CS107, Lecture 20
Assembly: Function Calls and the Runtime Stack

Reading: B&O 3.7

This document is copyright (C) Stanford Computer Science and Nick Troccoli, licensed under Creative Commons Attribution 2.5 License. All rights reserved.
Based on slides created by Cynthia Lee, Chris Gregg, Jerry Cain, Lisa Yan and others.
NOTICE RE UPLOADING TO WEBSITES: This content is protected and may not be shared, uploaded, or distributed. (without expressed written permission)

CS107 Topic 5

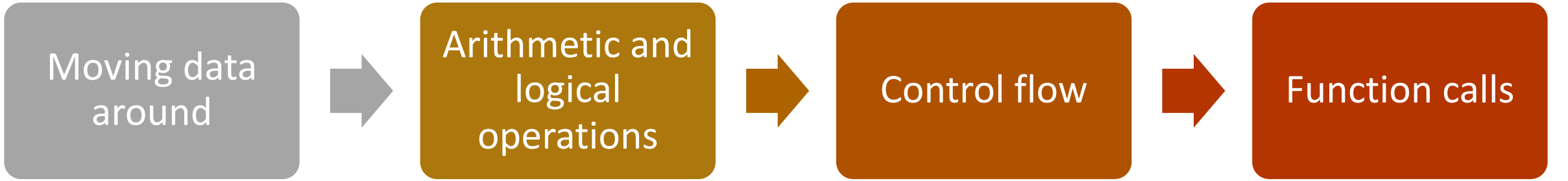
How does a computer interpret and execute C programs?

Why is answering this question important?

- Learning how our code is really translated and executed helps us write better code
- We can learn how to reverse engineer and exploit programs at the assembly level

assign5: find and exploit vulnerabilities in an ATM program, reverse engineer a program without seeing its code, and de-anonymize users given a data leak.

Learning Assembly



Reference Sheet: `cs107.stanford.edu/resources/x86-64-reference.pdf`
See more guides on Resources page of course website!

Learning Goals

- Learn how assembly calls functions and manages stack frames.
- Learn the rules of register use when calling functions.

Lecture Plan

- **Other Instructions That Depend On Condition Codes**

Condition Code-Dependent Instructions

There are three common instruction types that use condition codes:

- **jmp** instructions conditionally jump to a different next instruction
- **set** instructions conditionally set a byte to 0 or 1
- new versions of **mov** instructions conditionally move data

`set`: Read condition codes

`set` instructions conditionally set a byte to 0 or 1.

- Reads current state of flags
- Destination is a single-byte register (e.g., %al) or single-byte memory location
- Does not perturb other bytes of register
- Typically followed by movzbl to zero those bytes

```
int small(int x) {
    return x < 16;
}
```

```
cmp $0xf,%edi
setle %al
movzbl %al, %eax
retq
```

`set`: Read condition codes

Instruction	Synonym	Set Condition (1 if true, 0 if false)
`sete D`	`setz`	Equal / zero
`setne D`	`setnz`	Not equal / not zero
`sets D`		Negative
`setns D`		Nonnegative
`setg D`	`setnle`	Greater (signed >)
`setge D`	`setnl`	Greater or equal (signed >=)
`setl D`	`setnge`	Less (signed <)
`setle D`	`setng`	Less or equal (signed <=)
`seta D`	`setnbe`	Above (unsigned >)
`setae D`	`setnb`	Above or equal (unsigned >=)
`setb D`	`setnae`	Below (unsigned <)
`setbe D`	`setna`	Below or equal (unsigned <=)

cmov: Conditional move

`cmovx src,dst` conditionally moves data in `src` to data in `dst`.

- Mov `src` to `dst` if condition `x` holds; no change otherwise
- `src` is memory address/register, `dst` is register
- May be more efficient than branch (i.e., jump)
- Often seen with C ternary operator: `result = test ? then: else;`

```
int max(int x, int y) {
    return x > y ? x : y;
}
```

```
cmp     %edi,%esi
mov     %edi, %eax
cmovge  %esi, %eax
retq
```

cmov: Conditional move

Instruction	Synonym	Move Condition
`cmove S,R`	`cmovz`	Equal / zero (ZF = 1)
`cmovne S,R`	`cmovnz`	Not equal / not zero (ZF = 0)
`cmovs S,R`		Negative (SF = 1)
`cmovns S,R`		Nonnegative (SF = 0)
`cmovg S,R`	`cmovnle`	Greater (signed >) (SF = 0 and SF = OF)
`cmovge S,R`	`cmovnl`	Greater or equal (signed >=) (SF = OF)
`cmovl S,R`	`cmovnge`	Less (signed <) (SF != OF)
`cmovle S,R`	`cmovng`	Less or equal (signed <=) (ZF = 1 or SF! = OF)
`cmova S,R`	`cmovnbe`	Above (unsigned >) (CF = 0 and ZF = 0)
`cmovae S,R`	`cmovnb`	Above or equal (unsigned >=) (CF = 0)
`cmovb S,R`	`cmovnae`	Below (unsigned <) (CF = 1)
`cmovbe S,R`	`cmovna`	Below or equal (unsigned <=) (CF = 1 or ZF = 1)

Last Lab: Conditional Move

```
int signed_division(int x) {
    return x / 4;
}
```

```
signed_division:
  leal 3(%rdi), %eax
  testl %edi, %edi
  cmovns %edi, %eax
  sarl $2, %eax
  ret
```

Put x + 3 into %eax

Check the sign of x

If x is nonnegative, put x into %eax

Divide %eax by 4

Lecture Plan

- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage
- Register Restrictions

```
cp -r /afs/ir/class/cs107/lecture-code/lect20 .
```

Lecture Plan

- **Calling Functions**
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage
- Register Restrictions

```
cp -r /afs/ir/class/cs107/lecture-code/lect20 .
```

Calling Functions In Assembly

To call a function in assembly, we must do a few things:

- **Pass Control** – %rip must be adjusted to execute the callee's instructions, and then resume the caller's instructions afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

How does assembly interact with the stack?

Terminology: **caller** function calls the **callee** function.

