



CS107 Lab 1 — Welcome!

Bits, Bytes, and Integers 🐙

Ben Yan, Winter 2026 🌨️

Slides Link: stanford.edu/~bbyan/cs107 :)



Hope that you're looking forward to a fun time in Jerryland!

Stanford | ENGINEERING
Computer Science

Hi everyone – so excited to meet you all! 🙌

Here's where we are: our quarter's just getting started



Winter



Week 2



Spring

Hope that you had a beautiful winter break!

Hello I'm Ben! (Pronouns: he/they), bbyan@stanford.edu



Short Bio

- ❑ **CS co-term!**  For my undergrad: I double majored in CS and Math,  minored in Creative Writing, also studied poetry/lit abroad at Oxford
- ❑ **It was a lot lol, kind of fun but very messy**
- ❑ Jerry is my advisor, helped me out of tough spots



Hello I'm Ben! (Pronouns: he/they), bbyan@stanford.edu



What I've loved most here has been CS teaching

- ❑ **TAed 107 last winter + spring, elated to be back!**
- ❑ **Also SLing 106A this quarter (oh the double life)**
- ❑ **Once upon a time, Head TA of 106AX! Jerry made the terrible mistake once of letting me lecture :)**
- ❑ **Formerly lectured a chic 1-unit wonder CS106S**



So blessed I get to be here!



My Interests, What Gives Me Joy

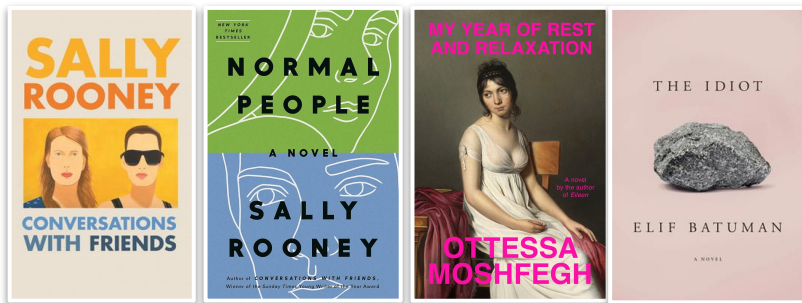
I'm a bit of a diva for dedicating an entire slide to this, but lmk if we have shared interests!



Anime, Manga, Visual Novels



Rock Music, K-pop, Nirvana



**Novels by Sally Rooney,
Otessa Moshfegh, Elif Batuman**



Language Learning (currently Korean!)



But more importantly, introduce yourselves!

- ❑ Your name!
- ❑ Your pronouns if you feel comfortable sharing
- ❑ What you're studying / plan to study
- ❑ 📦 **Icebreaker: A story about your name**, or alternatively, **any one of the following!**



What are you looking forward to this quarter?



Something fun you did over the winter break?



Music, album, or book recommendations?

Or anything else you'd like to share! :)

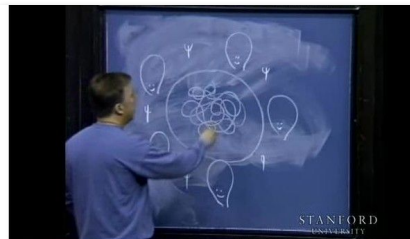
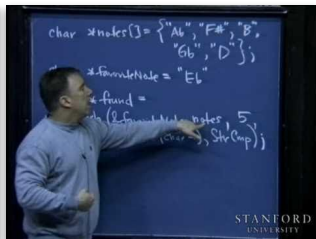
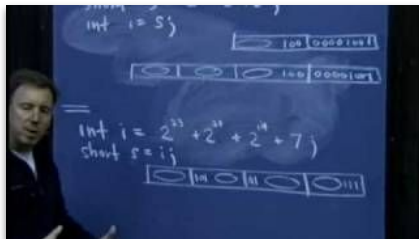




Lab Mechanics — How It Works

- ❑ Think of **Jerry's lecture as a dramatic, stunning YA novel** (his voice is indeed audiobook quality)
- ❑  **Lab is our cozy little book club**: we debrief the plot/drama and tea and what's going on each week :)

Section and it's
the same but
there's 30
more minutes
so it's not

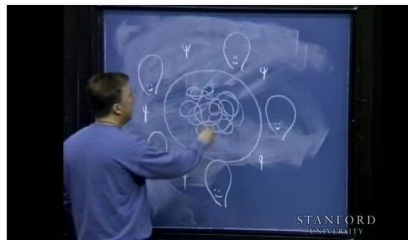
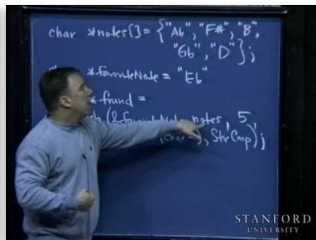
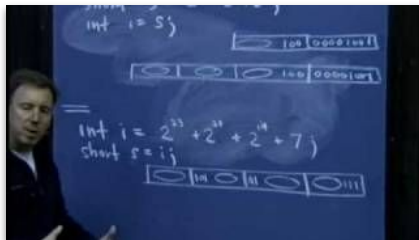




Lab Mechanics — How It Works

- ❑ Think of **Jerry's lecture as a dramatic, stunning YA novel** (his voice is indeed audiobook quality)
- ❑  **Lab is our cozy little book club**: we debrief the plot/drama and tea and what's going on each week :)

Section and it's the same but there's 30 more minutes so it's not



Summary: We'll engage with many exercises—from debugging to memory sorcery to code-writing—to **get practice with the material + prepare for assignments!** ✨

Note: Weekly lab attendance is expected, but let me and/or Jerry know if difficulties arise during the quarter! **As a staff, we are here to support you in any way we can** ❤️



Lab Mechanics — More Logistical Details



Contact

bbyan@stanford.edu, or over Slack!



Structure

1. **Overview of most significant bits** of prior week's lecture material, Q & A
2. **Group work on lab handout/exercises**, then we debrief together
3. Lastly, a **check-off form** to submit! ✓



Lab Place & Time

Wed, 1:30 – 2:50 PM, Wallenberg 330
or **Thurs, 1:30 – 2:50 PM, McMurtry Art 360**

If you can't make lab any given week, feel free to attend another one (we'll miss you!!), and email me + the other TA.



