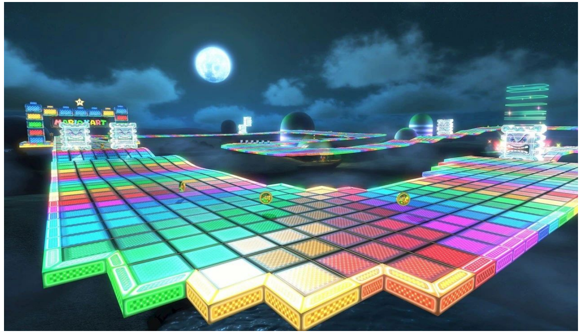
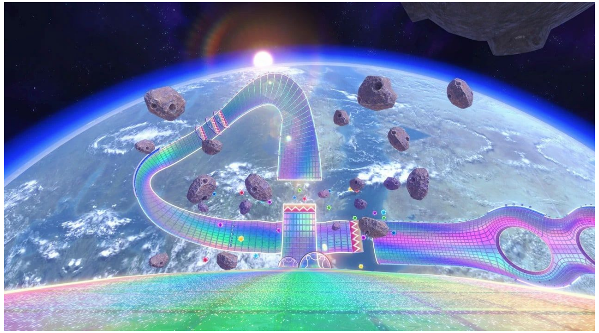
📽️ Slides link at stanford.edu/~bbyan/cs107





CS107 Final Review Session 🌠

Ben Yan << Spring 24-25 🌸

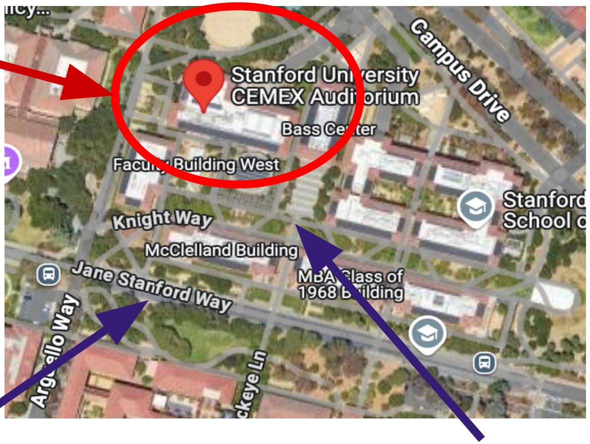
Slides adapted and remixed from Carolina Borbon Miranda, Sophie Andrews, and many wonderful TAs, past and present, for this class! :)

Welcome and thank you!

manifesting a
wonderful final
for you

Logistics

📍 **Exam here!**



Jane Stanford Way

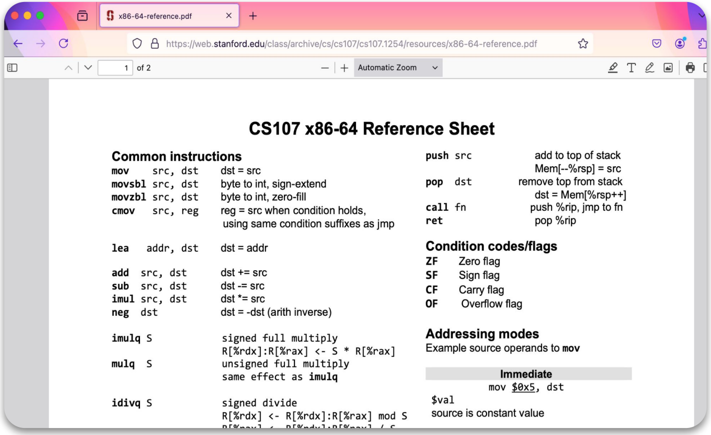
Stanford GSB

🧭 Place & Time

- ❑ Time: **Wednesday**, June 11, **8:30 AM – 11:30 AM (3 hrs)**
- ❑ Location: CEMEX Auditorium

⛵ Mechanics

- ❑ You won't be able to look at your own notes during the exam :(
- ❑ However! We provide a:
  - ❑ C reference sheet
  - ❑ Assembly reference sheet
- ❑ No need to bring them, we'll print both :)


x86-64-reference.pdf
https://web.stanford.edu/class/archive/cs/cs107/cs107.1254/resources/x86-64-reference.pdf
1
of 2
Automatic Zoom

CS107 x86-64 Reference Sheet

Common instructions

```
mov     src, dst    dst = src
movsbl  src, dst    byte to int, sign-extend
movzbl  src, dst    byte to int, zero-fill
cmov    src, reg    reg = src when condition holds,
                     using same condition suffixes as jmp

lea   addr, dst     dst = addr

add  src, dst       dst += src
sub  src, dst       dst -= src
imul src, dst       dst *= src
neg  dst            dst = -dst (arith inverse)

imulq S             signed full multiply
                    R[%rdx]:R[%rax] <- S * R[%rax]
mulq  S             unsigned full multiply
                    same effect as imulq

idivq S             signed divide
                    R[%rdx] <- R[%rdx]:R[%rax] mod S
                    R[%rax] <- R[%rdx]:R[%rax] / S
```

```
push src        add to top of stack
                Mem[--%rsp] = src
pop  dst        remove top from stack
                dst = Mem[%rsp++]
call fn         push %rip, jmp to fn
ret             pop %rip
```

Condition codes/flags

ZF	Zero flag
SF	Sign flag
CF	Carry flag
OF	Overflow flag

Addressing modes

Example source operands to **mov**

Immediate

`mov $0x5, dst`

`$val`

source is constant value

📚 Topics

For more detail on pre-midterm topics, see Carolina's wonderful **midterm review slides**!

❑ Final is **cumulative**, with a **focus on post-midterm material**

Binary, Data Representation	Bits and Bytes, ASCII, signed/unsigned, two's complements integers
Strings, Pointers, Stack/Heap	C strings (why char*), string functions, pointers, array and structs, stack arrays, heap memory (e.g., malloc, realloc), memory errors
Generics	void* pointers, memcpy/memmove, bsearch/qsort, cmp functions
Assembly + Stack Layout	Registers, addressing modes, assembly instructions, call/return, parameter passing, caller/callee protocol, layout of function stack
Heap Allocation	Allocator implementations, strategies and trade-offs
Optimization	Compiler optimizations, profiling (e.g., static vs dynamic instruction #)
Ethics	Disclosure policies, four degrees of partiality, privacy and trust

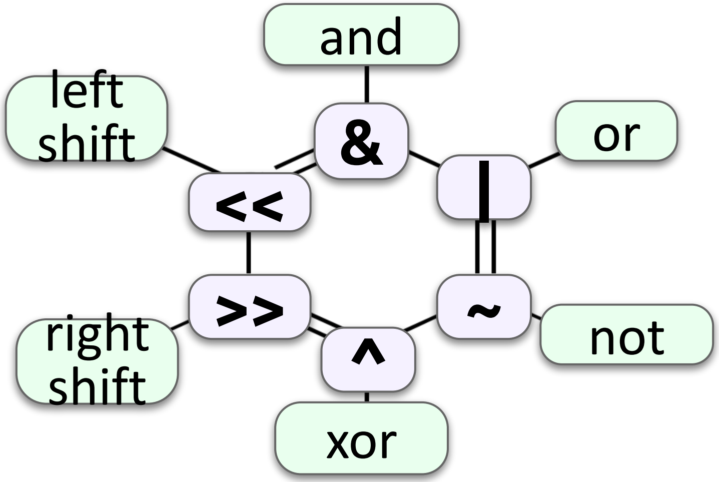
*Remember – **this is all stuff you've worked with**, and leveraged beautifully with your work on SecureVault, Heap Allocator, etc* 🔥. ***Keep doing what you do.***

👾 Bits and Bitmasks

Relevant throughout the material (everything is binary!), especially with allocators, e.g., for retrieving payload size/status stored in headers

👉 6 key bitwise operators — shown on the right!

🎭 Bitmasks are useful for **manipulating bits**



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 3rd lowest bit
Bitmask: OR (|) with (1 << 2)

num		1	…	0	1	0	0	1
bitmask	|	0	…	0	0	1	0	0
result		1	…	0	1	1	0	1

Note that all bits are **retained, except the bit (3rd LSB) we just turned on!**

Bits and Bitmasks

For **building bitmasks**, I recommend:

- ❖ Breaking it up in a few steps
- ❖ Constructing it from a bitmask 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
2^n	0	…	0	1	0	0	0	1<<n
2^n-1	0	…	0	0	1	1	1	(1<<n) -1

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

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 the bitmask is AND (&) with (**-1L << 4**)

How do we build this mask?

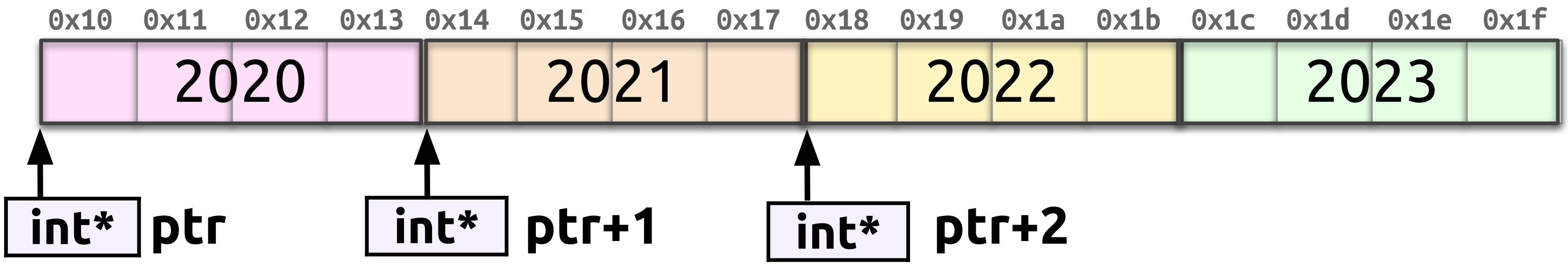
1	…	1	1	1	1	1	1

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

1	…	1	1	0	0	0	0

👉 Pointers and Pointer Arithmetic

A gentle reminder: Pointer arithmetic works in terms of the **size of the data type being pointed to** (e.g., +1 for an **int*** means advance 1 int, so 4 bytes)



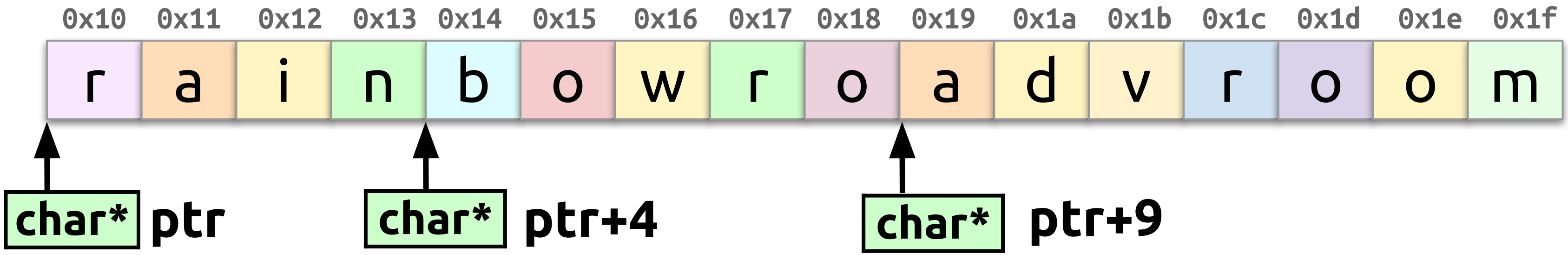
0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f
2020				2021				2022				2023			

int* **ptr**

int* **ptr+1**

int* **ptr+2**

Meanwhile, +1 for a **char*** means advance 1 char, so just 1 byte instead.
Takeaway: Be mindful of the pointer type you're working with!



0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f
r	a	i	n	b	o	w	r	o	a	d	v	r	o	o	m

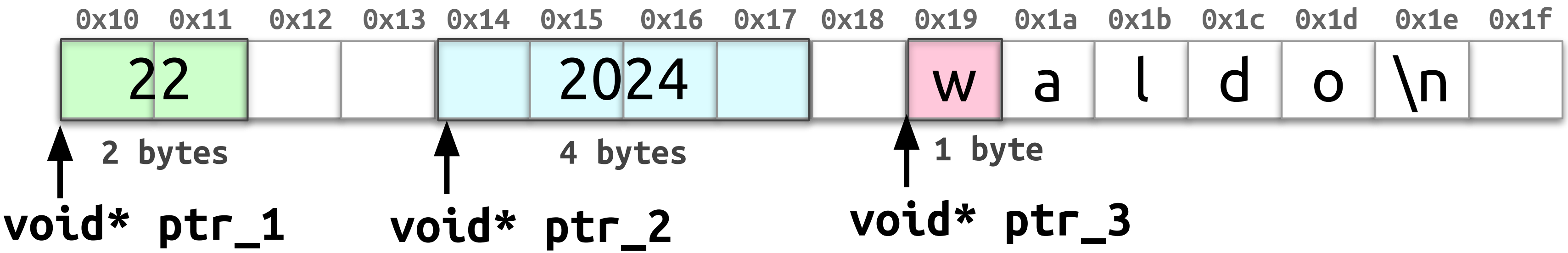
char* **ptr**

char* **ptr+4**

char* **ptr+9**

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.



0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f
22				2024					w	a	l	d	o	\n	

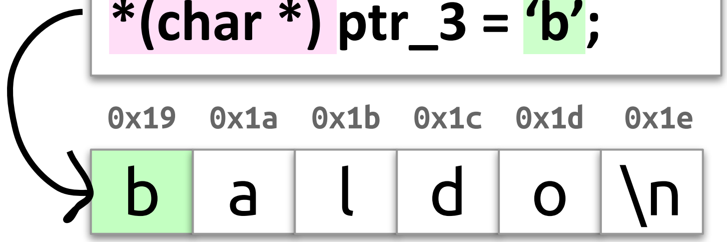
❌ **We can’t dereference void* pointers right away!** Don’t do this plzzz :(

Instead, to change memory a **void*** points to:

★ Use memcpy / memmove

★ Or, if we know what type is actually being pointed to, we can **first cast, then dereference**

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



0x19	0x1a	0x1b	0x1c	0x1d	0x1e
b	a	l	d	o	\n

👉 Generics: Pointer Arithmetic

Why do we cast to a **char*** for pointer arithmetic?

