Procedures II

CSE 351 Winter 2019

Instructors:

Max Willsey

Luis Ceze

Teaching Assistants:

Britt Henderson

Lukas Joswiak

Josie Lee

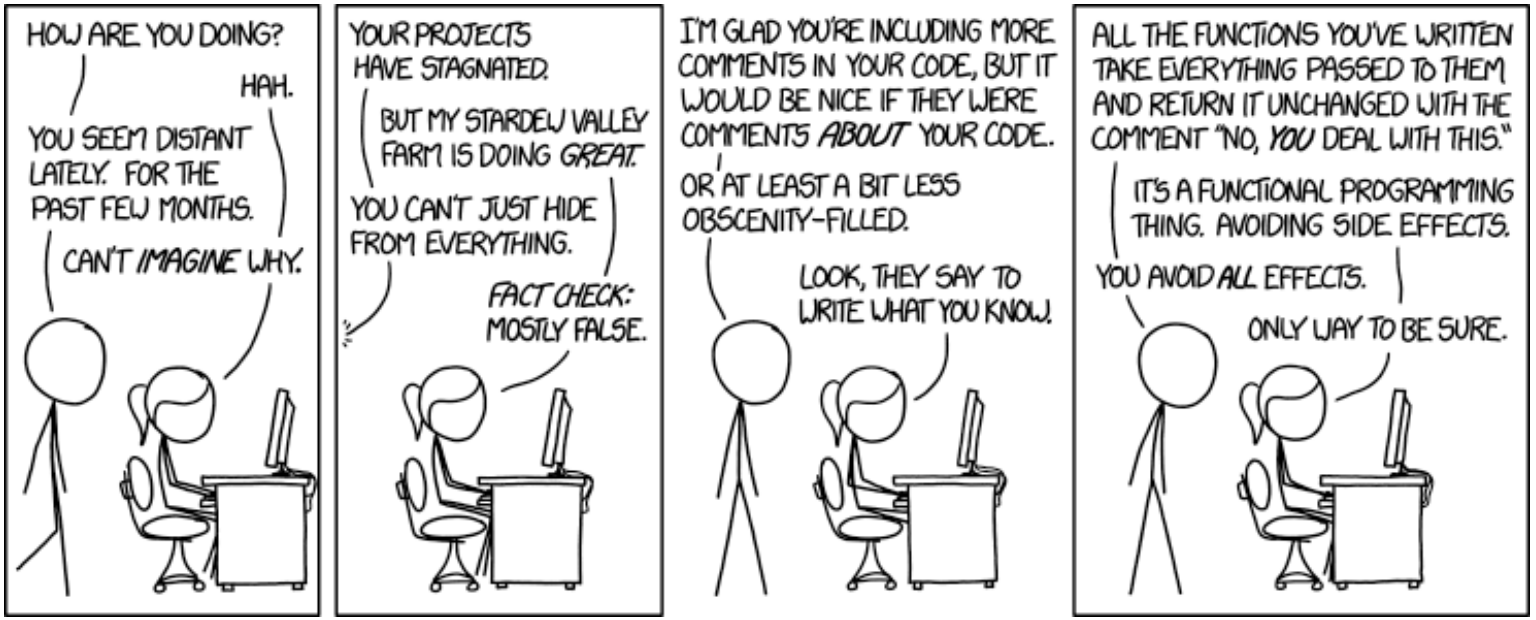
Wei Lin

Daniel Snitkovsky

Luis Vega

Kory Watson

Ivy Yu



http://xkcd.com/1790/

Administrivia

- Lab 2 due Friday (Feb 8)

- **Midterm** next Wednesday (Feb 13, in class)
  - Make a cheat sheet! – two-sided letter page, ***handwritten***
  - You get a reference sheet (see website)
  - Check Piazza this week for announcements
  - **Review session** this week in section

Code Example (Preview)

