# Lecture 09
# Memory and Binary

CS211 – Fundamentals of Computer Programming II

Branden Ghena – Spring 2023

Slides adapted from:
Jesse Tov

# Administrivia

- Homework 3 part 1 due today
  - Only need to submit code in `ballot.c` and `test_ballot.c`
  - (Unless you made any `Resources/` files. Submit those!)

- Homework 3 part 2 due next week Thursday
  - Can start submitting to Gradescope later today
  - Continuation of Part 1, so it shouldn’t be too hard to get started

# End of C!!

- Today is the last lecture on C

- Next week we’ll be starting C++!

- That means it’s time for another Lab
  - Will release sometime on Friday
  - Setup for CLion IDE and the SDL2 game engine
  - Reach out to me for help with this!

# Today’s Goals

- Discuss concept of pointers to pointers

- Practice dynamic memory allocation with arrays
  - How do we make an array the dynamically changes size?

- Go below the level of C and understand how the computer thinks about data with bits and bytes
  - Understand how this leads to the boundaries of common C types

# Getting the code for today

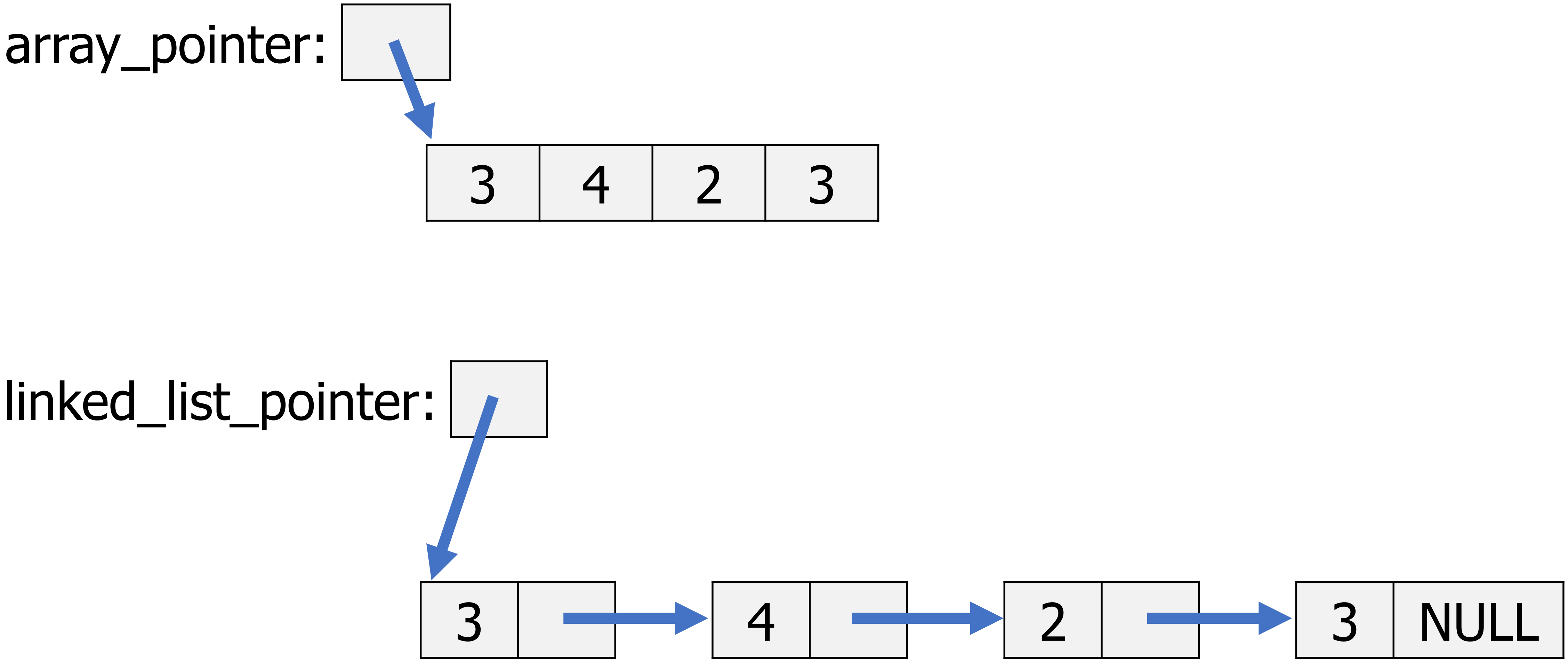
Same files as last lecture!

```
cd ~/cs211/lec/            (or wherever you put stuff)
tar -xkvf ~cs211/lec/08_linked_lists.tgz
cd 08_linked_lists/
```

# Outline

- **Linked Lists**
- Pointers to Pointers
- Dynamic Arrays
- Bits and Bytes
- Integer Encodings

# An alternative: linked allocations



array_pointer:

| 3 | 4 | 2 | 3 |
|---|---|---|---|

linked_list_pointer:

3 → 4 → 2 → 3 NULL

# C code for a linked list structure

- Array version:

```
int myarray[];
```

- Linked List version:

```
struct node {
  int value;
  struct node* next;
};
typedef struct node node_t;

node_t* head;
```

# Items can be added at any point in the list

linked_list-starter.c
linked_list-complete.c

- We can add/remove the middle item of the list
  - Just make sure you get the next pointer right

