



CS107 Midterm Review Session

★ Keep up the amazing work everyone!

Ben Yan, Winter 2026 ❄️

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



🎵 My Song of the Week: Golden from Kpop Demon Hunters

Stanford | ENGINEERING
Computer Science

👋 Awesome to meet you if we haven't already!!

Hi I'm Ben (bbyan@stanford.edu), elated to be TAing CS107 again :)

- ★ **CS co-term!** 🦖 For undergrad, ~~chaos~~ double majored in CS and Math, minored in Creative Writing (took 12 English classes), also studied poetry abroad at Oxford
- ★ **Enemies to lovers relationship with CS107 (and CS here in general lol)**, sort of
- ★ So blessed to have such amazing students + fellow TAs, and Jerry as my advisor!
- ★ TA for CS106A, CS106AX, CS107; co-taught a 1-unit wonder CS106S with Jerry

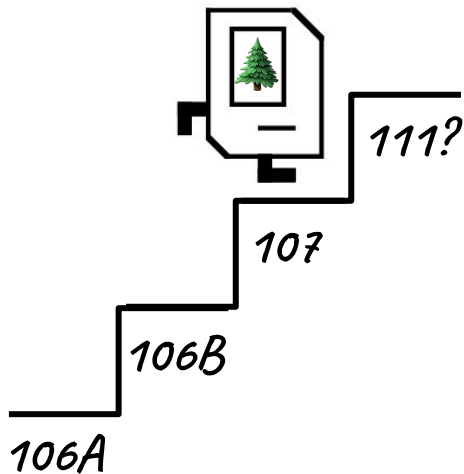


Hope that you're having a fun time in Jerryland this winter!



The Map For Today

- 1 Quick overview of midterm structure, logistics, what you can expect, resources
- 2 **Recap of course material** (up to Lecture Deck #13), from Bits/Bytes to Generics
- 3 Walk through **practice problems and fun coding exercises** together 💖
- 4 Happy to take questions along the way!
I will throw you a beautiful mochi





Midterm Logistics and Topics Coverage



When/where is the midterm?

- ❑ Time: **Wed, Feb 11th, 3:30 - 5:30pm (Hewlett 201)** or **7 – 9 PM (CEMEX)**
- ❑ Two hour, paper and pencil exam
- ❑ Closed notes, reference sheet provided



What are Jerry exams like?

- ❑ Jerry approaches exam writing as **literature and art**. It's storytelling :)
- ❑ Practice exams are key, and especially ones with **poohbear** as the writer.



What's covered on the midterm? What should I brush up on?

Binary, Data Representation

Bits and Bytes, ASCII, signed/unsigned, two's complements integers

Strings, Pointers

C strings (why char*), string functions, pointers, buffers and structs

Stack/Heap

Stack arrays, heap memory (e.g., malloc, realloc, free), memory errors

Generics

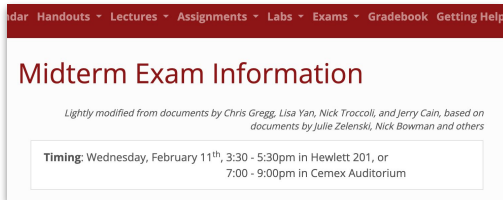
void* pointers, memcpy/memmove, bsearch/qsort, cmp functions



Midterm Study Resources



Midterm Info Page: Also includes a string quartet of previous CS107 exams!



Panopto Course Videos: Aka Jerry's reality TV show. Guest star Kate in W3!



My Lab Website: Shameless plug, but I post all my lab materials + review slides at stanford.edu/~bbyan/cs107 :)



Lab 4

Void Star, Function Pointers, Generics — Feb 4-5

Slides

Slides (PDF)

Lab Handout

Check-Off Form

Lecture Recap

Practice Midterms

Prep

★ Extra Midterm Prep Materials

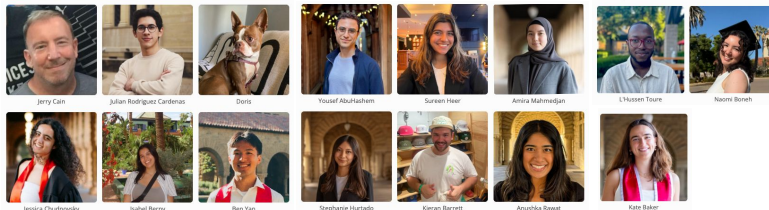
Slides (TBA)

Ben's Practice Problems

Practice Solutions



Office Hours + EdStem Forum: We're always around if you have questions!



Let's Jump Right Into the Fun!



CS107 Scenic Recap: Jerry Cain High-Speed Train



**Bits, Bytes,
Integers!**

Exercise: Power
of Friendship



**C Strings &
Methods**

Exercise: Life
Comes Full Circle



**Pointers,
Arrays, Heap**

Exercise:
Malevolent Heap



**Generics,
Void Stars**

Exercise: Yeet
That Element

Let's Jump Right Into the Fun!



CS107 Scenic Recap: Jerry Cain High-Speed Train



**Bits, Bytes,
Integers!**

**Exercise: Power
of Friendship**



**C Strings &
Methods**

Exercise: Life
Comes Full Circle



**Pointers,
Arrays, Heap**

Exercise:
Malevolent Heap



**Generics,
Void Stars**

Exercise: Yeet
That Element



Before we get onboard, do we have any questions?



Binary Representation of Integer Values

All values in C are represented in memory as **binary**: a sequence of 0/1 bits denoting the powers of 2 that make up a number. Decimal is our own lens of interpretation.

$$7 = 2^2 + 2^1 + 2^0 \quad \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline \end{array} \\ 2^2 \quad 2^1 \quad 2^0$$

$$10 = 2^3 + 2^1 \quad \begin{array}{|c|c|c|c|} \hline 1 & 0 & 1 & 0 \\ \hline \end{array} \\ 2^3 \quad 2^2 \quad 2^1 \quad 2^0$$

C offers integers in two flavors: **signed and unsigned**. Signed ints use the **2C (two's complement's) representation**: the highest bit / MSB is 1 if negative, 0 if positive.

➤ For 2C, I think about MSB as having  negative weight, e.g., -2^7 for a 8-bit char.

CHAR_MIN
(-2^7 , -128)

$$\begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline \end{array} \\ -2^7 \quad 2^6 \quad 2^5 \quad 2^4 \quad 2^3 \quad 2^2 \quad 2^1 \quad 2^0$$

-1

$$\begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \hline \end{array} \\ -2^7 \quad 2^6 \quad 2^5 \quad 2^4 \quad 2^3 \quad 2^2 \quad 2^1 \quad 2^0$$

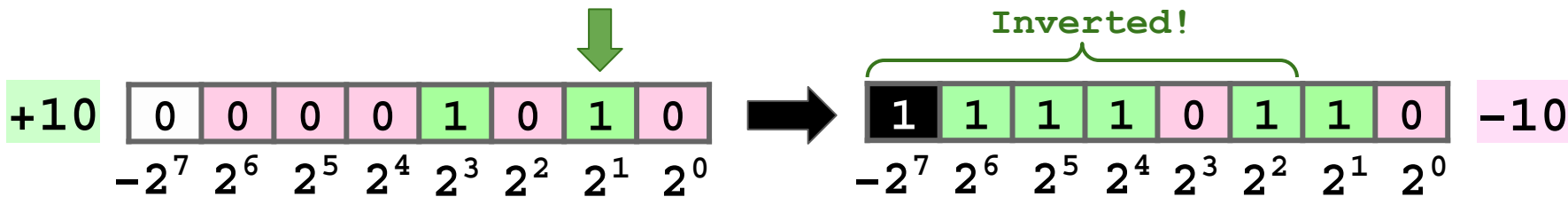
MSB contributes immense negativity, and no positive 1 bits to lift its spirits up 🙄

MSB contributes much negativity, but just enough positivity to stay nearly afloat 🙄



Binary Representation of Integer Values

2C Hack: To convert binary of positive integer to binary of its negative equivalent, (1) find the rightmost 1 bit, and (2) invert all bits to its left. Don't forget leading 0 bits!



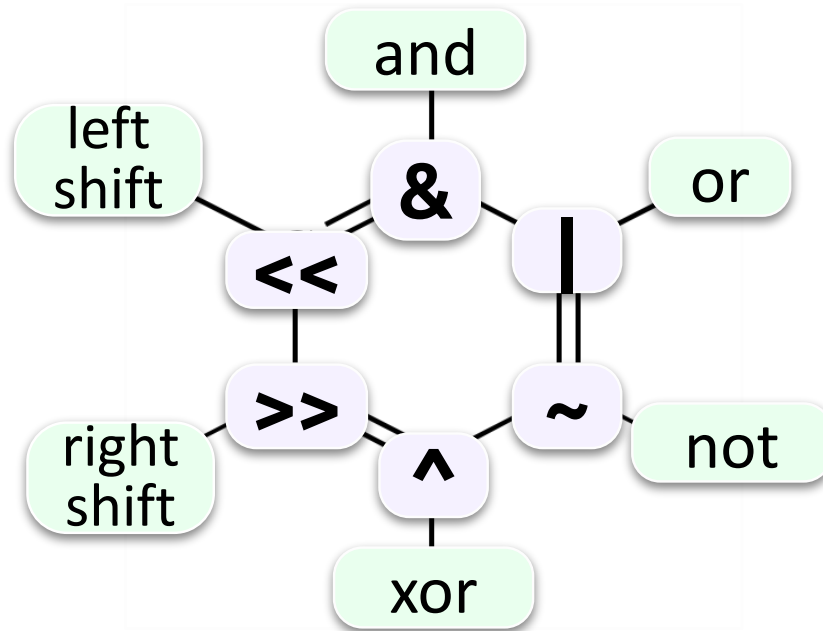
- ❑ When casting from signed to unsigned, the underlying bits don't change – just the way they're interpreted (two's complement $\leftrightarrow$ ordinary binary)
- ❑ When comparing signed and unsigned ints, C implicitly casts signed to unsigned

Signed	-1	1 1 1 1 1 1 1 1	0xff
Unsigned	255	1 1 1 1 1 1 1 1	0xff



Bit Ops: Performing Operations on Bits

We have 6 key bitwise operators! Let's review what each of them are :)





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
	1	0	0	0

→ 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**.

OR

	1	1	0	0
	1	0	1	0
	1	1	1	0

XOR

	1	1	0	0
^	1	0	1	0
	0	1	1	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**.

NOT

	1	1	0	0
~	1	1	0	0
	0	0	1	1



Bit Operators: << and >> 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 left end.

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.

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 1 0 ... 0 1 0

↓ ↓

0 0 0 ... 0 1 1 1 0 ... 0 1

Arithmetic (fill with **sign bit**)



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!



Bitmask Techniques

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).
- May need to shift the input itself!

👉 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
---	-----	---	---	---	---	---	---

Happy to take questions!

Bits, Bytes, Integers, Bit Ops, Bitmasks, etc.

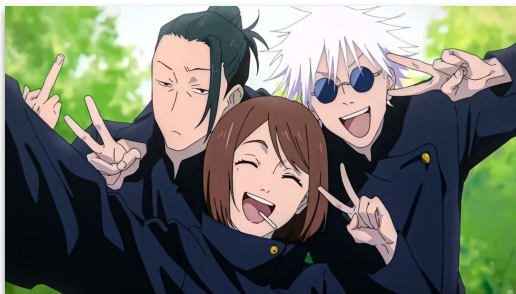




Bits and Bytes Exercise: Power of Friendship!

Implement the function `hasBitFriends(int num)`, which returns:

- `true` if the binary representation of `num` has a pair of consecutive 1 bits
- `false` otherwise, i.e. all 1 bits are lonely / spaced apart :(



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

has friends?



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

has friends?



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

has friends?

Anime Trivia: The image on the right is a breakup scene That KFC is now famous



Bits and Bytes Exercise: Power of Friendship!

Implement the function `hasBitFriends(int num)`, which returns:

- `true` if the binary representation of `num` has a **pair of consecutive 1 bits**
- `false` otherwise, i.e. all 1 bits are lonely / spaced apart :(

Examples: Toy inputs with a smaller bit count (8 bits) for simplicity!

num	<table border="1"><tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr></table>	1	0	1	0	1	0	1	0	has friends? 
1	0	1	0	1	0	1	0			

num	<table border="1"><tr><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td></tr></table>	0	1	0	1	0	1	0	0	has friends? 
0	1	0	1	0	1	0	0			

num	<table border="1"><tr><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td></tr></table>	0	1	0	1	1	0	1	0	has friends? 
0	1	0	1	1	0	1	0			

num	<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	1	1	1	1	1	1	1	has friends? 
1	1	1	1	1	1	1	1			



Power of Friendship: Approach #1 (Loop)

- ❑ One approach is to **check each pair of bits iteratively**, i.e. use a loop.
- ❑ We can **AND (&) each pair** with **bitmask 11** to check if they're both 1s!

Let's diagram out how to check the  i th and $(i+1)$ th bits from the left!

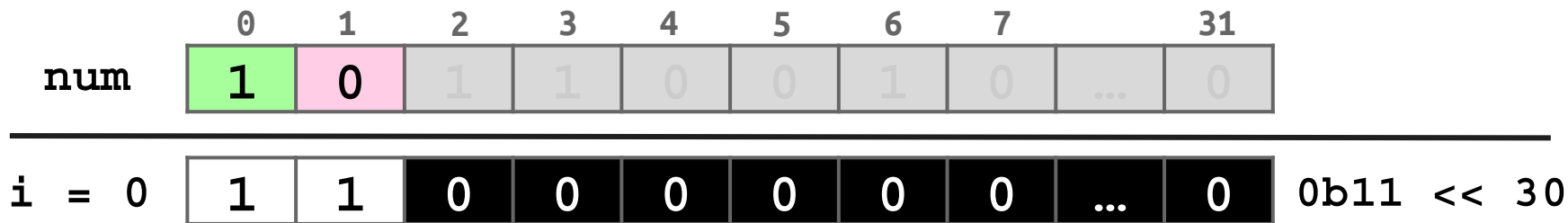
	0	1	2	3	4	5	6	7		31
num	1	0	1	1	0	0	1	0	...	0



Power of Friendship: Approach #1 (Loop)

- ❑ One approach is to **check each pair of bits iteratively**, i.e. use a loop.
- ❑ We can **AND (&) each pair** with **bitmask 11** to check if they're both 1s!

Let's diagram out how to check the  i th and $(i+1)$ th bits from the left!