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

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

```
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
```

```
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`
  1) Push return address on stack (*why? which address?*)
  2) Jump to ***label***

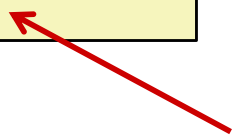
Procedure Control Flow

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

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

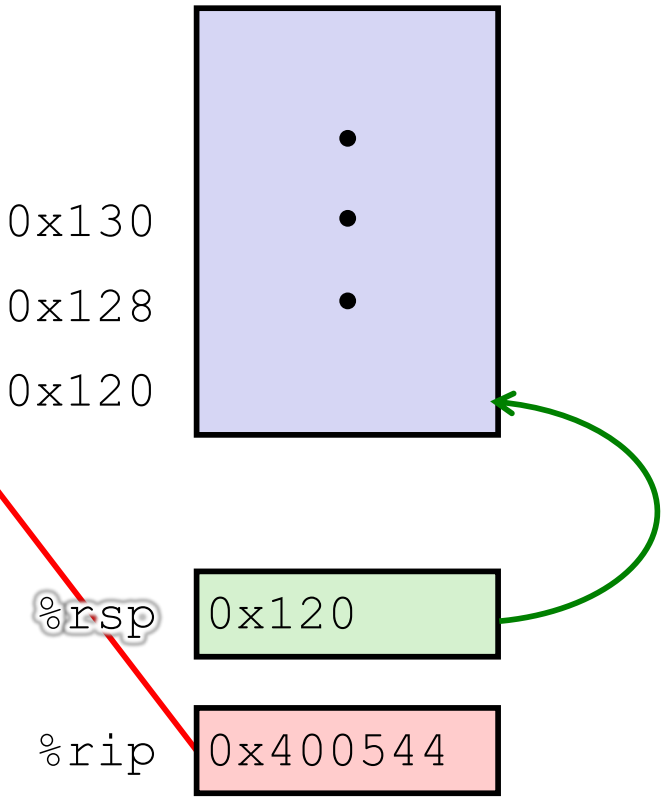


next instruction happens to be a move, but could be anything

Procedure Call Example (step 1)

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

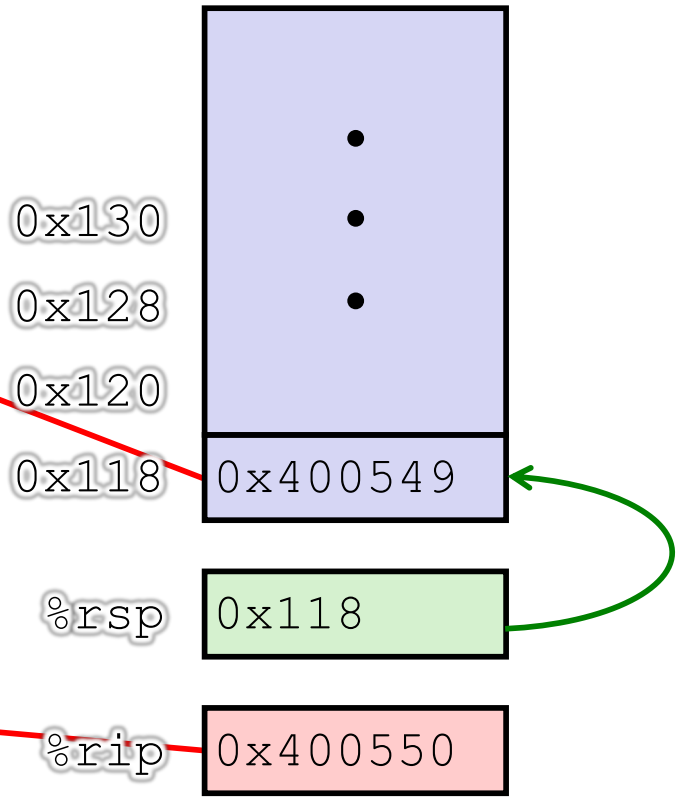
```
0000000000400550 <mult2>:
   400550:  movq   %rdi,%rax
