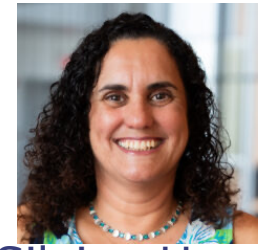
Welcome to 6.1903

Introduction to C and Assembly Language

Fall 2026

6.190/6.0004 Course Staff

Instructors



Silvina Hanono Wachman
silvina@mit.edu

Will Leiserson
yoonhkim@mit.edu

Martin Rinard
rinard@csail. mit.edu

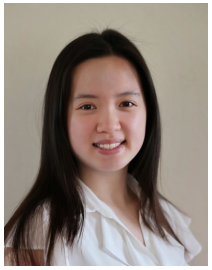
Joe Steinmeyer
jodalyst@mit.edu

Teaching Assistants

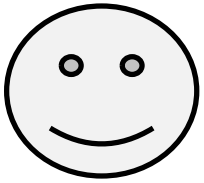
Justin Malloy

Elise Wingard

Kenny Zhang



Abrianna



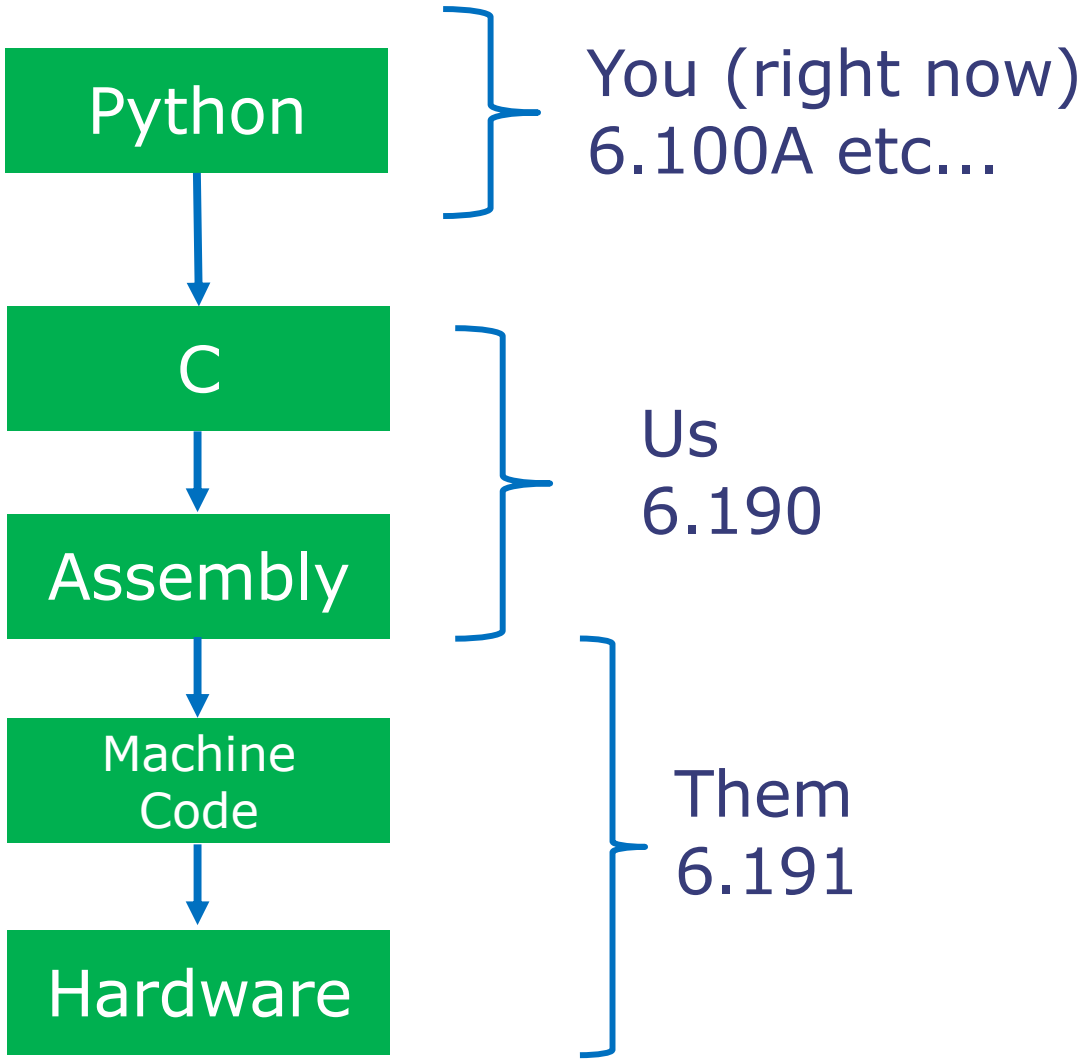
Alex

May

Course Goals

- How to write basic programs in C
  - Understand the difference between Python and C
- Understand how memory is managed in a computer and in C!
  - What happens when you declare a variable
  - How to access complex data structures in memory
- How high-level programs are translated to machine level (assembly language) programs that can directly be executed on our hardware.
- Prepare you for 6.191 (6.004) (Computation Structures) and beyond...

Big Idea


Python
C
Assembly
Machine Code
Hardware
You (right now)
6.100A etc...
Us
6.190
Them
6.191

Course Operation

- Basically have ~six/seven weeks of work spread over the whole semester...SO LOOK AT CALENDAR!!
- Monday: 90-minute lecture approximately every two weeks: 9:30-11am in 32-123: slides and reference materials on website
- Tuesday: 90-minute recitation approximately every two weeks seven recitation sections (see website for recitation rooms): work through problems using skills and concepts from previous lecture
- Wednesday: 2.5hr lab approximately every two weeks. four Wed lab sections in 38-530: work using ESP32 embedded system
  - Post-Lab comes out after Lab is complete

Course Operation Continued

- All assignments for a week are then due at 11:59pm on Sunday evening
  - Online exercises
  - Completed lab checkoffs in case you didn't complete them in class
  - Completed post lab checkoffs
- Three quizzes:
  - In lecture time Monday 9/28, 11/2, 11/16
- Two evening exams:
  - 7:30-9:30 on 10/15 and 12/3
- No calculator on either
- No "cheat-sheets" on either

Lab Attendance

- We expect you to attend your lab session and to work for **the 2.5 hours allotted**. Lab attendance is **mandatory**!
- You need to show up on time and stay the full lab period.
- Sign *in* with QR Code and then work on the assignment.

- ***WE EXPECT YOU TO WORK ON LAB ASSIGNMENT DURING LAB TIME. You will lose points if you leave early, skip lab, show up late.***

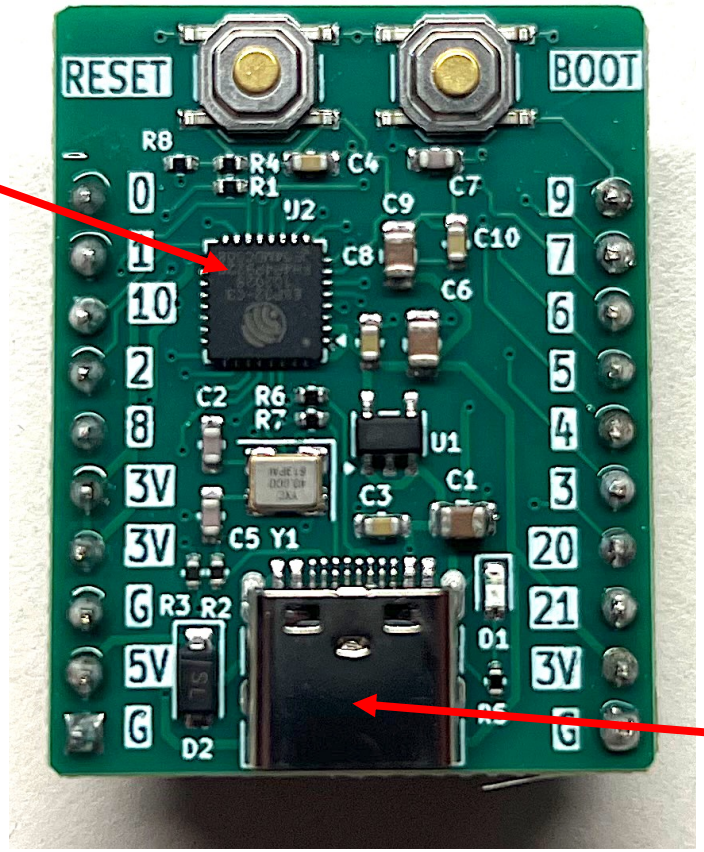
- **Please review course information for details**

Recitation Attendance

- Tuesdays...
- 3% of total grade
- Show up and work through problems to review some lecture material and prep for lab. Bring your laptop.
- OR if you do the required portion of the recitation work by 11:59pm on the recitation day on your own, then you don’t need to attend in person. This will be counted as “attending”

Labkit (Based around RISCV Microcontroller)

ESP32-C3
RV32IMC microcontroller
Running at 160 MHz



USB Connector

Lab Kit

- You'll get a lab kit: Pick it up at 38-530 after class (11:15am-12:30pm) or during Tuesday office hours (in 38-530)

- You will use this to do the labs and post-labs.
- You must bring your lab kit to each lab.

- **You must return this lab kit at the end of the quarter to pass the class.**
- **You must take care of your labkit!!!**

Grading

- Components:
  - Exercises: 6%
  - Labs: 18%
  - Post-Labs: 9%
  - Recitation Attendance: 3%
  - Three quizzes: 9%
  - Exam 1: 25%
  - Exam 2: 30%

- *Additionally: to pass the class all lab and postlab checkoffs must be completed*

Resources

- The course website has up-to-date information, handouts, and references to supplemental reading: https://llp.mit.edu/6190/FA26
- Piazza
  - Fastest way to get your questions answered
  - All course announcements will be made on Piazza
- We will hold office hours to help you with exercises, labs, post-labs, and general questions
  - Office hours are in 38-530
  - Office hours schedule is available on course website
  - Help queue also available through course website

AI Policy

- We expect everything you do in this class to be your own work.
- You're here to learn, not to have some other thing do work for you.
- If we catch you using AI, agents, etc...on your assignments, there will be scores of 0 and not good letter grades like F getting handed out as well as letters to the COD getting filed.
- Do the work! We're here to support you! Don't cheat yourself.

Course Material

Now the stuff.....

Part 1: Information

What *is* Information?

- Anything that provides an answer to a question of some kind
- Anything that can resolve uncertainty…
- And the more uncertainty you resolve, the more information you are providing
- Fundamental ***unit*** is the **bit**: one yes/no amount of information

Bits in Information

- As you increase the amount of information, the amount of bits goes up as well:
  - Do you need to represent/differentiate 4 things?
  - You need 2 bits:
    - 0: no no (00)
    - 1: no yes (01)
    - 2: yes no (10)
    - 3: yes yes (11)

$$n = \log_2(m)$$

$n =$ **Number of bits**

$m =$ **Number of values**

Claude Shannon
Formalized this in 1948

Goes Both Ways

- Do you have 8 bits? You can represent/specify 256 things

$2^n = m \quad 2^8 = 256$

00000000

through

11111111

Usually lump 8 bits into a "byte"

- Is a "thing" capable of specifying one of 2341 options?

That thing contains

$\log_2(2341) = 11.19$ bits

…of information

Representing Information

- There are two primary entities in electronics: voltage and current
- Most information in modern EECS is expressed in voltage variations, although current plays an important part too.
- A voltage, either static (fixed in time) or dynamic (changing in time), conveys meaning based upon its value.
- A system where voltage can be expressed and interpreted as **many values** is an **Analog** system:
- A system where voltage is limited (almost always to two categories of values), is a **Digital** system:

Differences?

- With **Analog**, theoretically with a single voltage measurement you can represent hundreds/thousands/many of things:
  - For example: a voltage varying from 0 to 10V in 0.01 V steps means a single voltage can represent up to 1000 options or $\mathbf{\log_2(1000) \approx 9.96}$ **bits** of information! Easy!
- With **Digital**, theoretically with a single voltage measurement you can represent two things:
  - For example, a voltage is limited to "high" (5V to 10V) or "low" (0V to 5V). That means voltage reading represents 2 options or $\mathbf{\log_2(2) = 1}$ **bit** of information! ☹

Analog is Obviously the Way to Go

- Can represent way more stuff (more information) with a single voltage measurement!
- More is better…(think about cookies or money or cats)
- Analog voltages must obviously what we use…right?
- Right?

- Right…?

- Tell me I'm right...

NO Digital Is It!

- Almost all modern electronics use a digital scheme to represent and transfer information (only two levels for their voltages!)

What is unique about Gen Z?

While there are substantive differences within the cohort known as Gen Z, there are a few commonalities its members share.

As the first real digital natives, Gen Zers—speaking generally—are *extremely online*. Gen Zers are known for working, shopping, dating, and making friends online; in Asia, Gen Zers spend six or more hours per day on their phones.

Digital natives often turn to the internet when looking for any kind of information, including news and reviews prior to making a purchase. They flit between sites, apps, and social media feeds, each one forming a different part of their online ecosystem. Having grown up with social media, Gen Zers curate their online selves more carefully than those in prior generations have, and they are more likely to turn to trends of anonymity, more personalized feeds, and a smaller online presence, even as they voraciously consume media online.

