Quiz Time!

- Please sit with max 4 per table
- Keep all electronics away + have out only your writing utensils/printed resources
- You'll have 15 minutes to complete the quiz
- Good luck! ☺ 

- NOTE: If you are a CGOE student/have OAE accommodations, you should have heard from us about how to take your quiz!

CS107 Lecture 3

Byte Ordering & Bitwise Operators

Reading:

Bryant & O'Hallaron, Ch. 2.1

Announcements

- Assign 0 due tonight @ 11:59PM, late (without penalty) Sunday 11:59PM!
- We hope Lab 1 went well! Feel free to switch amongst labs if you have a conflict on the Lab signup form.
- Office Hours are after lecture M/W/F!

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)

- **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
…
```

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.

The `sizeof` Operator

As we have seen, integer types are limited by the number of bits they hold. On the 64-bit myth machines, we can use the `sizeof` operator to find how many bytes each type uses:

```
int main() {
    printf("sizeof(char): %d\n", (int) sizeof(char));
    printf("sizeof(short): %d\n", (int) sizeof(short));
    printf("sizeof(int): %d\n", (int) sizeof(int));
    printf("sizeof(unsigned int): %d\n", (int) sizeof(unsigned int));
    printf("sizeof(long): %d\n", (int) sizeof(long));
    printf("sizeof(long long): %d\n", (int) sizeof(long long));
    printf("sizeof(size_t): %d\n", (int) sizeof(size_t));
    printf("sizeof(void *): %d\n", (int) sizeof(void *));
    return 0;
}
```

```
$ ./sizeof
sizeof(char): 1
sizeof(short): 2
sizeof(int): 4
sizeof(unsigned int): 4
sizeof(long): 8
sizeof(long long): 8
sizeof(size_t):  8
sizeof(void *): 8
```

Type	Width in bytes	Width in bits
char	1	8
short	2	16
int	4	32
long	8	64
void *	8	64

Expanding Bit Representation

- 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
```

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 // -12345

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 // sx is still -3

This is -3! **If the number does "fit," it will convert fine.**

1111 1111 1111 1111 1111 1111 1111 1101 // y is 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 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 // sx is 62464

This is 62464! **Unsigned numbers can lose info, too.**

0000 0000 0000 0000 1111 0100 0000 0000 // y is 62464! 😭

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

- `int` type is 4 bytes long on our machines. How to store, if memory is only byte-addressable?

Addressing and Byte Ordering

- `int` type is 4 bytes long on our machines. How to store, if memory is only byte-addressable? **Contiguously, back-to-back!**
- How do we organize those bytes in a back-to-back fashion?
- Example: a 4-byte integer, 0x01234567

0x01	23	45	67
1st byte?	2nd byte?	3rd byte?	4th byte?

0x01	23	45	67
4th byte?	3rd byte?	2nd byte?	1st byte?

Some other ordering of bytes?

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

- A **little-endian** representation of `0x 01 23 45 67` would look like this in memory (which is how Myth computers store integers!):

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

Often, 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.

Introducing GDB

Is there a way to step through the execution of a program and print out values as it's running?
e.g., to view binary representations? **Yes!**

The GDB Debugger

- GDB is a **command-line debugger**, a text-based debugger with similar functionality to other debuggers you may have used, such as in Qt Creator
- It lets you put **breakpoints** at specific places in your program to pause there
- It lets you step through execution line by line
- It lets you print out values of variables in various ways (including binary)
- It lets you track down where your program crashed
- And much, much more!

GDB is essential to your success in CS107 this quarter! We'll be building our familiarity with GDB over the course of the quarter.

GDB on a Program

gdb live_session	run gdb on executable (e.g., live_session is my executable)
break	Set breakpoint on a function (e.g., b main) or line (b 42)
run 82	Run with provided args
next, step, continue	control forward execution (next, step into, continue)
print • **p/t, p/x** • **p/d, p/u, p/c**	print variable (p varname) or evaluated expression (p 3L == 10) binary and hex formats
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 vim, 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!

Avoiding Frustrations with GDB

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.