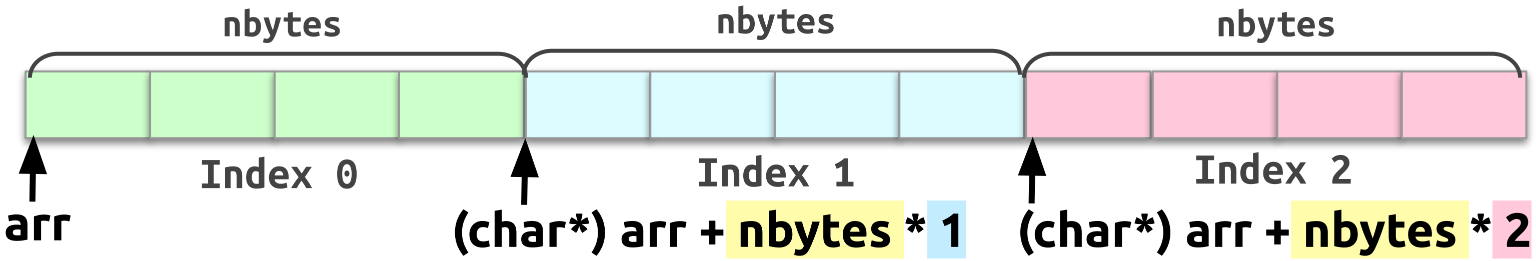
Consistency: Points to a **char** (only 1 byte), so we can work/add **in terms of bytes!**

Idiom: We have a **generic array arr** where each element has width **nbytes**.
How do we access / get a **pointer to the ith element** of the array?

```
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



nbytes nbytes nbytes

Index 0 Index 1 Index 2

arr

(char*) arr + nbytes * 1

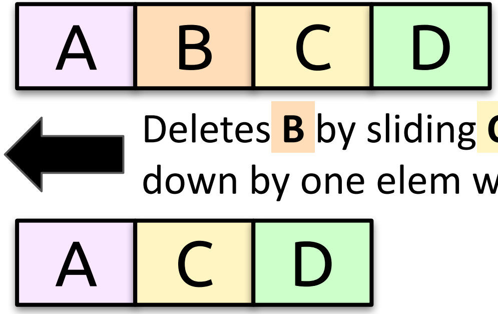
(char*) arr + nbytes * 2

✅ If you're unsure about an expression, try checking with a small value(s), e.g., here, for **i = 0**, make sure our expression is a **pointer to 0th element, or just arr itself**

A “Generic” Generics Problem Walkthrough

We want to write a generic function **yeet** that given an array **base** and **index**, deletes / “yeets” the element at that index—by sliding all the elements to the right of **index** down (the array base should stay in-place).

We assume 0 <= index < nelems. **Fill in the blanks.**



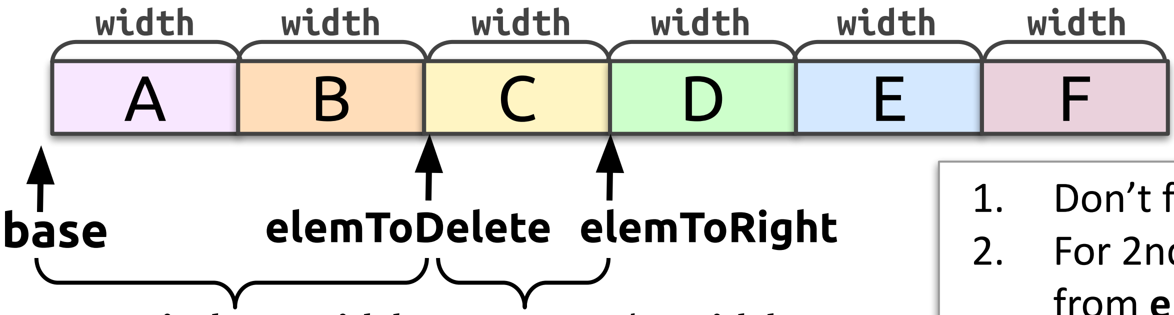
Deletes **B** by sliding **C, D** down by one elem width!

```
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
    ________________________;
}
```

Generics Walkthrough: First 2 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.

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



Increment **index * width** bytes from base

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

Example: Say we're deleting C (index 2)

1. Don't forget to cast **base** to **char***!
2. For 2nd blank, we're incrementing from **elemToDelete (not from base)**, so **width,** not (index + 1) * width

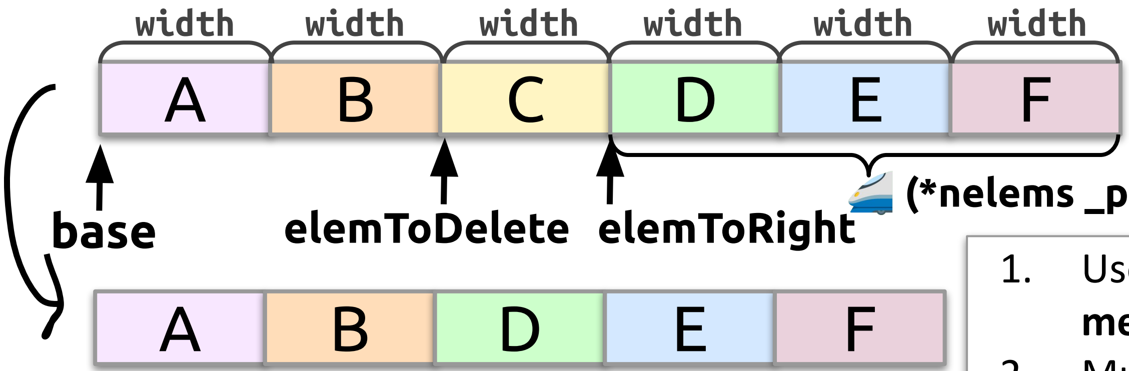
✅

```
void* elemToDelete = (char*) base + index * width;
void* elemToRight = (char*) elemToDelete + width;
```

Generics Walkthrough: Next 2 Blanks

Task: Sliding all elements to the right down, and updating number of elements.

```
void yeet(void* base, int index, int* nelems_ptr, int width){
    // previous 2 blanks getting elemToDelete and elemToRight pointers
    ________________________________; // slide all elements to the right down
    ________________________________; // finally, update the number of elements
```



Example: Say we're deleting C (index 2)

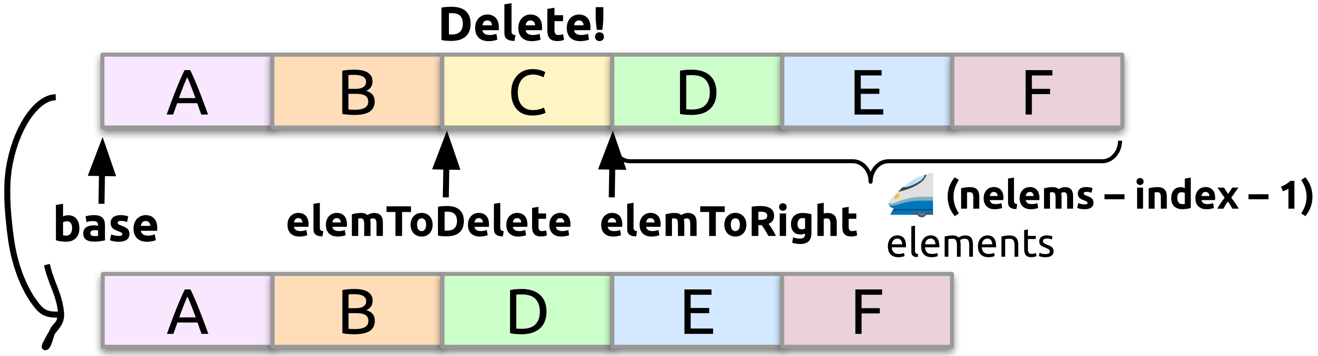
🚄 **(*nelems _ptr– index – 1)** elements being moved

1. Use **memmove**(dest, src, bytes), **not memcpy** due to memory overlap
2. Multiply by width to get # of bytes

✅ **memmove(elemToDelete, elemToRight, (*nelems_ptr – index – 1) * width);**
(*nelems_ptr) – – ; // don't forget to dereference the pointer!

Generics Walkthrough: And we’re done! 🎊

```
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 – index – 1) * width);
        // finally, update the number of elements
        (*nelems_ptr) – – ;
}
```



Delete!

A B C D E F

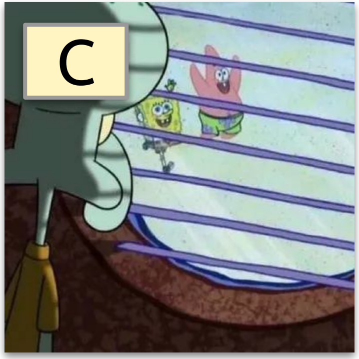
base

elemToDelete

elemToRight

🚄 **(nelems – index – 1)** elements

A B D E F



Notes on memcpy, memmove, bsearch, etc.

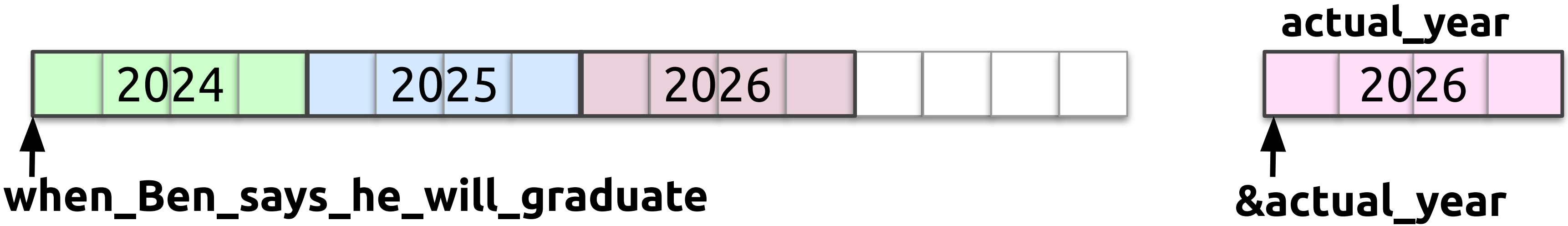
👉 When calling generic functions, you typically **pass pointers to the data** you're trying to copy, modify, compare, etc., **not the data value itself**. Though to be sure, you can check the **C reference guide** for function specs!

```
int[] when_Ben_says_he_will_graduate = {2024, 2025, 2026};
int actual_year = 2026;  // 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, …)



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

👉 We need pointer to key

Comparison Function: 3 Key Steps

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

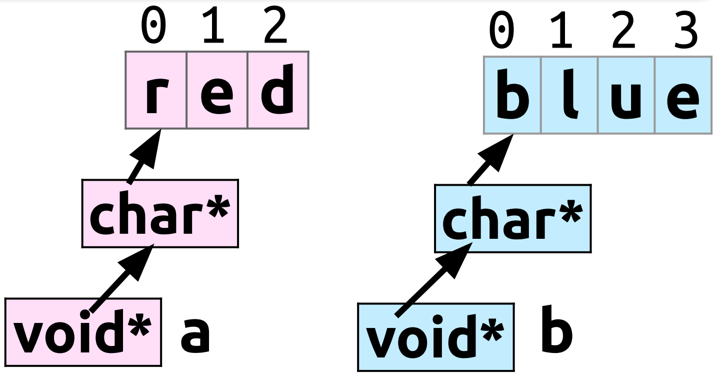
mycmp(a,b) should return:

`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!

Let's write a function to compare **strings** in order of **ascending first character**!

```
int mycmp(const void *a, const void *b){

}
```



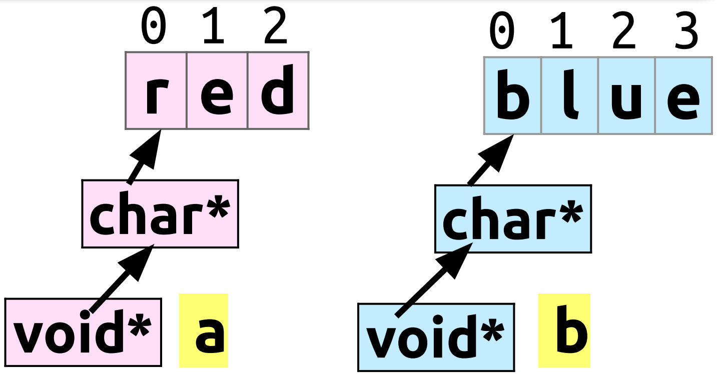
📊 Comparison Function: 3 Key Steps

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

Note: Steps 1 and 2 are often combined to cast and deference in one expression!

Let's write a function to compare **strings** in order of **ascending first character**!

```
int mycmp(const void *a, const void *b){
                              (const char**) a;
                              (const char**) b;


}
```



👉 *Note the inputs a and b are pointers to strings (char*), so cast to char***

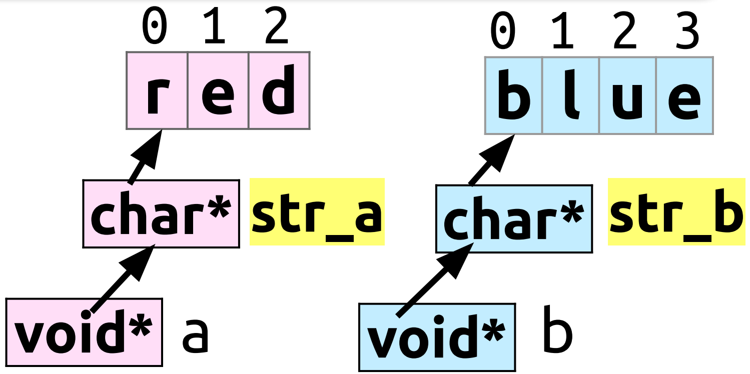
Comparison Function: 3 Key Steps

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

Note: Steps 1 and 2 are often combined to cast and deference in one expression!

Let’s write a function to compare **strings** in order of **ascending first character**!

```
int mycmp(const void *a, const void *b){
    const char* str_a = *(const char**) a;
    const char* str_b = *(const char**) b;


}
```



0 1 2

r e d

char* str_a

void* a

0 1 2 3

b l u e

char* str_b

void* b

👉 *Note the inputs a, b are char** pointers*

Comparison Function: 3 Key Steps

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

mycmp(a,b) should return:

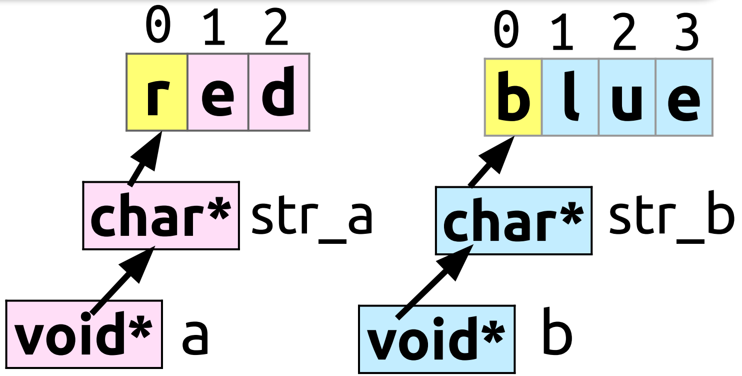
0	if **a** and **b** are **equal**
<0	if **a comes before b**
>0	if **a comes after b**

Note: Steps 1 and 2 are often combined to cast and deference in one expression!

