CS107 Lecture 4
Bits and Bytes; Bitwise Operators

Reading: Bryant & O'Hallaron, Ch. 2.1
Ed Discussion: https://edstem.org/us/courses/46162/discussion/3538902

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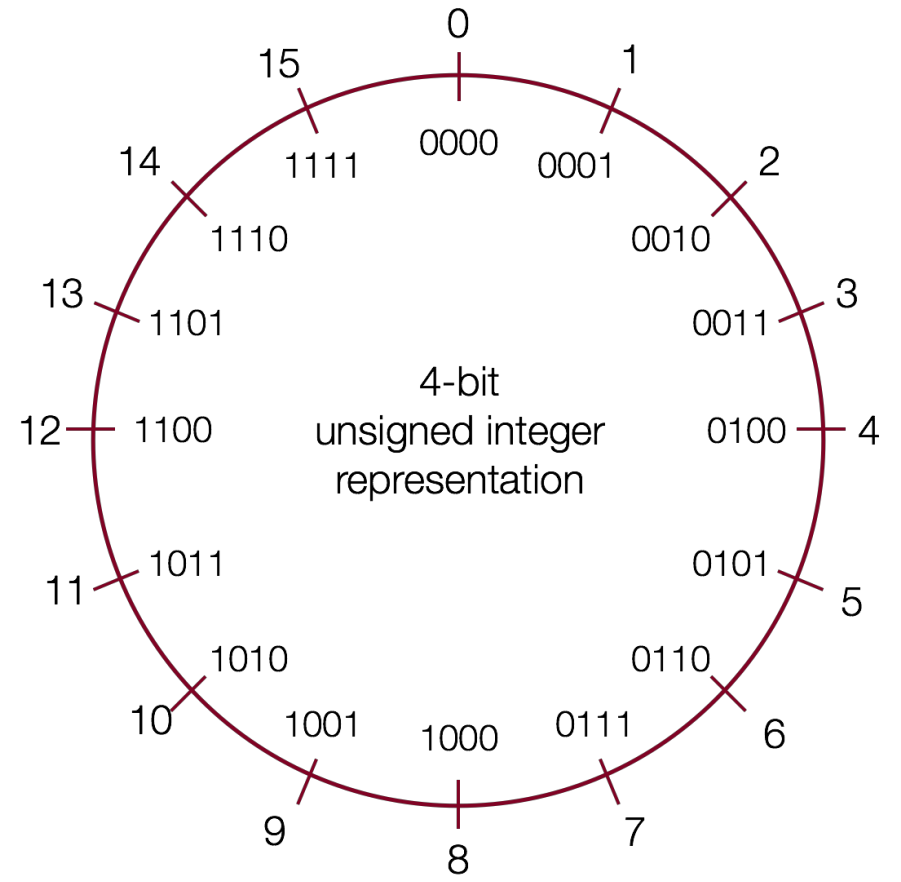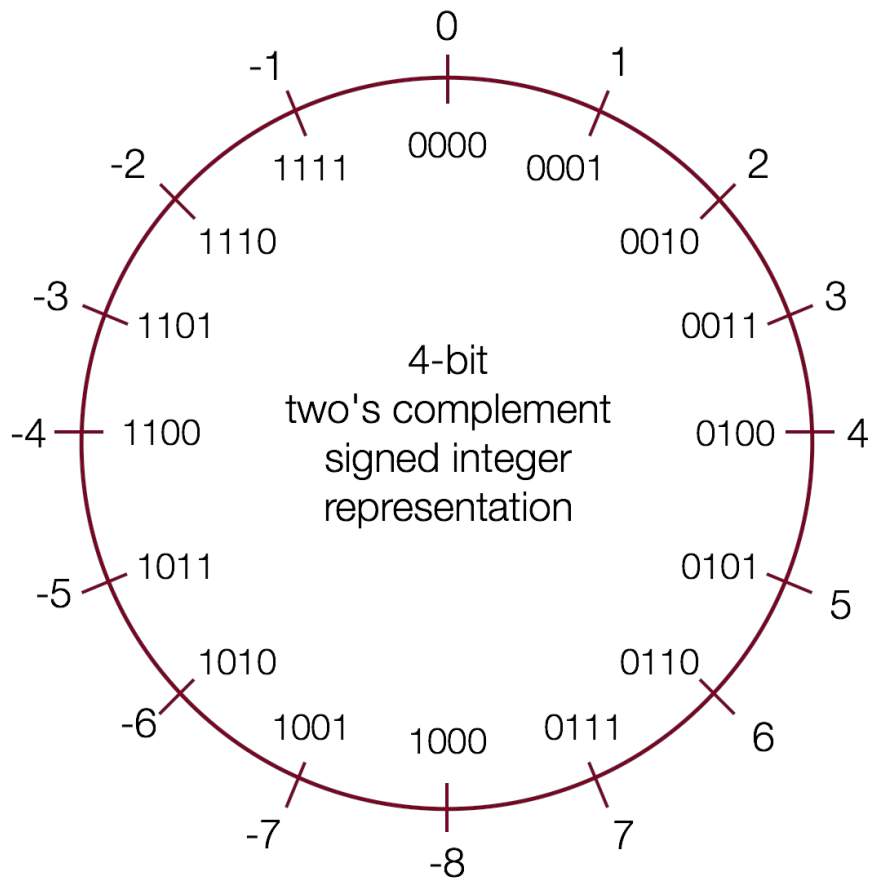
```
int v = -12345;
unsigned int uv = v;
printf("v = %d, uv = %u\n", v, uv);
```

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

The bit representation for -12345 is 0b**11111111111111111100111111000111.**

If we treat this binary representation as a positive number, it's *huge*!

Casting

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 tomorrow, we'll be introducing 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              OR              XOR             NOT

     0110            0110            0110
   & 1100          | 1100          ^ 1100          ~ 1100
   ----            ----            ----            ----
     0100            1110            1010            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	OR	XOR	NOT

` 0110`
`& 1100`
`----`
` 0100`
	` 0110`
	`| 1100`
	`----`
	` 1110`
		` 0110`
		`^ 1100`
		`----`
		` 1010`
			`~ 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	OR	XOR	NOT
``` 0110 ```	``` 0110 ```	``` 0110 ```	``` ```
```& 1100 ```	```	1100 ```	```^ 1100 ```	```~ 1100 ```
```----```	```----```	```----```	```----```
```0100```	```1110```	```1010```	```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	OR	XOR	NOT
``` 0110 & 1100 ---- 0100 ```	``` 0110 | 1100 ---- 1110 ```	``` 0110 ^ 1100 ---- 1010 ```	``` ~ 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	OR	XOR	NOT

``` 0110```
 ```& 1100```
 ```----```
 ``` 0100```
	``` 0110```
	 ```| 1100```
	 ```----```
	 ``` 1110```
		``` 0110```
		 ```^ 1100```
		 ```----```
		 ``` 1010```
			```~ 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).

Demo: Bits Playground