CS107 Lecture 3

Byte Ordering & Bitwise Operators

reading:

Bryant & O'Hallaron, Ch. 2.1

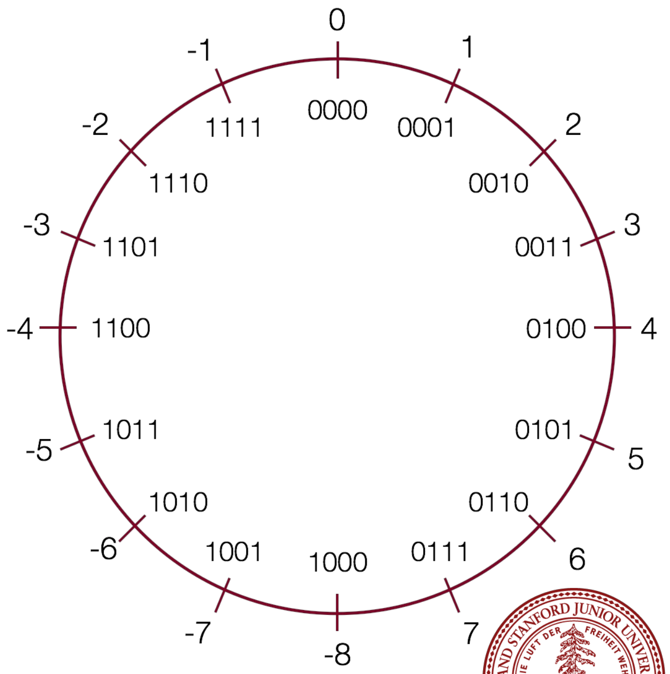
Where we left off

Comparison between signed and unsigned integers

When a C expression has combinations of signed and unsigned variables, you need to be careful!

If an operation is performed that has both a signed and an unsigned value, **C implicitly casts the signed argument to unsigned** and performs the operation assuming both numbers are non-negative. Let's take a look…

Expression	Type	Evaluation
`0 == 0U`	Unsigned	1
`-1 < 0`	Signed	1
`-1 < 0U`	Unsigned	0
`2147483647 > -2147483647 - 1`	Signed	1
`2147483647U > -2147483647 - 1`	Unsigned	0
`2147483647 > (int)2147483648U`	Signed	1
`-1 > -2`	Signed	1
`(unsigned)-1 > -2`	Unsigned	1



Note: In C, 0 is false and everything else is true. When C produces a boolean value, it allways chooses 1 to represent true.

Comparison between signed and unsigned integers

Let's try some more…a bit more abstractly.

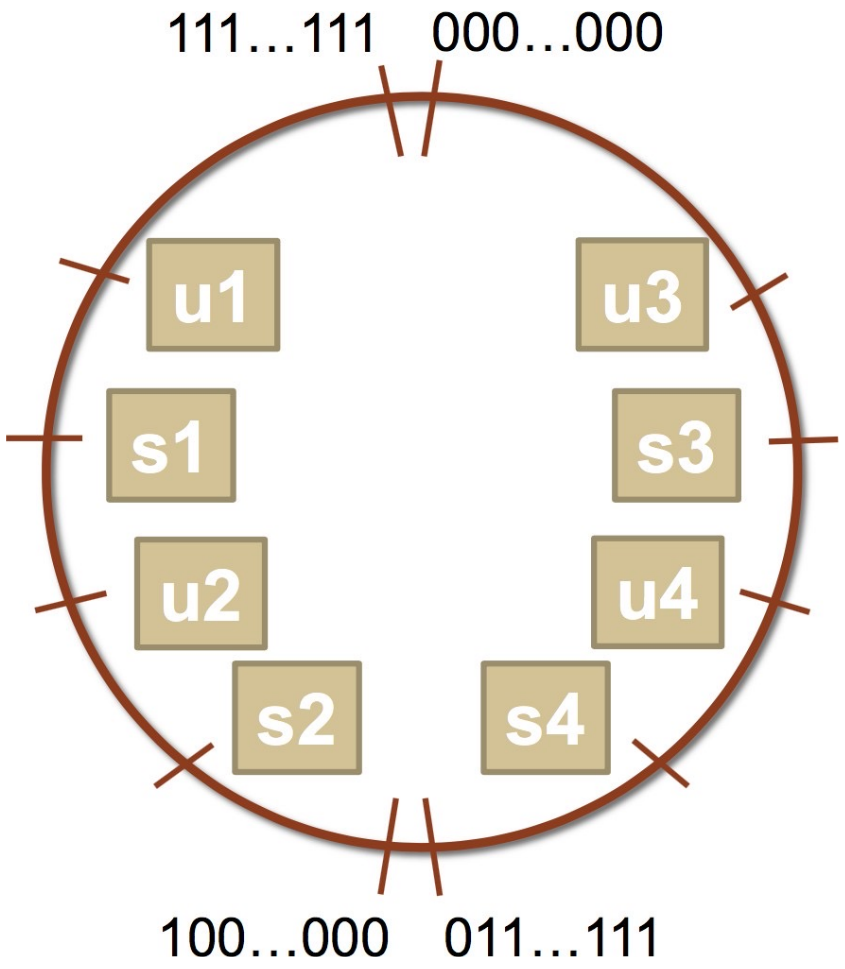
```
int s1, s2, s3, s4;
unsigned int u1, u2, u3, u4;
```

