CS107 Lecture 4
Bits and Bytes Wrap-up, Bitwise Operators

Reading: Bryant & O'Hallaron, Ch. 2.1

Casting Between Signed and Unsigned

What happens at the byte level when we cast between variable types?

The bit patterns remain the same, but their interpretations are dictated by the type.

```
int v = -12345;
unsigned int uv = v;
printf("v = %d, uv = %u\n", v, uv);
```

Output: `v = -12345, uv = 4294954951`.

Why the difference?

`-12345` in two's complement binary is **0b11111111111111111100111111000111.**

When treated as an unsigned, inherently nonnegative number, `-12345` is **all magnitude**.

Casting Between Signed and Unsigned

C-style casts can be used to on-the-fly reinterpret a value under a different type system, as with:

```
int v = -12345;
printf("v = %d, uv = %u\n", v, (unsigned int) v);
```

You can also append a **U** to a numeric literal to treat it as an unsigned, as with:

```
-12345U
```

Here, the **-12345** is evaluated for its 32-bit, **signed int** bit pattern, but because of that **U**, its bits are interpreted as an **unsigned int** instead.

Comparisons Between Different Types

Note: When comparing signed and unsigned integers. **C will implicitly cast** the signed value to **unsigned** and evaluate the expression assuming both values are **unsigned** and therefore **nonnegative**.

expression	comparison type?	evaluates to?	mathematically accurate?
`0 == 0U`			

Comparisons Between Different Types

Note: When comparing signed and unsigned integers. **C will implicitly cast** the signed value to **unsigned** and evaluate the expression assuming both values are **unsigned** and therefore **nonnegative**.

expression	comparison type?	evaluates to?	mathematically accurate?
`0 == 0U`	unsigned	`true`	yes
`-1 < 0`	signed	`true`	yes
`-1 < 0U`	unsigned	`false`	nope
`2147483647 > -2147483648`	signed	`true`	yes
`2147483647U > -2147483648`	unsigned	`false`	nada
`-1 > -2`	signed	`true`	yes
`(unsigned long)-1 > -2`	unsigned	`true`	yes

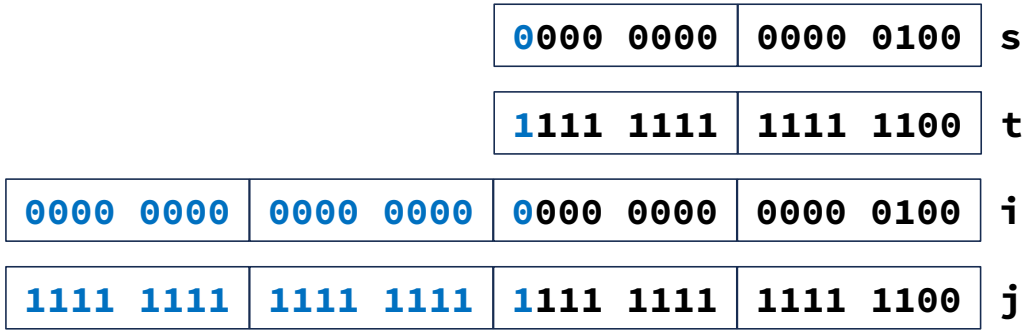
Extending Bit Representations

What happens when we initialize a variable of type **`int`** from a **`short`**?

```
short s = 4, t = -4;
int i = s, j = t;
```

Though it's uncommon, C allows it and defines clear rules for how the conversion works, choosing a behavior that's **efficient** for the hardware and **intuitive** for the programmer.

In this case, the easiest thing to preserve the **signed value** is to **sign extend** the bit pattern of the smaller integer to fill the extra bits of the larger one.



`i` and **`j`** are each four-byte quantities that evaluate to 4 and -4, respectively. By **sign extending** the two-byte representations—that is, by replicating the sign bit to full up the extra space, we preserve both **magnitude** and **sign**.