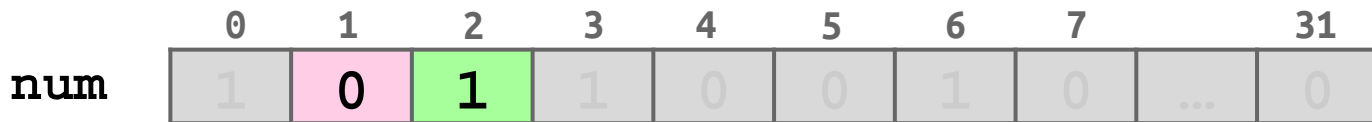




Power of Friendship: Approach #1 (Loop)

- ❑ One approach is to **check each pair of bits iteratively**, i.e. use a loop.
- ❑ We can **AND (&) each pair** with **bitmask 11** to check if they're both 1s!

Let's diagram out how to check the  i th and $(i+1)$ th bits from the left!



$i = 1$

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

`0b11 << 29`



Power of Friendship: Approach #1 (Loop)

- ❑ One approach is to **check each pair of bits iteratively**, i.e. use a loop.
- ❑ We can **AND (&) each pair** with **bitmask 11** to check if they're both 1s!

Let's diagram out how to check the  i th and $(i+1)$ th bits from the left!

	0	1	2	3	4	5	6	7		31
num	1	0	1	1	0	0	1	0	...	0

$i = 2$

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

`0b11 << 28`



Power of Friendship: Approach #1 (Loop)

- ❑ One approach is to **check each pair of bits iteratively**, i.e. use a loop.
- ❑ We can **AND (&) each pair** with **bitmask 11** to check if they're both 1s!

Let's diagram out how to check the  i th and $(i+1)$ th bits from the left!

	0	1	2	3	4	5	6	7	...	31	
num	1	0	1	1	0	0	1	0	...	0	
i = 0	1	1	0	0	0	0	0	0	...	0	0b11 << 30
i = 1	0	1	1	0	0	0	0	0	...	0	0b11 << 29
i = 2	0	0	1	1	0	0	0	0	...	0	0b11 << 28

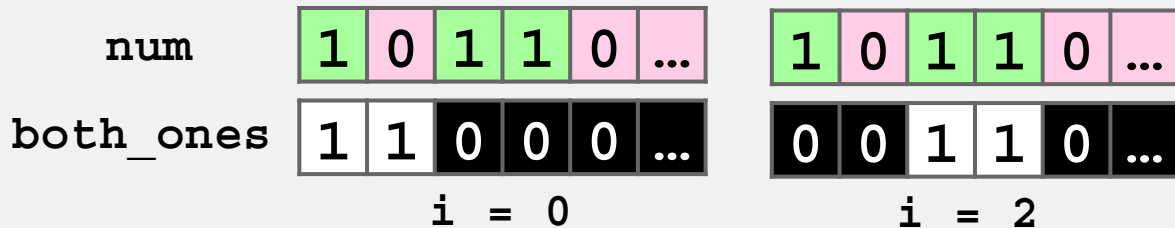
In general, & with mask $0b11 \ll (30 - i)$. If result is same as mask, bit friends!



Power of Friendship: Approach #1 (Loop)

Implement `hasBitFriends(int num)`, which returns `true` if the binary representation of `num` has a pair of consecutive 1 bits, and `false` otherwise,

```
bool hasBitFriends(int num){  
    // checks pairs of neighboring bits starting from left  
    for (int i = 0; i < 31; i++){ // 31 pairs for 32 bits  
        // checks if bits i and i + 1 are both 1 bits  
        int both_ones = 0b11 << (30 - i);  
        if (num & both_ones == both_ones){  
            return true;  
        }  
    }  
    return false;  
}
```





Power of Friendship: Approach #1 (Loop)

Implement `hasBitFriends(int num)`, which returns `true` if the binary representation of `num` has a pair of consecutive 1 bits, and `false` otherwise,

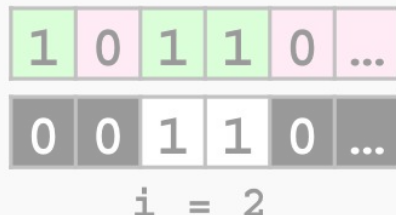
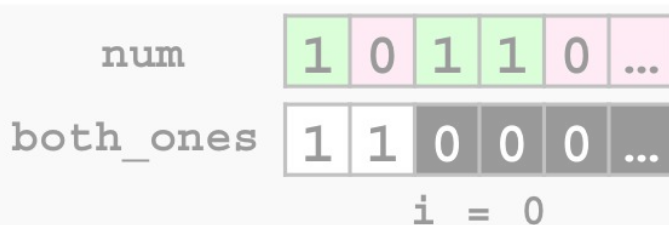
```
bool hasBitFriends(int num){
```

```
// checks each pair of bits
```

Problem: This looping approach **effectively requires 31 different bitmasks** (one to check each pair of bits) in order to work. Yikes.



```
    both_ones = 0b11 << (31 - i);  
    if (num & both_ones == both_ones){  
        return true;  
    }  
}  
return false;  
}
```





Power of Friendship: Approach #1 (Loop)

Implement `hasBitFriends(int num)`, which returns `true` if the binary representation of `num` has a pair of consecutive 1 bits, and `false` otherwise,

```
bool hasBitFriends(int num){
```

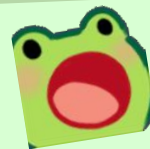
```
// checks each pair of bits
```

Problem: This looping approach **effectively requires 31 different bitmasks** (one to check each pair of bits) in order to work. Yikes.



```
    both_ones = 0b11 << (31 - i);  
    if (num & both_ones == both_ones){  
        return true;  
    }  
}
```

Can we check more efficiently? (i.e. without using any loop at all!)



How might we construct a **single bitmask** that can simultaneously **monitor all pairs of bits** in the binary, and deduce if a pair has both 1 bits?

`i = 0`

`i = 2`



Power of Friendship: Approach #2 (Musical Chairs)



Key: We can use the input binary (seating chart of bits) as part of our bitmask!





Power of Friendship: Approach #2 (Musical Chairs)

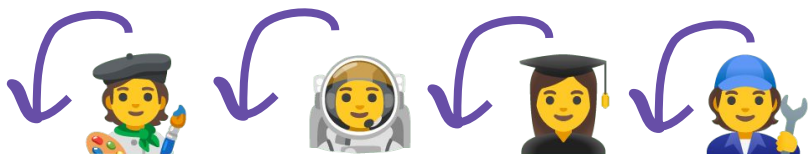


Key: We can use the input binary (seating chart of bits) as part of our bitmask!



Observation: When people sit apart from each other, anyone could move over one chair to the left, and be in an open seat. No overlap wahoo!

ON CALL
LATE NIGHT





Power of Friendship: Approach #2 (Musical Chairs)

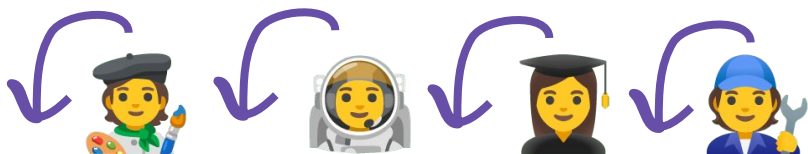


Key: We can use the input binary (seating chart of bits) as part of our bitmask!



Observation: When people sit apart from each other, anyone could move over one chair to the left, and be in an open seat. **No overlap wahoo!**

ON CALL
LATE NIGHT



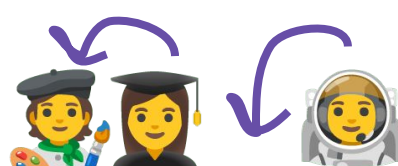
This isn't the case if **two bit friends sit next to each other**.

They would collide/overlap! 💥

COLLISION!



COLLISION!





Power of Friendship: Approach #2 (Musical Chairs)



Key: We can use the input binary (seating chart of bits) as part of our bitmask!

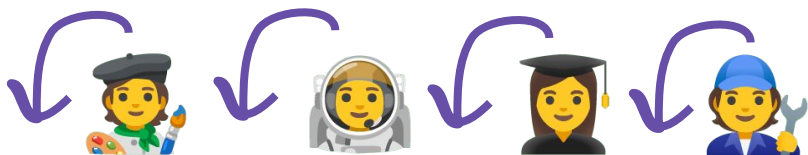


Observation: When people sit apart from each other, anyone could move over one chair to the left, and be in an open seat. **No overlap wahoo!**



This isn't the case if **two bit friends sit next to each other**.

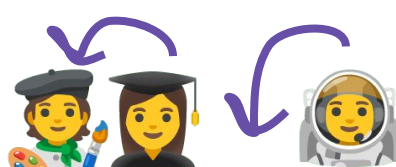
They would collide/overlap! 💥



COLLISION!



COLLISION!



Idea: What if we move the input bits around? We can have them all play musical chairs, i.e. shift the bits to the left by one, and tune in for collisions! 💥



Power of Friendship: Approach #2 (Musical Chairs)

- ❑ **Idea: What if we move the input bits around?** We can have them all play musical chairs, i.e. **shift the bits to the left by one**, and check for a collision!

If num has a pair of 1 bit friends

num

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

num << 1

num &
(num<<1)

If num has no pairs of 1 bit friends

num

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

num << 1

num &
(num<<1)



Power of Friendship: Approach #2 (Musical Chairs)

- ❑ **Idea: What if we move the input bits around?** We can have them all play musical chairs, i.e. **shift the bits to the left by one**, and check for a collision!

If num has a pair of 1 bit friends

num	1	1	0	1	0	0	1	1
num << 1	1	0	1	0	0	1	1	0

num &
(num<<1)

If num has no pairs of 1 bit friends

num	0	1	0	1	0	1	0	1
num << 1	1	0	1	0	1	0	1	0

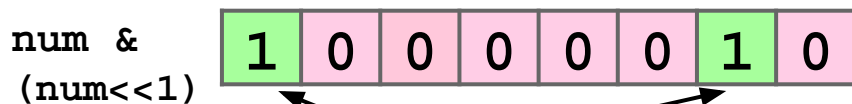
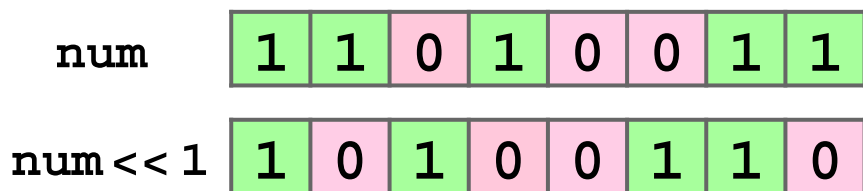
num &
(num<<1)



Power of Friendship: Approach #2 (Musical Chairs)

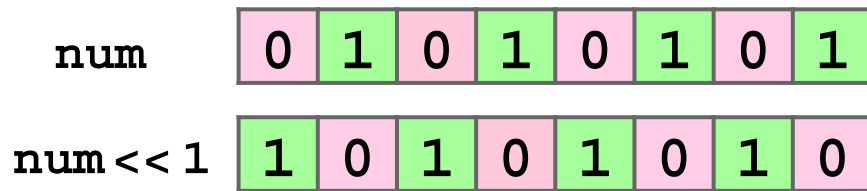
- ❑ **Idea: What if we move the input bits around?** We can have them all play musical chairs, i.e. **shift the bits to the left by one**, and check for a collision!

If num has a pair of 1 bit friends



Not all bits cancel out – the leftover 1 bits are a crash record of bit friends!

If num has no pairs of 1 bit friends



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



Power of Friendship: Approach #2 (Musical Chairs)

Implement `hasBitFriends(int num)`, which returns `true` if the binary representation of `num` has a pair of consecutive 1 bits, and `false` otherwise,

```
bool hasBitFriends(int num) {  
    return num & (num << 1) != 0;  
}  
// solution! true means overlapping  
// 1 bits between num and num << 1
```



Could right-shifting (`num >> 1`) work instead?

➤ i.e. imaginary friend! 🦄

Yes! But due to arithmetic right-shifting (may add an **extra 1 bit** by filling the MSB with a negative sign bit) **will need to cast**, i.e. `num & ((unsigned int) num >> 1) != 0`

We've reached the next station!



CS107 Scenic Recap: Jerry Cain High-Speed Train



**Bits, Bytes,
Integers!**

**Exercise: Power
of Friendship**



**C Strings &
Methods**

**Exercise: Life
Comes Full Circle**



**Pointers,
Arrays, Heap**

**Exercise:
Malevolent Heap**



**Generics,
Void Stars**

**Exercise: Yeet
That Element**



Time to start stringing together our quirkiest data type yet!

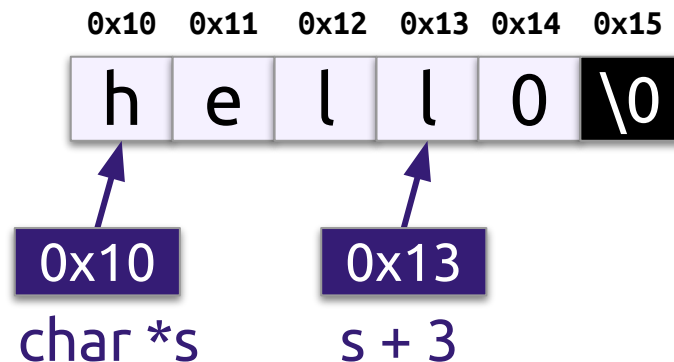


C Strings — Different from Python/C++!

A C string is an **array of characters** (`char []`) with a **null-terminating character** `'\0'` at the end. **It is vital that a string be properly null-terminated.**

❑ For common operations (e.g., comparing/concatenating/searching strings), we can use string library functions in `<string.h>`

❑ C strings can be represented as **`char*`**: **pointers to characters**, and we can move those pointers to  advance through chars.



❑ **When passing strings as parameters, we always pass them as `char*` (a pointer to the first char).** Thus, chars changed inside a function will persist outside of it.

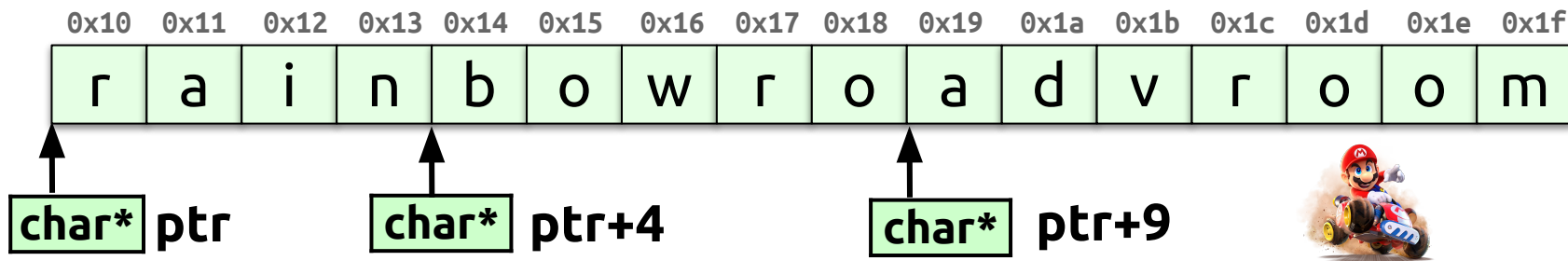
In summary, we usually find it more convenient and powerful to **use a `char*` / string pointer**  **to access, share, and manipulate the data** inside an array of characters.



