



CS107 Lab 6 – We're Almost There!



Runtime Stack, Smash Attack

Ben Yan, Winter 2026 ❄️

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



My Song of the Week: SPECIALZ by King Gnu (Debugger)

Stanford | ENGINEERING
Computer Science

Welcome Back to Lab!





In other news, was on TV last week! Sort of lol

Stanford University

Powered by Panopto

Review Sessions > CS107 Review Session on 2/9/2026 (Mon)

🔍 ? BY



Ben Yan, Winter 2026
Slides Link: stanford.edu/~bbyan/cs107

Search this recording 🔍

Details ⓘ Auto-generated captions may contain errors.

Captions

Discussion

Notes

Bookmarks

Um. Okay. Good to kick off. All right. 0:08

Oh, everyone, uh, welcome to the CSE 1 to 7 mid-term review session. I hope that you're all having a really good day. 0:17

Um, first things first. That's the link to the slides. You can find it on my lab website. 0:22

Or I just made a announcement, um, on ATS ten. So it should be like the first post that comes up. 0:25

Um, so you can sort of follow along the slides there. I also post to the handouts because I don't know if anyone's noticed, 0:30

but the CSE websites aren't really working at the moment or there's not very convenient access and during midterms week. 0:35

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

5:47

0:19 -1:24:53

🔍 ⚙️ 📄

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

Midterm Logistics and Topics Coverage

- When/where is the midterm?
- What are Jerry exams like?
- What's covered on the midterm? What should I brush up on?

Binary Representation of Integer Values

All values in C are represented in memory as binary: a sequence of 0's and 1's (bits) denoting the powers of 2 that make up a number. Decimal is our own way of representation.

7 = $2^2 + 2^1 + 2^0$ **111** 10 = $2^3 + 2^1$ **1010**

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

For 32, I think about MSB as having 1 negative weight, e.g., -2³¹ for a 32-bit int.

32-bit int: **01000000000000000000000000000000** 32-bit int: **11111111111111111111111111111111**



May need to work on mochi throwing skills



Welcome Back to Lab!

Hope that this week is going well and warmer!



Winter



Week 7



Spring



- ☐  **Assign5: Banking on Security** (once known as **Binary Bomb**) released! Please allocate time wisely for this one!
- ☐ It's ~~notorious and villainous~~ fun but challenging and rewarding, and will make you magical at reading assembly 

Welcome Back to Lab!

Hope that this week is going well and warmer!



**For Binary Bomb, PLZ PLZ PLZ DO NOT
START A DAY BEFORE THE DEADLINE!**



- ❑  **Assign5: Banking on Security** (once known as **Binary Bomb**) released! Please allocate time wisely for this one!
- ❑ It's ~~notorious and villainous~~ fun but challenging and rewarding, and will make you magical at reading assembly 



FAQ – How long does Binary Bomb take?

- Longer than the other assignments. Please **start ASAP. *ALL* of you!!**
 - On the plus side, as long as you leave enough time, this assignment is *absolutely doable* for everyone.
- Data from Andrew Benson's Spr 21 ACE section (11 students). This is 1 datapoint, individual times vary WIDELY (data from Winter 21 was significantly less time)

◦ ATM Part a)	1.77 hours
◦ ATM Part b)	3.09 hours
◦ ATM Part c)	4.09 hours
◦ Dataset Aggregation	1.16 hours
◦ Secure Vault L1	1.43 hours
◦ Secure Vault L2	4.32 hours
◦ Secure Vault L3	6.64 hours
◦ Secure Vault L4	9.95 hours

← This may include spec reading / onboarding

32.45 hours total

This is not meant to scare you, I promise! 😭

Just motivation to start early and budget your time!

You can do this! 🙏

From the CS107 ACE (Win '25) slides

If you want support, come to our OH! 🌟



Barack Obama ✓
@BarackObama



Why would we spend taxpayer dollars on a
Secure Vault with a fundamental flaw that can
be exploited by the sheer power of CS107 friendship
[OFA.BO/yfxWt3](#)

107 (especially binary bomb + heap allocator) tends to make a lot of close, emotional bonds. no idea why 🙌💚

Getting Setup for Lab 6



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

```
git clone /afs/ir/class/cs107/repos/lab6/shared lab6
```

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

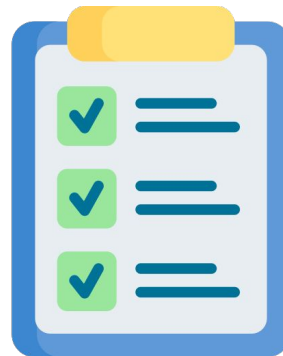


[Lab Handout](#)



[Lab Check-Off Form](#)

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

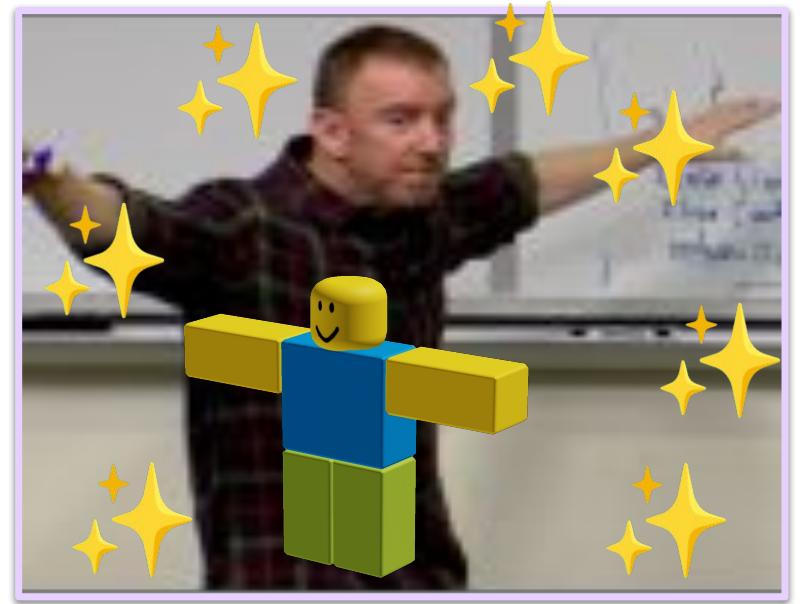


Lecture Recap!



~10 minutes

Now that we have all the
assembly puzzle pieces :)



Penultimate Jerry picture of the week!



Lab 5 Materials: Assembly Reading Techniques

Lab 5



Assembly, Translating and Tracing Machine-Level Code — Feb 11-12

Slides

Slides (PDF)

Lab Handout

Check-Off Form

Lecture Recap



Ben's Waterfall Method

x86-64 Reference

Ben Yan, CS107 Win26



Lab 5 Recap: Reading Assembly



Assembly & Registers



The **compiler** converts our **C code** into low-level **assembly instructions**, which can then be converted into machine code (0s and 1s) run by your computer.

```
int gold = 10;
int treasure = 5;
gold += treasure;
```

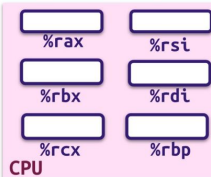


```
mov $10, %rax
mov $5, %rbx
addl %rbx, %rax
```

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 values

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



RAM/Memory



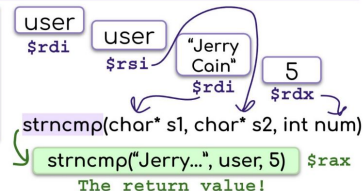
Ben's 'Waterfall Method' of Reading Assembly

General Guidelines:

(may have to adapt to specific assembly task!)

- Write each **register value** (or corresponding variable) in a separate, **labeled box**.
- Assembly is poetry, flowing top to bottom – **for each line, write a new box lower than the previous one and indented to the right**, with the updated value (don't cross out old boxes!).
- Treat function calls as a stanza break – **write each function call explicitly, and draw arrows from the registers being passed in**. Write a box for return **\$rax** after, and reset indentation.

```
is_not_poohbear(user):
<+0> mov $rdi, $rsi
<+4> mov $0x106a106b, $rdi
<+8> mov $0x5, $rdx
<+12> call strncmp
<+16> ret
```



Small toy example! Assume `$0x106a106b` corresponds to the string "Jerry Cain".



Recall: New GDB Commands for Assembly!



disassemble: prints the assembly for the currently executing function (can write *(gdb) disassemble func_name* to disassemble another function)



b *address: Places a breakpoint on a specific assembly instruction, via its address



stepi, nexti: Assembly versions of `step`, `next`, going one line of assembly at a time



info reg: Prints out the values in all integer registers



print \$register: Prints out a specific register's contents, e.g., `print $rax`

Note: Can print using a typecast, e.g., `(gdb) print (char*) ptr`, or in hex, via `print /x`

```
(gdb) p (char *)$rax
$3 = 0x400720 "Hello, world!\n"
```

```
(gdb) p *(long *)$rax
(gdb) p *(long *)((char *)$rsp + 8)
```



set \$register = value: manually set a register's contents, useful for testing behavior

In Terminal / myth, can run **objdump -d executable_name** to list all assembly instructions



Other Cool GDB Commands for Assembly



(gdb) **layout split** is a neat way to view both the assembly + GDB panel!



display: Reprints something automatically every time you step, e.g., (gdb) **display \$rax** will print out the value in **\$rax** each time we move forward



Conditional breakpoints: Pauses only if a specific condition is met

- (gdb) **break [function or instruction #] if (condition)**
- **Example:** Can help for **loops** if you want to stop only on the last iteration, not every!



Watchpoints: Special and nifty breakpoint — breaks only when variable, expression, or memory location value changes

- **Example:** (gdb) **watch *(int *) (\$rsp + 0x4)**
- This monitors an **int** variable stored at address **\$rsp + 0x4** for any changes!



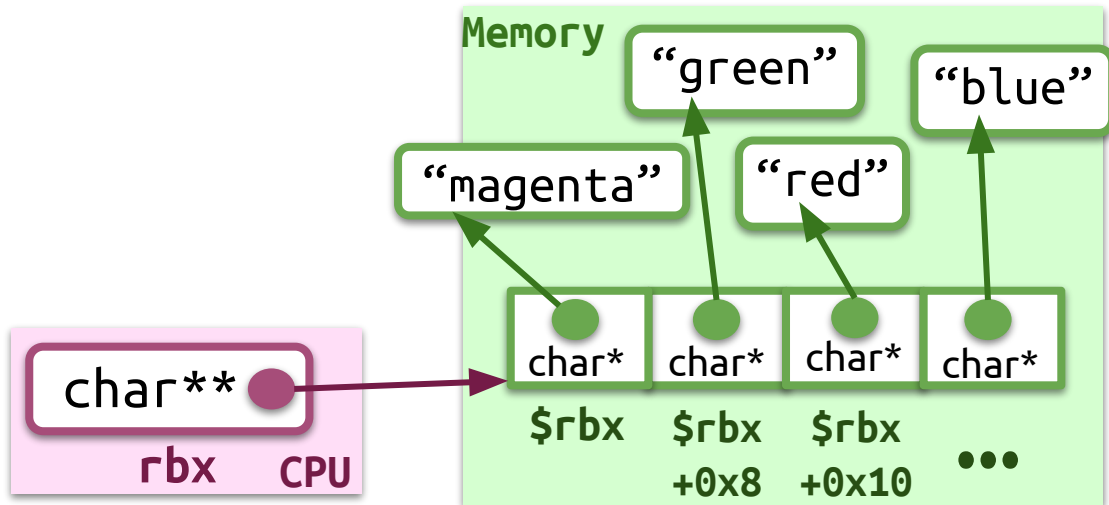
I've personally found **conditional breakpoints / watchpoints** a life-saver on **Assign5/6!**



Check-Off Q: GDB Printing of Arrays

Premise: In some program far far away, we know that `%rbx` holds an address for an **array of strings** (i.e. is a pointer to the first element, which is a `char*`).