Addressing and Byte Ordering

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int   main() {
5        // a variable
6        int a = 0x01234567;
7
8       // print the variable in big endian format
9       printf("a's value: 0x%.8x\n", a);
10      return 0;
11 }
```

```
yasminealonso — yalonso@myth59: ~/sum26-cs107 — ssh yalonso@myth.stanford.edu — 118×50
[yalonso@myth59:~/sum26-cs107$ gcc -g -O0 -std=gnu99 big_endian.c -o big_endian
[yalonso@myth59:~/sum26-cs107$
[yalonso@myth59:~/sum26-cs107$ ./big_endian
a's value: 0x01234567
[yalonso@myth59:~/sum26-cs107$
[yalonso@myth59:~/sum26-cs107$ gdb big_endian
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
The target architecture is set to "i386:x86-64".
Reading symbols from big_endian...
[(gdb) break main
Breakpoint 1 at 0x1155: file big_endian.c, line 6.
[(gdb) run
Starting program: /afs/.ir.stanford.edu/users/y/a/yalonso/sum26-cs107/big_endian

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.ubuntu.com>
[Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
Downloading separate debug info for system-supplied DSO at 0x7ffff7fc3000
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at big_endian.c:6
6       int a = 0x01234567;
[(gdb) next
9       printf("a's value: 0x%.8x\n", a);
[(gdb) p/x a
$1 = 0x1234567
[(gdb) p &a
$2 = (int *) 0x7fffffffe7ec
[(gdb) x/16bx &a
0x7fffffffe7ec: 0x67    0x45    0x23    0x01    0x90    0xe8    0xff    0xff
0x7fffffffe7f4: 0xff    0x7f    0x00    0x00    0xca    0xa1    0xc2    0xf7
(gdb)
```

Note the ordering: `0x01234567` is stored as Little Endian!

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, 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
```

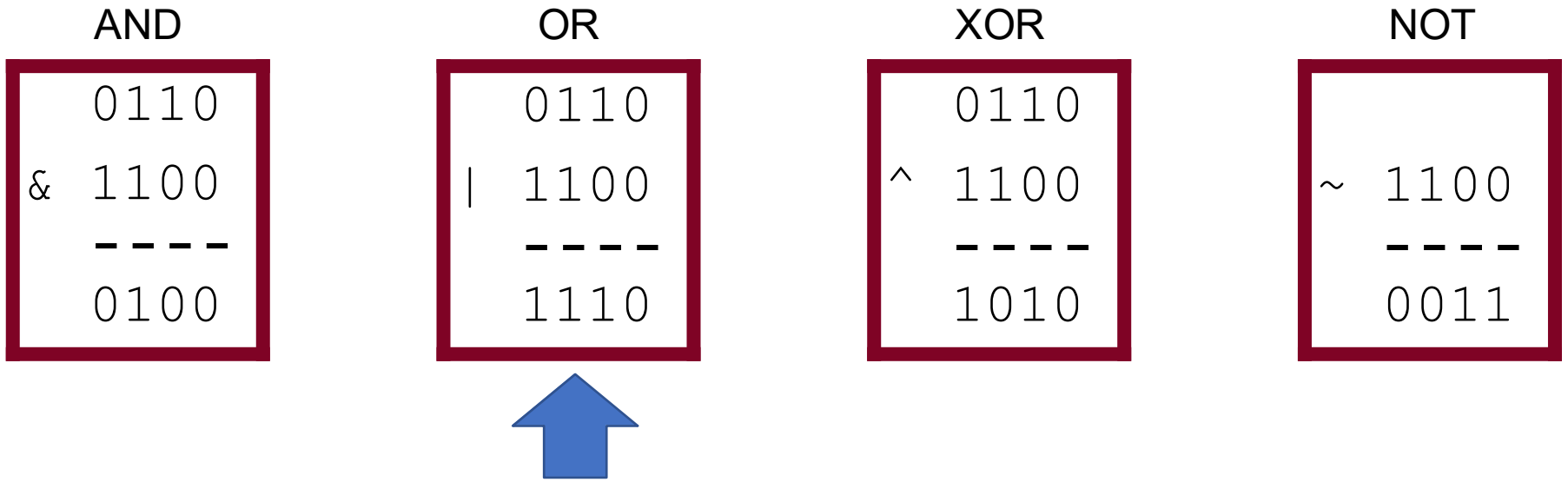
NOT

```
  
~ 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:



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).

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 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;          // 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)`

Shift Operation 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;
```

