CS107 Lecture 4

Bits and Bytes Wrap-up, Bitwise Operators

Reading: Bryant & O'Hallaron, Ch. 2.1

[Ed Discussion](#)

Casting

What happens at the byte level when we cast between variable types? **The bytes remain the same! This means they may be interpreted differently depending on the type.**

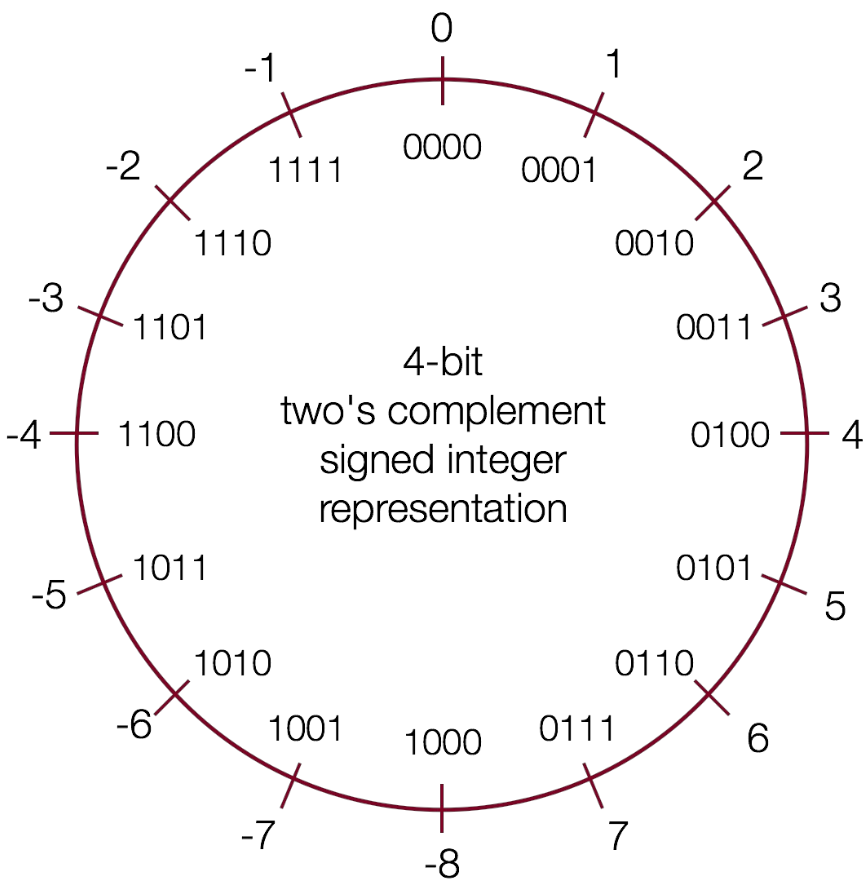
```
int v = -12345;
unsigned int uv = v;
printf("v = %d, uv = %u\n", v, uv);
```

