CS 31: Introduction to Computer Systems

14: Arrays and Structs

03-18-2025

SWARTHMORE COLLEGE

Four Types of Assembly Instructions

1. Arithmetic: use ALU to compute a value
2. Data movement: load and store
3. Control Flow: branch, jump, etc.
4. Stack Instructions: push and pop stack frames
   – Shortcut instructions for common operations (we'll cover these in detail later)

Overview

- Stack data structure, applied to memory

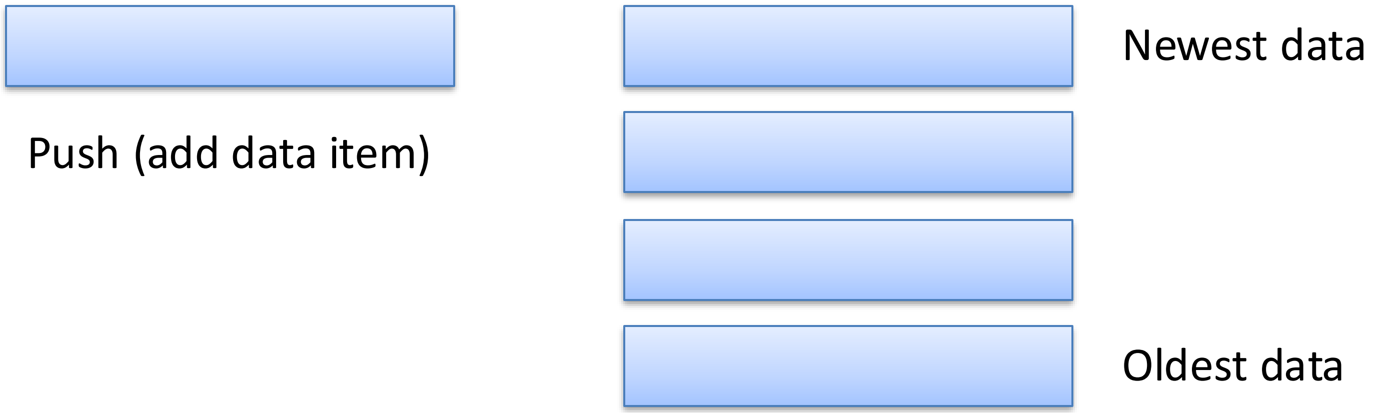
- Behavior of function calls

- Storage of function data, at assembly level

“A” Stack

- A stack is a basic data structure
  - Last in, first out behavior (LIFO)
  - Two operations
    - Push (add item to top of stack)
    - Pop (remove item from top of stack)

Pop (remove and return item)



Newest data

Push (add data item)

Oldest data

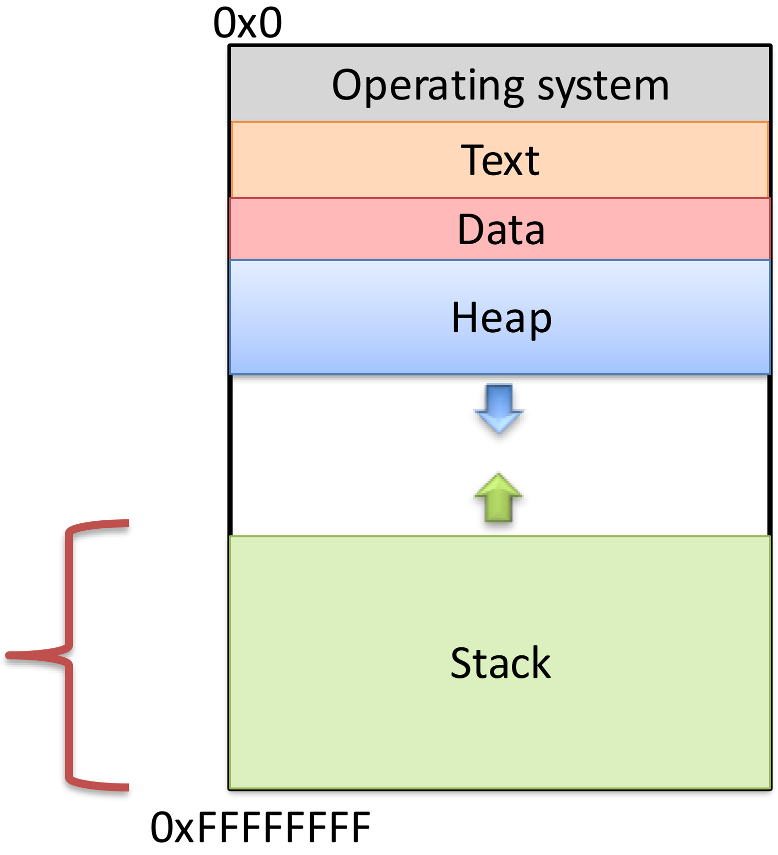
“The” Stack

- Apply stack data structure to memory
  - Store local (automatic) variables
  - Maintain state for functions (e.g., where to return)

- Organized into units called *frames*
  - One frame represents all of the information for one function.
  - Sometimes called *activation records*

Memory Model

- Starts at the highest memory addresses, grows into lower addresses.



What is responsible for creating and removing stack frames?

A. The user

B. The compiler

C. C library code

D. The operating system

E. Something / someone else

Insight: EVERY function needs a stack frame. Creating / destroying a stack frame is a (mostly) generic procedure.

What is responsible for creating and removing stack frames?

A. The user

B. The compiler

C. C library code

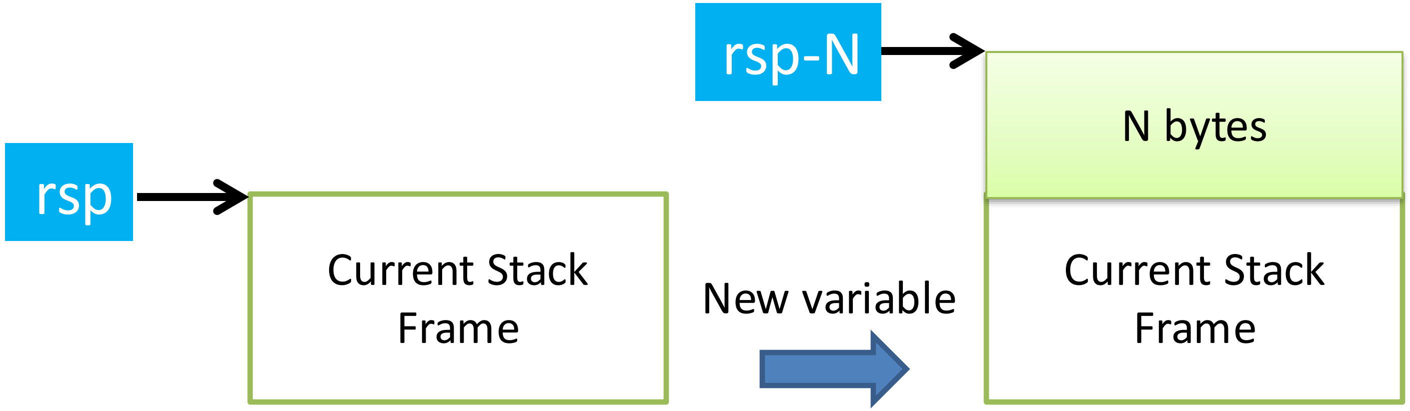
D. The operating system

E. Something / someone else

Insight: EVERY function needs a stack frame. Creating / destroying a stack frame is a (mostly) generic procedure.

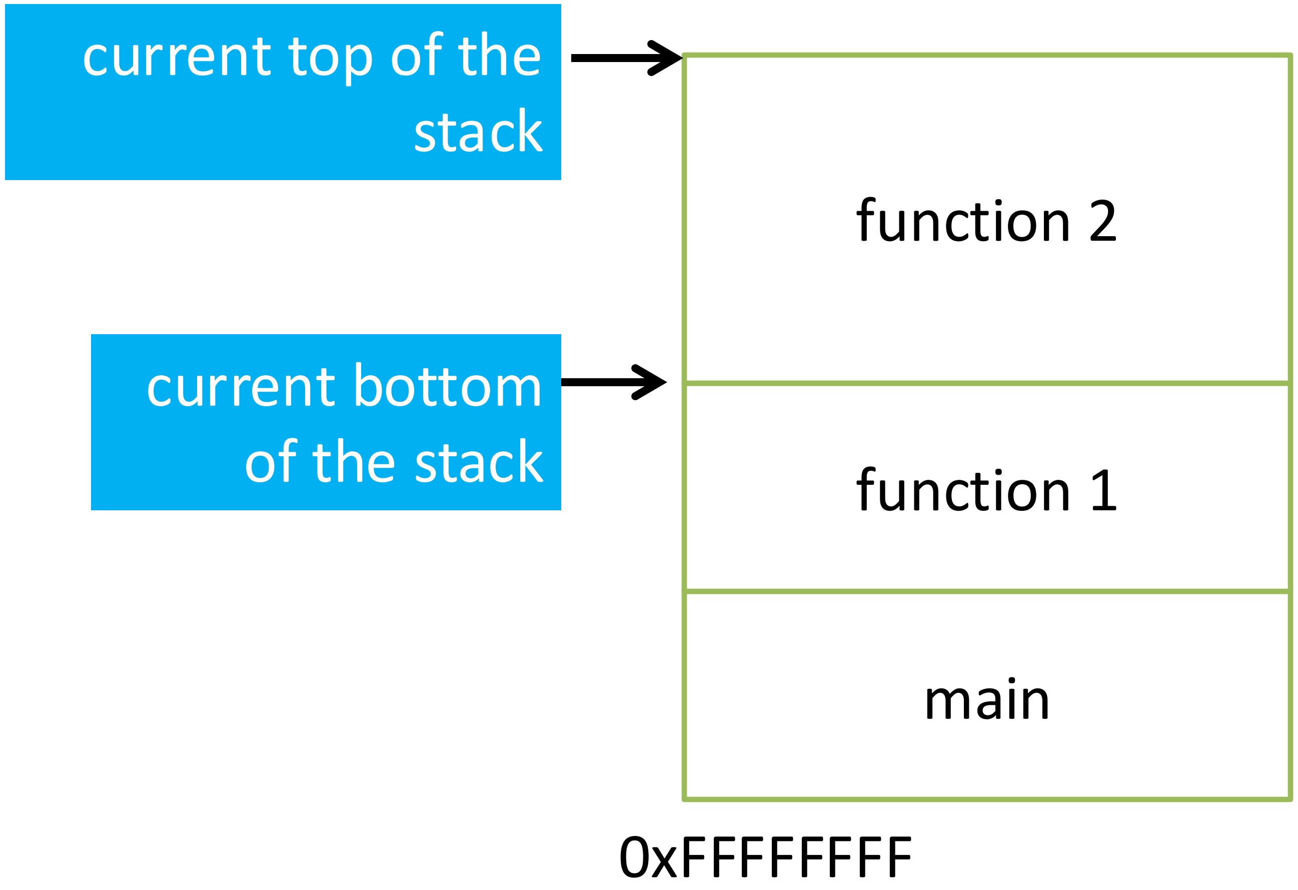
Local Variables

Compiler can allocate N bytes on the stack by subtracting N from the stack pointer: (rsp)



Stack Frame Location

Where in memory is the current stack frame?

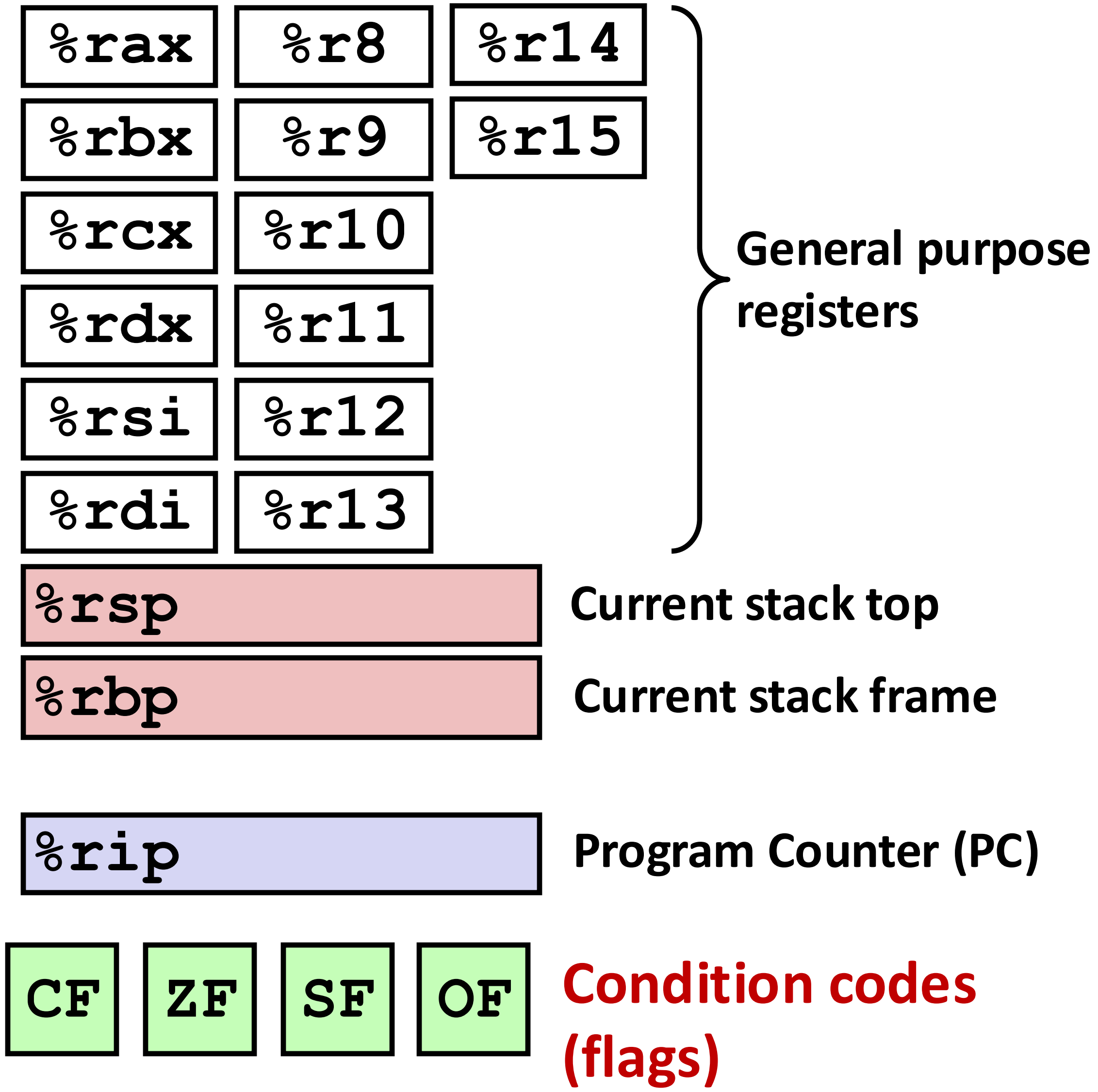


current top of the stack

current bottom of the stack

Recall: x86_64 Register Conventions

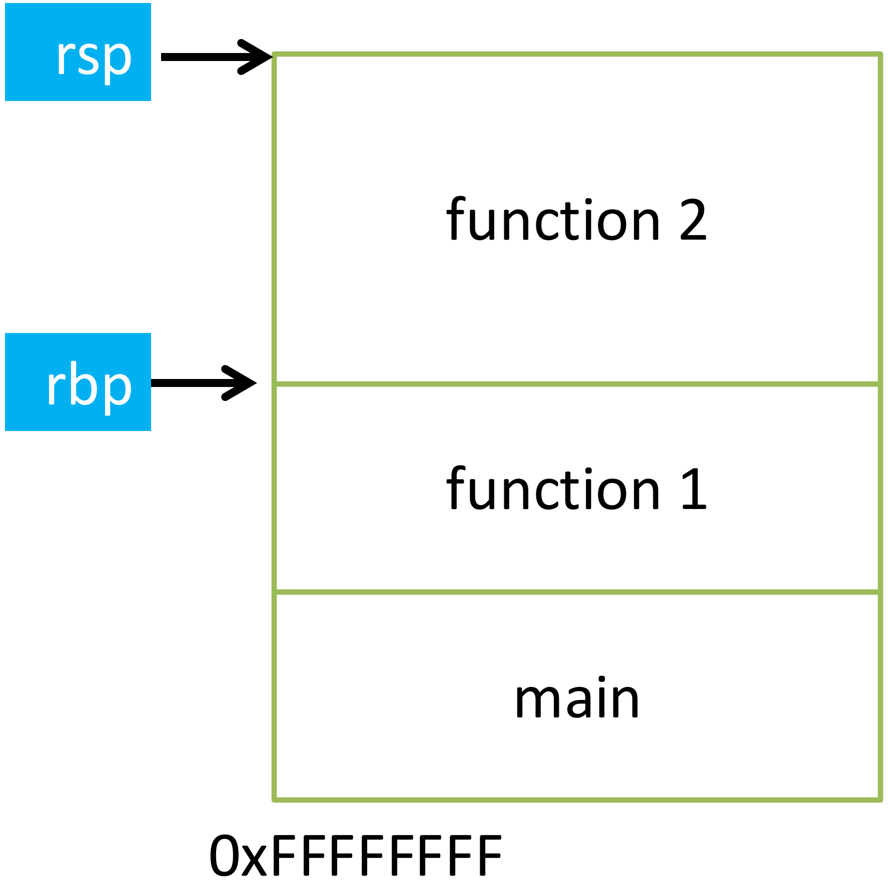
- Working memory for currently executing program
  - Address of next instruction to execute (%rip)
  - Location of runtime stack (%rbp, %rsp)
  - Temporary data (%rax - %r15)
  - Status of recent ALU tests (CF, ZF, SF, OF)



Stack Frame Location

- Compiler ensures that this invariant holds.
- This is why all local variables we've seen in assembly are relative to `rbp` or `rsp`!

invariant:
The current function's stack frame is always between the addresses stored in `rsp` and `rbp`



How would we implement pushing x to the top of the stack in x86_64?

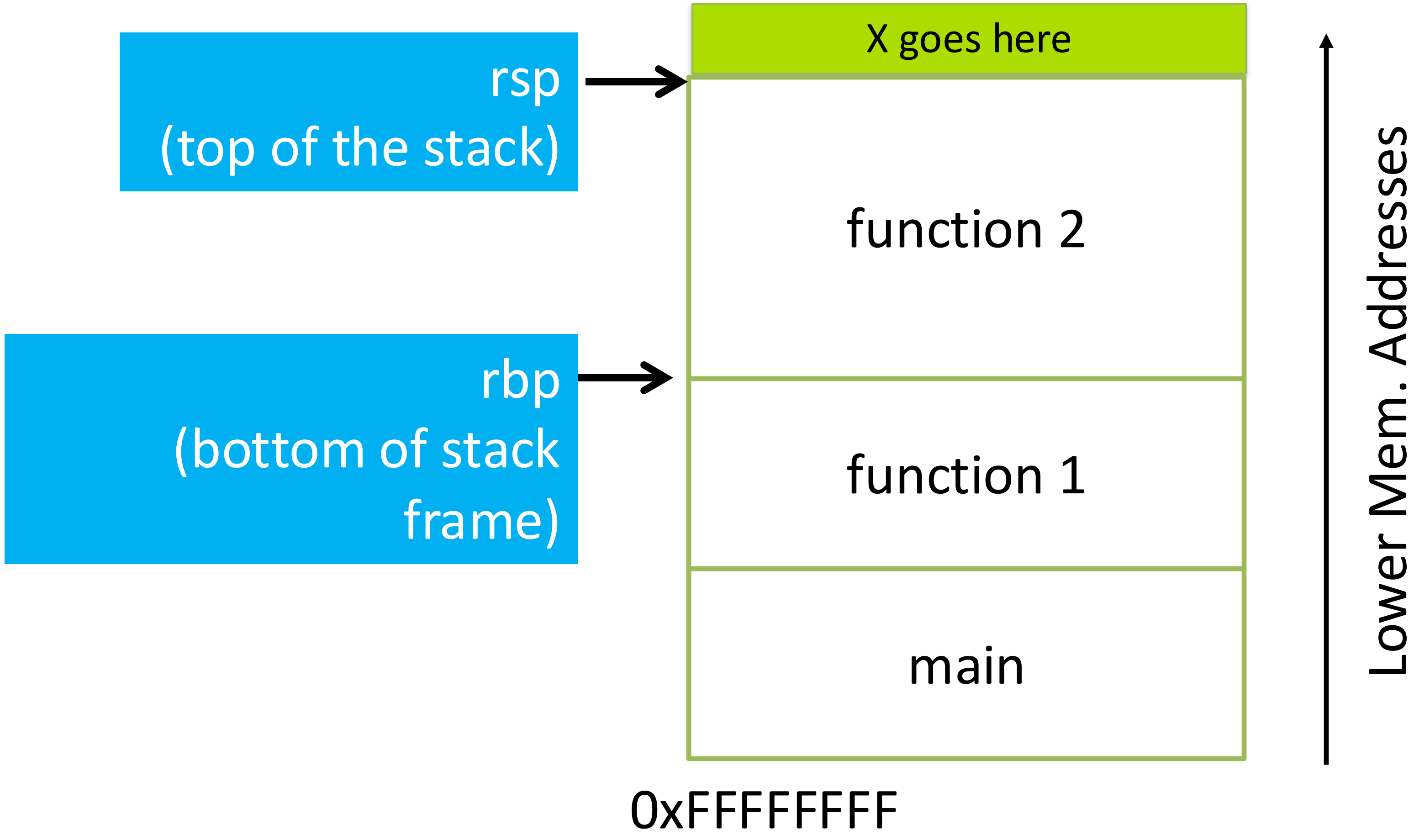
A. Increment rsp
Store x at (rsp)

B. Store x at (rsp)
Increment rsp

C. Decrement rsp
Store x at (rsp)

D. Store x at (rsp)
Decrement rsp

E. Copy rsp to rbp
Store x at rbp



How would we implement pushing x to the top of the stack in x86_64?

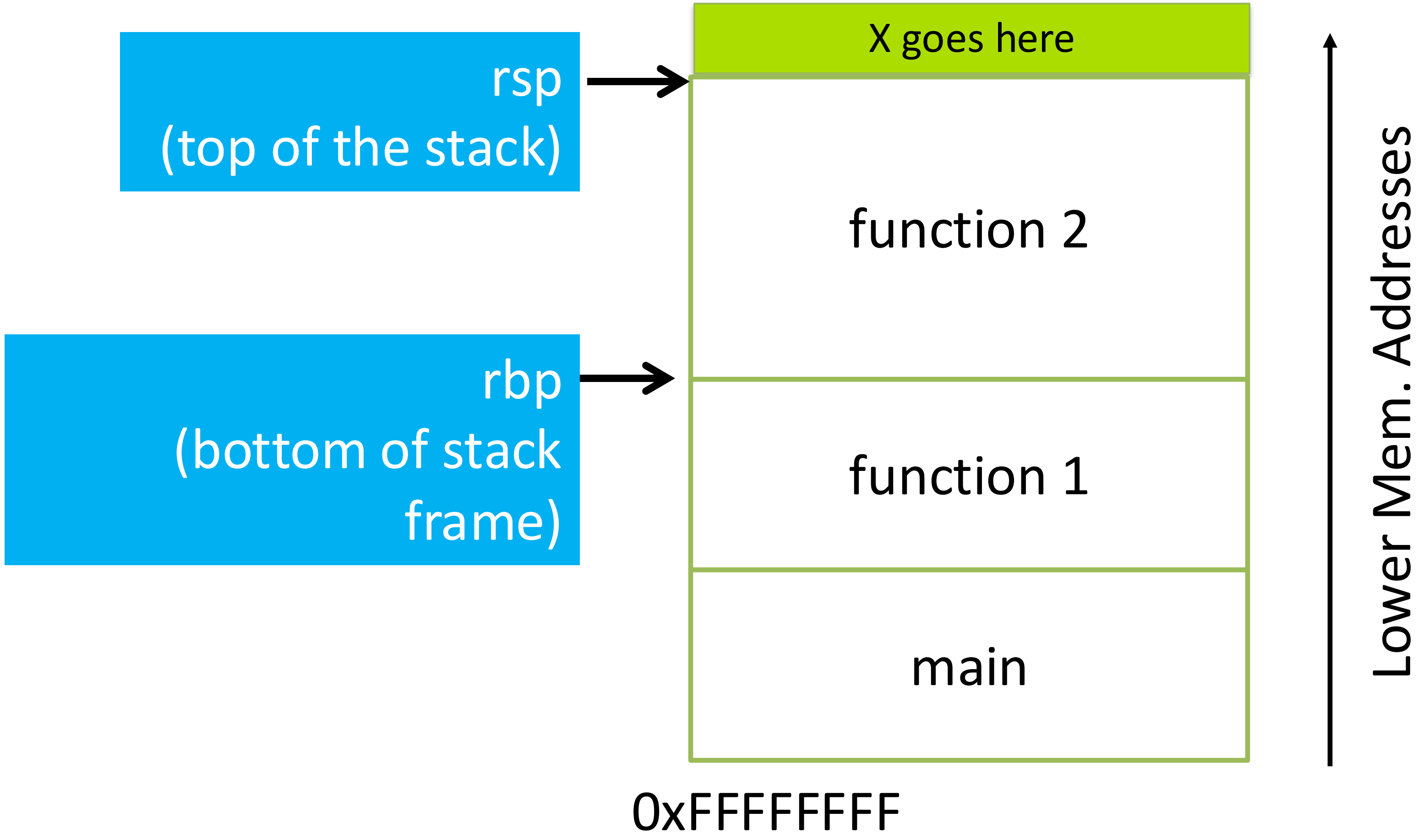
A. Increment rsp
   Store x at (rsp)

B. Store x at (rsp)
   Increment rsp

C. Decrement rsp
   Store x at (rsp)

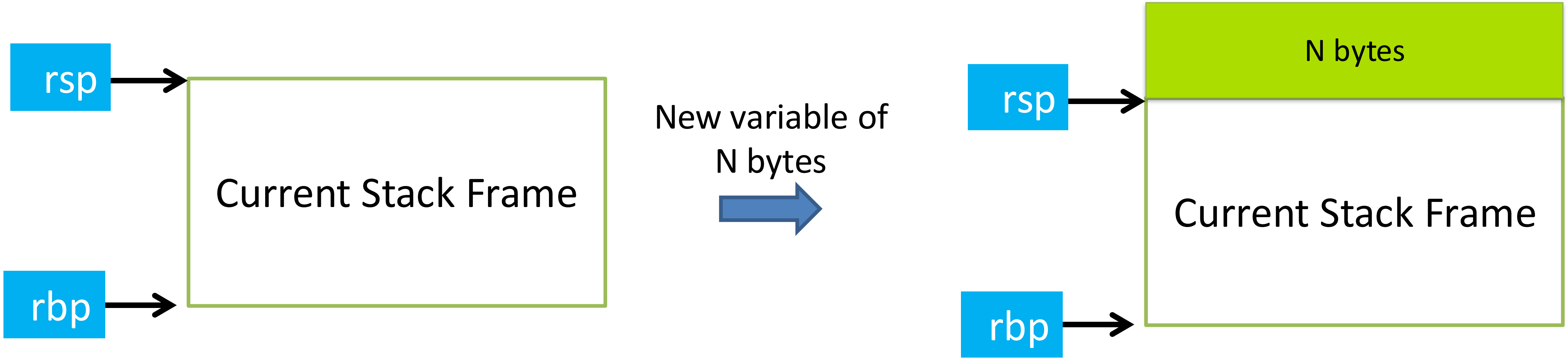
D. Store x at (rsp)
   Decrement rsp

E. Copy rsp to rbp
   Store x at rbp



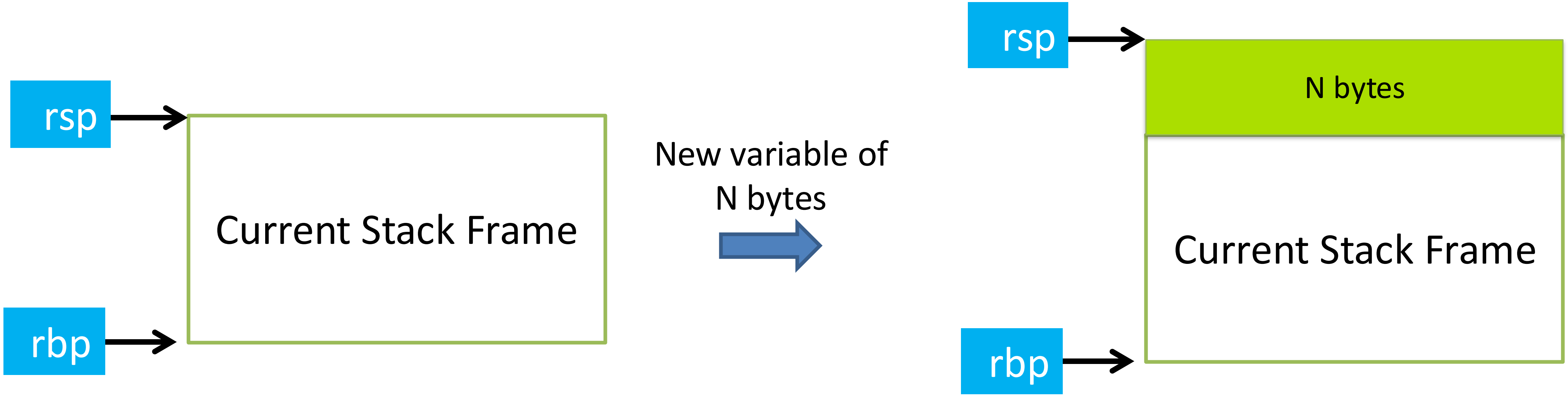
Local Variables

- Generally, we can make space on the stack for N bytes by:
  - <u>subtracting N from rsp</u>



Local Variables

- When we’re done, free the space by adding N back to `rsp`
  - `rsp + N`

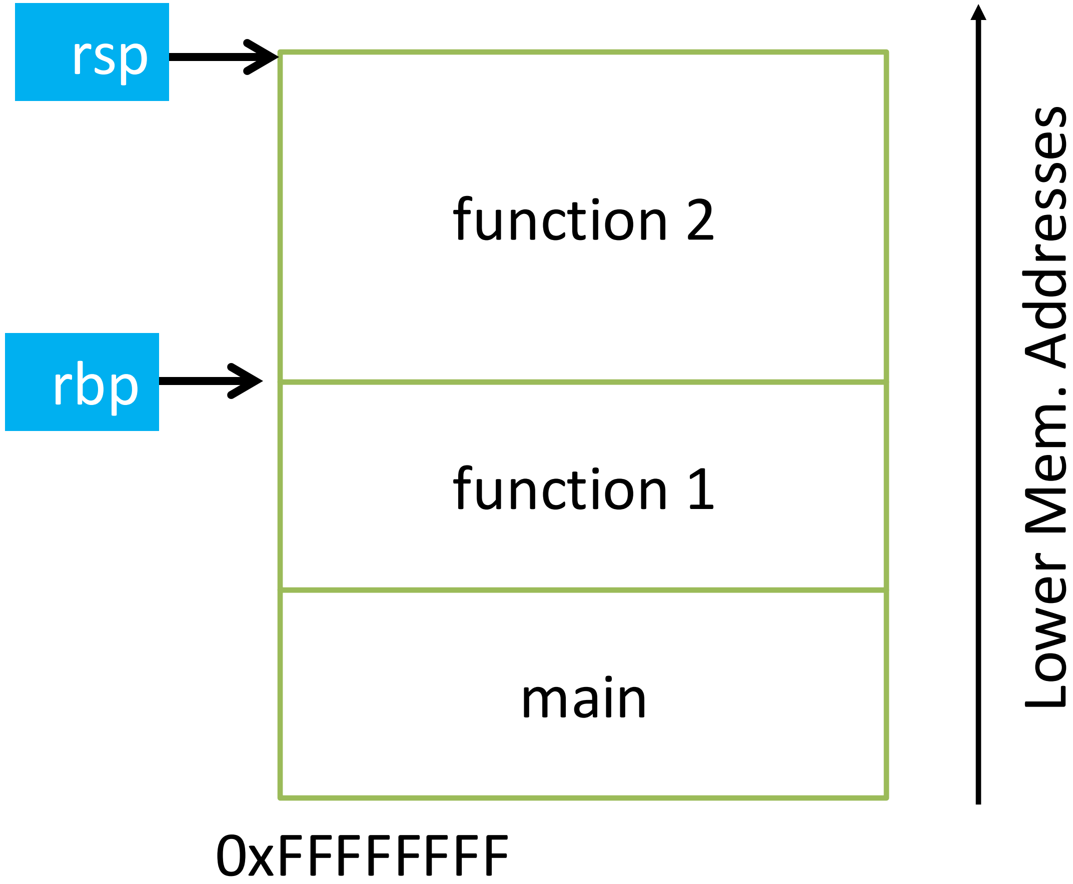


Stack Frame Contents

What needs to be stored in a stack frame? What *must* a function know?