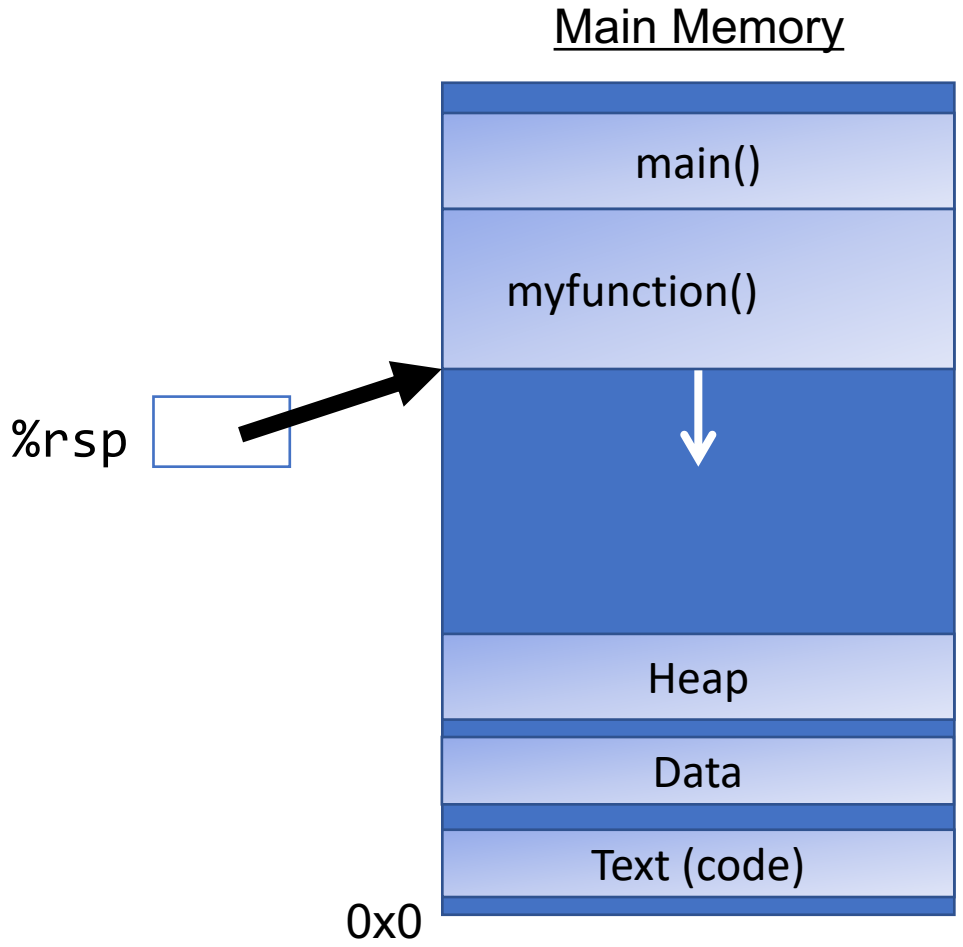
Lecture Plan

- **Calling Functions**
  - **The Stack**
  - Passing Control
  - Passing Data
  - Local Storage
- Register Restrictions

```
cp -r /afs/ir/class/cs107/lecture-code/lect20 .
```

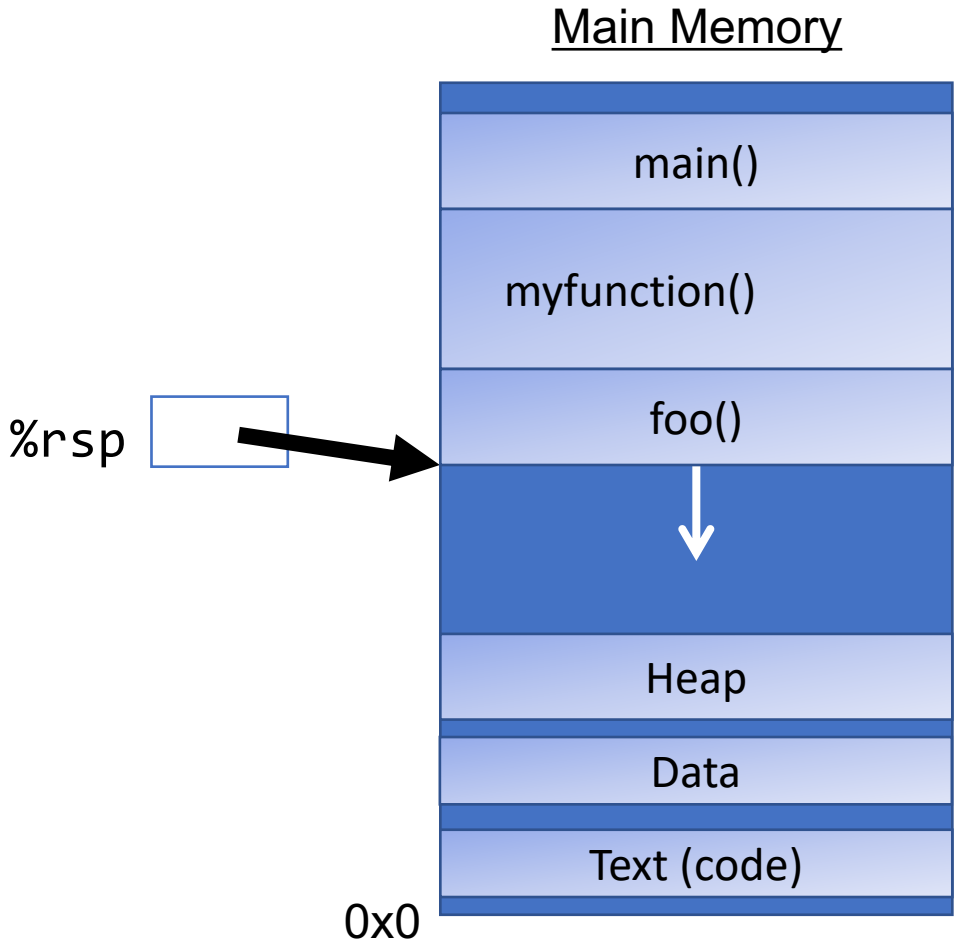
%rsp

- **%rsp** is a special register that stores the address of the current "top" of the stack (the bottom in our diagrams, since the stack grows downwards).



