The Stack & Procedures

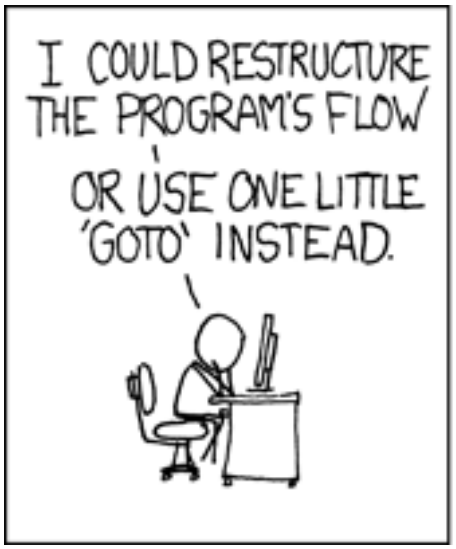
CSE 351 Autumn 2018

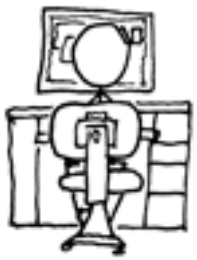
Instructor:

Justin Hsia

Teaching Assistants:

Akshat Aggarwal
Brian Dai
Kevin Bi
Sophie Tian
An Wang
Britt Henderson
Kory Watson
Teagan Horkan
Andrew Hu
James Shin
Riley Germundson





http://xkcd.com/571/

Administrivia

- Homework 2 due tonight
- Lab 2 due next Friday (10/26)
  - Ideally want to finish well before the midterm
- Homework 3 released next week
  - On midterm material, but due after the midterm

- **Midterm** (10/29, 5:10-6:20 pm, KNE 210 & 220)
  - Reference sheet + 1 *handwritten* cheat sheet
  - Find a study group! Look at past exams!
  - Average is typically around 75%
  - **Review session** (10/26) in EEB 105 from 5-7 pm

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();
```

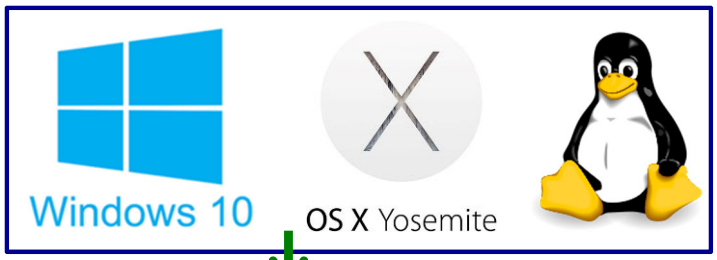
Assembly language:

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

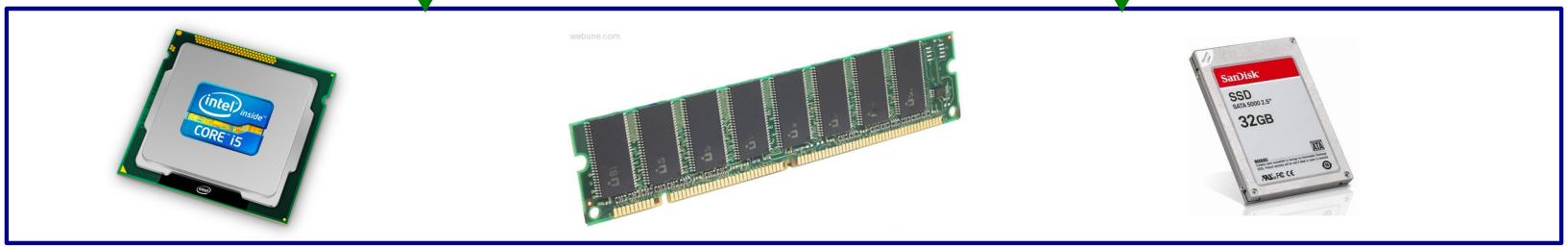
Machine code:

```
0111010000011000
100011010000010000000010
100100111000010
11000001111110100011111
```

OS:



Computer system:

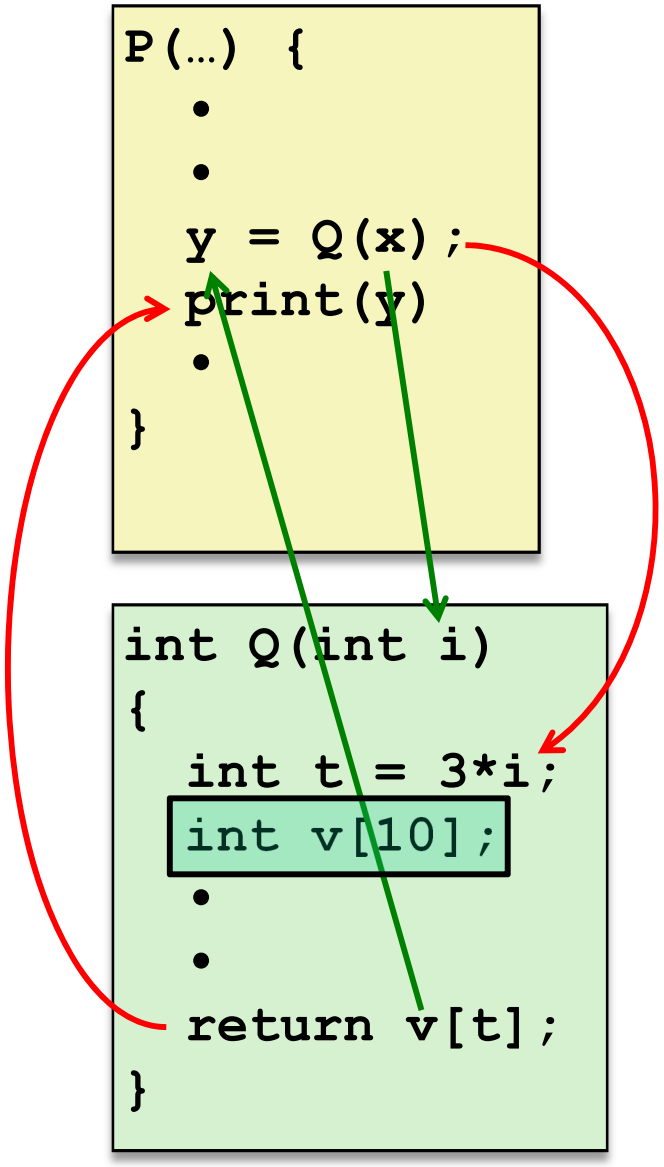


- Memory & data
- Integers & floats
- x86 assembly
- Procedures & stacks
- Executables
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

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

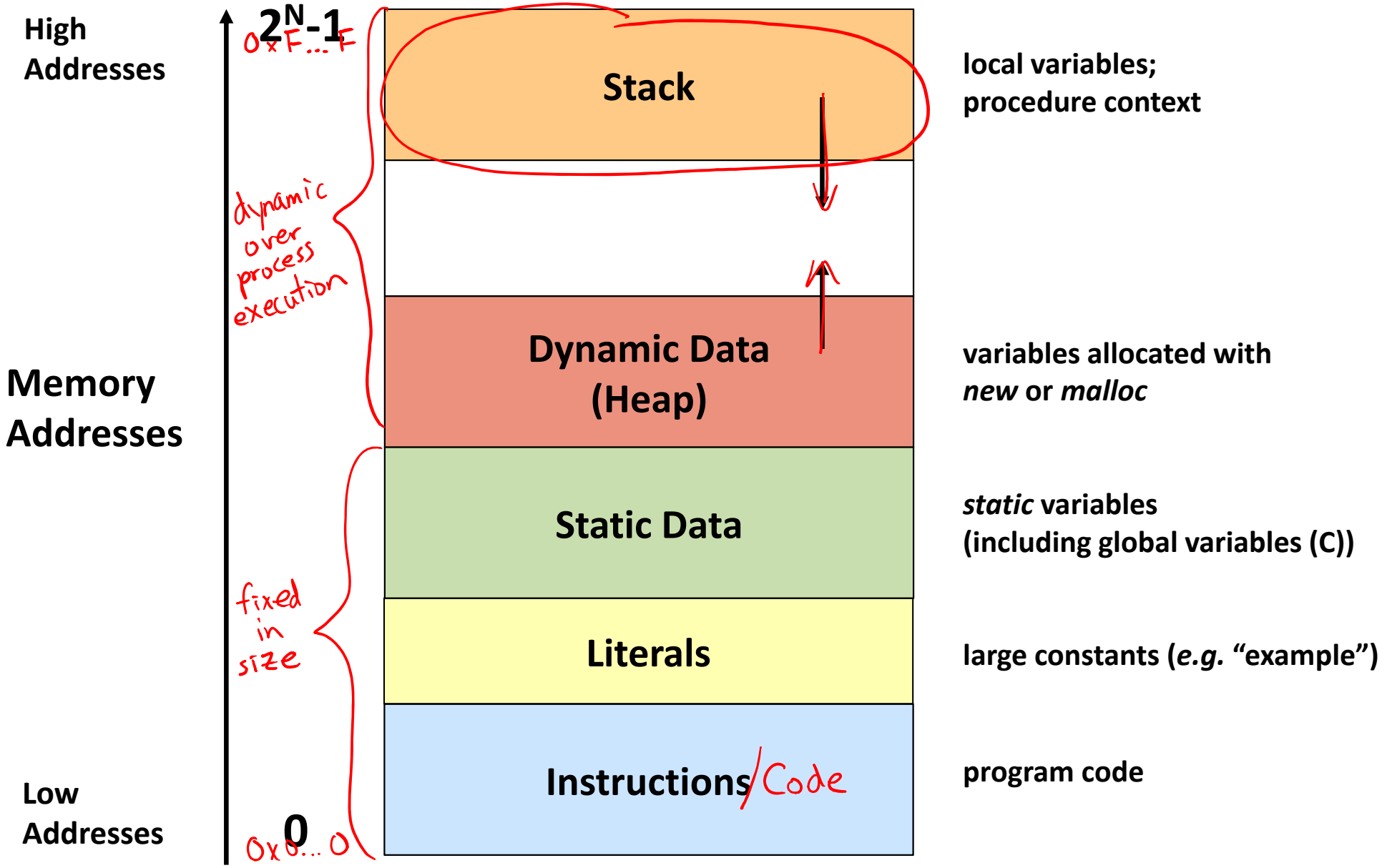
❖ All implemented with machine instructions!
   - An x86-64 procedure uses only those mechanisms required for that procedure



Procedures

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

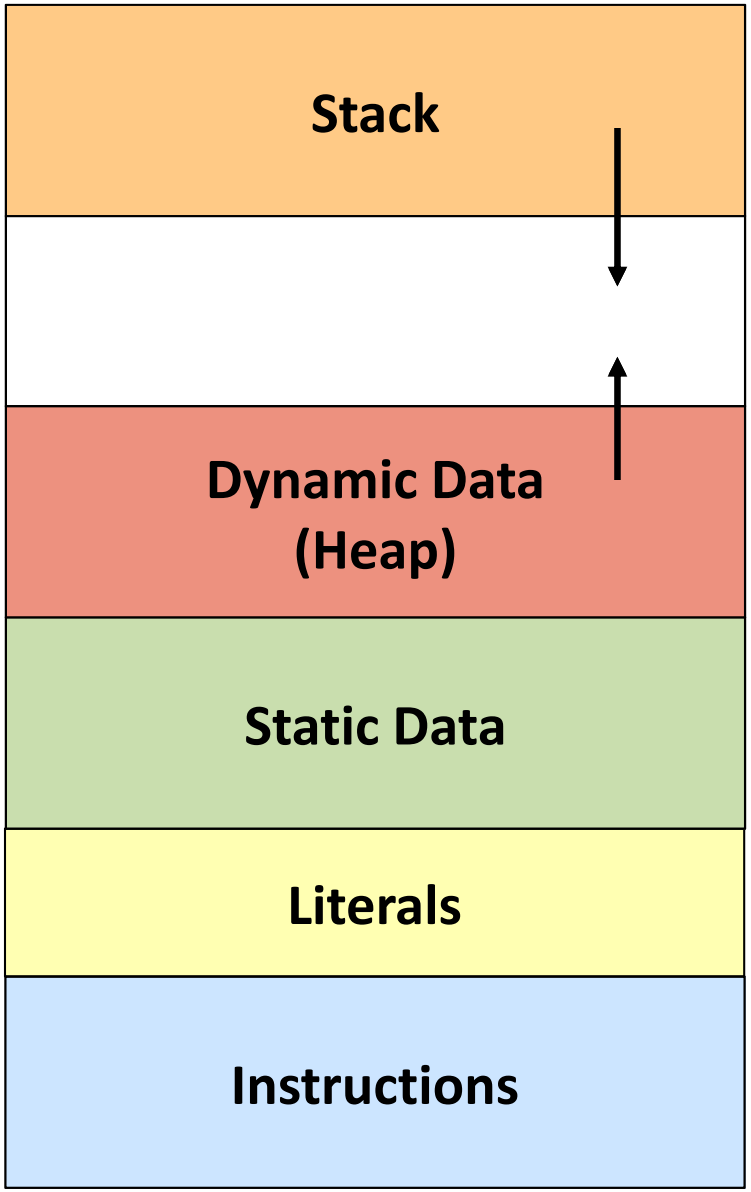
Simplified Memory Layout



Memory Permissions

segmentation faults?