Extending Bit Representations

What happens when we initialize an **unsigned int** from an **unsigned short**?

```
unsigned short us = 0b1111111111110010;
unsigned int ui = us;
```

This is simpler, because we can always fill the extra bits with zeroes. In an unsigned world, leading zero bits leave the value alone.

Sign and zero extension aren't unique to **short**s.

longs can be initialized from **int**s, **short**s, or **char**s.

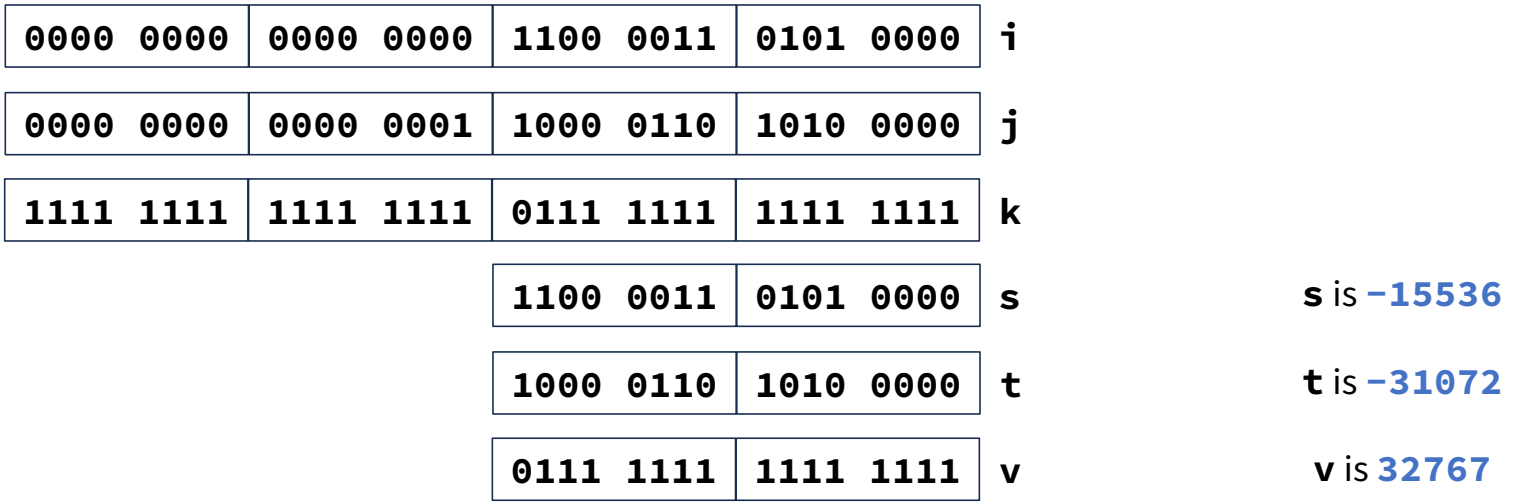
ints and **short**s can be initialized from **char**s.

Truncating Bit Representations

What about the opposite? What happens when we initialize a **`short`** from an **`int`**?

```
int i = 50000, j = 100000, k = -32769;
short s = i, t = j, v = k;
```

C can't wedge four bytes' worth of data into two bytes of memory, so it **discards**—or rather, **truncates**—the most significant bits and **retains** the least significant ones.