   •
   •
   400557:  ret
```



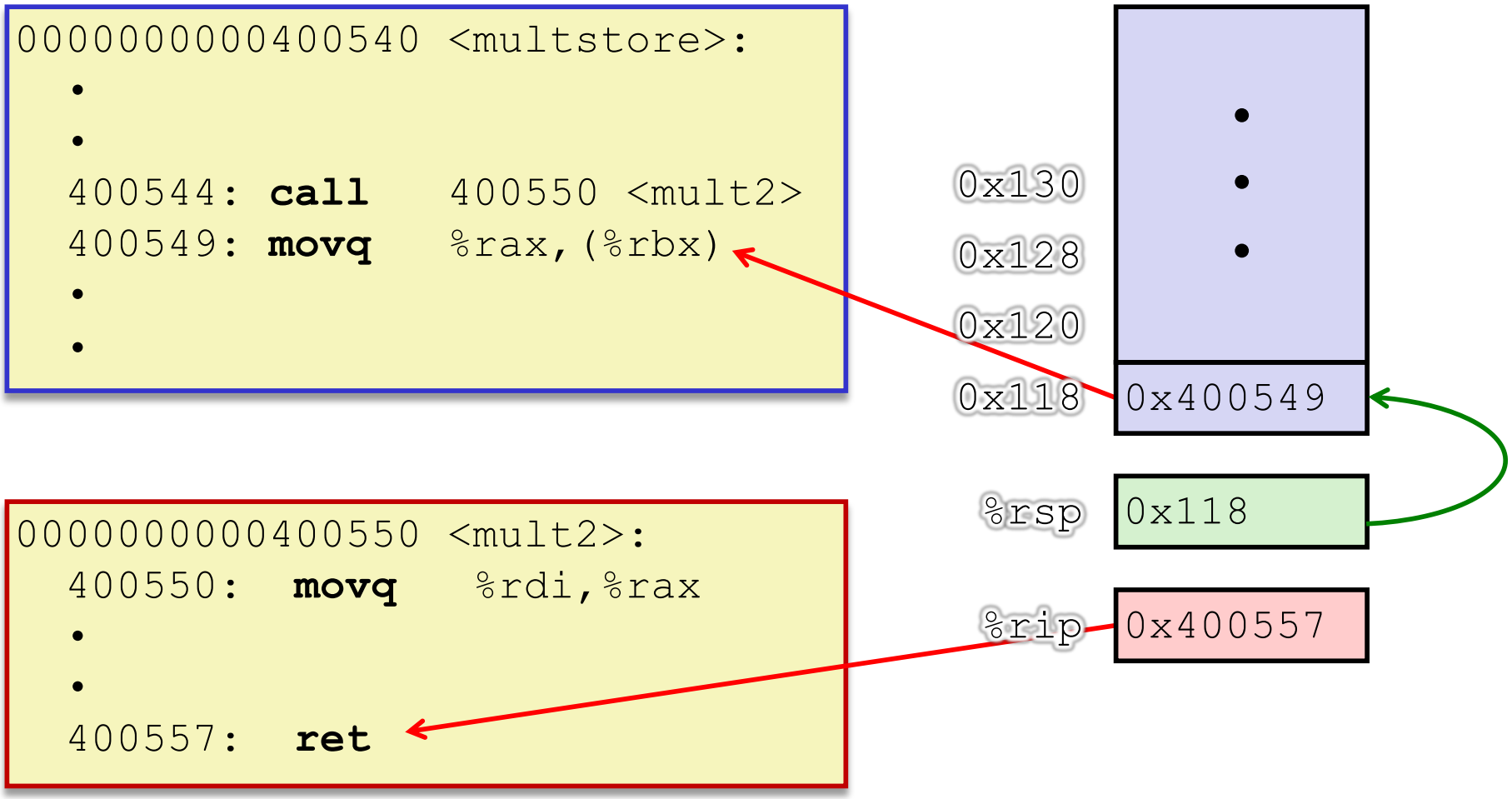
Procedure Call Example (step 2)

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

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



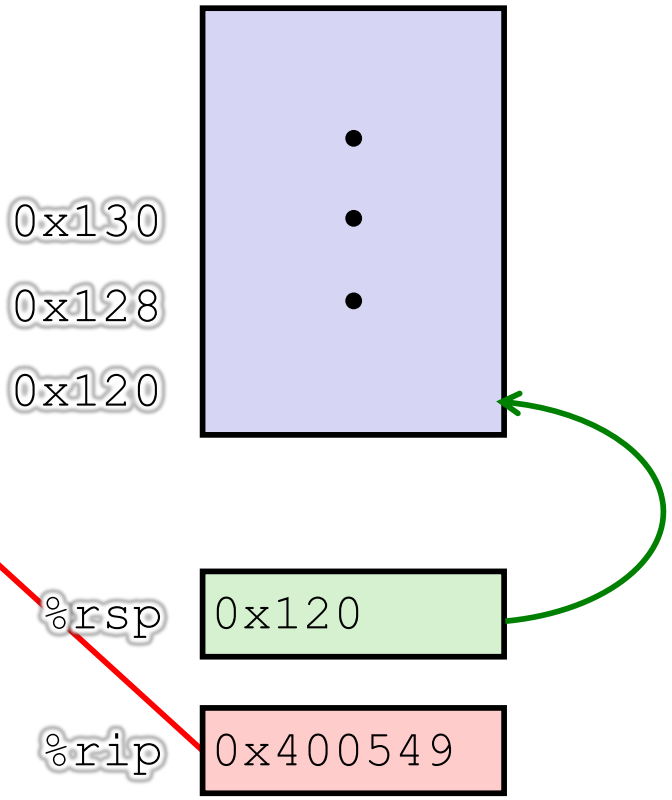
Procedure Return Example (step 1)



Procedure Return Example (step 2)

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

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



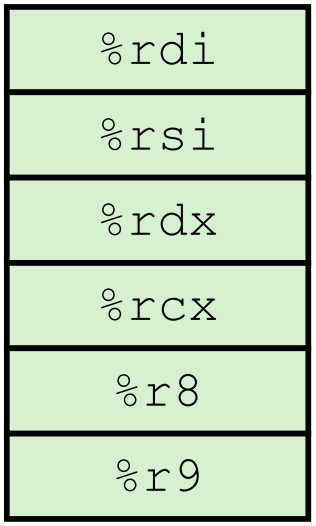
Procedures

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

Procedure Data Flow

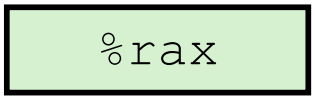
Registers (NOT in Memory)

- First 6 arguments

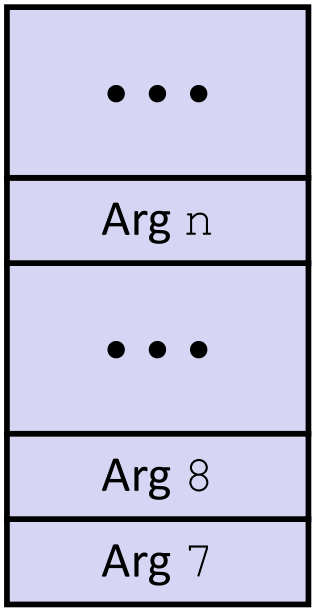


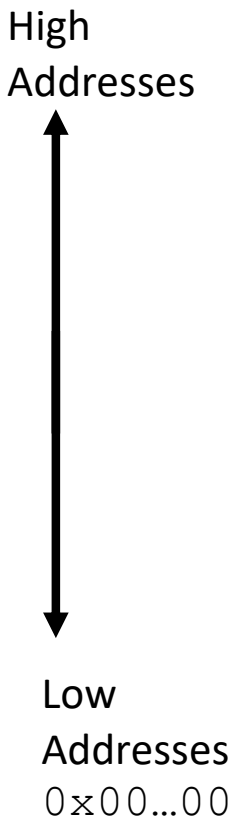
Diane's Silk Dress Costs $8 9

