# Lecture 09
# Bits, Bytes, and Integer Encoding

CS211 – Fundamentals of Computer Programming II

Branden Ghena – Winter 2022

Slides adapted from:
Jesse Tov

Northwestern

## Administrivia

- Homework 4 due on Thursday
  - You can do it!

- Remember that office hours get busy right before the deadline
  - It’ll be harder to get help and you’ll get less time

## Administrivia

- No lecture on Thursday
  - Take a nap instead so you can recharge
- Next week starts C++ 🎉

## Today's Goals

- Discuss concept of pointers to pointers

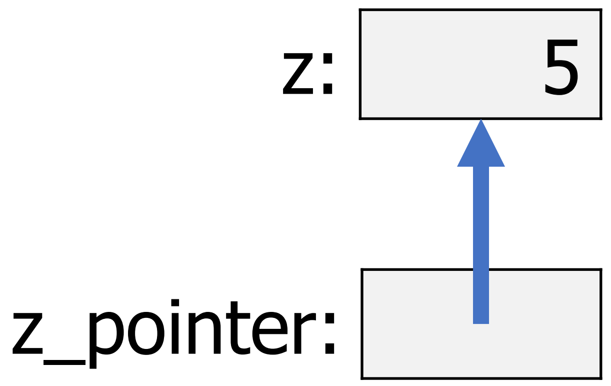
- Go below the level of C and understand how the computer thinks about data with bits and bytes
  - Understand how this leads to the boundaries of common C types
  - Note: this isn't a main focus of this class
    - I just wanted to take today to explain more deeply
    - This will all come up again if you take CS213

## Outline

- **Pointers to Pointers**
- Bits and Bytes
- Integer Encoding
- C Type Bounds

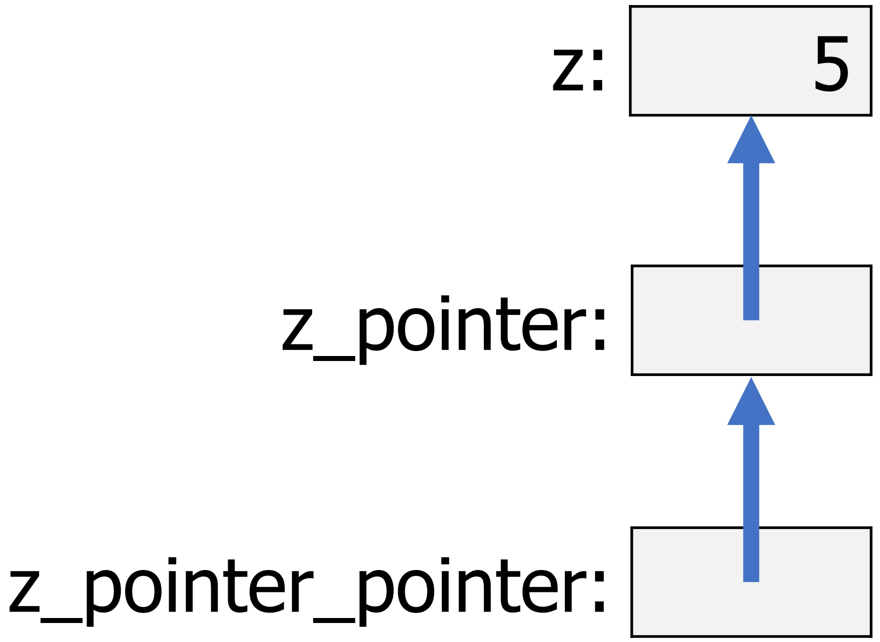
## Reminder: Pointers are another type of value

