The Stack & Procedures

CSE 410 Winter 2017

Instructor:

Justin Hsia

Teaching Assistants:

Kathryn Chan, Kevin Bi, Ryan Wong, Waylon Huang, Xinyu Sui

AI Decisively Defeats Human Poker Players

Libratus lived up to its "balanced but forceful" Latin name by becoming the first AI to beat professional poker players at heads-up, no-limit Texas Hold'em. Developed by Carnegie Mellon University, the AI won the "Brains Vs. Artificial Intelligence" tournament against four poker pros by $1,766,250 in chips over 120,000 hands (games). Researchers can now say that the victory margin was large enough to count as a statistically significant win (99.7 percent certainty).

Libratus focuses on improving its own play, [described] as safer and more reliable compared to the riskier approach of trying to exploit opponent mistakes.

- http://spectrum.ieee.org/automaton/robotics/artificial-intelligence/ai-learns-from-mistakes-to-defeat-human-poker-players

Administrivia

- Homework 3 released today, due next Thu (2/9)
- Lab 2 deadline pushed to Monday (2/13)
  - Definitely want to start before the Midterm
- **Midterm** (2/10) in lecture
  - Reference sheet + 1 *handwritten* cheat sheet
  - Find a study group! Look at past exams!
  - Aiming for average of 75%
- **Midterm review session** (2/7) in BAG 261 from 5-7:30pm

Procedures

- Stack Structure
- **Calling Conventions**
  - Passing control
  - **Passing data**
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion

Procedure Data Flow

Registers (NOT in Memory)

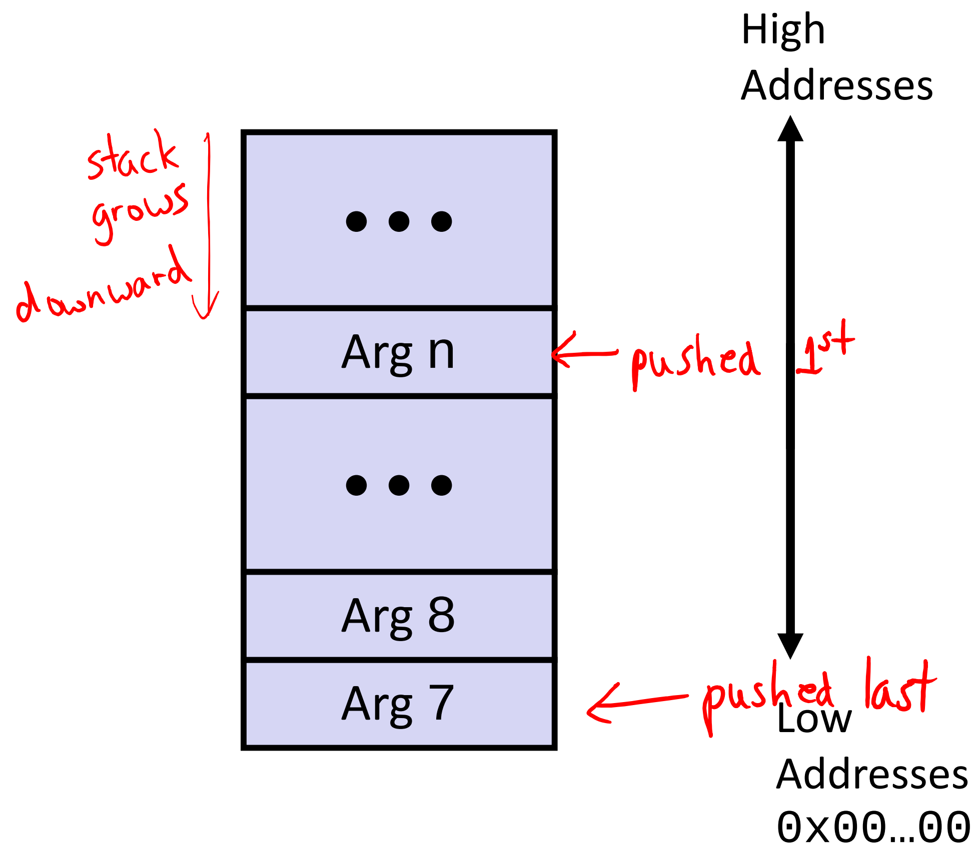
- First 6 arguments

`%rdi`	*Diane's*
`%rsi`	*Silk*
`%rdx`	*Dress*
`%rcx`	*Costs*
`%r8`	*$8 9*
`%r9`	

- Return value

`%rax`

Stack (Memory)



- Only allocate stack space when needed

x86-64 Return Values

- By convention, values returned by procedures are placed in `%rax`
  - Choice of `%rax` is arbitrary

1) Caller must make sure to save the contents of `%rax` before calling a callee that returns a value
   - Part of register-saving convention

2) Callee places return value into `%rax`
   - Any type that can fit in 8 bytes – integer, float, pointer, etc.
   - For return values greater than 8 bytes, best to return a *pointer* to them

3) Upon return, caller finds the return value in `%rax`

Data Flow Examples

```
void multstore
 (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

rdi rsi rdx

rdi rsi

lined up nicely so we didn't have to manipulate arguments

```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
    • • •
  400541: movq   %rdx,%rbx        # "Save" dest
  400544: call   400550 <mult2> # mult2(x,y)
  # t in %rax
  400549: movq   %rax,(%rbx)      # Save at dest
    • • •
```

(will explain later)

```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550:  movq   %rdi,%rax   # a
  400553:  imulq  %rsi,%rax   # a * b
  # s in %rax
  400557:  ret                # Return
```

Procedures

- Stack Structure
- **Calling Conventions**
  - Passing control
  - Passing data
  - **Managing local data**
- Register Saving Conventions
- Illustration of Recursion

Stack-Based Languages

- Languages that support recursion
  - *e.g.* C, Java, most modern languages
  - Code must be *re-entrant*
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store *state* of each instantiation
    - Arguments, local variables, return pointer
- Stack allocated in *frames*
  - State for a single procedure instantiation
- Stack discipline
  - State for a given procedure needed for a limited time
    - Starting from when it is called to when it returns
  - Callee always returns before caller does

Call Chain Example

```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   •
   amI();
   •
   amI();
   •
}
```

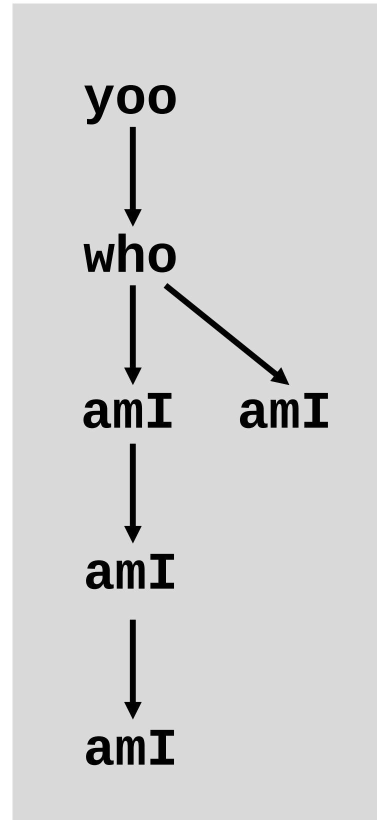
1st call recurses twice

2nd call doesn't recurse