Question: How could we **print in GDB** to see the **index-2 element** in this array?





Check-Off Q: GDB Printing of Arrays

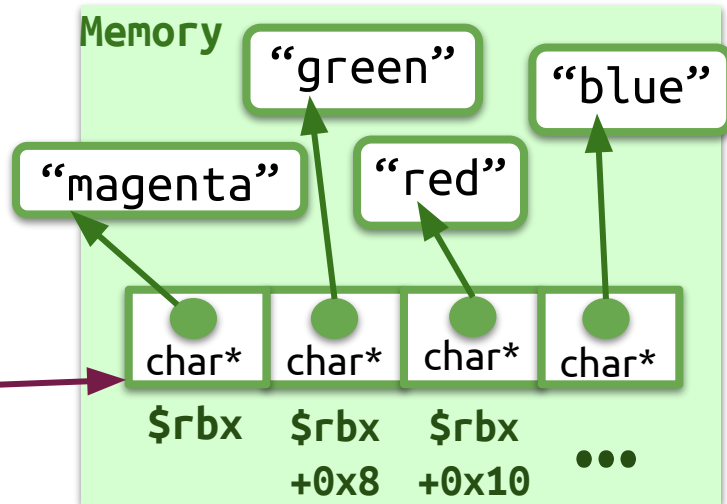
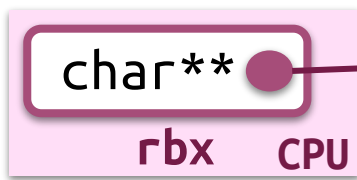
Premise: In some program far far away, we know that `%rbx` holds an address for an **array of strings** (i.e. is a pointer to the first element, which is a `char*`).

Question: How could we **print in GDB** to see the **index-2 element** in this array?

Solution: As `%rbx` is a pointer to a `char*`, we need to **cast it to a `char**`** data type before we can index / dereference.

```
(gdb) print ((char **) $rbx) [2]
```

Extra: To print out the entire array in GDB, we can write:

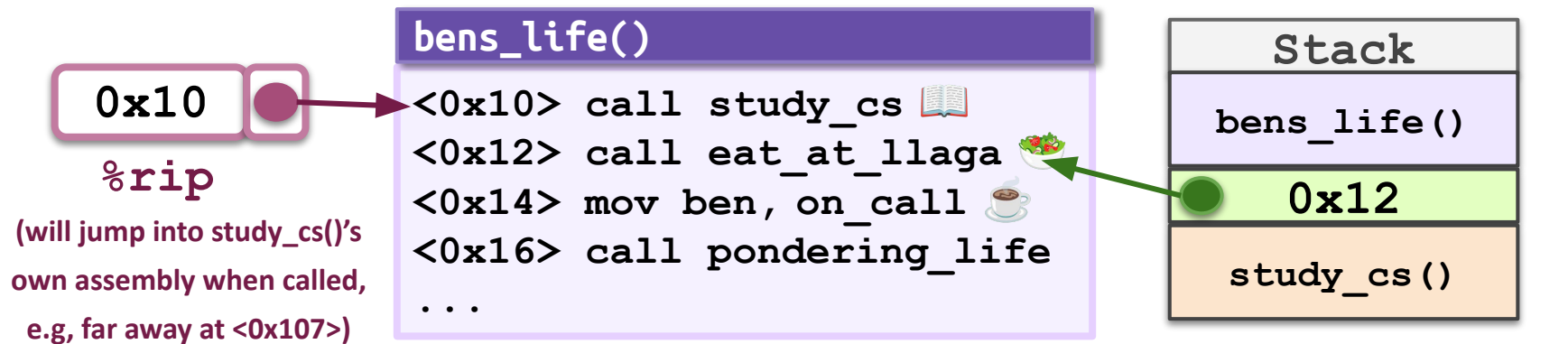


```
(gdb) print ((char **) $rbx) [0] @ number of elements
```

👉 Special Registers: Instruction Pointer `%rip`

`%rip` stores the **address of the next instruction** to execute:

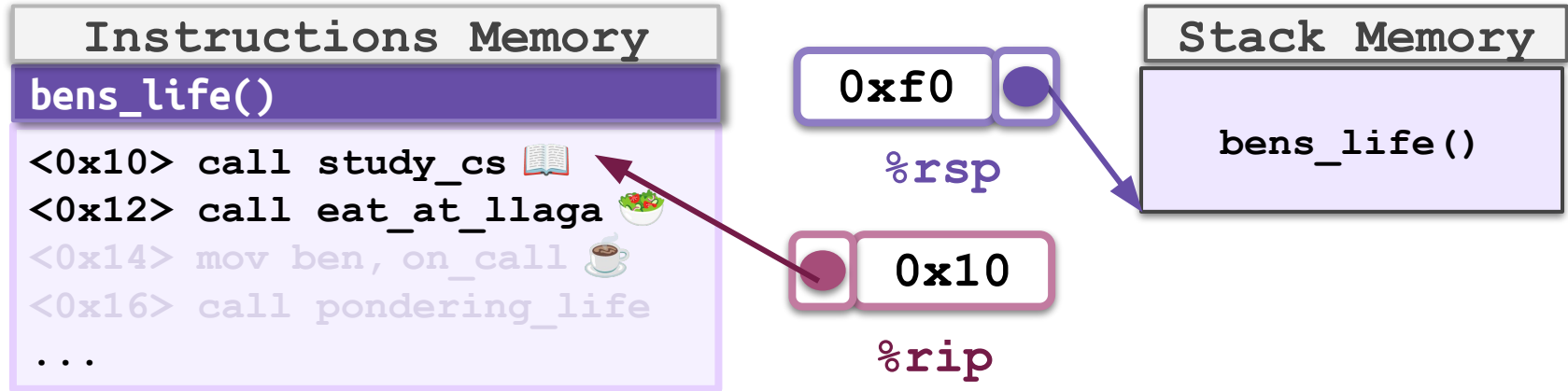
- If we jump to another instruction or call a new function, `%rip` is updated
- When we call / jump into another function, **we must save the caller's next instruction** to execute **so we can resume there** when the callee finishes
- `call` instruction does this for us automatically by **storing it on the stack**, and the `ret` instruction pops the 📌 value / breadcrumb off into `%rip`



“When Ben is done studying CS, his life resumes at Llaga dining”



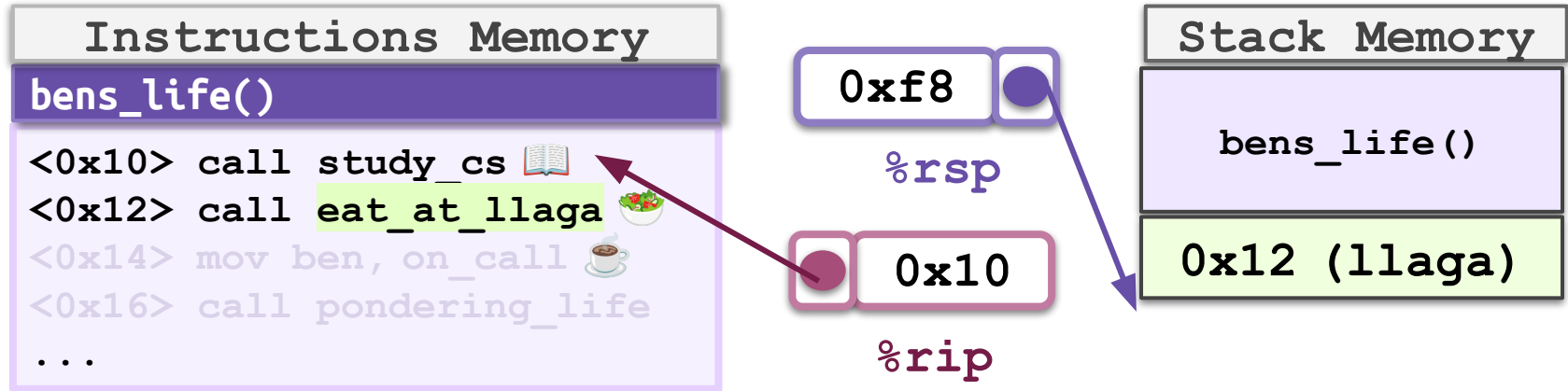
Example: `bens_life()` calls `study_cs()`



In `bens_life()`: `<0x10>` is assembly instruction that calls `study_cs()`



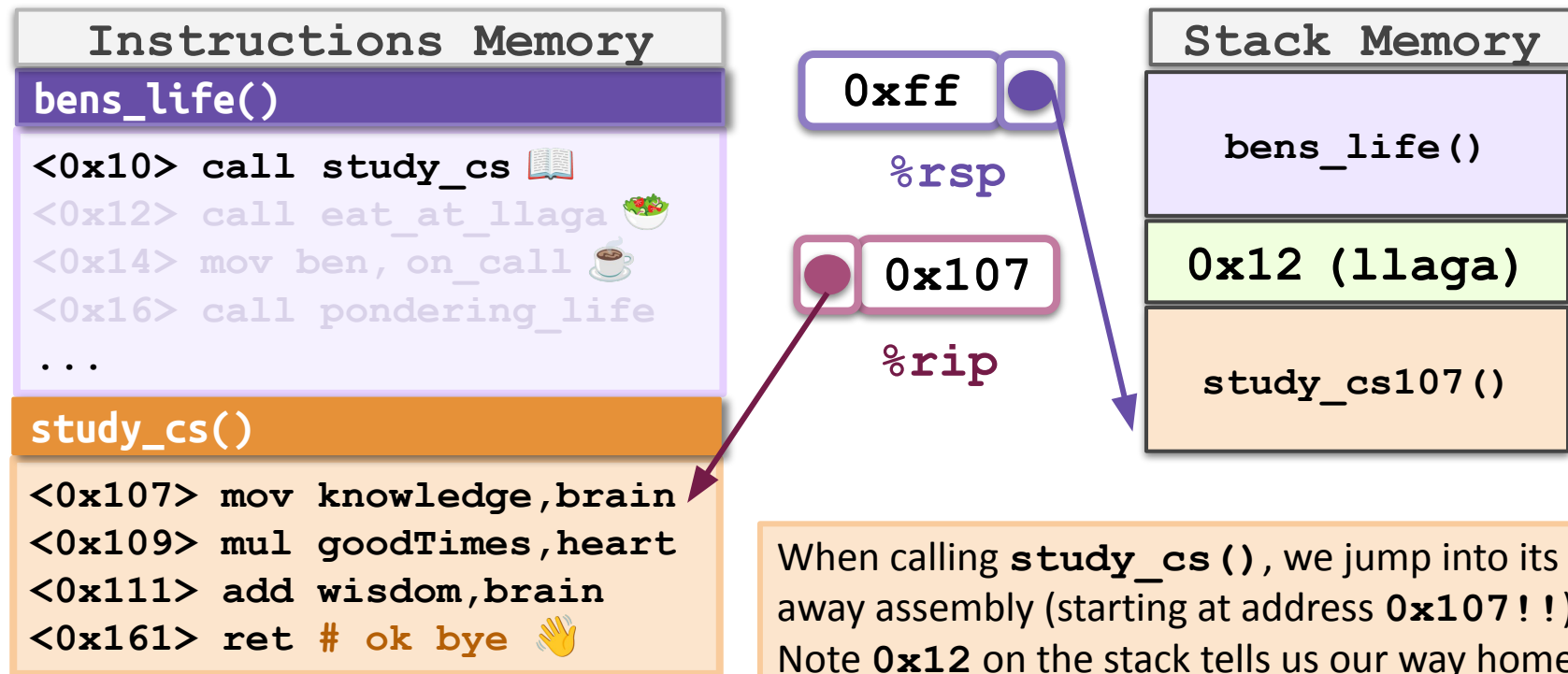
Example: `bens_life()` calls `study_cs()`