Bitmasks

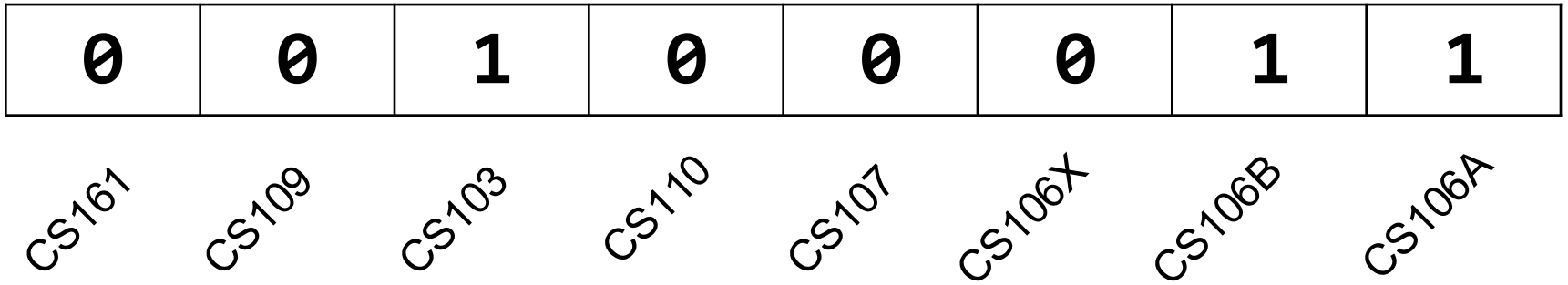
We will frequently want to manipulate or otherwise isolate specific bits in a larger collection of them. A **bitmask** is a constructed bit pattern that we can use, along with standard bit operators like **&**, **|**, **^**, **~**, **<<**, and **>>**, to do this.

Motivating Example: Bit vectors

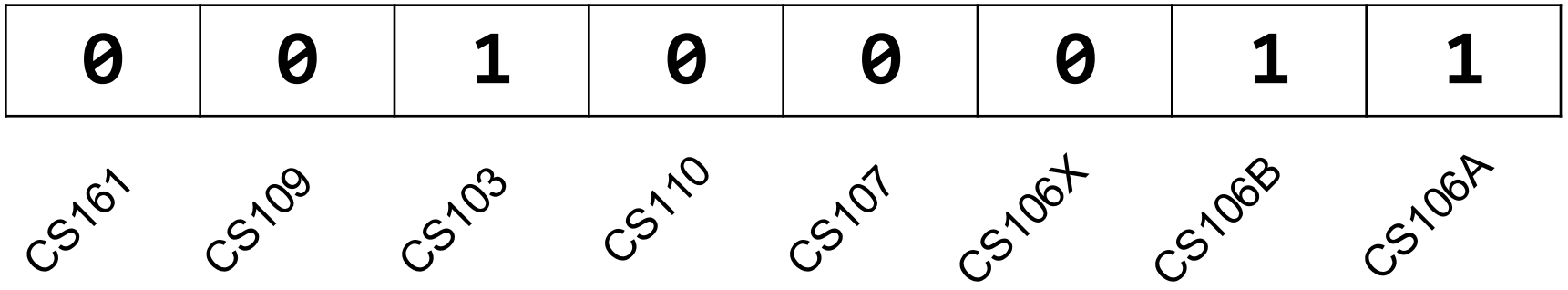
Aside: C++ relies on bit vectors to efficiently implement **vector<bool>**.

Bit Vectors and Sets

- 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`**.



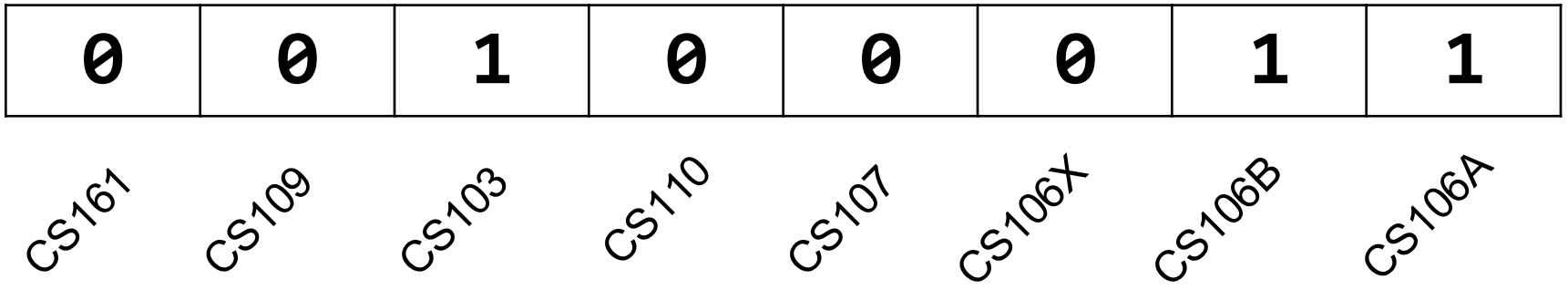
Bit Vectors and Sets



- How do we find the union of two sets of courses taken? Use OR:

