The Stack & Procedures

CSE 351 Spring 2019

Instructor:

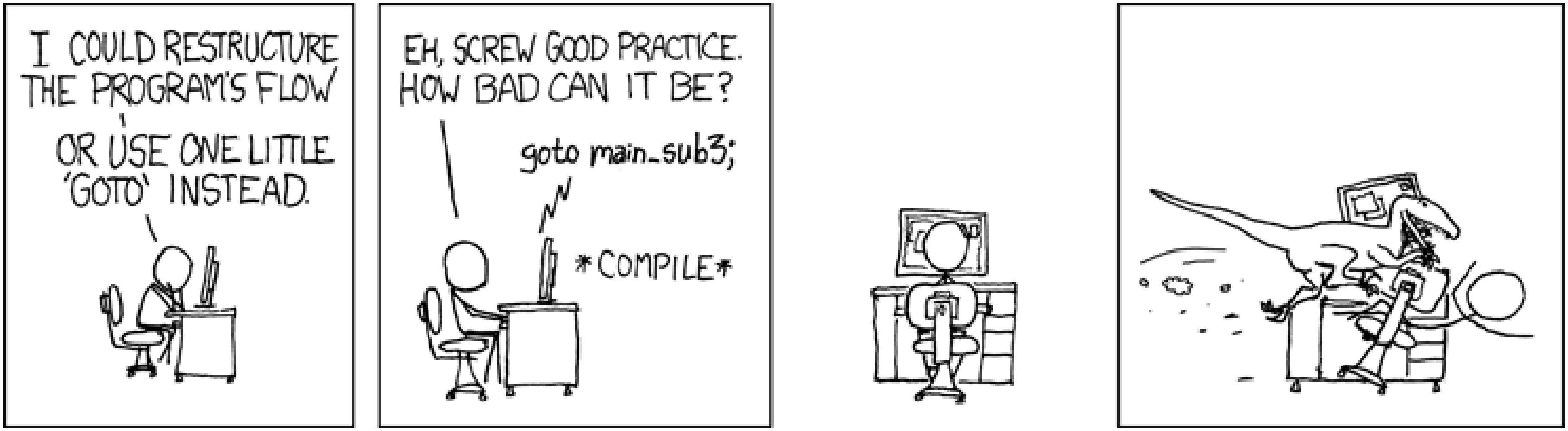
Ruth Anderson

Teaching Assistants:

Gavin Cai
Britt Henderson
Sophie Tian
Casey Xing

Jack Eggleston
Richard Jiang
Connie Wang
Chin Yeoh

John Feltrup
Jack Skalitzky
Sam Wolfson


I COULD RESTRUCTURE THE PROGRAM'S FLOW
OR USE ONE LITTLE 'GOTO' INSTEAD.
EH, SCREW GOOD PRACTICE. HOW BAD CAN IT BE?
goto main_sub3;
COMPILE

http://xkcd.com/571/

Administrivia

- Homework 2 due TONIGHT Wednesday (4/24)
- Lab 2 (x86-64) due Wednesday (5/01)
  - Ideally want to finish well before the midterm
- Homework 3, coming soon
  - On midterm material, but due after the midterm

- Section tomorrow on Assembly and GDB
  - Bring your laptops!

- **Midterm** (Fri 5/03, 4:30-5:30pm in KNE 130)

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

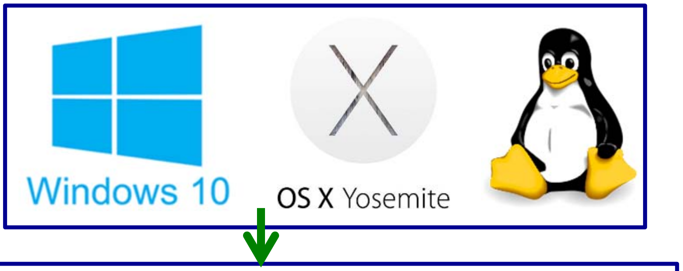
Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

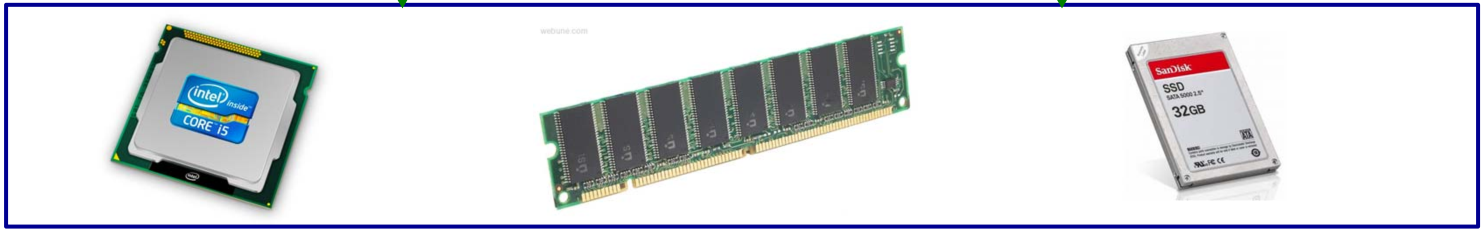
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:



Computer system:

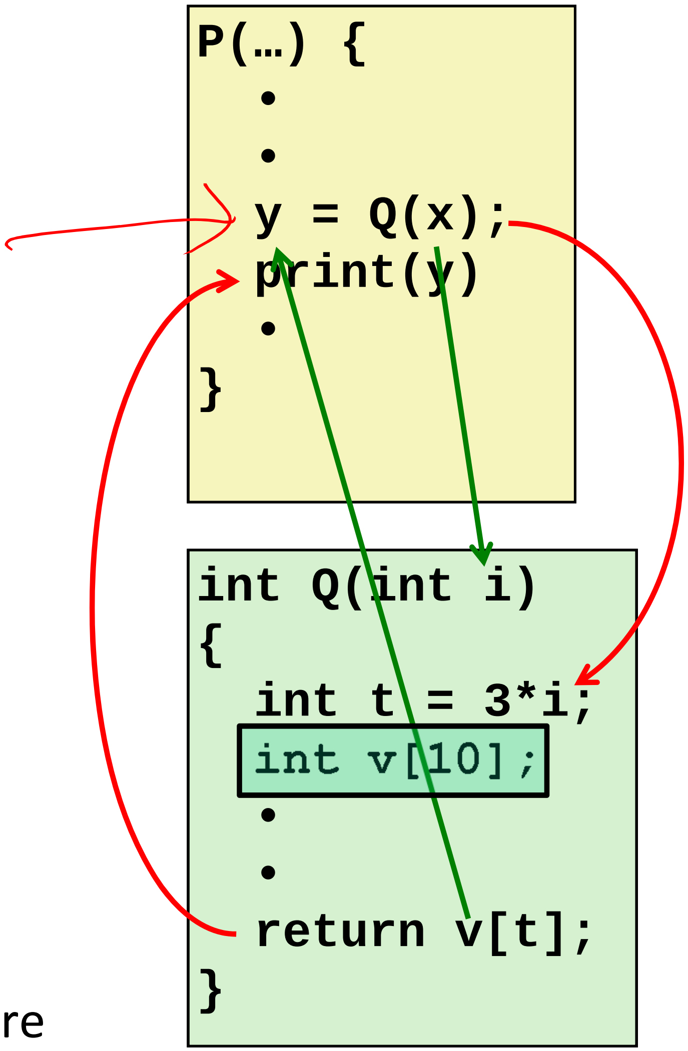


Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

Mechanisms required for *procedures*

1) Passing control
   - To beginning of procedure code
   - Back to return point
2) Passing data
   - Procedure arguments
   - Return value
3) Memory management
   - Allocate during procedure execution
   - Deallocate upon return

- All implemented with machine instructions!
  - An x86-64 procedure uses only those mechanisms required for that procedure



