Lecture 08
Procedures

CS213 – Intro to Computer Systems

Branden Ghena – Winter 2025

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

Northwestern

Administrivia

- Homework 2 due today
  - Good practice for the exam
  - I’m hoping to post the solutions before the exam if everyone can submit

- Midterm Exam 1:
  - Next week Thursday, during class time in class room
  - Bring a pencil!

  - Bring one 8.5x11 inch sheet of paper with notes on front and back
    - Write down everything you don’t want to memorize

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?

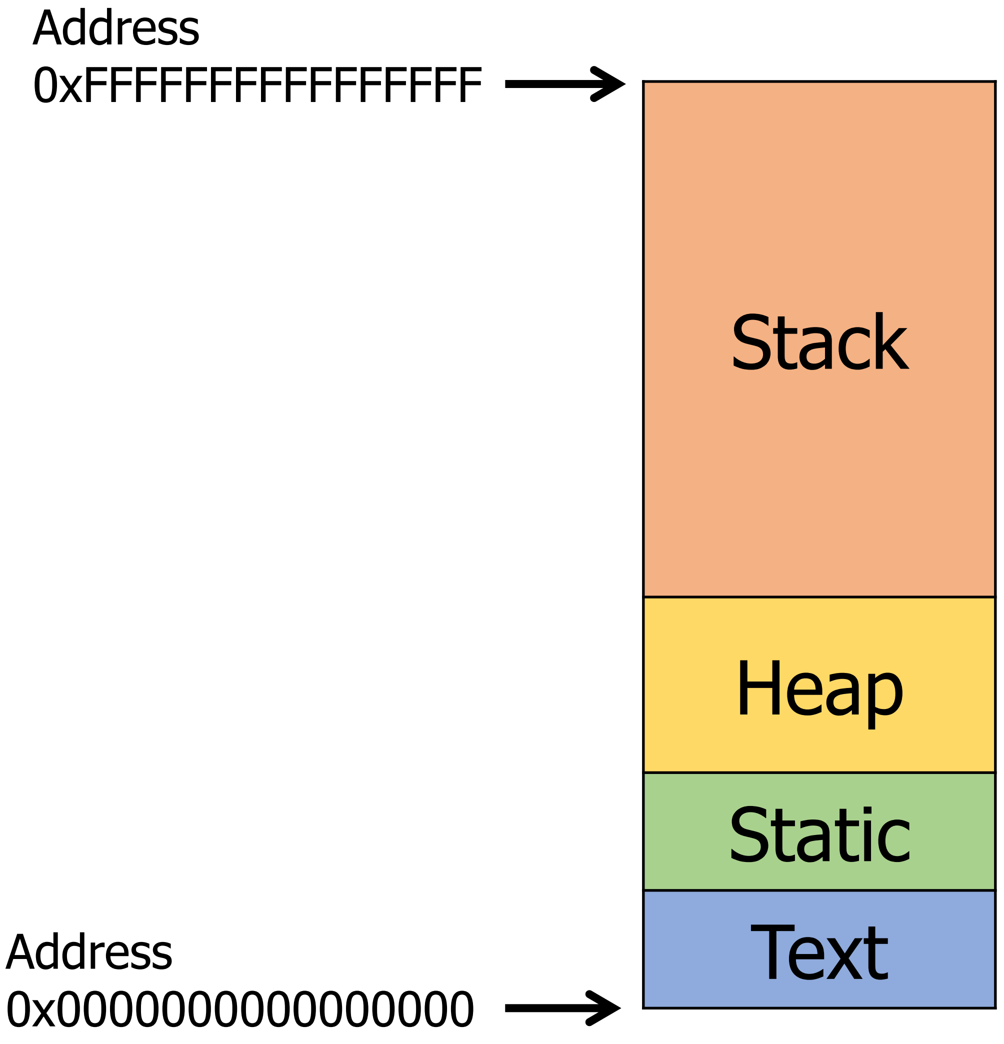
- Understand how we manage register use between functions

Outline

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

C memory layout

- Stack Section
  - Local variables
  - 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

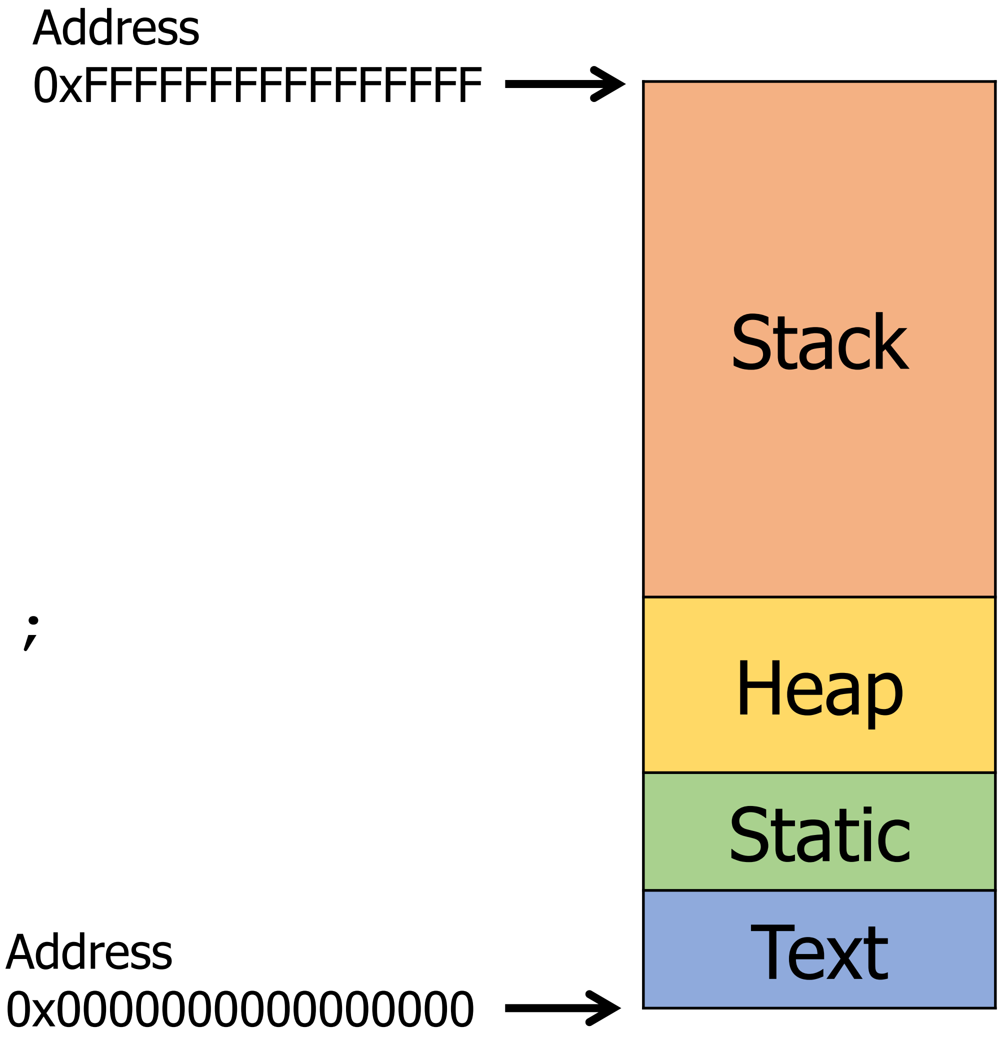


C memory layout

```
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");
}
```

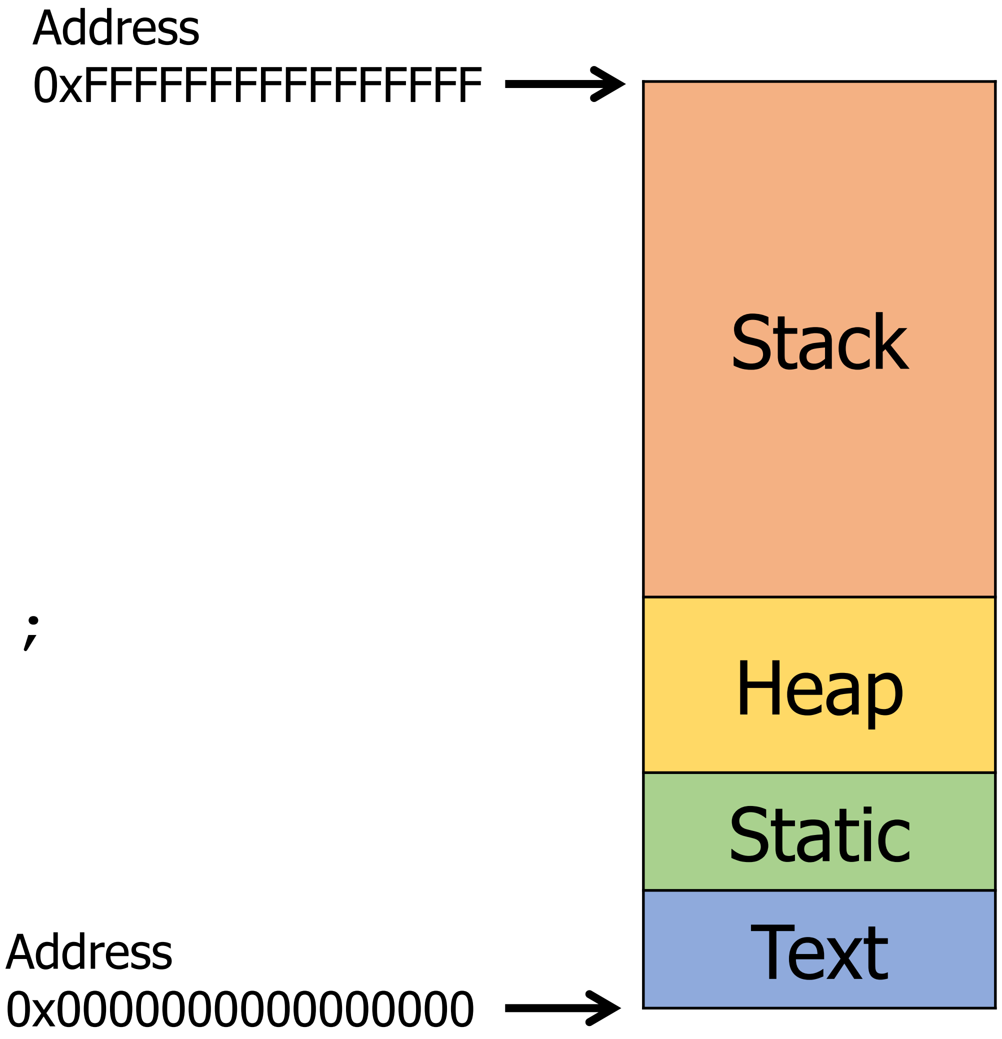


C memory layout

```
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");
}
```

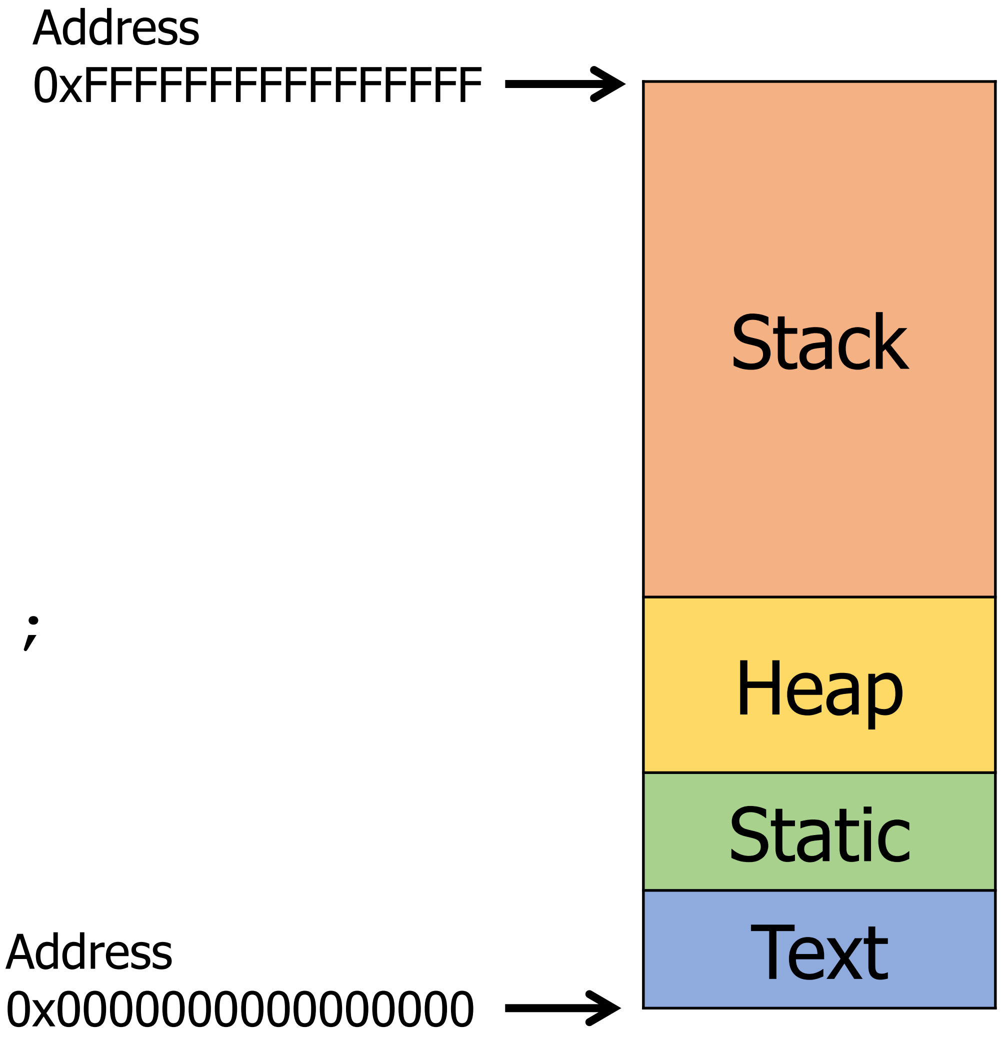


C memory layout

```
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");
}
```

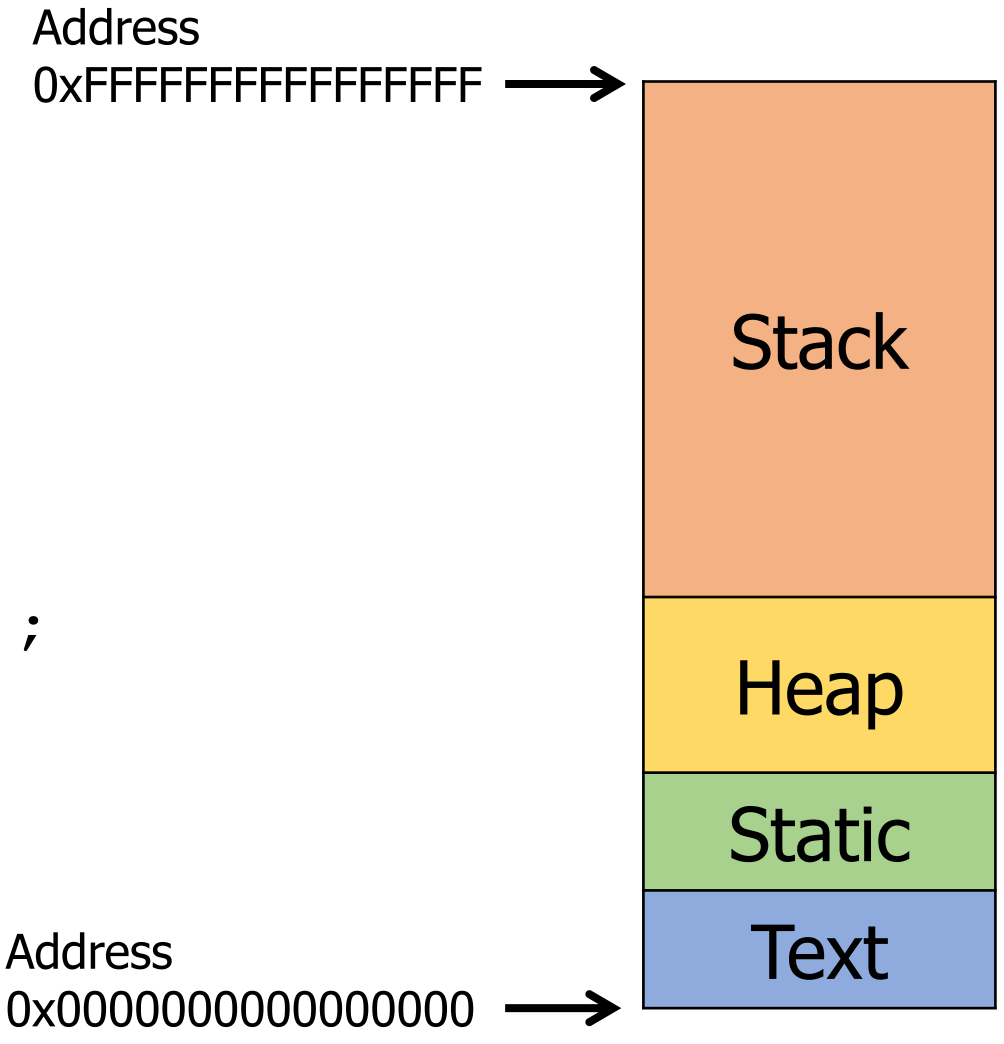


C memory layout

```
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");
}
```

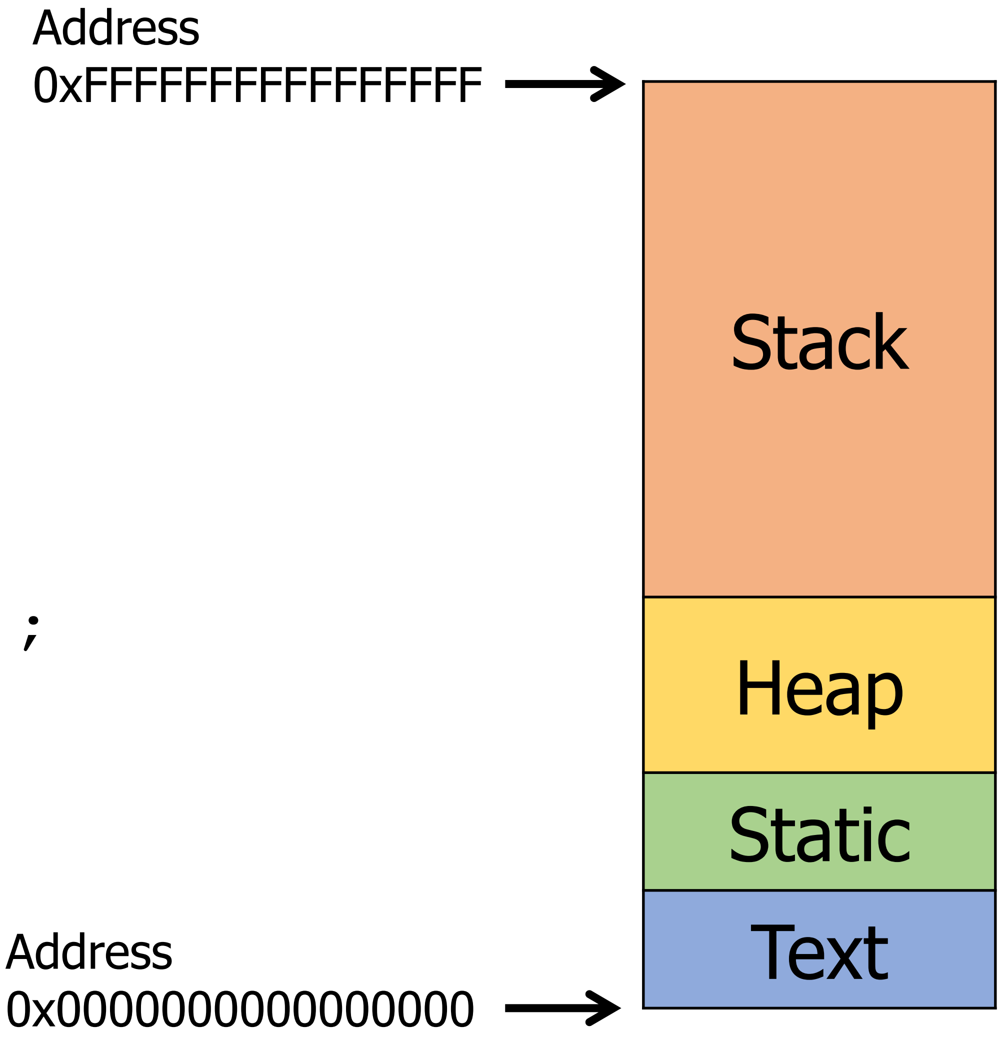


C memory layout

```
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”);
}
```