- 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;
}
```

```
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
    • • •
```

```
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

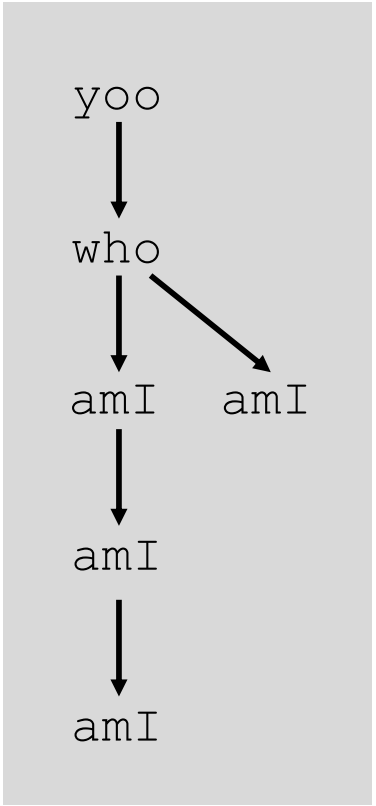
Call Chain Example

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

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

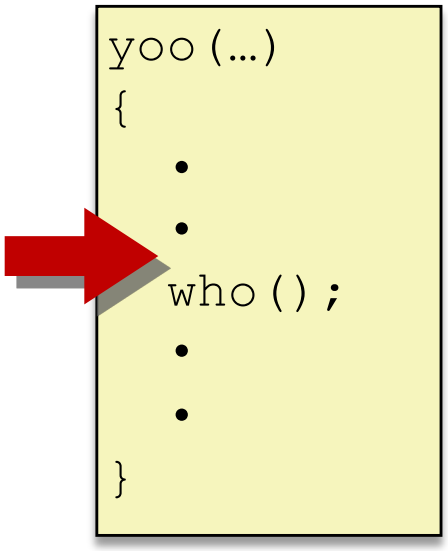
```
amI(…)
{
    •
  if(…){
    amI()
  }
    •
}
```

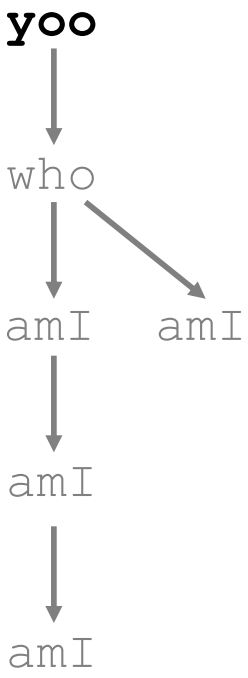
Example
Call Chain

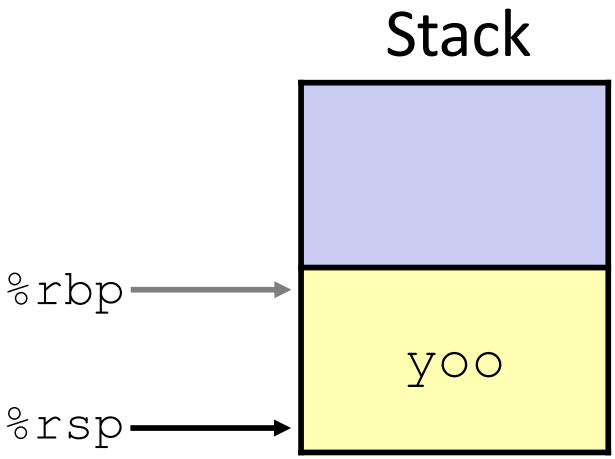


Procedure `amI` is recursive
(calls itself)

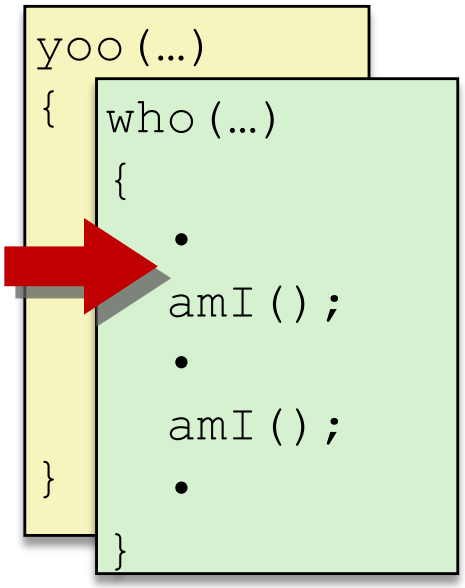
1) Call to yoo

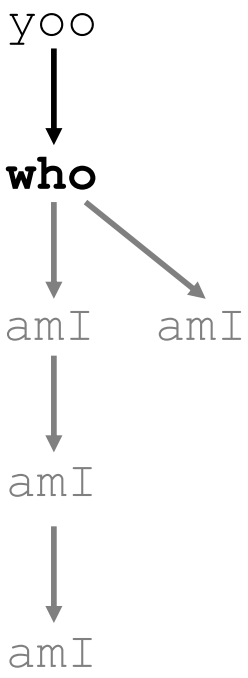


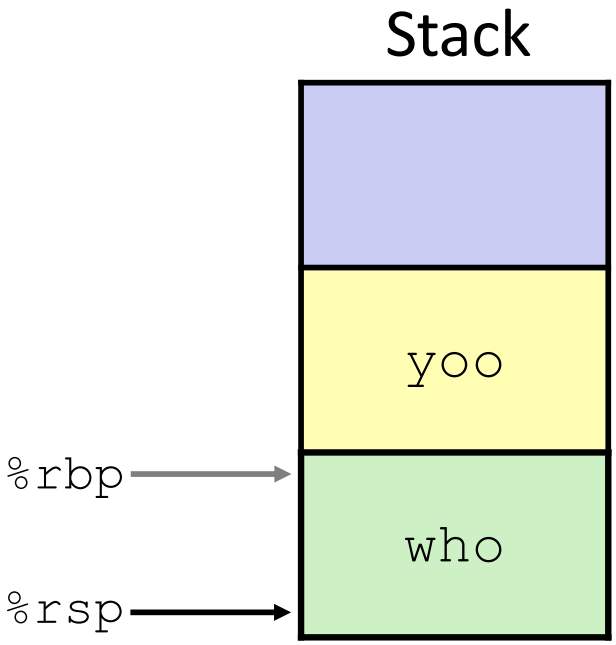