- Local variables
- Previous stack frame base address
- Function arguments
- Return value
- Return address

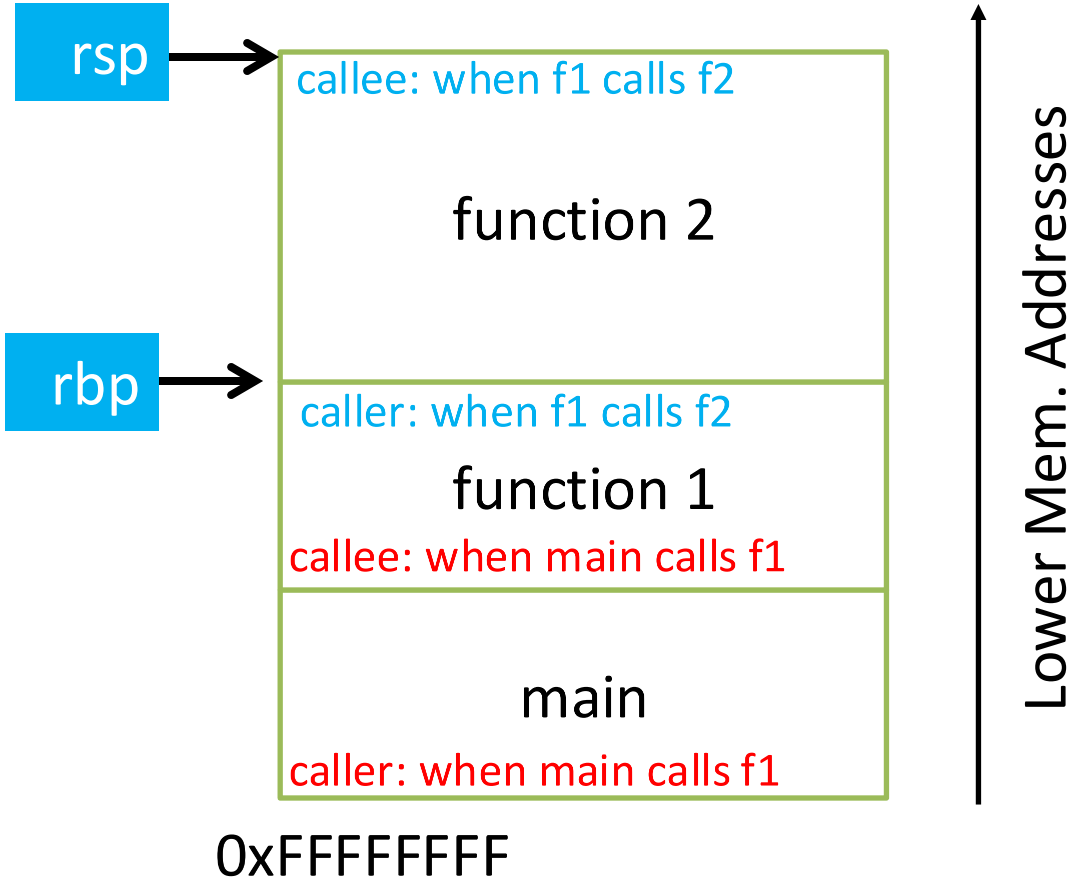
- Saved registers
- Spilled temporaries



Stack Frame Relationships

- If function 1 calls function 2:
  - function 1 is the *caller*
  - function 2 is the *callee*

- With respect to main:
  - main is the *caller*
  - function 1 is the *callee*



Where should we store the following stuff?

```
Previous stack frame base address
Function arguments
Return value
Return address
```

A. In registers

B. On the heap

C. In the caller’s stack frame

D. In the callee’s stack frame

E. Somewhere else

Calling Convention

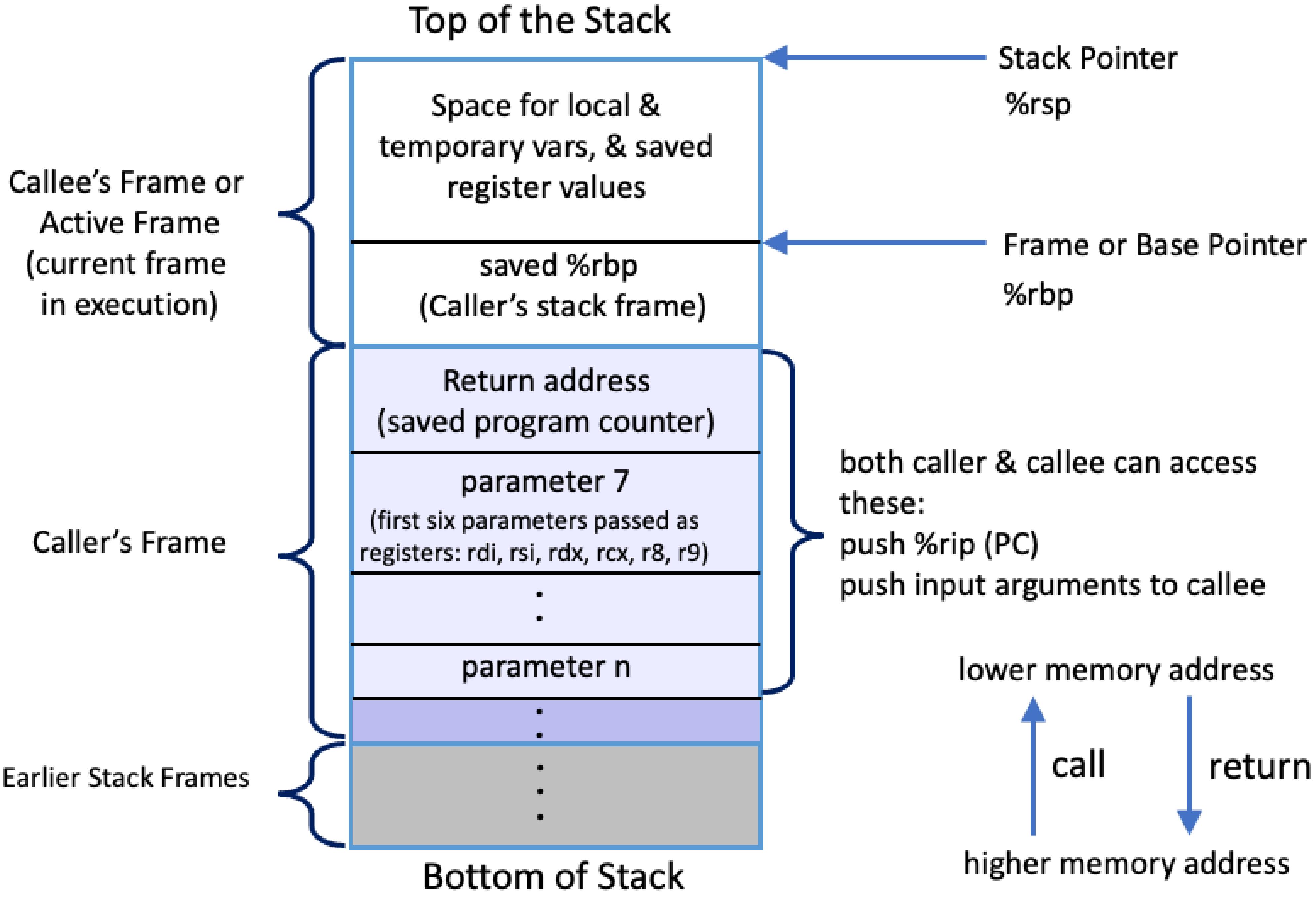
- You could store this stuff wherever you want!
  - The hardware does NOT care.
  - What matters: everyone agrees on where to find the necessary data.

- Calling convention: agreed upon system for exchanging data between caller and callee

- When possible, keep values in registers (why?)
  - Accessing registers is faster than memory (stack)

x86_64 Calling Convention

- The function’s return value: In register %rax
- The caller’s %rbp value (caller’s saved frame pointer)
  - Placed on the stack in the callee’s stack frame
- The return address (saved PC value to resume execution on return)
  - Placed on the stack in the caller’s stack frame
- Arguments passed to a function:
  - First six passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)
  - Any additional arguments stored on the caller’s stack frame (shared with callee)



Top of the Stack

Space for local & temporary vars, & saved register values

saved %rbp (Caller's stack frame)

Return address (saved program counter)

parameter 7

(first six parameters passed as registers: rdi, rsi, rdx, rcx, r8, r9)

parameter n

Bottom of Stack

Stack Pointer %rsp

Frame or Base Pointer %rbp

Callee's Frame or Active Frame (current frame in execution)

Caller's Frame

Earlier Stack Frames

both caller & callee can access these:
push %rip (PC)
push input arguments to callee

lower memory address

call

return

higher memory address

x86_64 Calling Convention

- The function's return value: In register %rax

- The caller's %rbp value (caller's saved frame pointer)
  - Placed on the stack in the callee's stack frame

- The return address (saved PC value to resume execution on return)
  - Placed on the stack in the caller's stack frame

- Arguments passed to a function:
  - First six passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)
  - Any additional arguments stored on the caller's stack frame (shared with callee)

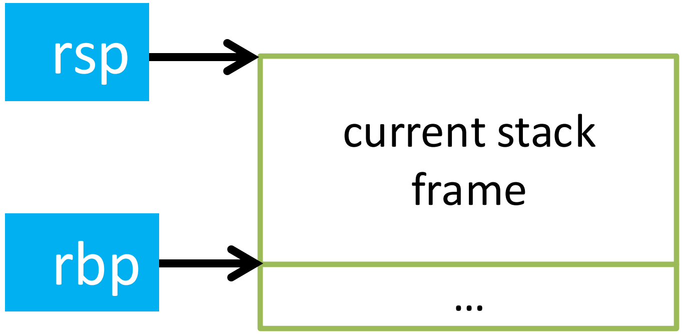
Return Value

- If the callee function produces a result, the caller can find it in %rax

- We saw this when we wrote our function in the weekly lab last friday
  - Copy the result to %rax before we finishing up

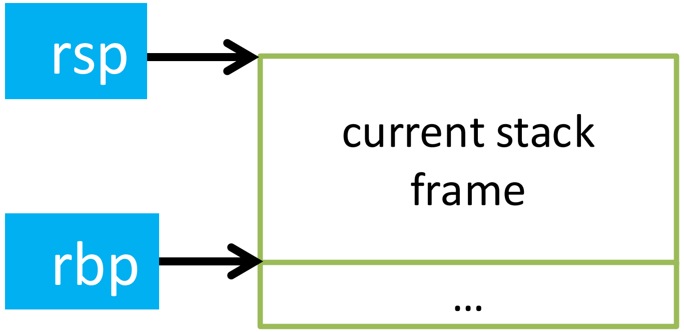
Dynamic Stack Accounting

- Dedicate CPU registers for stack bookkeeping
  - %rsp (stack pointer): Top of current stack frame
  - %rbp (frame pointer): Base of current stack frame
- Compiler maintains these pointers
  - Does the compiler know the exact address they point to?
  - Compiler doesn’t know or care! (job of the OS to figure that out)
- To the compiler: every variable access is relative to %rsp and %rbp!



Compiler: updates to rsp/rbp on function call/return

invariant:
The current function's stack frame is always between the addresses
stored in `rsp` and `rbp`

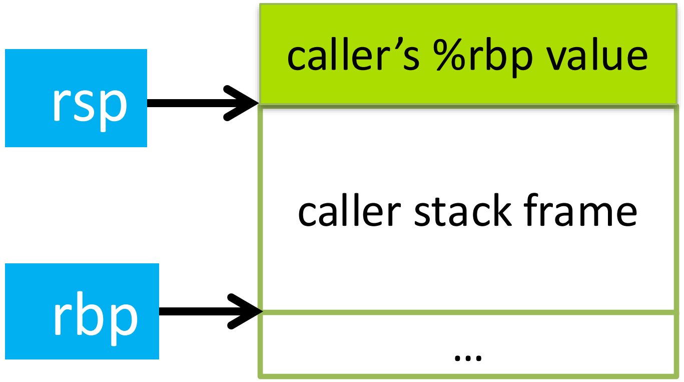


Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp

invariant:
The current function’s stack frame is always between the addresses
stored in `rsp` and `rbp`

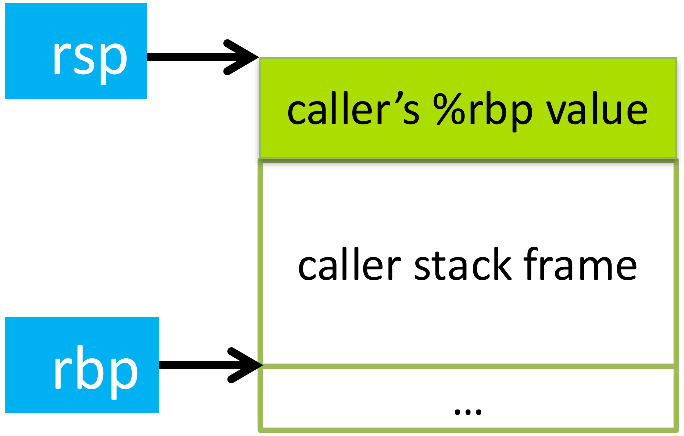


Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp

invariant:
The current function's stack frame is always between the addresses stored in rsp and rbp

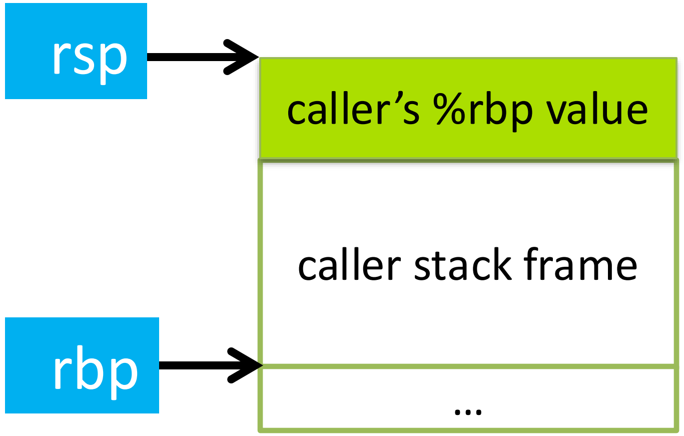


Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp
2. Set %rbp = %rsp

invariant:
The current function's stack frame is always between the addresses
stored in `rsp` and `rbp`

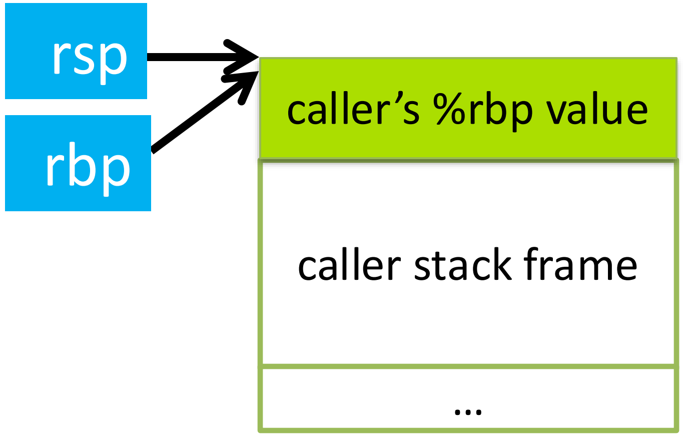


Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp
2. Set %rbp = %rsp

invariant:
The current function's stack frame is always between the addresses
stored in rsp and rbp

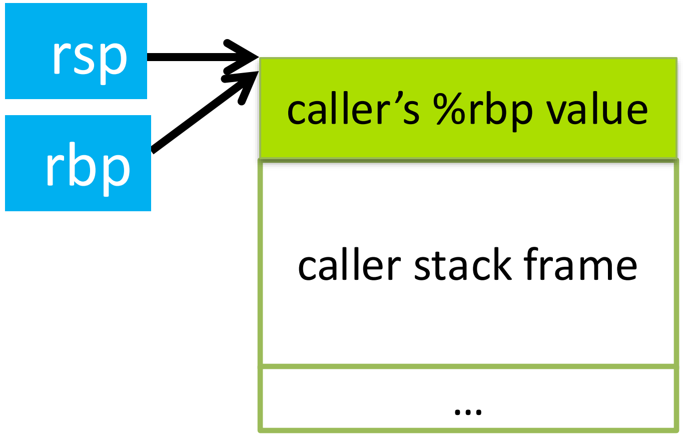


Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp
2. Set %rbp = %rsp
3. Subtract N from %rsp

invariant:
The current function's stack frame is always between the addresses
stored in rsp and rbp

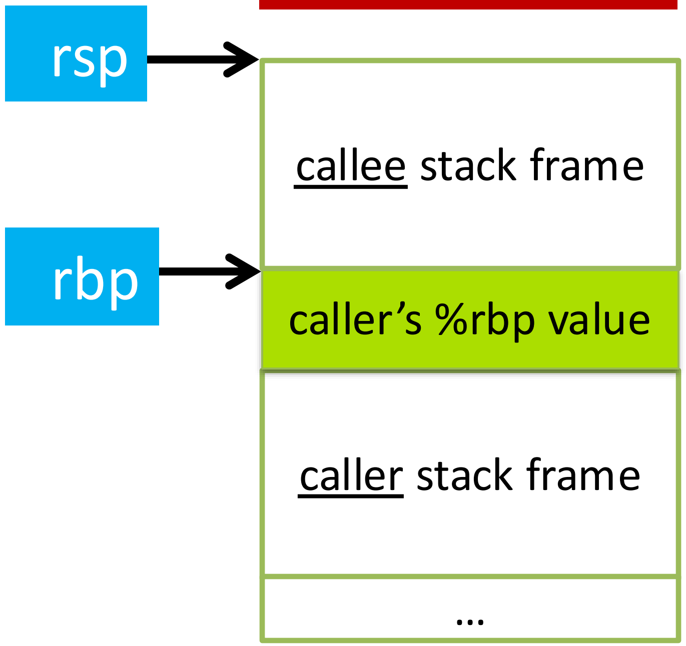


Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp
2. Set %rbp = %rsp
3. Subtract N from %rsp

Callee can now execute.

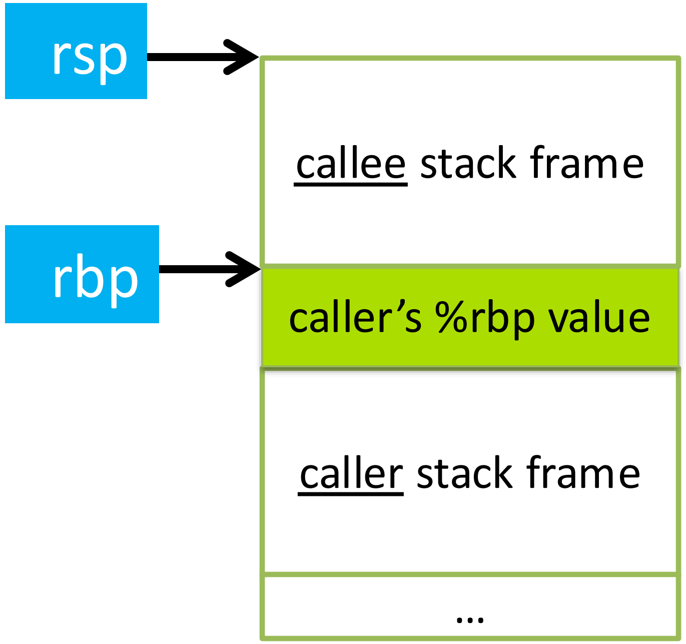


invariant:
The current function’s stack frame is always between the addresses
stored in rsp and rbp

Compiler: Returning from a function call..

Returning from a function:

1. Set %rsp = %rbp

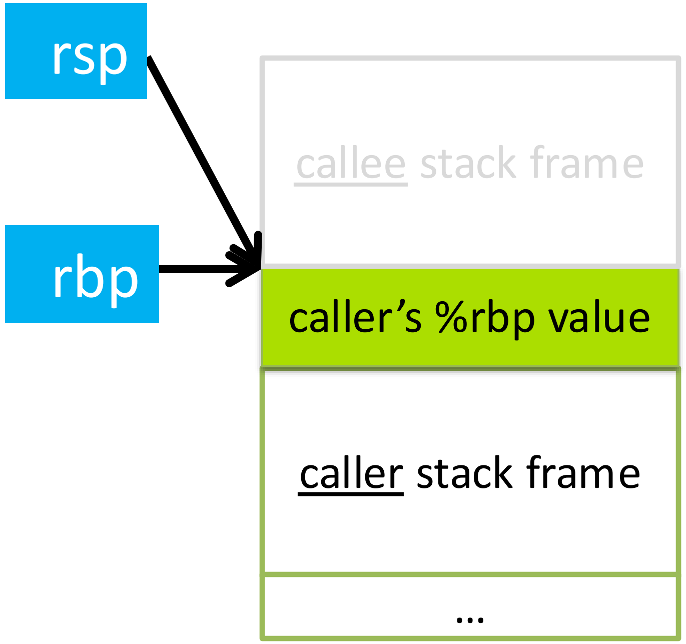


invariant:
The current function's stack frame is always between the addresses stored in rsp and rbp

Compiler: Returning from a function call..

Returning from a function:

1. Set %rsp = %rbp (callee stack frame no longer exists)



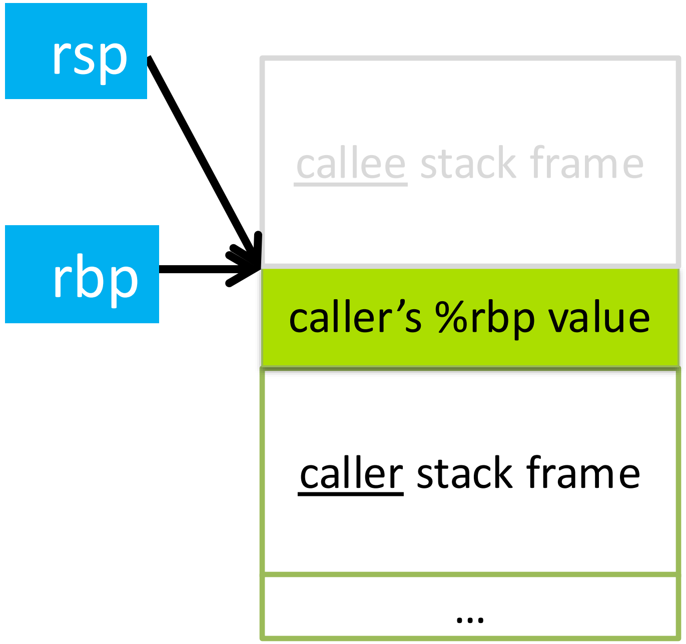
invariant:
The current function's stack frame is always between the addresses
stored in rsp and rbp

Compiler: Returning from a function call..

Returning from a function:

1. Set %rsp = %rbp (callee stack frame no longer exists)
2. pop %rbp

invariant:
The current function's stack frame is always between the addresses
stored in `rsp` and `rbp`



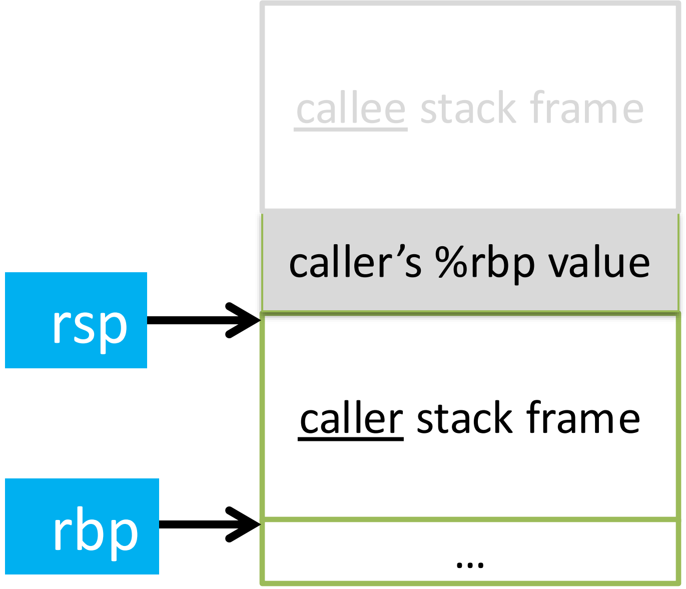
Compiler: Returning from a function call..

Returning from a function:

1. Set %rsp = %rbp
2. pop %rbp
   - pop caller’s rbp off the stack and set it to the value of rbp
   - decrement rsp

X86_64 has another convenience instruction for this: leaveq

invariant:
The current function’s stack frame is always between the addresses
stored in rsp and rbp



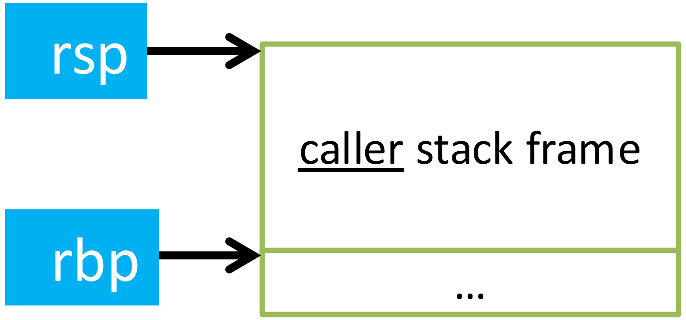
Compiler: Returning from a function call..