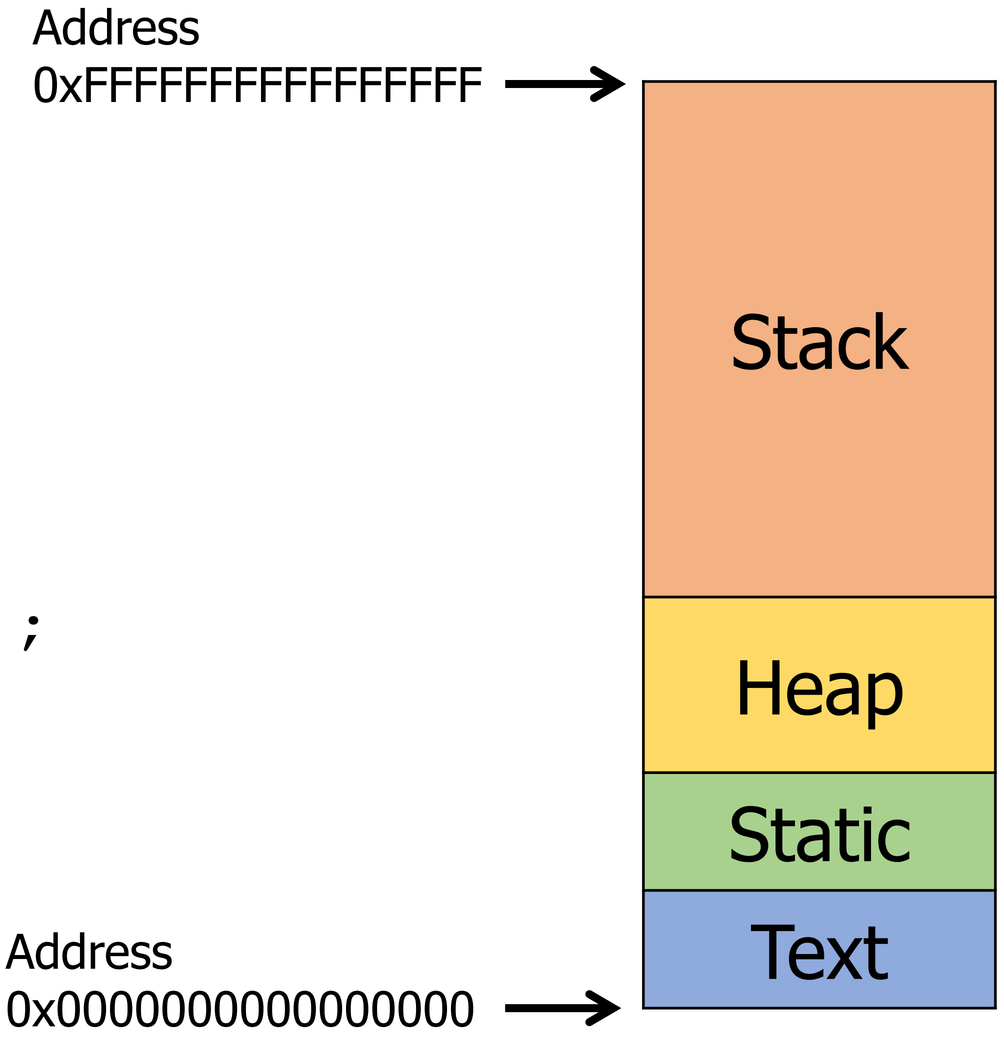
C memory layout

```
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”);

}
```

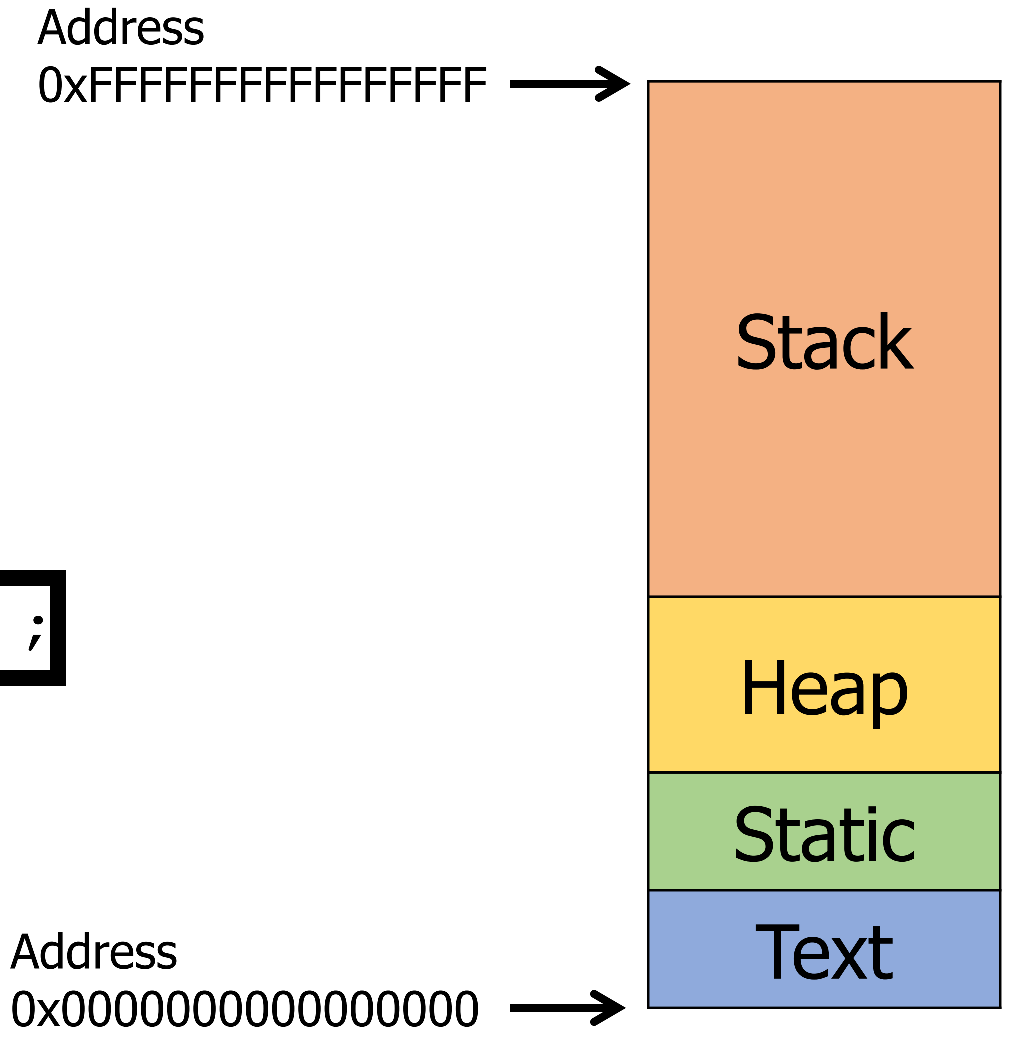


C memory layout

```
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”);
}
```

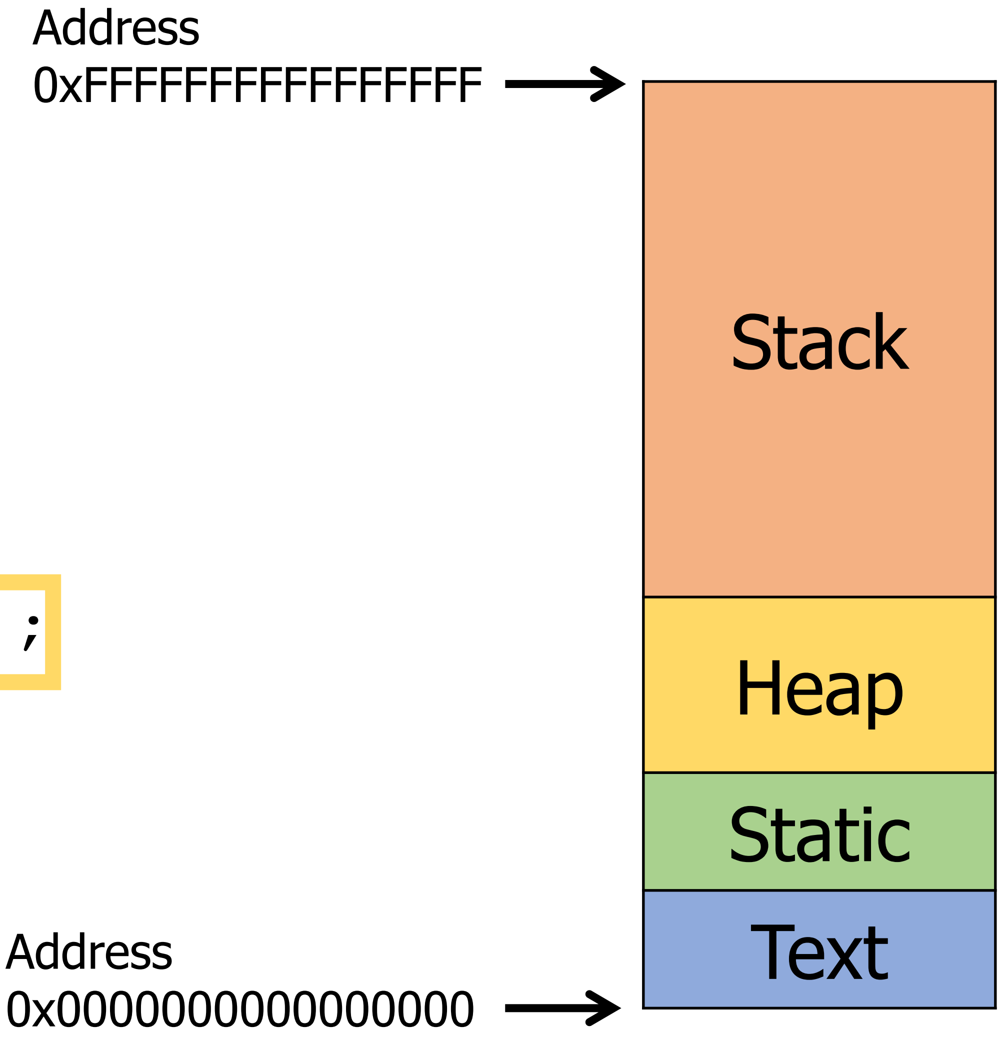


C memory layout

```c
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”);
}
```

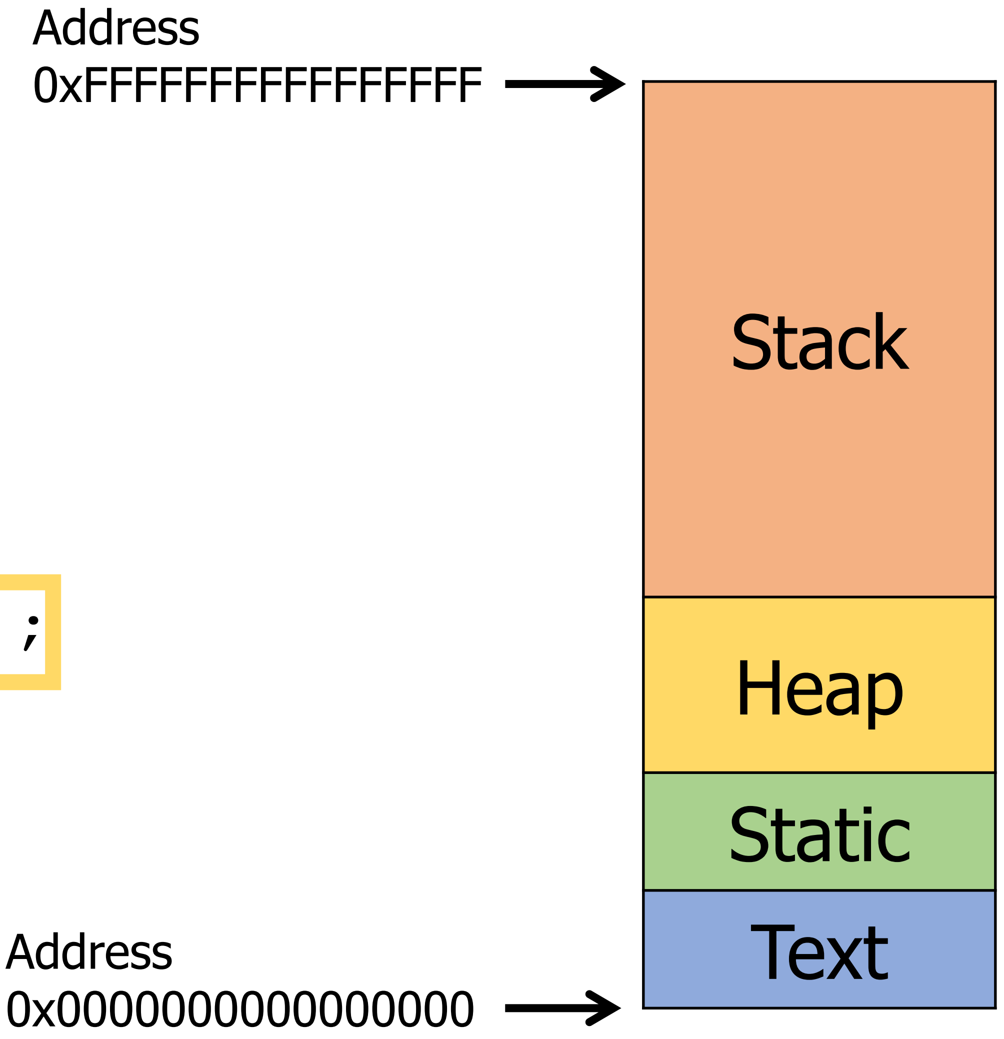


C memory layout

```c
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");
}
```

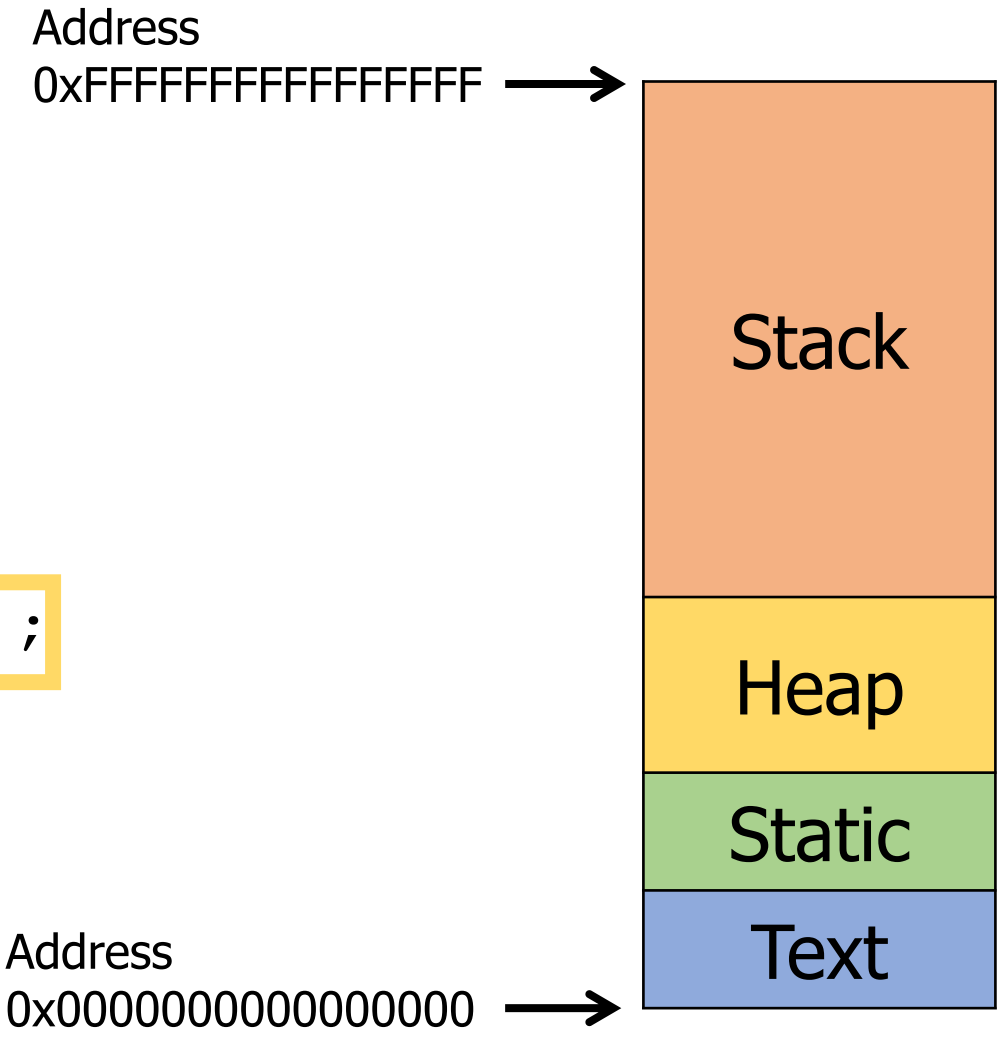


C memory layout

```
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");
}
```