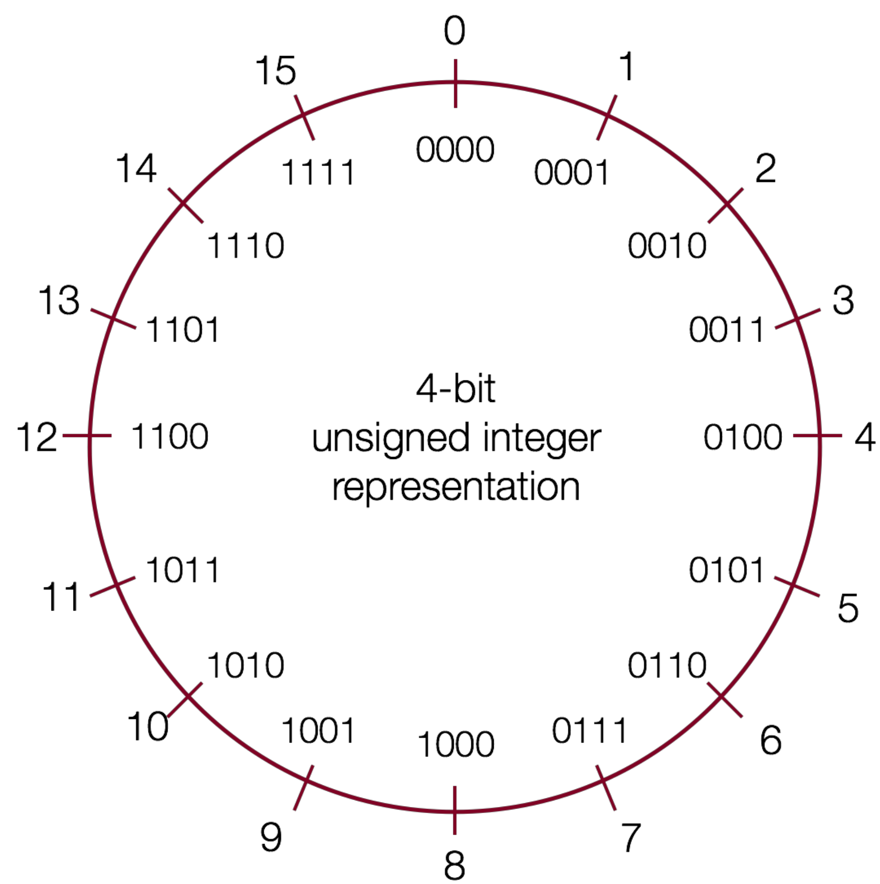
This prints out: `"v = -12345, uv = 4294954951"`. **Why?**

The bit representation for -12345 is `0b11111111111111111100111111000111`.

If we treat this as an unsigned, inherently positive number, it's *huge*!

Casting


0
1
2
3
4
5
6
7
-8
-7
-6
-5
-4
-3
-2
-1
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
4-bit
two's complement
signed integer
representation


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
4-bit
unsigned integer
representation

Casting

You can cast something to another type by putting that type in parentheses in front of the value:

```
int v = -12345;
...(unsigned int)v...
```

You can also use the **U** suffix after a number literal to treat it as unsigned:

```
-12345U
```

Comparisons Between Different Types

Be careful when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned and then evaluate the expression assuming both numbers are unsigned and nonnegative.

`Expression`	Comparison Type?	Evaluates To?	Mathematically correct?
`0 == 0U`			

Comparisons Between Different Types

Be careful when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned and then evaluate the expression assuming both numbers are unsigned and nonnegative.

`Expression`	Comparison Type?	Evaluates To?	Mathematically correct?
`0 == 0U`	Unsigned	true	yes

Comparisons Between Different Types

Be careful when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned and then evaluate the expression assuming both numbers are unsigned and nonnegative.

`Expression`	Comparison Type?	Evaluates To?	Mathematically correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`			

Comparisons Between Different Types

Be careful when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned and then evaluate the expression assuming both numbers are unsigned and nonnegative.

`Expression`	Comparison Type?	Evaluates To?	Mathematically correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes

Comparisons Between Different Types

Be careful when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned and then evaluate the expression assuming both numbers are unsigned and nonnegative.

`Expression`	Comparison Type?	Evaluates To?	Mathematically correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes
`-1 < 0U`			

Comparisons Between Different Types

Be careful when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned and then evaluate the expression assuming both numbers are unsigned and nonnegative.

`Expression`	Comparison Type?	Evaluates To?	Mathematically correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes
`-1 < 0U`	Unsigned	false	No!

Comparisons Between Different Types

Be careful when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned and then evaluate the expression assuming both numbers are unsigned and nonnegative.

`Expression`	Comparison Type?	Evaluates To?	Mathematically correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes
`-1 < 0U`	Unsigned	false	No!
`2147483647 > -2147483648`			

Comparisons Between Different Types

Be careful when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned and then evaluate the expression assuming both numbers are unsigned and nonnegative.

`Expression`	Comparison Type?	Evaluates To?	Mathematically correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes
`-1 < 0U`	Unsigned	false	No!
`2147483647 > -2147483648`	Signed	true	yes

Comparisons Between Different Types

Be careful when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned and then evaluate the expression assuming both numbers are unsigned and nonnegative.