What is the value of this expression?

```
u1 > s3
```

Go to

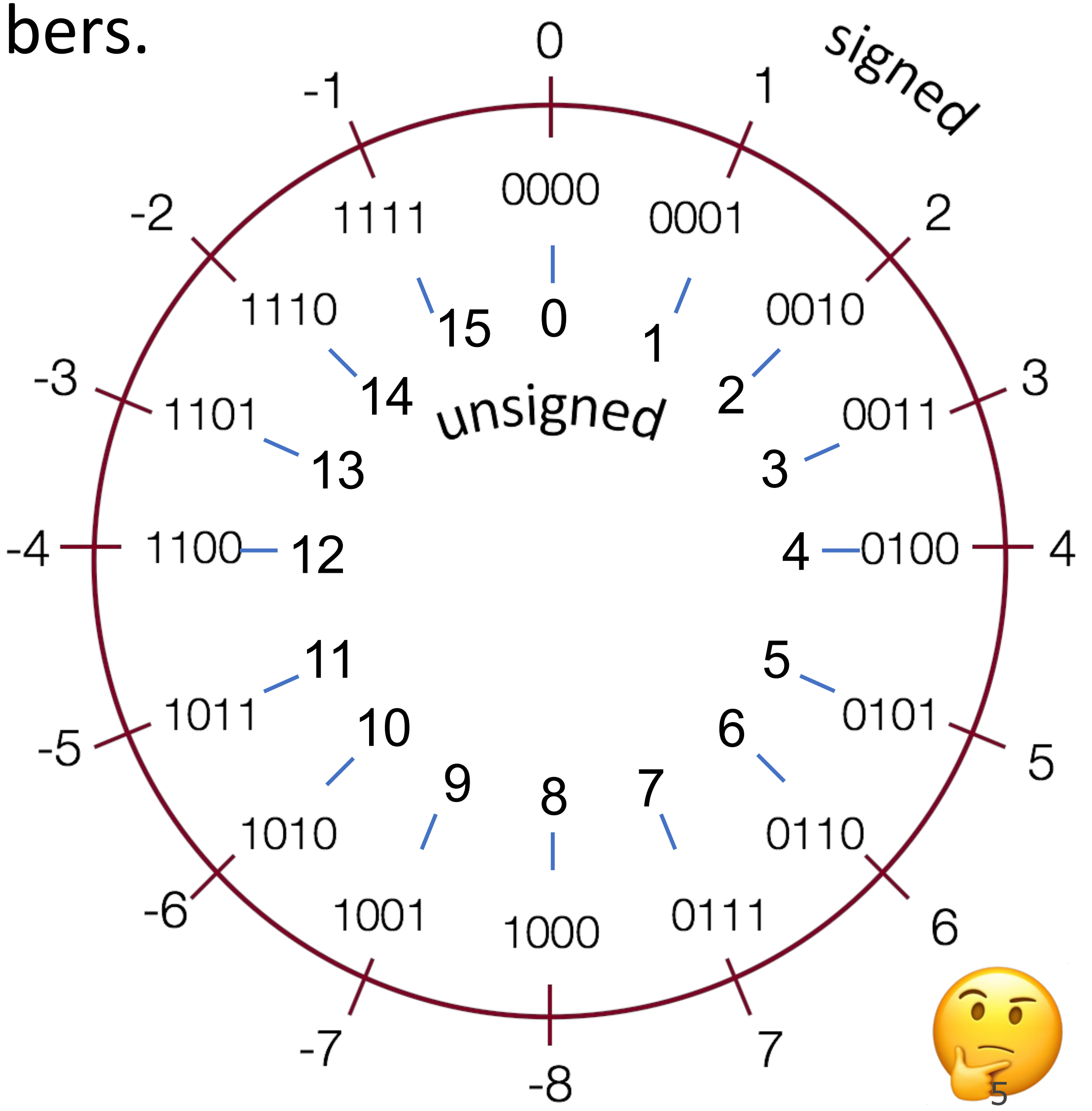
https://pollev.com/cs107summer



Overflow

- What is happening here? Assume 4-bit numbers.