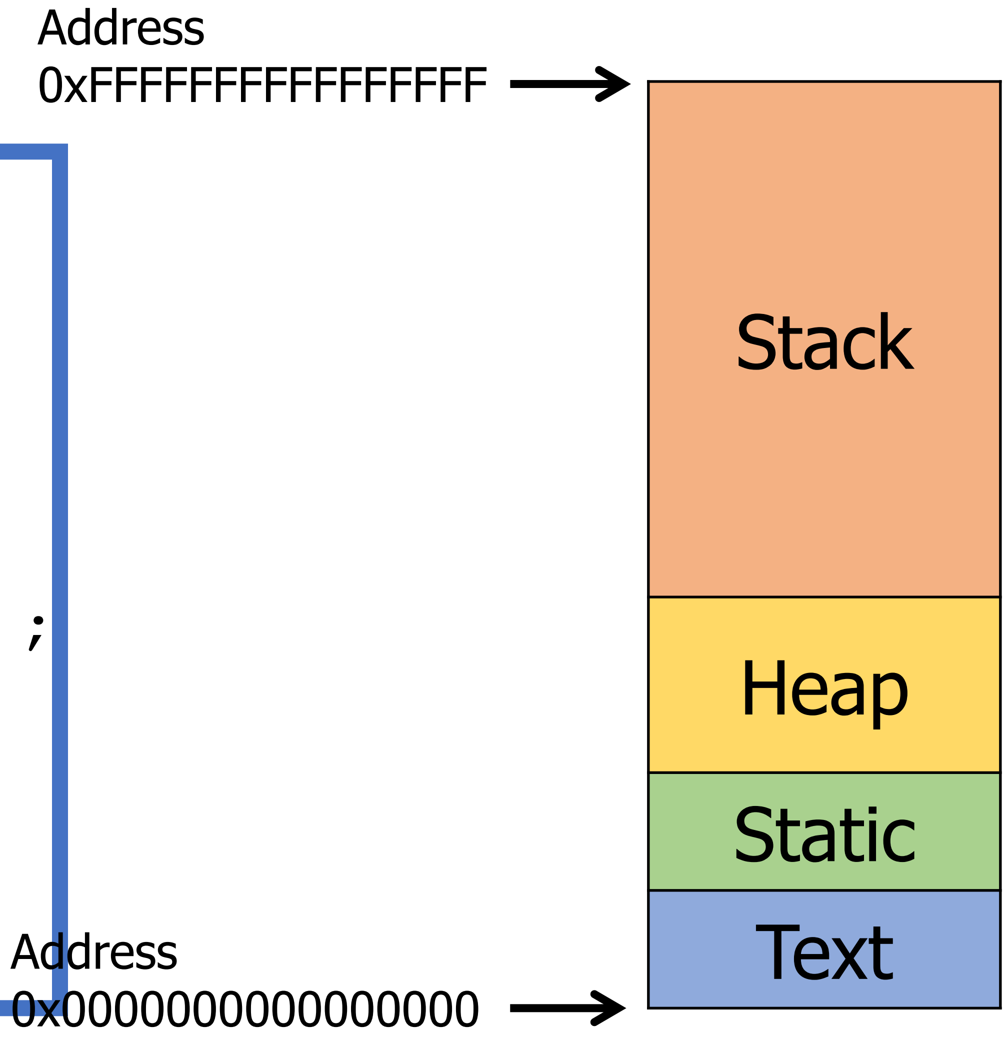
C memory layout

```
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



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
  - Assembly directive can place values into Static section
- 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)

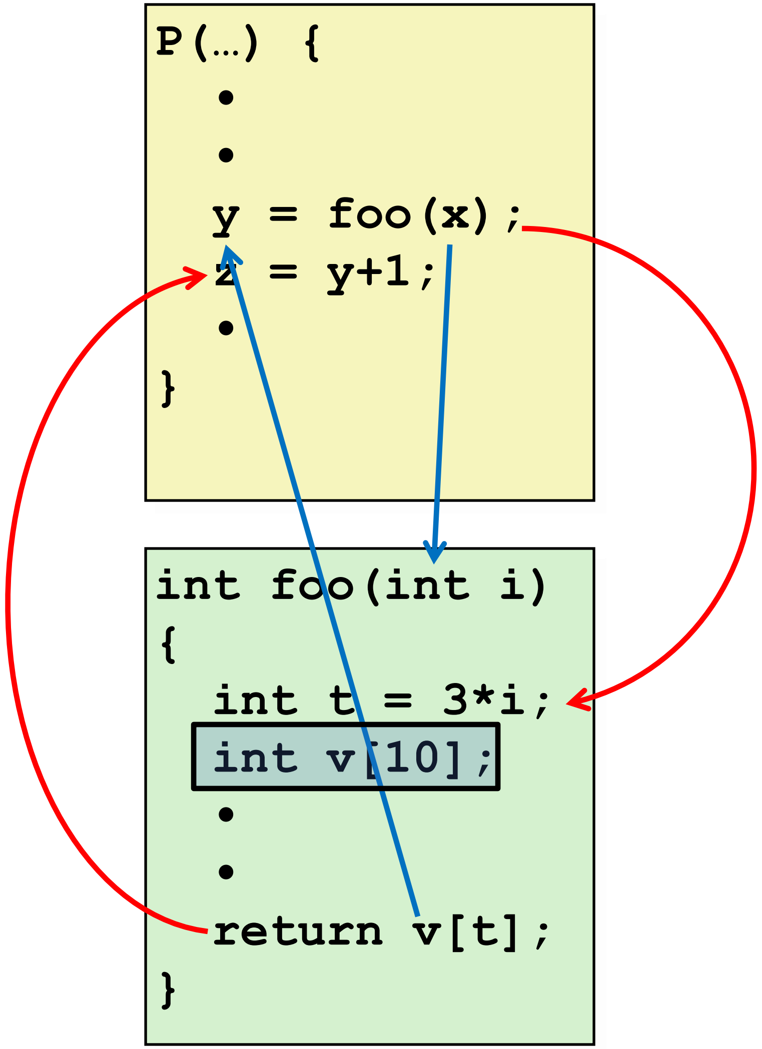
Outline

- 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!



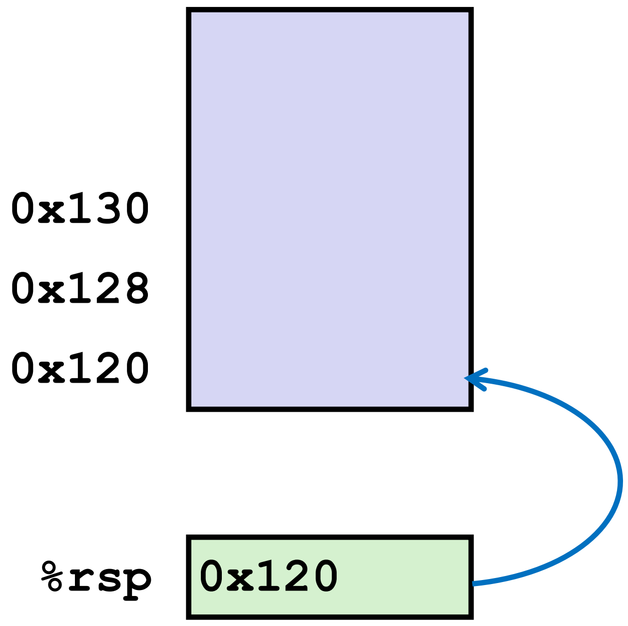
The Stack in Assembly

- Chunk of memory that code can use

- %rsp points at most recently used position

- Stack grows downward
  - Subtract from %rsp to make more space
  - 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)



Procedure control flow

- Use stack to support procedure call and return!
- Procedure call
  `callq` ***`label`*** Push return address on stack; jump to ***`label`***
- Procedure return
  `retq` Pop address from stack; jump there
  (stack should be as it was when the call began)
  - Return value is in `%rax`

If you don't know where you're going, you may not get there.
— Yogi Berra

- Return address value
  - Address of instruction immediately following **`callq`**
  - Example from disassembly

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

Return address: **`0x400549`**

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

Code Examples

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

```
0000000000400540 <multstore>:
  ... (we'll fill the start in soon)
  400541: movq   %rdx,%rbx        # Save dest
  400544: callq  400550 <mult2>   # mult2(x,y)
  400549: movq   %rax,(%rbx)      # Store at *dest
  ... (we'll fill the end in soon too)
  40054d: retq                    # Return
```

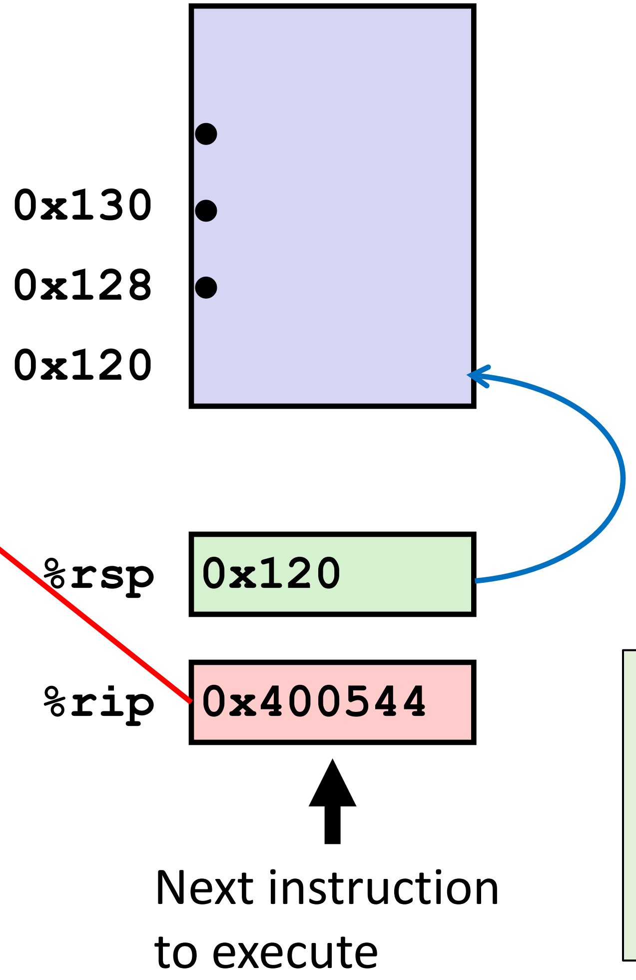
```
long mult2 (long a, long b){
  long s = a * b;
  return s;
}
```

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

Control Flow Example
about to execute `callq`

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

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



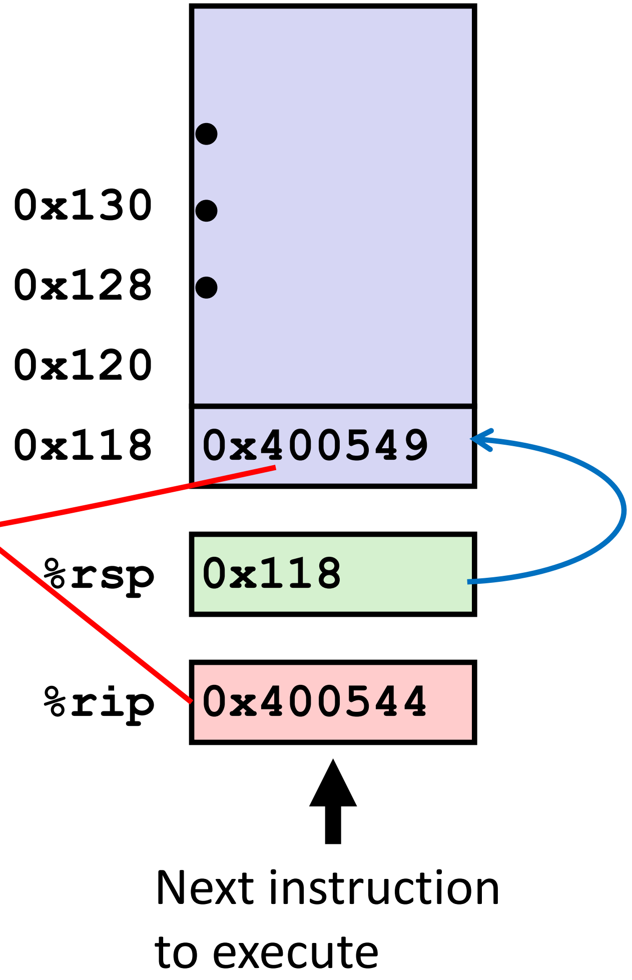
`%rip`: instruction pointer

Can't be directly modified

Control Flow Example
`callq` step 1

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

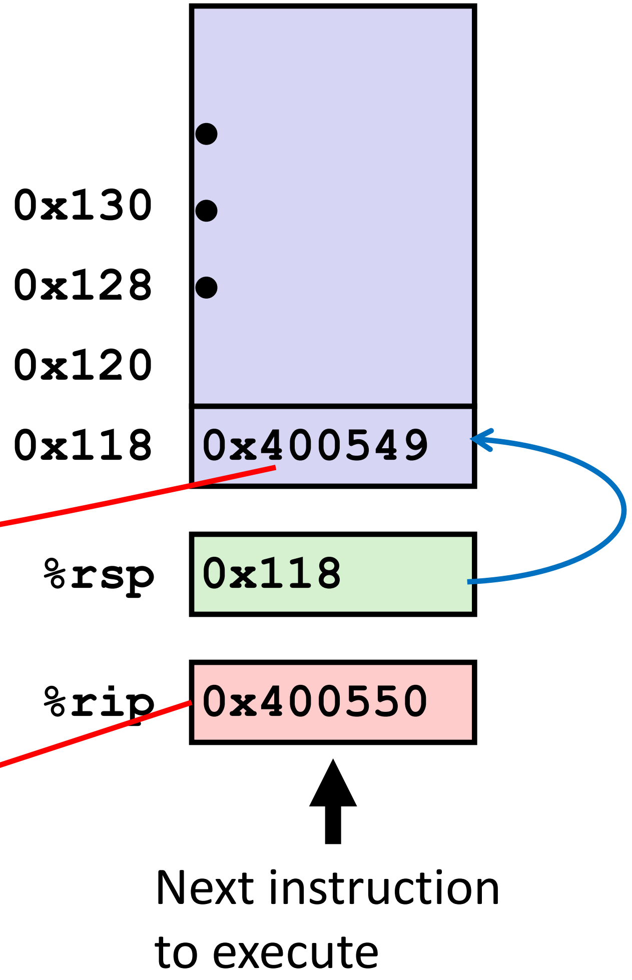
```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example
`callq` step 2

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

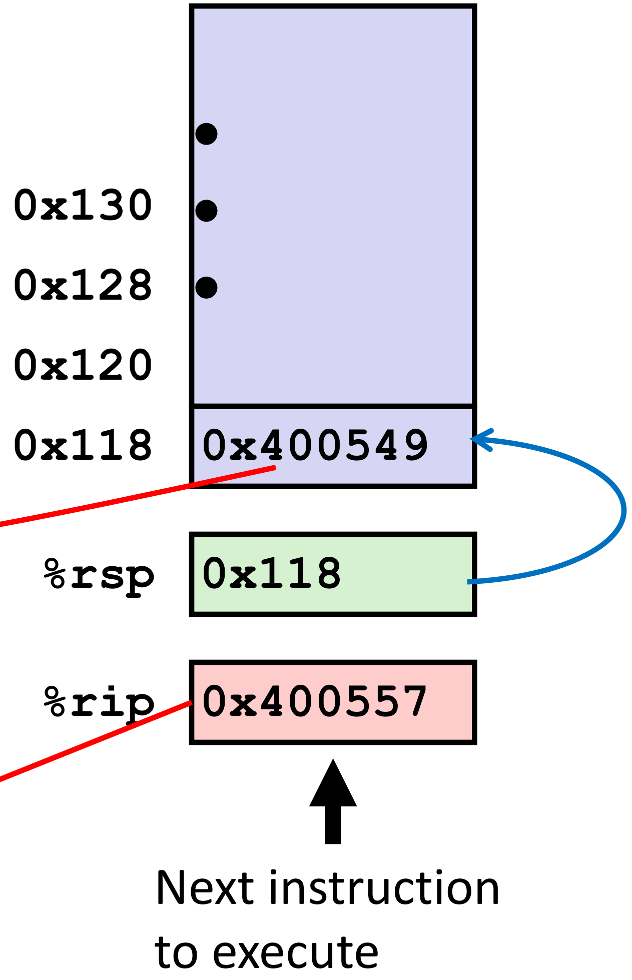
```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example
about to execute `retq`

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

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

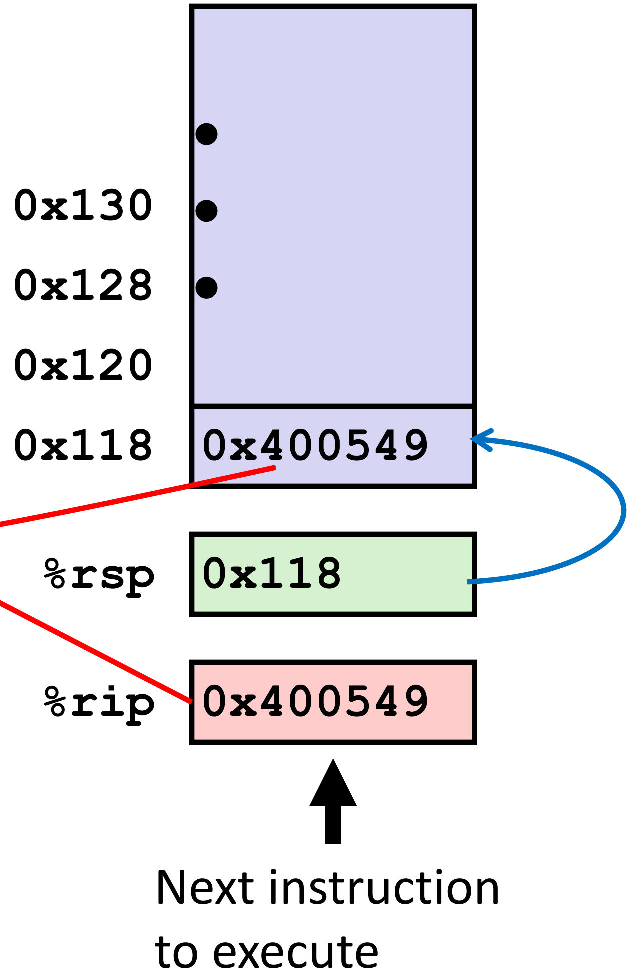


QUIZ: What is the address of the instruction we execute after **`retq`**?

Control Flow Example
`retq` step 1

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

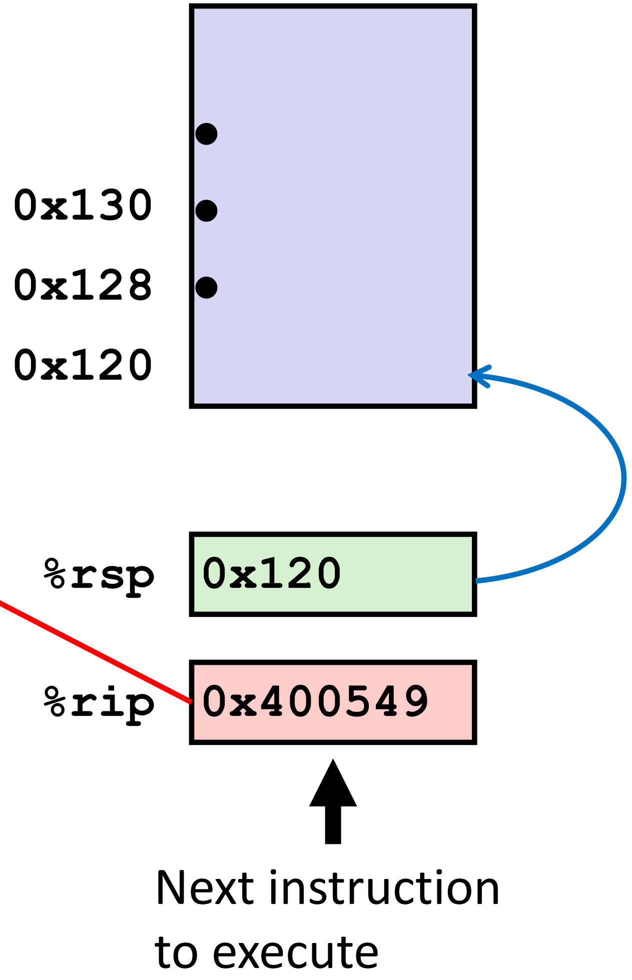
```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example
`retq` step 2