- 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



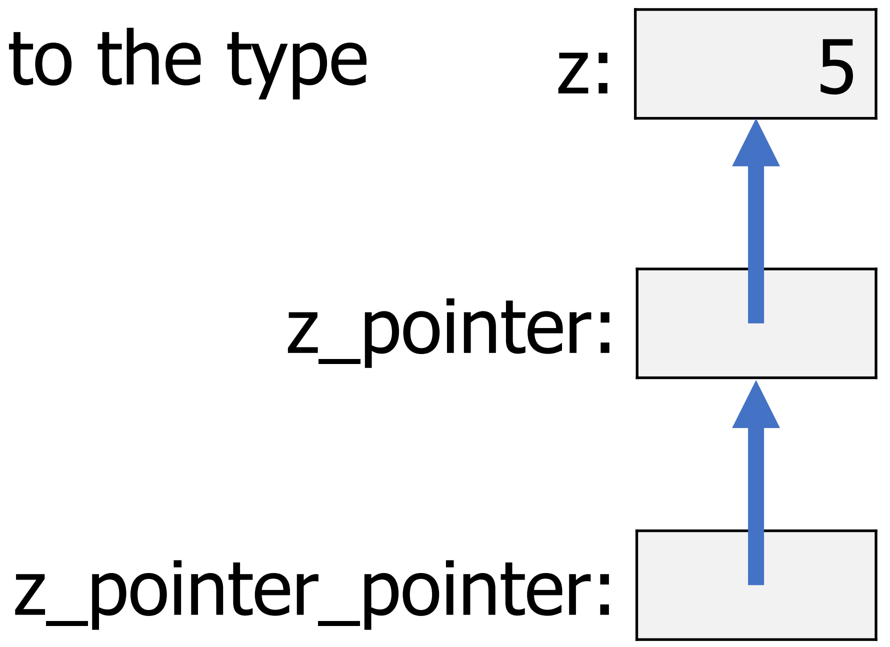
## We can make a pointer to another pointer

- 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

- Pointers to Pointers
- **Bits and Bytes**
- Integer Encoding
- C Type Bounds

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

# Positional Numbering Systems

- 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 (bits remain in French today)

  - Base 2: $10010010_2$

    $= 1*2^7 + 1*2^4 + 1*2^1$

    $= 128_{10} + 16_{10} + 2_{10}$

    $= 146_{10}$

## Base 2 Example

- Computer Scientists use base 2 a ***LOT***
- Let's convert $134_{10}$ to base 2
- We need to decompose $134_{10}$ into a sum of powers of 2
  - Start with the largest power of 2 that is smaller or equal to $134_{10}$
  - Subtract it, then repeat the process

$$134_{10} - \boxed{128_{10}} = 6_{10}$$
$$6_{10} - \boxed{4_{10}} = 2_{10}$$
$$2_{10} - \boxed{2_{10}} = 0_{10}$$

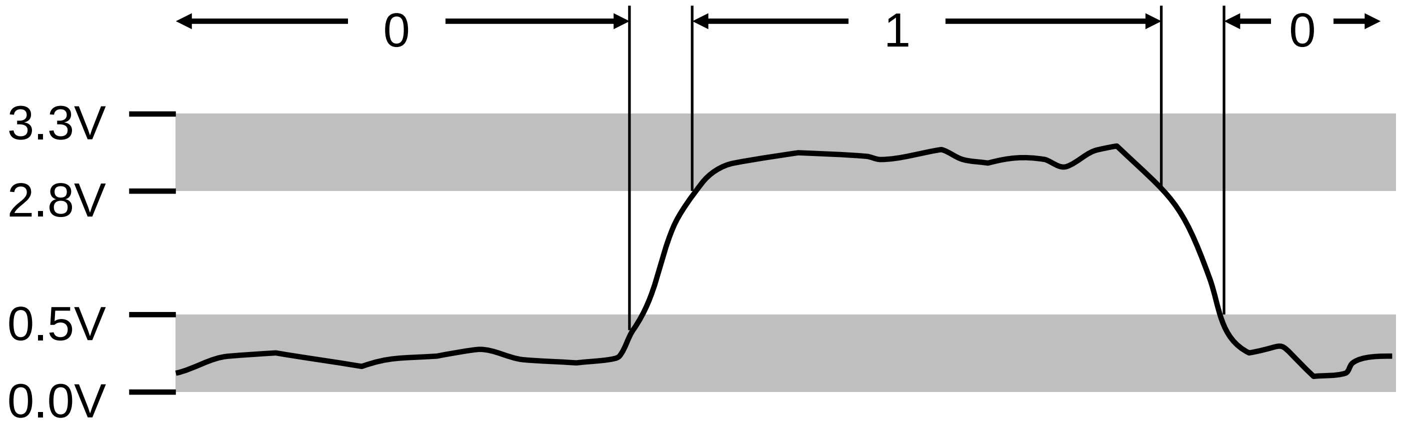
$$134_{10} = \underline{\mathbf{1}}\times128 + 0\times64 + 0\times32 + 0\times16 + 0\times8 + \underline{\mathbf{1}}\times4 + \underline{\mathbf{1}}\times2 + 0\times1$$