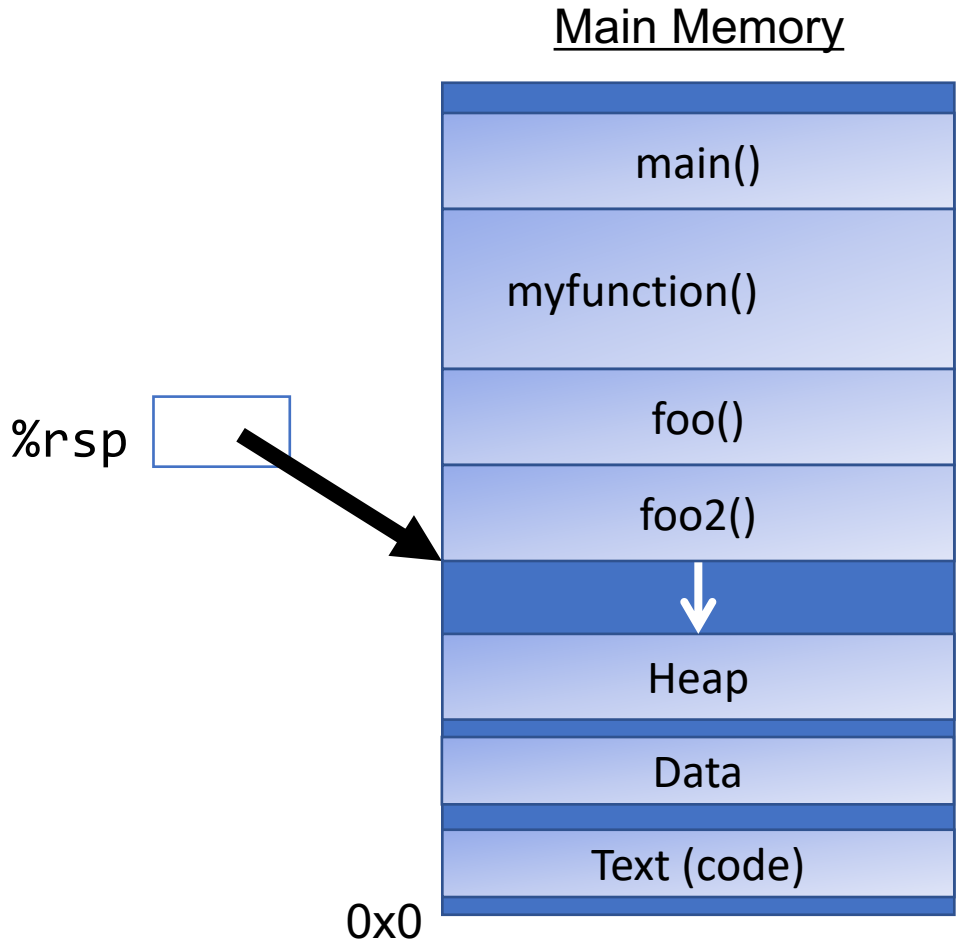
%rsp

- **%rsp** is a special register that stores the address of the current "top" of the stack (the bottom in our diagrams, since the stack grows downwards).



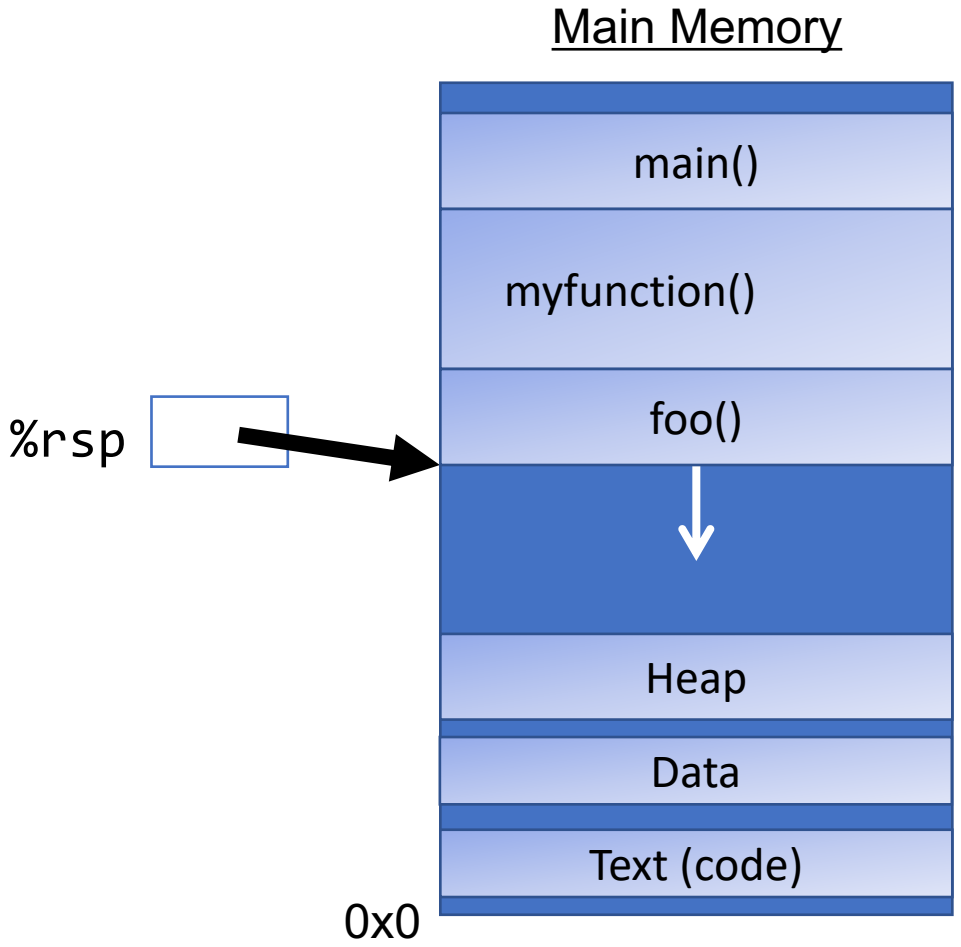
%rsp

- **%rsp** is a special register that stores the address of the current "top" of the stack (the bottom in our diagrams, since the stack grows downwards).