Getting Help

See the course website for EdStem and the mosaic of CoDa office hours! Mine are typically:



Tues, 1:30 – 3:30 PM



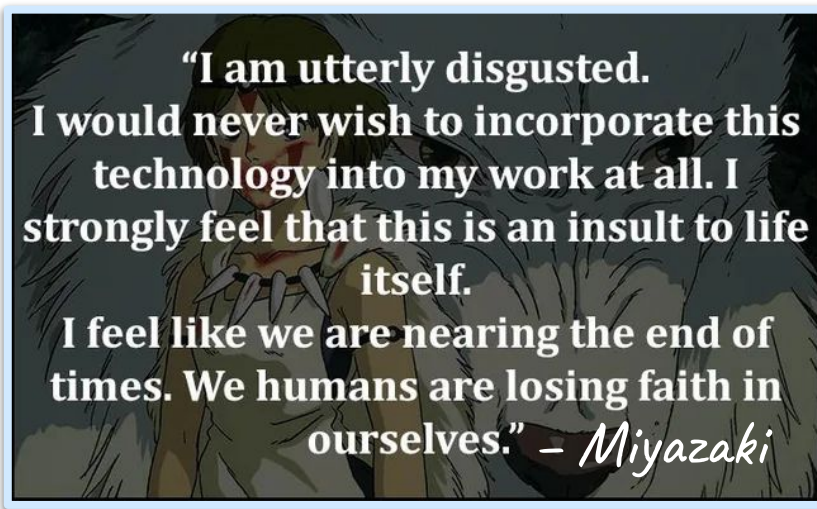
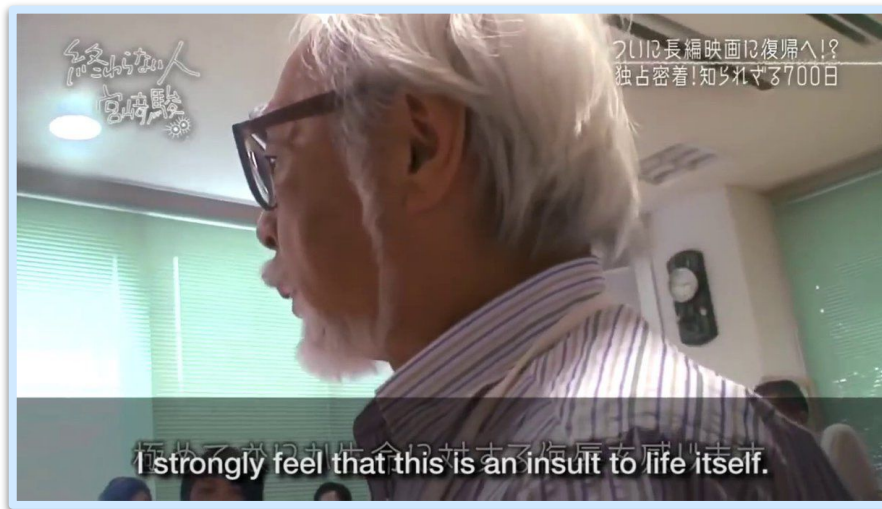
Tues, 5 – 7 PM



Will generally stay after lab for questions!

CS107 Policy: No AI Use for Writing Code Please 🙅

I promise you'll learn so much more without it! You >> ChatGPT





My (Embarrassingly Designed) Lab Website



I'll have lab slides, materials, and helpful links posted each week at:

`stanford.edu/~bbyan/cs107` :)

Ben's CS107 Lab

stanford.edu/~bbyan/cs107/

CS107 Lab

Ben Yan (bbyan@stanford.edu)

Winter 2026 Stanford University Welcome to Jerryland!

Hello and welcome! This is my homepage for CS107 lab slides, links to handouts, and general lab resources. If you have any questions, please reach out! So incredibly excited to meet you all and for a beautiful quarter in Jerryland.

My two lab times are listed below, though you're only required to attend one :)

- Wednesday 1:30 - 2:50 PM, Wallenberg Room 330
- Thursday 1:30 - 2:50 PM, McMurtry Art Room 360

Schedule

Lab 1 Bits, Bytes, and Integers — Jan 14-15

Slides Lab Handout Check-Off Form

Lab Syllabus Lecture Recap Debugging Guide Emacs Guide

Lab 2 C Strings, String Tokenization — TBD

Lab 3 Pointers, Arrays and the Heap — TBD

Ben's CS107 Lab

stanford.edu/~bbyan/cs107/

Lab 2 C Strings, String Tokenization — TBD

Lab 3 Pointers, Arrays and the Heap — TBD

Lab 4 Void Star, Function Pointers, Generics — TBD

Lab 5 Assembly, Translating and Tracing Machine-Level Code — TBD

Lab 6 Runtime Stack, Stack Smash Attack — TBD

Lab 7 Optimizing, Ethics, Parting Thoughts, Beyond CS107 — TBD

Slides Info

Lab Mechanics

Lab is graded on attendance and participation i.e. showing up and filling out an online check-off form as you work, with credit being awarded for a sincere effort as opposed to getting every detail correct. If you have any questions, please feel free to ask me!

For getting help in CS107 generally, please see the [course website](#) on the office hours mosaic each week and the EdStem forum. My office hours are typically:

- Tuesdays, 1:30 - 3:30 PM, CoDa Basement
- Tuesdays, 5 - 7 PM, CoDa Basement

The hours may vary by week, so the calendar has the most up-to-date information :)

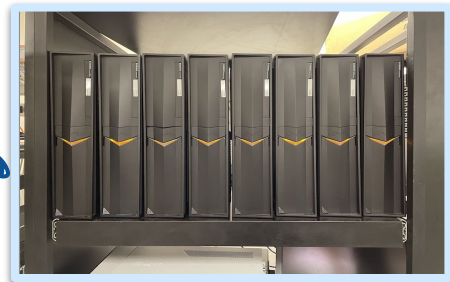
Feedback

Please feel free to submit anonymous feedback at this [form link](#)! I'm always looking for ways to improve as a TA, so this'll help me a lot: thank you so much in advance!

© Ben Yan. Oh look, I'm twinning with Kevin Wang's stunning [section website](#) here



Getting Setup for Lab 1



- ❑ To get started, log into the **myth machines**, and clone the lab starter code using the command:

```
git clone /afs/ir/class/cs107/repos/lab1/shared lab1
```

- ❑ Next, also open the **online lab check-off** linked below (or in the lab 1 handout), so you can write as you go.



[Lab Handout](#)



[Lab Check-Off Form](#)

Don't worry at all if you fall behind on the write up! I'll aim to go over everything, though just see me after lab if anything is still left



Lecture Recap!



Binary

Note: Slide skipped during lab, but may be a helpful reference!

- Decimal (standard) is a base-10 number system, i.e. $0, 1, 2, \dots, 9$

1	2	3	4	
1000	100	10	1	
10^3	10^2	10^1	10^0	

$$1234 = 1 * 10^3 + 2 * 10^2 + 3 * 10^1 + 4 * 10^0$$

- Binary is a base-2 number system i.e. 0s and 1s

1	1	0	1	
8	4	2	1	
2^3	2^2	2^1	2^0	

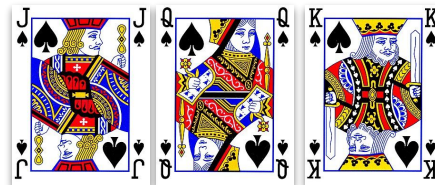
$$13 = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$$

- In general, a base- n number system has digits $0, 1, 2, \dots, n-1$; each digit has a “weight” n times greater than the digit to the right

Note: Slide skipped during lab, but may be a helpful reference!