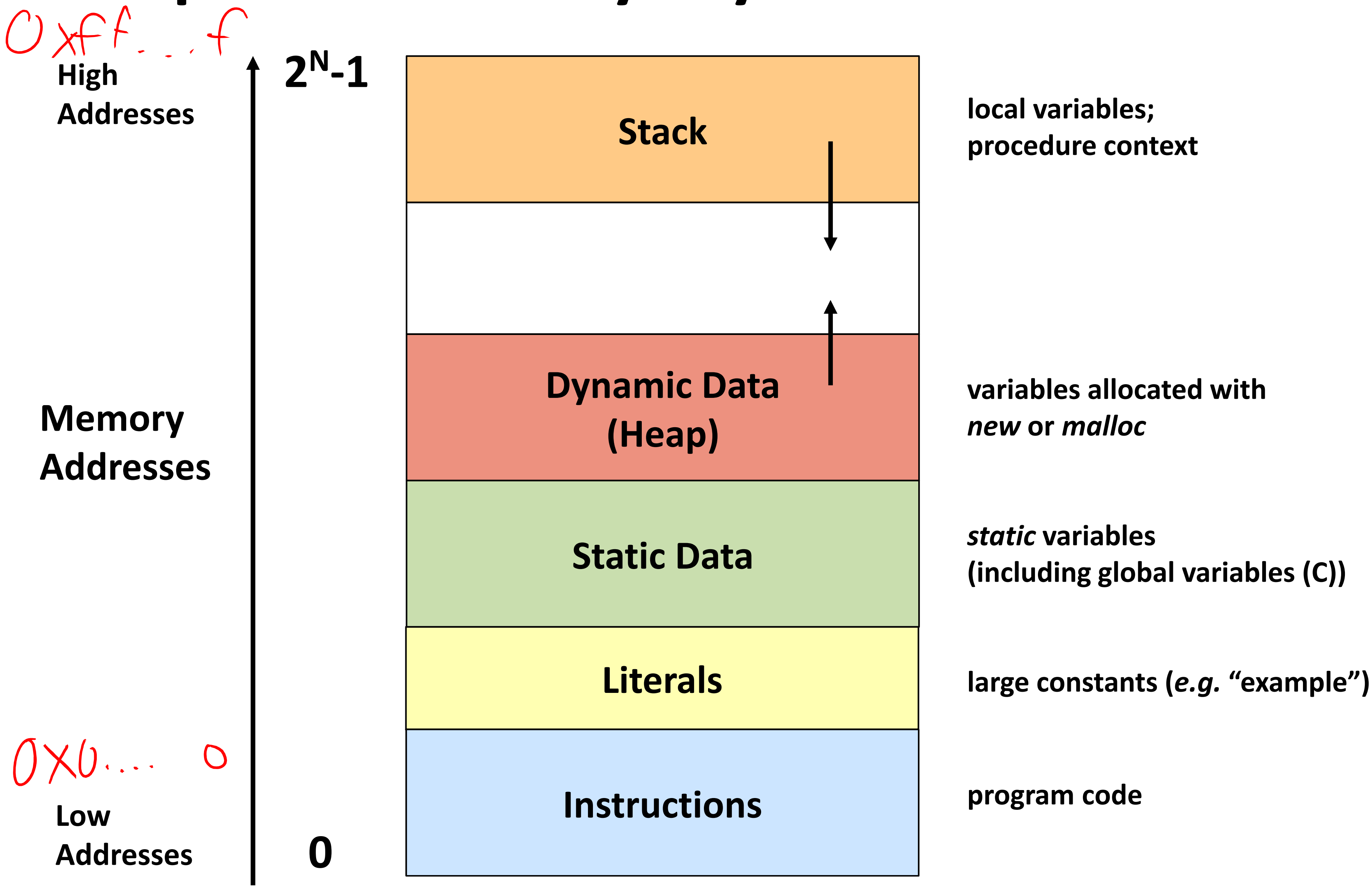
```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

Procedures

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

Simplified Memory Layout



0xff...f

High Addresses

2^N-1

Memory Addresses

0x0.... 0

Low Addresses

0

Stack

local variables;
procedure context

Dynamic Data
(Heap)

variables allocated with
new or *malloc*

Static Data

static variables
(including global variables (C))

Literals

large constants (*e.g.* "example")

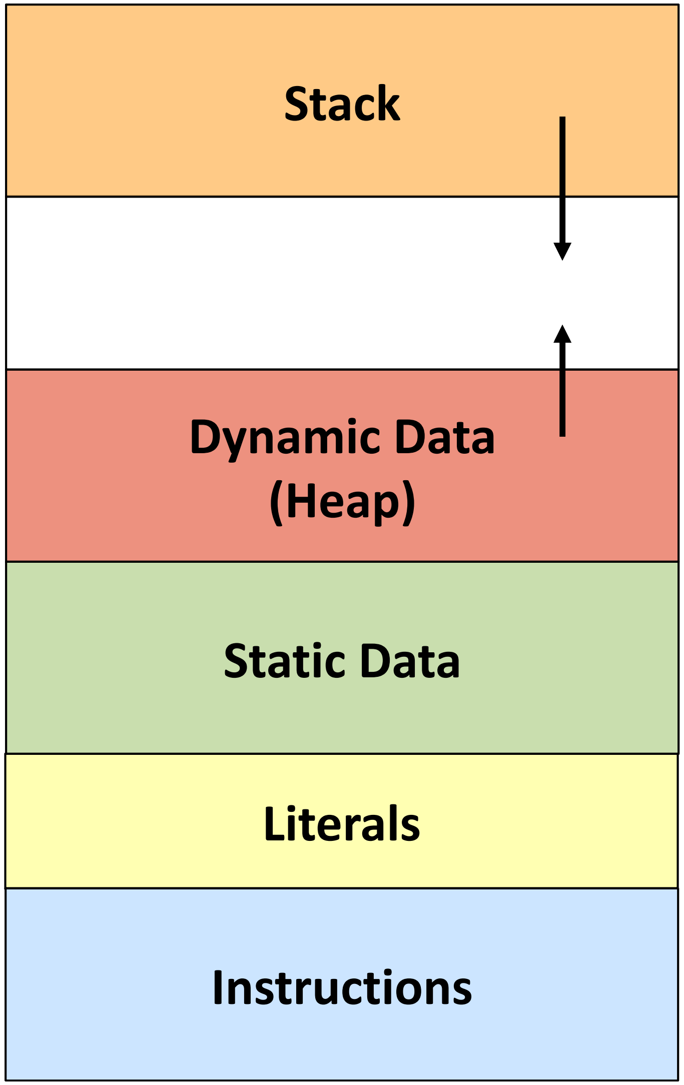
Instructions

program code

Memory Permissions

segmentation faults?

accessing memory in a way that you are not allowed to



writable; not executable — Stack — Managed “automatically” (by compiler)

grow towards each other to maximize use of space

writable; not executable — Dynamic Data (Heap) — Managed by programmer

writable; not executable — Static Data — Initialized when process starts

read-only; not executable — Literals — Initialized when process starts

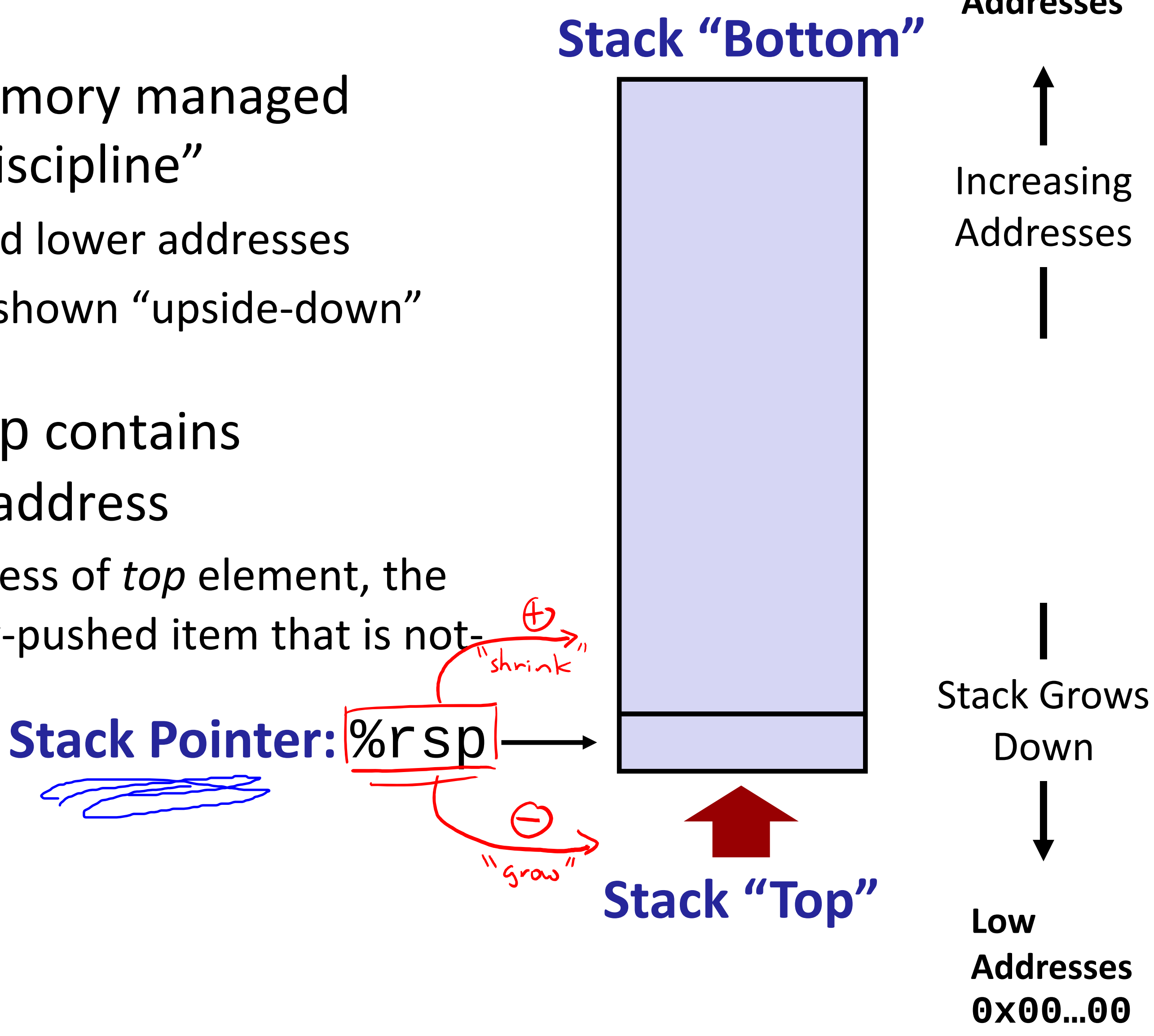
read-only; executable — Instructions — Initialized when process starts

x86-64 Stack

Last In, First Out (LIFO)

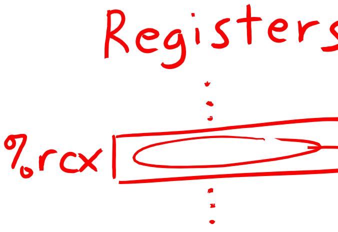
- Region of memory managed with stack “discipline”
  - Grows toward lower addresses
  - Customarily shown “upside-down”

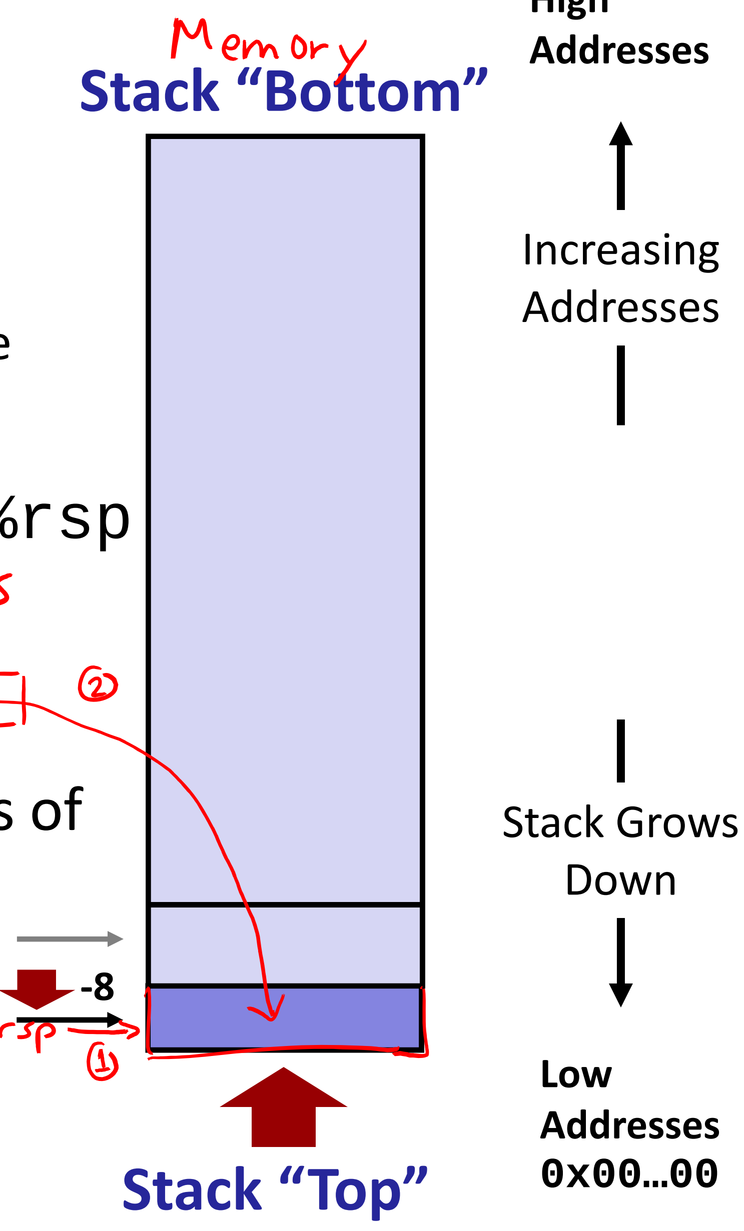
- Register `%rsp` contains *lowest* stack address
  - `%rsp` = address of *top* element, the most-recently-pushed item that is not-yet-popped



x86-64 Stack: Push

- pushq *src*
  size specifier
  - Fetch operand at *src*
    - *Src* can be reg, memory, immediate
  - ***Decrement*** `%rsp` by 8
  - Store value at address given by `%rsp`
- Example:
  - **`pushq %rcx`**
  - Adjust `%rsp` and store contents of `%rcx` on the stack





Stack Pointer: `%rsp`

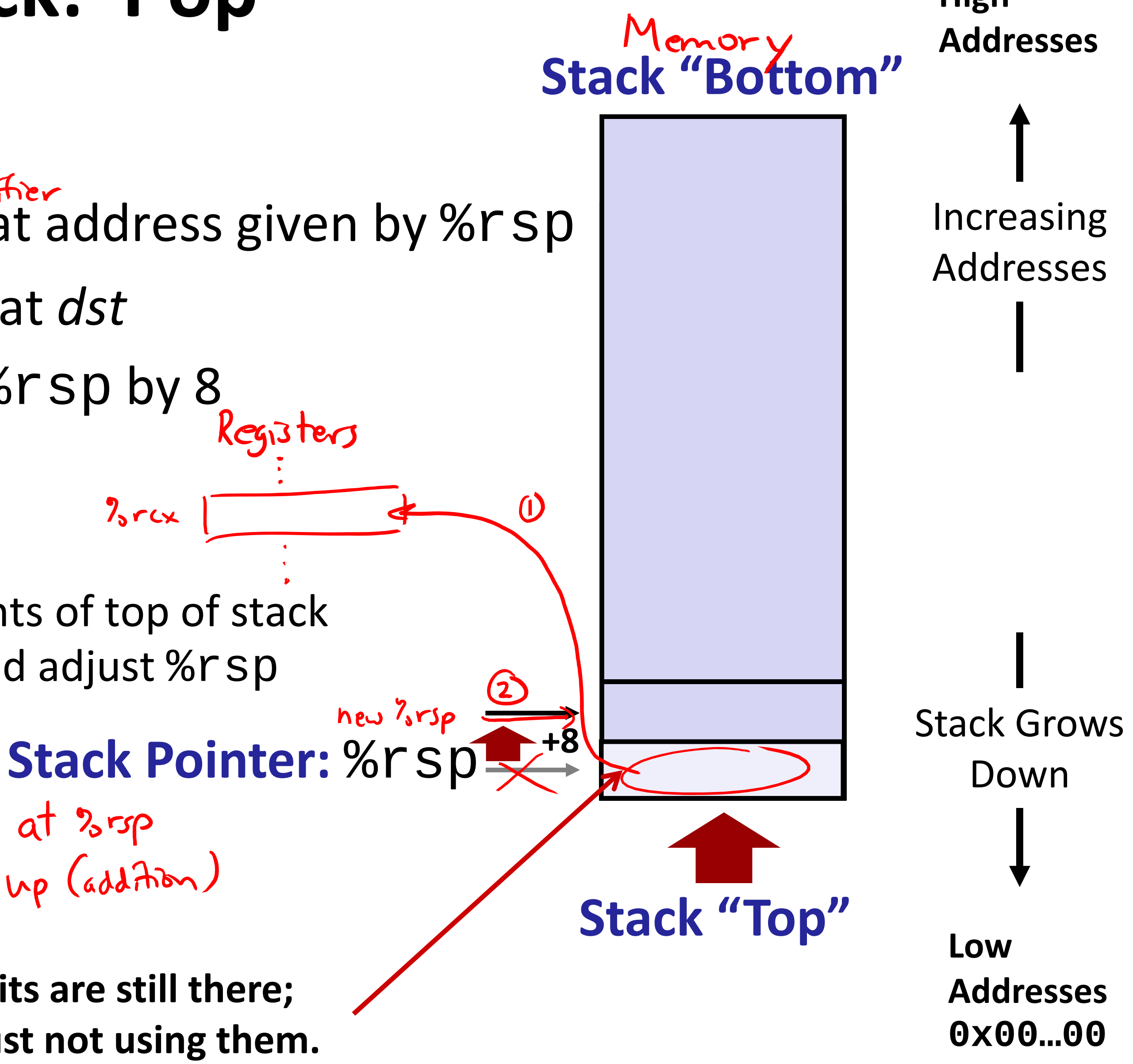
① move %rsp down (subtract)
② store src at %rsp

x86-64 Stack: Pop

- `popq` *dst*

  ↑ size specifier

  - Load value at address given by `%rsp`
  - Store value at *dst*
  - ***Increment*** `%rsp` by 8
- Example:
  - **`popq %rcx`**
  - Stores contents of top of stack into `%rcx` and adjust `%rsp`



① read out data at %rsp

② move %rsp up (addition)

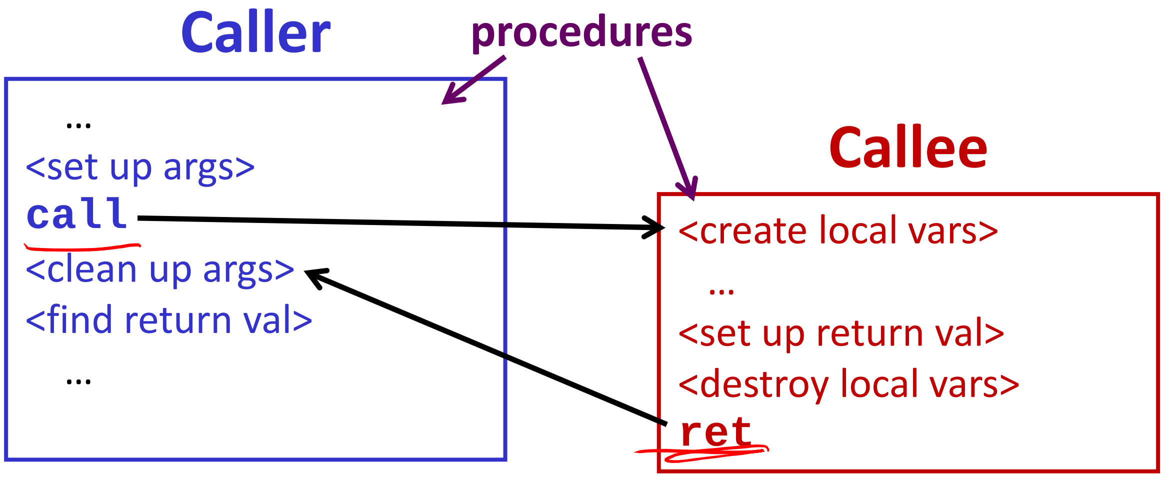
Those bits are still there; we're just not using them.

no point in overwriting them or zeroing them out; we will just overwrite them later when we store another piece of data there!

Procedures

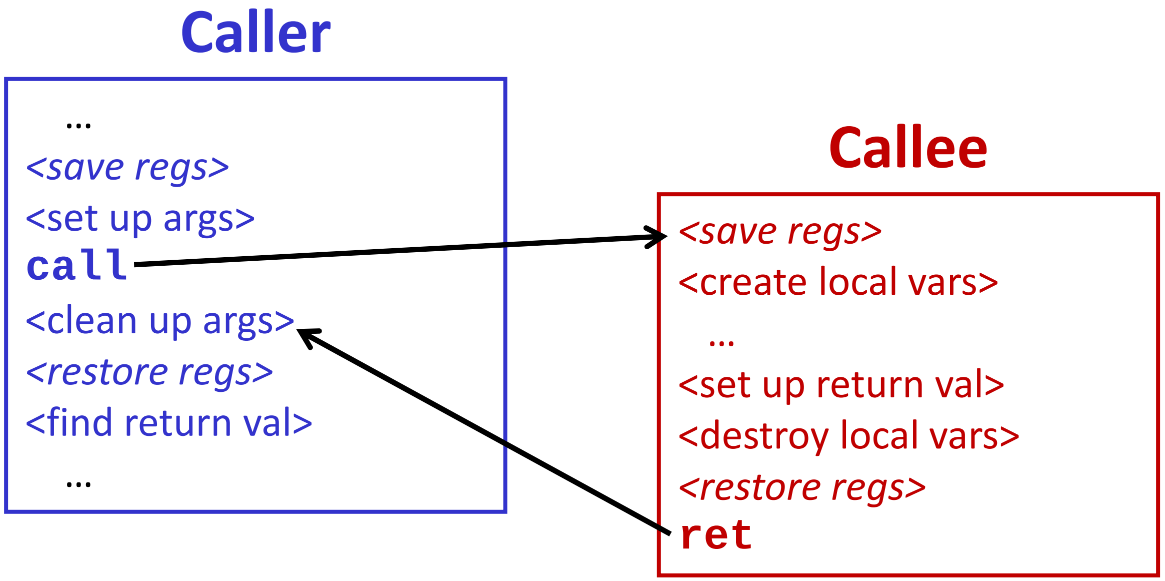
- Stack Structure
- **Calling Conventions**
  - **Passing control**
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion

Procedure Call Overview



- **Callee** must know where to find args
- **Callee** must know where to find *return address*
- **Caller** must know where to find *return value*
- **Caller** and **Callee** run on same CPU, so use the same registers
  - How do we deal with register reuse?
- Unneeded steps can be skipped (*e.g.* no arguments)

Procedure Call Overview



- The *convention* of where to leave/find things is called the calling convention (or procedure call linkage)
  - Details vary between systems
  - We will see the convention for x86-64/Linux in detail
  - What could happen if our program didn't follow these conventions?

If our program didn't follow these conventions, we might have a case where a callee overwrites some of the variables in the registers that the caller was going to use after the return statement.

Code Example (Preview)

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

Caller

Compiler Explorer:
https://godbolt.org/g/cKKDZn

by moving to a register reserved for caller (see textbook page 223)

by pushing it onto the top of the stack

executable disassembly