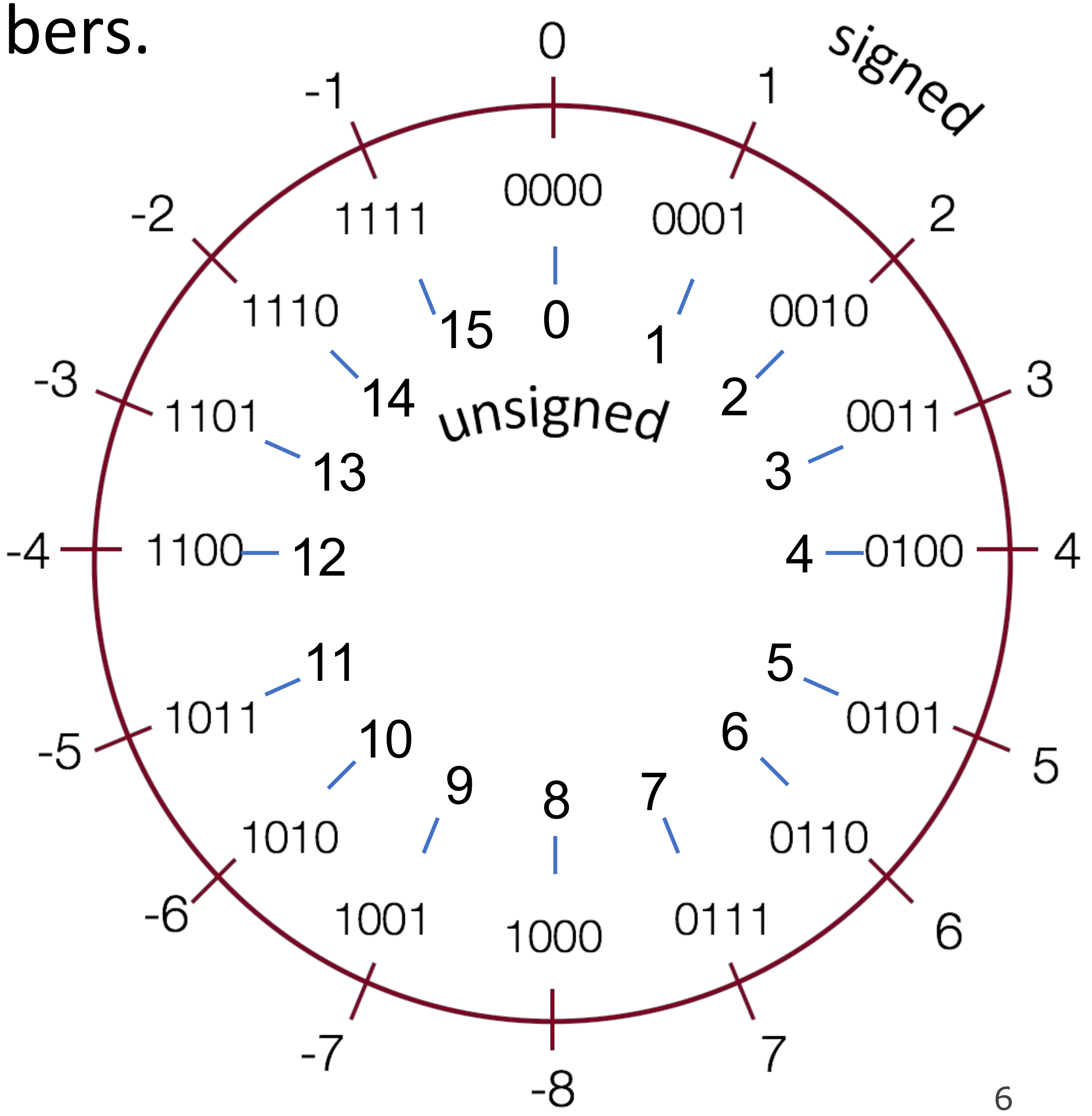
```
  0b1101
+ 0b0100
```



Overflow

- What is happening here? Assume 4-bit numbers.

```
  0b1101
+ 0b0100
```



<u>Signed</u>

-3 + 4 = 1

No overflow

<u>Unsigned</u>

13 + 4 = 1

Overflow

Limits and Comparisons

1. What is the…

	Largest unsigned?	Largest signed?	Smallest signed?
`char`	$2^8 - 1 = 255$	$2^7 - 1 = 127$	$-2^7 = -128$
`int`	$2^{32} - 1 = 4294967296$	$2^{31} - 1 = 2147483647$	$-2^{31} = -2147483648$

These are available as UCHAR_MAX, INT_MIN, INT_MAX, etc. in the `<limits.h>` header.

Limits and Comparisons

2. Will the following `char` comparisons evaluate to `true` or `false`?
   i. `-7 < 4`
   ii. `-7 < 4U`
   iii. `(char) 130 > 4`
   iv. `(char) -132 > 2`

By default, numeric constants in C are signed ints, unless they are suffixed with u (unsigned) or L (long).

The `sizeof` Operator

```
long sizeof(type);

    // Example
    long int_size_bytes = sizeof(int);    // 4
    long short_size_bytes = sizeof(short); // 2
    long char_size_bytes = sizeof(char);   // 1
```

`sizeof` takes a variable type as a parameter and returns the size of that type, in bytes.

MIN and MAX values for integers

Because we now know how bit patterns for integers works, we can figure out the maximum and minimum values, designated by `INT_MAX`, `UINT_MAX`, `INT_MIN`, (etc.), which are defined in `limits.h`

Type	**Width (bytes)**	**Width (bits)**	**Min in hex (name)**	**Max in hex (name)**
char	1	8	80 (CHAR_MIN)	7F (CHAR_MAX)
unsigned char	1	8	0	FF (UCHAR_MAX)
short	2	16	8000 (SHRT_MIN)	7FFF (SHRT_MAX)
unsigned short	2	16	0	FFFF (USHRT_MAX)
int	4	32	80000000 (INT_MIN)	7FFFFFFF (INT_MAX)
unsigned int	4	32	0	FFFFFFFF (UINT_MAX)
long	8	64	8000000000000000 (LONG_MIN)	7FFFFFFFFFFFFFFF (LONG_MAX)
unsigned long	8	64	0	FFFFFFFFFFFFFFFF (ULONG_MAX)

Min and Max Integer Values

- You can also find constants in the standard library that define the max and min for each type on that machine(architecture)

- Visit <limits.h> or <cstdint.h> and look for variables like:

```
INT_MIN
INT_MAX
UINT_MAX
LONG_MIN
LONG_MAX
ULONG_MAX
```

…

Expanding Bit Representations

- Sometimes, we want 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, but we do want to always be able to convert from a **smaller** data type to a **bigger** data type.
- For **unsigned** values, we can add *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 to the larger type*.

