

Lecture 08

Procedures

CS213 – Intro to Computer Systems
Branden Ghen a – Winter 2026

Slides adapted from:

St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Today's Goals

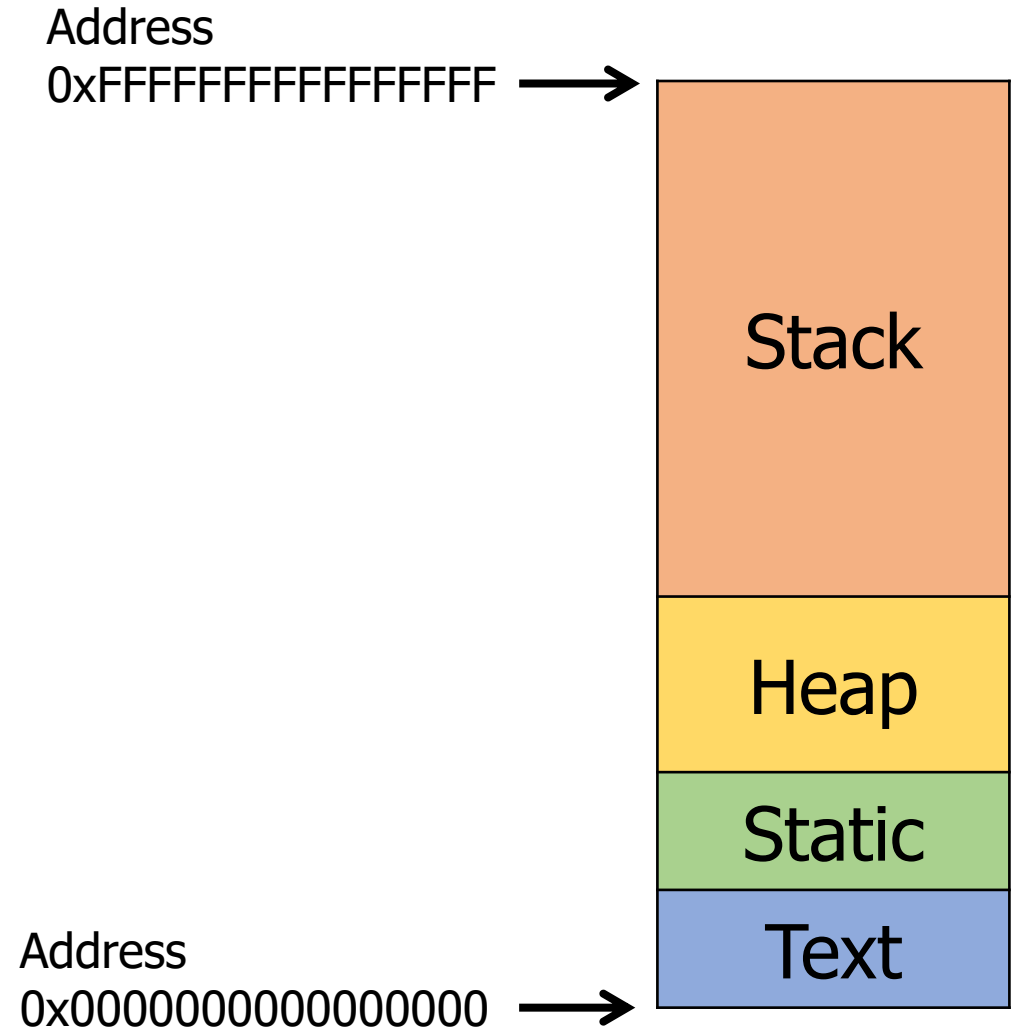
- Describe C memory layout
- Explore functions in assembly
 - How do we call them and return from them?
 - How do we create local variables?
- Understand how we manage register use between functions

Outline

- **C Code Layout**
- x86-64 Calling Convention
- Managing Local Data
- Register Saving
 - Recursion Example

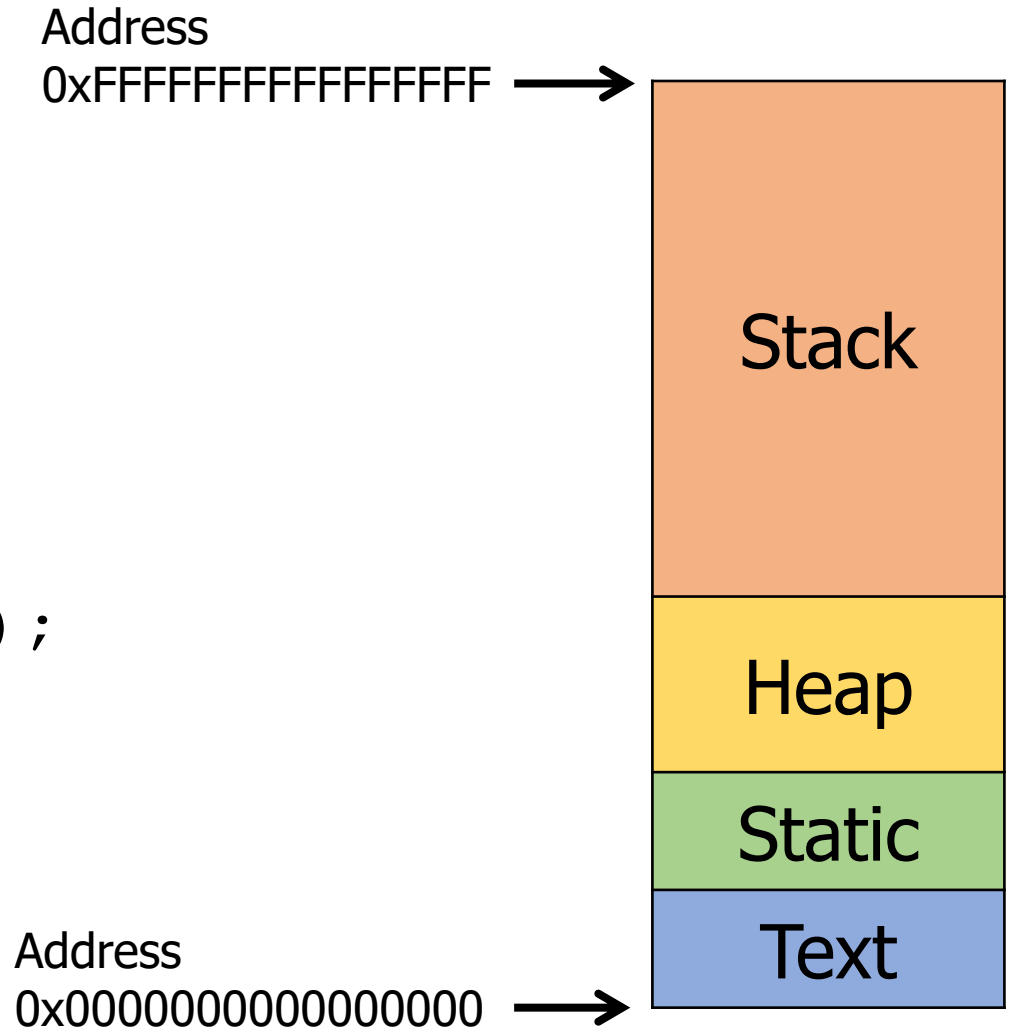
C memory layout

- Stack Section
 - Local variables (that aren't in registers)
 - Function arguments
- Heap Section
 - Memory granted through `malloc()`
- Static Section (a.k.a. Data Section)
 - Global variables
 - Static function variables
- Text Section (a.k.a Code Section)
 - Program code



C memory layout

```
char glob_str[80] = {0};  
  
void func(short b, int* f) {  
    static int c = 3;  
  
    char* d = "Test";  
    int* e = malloc(sizeof(int));  
  
    printf("Hello CS213\n");  
}
```



C memory layout

```
char glob_str[80] = {0};
```

```
void func(short b, int* f) {  
    static int c = 3;
```

```
    char* d = "Test";
```

```
    int* e = malloc(sizeof(int));
```

```
    printf("Hello CS213\n");
```

```
}
```

Address

0xFFFFFFFFFFFFFFFF →

Stack

Heap

Static

Text

Address

0x0000000000000000 →

C memory layout

```
char glob_str[80] = {0};  
void func(short b, int* f) {  
    static int c = 3;  
  
    char* d = "Test";  
    int* e = malloc(sizeof(int));  
  
    printf("Hello CS213\n");  
}
```

Address

0xFFFFFFFFFFFFFFFF →

Stack

Heap

Static

Text

Address

0x0000000000000000 →

C memory layout

```
char glob_str[80] = {0};  
void func(short b, int* f) {  
    static int c = 3;  
  
    char* d = "Test";  
    int* e = malloc(sizeof(int));  
  
    printf("Hello CS213\n");  
}
```

Address

0xFFFFFFFFFFFFFFFF →

Stack

Heap

Static

Text

Address

0x0000000000000000 →

C memory layout

```
char glob_str[80] = {0};  
void func(short b, int* f) {  
    static int c = 3;  
  
    char* d = "Test";  
    int* e = malloc(sizeof(int));  
  
    printf("Hello CS213\n");  
}
```

Address

0xFFFFFFFFFFFFFFFF →

Stack

Heap

Static

Text

Address

0x0000000000000000 →

C memory layout

```
char glob_str[80] = {0};  
void func(short b, int* f) {  
    static int c = 3;  
  
    char* d = "Test";  
    int* e = malloc(sizeof(int));  
  
    printf("Hello CS213\n");  
}
```

Address

0xFFFFFFFFFFFFFFFF →

Stack

Heap

Static

Text

Address

0x0000000000000000 →

C memory layout

```
char glob_str[80] = {0};  
void func(short b, int* f) {  
    static int c = 3;  
  
    char* d = "Test";  
    int* e = malloc(sizeof(int));  
  
    printf("Hello CS213\n");  
}
```

Address

0xFFFFFFFFFFFFFFFF →

Stack

Heap

Static

Text

Address

0x0000000000000000 →

C memory layout

```
char glob_str[80] = {0};  
void func(short b, int* f) {  
    static int c = 3;  
  
    char* d = "Test";  
    int* e = malloc(sizeof(int));  
  
    printf("Hello CS213\n");  
}
```

Address

0xFFFFFFFFFFFFFFFF →

Stack

Heap

Static

Text

Address

0x0000000000000000 →

C memory layout

```
char glob_str[80] = {0};  
void func(short b, int* f) {  
    static int c = 3;  
  
    char* d = "Test";  
    int* e = malloc(sizeof(int));  
  
    printf("Hello CS213\n");  
}
```

Address

0xFFFFFFFFFFFFFFFF →

Stack

Heap

Static

Text

Address

0x0000000000000000 →

C memory layout

```
char glob_str[80] = {0};  
void func(short b, int* f) {  
    static int c = 3;  
  
    char* d = "Test";  
    int* e = malloc(sizeof(int));  
  
    printf("Hello CS213\n");  
}
```

Address

0xFFFFFFFFFFFFFFFF →

Stack

Heap

Static

Text

Address

0x0000000000000000 →

C memory layout

```
char glob_str[80] = {0};
```

```
void func(short b, int* f) {  
    static int c = 3;  
  
    char* d = "Test";  
    int* e = malloc(sizeof(int));  
  
    printf("Hello CS213\n");  
}
```

Assembly code goes in the Text section

Address

0xFFFFFFFFFFFFFFFF →

Stack

Heap

Static

Text

Address

0x0000000000000000 →

Interacting with data sections in assembly

- Stack
 - Stack pointer is saved in `%rsp` and can be moved as needed
 - We'll discuss this today
- Heap
 - C library (malloc) handles this above the machine level
 - i.e. from the machine point of view, there is no heap
- Static
 - Arbitrary pointers to memory can be created and used
 - With memory addressing instructions
 - Might references global values
- Text
 - Assembly code is placed here automatically
 - Labels are just addresses within the Text section

Break + Open Question

- Which sections are absolutely required, and which aren't?
- Text
- Static
- Heap
- Stack

Break + Open Question

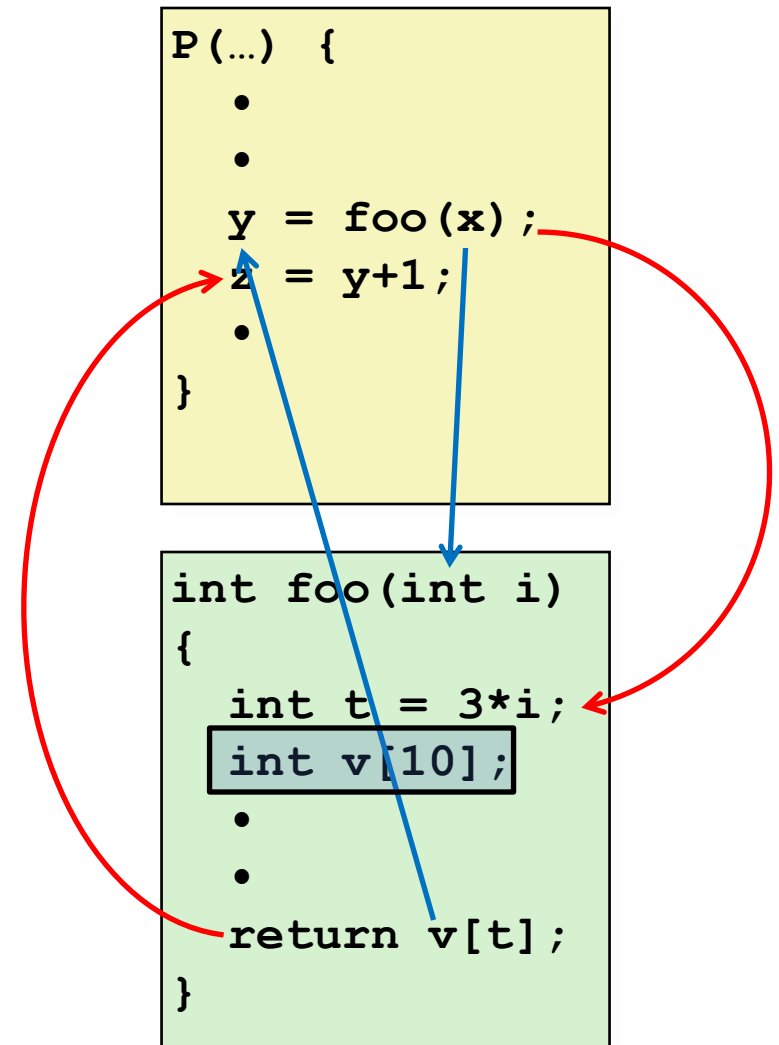
- Which sections are absolutely required, and which aren't?
- Text: necessary since it holds the code
- Static: only necessary if you use globals or strings
- Heap: only necessary if you heap-allocate
(with malloc or automatically in other languages)
- Stack: necessary if you use variables or call functions
(so probably always necessary unless you write in assembly)

Outline

- C Code Layout
- **x86-64 Calling Convention**
- Managing Local Data
- Register Saving
 - Recursion Example

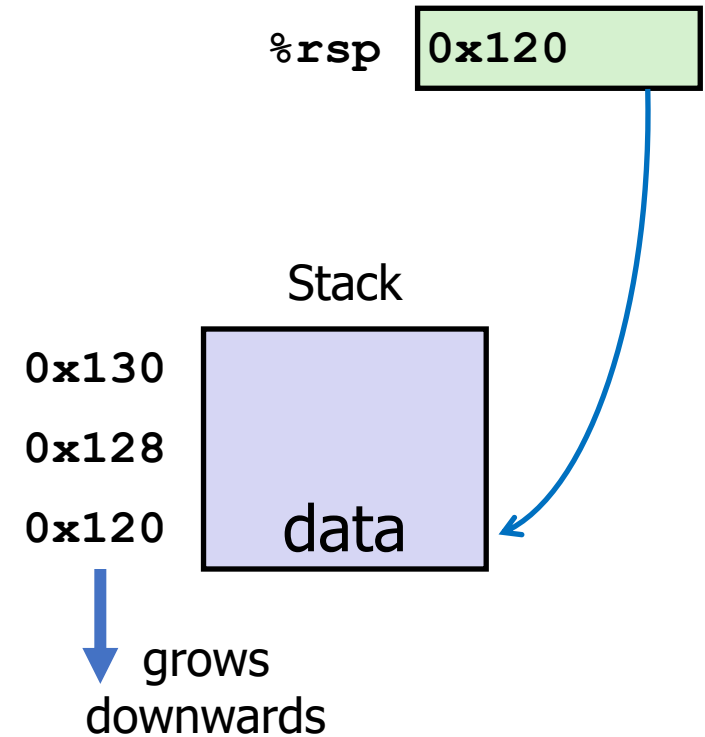
Mechanisms in Procedures

- Passing control
 - To beginning of procedure code
 - Back to return point
- Passing data
 - Procedure arguments
 - Return value
- Local memory management
 - Allocate during procedure execution
 - Deallocate upon return
- No one instruction does all that
 - Need instructions for each
- The stack is the key to all 3 of these!