$$134_{10} = \underline{\mathbf{1}}\times2^7 + 0\times2^6 + 0\times2^5 + 0\times2^4 + 0\times2^3 + \underline{\mathbf{1}}\times2^2 + \underline{\mathbf{1}}\times2^1 + 0\times2^0$$

$$134_{10} = 10000110_2$$

# Why computers use Base 2

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

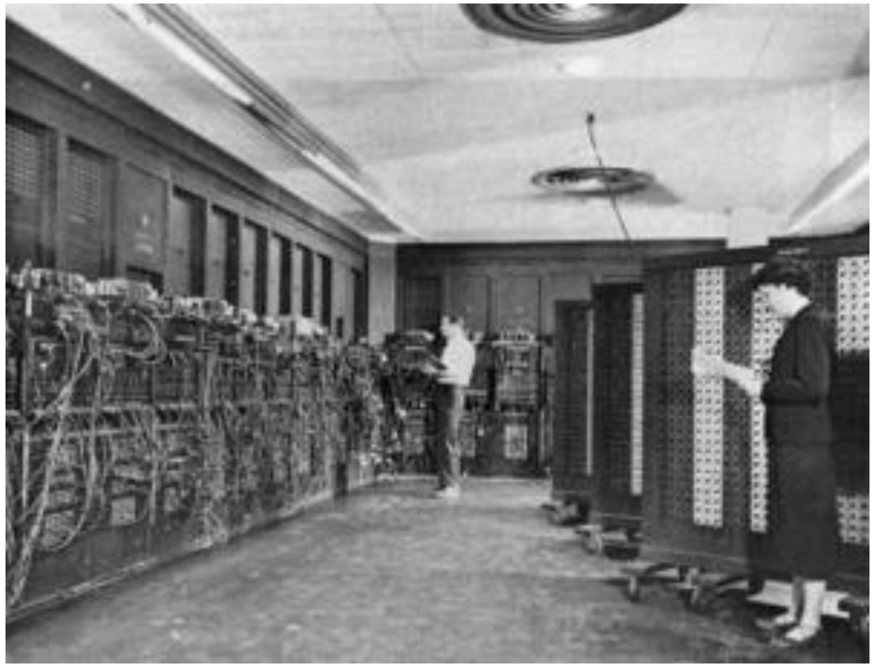


- Straightforward implementation of arithmetic functions

- (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

## Break + Practice problem

- **Convert 0x42 to decimal**

- Steps
  - Convert 0x42 to binary:

  - Convert binary to decimal:

## Break + Practice problem

- **Convert 0x42 to decimal**

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

## Break + Practice problem

- **Convert 0x42 to decimal**

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

## Break + Practice problem

- **Convert 0x42 to decimal**

- Critical thinking:
  - What are the maximum and minimum values?
    - Minimum 0 (0x00)
    - Maximum 255 (0xFF)
  - How big is 0x42 out of 0xFF?
    - ~25% (0x40, 0x80, 0xC0, 0x100)
    - So 255/4 ≈ 256/4 ≈ 64

## Outline

- Pointers to Pointers
- Bits and Bytes
- **Integer Encoding**
- C Type Bounds

## 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

## Expressing C types in bits

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

- Size + encoding family determine which C type we're representing
    - Each type will use a fixed size (# of bits)
        - For a given machine
        - 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