```
0000000000400540 <multstore>:
  400540: push   %rbx              # Save %rbx
  400541: movq   %rdx,%rbx         # Save dest
  400544: call   400550 <mult2> # mult2(x,y)
  400549: movq   %rax,(%rbx)       # Save at dest
  40054c: pop    %rbx              # Restore %rbx
  40054d: ret                      # Return
```

these are instruction addresses

address where instructions for <mult2> are stored

Callee

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

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

Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: `call label` (special push)
  1) Push return address on stack (*why? which address?*)
  2) Jump to ***`label`***

① move %rsp down
② store ret addr at %rsp (on the stack)
③ label → %rip

Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: `call label` (special push)
  1) Push return address on stack (*why? which address?*) → ① move %rsp down ② store ret addr at %rsp ③ label → %rip
  2) Jump to ***label***
- Return address:
  - Address of instruction immediately after **`call`** instruction
  - Example from disassembly:

```
400544: call   400550 <mult2>
400549: movq   %rax,(%rbx)
```

Return address = **0x400549**

next instruction happens to be a move, but could be anything

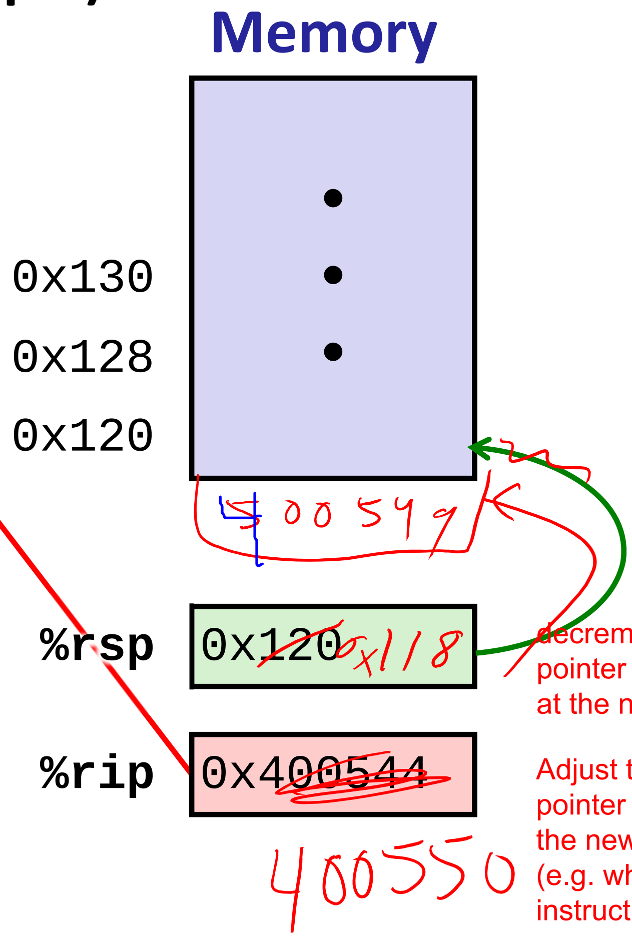
- Procedure return: `ret` (special pop)
  1) Pop return address from stack ① read ret addr at %rsp into %rip
  2) Jump to address ② move %rsp up

Procedure Call Example (step 1)