The Stack in Assembly

- Chunk of memory that code can use
 - Think of it like scratch paper for taking notes
- `%rsp` points at most recently used position
- Stack grows downward
 - Subtract from `%rsp` to claim stack memory
 - Add to `%rsp` to release previously used memory
- Stack is used for multiple purposes
 - Scratch space if not enough registers
 - Function calls (extra arguments, return addresses)



Function control flow

- Two assembly instructions call functions and return from them
 - `callq label`
 - Subtracts 8 from `%rsp` (allocates 8 bytes on the stack)
 - Saves the address of the *next* instruction to the stack at `%rsp`
 - Jumps to label
 - `retq`
 - Loads an address from the stack at `%rsp`
 - Adds 8 to `%rsp` (frees 8 bytes on the stack)
 - Jumps to loaded address

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

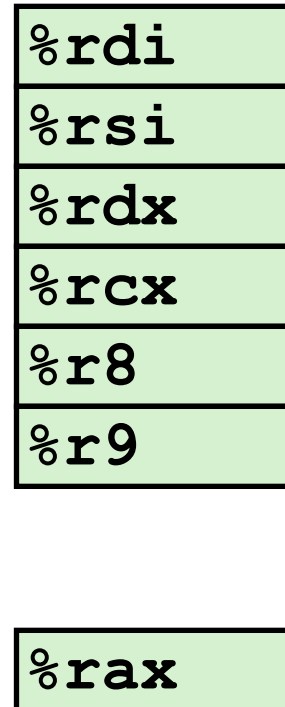
Return address: 0x400549

Just `call` and `ret` are fine by the way, the `q` is assumed (there is no other option)

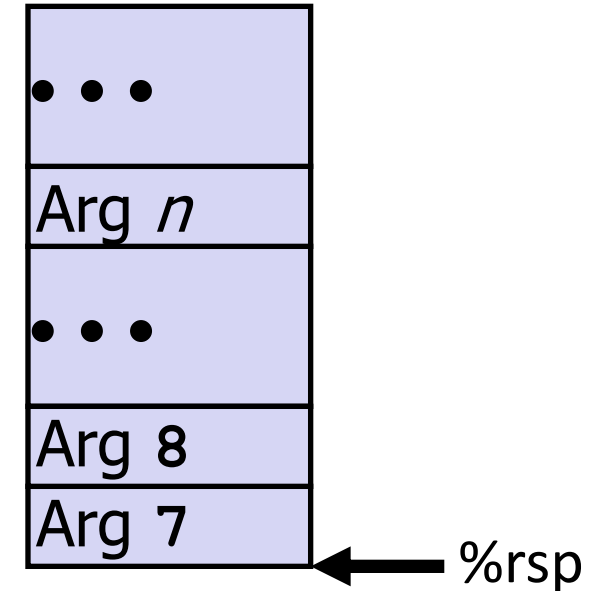
Function data flow in x86-64

- First 6 arguments are in registers
 - `%rdi` is first argument
- Next `n` arguments are on the stack
 - Too many arguments hurts performance
- Return value is in `%rax`

Registers



Stack



(Only allocate stack space when needed)

Code Example

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

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

Which registers hold arguments?

No return

%rdi %rsi %rdx

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

```
0000000000400540 <multstore>:  
400544: callq 400550 <mult2>    # mult2(x,y)  
400549: movq  %rax, (%rdx)      # Store at *dest  
40054d: retq                    # Return
```

%rax

%rdi %rsi

```
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: retq                   # Return
```

Control flow

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

0000000000400540 <multstore>:

400544:	callq	400550 <mult2>	# mult2(x,y)
400549:	movq	%rax, (%rdx)	# Store at *dest
40054d:	retq		# Return

Calls the mult2 function

```
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:	retq		# Return

Return instructions

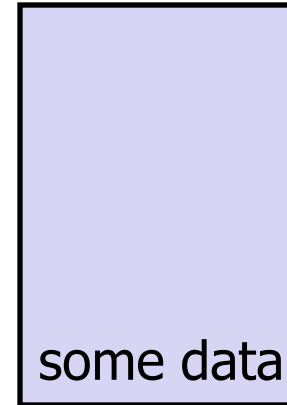
Control Flow Example

about to execute `callq`

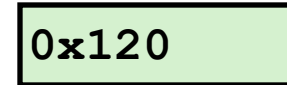
```
0000000000400540 <multstore>:  
.  
.  
400544: callq 400550 <mult2>  
400549: movq %rax, (%rdx)  
.  
.
```

```
0000000000400550 <mult2>:  
400550: movq %rdi, %rax  
.  
.  
400557: retq
```

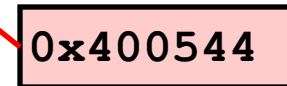
0x130
0x128
0x120



%rsp



%rip



Next instruction
to execute

%rip: instruction
pointer

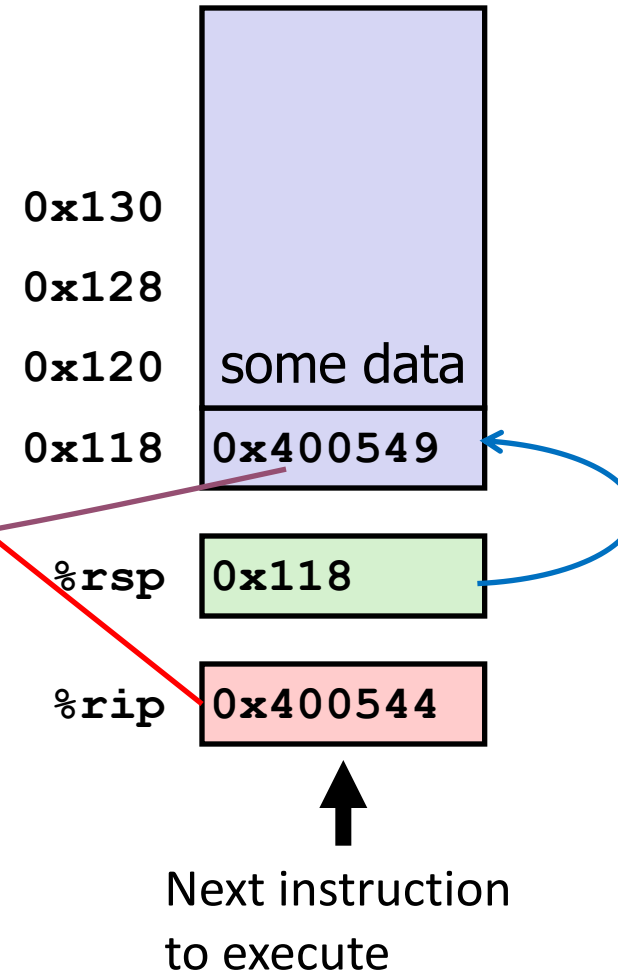
Can't be directly
modified

Control Flow Example

callq step 1

```
0000000000400540 <multstore>:  
.  
.  
400544: callq 400550 <mult2>  
400549: movq %rax, (%rdx)  
.  
.
```

```
0000000000400550 <mult2>:  
400550: movq %rdi, %rax  
.  
.  
400557: retq
```



Control Flow Example

callq step 2

```
0000000000400540 <multstore>:
```

•

•

```
400544: callq 400550 <mult2>
```

```
400549: movq %rax, (%rdx)
```

•

•

```
0000000000400550 <mult2>:
```

```
400550: movq %rdi, %rax
```

•

•

```
400557: retq
```

0x130

0x128

0x120

some data

0x118

0x400549

%rsp

0x118

%rip

0x400550

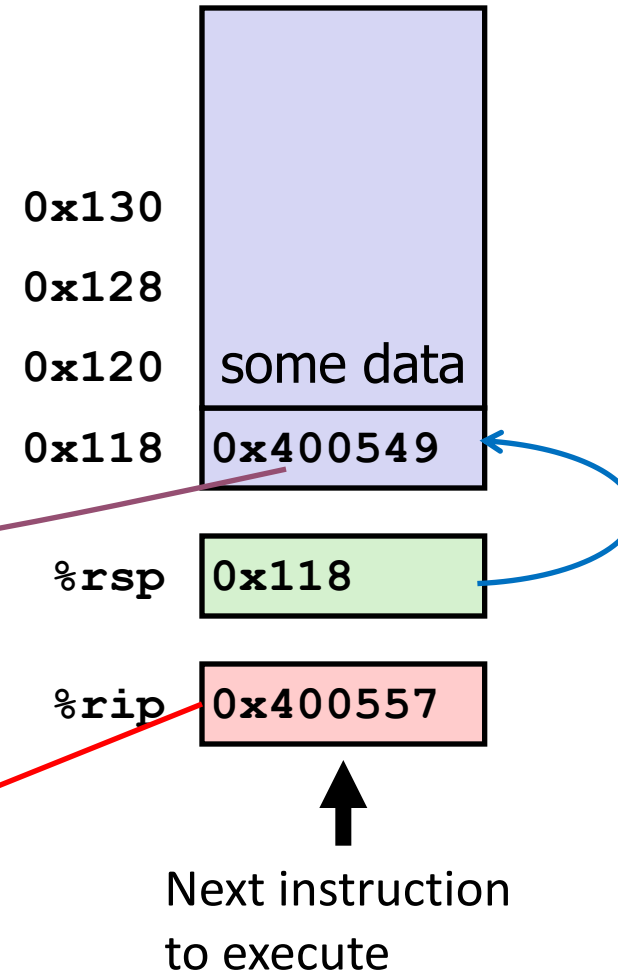
Next instruction
to execute

Control Flow Example

about to execute `retq`

```
0000000000400540 <multstore>:  
.  
.  
400544: callq 400550 <mult2>  
400549: movq %rax, (%rdx)  
.  
.
```

```
0000000000400550 <mult2>:  
400550: movq %rdi, %rax  
.  
.  
400557: retq
```



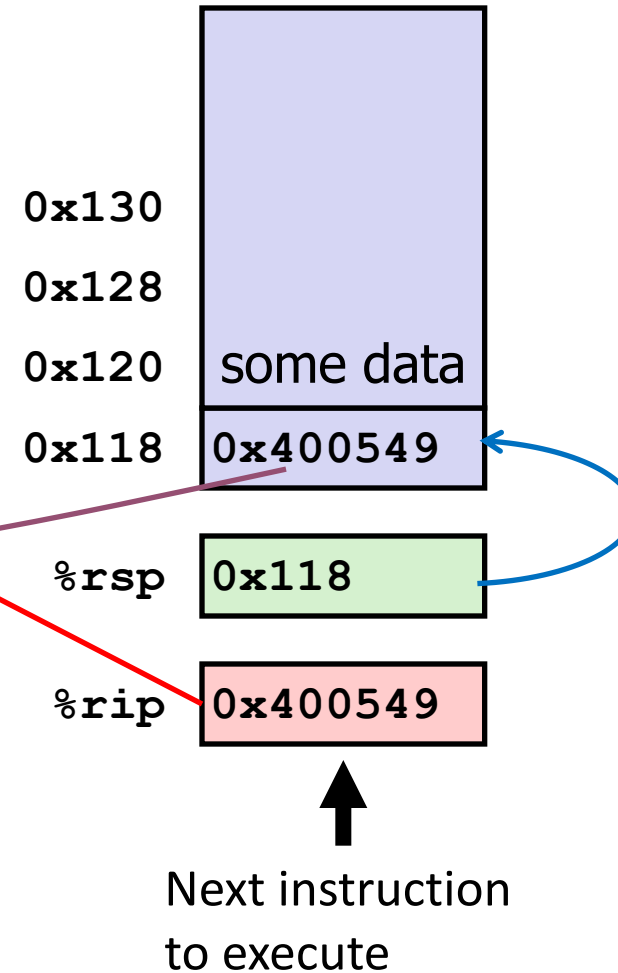
QUIZ: What is the address of the instruction we execute after `retq`?

Control Flow Example

retq step 1

```
0000000000400540 <multstore>:  
.  
.  
400544: callq 400550 <mult2>  
400549: movq %rax, (%rdx)  
.  
.
```

```
0000000000400550 <mult2>:  
400550: movq %rdi, %rax  
.  
.  
400557: retq
```



Control Flow Example

retq step 2

```
0000000000400540 <multstore>:  
.  
.  
400544: callq 400550 <mult2>  
400549: movq %rax, (%rdx)  
.  
.
```

```
0000000000400550 <mult2>:  
400550: movq %rdi, %rax  
.  
.  
400557: retq
```

0x130

0x128

0x120

some data

%rsp

0x120

%rip

0x400549

Next instruction
to execute

Break + Open Question

- How did we decide how many registers to use for arguments and return values?

<code>%rdi</code>
<code>%rsi</code>
<code>%rdx</code>
<code>%rcx</code>
<code>%r8</code>
<code>%r9</code>

- Do all functions have to use this same convention?

<code>%rax</code>

Break + Open Question

- How did we decide how many registers to use for arguments and return values?
 - Testing lots of real-world programs
 - Many style guides suggest you use four or less arguments
 - x86 (32-bit) only had four arguments
 - x86-64 added two more
 - C only has one return result, so one register is fine
- Do all functions have to use this same convention?
 - All functions within a program must, or they won't work
 - Different programs, or different OSes, could choose different

<code>%rdi</code>
<code>%rsi</code>
<code>%rdx</code>
<code>%rcx</code>
<code>%r8</code>
<code>%r9</code>

<code>%rax</code>

Outline

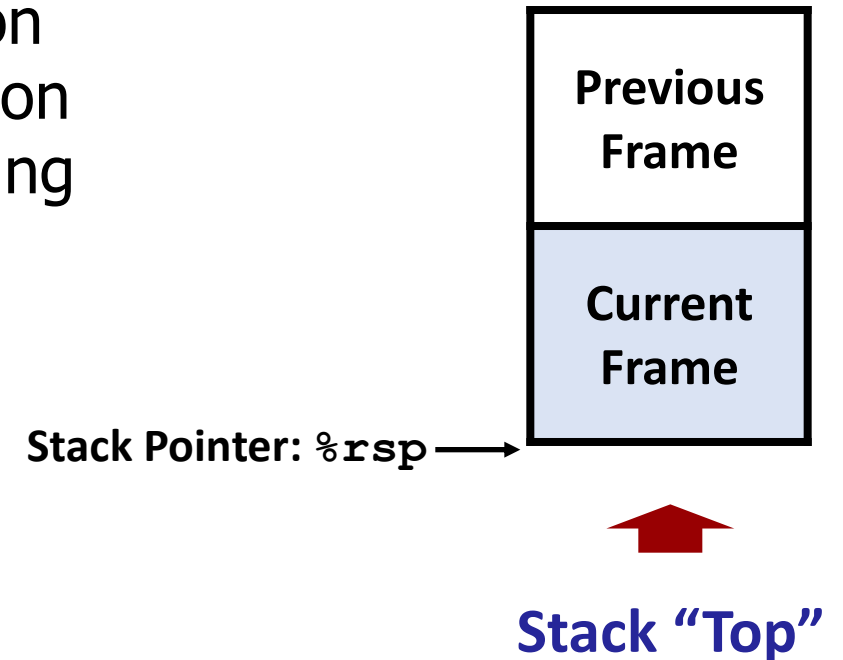
- C Code Layout
- x86-64 Calling Convention
- **Managing Local Data**
- Register Saving
 - Recursion Example

Call-Local State

- Need some place to store state for each call
 - Return address
 - Arguments
 - Local variables
 - Temporary space (if needed)
- Note: these are separate for each call, not each function
 - Function could be called recursively, but each call needs its own local variables
- State only needs to exist until the function returns

Using the Stack for Call-Local State

- Place local state on the stack
- Stack allocated in **Frames**
 - Frame = State for a single procedure invocation
 - Allocated by "setup" code at the start of function
 - Deallocated by "teardown" code before returning
- Caller functions runs the Callee function
 - **Caller** – function that called this function
 - Its state is in the "previous frame"
 - **Callee** – the currently running function
 - Its state is in the "current frame"



Call Chain Example

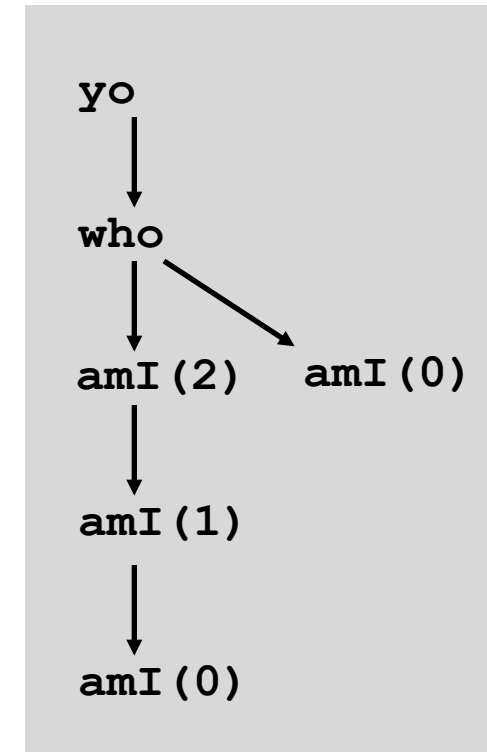
```
yo (...)  
{  
  .  
  .  
  who ();  
  .  
  .  
}
```