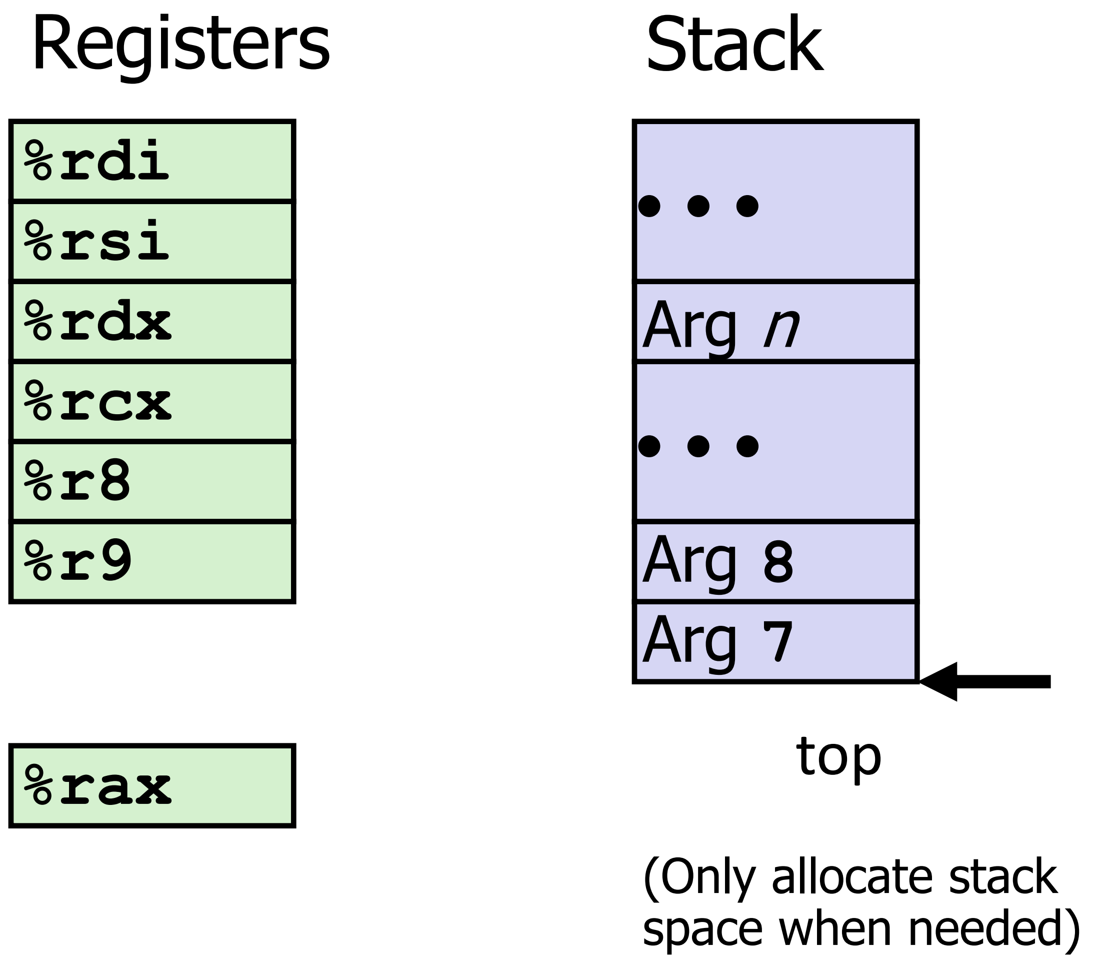
```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: movq   %rax,(%rbx)
  •
  •
```

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



Function data flow

- First 6 arguments are in registers
  - `%rdi` is first argument
- Next **`n`** arguments are on the stack
  - This means more arguments is slower
- Return value is 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: callq  400550 <mult2>     # mult2(x,y)
  # t in %rax
  400549: movq   %rax,(%rbx)        # *dest = t
  • • •
```

```
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:  retq                   # Return
```

Break + Open Question

- How did we decide how many registers to use for arguments and return values?
- Do all functions have to use this same convention?

`%rdi`
`%rsi`
`%rdx`
`%rcx`
`%r8`
`%r9`

`%rax`

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

`%rdi`
`%rsi`
`%rdx`
`%rcx`
`%r8`
`%r9`

`%rax`

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

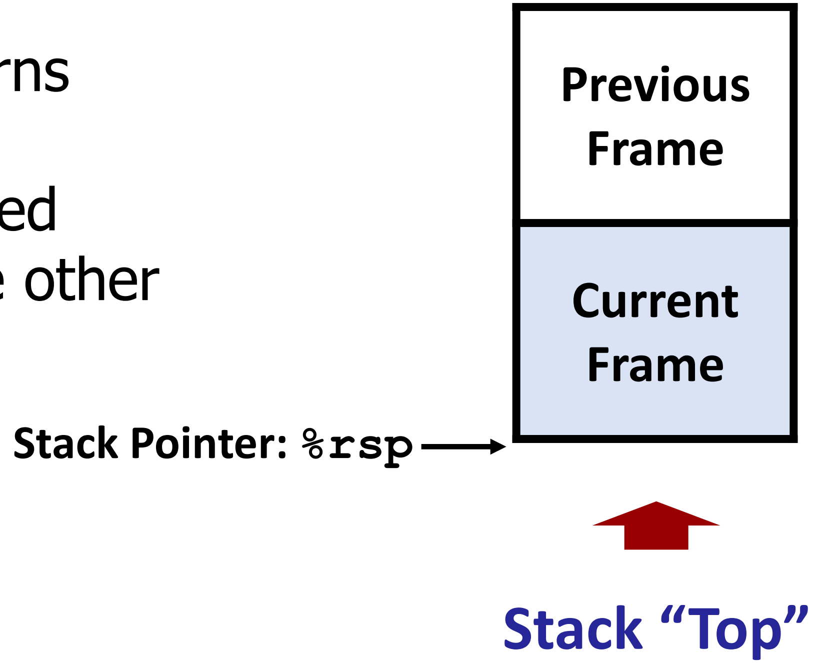
- State only needs to exist until the function returns

Using the Stack for Call-Local State

- Place local state on the stack

- Stack discipline
  - That state is only needed for limited time
    - Starts when function is called; ends when it returns
  - ***Callee*** returns before ***caller*** does
    - ***Callee***: <u>for a specific call</u>, the function being called
    - ***Caller***: <u>for a specific call</u>, the function calling the other

- 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



Call Chain Example

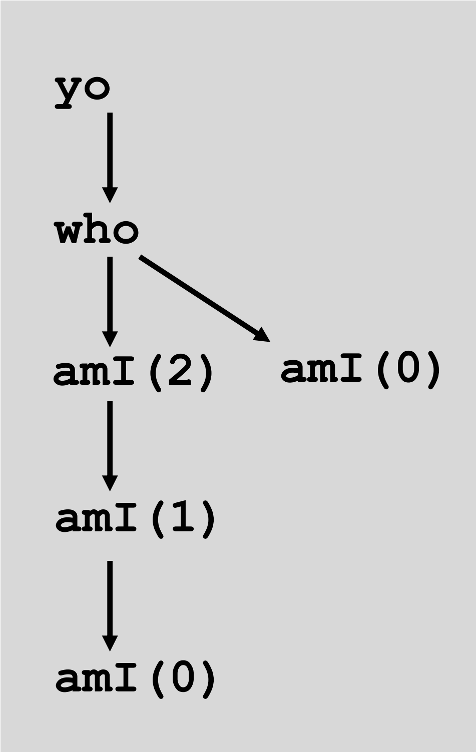
```
yo(…)
{
  •
  •
  who();
  •
  •
}
```