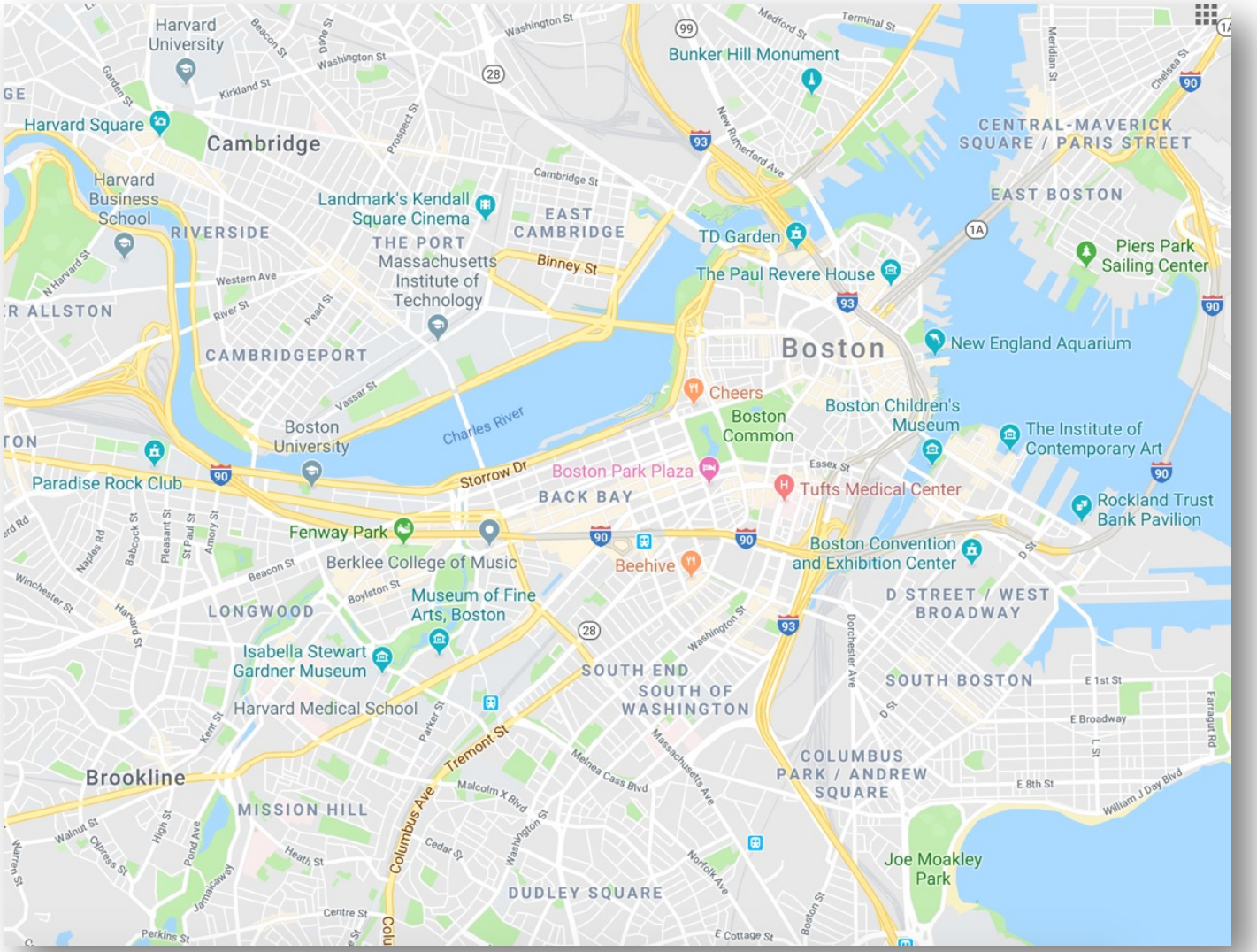
https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-gen-z

Raises an Interesting Question...

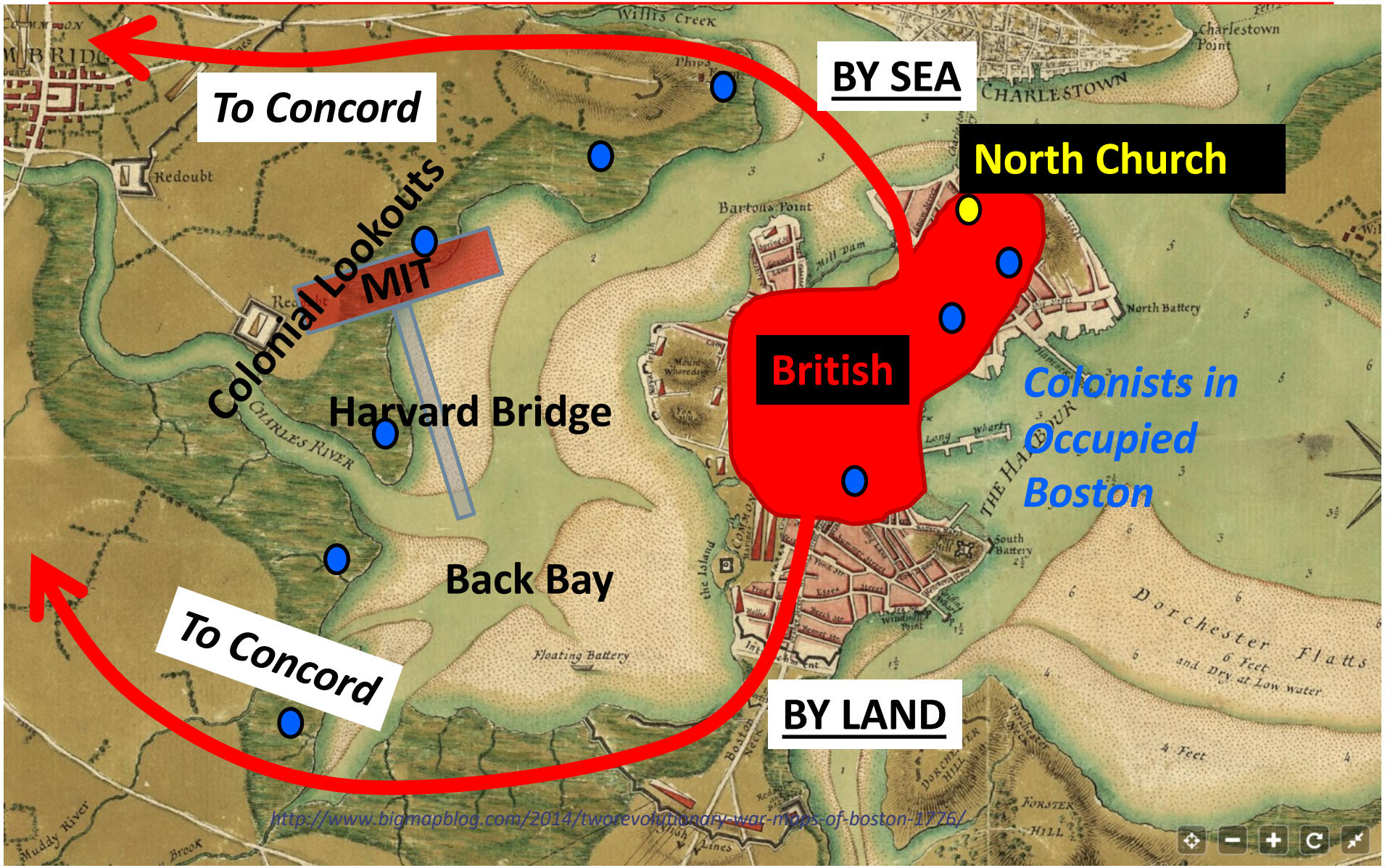
- Why did we choose an information-encoding scheme that seems so less “efficient”?
- If you look at any computer’s physical memory with a measurement device all you would see are huge chains of high or low voltages (represented as 1’s and 0’s):
  - 0b10101111000111001001110010110 10, etc…
- You do not see analog voltage chains:
  - 243, 112, -15, 67, 1, 0, 889, 123,-112, etc…

- To understand why we have to go back in time…

Boston 2026


Harvard University
Harvard Square
Cambridge
Harvard Business School
RIVERSIDE
Landmark's Kendall Square Cinema
EAST CAMBRIDGE
THE PORT
Massachusetts Institute of Technology
Binney St
CAMBRIDGEPORT
Boston University
Charles River
Paradise Rock Club
Storrow Dr
Fenway Park
Berklee College of Music
Museum of Fine Arts, Boston
LONGWOOD
Isabella Stewart Gardner Museum
Harvard Medical School
Brookline
MISSION HILL
Bunker Hill Monument
CENTRAL-MAVERICK SQUARE / PARIS STREET
EAST BOSTON
TD Garden
The Paul Revere House
Piers Park Sailing Center
Boston
New England Aquarium
Cheers
Boston Common
Boston Children's Museum
The Institute of Contemporary Art
Boston Park Plaza
BACK BAY
Tufts Medical Center
Rockland Trust Bank Pavilion
Beehive
Boston Convention and Exhibition Center
D STREET / WEST BROADWAY
SOUTH END
SOUTH OF WASHINGTON
SOUTH BOSTON
COLUMBUS PARK / ANDREW SQUARE
DUDLEY SQUARE
Joe Moakley Park
Tremont St
Columbus Ave

Boston in 1775



Boston in 1775

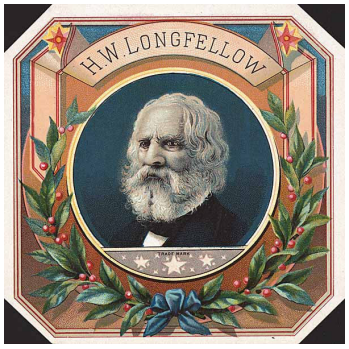
He said to his friend, "If the British march
By land or sea from the town to-night,
Hang a lantern aloft in the belfry-arch
Of the North-Church-tower, as a signal-light,—

CENSORED...Must decide an information representation scheme to express how the British are coming

And I on the opposite shore will be,
Ready to ride and spread the alarm
Through every Middlesex village and farm,
For the country-folk to be up and to arm."

…

-Henry Wadsworth Longfellow



https://poets.org/poem/paul-reveres-ride

Analog Solution

- Use One Lantern
- Turn lantern on full if by land
- Turn lantern on half if by sea

*Encode **three** things with one lantern*

One of the Paul Revere Lanterns

From Far Distance at Night:

By Land:

OR

By Sea:

OR

Not Coming Tonight:

From Far Distance at Night (WITH FOG):

They're coming?:

OR

They're coming?:

OR

Not Coming:

http://www.concordmuseum.org/lantern.php

The Digital Solution

- Hang **one** lantern if by land
- Hang **two** lanterns if by sea
- Hang **none** if they're not coming

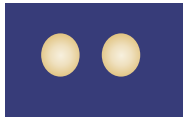
*Each lantern encodes **two** things*

From Far Distance at Night:

By Land:

OR

By Sea:



OR

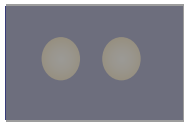
Not Coming Tonight:

From Far Distance at Night (WITH FOG):

By Land:

OR

By Sea:



OR

Not Coming Tonight:

Feel the Noise

- So...yeah...a voltage varying from 0 to 10V in 0.01 V steps means a single reading can contain up to 1000 options or $\log_2(1000) \approx 9.96$ bits of information.
- ...*theoretically...*
- **BUT, BUT BUT!!!** If you get random voltage swings of up to +/- 0.1V during transmission/storage…
- When you read back…does 2.01V mean 2.01 or is a 1.98 with +0.03V of noise or 2.02 with -0.01V of noise? Who knows? Lost!!!
- Encoding information in analog is highly susceptible to noise

Digital Is the Way to Go

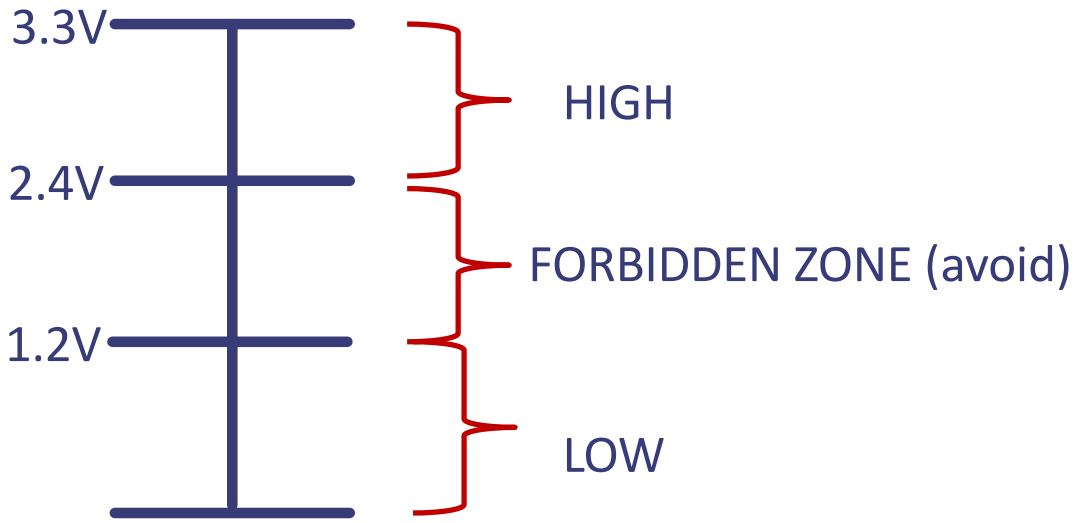
- Theoretically, analog signals are the way to go.
  - You could represent many, many things with one value
- But our ability to measure that signal is impaired because of noise, so what's the point?
- Digital Wins Out

Digital Solution

- In a digital system, for example, voltage can be 3.3V or 0V. That means voltage reading represents 2 options or $\log_2(2) = 1$ bit of information!
- Let's say we treat everything within 1V of 3.3V as "1" and everything within 1V of 0V as "0".
- Then if we send a 3.3V and receive a 3.2V we know what we mean even if it isn't perfect!
- Convey *less* information for a given measurement, but do so way more robustly

Digital in Electronics ("Digital Contract")

- Use voltages to represent the yes/no 1/0:
  - High Voltage: 1
  - Low Voltage: 0
- More specifically we set up a "contract" which all parts agree to:
  - Above a certain amount is 1
  - Below a certain amount is 0



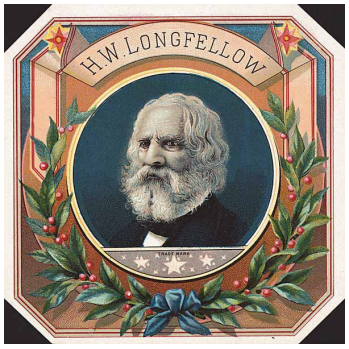
Contract for processor in our lab kit

Boston in 1775

He said to his friend, "If the British march
By land or sea from the town to-night,
Hang a lantern aloft in the belfry-arch
Of the North-Church-tower, as a signal-light,—
One if by land, and two if by sea;
And I on the opposite shore will be,
Ready to ride and spread the alarm
Through every Middlesex village and farm,
For the country-folk to be up and to arm."