Returning from a function:

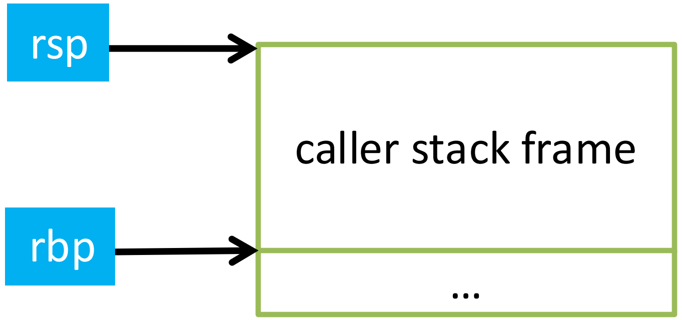
1. Set %rsp = %rbp
2. pop %rbp
   - pop caller’s rbp off the stack and set it to the value of rbp
   - decrement rsp

invariant:
The current function’s stack frame is always between the addresses
stored in `rsp` and `rbp`

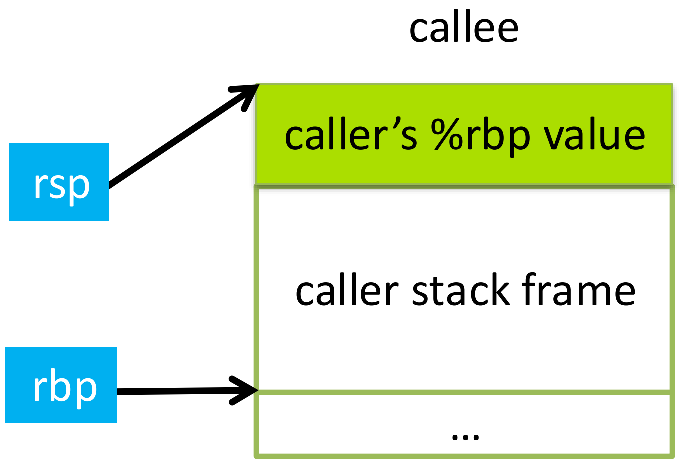
Back to where we started



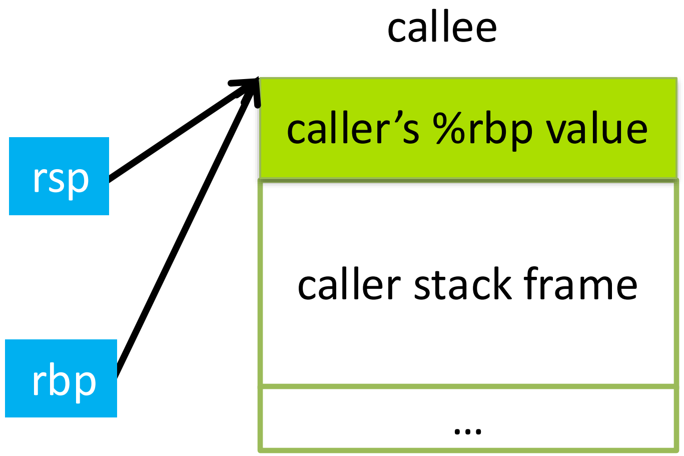
x86 Calling Conventions: Function Call



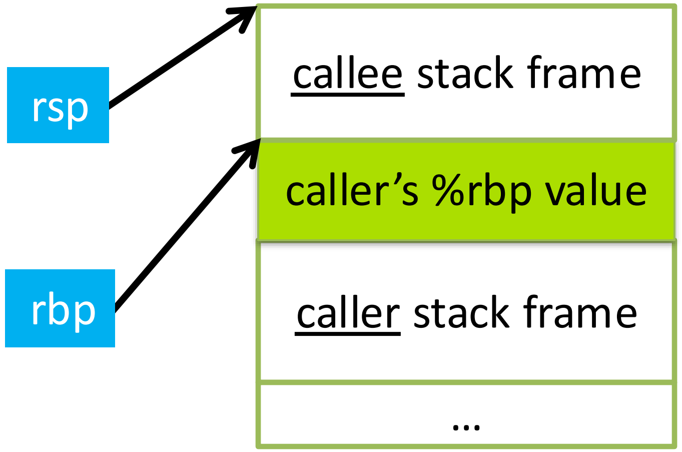
Initial state



push %rbp (store caller’s base pointer)

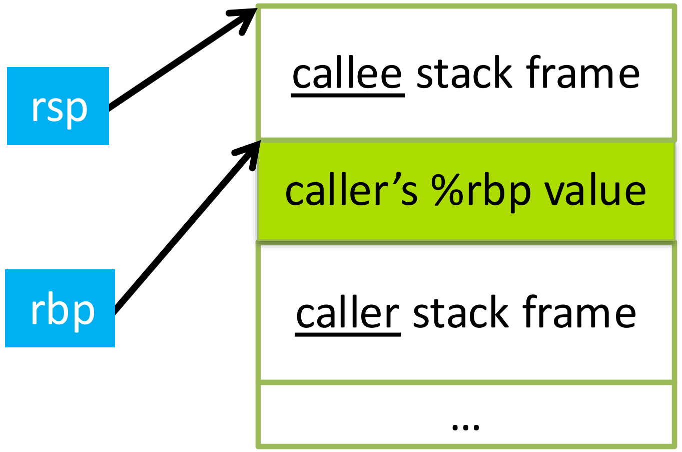


mov %rsp, %rbp
(establish callee’s frame pointer)



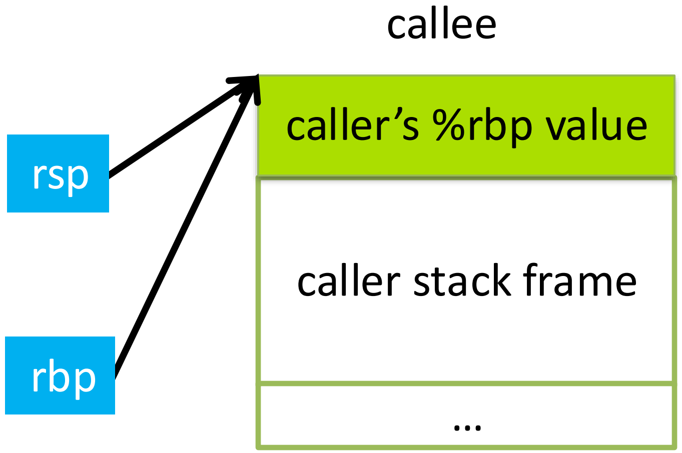
sub $SIZE, %rsp
(allocate space for callee’s locals)

x86 Calling Conventions: Function Return

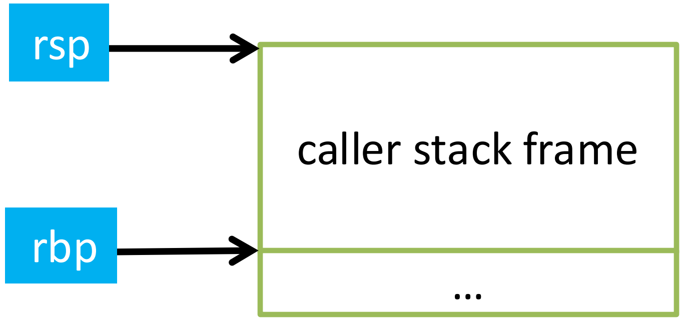


we want to restore the caller's frame

x86_64 provides a convenience instruction that does all of this: `leaveq`



mov %rbp, %rsp
(restore caller's stack pointer)



pop %rbp (restore caller's frame pointer)

x86_64 Calling Convention

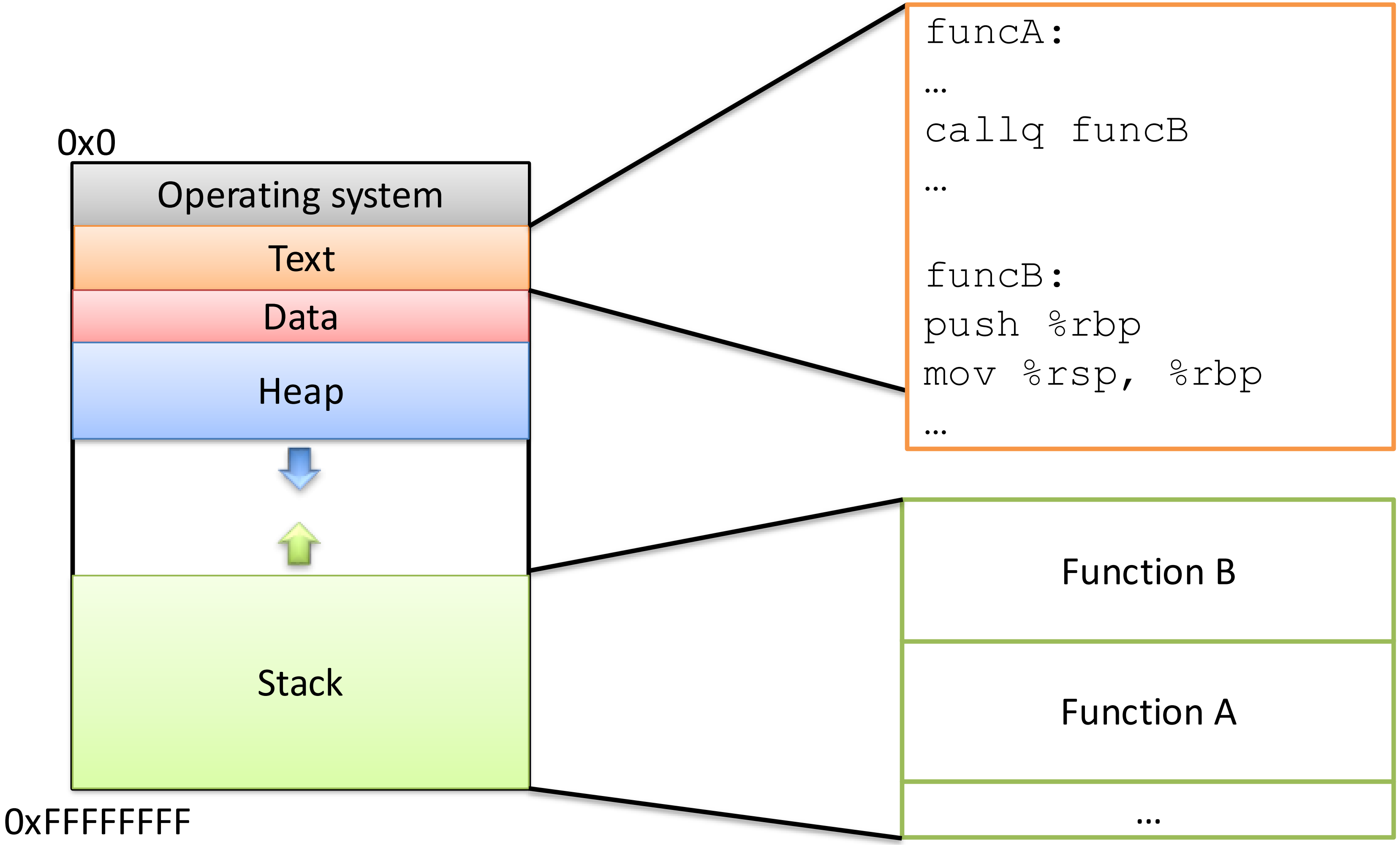
- The function's return value:
  - In register %rax

- The caller's %rbp value (caller's saved frame pointer)
  - Placed on the stack in the callee's stack frame

- The return address (saved PC value to resume execution on return)
  - Placed on the stack in the caller's stack frame

- Arguments passed to a function:
  - First six passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)
  - Any additional arguments stored on the caller's stack frame (shared with callee)

Instructions in Memory



0x0

Operating system

Text

Data

Heap

Stack

0xFFFFFFFF

```
funcA:
…
callq funcB
…

funcB:
push %rbp
mov %rsp, %rbp
…
```

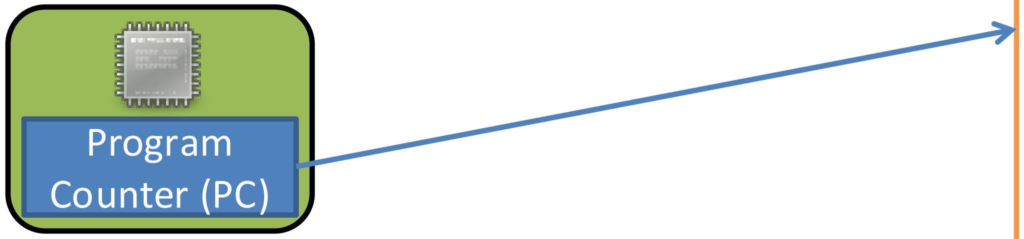
Function B

Function A

…

Program Counter

Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)



What do we do now?

Follow PC, fetch instruction:

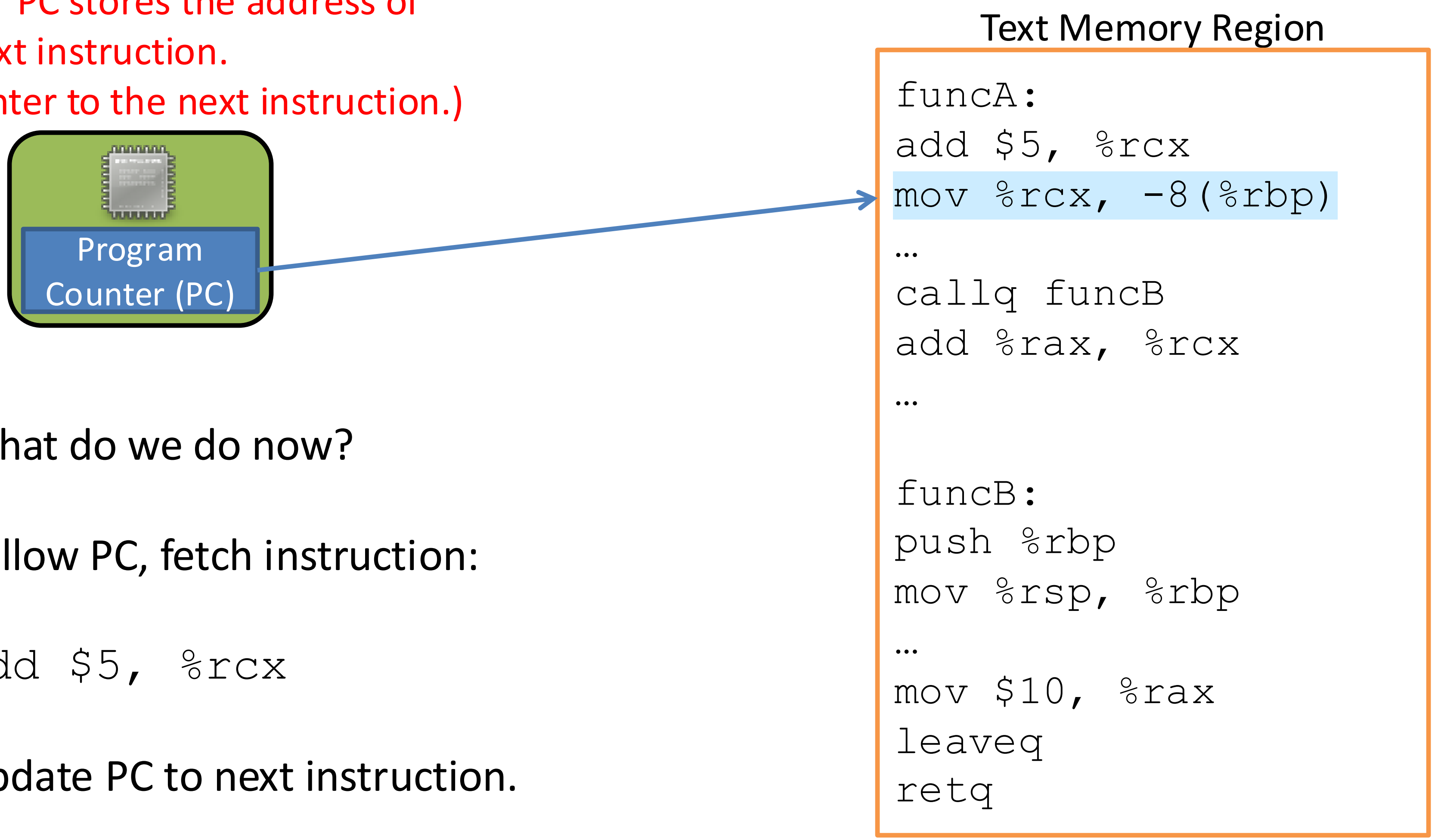
```
add $5, %rcx
```

Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

Program Counter

Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)



```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

What do we do now?

Follow PC, fetch instruction:

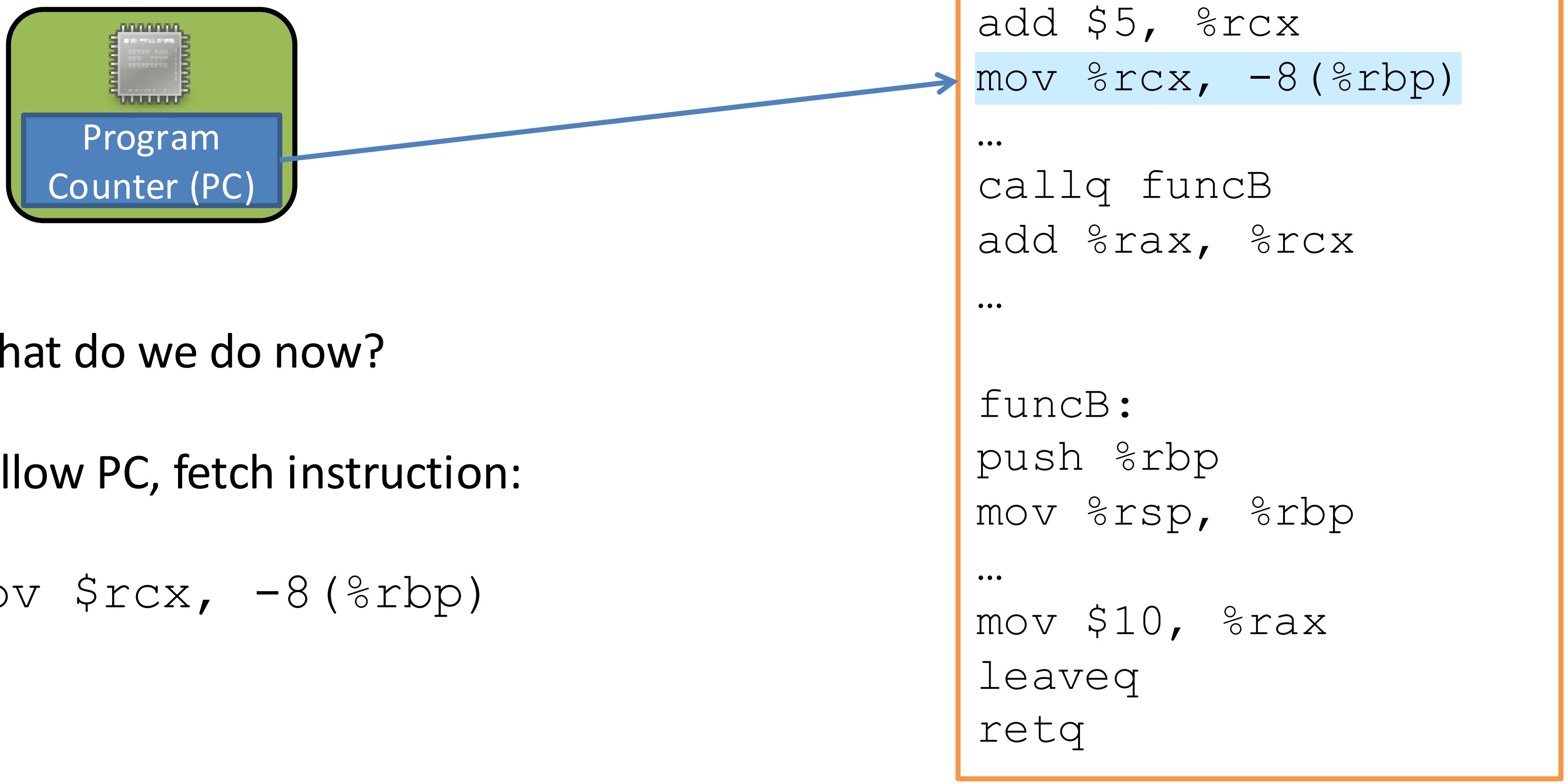
`add $5, %rcx`

Update PC to next instruction.

Execute the `addl`.

Program Counter

Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)



Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

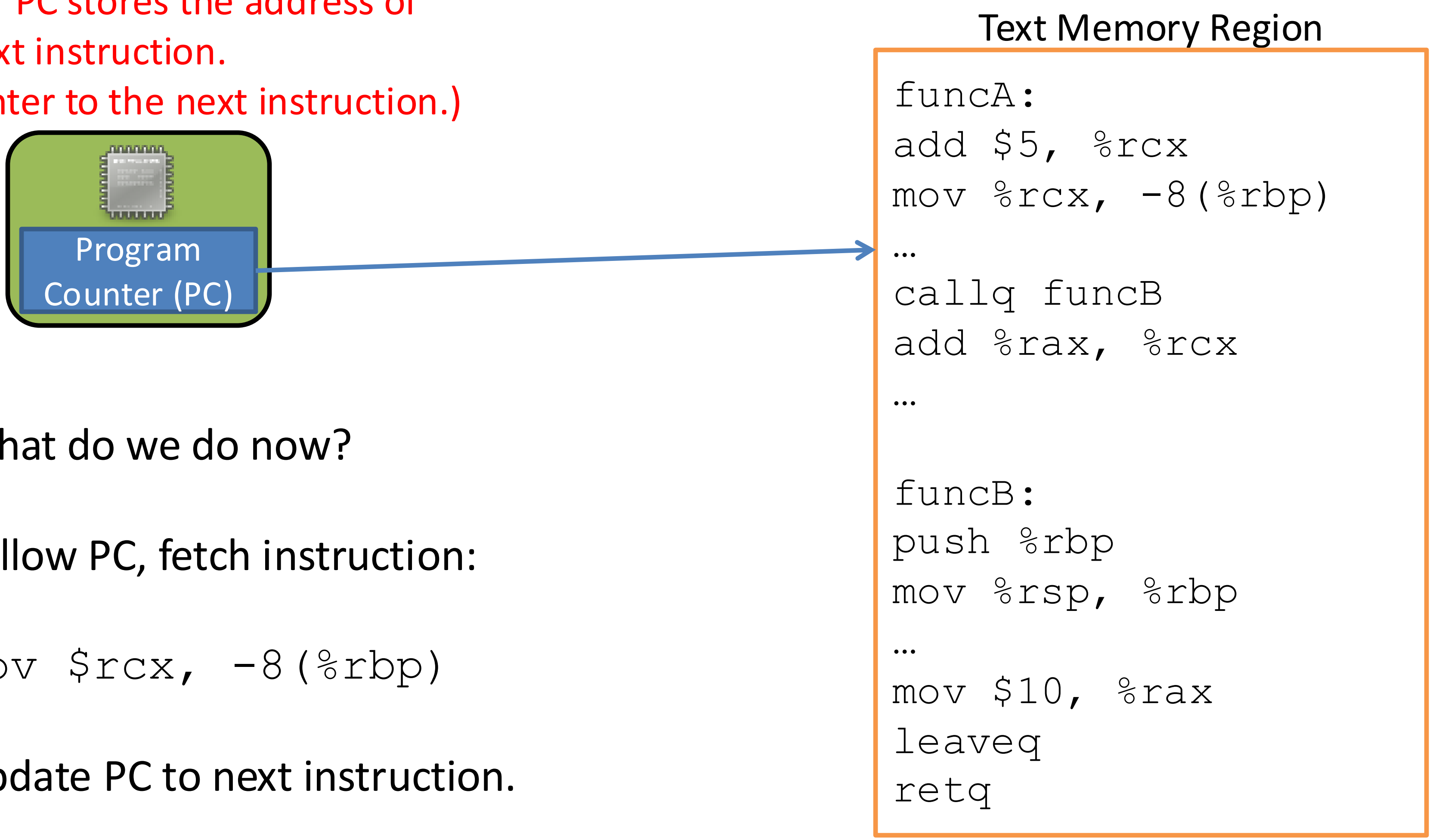
What do we do now?

Follow PC, fetch instruction:

```
mov $rcx, -8(%rbp)
```

Program Counter

Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)



```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

What do we do now?

Follow PC, fetch instruction:

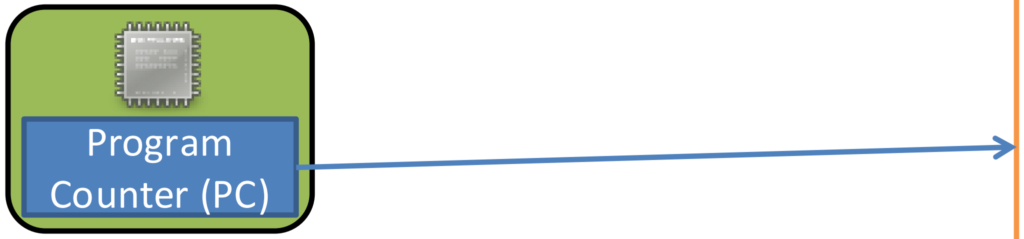
`mov $rcx, -8(%rbp)`

Update PC to next instruction.

Execute the `mov`.

Program Counter

Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)



What do we do now?

Keep executing in a straight line downwards like this until:

We hit a jump instruction.
We call a function.

Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

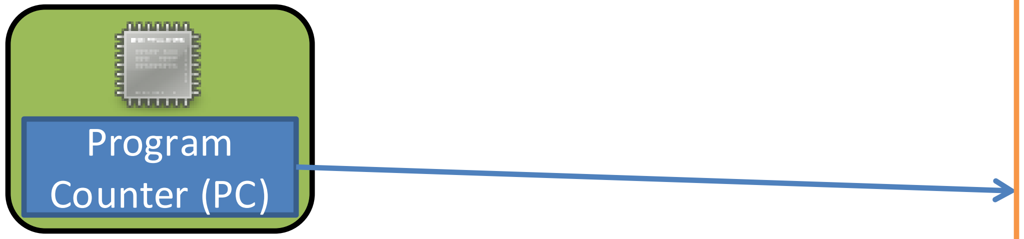
Changing the PC: Jump

- On a (non-function call) jump:
  - Check condition codes
  - Set PC to execute elsewhere (usually not the next instruction)

- Do we ever need to go back to the instruction after the jump?

Maybe (and if so, we'd have a label to jump back to), but usually not.