Expanding the bit representation of a number

For signed values, we want the number to remain the same, just with more bits. In this case, we perform a "sign extension" by repeating the sign of the value for the new digits. E.g.,

```
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
```

Converting from a smaller type to a larger type is also often called promotion
I.E. the number was promoted from short to int

Sign-extension Example

```
// show_bytes() defined on pg. 45, Bryant and O'Halloran
int main() {
    short sx = -12345;        // -12345
    unsigned short usx = sx; // 53191
    int x = sx;               // -12345
    unsigned ux = usx;        // 53191

    printf("sx = %d:\t", sx);
    show_bytes((byte_pointer) &sx, sizeof(short));
    printf("usx = %u:\t", usx);
    show_bytes((byte_pointer) &usx, sizeof(unsigned short));
    printf("x  = %d:\t", x);
    show_bytes((byte_pointer) &x, sizeof(int));
    printf("ux = %u:\t", ux);
    show_bytes((byte_pointer) &ux, sizeof(unsigned));

    return 0;
}
```

```
$ ./sign_extension
sx = -12345:   c7 cf
usx = 53191:   c7 cf
x  = -12345:   c7 cf ff ff
ux = 53191:    c7 cf 00 00
```

(careful: this was printed on the little-endian myth machines!)

Truncating Numbers: Signed

What if we want to reduce the number of bits that a number holds? E.g.

```
int x = 53191;            // 53191
short sx = (short) x;  // -12345
int y = sx;
```

This is a form of *overflow*! We have altered the value of the number. Be careful!

We don't have enough bits to store the int in the short for the value we have in the `int`, so the strange values occur.

What is y above? We are converting a short to an int, so we sign-extend, and we get -12345!

`1100 1111 1100 0111` becomes

`1111 1111 1111 1111 1100 1111 1100 0111`

Play around here: http://www.convertforfree.com/twos-complement-calculator/

Truncating Numbers: Signed

If the number does fit into the smaller representation in the current form, it will convert just fine.

```
int x = -3;               // -3
short sx = (short) -3; // -3
int y = sx; // -3
```

x: `1111 1111 1111 1111 1111 1111 1111 1101` becomes

sx: `1111 1111 1111 1101`

Play around here: http://www.convertforfree.com/twos-complement-calculator/

Truncating Numbers: Unsigned

We can also lose information with unsigned numbers:

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

Bit representation for x = 128000 (32-bit unsigned int):

`0000 0000 0000 0001 1111 0100 0000 0000`

Truncated unsigned short sx:

`1111 0100 0000 0000`

which equals 62464 decimal.

Converting back to an unsigned int, y = 62464

Overflow In Practice: PSY

YouTube: “We never thought a video would be watched in numbers greater than a 32-bit integer (=2,147,483,647 views), but that was before we met PSY. "Gangnam Style" has been viewed so many times we had to upgrade to a 64-bit integer (9,223,372,036,854,775,808)!”

Overflow in Signed Addition

In the news on January 5, 2022 (!):

ars TECHNICA

BIZ & IT TECH SCIENCE POLICY CARS GAMING & CUL

GOOD THING ANDROID IS GREAT AT ROLLING OUT UPDATES —

Google fixes nightmare Android bug that stopped user from calling 911

An integer overflow/underflow crash lets misbehaving apps lock users out of 911.

RON AMADEO - 1/5/2022, 3:09 PM

https://arstechnica.com/gadgets/2022/01/google-fixes-nightmare-android-bug-that-stopped-user-from-calling-911/

Overflow in Signed Addition

Signed overflow wraps around to the negative numbers.

```
#include<stdio.h>
#include<stdlib.h>
#include<limits.h> // for INT_MAX

int main() {
    int a = INT_MAX;
    int b = 1;
    int c = a + b;

    printf("a = %d\n",a);
    printf("b = %d\n",b);
    printf("a + b = %d\n",c);

    return 0;
}
```

```
$ ./signed_overflow
a = 2147483647
b = 1
a + b = -2147483648
```