…

-Henry Wadsworth Longfellow



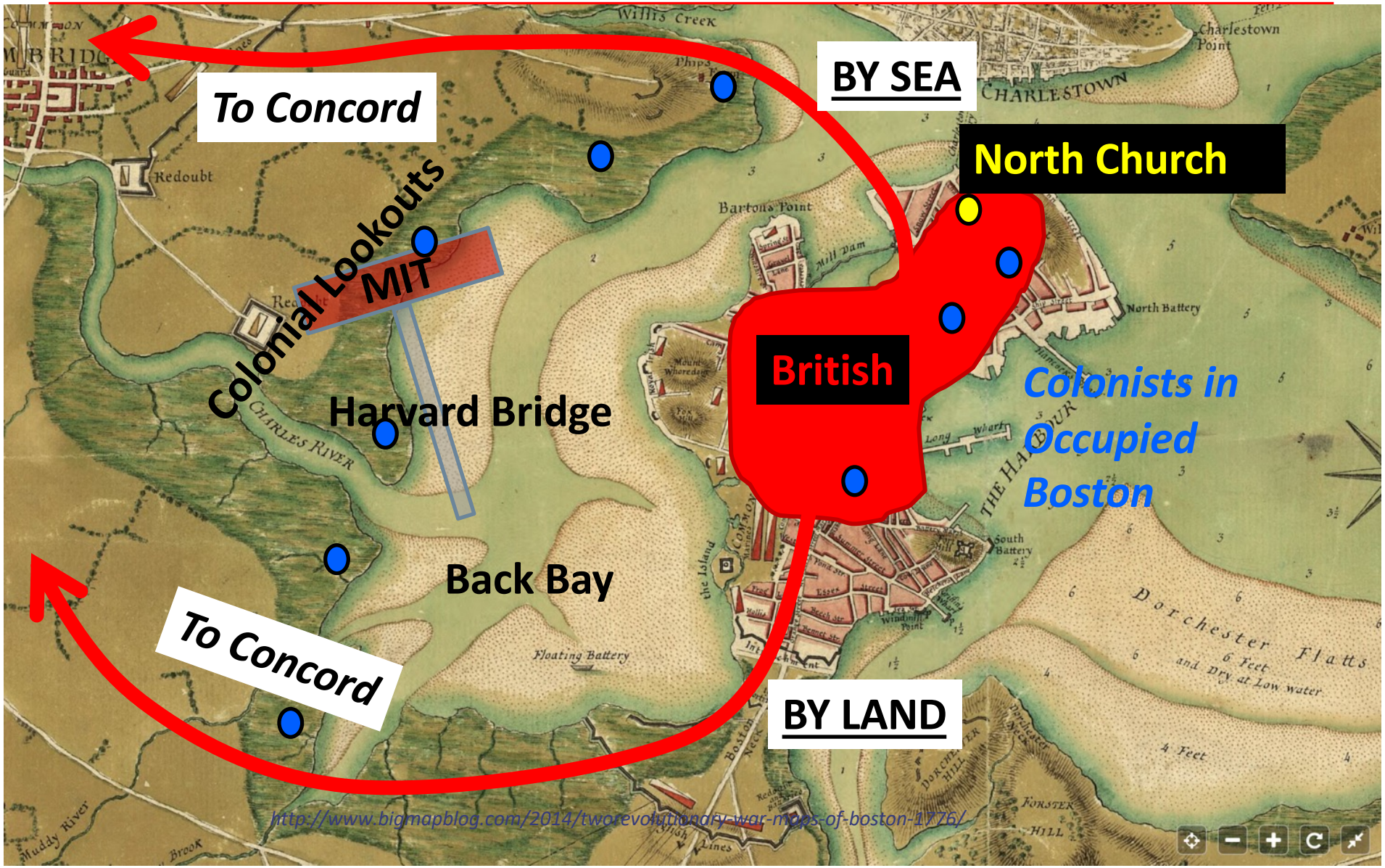
Digital contract in lantern form

https://poets.org/poem/paul-reveres-ride

Scaling It

How to Send More Data

Boston in 1775



Representing More Information

He said to his friend, "If the British march
By land or sea from the town to-night,
Hang a lantern aloft in the belfry-arch
Of the North-Church-tower, as a signal-light,—
One if by land, and two if by sea;

…and also tell me the hour that they're leaving and how many troops they're bringing!

And I on the opposite shore will be,
Ready to ride and spread the alarm
Through every Middlesex village and farm,
For the country-folk to be up and to arm."

Information Budget

- Coming by Land, coming by Sea, not coming?:
  - Three total options… $\text{ceil}(\log_2(3)) =$ **2 bits**
- Hour of day?:
  - 24 options…$\text{ceil}(\log_2(24)) =$ **5 bits**
- Number of troops?
  - Historians estimate about 4,000 troops garrisoned in Boston leading up to Battles of Lex/Concord, so assuming any integer number of troops in range [0 to 4000]
  - So: $\text{ceil}(\log_2(4000)) =$**12 bits**

Representing More Information

He said to his friend, "If the British march
By land or sea from the town to-night,
Hang a lantern aloft in the belfry-arch
Of the North-Church-tower, as a signal-light,—
One if by land, and two if by sea;

And an additional five lanterns conveying time of day;

And and additional twelve lanterns to indicate the number of soldiers;

And I on the opposite shore will be,
Ready to ride and spread the alarm
Through every Middlesex village and farm,
For the country-folk to be up and to arm."

So its April 18, 1775…

- You're the Revere's BFF in Boston spying on the British. You've got your lanterns and matches ready.
- You overhear that the British are going to do a night march to get the weapons at Concord:
  - Taking boats (aka "by sea")
  - Leaving at 9pm (21:00 hours)
  - 2027 soldiers are going to march out
- It is time. You must send your message digitally
  - First one ("by sea") is special encoding: not number
  - Second two are numbers

Numbers

- Encoding numbers digitally isn't just "encoding", it is a legitimate numerical base which you can work with….Base 2!
- You can add, subtract, multiply, divide and do everything else
- Some things are more convenient than others:
  - For example in a decimal (base 10), dividing and multiplying by 10 are easy
  - In Binary (base 2), dividing and multiplying by 2 are easy/trivial

Encoding Positive Integers

It is natural to encode positive integers as a sequence of bits. Each bit is assigned a weight. Ordered from right to left, these weights are increasing powers of 2. The value of an N-bit number is given by the following formula:

$$v = \sum_{i=0}^{N-1} 2^i b_i$$

2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	1	1	1	1	0	1	0	0	0	0

What value does 011111010000 encode?

$V = 0*2^{11} + 1*2^{10} + 1*2^9 + \ldots$

$= 1024 + 512 + 256 + 128 + 64 + 16 = 2000$

Smallest number? 0 *Largest number? 2^N-1*

Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

Base 10	Base 2
1 ← *carry* 14 + 7 *21*	*111* 1110 + 111 *10101*
-1 ← *borrow* 14 − 7 *07*	*-1-1-1* 1110 − 111 *0111*

Overflow

- If we use a fixed number of bits, addition and other operations may produce results outside the range that the output can represent
  - This is known as an *overflow*
  - For example if our answer is limited to four bits:

```
  111
  1110
+ 0111
------
 10101
```

  - Same thing can happen on subtraction (call that an *underflow*)
- We'll worry about negative numbers next week.

Modular Arithmetic

- Overflow and Underflow can be bad, but they can also be good. They give us modular arithmetic (in powers of 2 for free!)

$$\begin{array}{r} \textit{111} \\ 1110 \\ +\ 0111 \\ \hline \textit{10101} \end{array}$$

- Limiting output to N bits is same as $\%2^N$
- So above is same as $(14 + 7)\%16 = 5$

So its April 18, 1775…

- Back to Boston...
- You overhear that the British are going to do a night march to get the weapons at Concord:
  - Taking boats (by sea)
  - Leaving at 9pm (21:00 hours)
  - 2027 soldiers are going
- It is time. You must send your message in binary
  - First one is special encoding: not number
  - Second two are numbers

To Encode 21 in Binary

- We need to encode 21 into base-2 binary for our lantern display
- We need an "algorithm" How would you do that? The same way you'd do it in base 10 actually except we're just in base 2.
  1. Find the biggest power of 2 you can fit in the number, mark a 1 in that place, subtract the power of 2 from number...
  2. Repeat for next smallest power of 2 until leftover is 0

Encoding

START: 21

2^4 (16) fit in 21?

Yes: 0b1
21-16 = 5

2^3 (8) fit in 5?

No: 0b10
5-0 = 5

2^2 (4) fit in 5?

Yes: 0b101
5-4 = 1

2^1 (2) fit in 1?

No: 0b1010
1-0 = 1

2^0 (1) fit in 1?

Yes: 0b10101
1-1 = **0**

Answer:
0b10101

The leading "0b" means we're in base 2

To Encode 2027 in binary

- We could also encode 2027 into base-2 binary for our lantern display.
- We need an "algorithm" How would you do that? The same way you'd do it in base 10 actually except we're just in base 2.
  1. Find the biggest power of 2 you can fit in the number, mark a 1 in that place, subtract the power of 2 from number.
  2. Repeat for next smallest power of 2 until leftover is 0

Encoding (do for practice...)

START: 2027

2^{11} (2048) fit in 2027?

No: 0b0
2027-0 = 2027

2^{10} (1024) fit in 2027?

Yes: 0b01
2027-1024 = 1003

2^{9} (512) fit in 1003?

Yes: 0b011
1003-512 = 491

2^{8} (256) fit in 491?

Yes: 0b0111
491-256 = 235

2^{7} (128) fit in 235?

Yes: 0b01111
235-128 = 107

2^{6} (64) fit in 107?

Yes: 0b011111
107-64 = 43

2^{5} (32) fit in 43?

Yes: 0b0111111
43-32 = 11

2^{4} (16) fit in 11?

No: 0b01111110
11-0 = 11

2^{3} (8) fit in 11?

Yes: 0b011111101
11-8 = 3

2^{2} (4) fit in 3?

No: 0b0111111010
3-0 = 3

2^{1} (2) fit in 3?

Yes: 0b01111110101
3-2 = 1

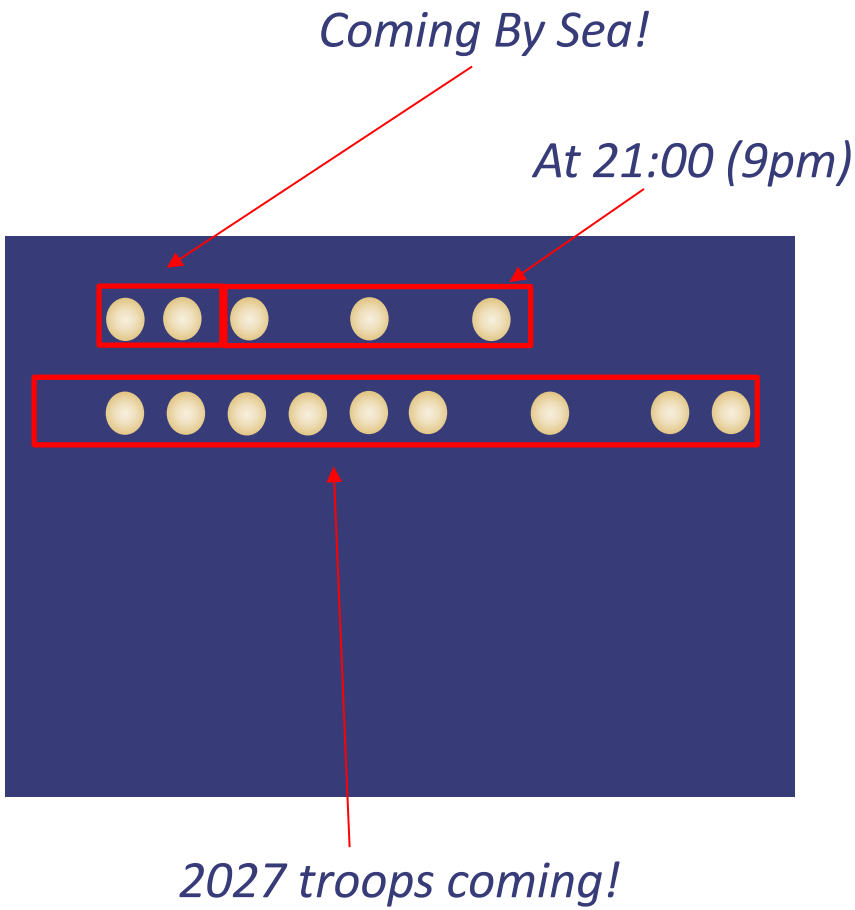
2^{0} (1) fit in 1?

Yes: 0b011111101011
1-1 = 0

Answer:
0b011111101011

The "0b" means we're in base 2

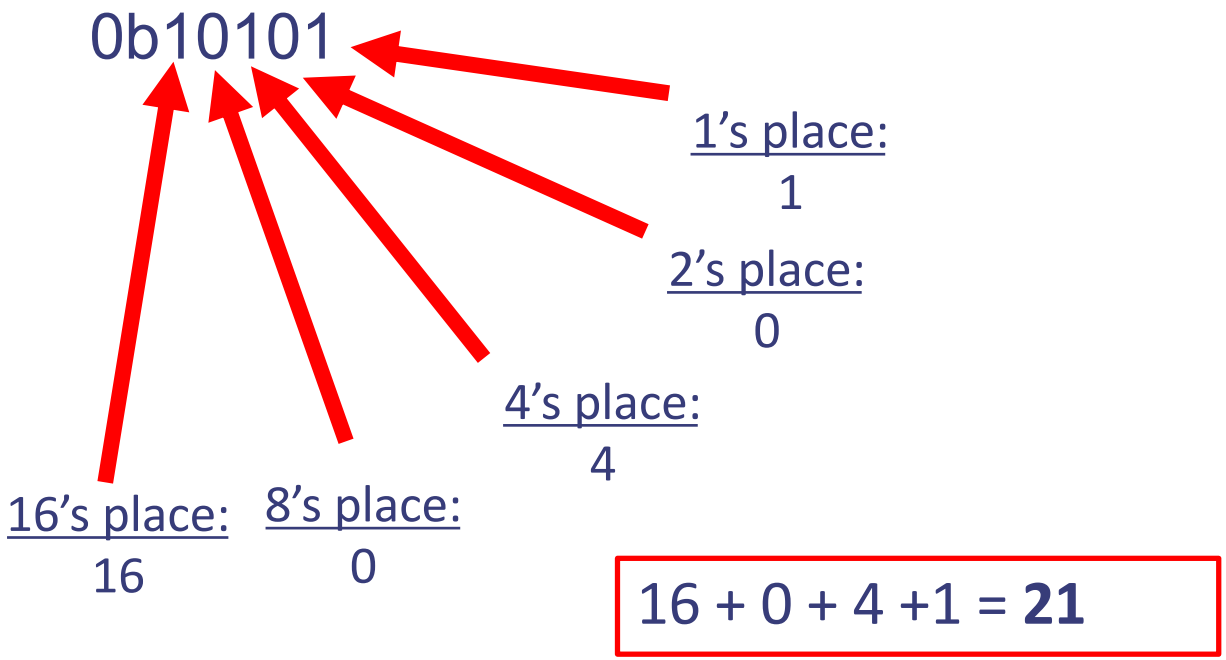
Transmit the Message:



And Paul Revere Would See it and need to decode it...