accessing memory in a way that you are not allowed to



writable; not executable

Managed "automatically" (by compiler)

grow towards each other to maximize use of space

writable; not executable

Managed by programmer

writable; not executable

Initialized when process starts

read-only; not executable

Initialized when process starts

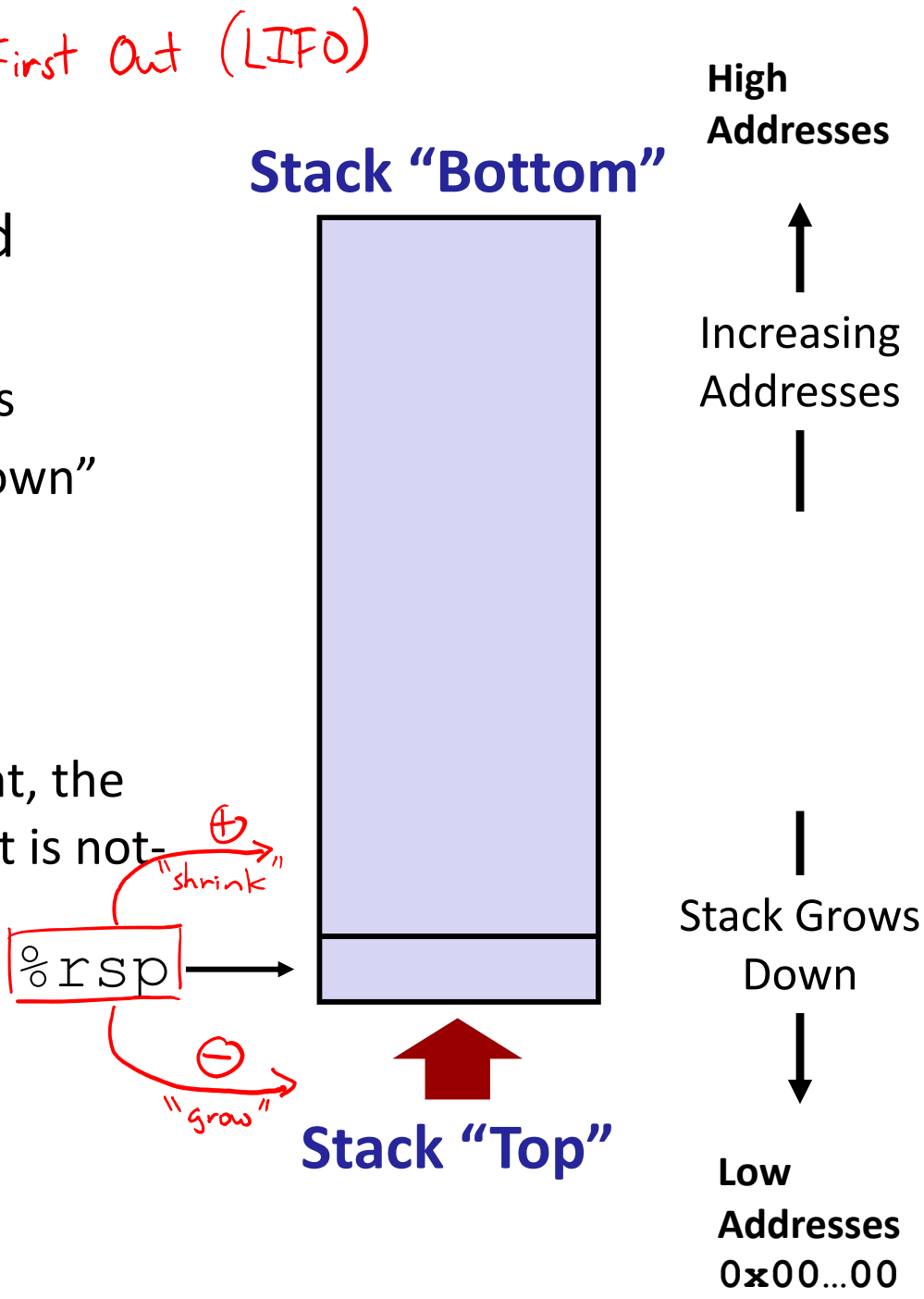
read-only; executable

Initialized when process starts

x86-64 Stack

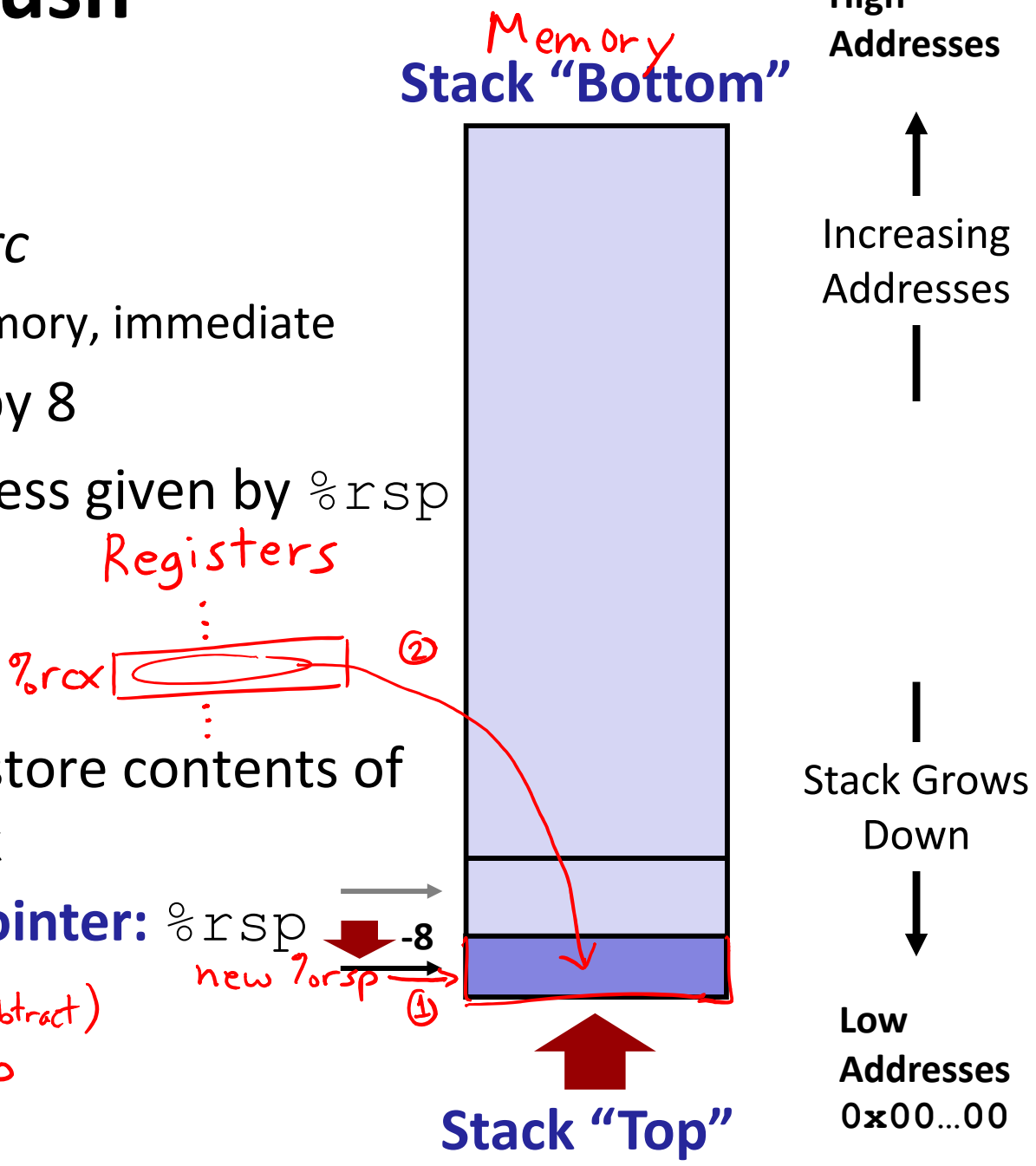
Last In, First Out (LIFO)

- 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

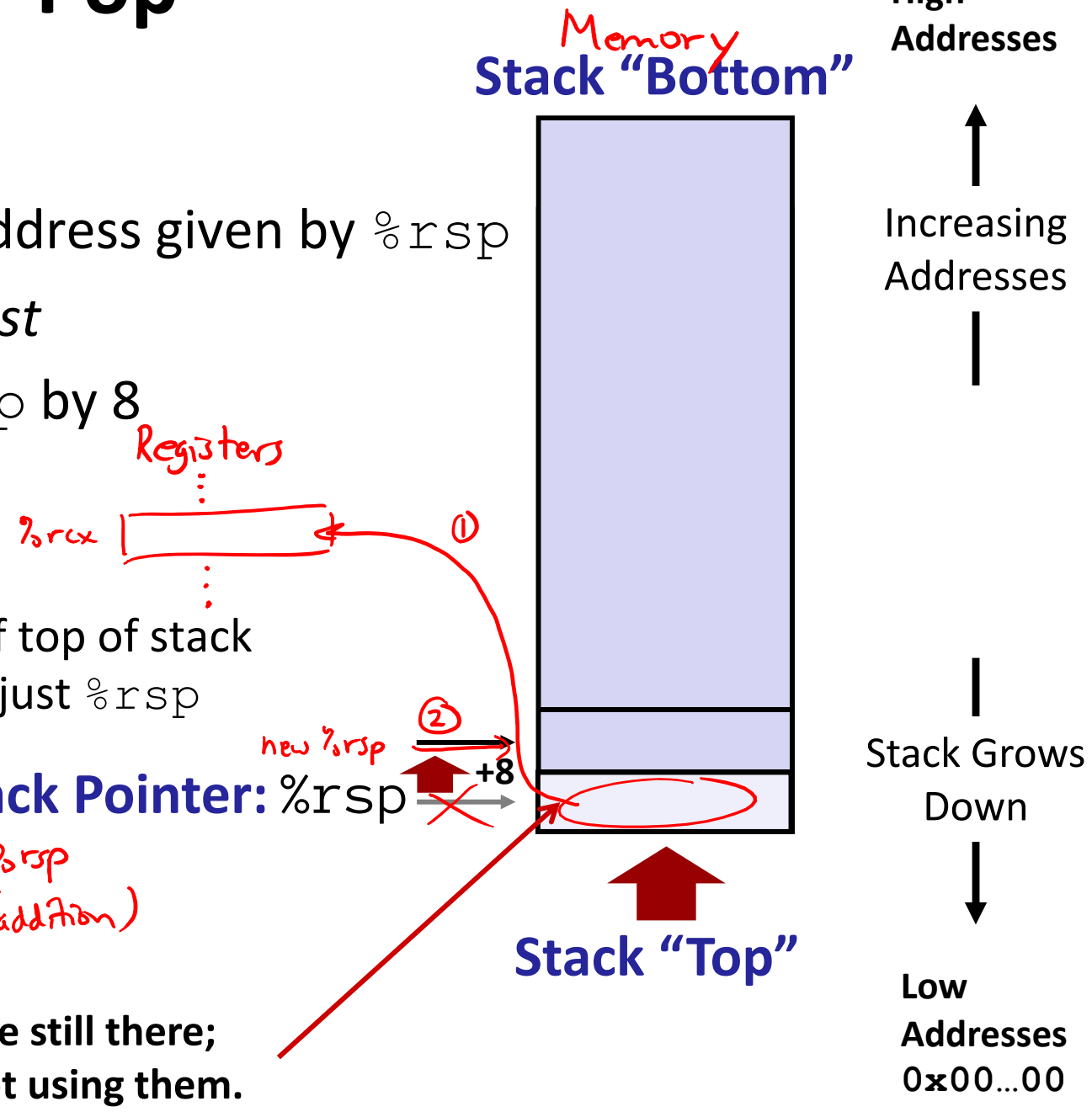
- `pushq` *src*
  - Fetch operand at *src*
    - *Src* can be reg, memory, immediate
  - ***Decrement*** `%rsp` by 8
  - Store value at address given by `%rsp`
- Example:
  - **`pushq %rcx`**
  - Adjust `%rsp` and store contents of `%rcx` on the stack



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

x86-64 Stack: Pop

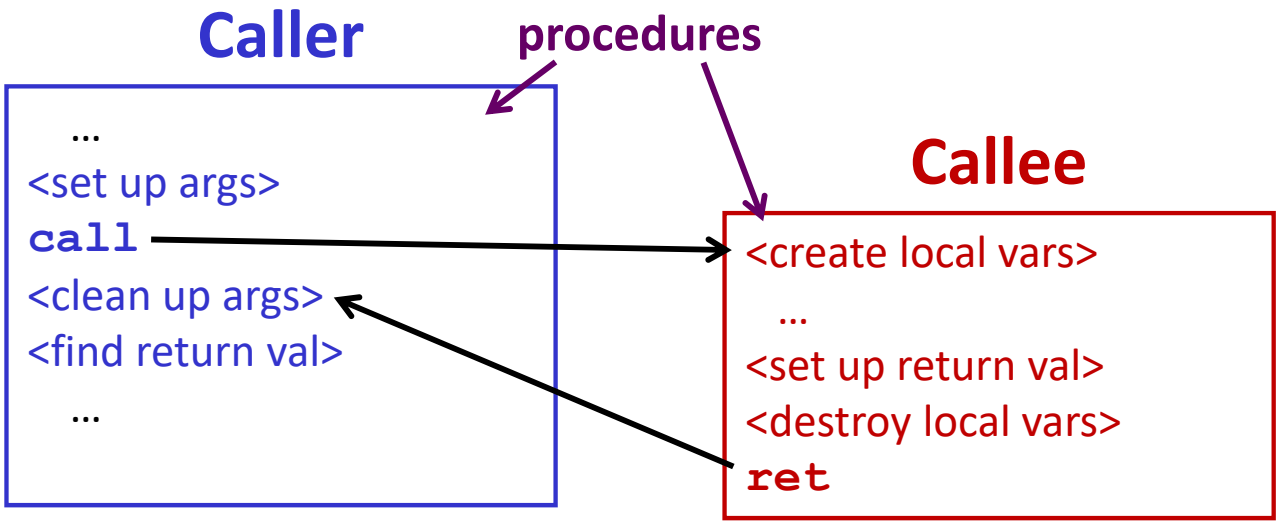
- `popq` *dst*
  - Load value at address given by `%rsp`
  - Store value at *dst*
  - ***Increment*** `%rsp` by 8
- Example:
  - **`popq %rcx`**
  - Stores contents of top of stack into `%rcx` and adjust `%rsp`



Procedures

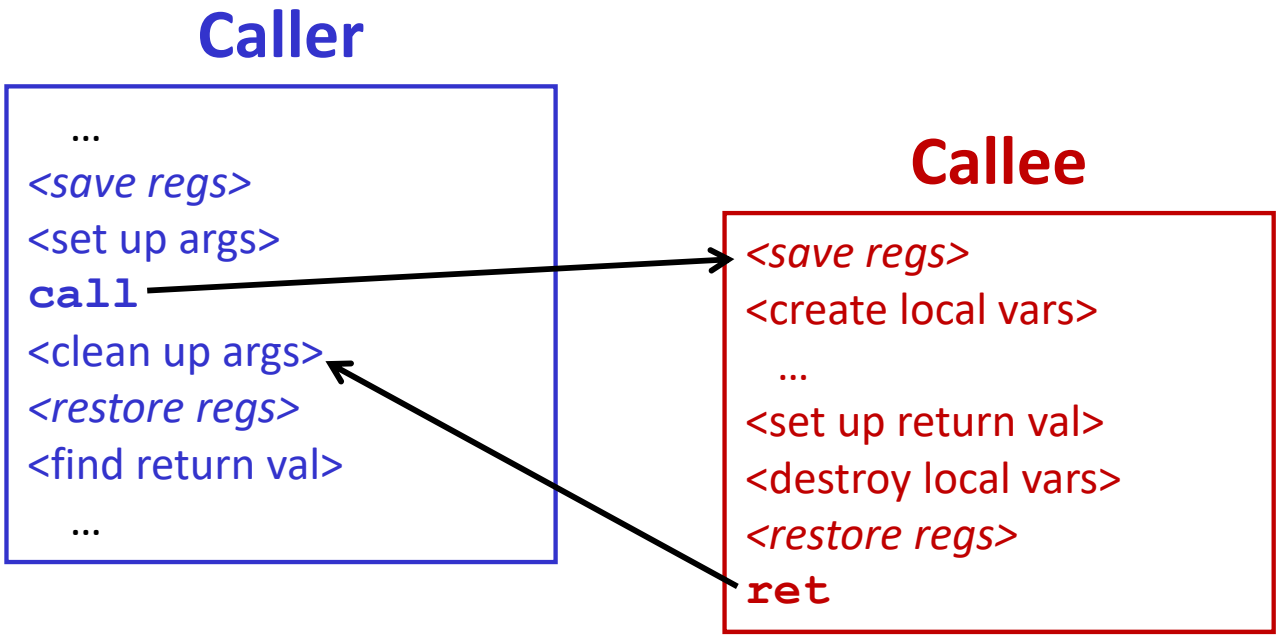
- Stack Structure