Hexadecimal

- Hexadecimal is a base-16 number system, i.e. 0, 1, 2, ..., 9 and also A (10), B (11), C (12), D (13), E (14), F (15)
- As an analogue, think of *playing cards*, with values extending beyond 9 to Jack (10), Queen (11), King (12)



0x A 2 4
 16^2 16^1 16^0

$$2596 = 10 (A) * 16^2 + 2 * 16^1 + 4 * 16^0$$

- To toggle between binary (base 2) and hexadecimal (base 2^4), demarcate the bits in groups of 4 **starting from the right**—each group is one hexadecimal digit

011101001111
0x 7 4 F

0x 1 2 A
000100101010

Note: Slide skipped during lab, but may be a helpful reference!

Unsigned vs Signed

- ❑ C offers integers in two flavors: **signed & unsigned**
- ❑ Signed integers use the **two's complements** system
 - **Positive numbers:** just their binary representation
 - **Negative numbers:** (1) Take the binary of the positive equivalent, (2) invert each bit, (3) and add 1
 - 0 is all zero bits 🥰

Note that **most significant bit** indicates sign (0 for +, 1 for -)

(1) 0101 +5

(2) 1010

(3) 1011 -5

- ❑ **When casting from signed to unsigned, the underlying bits don't change** – just the way they're interpreted (two's complement ↔ ordinary binary)
- ❑ When comparing signed and unsigned ints, C implicitly casts signed to unsigned

Signed	-1	11111111	0xff
Unsigned	255	11111111	0xff



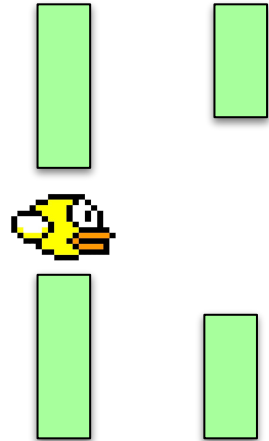
Binary and Bit Vectors

Goal: To think about binary representations as a kind of **array data structure** – with each bit storing true/false information

Example: In *Flappy Bird*, we can represent a vertical barrier as a 64-bit number / vector: **0 bit** for opening, **1 bit** otherwise

1	1	1	0	...	0	1	1	1
---	---	---	---	-----	---	---	---	---

`long barrier = 0b1110...0111`



Binary and Bit Vectors

Goal: To think about binary representations as a kind of **array data structure** – with each bit storing true/false information

Example: In *Flappy Bird*, we can represent a vertical barrier as a **64-bit number** / vector: **0 bit** for opening, **1 bit** otherwise

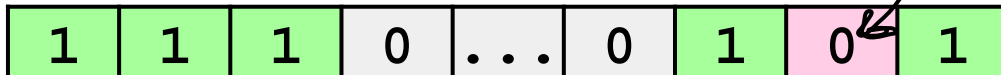


`long barrier = 0b1110...0111`

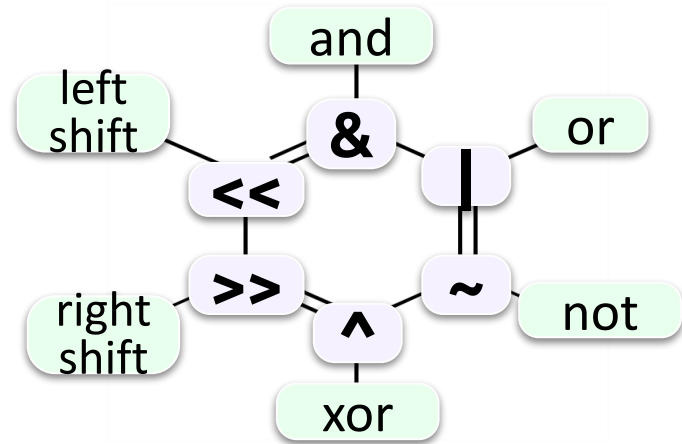
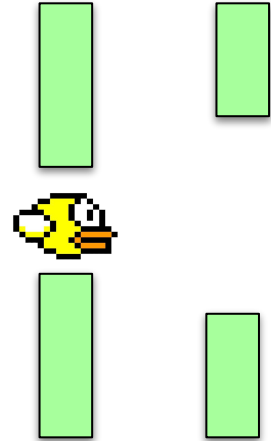
Task: How to **modify / access** the data within?

👉 **6 key bitwise operators** — shown on the right!

🧠 Bitmasks are useful for **manipulating bits** (e.g., getting the lowest bit, flipping a specific bit OFF)



Visually, this would be like opening a new hole in the barrier





Bit Operators: AND, OR, XOR, NOT

- ❑ AND, OR, XOR are binary operators (2 bits as input), NOT is unary (1 bit as input)
- ❑ When applied on numbers (multiple bits), **applied to corresponding bits** in each

AND

1	1	0	0
---	---	---	---

&

1	0	1	0
---	---	---	---

OR

1	1	0	0
---	---	---	---

|

1	0	1	0
---	---	---	---

XOR

1	1	0	0
---	---	---	---

^

1	0	1	0
---	---	---	---

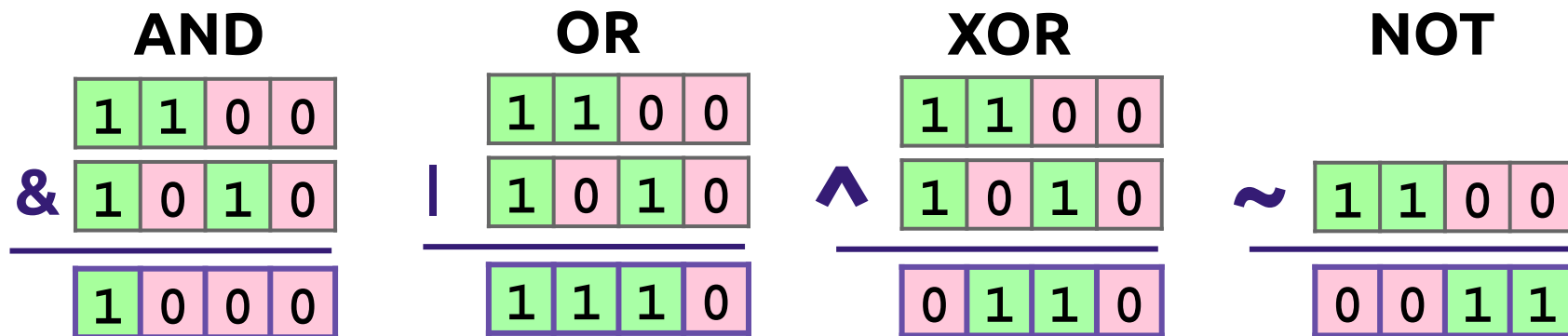
NOT

~

1	1	0	0
---	---	---	---

Bit Operators: AND, OR, XOR, NOT

- ❑ AND, OR, XOR are binary operators (2 bits as input), NOT is unary (1 bit as input)
- ❑ When applied on numbers (multiple bits), **applied to corresponding bits** in each



- The **AND** of two bits is **1** only if both bits are **1**, and otherwise **0**.
- The **OR** of two bits is **1** if either bit (or both) is a **1**, and otherwise **0**.
- The **XOR** (exclusive OR) of two bits is **1** if exactly one of the bits is a **1**, and **0** if both bits or neither bit is a **1**.
- The **NOT** of a bit is its **exact opposite**: **1** if it's a **0**, and **0** if the bit's a **1**.