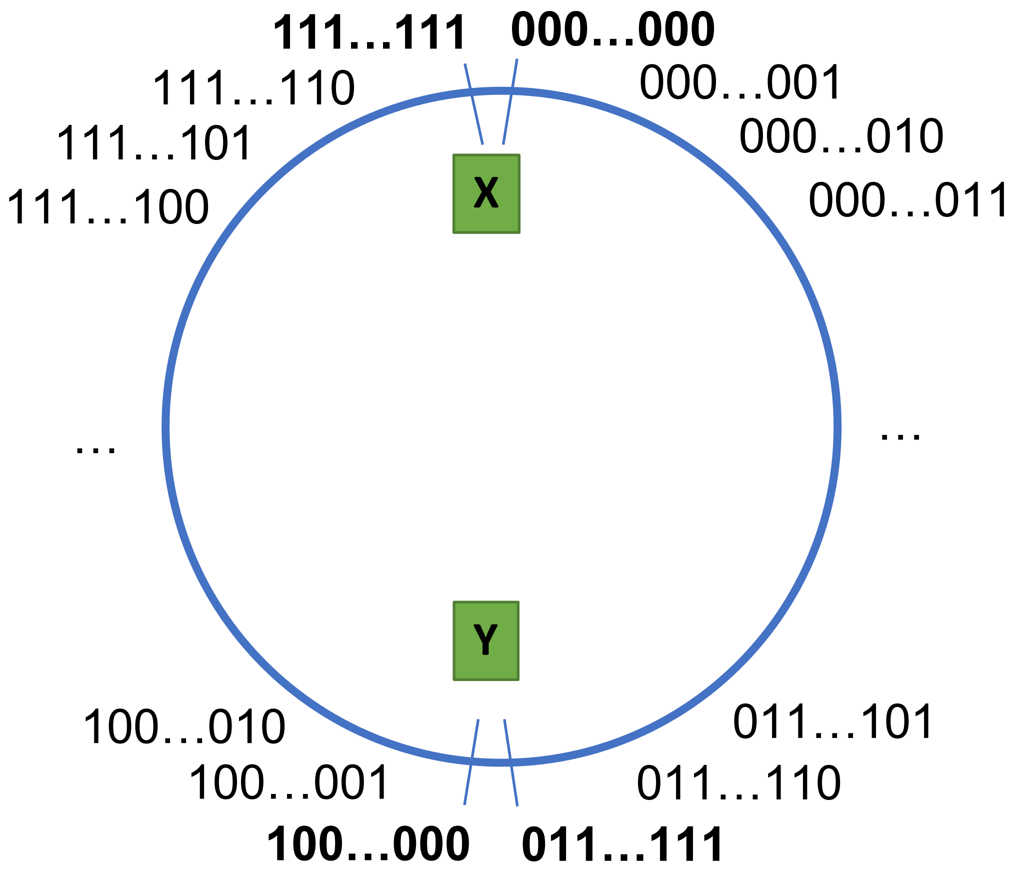
Technically, *signed integers in C* ***produce undefined behavior*** *when they overflow. On two's complement machines (virtually all machines these days), it does overflow predictably. You can test to see if your addition will be correct:*

```
// for addition
#include <limits.h>
int a = <something>;
int x = <something>;
if ((x > 0) && (a > INT_MAX - x)) /* `a + x` would overflow */;
if ((x < 0) && (a < INT_MIN - x)) /* `a + x` would underflow */;
```

Overflow

At which points can overflow occur for signed and unsigned int? *(assume binary values shown are all 32 bits)*

A. Signed and unsigned can both overflow at points X and Y
B. Signed can overflow only at X, unsigned only at Y
C. Signed can overflow only at Y, unsigned only at X
D. Signed can overflow at X and Y, unsigned only at X
E. Neither.