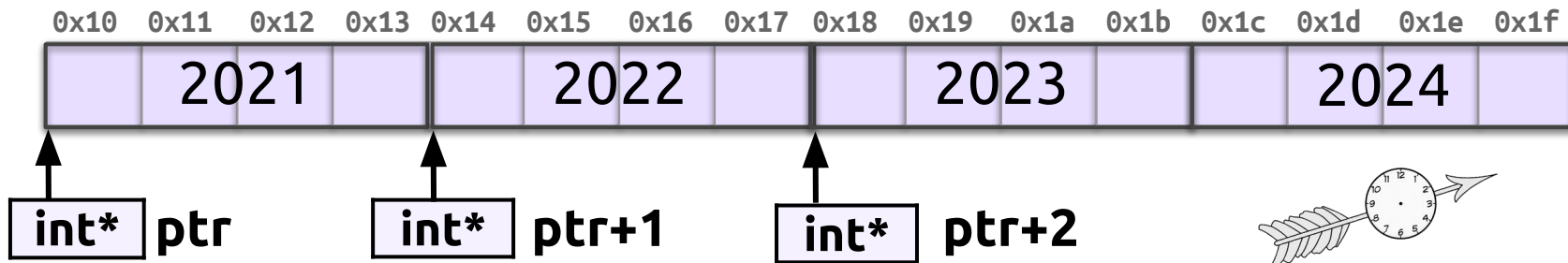
Pointers and Pointer Arithmetic

Pointer arithmetic works in terms of the **size of the data type being pointed to**

→ **+1** for a **char* pointer** means advance one char, so just **1 byte**



→ **+1** for an **int* pointer** means advance one int, so **4 bytes**





Pointer Fun! Addressing (&), Dereferencing (*)

Addressing: & is the **address-of** operator, and the expression **&variable** will tell us where **variable** lives in memory by producing its 8-byte address.

```
int jerryland = 10;  
int* intPtr = &jerryland;
```

intPtr

0x107

jerryland



enchanted
Troccoli
Forest



bitWorld





Pointer Fun! Addressing (&), Dereferencing (*)

Addressing: & is the **address-of** operator, and the expression **&variable** will tell us where **variable** lives in memory by producing its 8-byte address.

```
int jerryland = 10;  
int* intPtr = &jerryland;
```

intPtr

0x107

jerryland



enchanted
Troccoli
Forest



bitWorld



Dereferencing: * is the **magic wand of dereferencing**: ***ptr** obtains the value being **pointed at**, like entering or clicking a URL. Recall that pointers hold address values.

```
*intPtr = 40; // new value!
```

intPtr

0x107



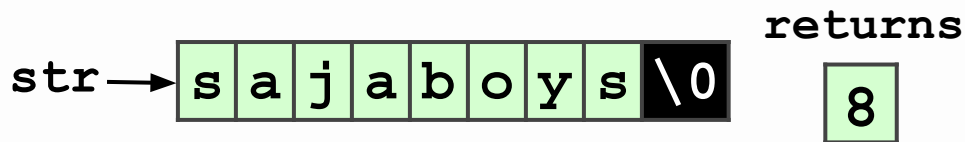
*intPtr knows to get/alter a 4-byte value starting at 0x107



Hot 100: Popular `<string.h>` Functions!

`size_t strlen(str)`

Returns length of string, **not including the null-terminating `\0`**





Hot 100: Popular `<string.h>` Functions!

`size_t strlen(str)`

Returns length of string, **not including** the null-terminating `\0`

`str` →

s	a	j	a	b	o	y	s	\0
---	---	---	---	---	---	---	---	----

 returns

8

`int strcmp(str1, str2)`

Returns **0** if `str1` and `str2` are identical
Returns **negative** if `str1` comes first in the dictionary, **positive** if `str2` comes first.

`str1` →

black

`str2` →

pink

 returns

<0

`str1` →

pink

`str2` →

pink

 returns

0

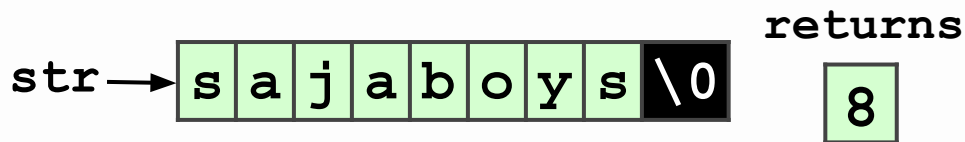
Note: The variant function `strncmp(str1, str2, n)` will **compare only up to n chars!**



Hot 100: Popular `<string.h>` Functions!

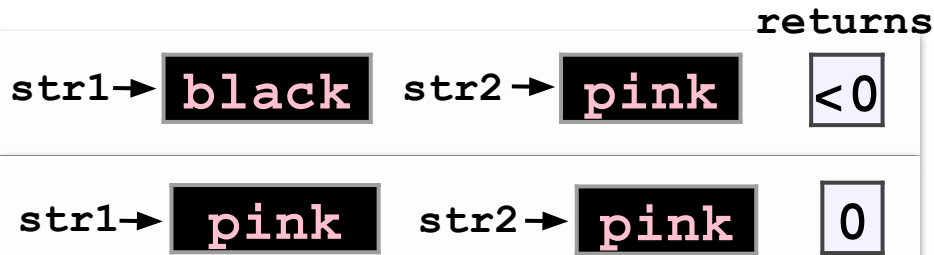
`size_t strlen(str)`

Returns length of string, **not including the null-terminating `\0`**



`int strcmp(str1, str2)`

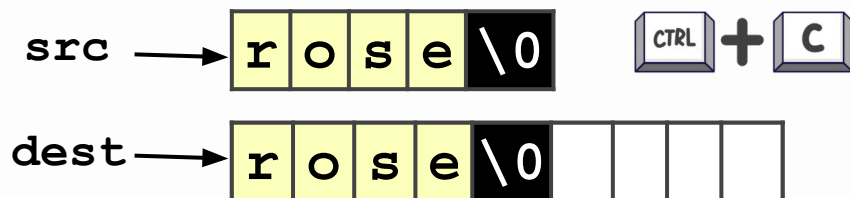
Returns **0** if `str1` and `str2` are identical
Returns **negative** if `str1` comes first in the dictionary, **positive** if `str2` comes first.



Note: The variant function `strncmp(str1, str2, n)` will compare only up to `n` chars!

`char* strcpy(dest, src)`

Copies the characters of a source string (including `\0`) to a destination buffer.





Hot 100: More String Functions!

```
char* strncpy(dest, src, n)
```

Copies **n bytes** from **source** into destination (1 char = 1 byte), but doesn't guarantee result will be null-terminated unless the **\0** is copied over

src →

r	o	s	e	b	u	d	\0
---	---	---	---	---	---	---	----

dest →

r	o	s	e				
---	---	---	---	--	--	--	--

 n

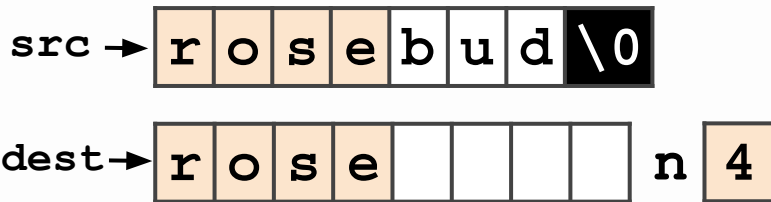
4



Hot 100: More String Functions!

```
char* strncpy(dest, src, n)
```

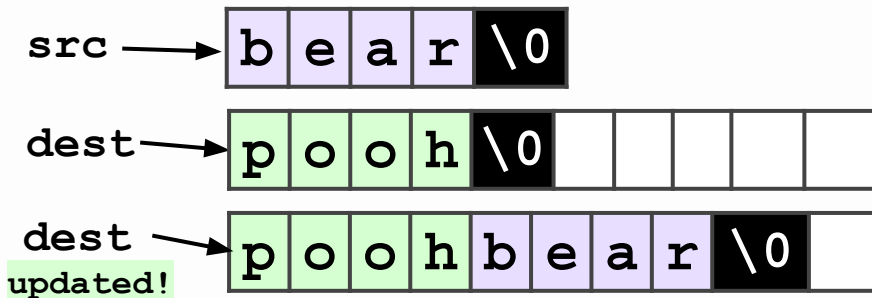
Copies **n bytes** from **source** into **destination** (1 char = 1 byte), but doesn't guarantee result will be null-terminated unless the **\0** is copied over



```
char* strcat(dest, src)
```

Ben's favorite function, because cats 🐱.

Appends a copy of destination string to source, overriding the source's **\0** char, and tacking on a new **\0** at the end.

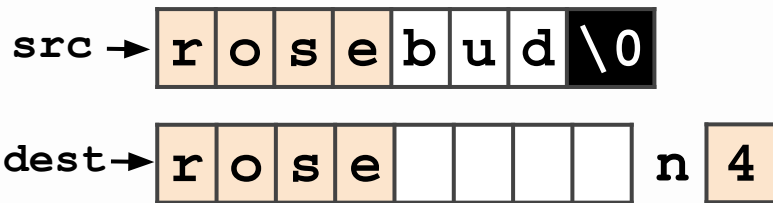




Hot 100: More String Functions!

char* strncpy(dest, src, n)

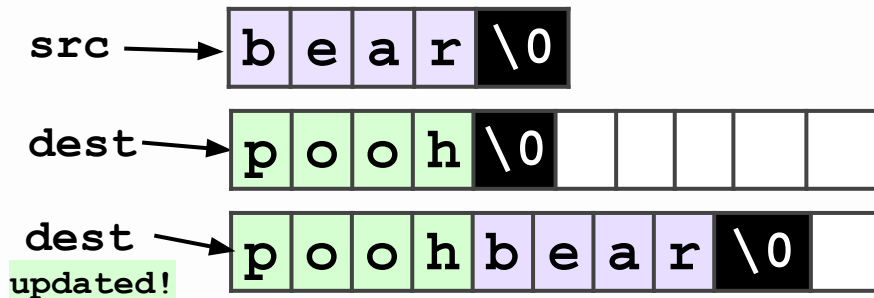
Copies **n** bytes from **source** into destination (1 char = 1 byte), but doesn't guarantee result will be null-terminated unless the **\0** is copied over



char* strcat(dest, src)

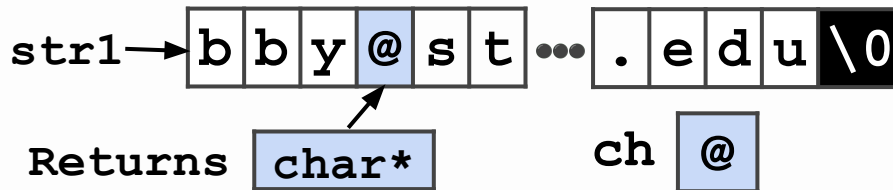
Ben's favorite function, because cats 🐱.

Appends a copy of destination string to source, overriding the source's **\0** char, and tacking on a new **\0** at the end.



char* strchr(str, ch)

Returns pointer to first occurrence of character **ch** in **str**, NULL if not found



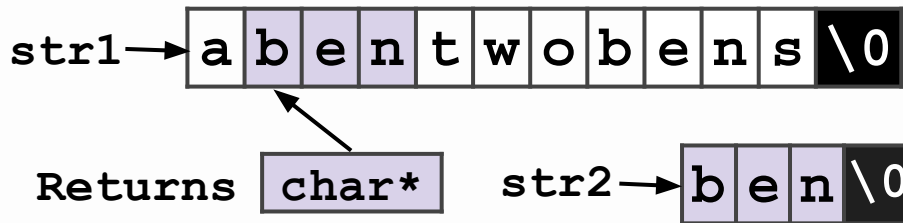


Hot 100: Even More String Functions!



```
char *strstr(str1, str2)
```

Returns pointer to **first occurrence** of **str2** in **str1**, or **NULL** if not found.



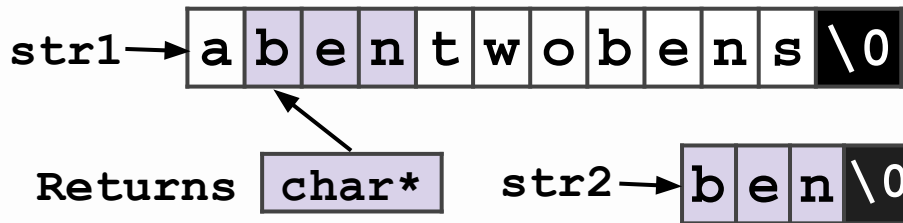


Hot 100: Even More String Functions!



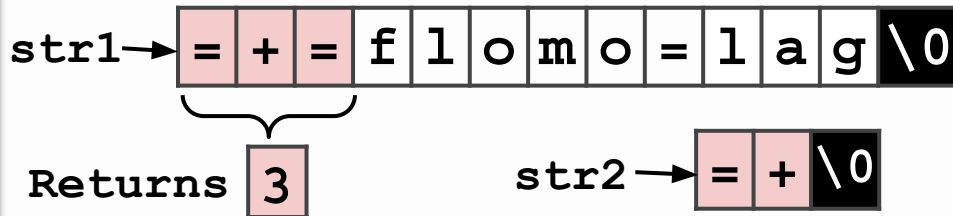
`char *strstr(str1, str2)`

Returns pointer to **first occurrence** of `str2` in `str1`, or `NULL` if not found.



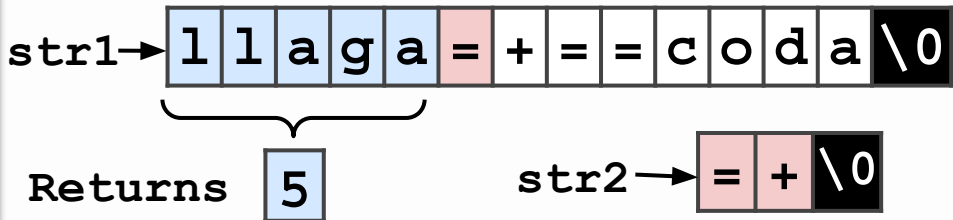
`size_t strspn(str1, str2)`

Returns **length** of initial portion of `str1` that only contains characters from `str2`



`size_t strcspn(str1, str2)`

Returns **length** of initial portion of `str1` that **doesn't** contain characters from `str2`





Some Advice Pointers for C String Tasks

1

Hot Take: I've found **107** to be as much a 🎨 **WAYS-CE (arts and crafts) course as a programming one**: always have a paper at your side to sketch memory / pointer diagrams, examples, or illustrate your thoughts!

```
- - r e d \0 g r e e n - - - b l u e \0
```

s ↑

p ↑

s 👉 "red"

p 👉 "green---blue"

(at the end of first call to strtok)

```
char* → b l u e \0
```

```
void * a
```

```
char* → r e d \0
```

```
void * b
```



*Sometimes, a diagram can help disambiguate if a pointer is a **char*** or **char****!*

2

If you get stuck, try **tangible examples**, e.g., choose and work with a small string like 🧵 "token1--token2" or 🧵 "-red-green-blue", and see if you can find patterns! You will find a way!