Let's write a function to compare **strings** in order of **ascending first character**!

```
int mycmp(const void *a, const void *b){
    const char* str_a = *(const char**) a;
    const char* str_b = *(const char**) b;
    return str_a[0] – str_b[0];
}
```

Can also write ***str_a – *str_b** instead



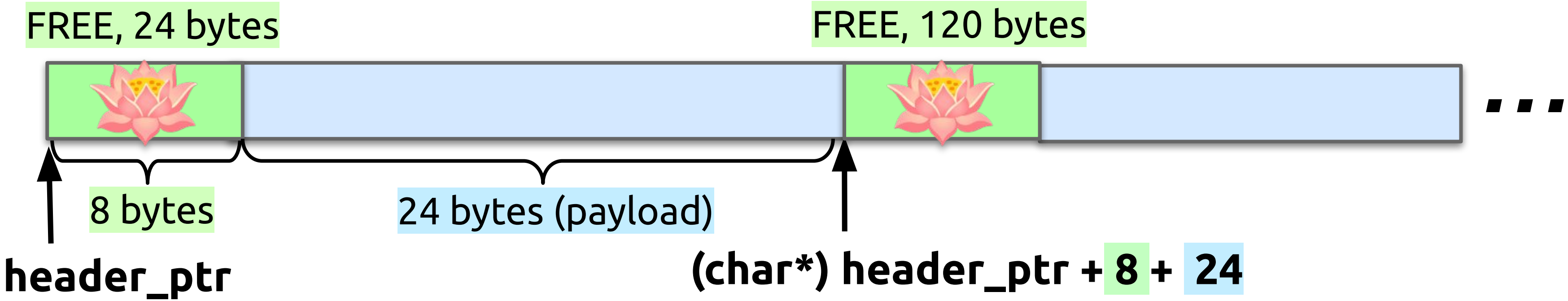
👉 *Note the inputs a, b are char** pointers*

Questions on generics, comparison functions?

Next up: Heap allocation!

📦 Heap Allocator

🎨 **Tip: CS107 is as much an art class as a programming class (imo) – so make a sketch / example!** This can help make sure you're updating payloads and moving pointers by the right number of bytes, not forgetting header bytes 🪷, etc.



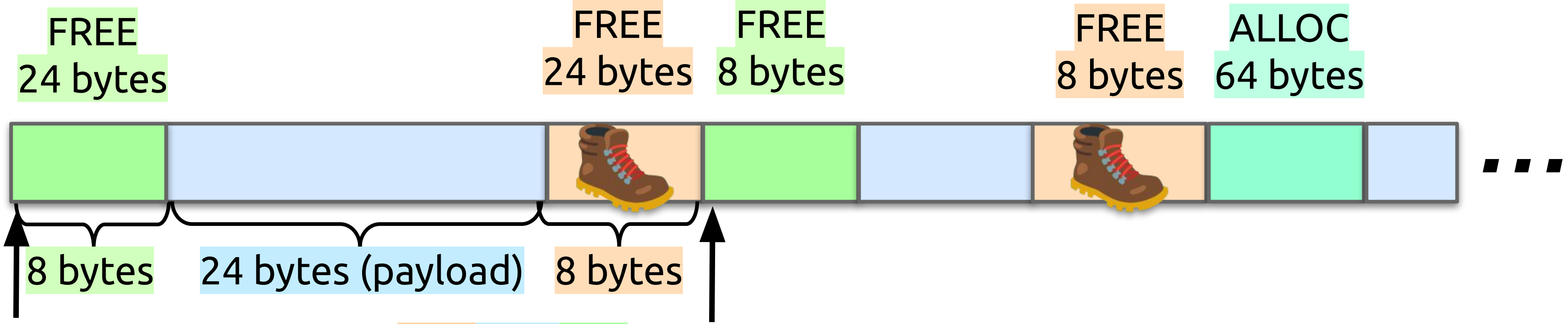
👉 *To get to the next header, we moved the pointer by 32 bytes total here*

Tip: I would also review Assign6 code, not to memorize it 😭, but to re-familiarize with the high-level details of, e.g., navigating headers, extracting header size/status

📦 Heap Allocator

Example: Now, suppose that each heap block also stores an 8-byte footer 🥾, starting precisely where the payload ends (and before the next header 🪷 begins).

🤔 *How can we move backwards from a header to the previous header?*



(char*) header_ptr – 8 – 24 – 8 **header_ptr**

👉 *To get to the previous header, we **moved the pointer back** by **40 bytes** total here*

Takeaway: Draw an example! And visually check your pointers are pointing to the right place (e.g., header_ptr doesn't point to payload by accident) **Then, generalize to code.**

Heap Allocator: General Concepts

Throughput: How fast can the allocator service requests?

Utilization: How efficiently/tightly can the allocator use the segment space?

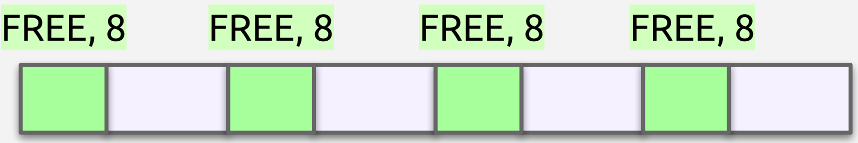
Fragmentation: Primary cause of poor utilization, occurring when unused memory is unable to satisfy allocation requests. External vs Internal

External Fragmentation

Lots of free memory but split across many small free blocks → can't service one large request

Request: Hey, a 200-byte payload please?

Allocator: Sorry it's absolutely cooked

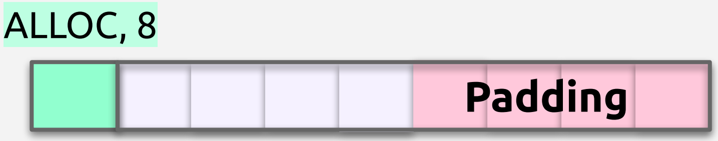


Internal Fragmentation

More space is allocated for a used block than necessary (e.g., padding)

Request: Ok, just 4 bytes please?

Allocator: Here's 8, keep the change



📦 Allocator Designs and Trade-Offs

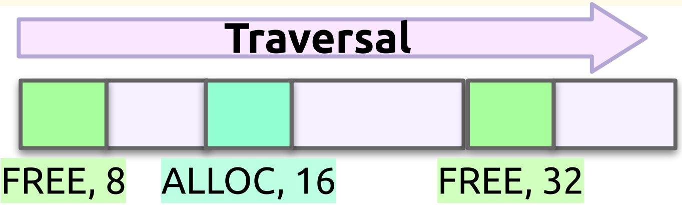
⚖️ It's hard for an allocator to have both high **throughput** and **utilization** (e.g., may take longer to search for a good block) – **finding an appropriate balance is key!**

🐦‍🔥 Design Choice: Maintain **list of free blocks** to reuse in the future

😊 Implicit Allocator

📌8-byte header with payload size + used/free

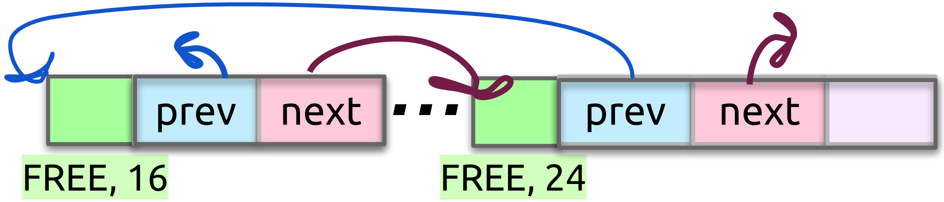
📌Traverse both free/used blocks for request



😫 Explicit Allocator

📌Also 8-byte header with payload size + used/free

📌Free block tracked in **linked list**, with 8-byte prev/next pointers stored in payload (>=16 bytes)



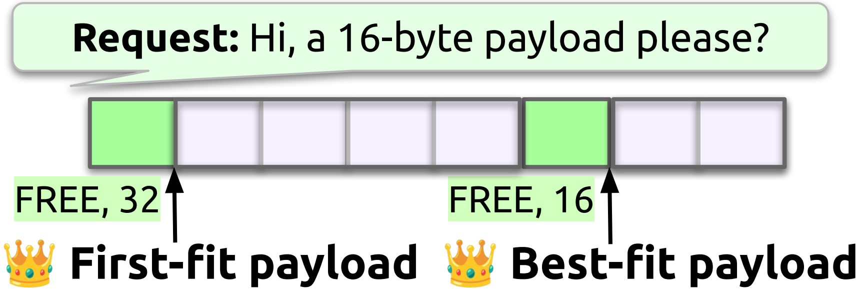
Some Benefits and Downsides of Each

- **Explicit only has to navigate through free blocks**, improving efficiency in most cases
- But **explicit requires larger minimum payloads** → more padding, internal fragmentation
- **Implicit has simpler design** than explicit, with more straightforward linear traversal

Allocator Designs and Trade-Offs

First-fit vs best-fit trade-offs?

- **First-fit is generally faster** – just find the first block that's free and large enough
- **Best-fit** has less throughput (speed), but **tends to reduce external fragmentation**, by avoiding splitting large, versatile blocks



Request: Hi, a 16-byte payload please?

FREE, 32

FREE, 16

👑 **First-fit payload**

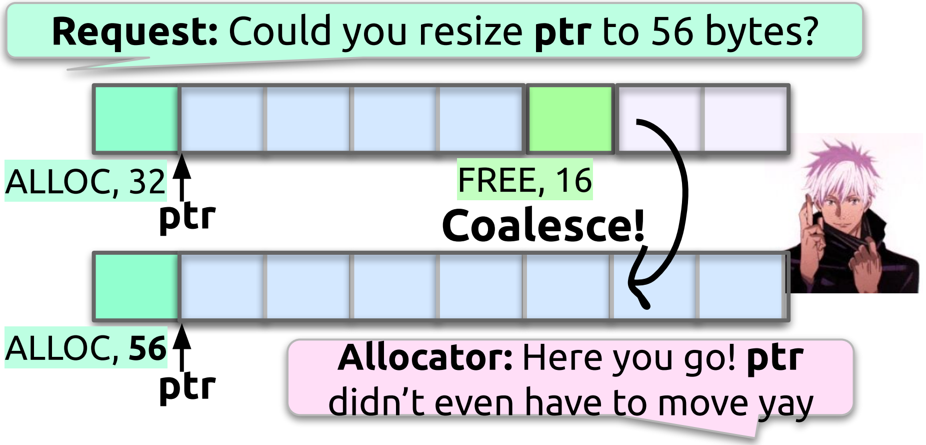
👑 **Best-fit payload**

Explicit free list: Order by address, size, adding new nodes to front or back?

- **E.g., Last-in first-out** makes adding to list quick (just add new node to head/front of list)

Coalescing – what's the point?

- **In-place realloc** by absorbing free blocks to right
- **Reduces external fragmentation** by merging small free blocks together into a large block



Request: Could you resize **ptr** to 56 bytes?

ALLOC, 32

ptr

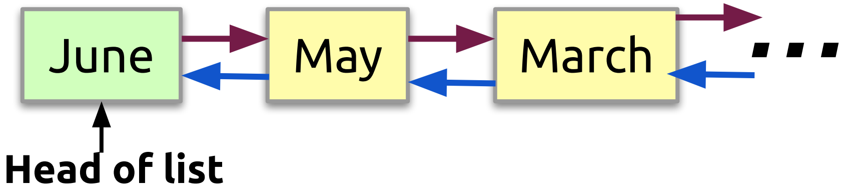
FREE, 16

Coalesce!

ALLOC, **56**

ptr

Allocator: Here you go! **ptr** didn't even have to move yay



June

May

March

Head of list

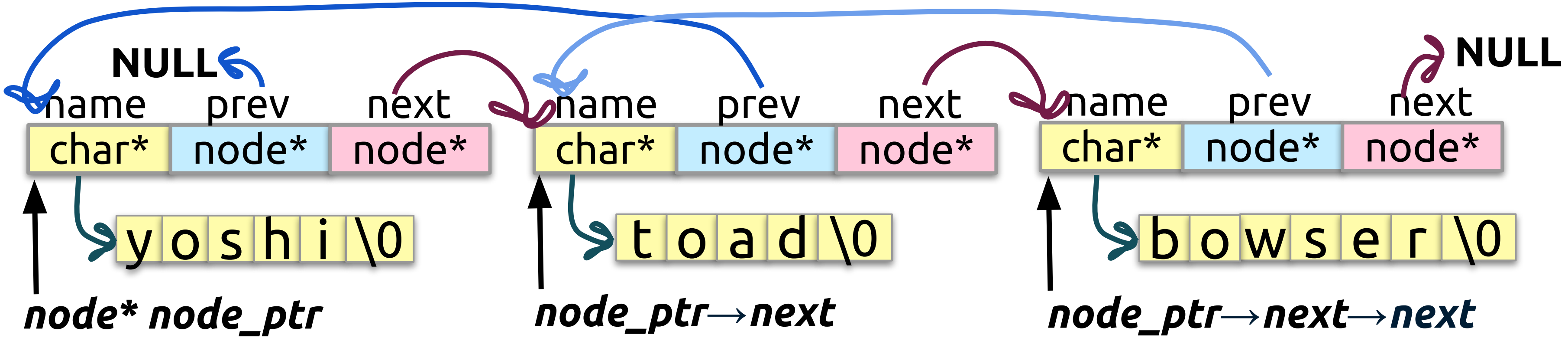
Linked List: Structs!

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

Example: A doubly-linked list of names, where each node stores 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*



NULL

NULL

name	prev	next
char*	node*	node*

y o s h i \0

t o a d \0

b o w s e r \0

node* node_ptr

node_ptr→next

node_ptr→next→next

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**

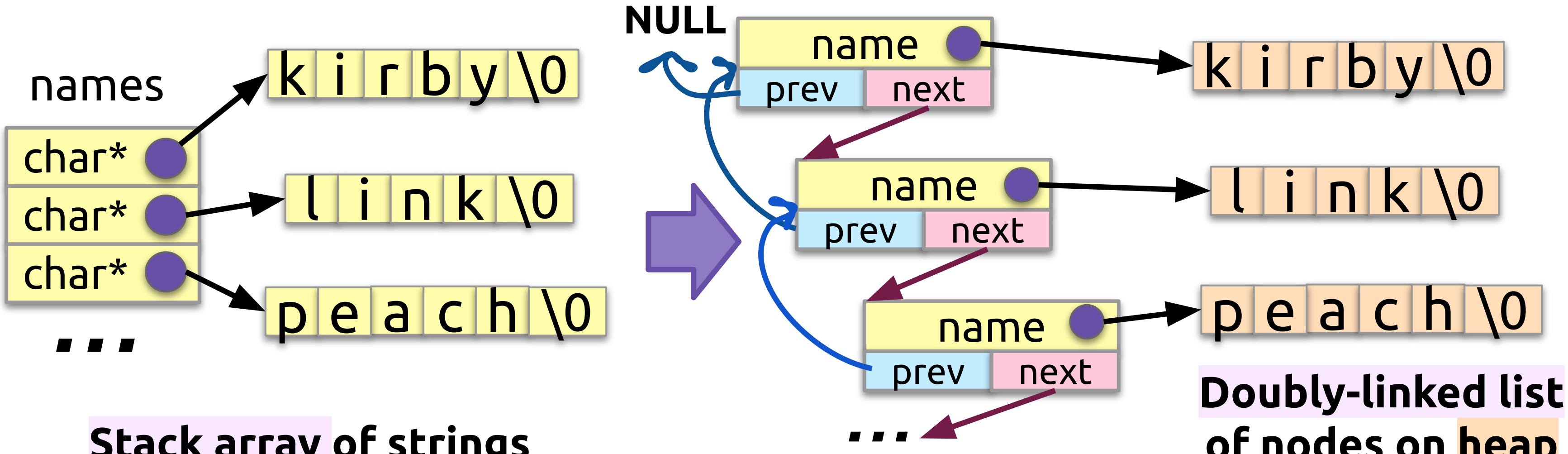
🔗 Linked List and Struct Pointers Example

We want to write a function **switch_ituplike_nintendo** that takes a stack array of name strings, and converts it into a doubly-linked list of **node structs, one per name.**

📌 Each **node** should be heap-allocated, along with each node’s **name / string.**

📌 The function should **return a pointer to the first node** in the list.

node* switch_ituplike_nintendo(char** names, int nelems)



Stack array of strings

Doubly-linked list of nodes on heap

🔗 Linked List and Struct Pointers Example

```
node* switch_ituplike_nintendo(char** names, int nelems){









}
```

Let’s get started!