```
0000000000400540 <multstore>:
  •
  •
  400544: call   400550 <mult2>
  400549: movq   %rax,(%rbx)
  •
  •
```

return value; e.g. where the instructions are supposed to pick up again after call finishes

```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  ret
```



return value is pushed onto the stack

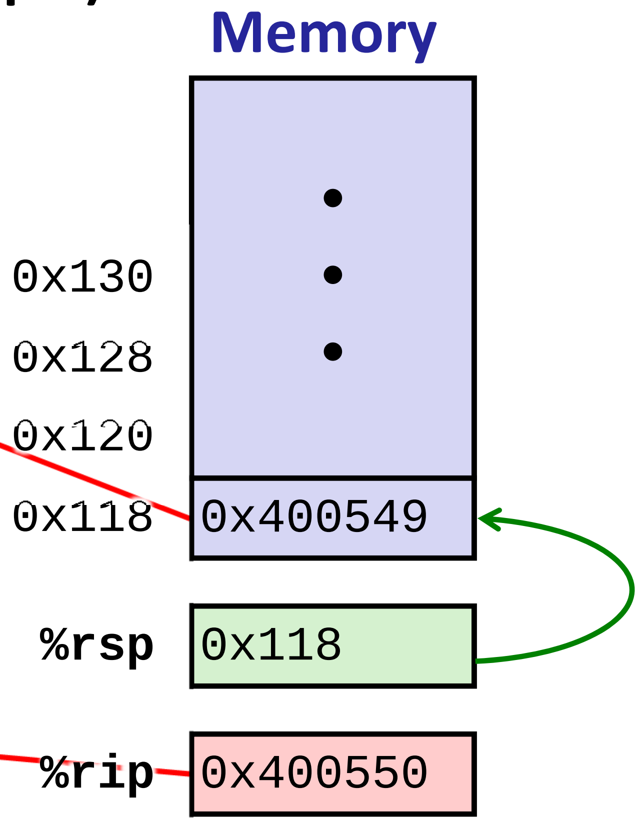
decrement the stack pointer to now point at the new top of stack

Adjust the instruction pointer rip to point to the new set of instructions (e.g. where the mult2 instructions start)

Procedure Call Example (step 2)

```
0000000000400540 <multstore>:
  •
  •
  400544: call   400550 <mult2>
  400549: movq   %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  ret
```

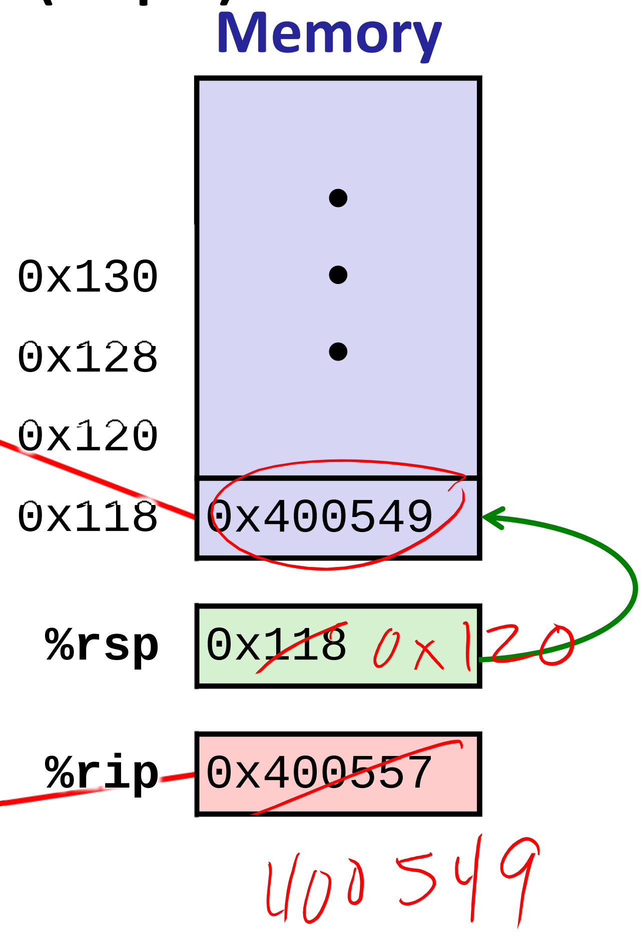


Now carry out the instructions is <mult2>

Procedure Return Example (step 1)

```
0000000000400540 <multstore>:
  •
  •
  400544: call   400550 <mult2>
  400549: movq   %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  ret
```

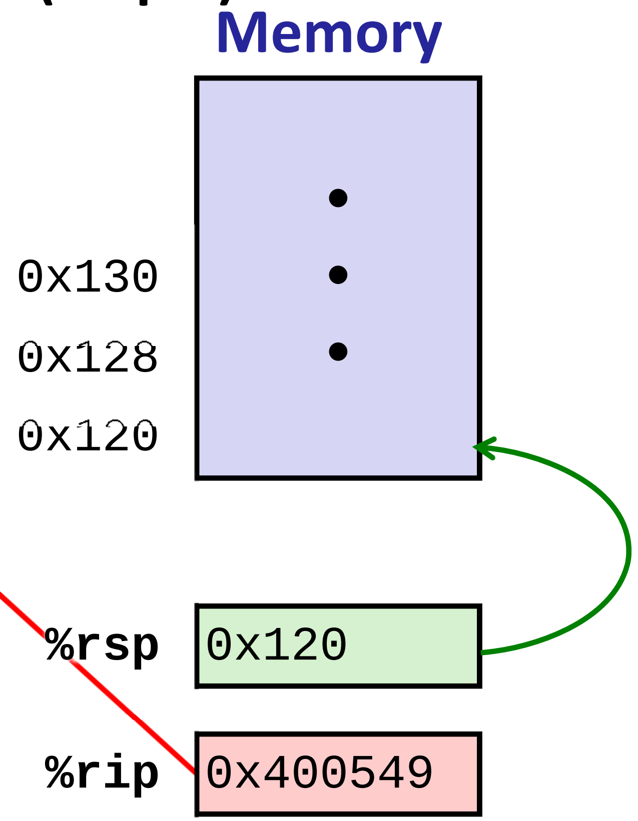


Once we complete all the instructions in <mult2>
(e.g once we reach the ret statement) reset the instruction pointer
%rip to the next command following the call command (e.g.
instruction 400549 above). <mult2> can find this value as it was
pushed onto the stack by call.

Procedure Return Example (step 2)

```
0000000000400540 <multstore>:
  •
  •
  400544: call   400550 <mult2>
  400549: movq   %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  ret
```



Procedures

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

So we have seen how we can use call/ret to move the instruction pointer around. This is how we pass control to different functions/instructions stored in memory. Now, how do we pass data (e.g. the arguments of a function)?

Procedure Data Flow

Registers (NOT in Memory)

- First 6 arguments

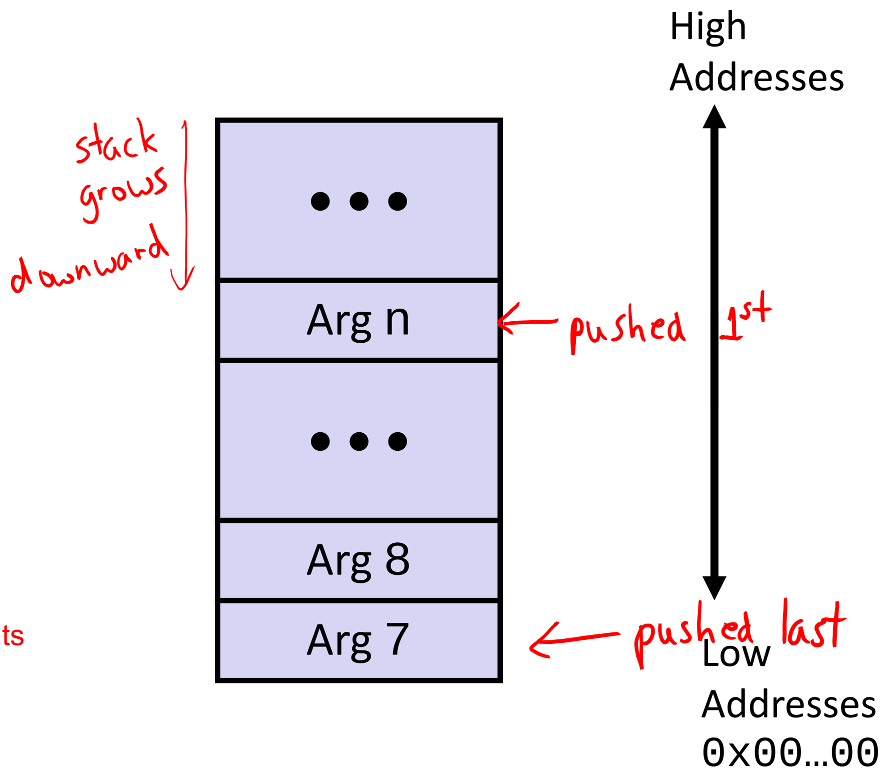
①	%rdi	Diane's
②	%rsi	Silk
③	%rdx	Dress
④	%rcx	Costs
⑤	%r8	$8
⑥	%r9	9

e.g. if we can, store the arguments in the registers.

- 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

save to the stack

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

because the calle is going to overwrite that stuff to store the return value there

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

Caller

```
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)

Callee

```
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 address
- 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(…)
{
    •
  ①amI();
    •
  ②amI();
    •
}
```