Changing the PC: Functions

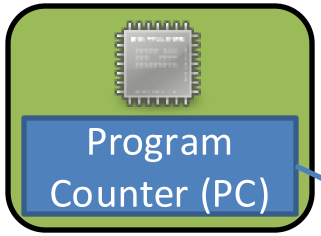


Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

What we’d like this to do:

Changing the PC: Functions



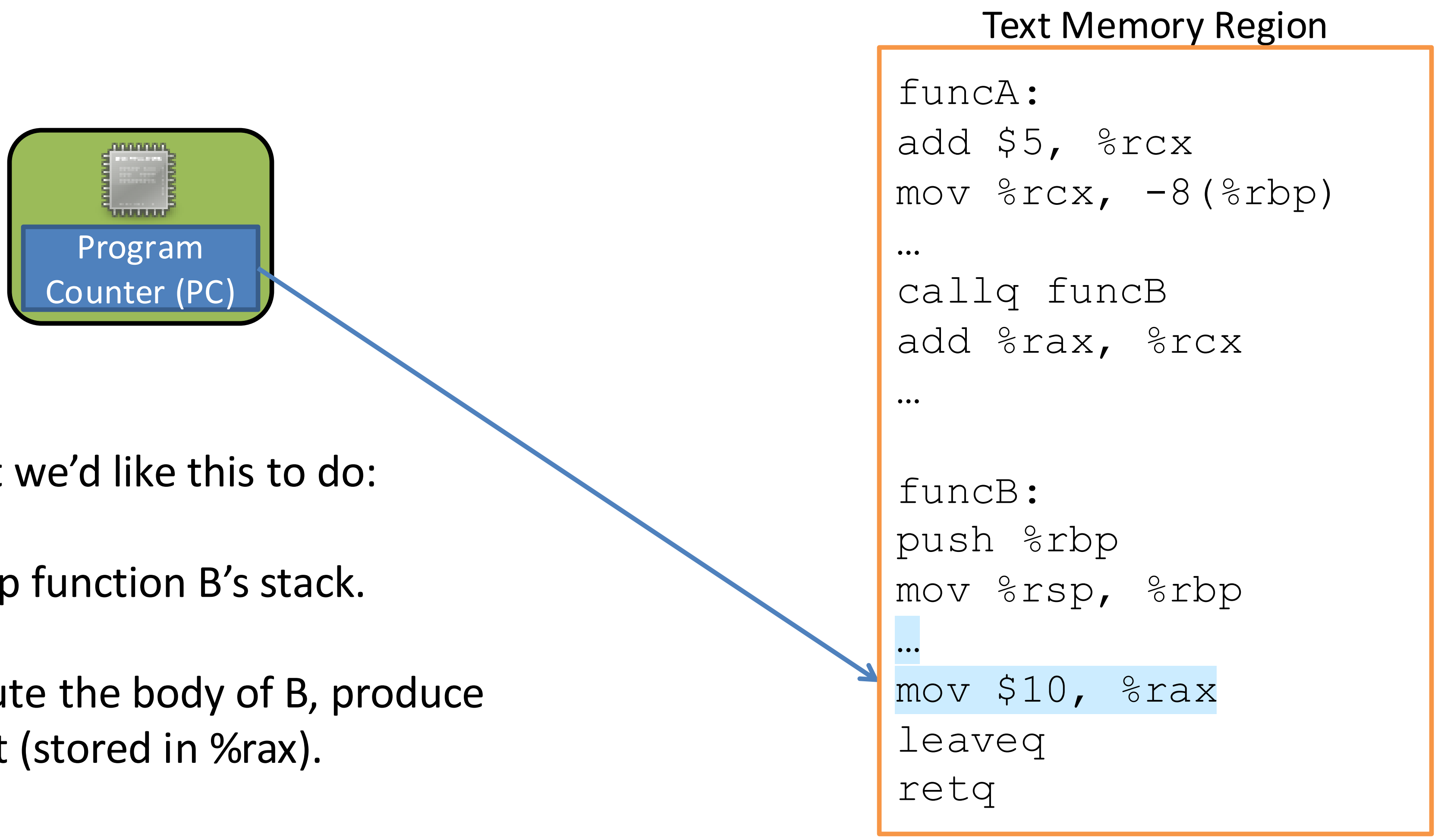
What we'd like this to do:

Set up function B's stack.

Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

Changing the PC: Functions



Text Memory Region

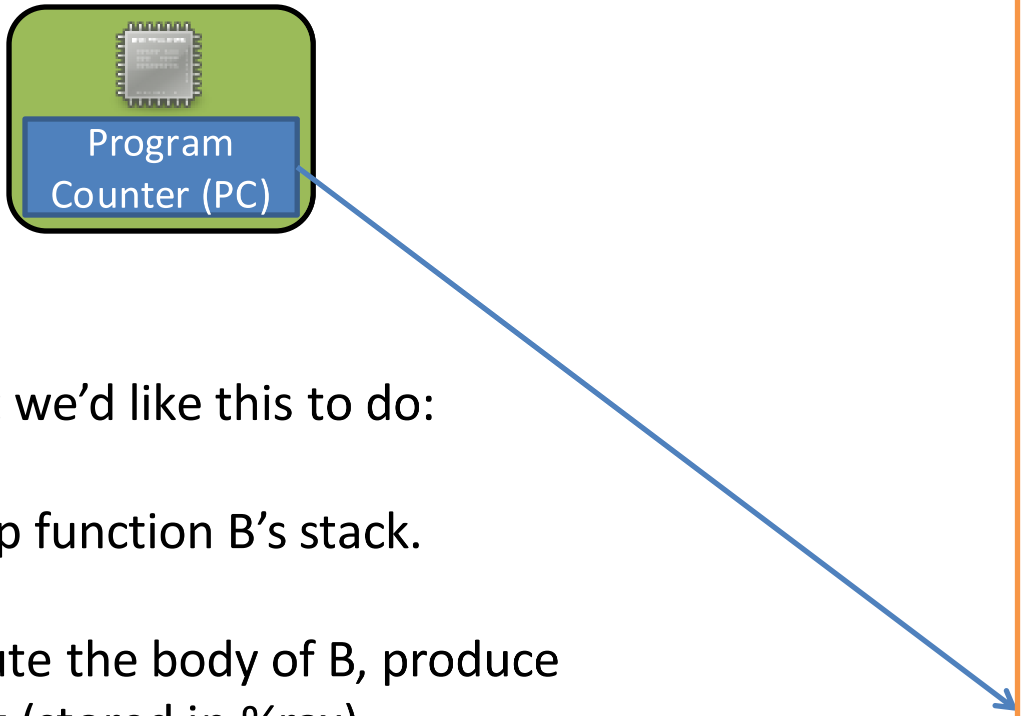
```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

What we’d like this to do:

Set up function B’s stack.

Execute the body of B, produce result (stored in %rax).

Changing the PC: Functions



Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

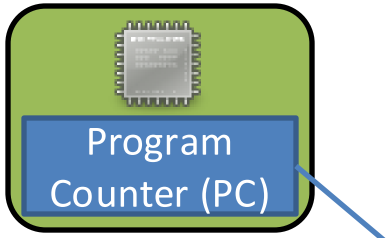
What we'd like this to do:

Set up function B's stack.

Execute the body of B, produce result (stored in %rax).

Restore function A's stack.

Changing the PC: Functions



What we’d like this to do:

Return:
Go back to what we were doing before funcB started.

Unlike jumping, we intend to go back!

Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

Like `push`, `pop`, and `leave`, `call` and `ret` are convenience instructions. What should they do to support the PC-changing behavior we need? (The PC is %rip.)

call

In words:

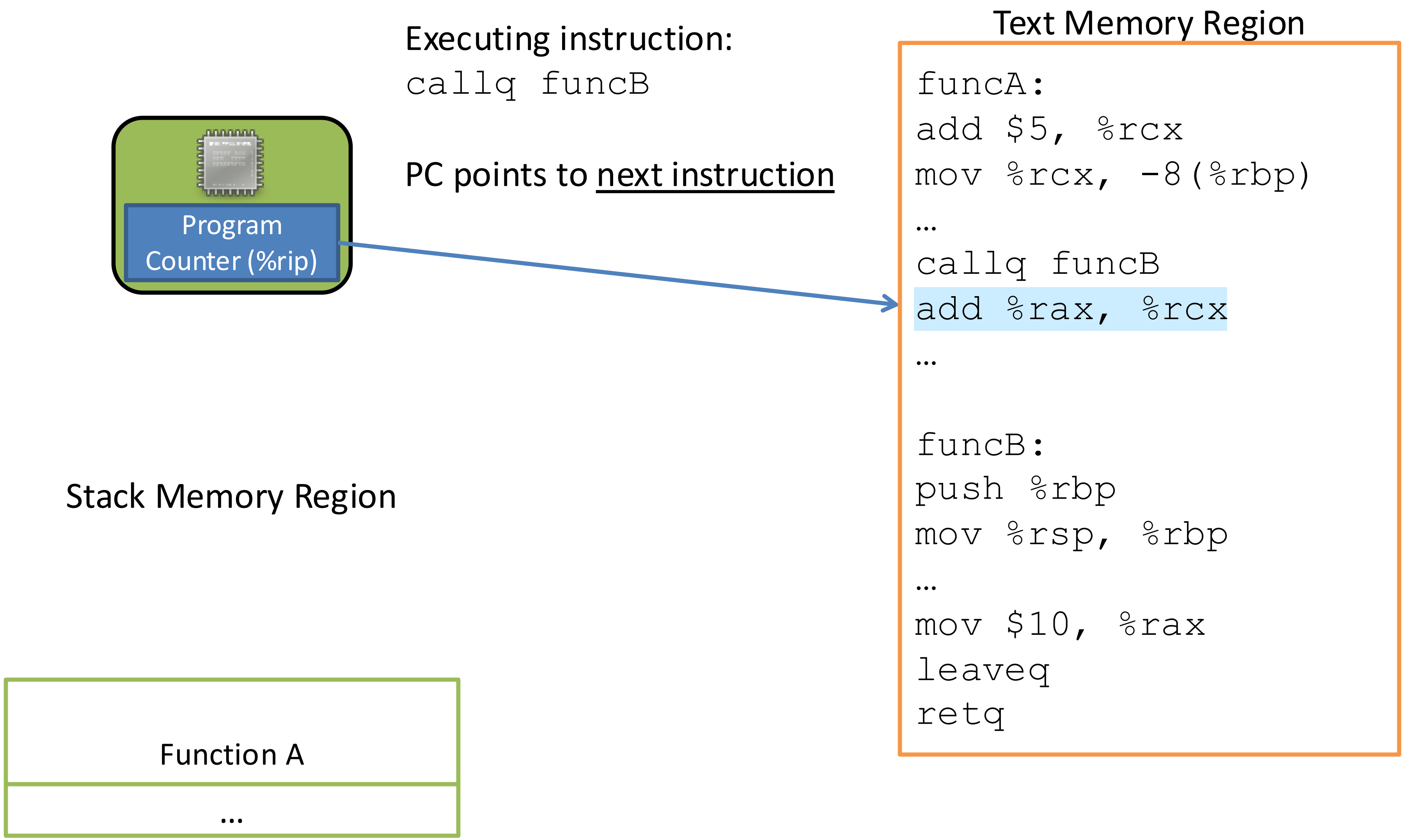
In instructions:

ret

In words:

In instructions:

Functions and the Stack



Executing instruction:
`callq funcB`

PC points to next instruction

Text Memory Region

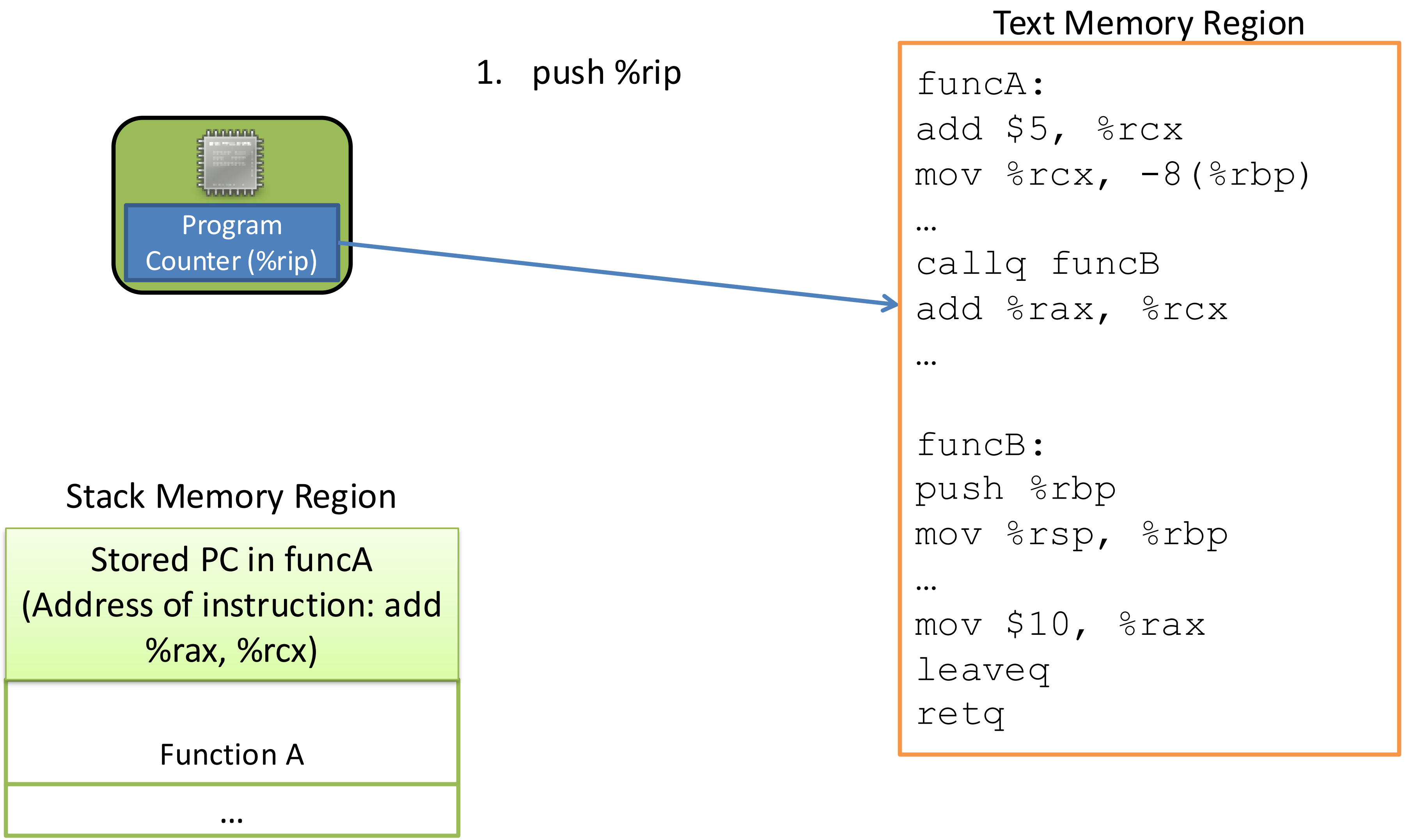
```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

Stack Memory Region

Function A
…

Functions and the Stack

1. push %rip



Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

Stack Memory Region

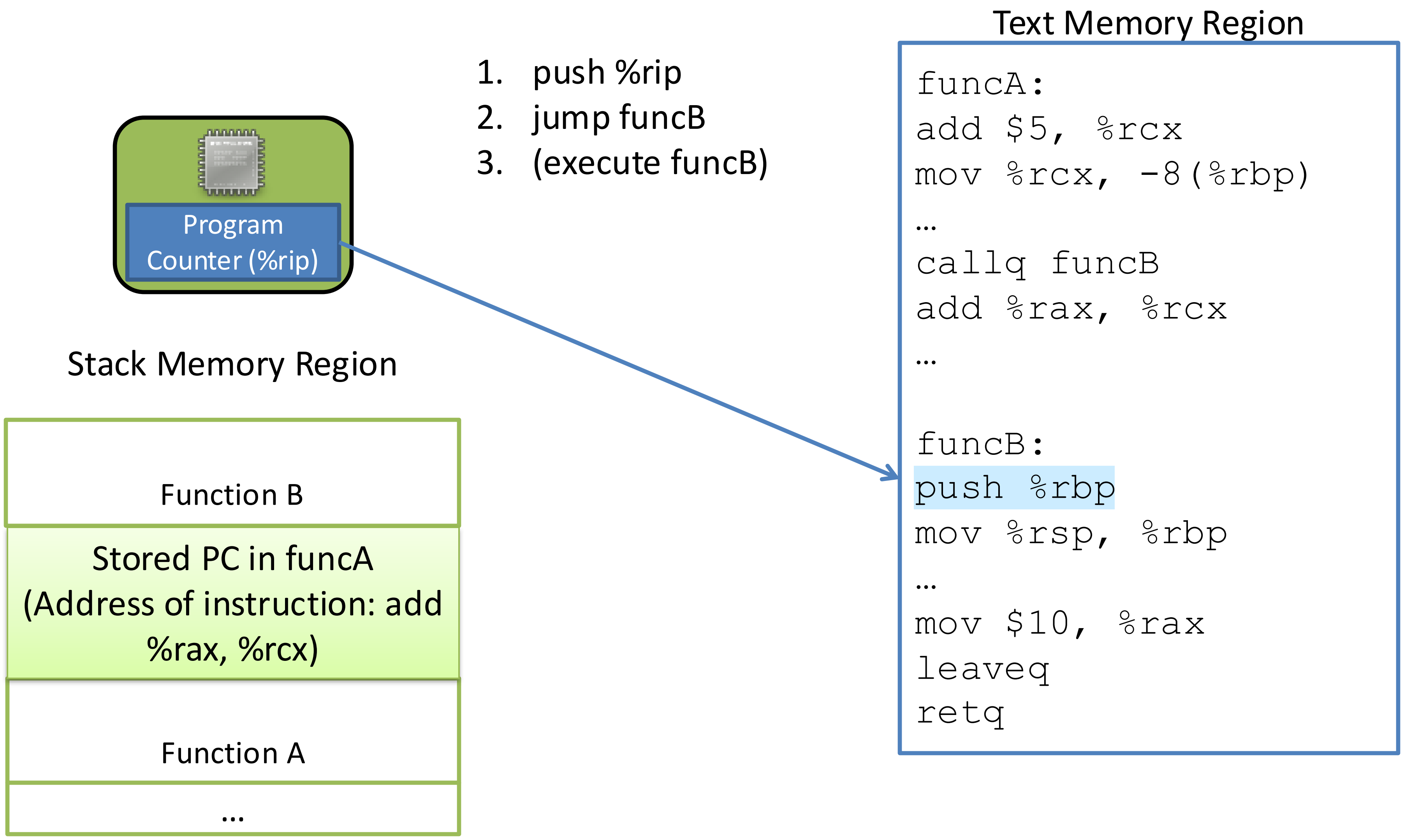
Stored PC in funcA (Address of instruction: add %rax, %rcx)

Function A

…

Functions and the Stack

1. push %rip
2. jump funcB
3. (execute funcB)



Program Counter (%rip)

Stack Memory Region

Function B

Stored PC in funcA (Address of instruction: add %rax, %rcx)

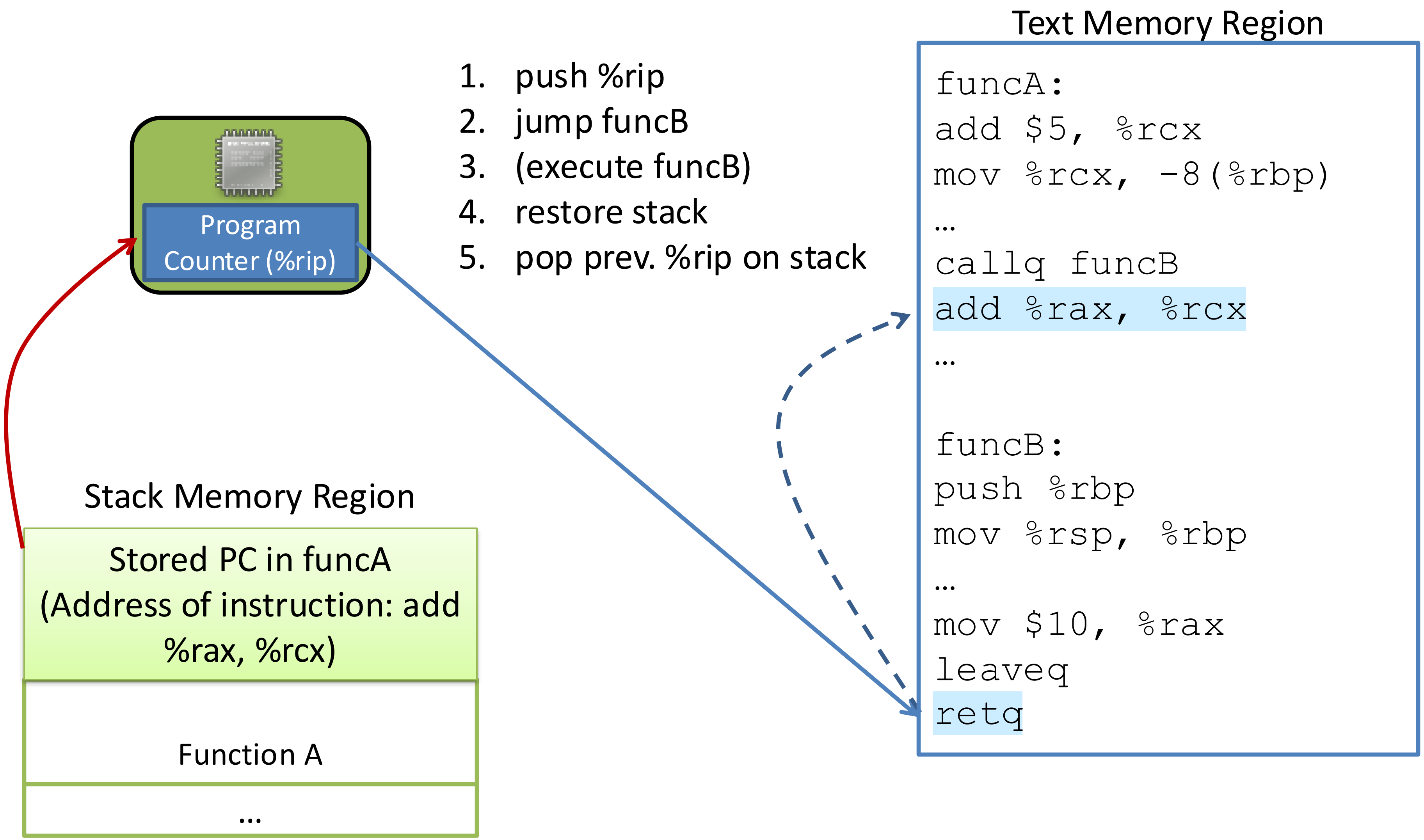
Function A

…

Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

Functions and the Stack



1. push %rip
2. jump funcB
3. (execute funcB)
4. restore stack
5. pop prev. %rip on stack

Program Counter (%rip)

Stack Memory Region

Stored PC in funcA (Address of instruction: add %rax, %rcx)

Function A

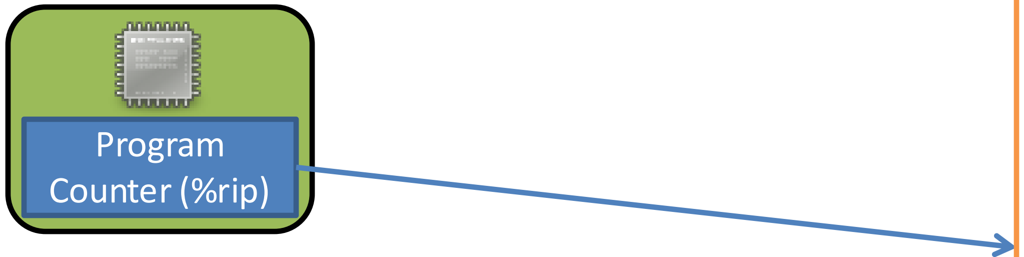
…

Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

Functions and the Stack

6. (resume funcA)