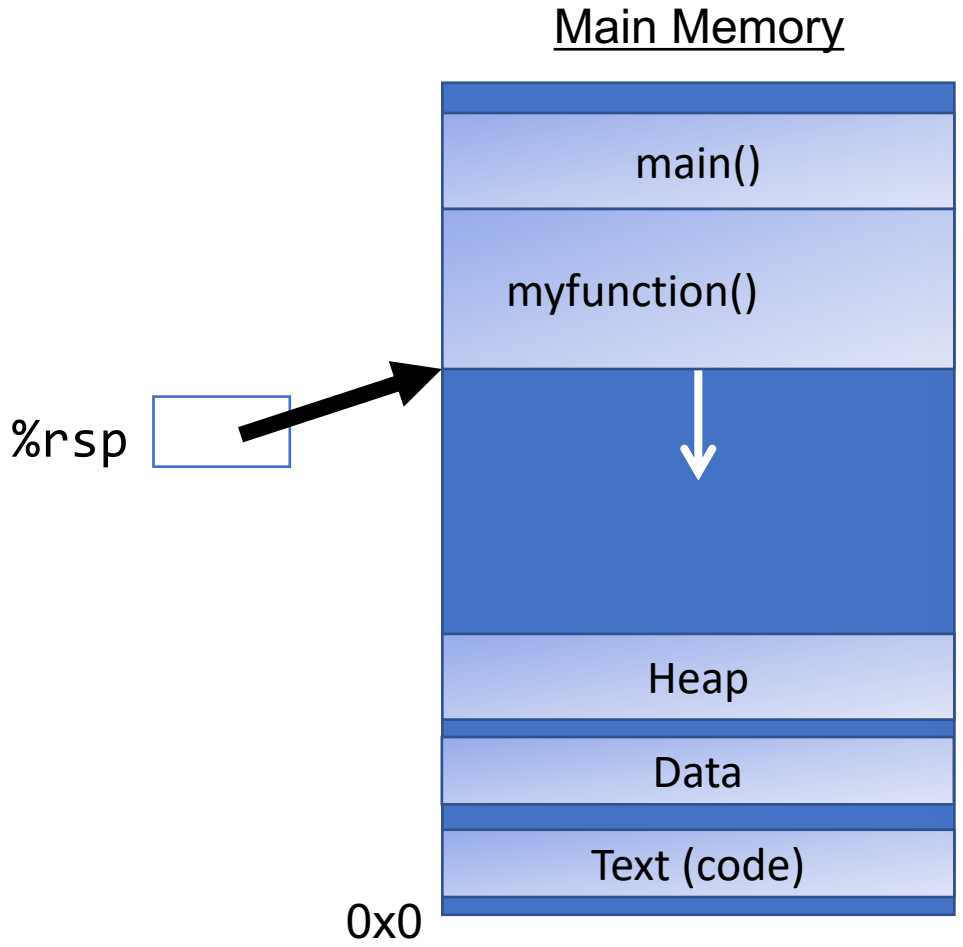
%rsp

- **%rsp** is a special register that stores the address of the current "top" of the stack (the bottom in our diagrams, since the stack grows downwards).



%rsp

- **%rsp** is a special register that stores the address of the current "top" of the stack (the bottom in our diagrams, since the stack grows downwards).



Key idea: %rsp must point to the same place before a function is called and after that function returns, since stack frames go away when a function finishes.

push

- The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting **%rsp** accordingly.

Instruction	Effect
`pushq S`	`R[%rsp] ⟵ R[%rsp] - 8;` `M[R[%rsp]] ⟵ S`

push

- The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting **%rsp** accordingly.

Instruction	Effect
`pushq S`	`R[%rsp] ⟵ R[%rsp] - 8;` `M[R[%rsp]] ⟵ S`

push

- The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting **%rsp** accordingly.

Instruction	Effect
`pushq S`	`R[%rsp] ⟵ R[%rsp] - 8;` **`M[R[%rsp]] ⟵ S`**

push

The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting **%rsp** accordingly.

Instruction	Effect
`pushq S`	`R[%rsp] ⟵ R[%rsp] - 8;` `M[R[%rsp]] ⟵ S`

- This behavior is equivalent to the following, but pushq is a shorter instruction:

```
subq $8, %rsp
movq  S, (%rsp)
```

- Sometimes, you'll see instructions just explicitly decrement the stack pointer to make room for future data.

pop

The **pop** instruction pops the topmost data from the stack and stores it in the specified destination, adjusting **%rsp** accordingly.

Instruction	Effect
`popq D`	`D ⟵ M[R[%rsp]]` `R[%rsp] ⟵ R[%rsp] + 8;`

- **Note**: this *does not* remove/clear out the data! It just increments %rsp to indicate the next push can overwrite that location.

pop

The **pop** instruction pops the topmost data from the stack and stores it in the specified destination, adjusting **%rsp** accordingly.

Instruction	Effect
popq D	D ⟵ M[R[%rsp]] R[%rsp] ⟵ R[%rsp] + 8;

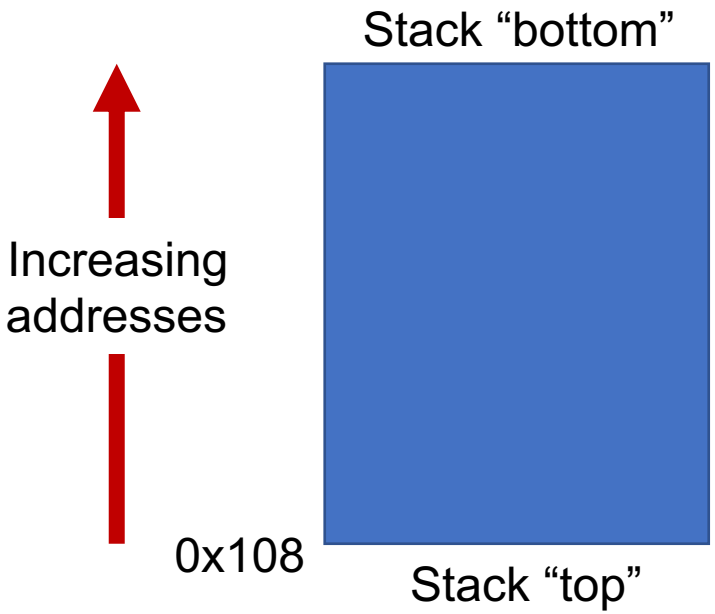
- This behavior is equivalent to the following, but popq is a shorter instruction:

```
movq (%rsp), D
addq $8, %rsp
```