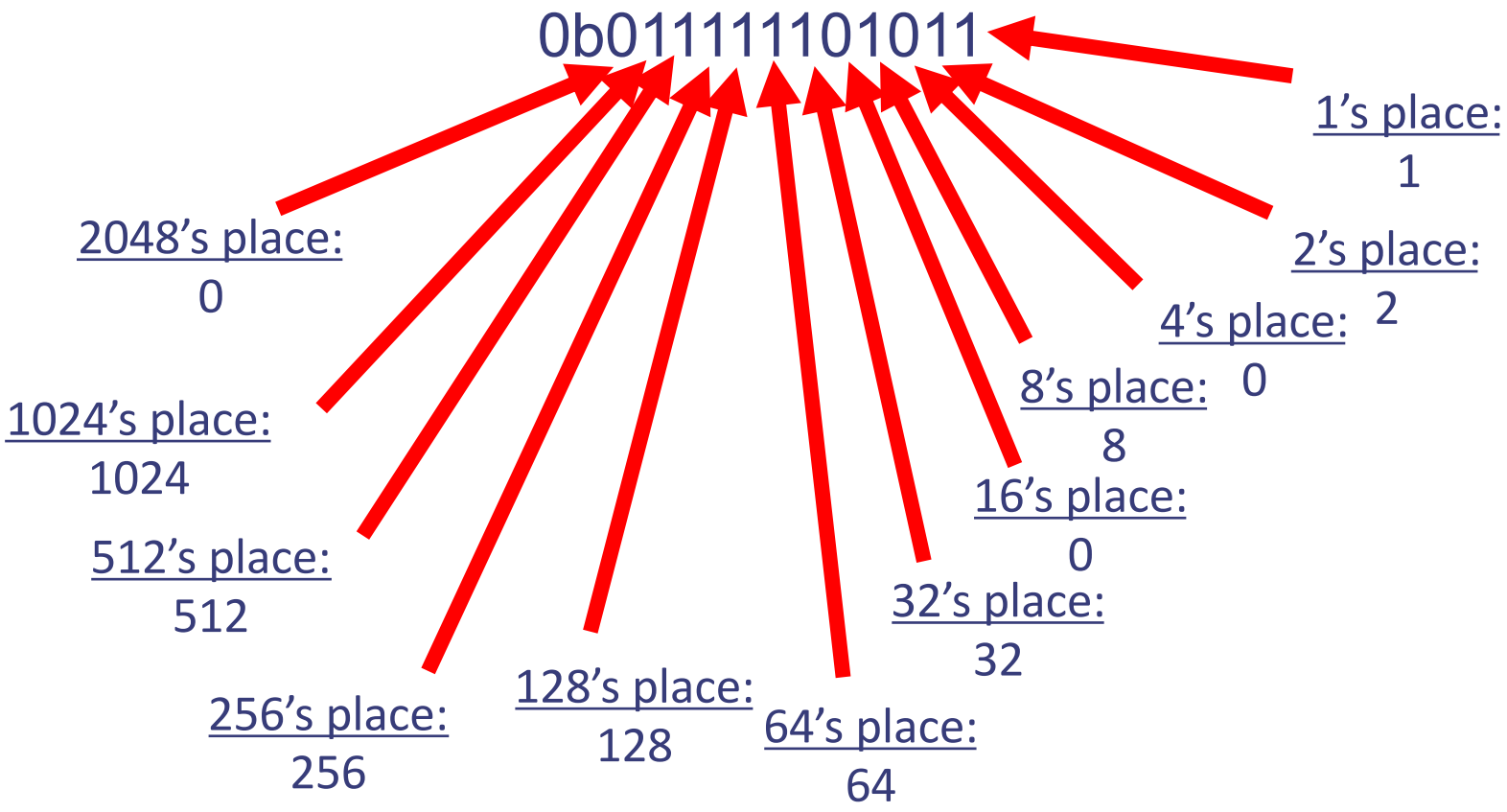
Decoding of Hour

The 0b means binary



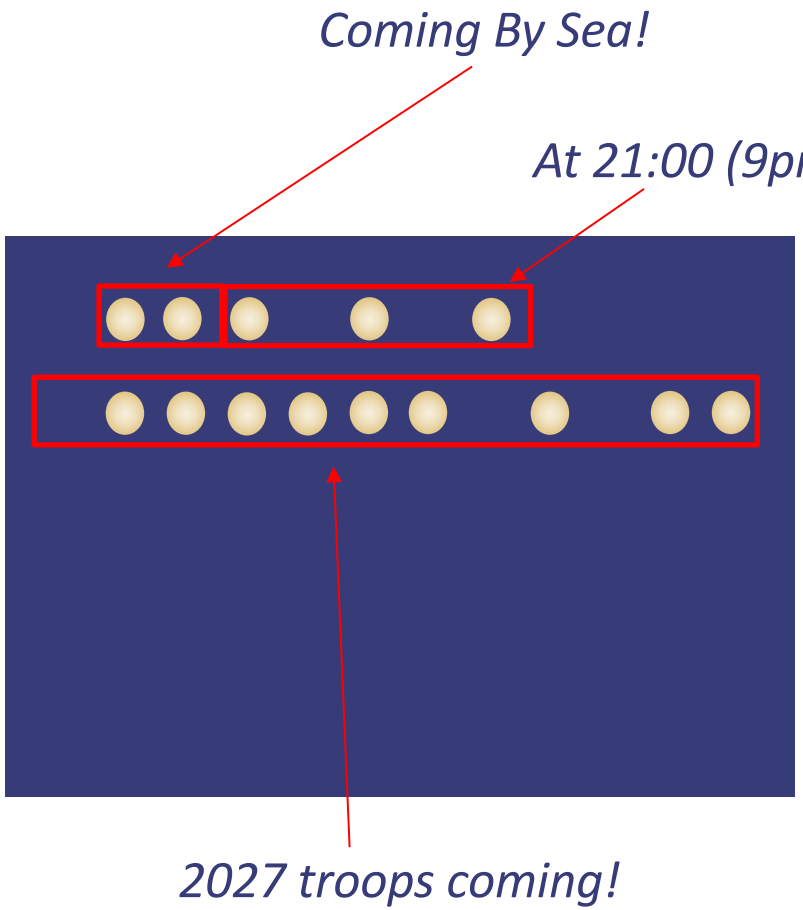
Decoding of Troop Count

The 0b means binary



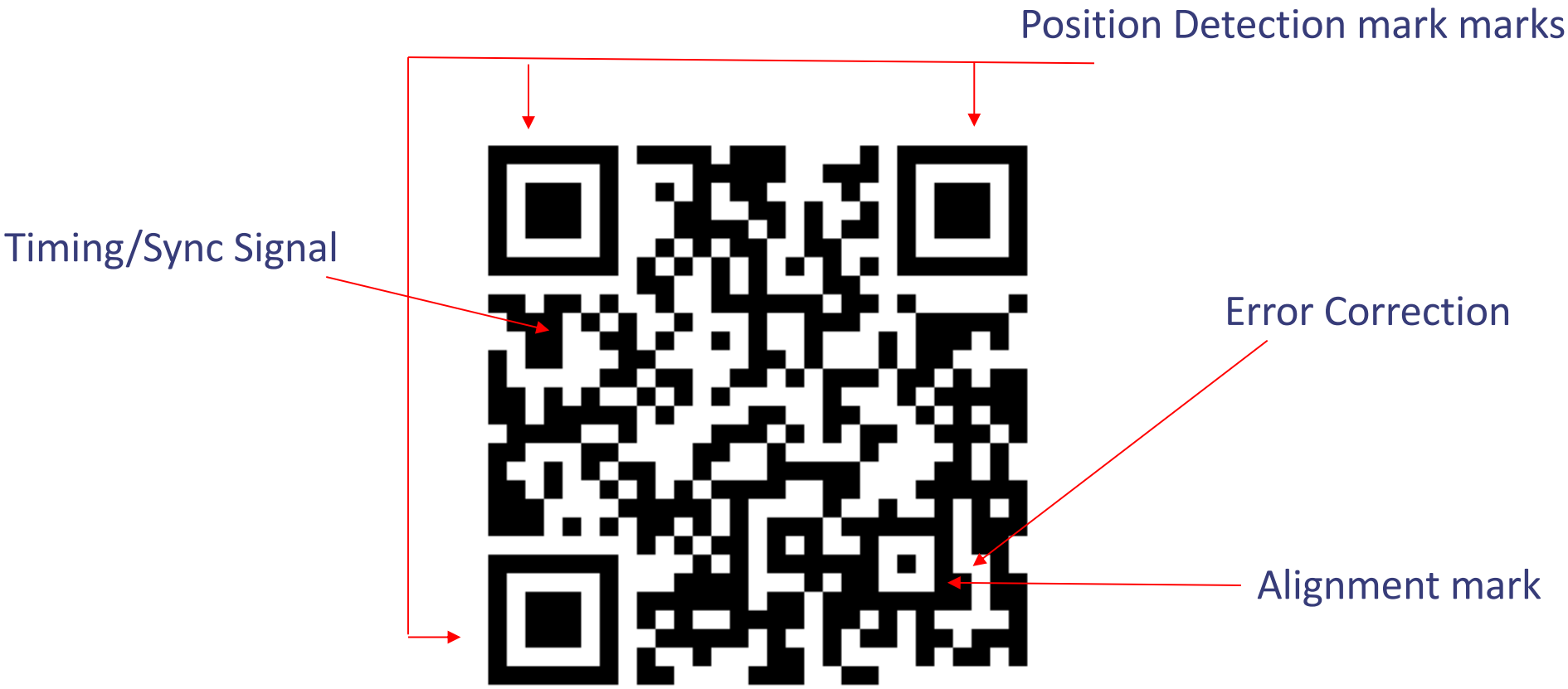
0+1024+512+256+128+64+32+0+128+8+0+2+1 = **2027**

Full Decoding of the Message:



And Paul Revere Would See it and need to decode it...

And Actually Not that Weird of an Example



Black and White are 1's and 0's

Conclusion

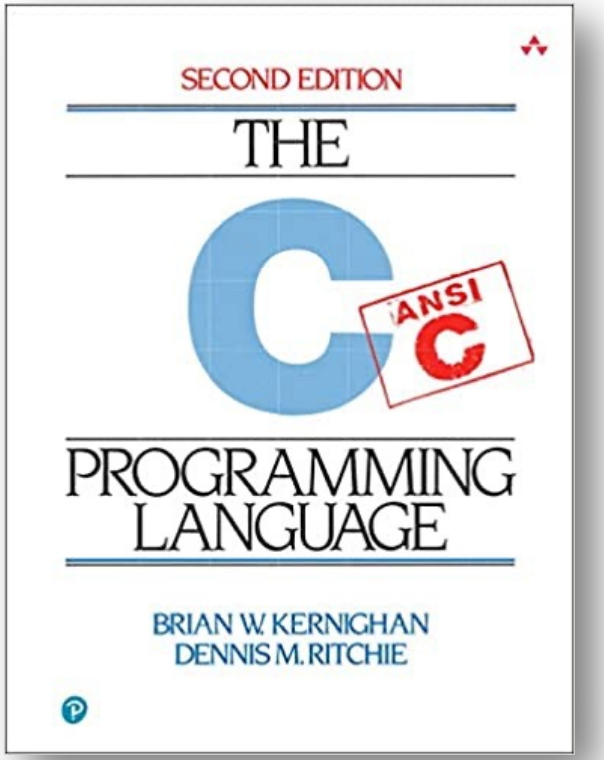
- What we just went through is very similar to how everything is set up in most modern digital computing devices.
- You come up with encoding schemes based on what you need to represent, and then do it using specifically ordered patterns of 1’s and 0’s:
  - All the variables
  - All the steps of the algorithms
  - All the inputs and outputs
  - Everything
  - Everything

Part 2:

The C Programming Language and Memory

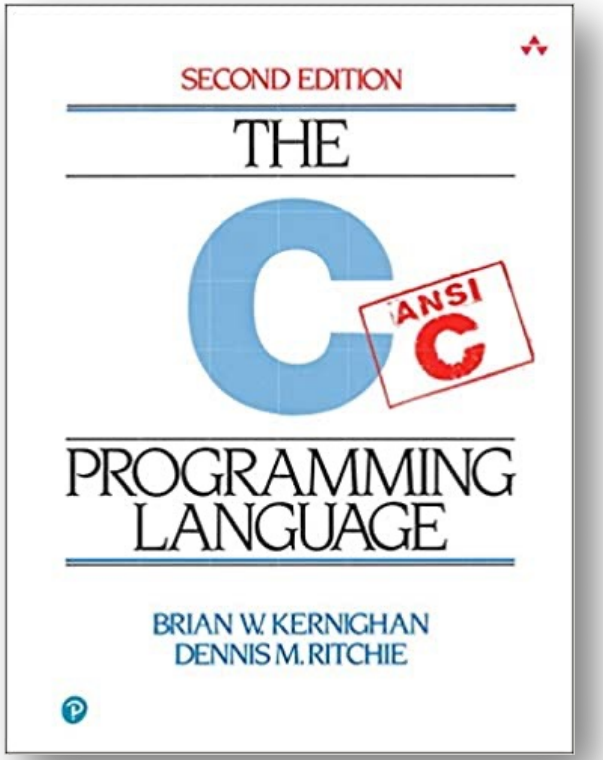
C: The Language

- C has been around for 50 years
- It is an old language and very low-level compared to Python
- It is a very unsafe language compared to Python
- It is also a small language compared to Python



Learning C

- Learn best by doing, but references are always good.
- The internet is always a good reference, and books are good too
  - The K&R C book is excellent and we have linked a copy on the site for your use.



Key Differences between C and Python

Python	C
Interpreted - Slower	Compiled - Faster
Don’t need to declare types	Must declare types
No pointers	Pointers
Automatic memory management	Manual memory management

- C is in general “closer to the metal” meaning there are less abstractions between you and how the data is actually represented in the computer
- When used properly, can result in very efficient programs!

Understanding C Informs

- Python is written in C
- Historically, other languages came in to ”fix” problems with C
- Being conversational in C helps you appreciate EECS as a field and makes you a better engineer even if you don’t use it in your job every day.

I believe in C as I believe that the Sun has risen:
not only because I see it,
but because by it I see everything else.

-C.S. Lewis

So let's write/run some C

- Consider this C code:

```
void app_main(){
  int x;
  x = 11;
  int y = 15;
          int z = x+y;
  printf("the value of x is %d\n", x);
  printf("the value of y is %d\n", y);
  printf("the value of z is %d\n", z);
  // what did this "make" in memory?


}
```

- Prints Out:

```
the value of x is 11
the value of y is 15
the value of z is 26
```

...with comments

- Consider this C code:

```
//comments are with "//"
//This is our "entry point" function called app_main...start here
void app_main(){ //gotta use curly braces (and ; at end of most lines!)
  int x; //declare a variable of type int without defining it
  x = 11; //assign 11 to x
  int y = 15; //declare *and* define a variable of type int called y
          int z = x+y; //indenting doesn't matter
  printf("the value of x is %d\n", x); //print using C-string format
  printf("the value of y is %d\n", y);
  printf("the value of z is %d\n", z);
  // what did this "make" in memory?
  /* multi-line comments possible like this
  words words words blah blah comments */
}
```

- Prints Out:

```
the value of x is 11
the value of y is 15
the value of z is 26
```

What did that code *make*?