Text Memory Region

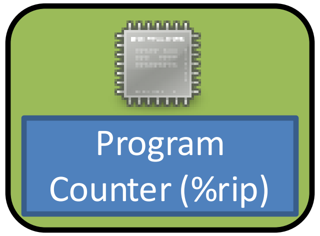
```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

Stack Memory Region

Function A

…

Recap: PC upon a Function Call



1. push %rip
2. jump funcB
3. (execute funcB)
4. restore stack
5. pop prev. %rip on stack
6. (resume funcA)

Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

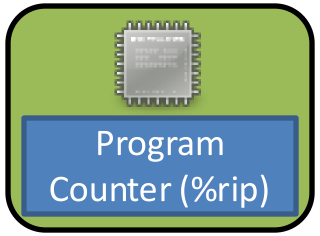
Stack Memory Region

Stored PC in funcA (Address of instruction: add %rax, %rcx)

Function A

…

Functions and the Stack



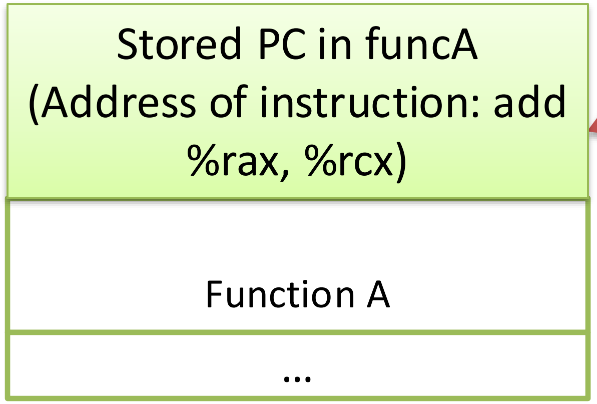
1. push %rip
2. jump funcB
3. (execute funcB)
4. restore stack
5. pop prev. %rip on stack
6. (resume funcA)

callq

leaveq

retq

Stack Memory Region



Stored PC in funcA (Address of instruction: add %rax, %rcx)

Function A

…

Return address:

Address of the instruction we should jump back to when we finish (return from) the currently executing function.

x86_64 Stack / Function Call Instructions

push	Create space on the stack and place the source there.	sub $8, %rsp mov src, (%rsp)
pop	Remove the top item off the stack and store it at the destination.	mov (%rsp), dst add $8, %rsp
callq	1. Push return address on stack 2. Jump to start of function	push %rip jmp target
leaveq	Prepare the stack for return (restoring caller’s stack frame)	mov %rbp, %rsp pop %rbp
retq	Return to the caller, PC ← saved PC (pop return address off the stack into PC (rip))	pop %rip

x86_64 Calling Convention

- The function’s return value:
  - In register %rax

- The caller’s %rbp value (caller’s saved frame pointer)
  - Placed on the stack in the callee’s stack frame

- The return address (saved PC value to resume execution on return)
  - Placed on the stack in the caller’s stack frame

- Arguments passed to a function:
  - First six passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)
  - Any additional arguments stored on the caller’s stack frame (shared with callee)

Function Arguments

- Most functions don’t receive more than 6 arguments, so x86_64 can simply use registers most of the time.

- If we *do* have more than 6 arguments though (e.g., perhaps a `printf` with lots of placeholders), we can’t fit them all in registers.

- In that case, we need to store the extra arguments on the stack. By convention, they go in the caller’s stack frame.

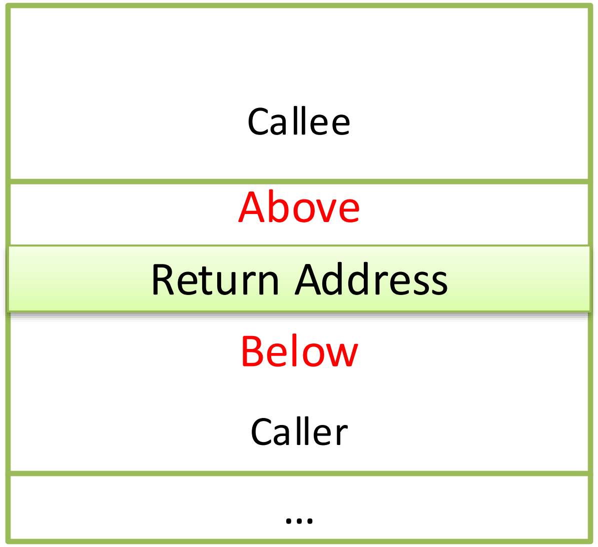
If we need to place arguments in the caller's stack frame, should they go above or below the return address?

A. Above

B. Below

C. It doesn't matter

D. Somewhere else



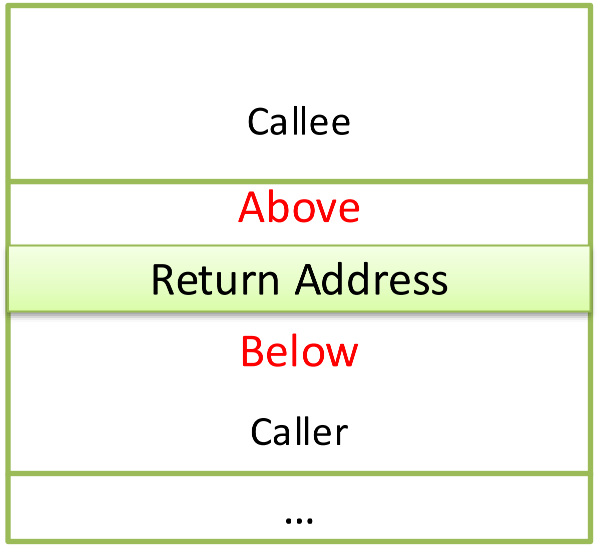
If we need to place arguments in the caller's stack frame, should they go above or below the return address?

A. Above

B. Below

C. It doesn't matter

D. Somewhere else



x86_64 Stack / Function Call Instructions

`push`	Create space on the stack and place the source there.	`sub $8, %rsp`
		`mov src, (%rsp)`

`pop`	Remove the top item off the stack and store it at the destination.	`mov (%rsp), dst`
		`add $8, %rsp`

`callq`	1. Push return address on stack
	2. Jump to start of function
		`push %rip`
		`jmp target`

`leaveq`	Prepare the stack for return (restoring caller's stack frame)	`mov %rbp, %rsp`
		`pop %rbp`

`retq`	Return to the caller, PC ← saved PC (pop return address off the stack into PC (rip))	`pop %rip`

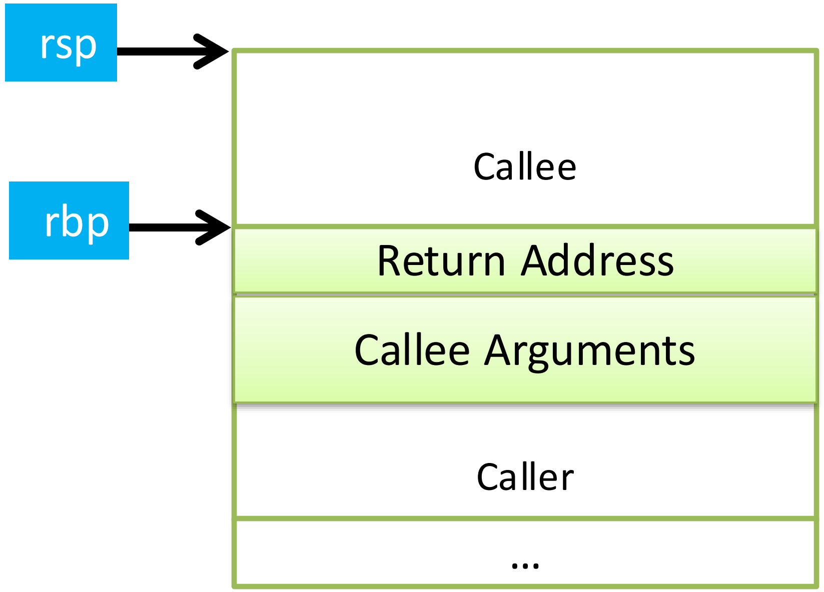
Arguments

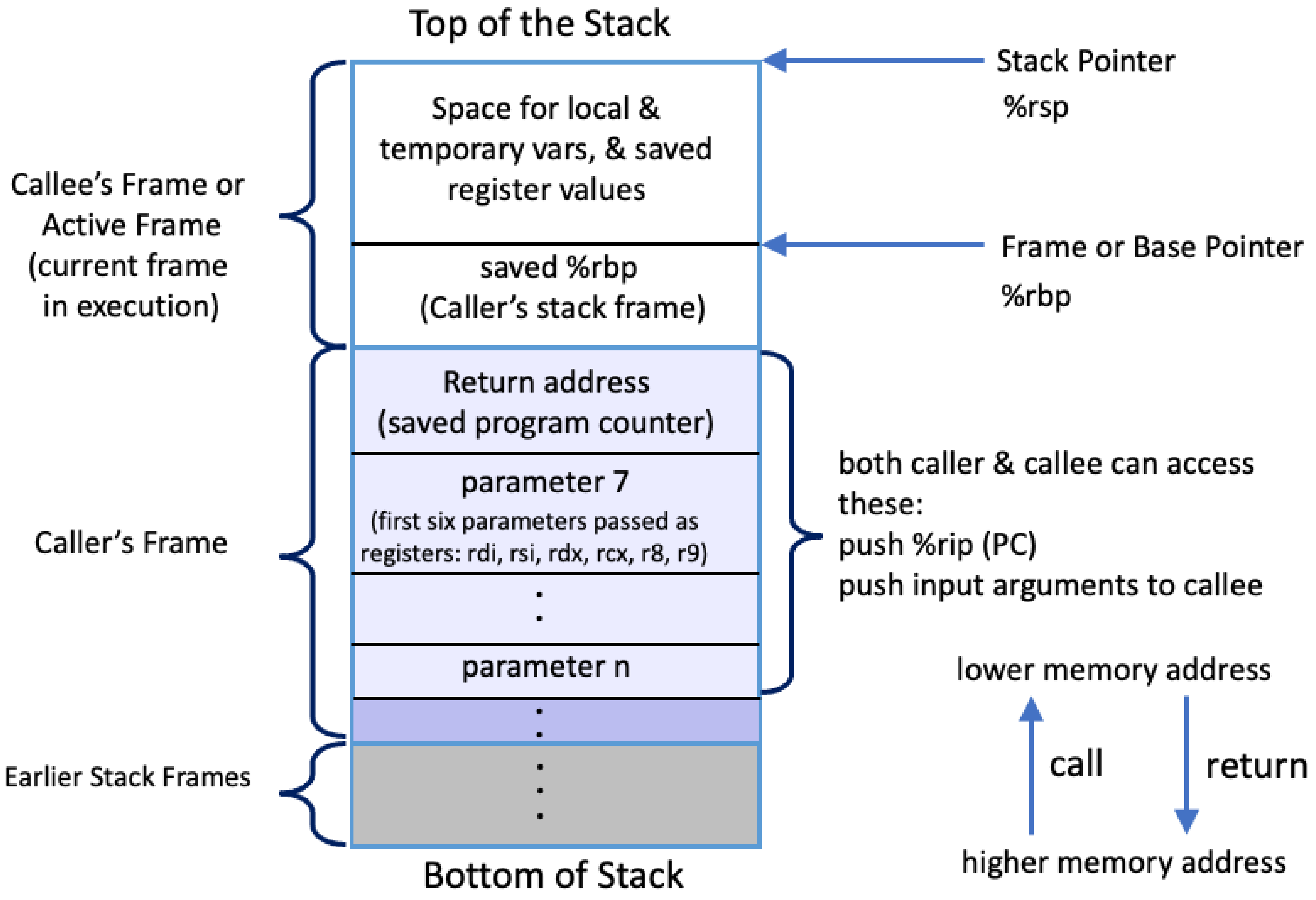
- Extra arguments to the callee are stored just underneath the return address.

- Does it matter what order we store the arguments in?

- Not really, as long as we're consistent (follow conventions).

This is why arguments can be found at positive offsets relative to %rbp.



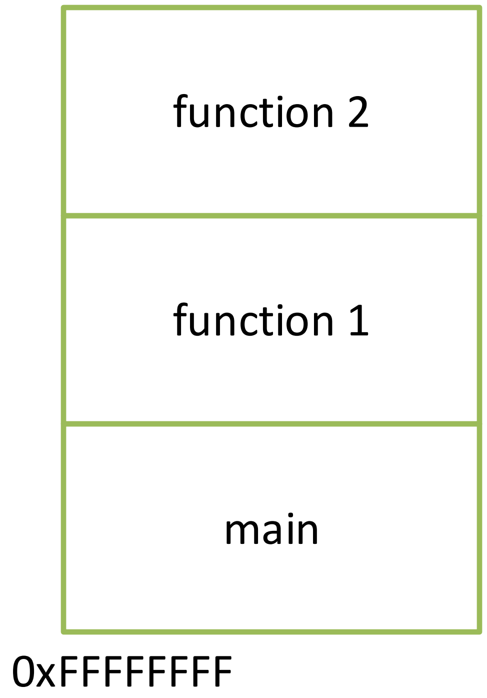
Top of the Stack
Stack Pointer
%rsp
Space for local & temporary vars, & saved register values
Callee's Frame or Active Frame (current frame in execution)
saved %rbp (Caller's stack frame)
Frame or Base Pointer
%rbp
Return address (saved program counter)
parameter 7
(first six parameters passed as registers: rdi, rsi, rdx, rcx, r8, r9)
parameter n
Caller's Frame
both caller & callee can access these:
push %rip (PC)
push input arguments to callee
lower memory address
call
return
higher memory address
Earlier Stack Frames
Bottom of Stack

Stack Frame Contents

- What needs to be stored in a stack frame?
  - Alternatively: What *must* a function know?

- Local variables
- Previous stack frame base address
- Function arguments
- Return value
- Return address

- Saved registers
- Spilled temporaries



Saving Registers

- Registers are a relatively scarce resource, but they’re fast to access. Memory is plentiful, but slower to access.

- Should the caller save its registers to free them up for the callee to use?
- Should the callee save the registers in case the caller was using them?
- Who needs more registers for temporary calculations, the caller or callee?

- Clearly the answers depend on what the functions do…

Splitting the difference…

- We can't know the answers to those questions in advance…
- Divide registers into two groups:

Caller-saved: %rax, %rdi, %rsi, %rdx, %rcx, %r8, %r9, %r10, %r11

Caller must save them prior to calling callee
callee free to trash these,
Caller will restore if needed

Callee-saved: %rbx, %r12, %r13, %r14, %r15

Callee must save them first, and restore them before returning
Caller can assume these will be preserved

Running Out of Registers

- Some computations require more than 16 general-purpose registers to store temporary values.

- *Register spilling*: The compiler will move some temporary values to memory, if necessary.
  - Values pushed onto stack, popped off later
  - No explicit variable declared by user
  - This is getting to the limits of CS 31!
    - – take CS 75 (compilers) for more details.

Today on CS31

How 1D arrays are stored in memory & accessed:

- In C and Assembly
- Static vs. Dynamic

How complex structures are stored in memory & accessed:

- 2D arrays
  - Static vs. Dynamic
  - One contiguous block of memory vs. array of arrays
- Structs

So far: Primitive Data Types

- We’ve been using ints, floats, chars, pointers

- Simple to place these in memory:
  - They have an unambiguous size
  - They fit inside a register*
  - The hardware can operate on them directly

(*There are special registers for floats and doubles that use the IEEE floating point format.)

Composite Data Types

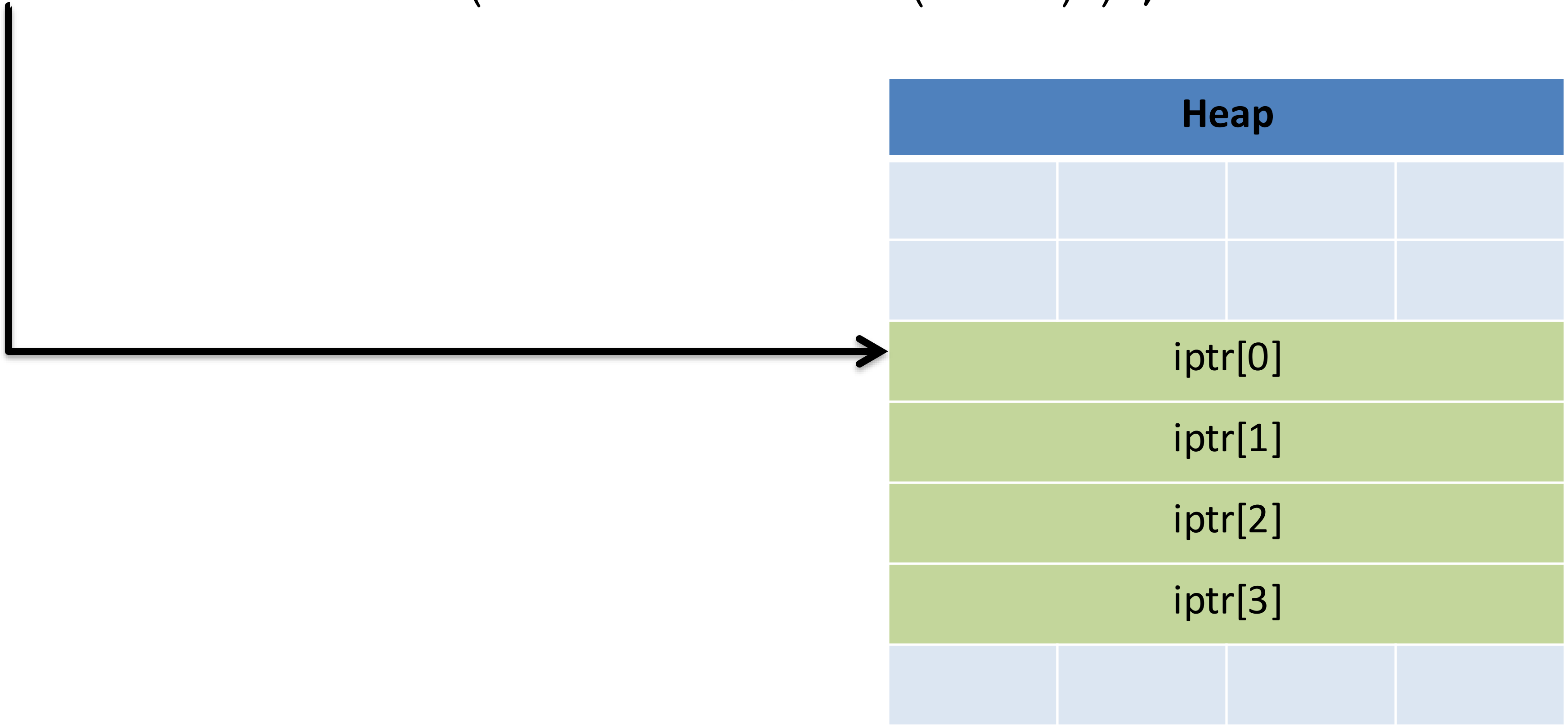
- Combination of one or more existing types into a new type. (e.g., an array of *multiple* ints, or a struct)

- Example: a queue
  - Might need a value (int) plus a link to the next item (pointer)

```
struct queue_node{
  int value;
  struct queue_node *next;
}
```

Recall: Arrays in Memory

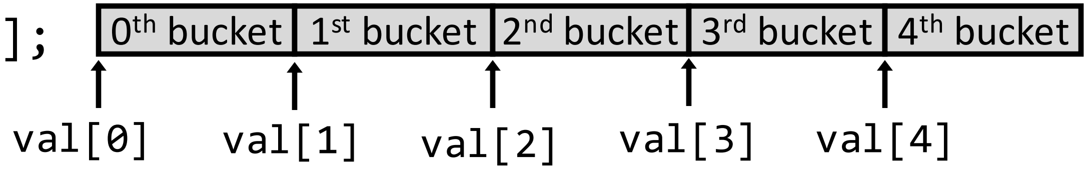
```
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
```



Base + Offset

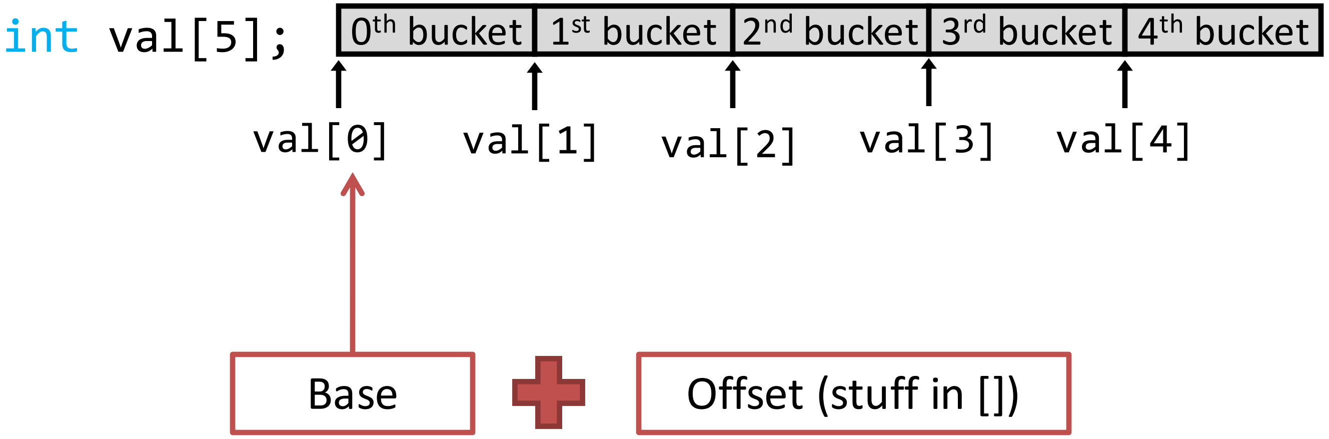
- We know that arrays act as a pointer to the first element. For bucket [N], we just skip forward N.

```
int val[5];
```



Base + Offset

- We know that arrays act as a pointer to the first element. For bucket [N], we just skip forward N.



```
int val[5];
```

This is why we start counting from zero!
Skipping forward with an offset of zero ([0]) gives us the first bucket…

Which expression would compute the address of iptr[3]?

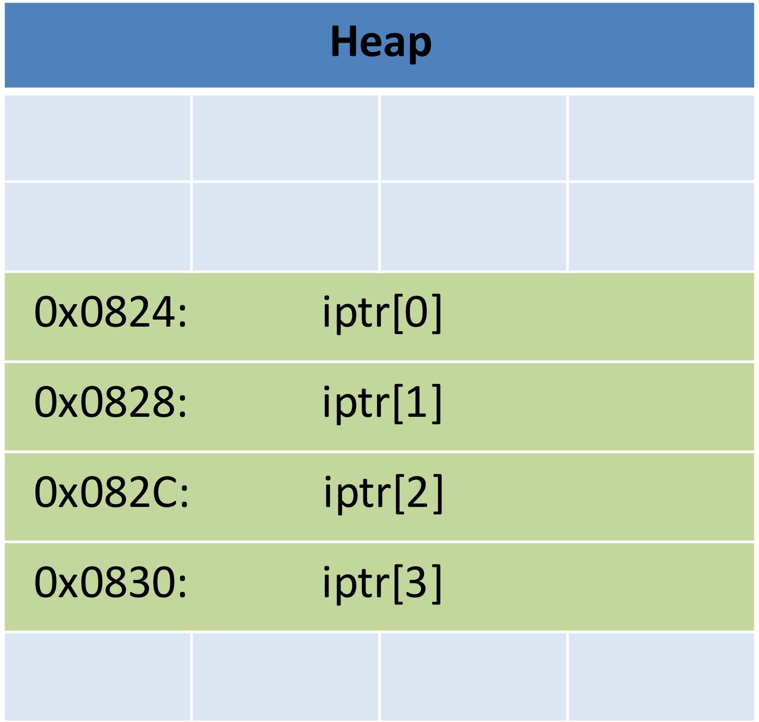
A. 0x0824 + 3 * 4

B. 0x0824 + 4 * 4

C. 0x0824 + 0xC

D. More than one (which?)

E. None of these



Heap	
0x0824:	iptr[0]
0x0828:	iptr[1]
0x082C:	iptr[2]
0x0830:	iptr[3]

Which expression would compute the address of iptr[3]?

What if this isn’t known at compile time?

A. 0x0824 + 3 * 4

B. 0x0824 + 4 * 4

C. 0x0824 + 0xC

D. More than one (which?)

E. None of these

Heap	
0x0824:	iptr[0]
0x0828:	iptr[1]
0x082C:	iptr[2]
0x0830:	iptr[3]

Recall Addressing Mode: Memory

- Accessing memory requires you to specify which address you want.
  - Put the address in a register.
  - Access the register with () around the register’s name.

```
mov (%rcx), %rax
```

  - Use the address in register %rcx to access memory, store result in register %rax

Recall Addressing Mode: Displacement

- Like memory mode, but with a constant offset
  - Offset is often negative, relative to %rbp

```
mov -24(%rbp), %rax
```

  - Take the address in %rbp, subtract 24 from it, index into memory and store the result in %rax.

Addressing Mode: Indexed

- Instead of only using one register to store the base address of a memory address, we can use a base address register **and** an offset register value.

```
mov (%rax, %rcx), %rdx
```

  – Take the base address in %rax, add the value in %rcx to produce a final address, index into memory and store the result in %rdx.

Addressing Mode: Indexed

Instead of only using one register to store the base address of a memory address, we can use a base address register **and** an offset register value.

One register to keep track of base address.

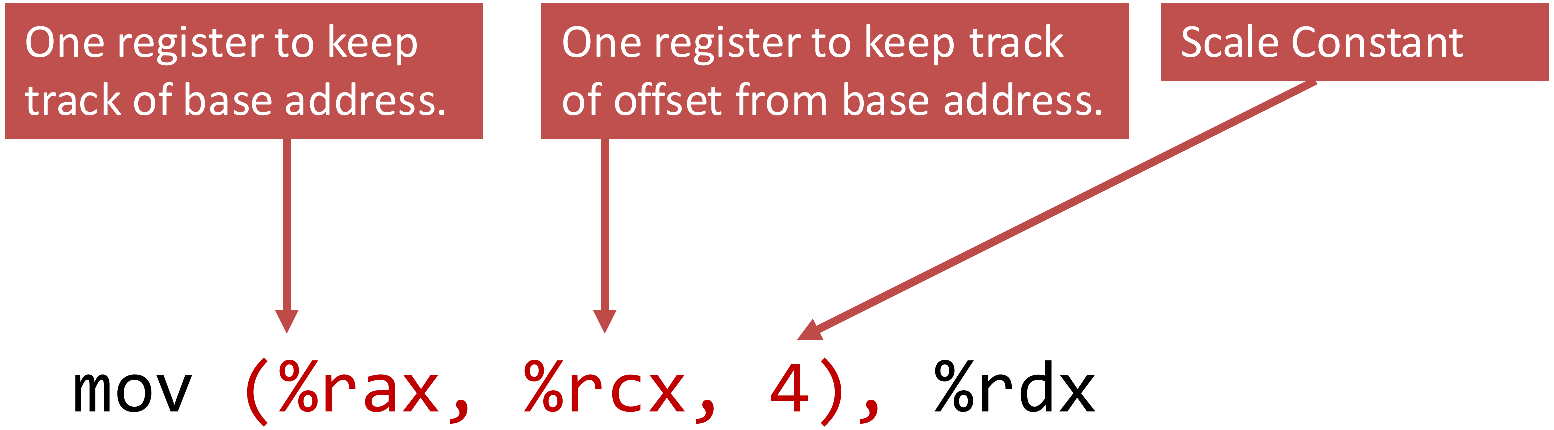
One register to keep track of offset from base address.

```
mov (%rax, %rcx), %rdx
```

- Take the base address: %rax,
- add the value in %rcx: %rax + %rcx
- index into memory and store the result in %rdx.

Addressing Mode: Indexed

The offset (%rcx) can also be scaled by a constant.



One register to keep track of base address.

One register to keep track of offset from base address.

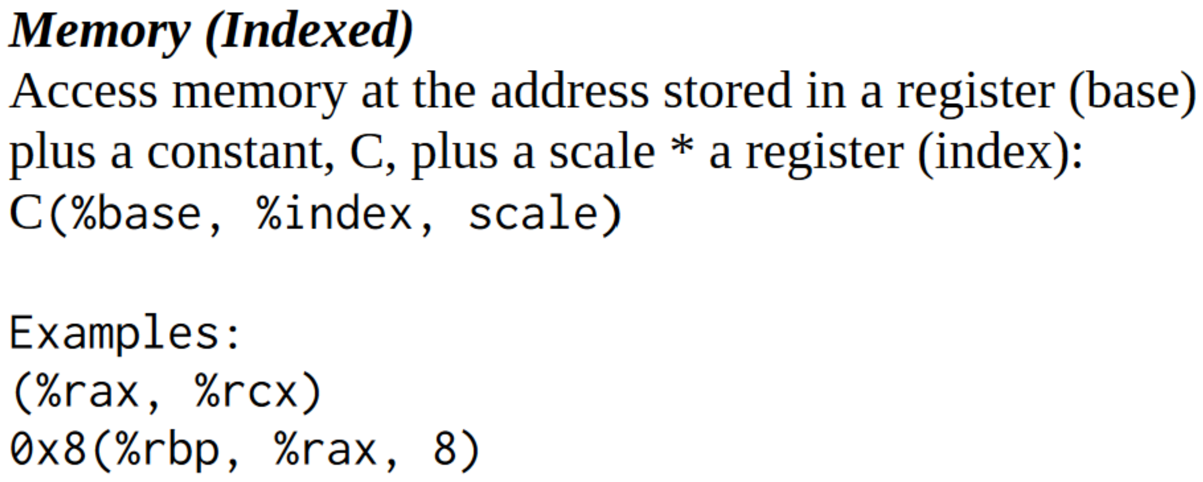
Scale Constant

```
mov (%rax, %rcx, 4), %rdx
```

- – Take the base address: %rax
- – Multiply the offset by the scale: %rcx * 4
- – Add the scaled offset to the base: %rax + %rcx * 4
- – Now, index into memory at (%rax + %rcx * 4) and store the result in %rdx.

Assembly Reference

This mode has been on your assembly reference sheet all along!



Memory (Indexed)

Access memory at the address stored in a register (base) plus a constant, C, plus a scale * a register (index):

```
C(%base, %index, scale)

Examples:
(%rax, %rcx)
0x8(%rbp, %rax, 8)
```

Let’s try an example

Suppose:

```
int *iptr = malloc(4*sizeof(int));
```

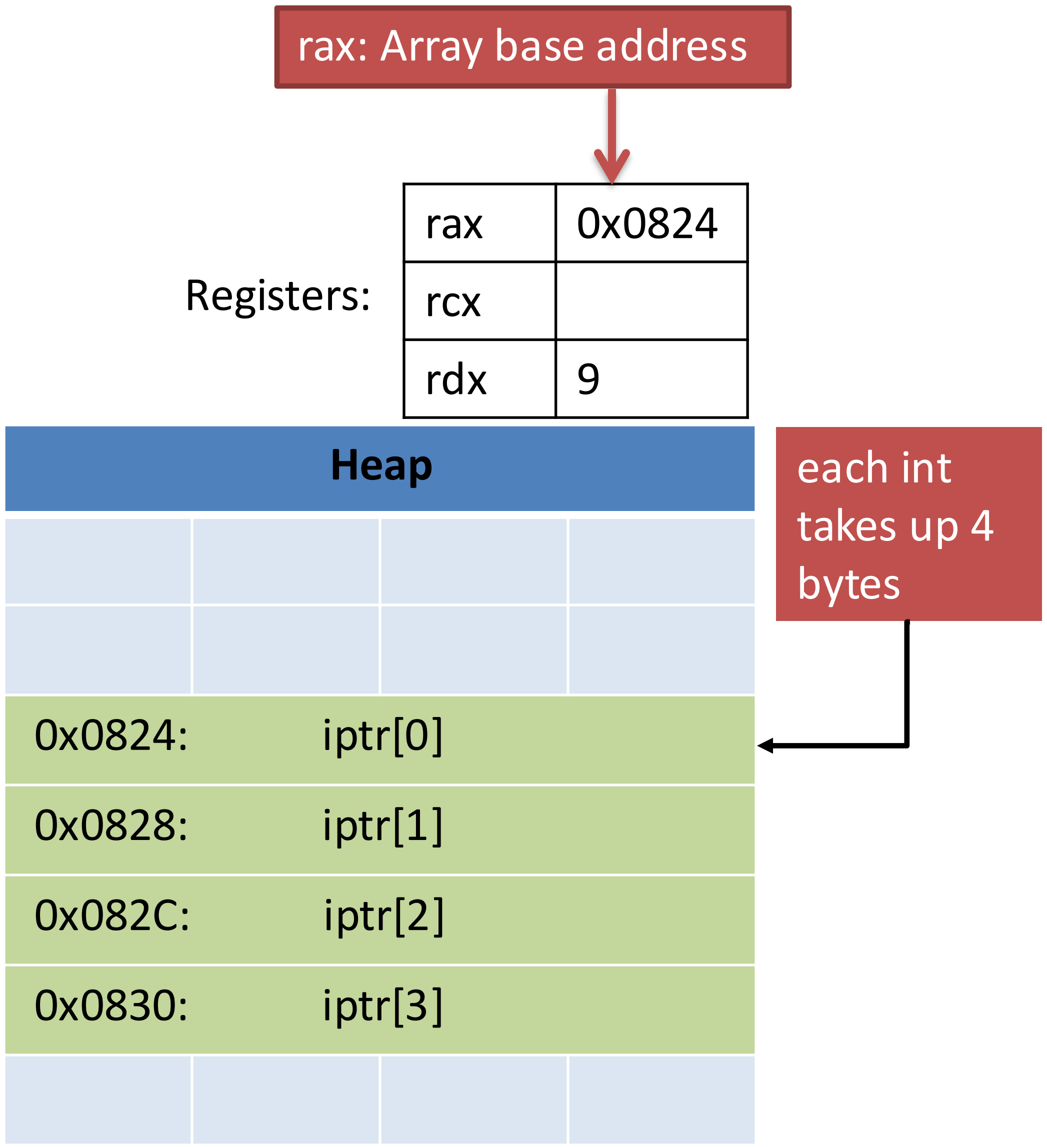
//iptr is stored in register %rax.

int i=2; is stored at %rbp-8

C code says:

```
iptr[i] = 9;
```

Using what we just learnt, what does the C code above translate to, in assembly?



rax: Array base address

Registers:

rax	0x0824
rcx	
rdx	9

Heap

0x0824:	iptr[0]
0x0828:	iptr[1]
0x082C:	iptr[2]
0x0830:	iptr[3]

each int takes up 4 bytes

Let's try an example

Suppose:

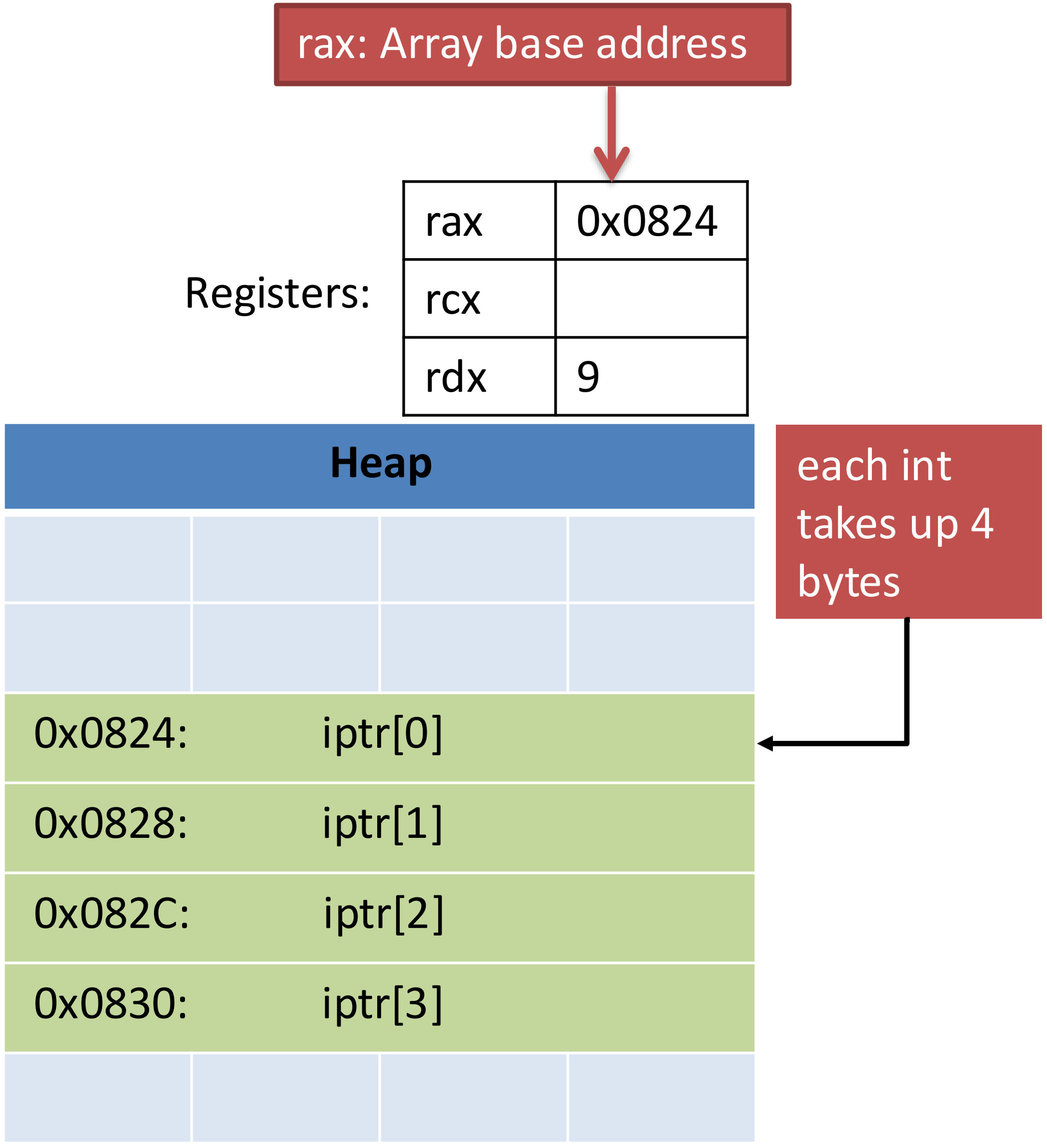
```
int iptr = malloc(4*sizeof(int));
//iptr is stored in register %rax.
int i=2; is stored at %rbp-8
```

C code says:

```
iptr[i] = 9;
```

Using what we just learnt, what does the C code above translate to, in assembly?