Bit Operators Example: Watching TV!



Say we have a bit vector storing which shows you've watched recently.

OR operator



my_shows

today_watched

	Pokemon	Squid Game	Bojack Horseman	Sever- ance	Demon Slayer	Jujutsu Kaisen	Ted Lasso	Heart stopper
my_shows	0	1	0	0	1	0	0	0
today_watched	0	0	1	1	0	0	0	0

Bit Operators Example: Watching TV!



Say we have a bit vector storing which shows you've watched recently.

OR operator

	Pokemon	Squid Game	Bojack Horseman	Severance	Demon Slayer	Jujutsu Kaisen	Ted Lasso	Heart stopper
my_shows	0	1	0	0	1	0	0	0
today_watched	0	0	1	1	0	0	0	0

my_shows (Updated)	0	1	1	1	1	0	0	0
---------------------------	---	---	---	---	---	---	---	---

Note this updates the bit vector to indicate we've also watched Bojack & Severance!

Bit Operators Example: Watching TV!



Say we have a bit vector storing which shows you've watched recently.

OR operator

	Pokemon	Squid Game	Bojack Horseman	Severance	Demon Slayer	Jujutsu Kaisen	Ted Lasso	Heart stopper
my_shows	0	1	0	0	1	0	0	0
today_watched	0	0	1	1	0	0	0	0

my_shows (Updated)

0	1	1	1	1	0	0	0
---	---	---	---	---	---	---	---

Note this updates the bit vector to indicate we've also watched Bojack & Severance!

Say your friend has the **exact opposite taste** – they've watched all the shows you haven't seen, and none of the shows you have. How do we get their bit vector?

Solution: Use the NOT (~) operator, which inverts all the bits!

~my_shows	1	0	0	0	0	1	1	1
-----------	---	---	---	---	---	---	---	---



Bit Operators: Left/Right Shifting



Left shifting (<<) involves sliding a bit pattern a **certain number of positions to the left** — filling in the new lower bits with 0s, and discarding any bits shifted off the end.

Left Shift
(fill with 0s)

1 << 3 → 1000
3 new zero bits

11 << 30 → 11000...000
30 new zero bits

Right shifting (>>) is similar, but slides to the right instead. However, **filling in the higher-order bits is trickier**, as we don't want to alter the sign bit / MSB.

Solution: For signed values, fill with sign bit; for unsigned, just fill with leading 0s.

Right Shift

Unsigned Values

num
1 0 ... 0 1 0



num >> 1

0 1 0 ... 0 1

Logical (fill with **leading 0s**)

Signed Values

0 0 ... 0 1 0



0 0 0 ... 0 1

Arithmetic (fill with **sign bit**)

1 0 ... 0 1 0



1 1 0 ... 0 1



Bitmasks to Modify Bits Strategically

To **manipulate bits in a certain way**, we often construct a help value / **bitmask**, e.g.,



Turn a bit on

`bit | 1`



Invert/flip a bit

`bit ^ 1`



Turn a bit off

`bit & 0`



Retrieving the last bit

`num & 1`



Bitmasks to Modify Bits Strategically

To **manipulate bits in a certain way**, we often construct a help value / **bitmask**, e.g.,

Turn a bit on `bit | 1`

Invert/flip a bit `bit ^ 1`

Turn a bit off `bit & 0`

Retrieving the last bit `num & 1`

To see how a bitmask works, I recommend trying it with **concrete examples**, e.g.,



Task: Getting the lowest bit (LSB)

Bitmask: AND (&) with 1

num	1	...	0	1	1	0	1	1
bitmask &	0	...	0	0	0	0	0	1
result	0	...	0	0	0	0	0	1

Note that all bits are **switched off**, except for the **last one!** (preserved from original **num**)



Task: Turning on the lowest bit

Bitmask: OR (|) with 1

num	1	...	0	1	0	0	0
bitmask	0	...	0	0	0	0	1
result	1	...	0	1	0	0	1

Note that all bits are **retained**, except the **lowest bit (LSB)** we just turned on!

Any questions?

How are we feeling with course material so far?





Emacs

- **M-g M-g**: Jump to Line
- **C-s**: Search forwards. **C-r**: Search backwards
- **C-g**: Cancel/quit unfinished command
- **C-SPACE** to put a marker down, and move the cursor to select text
- **M-w**: Copy region. **C-w**: Cut region. **C-y**: Yank (Paste) region
- **C-k**: Cut ("kill") the text on the current line to the right of the cursor
- **Ctl-x u**: Undo the last action
- **C-x C-s**: Save file
- **C-x C-c**: Exit Emacs
- **C-h t**: Tutorial

Check out the [emacs guide](#) on the course website!



Unix Shortcuts

Ctrl + A: move to beginning of line

Ctrl + E: move to end of line

Ctrl + R: search for previously used commands

Ctrl + U: erases the current line

Ctrl + C: cancels the currently running command

tab for autocomplete

Up ↑ / down ↓ arrows to scroll through previous commands

Check out the [Unix guide](#) on the course website!



Exercise 1: Round Up!

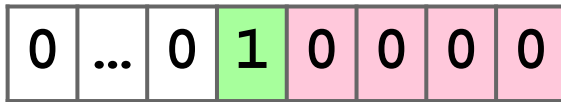
In the **lab1** directory, examine the **round.c** file, and review the **is_power_of_two** and **round_up** functions as outlined in the lab handout. How do they work?

Constellate into groups, and move into the next exercise if you finish early. Please lmk if you have any questions!

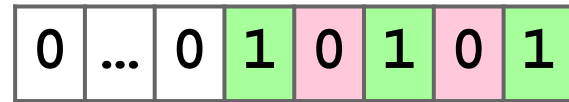


~ 20 min + 10 min all-lab discussion

Tip: Try working with small example numbers / bit representations on paper!



e.g., power of two



e.g., **not** power of two



Q1: is_power_of_2

```
bool is_power_of_2(unsigned int num) {  
    return num != 0 && (num & (num - 1)) == 0;  
}
```



Q1: is_power_of_2

```
bool is_power_of_2(unsigned int num) {  
    return num != 0 && (num & (num - 1)) == 0;  
}
```

num is a **power of two** (e.g., 16)

num	0	...	0	1	0	0	0	0
num - 1	0	...	0	0	1	1	1	1
num & (num - 1)	0	...	0	0	0	0	0	0

All the bits between **num** and **num - 1** cancel each other out!