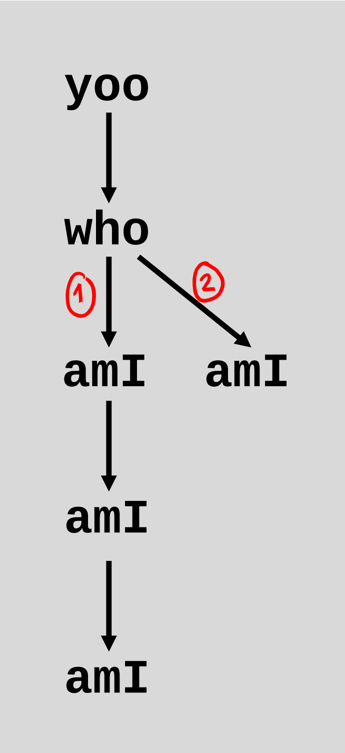
1st call recurses twice

2nd call doesn't recurse

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

based on condition

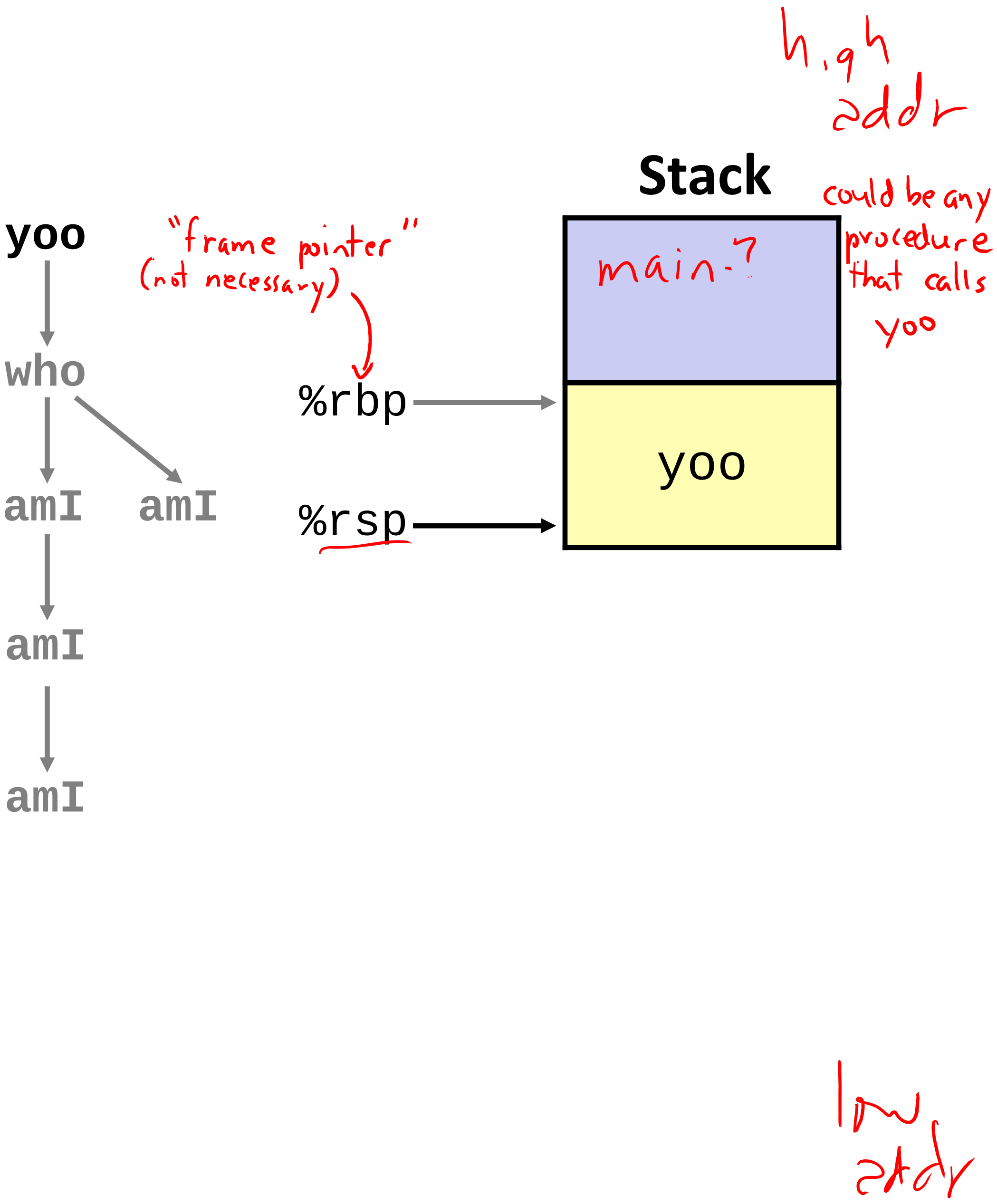
Example Call Chain



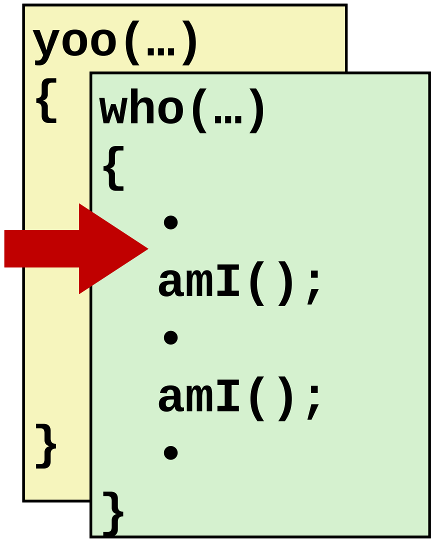
Procedure amI is recursive (calls itself)

1) Call to yoo

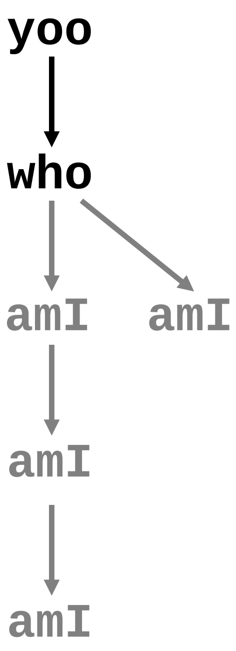
```
yoo(…)
{
  •
  •
  who();
  •
  •
}
```

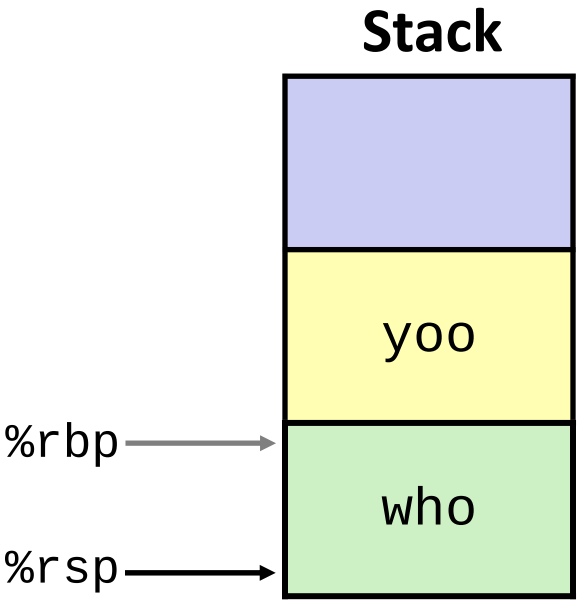

high addr
Stack
could be any procedure that calls yoo
yoo
"frame pointer" (not necessary)
who
amI
amI
amI
amI
%rbp
%rsp
main?
yoo
low addr

2) Call to who



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

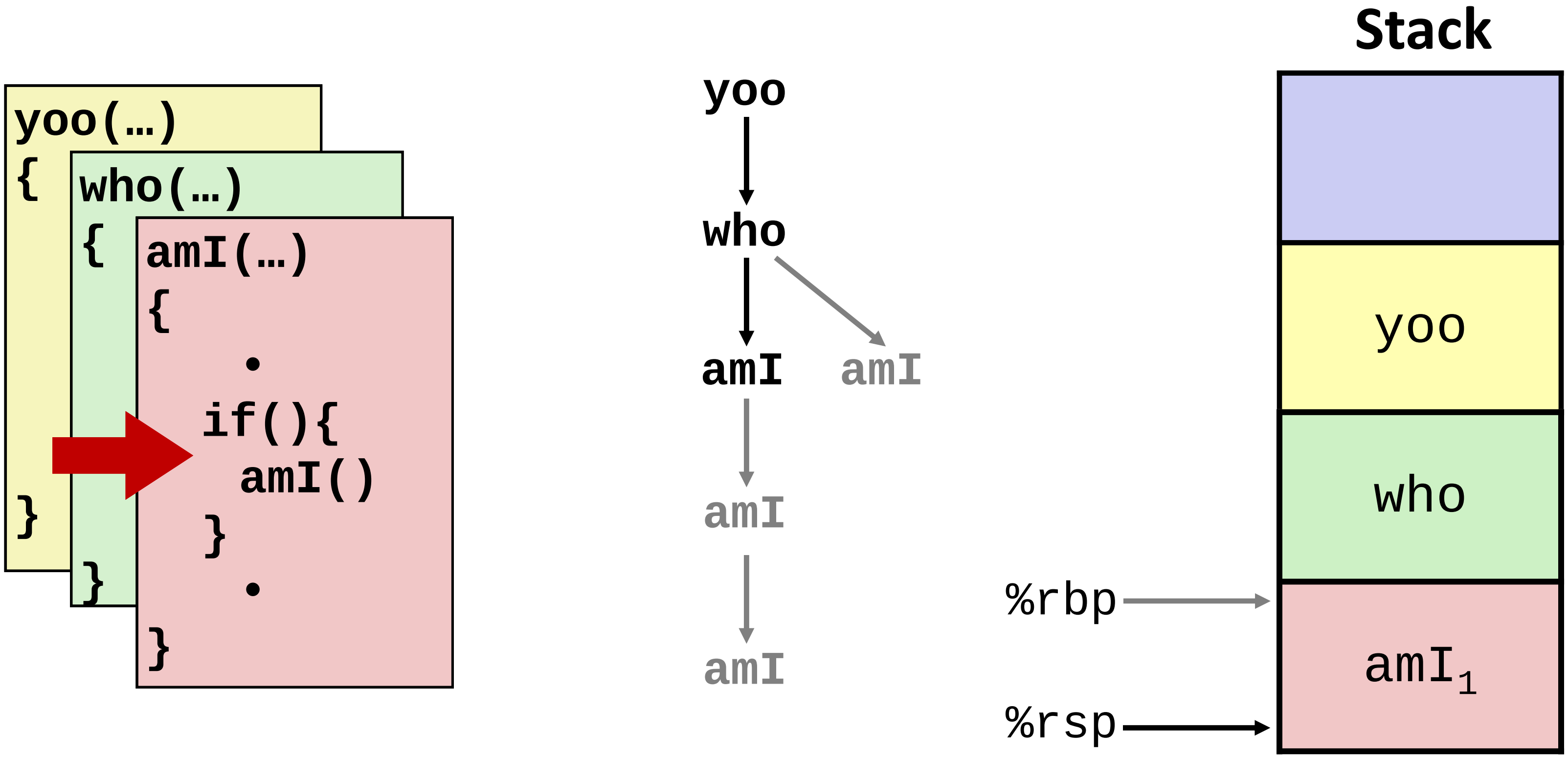




"create" frame
by manipulating %rsp

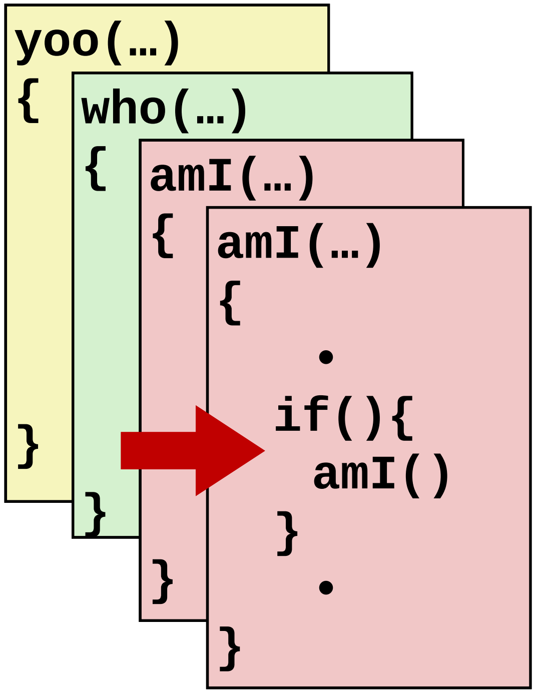
This frame will be allocated for storing local information about the who procedure

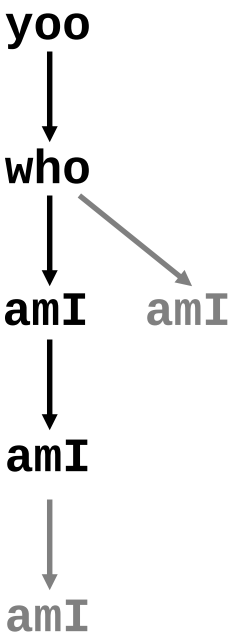
3) Call to amI (1)