```
mov -8(%rbp), %rcx
```



rax: Array base address

Registers:

rax	0x0824
rcx	
rdx	9

each int takes up 4 bytes

Heap	
0x0824:	iptr[0]
0x0828:	iptr[1]
0x082C:	iptr[2]
0x0830:	iptr[3]

Let’s try an example

Suppose:

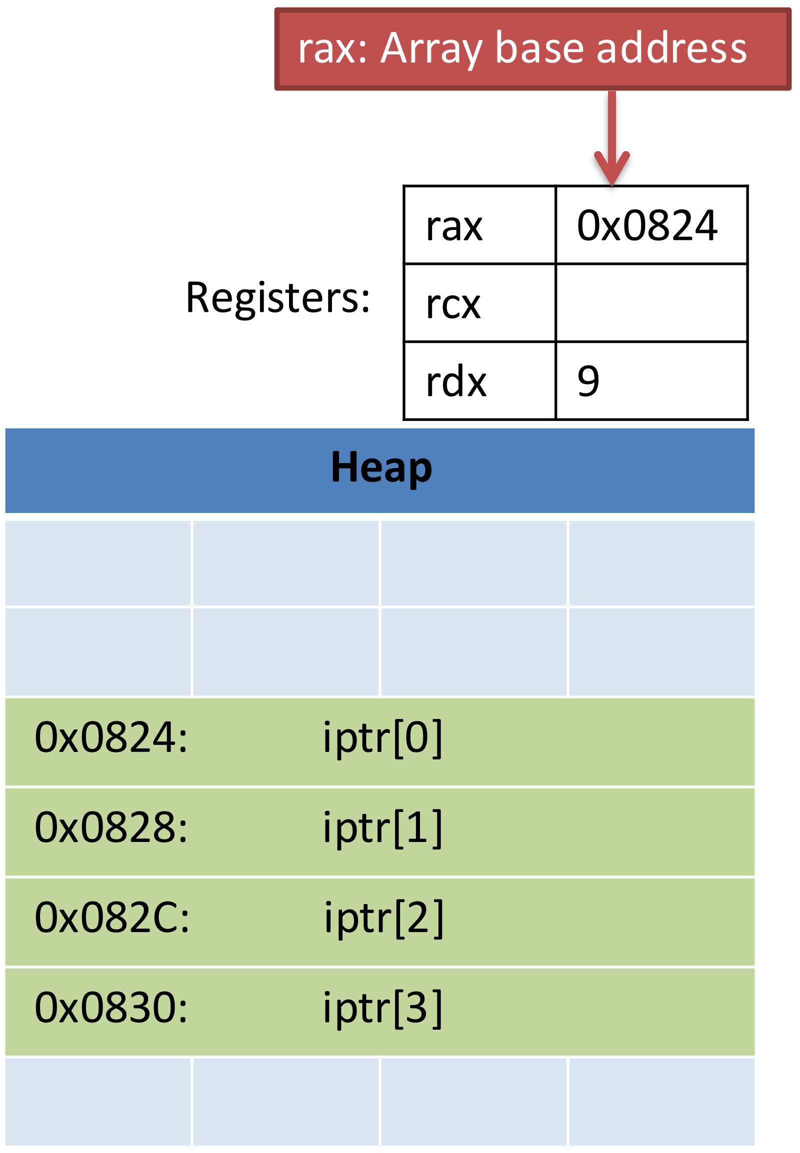
```
    int iptr = malloc(4*sizeof(int));
    //iptr  is stored in register %rax.
    int  i=2; is stored at %rbp-8
```

C code says:

```
iptr[i] = 9;
```

Using what we just learnt, what does the C code above translate to, in assembly?

```
mov -8(%rbp), %rcx
mov %rdx, (rax, rcx, 4)
```



rax: Array base address

Registers:

rax	0x0824
rcx	
rdx	9

Heap

0x0824:	iptr[0]
0x0828:	iptr[1]
0x082C:	iptr[2]
0x0830:	iptr[3]

Let’s try an example

Suppose:

`int iptr;` is stored in register `%rax.`

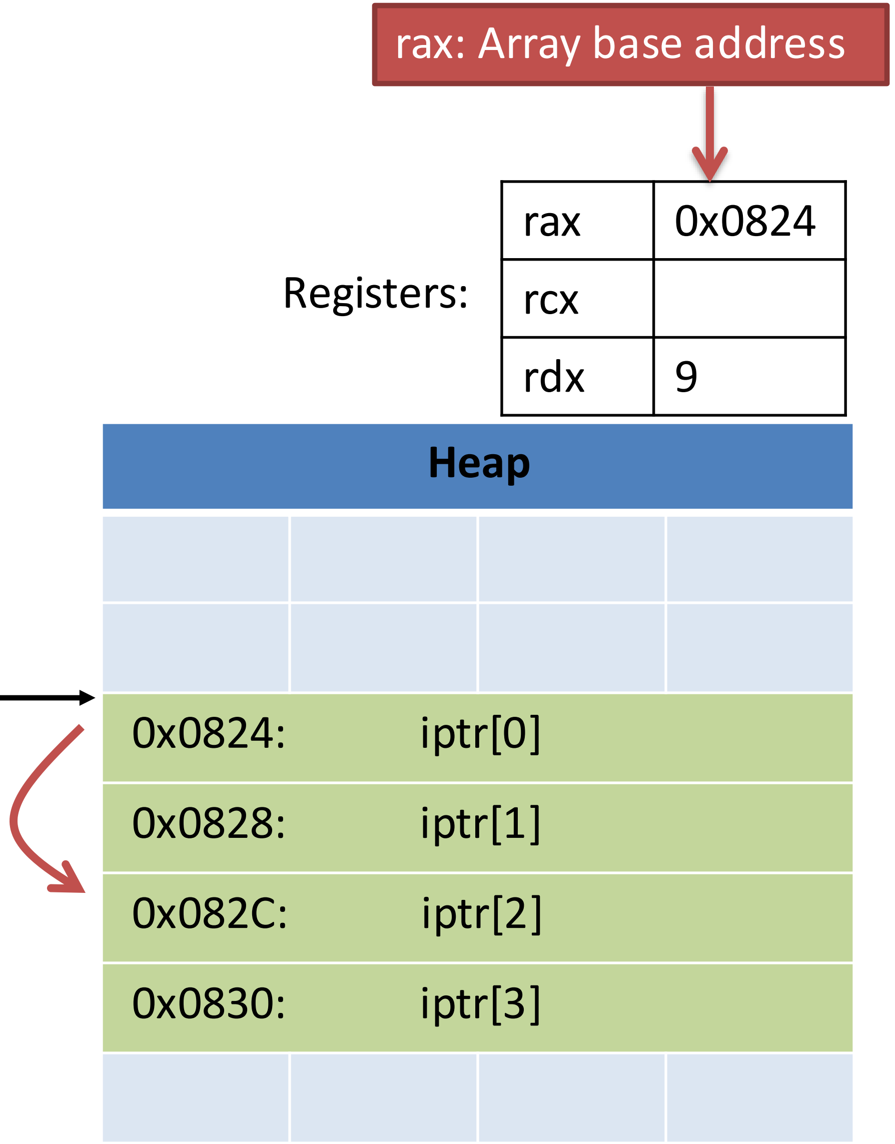
`int i=2;` is stored at %rbp-8

```
iptr[i] = 9; //iptr[2] = 9;
```

In assembly:

```
mov -8(%rbp), %rcx

mov %rdx, (rax, rcx, 4)
```

= add (rcx *4)
= add (2*4)
= add 8



rax: Array base address

Registers:

rax	0x0824
rcx	
rdx	9

Heap

0x0824:	iptr[0]
0x0828:	iptr[1]
0x082C:	iptr[2]
0x0830:	iptr[3]

Let’s try an example

Suppose:

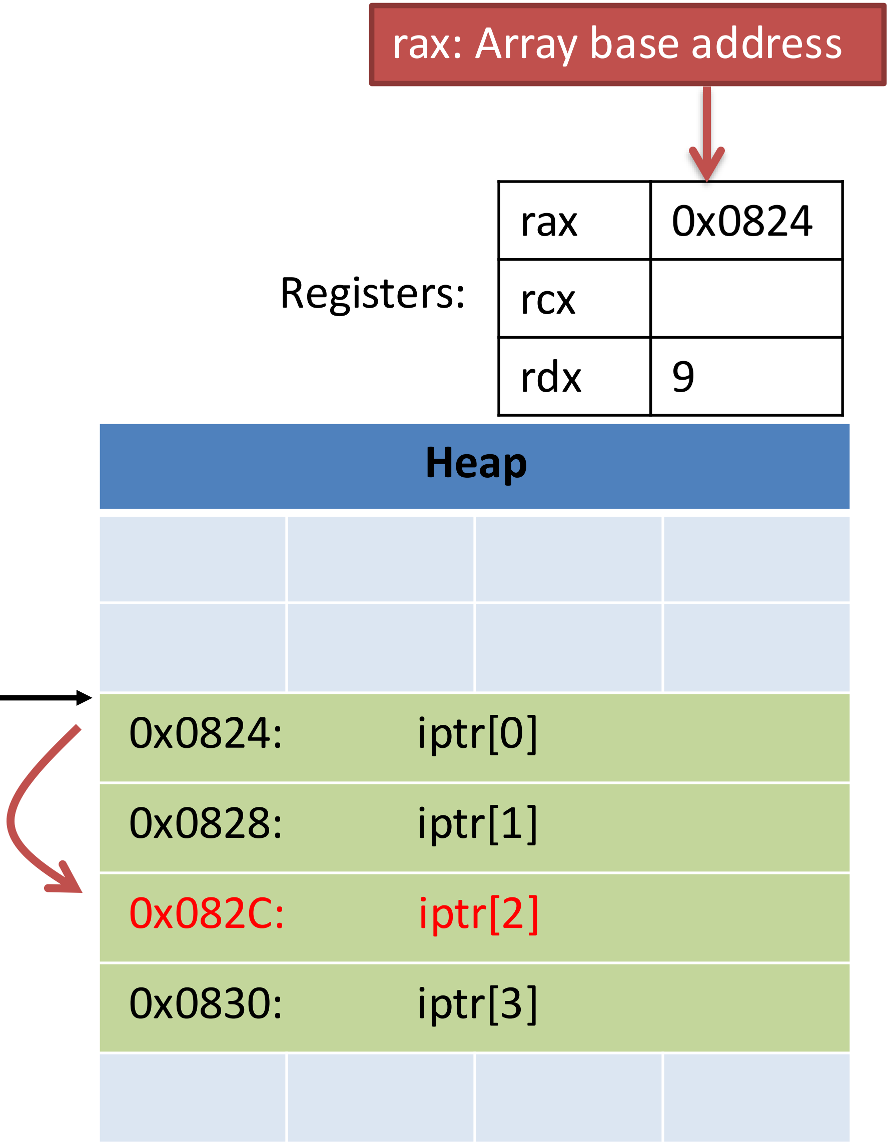
```
int iptr;  is stored in register %rax.
int i=2; is stored at %rbp-8
iptr[i] = 9; //iptr[2] = 9;
```

<u>In assembly:</u>

```
mov -8(%rbp), %rcx

mov %rdx, (rax, rcx, 4)
```

= add (rcx *4)
= add (2*4)
= add 8

rax: Array base address



Registers:

rax	0x0824
rcx	
rdx	9

Heap	
0x0824:	iptr[0]
0x0828:	iptr[1]
0x082C:	iptr[2]
0x0830:	iptr[3]

What happens when we increment i?
What changes do we make in assembly?

Suppose:

`int iptr;` is stored in register `%rax.`

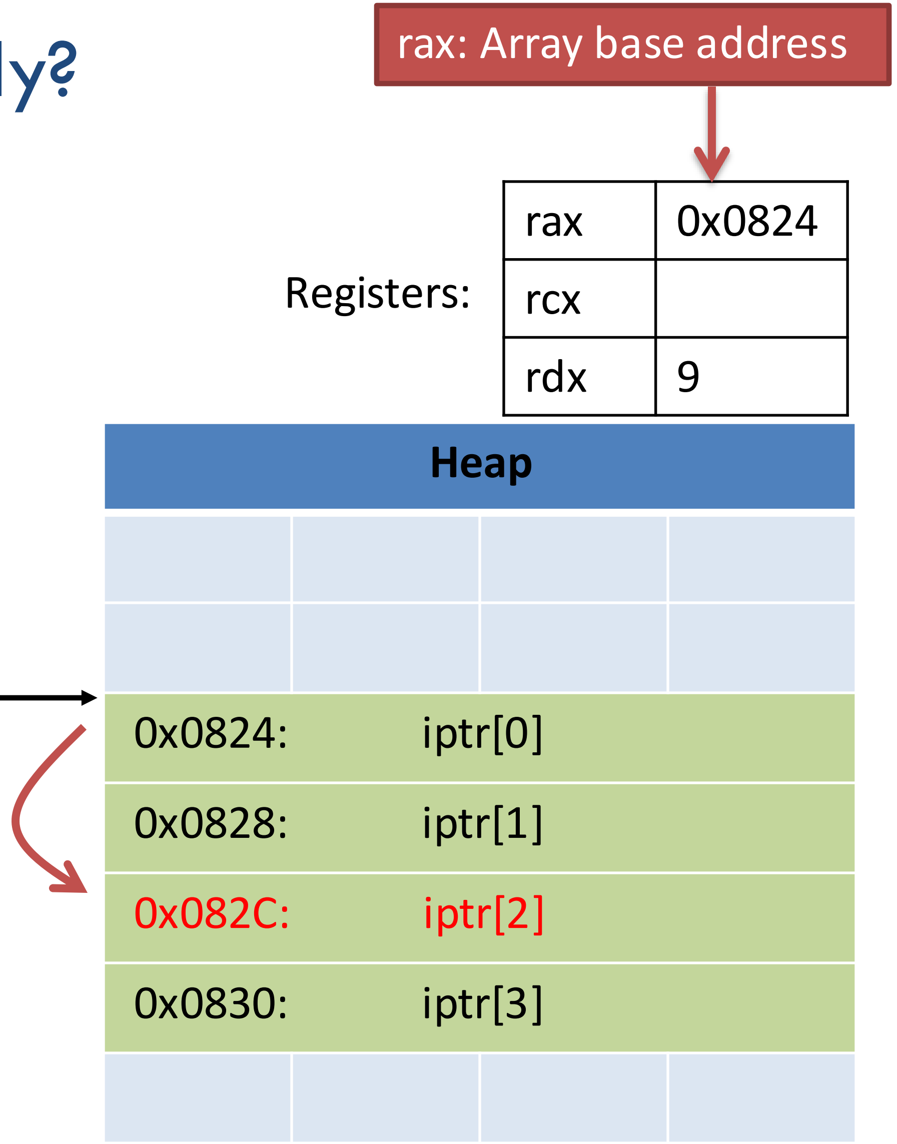
`int` i=3; is stored at %rbp-8

```
iptr[i] = 10; //iptr[3] = 10;
```

In assembly:

```
mov -8(%rbp), %rcx

mov %rdx, (rax, rcx, 4)
```

= add (rcx *4)
= add (2*4)
= add 8



rax: Array base address

Registers:

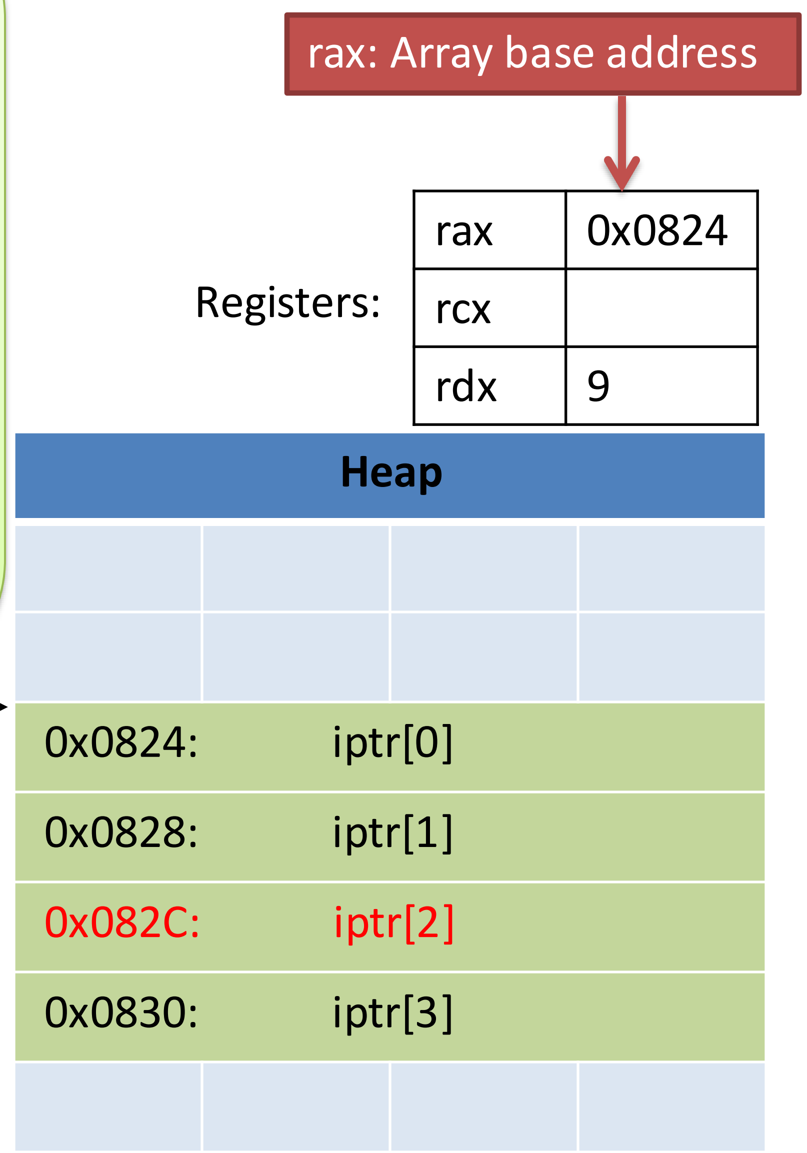
rax	0x0824
rcx	
rdx	9

From here, if the program increments i (e.g., in a loop) and accesses the array at the new (incremented) position of i:

Compiler can simply increment register rcx and access the next element of the array with the same `mov` command!

```
mov -8(%rbp), %rcx

mov %rdx, (rax, rcx, 4)
```

= add (rcx *4)
= add (2*4)
= add 8



rax: Array base address

Registers:

rax	0x0824
rcx	
rdx	9

Heap

0x0824:	iptr[0]
0x0828:	iptr[1]
0x082C:	iptr[2]
0x0830:	iptr[3]

So Far: One Dimensional Arrays

- We are not restricted to an array of ints..
  How about an array *of* arrays of ints?

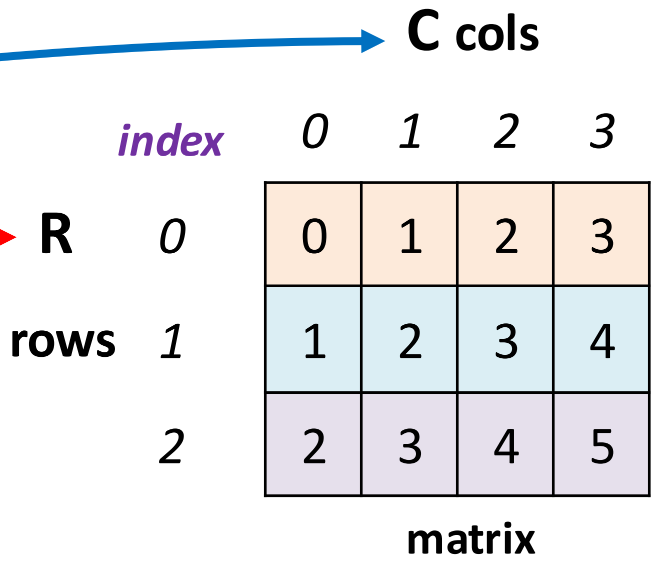
"Give me three sets of four integers"

```
int twodims[3][4];
```

- How should these be organized in memory?

Declaring Static 2D Arrays

```
#define R 3
#define C 4

int  matrix[R][C] , i, j;

for(i=0; i<R; i++) {
  for(j=0; j<C; j++) {
     matrix[i][j] = i+j;
   }
}
```



index	0	1	2	3
0	0	1	2	3
1	1	2	3	4
2	2	3	4	5

- Declare with row and column dimension
- Can use matrix[i][j] to index

Memory Layout of Static 2D Arrays

C cols

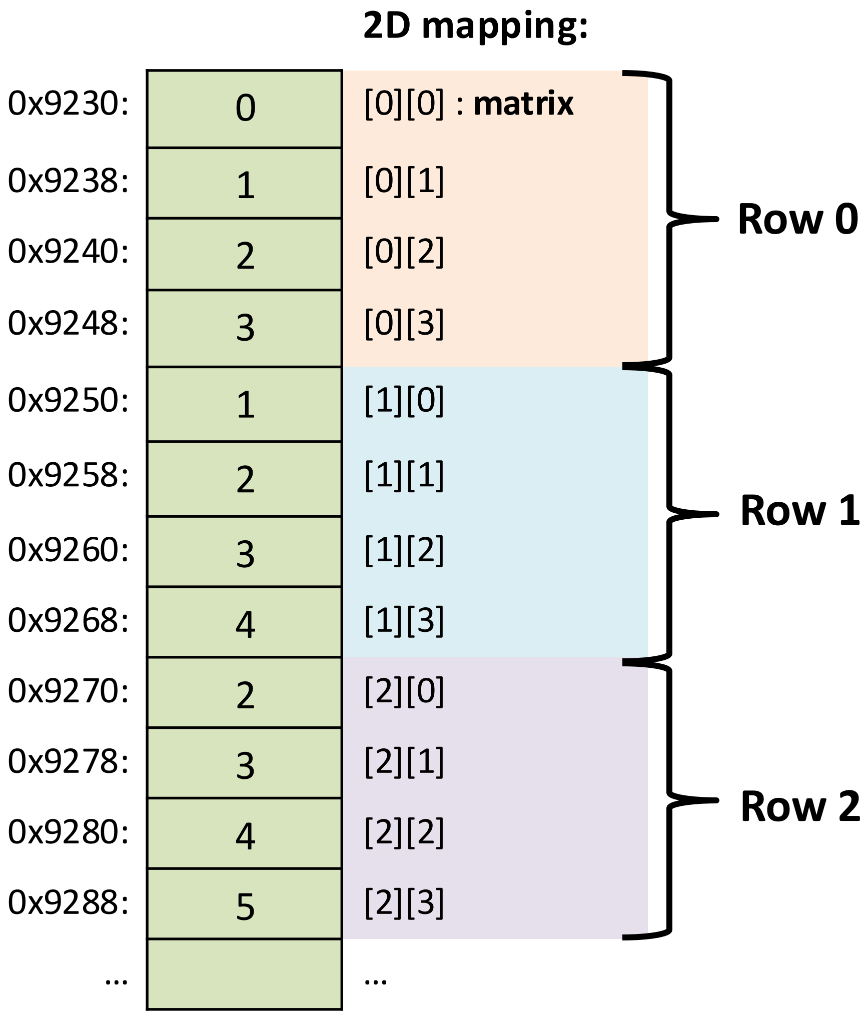
R rows

index	*0*	*1*	*2*	*3*
0	0	1	2	3
1	1	2	3	4
2	2	3	4	5

matrix

***Row Major* Order in C:**

all Row 0 buckets, followed by
all Row 1 buckets, followed by
all Row 2 buckets, …



2D mapping:

0x9230:	0	[0][0] : **matrix**
0x9238:	1	[0][1]
0x9240:	2	[0][2]
0x9248:	3	[0][3]
0x9250:	1	[1][0]
0x9258:	2	[1][1]
0x9260:	3	[1][2]
0x9268:	4	[1][3]
0x9270:	2	[2][0]
0x9278:	3	[2][1]
0x9280:	4	[2][2]
0x9288:	5	[2][3]
…		…

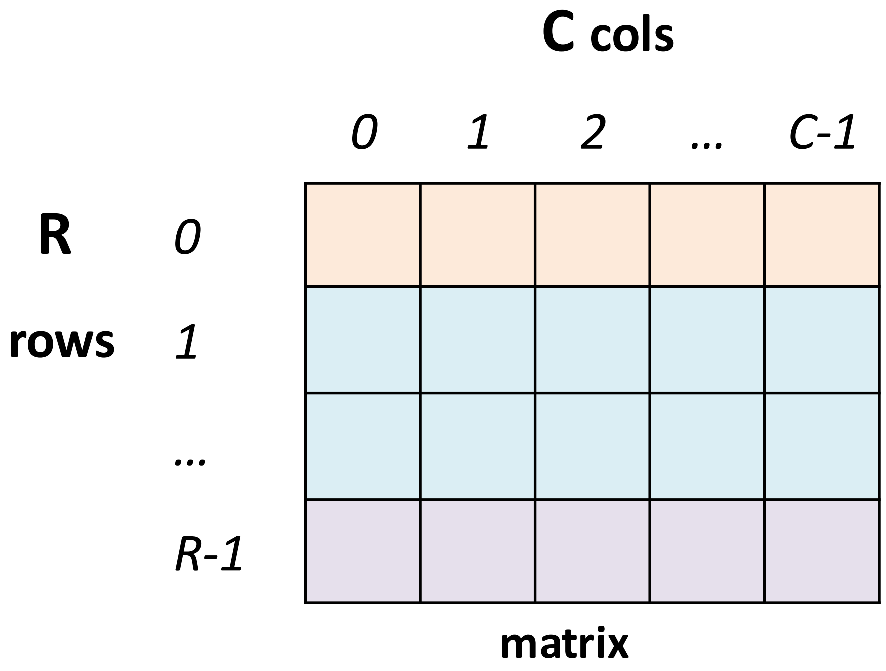
Using Static 2D Arrays as Parameters

- 2D array parameter must specify **column dimension**
  - **Why?** Compiler needs the column dimension to calculate offset from base address in memory of bucket [i][j]
- Row dimension passed as 2nd parameter to make function *more generic*
  - function can be passed any 2D array with same column dimension