Q1: is_power_of_2

```
bool is_power_of_2(unsigned int num) {  
    return num != 0 && (num & (num - 1)) == 0;  
}
```

num is a **power of two** (e.g., 16)

num	0	...	0	1	0	0	0	0
num - 1	0	...	0	0	1	1	1	1

num & (num - 1)	0	...	0	0	0	0	0	0
--------------------	---	-----	---	---	---	---	---	---

All the bits between **num** and **num - 1** cancel each other out!

num is **NOT** a power of two (e.g., 18)

num	0	...	0	1	0	0	1	0
num - 1	0	...	0	1	0	0	0	1

num & (num - 1)	0	...	0	1	0	0	0	0
--------------------	---	-----	---	---	---	---	---	---

The bits don't fully cancel each other out – there's leftover 1 bits!



Q2: round_up (not necessarily power of 2)

`((value + multiple - 1) / multiple) * multiple;` // multiple may be non-power of 2

The key is **integer division!**

Example: value = 20, multiple = 6





Q2: round_up (not necessarily power of 2)

$((\text{value} + \text{multiple} - 1) / \text{multiple}) * \text{multiple};$ // multiple may be non-power of 2

The key is integer division!

Example: value = 20, multiple = 6



1



overshoot

value + mult - 1

25

We first get a quantity that **exceeds the next multiple** above value

2



come back to rounded answer

value + mult - 1

mult

* mult

As integer division “rounds down”, these gets **how many full multiples** the quantity contains, e.g., $25 / 6 = 4$

We then multiply by **mult** again to get the rounded value, e.g., we have **4 full multiples of 6**, so $4 * 6 = 24!$



Q2: round_up (power of 2)

What makes **powers of 2** special so that we can “correct” without integer division?

```
(value + multiple - 1) & ~(multiple - 1); // multiple is power of 2, e.g., 2,4,8,...
```

Example: value = 20, multiple = 8





Q2: round_up (power of 2)

What makes **powers of 2** special so that we can “correct” without integer division?

```
(value + multiple - 1) & ~(multiple - 1); // multiple is power of 2, e.g., 2,4,8,...
```

Example: value = 20, multiple = 8



24

1



overshoot

value + mult - 1

27

Same as before, we first get a quantity that **exceeds the next multiple**

2



come back to rounded answer

(value + mult - 1) & ~(mult - 1);

How does this work now?
Let's think in terms of bits :)

Recall from
is_power_of_2

mult

0	...	0	1	0	0	0
---	-----	---	---	---	---	---

mult - 1

0	...	0	0	1	1	1
---	-----	---	---	---	---	---



~(mult - 1)

1	...	1	1	0	0	0
---	-----	---	---	---	---	---

This is our bitmask!



Check-Off Q3: Rounding

$(\text{value} + \text{multiple} - 1) \& \sim(\text{multiple} - 1);$ // multiple is power of 2, e.g., 2,4,8,...

Example: value = 20, multiple = 8

 24

value + mult - 1

0	...	0	1	1	0	1	1
---	-----	---	---	---	---	---	---

27

$\sim(\text{mult} - 1)$

1	1	...	1	1	0	0	0
---	---	-----	---	---	---	---	---



Check-Off Q3: Rounding

$(\text{value} + \text{multiple} - 1) \& \sim(\text{multiple} - 1);$ // multiple is power of 2, e.g., 2,4,8,...

Example: value = 20, multiple = 8

 24

value + mult - 1

0	...	0	1	1	0	1	1
---	-----	---	---	---	---	---	---

 27

$\sim(\text{mult} - 1)$

1	1	...	1	1	0	0	0
---	---	-----	---	---	---	---	---

$(\text{value} + \text{mult} - 1) \& \sim(\text{mult} - 1)$

0	...	0	1	1	0	0	0
---	-----	---	---	---	---	---	---

 24
32 16 8 4 2 1

The bitmask here makes the quantity a multiple of mult, by zeroing out the bits lower than the one affiliated with mult.

For instance, here, the number becomes a **multiple of 8**, by only keeping the bits for 8, 16, 32, ..., and discarding lower bits (1, 2, 4).

Exercise 1 Takeaways

For **building bitmasks**, I recommend:

- ❖ Breaking it up in separate steps
- ❖ Constructing it **from bitmask(s) you already know** (e.g., -1 is all 1s)

👉 On the right, some useful ones to know!



Bitmask building blocks / LEGOs

-1	1	1	1	...	1	1	1	-1
1	0	...	0	0	0	0	1	1

Note: 1L for signed long, 1UL for unsigned long

2^n	0	...	0	1	0	0	0	$1 \ll n$
-------	---	-----	---	---	---	---	---	-----------

$2^n - 1$	0	...	0	0	1	1	1	$(1 \ll n) - 1$
-----------	---	-----	---	---	---	---	---	-----------------

Task: Turning off the lowest 4 bits simultaneously

num	1	...	0	1	1	0	1	1
bitmask &	1	...	1	1	0	0	0	0
result	1	...	0	1	0	0	0	0

So a possible bitmask is AND (&) with $(-1L \ll 4)$

How do we build this mask?

1	...	1	1	1	1	1	1
---	-----	---	---	---	---	---	---



Start with $-1L$ (above) and shift 4 bits to the left!

1	...	1	1	0	0	0	0
---	-----	---	---	---	---	---	---

Alternatively, the bitmask from prior slide!



Exercise 2: Debugging Parity

In the **lab1** directory, examine the **parity.c** file, compile it (make), and run it on various values, e.g.,

`./parity 1` `./parity 8` `./parity -5`

Here, a value is considered “odd” if it has an **odd number of 1 bits in its binary representation**, and “even” if it has an even number of 1 bits.



~ 20 min + 10 min
all-lab discussion



Here, 5 is even!

0	...	0	0	0	1	0	1
---	-----	---	---	---	---	---	---

Even number of 1 bits



Here, 4 is odd!