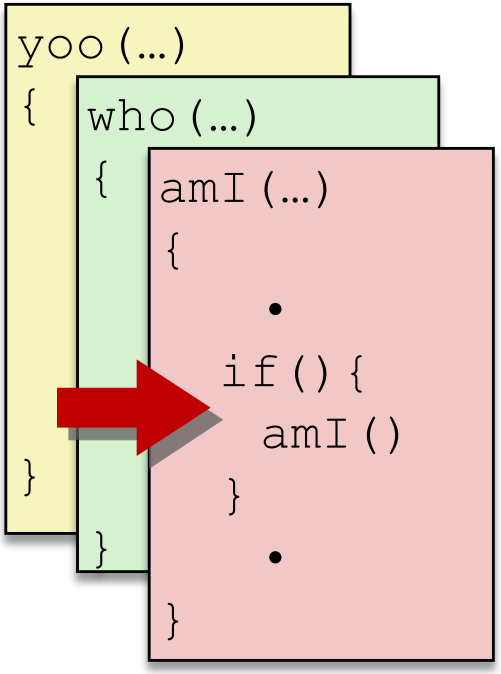
2) Call to who

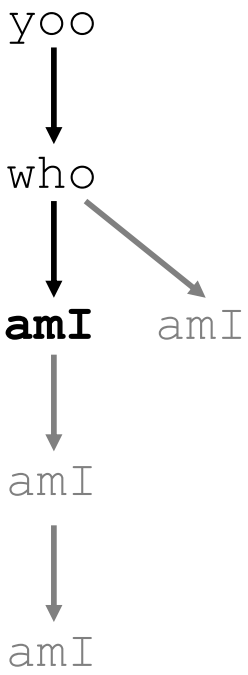


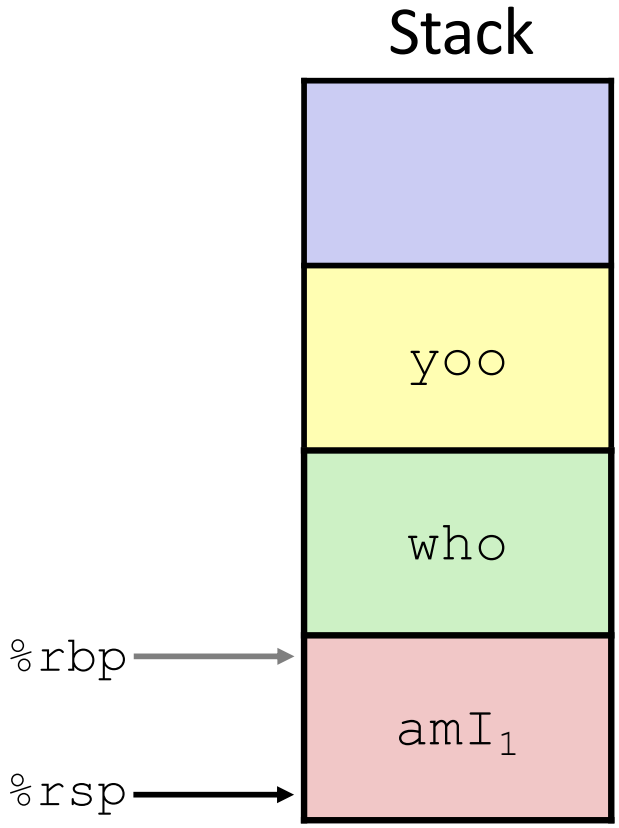




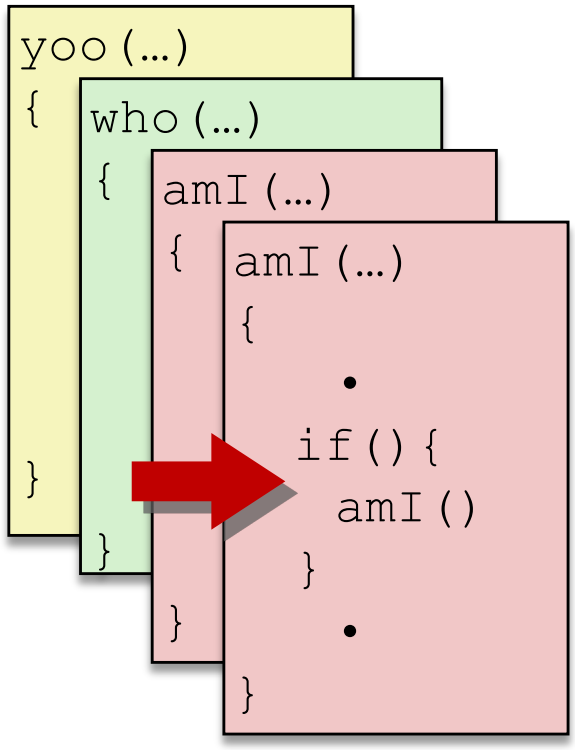
3) Call to amI (1)

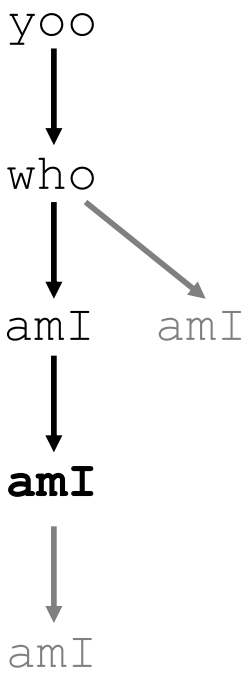


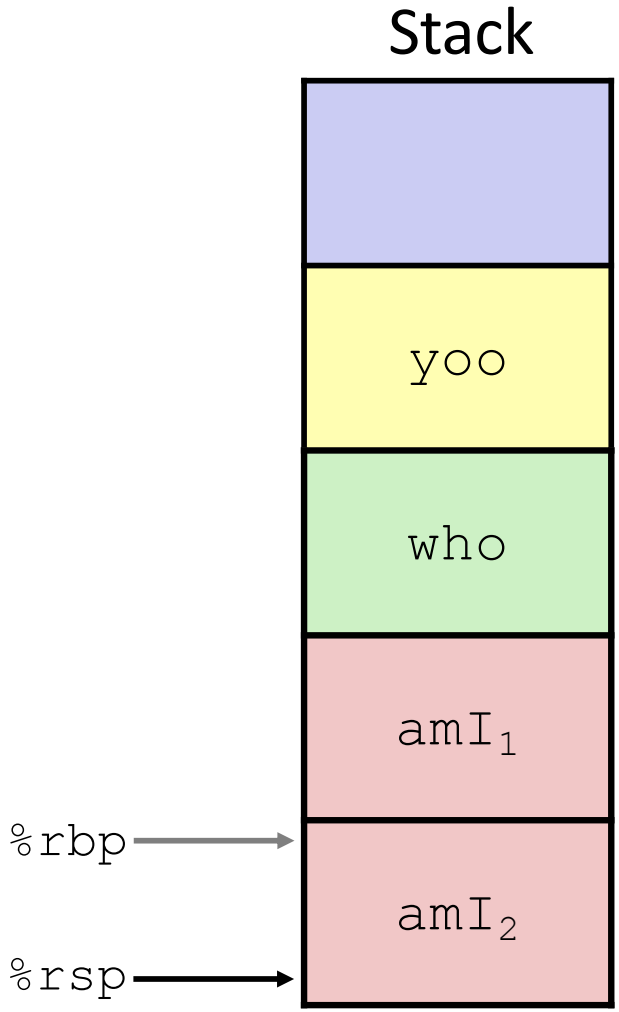




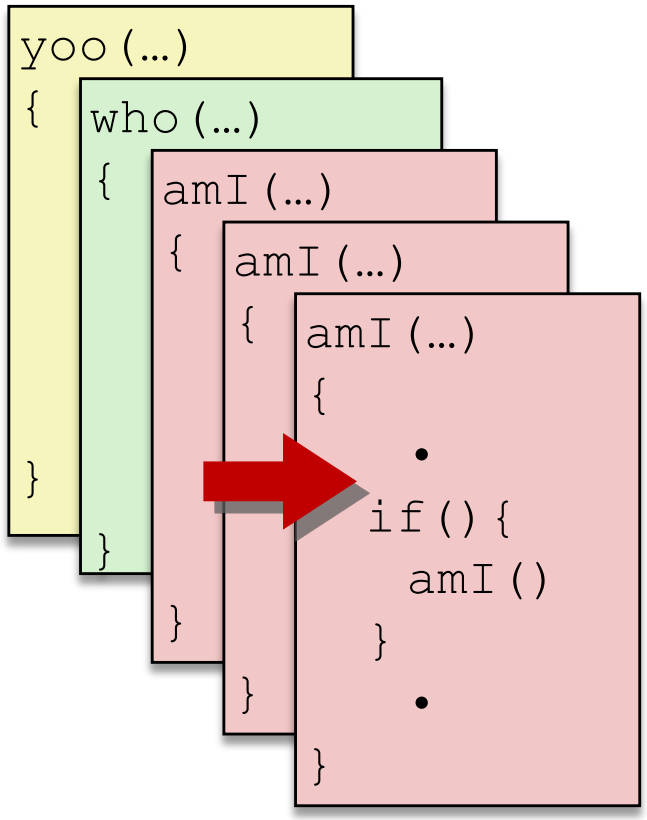
4) Recursive call to `amI` (2)

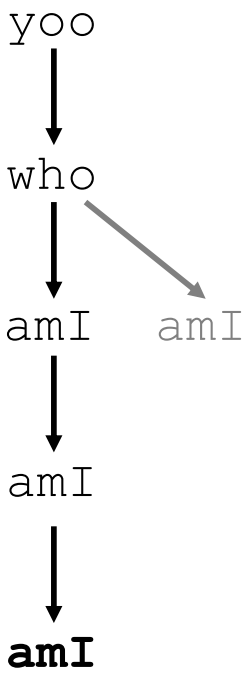


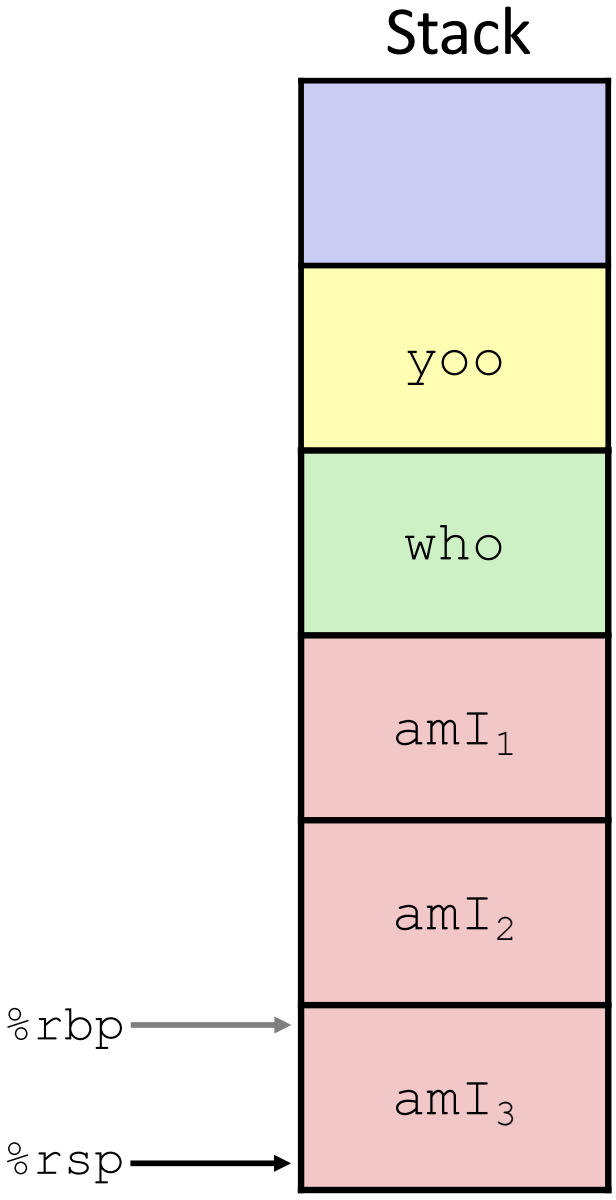




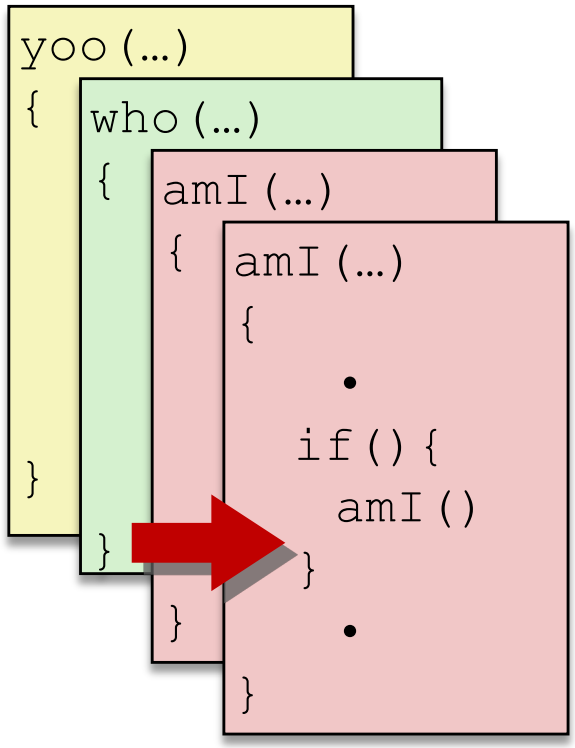
5) (another) Recursive call to `amI` (3)

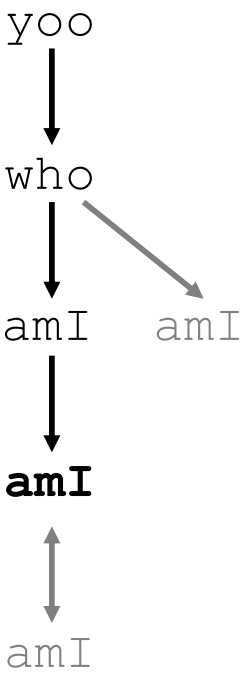


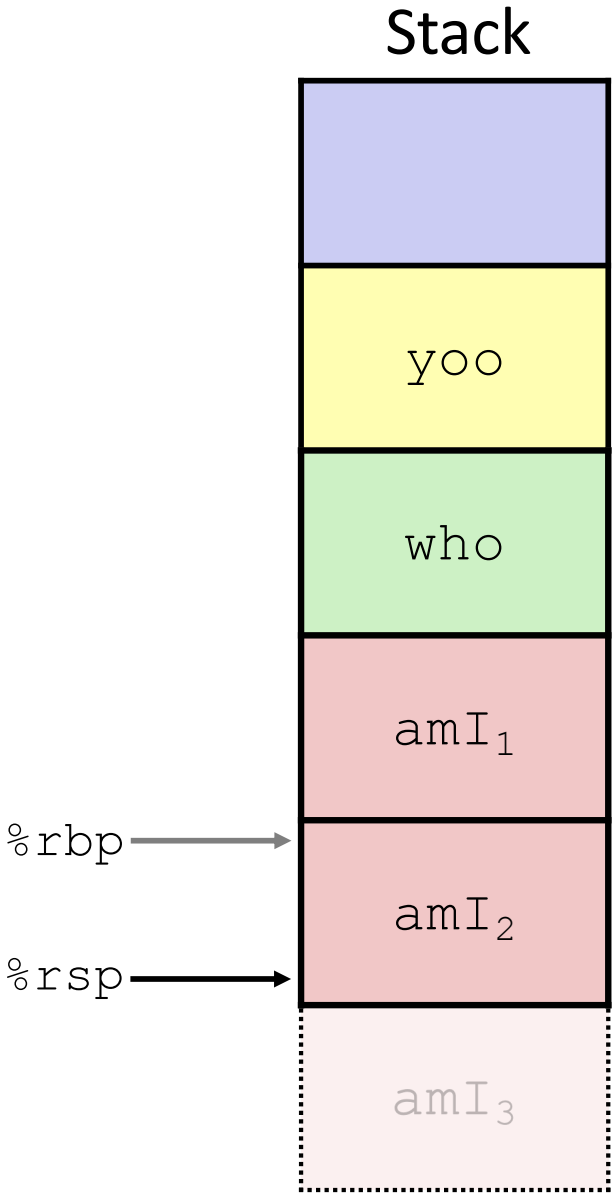




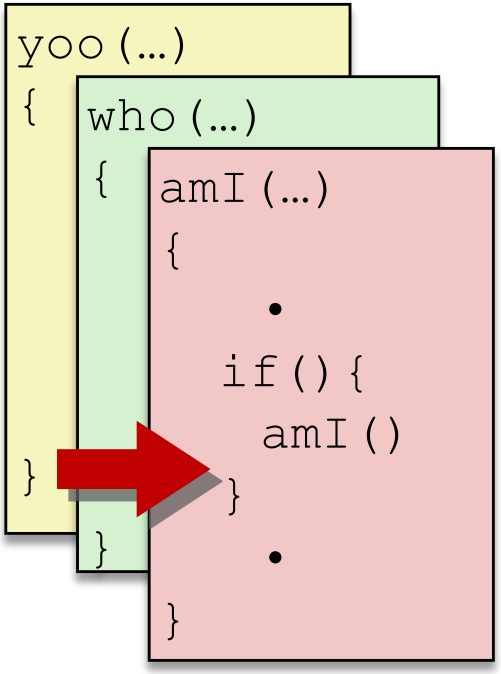
6) Return from (another) recursive call to amI

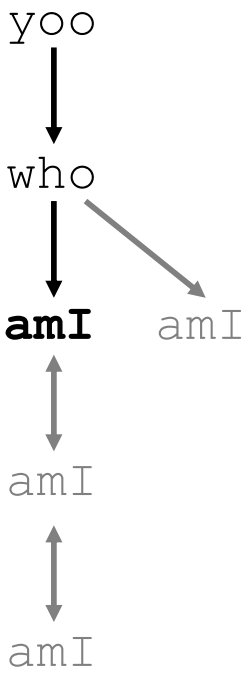