(1) What kinds of **control structures** (e.g., loops) might we need?

(2) What **functions** (e.g., malloc) might we use for **heap allocation**?

(3) **What variables** might we want to track to **wire up prev/next pointers** in the list correctly?

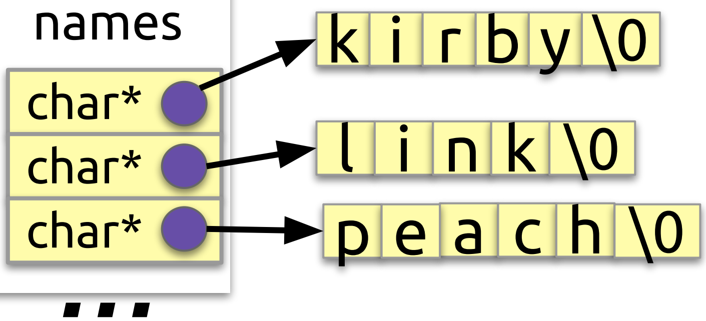
(4) What do we need to **return at the end**?

🔗 Linked List: Overall Code Structure, Goal

```
node* switch_ituplike_nintendo(char** names, int nelems){
        node* headNode = NULL;

        for (int i = 0; i < nelems; i++){




        }

        return headNode;
}
```

We need to return a **node* pointer to the first / head node** of the list, so let's initialize it to **NULL** first!

Overall Structure: We'll have to iterate through each string in the stack array, so **let's create a loop!**



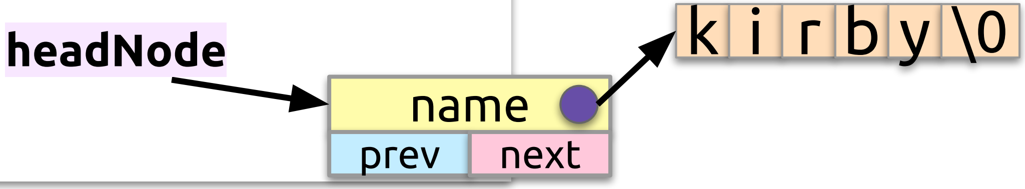
🔗 Linked List: Behavior Inside Loop

```
node* switch_ituplike_nintendo(char** names, int nelems){
        node* headNode = NULL;

        for (int i = 0; i < nelems; i++){
                // heap-allocate each node, and its name string
                node* curNode = malloc(sizeof(node));
                curNode→name = strdup(names[i]);

                If (headNode == NULL) headNode = curNode;



        }

        return headNode;
}
```

Each list node: We need to malloc the node first, which is sizeof(node) bytes

Each node's string: We also have to heap-allocated the ith string, so use strdup, and wire the pointer to node→name

Lastly, if the **head node** hasn't been set yet (first node), set it to be this node



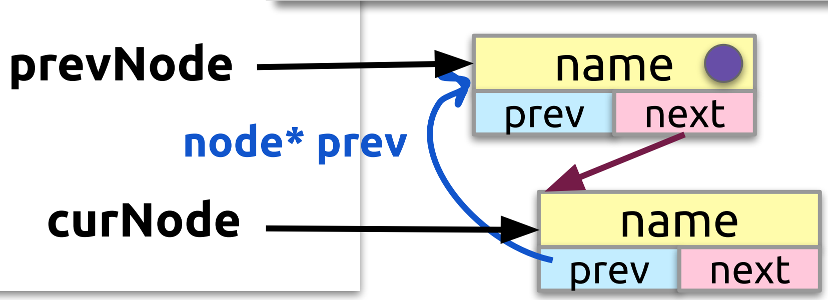
Linked List: Wiring up the list, Part 1

```
node* switch_ituplike_nintendo(char** names, int nelems){
        node* headNode = NULL;
        node* prevNode = NULL; // prev of first node is NULL
        for (int i = 0; i < nelems; i++){
                // heap-allocate each node, and its name string
                node* curNode = malloc(sizeof(node));
                curNode→name = strdup(names[i]);
                // wire up the current node to linked list
                If (headNode == NULL) headNode = curNode

                curNode→prev = prevNode;
                prevNode = curNode;
        }

        return headNode;
}
```

Let’s go ahead and set the **prev** pointer of each node.

We want to keep a **lagged node* variable prevNode** that tracks the node in the prior iteration.

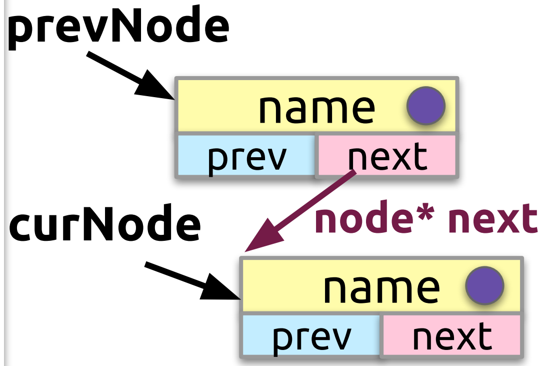
We then **set curNode→prev to be this previous node**, and lastly, update prevNode



🔗 Linked List: Wiring up the list, Part 2

```
node* switch_ituplike_nintendo(char** names, int nelems)
    node* headNode = NULL;
    node* prevNode = NULL; // prev of first node is NULL
    for (int i = 0; i < nelems; i++){
        // heap-allocate each node, and its name string
        node* curNode = malloc(sizeof(node));
        curNode→name = strdup(names[i]);
        // wire up the current node to linked list
        If (headNode == NULL) headNode = curNode
        if (prevNode != NULL) prevNode→next = curNode;
        curNode→prev = prevNode;
        prevNode = curNode;
    }
    prevNode→next = NULL; // next of last node is NULL
    return headNode;
}
```

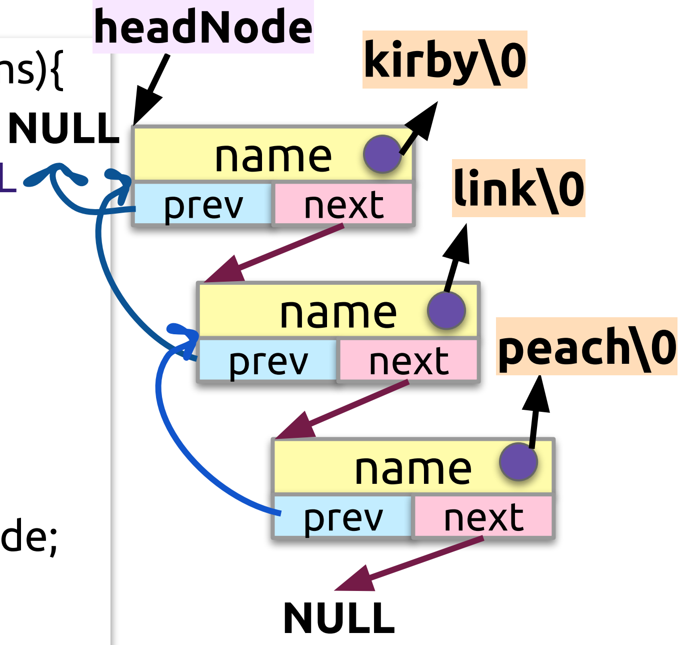
Now, let's set the **next** ptr, which **we can only do in the next iteration** (when we know who the next node is).



At the end, prevNode will be the last node in the list, so set its next to be **NULL**

🕹️ Switch it up: And we’re done! 🎊

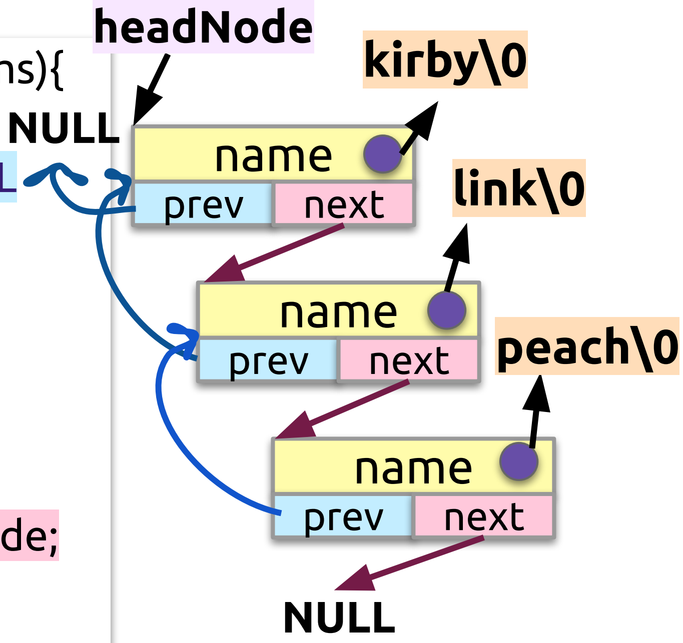
```
node* switch_ituplike_nintendo(char** names, int nelems){
    node* headNode = NULL;
    node* prevNode = NULL; // prev of first node is NULL
    for (int i = 0; i < nelems; i++){
        // heap-allocate each node, and its name string
        node* curNode = malloc(sizeof(node));
        curNode→name = strdup(names[i]);
        // wire up the current node to linked list
        If (headNode == NULL) headNode = curNode;
        if (prevNode != NULL) prevNode→next = curNode;
        curNode→prev = prevNode;
        prevNode = curNode;
    }
    prevNode→next = NULL; // next of last node is NULL
    return headNode;
}
```



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

🕹️ Switch it up: And we’re done! 🎊

```
node* switch_ituplike_nintendo(char** names, int nelems){
        node* headNode = NULL;
        node* prevNode = NULL; // prev of first node is NULL
        for (int i = 0; i < nelems; i++){
                // heap-allocate each node, and its name string
                node* curNode = malloc(sizeof(node));
                curNode→name = strdup(names[i]);
                // wire up the current node to linked list
                If (headNode == NULL) headNode = curNode;
                if (prevNode != NULL) prevNode→next = curNode;
                curNode→prev = prevNode;
                prevNode = curNode;
        }
        prevNode→next = NULL; // next of last node is NULL
        return headNode;
}
```



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

Questions on heap allocation or structs?

Next up: assembly!!

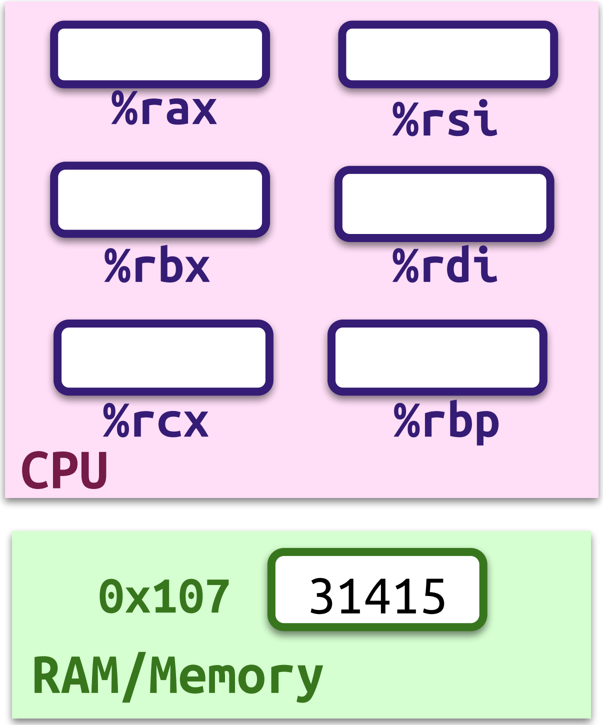
📮 Assembly and Registers

The **compiler** converts our **C code** into low-level **assembly instructions**, which can then be converted into machine code run by your computer.c

📝 Think in terms of **registers**: variable boxes used as temporary storage / **scratch paper** for programs.

⚙️ A register is an **very fast read/write slot** right on the CPU (not in memory), which can hold variable

📬 **Move data in & out of registers** to perform operations on it with the CPU, e.g., arithmetic operations, parameters into functions, return values



Key Instruction: mov src, dest

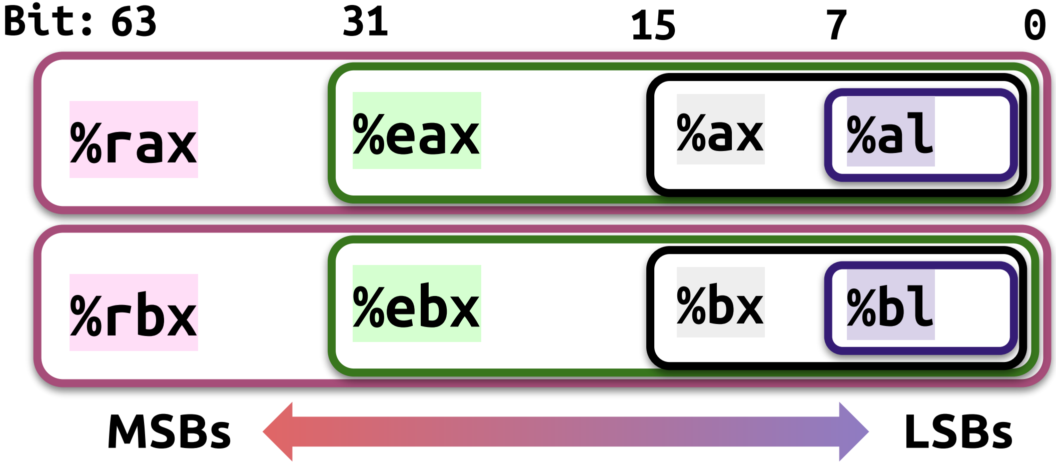
- Responsible for moving data around!
- **src/dest** can be register or memory location, **but not both memory locations** (must move into register first)
- **src** can also be immediate (constant value, e.g., 0x107)

📮 Register Sizes

🗃️ Registers can be **partially or fully referred to**. For instance, we can access just 32 bits of a register (e.g., $edi) instead of the full 64 bits (e.g., $rdi).