In `bens_life()`: `<0x10>` is assembly instruction that calls `study_cs()`

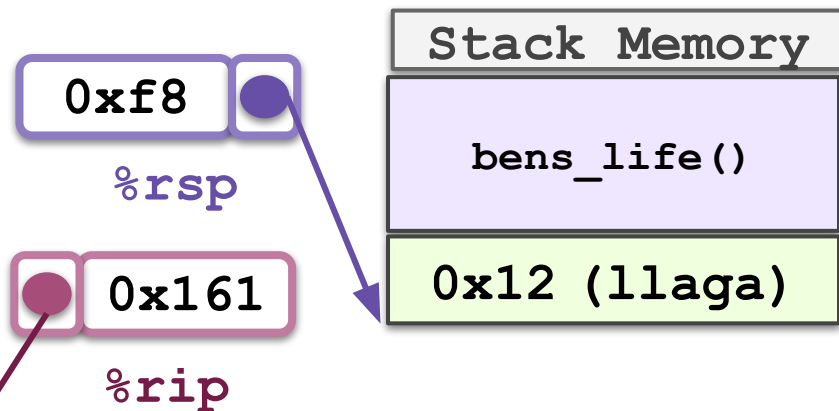
`<0x12>` is next instruction when we come back to `bens_life()`. We need to push this value to stack before we leave, so we remember later!

Example: `bens_life()` calls `study_cs()`



Example: bens_life() calls study_cs()

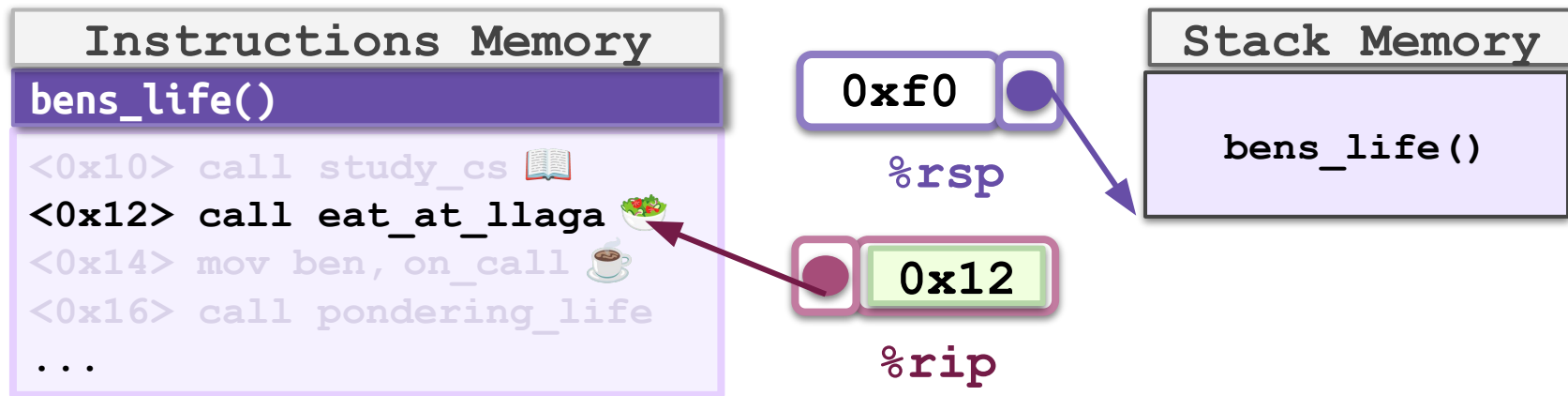
Instructions Memory
bens_life()
<0x10> call study_cs 
<0x12> call eat_at_llaga 
<0x14> mov ben, on_call 
<0x16> call pondering_life
...
study_cs()
<0x107> mov knowledge, brain
<0x109> mul goodTimes, heart
<0x111> add wisdom, brain
<0x161> ret # ok bye 



Now, when we're all done with **study_cs()**, we need to figure out the **next instruction to execute** — which is **stored right on the stack!**

0x12 call eat_at_llaga 

Example: `bens_life()` calls `study_cs()`

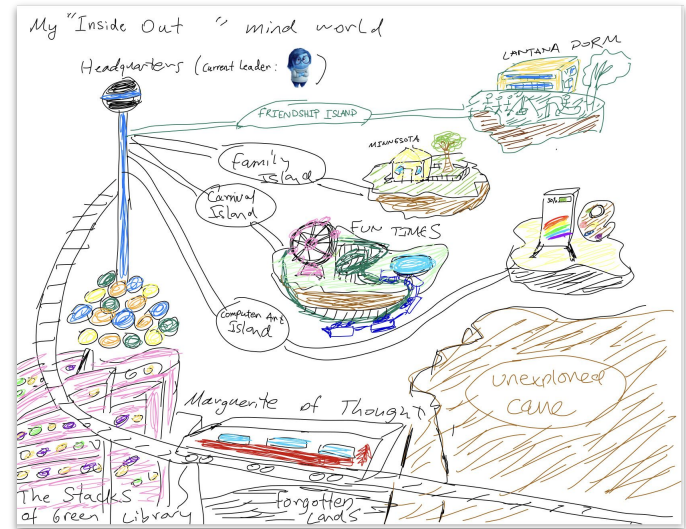
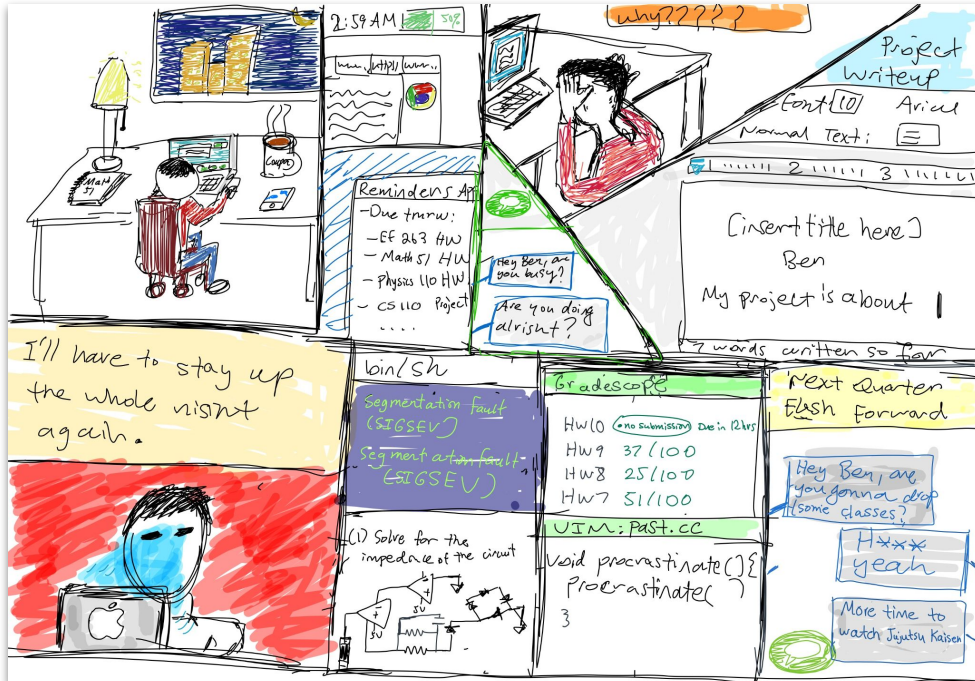


Lastly, we pop the tasty breadcrumb `0x12` off the stack and into `$rip`, so that we resume execution in `bens_life()` and eat at llaga! 🍕

Moral of the Story: There's more to life than CS! Don't forget to take care of yourself, whether that's including meals, break time, walking, etc. as part of your daily stack :)

Example: `bens_life()` calls `study_cs()`

Moral of the Story: There's more to life than CS! Don't forget to take care of yourself, whether that's including meals, break time, walking, etc. as part of your daily stack :)



Comics I made a while back in English 9CE creative workshop lol
(I'm still working on work-life balance myself! )



Exercise 1: GDB and Stack Mechanics

- In your **lab6** directory, go through the **stack.c** program, and its assembly representation to get familiarized with the new set of GDB commands 
- These kind of GDB techniques are similar to ones you'll employ for security explorations in Assign5! (i.e. breaking into a bank) 

Hands-on Practice

Let's try out all these new gdb commands! First, read over the program `stack.c`. Then compile the program and load it into gdb. Do each of the following steps:

1. `break kermi` to put a breakpoint at the start of the `kermi` function
2. `run` to run the program. It should pause right as it begins `kermi`.
3. `disassemble kermi` (or even just `disassemble` if you just broke at the call to `kermi`) to view the assembly for the `kermi` function
4. `display/3i $rip`. From now on, every time you step, it will print out the next 3 assembly instructions to execute.
5. `x/1gx $rsp` to print out the 8 byte value at the top of the stack. ("Examine (1) (g)iantword in he(x) starting at \$rsp")



~ 30 min + 10 min all-lab discussion



(gdb) `break kermi`

(gdb) `disassemble kermi`



Check-Off Q2: `kernit` ()



What is the significance of the **8 byte value at the top of the stack** right when **`kernit`** (or any other function) is called?

```
(gdb) x/1gx $rsp  
0x7fffffffef718: 0x0000000004011de
```

Value at top of stack!



Check-Off Q2: `kermi t` ()



What is the significance of the **8 byte value at the top of the stack** right when **`kermi t`** (or any other function) is called?

```
(gdb) x/1gx $rsp
```

```
0x7fffffffef718:0x00000000004011de
```

Value at top of stack!

```
(gdb) disass main
```

Dump of assembler code for function main:

```
0x00000000004011c4 <+0>:    push    %rbx
0x00000000004011c5 <+1>:    mov     $0x402008,%edi
0x00000000004011ca <+6>:    mov     $0x0,%eax
0x00000000004011cf <+11>:   call    0x401030 <printf@plt>
0x00000000004011d4 <+16>:   mov     $0x6b,%edi
0x00000000004011d9 <+21>:   call    0x4011a7 <kermi t>
0x00000000004011de <+26>:   mov     %eax,%ebx
0x00000000004011e0 <+28>:   mov     $0x2,%esi
```



Check-Off Q2: `kermi t ( )`



What is the significance of the **8 byte value at the top of the stack** right when **`kermi t`** (or any other function) is called?

```
(gdb) x/1gx $rsp
```

```
0x7fffffffef718: 0x00000000004011de
```

Value at top of stack!

```
(gdb) disass main
```

Dump of assembler code for function main:

```
0x00000000004011c4 <+0>:    push    %rbx
                                mov     $0x402008,%edi
                                mov     $0x0,%eax
                                :      call   0x401030 <printf@plt>
                                :      mov     $0x6b,%edi
0x00000000004011d9 <+21>:    call    0x4011a7 <kermi t>
0x00000000004011de <+26>:    mov     %eax,%ebx
0x00000000004011e0 <+28>:    mov     $0x2,%esi
```

Same as address of the instruction in **`main ( )`** which follows the **`call kermi t`** instruction.