```
who (...)  
{  
  . . .  
  amI (2) ;  
  . . .  
  amI (0) ;  
  . . .  
}
```

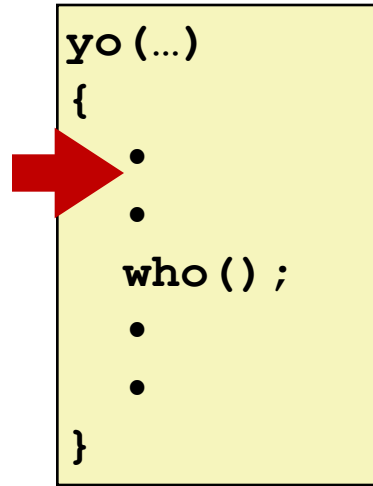
```
amI (int x)  
{  
  .  
  if (x)  
    amI (x-1) ;  
  .  
  .  
}
```

Procedure amI () is recursive

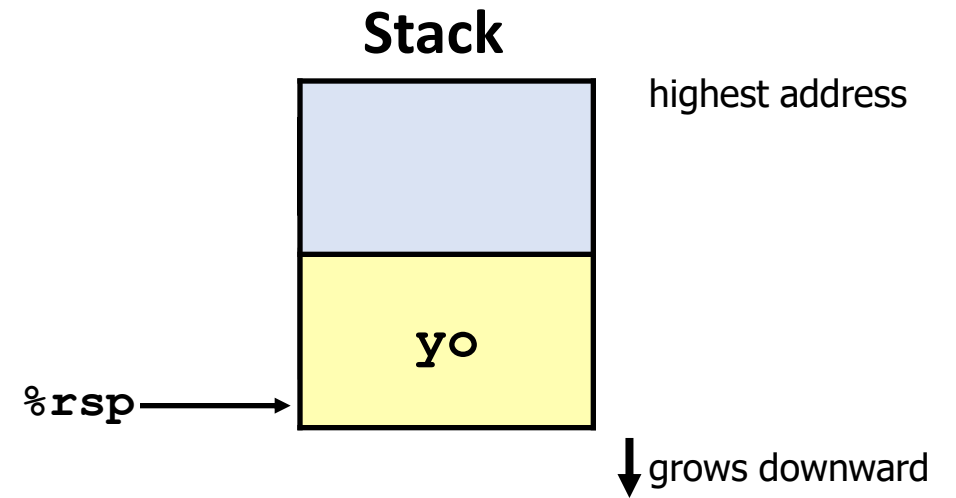
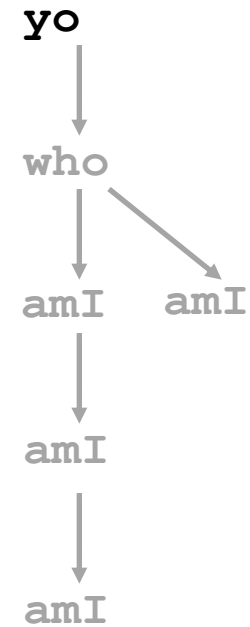
Example Call Chain



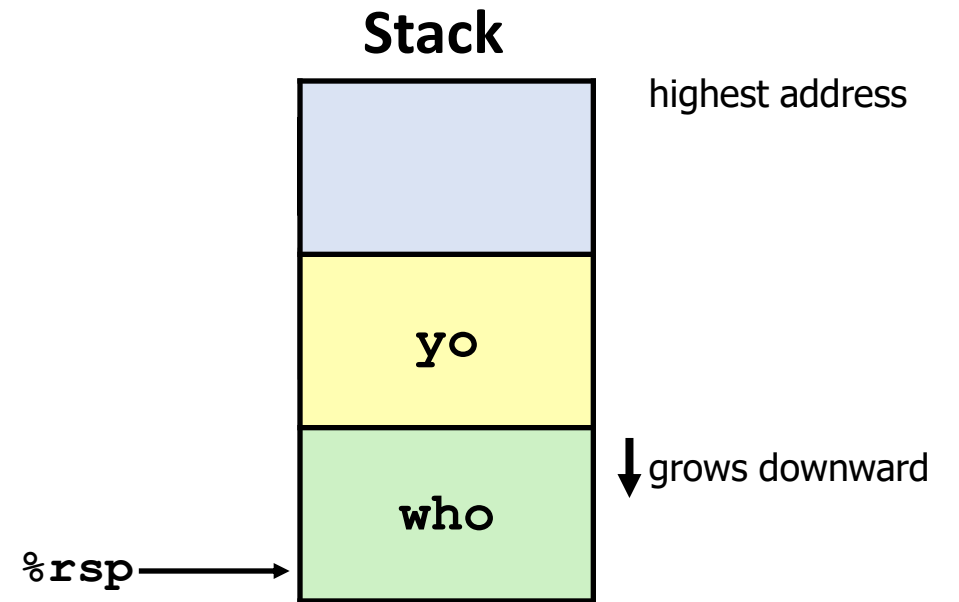
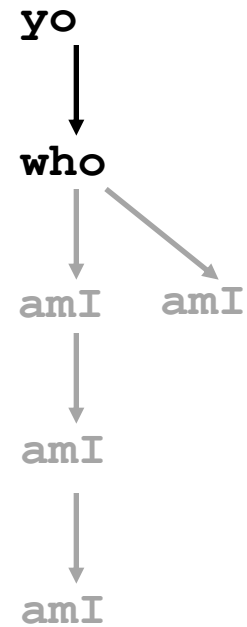
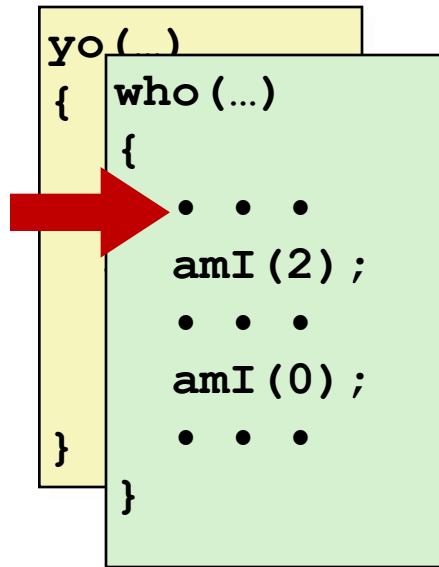
Example



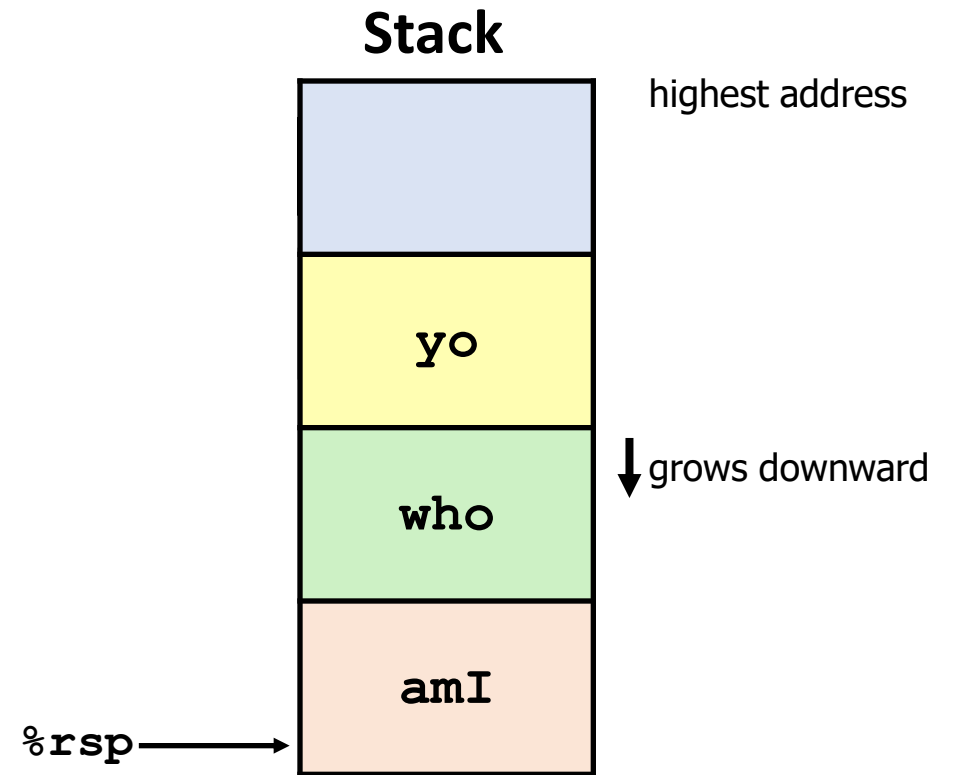
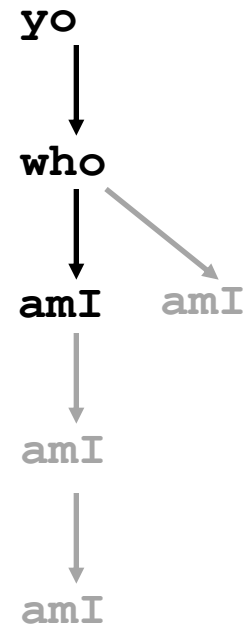
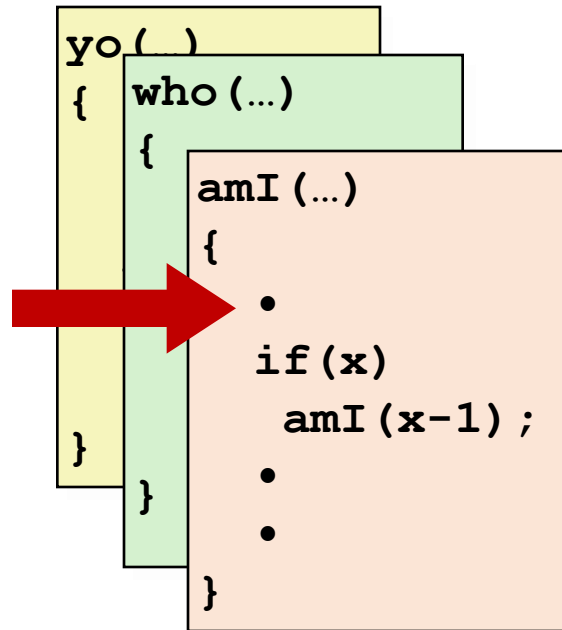
Call Chain



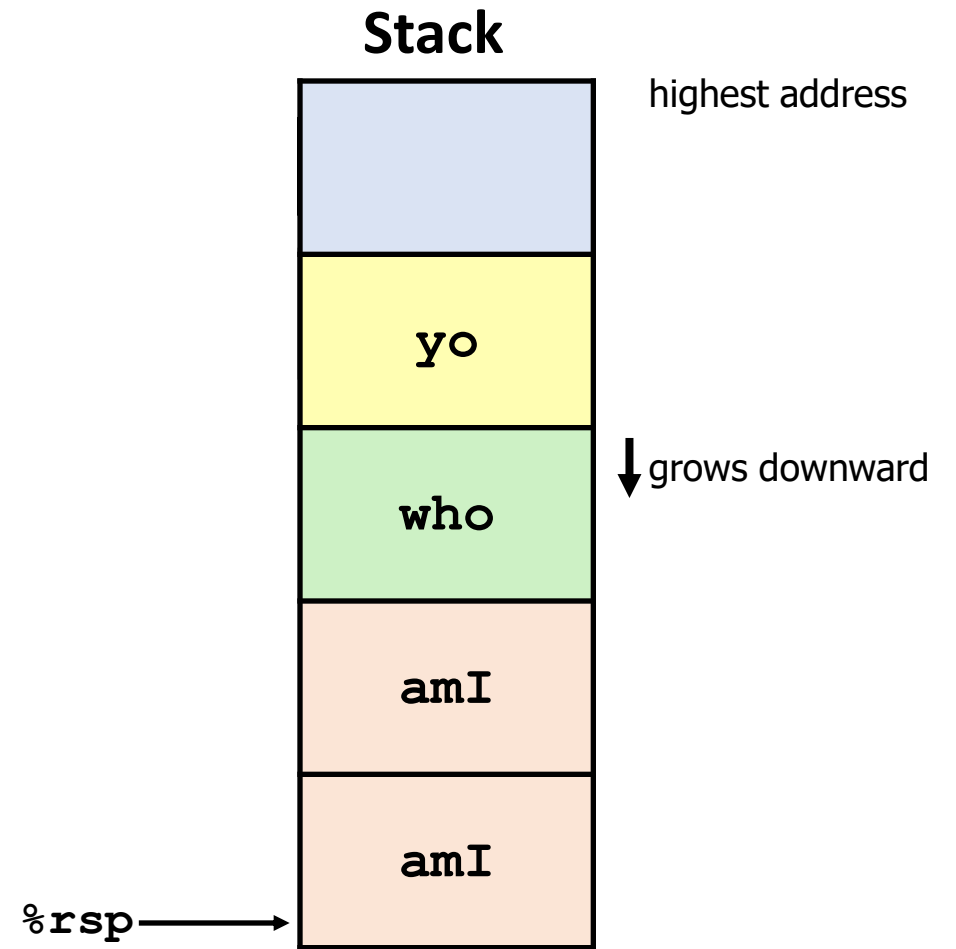
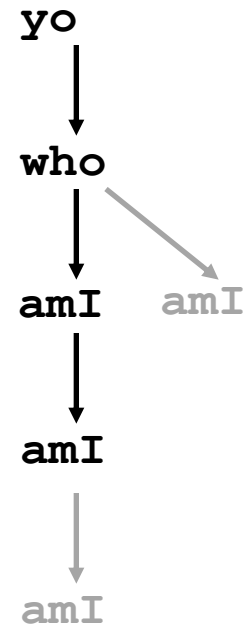
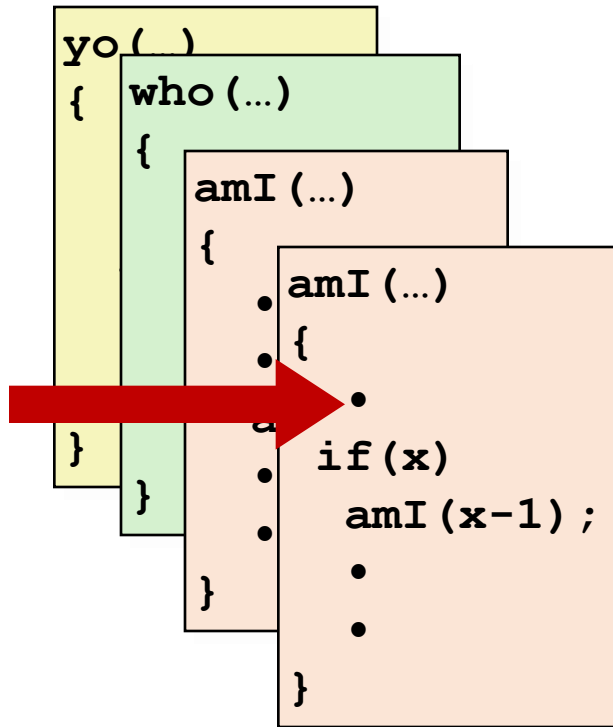
Example



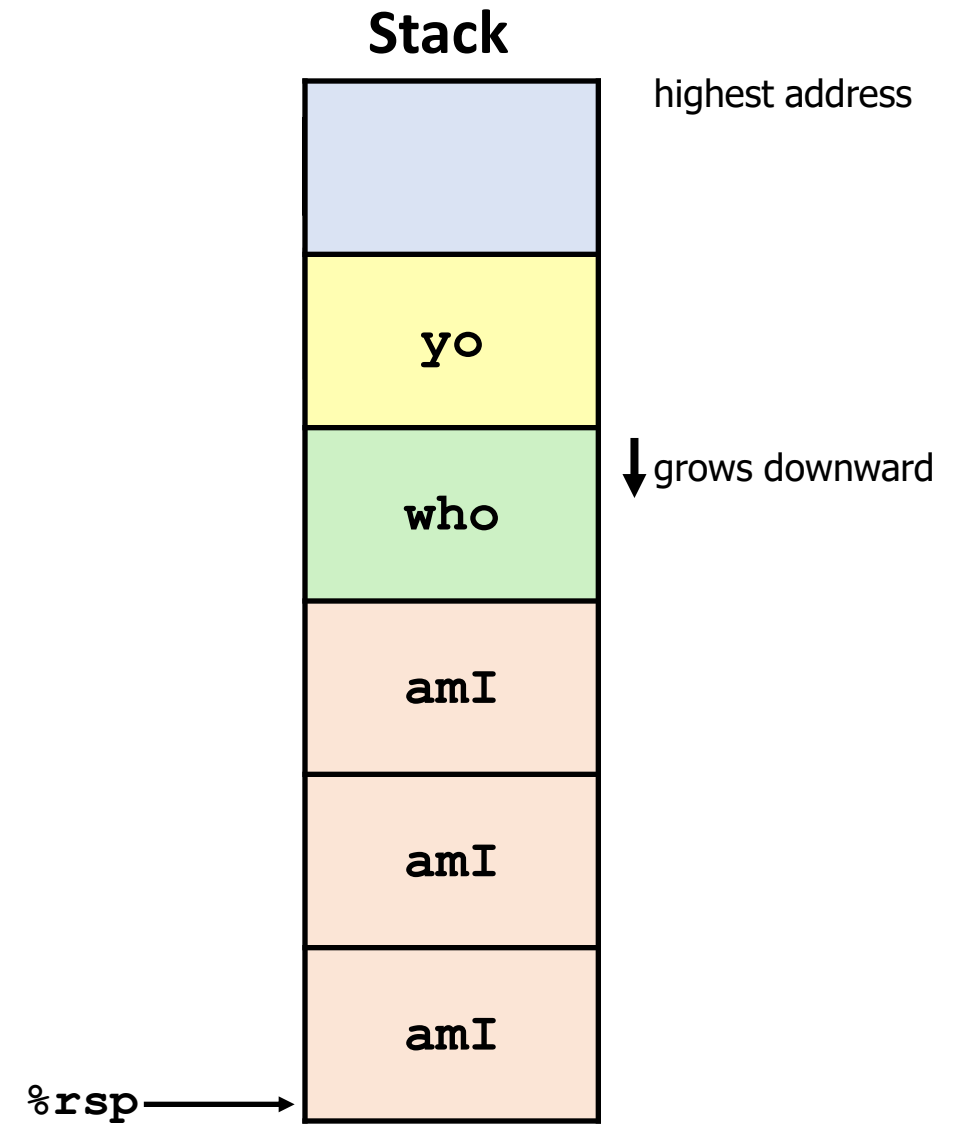
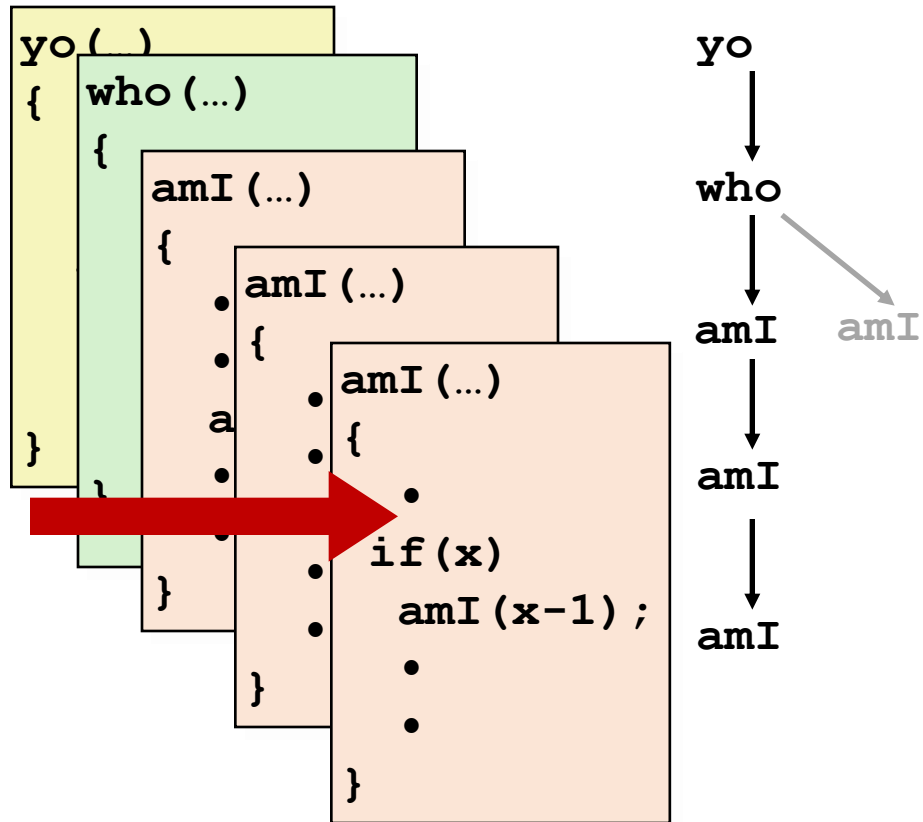
Example



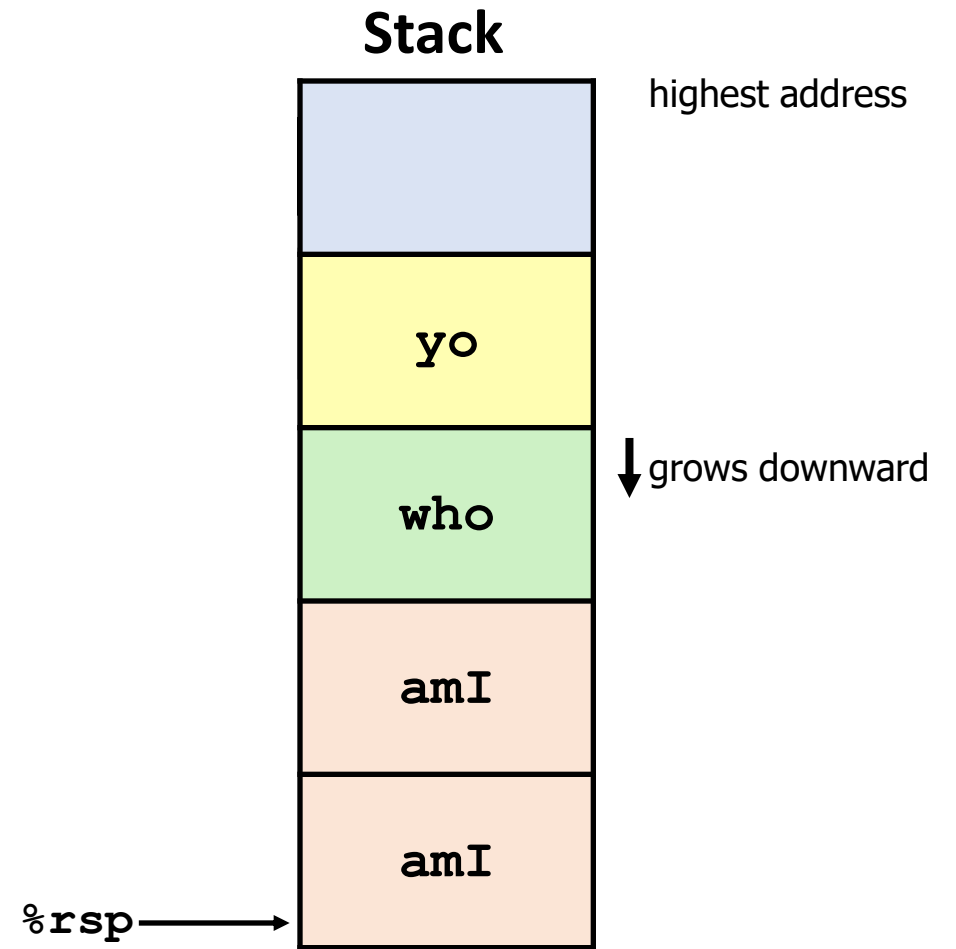
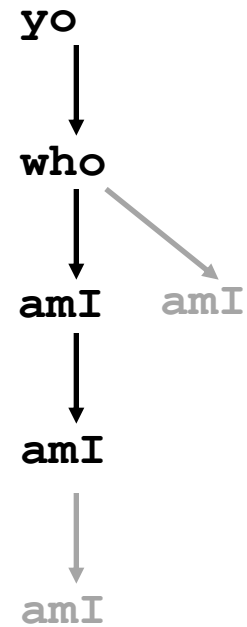
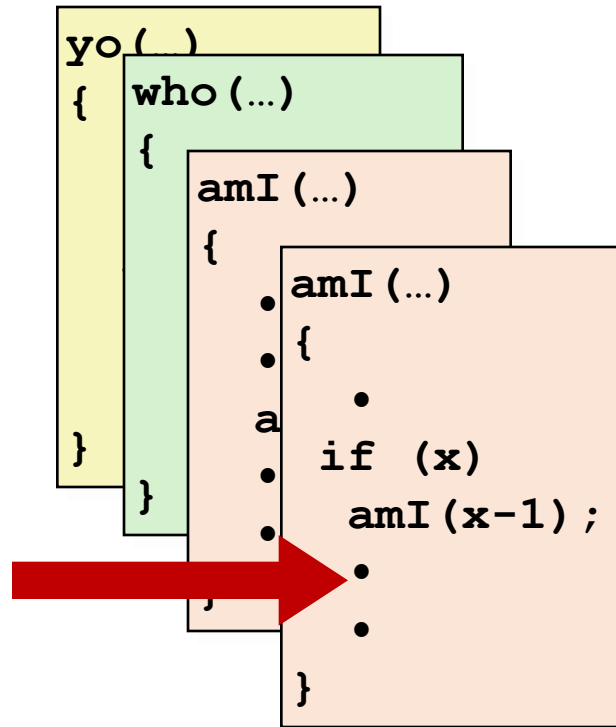
Example



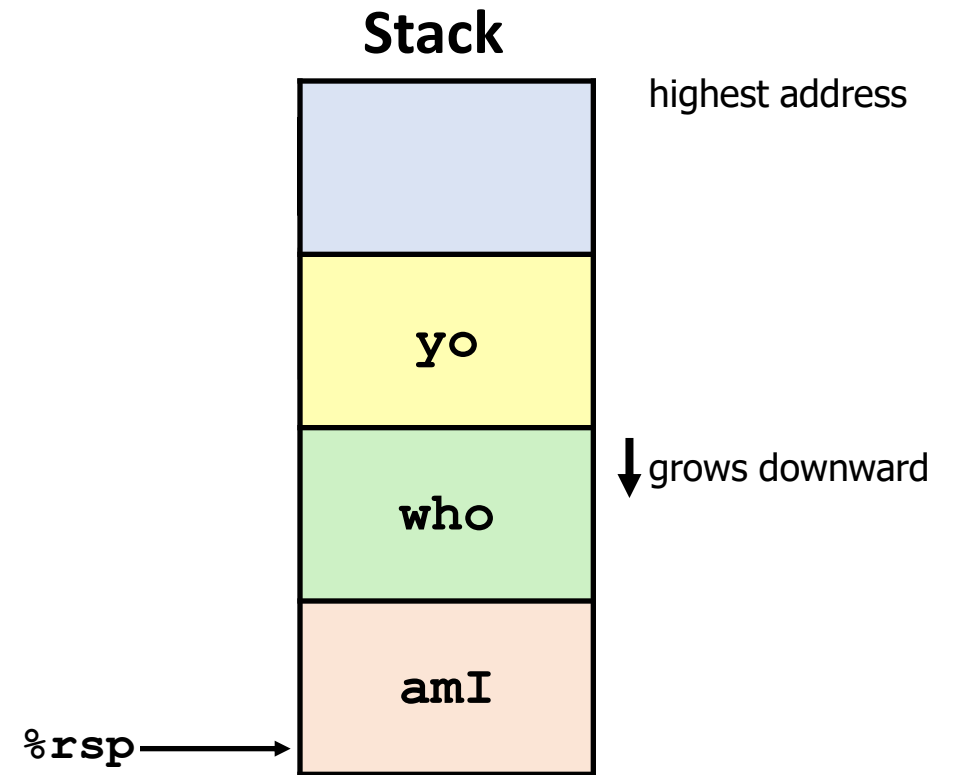
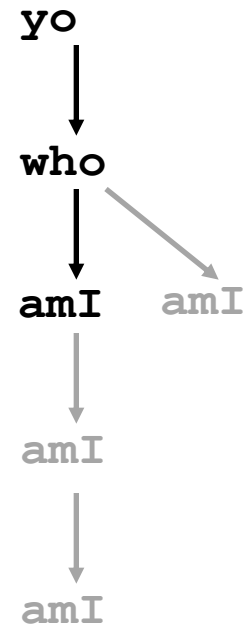
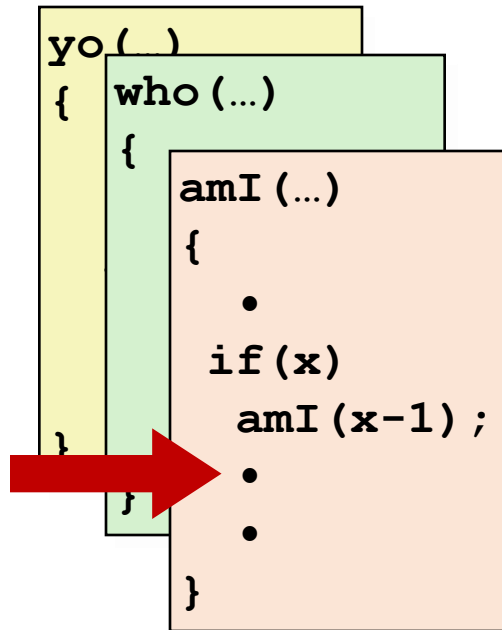
Example



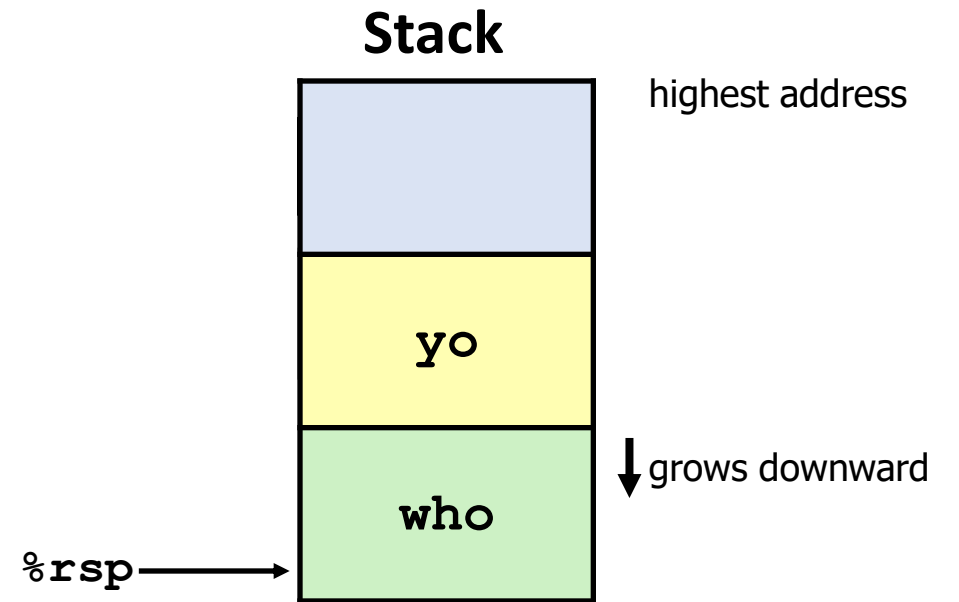
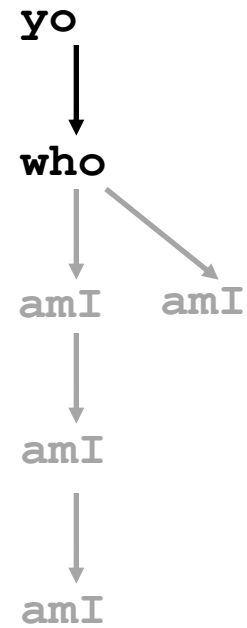
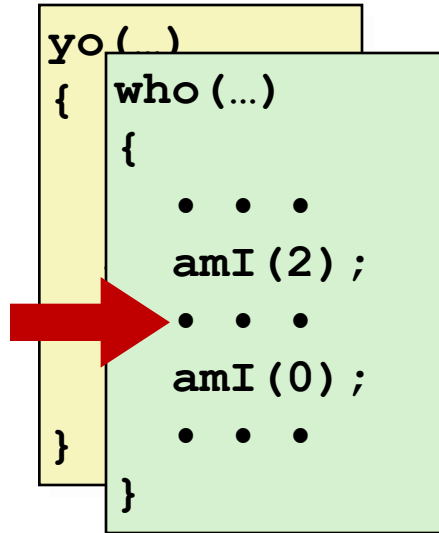
Example



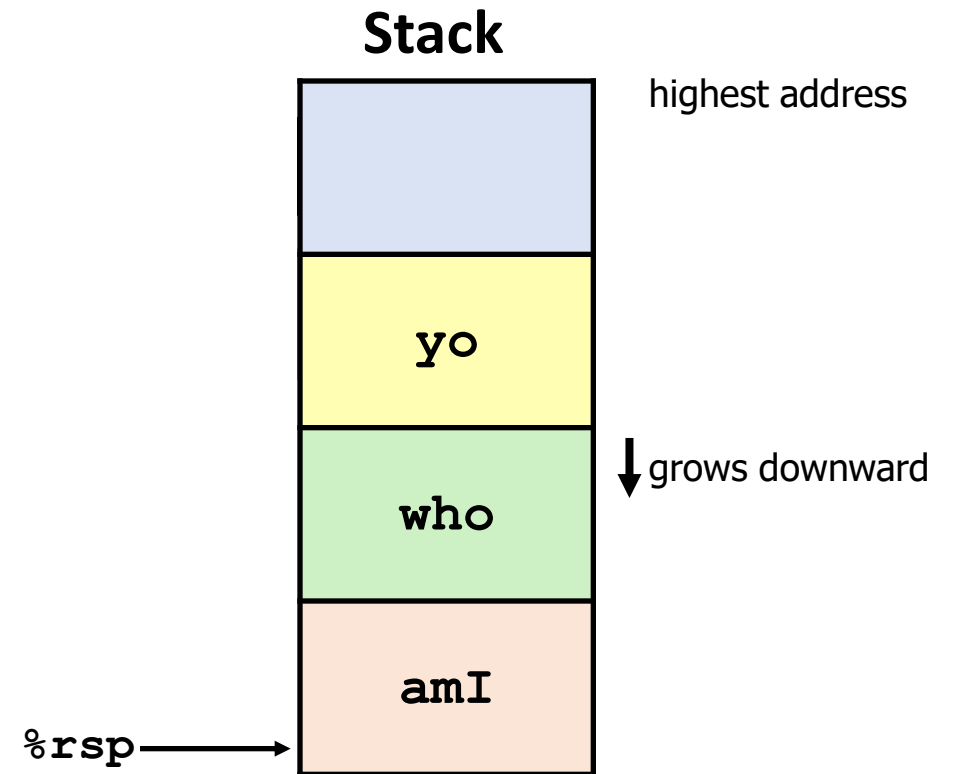
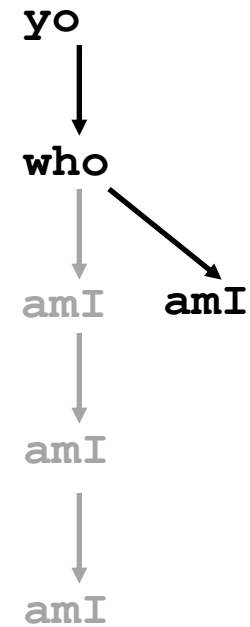
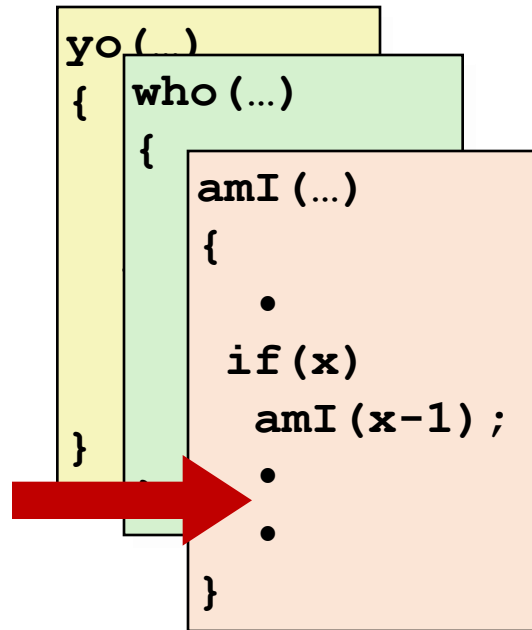
Example



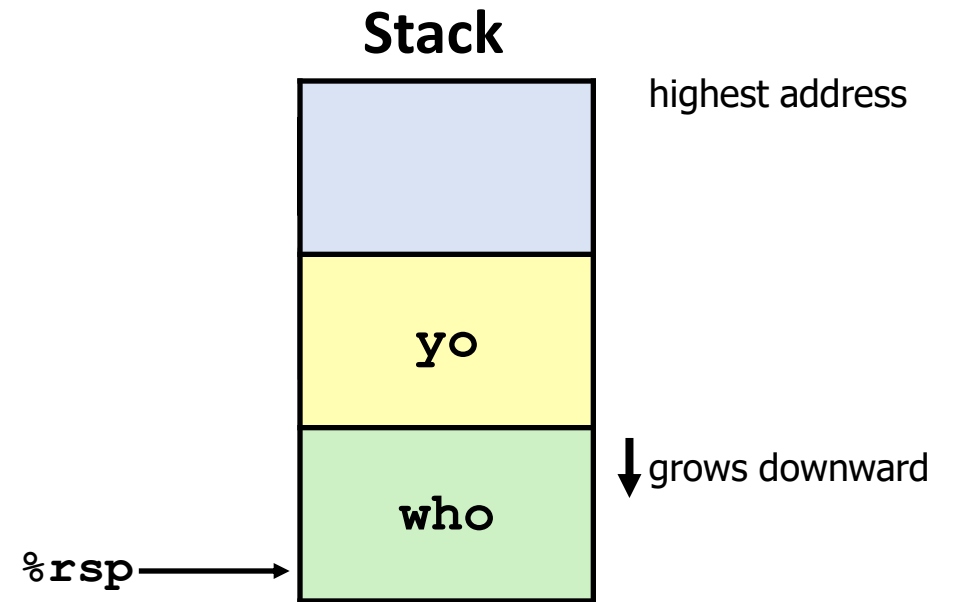
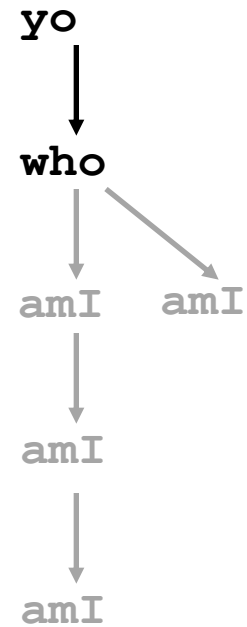
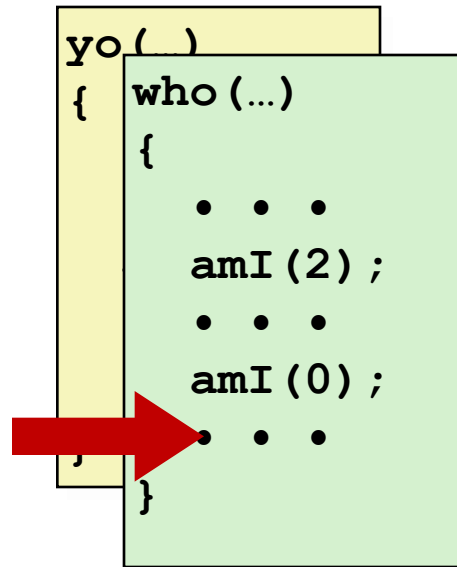
Example



Example

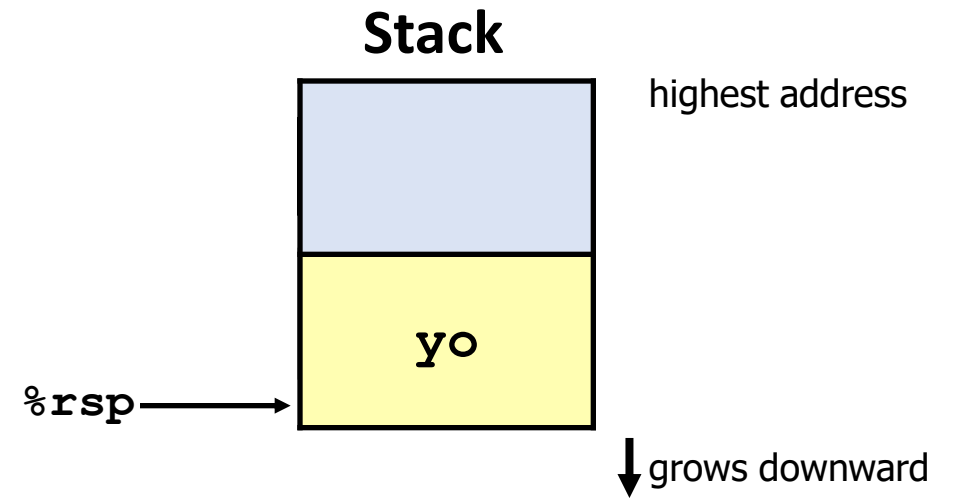
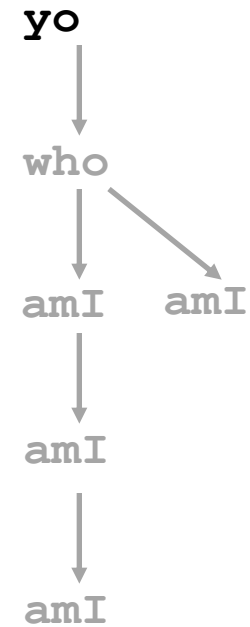


Example

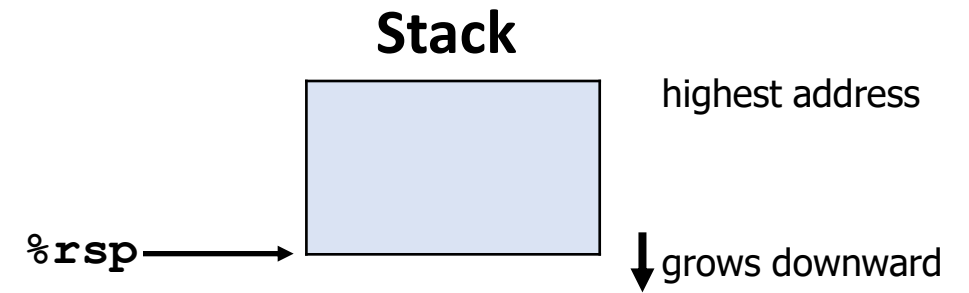


Example