0	...	0	0	0	1	0	0
---	-----	---	---	---	---	---	---

Odd number of 1 bits



compute_parity()

```
int compute_parity(int x) {  
    int result;  
    while (x != 0) {  
        // invert result if lsb is on  
        result ^= (x & 1);  
        // shift bits down  
        x = x >> 1;  
    }  
    return result;  
}
```



compute_parity(13)

0 ... 0 1 1 0 1

```
int compute_parity(int x) {  
    int result;  
    while (x != 0) {  
        // invert result if lsb is on  
        result ^= (x & 1);  
        // shift bits down  
        x = x >> 1;  
    }  
    return result;  
}
```

X
0 ... 0 1 1 0 1



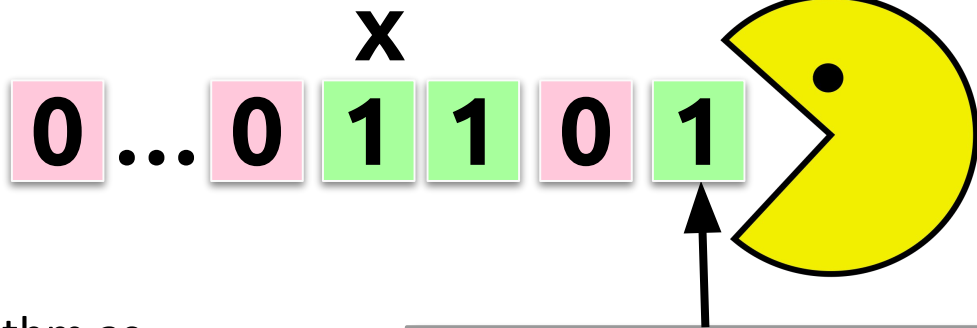
compute_parity(13)

0 ... 0 1 1 0 1

```
int compute_parity(int x) {  
    int result;  
    while (x != 0) {  
        // invert result if lsb is on  
        result ^= (x & 1);  
        // shift bits down  
        x = x >> 1;  
    }  
    return result;  
}
```

Bits eaten so far: *Nothing so far!*

1 bits eaten: 0 Result: Odd? No / 0



We use $x \& 1$ to get the current bit to eat/chomp

Hear me out lol – imagine this algorithm as **Pac-Man**: we read the bits of x one by one by right-shifting them into PacMan's mouth



compute_parity(13)

0 ... 0 1 1 0 1

```
int compute_parity(int x) {  
    int result;  
    while (x != 0) {  
        // invert result if lsb is on  
        result ^= (x & 1);  
        // shift bits down  
        x = x >> 1;  
    }  
    return result;  
}
```

Bits eaten so far:

1

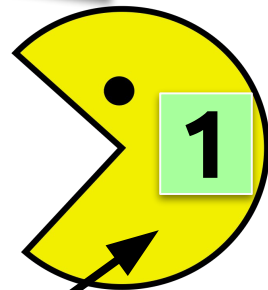
1 bits eaten:

1

Result: Odd?

Yes / 1

0 ... 0 1 1 0



We used `x & 1` to eat the 1 bit at the end—flipping **result**, as we've now eaten an **odd number** of 1 bits so far.



compute_parity(13)

0 ... 0 1 1 0 1

```
int compute_parity(int x) {  
    int result;  
    while (x != 0) {  
        // invert result if lsb is on  
        result ^= (x & 1);  
        // shift bits down  
        x = x >> 1;  
    }  
    return result;  
}
```

Bits eaten so far:

1

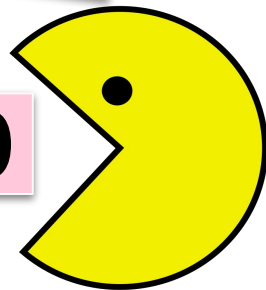
1 bits eaten:

1

Result: Odd?

Yes / 1

0 ... 0 1 1 0



We now **right-shift / shift the bits down once**, so that Pac-Man can eat the next bit!



compute_parity(13)

0 ... 0 1 1 0 1

```
int compute_parity(int x) {  
    int result;  
    while (x != 0) {  
        // invert result if lsb is on  
        result ^= (x & 1);  
        // shift bits down  
        x = x >> 1;  
    }  
    return result;  
}
```

Bits eaten so far:

0 1

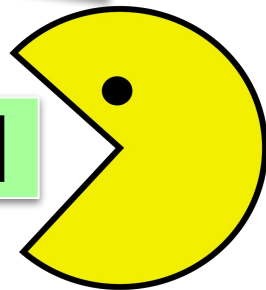
1 bits eaten:

1

Result: Odd?

Yes / 1

0 ... 0 1 1



We do the same two steps again: (1) eat lowest bit and (2) shift down—except **since it's a 0 bit, we don't flip the parity!**



compute_parity(13)

0 ... 0 1 1 0 1

```
int compute_parity(int x) {  
    int result;  
    while (x != 0) {  
        // invert result if lsb is on  
        result ^= (x & 1);  
        // shift bits down  
        x = x >> 1;  
    }  
    return result;  
}
```

Bits eaten so far:

1 0 1

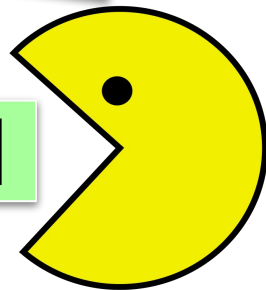
1 bits eaten:

2

Result: Odd?

No / 0

0 ... 0 1



We just ate a 1 bit – so we flip the parity again!



compute_parity(13)

0 ... 0 1 1 0 1

```
int compute_parity(int x) {  
    int result;  
    while (x != 0) {  
        // invert result if lsb is on  
        result ^= (x & 1);  
        // shift bits down  
        x = x >> 1;  
    }  
    return result;  
}
```

Bits eaten so far:

1 1 0 1

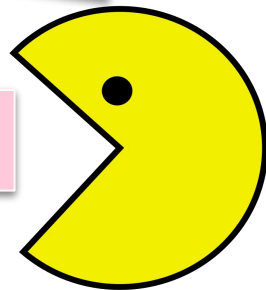
1 bits eaten:

3

Result: Odd?

Yes / 1

0 ... 0



And again! Just ate a 1 bit – so we flip the parity back!



compute_parity(13)

0 ... 0 1 1 0 1

```
int compute_parity(int x) {  
    int result;  
    while (x != 0) {  
        // invert result if lsb is on  
        result ^= (x & 1);  
        // shift bits down  
        x = x >> 1;  
    }  
    return result;  
}
```

Bits eaten so far:

1 1 0 1

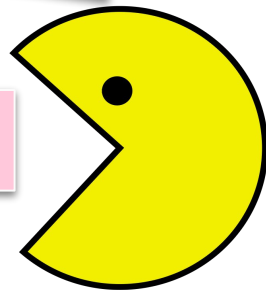
1 bits eaten:

3

Result: Odd?

Yes / 1

0 0 0 0 0 ... 0



Recall when we right-shift, we lose bits—so >> has to also fill in bits on the left!

Now, the loop will terminate, as **x is now 0 (all 1 bits eaten)**. Our final result is that **input 13 was odd!** (3 ON bits)



Infinite Loop with Negative Inputs



! However, issues can arise if the left side doesn't fill with 0 bits!, e.g.,

1 1 1 0 ... 0

Let's explore in **GDB**!



```
bbyan@myth: make
bbyan@myth: gdb parity
```

```
(gdb) run -2
```

```
Starting program ...
```

Press **CTRL-C** to interrupt

```
Program received signal SIGINT
compute_parity(x=-2) at parity.c: 14
14      while (x != 0){
```

Make sure to compile with **make** first, then to launch gdb, run **`gdb [program name]`**

To run your program in GDB, type **`run [inputs]`**, e.g., this here will run **`./parity -2`**

OH NO, infinite loop! let's interrupt the program to see where it got stuck

Neat – we know it got stuck in this while loop, let's investigate further!