```
amI(…)
{
   •
   if(…){
     amI()
   }
   •
}
```

based on condition

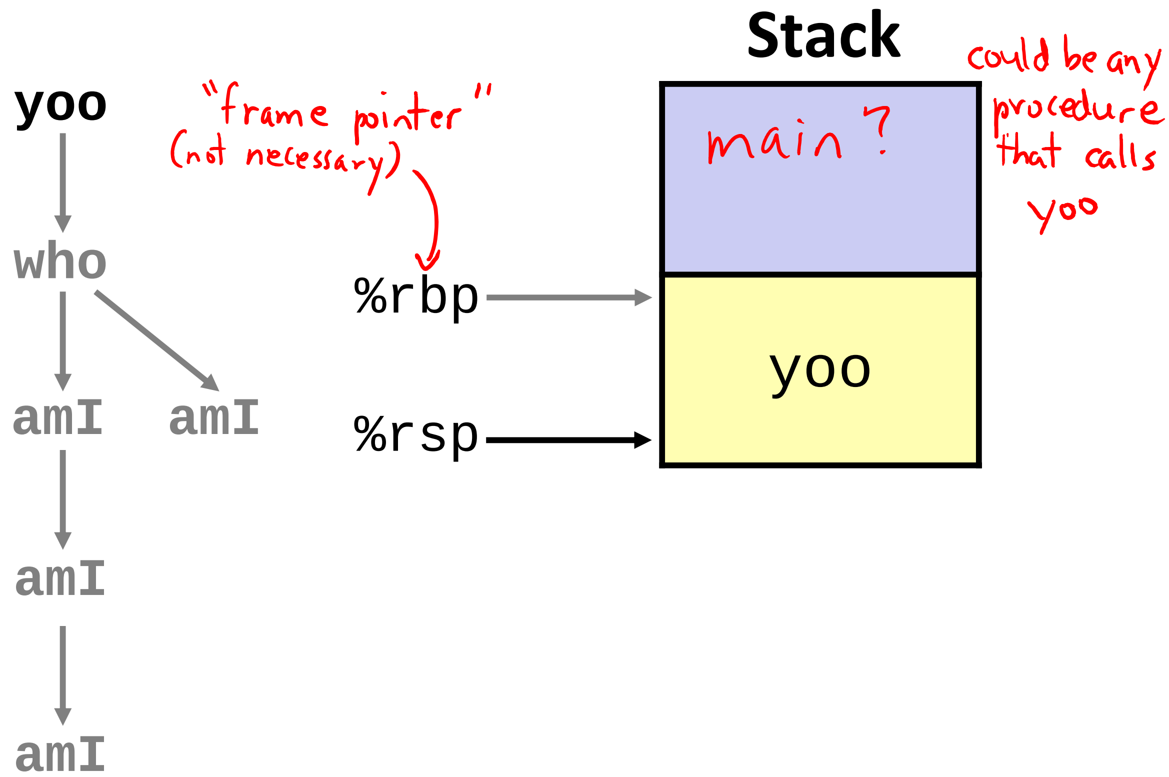
Example Call Chain



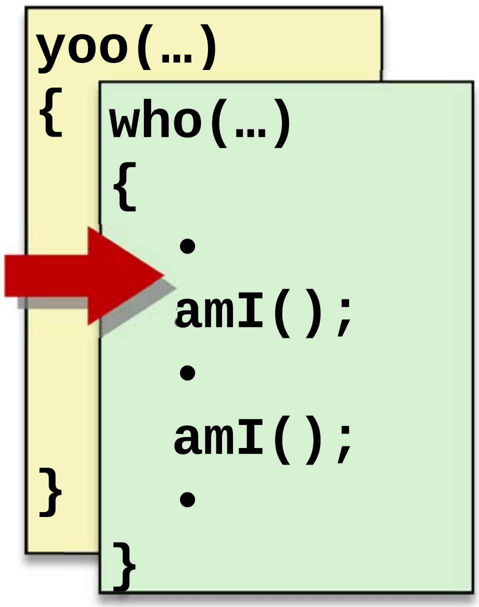
Procedure `amI` is recursive (calls itself)

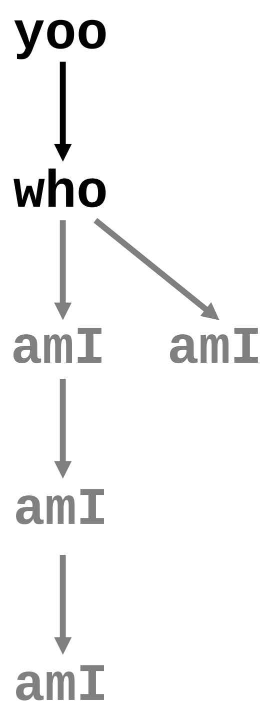
1) Call to yoo

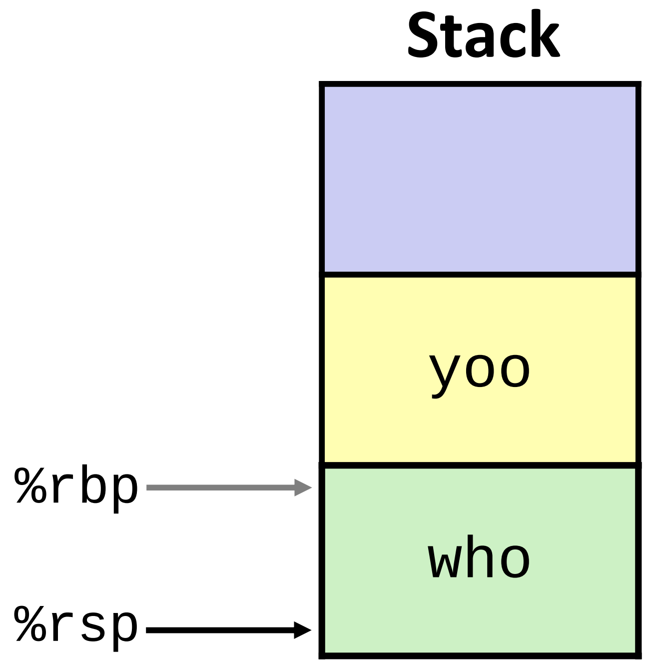
```
yoo(…)
{
   •
   •
  who();
   •
   •
}
```



2) Call to who

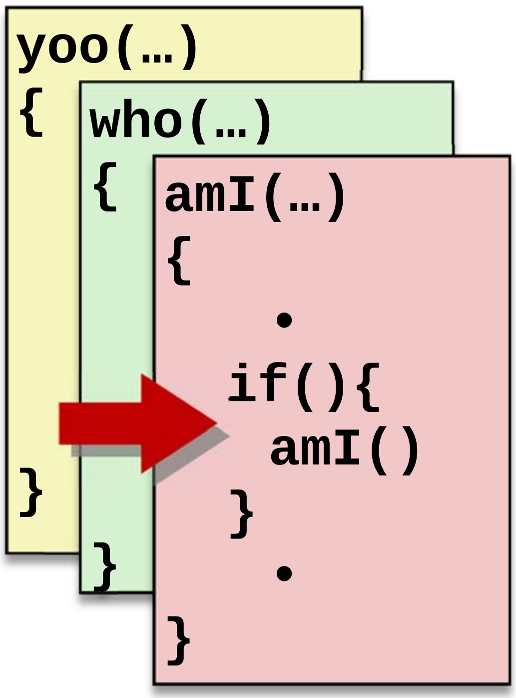