Happy to take questions!

C Strings, <string.h> functions, why char*, pointers, silly strings, string theory, etc.

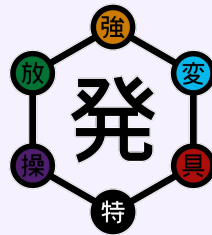




Strings Exercise: Life Comes Full Circle

Implement the function `comesFullCircle(str, N)`, which:

- Accepts a `char* str` and an integer `N` as input
- Returns `true` if the string `str` has a matching prefix and suffix that is precisely `N` chars long, and `false` otherwise.



In other words, we return true only if the **first N chars matches the last N chars** ✓, **which are allowed to overlap**. If `N` exceeds the string's length, simply return `false`.

`str` →

h	u	n	t	e	r		h	u	n	t	e	r	\0
---	---	---	---	---	---	--	---	---	---	---	---	---	----

`N`

3

 full circle?

`str` →

h	u	n	t	e	r		h	u	n	t	e	r	\0
---	---	---	---	---	---	--	---	---	---	---	---	---	----

`N`

6

 full circle?

`str` →

l	a	l	a	l	a	\0
---	---	---	---	---	---	----

`N`

2

full circle?

`str` →

l	a	l	a	l	a	\0
---	---	---	---	---	---	----

`N`

4

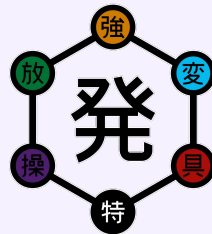
full circle?



Strings Exercise: Life Comes Full Circle

Implement the function `comesFullCircle(str, N)`, which:

- Accepts a `char* str` and an integer `N` as input
- Returns `true` if the string `str` has a matching prefix and suffix that is precisely `N` chars long, and `false` otherwise.



In other words, we return `true` only if the **first `N` chars matches the last `N` chars** ☒, **which are allowed to overlap**. If `N` exceeds the string's length, simply return `false`.

`str` →

h	u	n	t	e	r		h	u	n	t	e	r	\0
---	---	---	---	---	---	--	---	---	---	---	---	---	----

`N`

3

 full circle? ☒

`str` →

h	u	n	t	e	r		h	u	n	t	e	r	\0
---	---	---	---	---	---	--	---	---	---	---	---	---	----

`N`

6

 full circle? ☒

`str` →

l	a	l	a	l	a	\0
---	---	---	---	---	---	----

`N`

2

full circle? ☒

`str` →

l	a	l	a	l	a	\0
---	---	---	---	---	---	----

`N`

4

full circle? ☒



Strings Exercise: Life Comes Full Circle

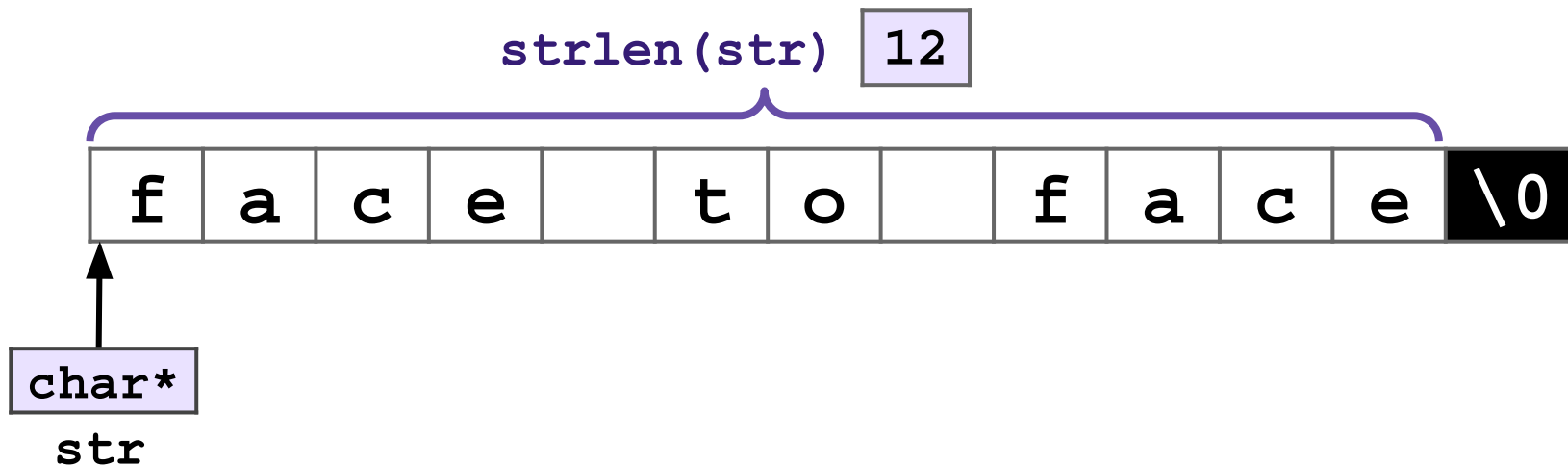
Implement `comesFullCircle(str, N)`, which returns `true` only if:



The first `N` chars matches the last `N` chars of string `str`. The chars can overlap.



If no such match or if `N` exceeds the string's length, simply return `false`.

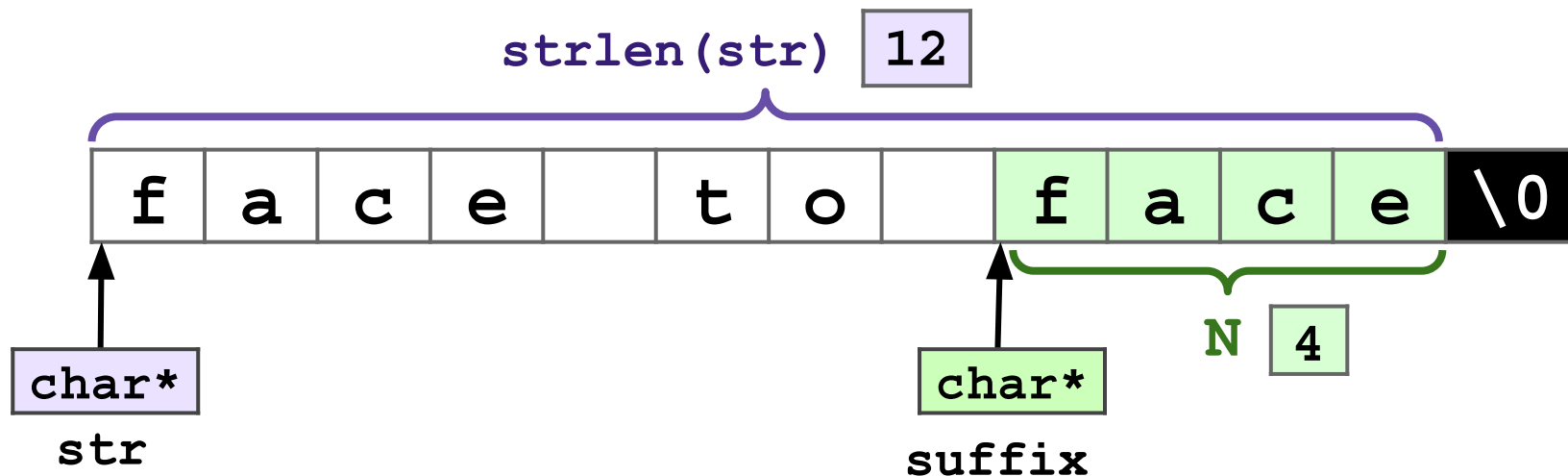


Example: `str = "face to face"`, `N = 4`



Strings Exercise: Life Comes Full Circle

Q: From `str`, do we obtain a pointer to its `suffix` that is `N` chars long?



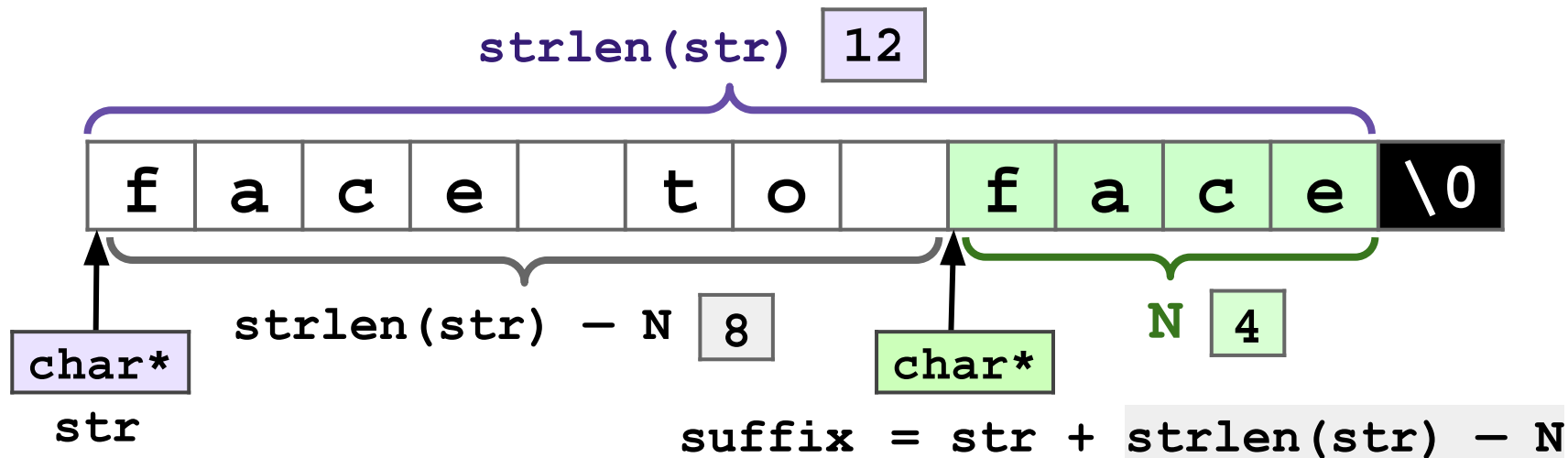
Example: `str = "face to face"`, `N = 4`



Strings Exercise: Life Comes Full Circle

Q: From `str`, do we obtain a pointer to its **suffix** that is `N` chars long?

A: Increment `str` by its length minus the `N` chars at the very end!



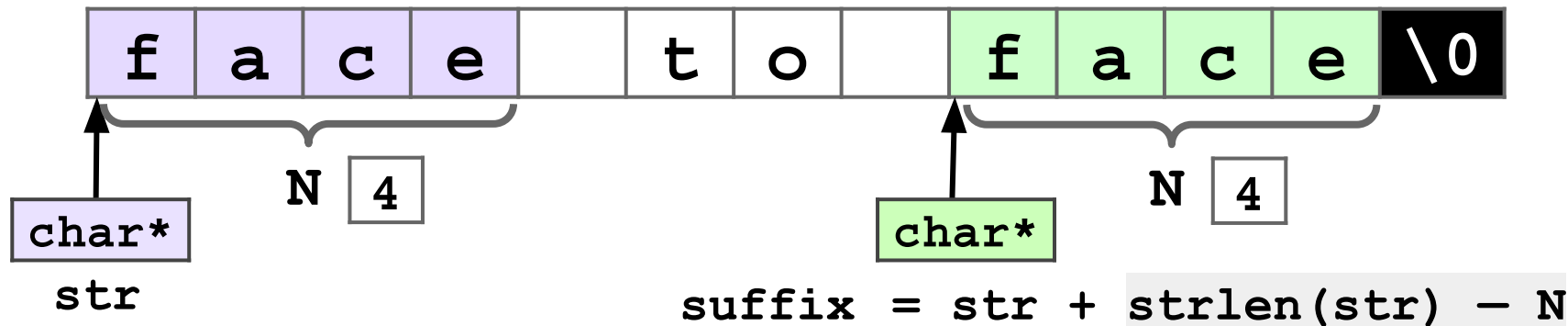
Example: `str = "face to face"`, `N = 4`



Strings Exercise: Life Comes Full Circle

Q: How can we compare `str` and `suffix`, but only up to `N` chars?

A: One way is `strncmp`, which returns 0 on a match!



Example: `str = "face to face"`, `N = 4`



Life Comes Full Circle: Solution Code

Implement `comesFullCircle(str, N)`, which returns `true` only if:

-  The **first N chars** matches the **last N chars** of string `str`. The chars can overlap.
-  If no prefix/suffix match or N exceeds the string's length, simply return `false`.

```
bool comesFullCircle(char* str, int N){  
    if (strlen(str) < N){  
        return false; // str is too short  
    }  
    char* suffix = str + strlen(str) - N;  
    // compares the first N chars (prefix)  
    // with the last N chars (suffix)  
    return strncmp(str, suffix, N) == 0;  
}
```

`strlen(str) - N`



`str`

`suffix`

N 2 full circle? 

Next Station! And onward we go :)



CS107 Scenic Recap: Jerry Cain High-Speed Train



**Bits, Bytes,
Integers!**

Exercise: Power
of Friendship



**C Strings &
Methods**

Exercise: Life
Comes Full Circle



**Pointers,
Arrays, Heap**

**Exercise:
Malevolent Heap**



**Generics,
Void Stars**

Exercise: Yeet
That Element



Arrays in C are just such a ray of sunshine, aren't they?



C is Pass by Value for Function Args

When passing parameters, **C passes a copy of whatever is passed.**

- ❑ To change values, and have those changes persist outside the function, **we must** 🙌 **pass a pointer to the value instead** (i.e. its address in memory).
 - ❑ For instance, to change an **int**, we have to pass in an **int *** (pointer to an int)
 - ❑ To change a **char*** (string), we have to pass in a **char**** (pointer to string).