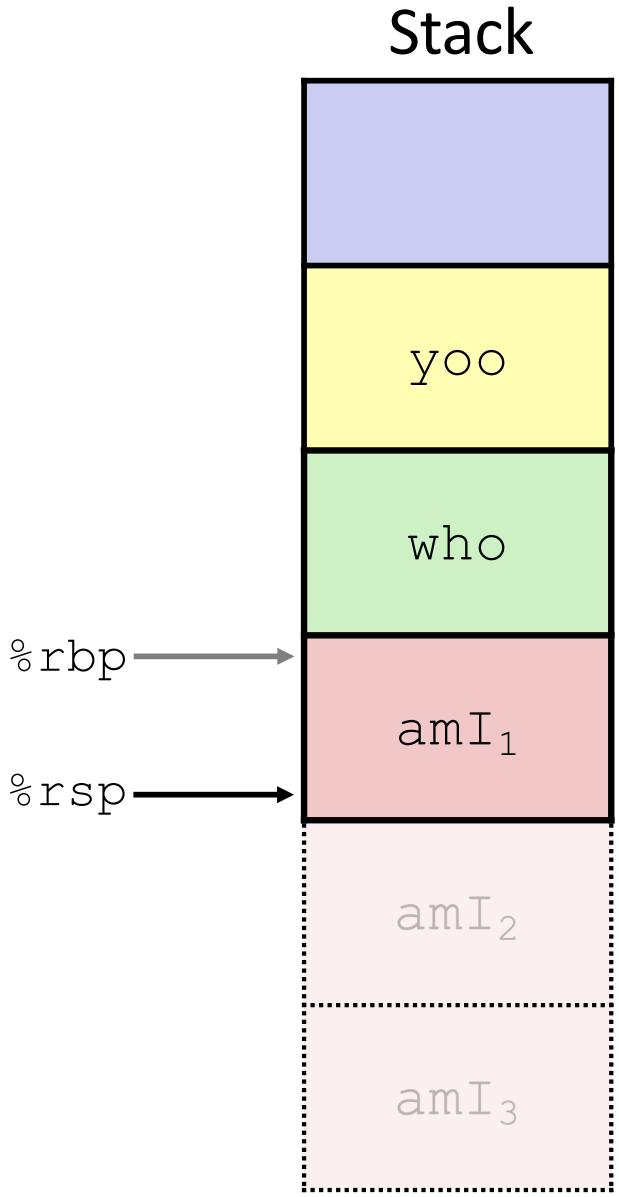




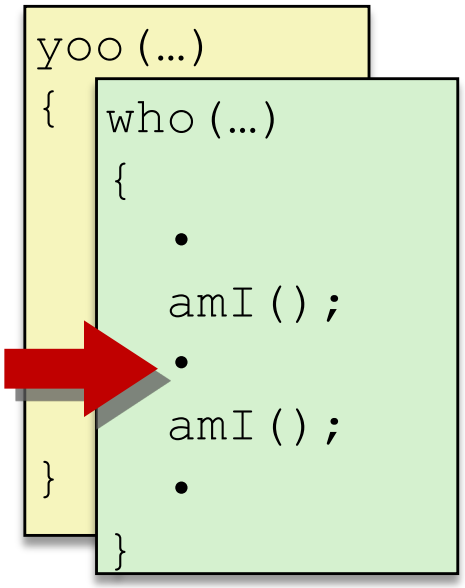
7) Return from recursive call to amI

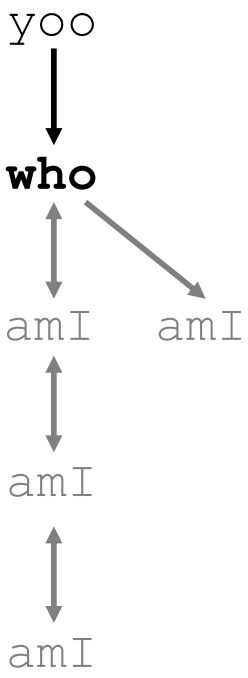


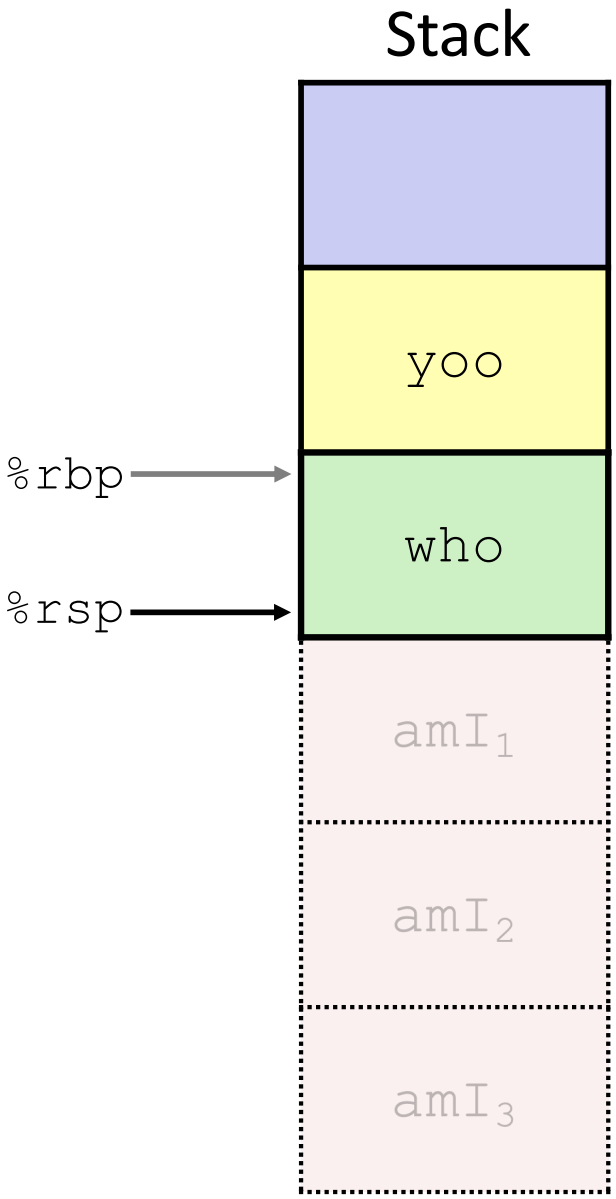




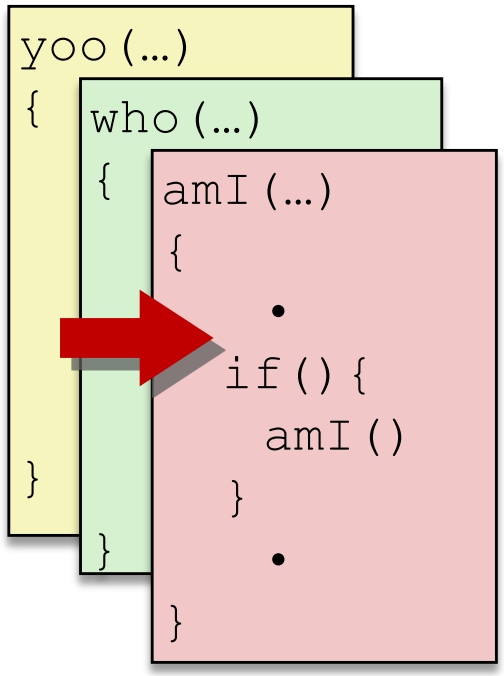
8) Return from call to amI

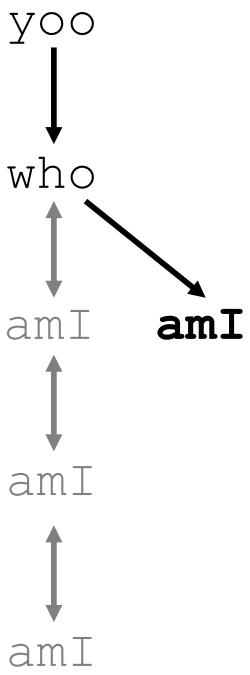


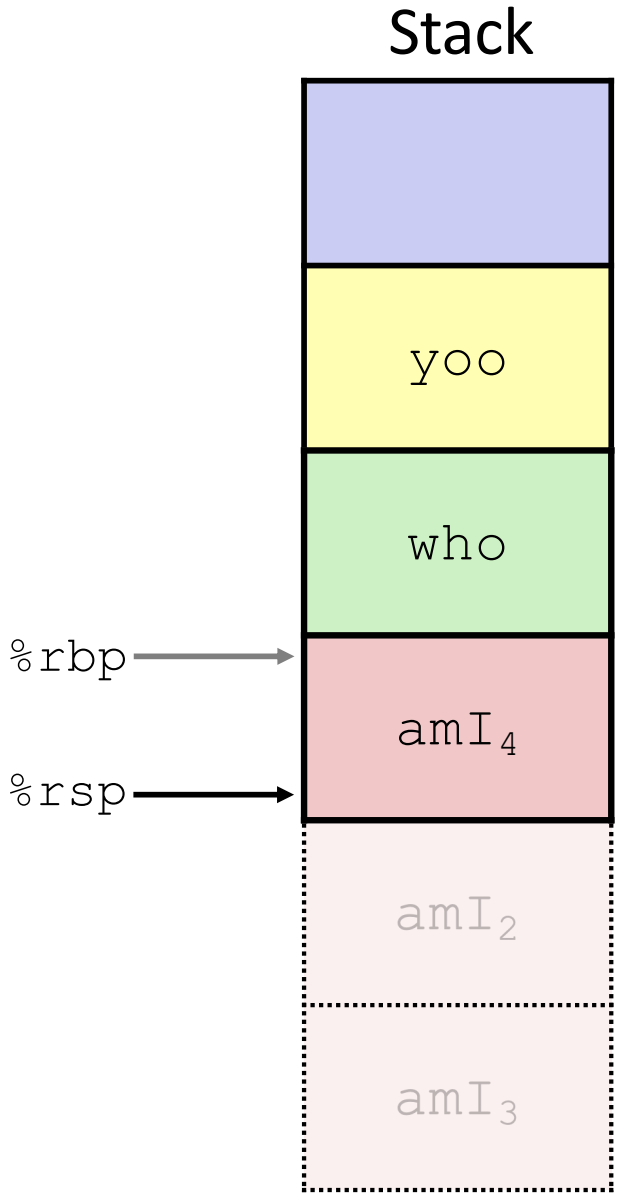




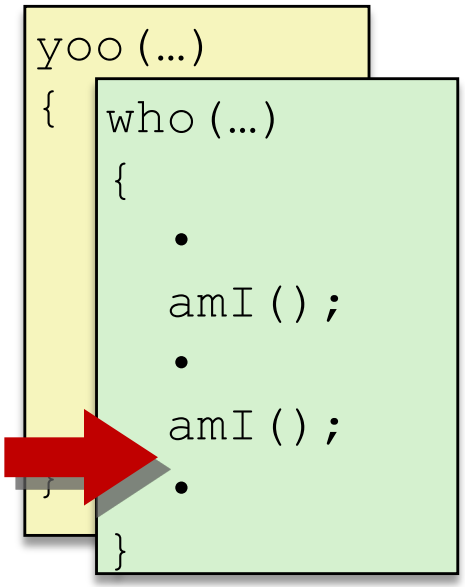
9) (second) Call to amI (4)

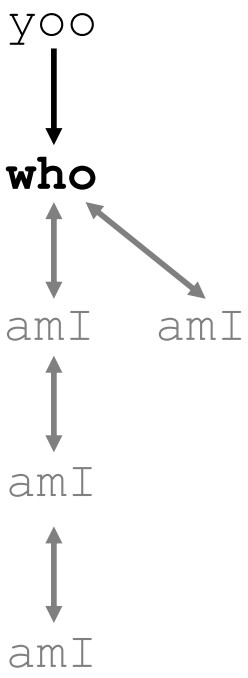


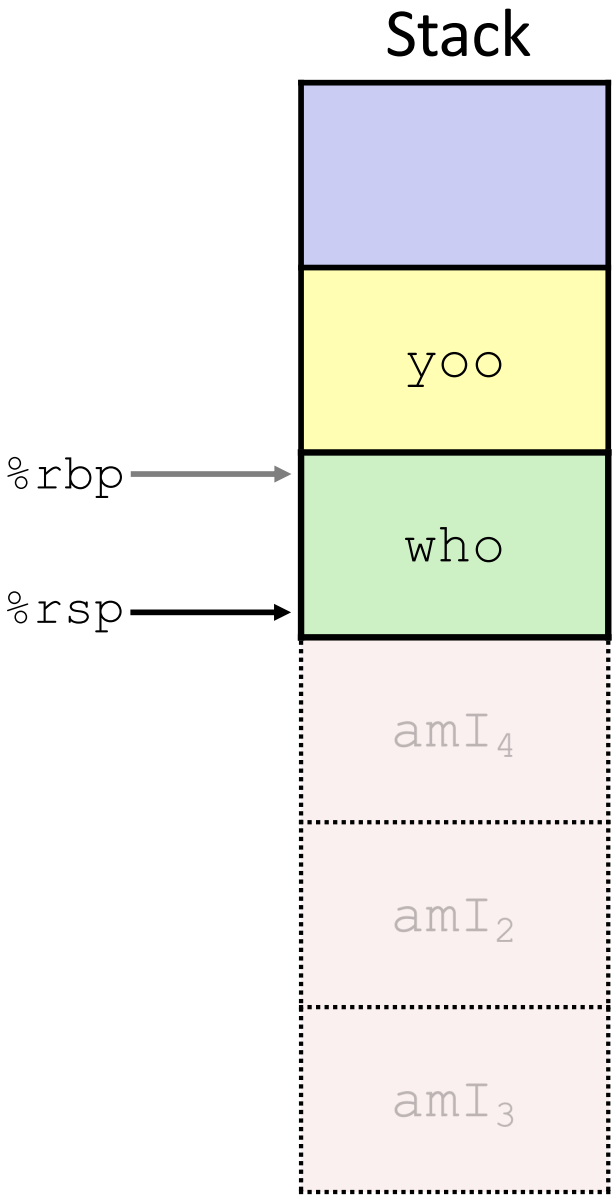




10) Return from (second) call to amI

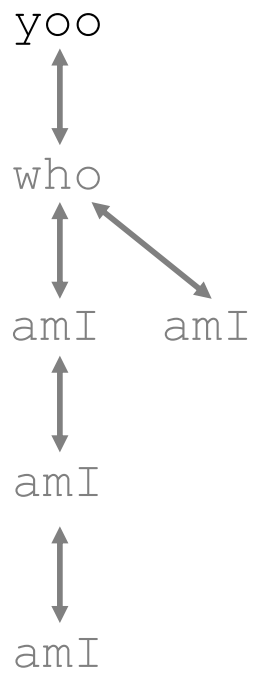


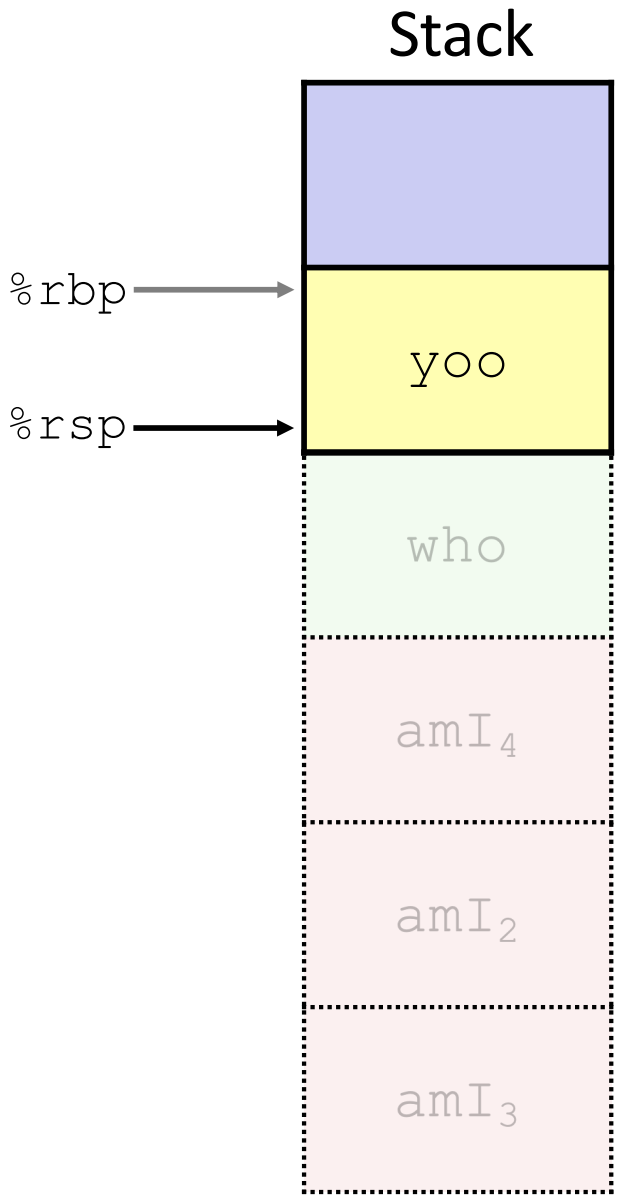




11) Return from call to who

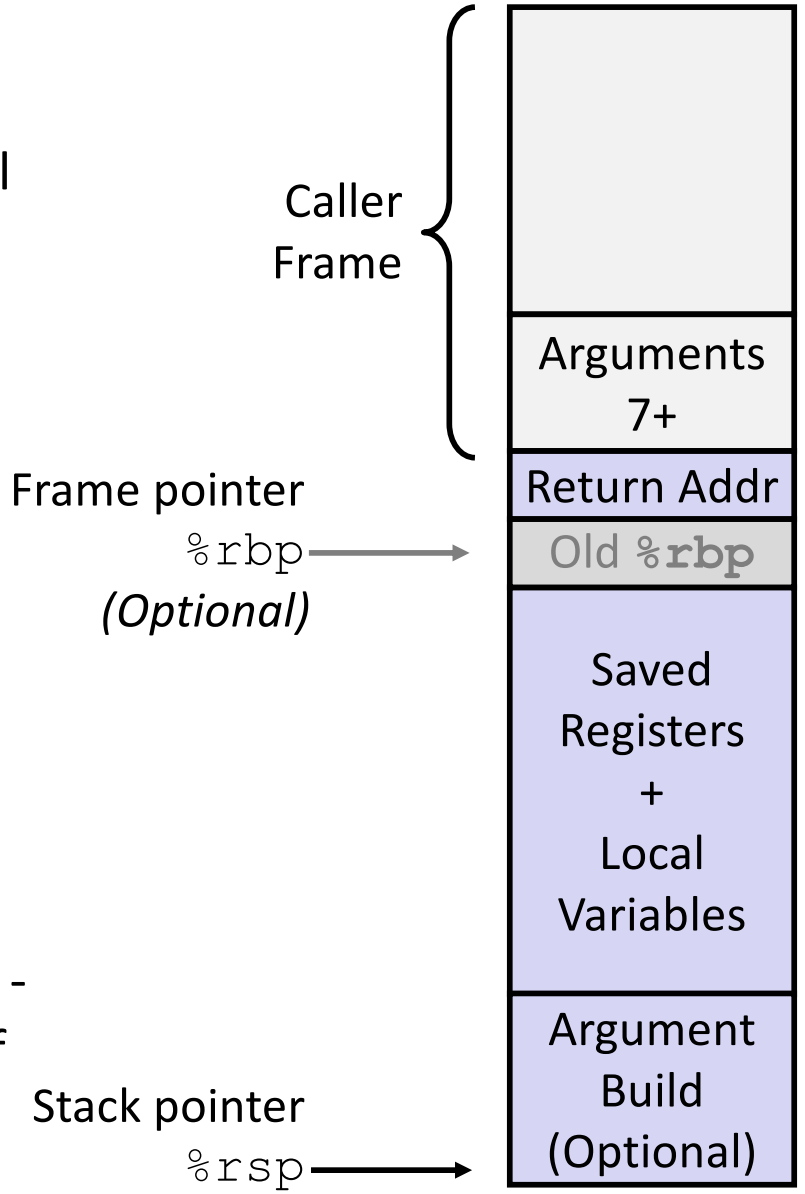
```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```