Check-Off Q4: Backtrace – where are we?



(gdb) **backtrace**

```
#0 compute_parity(x=-2) at parity.c:14
#1 main(argc=2,argv=...) at parity.c: 36
```

(gdb) **list**

```
14  while (x != 0){
15      // invert result if lsb is on
16      result ^= (x & 1);
18      // shift bits down
19  x = x >> 1;
...
```

(gdb) **break 19**

backtrace prints out a stack trace for where we currently are in the program

here, we're in the **compute_parity** helper function, which is being called by **main** currently

list prints 10 lines of code around where we are – to print more lines, just run **list** again!

Let's put a **breakpoint**  on line 19 (where x changes inside loop); GDB knows to stop there so we can investigate



Also, let's print out x before we continue!
/t will format it in binary (/x for hex)

Interesting, this is **-1** (all 1 bits)

We let the program **continue** running until a breakpoint – **how will x change?** (*if you instead want to run 1 line at a time, use **next***)

All 1 bits again!! Looks like >> fills with 1 bits on the left. One more time ...

UH OH! **It's cooked: x is still -1 (all 1 bits)**. This is a problem as the while loop only ends when $x = 0$ (no 1 bits left)

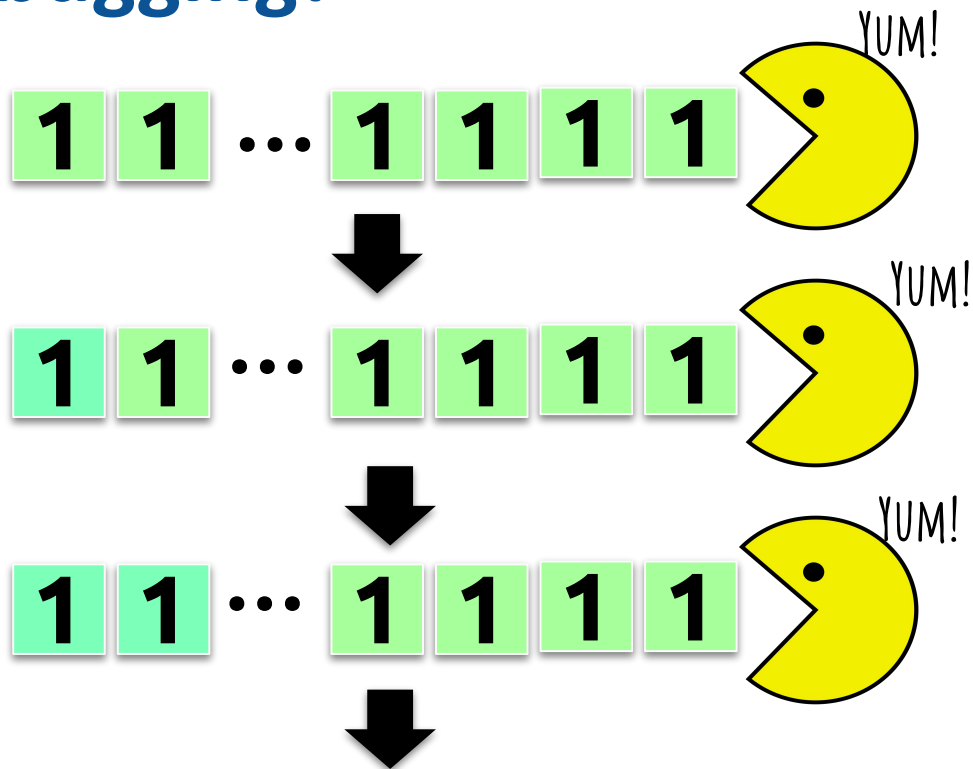
(gdb) continue

```
(gdb) print /t x
```

1111111111111111111111111111111111111111111111111



Check-Off Q5: Debugging!



**OH-NO! Pac-Man will
never stop eating 1 bits!**



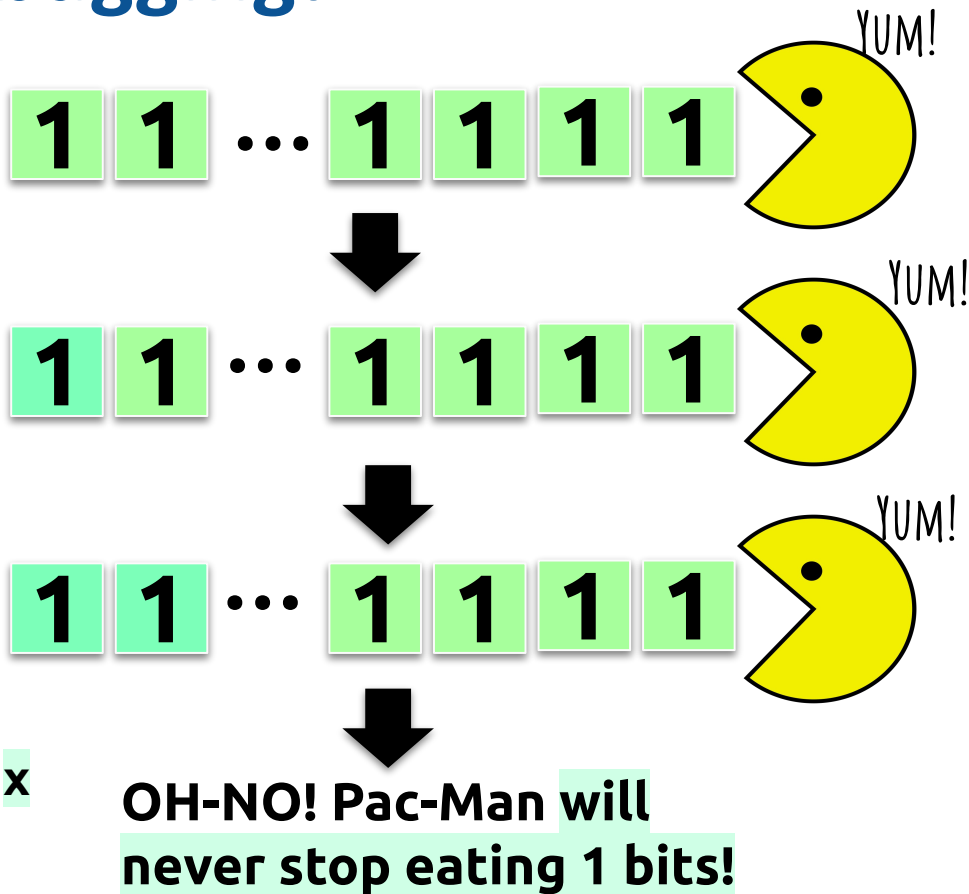
Check-Off Q5: Debugging!

Note that right-shift for signed ints fills with **signed bit**—when x is negative, this is a 1 bit.