Go to

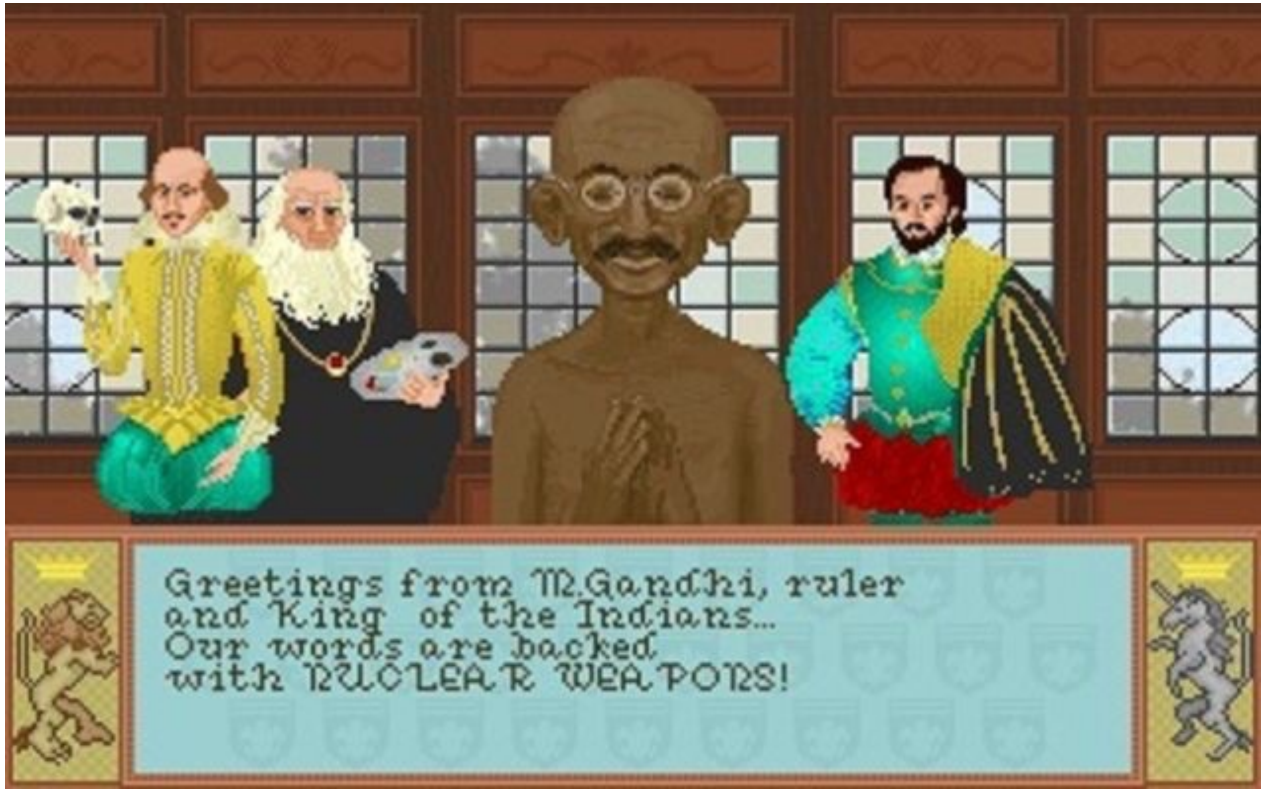
https://pollev.com/cs107summer

Overflow In Practice: Timestamps

- Many systems store timestamps as **the number of seconds since Jan. 1, 1970** in a **signed 32-bit integer**.
- **Problem:** the latest timestamp that can be represented this way is 3:14:07 UTC on Jan. 13 2038!

Overflow In Practice: Gandhi

- In the game "Civilization", each civilization leader had an "aggression" rating. Gandhi was meant to be peaceful, and had a score of 1.
- If you adopted "democracy", all players' aggression reduced by 2. Gandhi's went from 1 to **255**!
- Gandhi then became a big fan of nuclear weapons.



https://kotaku.com/why-gandhi-is-such-an-asshole-in-civilization-1653818245

Overflow in Practice:

- Pacman Level 256
- Make sure to reboot Boeing Dreamliners every 248 days
- Comair/Delta airline had to cancel thousands of flights days before Christmas
- Reported vulnerability CVE-2019-3857 in libssh2 may allow a hacker to remotely execute code
- Donkey Kong Kill Screen

printf and Integers

- There are 3 placeholders for 32-bit integers that we can use:
  - %d: signed 32-bit int
  - %u: unsigned 32-bit int
  - %x: hex 32-bit int
- **The placeholder—not the expression filling in the placeholder—dictates what gets printed!**

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?**

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);
```

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

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

Practice: Two's Complement

Fill in the below table:

It's easier to compute base-10 for positive numbers, so use two's complement first if negative.

	`char x = ____;`		`char y = -x;`	
	decimal	binary	decimal	binary
1.		`0b1111 1100`		
2.		`0b0001 1000`		
3.		`0b0010 0100`		
4.		`0b1101 1111`		

Practice: Two’s Complement

Fill in the below table:

It’s easier to compute base-10 for positive numbers, so use two’s complement first if negative.

	`char x = ____;`		`char y = -x;`	
	decimal	binary	decimal	binary
1.	-4	0b1111 1100	4	0b0000 0100
2.		0b0001 1000		
3.		0b0010 0100		
4.		0b1101 1111		

🤔

Practice: Two's Complement

Fill in the below table:

It's easier to compute base-10 for positive numbers, so use two's complement first if negative.

	`char x = ____;`		`char y = -x;`	
	decimal	binary	decimal	binary
1.	-4	0b1111 1100	4	0b0000 0100
2.	24	0b0001 1000	-24	0b1110 1000
3.	36	0b0010 0100	-36	0b1101 1100
4.	-33	0b1101 1111	33	0b0010 0001

Addressing and Byte Ordering

Every variable holds a value stored in memory.

A non-pointer variable (quietly) stores the address to that location in memory.

This means that when we use that variable it gives us the value at that location

A pointer (quietly) stores a location in memory, however the value at the location is another address.

Regardless of whether we are storing a value or an address, it is quite common for us to need more than one byte.

Addressing and Byte Ordering

The `int` type on our machines is 4 bytes long. So, how do we store 4 bytes, if memory is only byte-addressable?

We store it contiguously (back-to-back)!

That still leaves us with an important question, which way to go?
Right -> Left or Left -> Right?
We call this question the Endianness of the number

Lets begin by representing an `int` as an 8-digit hex numbers:

`0x01234567`

We can then separate out the bytes (remember 2 hex digits is 8 bits or 1 byte):

`0x   01  23  45  67`

Addressing and Byte Ordering

01 23 45 67

0000 0001 0010 0011 0100 0101 0110 0111

most significant → least significant

- Some machines choose to store the bytes ordered from least significant byte to most significant byte, called “little endian” (because the “little end” comes first).

- Other machines choose to store the bytes ordered from most significant byte to least significant byte, called “big endian” (because the “big end” comes first).

Addressing and Byte Ordering

- Our `0x 01 23 45 67` number would look like this in memory for a little endian computer (which, by the way, is the way the myth computers store ints):

byte:	67	45	23	01
address:	0x100	0x101	0x102	0x103

- A big-endian representation would look like this:

byte:	01	23	45	67
address:	0x100	0x101	0x102	0x103

Many times we don't care how our integers are stored, but in cs107 we will! Let's look at a sample program and dig under the hood to see how little-endian works.

Addressing and Byte Ordering

```c
#include<stdio.h>
#include<stdlib.h>

int   main() {
      // a variable
      int a = 0x01234567;