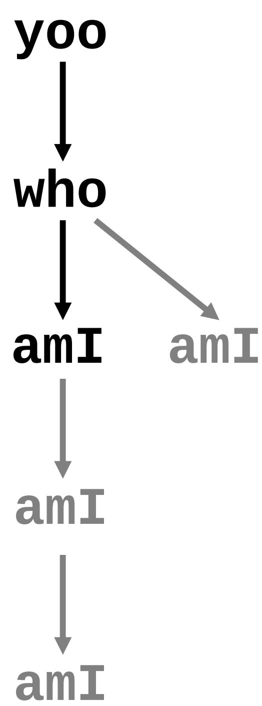


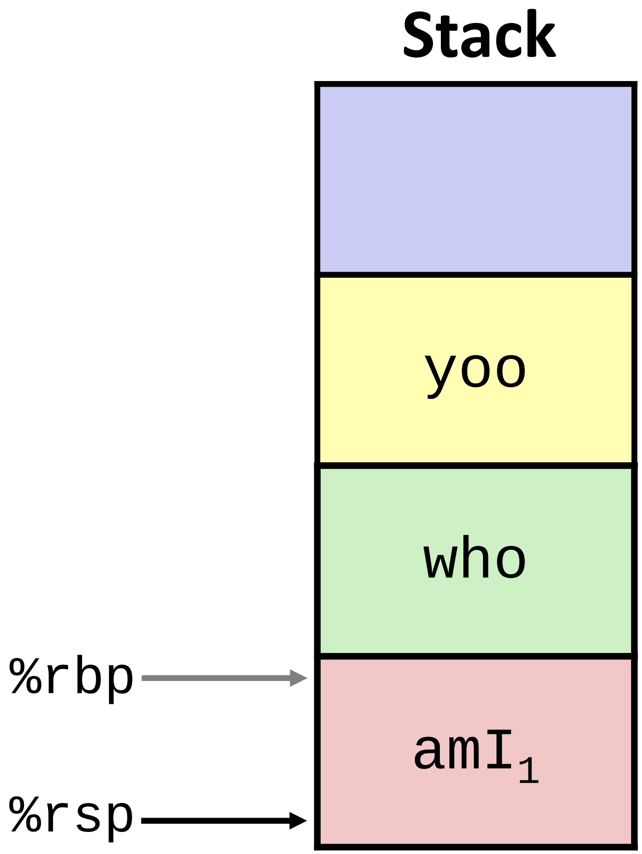


"create" frame
by manipulating %rsp

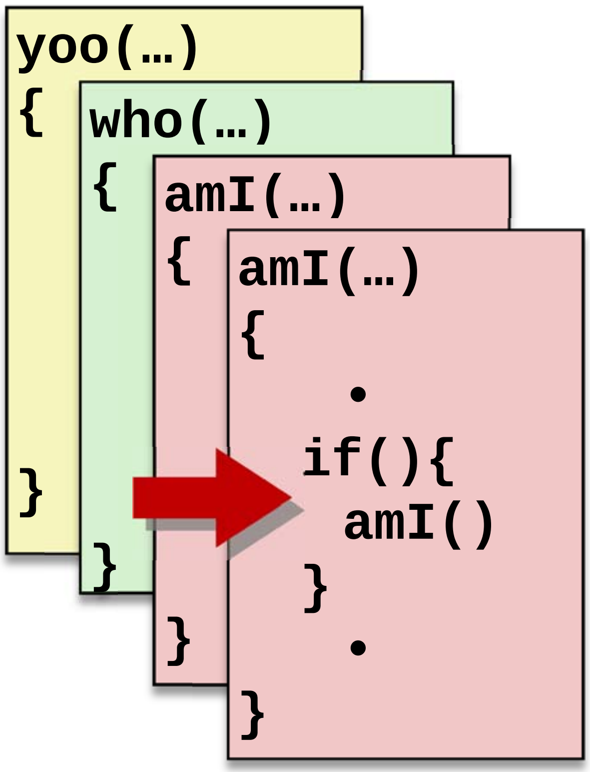
3) Call to amI (1)

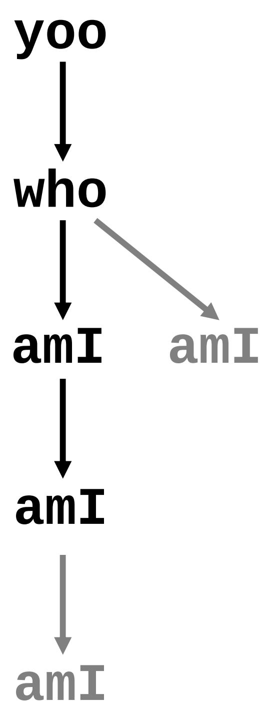


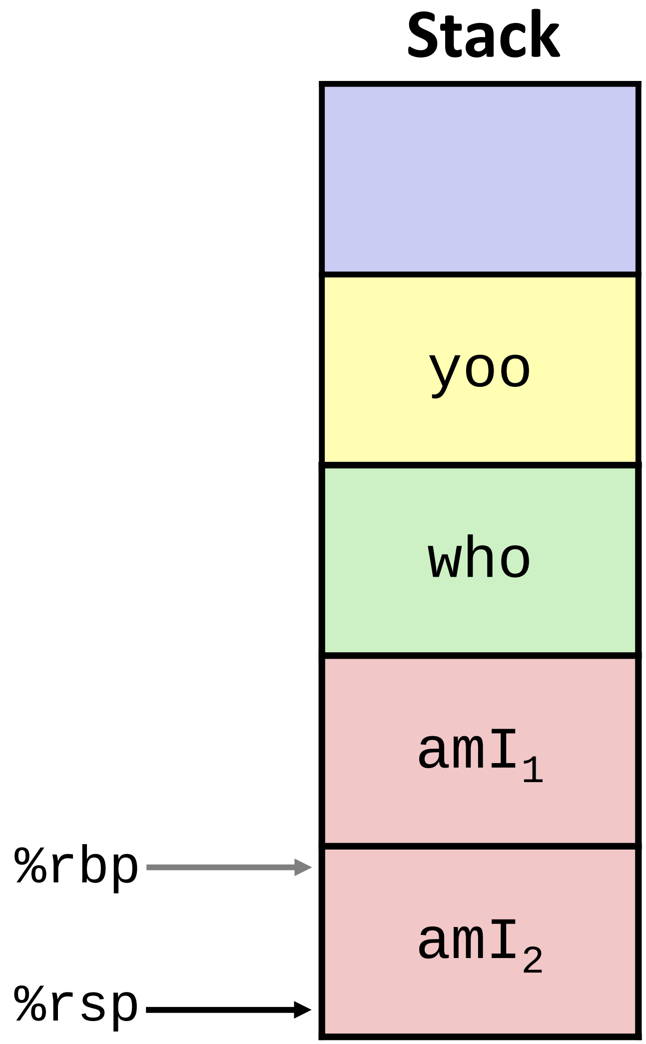




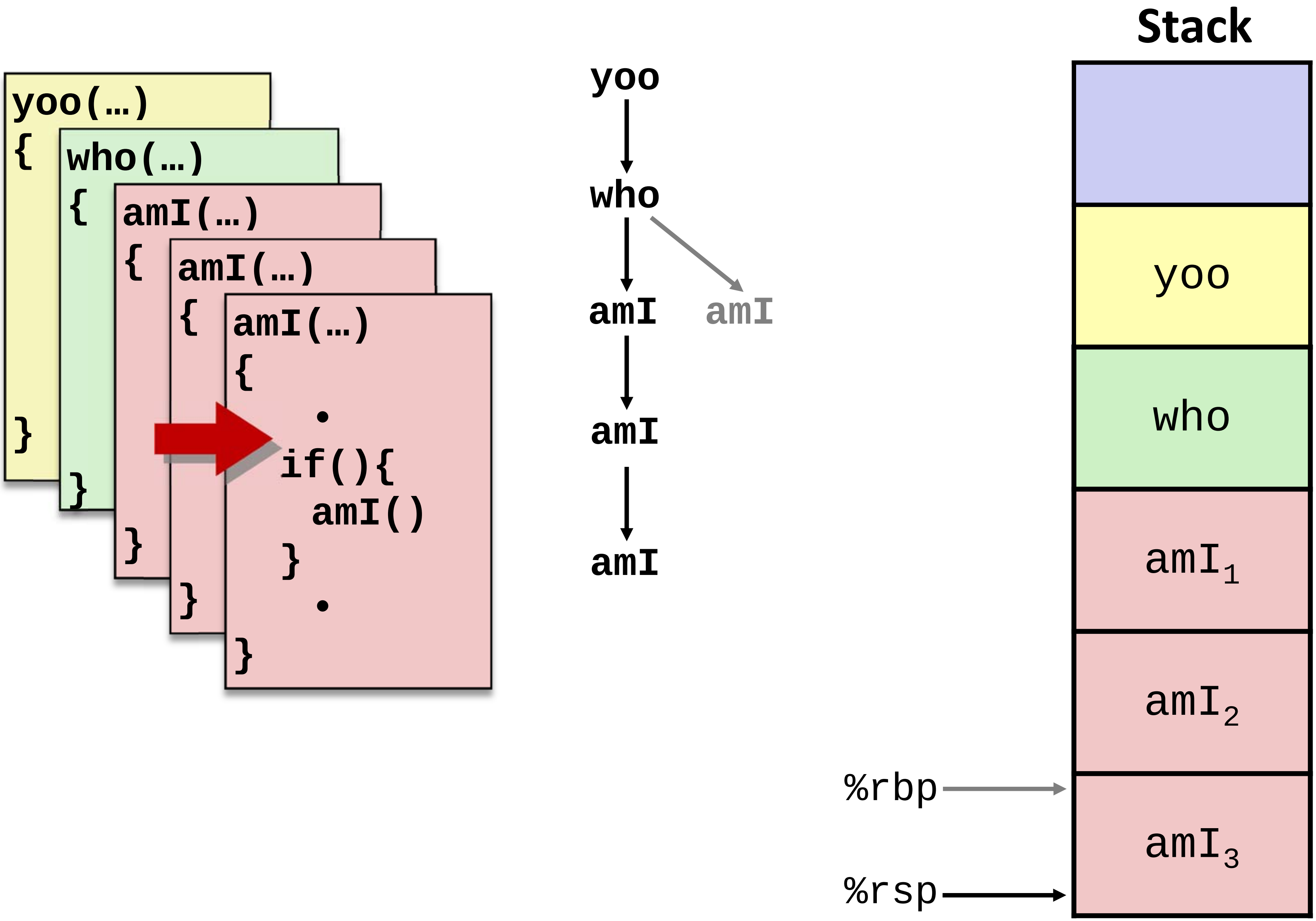
4) Recursive call to `amI` (2)



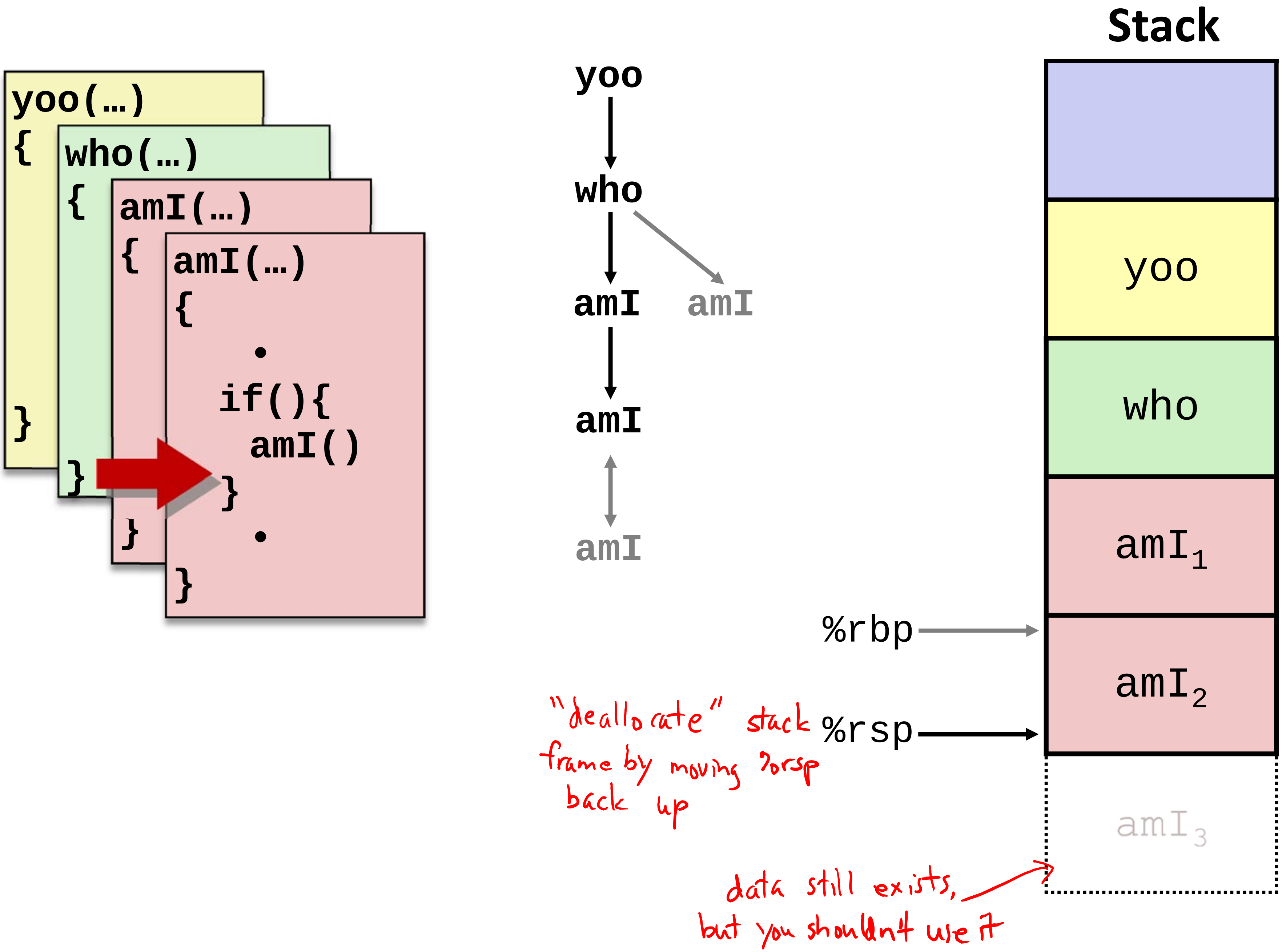