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

- 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*?
- What is the maximum *depth* (# of frames) of the Stack?

A. 1 B. 2 C. 3 D. 4

Procedures

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

Register Saving Conventions

- When procedure `yoo` calls `who`:
  - `yoo` is the caller
  - `who` is the callee
- Can registers be used for temporary storage?

```
yoo:
    • • •
    movq $15213, %rdx
    call who
    addq %rdx, %rax
    • • •
    ret
```

```
who:
    • • •
    subq $18213, %rdx
    • • •
    ret
```

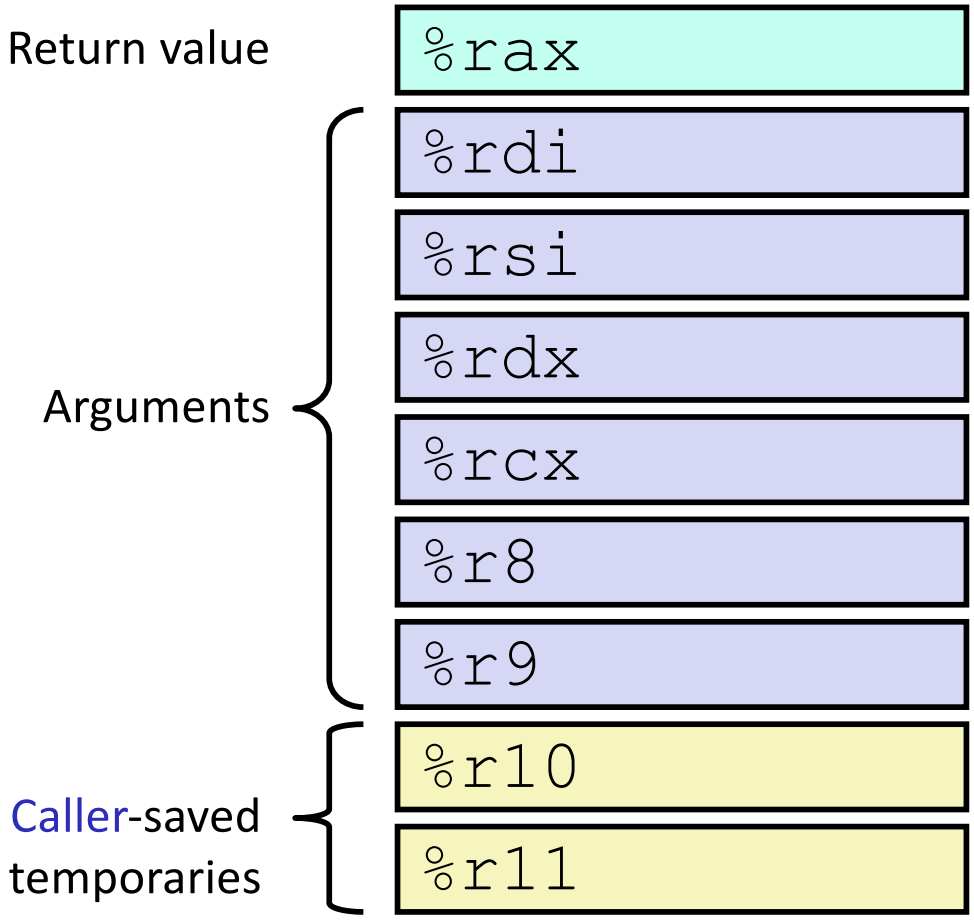
  - No! Contents of register `%rdx` overwritten by `who`!
  - This could be trouble – something should be done. Either:
    - Caller should save `%rdx` before the call (and restore it after the call)
    - Callee should save `%rdx` before using it (and restore it before returning)

Register Saving Conventions

- ***"Caller-saved" registers***
  - It is the **caller**'s responsibility to save any important data in these registers before calling another procedure (*i.e.* the **callee** can freely change data in these registers)
  - **Caller** saves values in its stack frame before calling **Callee**, then restores values after the call
- ***"Callee-saved" registers***
  - It is the callee's responsibility to save any data in these registers before using the registers (*i.e.* the **caller** assumes the data will be the same across the **callee** procedure call)
  - **Callee** saves values in its stack frame before using, then restores them before returning to **caller**

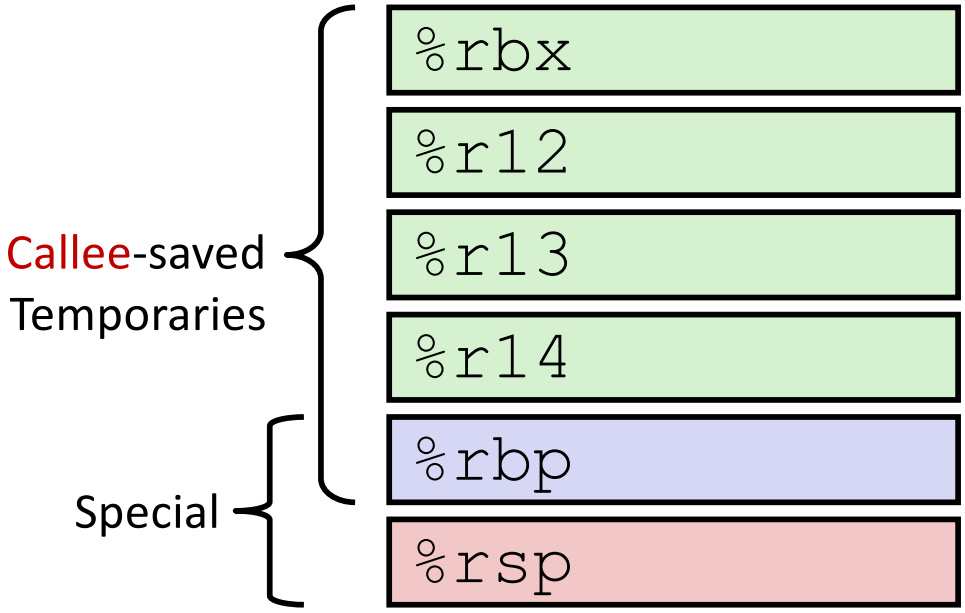
x86-64 Linux Register Usage, part 1

- **`%rax`**
  - Return value
  - Also **caller**-saved & restored
  - Can be modified by procedure
- **`%rdi, ..., %r9`**
  - Arguments
  - Also **caller**-saved & restored
  - Can be modified by procedure
- **`%r10, %r11`**
  - **Caller**-saved & restored
  - Can be modified by procedure



x86-64 Linux Register Usage, part 2

- **`%rbx, %r12, %r13, %r14`**
  - **Callee**-saved
  - **Callee** must save & restore
- **`%rbp`**
  - **Callee**-saved
  - **Callee** must save & restore
  - May be used as frame pointer to aid in debugging
- **`%rsp`**
  - Special form of **callee** save
  - Restored to original value upon exit from procedure



x86-64 64-bit Registers: Usage Conventions

%rax	Return value - Caller saved
%rbx	Callee saved
%rcx	Argument #4 - Caller saved
%rdx	Argument #3 - Caller saved
%rsi	Argument #2 - Caller saved
%rdi	Argument #1 - Caller saved
%rsp	Stack pointer
%rbp	Callee saved

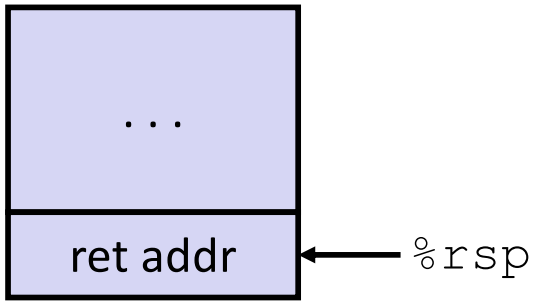
%r8	Argument #5 - Caller saved
%r9	Argument #6 - Caller saved
%r10	Caller saved
%r11	Caller Saved
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

Callee-Saved Example (step 1)