      // print the variable in big endian format
      printf("a's value: 0x%.8x\n",a);
      return 0;
}
```

GDB as an Interpreter

- `gdb live_session` run gdb on live_session executable
- `p` print variable (`p varname`) or evaluated expression (`p 3L << 10`)
  - `p/t, p/x` binary and hex formats.
  - `p/d, p/u, p/c`
- `<enter>` Execute last command again
- `q` Quit gdb

Important When first launching gdb:

- Gdb is not running any program and therefore can't print variables
- It can still process operators on constants

gdb on a program

- `gdb live_session` run gdb on executable
- `b` Set breakpoint on a function (e.g., `b main`) or line (`b 42`)
- `r 82` Run with provided args
- `n, s, continue` control forward execution (next, step into, continue)
- `p` print variable (`p varname`) or evaluated expression (`p 3L << 10`)
  - `p/t, p/x` binary and hex formats.
  - `p/d, p/u, p/c`
- `info` `args, locals`

Important: gdb does not run the current line until you hit "next"

gdb: highly recommended

At this point, setting breakpoints/stepping in gdb may seem like overkill for what could otherwise be achieved by copious **printf** statements.

However, gdb is incredibly useful for assign1 (and all assignments):

- A fast “C interpreter”: p + <expression>
  - Sandbox/try out ideas around bitshift operators, signed/unsigned types, etc.
  - Can print values out in binary!
  - Once you’re happy, then make changes to your C file
- **Tip**: Open two terminal windows and SSH into myth in both
  - Keep one for emacs, the other for gdb/command-line
  - Easily reference C file line numbers and variables while accessing gdb
- **Tip**: Every time you update your C file, **make** and then rerun gdb.

Gdb takes practice! But the payoff is tremendous! ©

gdb step, next, finish

I've seen a few students who have been frustrated with stepping through functions in gdb. Sometimes, they will accidentally step into a function like `strlen` or `printf` and get stuck.

There are three important gdb commands about stepping through a program:

step (abbreviation: s) : executes the next line and *goes into* function calls.

next (abbreviation: n) : executes the next line, and *does not go into function calls.* I.e., if you want to run a line with `strlen` or `printf` but don't want to attempt to go into that function, use **next**.

display (abbreviation: disp) : displays a variable (or other item) after each step.

finish (abbreviation: fin) : completes a function and returns to the calling function. This is the command you want if you accidentally go into a function like `strlen` or `printf`! This continues the program until the end of the function, putting you back into the calling function.

Bitwise Operations

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

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

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

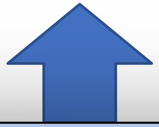
```
  0110
| 1100
  ----
  1110
```

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).

Bit Vectors and Sets

- We can use bit vectors (ordered collections of bits) to represent finite sets, and perform functions such as union, intersection, and complement.
- **Example:** we can represent current courses taken using a **char**.

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS110	CS107	CS106X	CS106B	CS106A

Bit Vectors and Sets

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS110	CS107	CS106X	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	CS110	CS107	CS106X	CS106B	CS106A

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

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

Bit Masking

- We will frequently want to manipulate or isolate out specific bits in a larger collection of bits. A **bitmask** is a constructed bit pattern that we can use, along with bit operators, to do this.
- **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	CS110	CS107	CS106X	CS106B	CS106A

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

Bit Masking

```
#define CS106A 0x1    /* 0000 0001 */
#define CS106B 0x2    /* 0000 0010 */
#define CS106X 0x4    /* 0000 0100 */
#define CS107  0x8    /* 0000 1000 */
#define CS110  0x10   /* 0001 0000 */
#define CS103  0x20   /* 0010 0000 */
#define CS109  0x40   /* 0100 0000 */
#define CS161  0x80   /* 1000 0000 */

char myClasses = ...;
myClasses = myClasses | CS107;   // Add CS107
```

Bit Masking

```
#define CS106A 0x1     /* 0000 0001 */
#define CS106B 0x2     /* 0000 0010 */
#define CS106X 0x4     /* 0000 0100 */
#define CS107  0x8     /* 0000 1000 */
#define CS110  0x10    /* 0001 0000 */
#define CS103  0x20    /* 0010 0000 */
#define CS109  0x40    /* 0100 0000 */
#define CS161  0x80    /* 1000 0000 */

char myClasses = ...;
myClasses |= CS107;      // Add CS107
```

Bit Masking

- **Example:** how do we update our bit vector to indicate we've *not* taken CS103?

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS110	CS107	CS106X	CS106B	CS106A

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

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

Bit Masking

- **Example:** how do we update our bit vector to indicate we've *not* taken CS103?

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS110	CS107	CS106X	CS106B	CS106A

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

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

Bit Masking

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

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS110	CS107	CS106X	CS106B	CS106A

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

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

Bit Masking

- **Example:** how do we check if we've *not* taken CS107?

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS110	CS107	CS106X	CS106B	CS106A

```
  00100011
& 00001000
----------
  00000000
```

```
char myClasses = ...;
if (!(myClasses & CS107)) {...
     // not taken CS107!
```

Bit Masking

- **Example:** how do we check if we've *not* taken CS107?

0	0	1	0	0	0	1	1
CS161	CS109	CS103	CS110	CS107	CS106X	CS106B	CS106A

```
  00100011      00000000
& 00001000    ^ 00001000
  --------      --------
  00000000      00001000
```

```
char myClasses = ...;
if ((myClasses & CS107) ^ CS107) {...
    // not taken CS107!
```

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 (>>)

The RIGHT SHIFT operator shifts a bit pattern a certain number of positions to the right. Bits shifted off the end are lost.

```
x >> k;     // evaluates to x shifted to the right by k bits
x >>= k;    // shifts x to the right by k bits
```

Question: how should we fill in new higher-order bits?

Idea: let's follow left-shift and fill with 0s.