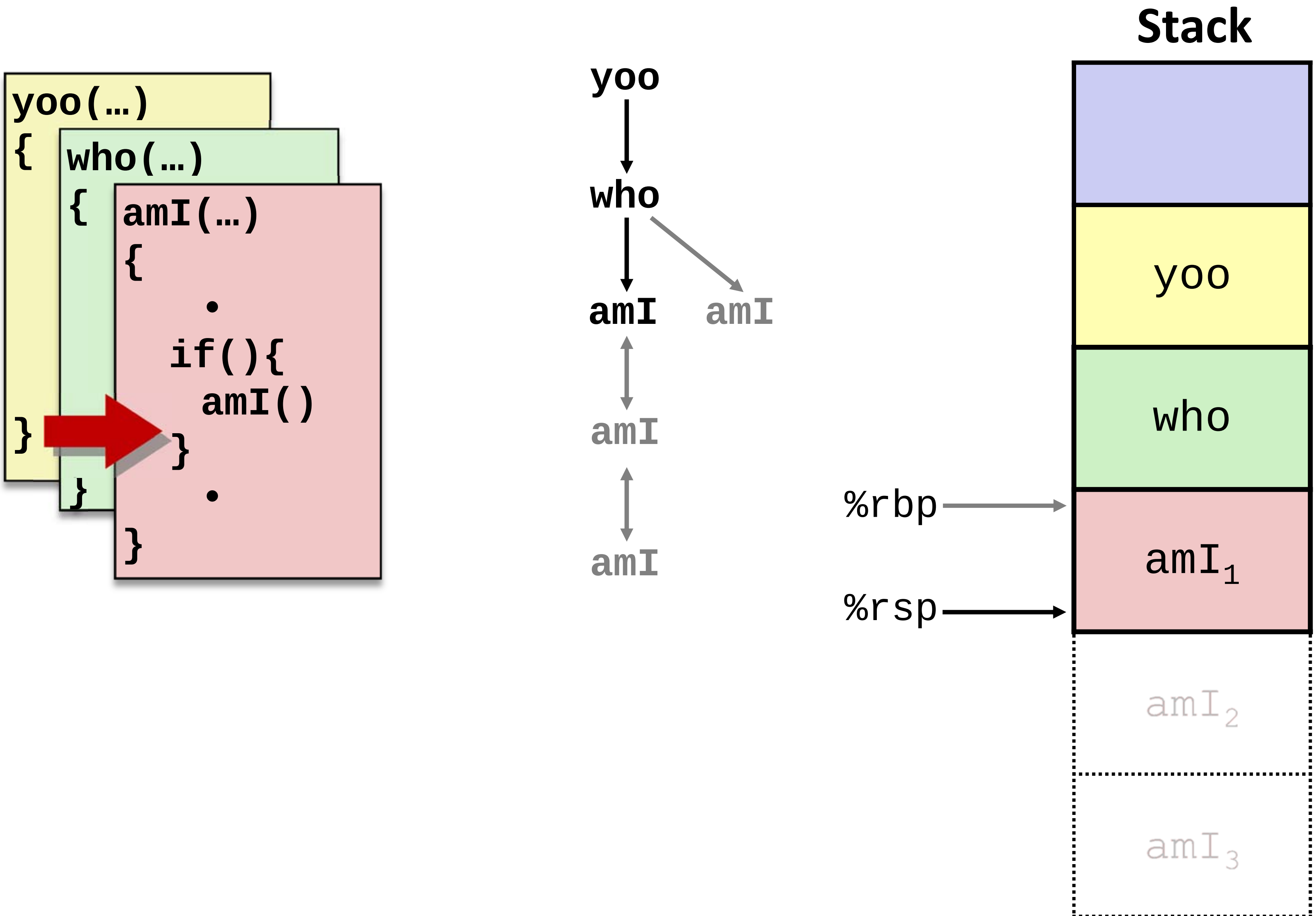
5) (another) Recursive call to amI (3)



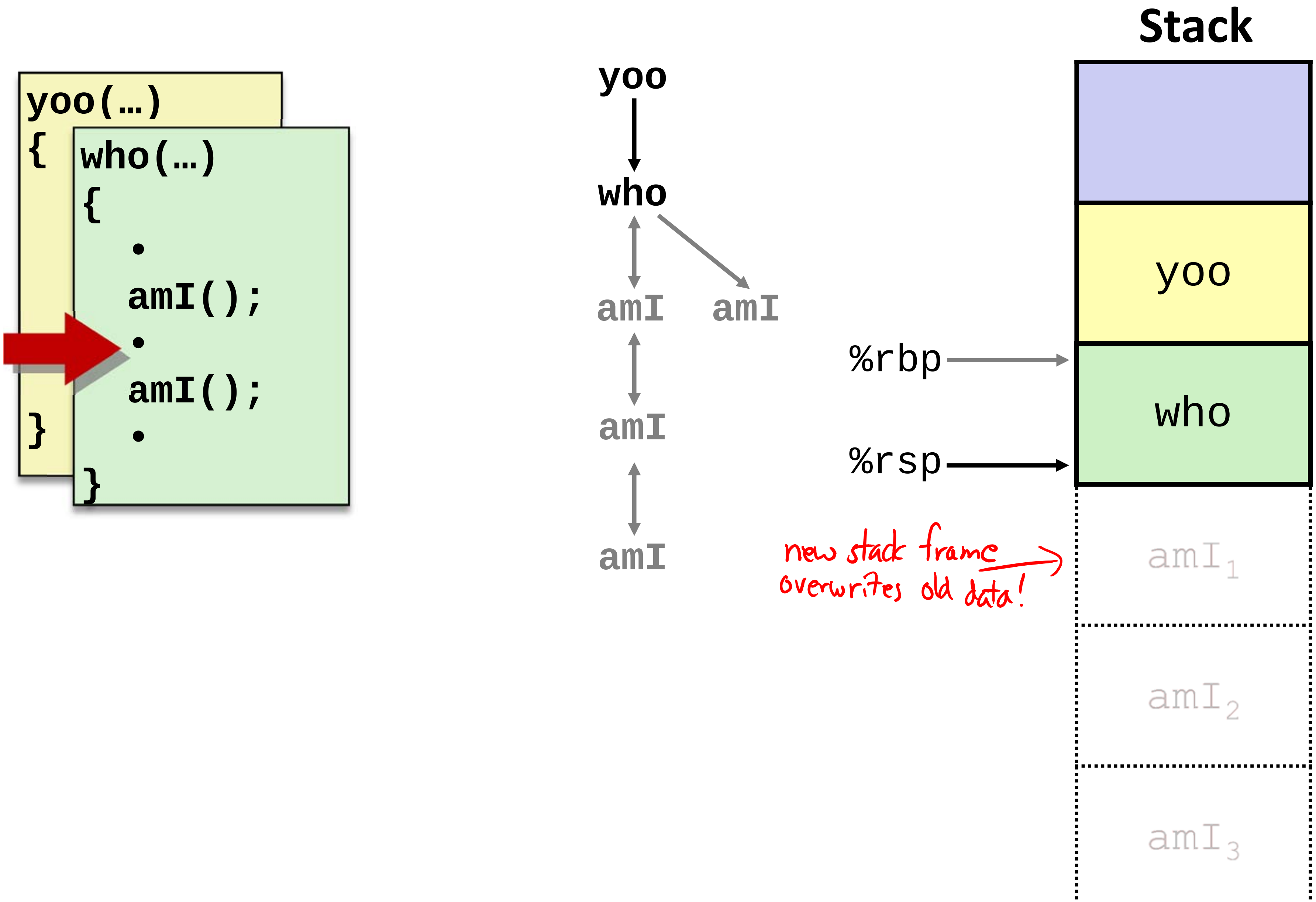
6) Return from (another) recursive call to amI



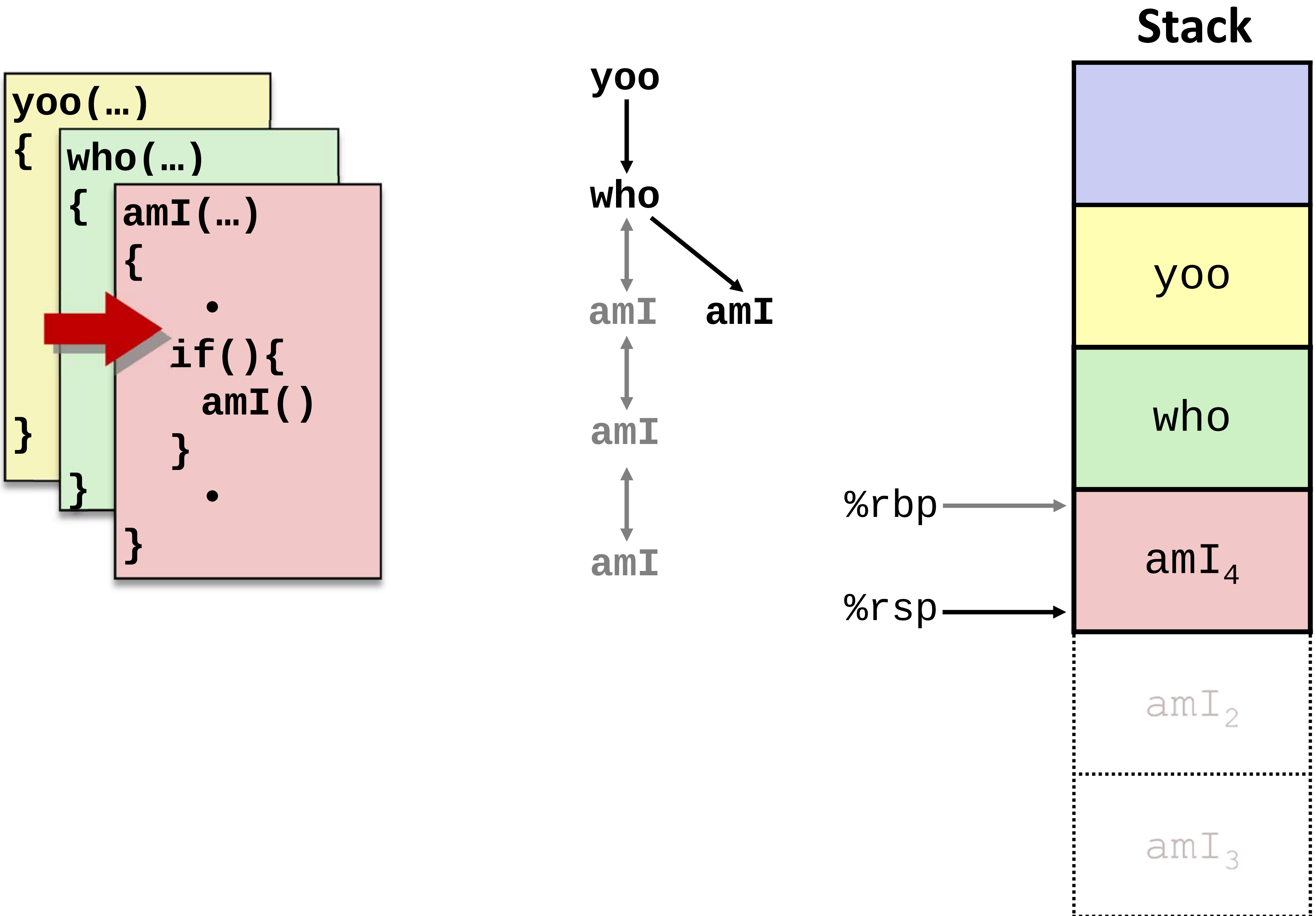
7) Return from recursive call to amI



8) Return from call to amI



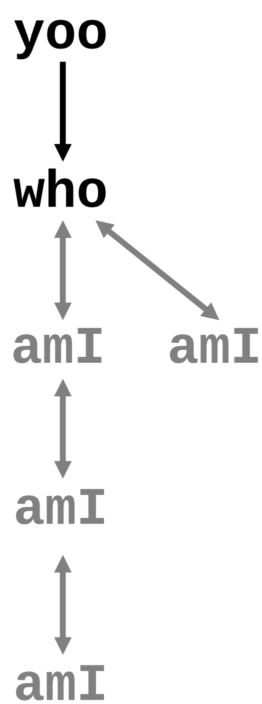
9) (second) Call to amI (4)

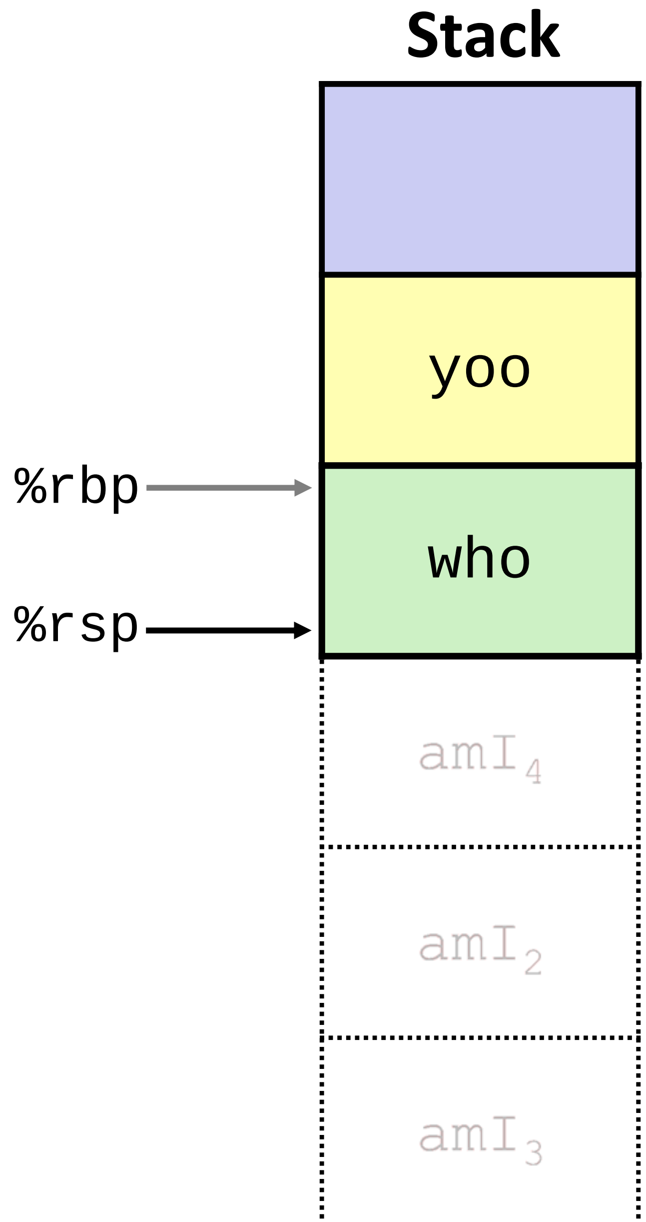


10) Return from (second) call to `amI`