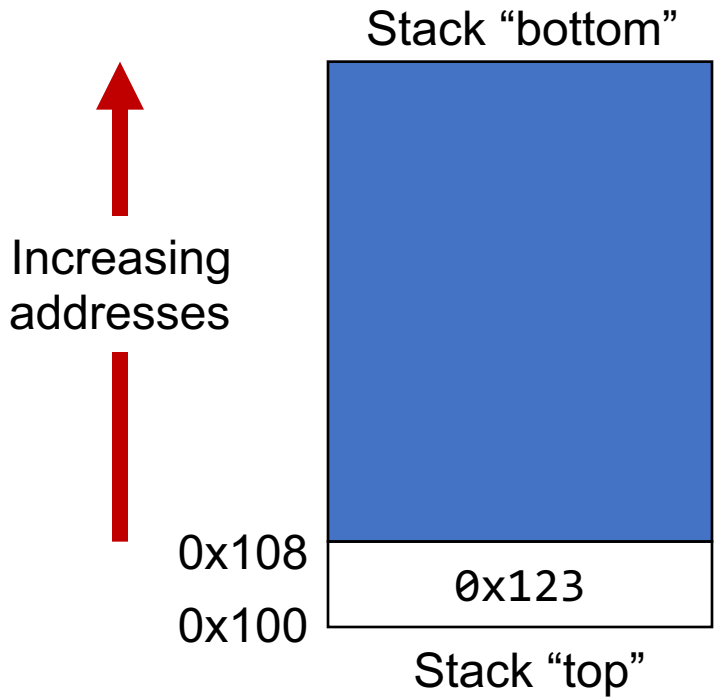
- Sometimes, you'll see instructions just explicitly increment the stack pointer to pop data.

Stack Example

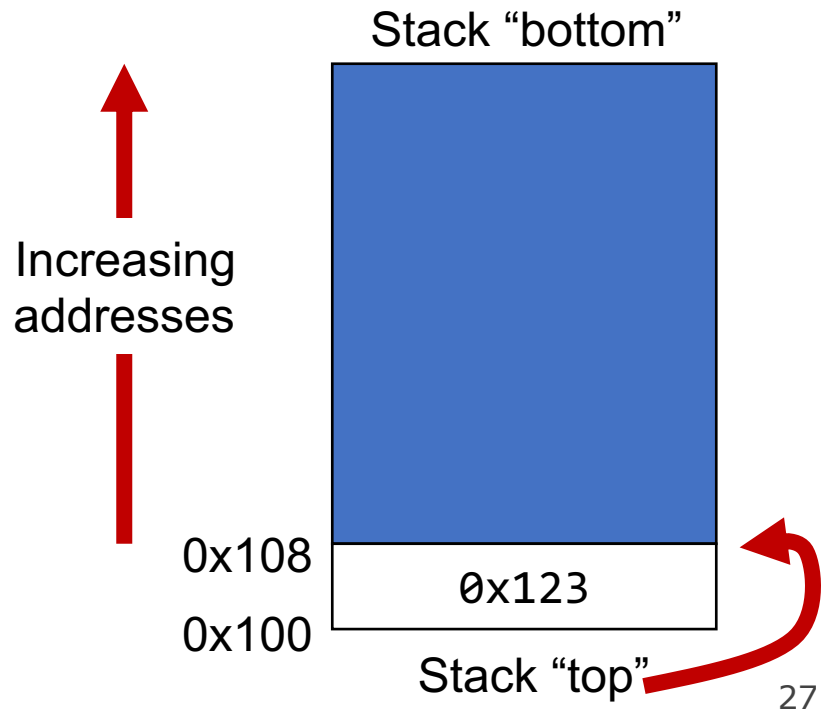
Initially	
%rax	0x123
%rdx	0
%rsp	0x108



pushq %rax	
%rax	0x123
%rdx	0
%rsp	0x100



popq %rdx	
%rax	0x123
%rdx	0x123
%rsp	0x108



Calling Functions In Assembly

To call a function in assembly, we must do a few things:

- **Pass Control** – %rip must be adjusted to execute the callee's instructions, and then resume the caller's instructions afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.

Lecture Plan

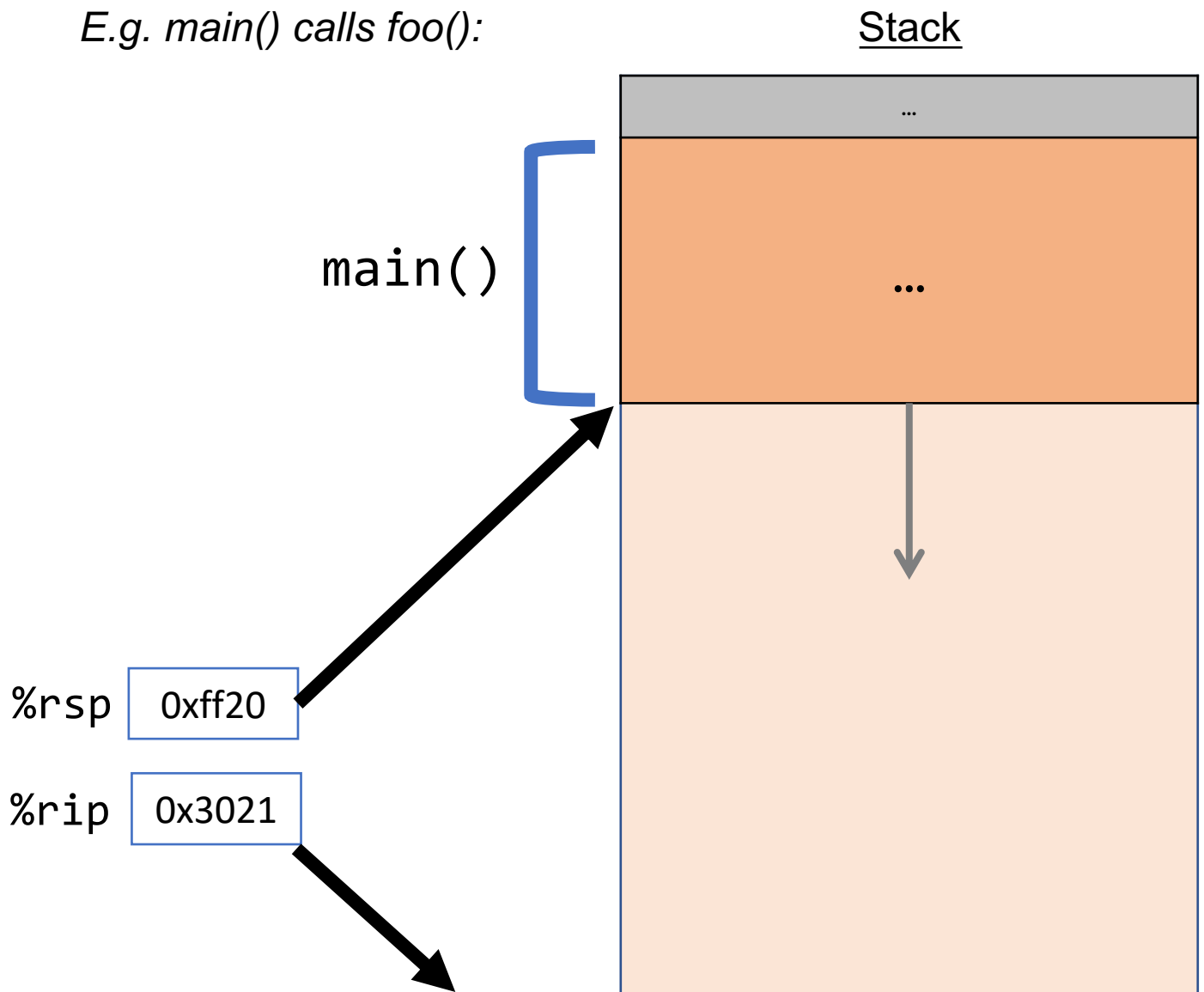
- **Calling Functions**
  - The Stack
  - **Passing Control**
  - Passing Data
  - Local Storage
- Register Restrictions

```
cp -r /afs/ir/class/cs107/lecture-code/lect20 .
```