```
who(…)
{
  • • •
  amI(2);
  • • •
  amI(0);
  • • •
}
```

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

Procedure `amI()` is recursive

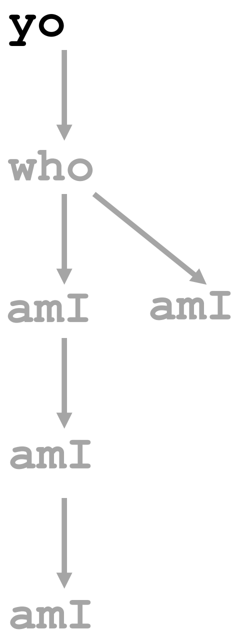
Example Call Chain



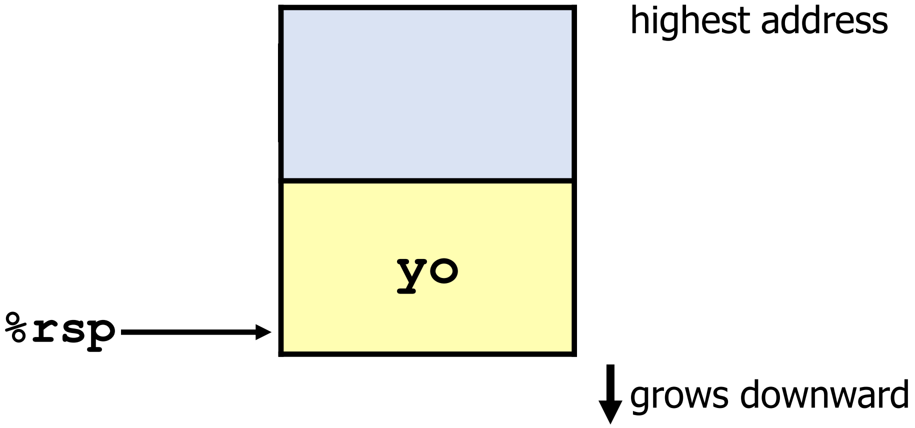
Example

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

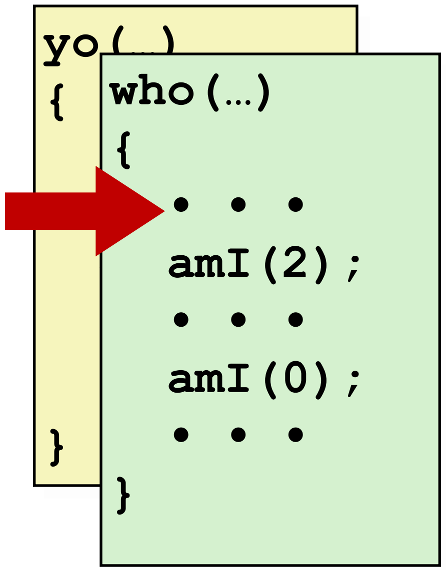
Call Chain



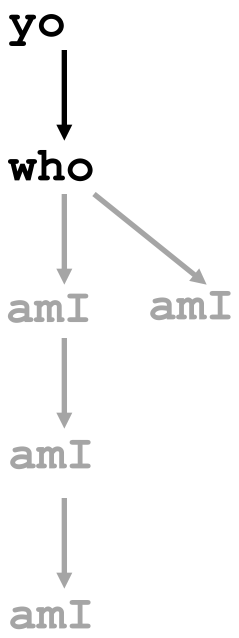
Stack



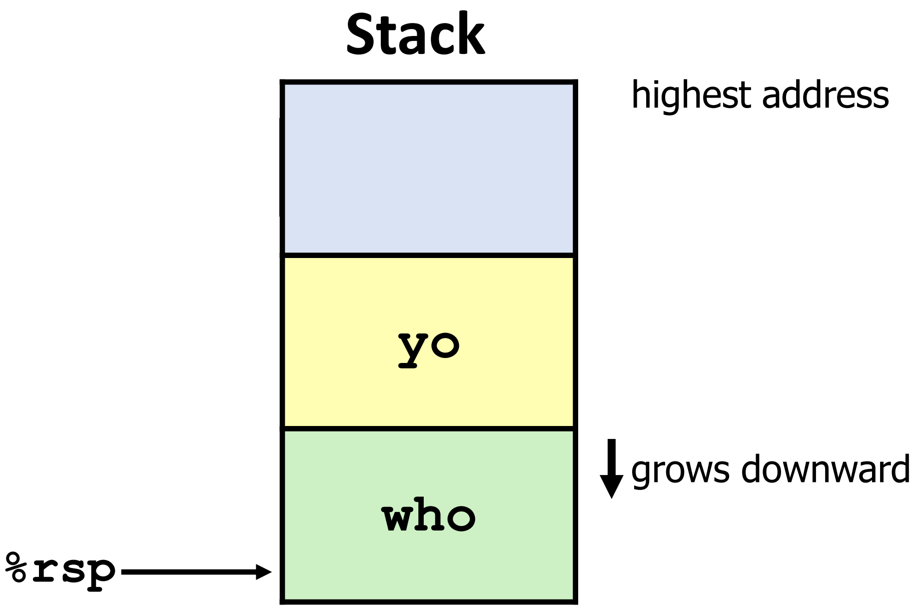
Example



yo(…)
{
}
who(…)
{
• • •
amI(2);
• • •
amI(0);
• • •
}

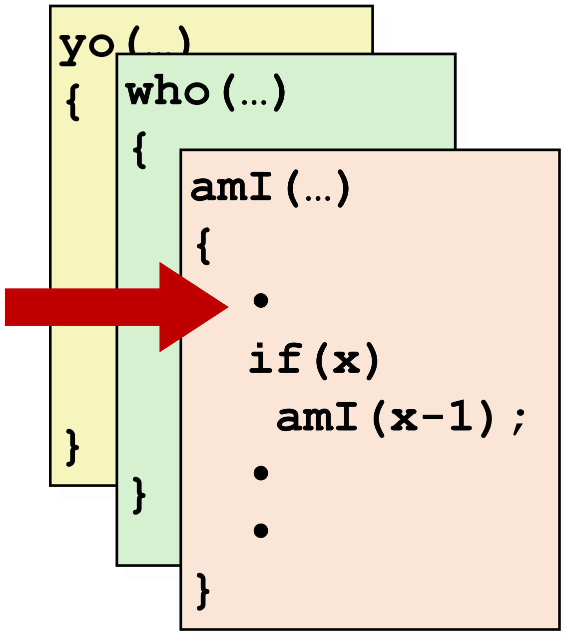


yo
who
amI
amI
amI
amI

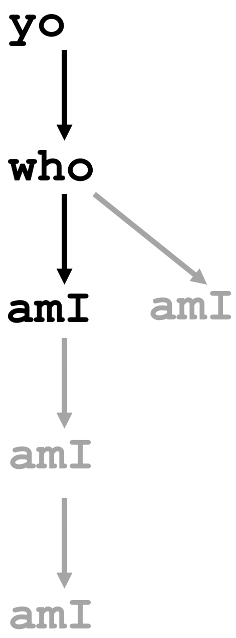


Stack
highest address
yo
who
grows downward
%rsp

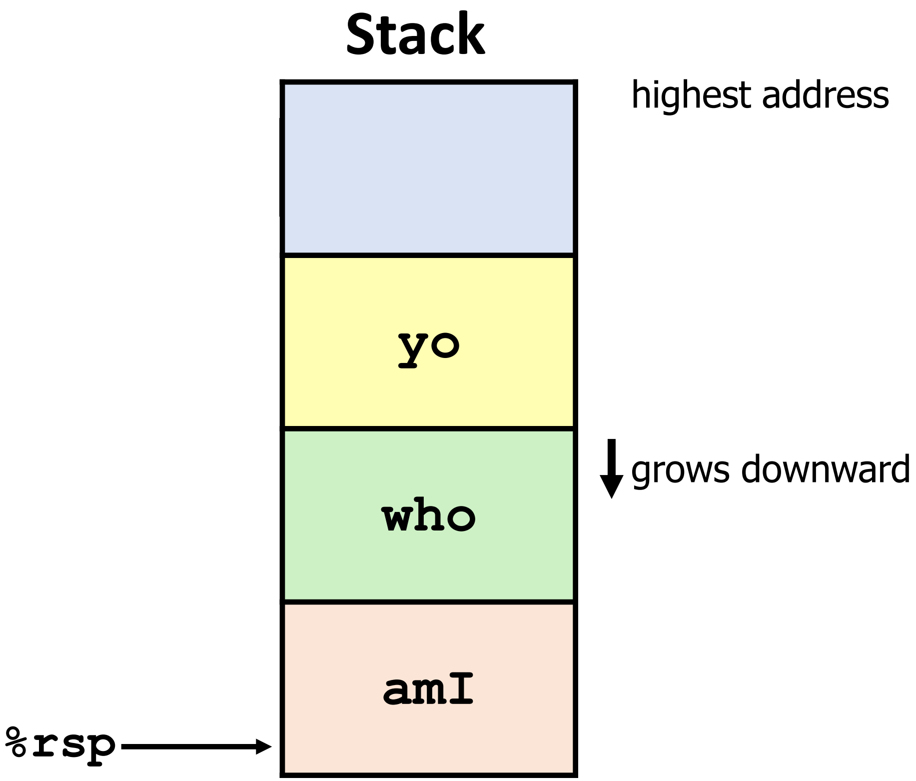
Example



yo(…)
{
}
who(…)
{
}
amI(…)
{
•
if(x)
amI(x-1);
•
•
}

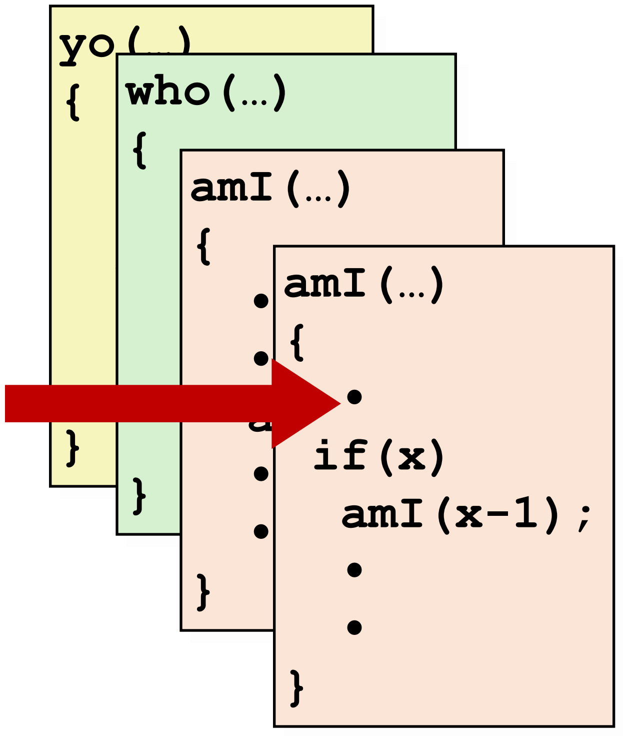


yo
who
amI
amI
amI
amI

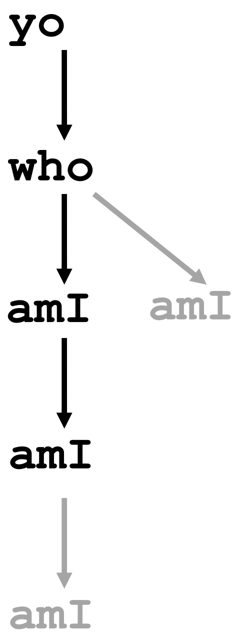


Stack
highest address
yo
who
amI
grows downward
%rsp

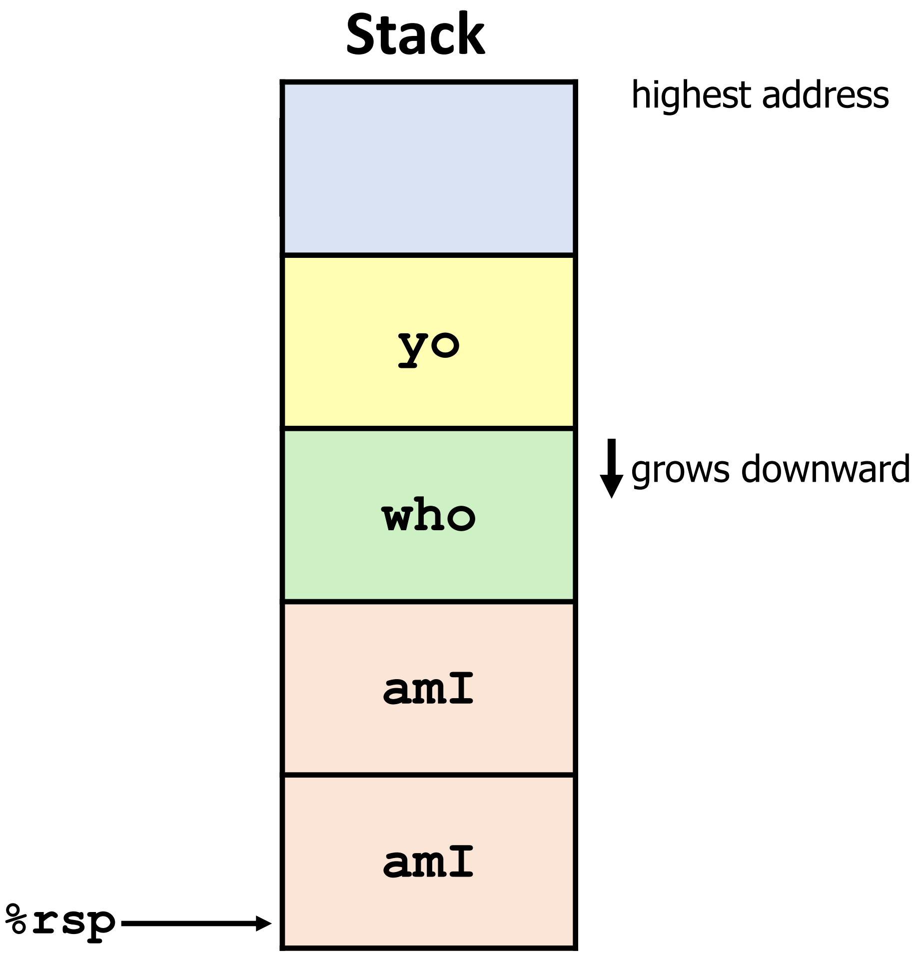
Example



yo(…)
{
}
who(…)
{
}
amI(…)
{
}
amI(…)
{
•
if(x)
amI(x-1);
•
•
}

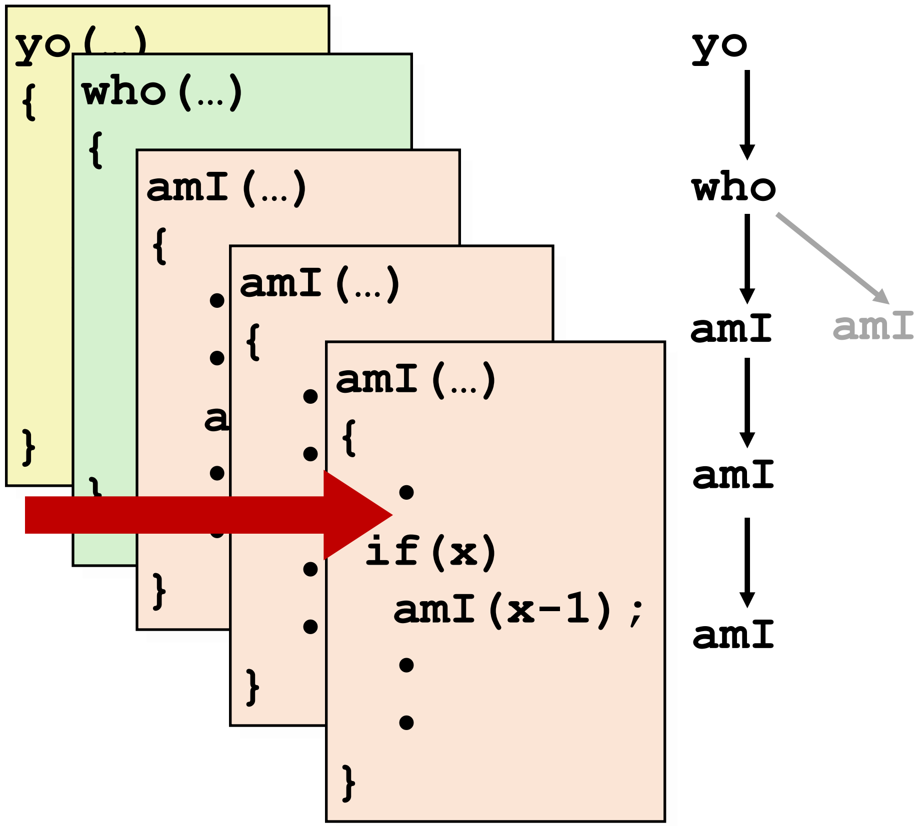


yo
who
amI
amI
amI
amI

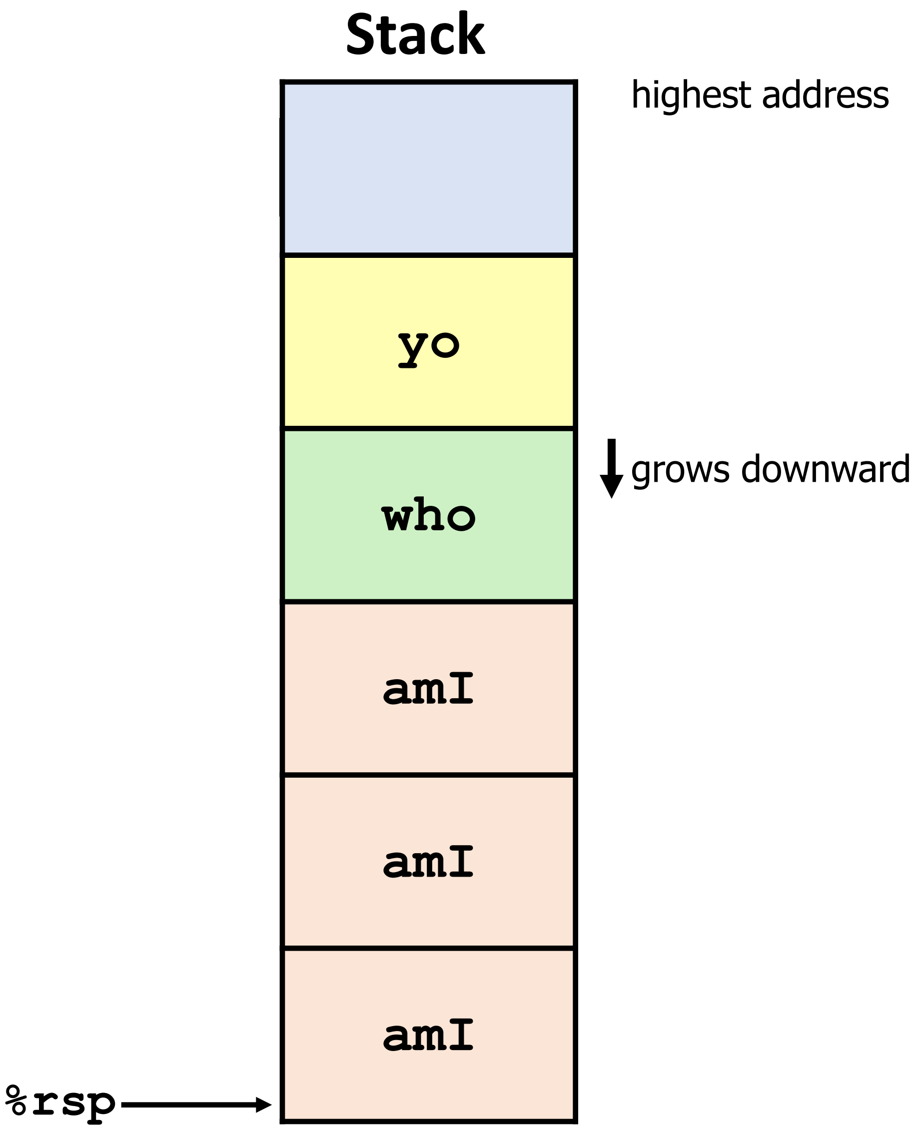


Stack
highest address
yo
who
amI
amI
grows downward
%rsp

Example

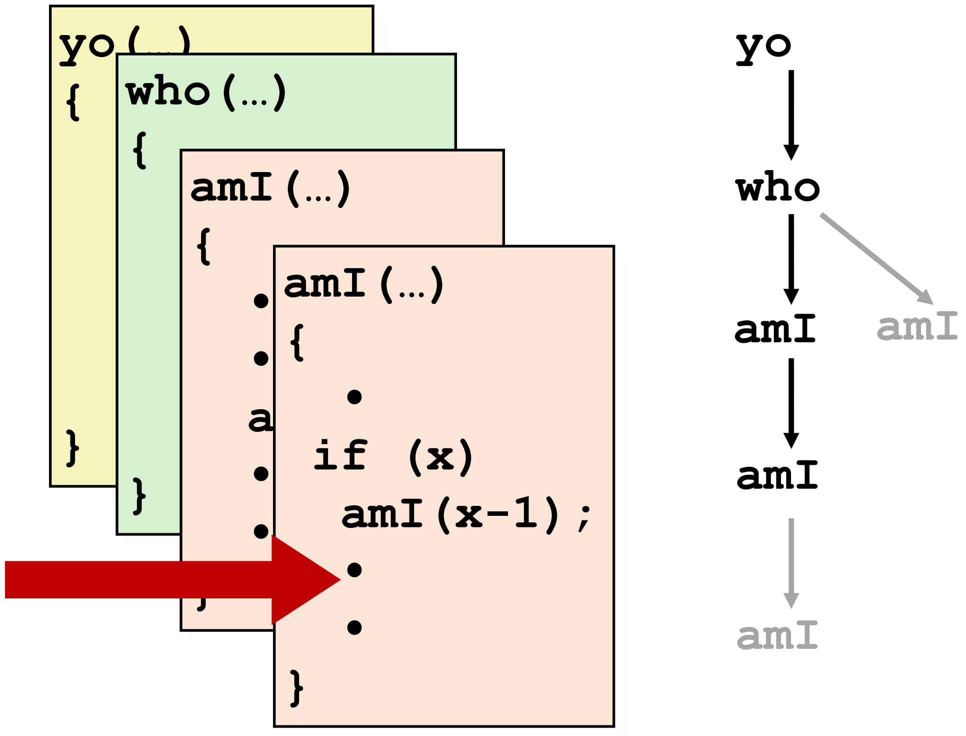


yo (…)
{
}
who (…)
{
}
amI (…)
{
}
amI (…)
{
}
amI (…)
{
if(x)
amI(x-1);
}
yo
who
amI
amI
amI
amI

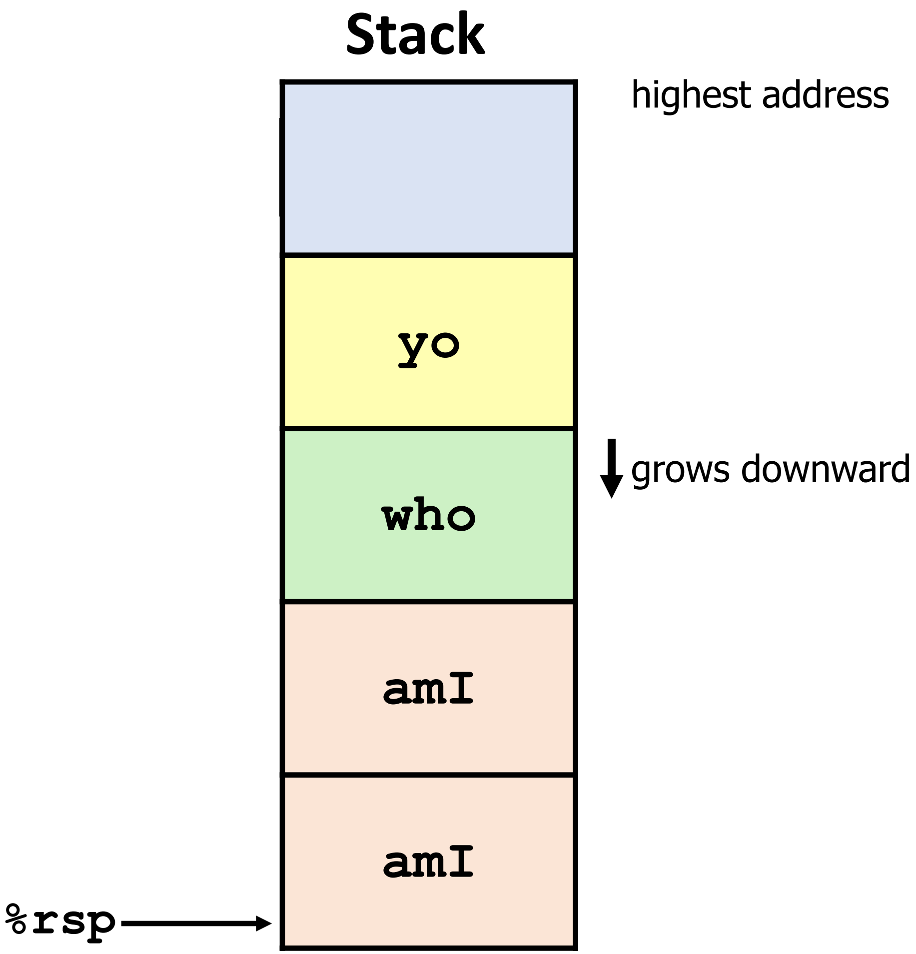


Stack
highest address
yo
who
amI
amI
amI
grows downward
%rsp

Example

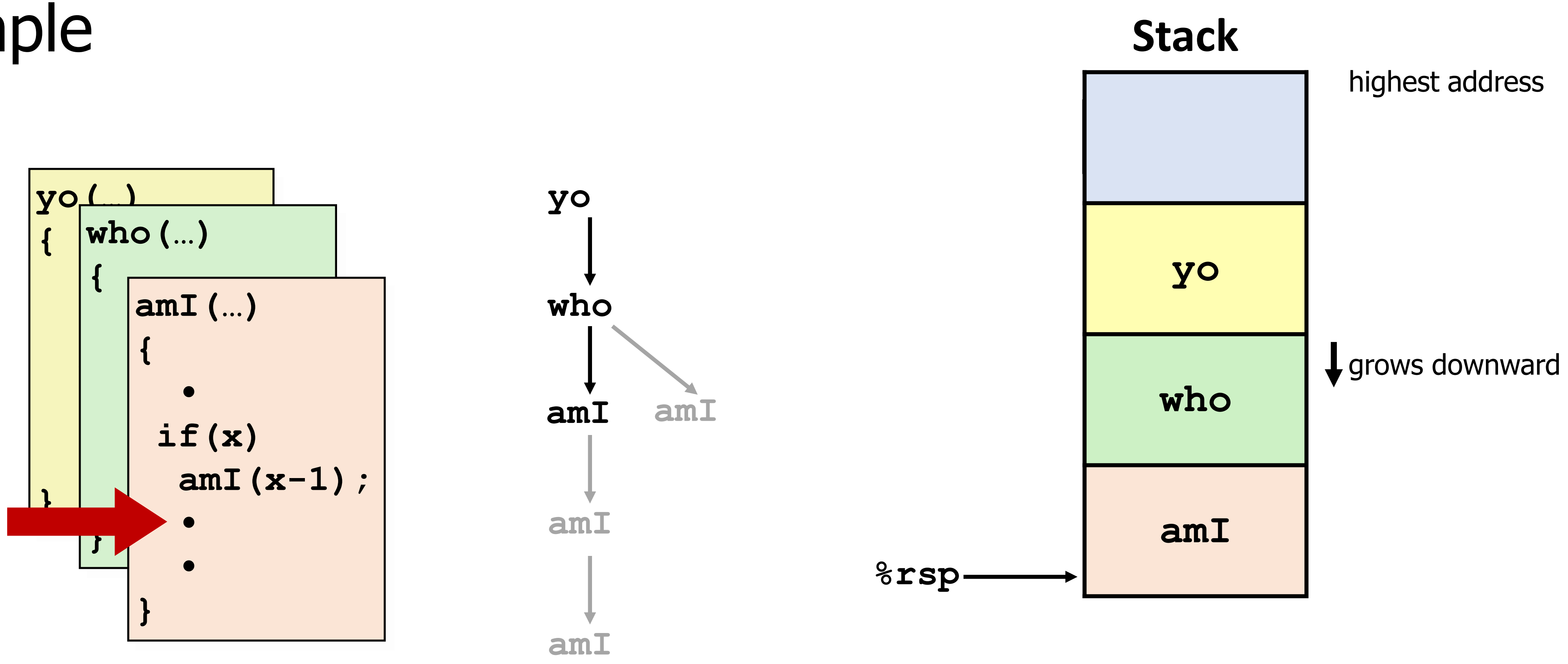


yo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
a
•
•
}
amI(…)
{
•
if (x)
amI(x-1);
•
•
}
yo
who
amI
amI
amI
amI

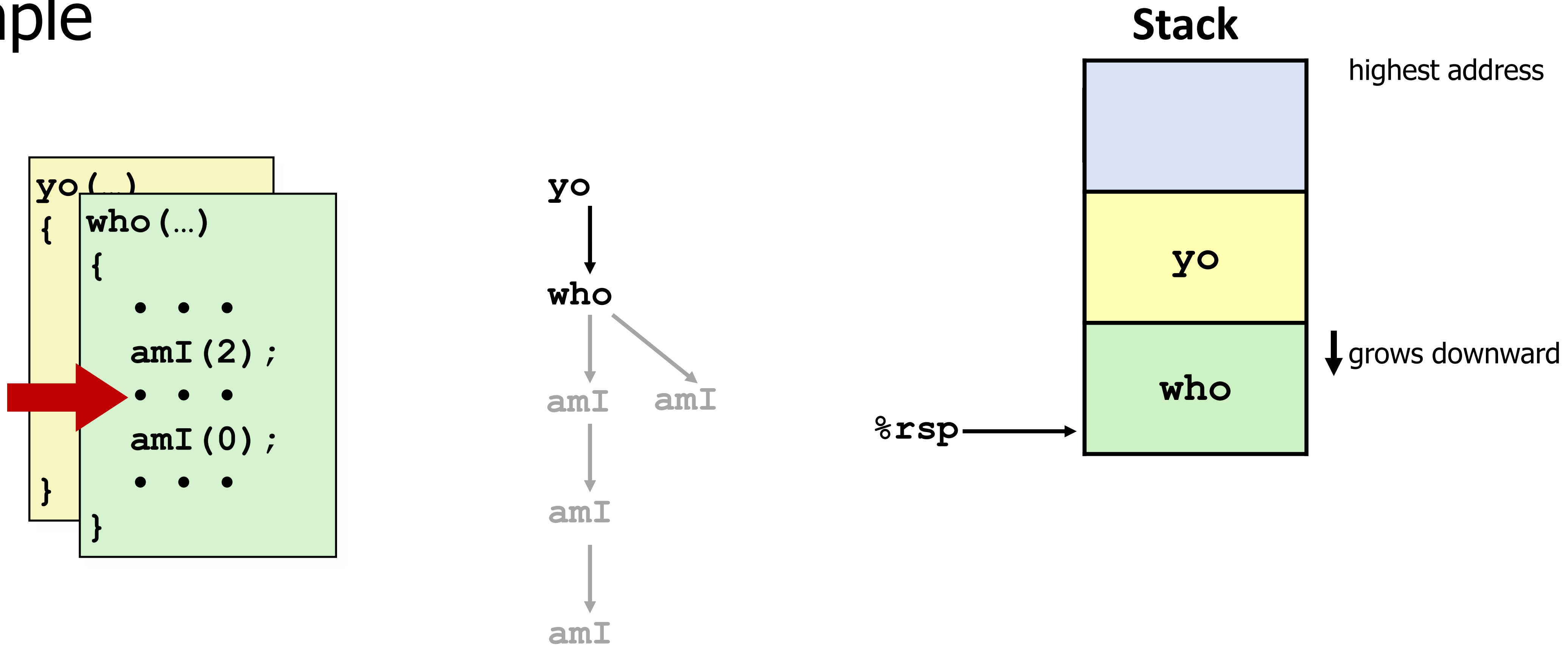


Stack
highest address