- Arrays can’t support that kind of thing
  - You would have to copy over all the later elements in the array

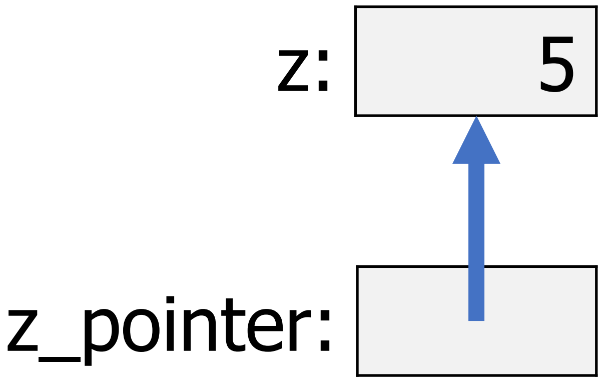
- Let’s write `list_append_front()` and `list_remove_front()` functions

# Outline

- Linked Lists
- **Pointers to Pointers**
- Dynamic Arrays
- Bits and Bytes
- Integer Encodings

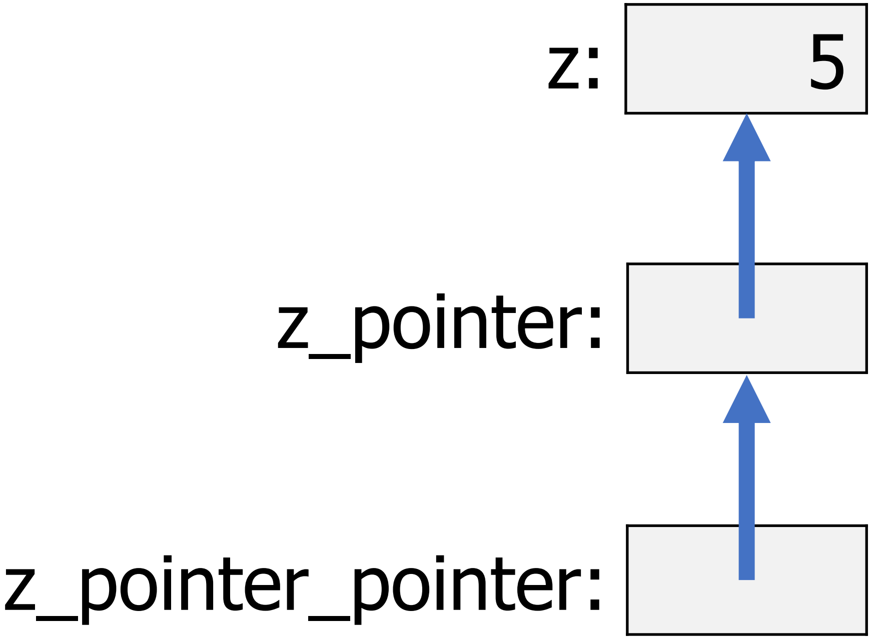
# Reminder: Pointers are another type of value

- Values could be a number, like 5 or 6.27
- Or they could be a “pointer” to an **object**
    - Points at the object, not the variable or value
    - It points at the “chunk of memory”
        - Technically, in C it holds the address of that memory



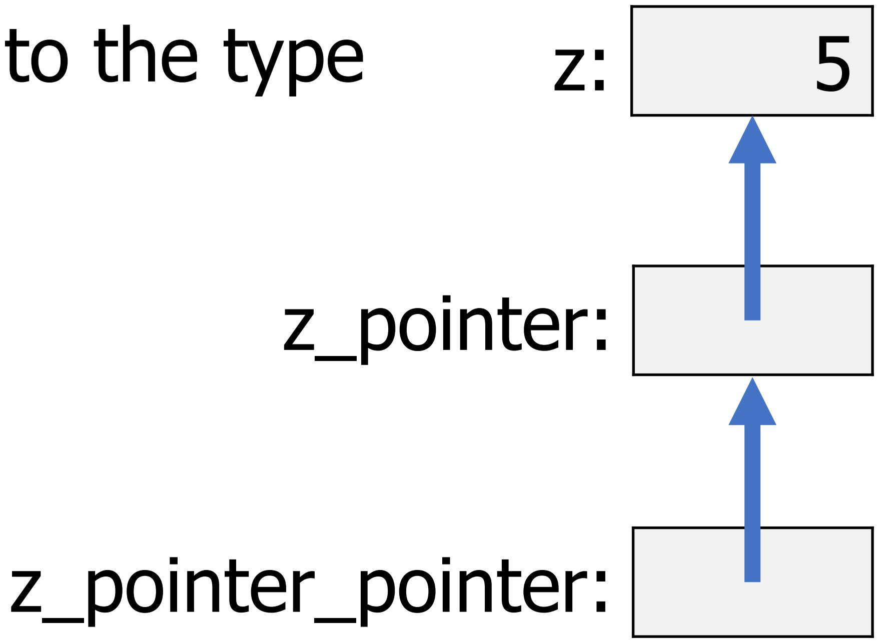
# We can make a pointer to another pointer

- Pointers are values stored in an object
    - That object has a memory address
    - We could make a pointer to a pointer



# Double pointers in C

- To make a pointer to something, add a * to the type



```
int z = 5;

int* z_pointer = &z;

int** z_pointer_pointer = &z_pointer;
```

# When is this useful?

linked_list.c
(from last lecture)

- Various functions in the linked list code need to return the new head of the linked list
  - Instead, they could update the linked list variable

```
struct node* list_append_front(struct node* list, int value);
```

could become

```
void list_append_front(struct node** list, int value);
```

# Also occurs in arguments to main

- argv is an array of strings
  - Strings are `char*`
  - So argv is `char**`

- `char* argv[]` is equivalent to `char** argv`

# Outline

- Linked Lists
- Pointers to Pointers
- **Dynamic Arrays**
- Bits and Bytes
- Integer Encodings

# Dealing with dynamic input

- What if you want to read in data, but you don't know how much data there might be?