`Expression`	Comparison Type?	Evaluates To?	Mathematically correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes
`-1 < 0U`	Unsigned	false	No!
`2147483647 > -2147483648`	Signed	true	yes
`2147483647U > -2147483648`			

Comparisons Between Different Types

Be careful when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned and then evaluate the expression assuming both numbers are unsigned and nonnegative.

Expression	Comparison Type?	Evaluates To?	Mathematically correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes
`-1 < 0U`	Unsigned	false	No!
`2147483647 > -2147483648`	Signed	true	yes
`2147483647U > -2147483648`	Unsigned	false	No!

Comparisons Between Different Types

Be careful when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned and then evaluate the expression assuming both numbers are unsigned and nonnegative.

Expression	Comparison Type?	Evaluates To?	Mathematically correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes
`-1 < 0U`	Unsigned	false	No!
`2147483647 > -2147483648`	Signed	true	yes
`2147483647U > -2147483648`	Unsigned	false	No!
`-1 > -2`			
`(unsigned)-1 > -2`			

Comparisons Between Different Types

Be careful when comparing signed and unsigned integers. **C will implicitly cast** the signed argument to unsigned and then evaluate the expression assuming both numbers are unsigned and nonnegative.

Expression	Comparison Type?	Evaluates To?	Mathematically correct?
`0 == 0U`	Unsigned	true	yes
`-1 < 0`	Signed	true	yes
`-1 < 0U`	Unsigned	false	No!
`2147483647 > -2147483648`	Signed	true	yes
`2147483647U > -2147483648`	Unsigned	false	No!
`-1 > -2`	Signed	true	yes
`(unsigned)-1 > -2`	Unsigned	true	yes

Expanding Bit Representations

- Sometimes, we need to convert between two integers of different sizes (e.g. **`short`** to **`int`**, or **`int`** to **`long`**).
- We might not be able to convert from a bigger data type to a smaller data type and retain all information, but we should always be able to convert from a **smaller** data type to a **larger** data type.
- For **unsigned** values, we can prepend *leading zeros* to the representation ("zero extension")
- For **signed** values, we can *repeat the sign of the value* for new digits ("sign extension")
- Note: when doing <, >, <=, >= comparison between different size types, it will *promote the smaller type to the larger one*.

Expanding Bit Representation

```
unsigned short s = 4;
// short is a 16-bit format, so                    s = 0000 0000 0000 0100b

unsigned int i = s;
// conversion to 32-bit int, so i = 0000 0000 0000 0000 0000 0000 0000 0100b
```

Expanding Bit Representation

```
short s = 4;
// short is a 16-bit format, so                     s = 0000 0000 0000 0100b

int i = s;
// conversion to 32-bit int, so i = 0000 0000 0000 0000 0000 0000 0000 0100b
```

— or —

```
short s = -4;
// short is a 16-bit format, so                     s = 1111 1111 1111 1100b

int i = s;
// conversion to 32-bit int, so i = 1111 1111 1111 1111 1111 1111 1111 1100b
```

Truncating Bit Representation

If we want to **reduce** the bit size of a number, C *truncates* the representation and discards the *more significant bits*.

```
int x = 53191;
short sx = x;
int y = sx;
```

What happens here? Let's look at the bits in x (a 32-bit int), 53191:

0000 0000 0000 0000 1100 1111 1100 0111

When we cast x to a short, it only has 16-bits, and C *truncates* the number:

1100 1111 1100 0111

This is -12345! And when we cast sx back an int, we sign-extend the number.