Check-Off Q2: `kermi t ( )`



What is the significance of the **8 byte value at the top of the stack** right when **`kermi t`** (or any other function) is called?

```
(gdb) x/1gx $rsp
```

```
0x7fffffffef718: 0x00000000004011de
```

Value at top of stack!

```
(gdb) disass main
```

Dump of assembler code for function main:

```
0x00000000004011c4 <+0>:      push    %rbx
```

Same as address of the instruction in `main ( )` which follows the `call kermi t` instruction.

Solution: This is the **return address** i.e. address of the caller instruction the program resumes from once `kermi t ( )` is done calling ☒

```
0x00000000004011d9 <+21>:      call     0x4011a7 <kermi t>
```

```
0x00000000004011de <+26>:      mov      %eax, %ebx
```

```
0x00000000004011e0 <+28>:      mov      $0x2, %esi
```



Check-Off Q: Printing Array in Clyde

Previously, we were in the `kermi t()` function. Now, let's dive into `clyde()` to print out an array in its vivid entirety, using clues gathered from assembly!

```
(gdb) break clyde  
Breakpoint 2 at 0x401138: file stack.c, line 14.  
(gdb) continue  
Continuing.
```

Breakpoint on `clyde(arr, length)`,
and `continue / run` from `kermi t()`.



Check-Off Q: Printing Array in Clyde

Previously, we were in the `kermi t()` function. Now, let's dive into `clyde()` to print out an array in its vivid entirety, using clues gathered from assembly!

```
(gdb) break clyde
Breakpoint 2 at 0x401138: file stack.c, line 14.
(gdb) continue
Continuing.

Breakpoint 2, clyde (arr=arr@entry=0x7fffffff774,
length=length@entry=3) at stack.c:14
14      for (int i = 0; i < length; i++) {
(gdb) print /x $rdi
$1 = 0x7fffffff774
(gdb) print $rsi
$2 = 3
(gdb) print ((int *)$rdi)[0]@$rsi
$3 = {1, 2, 3}
```

Breakpoint on `clyde(arr, length)`, and `continue / run` from `kermi t()`.

`$rdi`, 1st argument to `clyde`: address of the array's base!

`$rsi`, 2nd arg — # of elements!

Printing all `$rsi` (3) elements of integer array, starting from the base at address `$rdi`, casted to an `int*` ✓

```
(gdb) print ((int*)$rdi)[0] @ $rsi
```

Reading the Assembly Loop in Clyde

```
<+0>:    mov     $0x0,%eax
<+5>:    jmp     0x401150 <clyde+24>
<+7>:    movslq   %eax,%rdx
<+10>:   lea     (%rdi,%rdx,4),%rcx
<+14>:   mov     (%rcx),%edx
<+16>:   add     $0x1,%edx
<+19>:   mov     %edx, (%rcx)
<+21>:   add     $0x1,%eax
<+24>:   cmp     %esi,%eax
<+26>:   jl      0x40113f <clyde+7>
<+28>:   movslq   %esi,%rsi
<+31>:   mov     -0x4(%rdi,%rsi,4),%eax
<+35>:   ret
```

```
int clyde(int arr[], int length) {
    for (int i = 0; i < length; i++) {
        arr[i]++;
    }
    return arr[length - 1];
}
```

Reading the Assembly Loop in Clyde

With a loop, we look for:

(1) **initialization** of loop variable / counter

(2) **Loop check / condition** to keep iterating

(3) **Update** or incrementation to loop variable

```
<+0>:    mov     $0x0,%eax
<+5>:    jmp     0x401150 <clyde+24>
<+7>:    movslq   %eax,%rdx
<+10>:   lea     (%rdi,%rdx,4),%rcx
<+14>:   mov     (%rcx),%edx
<+16>:   add     $0x1,%edx
<+19>:   mov     %edx,(%rcx)
<+21>:   add     $0x1,%eax
<+24>:   cmp     %esi,%eax
<+26>:   jl      0x40113f <clyde+7>
<+28>:   movslq   %esi,%rsi
<+31>:   mov     -0x4(%rdi,%rsi,4),%eax
<+35>:   ret
```

```
int clyde(int arr[], int length) {
    for (int i = 0; i < length; i++) {
        arr[i]++;
    }
    return arr[length - 1];
}
```

Reading the Assembly Loop in Clyde

With a loop, we look for:

(1) initialization of loop variable / counter 

(2) Loop check / condition to keep iterating

(3) Update or incrementation to loop variable

```
<+0>:    mov     $0x0,%eax
<+5>:    jmp     0x401150 <clyde+24>
<+7>:    movslq   %eax,%rdx
<+10>:   lea     (%rdi,%rdx,4),%rcx
<+14>:   mov     (%rcx),%edx
<+16>:   add     $0x1,%edx
<+19>:   mov     %edx, (%rcx)
<+21>:   add     $0x1,%eax
<+24>:   cmp     %esi,%eax
<+26>:   jl      0x40113f <clyde+7>
<+28>:   movslq   %esi,%rsi
<+31>:   mov     -0x4(%rdi,%rsi,4),%eax
<+35>:   ret
```

```
int clyde(int arr[], int length) {
    for (int i = 0; i < length; i++) {
        arr[i]++;
    }
    return arr[length - 1];
}
```

Note: Moving **0** into **%eax** mirrors the initialization of counter **i = 0**

Reading the Assembly Loop in Clyde

With a loop, we look for:

(1) initialization of loop variable / counter 

(2) Loop check / condition to keep iterating 

(3) Update or incrementation to loop variable

```
<+0>:    mov     $0x0,%eax
<+5>:    jmp     0x401150 <clyde+24>
<+7>:    movslq  %eax,%rdx
<+10>:   lea     (%rdi,%rdx,4),%rcx
<+14>:   mov     (%rcx),%edx
<+16>:   add     $0x1,%edx
<+19>:   mov     %edx,(%rcx)
<+21>:   add     $0x1,%eax
<+24>:   cmp     %esi,%eax
<+26>:   jl      0x40113f <clyde+7>
<+28>:   movslq  %esi,%rsi
<+31>:   mov     -0x4(%rdi,%rsi,4),%eax
<+35>:   ret
```

```
int clyde(int arr[], int length) {
    for (int i = 0; i < length; i++) {
        arr[i]++;
    }
    return arr[length - 1];
}
```

Note: With **jl**, if **%eax < %esi** (equivalently **i < length**), we jump back into the loop. Else, we exit.

Reading the Assembly Loop in Clyde

With a loop, we look for:

(1) **initialization** of loop variable / counter 

(2) **Loop check / condition** to keep iterating 

(3) **Update** or incrementation to loop variable 

```
<+0>:    mov     $0x0,%eax
<+5>:    jmp     0x401150 <clyde+24>
<+7>:    movslq  %eax,%rdx
<+10>:   lea     (%rdi,%rdx,4),%rcx
<+14>:   mov     (%rcx),%edx
<+16>:   add     $0x1,%edx
<+19>:   mov     %edx,(%rcx)
<+21>:   add     $0x1,%eax
<+24>:   cmp     %esi,%eax
<+26>:   jl      0x40113f <clyde+7>
<+28>:   movslq  %esi,%rsi
<+31>:   mov     -0x4(%rdi,%rsi,4),%eax
<+35>:   ret
```

```
int clyde(int arr[], int length) {
    for (int i = 0; i < length; i++) {
        arr[i]++;
    }
    return arr[length - 1];
}
```

Note: Before heading back into the loop check, we perform update **\$eax += 1**, or equivalently **i++**;

Reading the Assembly Loop in Clyde

With a loop, we look for:

(1) initialization of loop variable / counter 

(2) Loop check / condition to keep iterating 

(3) Update or incrementation to loop variable 

```
<+0>:    mov     $0x0,%eax
<+5>:    jmp     0x401150 <clyde+24>
<+7>:    movslq   %eax,%rdx
<+10>:   lea     (%rdi,%rdx,4),%rcx
<+14>:   mov     (%rcx),%edx
<+16>:   add     $0x1,%edx
<+19>:   mov     %edx, (%rcx)
<+21>:   add     $0x1,%eax
<+24>:   cmp     %esi,%eax
<+26>:   jl      0x40113f <clyde+7>
<+28>:   movslq   %esi,%rsi
<+31>:   mov     -0x4(%rdi,%rsi,4),%eax
<+35>:   ret
```

```
int clyde(int arr[], int length) {
    for (int i = 0; i < length; i++) {
        arr[i]++;
    }
    return arr[length - 1];
}
```

Once we have these details, we now know **what's in the body of the loop** :), which leads to our check-off Q! 🎉



Check-Off Q: Watchpoints in Clyde

In the `clyde()` function, what assembly instruction is the culprit responsible for directly modifying `arr[1]`?

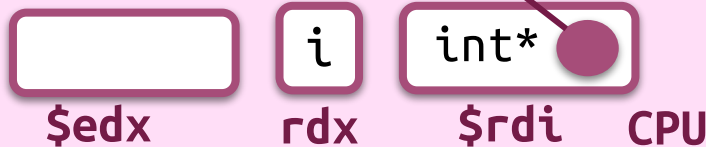
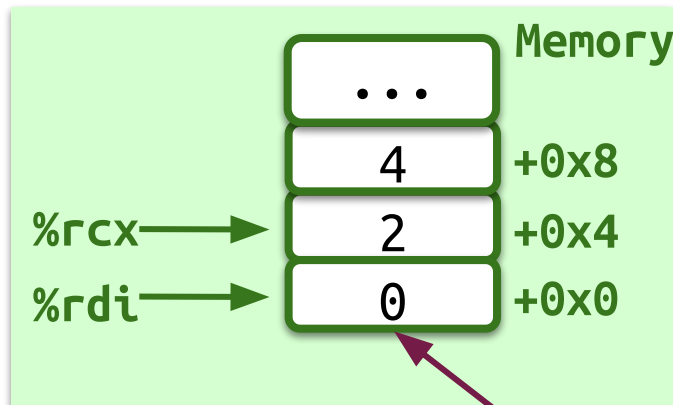


```
<+10>  lea  (%rdi,%rdx,4),%rcx
```

```
<+14>:  mov  (%rcx),%edx
```

```
<+16>:  add  $0x1,%edx
```

```
<+19>:  mov  %edx, (%rcx)
```



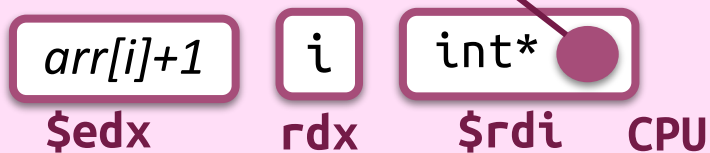
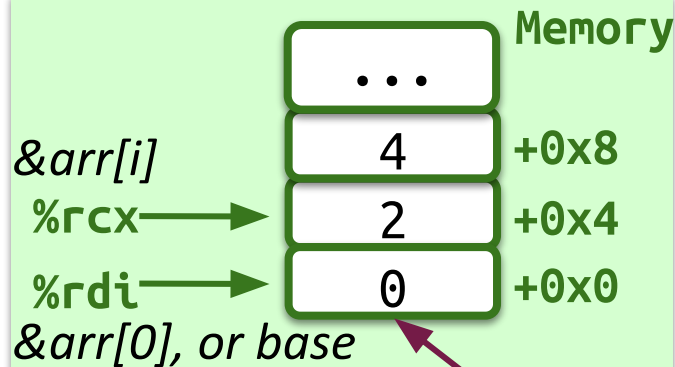


Check-Off Q: Watchpoints in Clyde

In the `clyde()` function, what assembly instruction is the culprit responsible for directly modifying `arr[1]`?



```
// gets address of ith element
<+10>  lea (%rdi,%rdx,4),%rcx
// moves arr[i] value into reg
<+14>:  mov (%rcx),%edx
// increments reg value by 1
<+16>:  add $0x1,%edx
// moves new value into arr[i]
<+19>:  mov %edx, (%rcx)
```





Check-Off Q: Watchpoints in Clyde

In the `clyde()` function, what assembly instruction is the culprit responsible for directly modifying `arr[1]`?