- **Calling Conventions**
  - **Passing control**
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion

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?

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

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

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
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?*)
  2) Jump to ***label***

  ① move %rsp down
  ② store ret addr at %rsp
  ③ label → %rip
- Return address:
  - Address of instruction immediately after **`call`** instruction
  - Example from disassembly:

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

Return address = **0x400549**

- Procedure return: `ret` (special pop)
  1) Pop return address from stack
  2) Jump to address

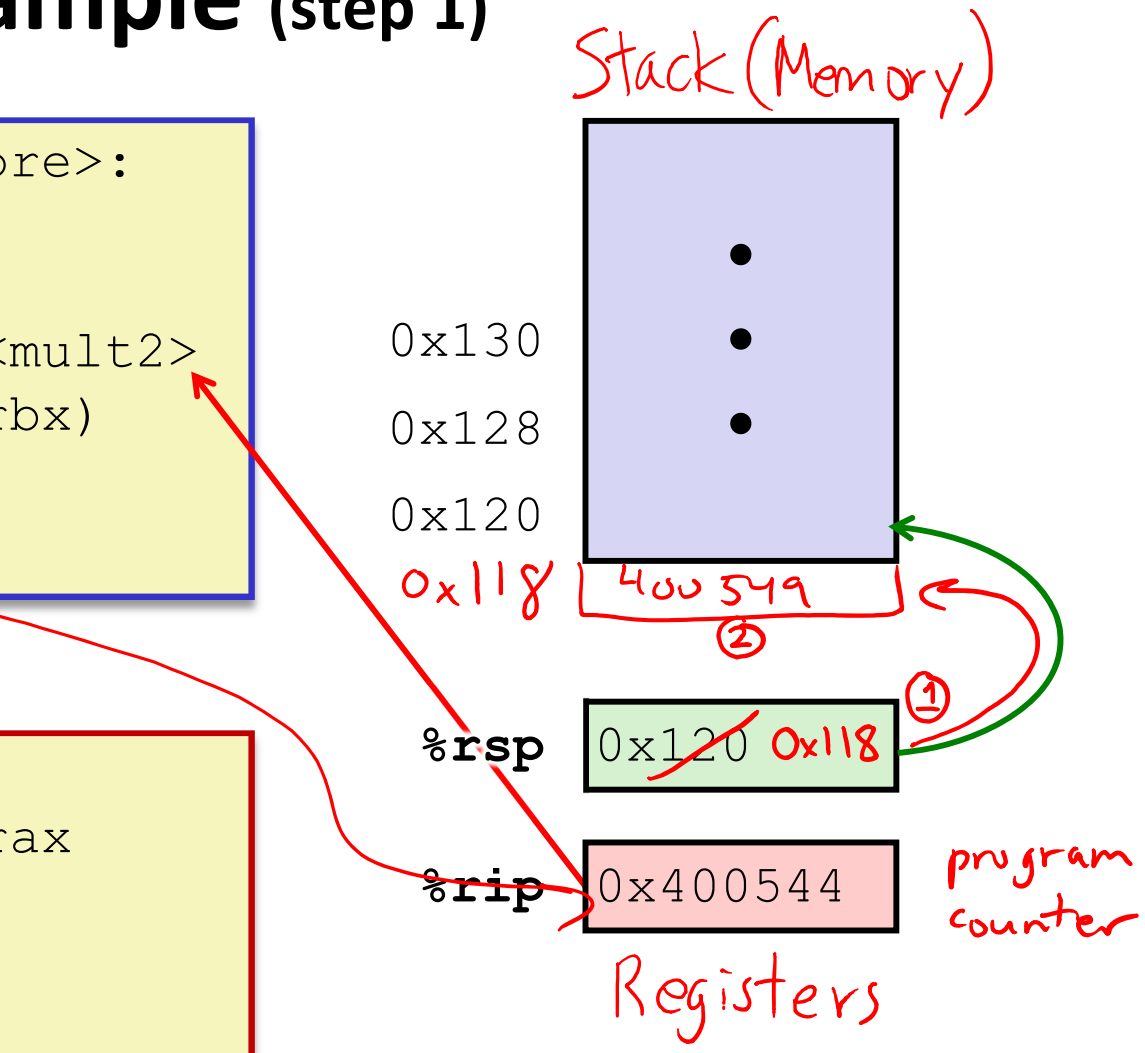
  ① read ret addr at %rsp into %rip
  ② move %rsp up

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

Procedure Call Example (step 1)

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

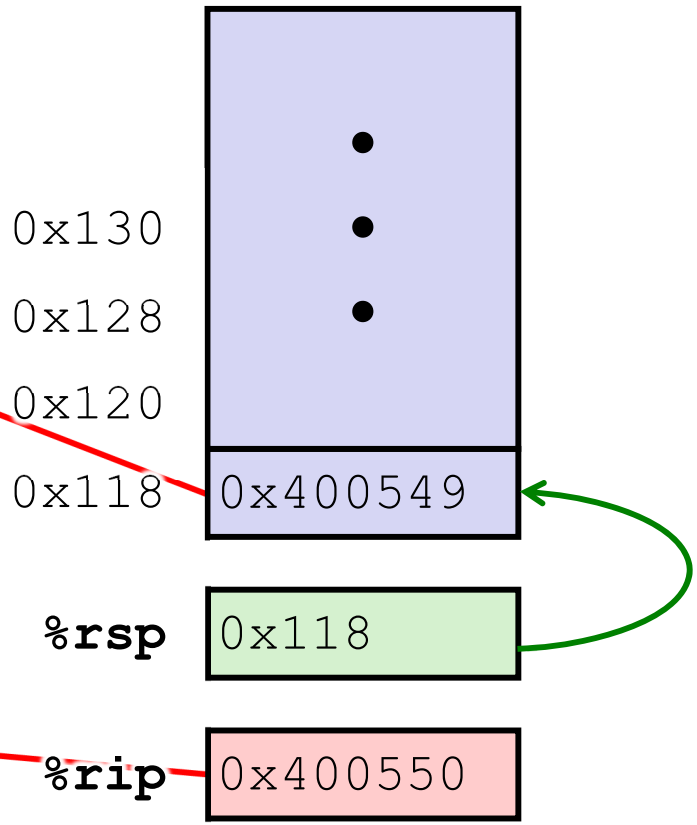
```
0000000000400550 <mult2>:
   400550:  movq    %rdi,%rax
   •
   •
   400557:  ret
```



Procedure Call Example (step 2)

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

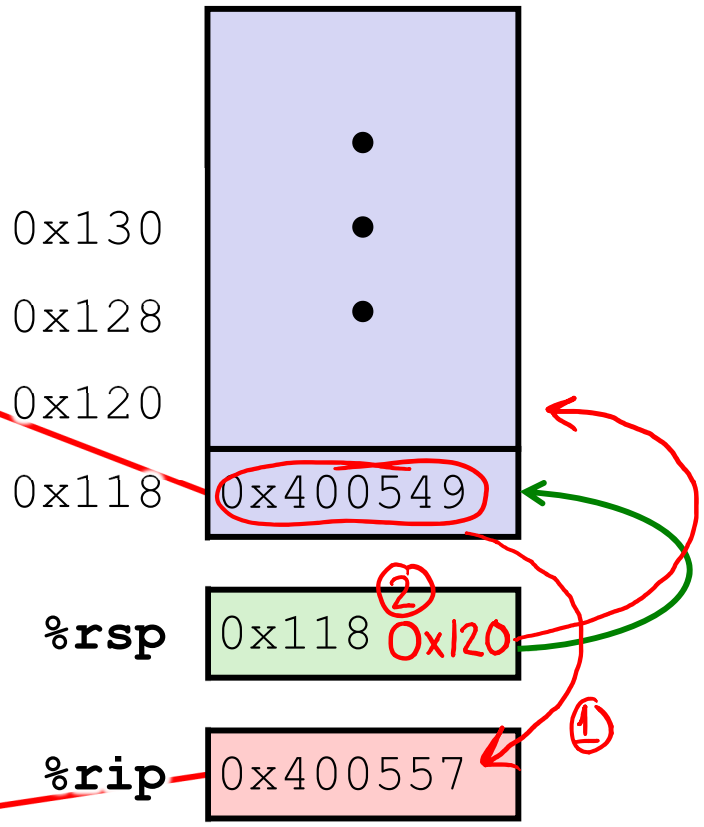