```
yoo(…)
{
```

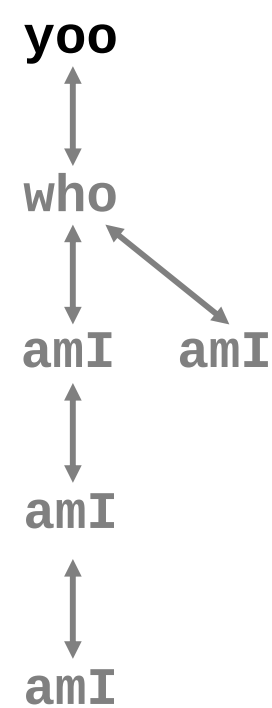
```
who(…)
{
   •
   amI();
   •
   amI();
   •
}
```

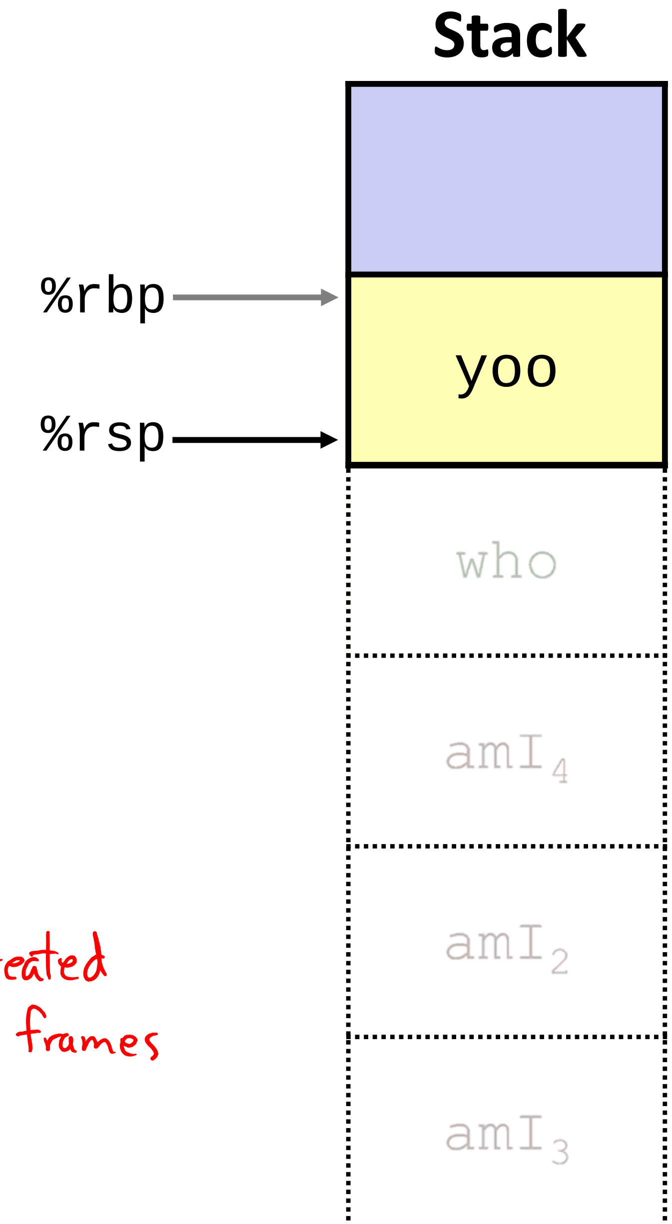




11) Return from call to who