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

Bit Vectors and Sets

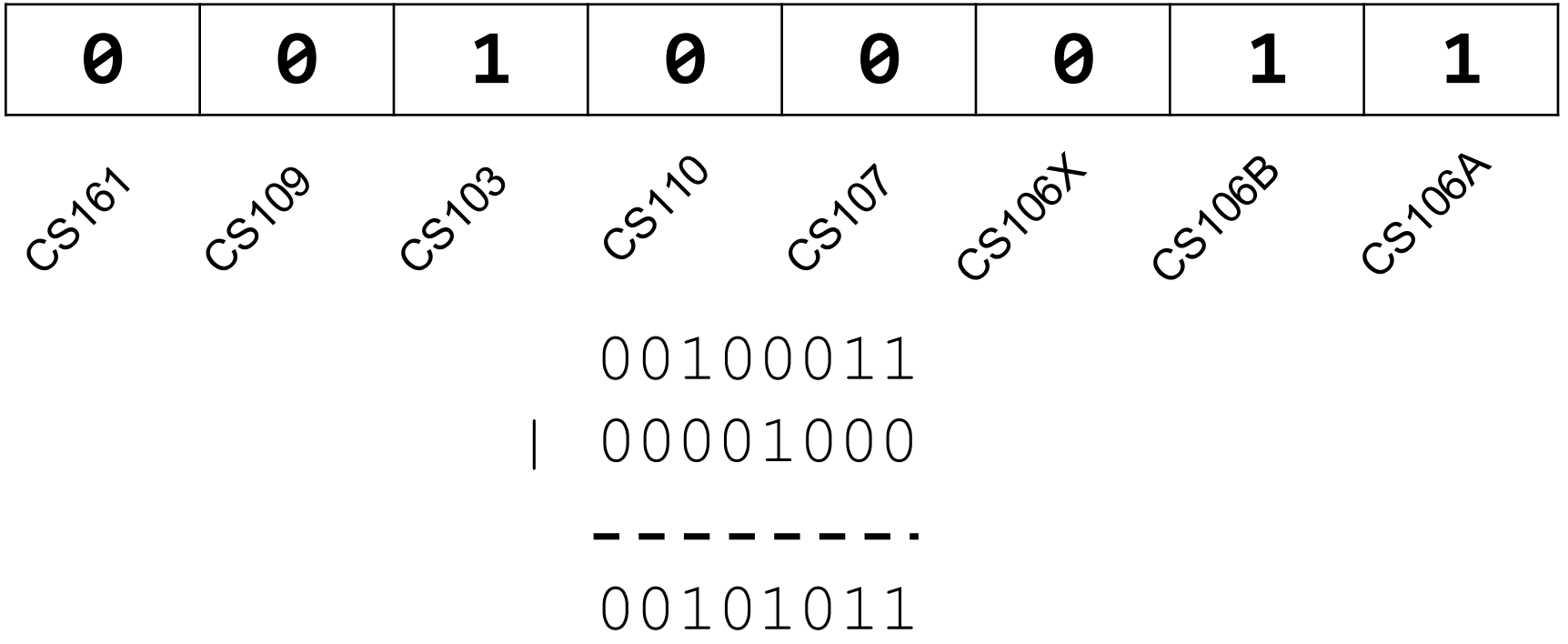


- How do we find the intersection of two sets of courses taken? Use AND:

```
  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?



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
```