## Two's complement encoding

- Good news: can represent both positive and negative numbers
- Bad news: need to make the encoding more complicated

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

- 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
  - $-6_{10} = 1 \times -2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \Rightarrow$ `0b1010` $\Rightarrow$ **`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

## Two's Complement Shortcut

- **Shortcut:** determine positive version of number, flip it, and add one

  - $100_{10}$ = 0b01100100
  - Flipped = 0b10011011
  - Plus 1 = 0b10011100 = 0x9C

Sidebar: binary addition

```
 0b01           0b011
+0b01          +0b001
------         -------
 0b10           0b100
```

## Interpreting binary signed values

- Converting binary to signed:

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

- Note: most significant bit tells us sign!! 1 -> negative
  - Checking if a number is negative is just checking that top bit
- Note: there is only one zero value
  - 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

# Unsigned & Signed Numeric Values

| *X* | B2U(*X*) | B2T(*X*) |
|---|---|---|
| 0000 | 0 | 0 |
| 0001 | 1 | 1 |
| 0010 | 2 | 2 |
| 0011 | 3 | 3 |
| 0100 | 4 | 4 |
| 0101 | 5 | 5 |
| 0110 | 6 | 6 |
| 0111 | 7 | 7 |
| 1000 | 8 | –8 |
| 1001 | 9 | –7 |
| 1010 | 10 | –6 |
| 1011 | 11 | –5 |
| 1100 | 12 | –4 |
| 1101 | 13 | –3 |
| 1110 | 14 | –2 |
| 1111 | 15 | –1 |

- **Equivalence**
  - Same encodings for non-negative values
- **Uniqueness**
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding
- $\Rightarrow$ **Can Invert Mappings**
  - Can go from bits to number and back, and vice versa
  - $\mathrm{U2B}(x) = \mathrm{B2U}^{-1}(x)$
    - Bit pattern for unsigned integer
  - $\mathrm{T2B}(x) = \mathrm{B2T}^{-1}(x)$
    - Bit pattern for two’s complement integer

## Practice + Break

- What range of integers can be represented with 5-bit two’s complement?

  - A -31 to +31
  - B -15 to +15
  - C 0 to +31
  - D -16 to +15
  - E -32 to +31

## Practice + Break

- What range of integers can be represented with 5-bit two's complement?

| | | |
|---|---|---|
| • A | -31 to +31 | No asymmetry and 6-bits |
| • B | -15 to +15 | No asymmetry |
| • C | 0 to +31 | Unsigned |
| [✓ • D] | [✓ -16 to +15] | Correct |
| • E | -32 to +31 | 6-bits |

## Outline

- Pointers to Pointers
- Bits and Bytes
- Integer Encoding
- **C Type Bounds**

## Standard sizes of C types on modern (64-bit) computers

- 1 byte
  - char, unsigned char, signed char
  - bool
- 2 bytes
  - short, unsigned short, signed short
- 4 bytes
  - int, unsigned int, signed int
  - float
- 8 bytes
  - long, unsigned long, signed long
  - double
  - Every pointer type! (for a 64-bit computer, which you’re all on)