```
0000000000400550 <mult2>:
   400550:  movq   %rdi,%rax
   •
   •
   400557:  ret
```



Procedure Return Example (step 1)

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

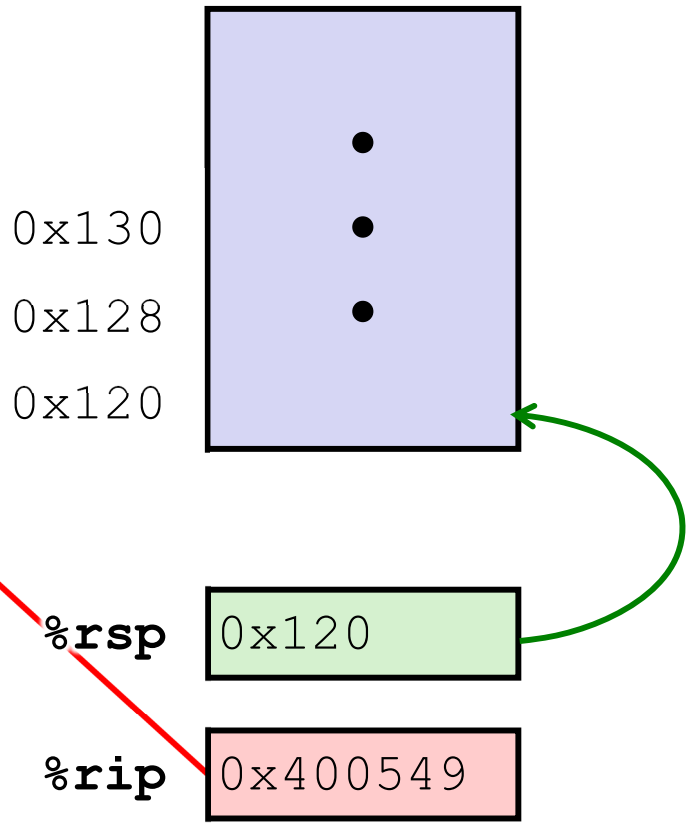
```
0000000000400550 <mult2>:
   400550:  movq   %rdi,%rax
   •
   •
   400557:  ret
```



Procedure Return Example (step 2)

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

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



Procedures

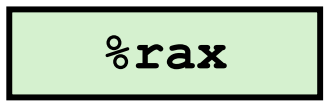
- Stack Structure
- **Calling Conventions**
  - Passing control
  - **Passing data**
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion

Procedure Data Flow

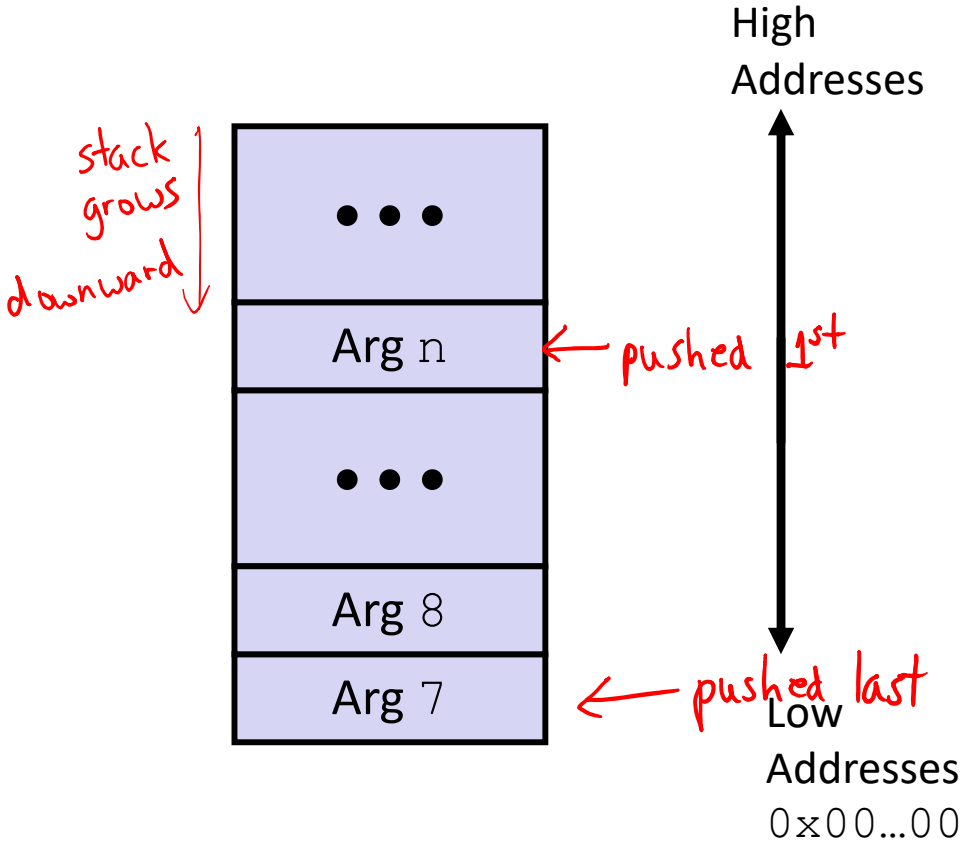
Registers (NOT in Memory)

- First 6 arguments

- Return value



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

1st call recurses twice

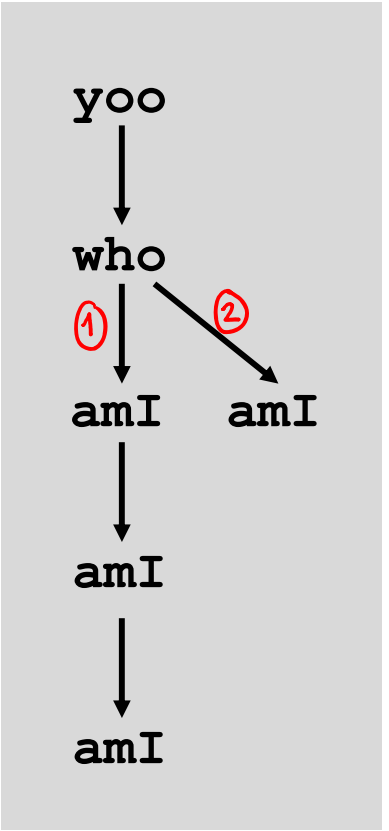
2nd call doesn't recurse

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

based on condition

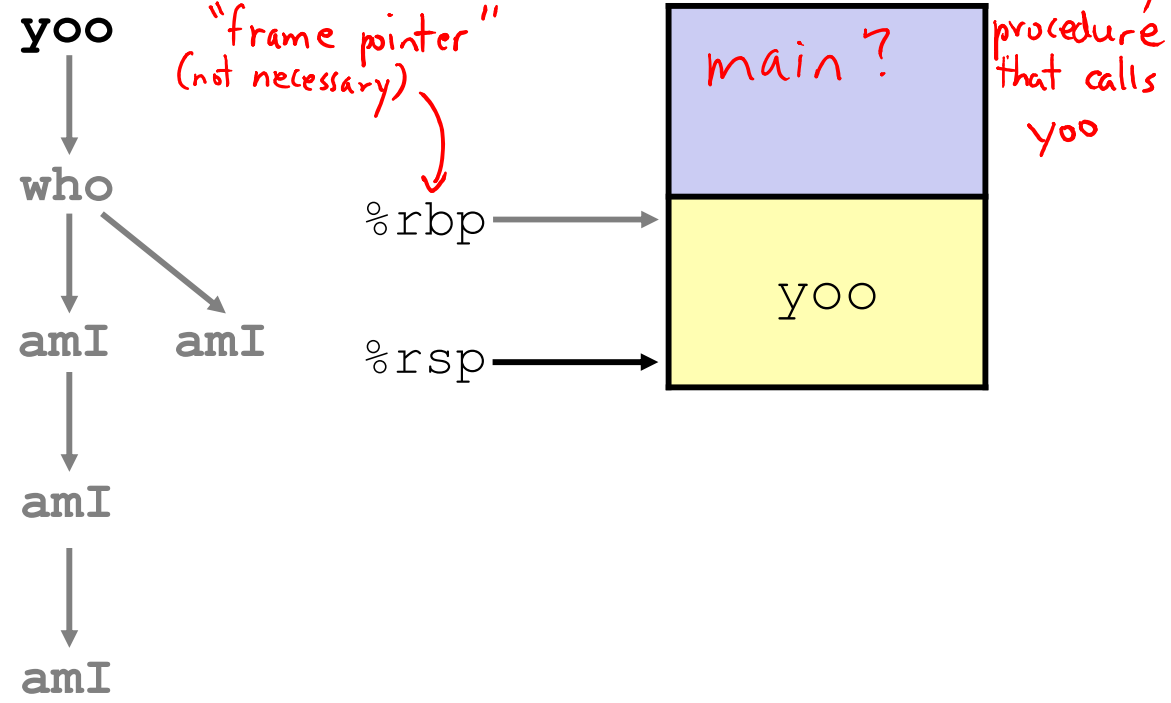
Procedure `amI` is recursive (calls itself)

Example Call Chain

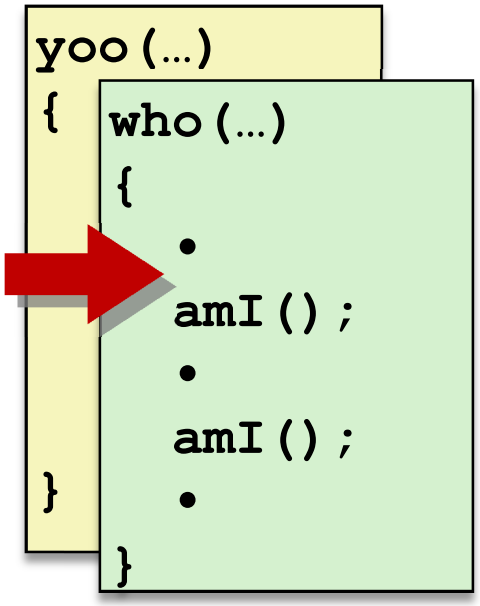


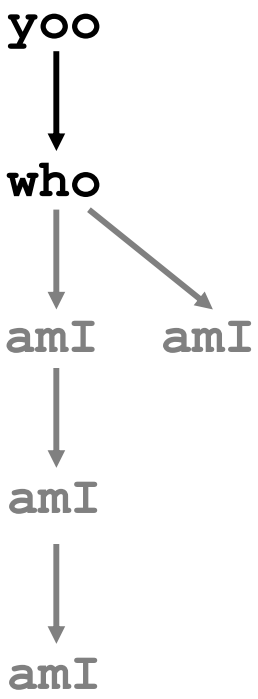
1) Call to yoo

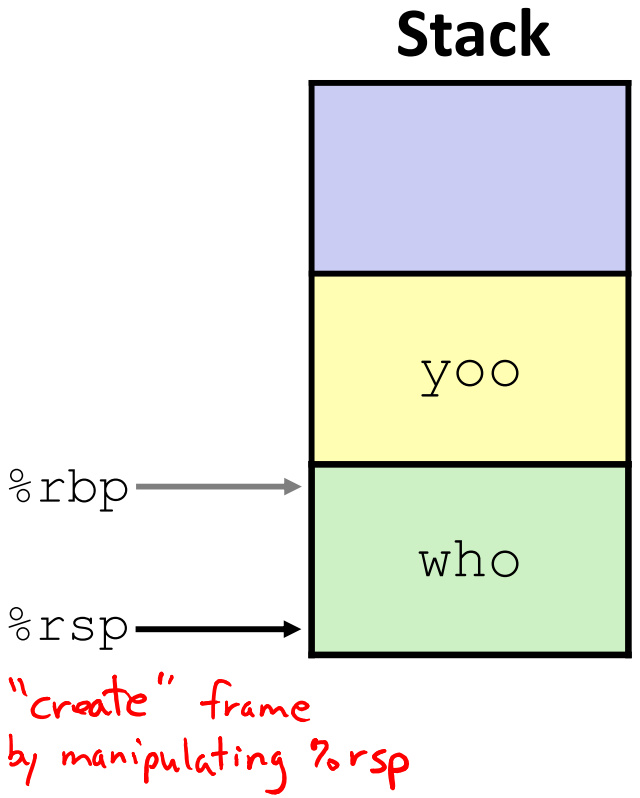
```
yoo(...)