Specifically, a reference to the **e-named register** accesses the **lower 32-bit subregister**

`%edi` is subregister of `%rdi`
`%eax` is subregister of `%rax`


Bit: 63 31 15 7 0

%rax %eax %ax %al

%rbx %ebx %bx %bl

MSBs **LSBs**

When do you want 32 bits (4 bytes) versus 64 bits (8 bytes)?

Tl;dr if you don’t need the extra bits. As examples:

- Operations on **pointers and longs** (64 bits) typically use the **full r-named registers**
- Whereas operations on **ints** use the **e-named registers.**

Assembly: Addressing Modes

Immediate: mov $0x5, dst

A **dollar sign ($)** indicates source is **constant value** to be moved into dst

Direct: mov 0x106b, dst

Fixed, static memory address

This reads from **Mem[0x106b]**, the memory at address 0x106b.

Register: mov **$rax**, dst

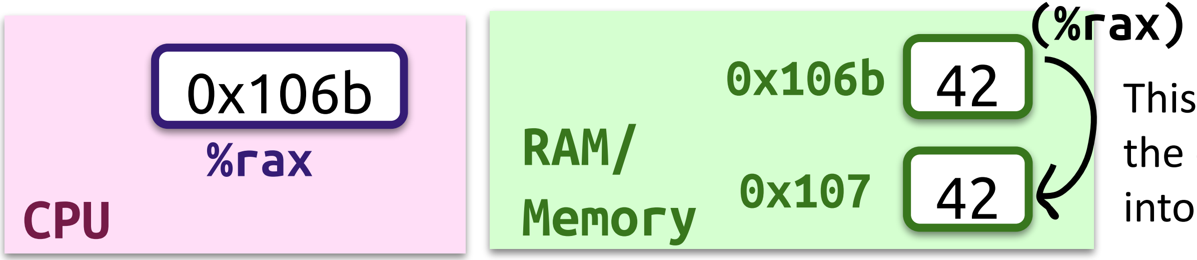
Moves **value held in register** to dst.

Indirect: mov **($rax)**, dst

This reads from **Mem[$rax]**, where the register $rax holds an address, e.g., 0x106b

Note **parentheses** to contrast with above.

Example: Indirect, with **mov ($rax), 0x107**, where **$rax** holds **address 0x106b**



This is effectively saying, move the data at **address 0x106b/$rax** into **address 0x107**.

📫 Addressing Modes

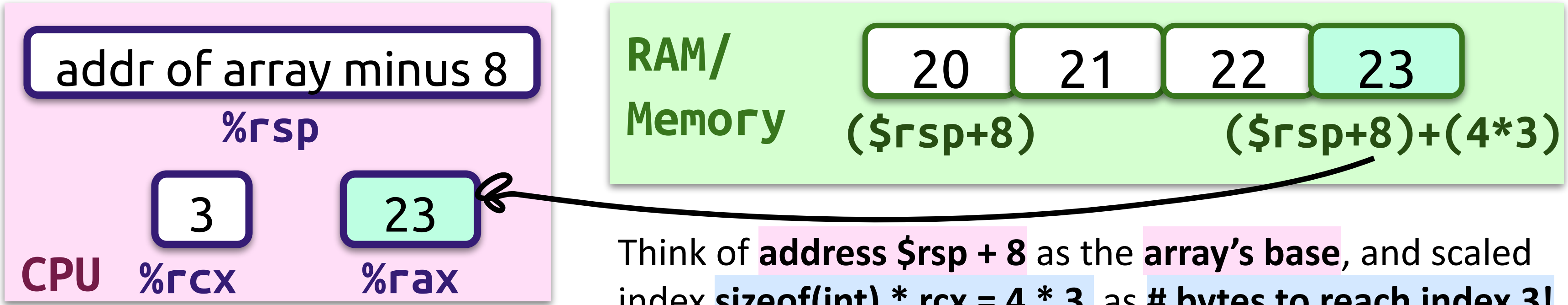
Indirect with Displacement: mov 8($rax), dst

Same as indirect, but with a **number prefix as bytes to offset the address** (think pointer arithmetic). The above reads from **Mem[$rax + 8]**, not Mem[$rax * 8]!

Indirect with Displacement + Scaled Index: mov 8($rsp, $rcx, 4), dst

More complex displacement, with **offset** + **scaling factor** (e.g., 1,2,4,8). Seen often when reading from arrays. This reads from **Mem[$rsp + 8 + (4 * $rcx)].**

Arrays Example: mov 8($rsp, $rcx, 4), $rax,** where **$rcx holds the value/index **3**

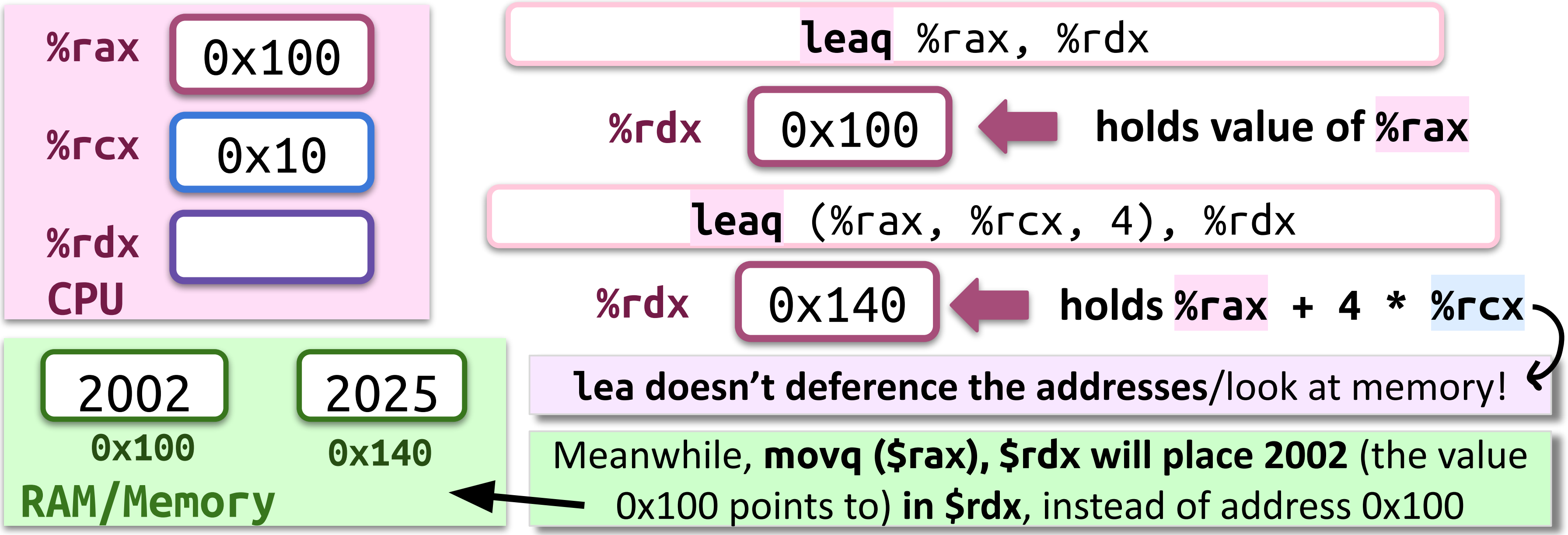


Think of **address $rsp + 8** as the **array's base**, and scaled index **sizeof(int) * rcx = 4 * 3** as **# bytes to reach index 3!**

📬 lea instruction: load effective address

Key Instruction: lea src, dest

- Similar to mov, except that it doesn’t dereference, e.g., **it copies of the value of src itself** to the destination, **not the data at the address src (which mov does).**
- It’s really just pointer arithmetic! 🤔 (also used for regular arithmetic)



leaq %rax, %rdx

holds value of %rax

leaq (%rax, %rcx, 4), %rdx

holds %rax + 4 * %rcx

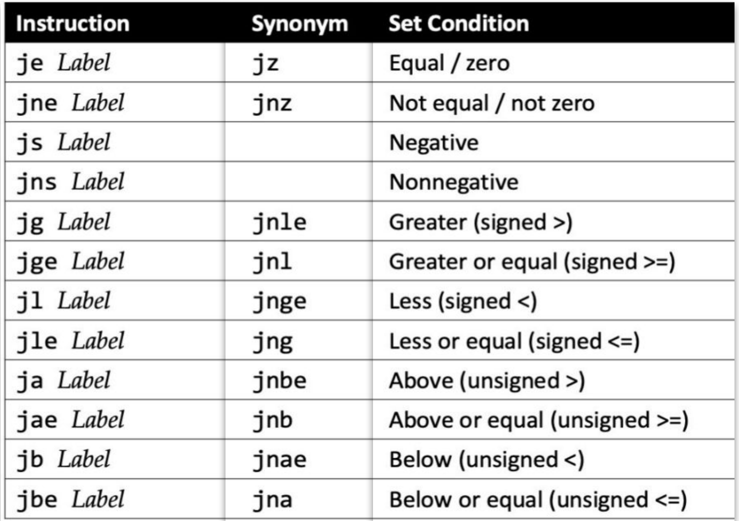
lea doesn’t deference the addresses/look at memory!

Meanwhile, **movq ($rax), $rdx will place 2002** (the value 0x100 points to) **in $rdx**, instead of address 0x100

📬 jumps and conditional jumps

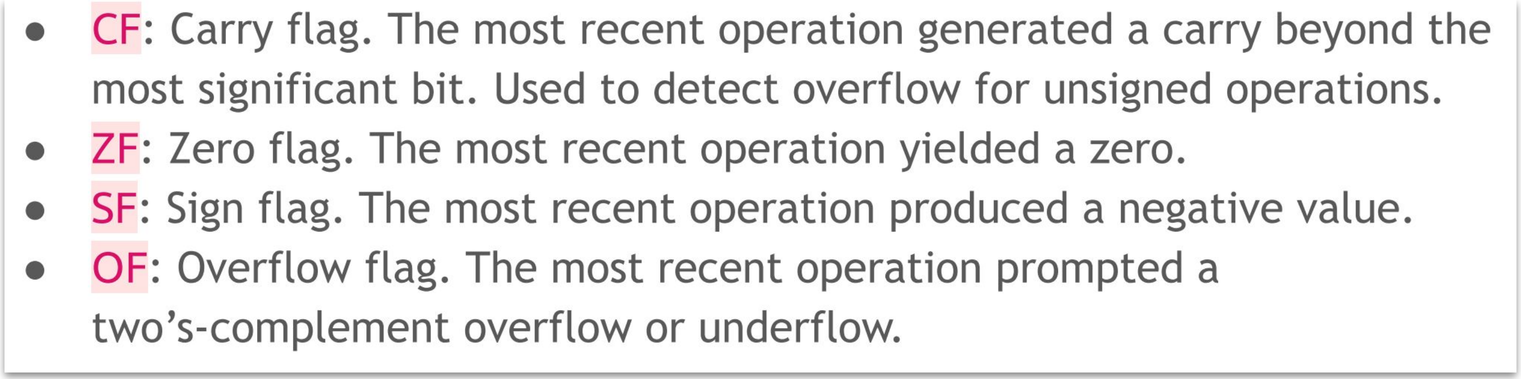
Some jumps (**jmp <address-of-next-instruction>**) are unconditional – when that **jmp** instruction is reached, the program will take the jump.

There are **variants of jmp that are conditional**: take the jump to the instruction address **if and only if certain conditions are met**.

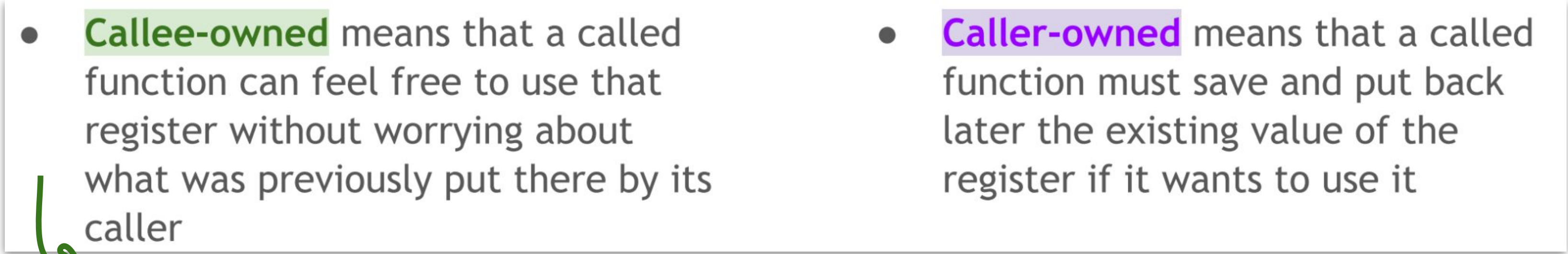


Instruction	Synonym	Set Condition
`je` *Label*	`jz`	Equal / zero
`jne` *Label*	`jnz`	Not equal / not zero
`js` *Label*		Negative
`jns` *Label*		Nonnegative
`jg` *Label*	`jnle`	Greater (signed >)
`jge` *Label*	`jnl`	Greater or equal (signed >=)
`jl` *Label*	`jnge`	Less (signed <)
`jle` *Label*	`jng`	Less or equal (signed <=)
`ja` *Label*	`jnbe`	Above (unsigned >)
`jae` *Label*	`jnb`	Above or equal (unsigned >=)
`jb` *Label*	`jnae`	Below (unsigned <)
`jbe` *Label*	`jna`	Below or equal (unsigned <=)

⛳ Assembly: flags, callee vs caller owned



- CF: Carry flag. The most recent operation generated a carry beyond the most significant bit. Used to detect overflow for unsigned operations.
- ZF: Zero flag. The most recent operation yielded a zero.
- SF: Sign flag. The most recent operation produced a negative value.
- OF: Overflow flag. The most recent operation prompted a two's-complement overflow or underflow.



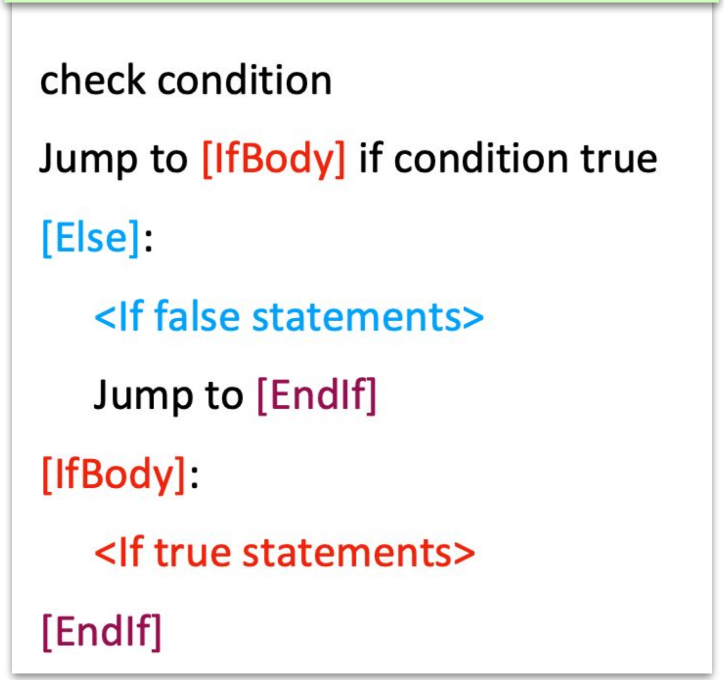
- Callee-owned means that a called function can feel free to use that register without worrying about what was previously put there by its caller

- Caller-owned means that a called function must save and put back later the existing value of the register if it wants to use it

Note that, however, **if the called function calls another function, it should probably save the register's contents somewhere** — as the callee might use the registers also.