- Arrays in C are a fixed size
- But you can `malloc()` as many times as needed
  - Request some memory
  - Use until you run out
  - Request more memory and copy existing values over

  - `realloc()` makes this simple, but it's still **slow**

# Example of dynamic memory: read_line()

```
char* read_line(void)
```

- Reads an entire line at a time from stdin
  - Can’t know in advance how many bytes there will be to read

  - Keeps reading in bytes until ‘\n’ character or end-of-file
  - Needs to request more memory until it holds the entire line

- Note: part of the 211 library, not standard C

# Live coding: implement read_line()

readline-starter.c
readline-complete.c

```
char* read_line(void)
```

- Requirements
  - Read from stdin until ‘\n’ or end-of-file (EOF)
  - Allocate an array to hold the read characters
    - Make sure to end it with a ‘\0’
  - Returns
    - NULL pointer if EOF was reached immediately
    - Pointer to string otherwise (not including the newline character)

# Realloc versus malloc

- We could just `malloc()` and copy ourselves, what does `realloc()` add?

- `realloc()` can be far more efficient
  - Doesn't have to copy data at all if there is room in the heap to expand

- Also simpler for programmers
  - Can't forget to free the old memory if `realloc()` does it for you

# Default string size will change efficiency

- Memory efficiency
  - Pointer returned could have way more memory than characters
  - User might hold on to memory for a while before freeing
  - The less wasted memory, the less memory the program needs

- Runtime speed
  - `malloc()` and `realloc()` are slow
  - The fewer times we call them, the faster the program will run

- Need to pick a sweet spot to balance the two of these
  - Real program: starts at 80 characters, doubles size when reallocating

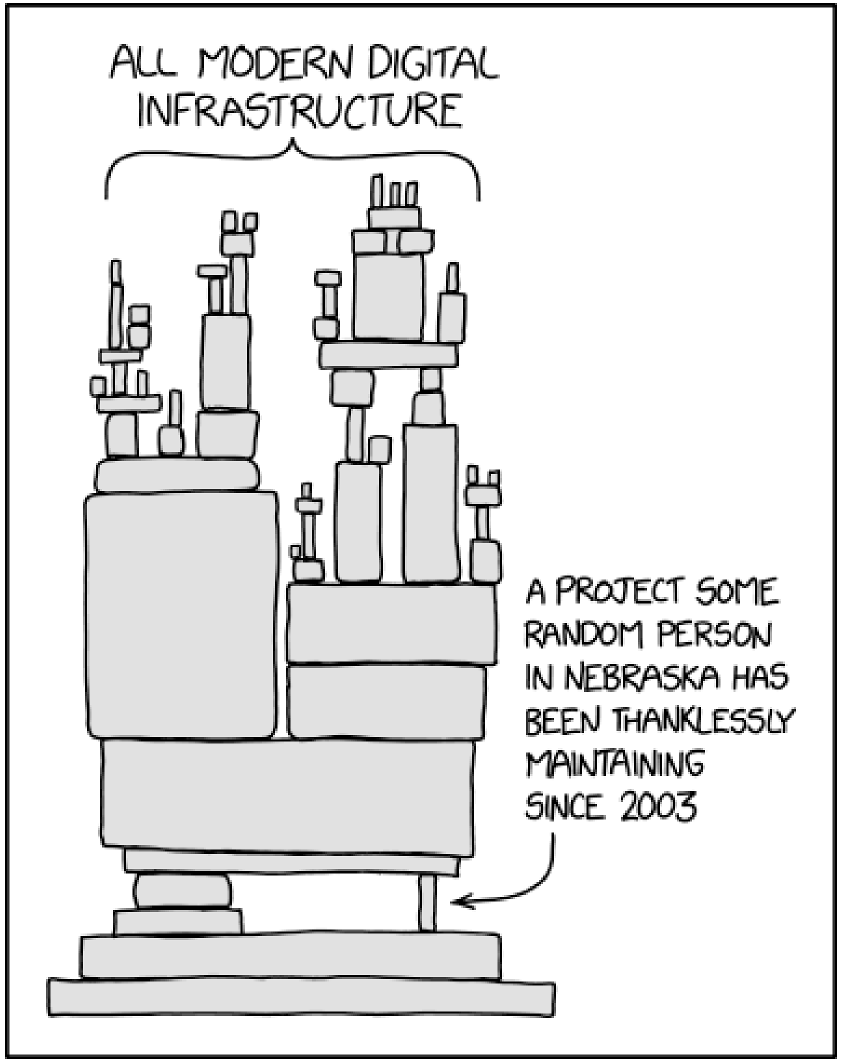
# Does efficiency really matter though?