```c
void foo(int matrix[][C], int rows){

   int i, j;

   for(i=0; i < rows; i++) {
    for(j=0; j< C; j++) {
       matrix[i][j] = i*j;
    }
   }
}
```

```c
#define R  3
#define C  4

int main() {

  int arr[R][C];
  int grid[100][C];

  foo(arr,  R);
  foo(grid, 100);
```

Calculating Offset for Static 2D Arrays



C cols

0 1 2 … C-1

R *0*

rows *1*

…

R-1

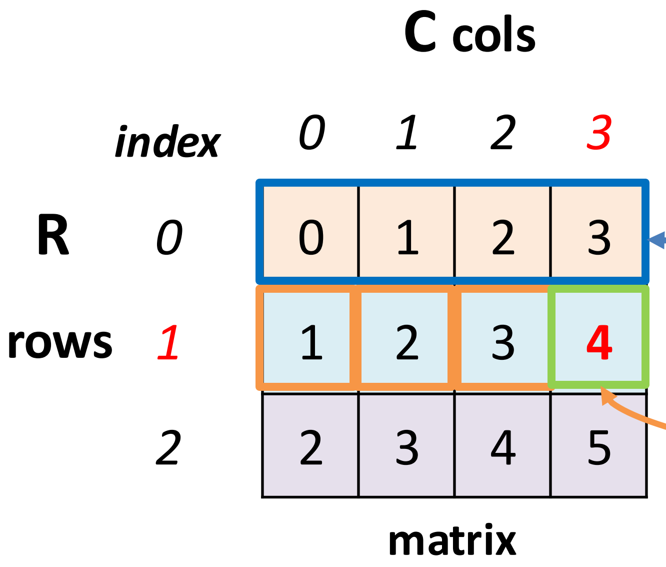
matrix

Offset of `matrix[row][col]` from base?

= row * MAX_COL + col

TIP: MAX_COL = how big each row is = max number of columns!

Calculating Offset for Static 2D Arrays



C cols

	index	*0*	*1*	*2*	*3*
R	*0*	0	1	2	3
rows	*1*	1	2	3	**4**
	2	2	3	4	5

matrix

Offset of `matrix[row][col]` from base?

= row * MAX_COL + col

E.g., location of `matrix[1][3]`?

= base + (1 * MAX_COL + 3) buckets // *skip 1 full row and 3 buckets*

= base + (1 * 4 + 3) buckets

= base + 7 buckets // *skip 7 buckets*

Calculating Offset for Static 2D Arrays

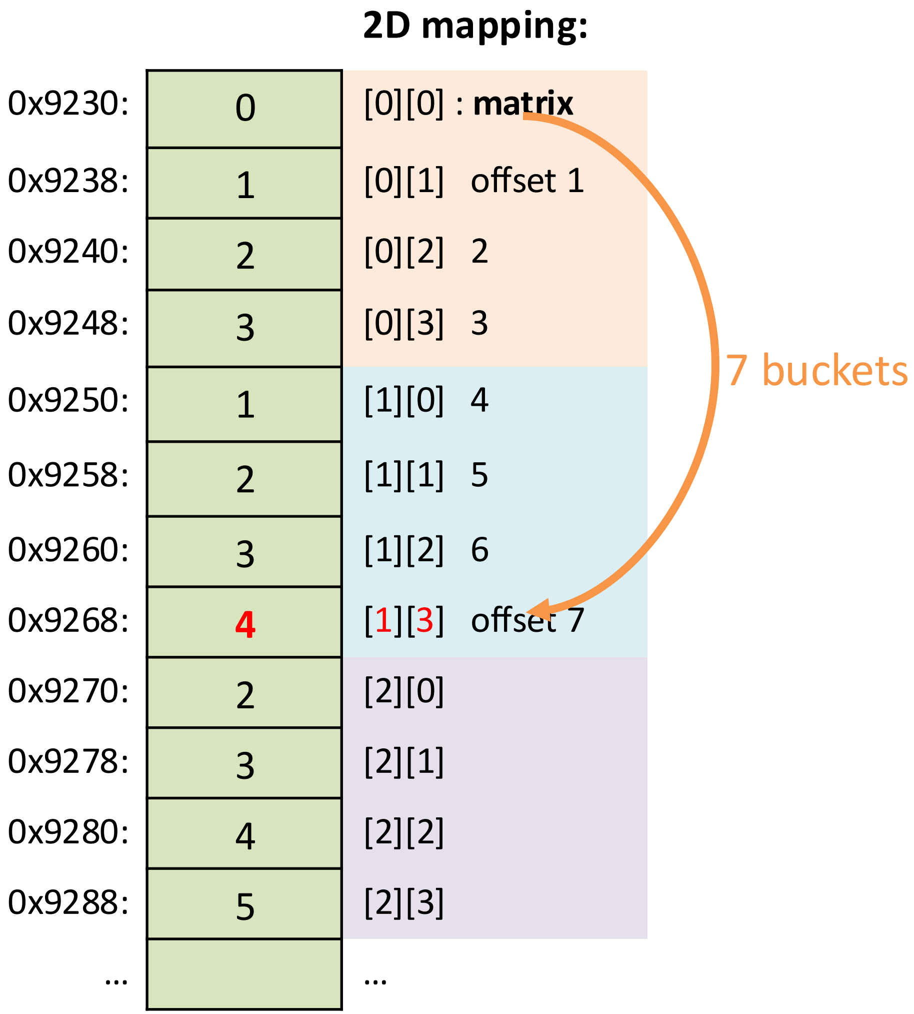
C cols

	index	*0*	*1*	*2*	*3*
R	*0*	0	1	2	3
rows	*1*	1	2	3	**4**
	2	2	3	4	5

matrix

Offset of `matrix[row][col]` from base?
= row * MAX_COL + col

E.g., location of `matrix[1][3]`?
= base + (1 * MAX_COL + 3) buckets
= base + (1 * 4 + 3) buckets
= base + 7 buckets



2D mapping:

0x9230:	0	[0][0] : **matrix**
0x9238:	1	[0][1] offset 1
0x9240:	2	[0][2] 2
0x9248:	3	[0][3] 3
0x9250:	1	[1][0] 4
0x9258:	2	[1][1] 5
0x9260:	3	[1][2] 6
0x9268:	**4**	[1][3] offset 7
0x9270:	2	[2][0]
0x9278:	3	[2][1]
0x9280:	4	[2][2]
0x9288:	5	[2][3]
…		…

Calculating Address for Static 2D Arrays

C cols

	index	*0*	*1*	*2*	*3*
R	*0*	0	1	2	3
rows	*1*	1	2	3	**4**
	2	2	3	4	5

SIZE

		2D mapping:
0x9230:	0	[0][0] : **matrix**
0x9238:	1	[0][1] offset 1
0x9240:	2	[0][2] 2
0x9248:	3	[0][3] 3
0x9250:	1	[1][0] 4
0x9258:	2	[1][1] 5
0x9260:	3	[1][2] 6
0x9268:	**4**	[1][3] offset 7
0x9270:	2	[2][0]
0x9278:	3	[2][1]
0x9280:	4	[2][2]
0x9288:	5	[2][3]
…		…

0x38 bytes

Address of `matrix[`**`row`**`][`**`col`**`]` from base?

= base address + row * MAX_COL*SIZE + col*SIZE

E.g., address of `matrix[`**`1`**`][`**`3`**`]`? Assume SIZE of bucket is 8 bytes

= base addr. + (1 * MAX_COL *SIZE + 3*SIZE) bytes

= base addr. + (1 * 4 * 8 + 3 * 8) bytes

= base addr. + (32 + 24) bytes

= base addr. + 0x38 → 0x9320 + 0x38 = 0x9268

If we declared `long int matrix[5][3];`, and the base of matrix is `0x3420`, what is the address of `matrix[3][2]`? Assume `sizeof(long int) = 8` bytes.

A. 0x3488

B. 0x3470

C. 0x3478

D. 0x344C

E. None of these

*address = base address + row * MAX_COL *SIZE + col*SIZE*

If we declared `long int matrix[5][3];`, and the base of matrix is `0x3420`, what is the address of `matrix[3][2]`? Assume `sizeof(long int) = 8` bytes.

A. 0x3488

B. 0x3470

C. 0x3478

D. 0x344C

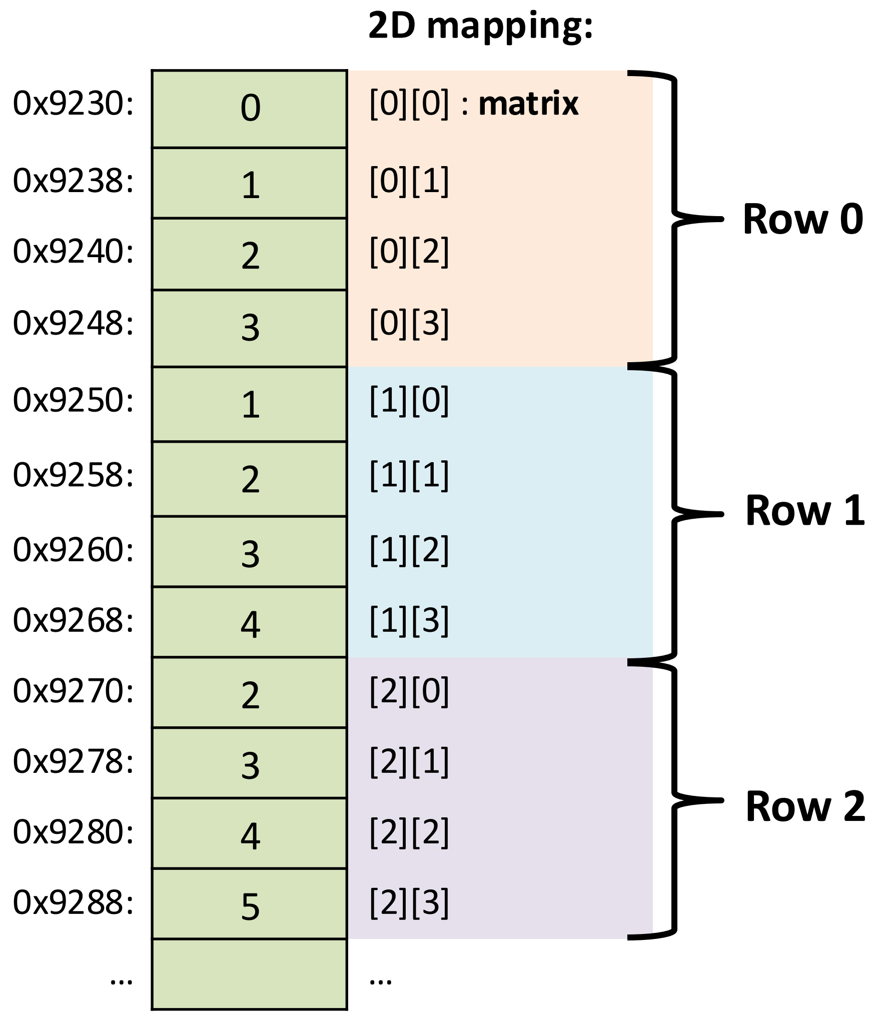
E. None of these

*address = base address + row * MAX_COL *SIZE + col*SIZE*

Dynamically Allocating 2D Arrays: Contiguous Memory

- Given the *row-major order* layout, a "two-dimensional array" is still just a **contiguous block** of memory:

  The malloc function just needs to return… a pointer to a contiguous block of memory! That is, you only need **one call** to malloc.



		2D mapping:	
0x9230:	0	[0][0] : **matrix**	**Row 0**
0x9238:	1	[0][1]	
0x9240:	2	[0][2]	
0x9248:	3	[0][3]	
0x9250:	1	[1][0]	**Row 1**
0x9258:	2	[1][1]	
0x9260:	3	[1][2]	
0x9268:	4	[1][3]	
0x9270:	2	[2][0]	**Row 2**
0x9278:	3	[2][1]	
0x9280:	4	[2][2]	
0x9288:	5	[2][3]	
…		…	

Dynamically Allocating 2D Arrays: Contiguous Memory

For this example, with three rows and four columns:

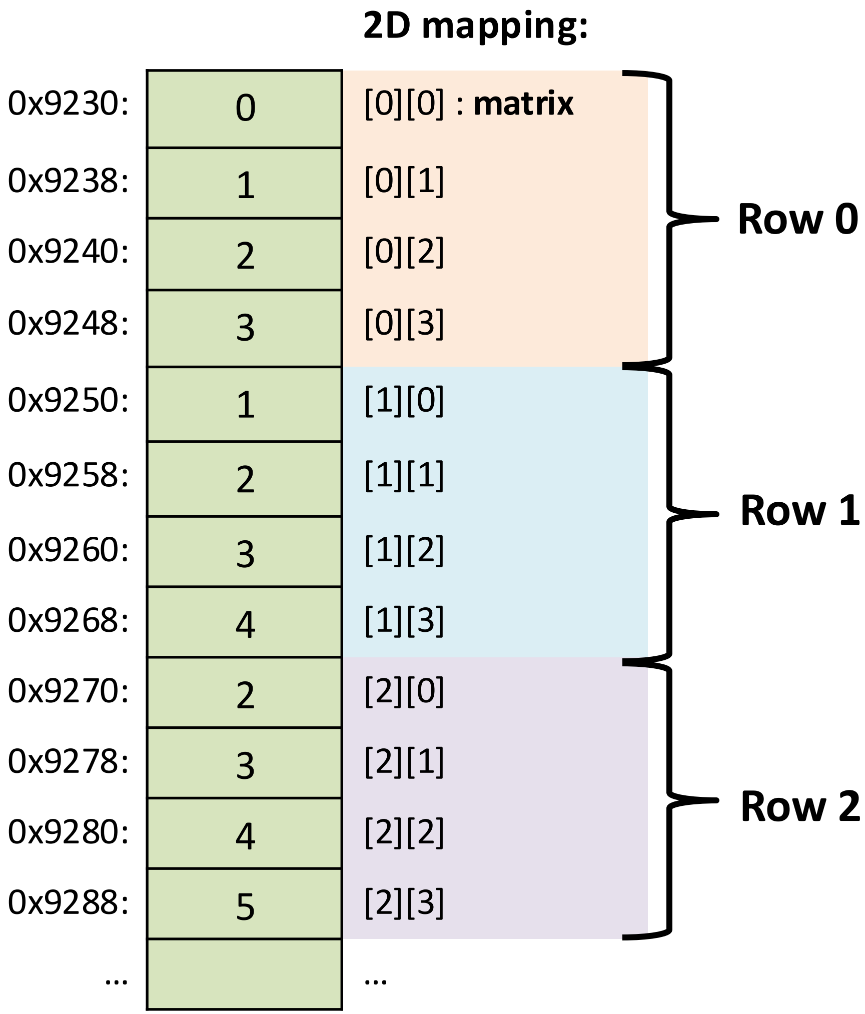
C cols

R rows		*0*	*1*	*2*	*3*
	0	0	1	2	3
	1	1	2	3	4
	2	2	3	4	5

```
long int * matrix = malloc(3 * 4 * sizeof (long int));
```

Caveat: the C compiler doesn't know that you're planning to use this block of memory with more than one index (i.e., row and column).

Can't access: `matrix[i][j]`!



2D mapping:

0x9230:	0	[0][0] : **matrix**	**Row 0**
0x9238:	1	[0][1]	
0x9240:	2	[0][2]	
0x9248:	3	[0][3]	
0x9250:	1	[1][0]	**Row 1**
0x9258:	2	[1][1]	
0x9260:	3	[1][2]	
0x9268:	4	[1][3]	
0x9270:	2	[2][0]	**Row 2**
0x9278:	3	[2][1]	
0x9280:	4	[2][2]	
0x9288:	5	[2][3]	
…		…	

Dynamically Allocating 2D Arrays: Contiguous Memory

For this example, with three rows and four columns:

C cols

		0	*1*	*2*	*3*
R	*0*	0	1	2	3
rows	*1*	1	2	3	4
	2	2	3	4	5

```
long int * matrix = malloc(3 * 4 * sizeof (long int));
```

To access matrix[i][j], compute the offset manually:

```
index = i * COL_MAX + j;

matrix[index] = …
```

2D mapping:

0x9230:	0	[0][0] : **matrix**	**Row 0**
0x9238:	1	[0][1]	
0x9240:	2	[0][2]	
0x9248:	3	[0][3]	
0x9250:	1	[1][0]	**Row 1**
0x9258:	2	[1][1]	
0x9260:	3	[1][2]	
0x9268:	4	[1][3]	
0x9270:	2	[2][0]	**Row 2**
0x9278:	3	[2][1]	
0x9280:	4	[2][2]	
0x9288:	5	[2][3]	
…		…	

Using Dynamically Allocated 2D Arrays as Parameters

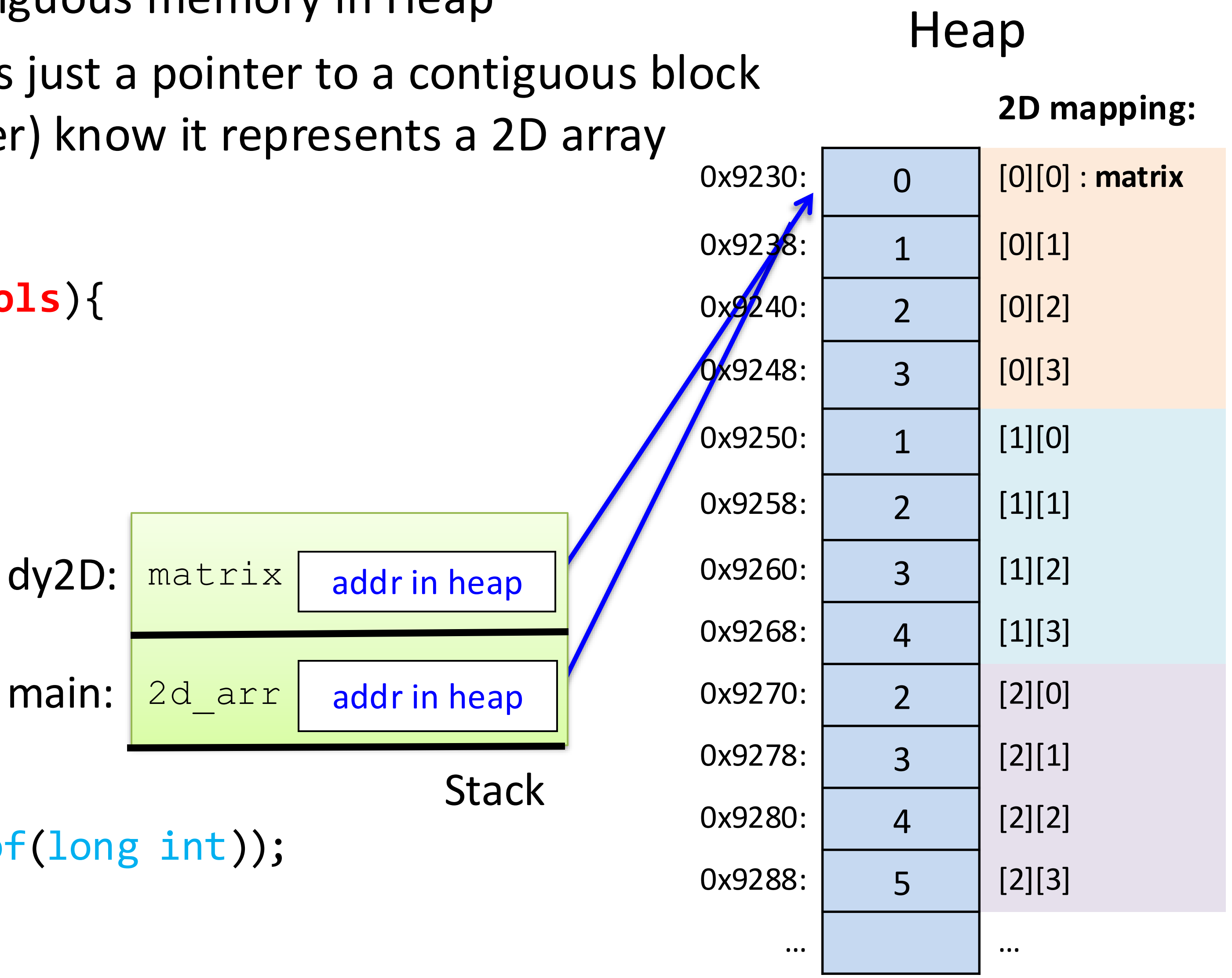
- Parameter gets base address of contiguous memory in Heap
- Just like 1D arrays (almost). **Why?** It’s just a pointer to a contiguous block of memory, only we (the programmer) know it represents a 2D array
- Pass *row* and *column* dimensions

```
void dy2D(int *matrix, int rows, int cols){
   int i, j;
   for(i=0; i < rows; i++) {
        for(j=0; j< cols; j++) {
            matrix[i*cols + j] = i*j;
    }
   }
}
int main() {
   long int *2d_arr = malloc(3 * 4 * sizeof(long int));
   dy2D(2d_arr, 3, 4);
}
```

Using Dynamically Allocated 2D Arrays as Parameters

- Parameter gets base address of contiguous memory in Heap
- Just like 1D arrays (almost). **Why?** It’s just a pointer to a contiguous block of memory, only we (the programmer) know it represents a 2D array
- Pass *row* and *column* dimensions

```
void dy2D(int *matrix, int rows, int cols){
   int i, j;
   for(i=0; i < rows; i++) {
        for(j=0; j< cols; j++) {
             matrix[i*cols + j] = i*j;
        }
    }
}
int main() {
  long int *2d_arr = malloc(3 * 4 * sizeof(long int));
  dy2D(2d_arr, 3, 4);
}
```



		2D mapping:
0x9230:	0	[0][0] : **matrix**
0x9238:	1	[0][1]
0x9240:	2	[0][2]
0x9248:	3	[0][3]
0x9250:	1	[1][0]
0x9258:	2	[1][1]
0x9260:	3	[1][2]
0x9268:	4	[1][3]
0x9270:	2	[2][0]
0x9278:	3	[2][1]
0x9280:	4	[2][2]
0x9288:	5	[2][3]
…		…

But… can't we have pointers to pointers?

- If we want a dynamic **array** of **ints**:
  - declare `int *array = malloc(N * sizeof(int))`
  - Treat this internally as a 2D array `(i*COL + j)`

- If we want an **array** of **int pointers**:
  - declare `int **array = malloc(…)`
  - For *each pointer*, dynamically allocate an array

But… can't we have pointers to pointers?

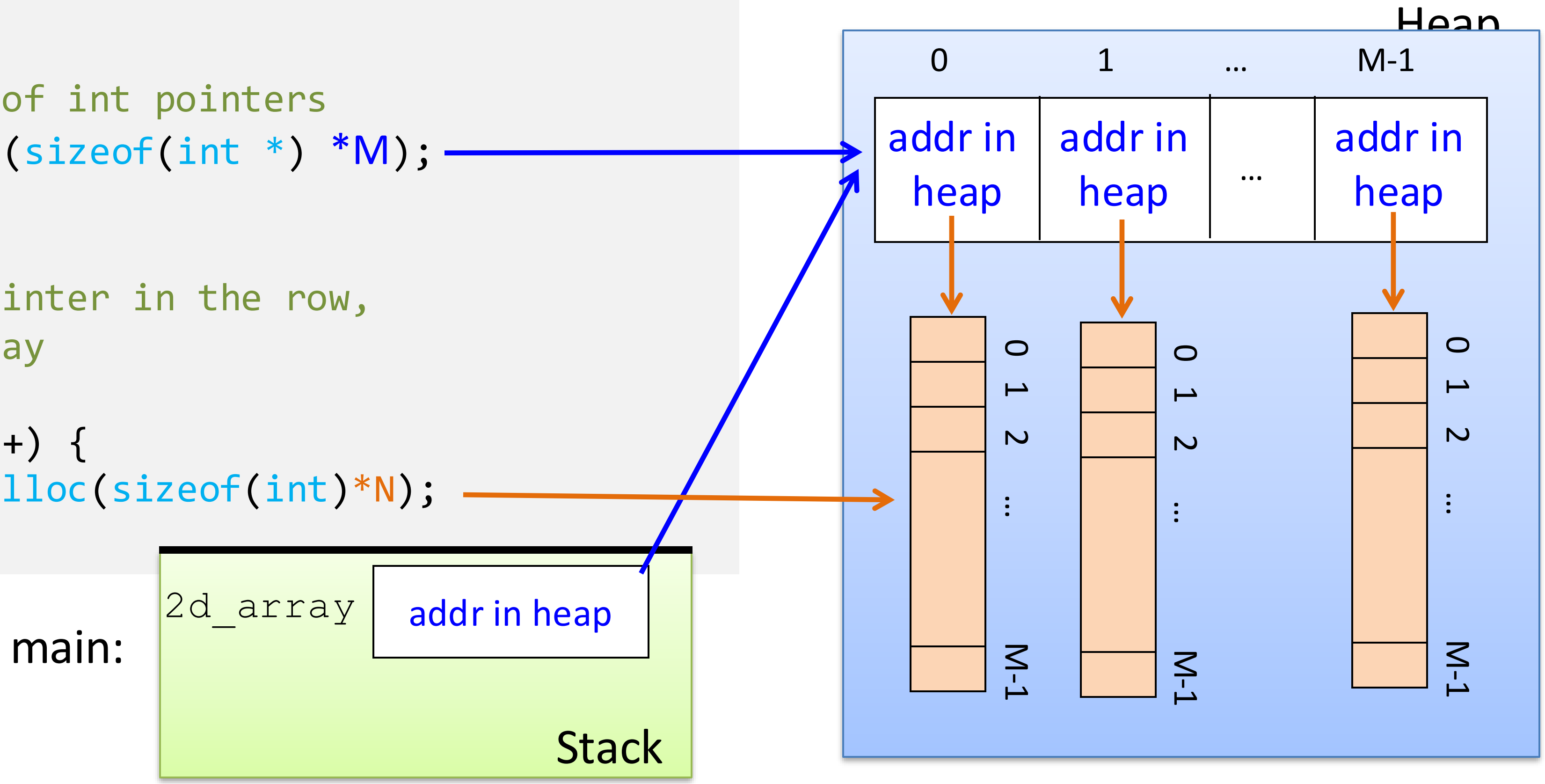
- If we want a dynamic **array** of **ints**:
  - declare `int *array = malloc(N * sizeof(int))`
  - Treat this internally as a 2D array (`i*COL + j`)

- If we want an **array** of **int pointers**:
  - declare `int **array = malloc(…)`
  - For *each pointer*, dynamically allocate an array
  - The type of array[0], array[1], etc. is: `int *`
  - For each one of those, we can malloc an array of ints:
    - `array[0] = malloc(M * sizeof(int))`

Dynamically Allocated 2D Array: Array of Pointers

- One malloc for an array of rows: an array of `int*`
- N mallocs for each row's column values: arrays of `int`
  - variable type is `int**`
  - stores address of rows array: an array of `int*`

```
int ** 2d_array;

// allocate a row of int pointers
2d_array = malloc (sizeof(int *) *M);


// for each int pointer in the row,
// allocate an array

for(i=0; i < M; i++) {
  2d_array[i] = malloc(sizeof(int)*N);
}
```

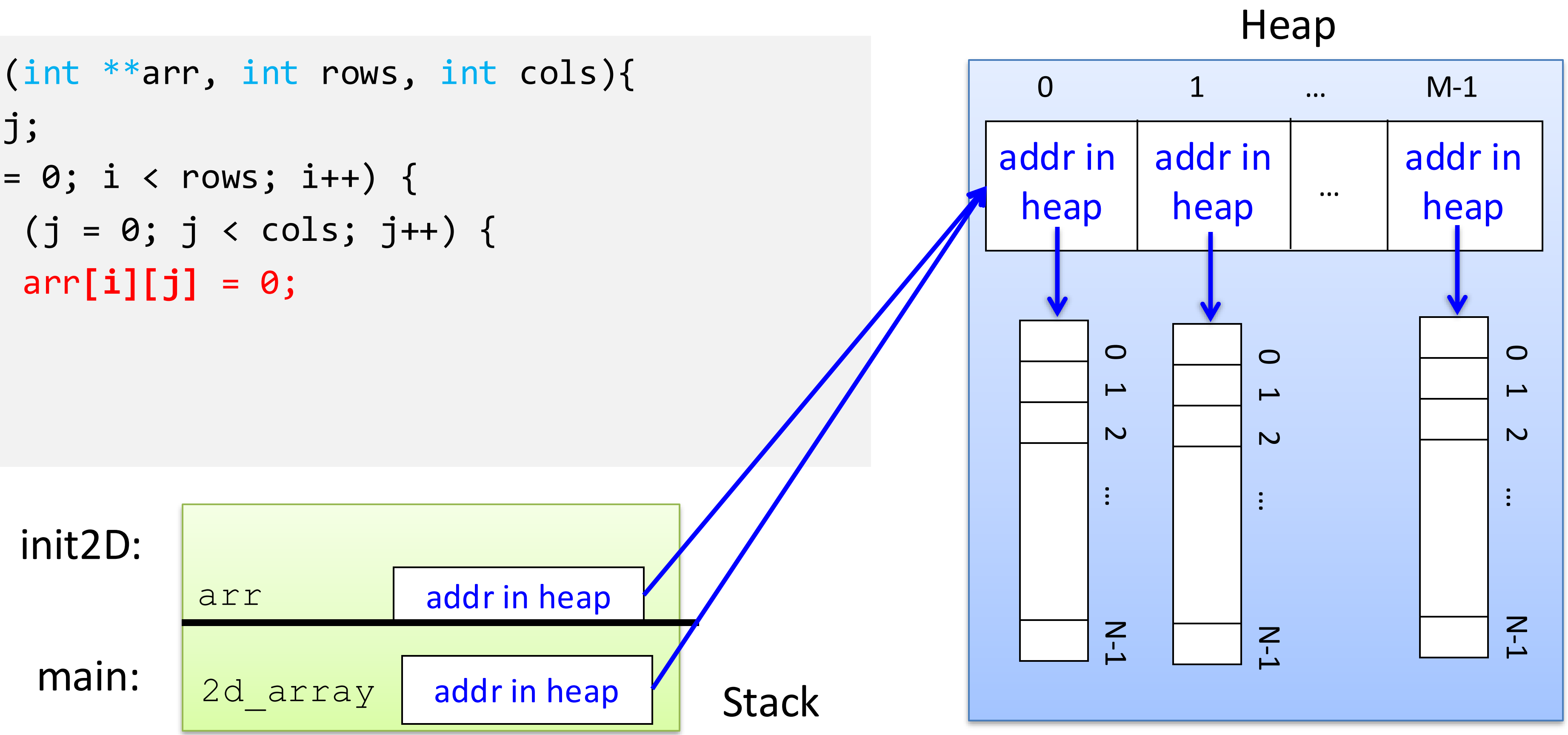


Using 2D Array (Array of Pointers) As Parameters

parameter gets base address of rows array of `int*`

- its type is `int**` : a pointer to `int*`: (with buckets of `int`)
- pass row and column dimension values
- Can use [i][j] to index into a specific location in 2D array.