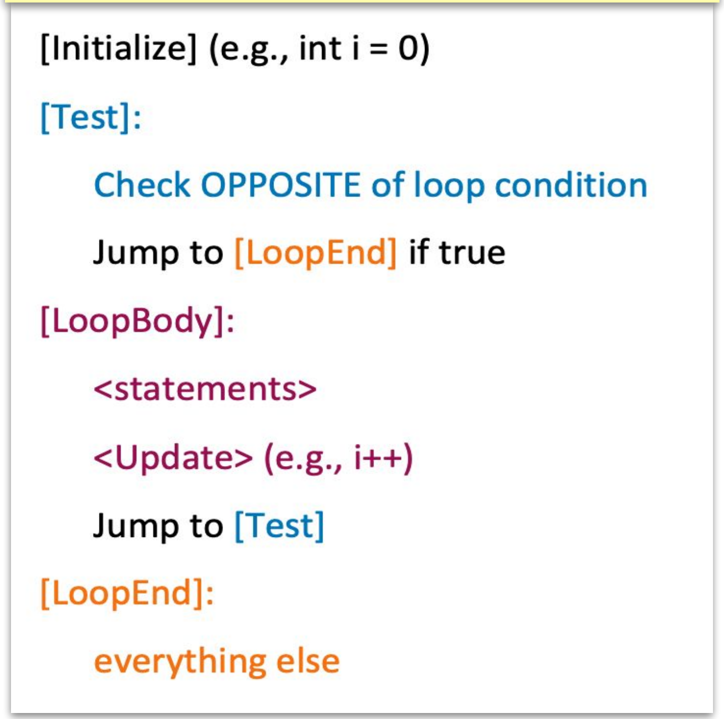
Assembly Control Patterns

if/else branching



```
check condition
Jump to [IfBody] if condition true
[Else]:
    <If false statements>
    Jump to [EndIf]
[IfBody]:
    <If true statements>
[EndIf]
```

Winking to Winky: **Loops**



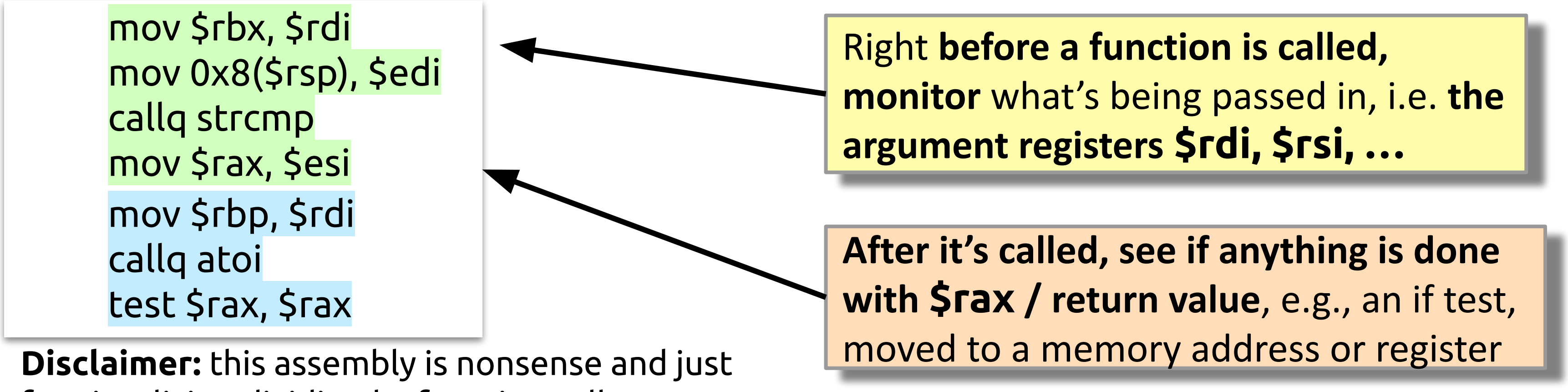
```
[Initialize] (e.g., int i = 0)
[Test]:
    Check OPPOSITE of loop condition
    Jump to [LoopEnd] if true
[LoopBody]:
    <statements>
    <Update> (e.g., i++)
    Jump to [Test]
[LoopEnd]:
    everything else
```

Reverse-Engineering / CodeGen Tips

When compiled without any optimization (-O0 flag), the **order of the C statements will very closely the order of the assembly instructions** they compile to.

No doubt, assembly-to-C can be tricky, but you can do it! Some tactics include:

Rely on function calls to **subdivide the assembly stream into sections**.



```
mov $rbx, $rdi
mov 0x8($rsp), $edi
callq strcmp
mov $rax, $esi
mov $rbp, $rdi
callq atoi
test $rax, $rax
```

Right **before a function is called, monitor** what's being passed in, i.e. **the argument registers $rdi, $rsi, …**

After it's called, see if anything is done with $rax / return value, e.g., an if test, moved to a memory address or register

Disclaimer: this assembly is nonsense and just for visualizing dividing by function calls

When you see a **function you know** (e.g., strcmp), cross-reference to check how many arguments and which data types, to **figure out what's being passed in**

🍀 Reverse-Engineering / CodeGen Tips

No doubt, assembly-to-C can be tricky, but you can do it! Some tactics include:

When you suspect if there's an **if test or if/else test,** look for **conditional and unconditional jumps to higher addresses**, to segment which instructions are apart of the if test, the if body, and potentially the body of an else.

When you know there's a **loop, look for the conditional jump forward, and the unconditional jump backwards**—to figure out which instructions are in the loop, which ones are before it, and which ones come after it.

🎡 Typically, you can segment a stream of many assembly lines into **(1) a pre-loop, (2) within-loop, and (3) post-loop section.** You'll then have **3 more manageable reverse-engineering problems** than one very large one.

Also, **ignore all push and pop operations**, as it's almost always boilerplate assembly to save and ultimately save and restore caller-owned register values.

🍀 Reverse-Engineering / CodeGen Tips

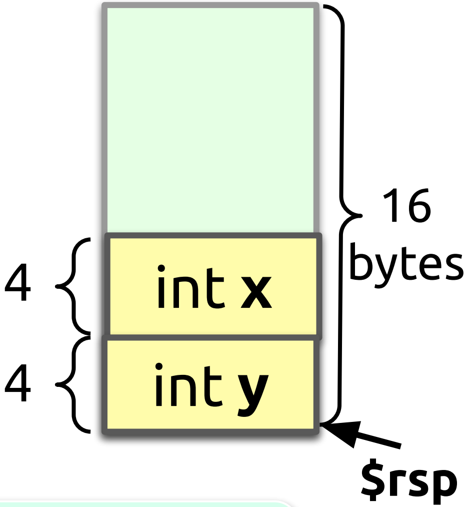
No doubt, assembly-to-C can be tricky, but you can do it! Some tactics include:

Look for **registers that alternate** between **being updated and being read**, because they typically correspond to **local variables**.

For instance, **mov 0x8, $rdx** that's followed by an instruction that reads from **$rdx (**without right away modifying it), then an instruction that updates $rdx → **suggests $rdx is affiliated with some local variable.**

Similarly, look for **portions of the stack frame** (e.g., an 8-byte sliver of it, like 0x8($rsp)) **being written to and updated by registers** → suggests it's affiliated with a local variable

Personally, I like always 🎨 **drawing a function stack diagram** where, when I see a memory address involved, I write / update its value on the stack diagram at the relevant address.

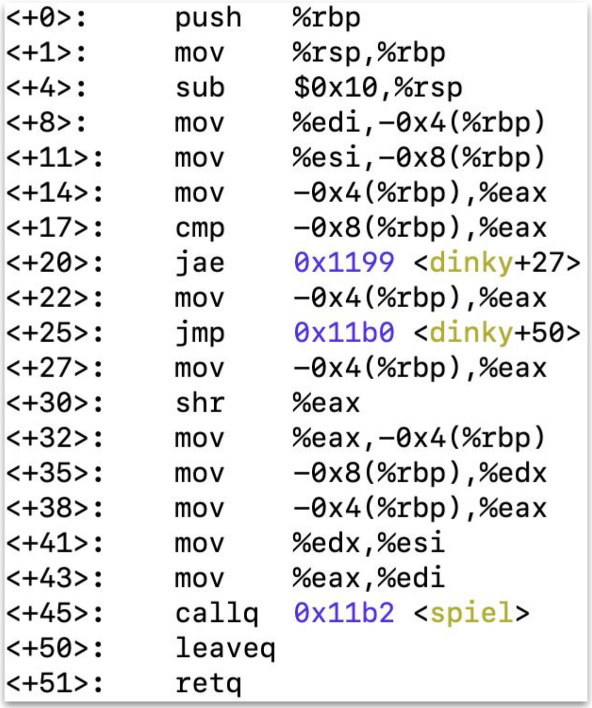


In general, be more concerned with **reverse-engineering equivalent C code**, and not necessarily the original C code.

CodeGen Problem Walkthrough

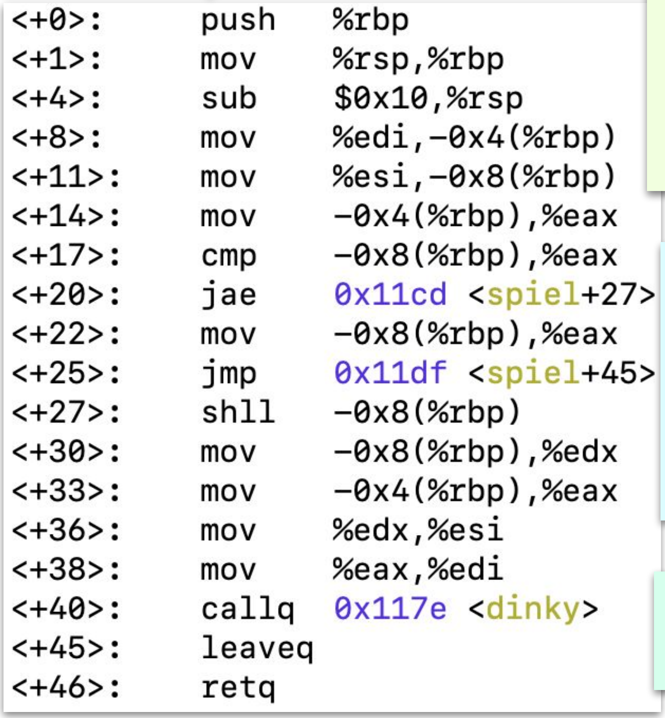
If you loved **binky** and **winky**, say hi to **dinky**! And its friend function **spiel**!

<dinky> function



```
<+0>:     push   %rbp
<+1>:     mov    %rsp,%rbp
<+4>:     sub    $0x10,%rsp
<+8>:     mov    %edi,-0x4(%rbp)
<+11>:    mov    %esi,-0x8(%rbp)
<+14>:    mov    -0x4(%rbp),%eax
<+17>:    cmp    -0x8(%rbp),%eax
<+20>:    jae    0x1199 <dinky+27>
<+22>:    mov    -0x4(%rbp),%eax
<+25>:    jmp    0x11b0 <dinky+50>
<+27>:    mov    -0x4(%rbp),%eax
<+30>:    shr    %eax
<+32>:    mov    %eax,-0x4(%rbp)
<+35>:    mov    -0x8(%rbp),%edx
<+38>:    mov    -0x4(%rbp),%eax
<+41>:    mov    %edx,%esi
<+43>:    mov    %eax,%edi
<+45>:    callq  0x11b2 <spiel>
<+50>:    leaveq
<+51>:    retq
```

<spiel> function



```
<+0>:     push   %rbp
<+1>:     mov    %rsp,%rbp
<+4>:     sub    $0x10,%rsp
<+8>:     mov    %edi,-0x4(%rbp)
<+11>:    mov    %esi,-0x8(%rbp)
<+14>:    mov    -0x4(%rbp),%eax
<+17>:    cmp    -0x8(%rbp),%eax
<+20>:    jae    0x11cd <spiel+27>
<+22>:    mov    -0x8(%rbp),%eax
<+25>:    jmp    0x11df <spiel+45>
<+27>:    shll   -0x8(%rbp)
<+30>:    mov    -0x8(%rbp),%edx
<+33>:    mov    -0x4(%rbp),%eax
<+36>:    mov    %edx,%esi
<+38>:    mov    %eax,%edi
<+40>:    callq  0x117e <dinky>
<+45>:    leaveq
<+46>:    retq
```

They're very similar—so **let's just reverse-engineer dinky** (spiel can be practice if you'd like!)

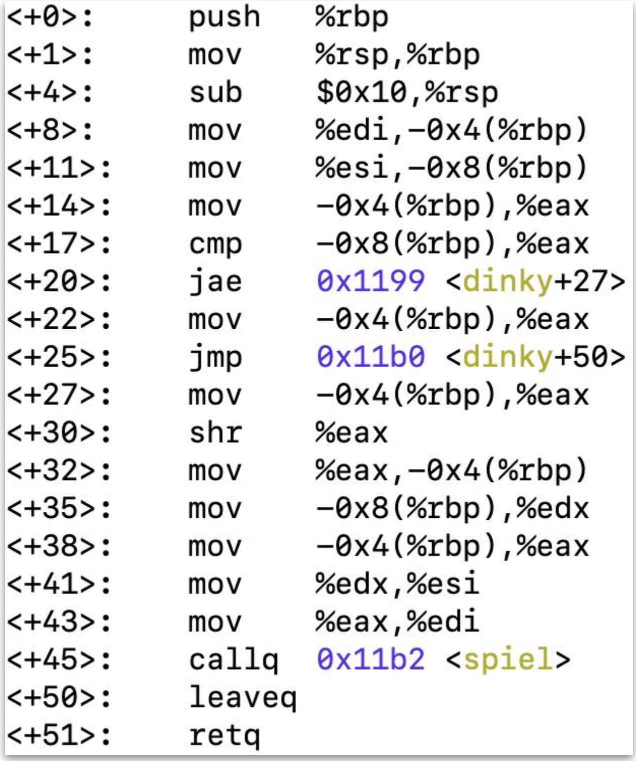
This is mainly testing:
(1) Reading function calls
(2) Reading/updating local variables on the stack
(3) Register tracing

See the **fill-in-the-blanks** on the next slide!

CodeGen Problem Walkthrough

🎵 If you loved **binky** and **winky**, say hi to **dinky**! And its friend function **spiel**!