```
yoo(…)
{
  •
  •
  who();
  •
  •
}
```





In total (counting main):

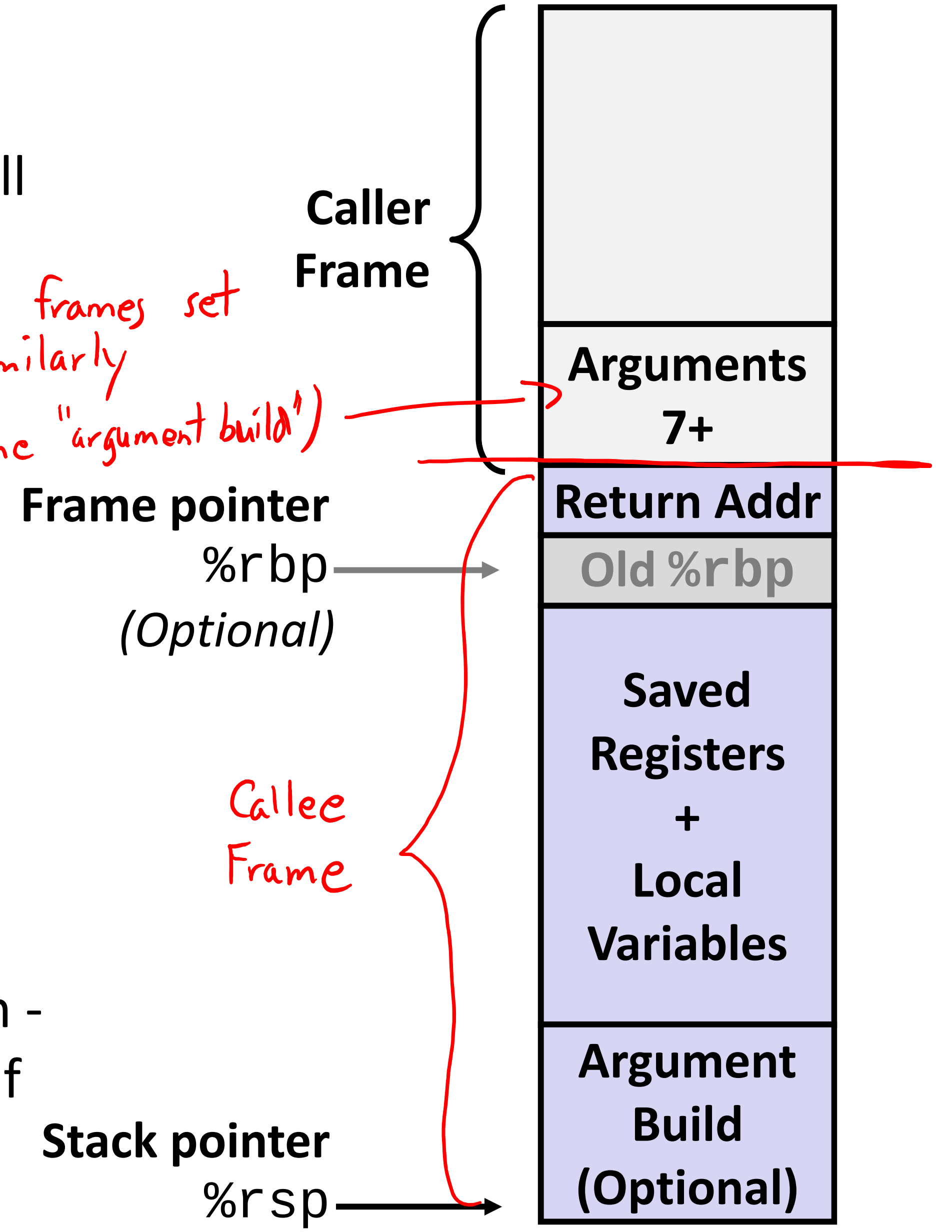
7 stack frames created

max depth of 6 stack frames

x86-64/Linux Stack Frame

- Caller's Stack Frame
  - Extra arguments (if > 6 args) for this call
- Current/Callee Stack Frame
  - Return address
    - Pushed by `call` instruction
  - Old frame pointer (optional)
  - Saved register context
    (when reusing registers)
  - Local variables
    (If can't be kept in registers)
  - "Argument build" area
    (If callee needs to call another function - parameters for function about to call, if needed)

all frames set up similarly
(this is the "argument build")



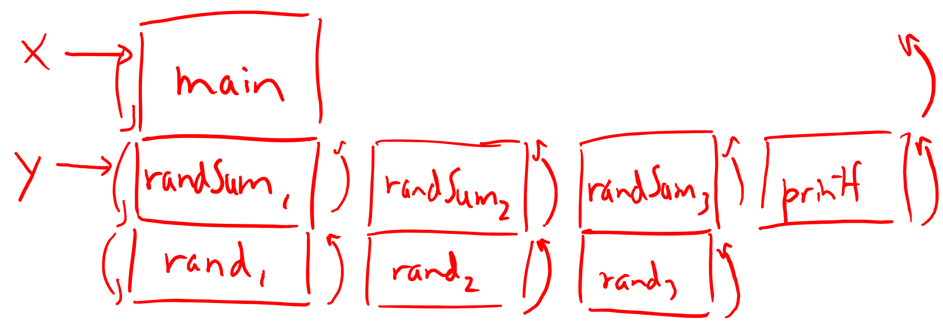
Peer Instruction Question

Vote only on 3rd question at
http://PollEv.com/justinh

- Answer the following questions about when `main()` is run (assume `x` and `y` stored on the Stack):