Remembering Where We Left Off

Problem: %rip points to the next instruction to execute. To call a function, we must remember the *next* caller instruction to resume at after.

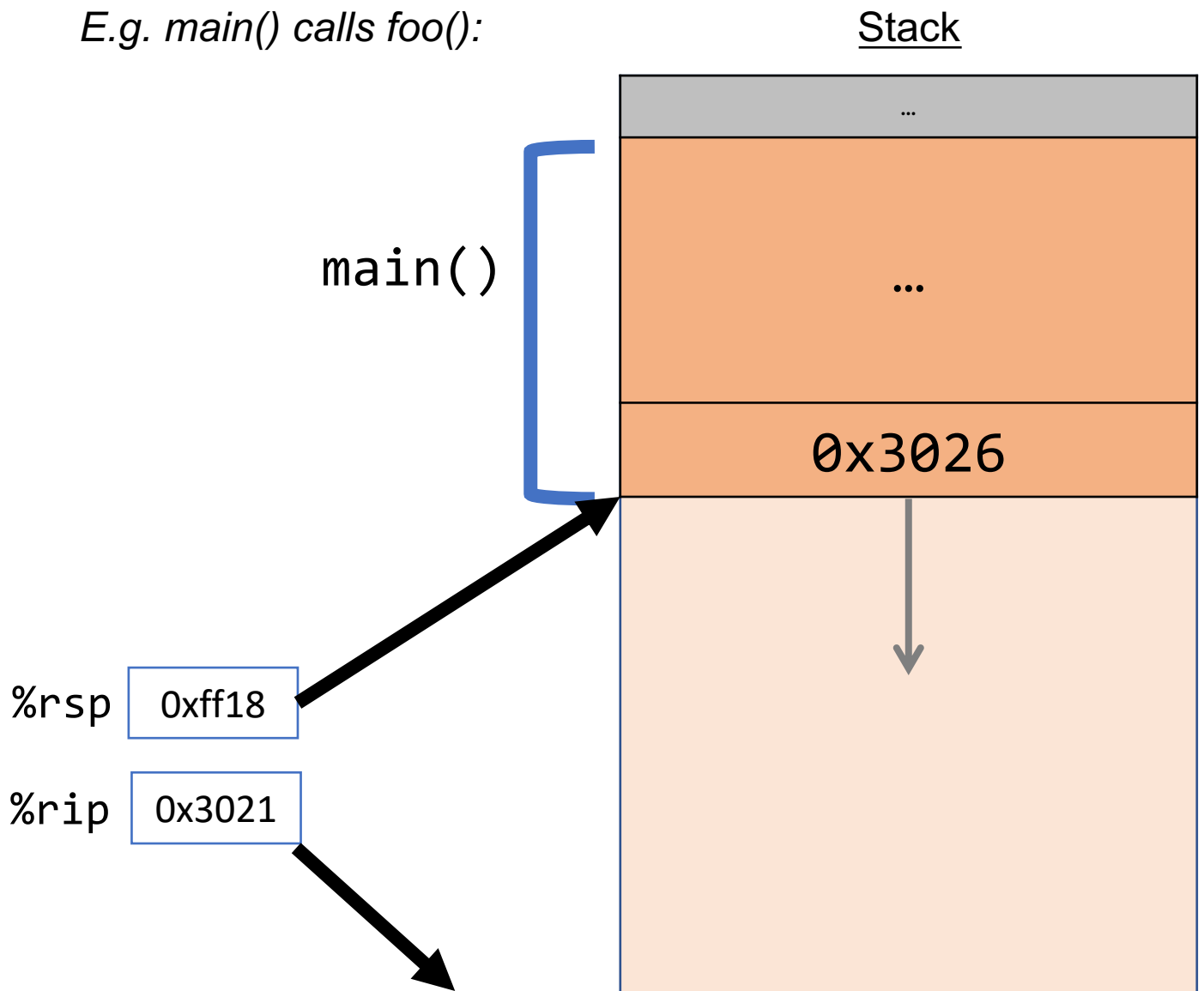
Solution: push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.



Remembering Where We Left Off

Problem: %rip points to the next instruction to execute. To call a function, we must remember the *next* caller instruction to resume at after.