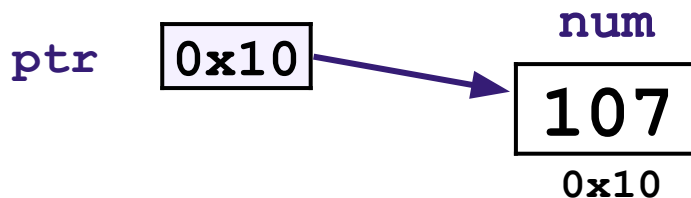
C is Pass by Value for Function Args

When passing parameters, **C passes a copy of whatever is passed.**

- ❑ To change values, and have those changes persist outside the function, **we must** 🙌 **pass a pointer to the value instead** (i.e. its address in memory).
- ❑ For instance, to change an **int**, we have to pass in an **int *** (pointer to an int)
- ❑ To change a **char *** (string), we have to pass in a **char **** (pointer to string).

Short Example

```
function mischief(int* ptrCopy) {  
    *ptrCopy = 701;  
}  
  
int main() {  
    int num = 107;  
    int* ptr = &num;  
    mischief(ptr);  
}
```





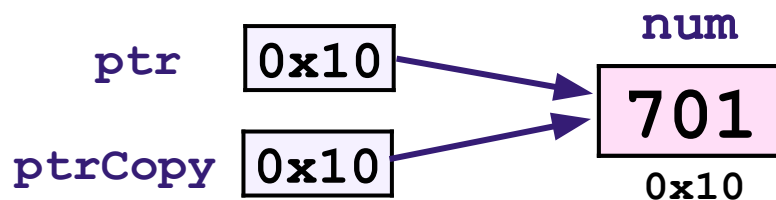
C is Pass by Value for Function Args

When passing parameters, **C passes a copy of whatever is passed.**

- ❑ To change values, and have those changes persist outside the function, **we must** 🙌 **pass a pointer to the value instead** (i.e. its address in memory).
- ❑ For instance, to change an **int**, we have to pass in an **int *** (pointer to an int)
- ❑ To change a **char *** (string), we have to pass in a **char **** (pointer to string).

Short Example

```
function mischief(int* ptrCopy) {  
    *ptrCopy = 701;  
}  
  
int main() {  
    int num = 107;  
    int* ptr = &num;  
    mischief(ptr);  
}
```



Though the pointer is itself duplicated, **it's still pointing to the same value** in memory, and so **we can directly modify that value!**



Heated Rivalry: Arrays, Pointers

- ❑ When passing **arrays as parameters**, they decay into a  **pointer to the first element**
- ❑ A common bug is calling `sizeof()` on an array that's decayed into an 8-byte pointer.

```
char dorm_name[100];
```

e	n	c	h	a	n	t	...	\n
---	---	---	---	---	---	---	-----	----

```
inhabit(char* dorm_name);
```

- ❑  **Array of pointers** = array where each element is a pointer to some data.
- ❑ For example, an **array of strings** stores **pointers to their first characters**, NOT the characters themselves. Those can be stored elsewhere in memory

Array of `char*`
e.g., `char* dorms[3];`

`dorms[0]`

`char*`



n	a	r	n	i	a	\0
---	---	---	---	---	---	----

`dorms[1]`

`char*`



p	l	u	t	o	\0
---	---	---	---	---	----

`dorms[2]`

`char*`



m	a	r	s	\0
---	---	---	---	----



Dynamic Memory: Heap Allocation Methods

- ❑ **Stack memory** is used for function call stack frames, local variables, etc.
 - ❑ **Heap memory** is a shared memory oasis for dynamic memory allocation!
- ❑ We can't freely give ourselves memory on the heap at random addresses 🧑
 - ❑ **Imagine the heap as being guarded fiercely by the heap alligator!** 🐊 (secret identity to be revealed on **Assign6**)
- ❑ To obtain heap memory, we need to 🤝
 - ❑ **The alligator understands requests such as `malloc`, `realloc`, `free`**, and can set aside precious heap memory for us!

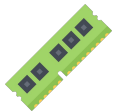
Heap Memory



What will Pikachu do?

`malloc` `realloc`
`free` `strdup`

Stack Memory



Dynamic Memory: Heap Allocation Methods

```
void* malloc(size_t size);
```

- ❑ **Input the number of raw bytes** to be allocated on the heap, i.e. the request size.
- ❑ **Returns base address** of freshly allocated memory.

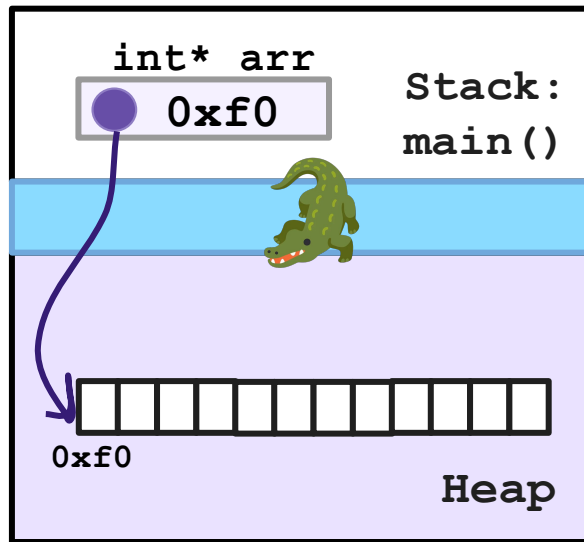
```
void* realloc(void *ptr, size_t new_size);
```

- ❑ **Input pointer to heap memory** given by malloc, and a **new total size to reallocate**.
- ❑ **Returns base address** of resized memory region.
- ❑ May move to new address if can't expand in-place! (will ferry the existing memory over)

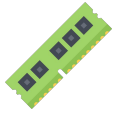
```
void free(void* ptr);
```

- ❑ **Accepts pointer to heap** from malloc, realloc, etc. and **recycles the memory back** to heap.

```
int* arr = malloc(  
    3 * sizeof(int));
```



Note: Don't forget to free your heap memory before the program ends! Otherwise a memory leak!



Note: Return pointer type from malloc, realloc, etc.

```
void* malloc(size_t size);
```

- ❑ **Input the number of raw bytes** to be allocated on the heap, i.e. the request size.
- ❑ **Returns base address** of freshly allocated memory.



Though `malloc()` has a `void*` return type to denote an address to generic (i.e. any kind of) memory, we can set a 👉 **pointer equal to without casting**, e.g.,

- `int* num_arr = malloc(n * sizeof(int));`
- `char** four_strs = malloc(4 * sizeof(char*));`
- `movie* moviePtr = malloc(sizeof(movie));`



Dynamic Memory: Other Allocation Methods

```
char* strdup(const char* str);
```

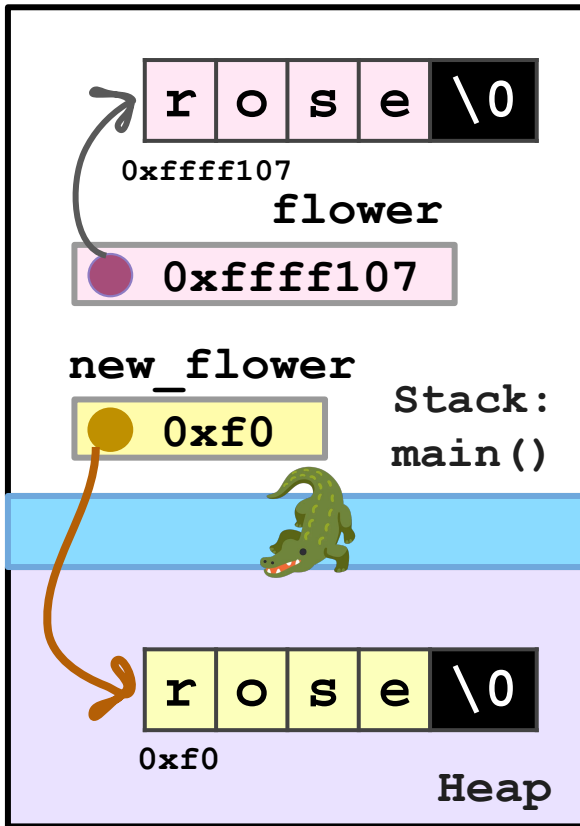
- ❑ A convenience function for **duplicating C strings onto the heap**.
- ❑ Example with duplicating 🌹 "rose" on right:

```
char[5] flower; 🌸  
strcpy(flower, "rose");  
char* new_flower = strdup(flower);
```

```
void* calloc(size_t count, size_t size);
```

- ❑ An allocation function that's just like **malloc**, except that it **zeroes out** the allocated memory!
- ❑ Example to create array of 3 integers set to 0:

```
int* ben_grades = calloc(3, sizeof(int)); 💔
```





Stack vs Heap Memory — Which to Use?



What are the general **benefits of heap memory** over stack memory?

- Heap memory is **more plentiful** + we can **resize memory** after its initial allocation!
- **Heap memory persists outside a function call** (unlike stack memory, which is erased when the call is over), so we have better control over its memory lifetime.



What are the general **disadvantages of heap memory** over stack memory?

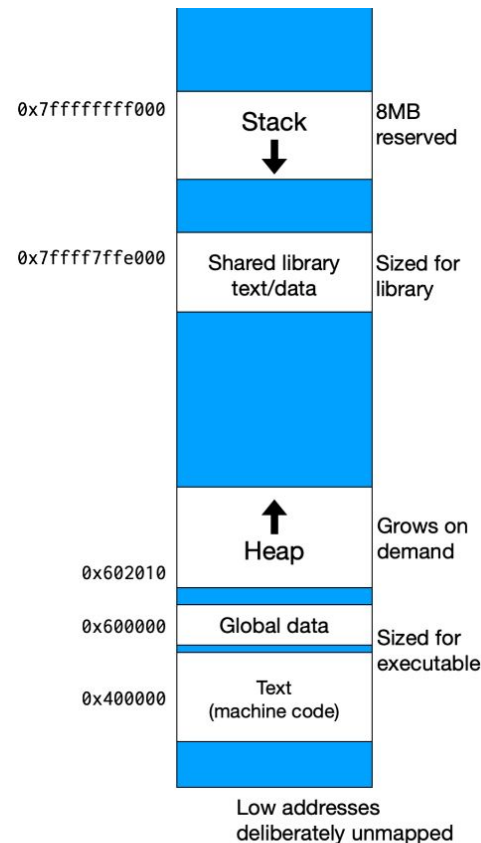
- **More opportunities for errors** ⚠️, such as type errors, forgetting to free all allocations before the program is done, allocating wrong size, pointer errors, etc.
- **Stack allocation is faster, more convenient** (automatic allocation and deallocation, can declare and initialize in one step), and has superior type safety.



Programs typically use both in combination, but **generally stick to stack allocation unless we need dynamic allocation**, e.g., may need to resize later.

Happy to take questions!

Heated rivalry (**pointers and arrays**), heated rivalry deux (**stack and heap memory**), etc.

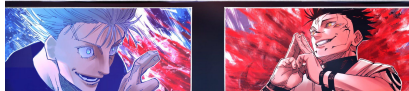




Exercise: Domain Expansion, Malevolent Heap

Implement the function `expand(strings, string_count)`, which receives a non-empty stack array of `char* strings` and integer `strings_count` as input.

It should return a **heap-allocated string which is the concatenation of all `strings_count` strings in `strings`**. Don't forget space for the null-terminator!



```
char* expand(char* strings[], int strings_count){  
    // present your implementation here!  
}
```

```
/* Sample Usage in main() */  
char* strings[] = {"cleave", "dismantle", "flame"};  
int count = sizeof(strings) / sizeof(strings[0]);  
char* result = expand(strings, count);  
// result should be string "cleavedismantleflame"
```



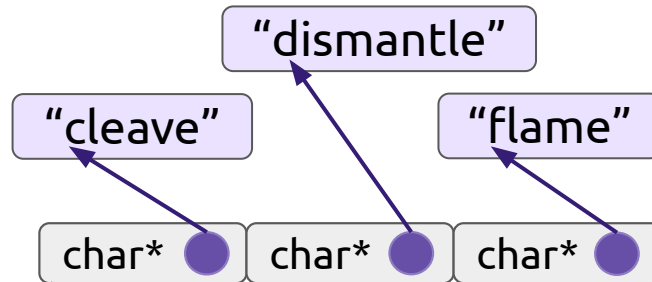
Exercise: Domain Expansion, Malevolent Heap

Implement `expand(strings, string_count)`, which returns a heap string which is the concatenation of all `strings_count` stack strings in `strings`.

Heap

Our goal is to create a heap-allocated, concatenated string and return it!

"cleavedismantleflame"



Stack:
main()

`char**` `strings`

Stack:
expand()

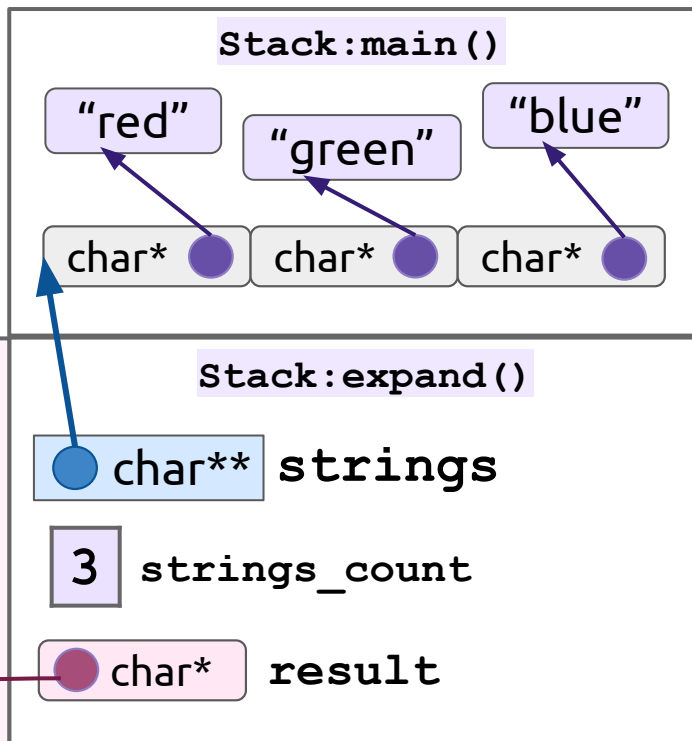
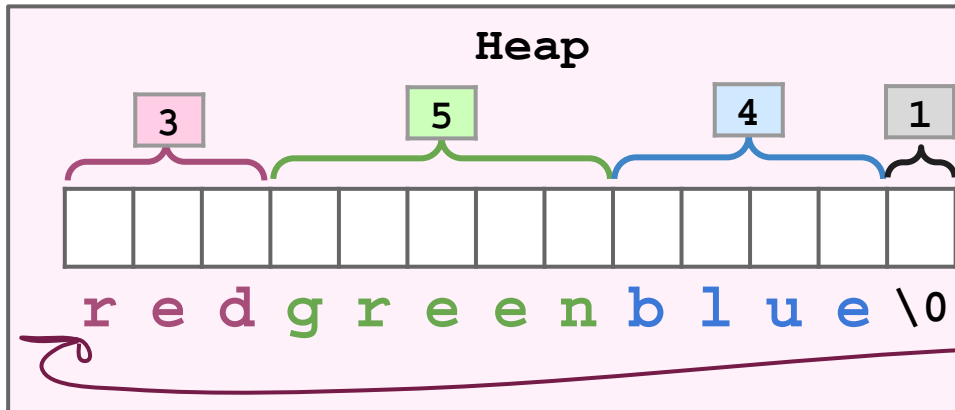
`char*` `result` `strings_count` 3



Domain Expansion: Approach #1 (Malloc Once)

Key Idea: Get Heap Allocation Done In One Go!