```c
int main() {
    int i, x = 0;
    for(i=0;i<3;i++)
        x = randSum(x);
    printf("x = %d\n",x);
    return 0;
}
```

```c
int randSum(int n) {
    int y = rand()%20;
    return n+y;
}
```



- *Higher/larger address*: $\boxed{x}$ or y?
- How many total stack frames are *created*? 8
- What is the maximum *depth* (# of frames) of the Stack?

A. 1 B. 2 $\boxed{\text{C. 3}}$ D. 4

Example: `increment`

adding val to value store at p

```
long increment(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

written this way to correspond to assembly

```
increment:
  movq    (%rdi), %rax    # x=*p
  addq    %rax, %rsi      # y=x+val
  movq    %rsi, (%rdi)    # *p=y
  ret
```

Register	Use(s)
%rdi	1st arg (p)
%rsi	2nd arg (val), y
%rax	x, return value

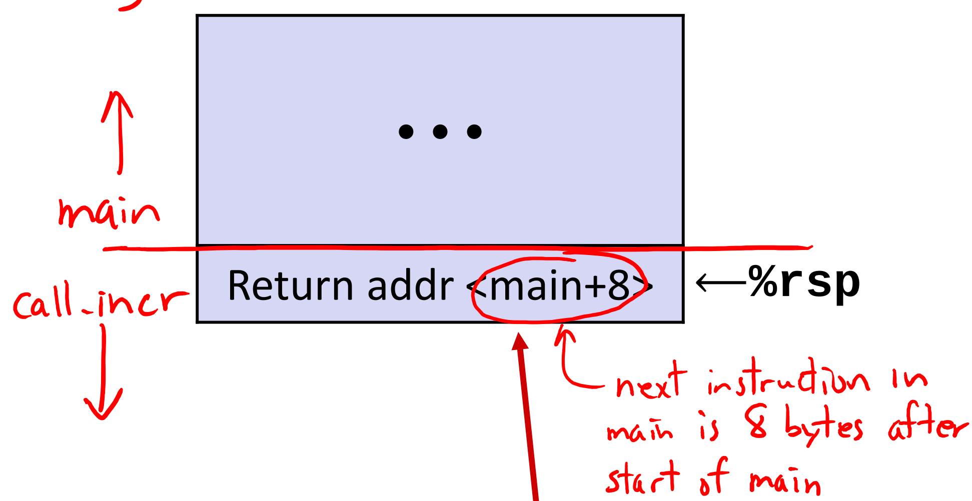
Procedure Call Example (initial state)

```
long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $410, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

frames

Initial Stack Structure



- Return address on stack is the address of instruction immediately *following* the call to "`call_incr`"
  - Shown here as `main`, but could be anything)
  - Pushed onto stack by call call_incr

Procedure Call Example (step 1)

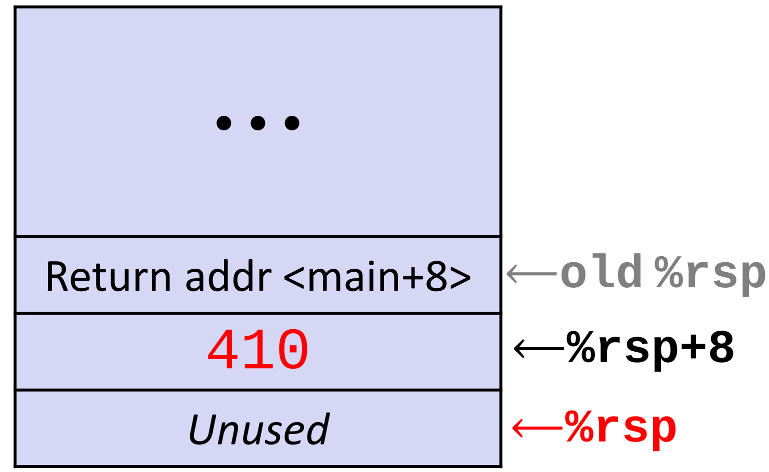
```
long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq     $16, %rsp
  movq     $410, 8(%rsp)
  movl     $100, %esi
  leaq     8(%rsp), %rdi
  call     increment
  addq     8(%rsp), %rax
  addq     $16, %rsp
  ret
```

Allocate space for local vars

manually "push"

Stack Structure



- Setup space for local variables
  - Only v1 needs space on the stack
- Compiler allocated extra space
  - Often does this for a variety of reasons, including alignment

Procedure Call Example (step 2)

```
long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $410, 8(%rsp)
  movl    $100, %esi     # set val
  leaq    8(%rsp), %rdi  # set p
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Set up parameters for call to increment

Stack Structure

• • •	
Return addr <main+8>	
410	← %rsp+8
Unused	← %rsp

Aside: `movl` is used because 100 is a small positive value that fits in 32 bits. High order bits of `rsi` get set to zero automatically. It takes *one less byte* to encode a `movl` than a `movq`.