1111 1111 1111 1111 1100 1111 1100 0111 // still -12345

Truncating Bit Representation

If we want to **reduce** the bit size of a number, C *truncates* the representation and discards the *more significant bits*.

```
int x = -3;
short sx = x;
int y = sx;
```

What happens here? Let's look at the bits in x (a 32-bit int), -3:

`1111 1111 1111 1111 1111 1111 1111 1101`

When we cast x to a short, it only has 16-bits, and C *truncates* the number:

`1111 1111 1111 1101`

This is -3! **If the number does fit, it will convert fine.** y looks like this:

`1111 1111 1111 1111 1111 1111 1111 1101` // still -3

Truncating Bit Representation

If we want to **reduce** the bit size of a number, C *truncates* the representation and discards the *more significant bits*.

```
unsigned int x = 128000;
unsigned short sx = x;
unsigned int y = sx;
```

What happens here? Let's look at the bits in x (a 32-bit unsigned int), 128000:

`0000 0000 0000 0001 1111 0100 0000 0000`

When we cast x to a short, it only has 16-bits, and C *truncates* the number:

`1111 0100 0000 0000`

This is 62464! **Unsigned numbers can lose info too.** Here is what y looks like:

`0000 0000 0000 0000 1111 0100 0000 0000` // still 62464

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**: &&, ||, !
- Today and Wednesday, we’ll be discussing a new category of operators: **bitwise operators:**

&, |, ~, ^, <<, >>

And (&)

AND is a binary operator. The AND 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

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

Or (|)

OR is a binary operator. The OR of 2 bits is 1 if either (or both) bits is 1.

```
output = a | b;
```

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

| with 1 to turn on a bit, | with 0 to let a bit go through

Not (~)

NOT is a unary operator. The NOT of a bit is 1 if the bit is 0, or 1 otherwise.

```
output = ~a;
```

a	output
0	1
1	0

Exclusive Or (^)

Exclusive Or (XOR) is a binary operator. The XOR of 2 bits is 1 if *exactly* one of the bits is 1, or 0 otherwise.

```
output = a ^ b;
```

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

^ with 1 to flip a bit, ^ with 0 to let a bit go through unmodified

Operators on Multiple Bits

- When these operators are applied to numbers (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 numbers (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
```

Note: these are different from the logical operators AND (&&), OR (||) and NOT (!).

Operators on Multiple Bits

- When these operators are applied to numbers (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 AND (&&). The logical AND returns true if both are nonzero, or false otherwise. With &&, this would be 6 && 12, which would evaluate to **true** (1).

Operators on Multiple Bits

- When these operators are applied to numbers (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 OR (||). The logical OR returns true if either are nonzero, or false otherwise. With ||, this would be 6 || 12, which would evaluate to **true** (1).

Operators on Multiple Bits

- When these operators are applied to numbers (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 NOT (!). The logical NOT returns true if this is zero, and false otherwise. With !, this would be !12, which would evaluate to **false** (0).

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

Aside: C++ relies on bit vectors to efficiently implement **vector<bool>**.

Bit Vectors and Sets

Instead of using arrays of Booleans, one can more compactly store Boolean information in bits instead.

- **Example:** we can represent current courses taken using a **`char`** and manipulate its contents using bit operators.

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 find the union of two sets of courses taken? Use OR:

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

Bit Vectors and Sets

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

- How do we find the intersection of two sets of courses taken? Use AND:

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

Bit Masking

Example: how do we update our bit vector to indicate we've taken 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 0x1 << 1 */
#define CS106AX 0x4   /* 0000 0100, or 0x1 << 2 */
#define CS107   0x8   /* 0000 1000, or 0x1 << 3 */
#define CS111   0x10  /* 0001 0000, or 0x1 << 4 */
#define CS103   0x20  /* 0010 0000, or 0x1 << 5 */
#define CS109   0x40  /* 0100 0000, or 0x1 << 6 */
#define CS161   0x80  /* 1000 0000, or 0x1 << 7 */

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

Bit Masking

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

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

Bit Masking

- **Example:** how do we update our bit vector to indicate we've *dropped* CS103?

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

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

```
char myClasses = ...;
myClasses &= ~CS103;    // Drop CS103
```

Bit Masking

- **Example:** how do we check if we've taken CS106B?

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

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

```
char myClasses = ...;
if (myClasses & CS106B) {...
     // taken CS106B!
```

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 isolated bits
- ~ is useful for flipping all bits