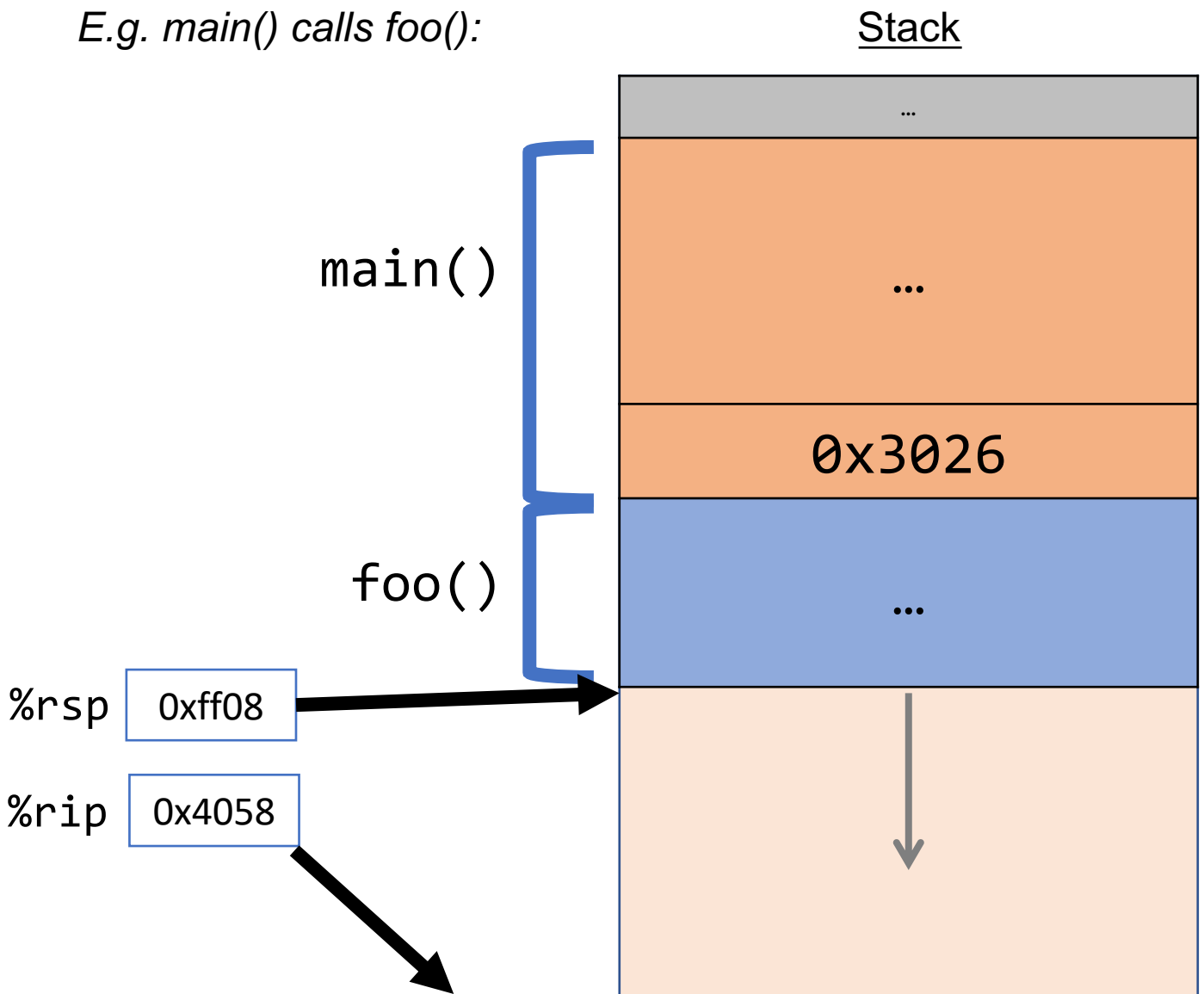
Solution: push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.



Remembering Where We Left Off

Problem: %rip points to the next instruction to execute. To call a function, we must remember the *next* caller instruction to resume at after.

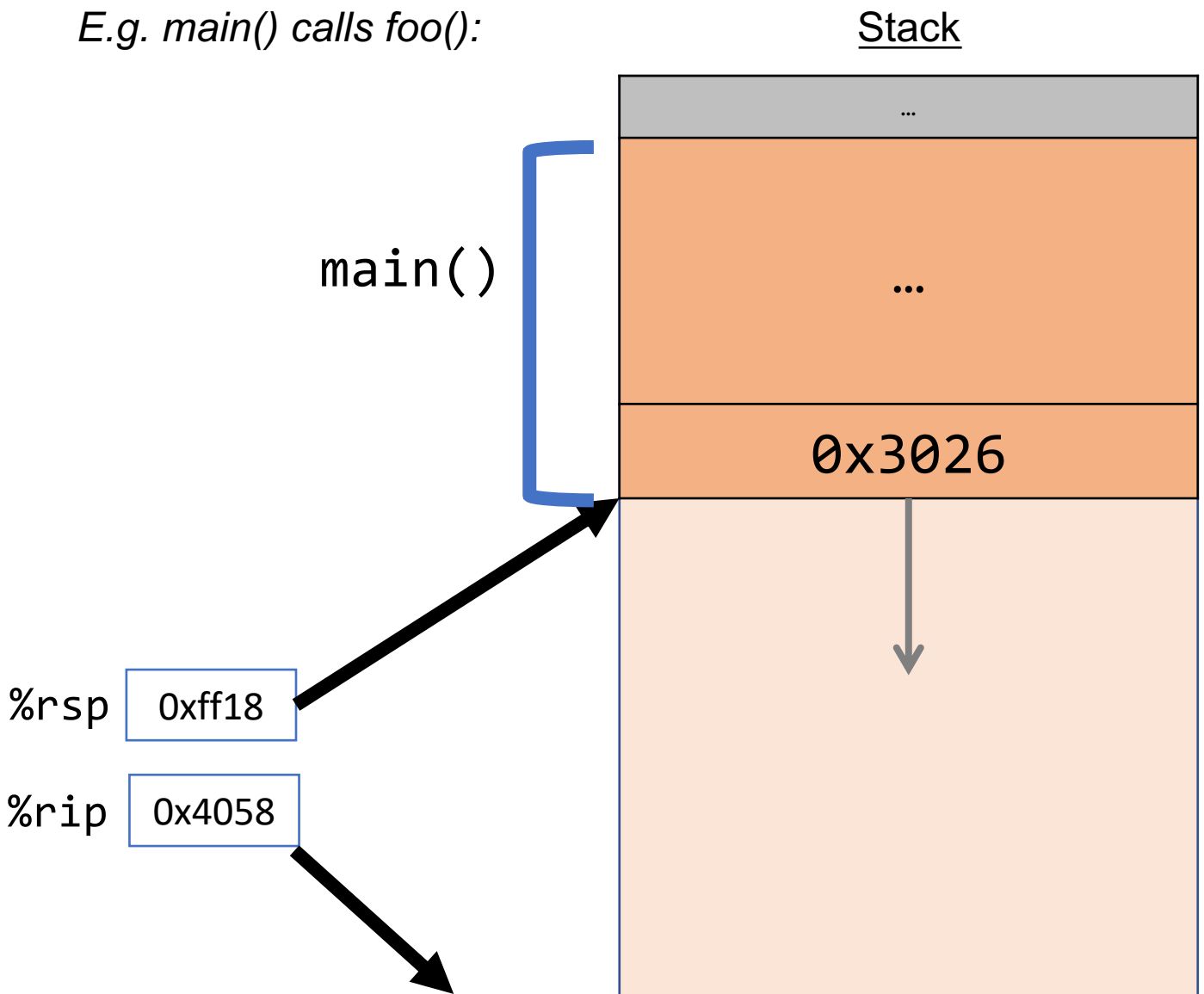
Solution: push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.