- If you’re writing a CS211 homework: **no**

- If you’re writing a Javascript interpreter for Firefox,
  - Which has millions of users
  - times hundreds of websites per day for each user
  - times hundreds of lines of code per website
  - and each line of code is read with `read_line()`

  - **YES**

# Break + relevant xkcd



ALL MODERN DIGITAL INFRASTRUCTURE
A PROJECT SOME RANDOM PERSON IN NEBRASKA HAS BEEN THANKLESSLY MAINTAINING SINCE 2003

https://xkcd.com/2347/

# Outline

- Linked Lists
- Pointers to Pointers
- Dynamic Arrays
- **Bits and Bytes**
- Integer Encodings

# Learning binary

- To understand how a computer really works we need to understand that data it operates on

- Computers hold data in memory as individual ones and zeros
  - These ones and zeros make up binary values

- So, we’re going to need to understand binary
  - Binary will ***definitely*** come up again in this and other classes

# Positional Numbering Systems

- The position of a *numeral* (e.g., digit) determines its contribution to the overall number
  - Makes arithmetic simple (compared to, say, roman numerals)
  - Any number has one canonical representation

- Example: base 10
  - $10456_{10} = 1*10^4 \quad + \quad 0*10^3 \quad + \quad 4*10^2 \quad + \quad 5*10^1 \quad + \quad 6*10^0$
  - Usually, we leave out the zeros:
    - $1*10^4 \quad + \quad 4*10^2 \quad + \quad 5*10^1 \quad + \quad 6*10^0$

# Other bases are also possible

- Base 60, used by the Babylonians
  - The source of 60 seconds in a minute, 60 minutes in an hour
  - And 360 degrees in a circle

- Base 20, used by the Maya and Gauls
  - Parts of this remain in French today

- Base 2, used by computers
  - Example: $10010010_2$
  - Same idea as before: $1*2^7 + 1*2^4 + 1*2^1 = 128_{10} + 16_{10} + 2_{10} = 146_{10}$

# Base 2 Example

- Computer Scientists use base 2 a ***LOT*** (especially in computer systems)
- Let's convert $138_{10}$ to base 2
- We need to decompose $138_{10}$ into a sum of powers of 2
  - Start with the largest power of 2 that is smaller or equal to $138_{10}$
  - Subtract it, then repeat the process

$$138_{10} - \boxed{128_{10}} = 10_{10}$$

$$10_{10} - \boxed{8_{10}} = 2_{10}$$

$$2_{10} - \boxed{2_{10}} = 0_{10}$$

$$138_{10} = \underline{\mathbf{1}}\times128 + 0\times64 + 0\times32 + 0\times16 + \underline{\mathbf{1}}\times8 + 0\times4 + \underline{\mathbf{1}}\times2 + 0\times1$$

$$138_{10} = \underline{\mathbf{1}}\times2^7 + 0\times2^6 + 0\times2^5 + 0\times2^4 + \underline{\mathbf{1}}\times2^3 + 0\times2^2 + \underline{\mathbf{1}}\times2^1 + 0\times2^0$$

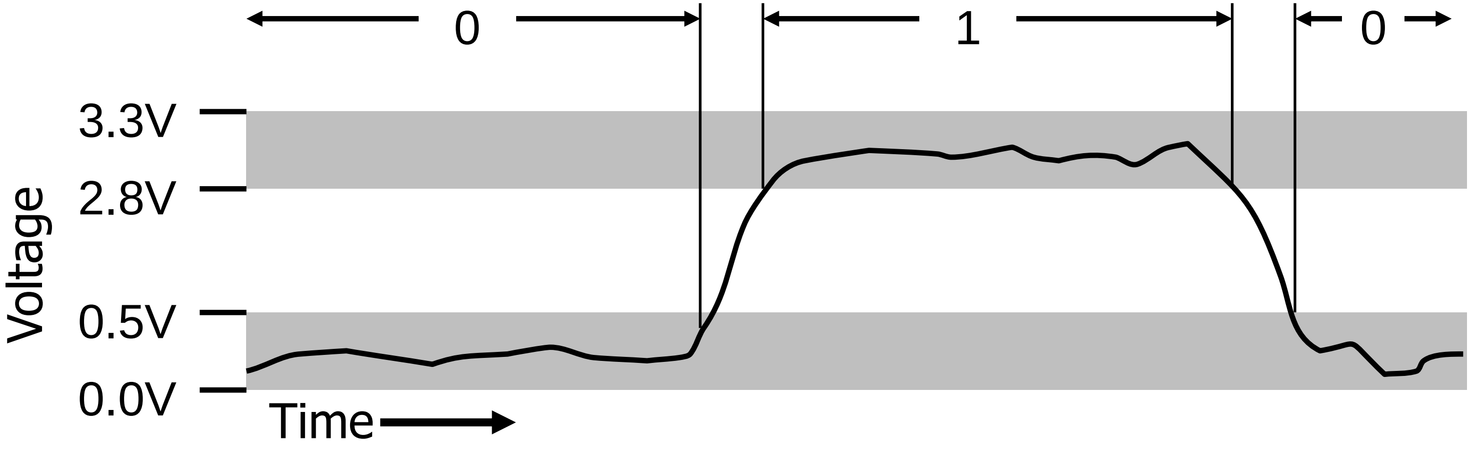
$$138_{10} = 10001010_2$$

# Binary practice

- Convert $101_2$ to decimal
  - $= 1\times2^2 + 0\times2^1 + 1\times2^0$
  - $= \quad 4 \;+\; 0 \;+\; 1$
  - $= \quad 5_{10}$
- Convert $4_{10}$ to binary: $100_2$ (one less than 5)
- Convert $6_{10}$ to binary: $110_2$ (one more than 5)