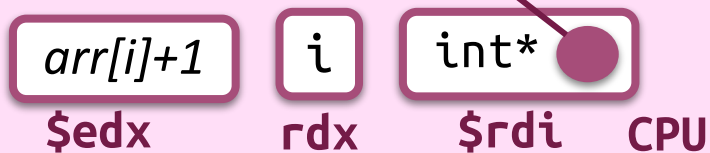
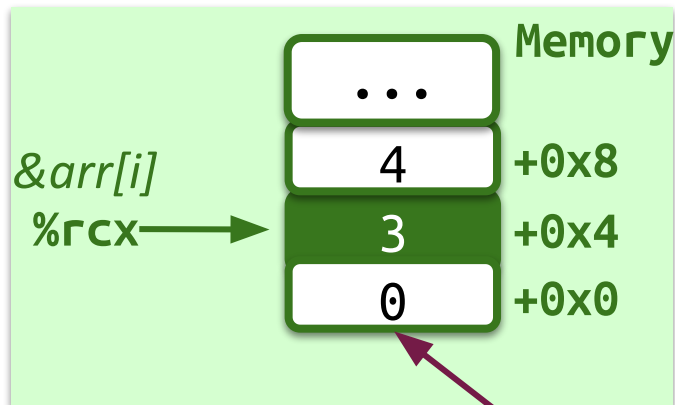
Solution: `mov %edx, (%rcx)` ✓

It's tricky! The prior instruction increments a register holding the value `arr[i]`, but the **new value** `arr[i] + 1` is **moved into memory** only in the next instruction, so `arr[i]` is modified there (note `%rcx` holds its address, and is being dereferenced).

```
<+16>: add $0x1,%edx
```

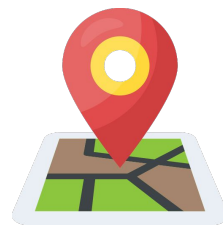
```
// moves new value into arr[i]
```

```
<+19>: mov %edx, (%rcx)
```



Happy to discuss any questions!

Navigating assembly with GDB, function calls in assembly, etc.

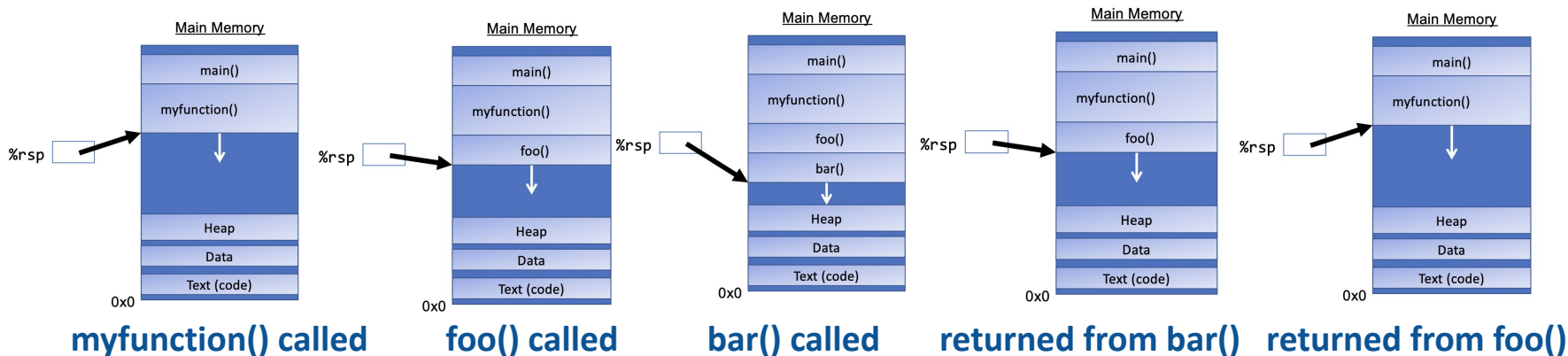




Special Registers: Stack Pointer `%rsp`

`%rsp` stores the **address of the current top of the stack** (note: visually,  the stack grows downward). Instructions can adjust `%rsp` by:

- **Explicitly incrementing / decrementing** it to make more or less space
- **Pushing or popping** things on/off the stack using the **push** and **pop** instructions, or **call**, which pushes the return address onto stack





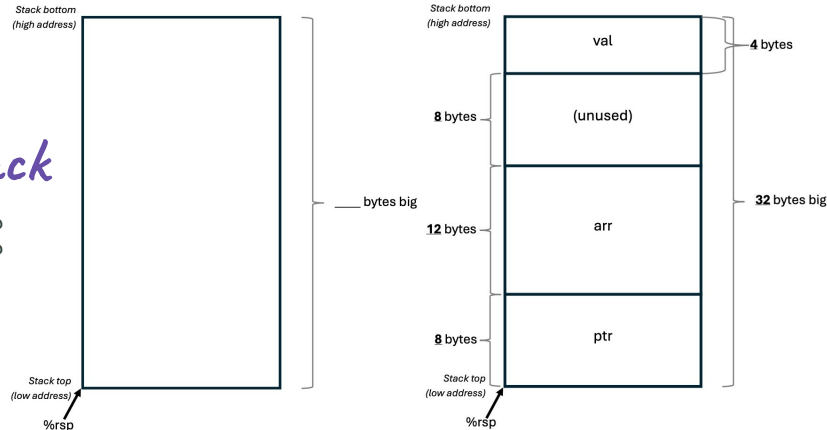
Exercise 2: Stack Smash

In your `lab6` directory, you'll want to examine the `smash.c` program, and try to **expose bugs** with stack management **by inspecting the underlying assembly**.



You'll want to get out a piece of paper to draw and annotate the stack diagram, and work in pairs/groups!

Example Stack
Diagram 



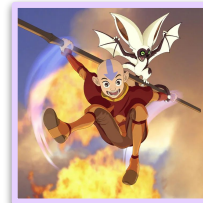
This exercise is very similar to a part of **A5: SecureVault!**



Exercise 2: Stack Smash

 **Goal:** Inside `./smash` is a function `greet()` that **returns 2** all the time, or does it? Here, we'll design an input that hacks it to **return 107** instead! :)

```
bbyan@myth64:~/CS107_TA/assignments/lab6$ ./smash
You've got 5 tries to blow this popsicle stand. Go!
What is your name? Jerry Cain
Hello, Jerry Cain! Buffer has space for 16 chars, your name is length 10
.
greet() returned 2
What is your name? Ben yan
Hello, Ben yan! Buffer has space for 16 chars, your name is length 7.
greet() returned 2
What is your name? Water. Earth. Fire. Air. Long ago, the four nations l
ived together in harmony. Then, everything changed when the Fire Nation
attacked.
Hello, Water. Earth. Fire. Air. Long ago, the four nations lived togethe
r in harmony. Then, everything changed when the Fire Nation attacked.! B
uffer has space for 16 chars, your name is length 134.
You have overrun the buffer by 119 chars.
Segmentation fault
bbyan@myth64:~/CS107_TA/assignments/lab6$
```



Vulnerability with C function `gets ()`

Our `greet ()` function relies on `gets (char* buf)` to read an input string from the command line, and store it in a C string `buf`.



`gets ()` also adds a `\0` at the end, which the buffer should accommodate.

```
bbyan@myth65:~/CS107_TA/assignments/lab6$ ./smash
You've got 5 tries to blow this popsicle stand. G
What is your name? Mclovin
Hello, Mclovin! Buffer has space for 16 chars, yo
```

M	c	l	o	v	i	n	\0	...
---	---	---	---	---	---	---	----	-----

`char buf[16];`

However, we don't know how many characters the user will enter, and `gets ()` will try writing all of them in memory, even if it overflows `buf`!



Thus, `gets ()` is very dangerous security-wise, so use `fgets ()` instead :)

**Drawing an Entire Stack
Diagram from Assembly**
Yay let's get started! ✨





Assembly and C code for greet ()



Compiler-generated assembly

```
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>
```



C source code

```
#define BUF_LEN 16

int greet() {
    int num = 2;
    char buf[BUF_LEN];

    printf("What is your name? ");
    if (gets(buf)) {
        printf("Hello, %s! Buffer
               buf, sizeof(buf), strlen(buf) - 1);
        if (strlen(buf) + 1 > BUF_LEN)
            printf("You have overran the buffer.
                   strlen(buf) - 1);
    }

    return num;
}
```



`greet ()` Assembly → Stack Diagram

Q1: How much stack space is initially reserved for `greet`?

`rsp`



other stack memory

```
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>
```

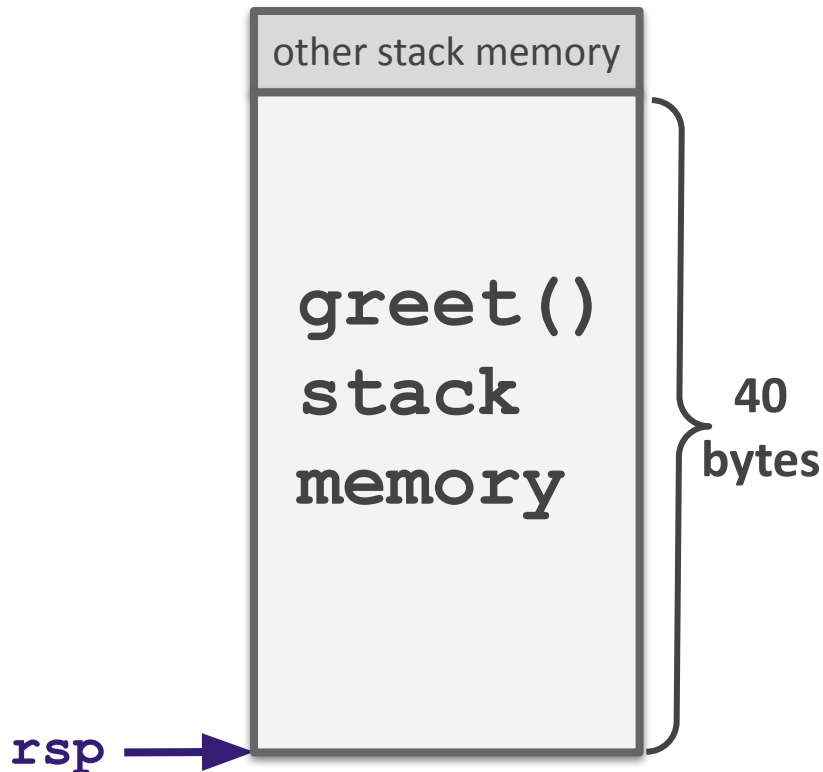


`greet ()` Assembly → Stack Diagram

Q1: How much stack space is initially reserved for `greet`?