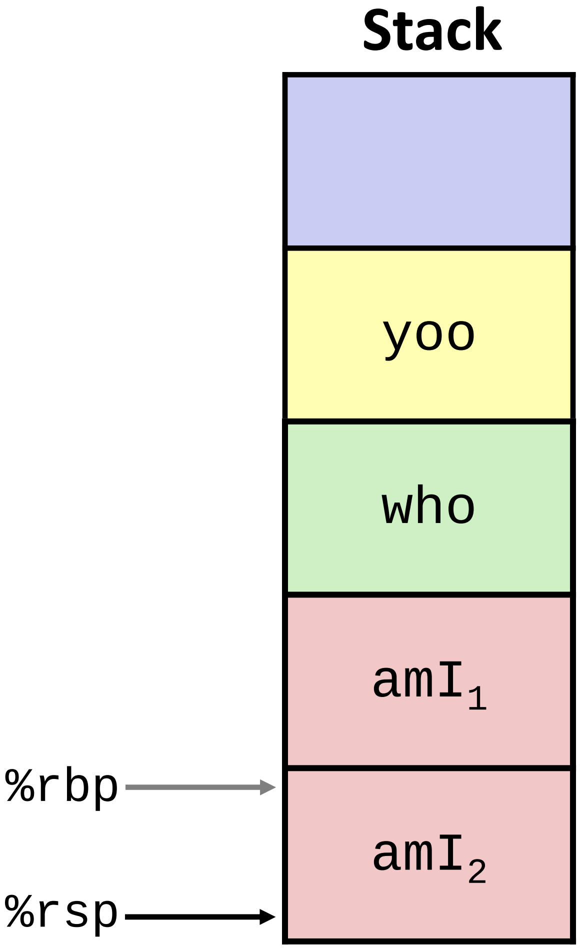


```
yoo(…)
{
}

who(…)
{
}

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

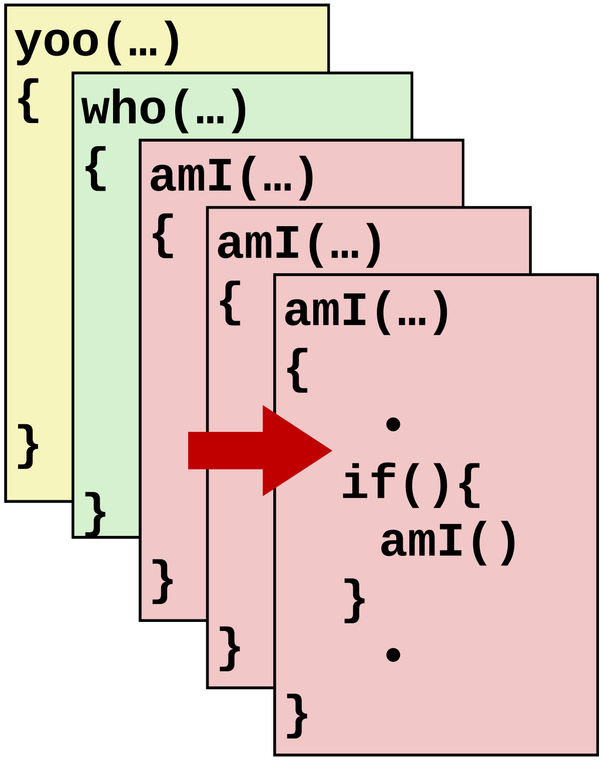
4) Recursive call to amI (2)



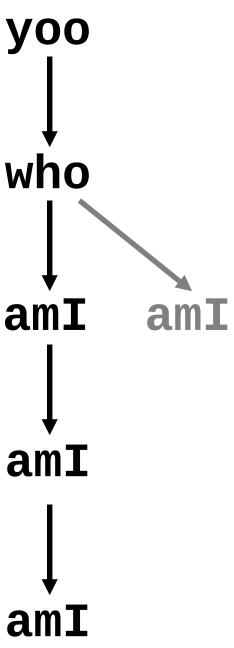


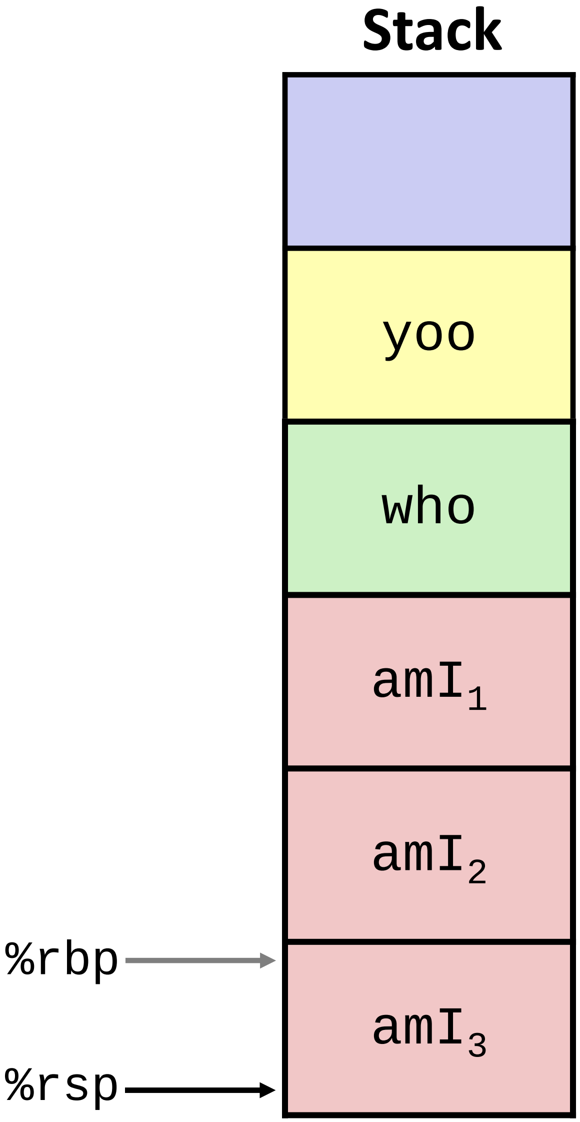


5) (another) Recursive call to amI (3)

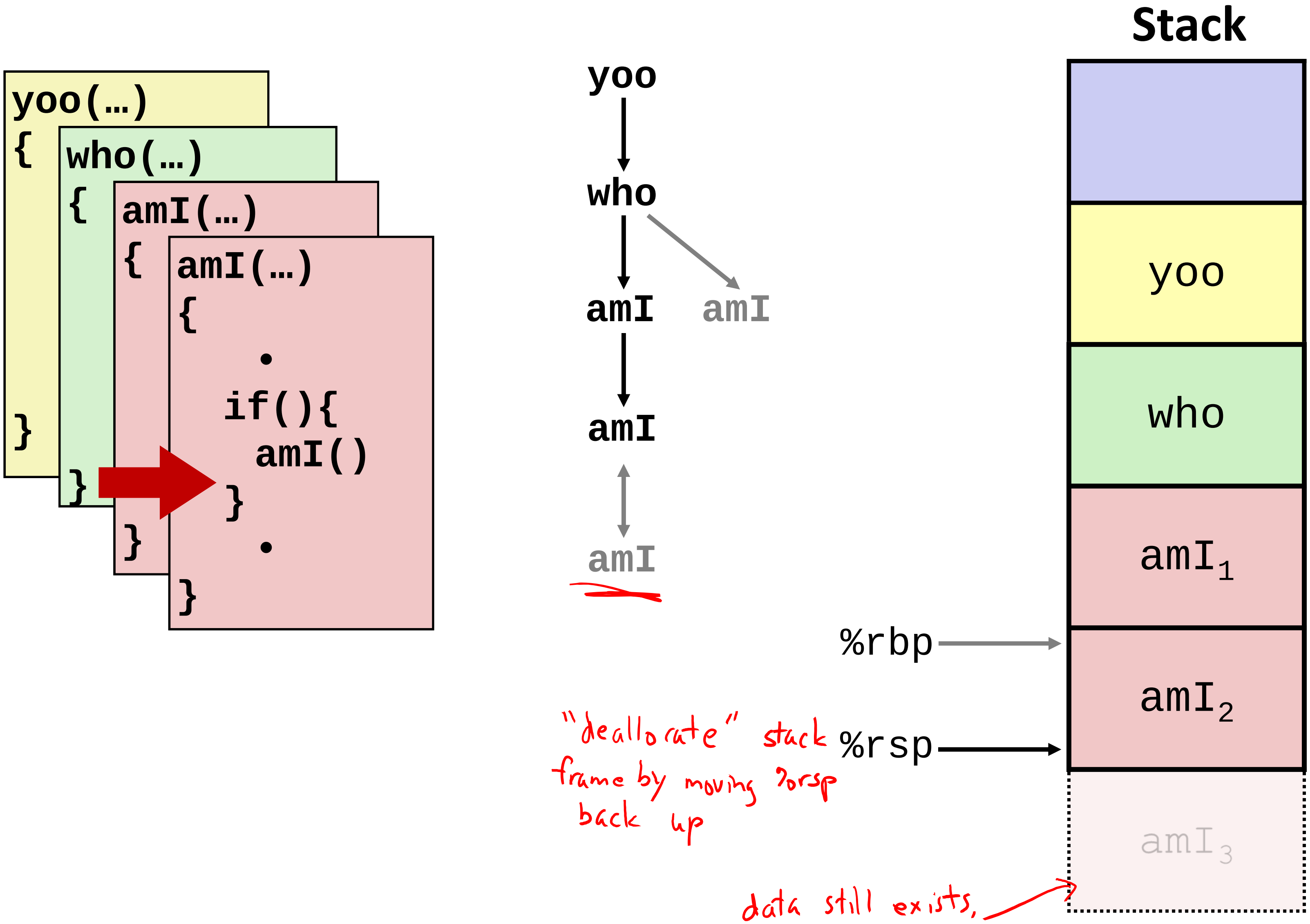