{
    •
    •
    who();
    •
    •
}
```



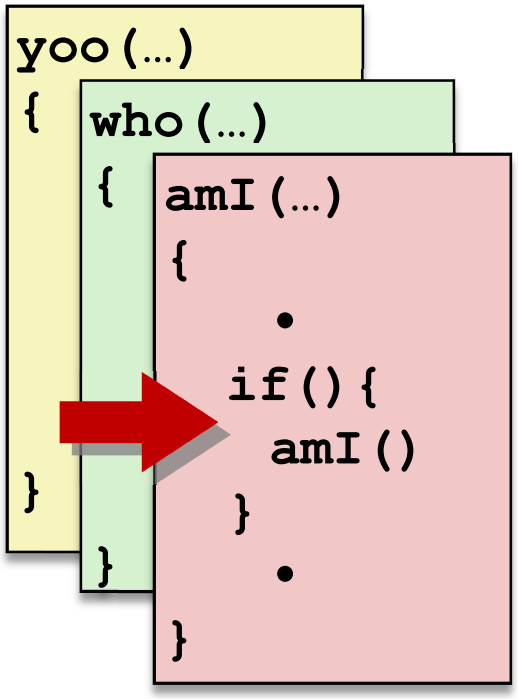
2) Call to who

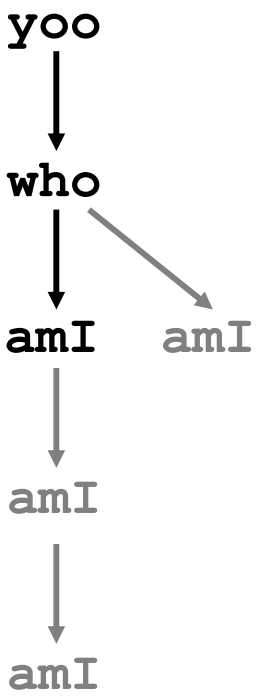

yoo(...)
{
who(...)
{
•
amI();
•
amI();
•
}
}

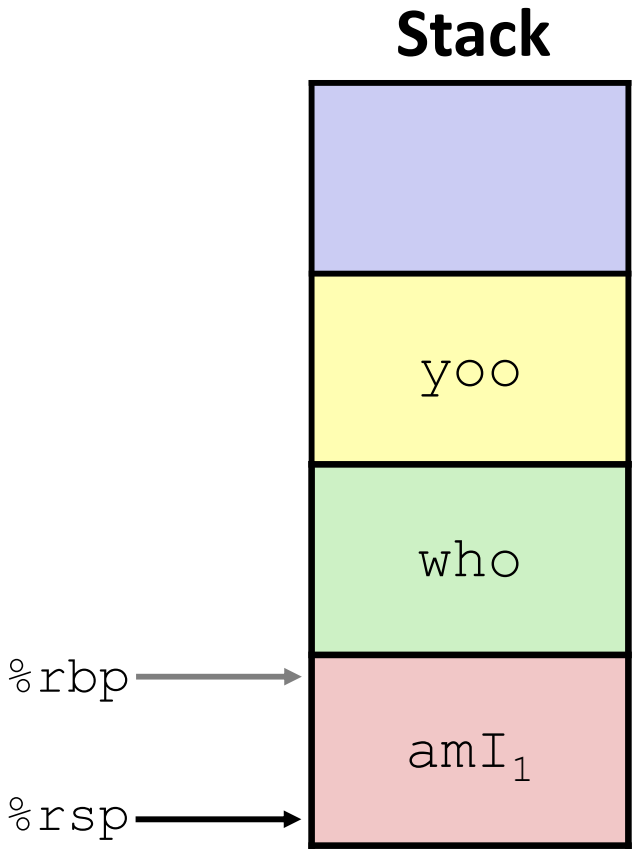

yoo
who
amI
amI
amI
amI


Stack
yoo
who
%rbp
%rsp
"create" frame
by manipulating %rsp

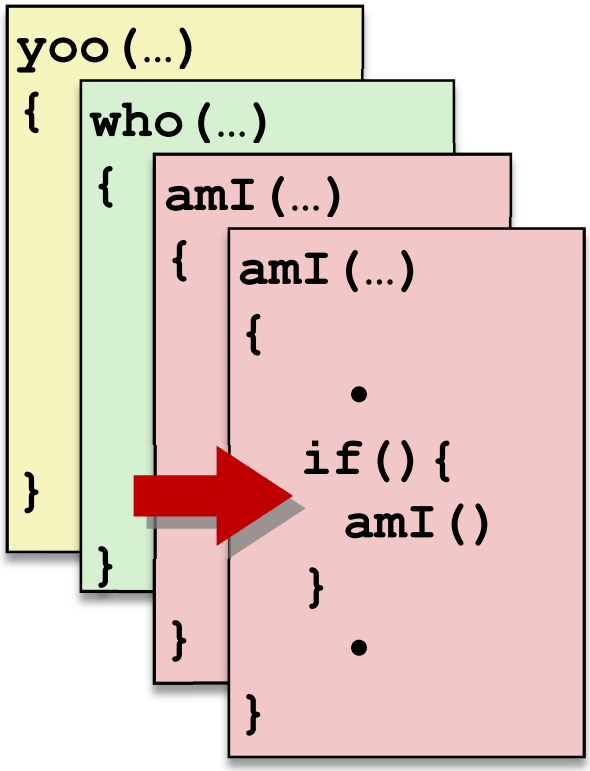
3) Call to amI (1)

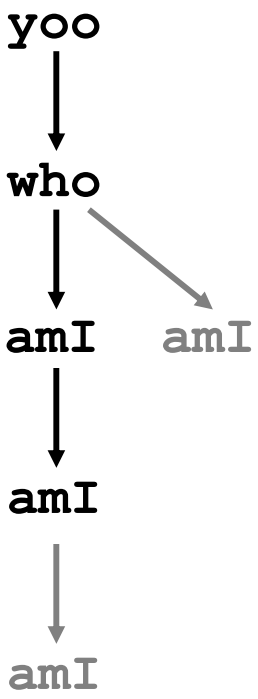


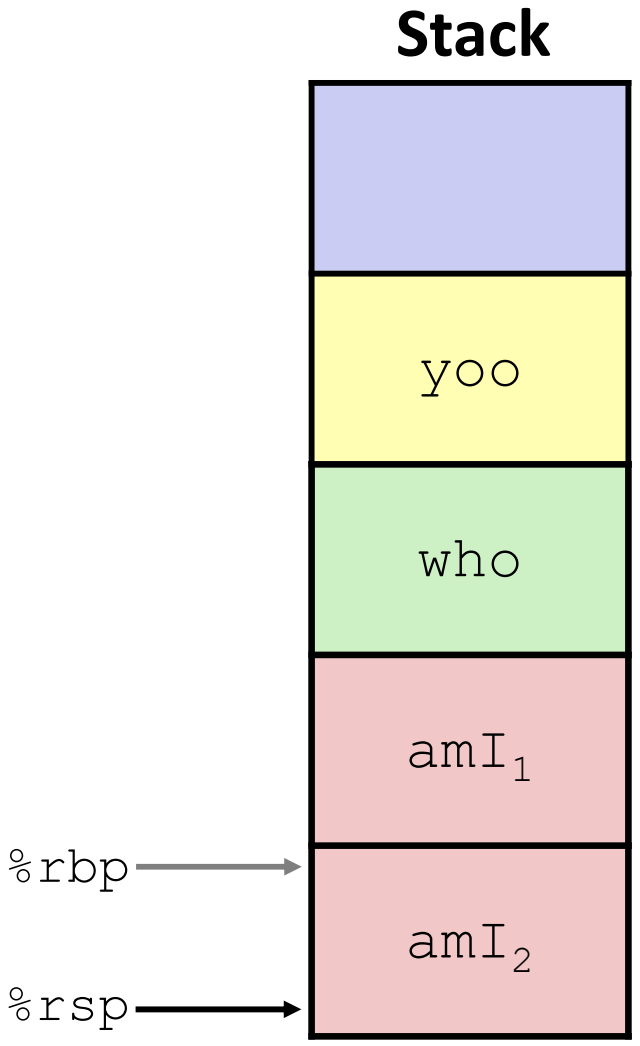




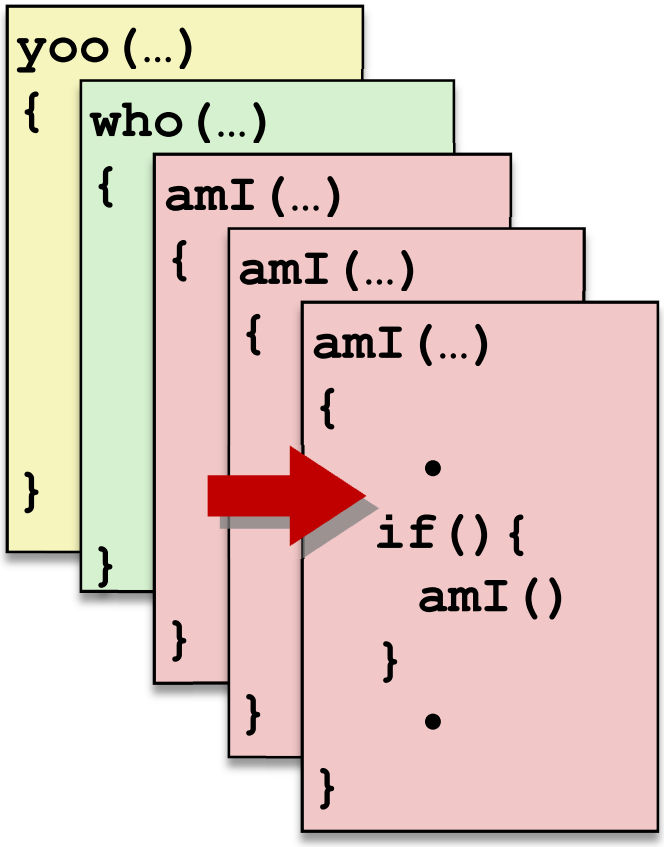