```
yo (...)  
{  
  .  
  .  
  who () ;  
  .  
  .  
}
```



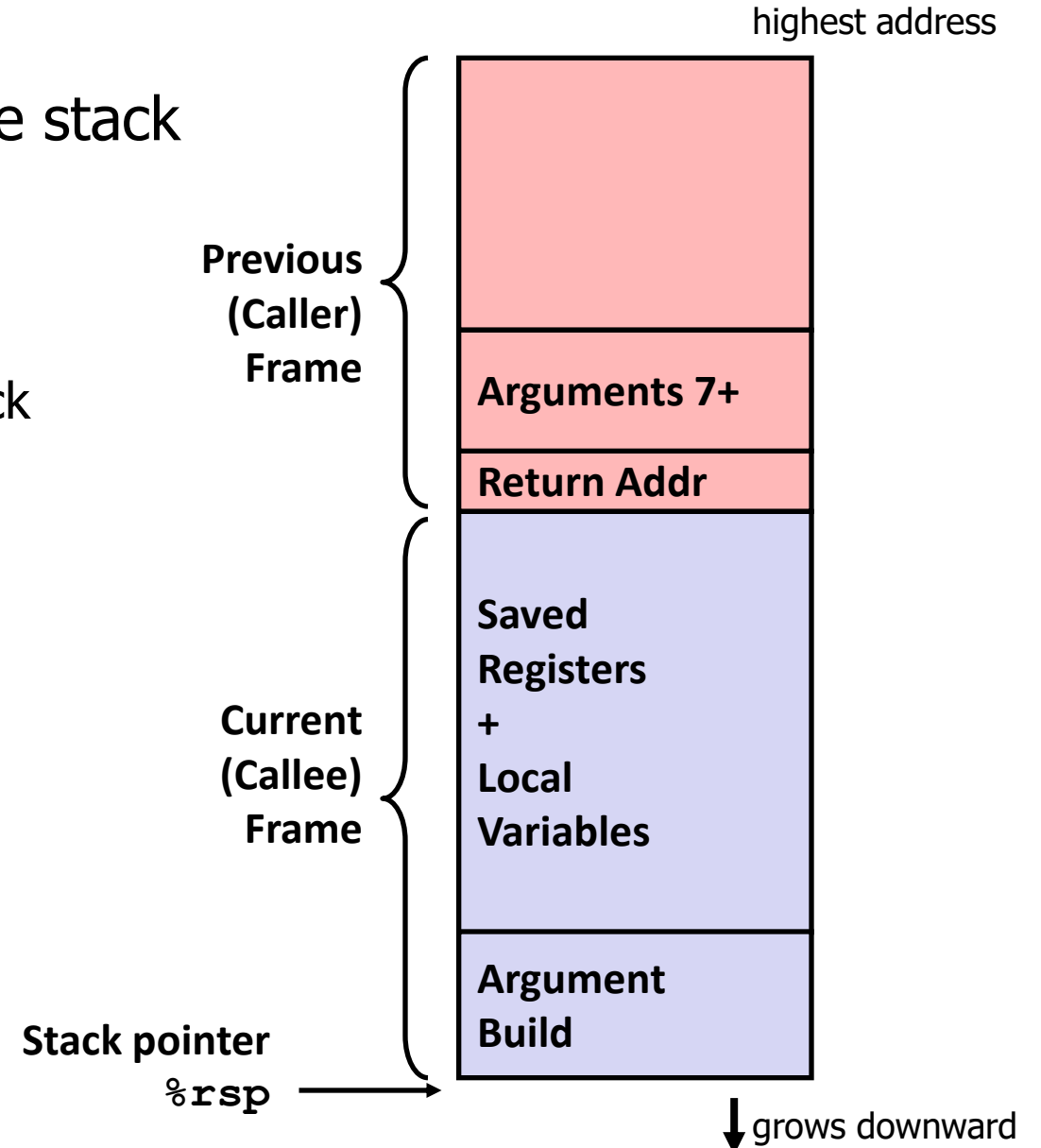
Returning to original stack



- Stack always eventually returns to its default state
 - Happens automatically in higher-level languages like C
 - Need to manage that ourselves if writing assembly
- Or the program can exit early from anywhere
 - Entire stack is deallocated when the program ends

x86-64/Linux Stack Frame

- Walking from `%rsp` backwards through the stack
- Current (Callee) Stack Frame
 - "Argument build":
Arguments for function we're about to call
If there are 7+ and they need to be on the stack
 - Local variables
If we can't keep them in registers
 - Saved register context (we'll get to that soon)
- Previous (Caller) Stack Frame
 - Return address
 - Pushed by `call` instruction
 - Arguments for callee (current function)



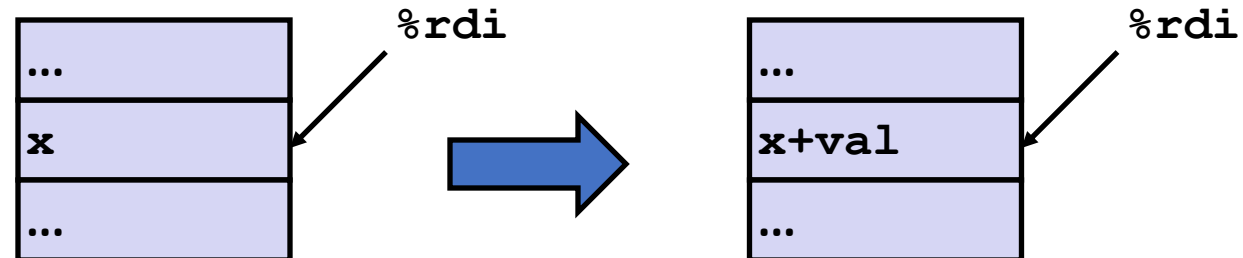
Example: `incr`

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

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

Register	Use(s)
<code>%rdi</code>	Argument <code>p</code>
<code>%rsi</code>	Argument <code>val</code> , also <code>y</code>
<code>%rax</code>	<code>x</code> , Return value

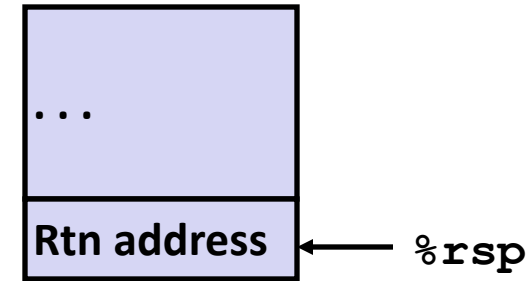
Memory



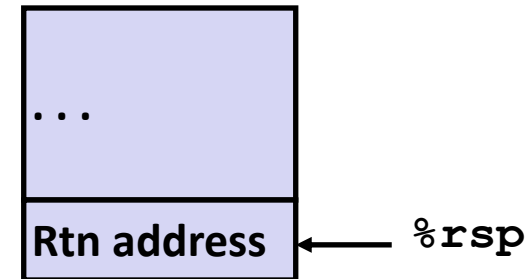
Example: Calling `incr` #1 (local variables)