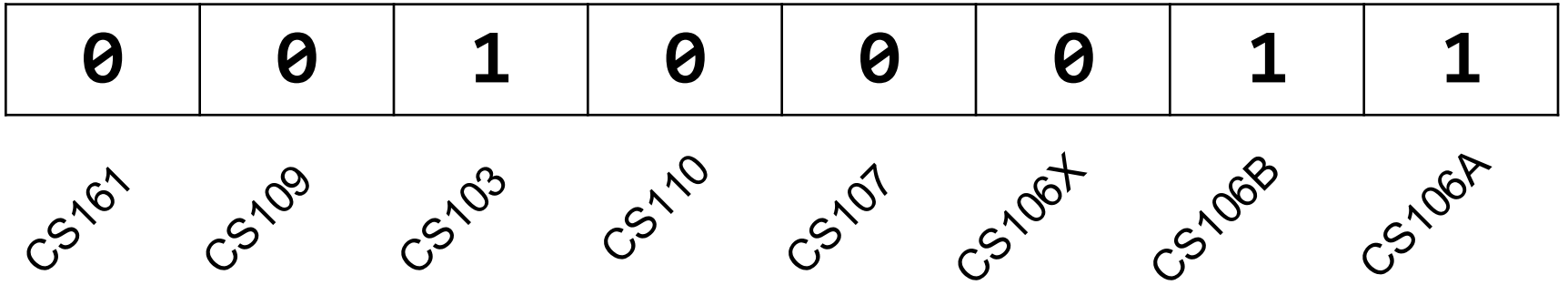
Bit Masking

```
#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?

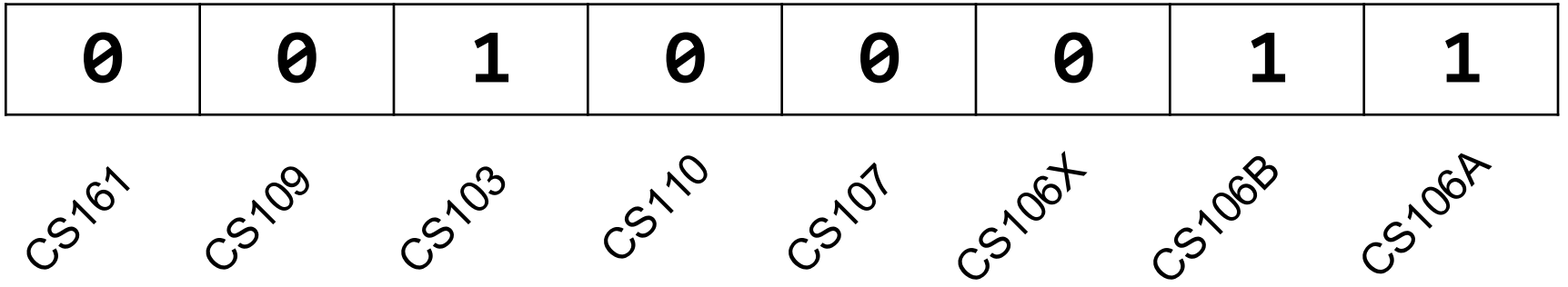



```
  00100011
& 11011111
----------
  00000011
```

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

Bit Masking

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

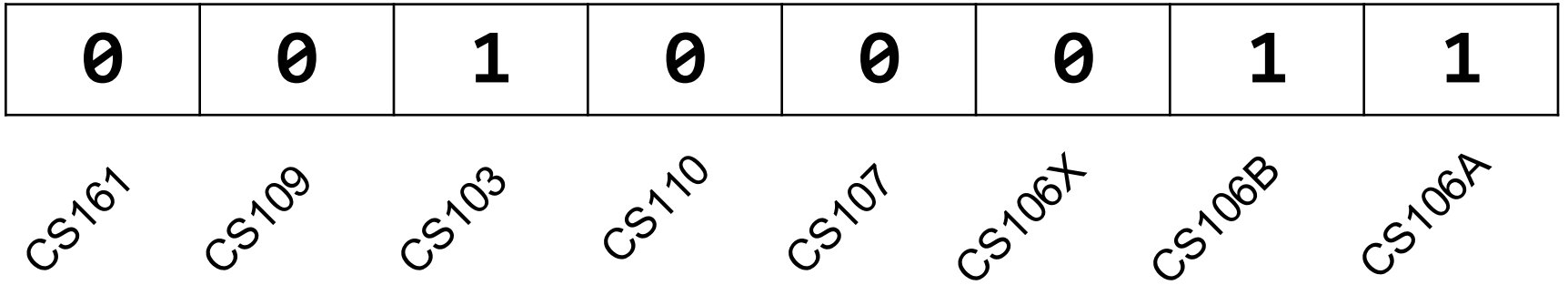