<dinky> function



```
<+0>:     push    %rbp
<+1>:     mov     %rsp,%rbp
<+4>:     sub     $0x10,%rsp
<+8>:     mov     %edi,-0x4(%rbp)
<+11>:    mov     %esi,-0x8(%rbp)
<+14>:    mov     -0x4(%rbp),%eax
<+17>:    cmp     -0x8(%rbp),%eax
<+20>:    jae     0x1199 <dinky+27>
<+22>:    mov     -0x4(%rbp),%eax
<+25>:    jmp     0x11b0 <dinky+50>
<+27>:    mov     -0x4(%rbp),%eax
<+30>:    shr     %eax
<+32>:    mov     %eax,-0x4(%rbp)
<+35>:    mov     -0x8(%rbp),%edx
<+38>:    mov     -0x4(%rbp),%eax
<+41>:    mov     %edx,%esi
<+43>:    mov     %eax,%edi
<+45>:    callq   0x11b2 <spiel>
<+50>:    leaveq
<+51>:    retq
```

```
// fill in the blanks here for dinky!
unsigned int dinky(unsigned int x, unsigned int y){
    if (____________){
        return ________________;
    }
    ____________;
    return ________________;
}
```

```
// you're given the function signature for spiel also!
unsigned int spiel(unsigned int x, unsigned int y);
```

uint dinky(uint x, uint y)

First, from dinky's function signature, it has **2 arguments**:

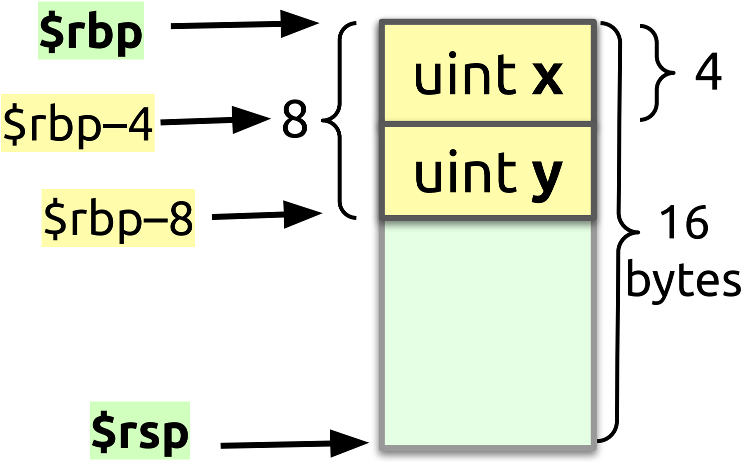
📌 Unsigned int **x (in $rdi register)**
📌 unsigned int **y (in $edi register)**

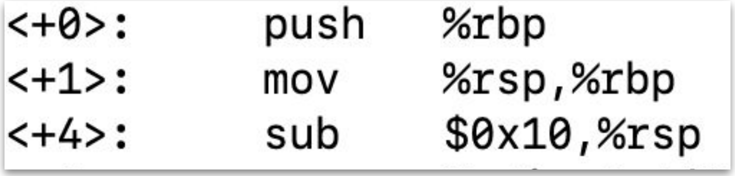
x

%rdi/%edi

y

%rsi/%esi





```
<+0>:       push    %rbp
<+1>:       mov     %rsp,%rbp
<+4>:       sub     $0x10,%rsp
```

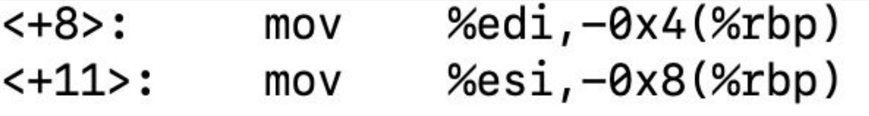
original $rsp

%rbp

original – 0x10 (16)

%rsp

Then, we **move stack pointer $rsp into $rbp** (to keep a copy of orignial address), then decrement it by **0x10 / 16 bytes** — which tells us the **size of function stack** yay!

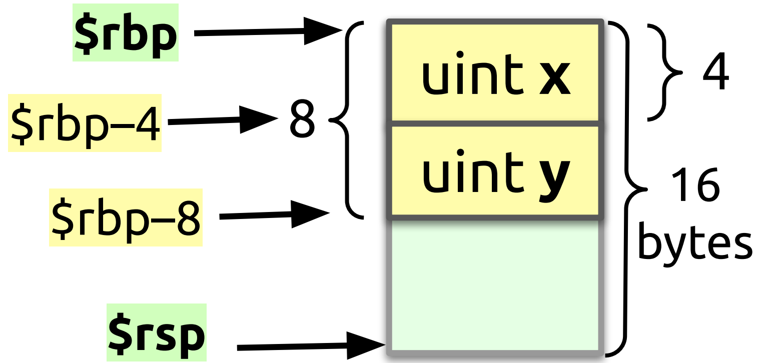


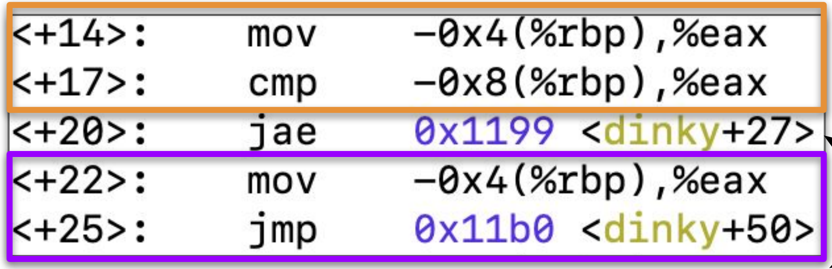
```
<+8>:       mov     %edi,-0x4(%rbp)
<+11>:      mov     %esi,-0x8(%rbp)
```

Next, we **move register values $edi (x) and $esi (y)** into addresses **4 and 8 bytes below the original $rsp**, respectively (see stack above).

CodeGen Problem Walkthrough

```
uint dinky(uint x, uint y){
    if (__x < y__){
        return ____x____;
    } // let's fill in the if loop!
```

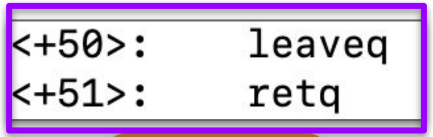




```
<+14>:    mov    -0x4(%rbp),%eax
<+17>:    cmp    -0x8(%rbp),%eax
<+20>:    jae    0x1199 <dinky+27>
<+22>:    mov    -0x4(%rbp),%eax
<+25>:    jmp    0x11b0 <dinky+50>
```

We have a **cmp and conditional jump** (and if not taken, a jmp later), suggesting an **if branch present!**

...



```
<+50>:    leaveq
<+51>:    retq
```

x

%eax

First, we move **-0x4($rbp), or x** from the stack diagram, **into $eax**, and **compare it with** -0x8($rbp), or **y**.

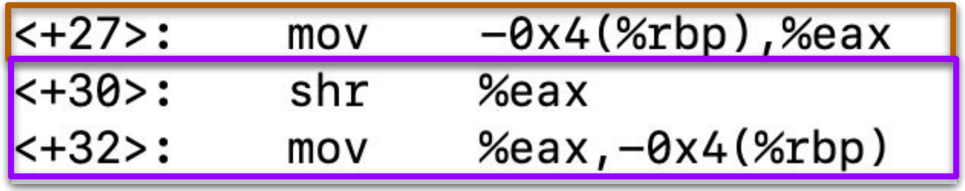
jae taken (x >= y)

Proceeds to <+27>

jae not taken (x < y)

Moves -0x4($rbp), or **x into $eax register, then jumps to return**

CodeGen Problem Walkthrough



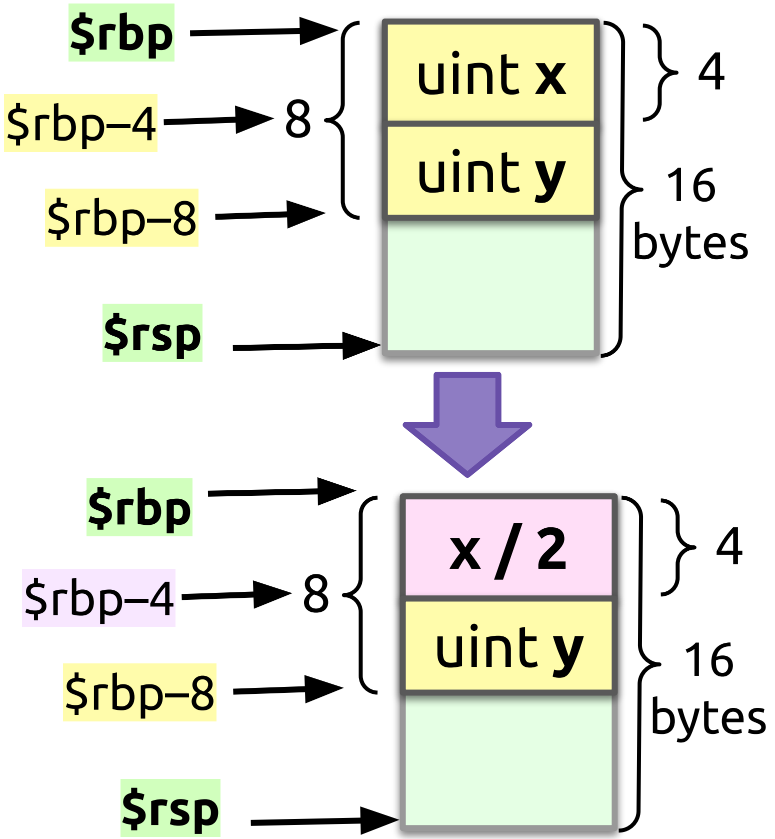
```
<+27>:      mov     -0x4(%rbp),%eax
<+30>:      shr     %eax
<+32>:      mov     %eax,-0x4(%rbp)
```

Now, we move **-0x4($rbp), or x, into $eax.**

Then, we **right-shift $eax**, so **it's now x / 2**.

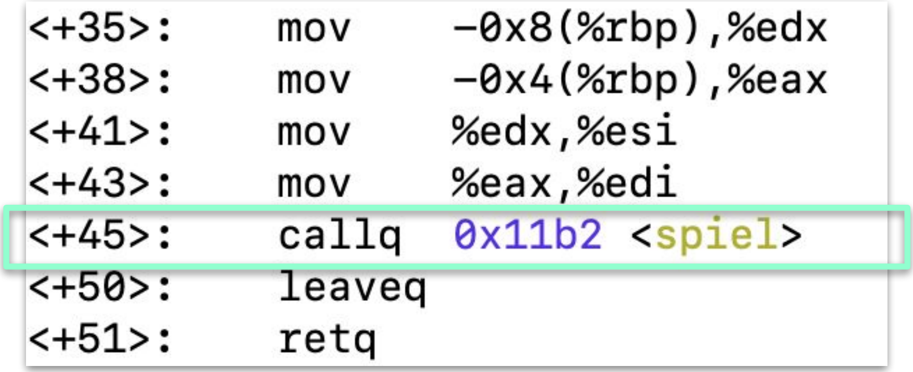
We move the value back into memory at -0x4($rbp), so **this is just x = x/2; !**

```
uint dinky(uint x, uint y){
  if (_x < y_){
    return __X__;
  }
__x = x/2_;
```

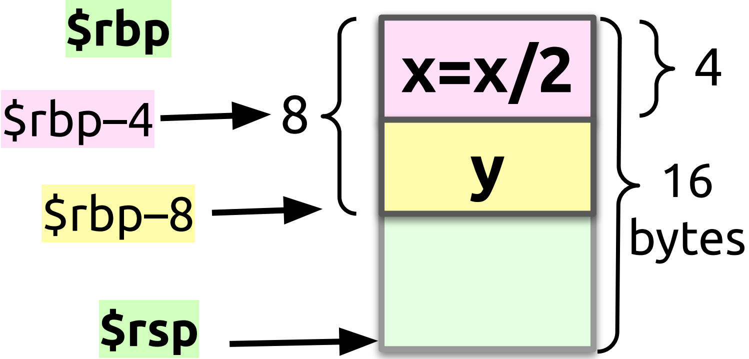


🪄 CodeGen Problem Walkthrough

🎵 **uint spiel(uint x, uint y);** // function signature of spiel!



```
<+35>:      mov     -0x8(%rbp),%edx
<+38>:      mov     -0x4(%rbp),%eax
<+41>:      mov     %edx,%esi
<+43>:      mov     %eax,%edi
<+45>:      callq   0x11b2 <spiel>
<+50>:      leaveq
<+51>:      retq
```



The assembly above is really just:

⛵ Move value at **-0x8($rdp), which is y,** into $edx and then **into $esi.**

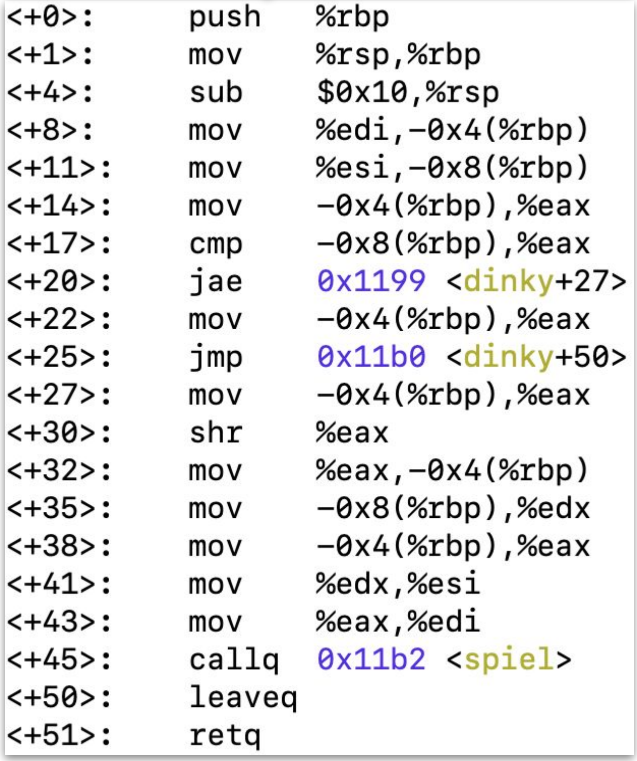
⛵ Move value at **-0x4($rdp), which is (original x)/2** into $eax and then **into $edi.**

Then, it calls **spiel($rdi = x, $rsi = y)**

```
uint dinky(uint x, uint y){
  if (_x < y_){
    return __x__;
  }
  _x = x/2_;
  return __spiel(x, y)__;
```

