5 */
#define CS109 0x40    /* 0100 0000, or 1 << 6 */
#define CS161 0x80    /* 1000 0000, or 1 << 7 */

char schedule = ...;
schedule = schedule | CS107; // include CS107!
```

Bit Masking

```
#define CS106A 0x1    /* 0000 0001 */
#define CS106B 0x2    /* 0000 0010, or 1 << 1 */
#define CS106AX 0x4    /* 0000 0100, or 1 << 2 */
#define CS107 0x8      /* 0000 1000, or 1 << 3 */
#define CS111 0x10     /* 0001 0000, or 1 << 4 */
#define CS103 0x20     /* 0010 0000, or 1 << 5 */
#define CS109 0x40     /* 0100 0000, or 1 << 6 */
#define CS161 0x80     /* 1000 0000, or 1 << 7 */

char schedule = ...;
schedule |= CS107;           // include CS107!
```

Bit Masking

How do we update our schedule to indicate we've **dropped** CS103?

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS111	CS107	CS106AX	CS106B	CS106A

```
00100011
& 11011111
-----
00000011
```

```
schedule &= ~CS103;
```

Bit Masking

How do we confirm we're **enrolled** in CS106B?

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS111	CS107	CS106AX	CS106B	CS106A

```
00100011
& 00000010
-----
00000010
```

```
if (schedule & CS106B) {
    // taking CS106B!
```