```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

Initial Stack Structure



Resulting Stack Structure



Example: Calling `incr` #1 (local variables)

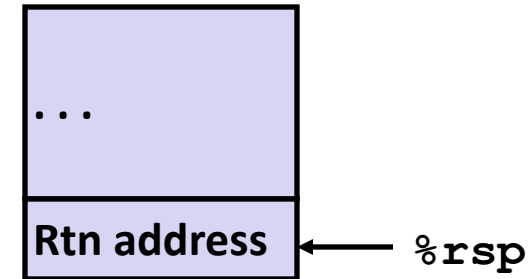
We take `v1`'s address, so must be in memory

```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

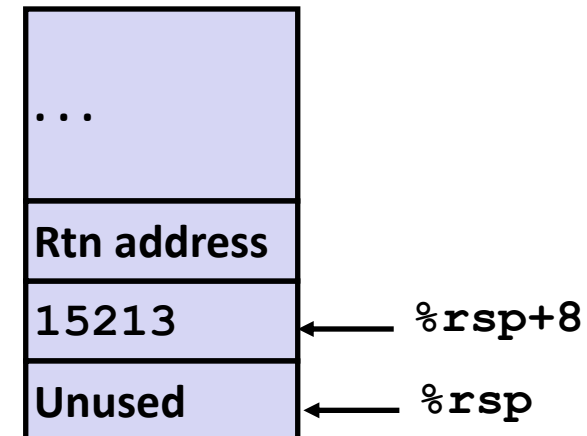
Stack pointer must be multiple of 16

```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movq    $3000, %rsi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Initial Stack Structure