```
void init2D(int **arr, int rows, int cols){
    int i, j;
    for (i = 0; i < rows; i++) {
        for (j = 0; j < cols; j++) {
            arr[i][j] = 0;
        }
    }
}
```

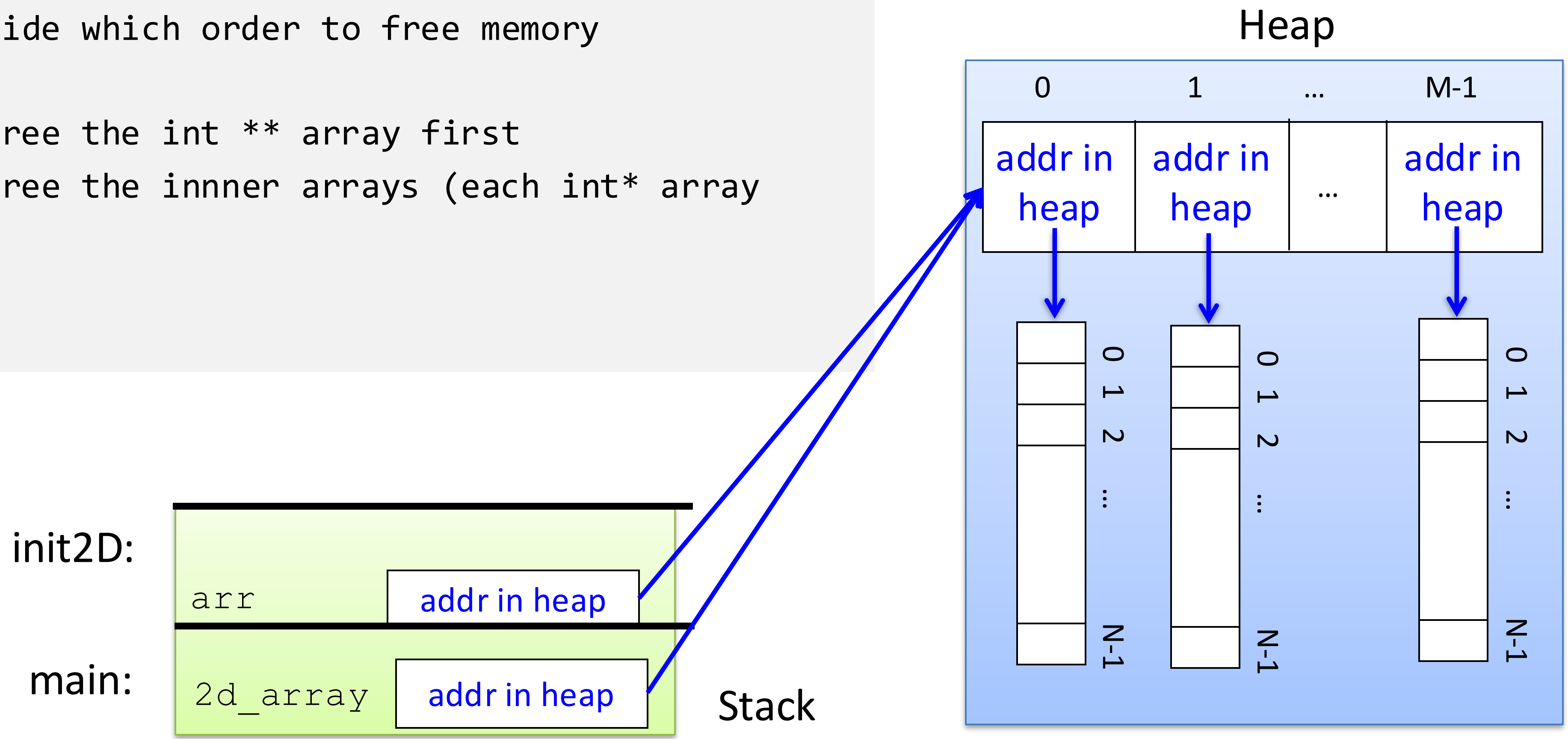


Using 2D Array (Array of Pointers): How about free-ing this memory?

parameter gets base address of rows array of int*

- its type is int** -> a pointer to an array of int*->
- each int* -> a pointer to an array of ints

```
void free(int **arr){
//TODO: decide which order to free memory

Option A: free the int ** array first
Option B: free the innner arrays (each int* array
first)

}
```



Two Ways for 2D Arrays

- We'll use BOTH methods in future labs:
  - **Lab 7**:
    - column-major, large chunk of memory that we treat as a 2D array,
    - use arr[index] where index = i * ROWSIZE + j to deference values
  - **Lab 8/9**:
    - array of integer pointers,
    - can use arr[N][M] to dereference values

Structs

- Multiple values (fields) stored together
  - Defines a new type in C's type system

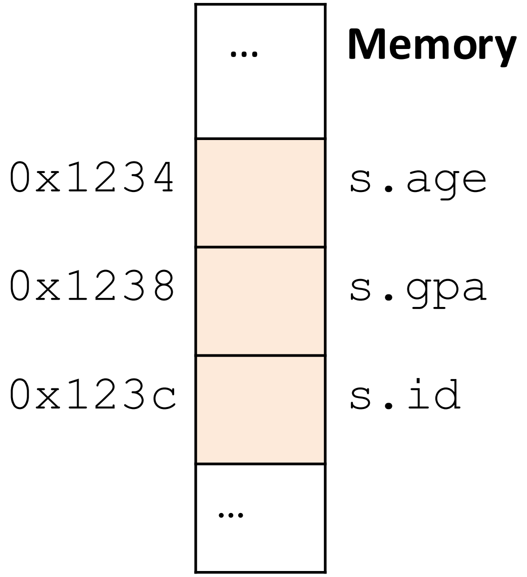
- Laid out contiguously by field (with a caveat we'll see later)
  - In order of field declaration.

Structs

Laid out contiguously by field (with a caveat we'll see later)

- In order of field declaration.

```
struct student{
  int age;
  float gpa;
  int id;
};

struct student s;
```

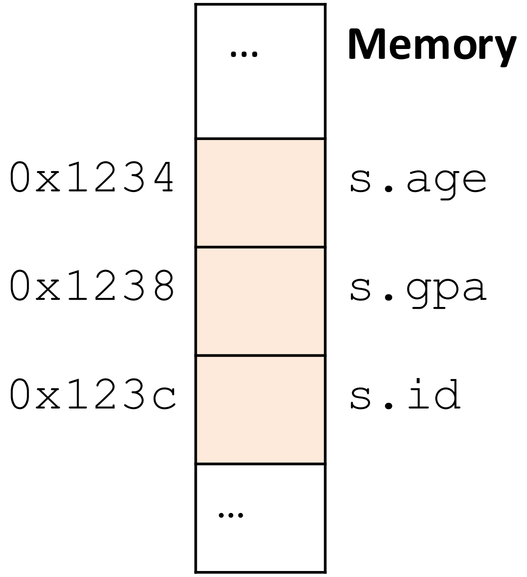


Structs

Struct fields accessible as a base + displacement

- Compiler knows (constant) displacement of each field

```
struct student{
   int age;
   float gpa;
   int id;
};

struct student s;
```



Memory

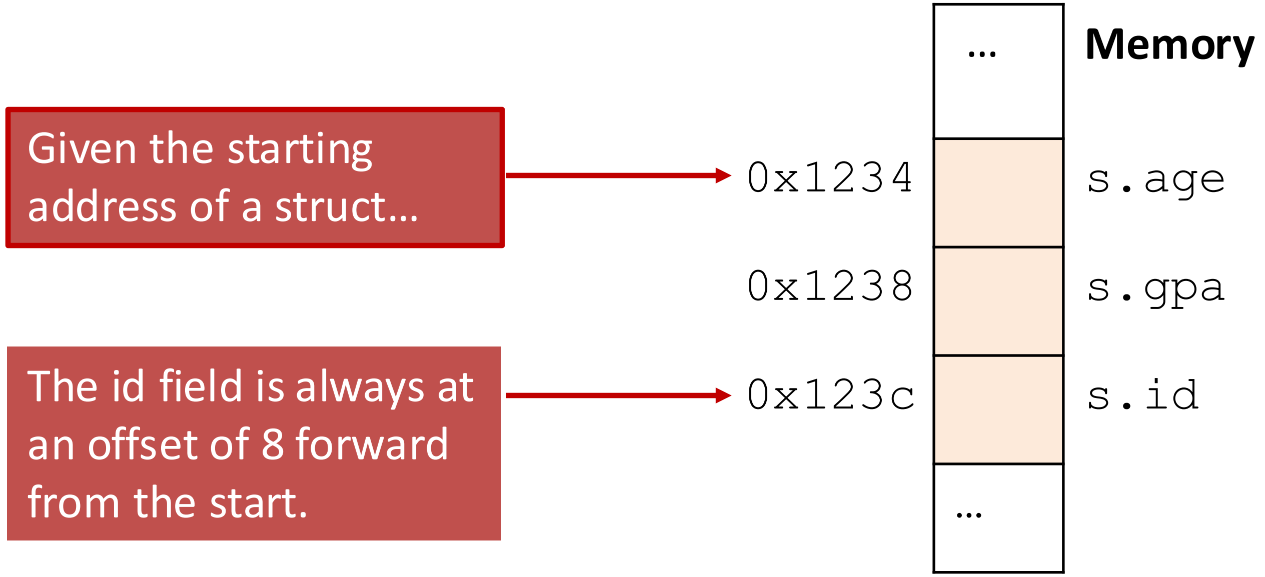
	…	
0x1234		s.age
0x1238		s.gpa
0x123c		s.id
	…	

Structs

Struct fields accessible as a base + displacement

- Compiler knows (constant) displacement of each field

```
struct student{
  int age;
  float gpa;
  int id;
};

struct student s;
```



Given the starting address of a struct…

The id field is always at an offset of 8 forward from the start.

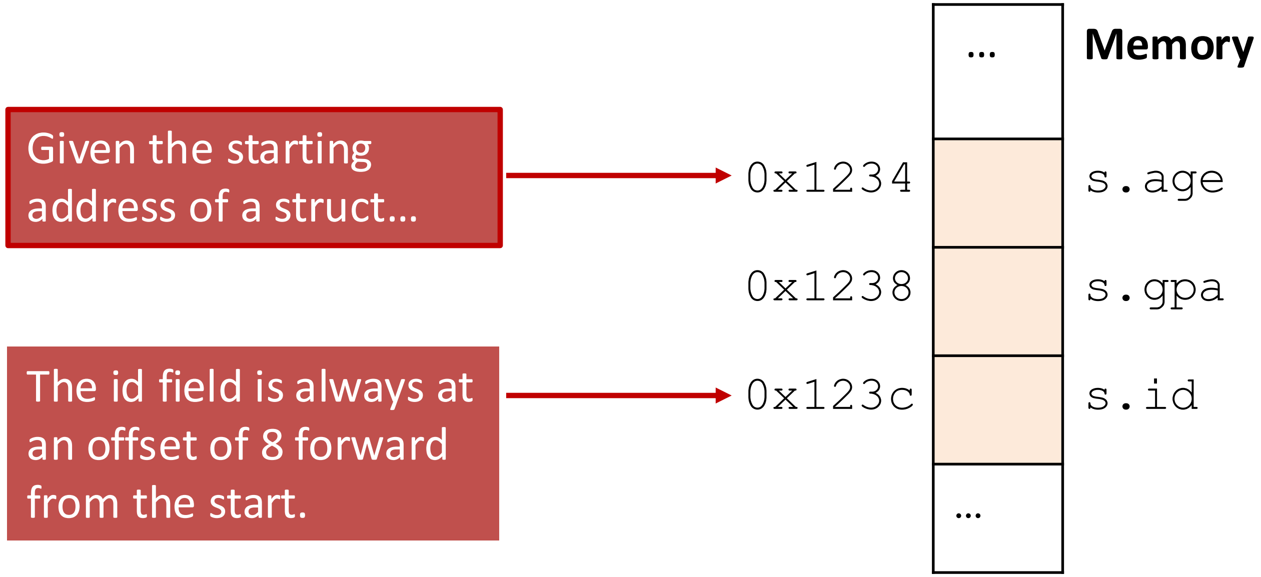
Structs

Struct fields accessible as a base + displacement
In assembly: `mov reg_value, 8(reg_base)`

Where:

- `reg_value` is a register holding the value to store (say, 12)
- `reg_base` is a register holding the base address of the struct

```
struct student{
  int age;
  float gpa;
  int id;
};

struct student s;
s.id = 12;
```

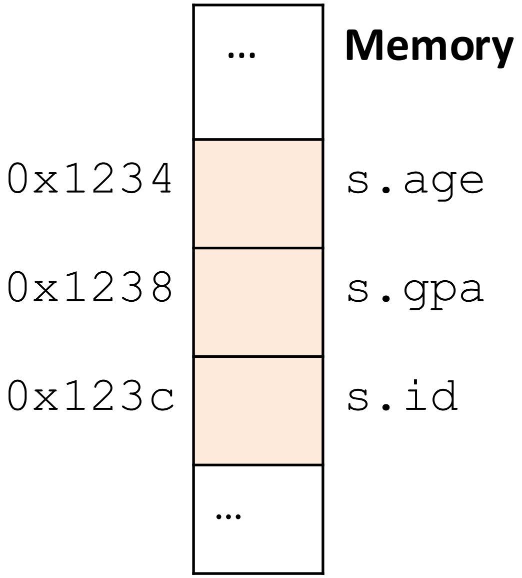


Given the starting address of a struct…

The id field is always at an offset of 8 forward from the start.

Structs

- Laid out contiguously by field
  - In order of field declaration.
  - May require some padding, for alignment.



Data Alignment:

- Where (which address) can a field be located?

- char (1 byte): can be allocated at any address:

  0x1230, 0x1231, 0x1232, 0x1233, 0x1234, …

- short (2 bytes): must be aligned on 2-byte addresses:

  0x123**0**, 0x123**2**, 0x123**4**, 0x123**6**, 0x123**8**, …

- int (4 bytes): must be aligned on 4-byte addresses:

  0x123**0**, 0x123**4**, 0x123**8**, 0x123**c**, 0x124**0**, …

Why do we want to align data on multiples of the data size?

A. It makes the hardware faster.

B. It makes the hardware simpler.

C. It makes more efficient use of memory space.

D. It makes implementing the OS easier.

E. Some other reason.

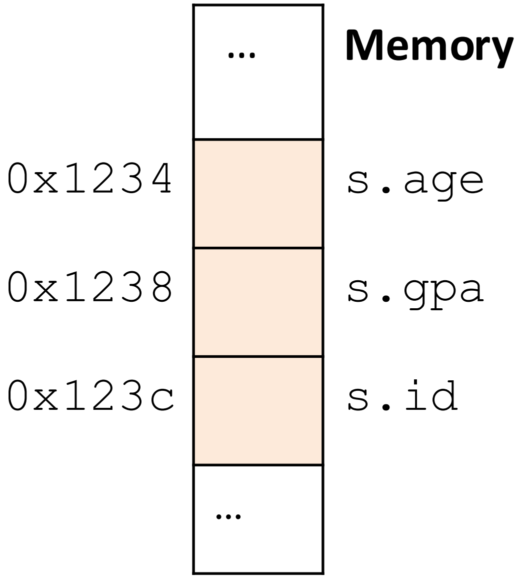
Data Alignment: Why?

- Simplify hardware
  - e.g., only read ints from multiples of 4
  - Don’t need to build wiring to access 4-byte chunks at any arbitrary location in hardware
- Inefficient to load/store single value across alignment boundary (1 vs. 2 loads)
- Simplify OS:
  - Prevents data from spanning virtual pages
  - Atomicity issues with load/store across boundary

Structs

- Laid out contiguously by field
  - In order of field declaration.
  - May require some padding, for alignment.

```
struct student{
  int age;
  float gpa;
  int id;
};

struct student s;
```



Structs

```c
struct student{
  char name[11];
  short age;
  int id;
};
```

How much space do we need to store one of these structures? Why?

```
struct student{
  char name[11];
  short age;
  int id;
};
```

A. 17 bytes

B. 18 bytes

C. 20 bytes

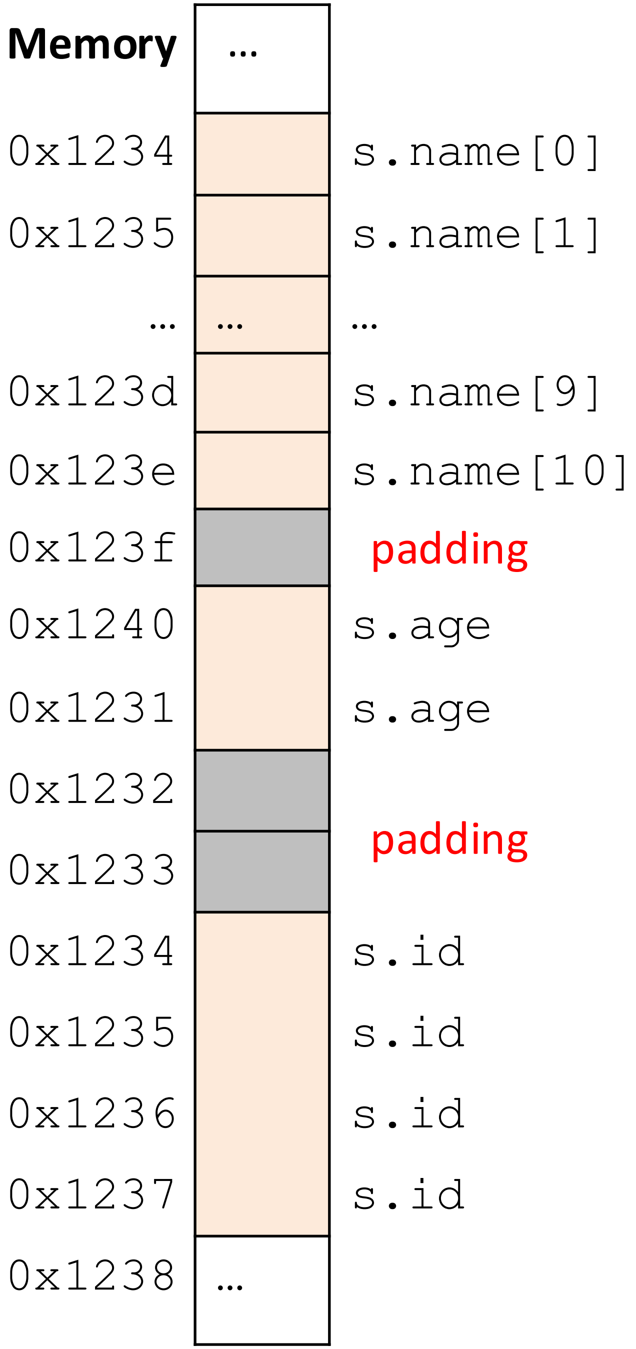
D. 22 bytes

E. 24 bytes

Structs

```
struct student{
  char name[11];
  short age;
  int id;
};

size of data:  17 bytes
size of struct: 20 bytes!
```

Use sizeof() when allocating structs with malloc()!



Memory	…	
0x1234		s.name[0]
0x1235		s.name[1]
…	…	…
0x123d		s.name[9]
0x123e		s.name[10]
0x123f	▨	padding
0x1240		s.age
0x1231		s.age
0x1232	▨	padding
0x1233	▨	
0x1234		s.id
0x1235		s.id
0x1236		s.id
0x1237		s.id
0x1238	…	

Alternative Layout

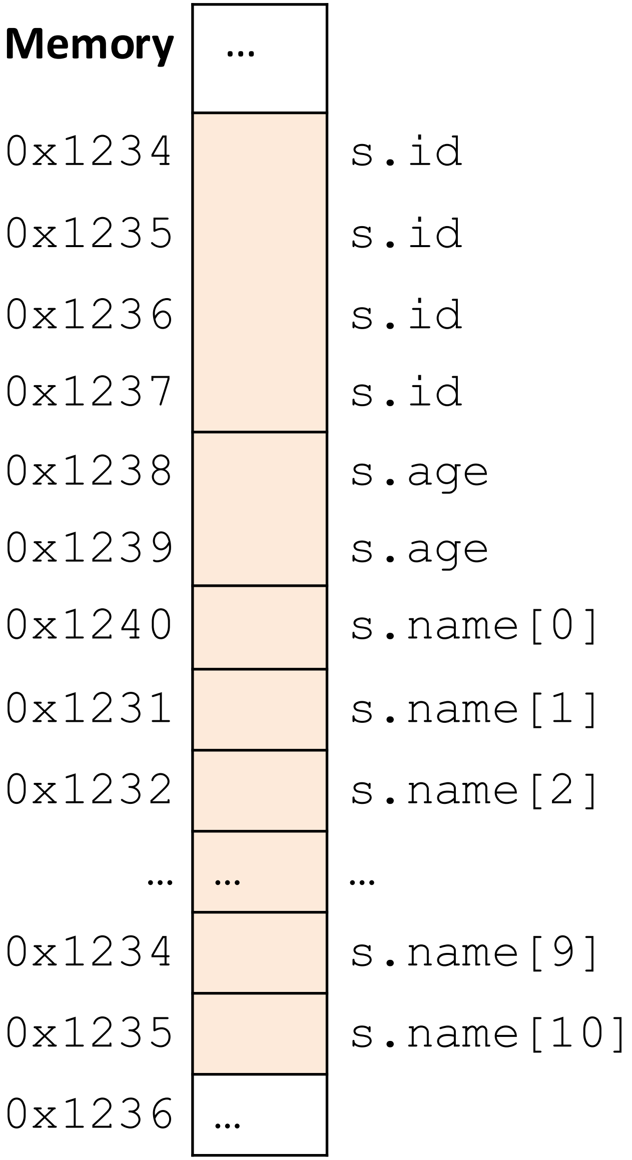
```
struct student{
   char name[11];
   short age;
   int id;
};
```

Same fields, declared in a different order.

Alternative Layout

```
struct student{
  char name[11];
  short age;
  int id;
};

size of data:  17 bytes
size of struct: 17 bytes
```

In general, this isn't a big deal on a day-to-day basis. Don't go out and rearrange all your struct declarations.



Memory	…	
0x1234		s.id
0x1235		s.id
0x1236		s.id
0x1237		s.id
0x1238		s.age
0x1239		s.age
0x1240		s.name[0]
0x1231		s.name[1]
0x1232		s.name[2]
…	…	…
0x1234		s.name[9]
0x1235		s.name[10]
0x1236	…	

Aside: Network Headers

- In networks, we attach metadata to packets
  - Things like destination address, port #, etc.
- Common for these to be a specific size/format
  - e.g., the first 20 bytes must be laid out like …
- Naïvely declaring a struct might introduce padding, violate format.

Cool, so we can get rid of this struct padding by being smart about declarations?

A. Yes (why?)

B. No (why not?)

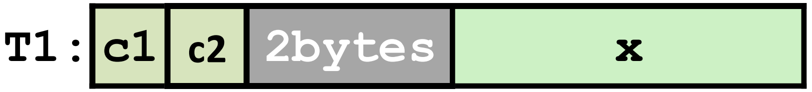
Cool, so we can get rid of this padding by being smart about declarations?

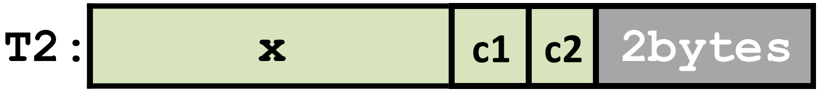
- Answer: Maybe.

- Rearranging helps, but often padding after the struct can’t be eliminated.

```
struct T1 {
    char c1;
    char c2;
    int  x;
};
```

```
struct T2 {
    int x;
    char c1;
    char c2;
};
```

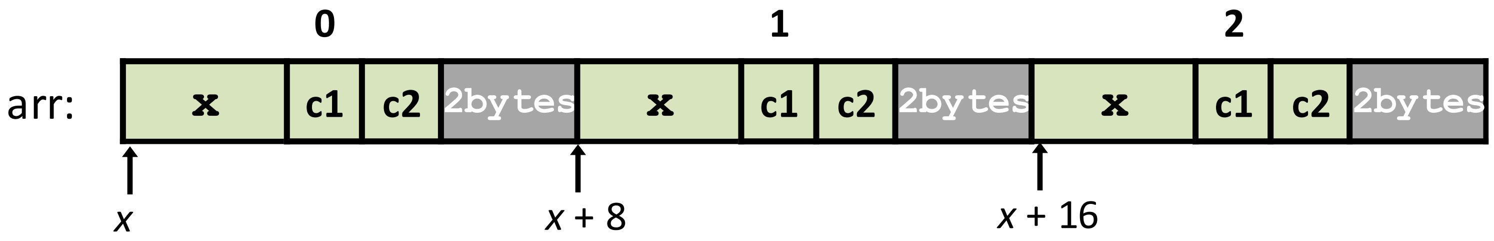




“External” Padding

Array of Structs: Field values in each bucket must be properly aligned:

```
struct T2 arr[3];
```



Buckets must be on a 8-byte aligned address

Struct field syntax…

```
struct student {
  int id;
  short age;
  char name[11];
};
struct student s;

s.id = 406432;
s.age = 20;
strcpy(s.name, “Alice”);
```

Struct is declared on the stack.
(NOT a pointer)

Struct field syntax…

What about this?

```
struct student {
  int id;
  short age;
  char name[11];
};
struct student *s = malloc(sizeof(struct student));
```

How do we get to the id and age?

Struct field syntax…

What about this?

```
struct student {
    int id;
    short age;
    char name[11];
};
struct student *s = malloc(sizeof(struct student));
```

How do we get to the id and age?

Option 1: Works but ugly

```
(*s).id = 406432;
(*s).age = 20;
strcpy((*s).name, “Alice”);
```

Option 2: Use struct pointer dereference!

```
s->id = 406432;
s->age = 20;
strcpy(s->name, “Alice”);
```

✓

Memory alignment applies elsewhere too!

```
int x;         vs.     double y;
char ch[5];            int x;
short s;               short s;
double y;              char ch[5];
```

In nearly all cases, *you shouldn't stress about this.* The compiler will figure out where to put things.

Exceptions: networking, OS

Structs and Arrays

- Use Structs & Arrays to build complex data types
- Very important to think about type!

  from the outside in: (e.g.) a[3].age

  - type of a is a pointer to an array of student
  - can use [i] notation to access a bucket of this array
  - type of a[3] is a student struct
  - can use **.** to access a field in struct
  - type of a[3].age is an int
- Remember how different types are passed
  - semantics of passing an array vs. a struct
  - it is all pass by value, but what value is differs by type