# Why computers use Base 2

- Simple electronic implementation
  - Easy to store with bi-stable elements
  - Reliably transmitted on noisy and inaccurate wires

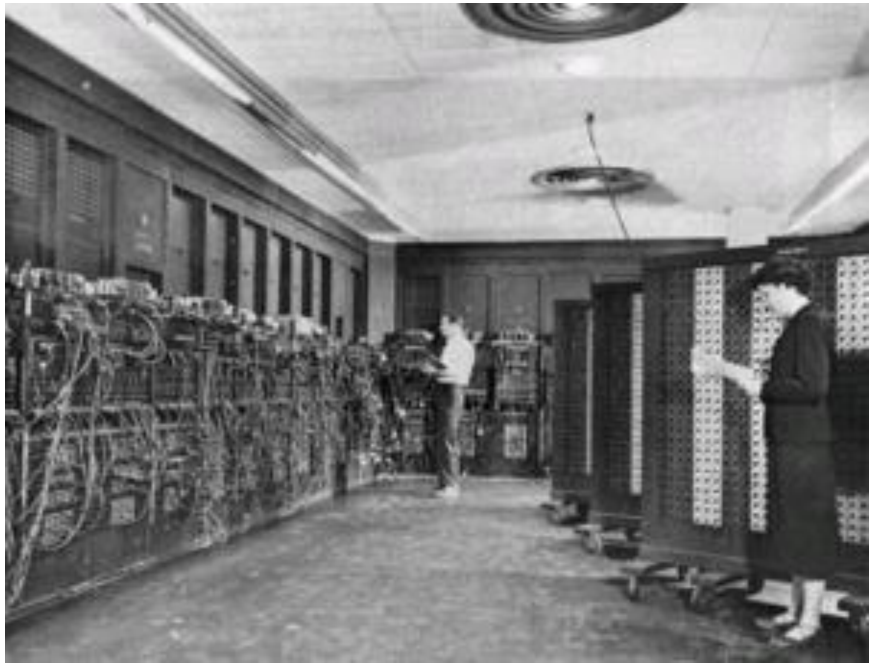


- Straightforward implementation of arithmetic functions

- (Pretty much) all computers use base 2

# Why don’t computers use Base 10?

- Because implementing it electronically is a pain
  - Hard to store
    - ENIAC (first general-purpose electronic computer) used 10 vacuum tubes / digit
  - Hard to transmit
    - Need high precision to encode 10 signal levels on single wire
  - Messy to implement digital logic functions
    - Addition, multiplication, etc.
    - (See CE203 for details)



# Base 16: Hexadecimal

- Writing long sequences of 0s and 1s is tedious and error-prone
  - And takes up a lot of space on a page!
- So we'll often use base 16 (also called *hexadecimal*)
- 
- Base 2 = 2 symbols (0, 1)
  Base 10 = 10 symbols (0-9)
  Base 16, need 16 symbols
  - Use letters A-F once we run out of decimal digits

| Hex | Decimal | Binary |
|---:|---:|---:|
| 0 | 0 | 0000 |
| 1 | 1 | 0001 |
| 2 | 2 | 0010 |
| 3 | 3 | 0011 |
| 4 | 4 | 0100 |
| 5 | 5 | 0101 |
| 6 | 6 | 0110 |
| 7 | 7 | 0111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| A | 10 | 1010 |
| B | 11 | 1011 |
| C | 12 | 1100 |
| D | 13 | 1101 |
| E | 14 | 1110 |
| F | 15 | 1111 |

# Base 16: Hexadecimal

- 16 = $2^4$, so every group of 4 bits becomes a hexadecimal digit (or *hexit*)
  - If we have a number of bits not divisible by 4, add 0s on the left (always ok, just like base 10)

0 0 1 0 | 1 0 0 1 | 0 1 1 1 | 1 0 1 1 → 0x297B

"0x" prefix = it's in hex

| Hex | Decimal | Binary |
|---|---|---|
| 0 | 0 | 0000 |
| 1 | 1 | 0001 |
| 2 | 2 | 0010 |
| 3 | 3 | 0011 |
| 4 | 4 | 0100 |
| 5 | 5 | 0101 |
| 6 | 6 | 0110 |
| 7 | 7 | 0111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| A | 10 | 1010 |
| B | 11 | 1011 |
| C | 12 | 1100 |
| D | 13 | 1101 |
| E | 14 | 1110 |
| F | 15 | 1111 |

# Bytes

- A single bit doesn't hold much information
  - Only two possible values: 0 and 1
  - So we'll typically work with larger groups of bits

- For convenience, we'll refer to groups of 8 bits as ***bytes***
  - And usually work with multiples of 8 bits at a time
  - Conveniently, 8 bits = 2 hexits