1. Figure out how large the heap buffer has to be to fit all strings joined together.
2. Allocate a buffer of that size with **malloc**
3. Copy / append all strings into that buffer.



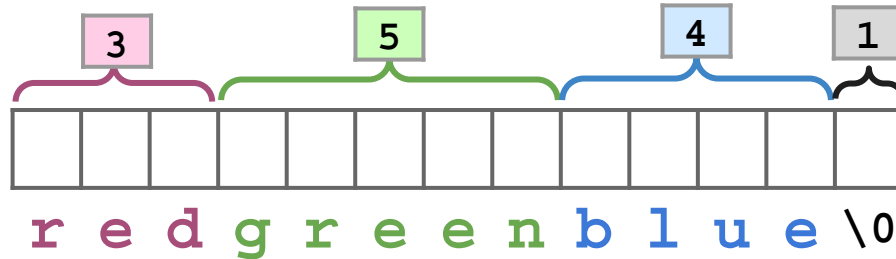
➤ This requires **adding up the strlen of each string, plus 1 for the \0 char.**



Domain Expansion: Approach #1 (Malloc Once)

Implement `expand(strings, string_count)`, which returns a heap string which is the concatenation of all `strings_count` strings in `strings`.

```
char* expand(char* strings[], int strings_count){  
    size_t final_len = 0; // length of eventual result  
    for (int i = 0; i < strings_count; i++){  
        final_len += strlen(strings[i]);  
    }  
    ... and continued! note: with malloc, +1 for \0 char
```



- This requires **adding up the `strlen` of each string, plus 1 for the `\0` char.**



Domain Expansion: Approach #1 (Malloc Once)

Implement `expand(strings, string_count)`, which returns a heap string which is the concatenation of all `strings_count` strings in `strings`.

```
char* expand(char* strings[], int strings_count){
    size_t final_len = 0; // length of eventual result
    for (int i = 0; i < strings_count; i++){
        final_len += strlen(strings[i]);
    }
    // create large enough heap buffer in one go, including /0
    char* result = malloc(final_len + 1);
    for (int i = 0; i < strings_count; i++){
        strcat(result, strings[i]); // append each string
    }
    return result;
}
```

Q: This is nearly correct. What's wrong?



Domain Expansion: Approach #1 (Malloc Once)

Implement `expand(strings, string_count)`, which returns a heap string which is the concatenation of all `strings_count` strings in `strings`.

```
char* expand(char* strings[], int strings_count){
    size_t final_len = 0; // length of eventual result
    for (int i = 0; i < strings_count; i++){
        final_len += strlen(strings[i]);
    }
    // create large enough heap buffer in one go, including /0
    char* result = malloc(final_len + 1);
    for (int i = 0; i < strings_count; i++){
        strcat(result, strings[i]); // append each string
    }
    return result;
}
```

Q: This is nearly correct. What's wrong?

A: This fails for the index-0 string, as we attempt to concatenate with uninitialized memory!



Domain Expansion: Approach #1 (Malloc Once)

Implement `expand(strings, string_count)`, which returns a heap string which is the concatenation of all `strings_count` strings in `strings`.

```
char* expand(char* strings[], int strings_count){
    size_t final_len = 0; // length of eventual result
    for (int i = 0; i < strings_count; i++){
        final_len += strlen(strings[i]);
    }
    // create large enough heap buffer in one go, including /0
    char* result = malloc(final_len + 1);
    strcpy(result, strings[0]); // copy first string in
    for (int i = 1; i < strings_count; i++){
        strcat(result, strings[i]); // append each string after
    }
    return result;
}
```

Solution: `strcpy` the first string into the uninitialized heap buffer, then `strcat` the other strings



Domain Expansion: Approach #2 (Resize as We Go)

Key Idea: Resize the heap string as we expand!

1. Allocate an initial buffer on the heap for just `strings[0]` with `strdup`.
2. For each other string, resize the buffer to fit the additional characters, and concatenate.

`r e d \0` (initial)

`r e d g r e e n \0` (next)

`r e d g r e e n b l u e \0`

(final)

Heap

Stack:main()

"red"

"green"

"blue"

char*

char*

char*

Stack:expand()

char** strings

3 strings_count

char* result

➤ To include a new string, we have to **expand the buffer by its string length**.

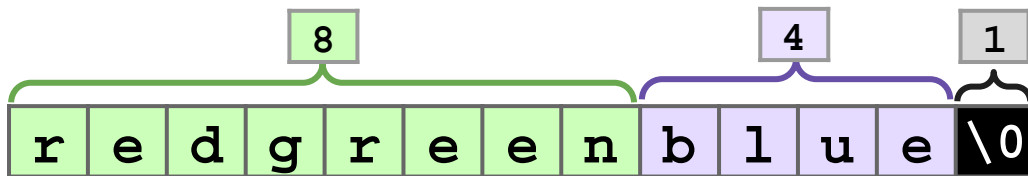


Domain Expansion: Approach #2 (Resize as We Go)

Implement `expand(strings, string_count)`, which returns a heap string which is the concatenation of all `strings_count` strings in `strings`.

```
char* expand(char* strings[], int strings_count){  
    char* result = strdup(strings[0]); // start with first string  
  
    // loop through other strings, concatenating with result  
    for (int i = 1; i < strings_count; i++){  
        size_t new_size = strlen(result) + strlen(strings[i]) + 1;  
        ... more code to realloc memory and concatenate  
    }  
}
```

To expand a heap buffer,
we'll want `realloc`!



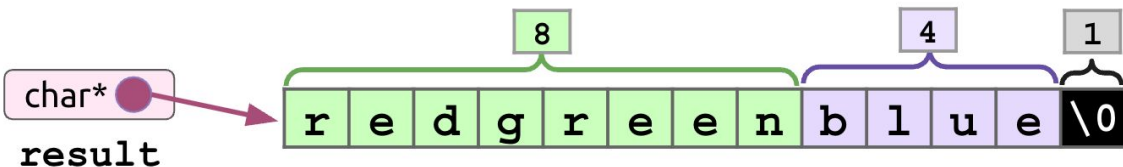
- To include a new string, we have to **expand the buffer by its string length**.



Domain Expansion: Approach #2 (Resize as We Go)

Implement `expand(strings, string_count)`, which returns a heap string which is the concatenation of all `strings_count` strings in `strings`.

```
char* expand(char* strings[], int strings_count){  
    char* result = strdup(strings[0]); // start with first string  
  
    // loop through other strings, concatenating with result  
    for (int i = 1; i < strings_count; i++){  
        size_t new_size = strlen(result) + strlen(strings[i]) + 1;  
        result = realloc(result, new_size); // enlarge heap buffer  
        strcat(result, strings[i]);  
    }  
    return result;  
}
```



Final station on the way! We're almost there ❤️



CS107 Scenic Recap: Jerry Cain High-Speed Train



**Bits, Bytes,
Integers!**

Exercise: Power
of Friendship



**C Strings &
Methods**

Exercise: Life
Comes Full Circle



**Pointers,
Arrays, Heap**

Exercise:
Malevolent Heap



**Generics,
Void Stars**

**Exercise: Yeet
That Element**

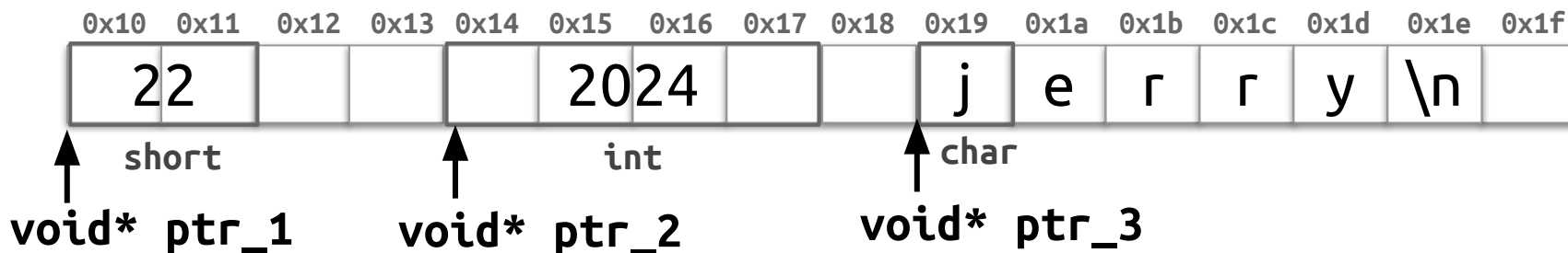


Despite the unflattering name, this content is hardly generic!



Generics: One Pointer to Rule Them All

- 👉 We can use **void*** to represent a generic pointer to “something”.
- 👉 It loses information about data types, allowing for more flexibility.

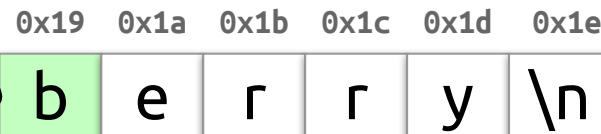


✗ We can't dereference void* pointers right away! Don't do this plz :(

To access/change memory a **void*** points to:

- ★ Use the generic memcpy / memmove
- ★ Or, if we know what type is actually being pointed to, we **first cast, then dereference**

***(char *) ptr_3 = 'b';**





Generics: Pointer Arithmetic

For pointer arithmetic with **void*** pointers, why do we cast to a **char***?

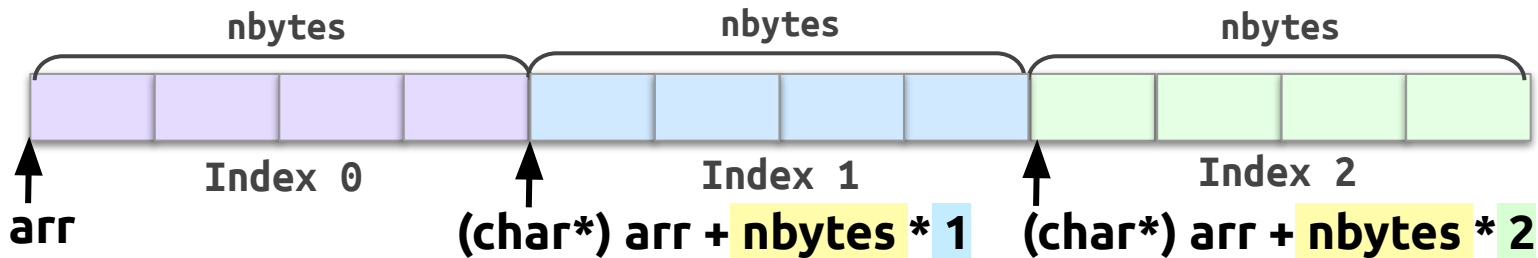
Consistency: Points to a **char** (only 1 byte), so we can work/add in terms of bytes!

Generics Idiom Consider a generic array **arr** where each element has width **nbytes**.
How do we get a **pointer to the ith element** of the array?

Solution: **void* elem_ptr = (char *) arr + i * nbytes;**

Cast to **char*** so we can work in individual bytes

Each element is **nbytes**, so multiply with number of elements (**i**) to move





Function Pointers: Sharing is Caring!

👉 We can **pass functions as parameters to other functions**, by specifying one or more function pointer parameters. These are termed **callback functions**.

- ❑ A **function pointer** lets us pass logic/wisdom from caller to callee
- ❑ For example, we can pass a function to **bubble_sort** that knows how to compare 2 elements of the type we are sorting

Function pointers have the form:

```
return_type (*function_name) (arg1,...)
```

For instance,

```
int (*sat_add) (int a, int b)
```

Comparison functions in particular have the following generic prototype:

```
int (*my_compare) (const void *a, const void *b)
```

Sign of **return value** tells us which elem goes first when sorting



Pointer to 1st element



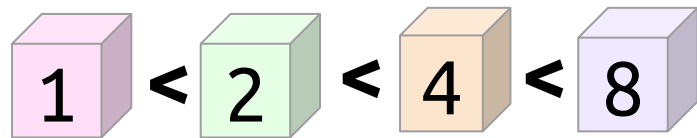
Pointer to 2nd element



Comparison Functions: Teaching C to Sort



Sorting is really just knowing **how to compare items** (i.e. which items go before which other items)



As such, to sort, we have to define a **comparison function** `cmp(a, b)`

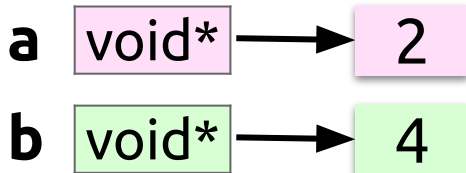
This function returns:

☐ `< 0` (Negative) if **a comes before b**
☐ `0` if **a and b are equal** ☐ `> 0` (Positive) if **a comes after b**



Most comparison functions we write, if not all, will have the blueprint:

```
int my_compare(const void *a, const void *b){  
    code to (1) cast and (2) dereference the received elements  
    code to (3) compare the values, often by subtracting them  
}
```



Due to generic `void*` args, **we must input pointers to what we're comparing!**



Generics: 3 Step Process for Comparison Functions!

```
int (*function_name) (const void *a, const void *b)
```

- 1 **Cast** the `void*` argument(s) to a known pointer type
- 2 **Deference** the typed pointer to access the value
- 3 **Compare** values to determine the result to return

Reminder! This returns:

0	if a and b are equal
<0	if a comes before b
>0	if a comes after b

***Note:** Steps 1 and 2 are often combined to cast and deference in one expression! e.g., `*(char*)`*



Generics: 3 Step Process for Comparison Functions!

```
int (*function_name) (const void *a, const void *b)
```

- 1 Cast the `void*` argument(s) to a known pointer type
- 2 Dereference the typed pointer to access the value
- 3 Compare values to determine the result to return

Reminder! This returns:

0	if a and b are equal
<0	if a comes before b
>0	if a comes after b

Note: Steps 1 and 2 are often combined to cast and dereference in one expression! e.g., `*(char*)`