A: 40 bytes. Stack pointer `$rsp` descends by `0x28` (40 decimal) bytes to make space.

```
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>
```

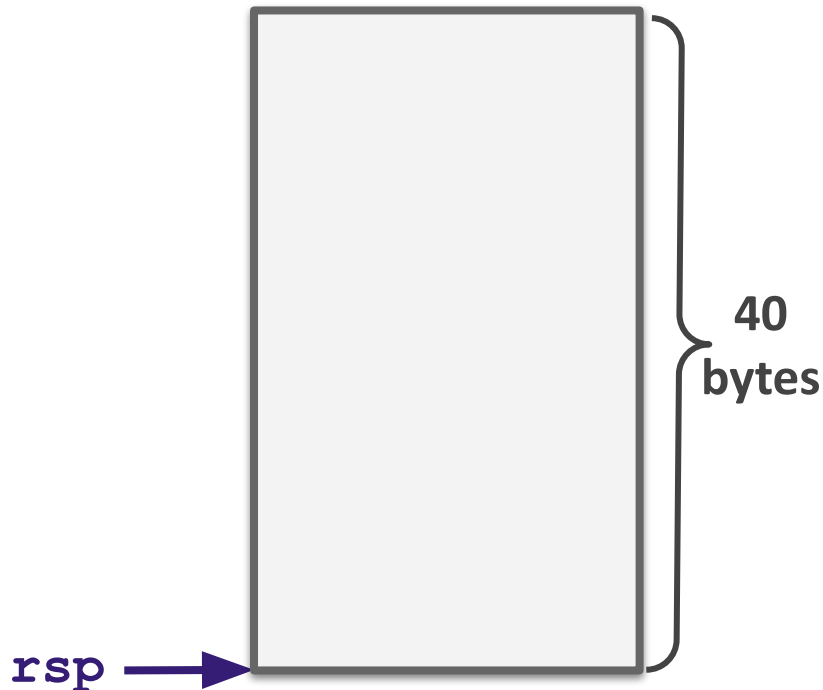




`greet ()` Assembly → Stack Diagram

Q2: Where is `num` (`int` variable, equal to 2) stored, relative to stack pointer `$rsp`?

```
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>
```





`greet ()` Assembly → Stack Diagram

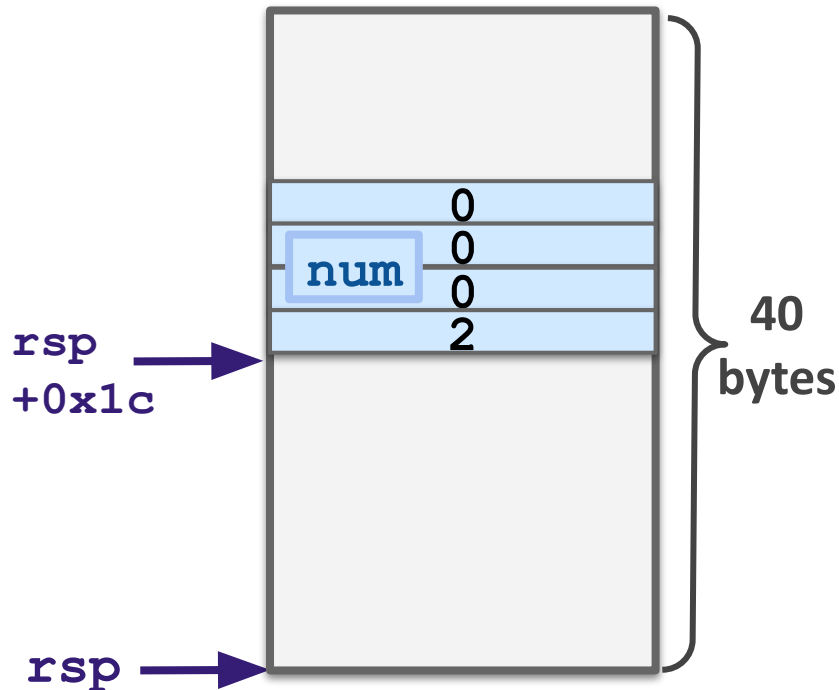
Q2: Where is `num` (`int` variable, equal to 2) stored, relative to stack pointer `$rsp`?

A: 28 bytes up from `$rsp` (bedrock of `greet`'s stack)

```
sub    $0x28,%rsp
movl   $0x2,0x1c(%rsp)
```

This moves a 4-byte integer (value 2) into address `$rsp + 28` in decimal.

```
mov    %rsp,%rax
mov    %rax,%rdi
callq  0x401060 <gets@plt>
```





`greet ()` Assembly → Stack Diagram

Q2: Where is `num` (`int` variable, equal to 2) stored, relative to stack pointer `$rsp`?

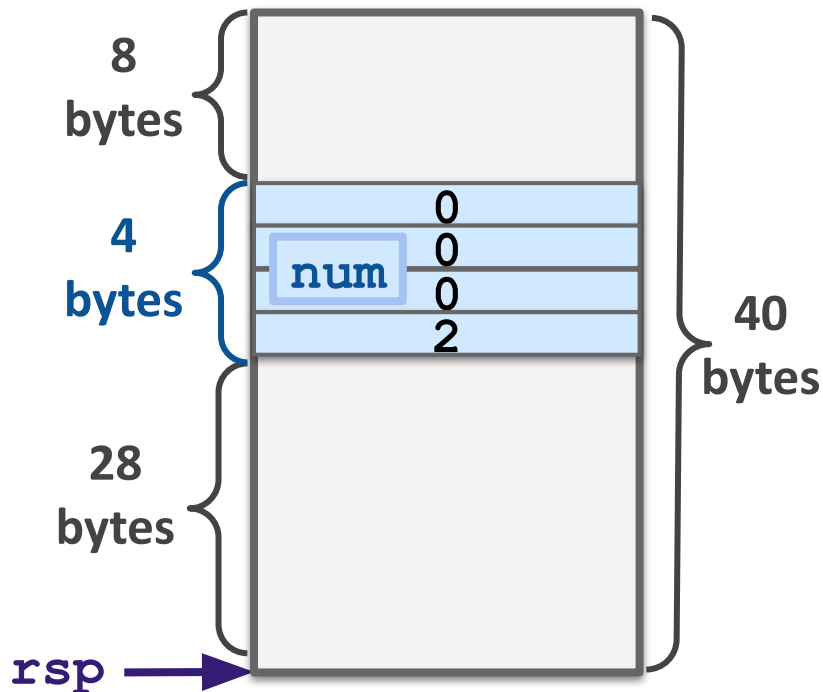
A: 28 bytes up from `$rsp` (bedrock of `greet`'s stack), or 8 bytes from the top.

```
sub    $0x28,%rsp
movl   $0x2,0x1c(%rsp)
```

This moves a 4-byte integer (value 2) into address `$rsp + 28` in decimal.

```
mov     %rsp,%rax
mov     %rax,%rdi
callq   0x401060 <gets@plt>
```

Note: Figure not drawn to scale

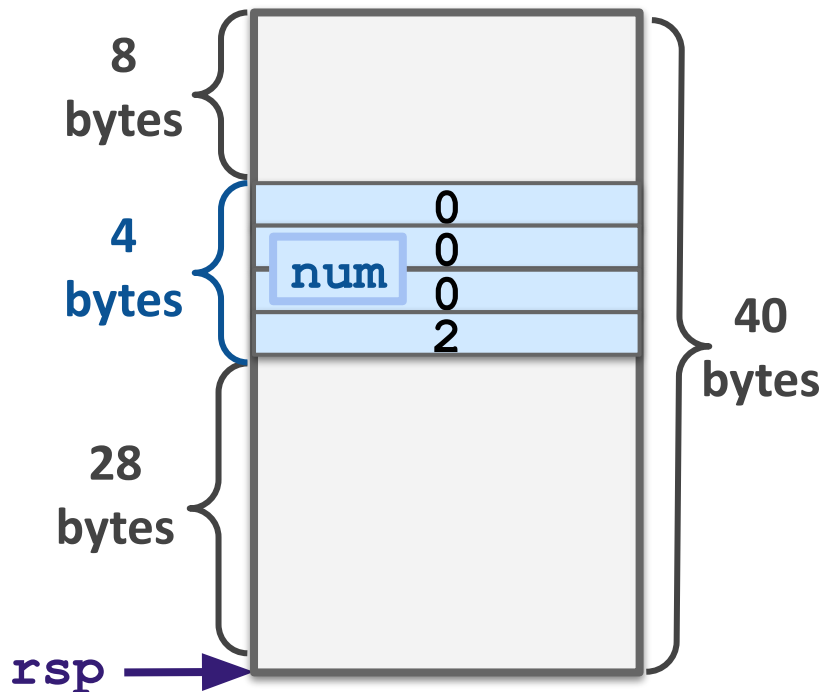




`greet ()` Assembly → Stack Diagram

Q3: Where is `buf` stored, relative to `$rsp`?

```
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>
```



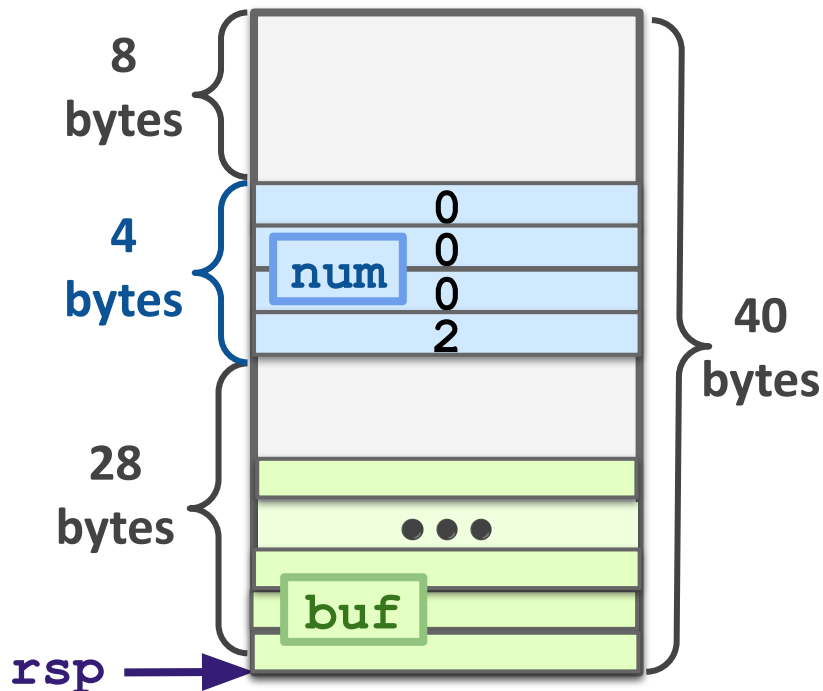


`greet ()` Assembly → Stack Diagram

Q3: Where is `buf` stored, relative to `$rsp`?

A: Directly at `$rsp`! We have `$rsp` passed into `gets (buf)` as its 1st parameter (`$rdi`), so that's where the buffer starts.

```
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>
```





`greet ()` Assembly → Stack Diagram

Q3: Where is `buf` stored, relative to `$rsp`?

A: Directly at `$rsp`! We have `$rsp` passed into `gets (buf)` as its 1st parameter (`$rdi`), so that's where the buffer starts.

```
sub    $0x28,%rsp
```

We know from the source code that `buf` is **16 bytes large**, which leaves 12 bytes between the  buffer's end and `num`.