- Some examples
  - 1 byte: 0b01100111 = 0x67
  - 2 bytes: 11000100 $00101111_2$ = 0xC42F

"0b" prefix = it's in binary

# Practice problem

- **Convert 0x42 to decimal**

- Steps
  - Convert 0x42 to binary:
  - Convert binary to decimal:

| Hex | Decimal | Binary |
|---|---|---|
| 0 | 0 | 0000 |
| 1 | 1 | 0001 |
| 2 | 2 | 0010 |
| 3 | 3 | 0011 |
| 4 | 4 | 0100 |
| 5 | 5 | 0101 |
| 6 | 6 | 0110 |
| 7 | 7 | 0111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| A | 10 | 1010 |
| B | 11 | 1011 |
| C | 12 | 1100 |
| D | 13 | 1101 |
| E | 14 | 1110 |
| F | 15 | 1111 |

# Practice problem

- **Convert 0x42 to decimal**

- Steps
  - Convert 0x42 to binary:
    - 0x4 -> 0b0100     0x2 -> 0b0010          0x42 -> 0b 0100 0010
  - Convert binary to decimal:

# Practice problem

- **Convert 0x42 to decimal**

- Steps
  - Convert 0x42 to binary:
    - 0x4 -> 0b0100     0x2 -> 0b0010     0x42 -> 0b 0100 0010
  - Convert binary to decimal:
    - $1*2^6 + 1*2^1 = 64 + 2 = 66$

# Outline

- Linked Lists
- Pointers to Pointers
- Dynamic Arrays
- Bits and Bytes
- **Integer Encodings**

# These two lines of code are equivalent

```
char mychar = 97;

char mychar = ‘a’;
```

- Per the ASCII table, the character ‘a’ has a decimal value 97
  - The character value and decimal value are equivalent
  - These two are also equivalent

```
char diff = ‘c’ - ‘a’;

char diff = 99 - 97;
```

# **Big idea:** bits can be used to represent anything

- Depending on the context, the bits **`11000011`** could mean
  - The number 195
  - The number -61
  - The number -1.1875
  - The value `True`
  - The character '├'
  - The **`ret`** x86 instruction

- You have to know the **context** to make sense of any bits you have!
  - People and software they write determine what the bits actually mean

# Integer types in C

- C type provides both size and encoding rules

- Integer types in C come in two flavors
  - Signed: **`short`**, **`signed short`**, **`int`**, **`long`**, …
  - Unsigned: **`unsigned char`**, **`unsigned short`**, **`unsigned int`**, …

- And in multiple different sizes
  - 1 byte: **`signed char`**, **`unsigned char`**
  - 2 bytes: **`short`**, **`unsigned short`**
  - 4 bytes: **`int`**, **`unsigned int`**
  - Etc.

# Sizes of C types are system dependent

| C Data Type | Intel IA32 | x86-64 | C Standard*<br>(C99) |
|---|---|---|---|
| char | 1 | 1 | ≥1 |
| short | 2 | 2 | ≥2 |
| int | 4 | 4 | ≥2 |
| long | 4 | 8 | ≥4 |
| long long | 8 | 8 | ≥8 |
| float | 4 | 4 | |
| double | 8 | 8 | |
| pointer | 4 | 8 | Widths for data, code pointers may differ! |

# Expressing C types in bits

- Two families of encodings to express integers using bits
  - ***Unsigned*** encoding for unsigned integers
  - ***Two's complement*** encoding for signed integers

- Each encoding will use a fixed size (# of bits)
  - For a given machine
  - Size + encoding family determine which C type we're representing
  - Fixed size is because computers are finite!

# Unsigned integer encoding

- Just write out the number in binary
  - Works for 0 and all positive integers

- Example: encode $104_{10}$ as an **unsigned** 8-bit integer
  - $104_{10} = 0\times2^7 + 1\times2^6 + 1\times2^5 + 0\times2^4 + 1\times2^3 + 0\times2^2 + 0\times2^1 + 0\times2^0$

$\Rightarrow$ **`01101000`**

$\Rightarrow$ **`0x68`**

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

(Binary To Unsigned)

# Bounds of unsigned integers

- For a fixed width ***w***, a limited range of integers can be expressed

  - Smallest value (we will call ***UMin***):
    - all 0s bit pattern: 000…0, value of 0

  - Largest value (we will call ***UMax***):
    - all 1s bit pattern: 111...1, value of $2^w - 1$

    - $2^w - 1 = 1\times2^{w-1} + 1\times2^{w-2} + ... + 1\times2^1 + 1\times2^0 = 11111...$

- Maximum 8-bit number = $2^8$-1 = 256-1 = 255

# Encoding signed integers

- What’s different about representing a signed number?
  - It can be negative!