```
short x = 2;   // 0000 0000 0000 0010
x >>= 1;       // 0000 0000 0000 0001
printf("%d\n", x); // 1
```

Right Shift (>>)

The RIGHT SHIFT operator shifts a bit pattern a certain number of positions to the right. Bits shifted off the end are lost.

```
x >> k;     // evaluates to x shifted to the right by k bit
x >>= k;    // shifts x to the right by k bits
```

Question: how should we fill in new higher-order bits?

Idea: let's follow left-shift and fill with 0s.

```
short x = -2; // 1111 1111 1111 1110
x >>= 1;      // 0111 1111 1111 1111
printf("%d\n", x); // 32767!
```

Right Shift (>>)

The RIGHT SHIFT operator shifts a bit pattern a certain number of positions to the right. Bits shifted off the end are lost.

```
x >> k;     // evaluates to x shifted to the right by k bit
x >>= k;    // shifts x to the right by k bits
```

Question: how should we fill in new higher-order bits?

Problem: always filling with zeros means we may change the sign bit.

Solution: let's fill with the sign bit!

Right Shift (>>)

The RIGHT SHIFT operator shifts a bit pattern a certain number of positions to the right. Bits shifted off the end are lost.

```
x >> k;     // evaluates to x shifted to the right by k bit
x >>= k;    // shifts x to the right by k bits
```

Question: how should we fill in new higher-order bits?

Solution: let's fill with the sign bit!

```
short x = 2;   // 0000 0000 0000 0010
x >>= 1;       // 0000 0000 0000 0001
printf("%d\n", x); // 1
```

Right Shift (>>)

The RIGHT SHIFT operator shifts a bit pattern a certain number of positions to the right. Bits shifted off the end are lost.

```
x >> k;     // evaluates to x shifted to the right by k bit
x >>= k;    // shifts x to the right by k bits
```

Question: how should we fill in new higher-order bits?

Solution: let's fill with the sign bit!

```
short x = -2; // 1111 1111 1111 1110
x >>= 1;      // 1111 1111 1111 1111
printf("%d\n", x); // -1!
```

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;
```

Bitwise Warmup

How can we use bitmasks + bitwise operators to…

`0b00001101`

1. …turn **on** a particular set of bits?

   `0b00001101`

   ____

   `0b00001111`

2. …turn **off** a particular set of bits?

   `0b00001101`

   ____

   `0b00001001`

3. …flip a particular set of bits?

   `0b00001101`

   ____

   `0b00001011` 🤔

Bitwise Warmup

How can we use bitmasks + bitwise operators to…

```
0b00001101
```

1. …turn **on** a particular set of bits? **OR**

```
  0b00001101
  0b00000010  |
--------------
  0b00001111
```

2. …turn **off** a particular set of bits? **AND**

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

3. …**flip** a particular set of bits? **XOR**

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

More Exercises

Suppose we have a 64-bit number. `long x = 0b1010010;`

How can we use bit operators, and the constant `1L` or `-1L` to…

- …design a mask that turns on the i-th bit of a number for any i (0, 1, 2, …, 63)?

- …design a mask that zeros out (i.e., turns off) the bottom i bits (and keeps the rest of the bits the same)?

More Exercises

Suppose we have a 64-bit number. `long x = 0b1010010;`

How can we use bit operators, and the constant `1L` or `-1L` to…

- …design a mask that turns on the i-th bit of a number for any i (0, 1, 2, …, 63)?

x | (1L << i)

- …design a mask that zeros out (i.e., turns off) the bottom i bits (and keeps the rest of the bits the same)?

x & (-1L << i)

🤔

On your own

- Print a variable
- Print (in binary, then in hex) result of left-shifting 14 and 32 by 4 bits.
- Print (in binary, then in hex) result of subtracting 1 from 128

`1 << 32`

- Why is this zero? Compare with `1 << 31`.
- Print in hex to make it easier to count zeros.

References and Advanced Reading

- **References:**
  - Two's complement calculator: http://www.convertforfree.com/twos-complement-calculator/
  - Wikipedia on Two's complement: https://en.wikipedia.org/wiki/Two%27s_complement
  - The `sizeof` operator: http://www.geeksforgeeks.org/sizeof-operator-c/

- **Advanced Reading:**
  - Signed overflow: https://stackoverflow.com/questions/16056758/c-c-unsigned-integer-overflow
  - Integer overflow in C: https://www.gnu.org/software/autoconf/manual/autoconf-2.62/html_node/Integer-Overflow.html
  - https://stackoverflow.com/questions/34885966/when-an-int-is-cast-to-a-short-and-truncated-how-is-the-new-value-determined

References and Advanced Reading

- **References:**
  - argc and argv: http://crasseux.com/books/ctutorial/argc-and-argv.html
  - The C Language: https://en.wikipedia.org/wiki/C_(programming_language)
  - Kernighan and Ritchie (K&R) C: https://www.youtube.com/watch?v=de2Hsvxaf8M
  - C Standard Library: http://www.cplusplus.com/reference/clibrary/
  - https://en.wikipedia.org/wiki/Bitwise_operations_in_C
  - http://en.cppreference.com/w/c/language/operator_precedence

- **Advanced Reading:**
  - After All These Years, the World is Still Powered by C Programming
  - Is C Still Relevant in the 21st Century?
  - Why Every Programmer Should Learn C