yo
grows downward
who
amI
amI
%rsp

Example

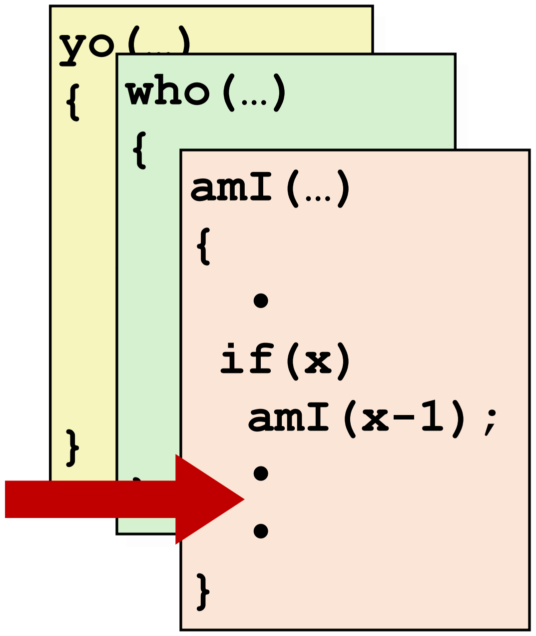

yo(…)
{
}
who(…)
{
}
amI(…)
{
•
if(x)
amI(x-1);
•
•
}
yo
who
amI
amI
amI
amI
Stack
highest address
yo
who
amI
grows downward
%rsp

Example

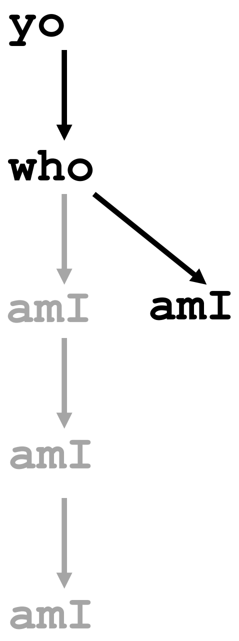


yo(…)
{
}
who(…)
{
• • •
amI(2);
• • •
amI(0);
• • •
}
yo
who
amI
amI
amI
amI
Stack
highest address
yo
who
grows downward
%rsp

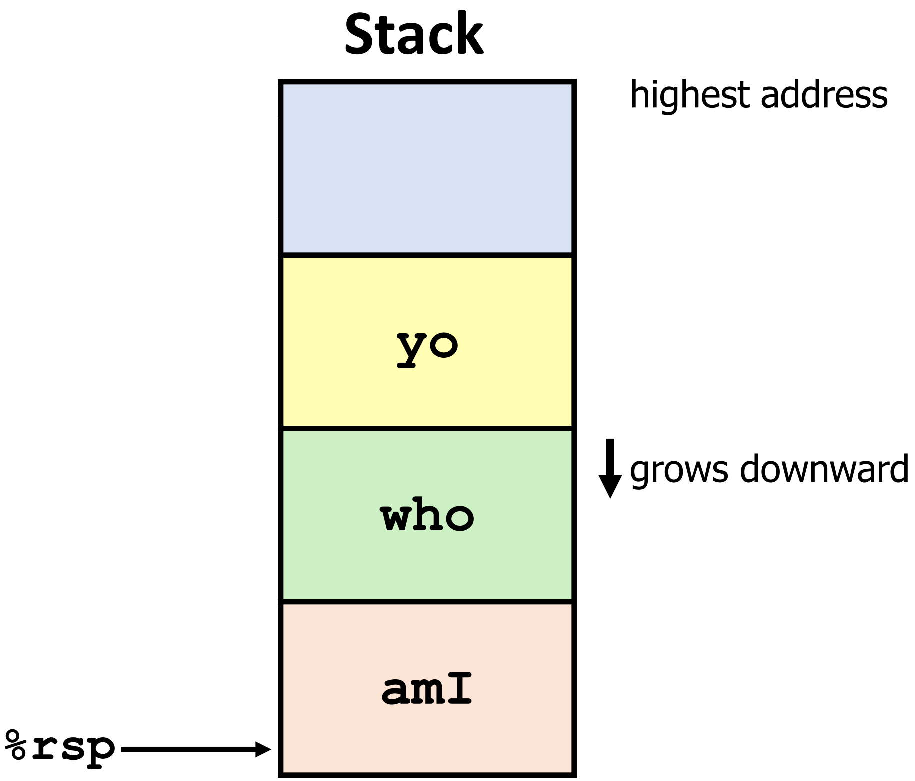
Example



yo(…)
{
}
who(…)
{
amI(…)
{
•
if(x)
amI(x-1);
•
•
}

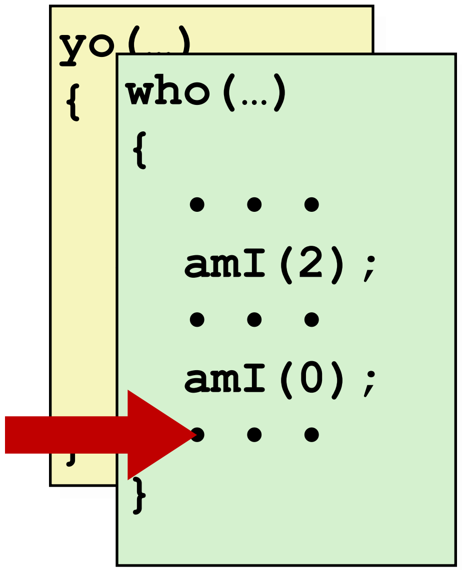


yo
who
amI
amI
amI
amI

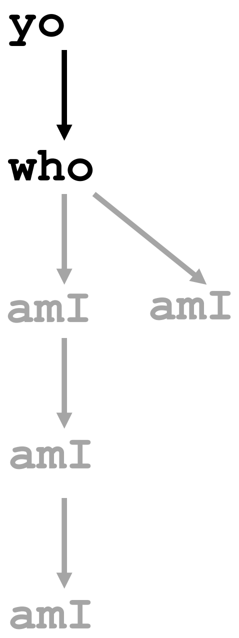


Stack
highest address
yo
who
amI
grows downward
%rsp

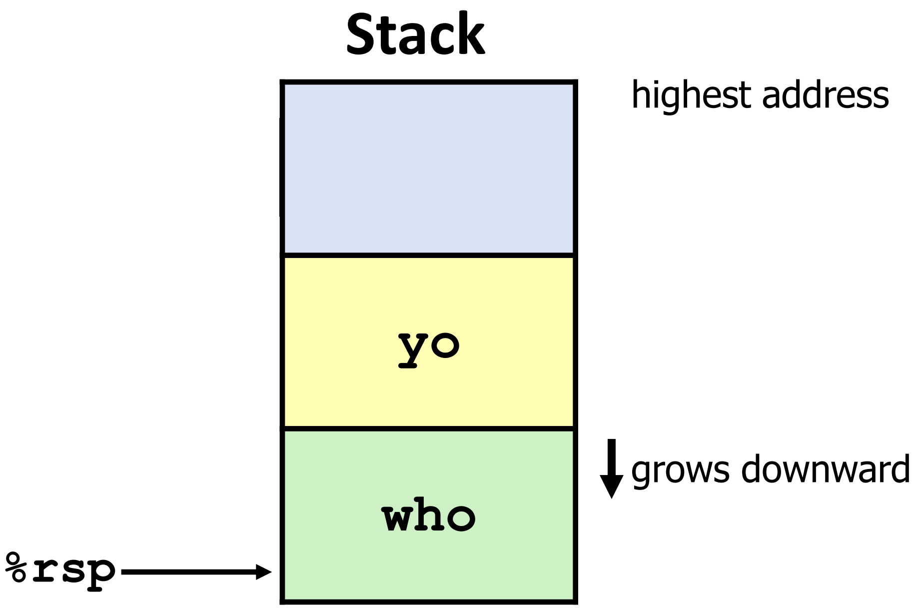
Example



yo(…)
{
who(…)
{
• • •
amI(2);
• • •
amI(0);
• • •
}



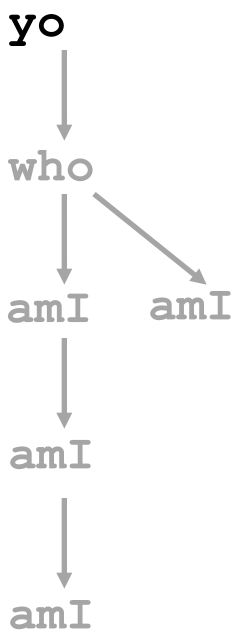
yo
who
amI
amI
amI
amI

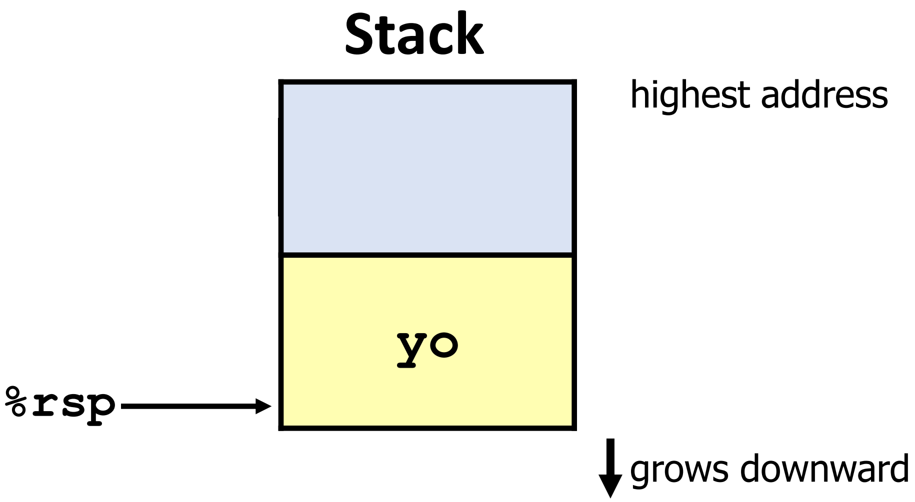


Stack
highest address
yo
who
grows downward
%rsp

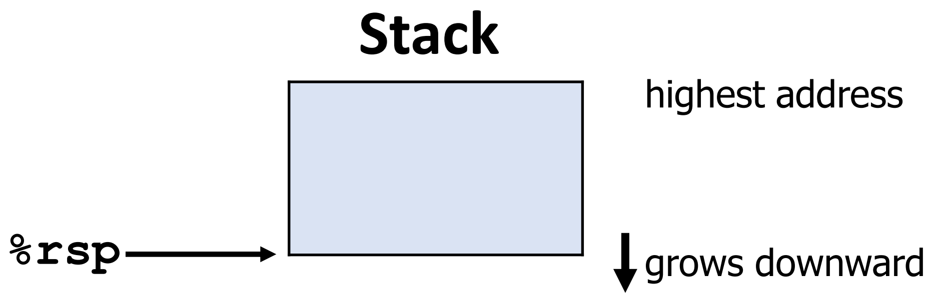
Example

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





Returning to original stack



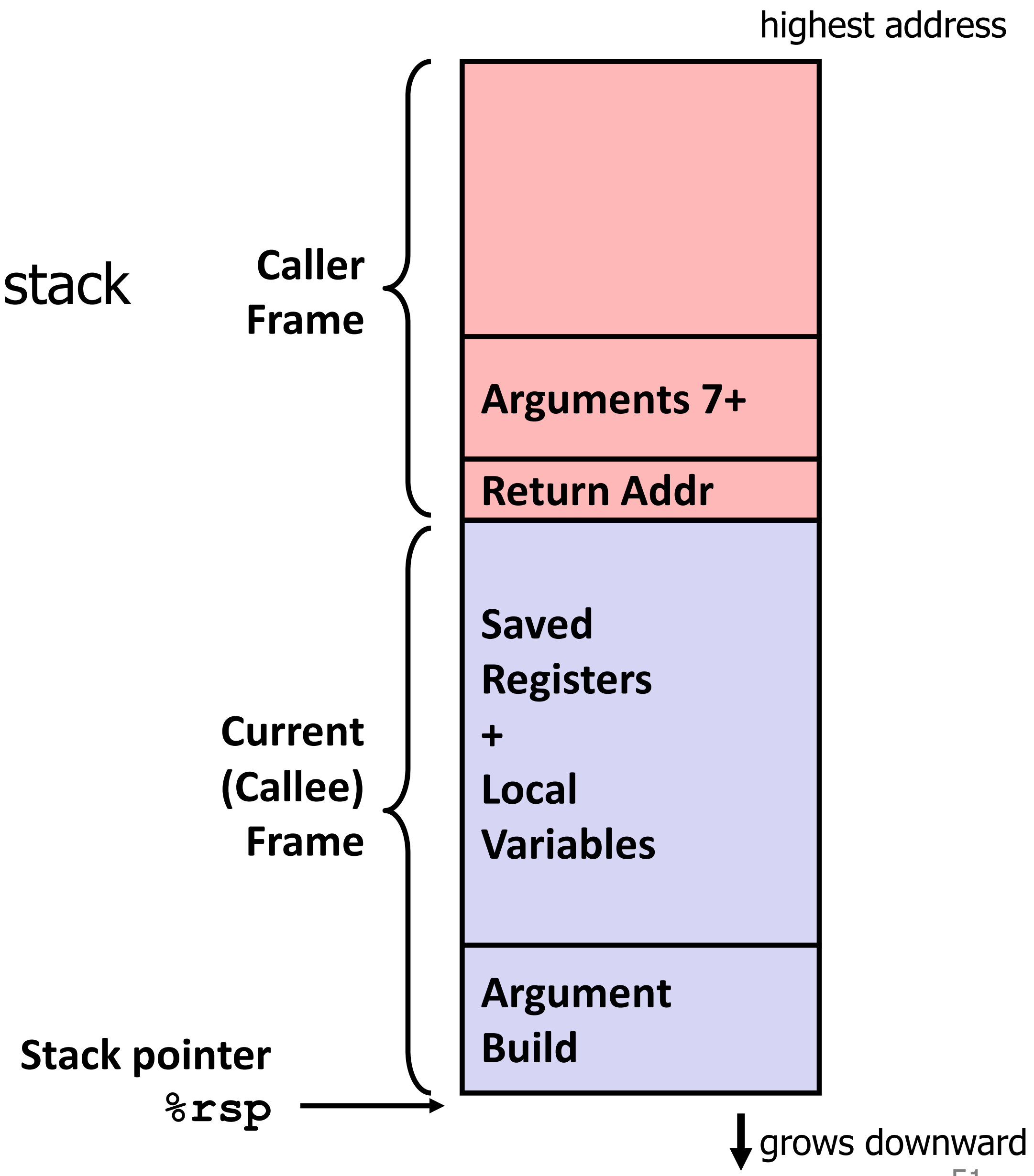
- Stack always eventually returns to its default state
  - Happens automatically in higher-level languages like C
  - Need to manage that ourselves if writing assembly

- Or the program can exit early from anywhere
  - Entire stack is deallocated when the program ends

x86-64/Linux Stack Frame

- Current Stack Frame ("Top" to Bottom)
  - "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
    (too many, or if must be in memory)
  - Saved register context
    (we'll get to that soon)

- Caller Stack Frame
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call



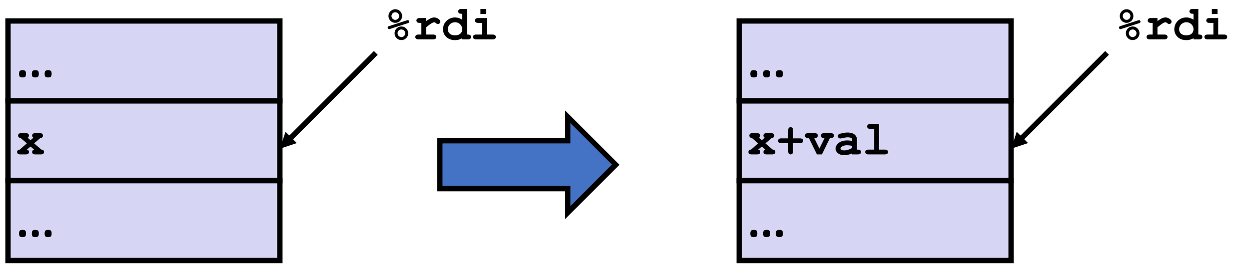
Example: `incr`