- So, we’re going to have to somehow represent values that are negative and positive

- There are actually many different encodings capable of doing this
  - This is when that “nice encoding” versus “annoying encoding” matters

# Two's complement encoding

- Plan:
  - Start with unsigned encoding, but make ONLY the largest power negative
  - Example: for 8 bits, most significant bit is worth $-2^7$ not $+2^7$ (other bits are still positive)

- To encode a negative integer
  - First, set the most significant bit to 1 to start with a big negative number
  - Then, add positive powers of 2 (the other bits) to "get back" to number we want

- Example: encode -6 as a 4-bit two's complement integer
  - $-6_{10} = 1 \times -2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^1$ ⇒ 0b**1010** ⇒ **0xa**

# Two's complement examples

- Encode -100 as an 8-bit two's complement number

  - $-100_{10}$ = $1 \times -2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$

    $-128 \quad + 0 \quad + 0 \quad + 16 \quad + 8 \quad +4 \quad +0 \quad +0$

    Problem becomes:
    encode +28 as a 7-bit unsigned number

  - $-100_{10}$ = 0b10011100 = 0x9C

# Interpreting binary signed values

- Converting binary to signed:

$$B2T(X) \;=\; -\underset{\substack{\uparrow\\ \text{Sign bit}}}{x_{w-1}} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^{i}$$

- Note: most significant bit still tells us sign!! 1-> negative
    - Checking if a number is negative is just checking that top bit

- Zero problem is always all zeros
    - 0b00000000 = 0 0b10000000 = -128

- -1: 0b111…1 = -1 (regardless of number of bits!)

# Bounds of two's complement integers

- For a fixed width ***w***, a limited range of integers can be expressed
  - Smallest value, most negative (we will call ***TMin***):
    - 1 followed by all 0s bit pattern: 100…0 = $-2^{w-1}$
  - Largest value, most positive (we will call ***TMax***):
    - 0 followed by all 1s bit pattern: 01...1, value of $2^{w-1} - 1$
- Beware the asymmetry! Bigger negative number than positive

# Ranges for different bit amounts

| | W | | | |
|---|---|---|---|---|
| | 8 | 16 | 32 | 64 |
| UMax | 255 | 65,535 | 4,294,967,295 | 18,446,744,073,709,551,615 |
| TMax | 127 | 32,767 | 2,147,483,647 | 9,223,372,036,854,775,807 |
| TMin | -128 | -32,768 | -2,147,483,648 | -9,223,372,036,854,775,808 |

- Observations
  - | *TMin* | = *TMax* + 1
    - Asymmetric range
  - *UMax* = 2 * *TMax* + 1

- C Programming
  - #include <limits.h>
  - Declares constants, e.g.,
    - ULONG_MAX
    - LONG_MAX
    - LONG_MIN
  - Values are platform specific

# Overflow

- What happens if you exceed the bound of a variable type?

# Overflow

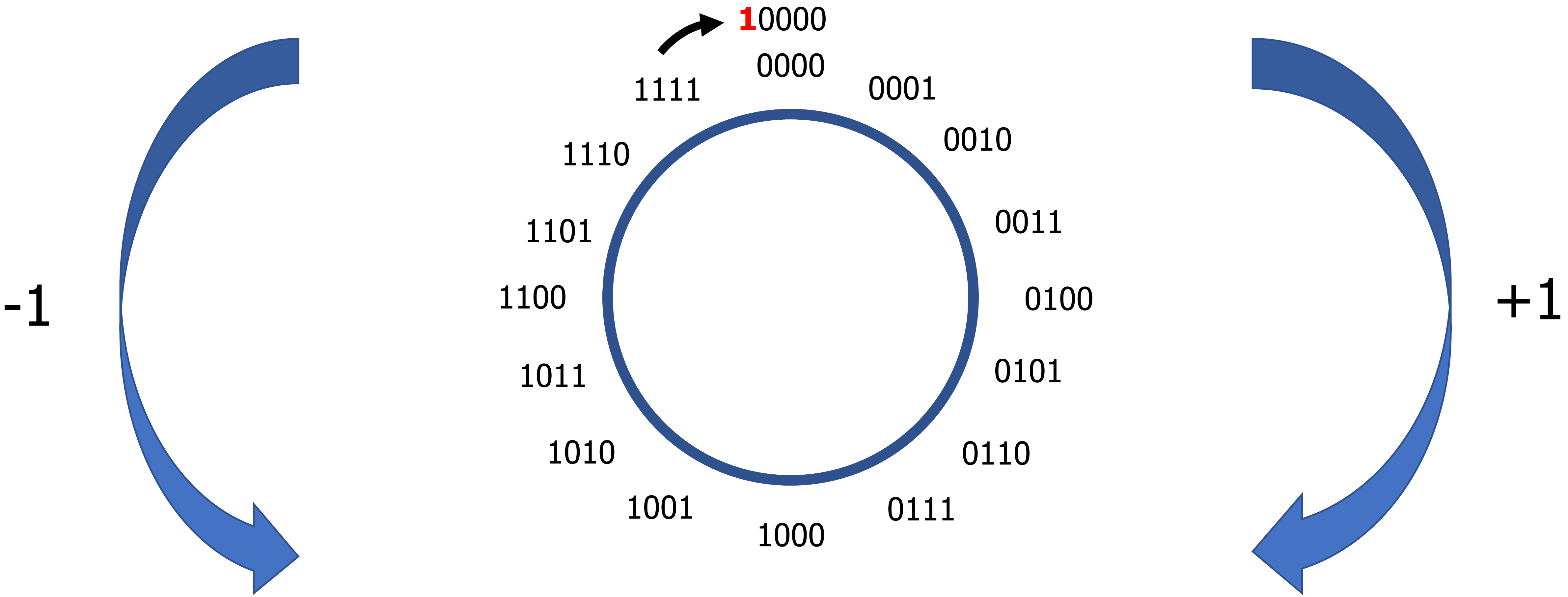
- What happens if you exceed the bound of a variable type?