So the **left side always replenishes with 1 bits**, causing x to never run out of them—loop never ends.

```
int compute_parity(int x) {  
    int result;  
    while (x != 0) {  
        ....  
    }  
}
```

We found this in **GDB** by noticing that x **stays as -1** due to the shift problem!





Infinite Loop Solution – Fixing the Bug :)

A solution is to **change x to be an unsigned** type – so that right-shifting **always fills with 0s!**

Also, we should **explicitly initialize result to 0** (even).



```
int compute_parity(unsigned int x) {  
    int result = 0;  
    while (x != 0) {  
        ....  
    }  
}
```

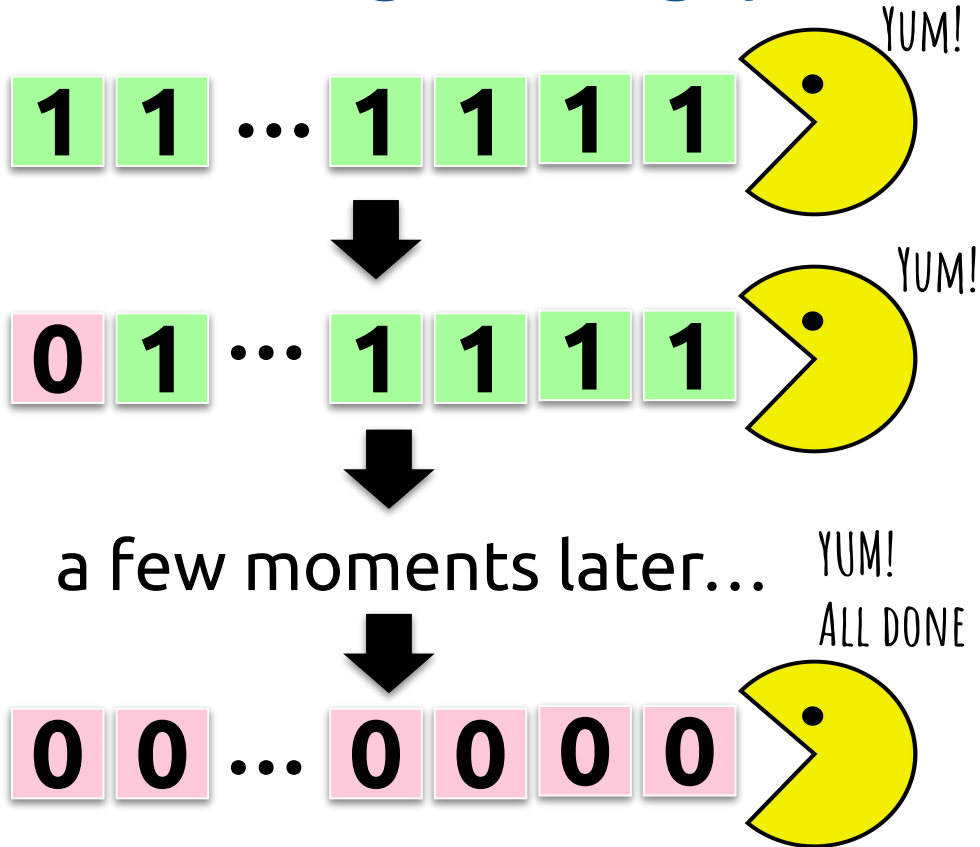


Infinite Loop Solution – Fixing the Bug :)

A solution is to **change x to be an unsigned** type – so that right-shifting **always fills with 0s!**

Also, we should **explicitly initialize result to 0** (even).

```
int compute_parity(unsigned int x) {  
    int result = 0;  
    while (x != 0) {  
        ....  
    }  
}
```





Takeaway: GDB is Your Friend in 107

Don't know how to really debug rn besides using print statements?

No worries! ~~Neither do I.~~ We'll learn together!

- **GDB** is the command-line debugger that we use in CS 107. **It's an absolute game-changer!** Using this tool, we can:

Pause a program at specific points

Step through execution

Inspect variable values

print a stack trace of program you're in

See where a crash / segfault happened

```
((gdb) list compute_parity
6      #include <error.h>
7      #include <stdio.h>
8      #include <stdlib.h>
9
10
11      int compute_parity(int x) {
12          int result;
13
14          while (x != 0) {
15              // invert result if lsb is on
((gdb) break 11
Breakpoint 1 at 0x1160: file parity.c, line 14.
((gdb) run 5
Starting program: /afs/.ir.stanford.edu/users/b/b/

Breakpoint 1, compute_parity (x=5) at parity.c:14
14      while (x != 0) {
((gdb) print result
$1 = 32767
```



GDB Guide

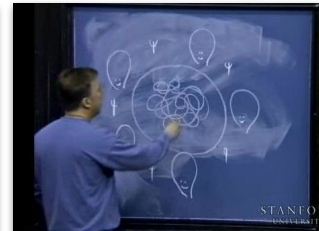
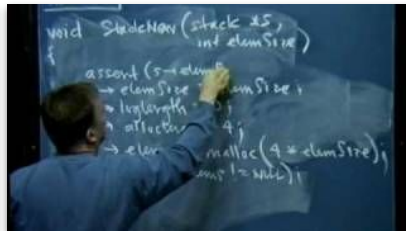
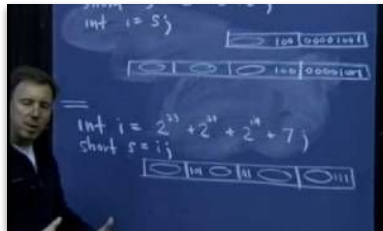
Helpful [debugging guide](#)
on the website!

Tip: You can use `p/t` and `p/x` to print in binary and hexadecimal formats, respectively!

Command	Description
<code>break [x]</code>	Put a breakpoint on line <code>x</code> or function <code>x</code> .
<code>run</code> <code>[args]</code>	Runs the current program in gdb with the specified command-line arguments.
<code>print [x]</code>	Print the value of a variable or expression.
<code>next</code>	Step to the next program line and <i>completely run any function(s) on that line.</i>
<code>step</code>	Step to the next program line, <i>or into the first function called by that line.</i>
<code>continue</code>	Continue running the program until the next breakpoint or until the program ends.
<code>backtrace</code>	Print a stack trace for where in the program you currently are.
<code>up</code> and <code>down</code>	Go up and down the stack trace to change your current view and what variables you can print. E.g. <code>up</code> places the program in the calling function, and then you can use <code>print</code> to look at the state of the program in that function.
<code>quit</code>	Quit <code>gdb</code>



Anonymous feedback form, appreciate any lab/teaching feedback! 



Have an amazing week, and welcome to 107!

- Please reach out to me if you have any questions, snack recommendations, or topics you want covered!
-  Contact: bbyan@stanford.edu, email or over Slack
- Also, it's not a bad time to exchange contact info and find study partners or groups [#powerOfFriendship](#)



Looking forward to a beautiful quarter in Jerryland!