```
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)
`%rdi`	Argument **`p`**
`%rsi`	Argument **`val`**, also **`y`**
`%rax`	**`x`**, Return value

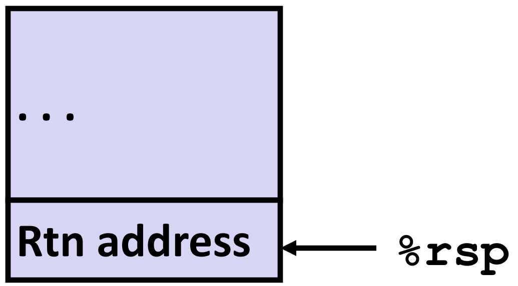
Memory



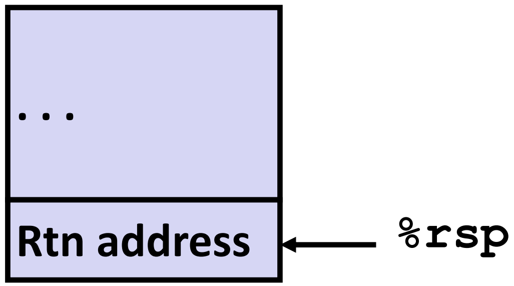
Example: Calling **incr** #1 (local variables)

```
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

Stack	
...	
Rtn address	← `%rsp`

Resulting Stack Structure

Stack	
...	
Rtn address	
15213	← `%rsp+8`
Unused	← `%rsp`

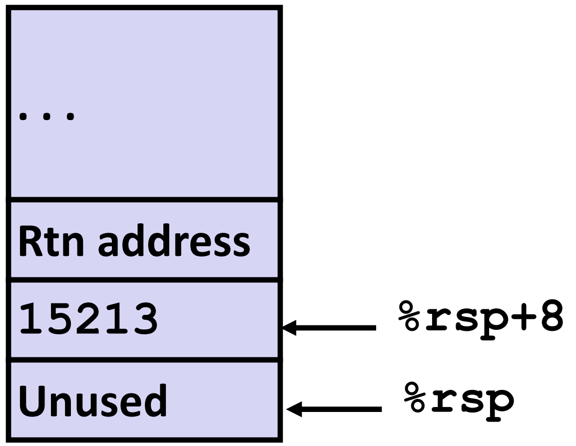
Example: Calling **`incr`** #2 (argument build)

```
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



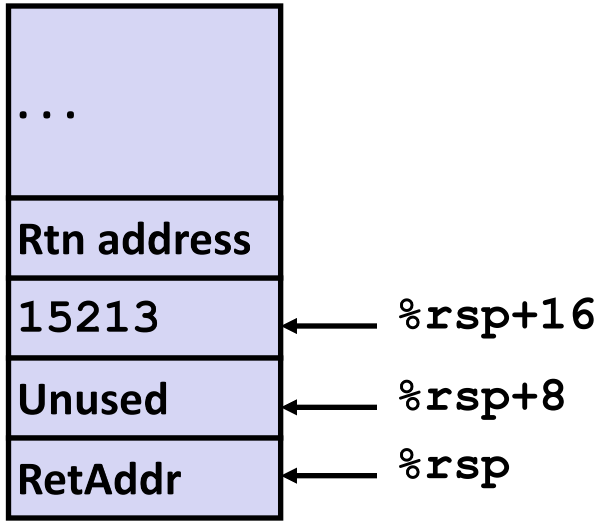
Example: Calling `incr` #3 (control transfer)

```
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`

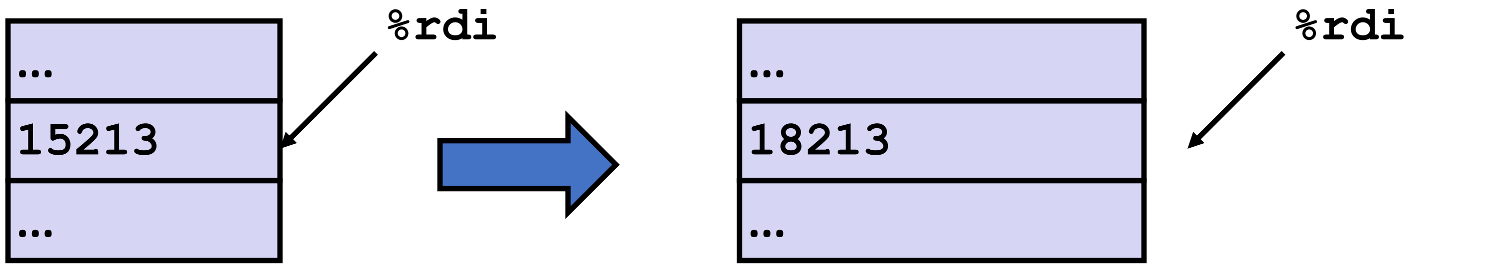
```c
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

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

Register	Use(s)
`%rdi`	Argument `p`
`%rsi`	Argument `val` (3000)
`%rax`	. . .

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

Memory

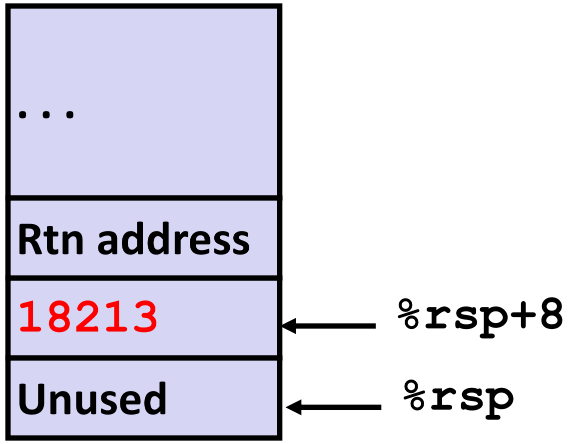


Example: right after executing `incr`

```
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

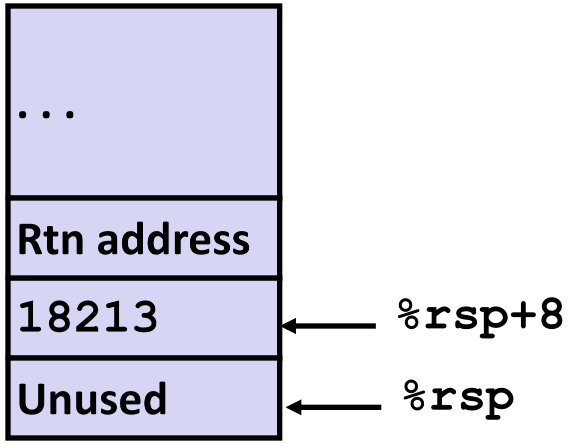
QUIZ: where do we find the return value of `incr`?

Example: Calling `incr` #4 (cleanup)

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

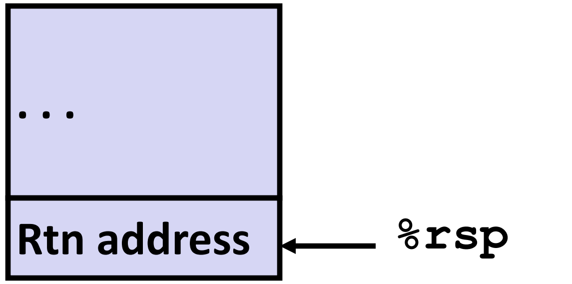
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movq    $3000, %rsi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Previous stack Structure



Register	Use(s)
`%rax`	Return value

Updated Stack Structure

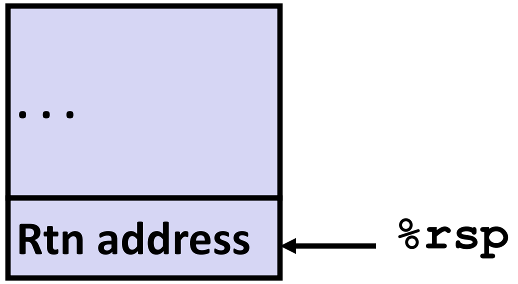


Example: Calling `incr` #5

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

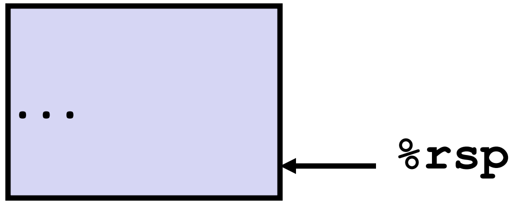
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movq    $3000, %rsi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Updated Stack Structure



Register	Use(s)
`%rax`	Return value

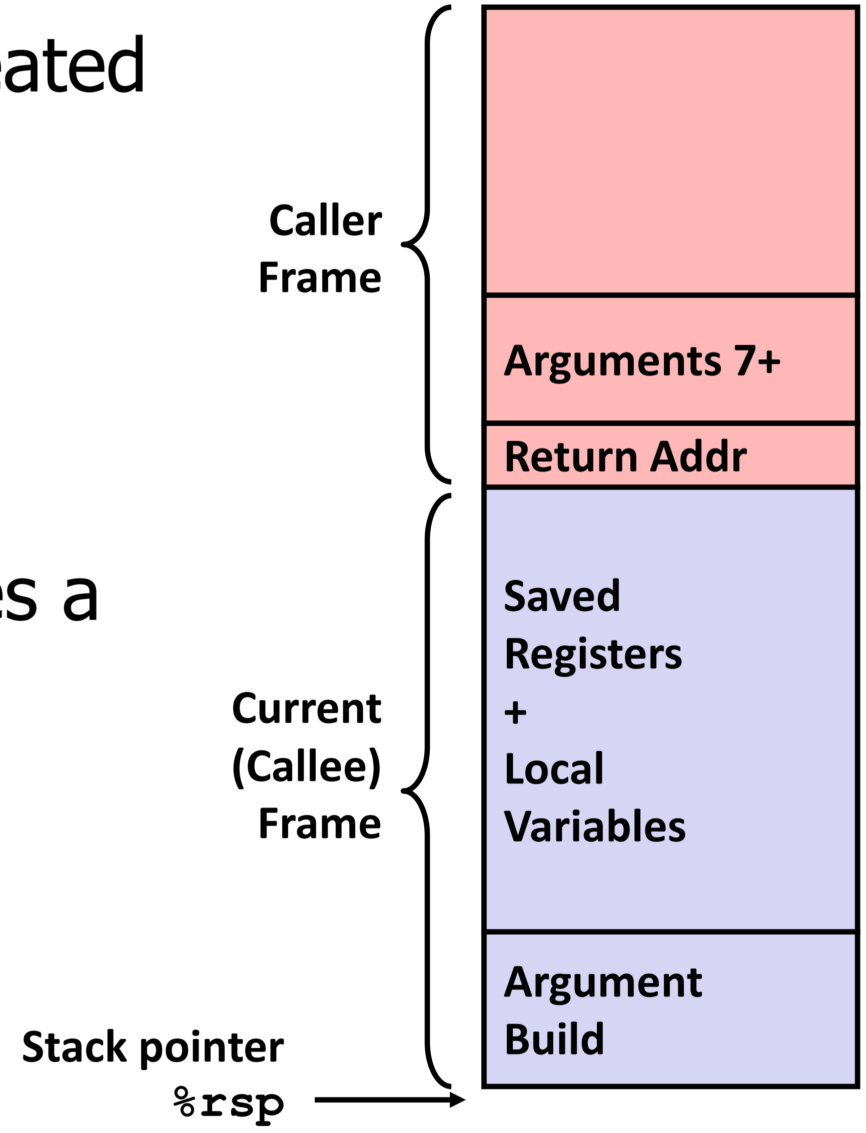
Final Stack Structure



Break + Open Questions

- What are the initial values of variables created on the stack?

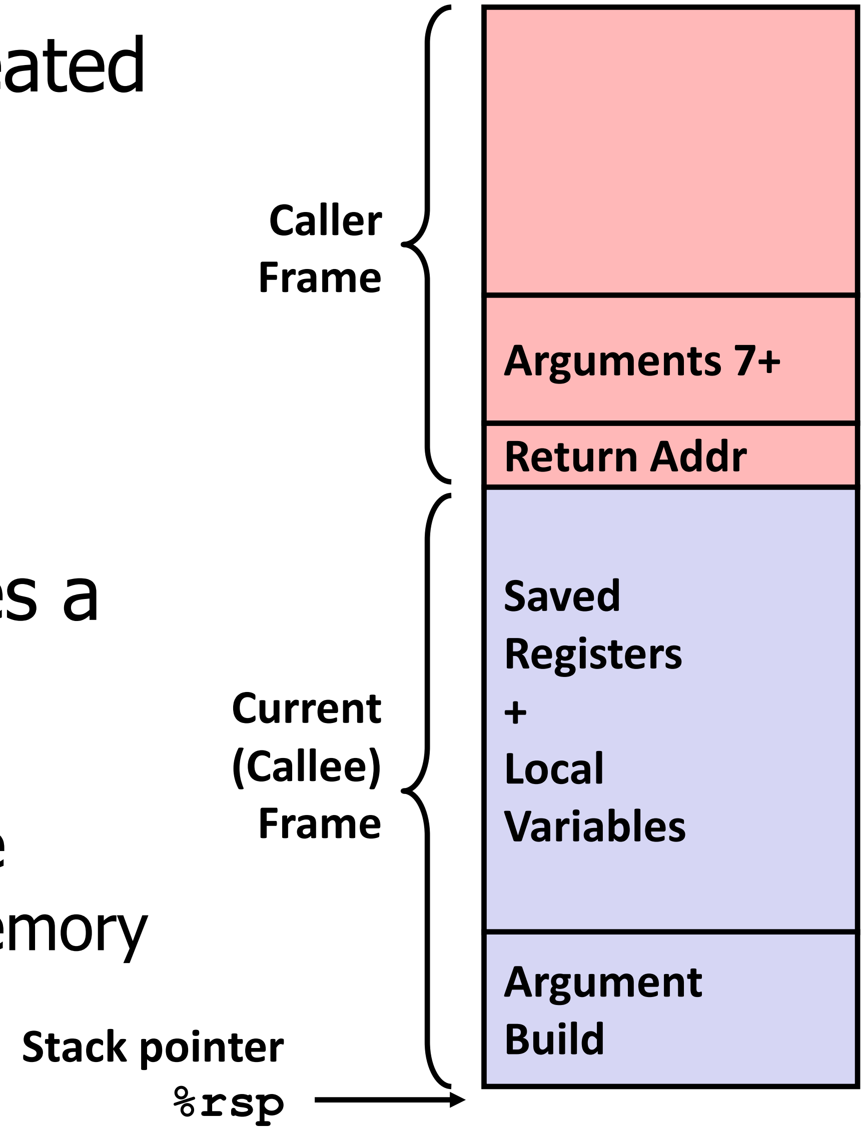
- Is there a limit to how many local variables a function can have?



Break + Open Questions

- What are the initial values of variables created on the stack?
  - Undefined behavior in C (compiler chooses)
  - 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 (function that’s run) could overwrite caller’s (code that’s calling the function) registers by accident

- 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 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`

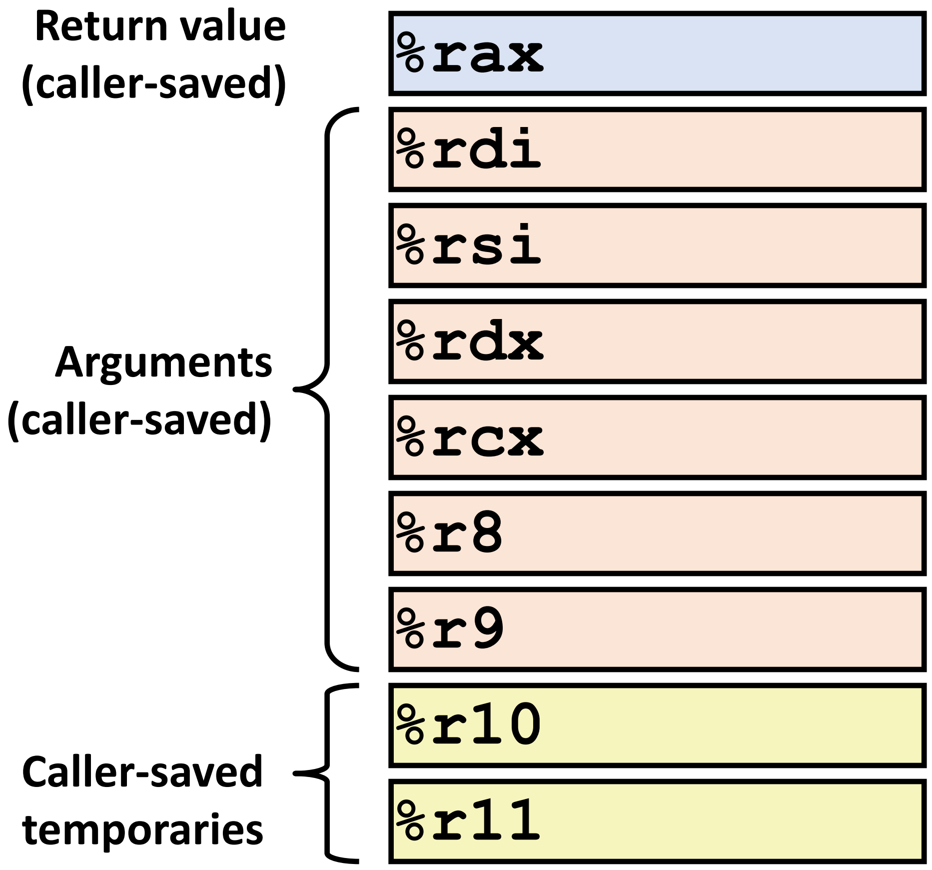
Full Rules for Register Saving

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

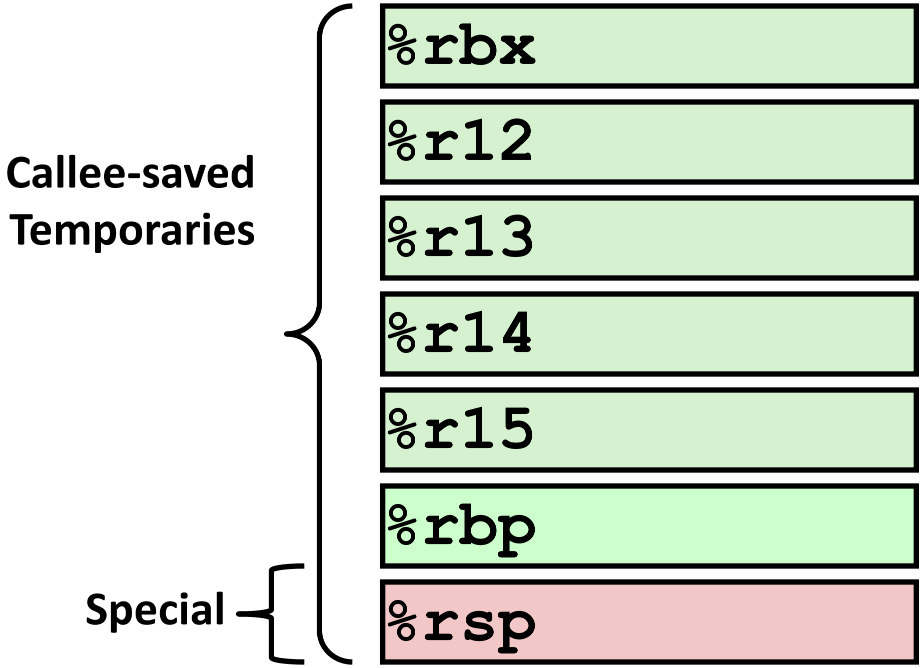
x86-64 Linux Register Usage #1 (caller-saved, in advance)

- `%rax`
  - Return value
  - Caller-saved
  - **Will** be modified by function we're about to call
- `%rdi`, …, `%r9`
  - Arguments
  - Caller-saved
  - Can be modified by function we're about to call
- `%r10`, `%r11`
  - Caller-saved
  - Can 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
  - Any function must save/restore the original values if it wants to use these registers



- **`%rsp`**
  - Special form of callee-saved
  - Restored to original value upon exit from procedure
    - Stack frame is removed

x86-64 Integer Registers: Usage Conventions

Caller Saved	In advance
Callee saved	On demand

`%rax`	Return value	`%r8`	Argument #5
`%rbx`	Callee saved	`%r9`	Argument #6
`%rcx`	Argument #4	`%r10`	Caller saved
`%rdx`	Argument #3	`%r11`	Caller Saved
`%rsi`	Argument #2	`%r12`	Callee saved
`%rdi`	Argument #1	`%r13`	Callee saved
`%rsp`	Stack pointer	`%r14`	Callee saved
`%rbp`	Callee saved	`%r15`	Callee saved

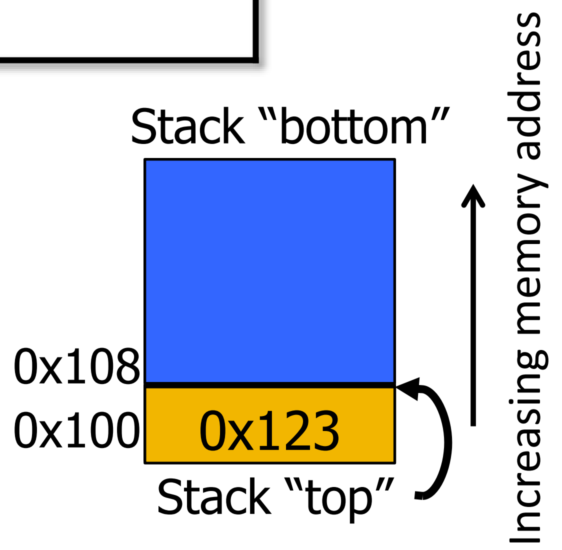
Push and Pop instructions

Instruction	Effect	Description
`pushq` S	R [%rsp] ← R [%rsp] – 8; M [R[%rsp]] ← S	Store S onto the stack
`popq` D	D ← M [R[%rsp]] R [%rsp] ← R [%rsp] + 8;	Retrieve D from the stack

- Example:

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

pushq %rax          %rsp = 0x100
popq %rdx           %rdx = 0x123; %rsp = 0x108
```