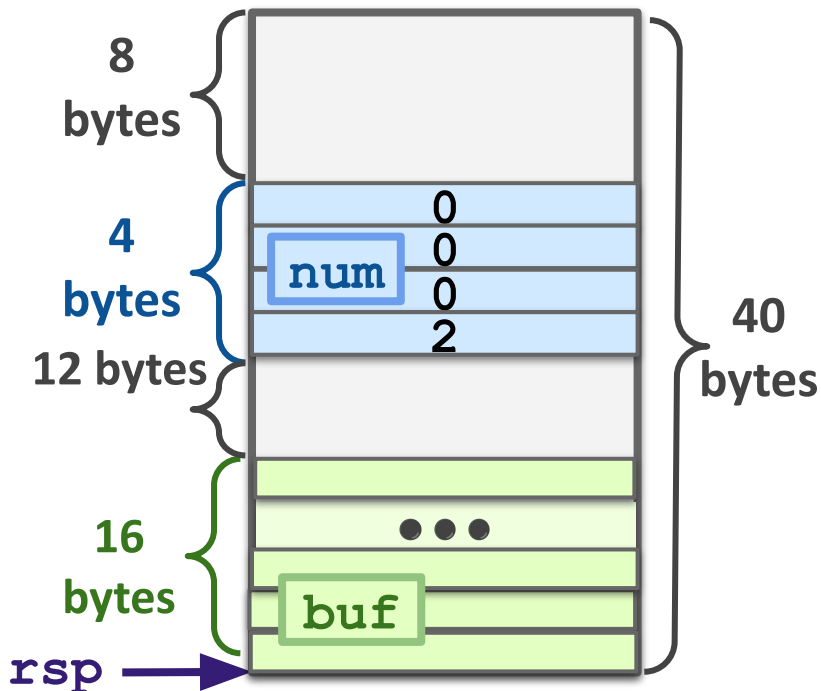
```
callq  0x401050 <printf@plt>
```

```
mov     %rsp,%rax
```

```
mov     %rax,%rdi
```

```
callq   0x401060 <gets@plt>
```

Figure is really not to scale lol



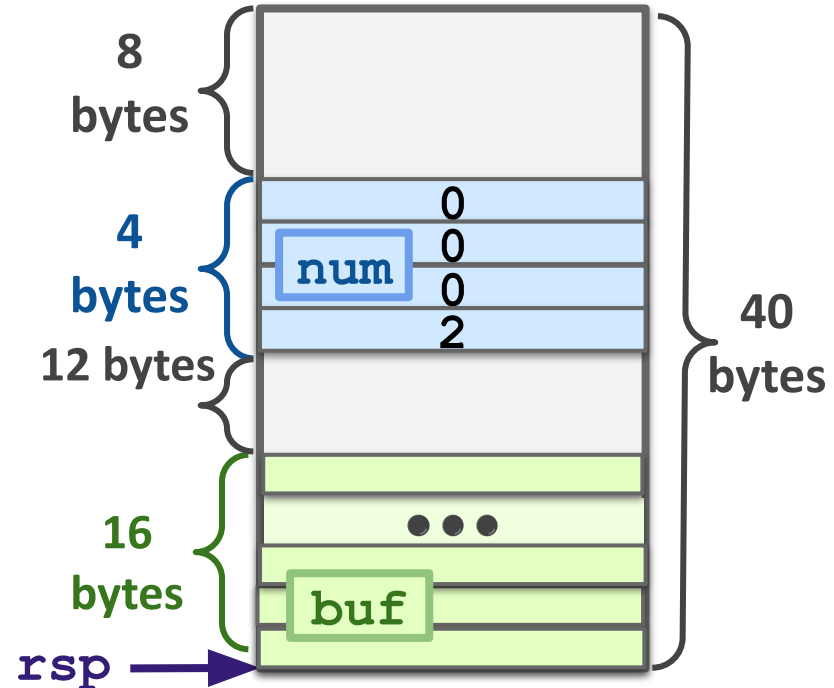


Check-Off Question: greet Stack Diagram

In the **greet** function, how much space (in bytes) is there on the stack **between** the end of **buf** and the start of **num**?

```
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>
```

Figure's scale is cooked



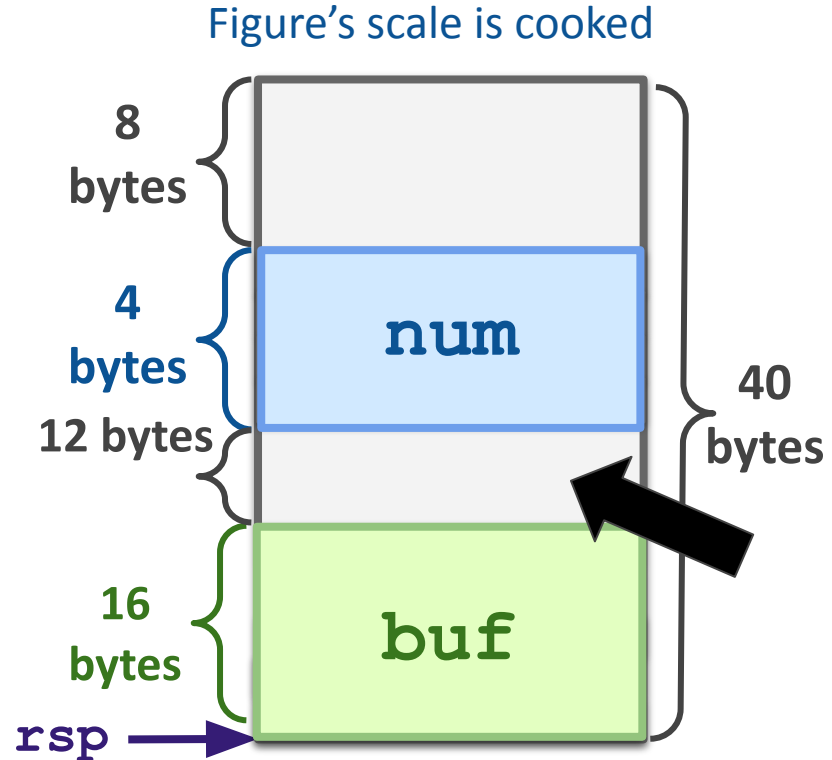


Check-Off Question: greet Stack Diagram

In the **greet** function, how much space (in bytes) is there on the stack **between** the end of **buf** and the start of **num**?

A: 12 bytes (middle gray area on right)

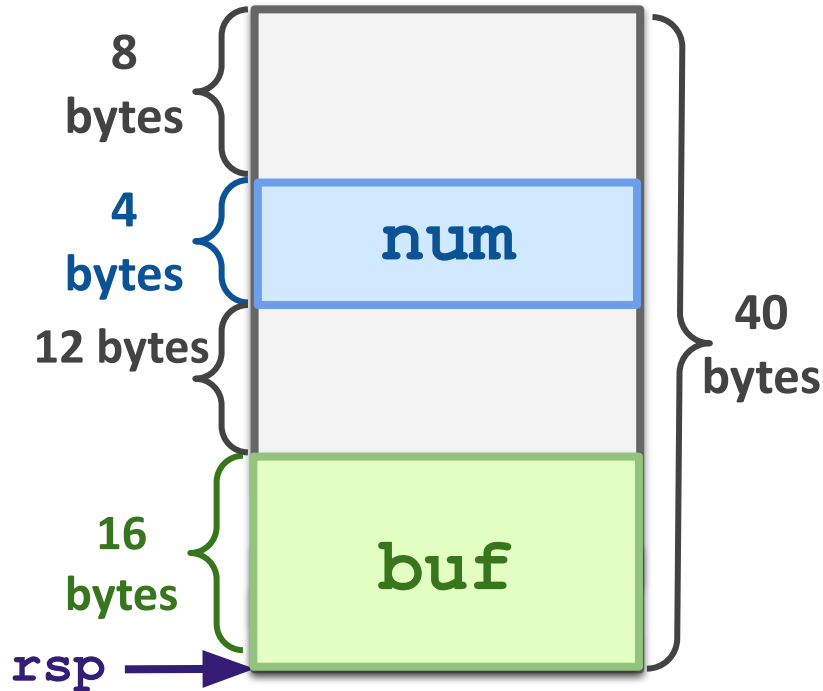
```
sub    $0x28,%rsp
movl   $0x2,0x1c(%rsp)
mov     $0x402008,%edi
mov     $0x0,%eax
callq   0x401050 <printf@plt>
mov     %rsp,%rax
mov     %rax,%rdi
callq   0x401060 <gets@plt>
```





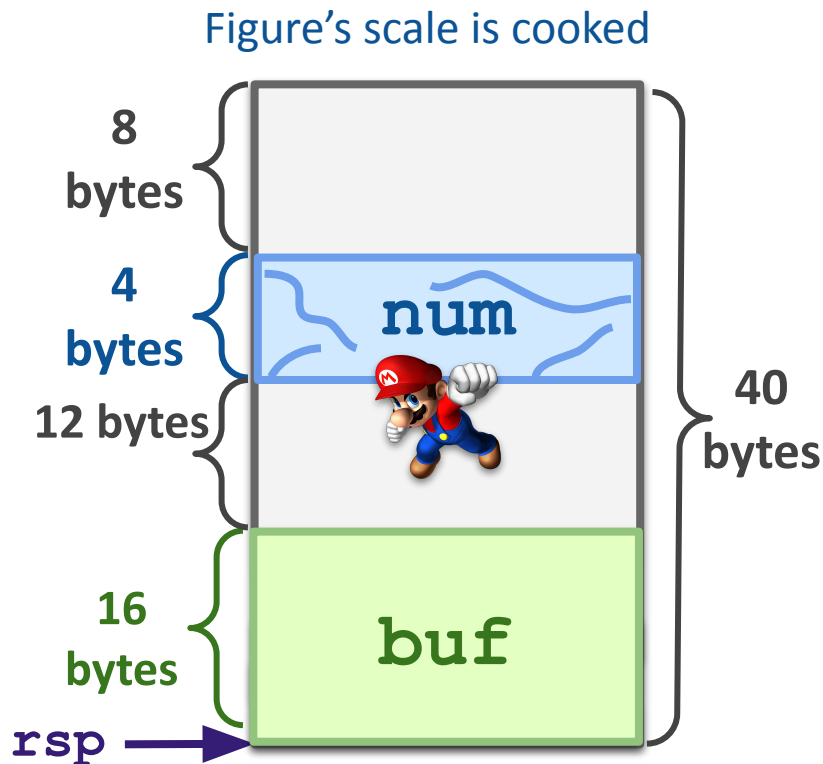
Voilà! The full stack diagram of `greet()`

Figure's scale is cooked





Voilà! The full stack diagram of greet()



Now, what happens when we write characters past **buf**?

And via buffer overflow, can we infiltrate the supervillain LaIR of **num**, the variable we return?



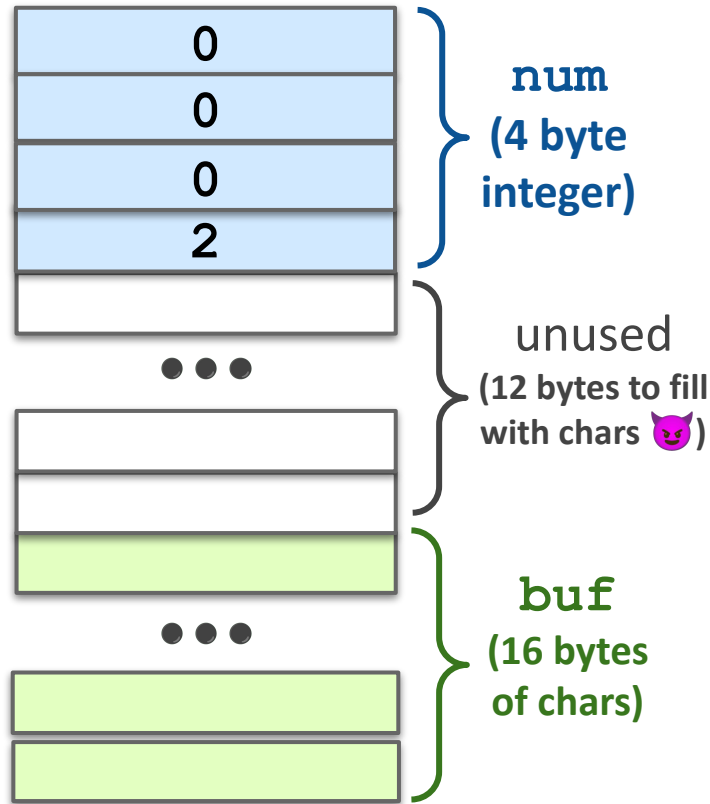
~~We not watching reels for 2 hours each morning~~



Check-Off Q: Stack Smash!