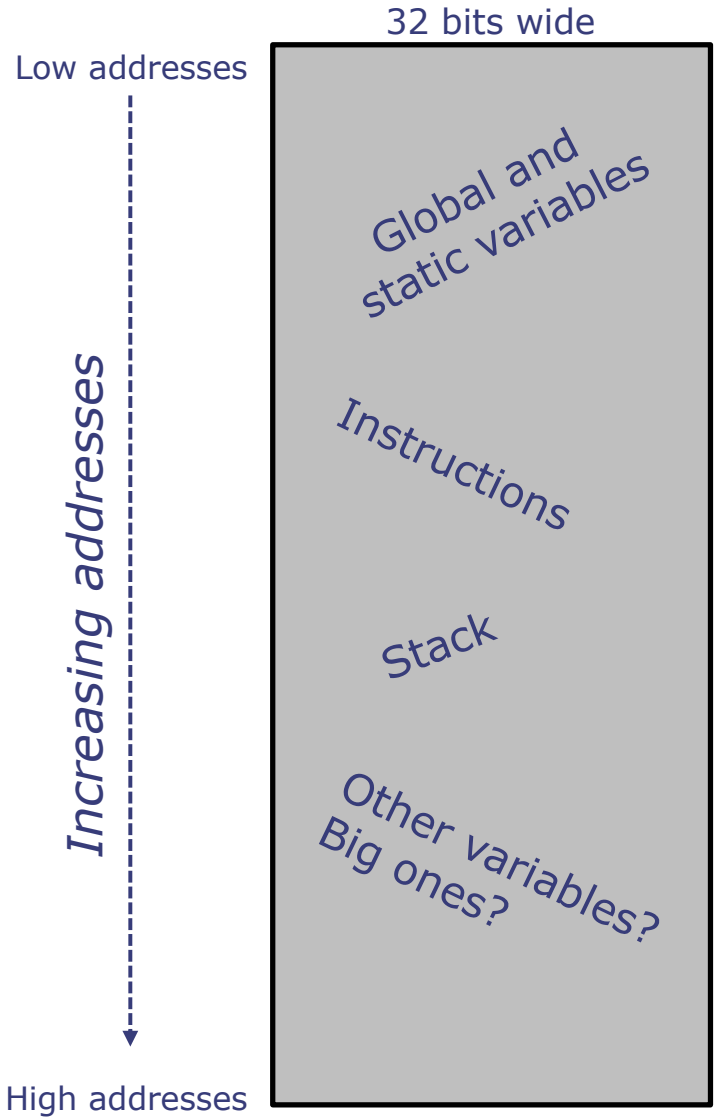
```c
//comments are with "//"
//This is our "entry point" function called app_main...start here
void app_main(){ //gotta use curly braces (and ; at end of most lines!)
  int x; //declare a variable of type int without defining it
  x = 11; //assign 11 to
  int y = 15; //declare and defie a variable of type int
  int z = x+y;
  printf("the value of x is %d\n", x); //print using C-string format
  printf("the value of y is %d\n", y);
  printf("the value of z is %d\n", z);
  // what did this "make" in memory?
}
```

- At the end of our code, our computer memory looked like this:

Address:	Value:
0x3fc93f48:	0x00000000
0x3fc93f4c:	0x420000e4
0x3fc93f50:	0x00000000
0x3fc93f54:	0x0000001a
0x3fc93f58:	0x0000000f
0x3fc93f5c:	0x0000000b
0x3fc93f60:	0x00000000
0x3fc93f64:	0x00000000
0x3fc93f68:	0x00000000
0x3fc93f6c:	0x4201652c

What *is* Memory?

- Memory is a large chunk of electrical storage
- *Everything* is stored digitally (in binary)
- Each part of memory has an address affiliated with it
  - Go from low…
  - To high
  - There are rules about how regions get used (later in class)



If you were to "look" at memory

- You'd just see a massive sea of 1's and 0's going on for ages.
- First you have to remember that these are:
  - NOT *base-10* 1's and 0's
  - **<u>but</u>** *base-2* 1's and 0's

```
00000110100011001110101111000100
01001110111110111100010101110011
00011100100100011100101100100111
01100111111000110101101000001001
01110111010001001001101010001110
11011111001010111001000101111111
01110000110111110000100101010111
00000111011110110101111011111001
10010000000010010111110000001001
11010001111011111111100001001100
11111101011010001001010001011110
00010100111011101111010001001110
01100001001101100000010010000011
11111111100011100110001001001111
10000000011000001000000010101001
00011011100110100011000011010111
```

If you were to "look" at memory

- To avoid confusion, we'll put in that "0b" at the start to indicate binary (base-2)!
- But this is a lot to digest as a human

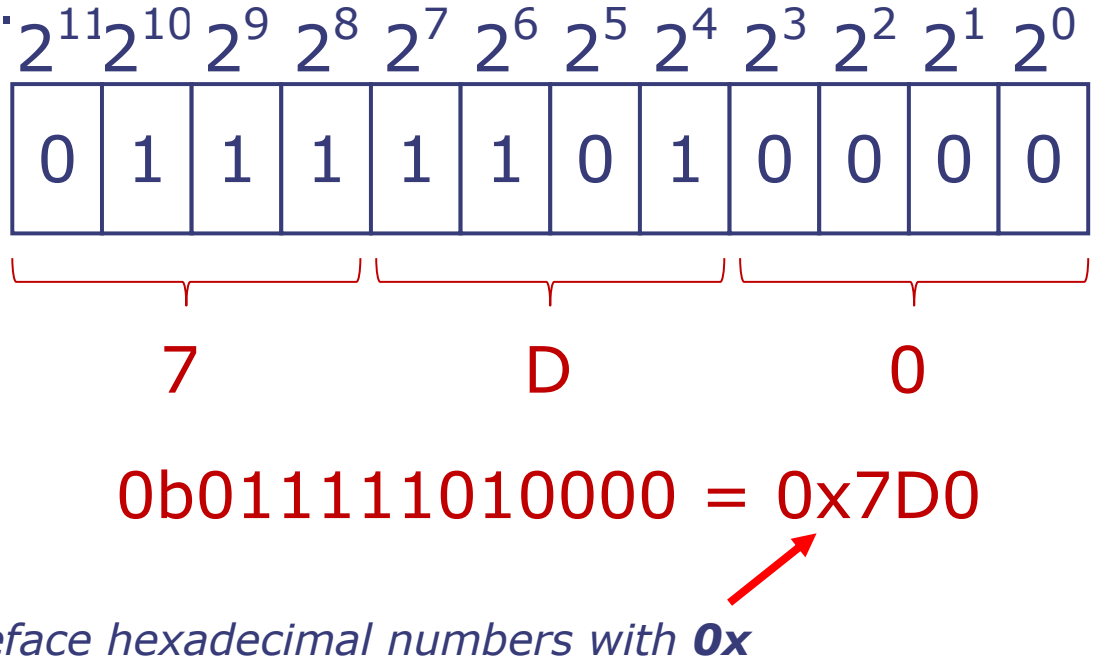
0b00000110100011001110101111000100
0b01001110111110111100010101110011
0b00011100100100011100101100100111
0b01100111111100011010110100001001
0b01110111010001001001101010001110
0b11011111100101011100100010111111
0b01110000110111110000100101010111
0b00000111011110110101111011111001
0b10010000000010010111110000001001
0b11010001111011111111100001001100
0b11111101011010001001010001011110
0b00010100111011101111010001001110
0b01100001001101100000001001000011
0b11111111100011100110001001001111
0b10000000011000001000000010101001
0b00011011100110100011000011010111

Hexadecimal Notation

- Long strings of bits are tedious and error-prone to transcribe, so we often use a higher-radix notation, choosing the radix so that it's simple to recover the original bit string.
- A popular choice is to transcribe numbers in base-16, called hexadecimal. Each group of 4 adjacent bits is represented as a single hexadecimal digit.

Hexadecimal - base 16

0000 - 0	1000 - 8
0001 - 1	1001 - 9
0010 - 2	1010 - A
0011 - 3	1011 - B
0100 - 4	1100 - C
0101 - 5	1101 - D
0110 - 6	1110 - E
0111 - 7	1111 - F



Why Not use Base-10 for this?

- Since each base-16 symbol can represent 16 things, it can cleanly express the information content of four binary digits (or four bits aka "nibbles")…you can easily move back and forth between the two formats with no bleed-over between digits (4-to-1…1-to-4)

- Unfortunately , each base-10 symbol represents $\log_2(10) =$ 3.32 binary digits…this clean representation is not possible...yucko.

So instead of this…

- I am overwhelmed

0b00000110100011001110101111000100
0b01001110111110111100010101110011
0b00011100100100011100101100100111
0b01100111111100011010110100001001
0b01110111010001001001101010001110
0b11011111100101011100100010111111
0b01110000110111110000100101010111
0b00000111011110110101111011111001
0b10010000000010010111110000001001
0b11010001111011111111100001001100
0b11111101011010001001010001011110
0b00010100111011101111010001001110
0b01100001001101100000001001000011
0b11111111100011100110001001001111
0b10000000011000001000000010101001
0b00011011100110100011000011010111

We do this....

- Same great data…
- Same great taste…
- Just more consumable and readable by humans for reading.

- BUT remember this is just a different symbolic representation of the same bits for our benefit!

0x068CEBC4
0x4EFBC573
0x1C91CB27
0x67E35A09
0x77449A8E
0xDF2B917F
0x70DF0957
0x077B5EF9
0x90097C09
0xD1EFF84C
0xFD68945E
0x14EEF44E
0x61360243
0xFF8E624F
0x806080A9
0x1B9A30D7

Now what about organization?

- Giant field of digital information…sure would be nice if we could organize it some way…
- Otherwise it will be difficult to navigate…
- Must have an addressing scheme!

0x068CEBC4
0x4EFBC573
0x1C91CB27
0x67E35A09
0x77449A8E
0xDF2B917F
0x70DF0957
0x077B5EF9
0x90097C09
0xD1EFF84C
0xFD68945E
0x14EEF44E
0x61360243
0xFF8E624F
0x806080A9
0x1B9A30D7

Organization

- The smallest unit accessible in memory of most modern computers is a **byte** which is 8 bits. (or two hexademical symbols)
- On our 6.190 systems we operate in a ***32 bit*** system which means the memory is organized into **four-byte words**

Address:	Value:				
"0" aka 0x00000000	Byte 3	Byte 2	Byte 1	Byte 0	Word @ 0x00000000
"4" aka 0x00000004	Byte 7	Byte 6	Byte 5	Byte 4	Word @ 0x00000004
"8" aka 0x00000008	Byte 11	Byte 10	Byte 9	Byte 8	Word @ 0x00000008
"12" aka 0x0000000C	Byte 15	Byte 14	Byte 13	Byte 12	Word @ 0x0000000C
"16" aka 0x00000010	Byte 19	Byte 18	Byte 17	Byte 16	Word @ 0x00000010
"…"	*ad nauseum…*				

Now what about organization?

- So instead of a giant unanchored set of data…
- We could have…

0x068CEBC4
0x4EFBC573
0x1C91CB27
0x67E35A09
0x77449A8E
0xDF2B917F
0x70DF0957
0x077B5EF9
0x90097C09
0xD1EFF84C
0xFD68945E
0x14EEF44E
0x61360243
0xFF8E624F
0x806080A9
0x1B9A30D7

An Organized Existence…

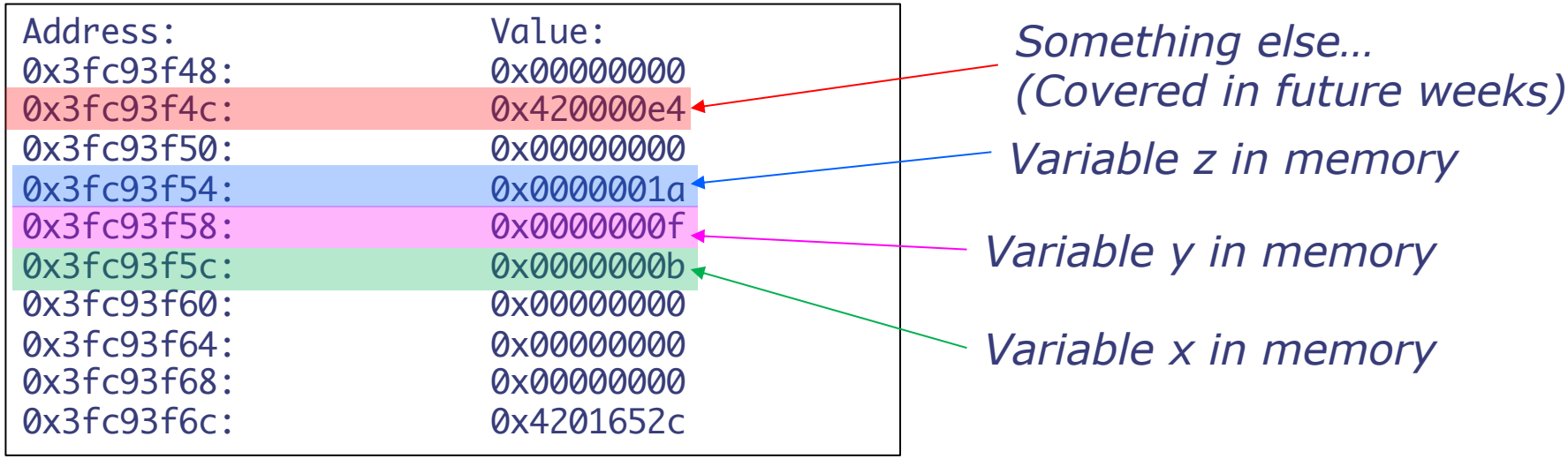
- 4-byte words with addresses.
- Each **word** contains 32 bits of data
- We'll express the addresses in hexadecimal as well

Address:	Value:
0x00000000	0x068CEBC4
0x00000004	0x4EFBC573
0x00000008	0x1C91CB27
0x0000000C	0x67E35A09
0x00000010	0x77449A8E
0x00000014	0xDF2B917F
0x00000018	0x70DF0957
0x0000001C	0x077B5EF9
0x00000020	0x90097C09
0x00000024	0xD1EFF84C
0x00000028	0xFD68945E
0x0000002C	0x14EEF44E
0x00000030	0x61360243
0x00000034	0xFF8E624F
0x00000038	0x806080A9
0x0000003C	0x1B9A30D7

So let us return to the question… What did this code make?

```
//comments are with "//"
//This is our "entry point" function called app_main...start here
void app_main(){ //gotta use curly braces (and ; at end of most lines!)
  int x; //declare a variable of type int without defining it
  x = 11; //assign 11 to
  int y = 15; //declare and defie a variable of type int
  int z = x+y;
  printf("the value of x is %d\n", x); //print using C-string format
  printf("the value of y is %d\n", y);
  printf("the value of z is %d\n", z);
  // what did this "make" in memory?
}
```

- At the end of our code, our memory looked like this:



Wait a sec...what is *Going On* with that Printing?