```
amI(…)
{
    •
  if(){
    amI()
  }
    •
}
```





6) Return from (another) recursive call to amI



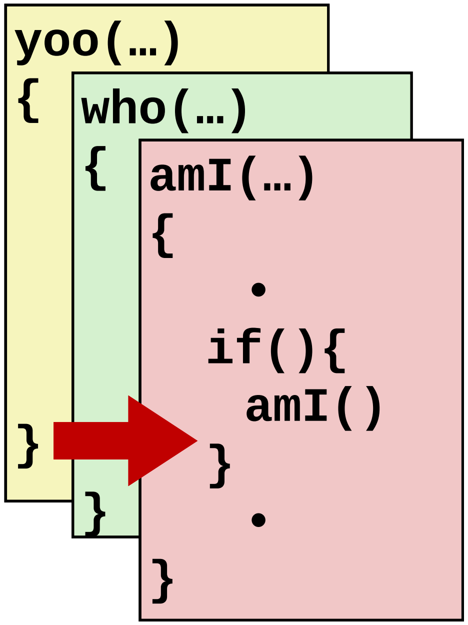
```
amI(…)
{
    •
  if(){
    amI()
  }
    •
}
```

"deallocate" stack frame by moving %rsp back up

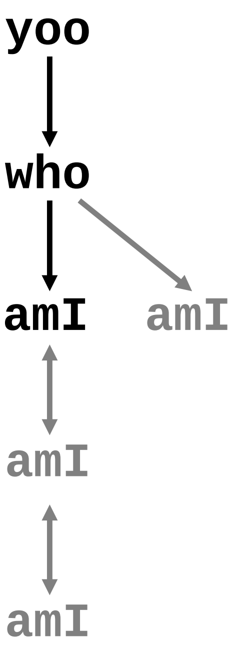
data still exists, but you shouldn't use it

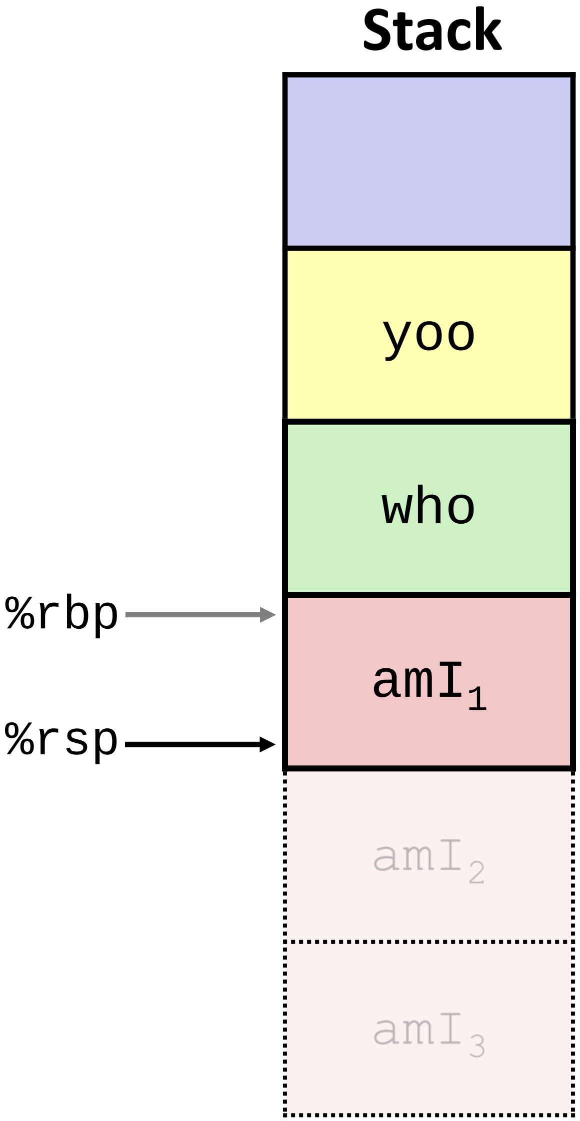
because at any point of time we could overwrite it;
since we plan on overwriting it anyways, no point in just "zeroing out" all the data here.

7) Return from recursive call to amI

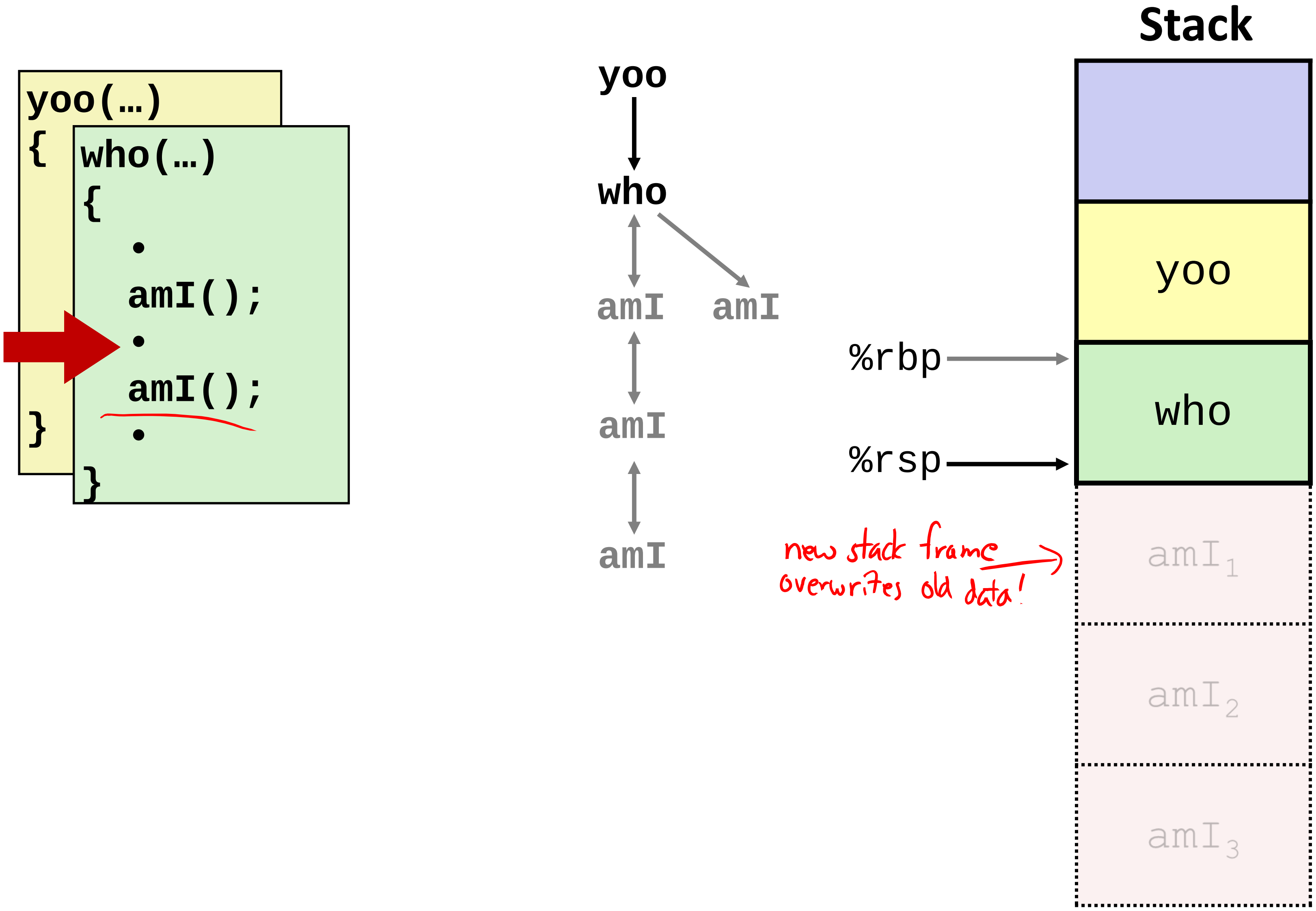


```
yoo(…)
{
  who(…)
  {
    amI(…)
    {
      •
      if(){
        amI()
      }
      •
    }
  }
}
```





8) Return from call to amI

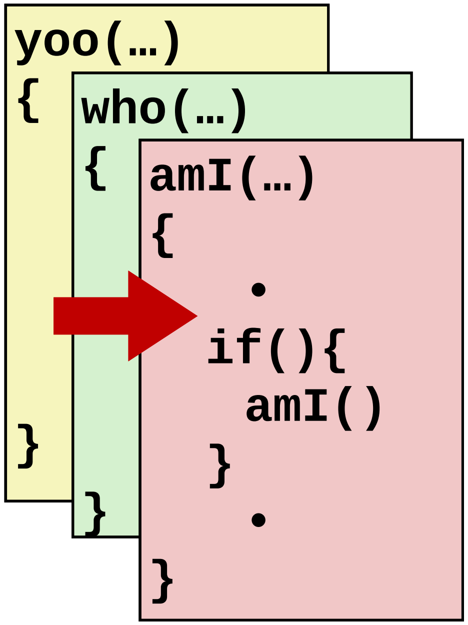


```
yoo(…)
{
}
```

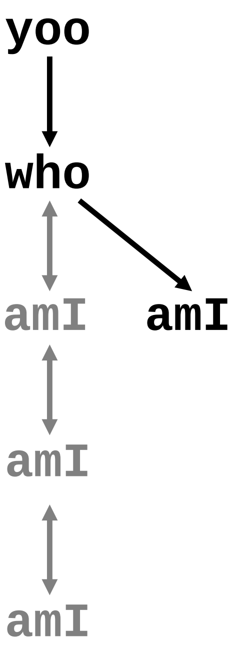
```
who(…)
{
   •
   amI();
   •
   amI();
   •
}
```

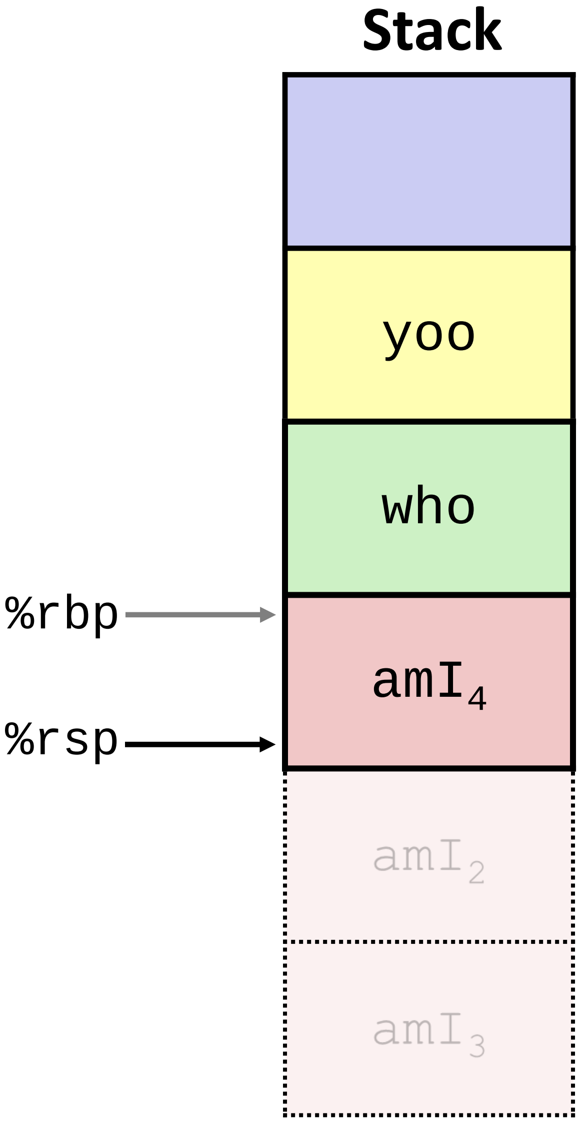
new stack frame overwrites old data!

9) (second) Call to `amI` (4)

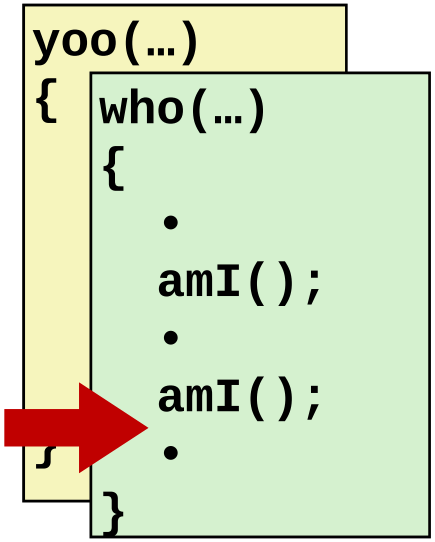