```
  00100011
& 11011111
----------
  00000011
```

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

Bit Masking

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

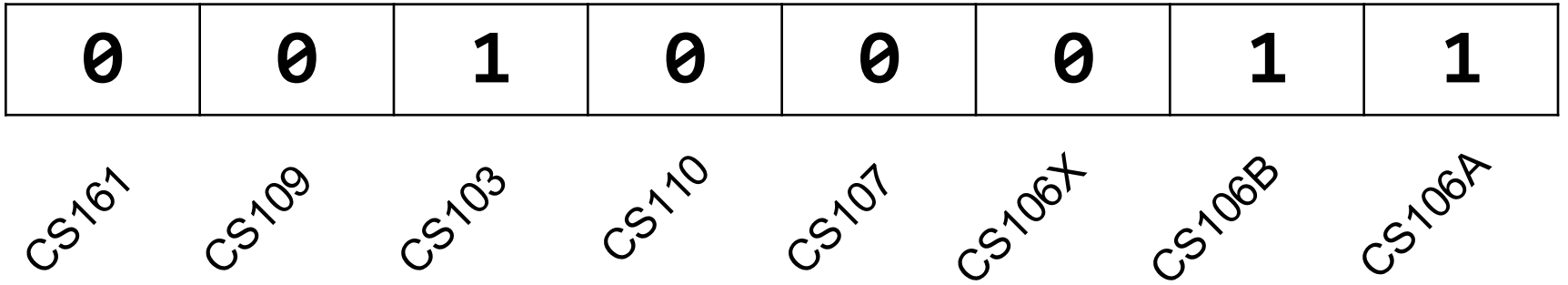



```
  00100011
& 00000010
  --------
  00000010
```

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

Bit Masking

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

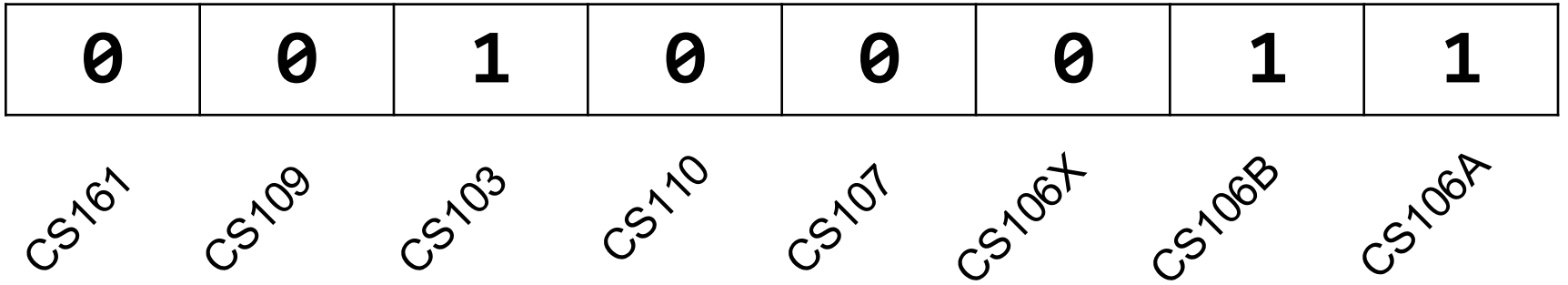



```
  00100011
& 00001000
  --------
  00000000
```

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

Bit Masking

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




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

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

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

Bitwise Practice

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`

   `0b00000111`

Bitwise Practice

How can we use bitmasks + bitwise operators to…

```
0b00001101
```

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

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

OR

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

```
  0b00001101
--------------
  0b00001001
```

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

```
  0b00001101
--------------
  0b00000111
```

Bitwise Practice

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?

```
  0b00001101
------------
  0b00000111
```

Bitwise Practice

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

Extra Practice: Two's Complement

Fill in the below table:

	`char x = ___;`		`char y = -x;`	
	decimal	binary	decimal	binary
1.				
2.				
3.				
4.				

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

Extra Practice: Two's Complement

Fill in the below table:

	`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

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

Extra Practice: Limits + Comparisons

1. What is the…

	Largest unsigned?	Largest signed?	Smallest signed?
`char`			
`int`			

Extra Practice: Limits + Comparisons

1. What is the…

	Largest unsigned?	Largest signed?	Smallest signed?
`char`	$2^8 - 1 = 255$	$2^7 - 1 = 127$	$-2^7 = -128$
`int`	$2^{32} - 1 = 4294967295$	$2^{31} - 1 = 2147483647$	$-2^{31} = -2147483648$

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

Extra Practice: Limits + 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`

Extra Practice: Limits + Comparisons

2. Will the following `char` comparisons evaluate to `true` or `false`?

i. `-7 < 4` **`true`**

ii. `-7 < 4U` **`false`**

iii. `(char) 130 > 4` **`false`**

iv. `(char) -132 > 2` **`true`**

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