4) Recursive call to amI (2)

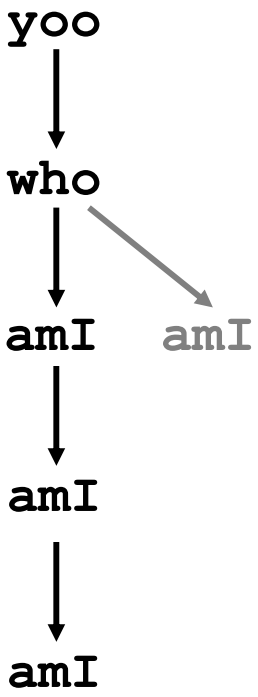


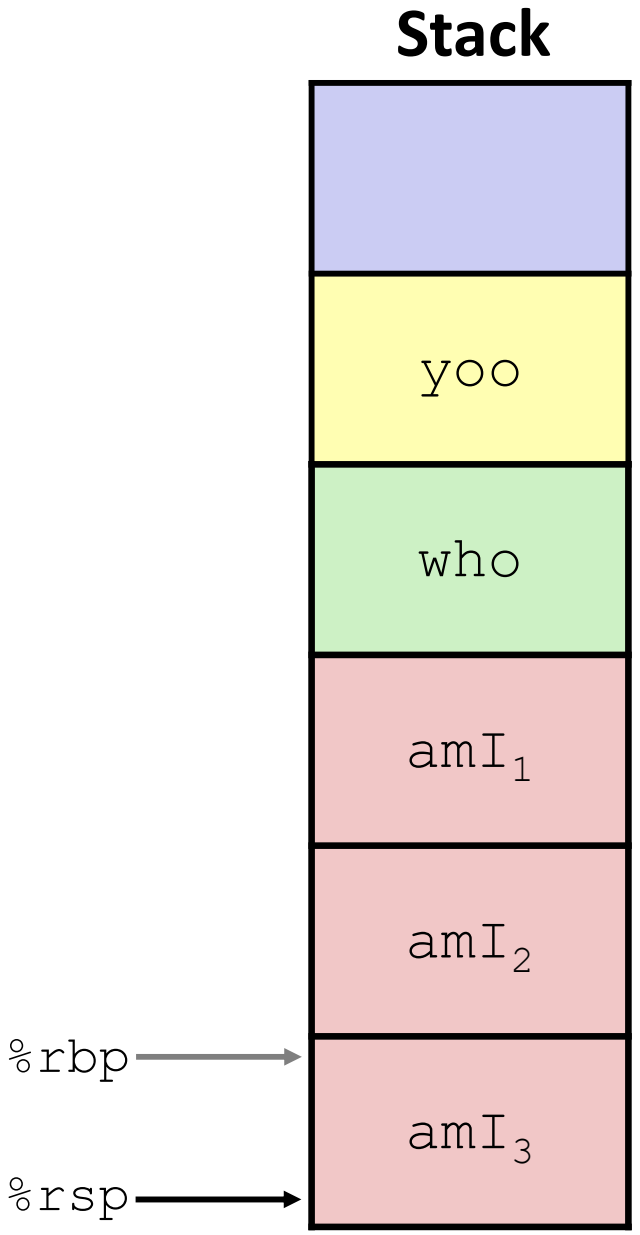




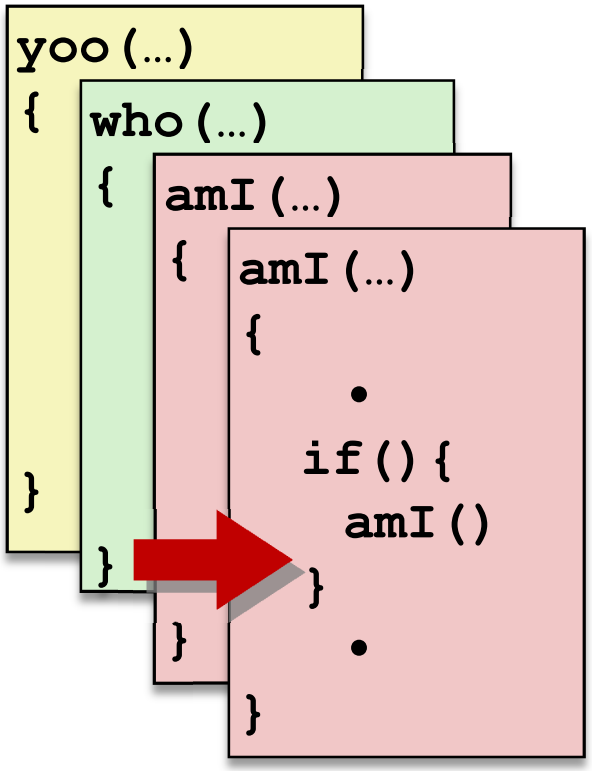
5) (another) Recursive call to amI (3)

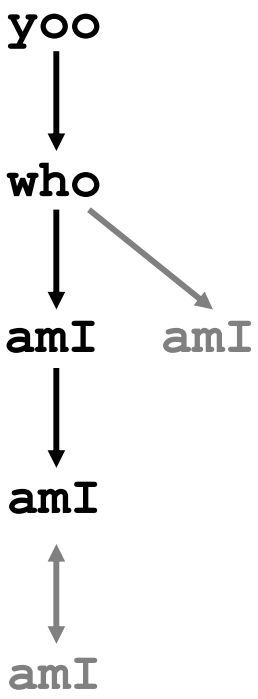


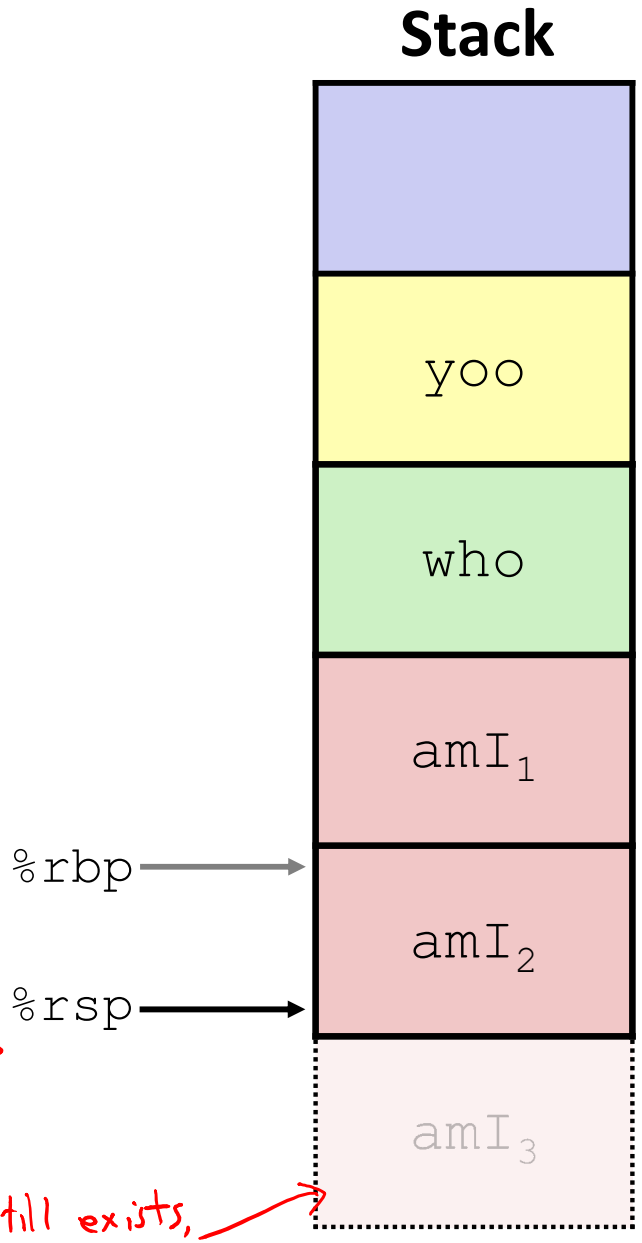




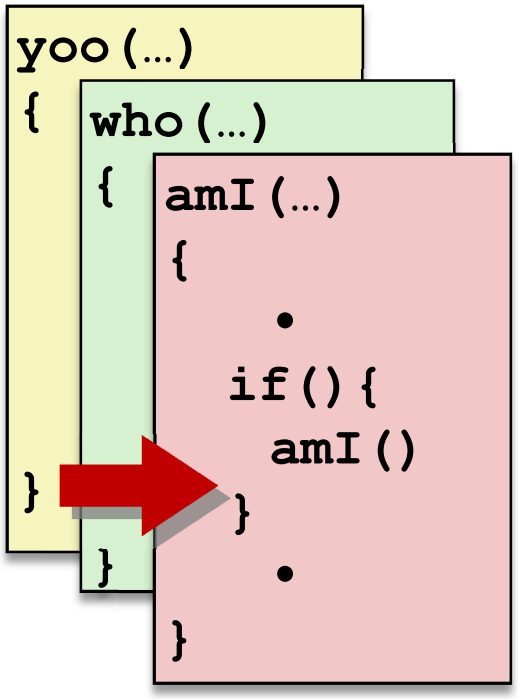
6) Return from (another) recursive call to amI

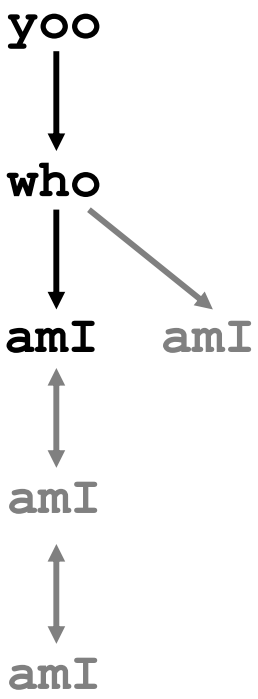


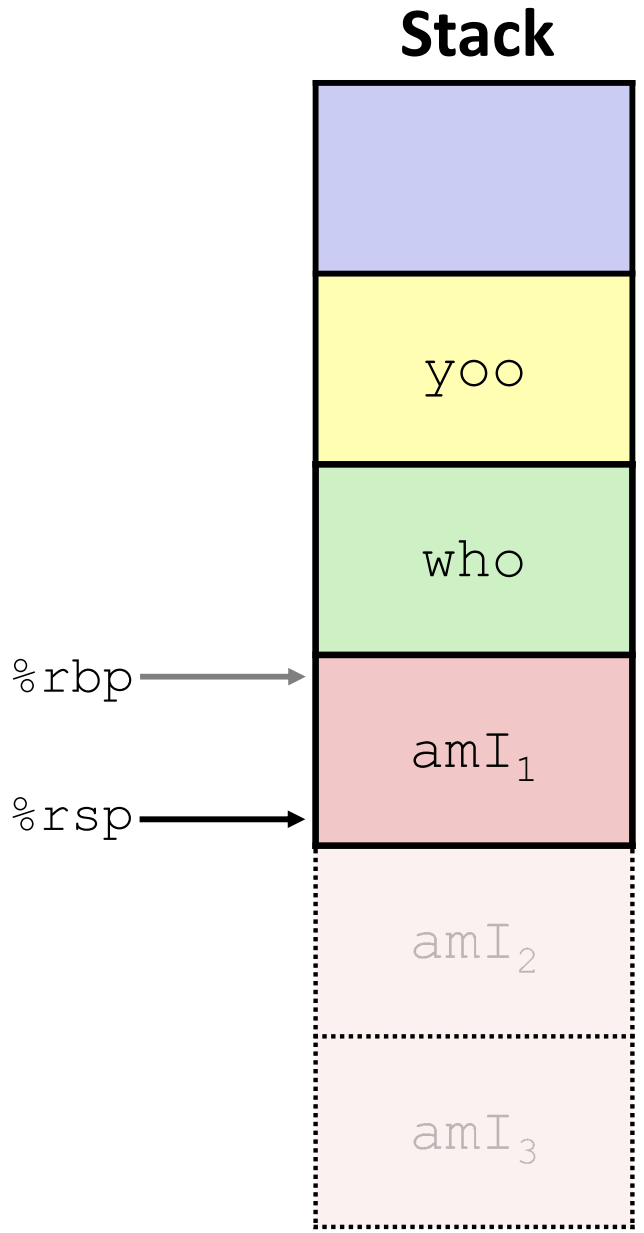




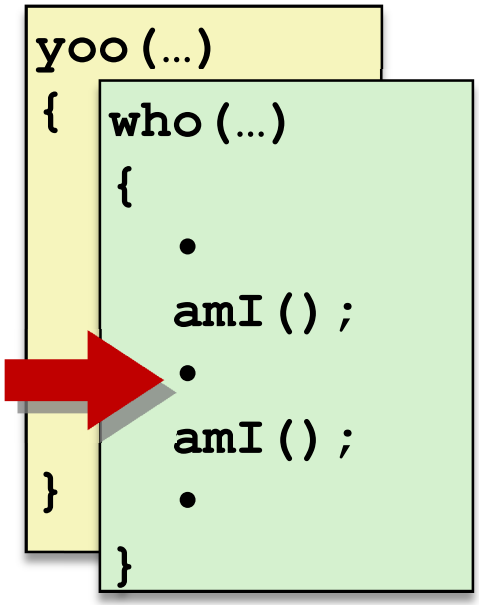