## 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

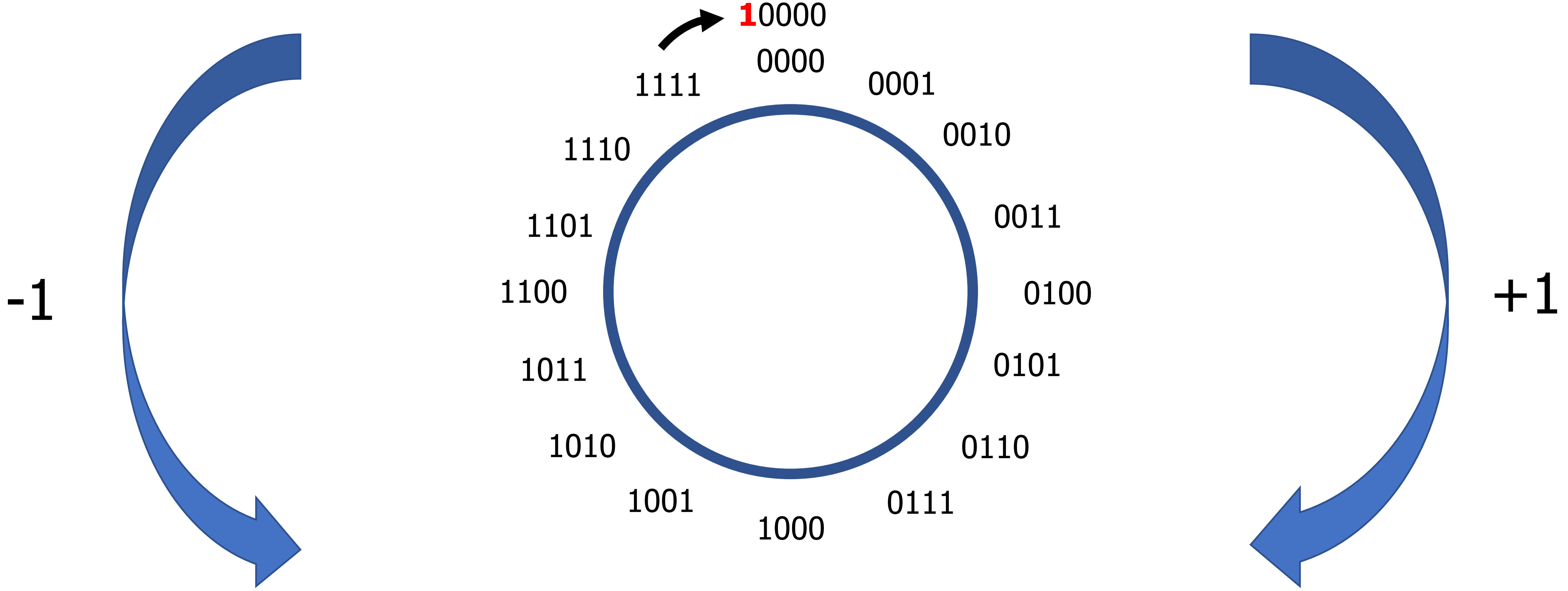
- What happens if you exceed the bound of a variable type?

- 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**

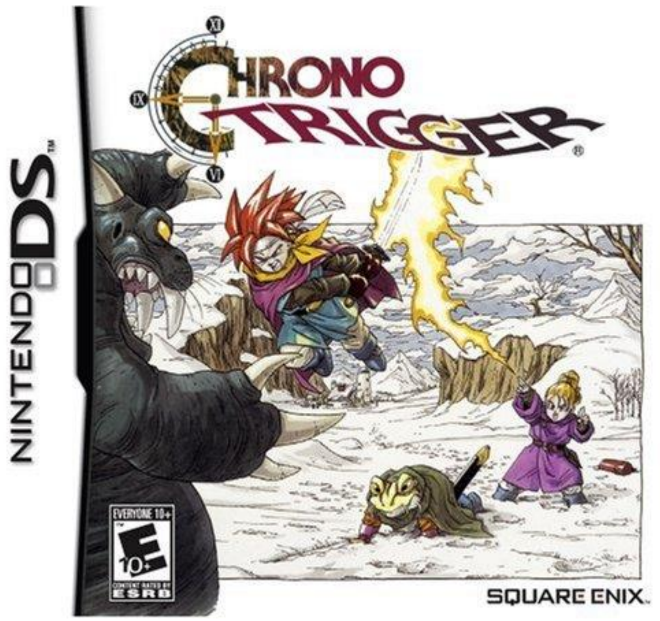
  - Usually they wrap (that’s what the hardware does)
  - But also the compiler can do anything it wants