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.

```
output = a ^ 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

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

OR

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

XOR

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

NOT

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

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

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

OR

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

XOR

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

NOT

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

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

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

OR

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

XOR

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

NOT

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

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`**.

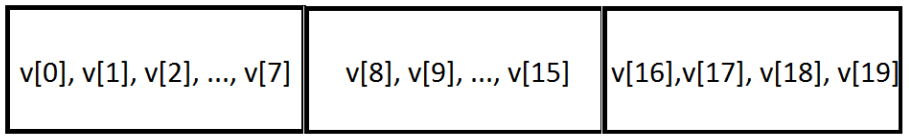
Bitwise Operators and Bitmasks

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);

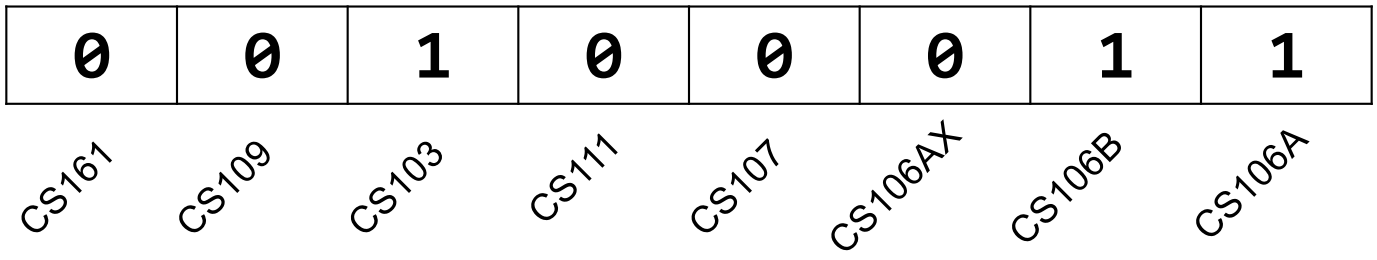


The idea? Allocate three **char**s for its 24 bits and ignore the last four bits of the third **char**.

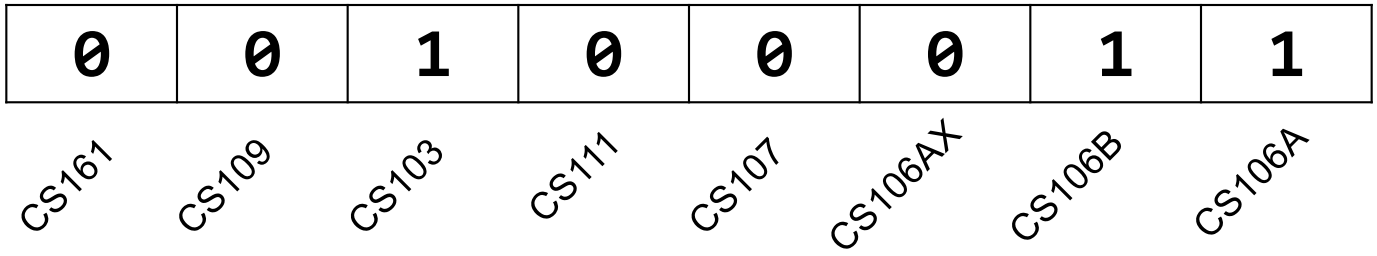
Bit Vectors and Sets

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`**.



Bit Vectors and Sets

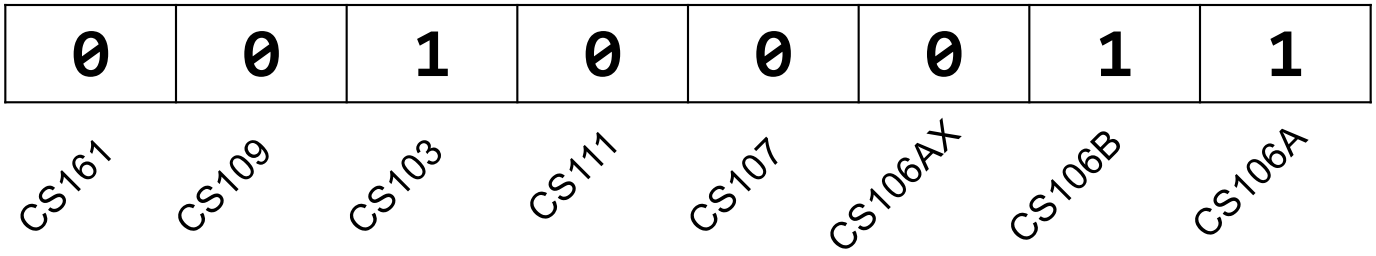


How do we compute the union of two course schedules?