7) Return from recursive call to amI

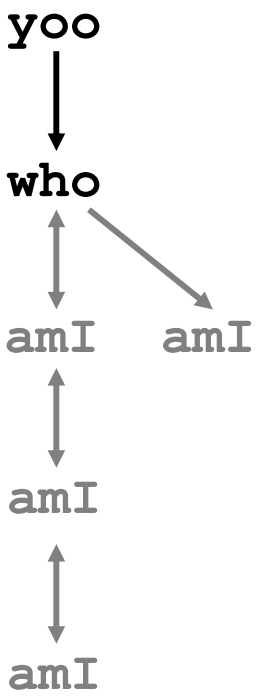


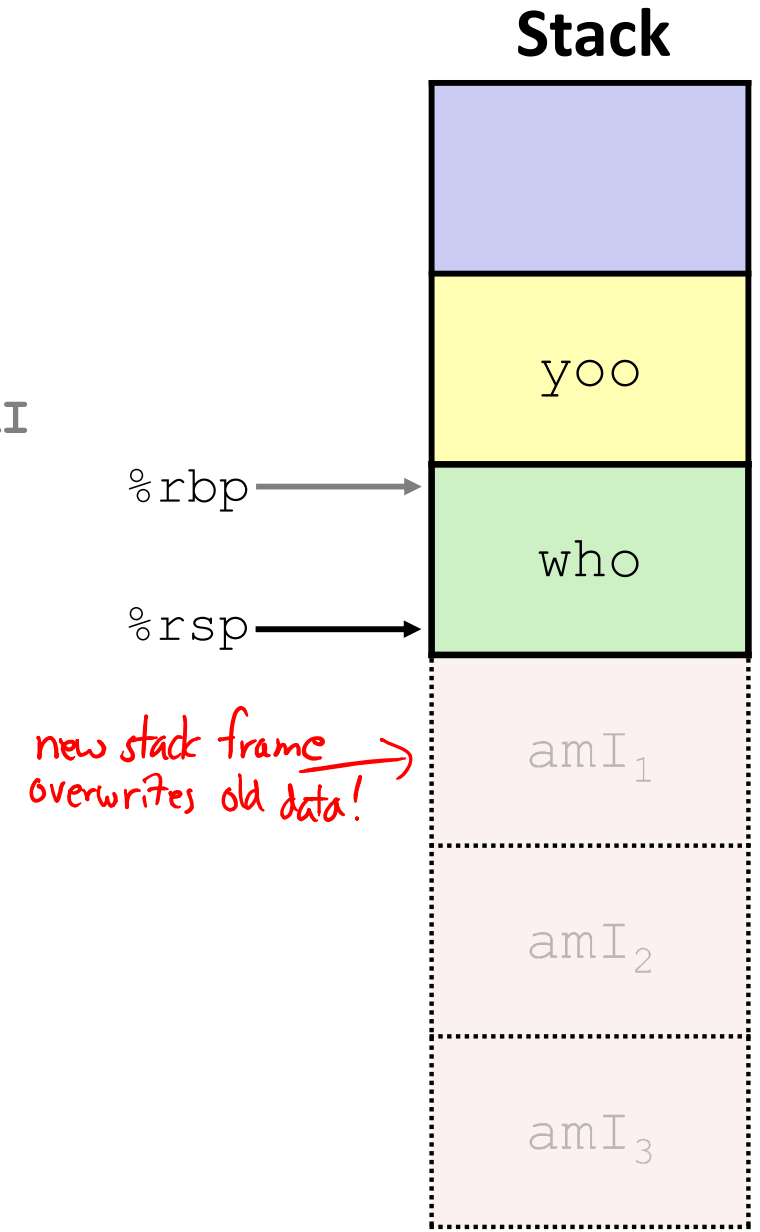




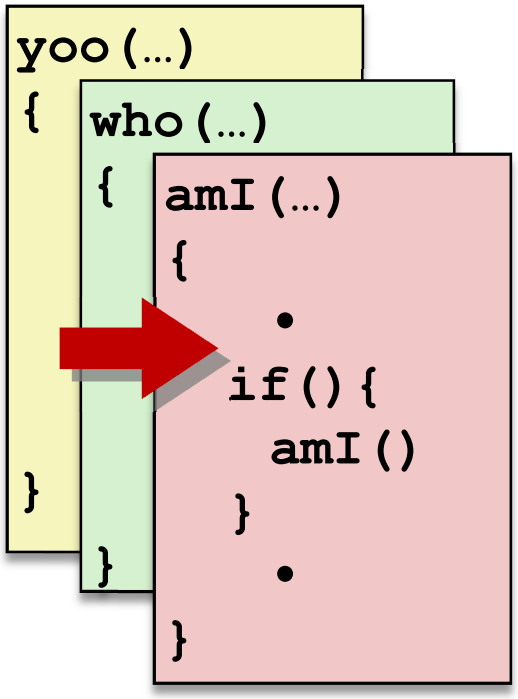
8) Return from call to amI

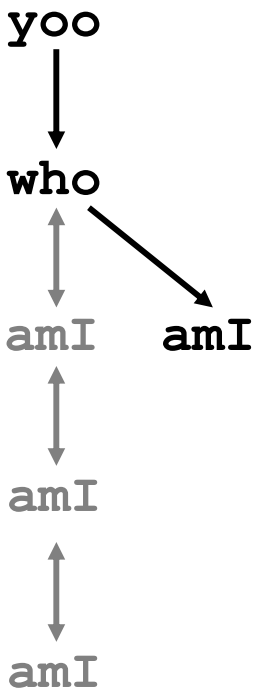


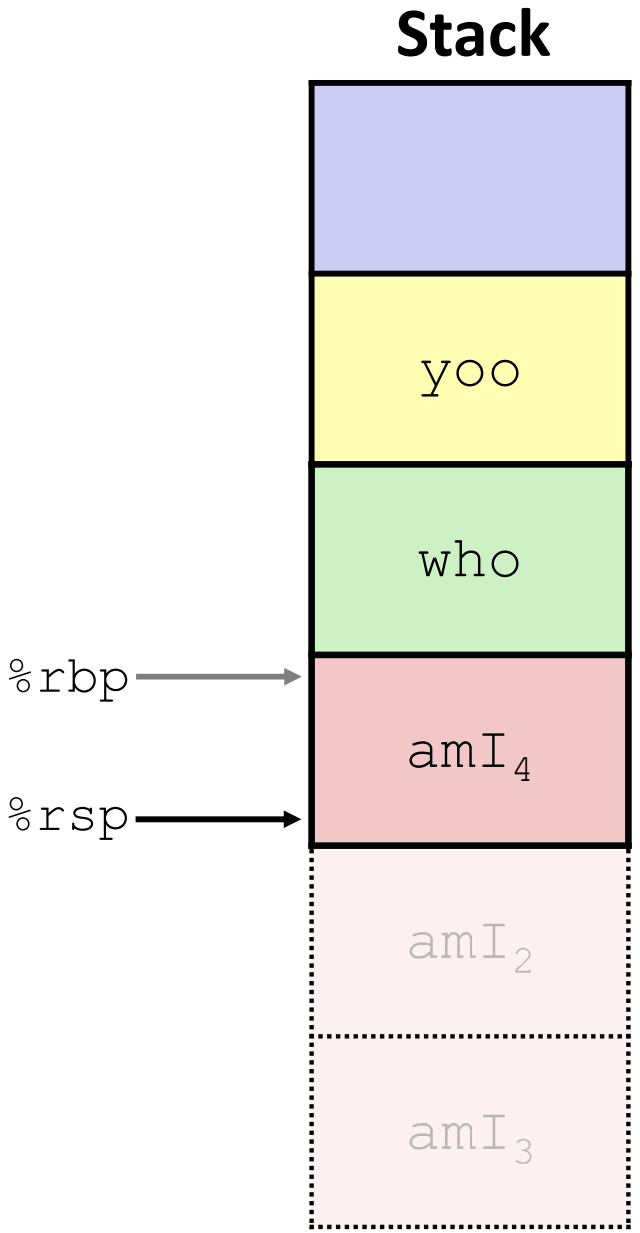




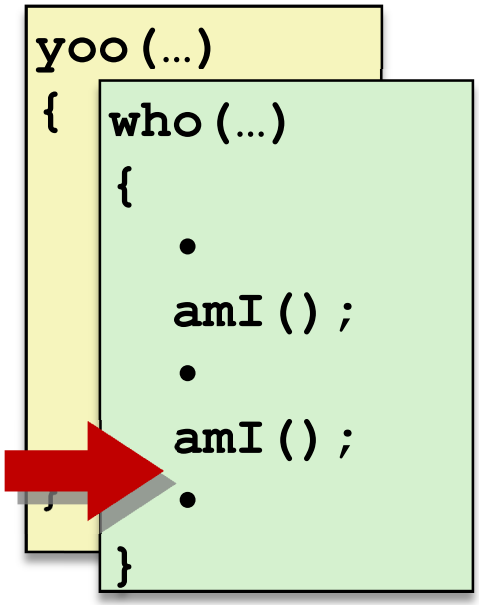
9) (second) Call to amI (4)

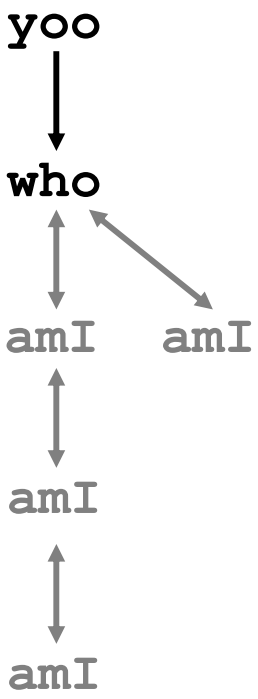


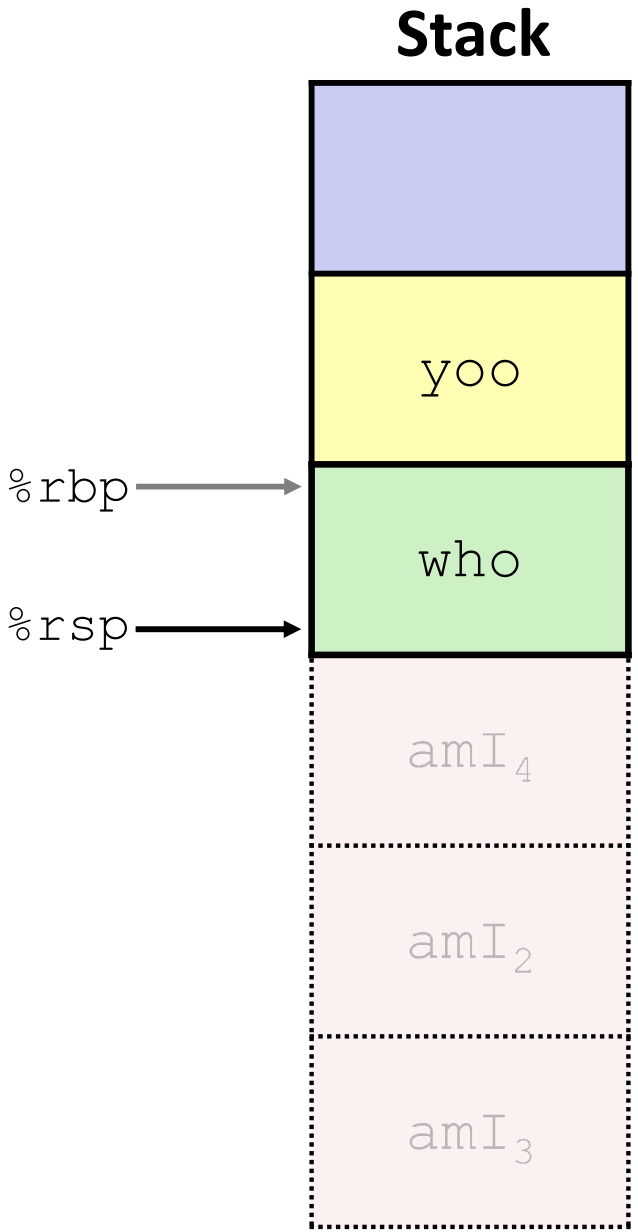




10) Return from (second) call to amI

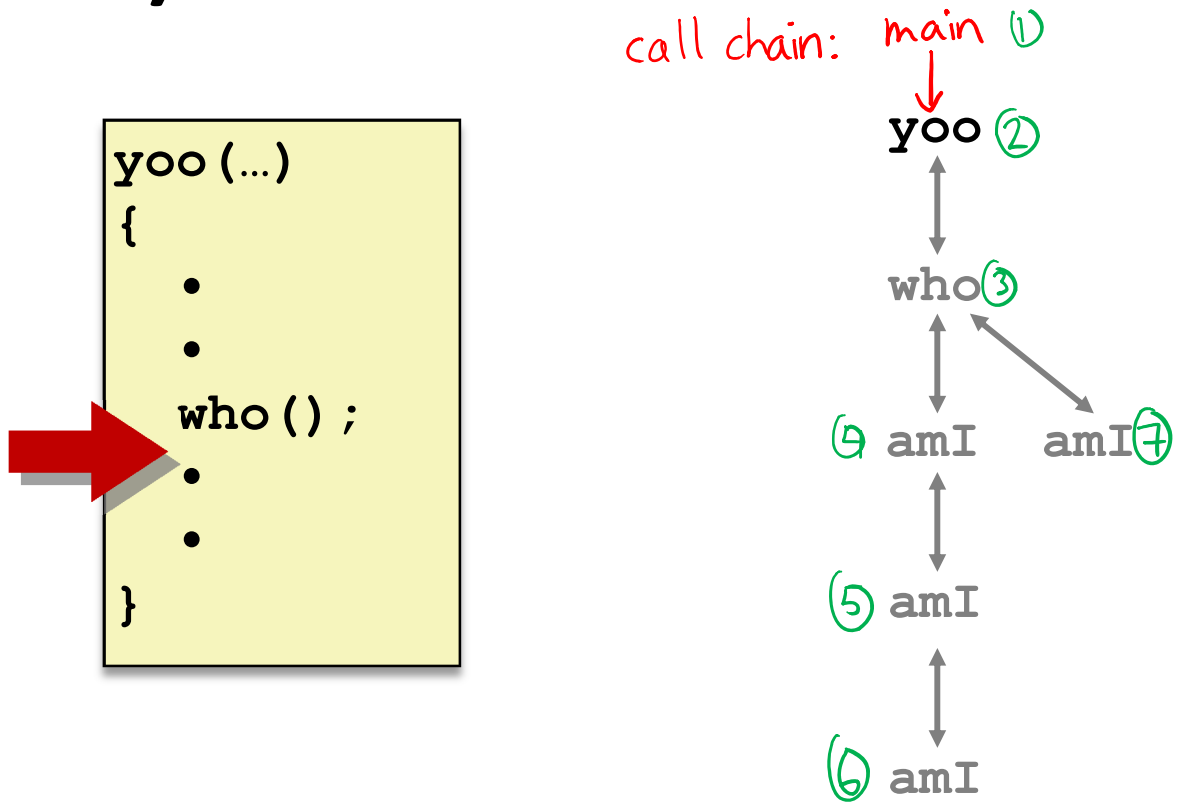