Resulting Stack Structure



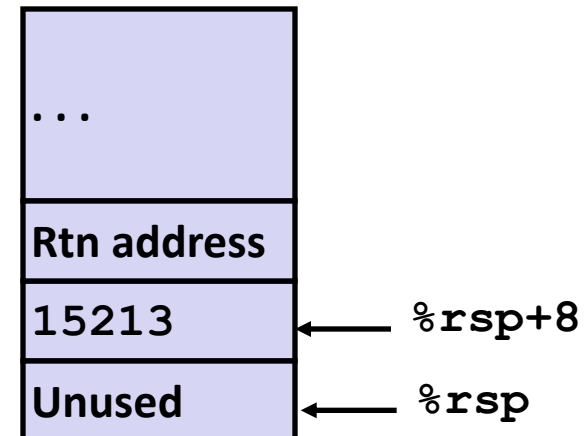
Example: Calling `incr` #2 (argument build)

```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

Register	Use(s)
%rdi	&v1
%rsi	3000

```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movq    $3000, %rsi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Stack Structure



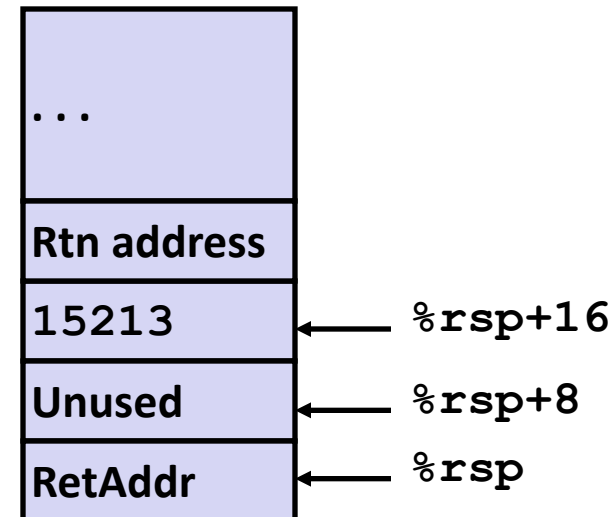
Example: Calling `incr` #3 (control transfer)

```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

Register	Use(s)
%rdi	&v1
%rsi	3000

```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movq    $3000, %rsi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Stack Structure



Example: executing `incr`

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

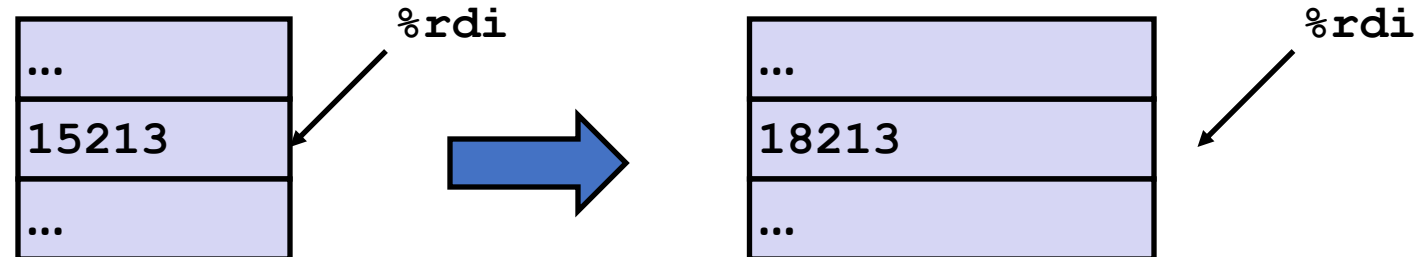
```
incr:  
    movq    (%rdi), %rax  
    addq    %rax, %rsi  
    movq    %rsi, (%rdi)  
    ret
```

Register	Use(s)
<code>%rdi</code>	Argument <code>p</code>
<code>%rsi</code>	Argument <code>val</code> (3000)
<code>%rax</code>	...



Register	Use(s)
<code>%rdi</code>	Argument <code>p</code>
<code>%rsi</code>	18213 (overwritten, that's fine)
<code>%rax</code>	15213 (return value)

Memory



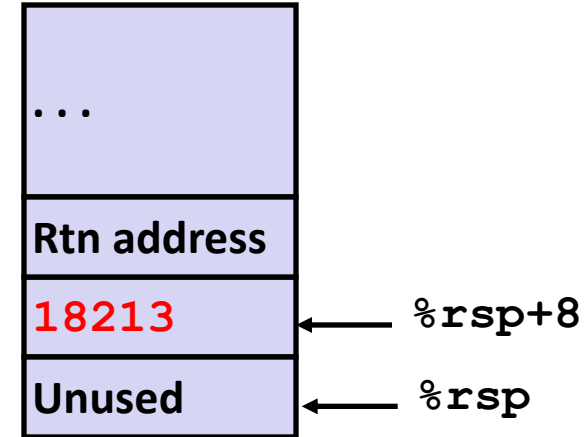
Example: right after executing `incr`

```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movq    $3000, %rsi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```



Stack Structure



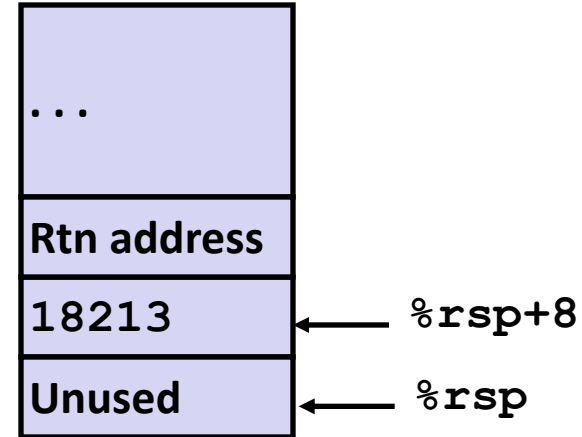
Register	Use(s)
%rdi	&v1
%rsi	18213
%rax	15213

QUIZ: where do we find
the return value of `incr`?

Example: Calling `incr` #4 (do the addition)

Previous stack Structure

```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    → return v1+v2;  
}
```



```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movq    $3000, %rsi  
    leaq    8(%rsp), %rdi  
    call    incr  
    → addq   8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

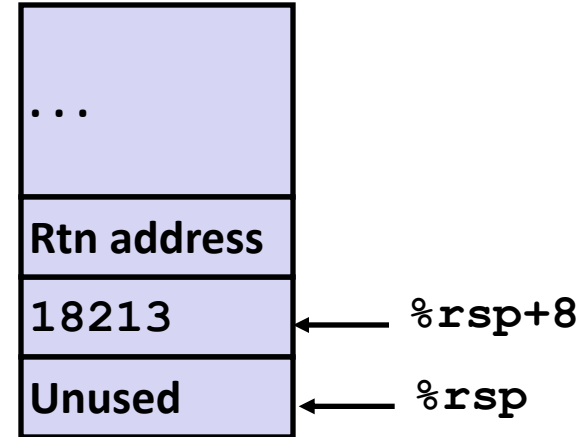
Register	Use(s)
%rax	Return value

Example: Calling `incr` #5 (cleanup)

```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    → return v1+v2;  
}
```

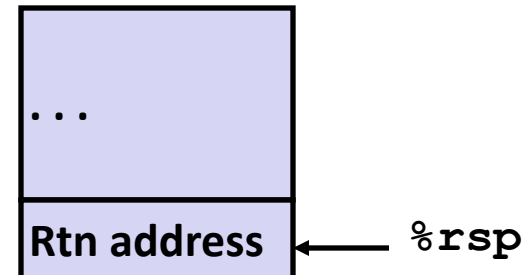
```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movq    $3000, %rsi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    → addq    $16, %rsp  
    ret
```

Previous stack Structure



Register	Use(s)
<code>%rax</code>	Return value

Updated Stack Structure

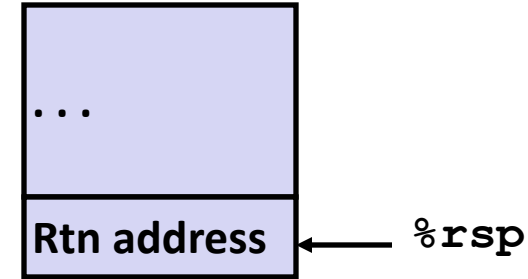


Example: Calling `incr` #6

```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    → return v1+v2;  
}
```

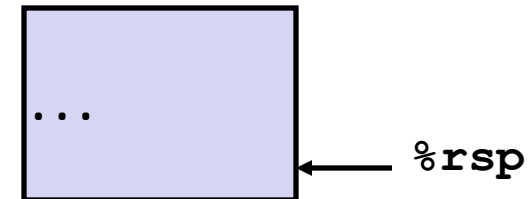
```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movq    $3000, %rsi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    → ret
```

Updated Stack Structure



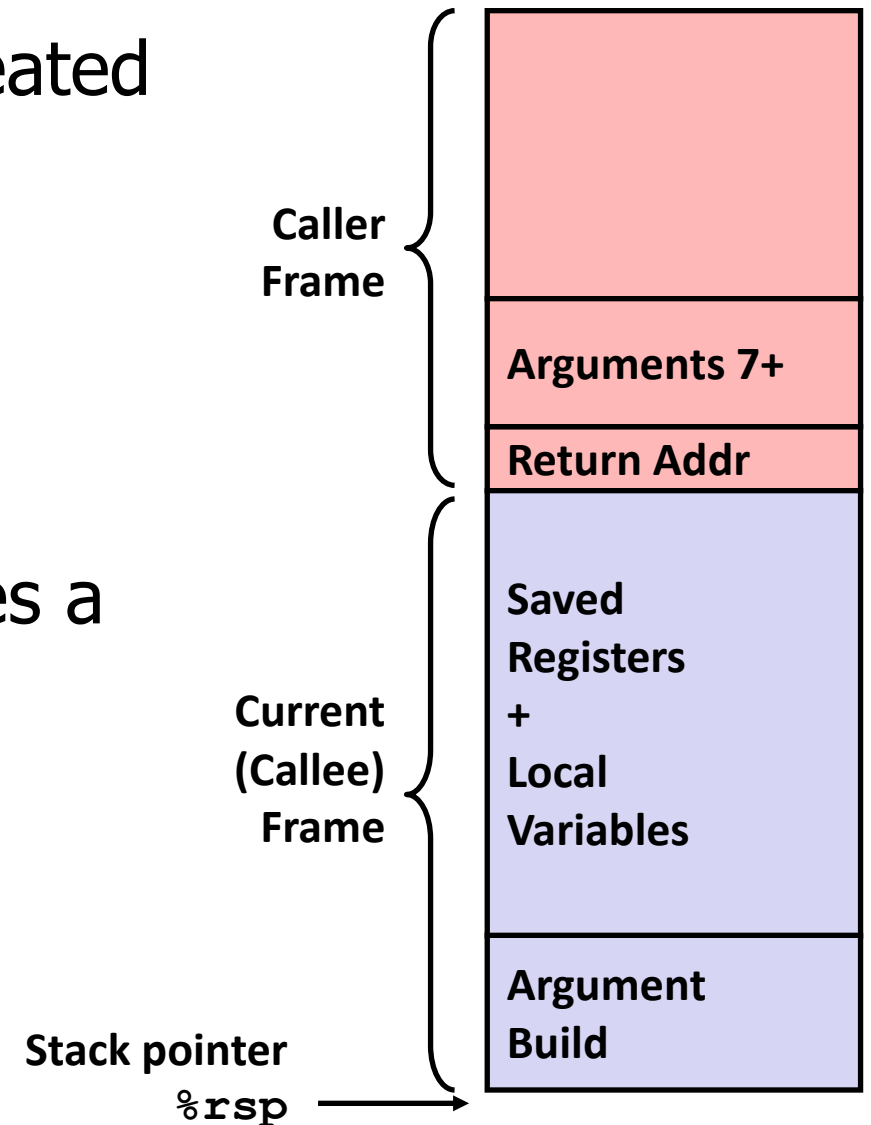
Register	Use(s)
%rax	Return value

Final Stack Structure



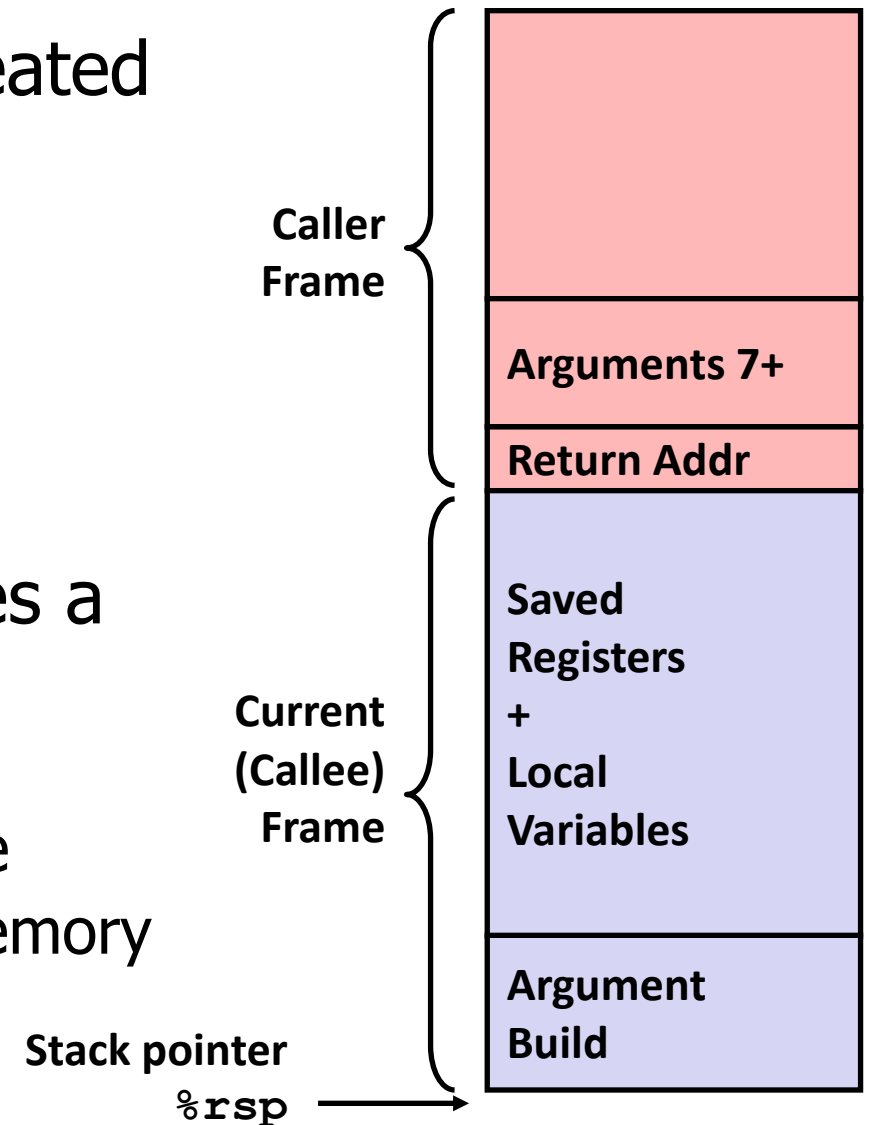
Break + Open Questions

- What are the initial values of variables created on the stack?
- Is there a limit to how many local variables a function can have?



Break + Open Questions

- What are the initial values of variables created on the stack?
 - Undefined behavior in C
 - Machine just creates a variable in the stack
 - Initial value is whatever was there before
- Is there a limit to how many local variables a function can have?
 - Based on memory limit of the process
 - Stack keeps growing until it runs out of space
 - OS can do lots of tricks to give it more memory



Outline

- C Code Layout
- x86-64 Calling Convention
- Managing Local Data
- **Register Saving**
 - Recursion Example

Register Saving

- Can a function use `%rdx` for temporary storage?

Caller

```
yo:
    . . .
    movq $15213, %rdx
    call who
    addq %rdx, %rax
    . . .
    ret
```

Callee

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

- Contents of register `%rdx` overwritten by `who`!
- This could be trouble → something should be done!
 - Need some coordination

Reusing registers

- Problem: registers are shared between functions
 - Callee could overwrite caller's registers by accident
 -  (function that's run)
 -  (code that's calling the function)
- How does each function know which registers are safe to use?
- Solution:
 - Save original register value to stack
 - Use register as needed
 - Restore original register value from stack
- New question: *when* should the saving happen? In advance or on demand?

Saving registers in advance

- New question: who should save the registers, Caller or Callee?
- Attempt 1: Save everything in advance
 - Caller knows which registers it is using
 - Before calling a function, save all registers it is going to need after the call
- Downside: Caller doesn't know what Callee needs
 - Wasted stores to memory if Callee doesn't need those registers
- Example: which registers does `printf()` need to use?

Saving registers on demand

- New question: who should save the registers, Caller or Callee?
- Attempt 2: Save everything on demand
 - Callee knows which registers it is using
 - At the start of a function, save all registers it is going to use
- Downside: Callee doesn't know what Caller was using
 - Wasted stores to memory if Caller wasn't using those registers
- Example: which registers does code that calls `printf()` use?