MISSION:IMPOSSIBLE

Here, we'll accomplish the impossible:
tricking `greet()` to return 107! 🎃



~~We not watching reels for 2 hours each morning~~

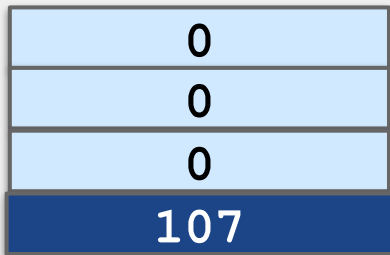


Check-Off Q: Stack Smash!

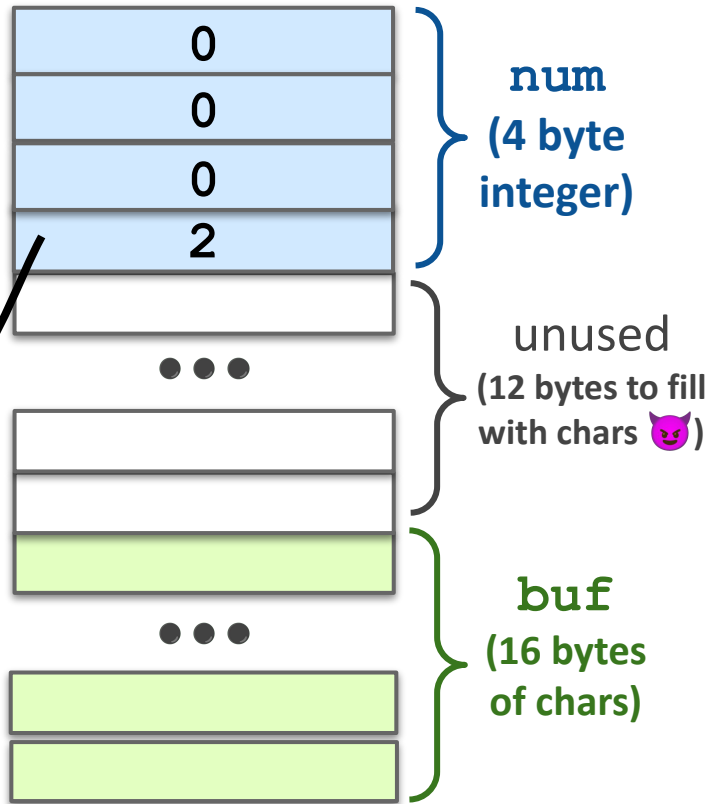
MISSION:IMPOSSIBLE

Here, we'll accomplish the impossible:
tricking `greet ()` to return 107! 🎃

In other words, in memory, can we can
override the lowest byte of num to be 107?



Note that `gets ()` will start writing input characters in **buf**, overflowing up to **num**.



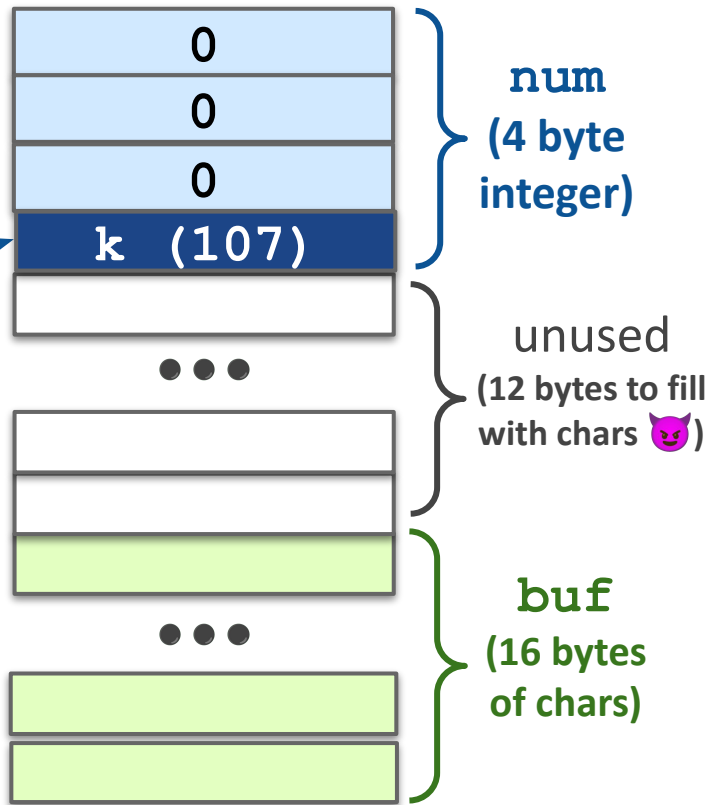


Check-Off Q: Stack Smash!

Here, we'll accomplish the impossible:
tricking `greet()` to return `107`! 🎃

Yes! Note that the ASCII character with value `107` is `k`, so in the bytes that belong to `num`, we have to place a `k`.

Due to `num` being an `int` data type, the `k` will be interpreted / casted as `107`.



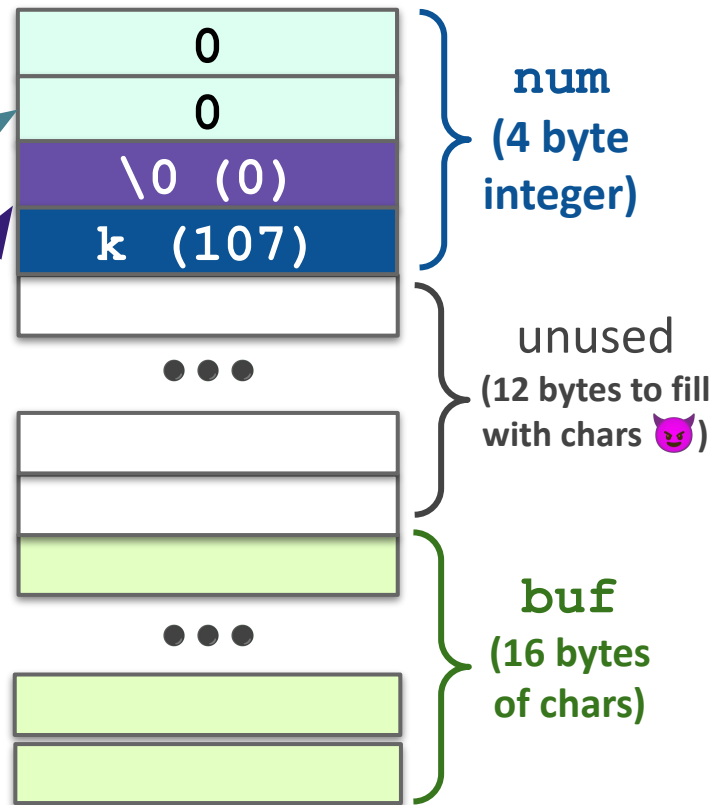


Check-Off Q: Stack Smash!

Here, we'll accomplish the impossible:
tricking `greet ()` to return 107! 🎃

Note higher bytes stay 0 as `num` was
initialized to 2. To make `num` even
larger, you can override them also :)

Also, `gets ()` follows the last character
with a `\0`, which keeps ASCII value 0!





Check-Off Q: Stack Smash!

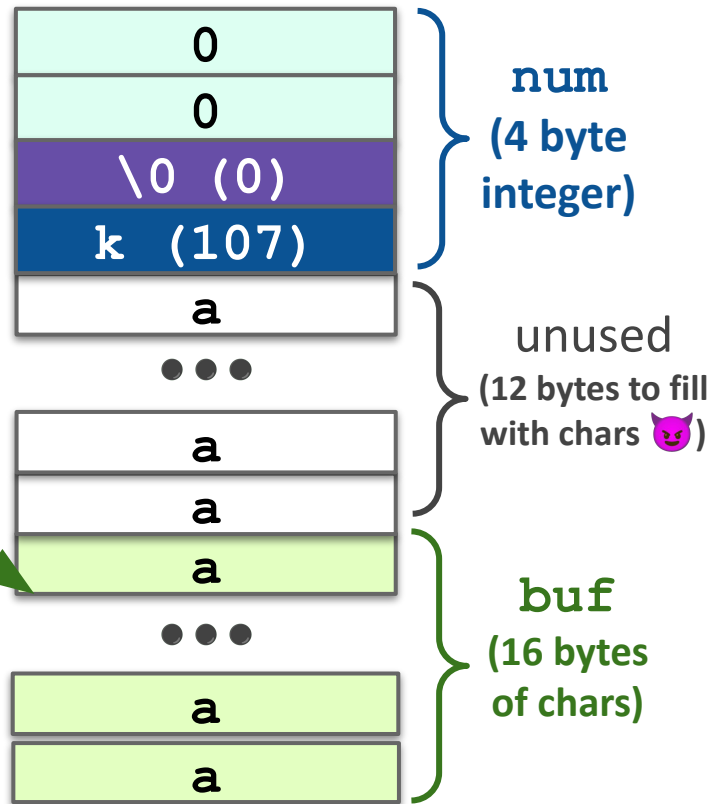
Here, we'll accomplish the impossible:
tricking `greet ()` to return 107! 🎃

The **buffer** + **unused** memory is 28 bytes, so our input should be a string of 29 chars total, ending in `k` :) ☐



Example: `aaaaaa...aak`

28 copies of `a`





Check-Off Q: Stack Smash!

“Everything is possible, even the impossible” – Mary Poppins



A more vivid solution: supercali . . . expialidk

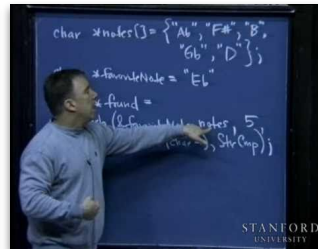
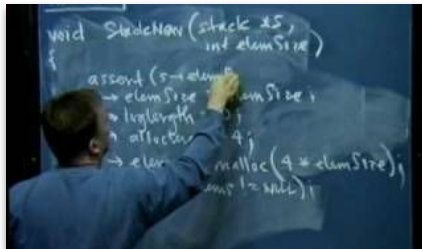
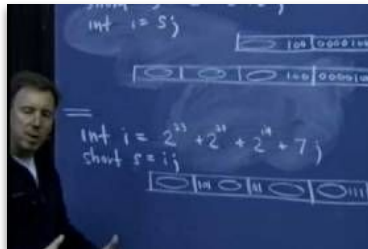
28 letters of anything you heart desires, followed by a **k**

```
bbyan@myth65:~/CS107_TA/assignments/lab6$ ./smash
You've got 5 tries to blow this popsicle stand. Go!
What is your name? supercalifragilisticexpialidk
Hello, supercalifragilisticexpialidk! Buffer has space
for 16 chars, your name is length 29.
You have overrun the buffer by 14 chars.
greet() returned 107
You did it?? But how??
```





Anonymous feedback form, appreciate any lab/teaching feedback! ❤️



Thank you and have a beautiful weekend!

- Please reach out to me if you have any questions (e.g., assembly reading tips), or topics you want covered!
-  Contact: bbyan@stanford.edu, email or over Slack
- We don't have lab next week (a 🎁 107 holiday of sorts for working on Binary Bomb), so see you in 2 weeks!



“What if we had a binary bomb but for CS106A, like a Pynary bomb”
– a random idea :)