- #include <stdio.h> - gives you access to printf()

```
int app_main ( ){
  printf ( "%5d\n", 123 ); /* Prints " 123" */
  printf ( "%*d\n", 5, 123 ); /* Prints " 123" */
  printf ( "%+05d\n", 123 ); /* Prints "+0123" */
  printf ( "%x\n", 123); /* Prints "7b" */
  printf ( "%#x\n", 123); /* Prints "0x7b" */
  printf ( "%#X\n", 123); /* Prints "0X7B" */
  printf ( "%-10.2f\n", 12.3 ); /* Prints "12.30" */
  printf ( "%10.2f\n", 12.3); /* Prints " 12.30" */
  printf ( "%lu\n", 123); /* Prints "123" */
  printf ( "%s\n", "Testing"); /* Prints "Testing" */
  printf ( "%c\n", 'A'); /* Prints "A" */
}
```

https://faq.cprogramming.com/cgi-bin/smartfaq.cgi?answer=1048379655&id=1043284385

Reference: Formatted Printing in C

- Specifying width

```
printf("%3d, %6d\n", a, b);
```

a will be given 3 columns of space, and b will be given 6 columns.

- Printing floating point

```
float fahr, celsius;
printf("%3.0f in F is %6.1f in C\n", fahr, celsius);
```

Variable F – printed to take up 3 chars with no digits after decimal

Variable C – printed to take up 6 characters of space, and has one digit after decimal.

https://www.cprogramming.com/tutorial/printf-format-strings.html

And so what does this print?

```
void app_main(){
  int x = 11;
  float y = 15.0;
  char z = 'c';
  char zz = 'b';

  printf("the value of x is %d\n", x);
  printf("the value of y is %f\n", y);
  printf("the value of z is %c\n", z);
  printf("the value of y is %d\n", y);
}
```

- Prints Out:

```
the value of x is 11
the value of y is 15.000000
the value of z is c
the value of y is 1076756480
```

Accidentally interpreted underlying bits of a float as an int (more on this next week)

What About Types?

```
void app_main(){
  int x = 11;
  float y = 15.0;
  char z = 'c';
  char zz = 'b';

  printf("the value of x is %d\n", x);
  printf("the value of y is %f\n", y);
  printf("the value of z is %c\n", z);
  printf("the value of y is %d\n", y);
}
```

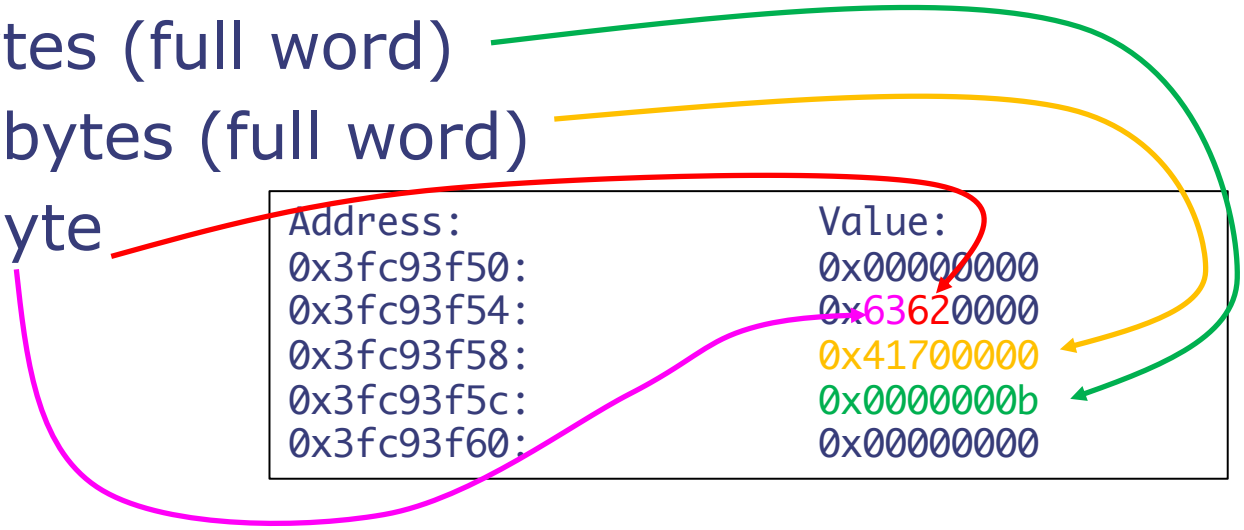
- Prints Out:

```
the value of x is 11
the value of y is 15.000000
the value of z is c
the value of y is 1076756480
```

Accidentally interpreted underlying bits of a float as an int (more on this next week)

Types

- In Python, types and variables are a more fluid concept. In C it is very strict.
- Creation of a variable requires specification of type since that relates to the size and encoding of the bits in the memory
- Cannot change a variable's type!
- `int`: 4 bytes (full word)
- `float`: 4 bytes (full word)
- `char`: 1 byte



More C Types

- #include <stdint.h>
  - Tells our program to include definitions in standard int C library.
- `int8_t`– signed 8-bit integer
- `uint8_t` – unsigned 8-bit integer
- `int32_t` – signed 32-bit integer
- `uint32_t` – unsigned 32-bit integer
- Some operations behave differently if the value is signed or if it is unsigned
  - We will learn about signed binary next week

Few Other C Things

- Conditionals?
  - Yep C has them.

Nothing wild here

Python:

```
if a < b:
   statement(s)
elif a == b:
   statement(s)
else:
   statement(s)
```

C:

```
if (a < b) {
   statement(s);
} else if (a == b) {
   statement(s);
} else {
   statement(s);
}
```

Few Other C Things

- Loops?
  - Yep C has them.

Nothing wild here

Python:

```
while n > 1:
    statement(s)

for i in range(1, 10):
    statements(s)
```

C:

```
while (n > 1) {
    statement(s); }

for ( i = 1; i < 10; i++) {
    statement(s);
}
```

Goto (not in Python)

No Python equivalent

- C allows you to "jump" to labeled locations in your code using the **goto** statement

```
C:

if (n > 1) {
    statement(s);
    goto spot_1;
}
function(5);
printf("Hey");
spot_1:
function(7);
printf("Hi!");
```

If n>1, code will "goto" spot_1 directly

Very close to how assembly language works actually…Exercise in week 1 to get some practice with this…

Few Other C Things

- Logical Operators?
  - Yep C has them.

Nothing wild here

Python:

- and
- or
- not

C:

- &&
- ||
- !

Logical Operators in C

- Comparison operators:
  - >, <, >=, <=, ==, !=
- Return 1 for true; return 0 for false;
  - 2 < 3 returns 1
- Logical operators:
  - && (AND), || (OR), ! (NOT)
  - X && 0 = 0
  - X || 1 = 1
  - !1 = 0, !0 = 1
- Any non-zero value is considered true
  - 3 && 7 = 1, 4 && 0 = 0
  - 3 || 7 = 1, 4 || 0 = 1, 0 || 0 = 0
  - !9 = 0, !0 = 1

Nothing wild here

Bitwise Operators

- Just like from Python there are bitwise operators

```
x << y; //shift x left by y bits
x >> y; //shift x right by y bits
x & y; //bitwise and x with y
x | y; //bitwise or x with y
~x;  //bitwise invert x
x ^ y; //bitwise xor x with y
```

These are all in Python too…you just probably didn't use them much

- You tend to use them a lot more in C since you're working with data at the bit level

Shift operations

- Suppose: x = 0b000101; shift left by 3 **(x<<3)**

000101	5
001010	10
010100	20
101000	40

  - Shift left by i is same as multiply by 2^i as long as don't fall off edge

- Suppose: x = 0b01100; shift right by 3. **(x>>3)**

01100	12
00110	6
00011	3
00001	1

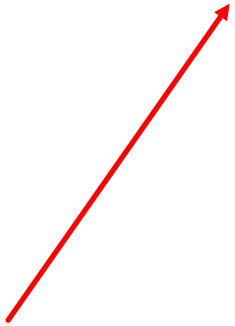
  - Shift right by i = divide by 2^i

Digital Manipulations

- Since our data is in Base-2, Boolean operations can be readily deployed on it!
- Given a single bit, you can do two things to it:

In	Out
0	0
1	1

In	NOT (~)
0	1
1	0



Don't really need an operator just to "be"

Bitwise NOT (~)

different than logical NOT (!)

Digital Manipulations (&, |, ^)

- Use 2 input bits to produce 1 output bit
- Three big functions: bitwise AND, OR, and XOR

In1	In2	AND (&)
0	0	0
0	1	0
1	0	0
1	1	1

In1	In2	XOR (^)
0	0	0
0	1	1
1	0	1
1	1	0

In1	In2	OR (|)
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise Binary Operations

- Bitwise binary operators apply these operations to individual bits of two values...
- Does so ”by column”
- Suppose: `x = 0b00101, y = 0b00110`

bitwise NOT x `~x`	**x bitwise AND y** `x & y`	**x bitwise OR y** `x | y`	**x bitwise XOR y** `x ^ y`
00101	00101	00101	00101
	00110	00110	00110
11010	00100	00111	00011

What About?

Nothing wild here
Just remember types

- Functions?
  - Yep C has them.

Python:

```
def summ(a, b):
   return a + b

x = 1
y = 2
z = summ(x, y)
```

C:

```
int summ(int a, int b){
   return a + b;
}
int x, y, z;
x = 1;
y = 2;
z = summ(x, y);
```

And then... there's the Pointer

- The pointer is a data type in C that "points" to other data.
- It does so by storing a memory address (called a "**reference**")
- We can then access what it points to it as needed ("**dereference**")
- Pointers allow direct access and manipulation of memory.

Why have a Pointer?

- Variables like ints, floats, and chars represent "external" concepts and information:
  - Money in a bank account
  - Temperature outside
  - Your letter grade in a class

- These variables live in memory, but sometimes it'd be nice to let other things know where those variables are in memory.

- *Pointers are variables* that allow the computer to represent locations its own memory

& Operator (“address-of operator”)

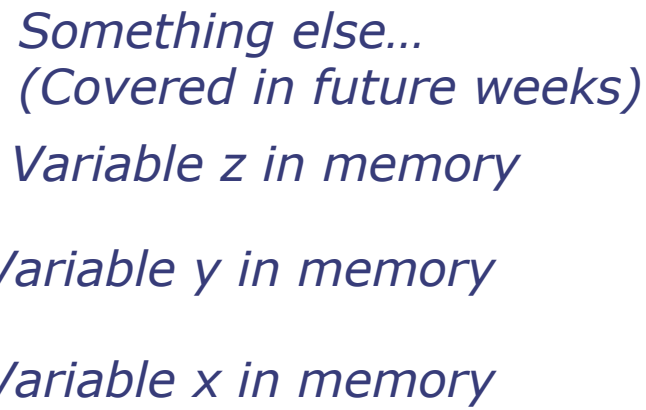
- To get access to a variable’s spot in memory, you use the & operator.

- This will return *where in memory* a variable resides.

Return to earlier code...

```
void app_main(){ //gotta use curly braces (and ; at end of most lines!)
  int x; //declare a variable of type int without defining it
  x = 11; //assign 11 to
  int y = 15; //declare and defie a variable of type int
  int z = x+y;
  //...
}
```

Address:	Value:
0x3fc93f48:	0x00000000
0x3fc93f4c:	0x420000e4
0x3fc93f50:	0x00000000
0x3fc93f54:	0x0000001a
0x3fc93f58:	0x0000000f
0x3fc93f5c:	0x0000000b
0x3fc93f60:	0x00000000
0x3fc93f64:	0x00000000
0x3fc93f68:	0x00000000
0x3fc93f6c:	0x4201652c

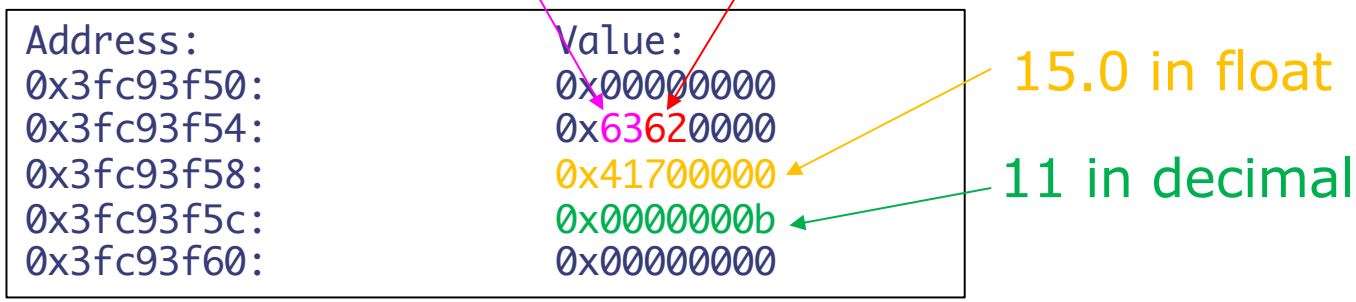


- What is &x? What is &y? What is &z?

&x: 0x3fc93f5c &y: 0x3fc93f58 &z: 0x3fc93f54

Slightly trickier... (code form earlier)

```
void app_main(){
  int x = 11;
  float y = 15.0;
  char z = 'c';
  char zz = 'b';
}
```



- What is &x? What is &y? What is &z? What is &zz?