Compromise: some registers in advance, some on demand

- Neither the Caller nor the Callee has perfect knowledge of register usage/availability
- Designate certain registers are saved in certain way
 - Some are saved in advance: Caller saved
 - Some are saved on demand: Callee saved
- Remember: Caller and Callee are just designations for one call event
 - Functions can and do act as both at different times
 - If $A()$ calls $B()$ calls $C()$, then $B()$ is both Callee of A and Caller of C

Full Rules for Register Saving

1. Does the function use any callee-saved (on-demand) registers?
 - They MUST be saved before use and restored before returning

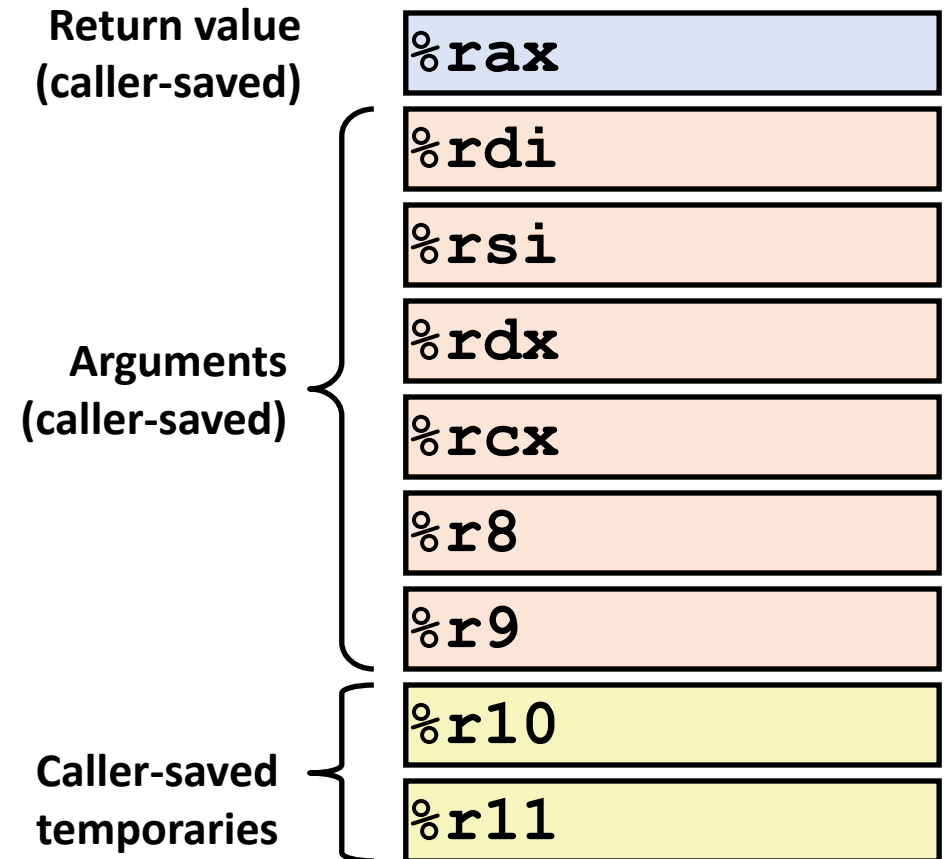
2. Does the code call any functions?
 - If no, you're done

 - If yes: do any caller-saved (in-advance) registers need to keep their original value after the function call returns?
 - If no, you're done (if we don't need the registers we don't save them)

 - If yes, save them before the function call and restore them after it

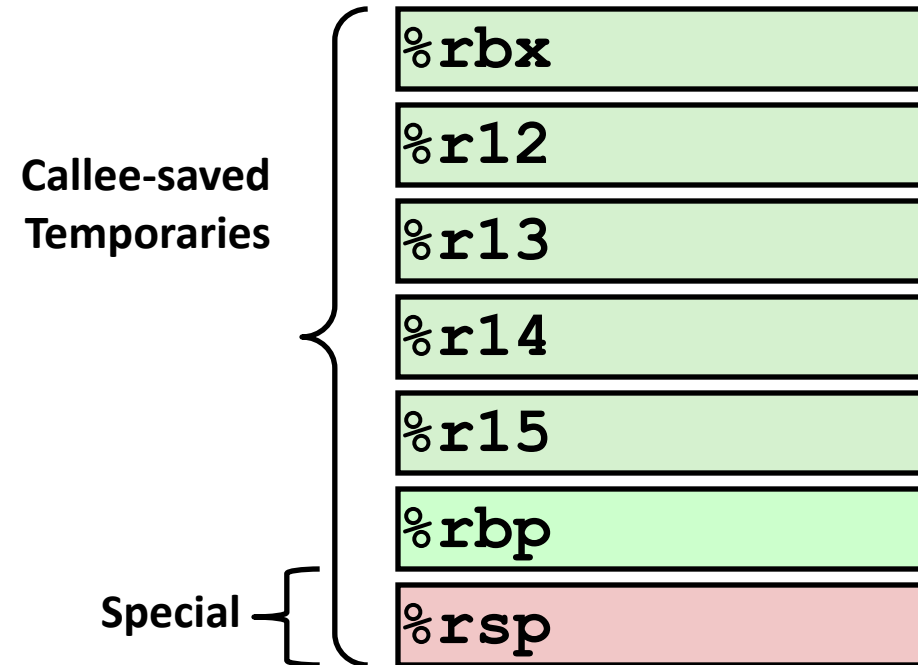
x86-64 Linux Register Usage #1 (caller-saved, in advance)

- **%rax**
 - Return value
 - Caller-saved
 - **Will** be modified by function we're about to call
- **%rdi, ..., %r9**
 - Arguments
 - Caller-saved
 - Could be modified by function we're about to call
- **%r10, %r11**
 - Caller-saved
 - Could be modified by function we're about to call



x86-64 Linux Register Usage #2 (callee-saved, on demand)

- **%rbx, %rbp, %r12-%r15**
 - Callee-saved
 - Every function must save/restore the original values if it wants to use these registers



- **%rsp**
 - Special form of callee-saved
 - Always restored to original value upon exit from procedure
 - The current stack frame is removed before returning

x86-64 Integer Registers: Usage Conventions

Caller Saved

In advance

Callee saved

On demand

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

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

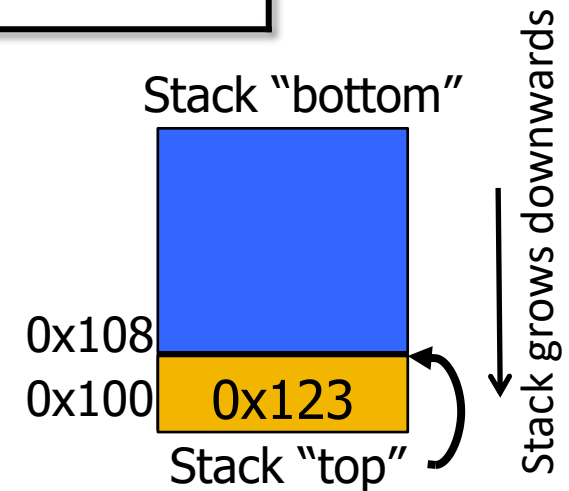
Push and Pop instructions

Instruction	Effect	Description
<code>pushq S</code>	$R[\%rsp] \leftarrow R[\%rsp] - 8;$ $M[R[\%rsp]] \leftarrow S$	Store S onto the stack
<code>popq D</code>	$D \leftarrow M[R[\%rsp]]$ $R[\%rsp] \leftarrow R[\%rsp] + 8;$	Retrieve D from the stack

- Example:

`%rax = 0x123, %rdx = 0x0, %rsp = 0x108`

<code>pushq %rax</code>	<code>%rsp = 0x100</code>
<code>popq %rdx</code>	<code>%rdx = 0x123; %rsp = 0x108</code>



- Remember, stack is just memory

- Can also use memory moves and modify `%rsp` manually!
- Functions often mix the two, push some registers and allocate extra space

Saving a register to the stack

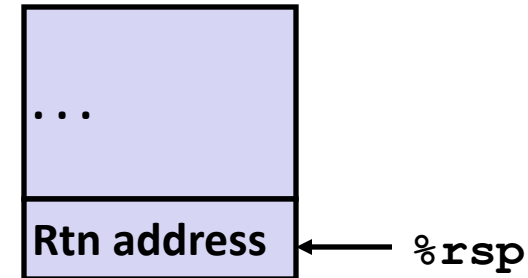
```
long call_incr2(long x) {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return x+v2;  
}
```

↑ Still need **x** after the call!

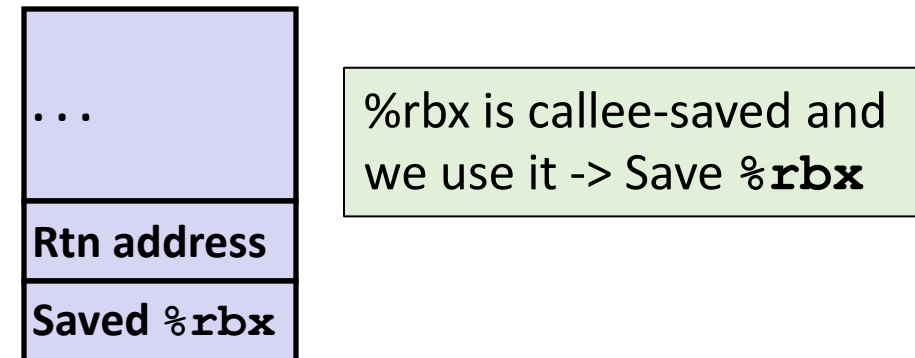
%rbx is callee-save (on demand)

```
call_incr2:  
→ pushq    %rbx  
  subq     $16, %rsp  
  movq     %rdi, %rbx  
  movq     $15213, 8(%rsp)  
  movq     $3000, %rsi  
  leaq     8(%rsp), %rdi  
  call     incr  
  addq     %rbx, %rax  
  addq     $16, %rsp  
  popq     %rbx  
  ret
```

Initial Stack Structure



Resulting Stack Structure



Manually allocating stack space

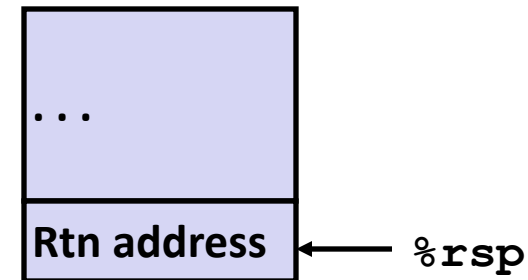
```
long call_incr2(long x) {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return x+v2;  
}
```

↑ Still need **x** after the call!

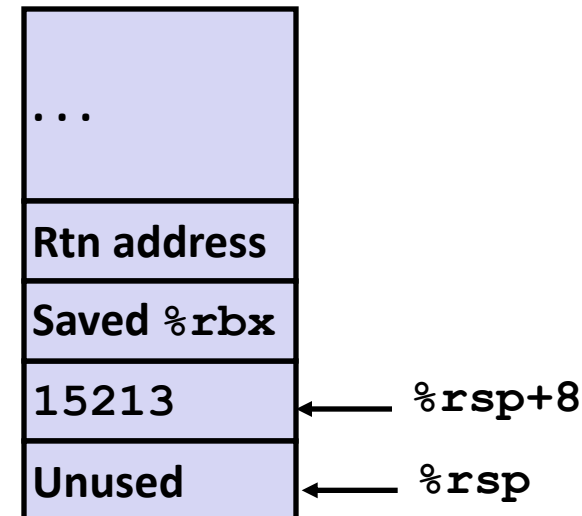
%rbx is callee-save (on demand)

```
call_incr2:  
    pushq    %rbx  
    → subq    $16, %rsp  
    movq     %rdi, %rbx  
    → movq     $15213, 8(%rsp)  
    movq     $3000, %rsi  
    leaq     8(%rsp), %rdi  
    call     incr  
    addq     %rbx, %rax  
    addq     $16, %rsp  
    popq     %rbx  
    ret
```

Initial Stack Structure



Resulting Stack Structure



FYI: Stack moves in multiples of 16 whenever possible.

This accommodates alignment for any 128-byte values on the stack.

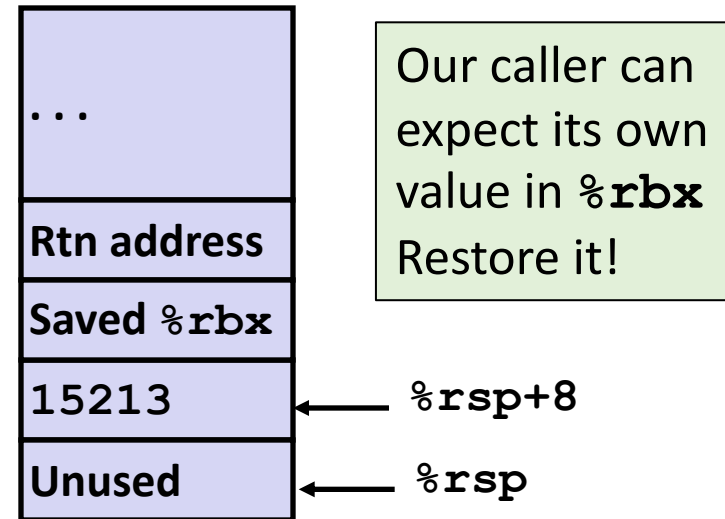
Restoring the stack and register before a return

```
long call_incr2(long x) {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return x+v2;  
}
```

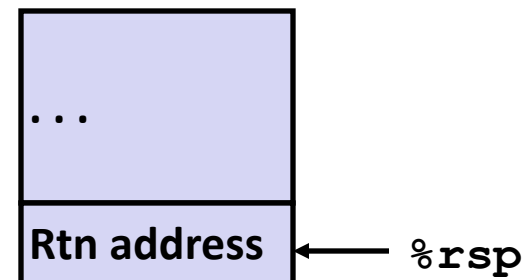
%rbx is callee-save (on demand)

```
call_incr2:  
    pushq    %rbx  
    subq     $16, %rsp  
    movq     %rdi, %rbx  
    movq     $15213, 8(%rsp)  
    movq     $3000, %rsi  
    leaq     8(%rsp), %rdi  
    call     incr  
    addq     %rbx, %rax  
    → addq    $16, %rsp  
    → popq    %rbx  
    ret
```

Resulting Stack Structure



Pre-return Stack Structure



Outline

- C Code Layout
- x86-64 Calling Convention
- Managing Local Data
- **Register Saving**
 - **Recursion Example**

Recursive Function

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

```
pcount_r:
    movq    $0, %rax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andq    $1, %rbx
    shrq    %rdi # (by 1)
    callq   pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```

Note: `rep` instruction inserted as no-op. You can ignore it.

Recursive Function Base Case

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
→  if (x == 0)
→  return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

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

pcount_r:

```
movq    $0, %rax
testq   %rdi, %rdi
je      .L6
```

Checks if
%rdi is zero

```
pushq   %rbx
movq     %rdi, %rbx
andq     $1, %rbx
shrq     %rdi # (by 1)
callq    pcount_r
addq     %rbx, %rax
popq     %rbx
```

.L6:

```
rep; ret
```

Recursive Function Register Save

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

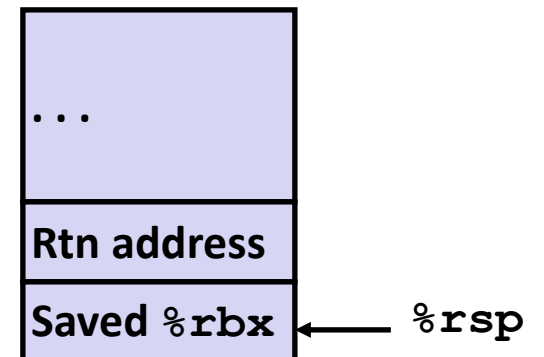
Register	Use(s)	Type
%rdi	x	Argument

pcount_r:

```
    movq    $0, %rax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andq    $1, %rbx
    shrq    %rdi # (by 1)
    callq   pcount_r
    addq    %rbx, %rax
    popq    %rbx
```

.L6:

```
    rep; ret
```



Recursive Function Call Setup

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) ← ↓
               + pcount_r(x >> 1);
}
```

Register	Use(s)	Type
%rdi	x >> 1	Rec. argument
%rbx	x & 1	Callee-saved

```
pcount_r:
    movq    $0, %rax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andq    $1, %rbx
    shrq    %rdi # (by 1)
    callq   pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```

Recursive Function Call

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```



```
pcount_r:
    movq    $0, %rax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andq    $1, %rbx
    shrq    %rdi, %rdi, # (by 1)
    callq   pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Recursive call return value	

Recursive Function Result

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```



```
pcount_r:
    movq    $0, %rax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andq    $1, %rbx
    shrq    %rdi # (by 1)
    callq   pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    rep; ret
```

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

Recursive Function Completion

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1)
            + pcount_r(x >> 1);
}
```

Register	Use(s)	Type
%rax	Return value	Return value

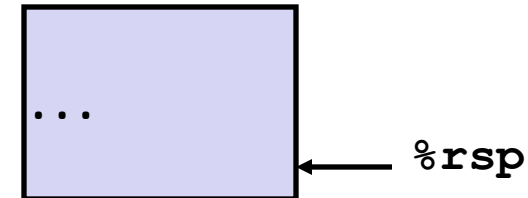
pcount_r:

```
movq    $0, %rax
testq   %rdi, %rdi
je      .L6
pushq   %rbx
movq    %rdi, %rbx
andq    $1, %rbx
shrq    %rdi # (by 1)
callq   pcount_r
addq    %rbx, %rax
```

```
popq    %rbx
```

.L6:

```
rep; ret
```



Example three recursions in

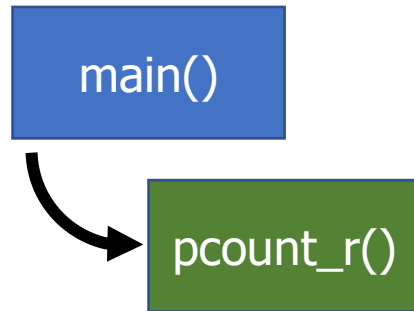
main()

Stack Structure

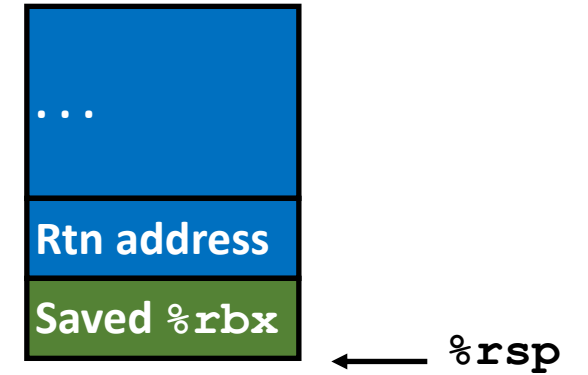


← %rsp

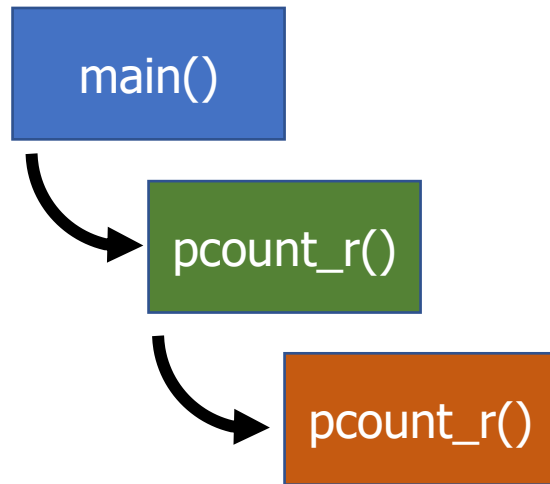
Example three recursions in



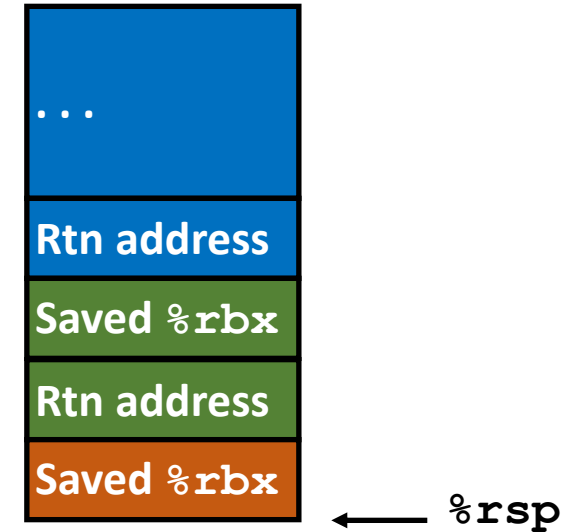
Stack Structure



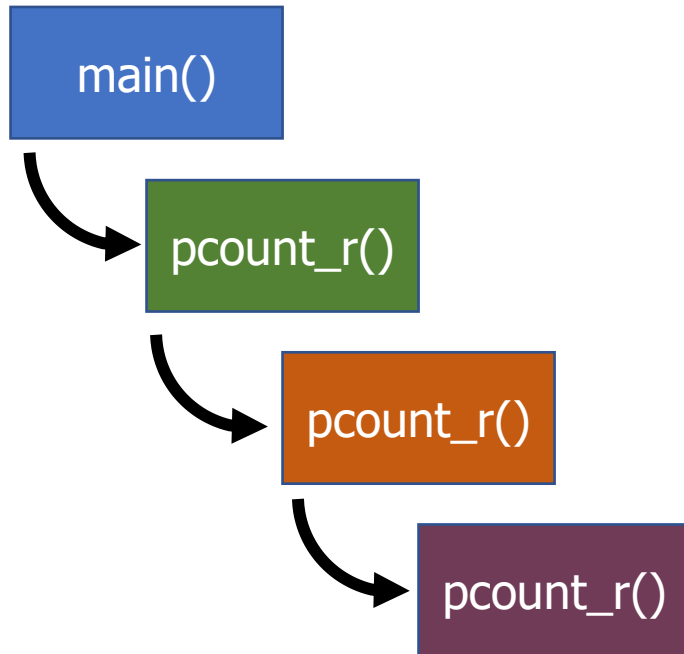
Example three recursions in



Stack Structure

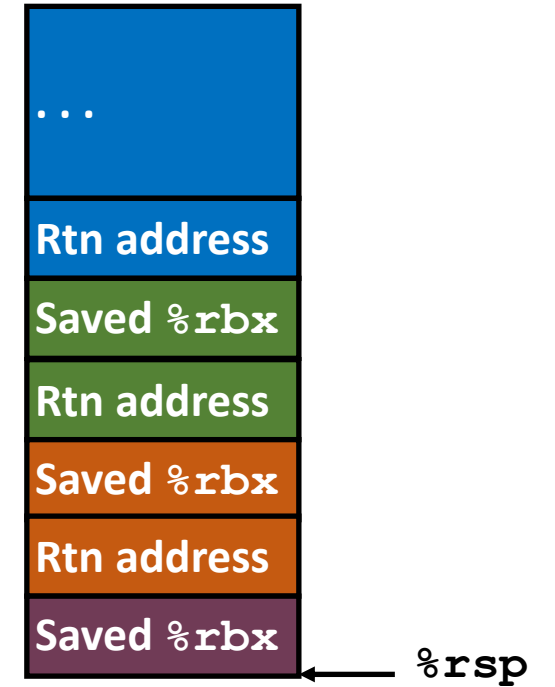


Example three recursions in



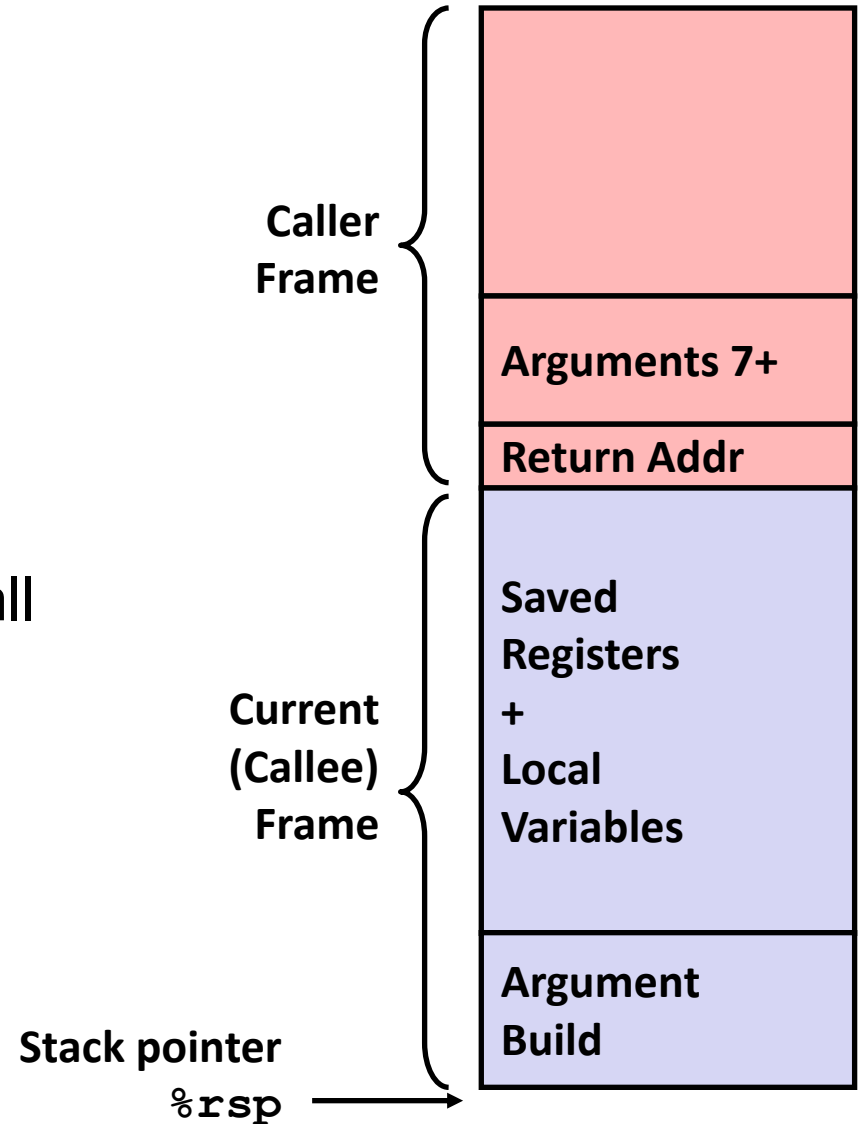
Executing, but has not yet
called `pcount_r()` again

Stack Structure



x86-64 Procedure Summary

- Important Points
 - The stack holds local function data
 - Builds and cleans like a stack data structure
 - The stack makes recursion work
- Calling convention
 - Caller-saved registers saved **in advance** before call
 - Put arguments in registers (1-6)
 - Put further arguments on top of stack (7+)
 - Put return address on top of stack
 - Callee can safely store values in local stack frame and in callee-saved registers (after saving them)
 - Result return in `%rax` and restore callee-saved registers before returning



Outline

- C Code Layout
- x86-64 Calling Convention
- Managing Local Data
- Register Saving
 - Recursion Example

Outline

- Bonus: Stack Frame Example

x86-64 Stack Frame Example

```
long sum = 0;
/* Swap a[i] & a[i+1] */
void
swap_ele_su(long a[], int i)
{
    swap(&a[i], &a[i+1]);
    sum += (a[i]*a[i+1]);
}
```

- Keeps values of `&a[i]` and `&a[i+1]` in callee-save registers
- Must set up stack frame to save these registers

```
swap_ele_su:
    movq    %rbx, -16(%rsp)
    movq    %rbp, -8(%rsp)
    subq    $16, %rsp
    movslq   %esi, %rax
    leaq    8(%rdi, %rax, 8), %rbx
    leaq    (%rdi, %rax, 8), %rbp
    movq    %rbx, %rsi
    movq    %rbp, %rdi
    call    swap
    movq    (%rbx), %rax
    imulq   (%rbp), %rax
    addq    %rax, sum(%rip)
    movq    (%rsp), %rbx
    movq    8(%rsp), %rbp
    addq    $16, %rsp
    ret
```

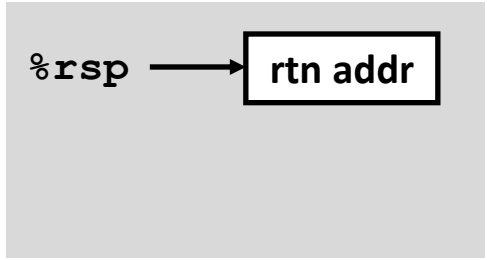
Understanding x86-64 Stack Frame

swap ele su:

movq %rbx, -16(%rsp)	# Save %rbx
movq %rbp, -8(%rsp)	# Save %rbp
subq \$16, %rsp	# Allocate stack frame
movslq %esi, %rax	# Extend i
leaq 8(%rdi, %rax, 8), %rbx	# &a[i+1] (callee save)
leaq (%rdi, %rax, 8), %rbp	# &a[i] (callee save)
movq %rbx, %rsi	# 2 nd argument
movq %rbp, %rdi	# 1 st argument
call swap	
movq (%rbx), %rax	# Get a[i+1]
imulq (%rbp), %rax	# Multiply by a[i]
addq %rax, sum(%rip)	# Add to sum
movq (%rsp), %rbx	# Restore %rbx
movq 8(%rsp), %rbp	# Restore %rbp
addq \$16, %rsp	# Deallocate frame
ret	

Understanding x86-64 Stack Frame

```
movq    %rbx, -16(%rsp)    # Save %rbx
movq    %rbp, -8(%rsp)     # Save %rbp
```



The diagram shows a grey rectangular area. On the left, the text `%rsp` is followed by a horizontal arrow pointing to a white rectangular box with a black border. Inside the box, the text `rtn addr` is written.

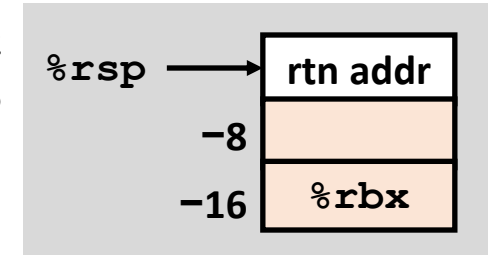
```
subq    $16, %rsp         # Allocate stack frame
```

● ● ●

```
movq    (%rsp), %rbx      # Restore %rbx
movq    8(%rsp), %rbp     # Restore %rbp
addq    $16, %rsp         # Deallocate frame
```

Understanding x86-64 Stack Frame

→ **movq** **%rbx, -16(%rsp)** # Save %rbx
movq **%rbp, -8(%rsp)** # Save %rbp



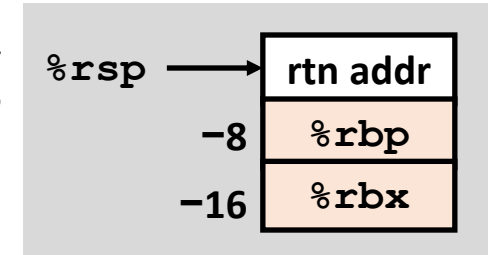
subq **\$16, %rsp** # Allocate stack frame

• • •

movq **(%rsp), %rbx** # Restore %rbx
movq **8(%rsp), %rbp** # Restore %rbp
addq **\$16, %rsp** # Deallocate frame

Understanding x86-64 Stack Frame

→ `movq %rbx, -16(%rsp)` # Save %rbx
→ `movq %rbp, -8(%rsp)` # Save %rbp



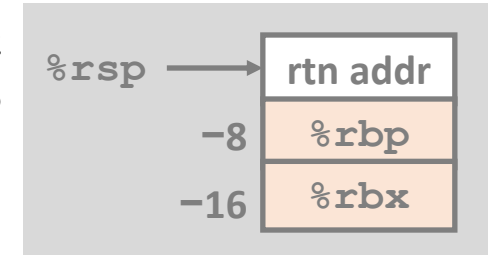
`subq $16, %rsp` # Allocate stack frame

• • •

`movq (%rsp), %rbx` # Restore %rbx
`movq 8(%rsp), %rbp` # Restore %rbp
`addq $16, %rsp` # Deallocate frame

Understanding x86-64 Stack Frame

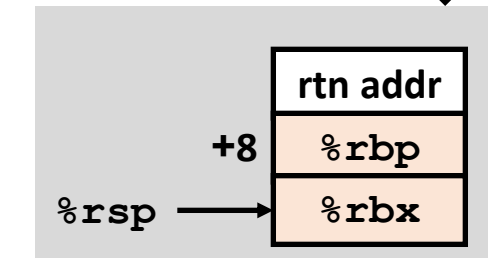
```
movq    %rbx, -16(%rsp)    # Save %rbx
movq    %rbp, -8(%rsp)     # Save %rbp
```



→ **subq \$16, %rsp**

Allocate stack frame

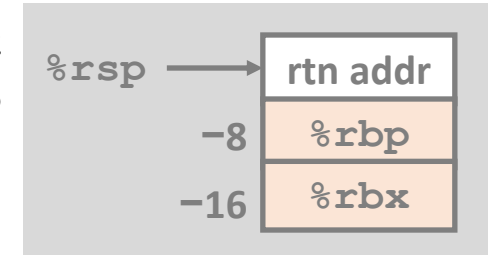
• • •



```
movq    (%rsp), %rbx       # Restore %rbx
movq    8(%rsp), %rbp      # Restore %rbp
addq    $16, %rsp          # Deallocate frame
```

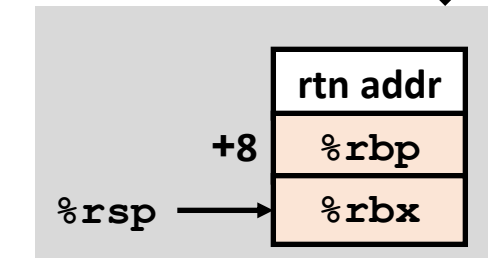
Understanding x86-64 Stack Frame

```
movq    %rbx, -16(%rsp)    # Save %rbx
movq    %rbp, -8(%rsp)     # Save %rbp
```



```
subq    $16, %rsp         # Allocate stack frame
```

...



```
movq    (%rsp), %rbx
movq    8(%rsp), %rbp
addq    $16, %rsp
```

```
# Restore %rbx
# Restore %rbp
# Deallocate frame
```