- Unsigned Variables
  - They wrap!

```
char a = 255;
a++;
// a now equals 0

char b = 2;
b = b-5;
// b now equals 253
```

# Modulo behavior in binary numbers



# Overflow

- What happens if you exceed the bound of a variable type?

- Signed Variables
  - **UNDEFINED BEHAVIOR**

  - Usually they wrap (that’s what the hardware does)
  - But also the compiler can do anything it wants

# Remember that overflow/underflow can occur in C

- Warning: programmers often fail to account for wrapping!
  - Sometimes it leads to unexpected behavior

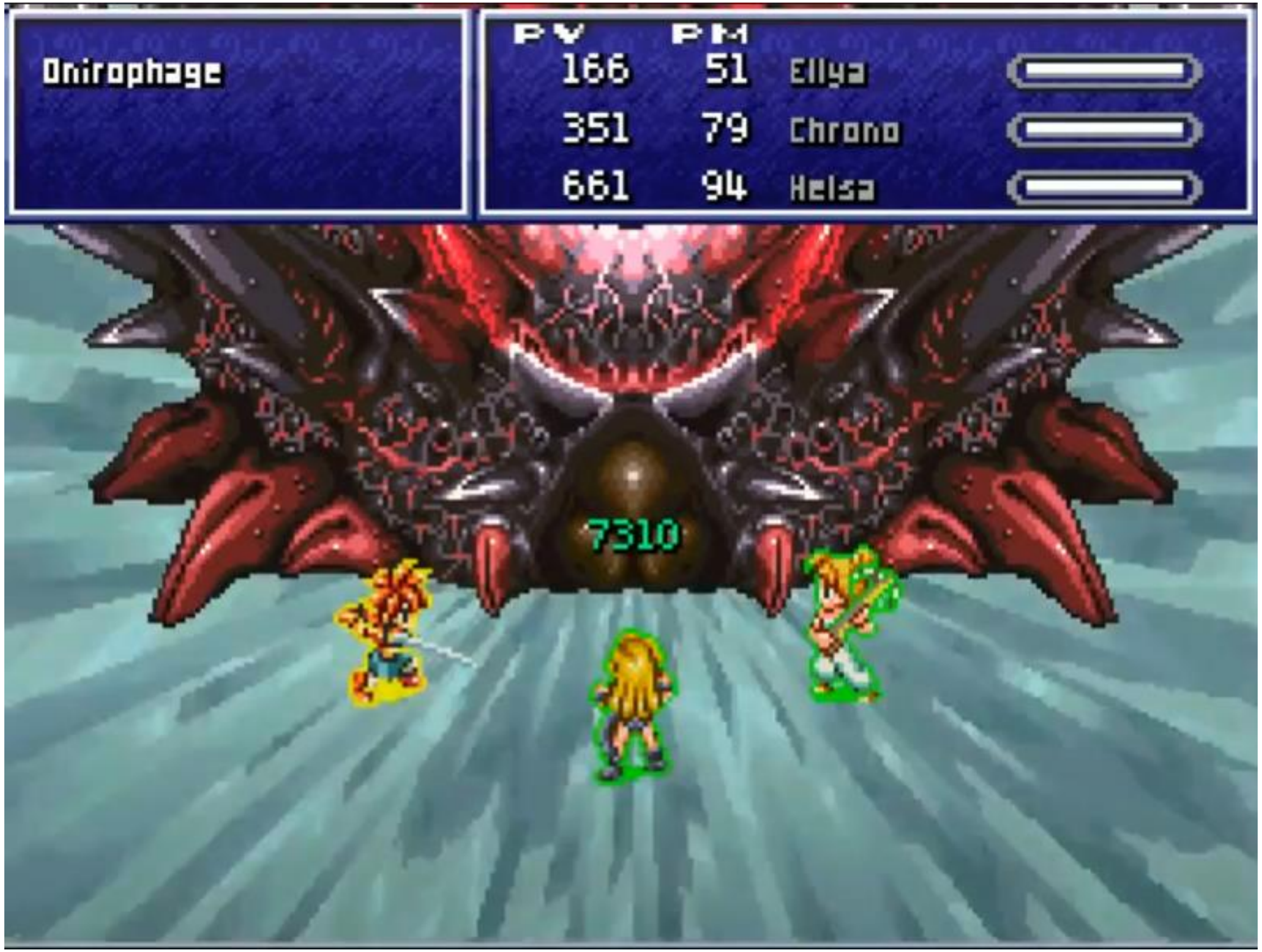
# Overflow example in the real world

- Dream Devourer
  - Special boss in the Nintendo DS edition
- Wanted to make it even more challenging
  - 32000 hit points
  - Takes *forever* to defeat
- Hit points stored as a 16-bit signed integer
  - Range: -32768 to +32767

# Chrono Trigger signed overflow bug

- Solution: heal it

- Hit points go negative and it dies



# Outline

- Linked Lists
- Pointers to Pointers
- Dynamic Arrays
- Bits and Bytes
- Integer Encodings