&x: 0x3fc93f5c &y: 0x3fc93f58

Byte Organization

- The smallest unit accessible in memory of most modern computers is a **byte** which is 8 bits. (or two hexademical symbols)
- On our 6.190 systems we operate in a ***32 bit*** system which means the memory is organized into **four-byte words**

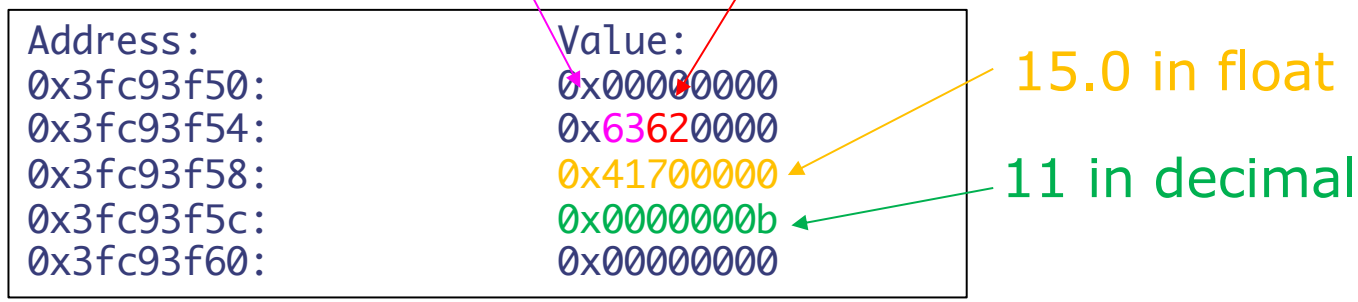
Address: Value:

"0" aka 0x00000000	Byte 3	Byte 2	Byte 1	Byte 0	Word @ 0x00000000
"4" aka 0x00000004	Byte 7	Byte 6	Byte 5	Byte 4	Word @ 0x00000004
"8" aka 0x00000008	Byte 11	Byte 10	Byte 9	Byte 8	Word @ 0x00000008
"12" aka 0x0000000C	Byte 15	Byte 14	Byte 13	Byte 12	Word @ 0x0000000C
"16" aka 0x00000010	Byte 19	Byte 18	Byte 17	Byte 16	Word @ 0x00000010

"…" *ad nauseum…*

Slightly trickier... (code form earlier)

```
void app_main(){
  int x = 11;
  float y = 15.0;
  char z = 'c';
  char zz = 'b';
}
```



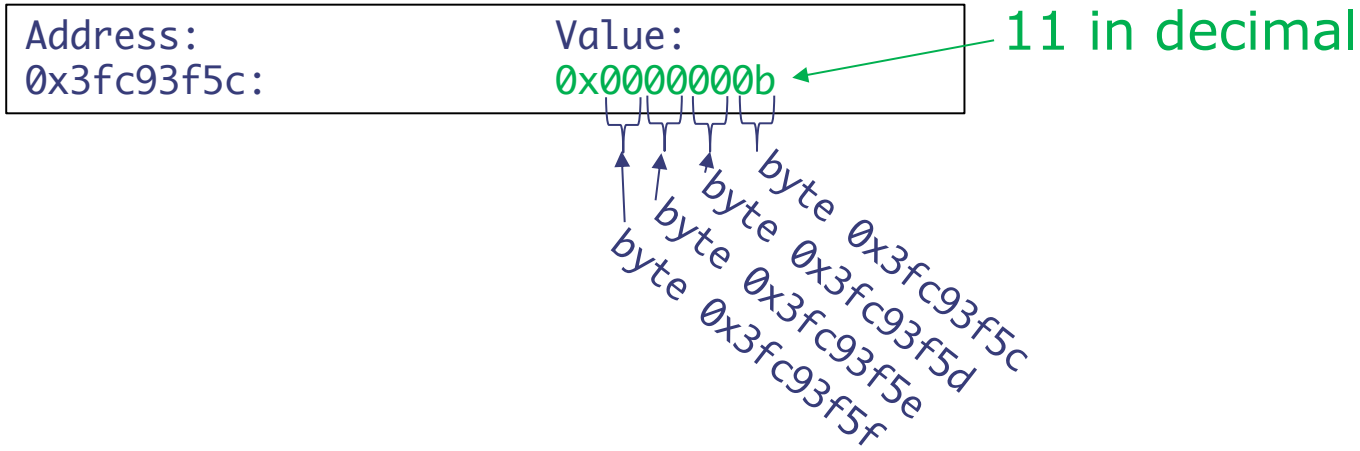
- What is &x? What is &y? What is &z? What is &zz?

&x: 0x3fc93f5c &y: 0x3fc93f58

&z: 0x3fc93f57 &z: 0x3fc93f56

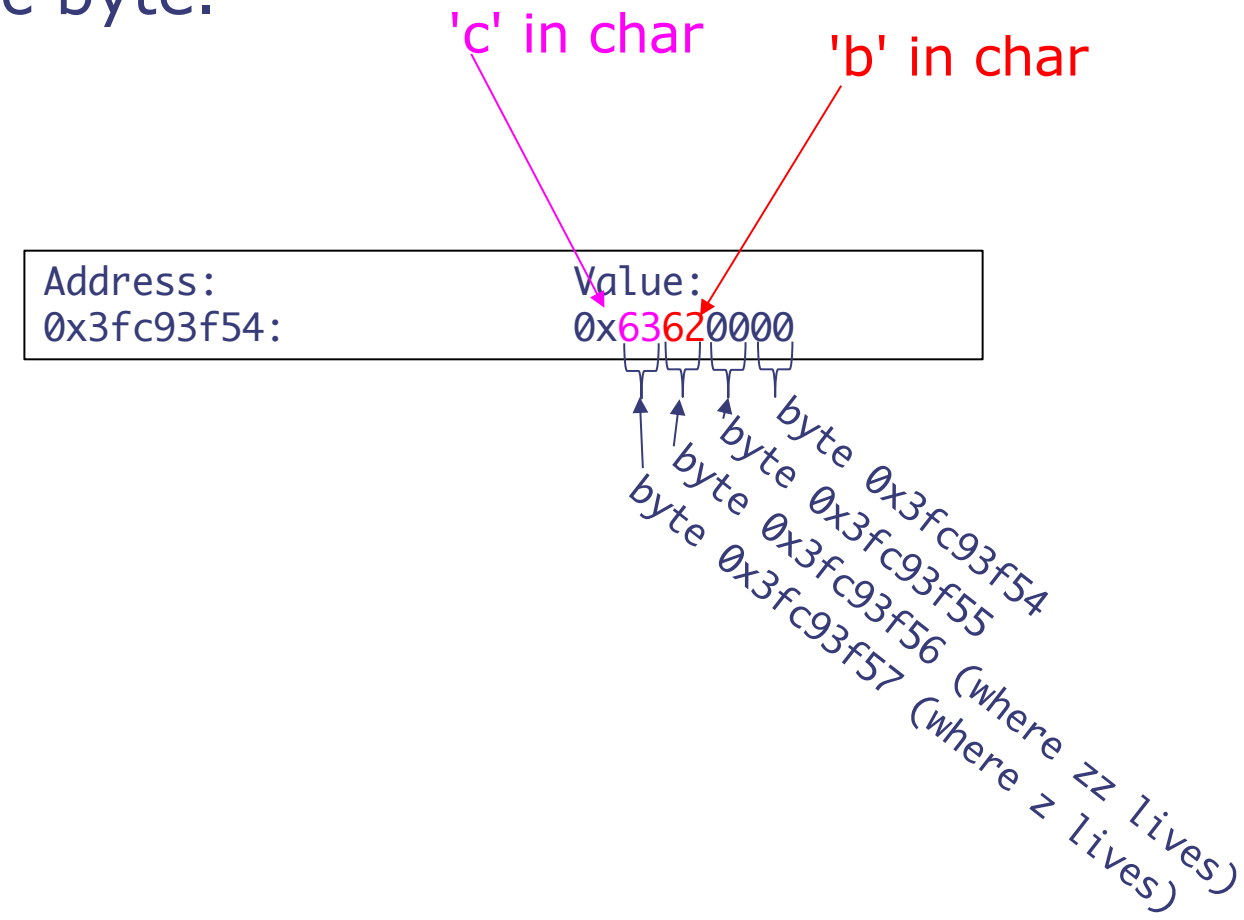
Address

- In multi-byte data types (int, float), the address of the variable is its *lowest byte*



Address

- In single byte data types (char), the address is its specific byte.



Why have a Pointer?

- Variables like ints, floats, and chars represent "external" concepts and information:
  - Money in a bank account
  - Temperature outside
  - Your letter grade in a class

- These variables live in memory, but sometimes it'd be nice to let other things know where those variables are in memory.

- *Pointers are variables* that allow the computer to represent locations its own memory

How to make a pointer

- To make a pointer, decide what type of thing in memory you want to point to (an integer? A float? A char?)
- Then add a * before deciding a name for your pointer

```
void app_main(){
  int x = 5; //make int called x
  int * ptr_to_x; //make "int pointer" called ptr_to_x
  ...
```

- Then to point that pointer to where x is in memory, you access x's memory address by using the & operator and assign it to ptr_to_x

```
ptr_to_x = &x; //give ptr_to_x the address of x
```

What's in the memory?...

```
void app_main(){
  int x = 5; //make int called x
  int * ptr_to_x; //make "int pointer" called ptr_to_x
  ptr_to_x = &x; //give ptr_to_x the address of x
}
```

Address:	Value:	
0x3fc93f58:	0x42016554	
0x3fc93f5c:	0x00000005	*x*
0x3fc93f60:	0x3fc93f5c	*ptr_to_x*
0x3fc93f64:	0x00000000	
0x3fc93f68:	0x00000000	
0x3fc93f6c:	0x42016554	

The pointer ptr_to_x lives in memory just like any other variable. Its value, is another variables memory location!

Dereferencing (* operator)

- This is the other side of pointers.
- It is how you "read" the memory location that is pointed to.

```
void app_main(){
  int x = 5; //make int called x
  int * ptr_to_x; //make "int pointer" called ptr_to_x
  ptr_to_x = &x; //give ptr_to_x the address of x
}
```

- To access and read what ptr_to_x points to therefore we need to do the following:

```
*ptr_to_x
```

Example

```
void app_main(){
  int x = 5; //make int called x
  int * ptr_to_x; //make int pointer called ptr_to_x
  ptr_to_x = &x; //give ptr_to_x the address of x
  printf("x is %d\n" , x);
  printf("ptr_to_x's value is %p\n" ,ptr_to_x); //%p prints value as address
  printf("The value ptr_to_x points at is %d\n", *ptr_to_x);
}
```

- This Prints:

```
x is 5
ptr_to_x's value is 0x3fc93f5c
The value ptr_to_x points at is 5
```

The value of "x"

The value of "ptr_to_x" Which is the address of x

The value contained in the memory spot that ptr_x points to (which should be x)

Syntactical Ambiguity of ∗ and &

- In C, ∗ means two-ish things depending on context:
  - ∗ can mean "multiply"
    - `int q = a * b; //no surprises`
  - ∗ can be used in conjunction with type to declare a pointer
    - `int * z; //declare int pointer named z`
  - ∗ can mean "**dereference**" which means get the value at the address the pointer points to
    - `*z = a; //set the value z points to to be int a`
    - `b = *z; //set int b to be the value z points to`
- In C, & means two things depending on context:
  - & can mean "and" (either logical or bitwise)
    - `int q = a&b; //q is bitwise and of a and b`
  - & can mean "reference to" or "address of"
    - `int * z = &a; //set pointer z's value to a's address`

Seems Confusing

- Yes. Yes. It can be. When all else fails, utilize parentheses to remove ambiguity with order of operations.

- As operators, * and & are opposites and can cancel out:

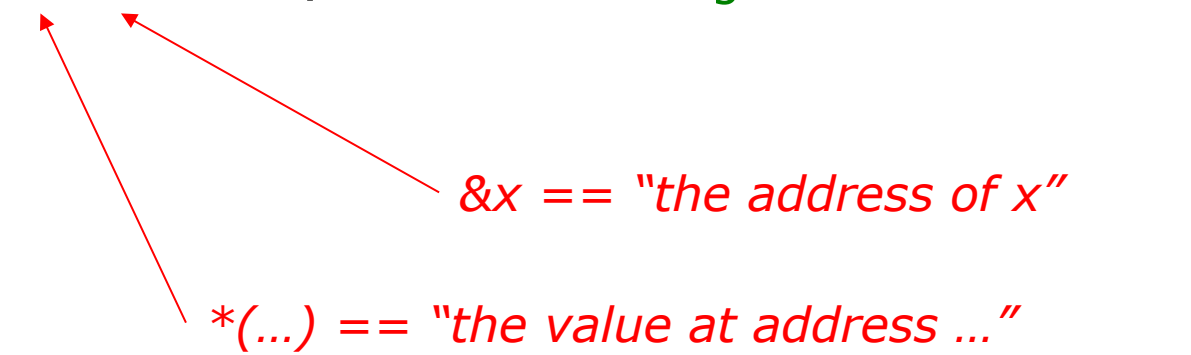
```
int x = 5; //create x and set to 5
x = 6; //set x to 6
*(&x) = 6; //same thing as line above
```

Important

```
int x = 5; //create x and set to 5
x = 6; //set x to 6
*(&x) = 6; //same thing as line above
```

- The last line is saying, "The value located at the address of x is 6"

```
*(&x) = 6; //same thing as line above
```



Declaring/Working with a Pointer

```
int * z; //declare an int pointer called z
```

```
          // stated another way...
int * z; // *z is type int. Therefore...
          // z must point to ints
```

```
int y = 6; //declare an int y
int q = 7; //declare an int q

//create, assign int pointer z to address of y:
int * z = &y;
// *z has value of 6 at this point.
//reassign z to address of q:
z = &q;
// *z now has value of 7 at this point
```

The Point of a Pointer

- The value of a pointer is a **reference** which is another way of saying **an address in memory**.

- Pointers have types in C because
  - Different variable types can have different sizes
  - The underlying binary is interpreted differently based on type
  - When you **dereference** it needs to know how to interpret binary (as an int? as a float? as a char?)
  - When you move pointer around ("pointer arithmetic", it will move the byte it points to based on size of type it points to)
    - +1 on an int pointer moves it by four bytes in memory
    - +1 on a char pointer moves it by one byte in memory

The Purpose/Utility of Pointers*

1. To point to a variable
2. To point to a group/region of data/variables
3. To refer to memory regions that aren't associated with variables

*there's more than three, but these are what we'll focus on here

1. Pointing to Variables

- Can allow you to modify variables *in place*. For example, this function has no return type, but does have an "output" in the form of the pointer z

```
void add_and_assign(int x, int y, int * z){
  //the value of z is a reference (address)
  //the value at that address (*z) is now to be x+y:
  *z = x+y;
}
```

```
void app_main(){
  int a = 1;
  int b = 2;
  int c = 0; //starts at 0
  add_and_assign(a,b,&c);//give int a, b, and ref to c
  //this function call will change variable c!
  printf("Value of c: %d\n",c); //will print 3
}
```

The Purpose/Utility of Pointers*

1. To point to a variable
2. To point to a group/region of data/variables
3. To refer to memory regions that aren't associated with variables

*there's more than three, but these are what we'll focus on here

2. Pointing to Groups of Data

- Large groups of data can be slow to copy around. If we can instead just hand around a pointer to the data group, working with the data can be much, much faster!

2. Larger Data Types in C

- In C there are really three fundamental types:
  - Integers
  - Characters
  - Floating Point Numbers
- Everything else are derived types comprised of variants or groups of the fundamental types above. This includes:
  - Arrays (lists)
  - Structs (sort of like classes)
- These larger data types are best handled via pointers.
- Let's look at **arrays**:

C Arrays

- The size of the array *must* be known at creation
- The *type* of all elements of the array must be the same
- The array's "variable" is approximately* the same as a pointer to the start of the array in memory.
  - Its value is the starting address of the array (index 0)

*more like an unchangeable pointer

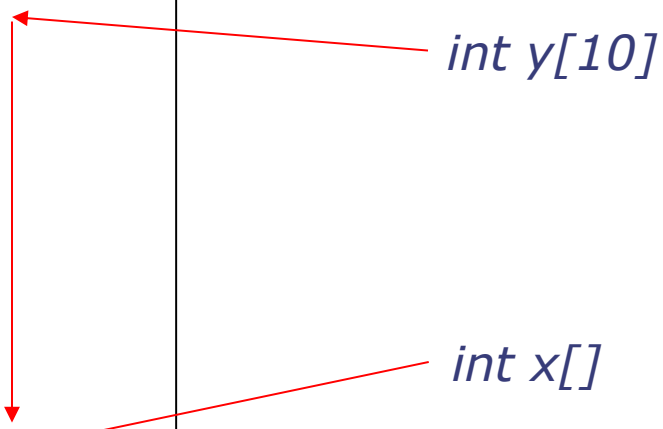
Consider this Code

```
void app_main(){
  int x[5] = {1,2,3,4,5};

  //10 long array of ints
  //only five vals specified
  int y[10] = {10,11,12,13};
}
```

Memory after Code. Can you figure what corresponds to what?