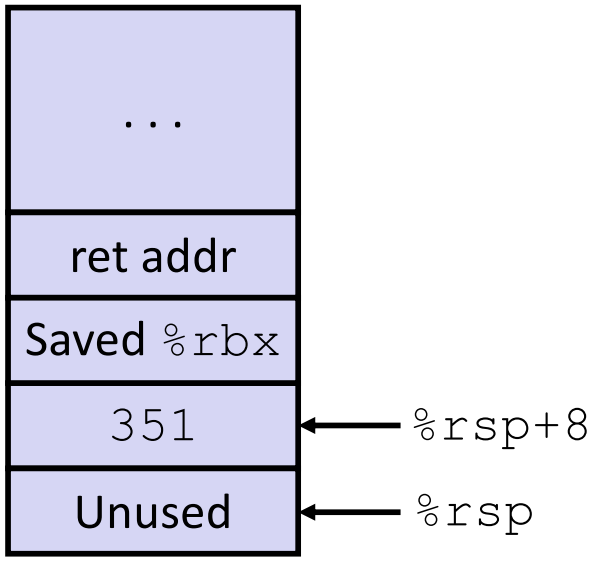
```
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Initial Stack Structure



Resulting Stack Structure

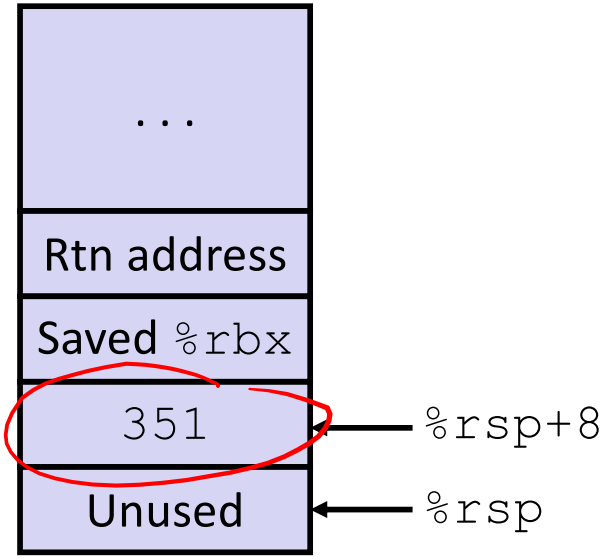


Callee-Saved Example (step 2)

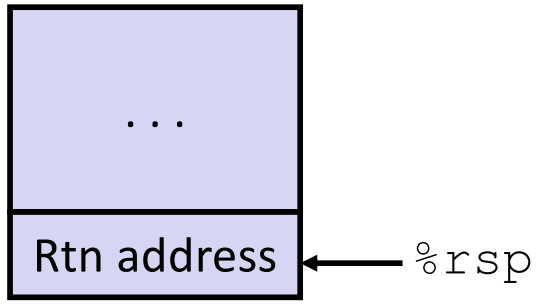
```
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Stack Structure



Pre-return Stack Structure



Register Conventions Summary

- **Caller**-saved register values need to be pushed onto the stack before making a procedure call *only if the Caller needs that value later*
  - **Callee** may change those register values
- **Callee**-saved register values need to be pushed onto the stack *only if the Callee intends to use those registers*
  - **Caller** expects unchanged values in those registers

- Don't forget to restore/pop the values later!

Procedures

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

Recursive Function

```
unsigned int fact(unsigned int n) {
    if (n == 0) {
        return 1;
    }
    return n * fact(n-1);
}
```

Compiler Explorer:
https://godbolt.org/z/pzs7N1
Try -O2!

```
fact:
  testl   %edi, %edi
  jne     .L8
  movl    $1, %eax
  ret
.L8:
  pushq   %rbx
  movl    %edi, %ebx
  subl    $1, %edi
  call    fact
  imull   %ebx, %eax
  popq    %rbx
  ret
```

Recursive Function: Base Case

```
unsigned int fact(unsigned int n) {
    if (n == 0) {
        return 1;
    }
    return n * fact(n-1);
}
```

Register	Use(s)	Type
%rdi	n	Argument
%rax	Return value	Return value

```
fact:
  testl   %edi, %edi
  jne     .L8
  movl    $1, %eax
  ret
.L8:
  pushq   %rbx
  movl    %edi, %ebx
  subl    $1, %edi
  call    fact
  imull   %ebx, %eax
  popq    %rbx
  ret
```

Recursive Function: Callee Register Save

```
unsigned int fact(unsigned int n) {
    if (n == 0) {
        return 1;
    }
    return n * fact(n-1);
}
```

Register	Use(s)	Type
%rdi	n	Argument

Need original value of n *after* recursive call to fact.

"Save" by putting in %rbx (**callee** saved), but need to save old value of %rbx before you change it.

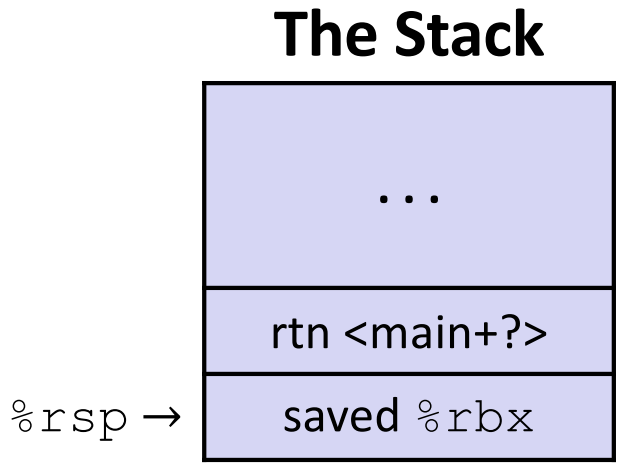
The Stack


```
fact:
  testl   %edi, %edi
  jne     .L8
  movl    $1, %eax
  ret
.L8:
  pushq   %rbx
  movl    %edi, %ebx
  subl    $1, %edi
  call    fact
  imull   %ebx, %eax
  popq    %rbx
  ret
```

Recursive Function: Call Setup

```
unsigned int fact(unsigned int n) {
    if (n == 0) {
        return 1;
    }
    return n * fact(n-1);
}
```

Register	Use(s)	Type
%rdi	n (new)	Argument
%rbx	n (old)	Callee saved




```
fact:
   testl   %edi, %edi
   jne     .L8
   movl    $1, %eax
   ret
.L8:
   pushq   %rbx
   movl    %edi, %ebx
   subl    $1, %edi
   call    fact
   imull   %ebx, %eax
   popq    %rbx
   ret
```

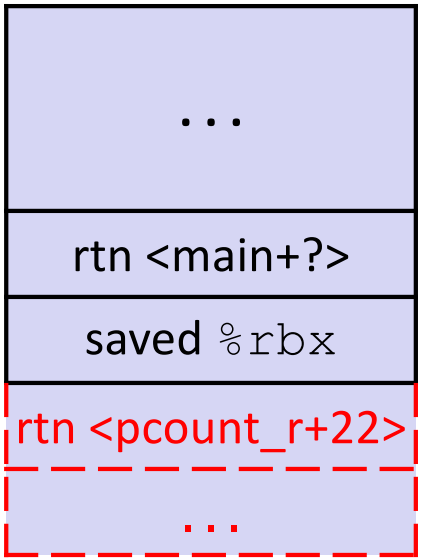
Recursive Function: Call

```
unsigned int fact(unsigned int n) {
    if (n == 0) {
        return 1;
    }
    return n * fact(n-1);
}
```

Register	Use(s)	Type
%rax	Recursive call return value	Return value
%rbx	n (old)	Callee saved

```
fact:
  testl   %edi, %edi
  jne     .L8
  movl    $1, %eax
  ret
.L8:
  pushq   %rbx
  movl    %edi, %ebx
  subl    $1, %edi
  call    fact
  imull   %ebx, %eax
  popq    %rbx
  ret
```

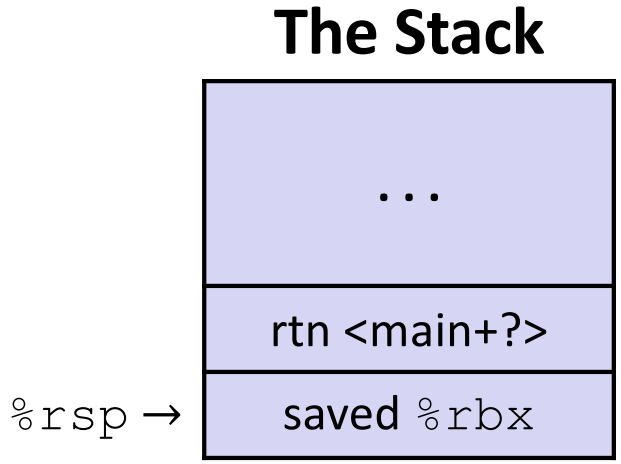
The Stack



Recursive Function: Result

```
unsigned int fact(unsigned int n) {
    if (n == 0) {
        return 1;
    }
    return n * fact(n-1);
}
```

Register	Use(s)	Type
%rax	Return value	Return value
%rbx	n	Callee saved




```
fact:
   testl    %edi, %edi
   jne      .L8
   movl     $1, %eax
   ret
.L8:
   pushq    %rbx
   movl     %edi, %ebx
   subl     $1, %edi
   call     fact
   imull    %ebx, %eax
   popq     %rbx
   ret
```

Observations About Recursion

- Works without any special consideration
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another's data
    - Unless the code explicitly does so (*e.g.* buffer overflow)
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out (LIFO)
- Also works for mutual recursion (P calls Q; Q calls P)

x86-64 Stack Frames

- Many x86-64 procedures have a minimal stack frame
  - Only return address is pushed onto the stack when procedure is called

- A procedure *needs* to grow its stack frame when it:
  - Has too many local variables to hold in **caller**-saved registers
  - Has local variables that are arrays or structs
  - Uses `&` to compute the address of a local variable
  - Calls another function that takes more than six arguments
  - Is using **caller**-saved registers and then calls a procedure
  - Modifies/uses **callee**-saved registers

x86-64 Procedure Summary

- Important Points
  - Procedures are a combination of *instructions* and *conventions*
    - Conventions prevent functions from disrupting each other
  - Stack is the right data structure for procedure call/return
    - If P calls Q, then Q returns before P
  - Recursion handled by normal calling conventions
- Heavy use of registers
  - Faster than using memory
  - Use limited by data size and conventions
- Minimize use of the Stack