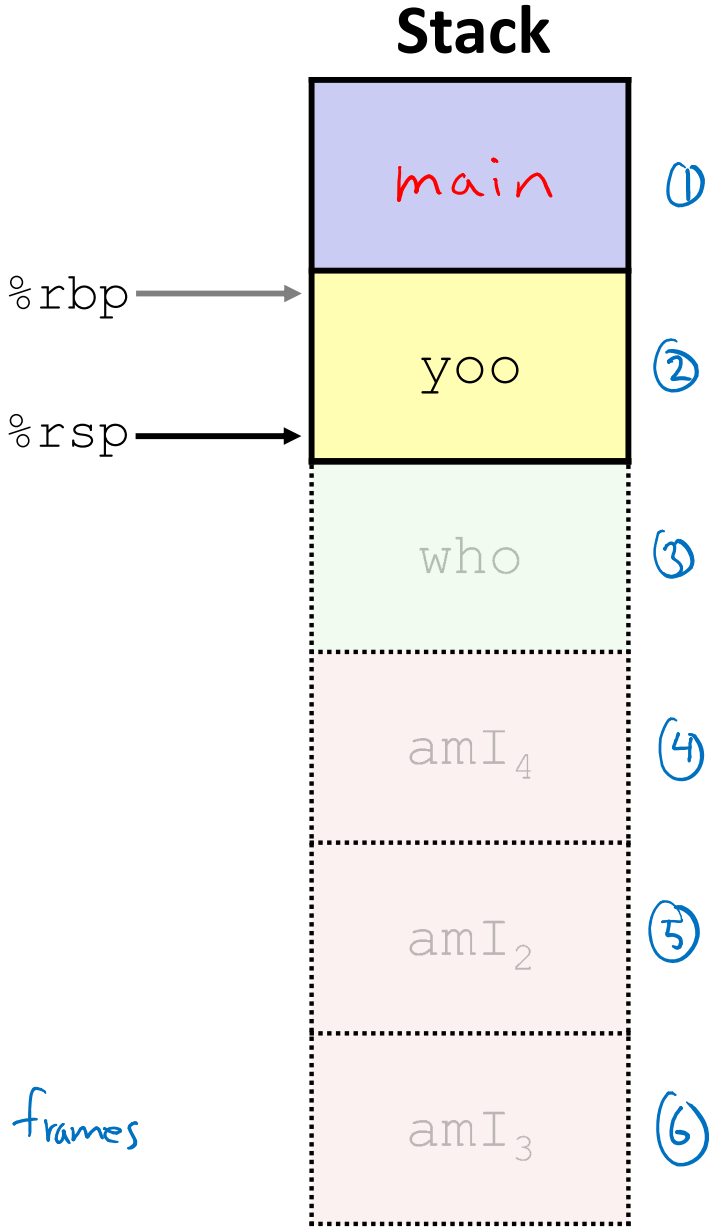




11) Return from call to who

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



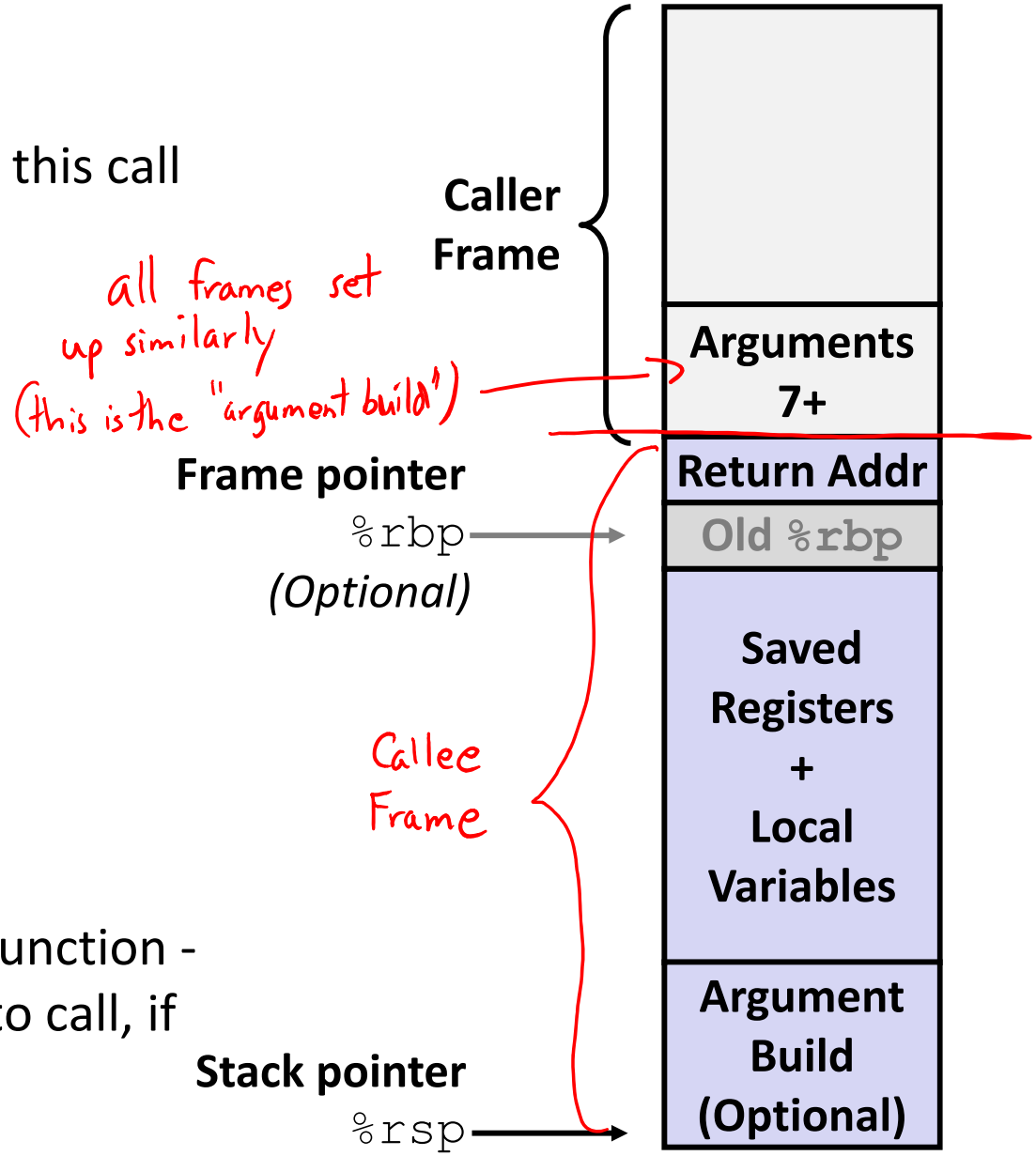


total stack frames created: 7

maximum stack depth: 6 frames

x86-64/Linux Stack Frame

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



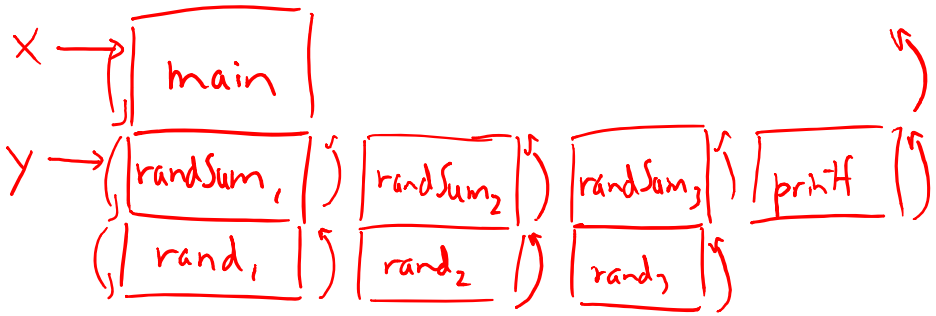
Peer Instruction Question

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):

```
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