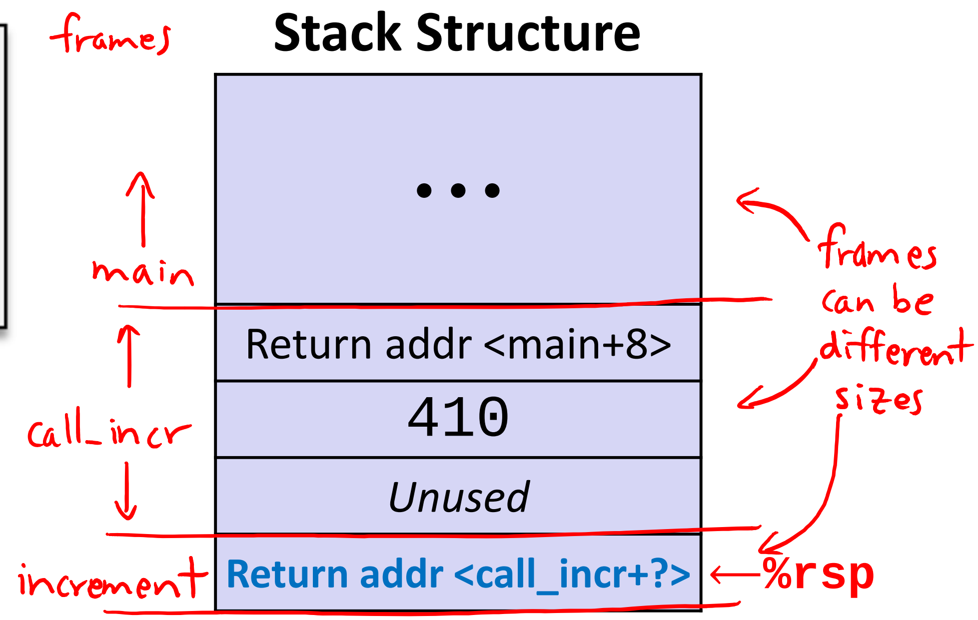
Register	Use(s)
%rdi	&v1
%rsi	100

Procedure Call Example (step 3)

```
long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $410, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

```
increment:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```



- State while inside `increment`
  - **Return address** on top of stack is address of the `addq` instruction immediately following call to `increment`

Register	Use(s)
%rdi	&v1
%rsi	100
%rax	

Procedure Call Example (step 4)

```
long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $410, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

```
increment:
① movq    (%rdi), %rax # x = *p
② addq    %rax, %rsi   # y = x+100
③ movq    %rsi, (%rdi) # *p = y
  ret
```

Stack Structure

• • •
Return addr <main+8>
510
Unused
Return addr <call_incr+?> ← %rsp

①read 410
②add 100
③store 510

popped off stack into %rip by ret instruction

- State while inside `increment`
  - *After* code in body has been executed

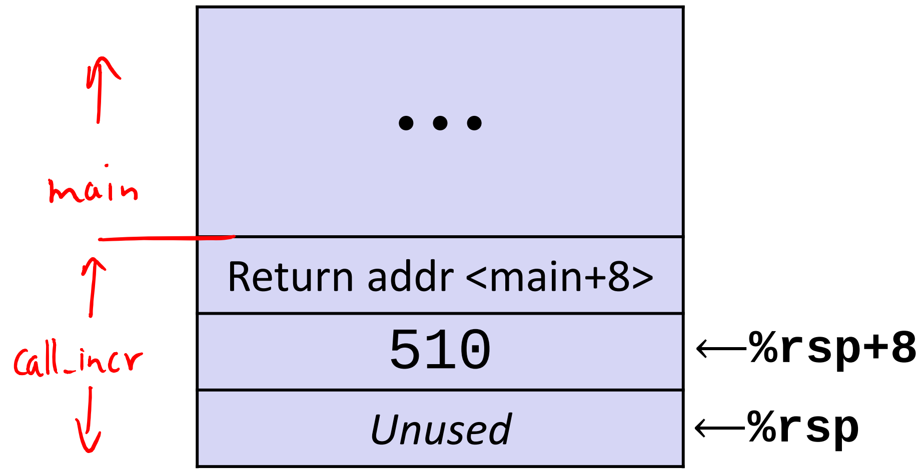
Register	Use(s)
%rdi	&v1
%rsi	510
%rax	410

Procedure Call Example (step 5)

```
long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $410, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



- After returning from call to `increment`
  - Registers and memory have been modified and return address has been popped off stack

Register	Use(s)
%rdi	&v1
%rsi	510
%rax	410

Procedure Call Example (step 6)

```
long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $410, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

← Update **%rax** to contain v1+v2

Stack Structure

• • •	
Return addr <main+8>	
510	←%rsp+8
Unused	←%rsp

Register	Use(s)
%rdi	&v1
%rsi	510
%rax	510+410

Procedure Call Example (step 7)

```
long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $410, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

De-allocate space for local **vars**
(make sure %rsp points to return addr before ret)

Stack Structure

...
Return addr <main+8> ← %rsp
410
Unused ← old %rsp

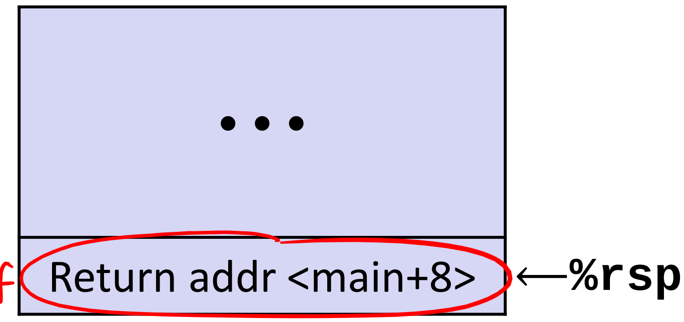
Register	Use(s)
%rdi	&v1
%rsi	510
%rax	920

Procedure Call Example (step 8)

```
long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $410, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



popped off stack into %rip by ret

- State *just before* returning from call to `call_incr`

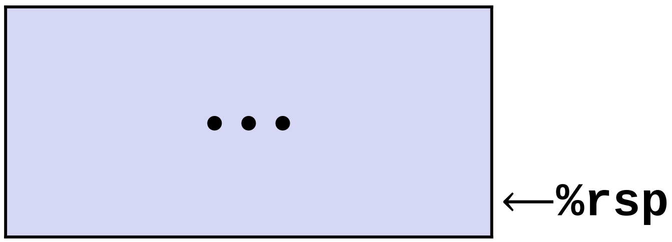
Register	Use(s)
%rdi	&v1
%rsi	510
%rax	920

Procedure Call Example (step 9)

```
long call_incr() {
    long v1 = 410;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $410, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Final Stack Structure



- State immediately *after* returning from call to `call_incr`
  - Return addr has been popped off stack
  - Control has returned to the instruction immediately following the call to `call_incr` (not shown here)

Register	Use(s)
%rdi	&v1
%rsi	510
%rax	920

Lab 2 Demo

- Let’s look at that binary bomb!

objdump -d bomb > bomb_disas //store disassembly of bomb in file
// called bomb_disas

In GDB:

stepi <#> // execute the next <#> asm instr (stepping into function calls)
nexti <#> // execute the next <#> asm instr (stepping over function calls)
print /<format> <expr> // print value of <expr> in <format>
x /<format> <addr> // dereference <addr> and print in <format>

Notes:

Annoyingly, register names in <expr> and <addr> in GDB are preceded by '$'
↳ so $rsp instead of %rsp
Common format characters are 'b' for binary
'x' for hex
's' for string