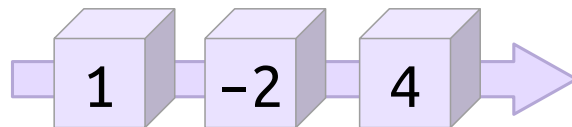


Example: Compare Integers by Absolute Value! (Lab 4)

```
int compare_abs(const void *a, const void *b){  
    2 1  
    int a_val = *(int *)a;  
    int b_val = *(int *)b;  
    return abs(a_val) - abs(b_val); 3  
}
```

a `void*` → 4

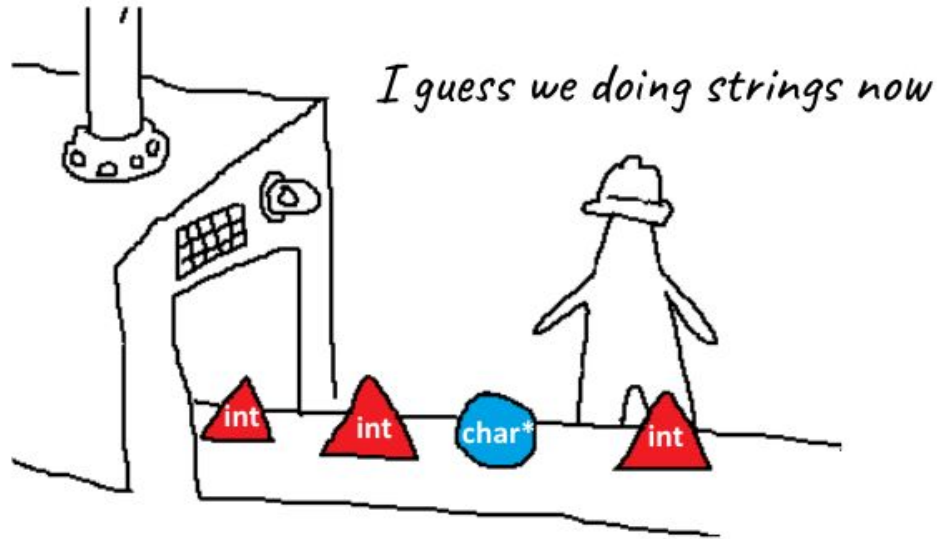
b `void*` → -4



Writing General-Purpose Code Requires Compromise



Generics is all about compromise! At a certain sacrifice (i.e. more indirection, pointers to values), we can make powerful functions work for all data types in C!



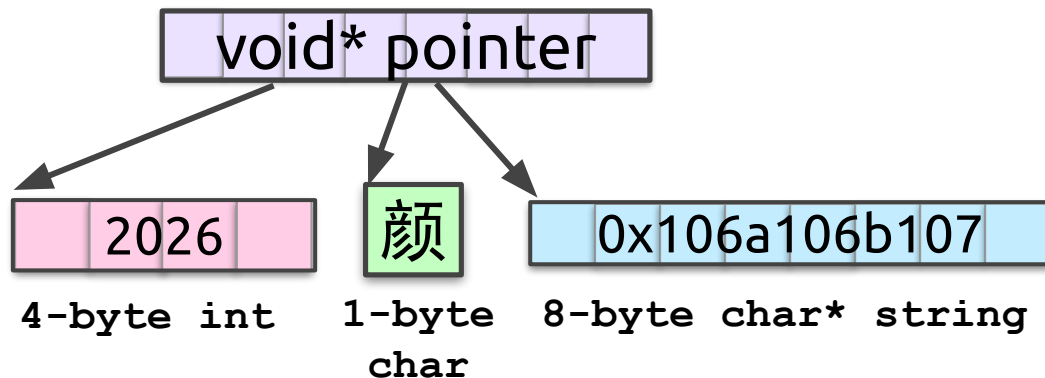
"But it is a sacrifice I'm willing to make" – ~~Lord Farquaad~~ C language

Writing General-Purpose Code Requires Compromise

🤝 **Generics is all about compromise!** At a certain sacrifice (i.e. more indirection, pointers to values), we can make powerful functions work for all data types in C!

e.g. qsort

```
void qsort( void* ptr, size_t count, size_t size,  
            int (*comp)(const void*, const void*) );
```



👉 *One pointer to rule them all
One pointer to find them
One pointer to bring them all
And in darkness bind them*

👉 **Why pointers?** There's no `void` data type that can hold any data value/size. The closest equivalent is a **8-byte `void*` address that could point to anything!**



Takeaway notes on using generic functions

👉 When calling generic functions, you typically **pass pointers to the block of data** you're trying to copy, modify, compare, etc., **not the data values itself**.

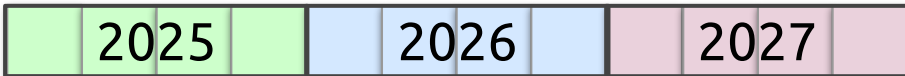
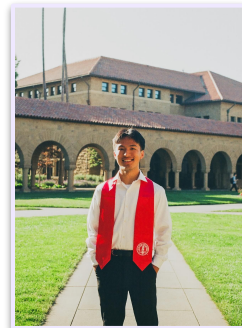
📖 Though to be sure, you can **check the C reference guide** for function specs!

```
int[] when_Ben_says_he_will_graduate = {2025, 2026, 2027};  
int actual_year = 2027;      // i'm almost almost there fr lol 🎓
```

❌ `bsearch(actual_year, when_Ben_says_he_will_graduate, ...)`

// we need the address of / a pointer to what we're searching for

✅ `bsearch(&actual_year, when_Ben_says_he_will_graduate, ...)`



when_Ben_says_he_will_graduate

👉 Note our array decays into a pointer when passed as arg



&actual_year

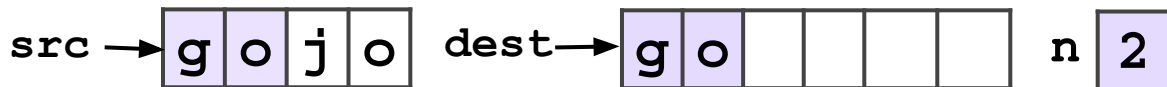
👉 We need pointer to key



Methods of Brotherly Love: Memcpy, Memmove

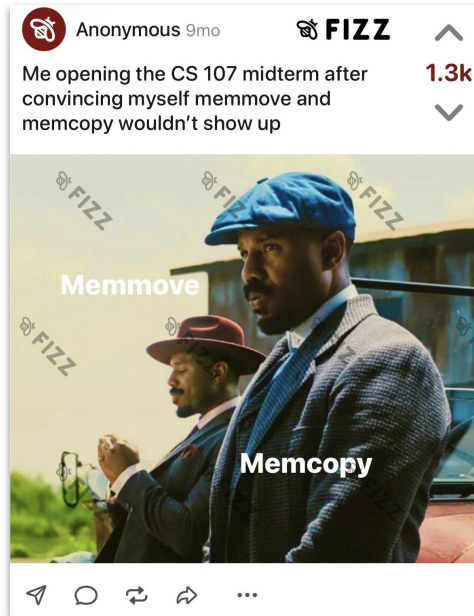
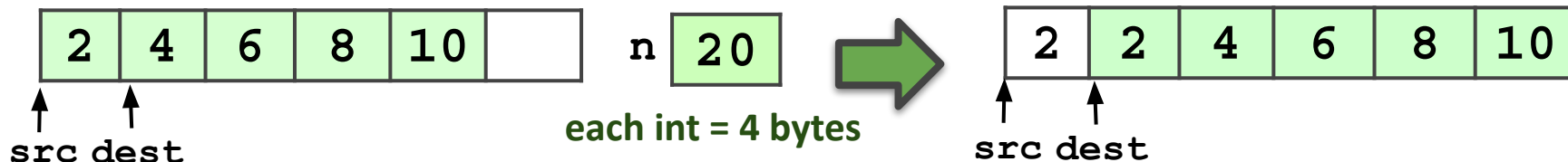
```
void* memcpy(void* dest, const void* src, size_t n)
```

- ❑ Copies **n bytes** from one address to another address (i.e. **dest** and **src** should be pointers!)
- ❑ Assumes the two memory regions **don't overlap**.



```
void* memmove(void* dest, const void* src, size_t n)
```

- ❑ Also copies **n bytes** from one address to another address, **but can handle overlapping memory!**



Happy to take questions!

Generics, void stars, comparison functions (i.e. the thief of joy), memcpy, memmove, etc.



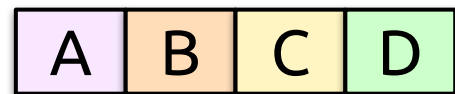


Generics Exercise: Yeet that Element!

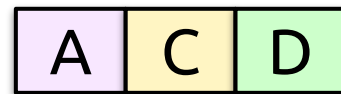
Implement a generic function `yeet(base, index, nelems_ptr, width)` that:

- Given an array with starting address **base** and a given integer **index**, **deletes (or “yeets”, in the colorful language of my generation) the element at that index.**
- We should **slide all the elements to the right of the index down**, ensuring that the array base should stay in-place. You may assume $0 \leq \text{index} < \text{nelems}$.

```
void yeet(void* base, int index, int* nelems_ptr, int width){  
    void* elemToDelete = _____;  
    void* elemToRight = (char*) elemToDelete + _____;  
    // slide all elements to the right down  
    _____;  
    // finally, update the number of elements  
    _____;  
}
```



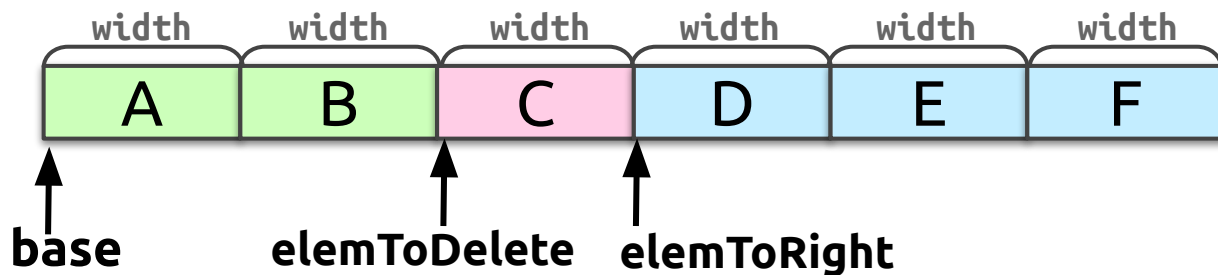
← Deletes **B** by sliding
elements **C**, **D** down





Generics Walkthrough: First Two Blanks

Task: Getting (1) a pointer to element to delete, (2) a pointer to the element right after, so we can slide it (and all other elements to the right) down.



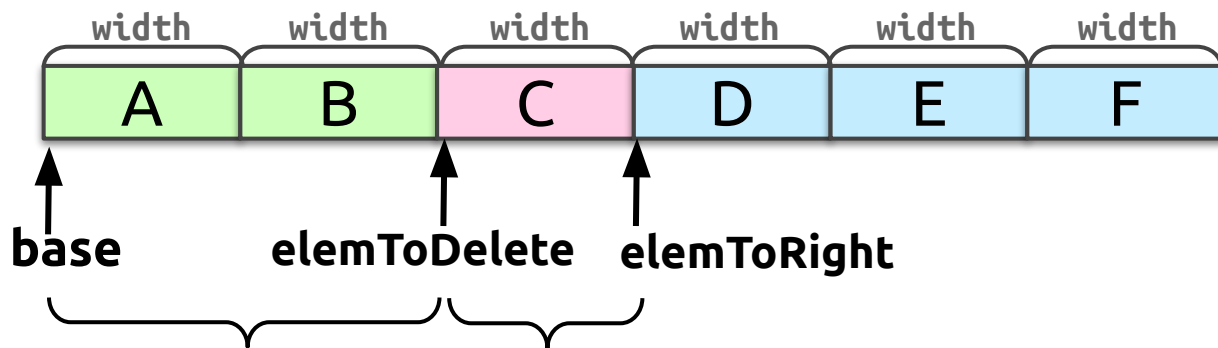
Example: Say we're deleting/yeeting element C at index 2.

```
void yeet(void* base, int index, int* nelems_ptr, int width){  
    void* elemToDelete = _____;  
    void* elemToRight = (char*) elemToDelete + _____;
```



Generics Walkthrough: First Two Blanks

Task: Getting (1) a pointer to element to delete, (2) a pointer to the element right after, so we can slide it (and all other elements to the right) down.



Example: Say we're deleting/yeeting element C at index 2.

Increment **index** *
width bytes from base

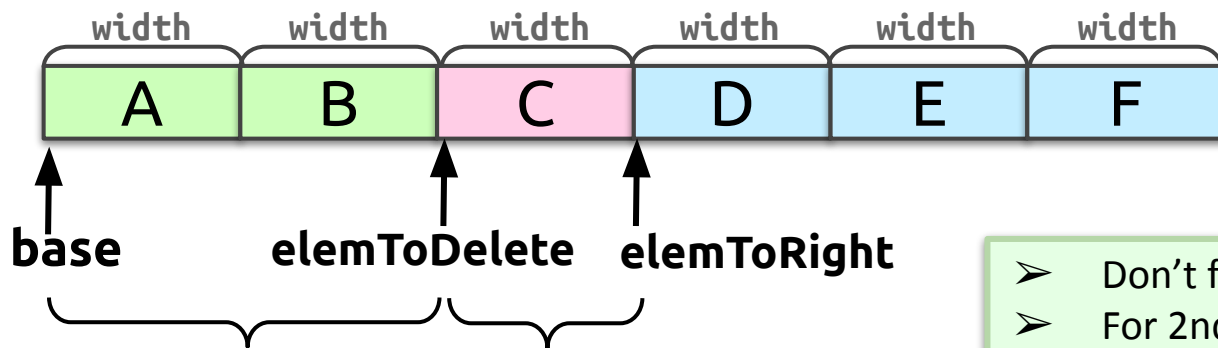
Increment by **width** more
bytes for 2nd pointer

```
void yeet(void* base, int index, int* nelems_ptr, int width){  
    void* elemToDelete = _____;  
    void* elemToRight = (char*) elemToDelete + _____;
```



Generics Walkthrough: First Two Blanks

Task: Getting (1) a pointer to element to delete, (2) a pointer to the element right after, so we can slide it (and all other elements to the right) down.



Increment **index** *
width bytes from base

Increment by **width** more
bytes for 2nd pointer

Example: Say we're deleting/yeeting element C at index 2.