## 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

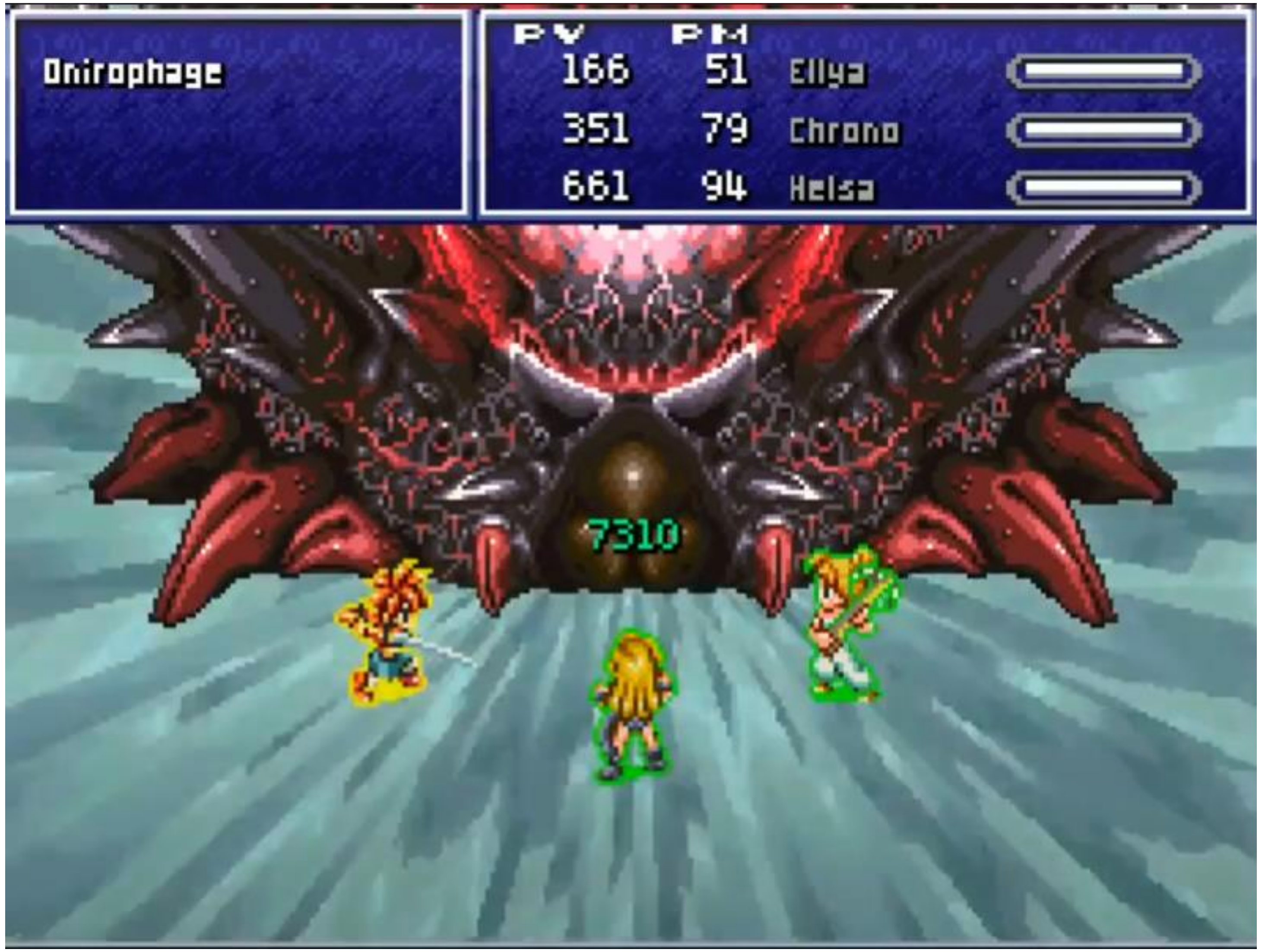
- Dream Devourer
  - Special boss in the Nintendo DS edition
- Wanted to make it even more challenging
  - 32000 hit points
  - Takes *forever* to defeat
- Hit points stored as a 16-bit signed integer
  - Range: -32768 to +32767



## Chrono Trigger signed overflow bug

- Solution: heal it

- Hit points go negative and it dies



## Outline

- Pointers to Pointers
- Bits and Bytes
- Integer Encoding
- C Type Bounds