🪄 CodeGen: And we're done! 🎊

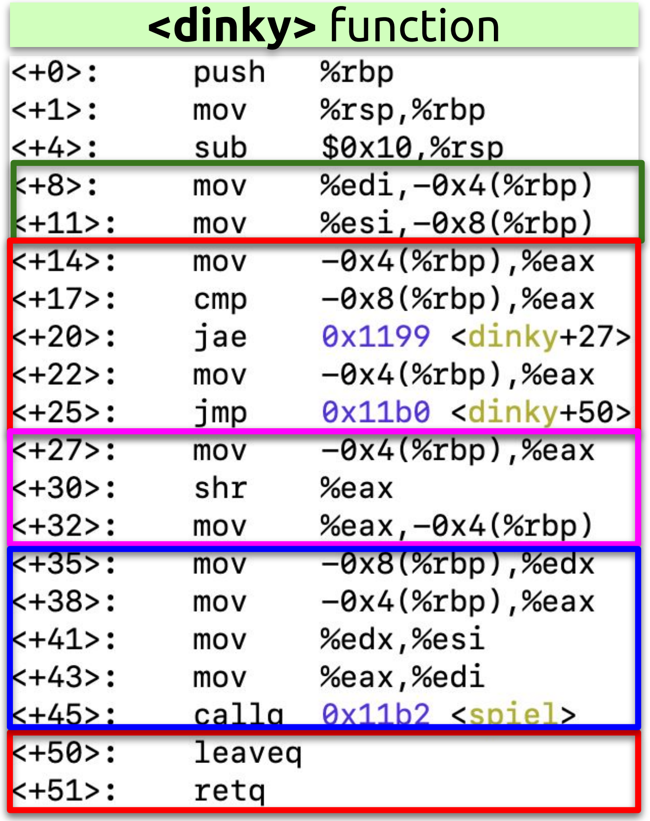
<dinky> function



```
<+0>:     push    %rbp
<+1>:     mov     %rsp,%rbp
<+4>:     sub     $0x10,%rsp
<+8>:     mov     %edi,-0x4(%rbp)
<+11>:    mov     %esi,-0x8(%rbp)
<+14>:    mov     -0x4(%rbp),%eax
<+17>:    cmp     -0x8(%rbp),%eax
<+20>:    jae     0x1199 <dinky+27>
<+22>:    mov     -0x4(%rbp),%eax
<+25>:    jmp     0x11b0 <dinky+50>
<+27>:    mov     -0x4(%rbp),%eax
<+30>:    shr     %eax
<+32>:    mov     %eax,-0x4(%rbp)
<+35>:    mov     -0x8(%rbp),%edx
<+38>:    mov     -0x4(%rbp),%eax
<+41>:    mov     %edx,%esi
<+43>:    mov     %eax,%edi
<+45>:    callq   0x11b2 <spiel>
<+50>:    leaveq
<+51>:    retq
```

```
// fill in the blanks here for dinky!
unsigned int dinky(unsigned int x, unsigned int y){
    if ( x < y ){
        return x;
    }
    x = x / 2; // alternatively, can do return spiel(x/2,y)
    return spiel( x , y );
}

// you're given the function signature for spiel also!
unsigned int spiel(unsigned int x, unsigned int y);
```

CodeGen: And we're done!



<dinky> function

```
<+0>:     push   %rbp
<+1>:     mov    %rsp,%rbp
<+4>:     sub    $0x10,%rsp
<+8>:     mov    %edi,-0x4(%rbp)
<+11>:    mov    %esi,-0x8(%rbp)
<+14>:    mov    -0x4(%rbp),%eax
<+17>:    cmp    -0x8(%rbp),%eax
<+20>:    jae    0x1199 <dinky+27>
<+22>:    mov    -0x4(%rbp),%eax
<+25>:    jmp    0x11b0 <dinky+50>
<+27>:    mov    -0x4(%rbp),%eax
<+30>:    shr    %eax
<+32>:    mov    %eax,-0x4(%rbp)
<+35>:    mov    -0x8(%rbp),%edx
<+38>:    mov    -0x4(%rbp),%eax
<+41>:    mov    %edx,%esi
<+43>:    mov    %eax,%edi
<+45>:    callq  0x11b2 <spiel>
<+50>:    leaveq
<+51>:    retq
```

```
// fill in the blanks here for dinky!
unsigned int dinky(unsigned int x, unsigned int y){
    if ( x < y ){
        return x;
    }

    x = x / 2; // alternatively, can do return spiel(x/2,y)

    return spiel( x , y );
}

// you're given the function signature for spiel also!
unsigned int spiel(unsigned int x, unsigned int y);
```

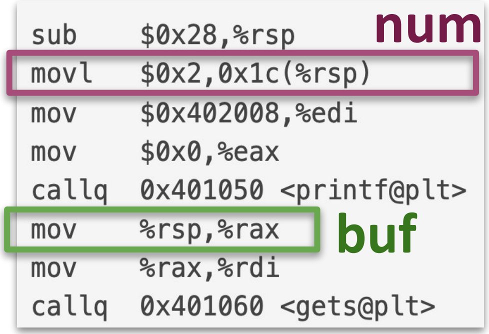
📚 Stack Vulnerabilities

Buffer Overflow

Functions such as gets() are bad as they **allow writing past the buffer**, and thus into other parts of the stack. ***Do not use gets!***

Overwriting variables, return, etc.

Variables, return address, etc. are laid out linearly on the function stack. **This can be exploited by buffer overwriting** to alter data values and thus a function's behavior.



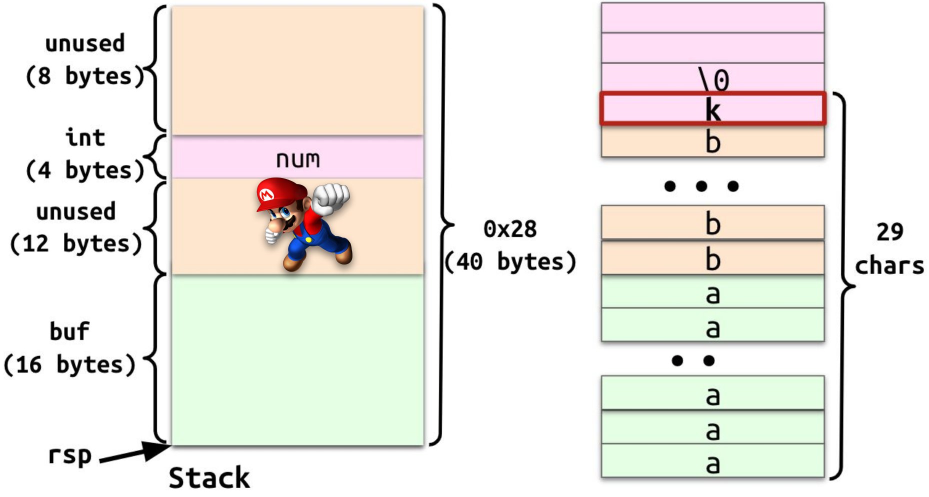
```
sub     $0x28,%rsp
movl    $0x2,0x1c(%rsp)
mov     $0x402008,%edi
mov     $0x0,%eax
callq   0x401050 <printf@plt>
mov     %rsp,%rax
mov     %rax,%rdi
callq   0x401060 <gets@plt>
```

I would recommend reviewing:

🔥 The **stack smash attack from Lab6**, where we infiltrated an int variable on the stack, to return 107 via buffer overflow.

🔥 **Your work from ATM** in Assign5

And, in general, **draw a stack diagram!** Use the assembly to trace where each component (e.g., buffer, variable) is placed in the stack.



Questions on assembly, reverse-engineering?

Next up: optimization!

🚞 Optimizations

For more detail and examples of each gcc optimization, see the **Lecture 25 slides**!

GCC can perform a variety of optimizations for us, when compiling with different flags (–O0 for none, –O2 for nearly all reasonable optimizations, –03 is more aggressive)

🔢 **Constant Folding:** Compiler can pre-compute constant values, including e.g., constant arithmetic, sizeof(int) → 4, strlen("LeBronJames") → 11

🧮 **Common subexpression elimination:** Avoids recalculating same result multiple times

🧸 **Strength reduction:** Avoids multiplying/dividing by shifting and adding instead

🧟 **Dead code elimination:** If a piece of code can never be reached (e.g., code in a function after a return statement), the compiler can just remove it

🚄 **Code motion:** Rearrange code for better performance

🗞️ **Loop unrolling:** Avoid many expensive conditions and jumps by copy-pasting the loop body (i.e. less iterations, with more work per iteration)

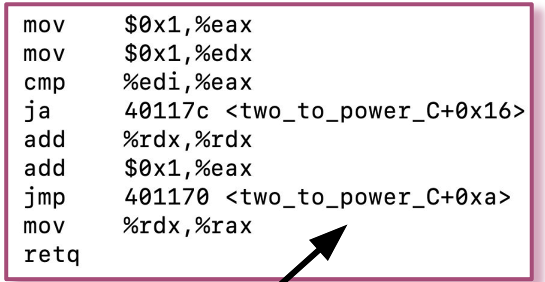
🌀 **Tail Recursion:** Alters some recursive patterns to be iterative instead

Optimizations

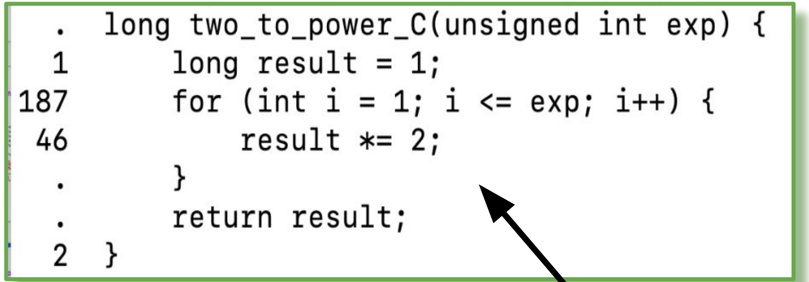
Static instruction count is the number of written assembly instructions

Dynamic instruction count is number of executed instructions **when a program is actually run** (note that a written line can be run multiple times, e.g., loops).

Example: This code computes 2^n by repeatedly multiplying with 2 inside a loop



```
mov     $0x1,%eax
mov     $0x1,%edx
cmp     %edi,%eax
ja      40117c <two_to_power_C+0x16>
add     %rdx,%rdx
add     $0x1,%eax
jmp     401170 <two_to_power_C+0xa>
mov     %rdx,%rax
retq
```



```
   .  long two_to_power_C(unsigned int exp) {
   1      long result = 1;
 187      for (int i = 1; i <= exp; i++) {
  46          result *= 2;
   .      }
   .      return result;
   2  }
```

Static count (lines of assembly) is just 9 instructions, so doesn't look too bad.

But **when actually run, dynamic count is >200!** The lines in the **loop** run many many times!

Takeaway: Dynamic counts are a better gauge of performance. Occasionally, the assembly code emitted with optimization can feature more static lines of assembly than the original, but that doesn't mean more instructions are actually executed!

🏛️ Ethics Content is Fair Game!

Full disclosure vs responsible disclosure

Full: Notify public immediately about vulnerability

Responsible: Privately notifying vendor first to make fixes 🛠️ – then disclosing to the public

Four degrees of partiality

1 Partiality

2 Partial Cosmopolitanism

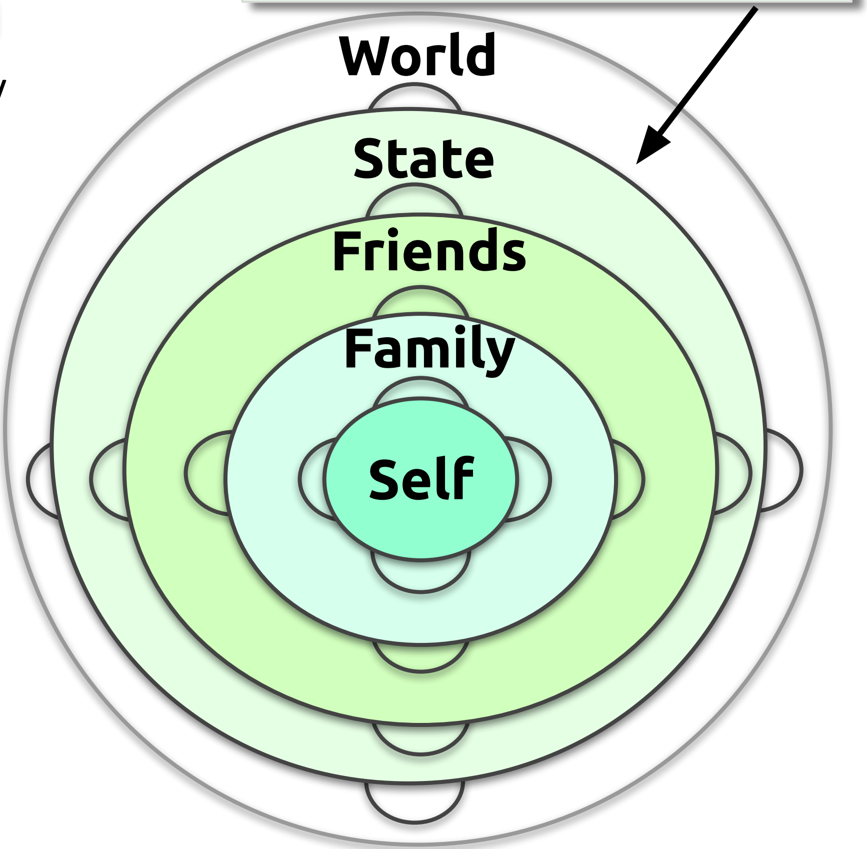
3 Universal Care

4 Impartial Benevolence

Privacy and trust

🗝️ **Privacy** as (1) control of information, (2) autonomy, (3) social good, and (4) trust

🏦 **Trust models:** Who is trusted / distrusted? Centralized or distributed?



Concentric circles representing groups towards whom one might demonstrate partiality

Closing Remarks!

📓 **Check out the Final Exam Page** on the 107 website—which has 5 different practice exams (all previous finals!) to study from.

Go for it!

Besides the practice exams, you can look over lab handouts / solutions to enrich understanding, as well as course assignments. **I'll upload these slides to the course website (and announce on Ed!) right after.**

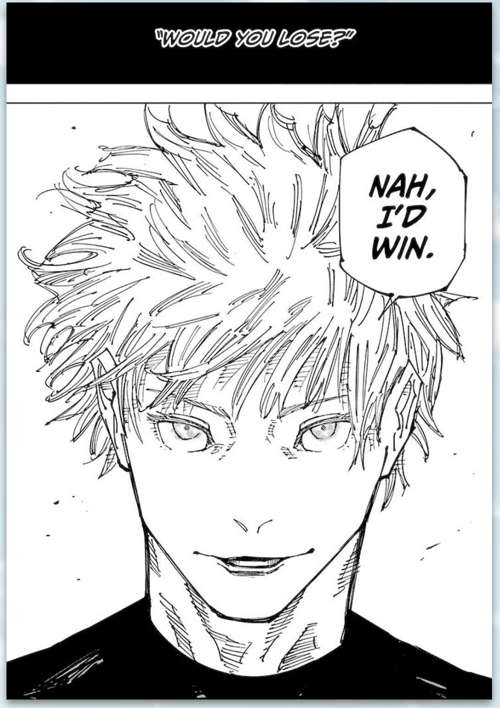
Please feel free to post on Ed with any questions! (practice exam/solution clarifications, conceptual, exam tips, etc.)

I'm also more than happy to take questions after this review session.

You can do this!

Good luck on the final!

You’re 👏 going 👏 to 👏 be 👏 awesome. 👏



And after finals, have an amazing summer break!!
It’s been such a joy being your TA, thank you ❤️