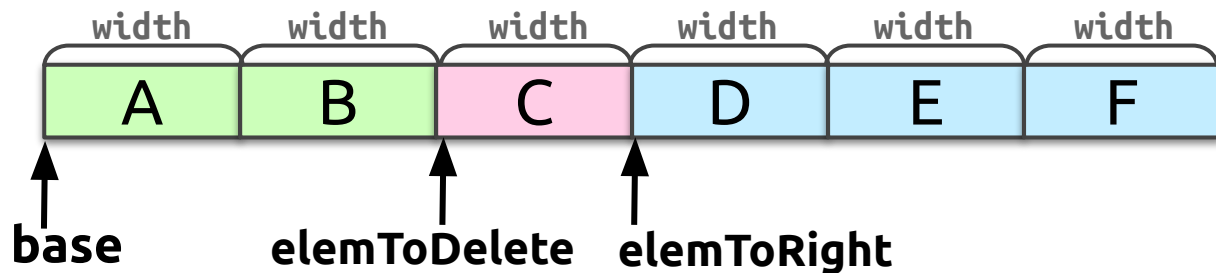
- Don't forget to cast **base** to **char***!
- For 2nd blank, we're incrementing from **elemToDelete** (not from **base**), so just **width** bytes forward!

```
void yeet(void* base, int index, int* nelems_ptr, int width){  
    void* elemToDelete = (char*) base + index * width;  
    void* elemToRight = (char*) elemToDelete + width;
```



Generics Walkthrough: Next Two Blanks!

(3) Sliding all elements to the right down, (4) updating number of elements.



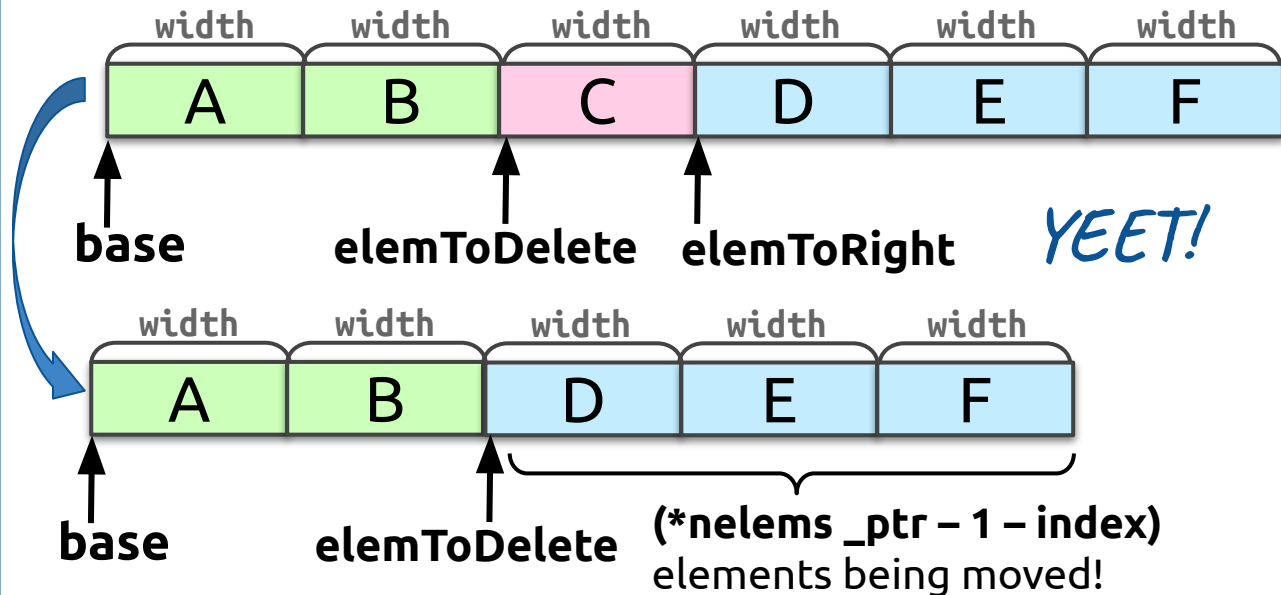
Example: Say we're deleting/yeeting element C at index 2.

```
_____  
_____; // don't forget to dereference, then decrement!
```



Generics Walkthrough: Next Two Blanks!

(3) Sliding all elements to the right down, (4) updating number of elements.



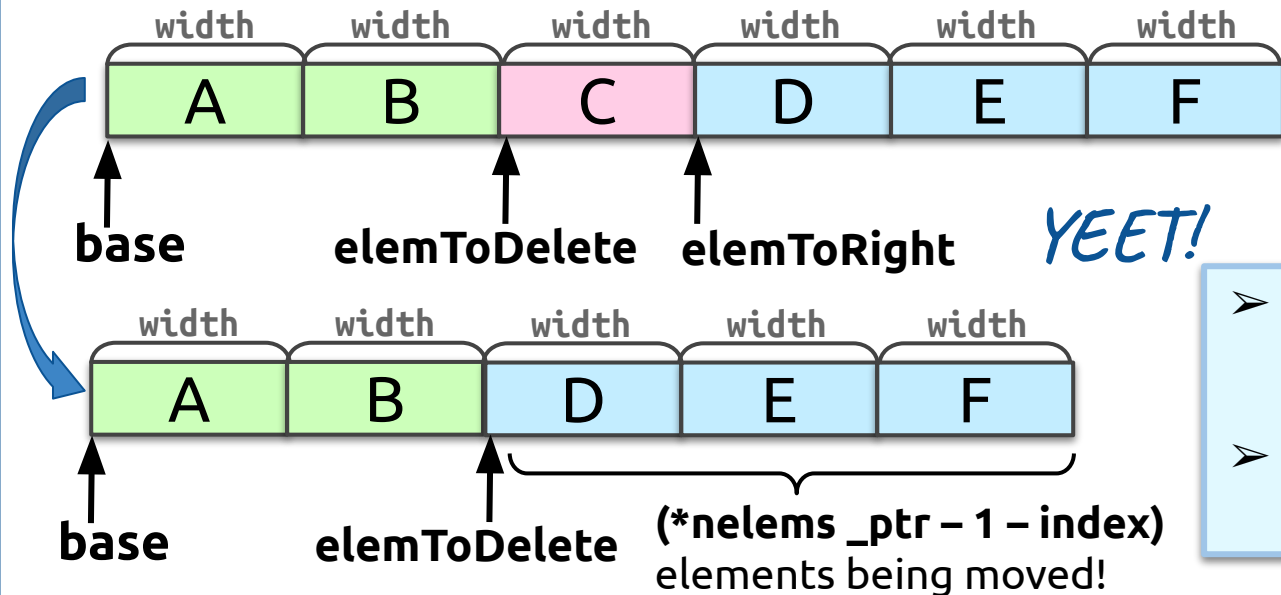
Example: Say we're deleting/yeeting element C at index 2.

```
_____  
_____; // don't forget to dereference, then decrement!
```



Generics Walkthrough: Next Two Blanks!

(3) Sliding all elements to the right down, (4) updating number of elements.



Example: Say we're deleting/yeeting element C at index 2.

- Use **memmove**, not **memcpy** due to memory regions overlapping
- Multiply by width to get # of bytes being moved!

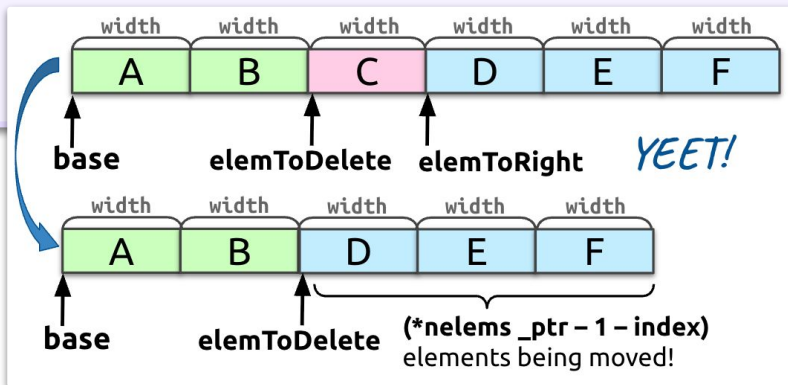
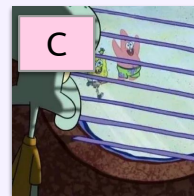
```
memmove(elemToDelete, elemToRight, (*nelems_ptr - 1 - index) * width) ;  
(*nelems_ptr) --; _____; // don't forget to dereference, then decrement!
```



Generics – Yeet that Element: Solution Code



```
void yeet(void* base, int index, int* nelems_ptr, int width){  
    void* elemToDelete = (char*) base + index * width ;  
    void* elemToRight = (char*) elemToDelete + width ;  
    // slide all elements to the right down  
    memmove(elemToDelete, elemToRight, (*nelems_ptr - 1 - index) * width) ;  
    // finally, update the number of elements  
    (*nelems_ptr) --; ;  
}
```



And with that, we're near the end of today's ride!



CS107 Scenic Recap: Jerry Cain High-Speed Train



**Bits, Bytes,
Integers!**

Exercise: Power of
Friendship

**C Strings &
Methods**

Exercise: Life
Comes Full Circle

**Pointers,
Arrays, Heap**

Exercise:
Malevolent Heap

**Generics, Void
Stars**

Exercise: Yeet
That Element

**Parting
Thoughts**

Teaser: Final Boss
Practice Problem!

I'll have all these materials uploaded to EdStem after this session!

Thank you all so much for being here, and happy studying ❤️



Ben's (I Promise Fun!) Final Boss Practice Problem

It's not that scary! I just wanted the problem to sound cool :)

Practice with strings, heap, malloc, structs, pointers in linked list!



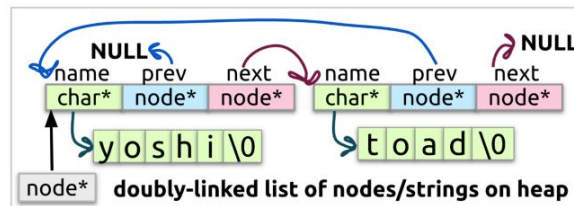
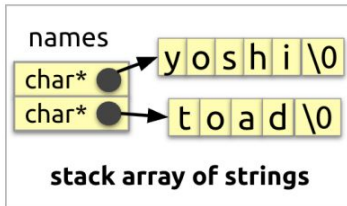
Solution on EdStem ✓

🔥 Problem 10: Switch It Up Like Nintendo (Dynamic Memory, Structs, ★Challenge)

Given a stack-allocated array **names** of valid C strings which represent character names (e.g., Yoshi, Kirby, Zelda, etc.), implement a function **switch_it_up_like_nintendo** that converts it into a dynamically-allocated, doubly-linked list of nodes of the following type:

```
typedef struct node {  
    char* name; // pointer for heap-allocated name string  
    node* prev; // pointer to previous node, NULL if none  
    node* next; // pointer to next node, NULL if none  
} node;
```

The list should preserve the order of the original array, and each node should be heap-allocated, along with each node's **name** string. The **prev** pointer of the starting node, and the **next** pointer of the last node, should be set to **NULL**. The function should return a pointer to the starting node. You can guarantee that the input array's length **num_names** ≥ 1 .



```
node *switch_it_up_like_nintendo(char* names[], int num_names){  
    // present your implementation below
```



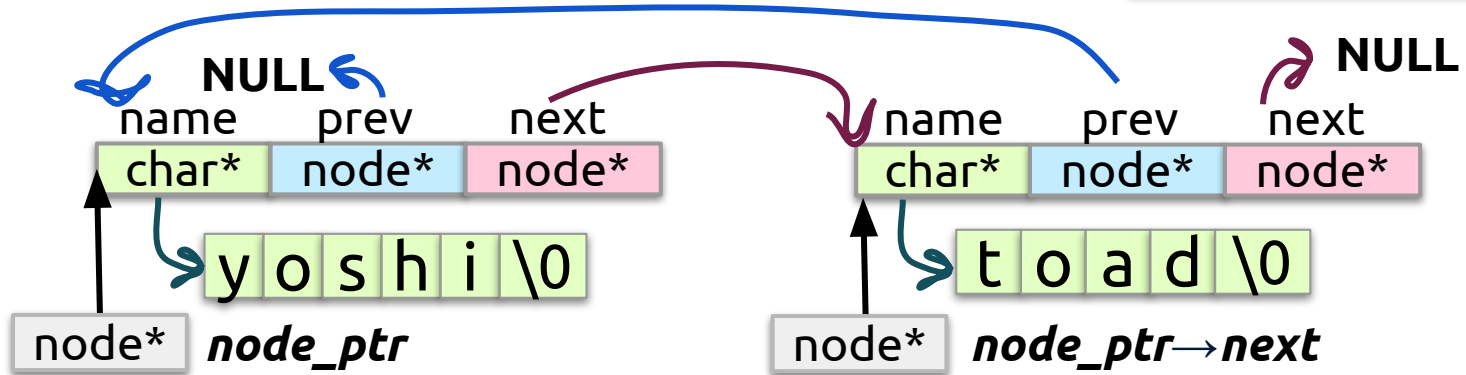
C Structs: The Power of Friendship

A struct **packs together several variables** / data types under a **single linear block of memory** with a name. Let's draw one!

🔗 **Example:** A doubly-linked list, where **each node is 24-byte struct** with a `char* name`, and pointers to `prev/next` nodes.

```
typedef struct node{
    char* name;
    node* prev;
    node* next;
} node;
```

name	prev	next
char*	node*	node*



Say you have a **node*** pointer **node_ptr** to the list's first node.

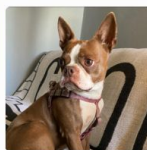
To get the name (yoshi), do **node_ptr->name**, which is a string ✓

To get pointers to prev and next nodes, **node_ptr->prev**, **node_ptr->next**, respectively, which are both **node*** ✓

Note: Arrow→ is for struct pointers. Use dot (.) if struct type itself, e.g., **(*node_ptr).name**



Julian Rodriguez Cardenas



Doris



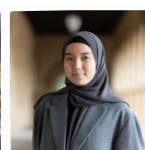
L'Hussen Toure



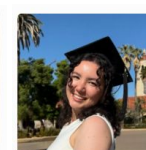
Yousef AbuHashem



Sureen Heer



Amira Mahmedjan



Naomi Boneh



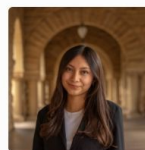
Jessica Chudnovsky



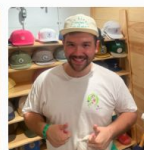
Isabel Berry



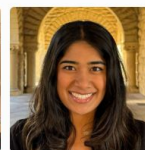
Ben Yan



Stephanie Hurtado



Kieran Barrett



Anushka Rawat



Kate Baker

Best wishes on the midterm!

You're 🖐️ going 🖐️ to 🖐️ be 🖐️ awesome,
and we're all rooting for you as a staff!



poohbear@stanford.edu

You slaying on the midterm

- **Happy to answer any questions!** I'll walk over to CoDa for improvised office hours until our staff meeting at 5PM if you'd like a group study space 📚❤️.
- **Exam kicks off on Wednesday:** Bring pencils or pens, erasers, and water!