```
  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



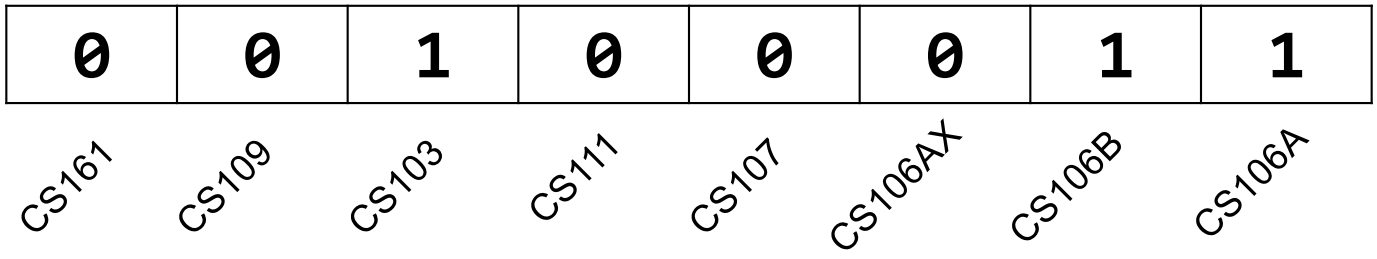
How do we compute the intersection of two course schedules?

```
  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?




```
  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!
```

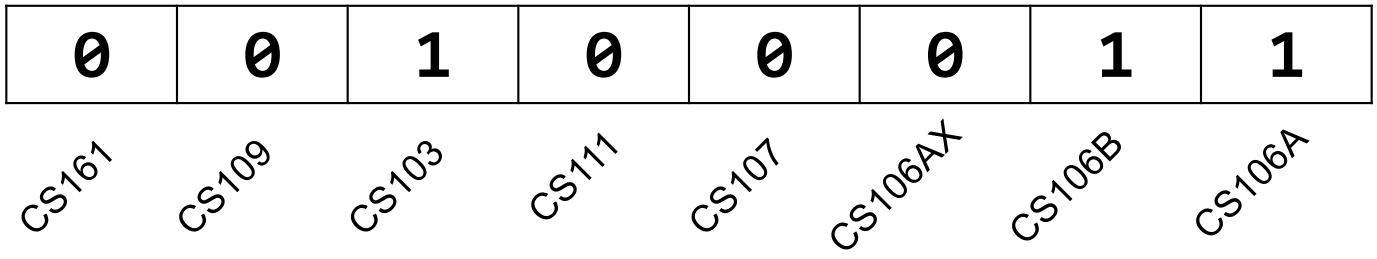
Bit Masking

```
#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?

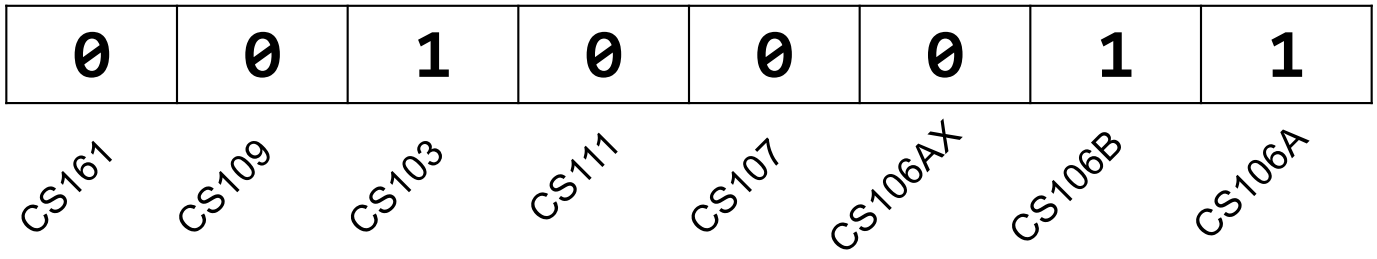



```
  00100011
& 11011111
  --------
  00000011
```

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

Bit Masking

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




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

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