```
yoo(…)
{  who(…)
   {  amI(…)
      {
          •
        if(){
          amI()
        }
          •
      }
   }
}
```

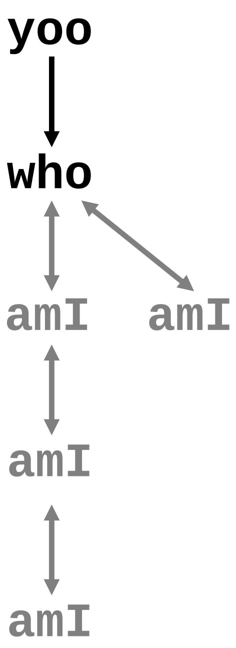


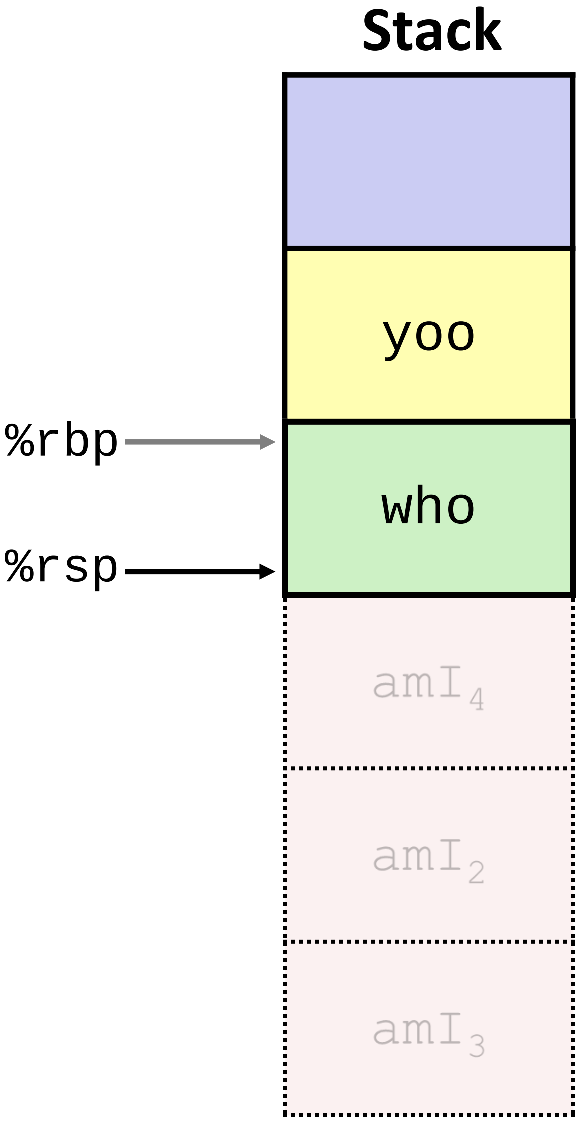


10) Return from (second) call to amI



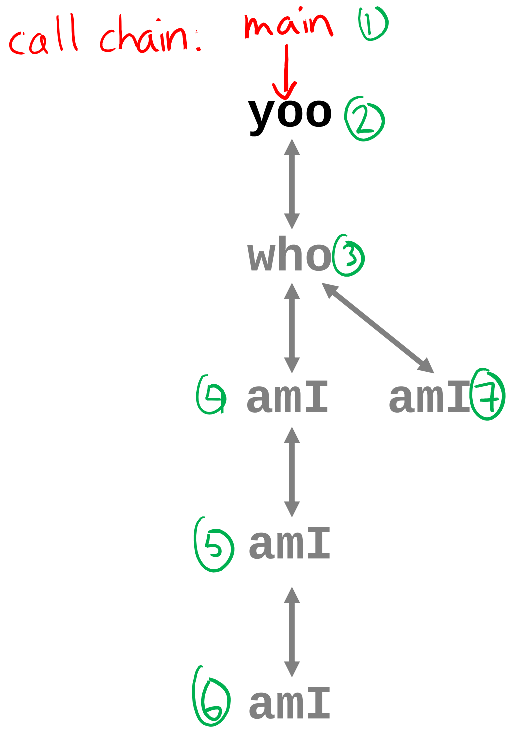
```
yoo(…)
{
  who(…)
  {
     •
     amI();
     •
     amI();
     •
  }
}
```

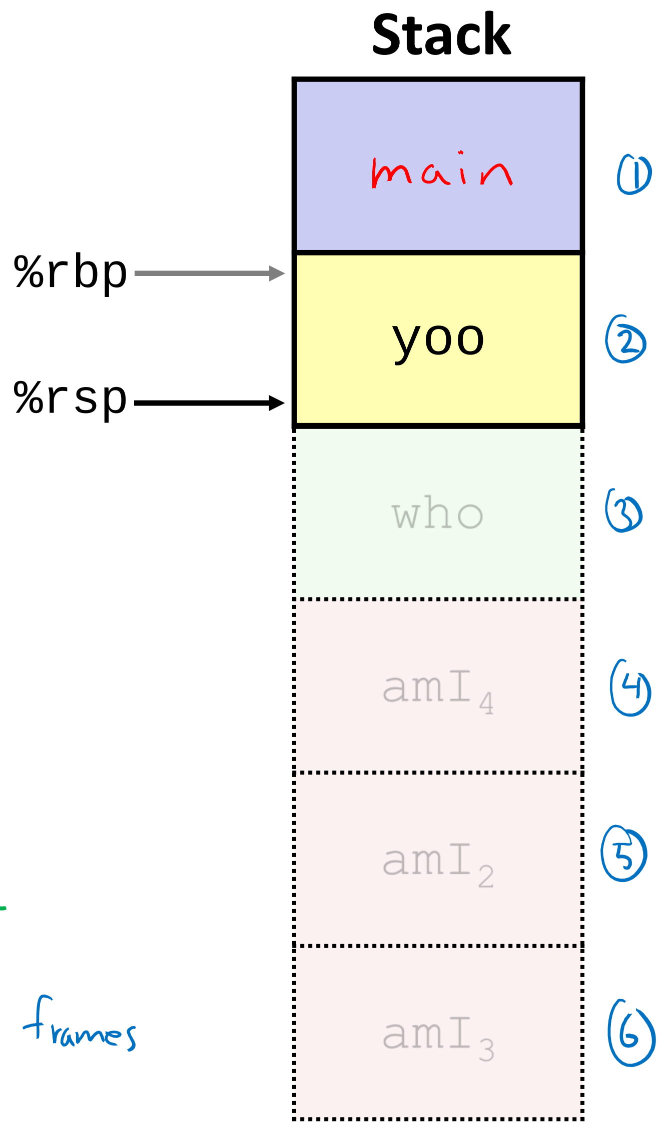




11) Return from call to who

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





total stack frames created: 7

maximum stack depth: 6 frames

x86-64/Linux Stack Frame

- Caller's Stack Frame
  - Extra arguments (if > 6 args) for this call

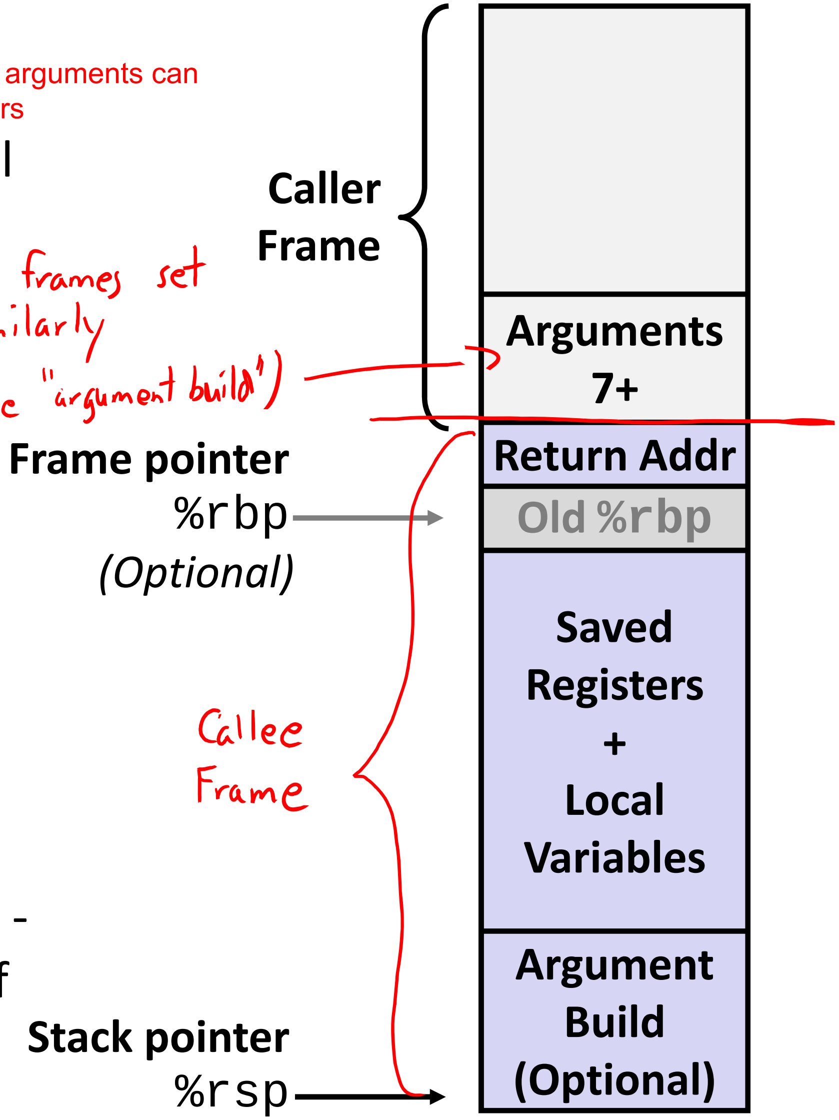
remember some of the arguments can be stored in the registers

- 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)

e.g. so that we don't overwrite arguments stored in the register by the caller (we save these to stack so we can replace the values in register at the end)

all frames set up similarly

(this is the "argument build")


Caller Frame
Arguments 7+
Return Addr
Frame pointer %rbp (Optional)
Old %rbp
Saved Registers + Local Variables
Argument Build (Optional)
Stack pointer %rsp
Callee Frame

Peer Instruction Question

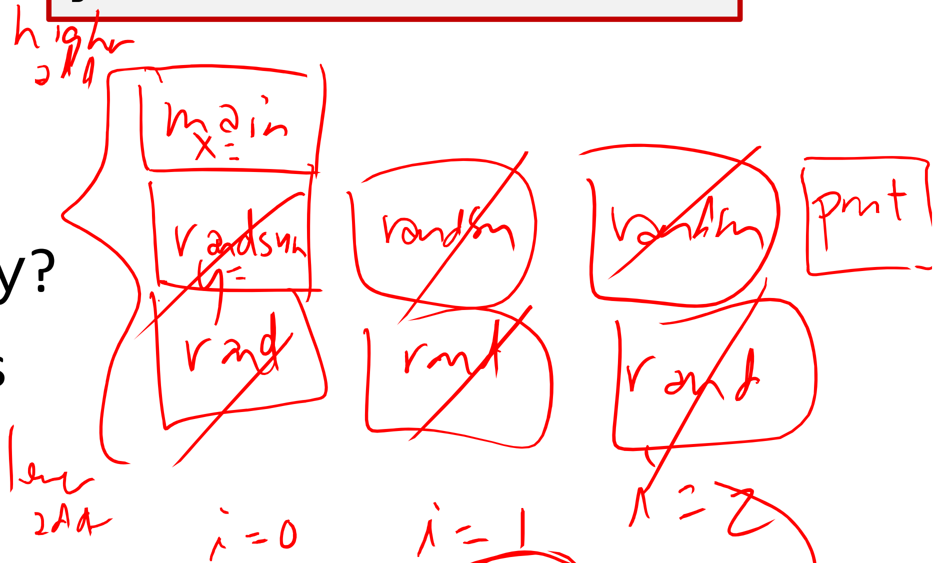
Vote only on 3rd question at http://pollev.com/rea

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

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

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

- *Higher/larger address:* [✓ 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 [✓ C. 3] D. 4