- 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

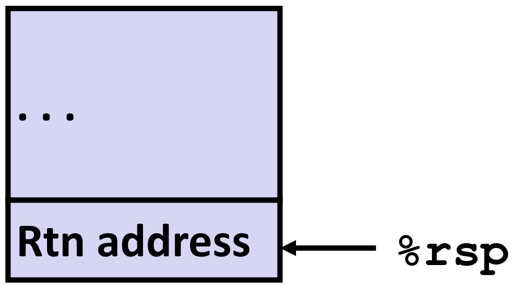
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

Still need **x** after the call!

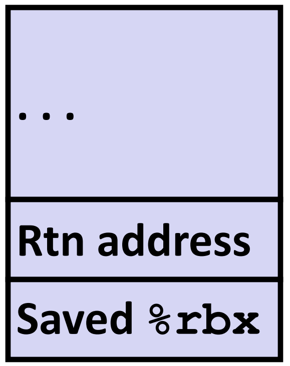
%rbx is callee-save (on demand)

```
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



%rbx is callee-saved and we use it -> Save **%rbx**

Manually allocating stack space

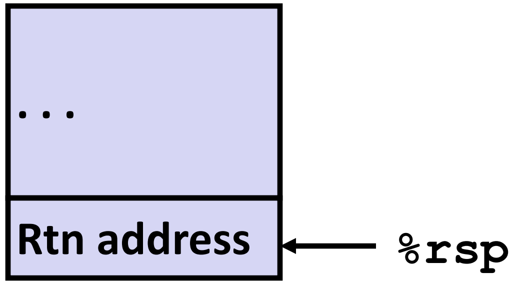
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

Still need **x** after the call!

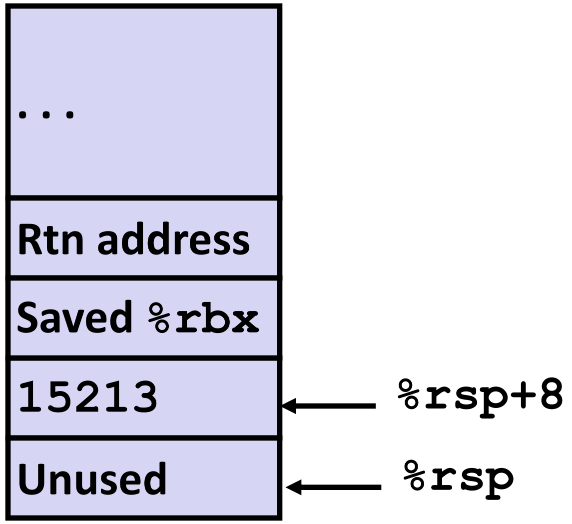
%rbx is callee-save (on demand)

```
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

Stack	
...	
Rtn address	
Saved %rbx	
15213	← %rsp+8
Unused	← %rsp

Our caller can expect its own value in %rbx

Restore it!

Pre-return Stack Structure

Stack	
...	
Rtn address	← %rsp

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);
}
```

```
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
```

Checks if `%rdi` is zero

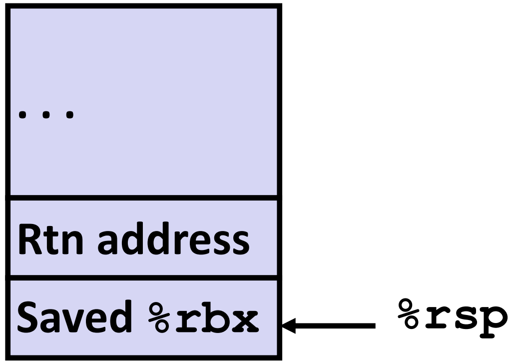
Register	Use(s)	Type
`%rdi`	`x`	Argument
`%rax`	Return value	Return value

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);
}
```

```
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
%rdi	x	Argument



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);
}
```

```
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
%rdi	x >> 1	Rec. argument
%rbx	x & 1	Callee-saved

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 # (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
```

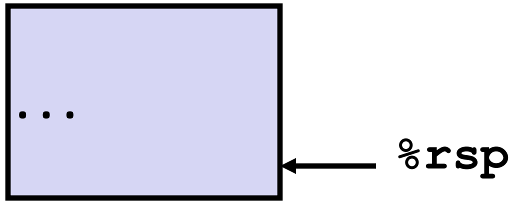
Register	Use(s)	Type
`%rbx`	`x & 1`	Callee-saved
`%rax`	Return value	

Recursive Function Completion

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

```
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
%rax	Return value	Return value



Example three recursions in

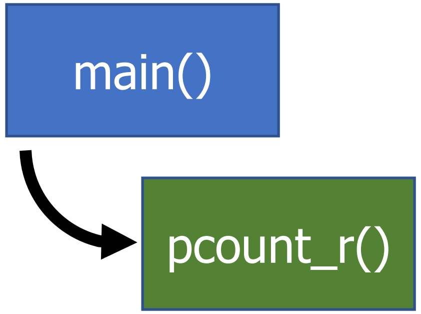
main()

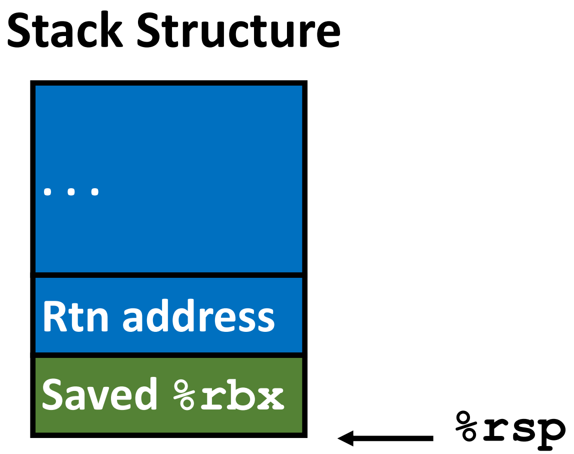
Stack Structure

...

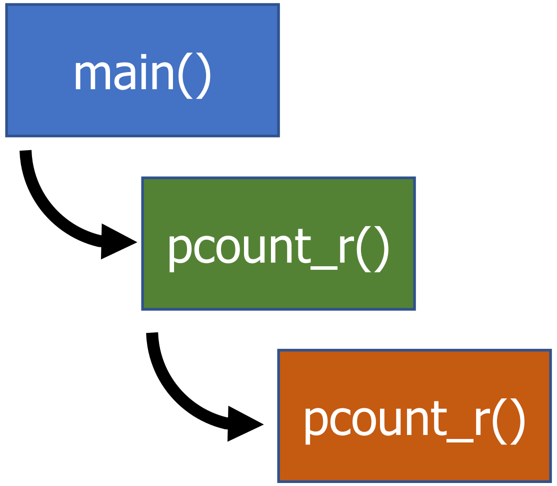
`%rsp`

Example three recursions in

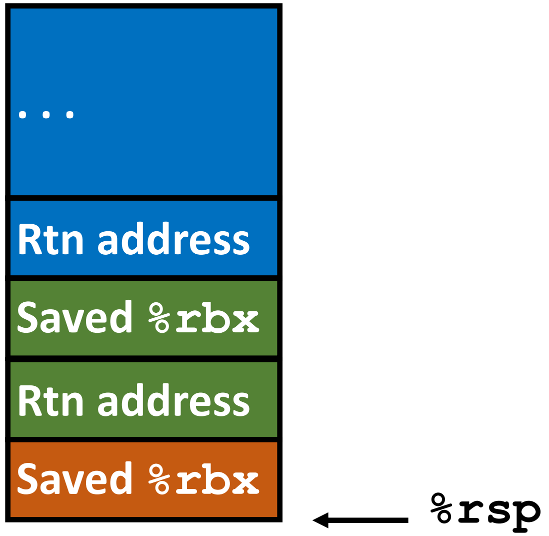




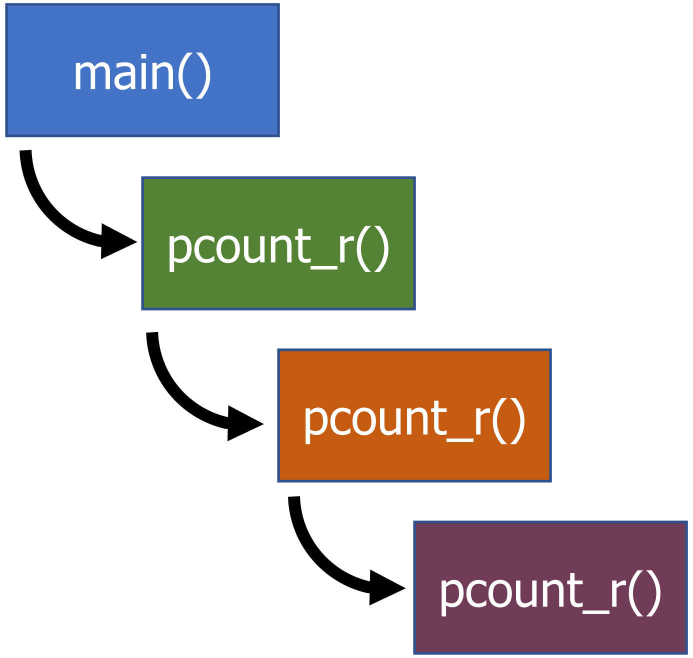
Example three recursions in

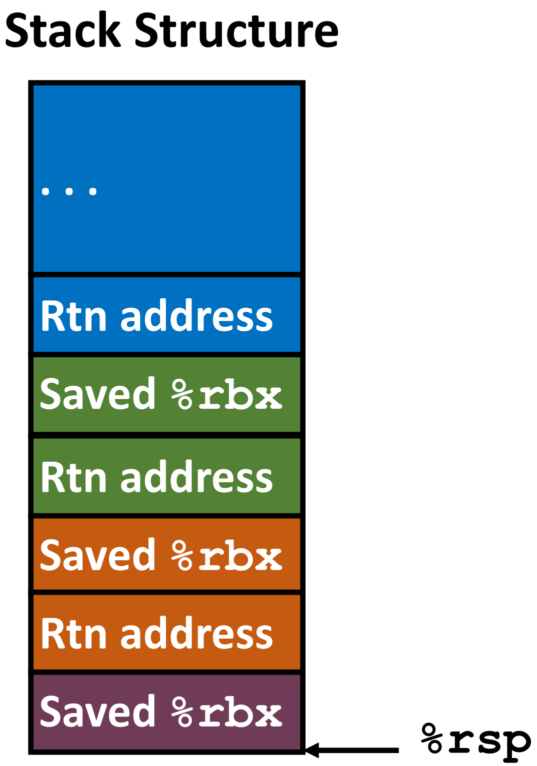


Stack Structure



Example three recursions in



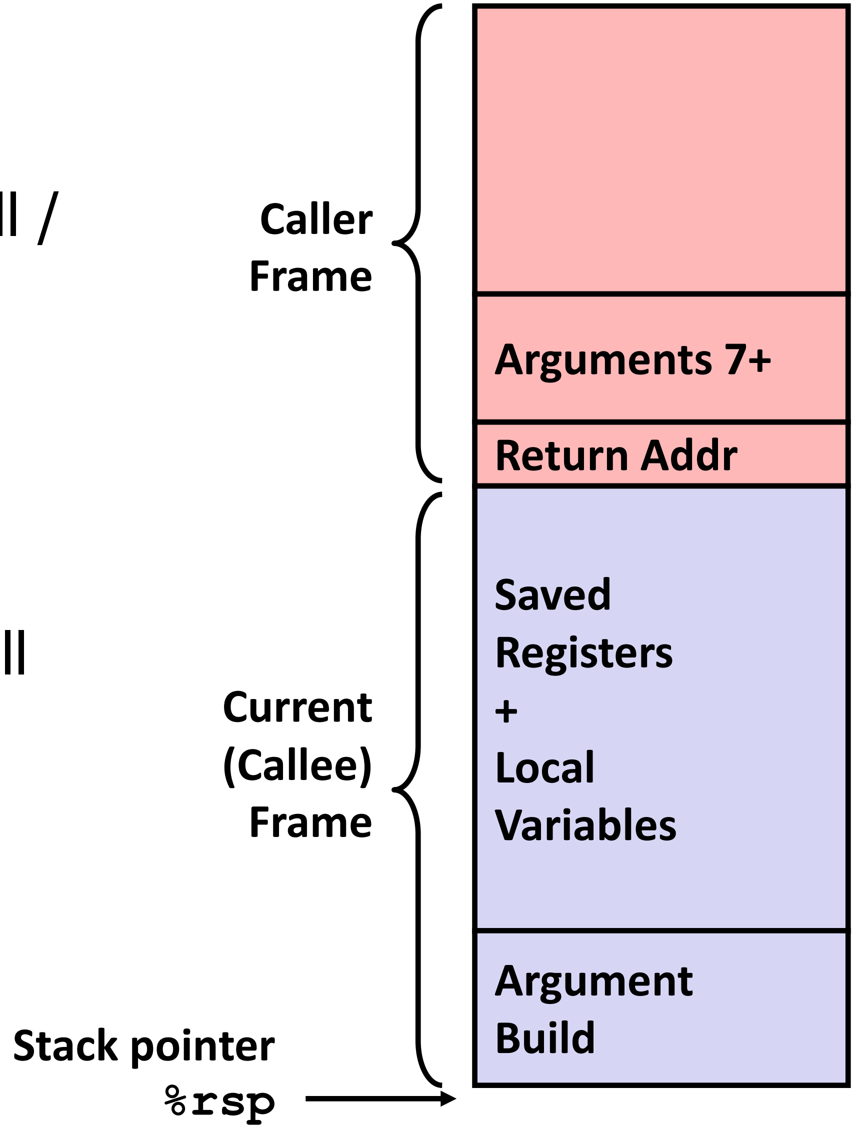


Executing, but has not yet called pcount_r() again

x86-64 Procedure Summary

- Important Points
  - A stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
  - 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           # 2nd argument
    movq    %rbp, %rdi           # 1st 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
```

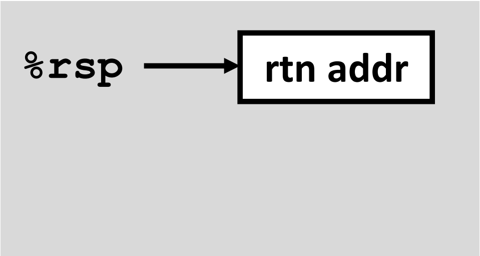
Understanding x86-64 Stack Frame

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

subq    $16, %rsp              # Allocate stack frame

• • •

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



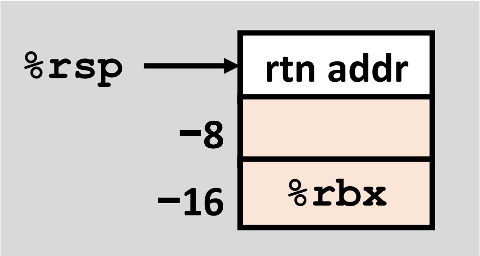
Understanding x86-64 Stack Frame

```
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
```



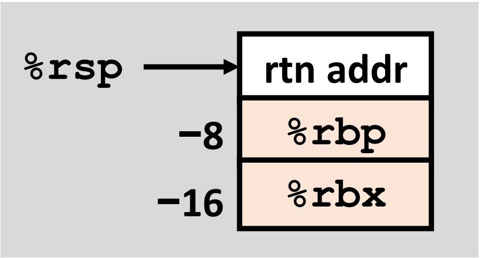
Understanding x86-64 Stack Frame

```
    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
```



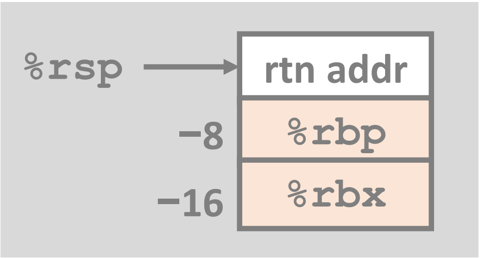
Understanding x86-64 Stack Frame

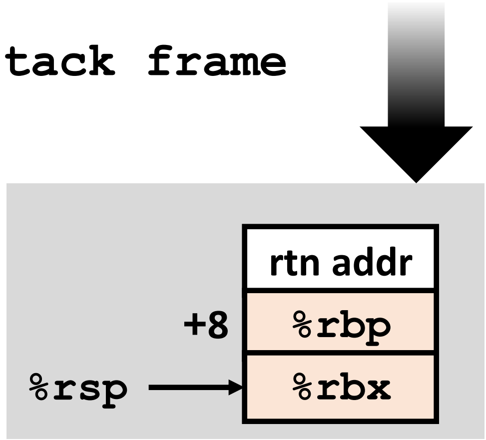
```
movq    %rbx, -16(%rsp)       # Save %rbx
movq    %rbp, -8(%rsp)        # Save %rbp

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