Address:	Value:
0x3fc93f0c:	0x420001ec
0x3fc93f10:	0x0000000a
0x3fc93f14:	0x0000000b
0x3fc93f18:	0x0000000c
0x3fc93f1c:	0x0000000d
0x3fc93f20:	0x00000000
0x3fc93f24:	0x00000000
0x3fc93f28:	0x00000000
0x3fc93f2c:	0x00000000
0x3fc93f30:	0x00000000
0x3fc93f34:	0x00000000
0x3fc93f38:	0x00000001
0x3fc93f3c:	0x00000002
0x3fc93f40:	0x00000003
0x3fc93f44:	0x00000004
0x3fc93f48:	0x00000005



A C array

```
void app_main(){
  int x[5] = {1,2,3,4,5};

  //10 long array of ints
  //only five vals specified
  int y[10] = {10,11,12,13};
}
```

- An array in C is literally just the values of the array in order in memory.

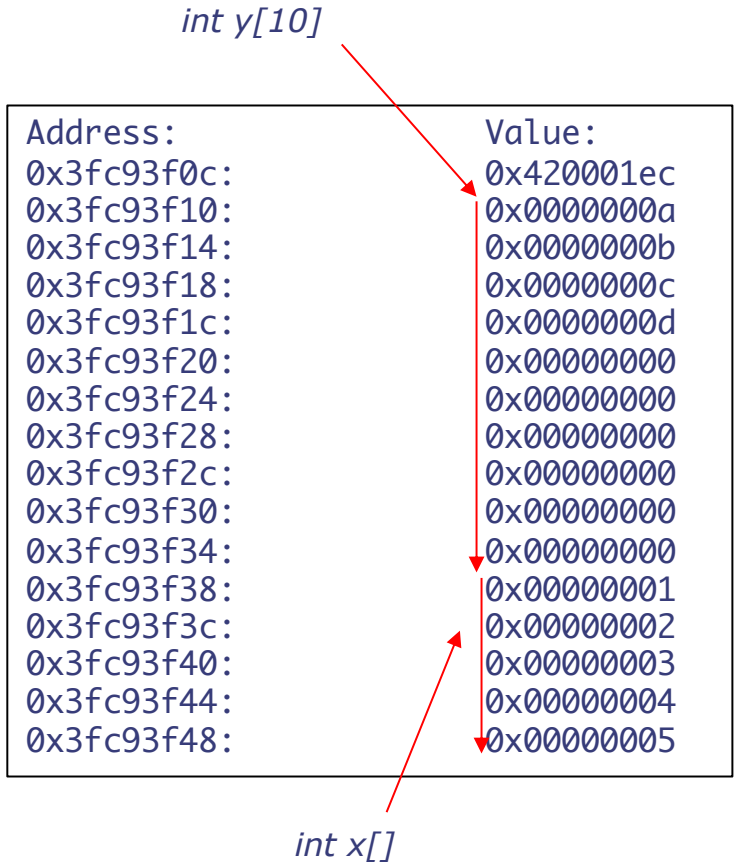
- The ”variable” that represents the array is basically* just a pointer to the starting address of that array.
  - So x[1] is an integer and x is a memory address

*not exactly 100% true, but ignore for now

Consider this Code

```
void app_main(){
  int x[5] = {1,2,3,4,5};
  //10 long array of ints
  //only five vals specified (rest are 0 init)
  int y[10] = {10,11,12,13};
  printf("%d\n", x[1]); //prints what?
  printf("%d\n", *x); //prints what?
  printf("%d\n", x); //prints what?
  printf("%d\n", x+1); //prints what?
  printf("%d\n", *(x+2)); //prints what?
  printf("%d\n", *(x+5)); //prints what?
  printf("%d\n", *(y+10)); //prints what?


}
```

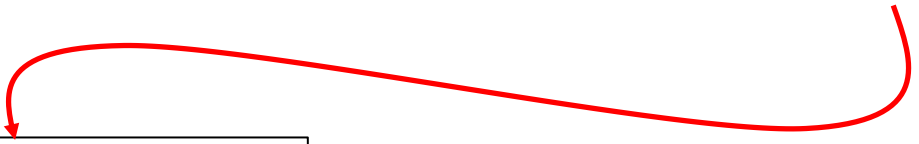


Consider this Code *(answers)*

```
void app_main(){
  int x[5] = {1,2,3,4,5};
  //10 long array of ints accessible through ptr y
  //only five vals specified (rest are unknown)
  int y[10] = {10,11,12,13};
  printf("%d\n", x[1]); //prints 2
  printf("%d\n", *x); //prints 1 same as x[0] or *(x+0)
  printf("%d\n", x); //prints 1070153528 (0x3fc93f38 in hex)
  printf("%d\n", x+1); //prints 1070153532 (0x3fc93f3c in hex)
  printf("%d\n", *(x+2)); //prints 3...same as x[2]
  printf("%d\n", *(x+5)); //prints undefined...same as x[5]…no idea what's in that memory
  printf("%d\n", *(y+10)); //prints 1...because you bled into where x is and x[0] is 1
  //wait does that mean I could do this???
  y[12]=15; //oh god please throw an error
  printf("%d\n", x[2]); //prints 15...oh no C has no boundary checking
}
```

An Array is Just an Array

- An array is an ordered list of contents in memory
- That's it. There is no "len" attribute that is callable
- So when an array is handed into a function, the function ***only sees a pointer***.
- We need to tell the function how long the array is!



```
int sum(int *a, int len) {
  int s = 0;
  for (int i=0; i<len; i++){
    s += a[i];
  }
  return s;
}
```

Python Lists versus C Arrays

Approximately same code in two languages:

```python
#in python:
a = [3, 9, 6, 1, 7, 2]
x = a[1]

def sum(a):
  s = 0
  for i in range(len(a)):
    s += a[i]
  return s

sum(a)
```

```c
// in C:
int a[6] = {3, 9, 6, 1, 7, 2};
int x;
x = a[1];

int sum(int *a, int len) {
  int s = 0;
  for (int i=0; i<len; i++){
    s += a[i];
  }
  return s;
}
sum(a, 6);
```

Some array and pointer fun

```
void app_main(){
  int a = 13; //int a = 13
  int b = 10; //int b = 10
  int* c = &a; //int ptr c takes on value of a's addr
  int *d; //int ptr d
  int * e; //int ptr e
  d = &b; //d takes on addr of b
  e = d; //e takes d's value (which is b's addr)
  //5 long array of ints accessible through ptr x
  int x[] = {1,2,3,4,5};
  //10 long array of ints accessible through ptr y
  //only five vals specified (rest are unknown)
  int y[10] = {10,11,12,13};
  *e = 2 * *e + *c;
}
```

Memory after that code. Can you figure what corresponds to what?

Address:	Value:
0x3fc93f0c:	0x420001ec
0x3fc93f10:	0x0000000a
0x3fc93f14:	0x0000000b
0x3fc93f18:	0x0000000c
0x3fc93f1c:	0x0000000d
0x3fc93f20:	0x00000000
0x3fc93f24:	0x00000000
0x3fc93f28:	0x00000000
0x3fc93f2c:	0x00000000
0x3fc93f30:	0x00000000
0x3fc93f34:	0x00000000
0x3fc93f38:	0x00000001
0x3fc93f3c:	0x00000002
0x3fc93f40:	0x00000003
0x3fc93f44:	0x00000004
0x3fc93f48:	0x00000005
0x3fc93f4c:	0x3fc93f58
0x3fc93f50:	0x3fc93f58
0x3fc93f54:	0x3fc93f5c
0x3fc93f58:	0x00000021
0x3fc93f5c:	0x0000000d
0x3fc93f60:	0x00000000

int y[10]

int x[]

*int * e*

*int * d*

*int * c*

int b

int a

Ask on piazza for clarification!!

The Purpose/Utility of Pointers*

1. To point to a variable

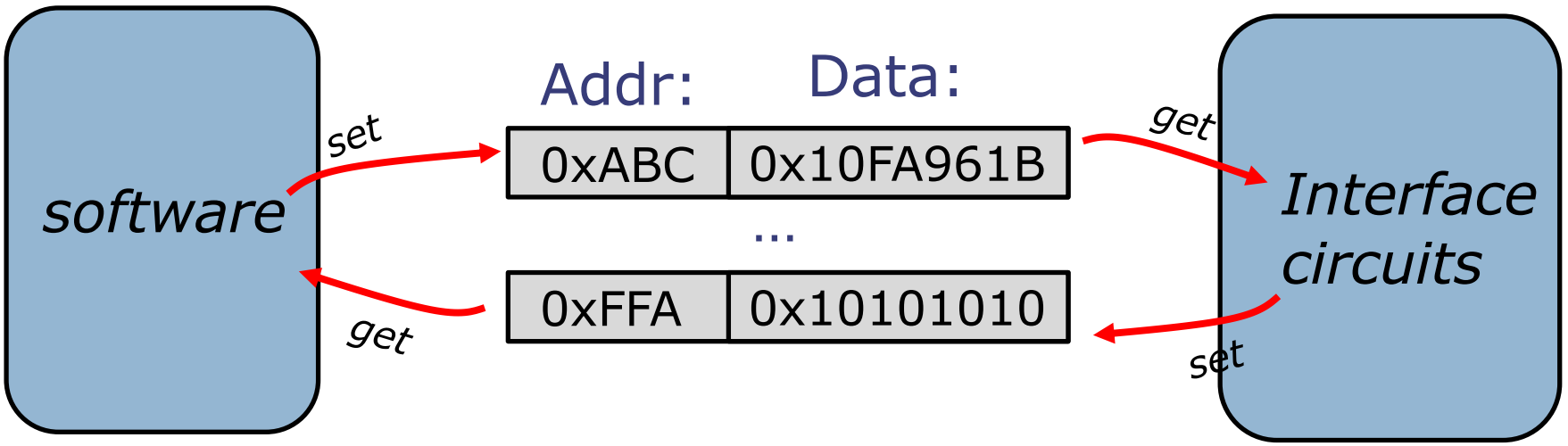
2. To point to a group/region of data/variables

3. To refer to memory regions that aren't associated with variables (very relevant for labs)

*there's more than three, but these are what we'll focus on here

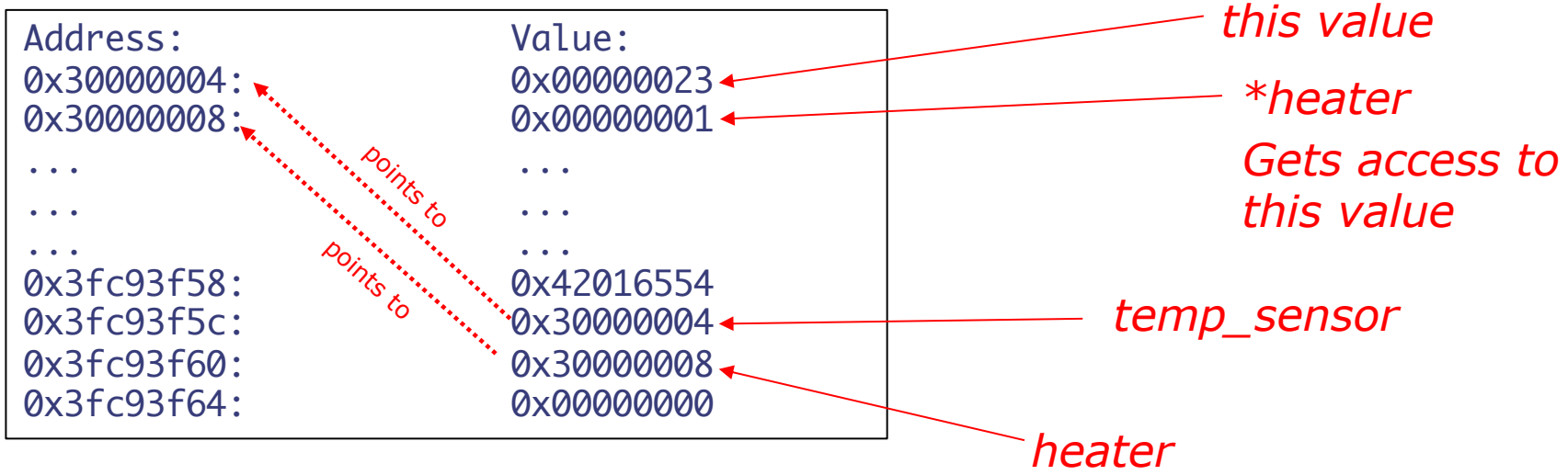
3. Pointing to Addresses that Aren't Variables or Groups of Variables

- Computers will have certain addresses in memory that are interfaces of the computer to the outside world
- Call this **Memory-Mapped-Input-Output (MMIO)**
- Certain addresses act like little mailboxes to set or get values from software to hardware/vice versa



MMIO Example… (Lab 1 related)

```
void app_main(){
  int * temp_sensor = 0x30000004; //set pointer to a known address value
  int * heater = 0x30000008; //set pointer to a known address value
  //The two addresses above come from datasheet of processor!
  while(1){//run forever
    //check temperature...
    if (*temp_sensor <60){ //get value...less than 60?
      *heater = 1; //set value to 1 (let it warm)
    }else{
      *heater = 0; //set value to 0 (let it cool)
    }
  }
}
```



MMIO Example… (Lab 1 related)

```
void app_main(){
  int * temp_sensor = 0x30000004; //set pointer to a known address value
  int * heater = 0x30000008; //set pointer to a known address value
  //The two addresses above come from datasheet of processor!
  while(1){//run forever
    //check temperature...
    if (*temp_sensor <60){ //get value...less than 60?
      *heater = 1; //set value to 1 (let it warm)
    }else{
      *heater = 0; //set value to 0 (let it cool)
    }
  }
}
```

Thermometer circuit will be writing values to this memory spot

Address:	Value:
0x30000004:	0x00000023
0x30000008:	0x00000001
...	...
...	...
...	...
0x3fc93f58:	0x42016554
0x3fc93f5c:	0x30000004
0x3fc93f60:	0x30000008
0x3fc93f64:	0x00000000

Heater circuit will be reading values from this memory spot to know what to do

Recitation this week

- Go over concepts we just discussed
- Then do more of them in lab this week!

Pick up your lab kit today or tomorrow…definitely before Lab on Wednesday/Thursday!

38-530 left hand side of room

11:15am-12:15pm TODAY

or

Office Hours 3-5pm TOMORROW