Remembering Where We Left Off

Problem: %rip points to the next instruction to execute. To call a function, we must remember the *next* caller instruction to resume at after.

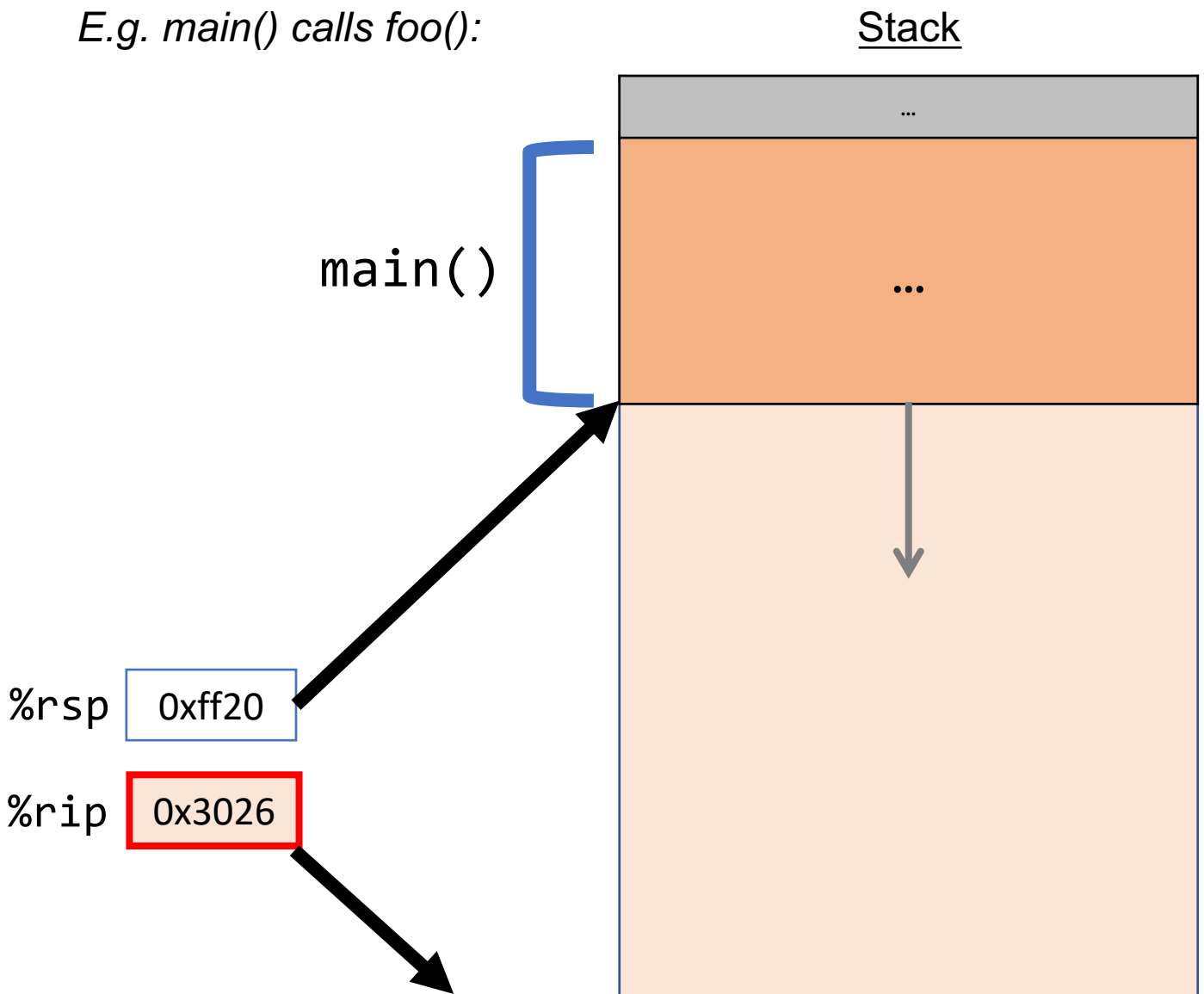
Solution: push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.



Remembering Where We Left Off

Problem: %rip points to the next instruction to execute. To call a function, we must remember the *next* caller instruction to resume at after.

Solution: push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.



Call And Return

The **call** instruction pushes the address of the instruction immediately following the **call** instruction onto the stack and sets %rip to point to the beginning of the specified function's instructions.

```
call Label
call *Operand
```

The **ret** instruction pops this instruction address from the stack and stores it in %rip.

```
ret
```

The stored %rip value for a function is called its **return address.** It is the address of the instruction at which to resume the function's execution. (not to be confused with **return value**, which is the value returned from a function).

Registers

What does **call** do?

call pushes the next instruction address onto the stack and points %rip to another function's instructions.

Registers

What does **ret** do?

ret pops off the 8 bytes from the top of the stack and puts it into %rip, thus resuming execution in the caller.

ret is separate from the *return value* of the function (put in %rax).

Function Pointers

The **call** instruction pushes the address of the instruction immediately following the **call** instruction onto the stack and sets %rip to point to the beginning of the specified function's instructions.

```
call Label
call *Operand
```

- Why would we use **call** with a register instead of hardcoding the function name in the assembly? *When would we not know the function to call until we run the code?*
- Function pointers! E.g. qsort – qsort calls a function stored in a parameter register.

Practice: call and ret

In the assembly below, what will the value of %rip be and what value will be stored on top of the stack as a result of the **call** instruction executing?

```
0000000000040112a <main>:
  40112a: 48 83 ec 08          sub    $0x8,%rsp
  40112e: bf 05 00 00 00       mov    $0x5,%edi
  401133: e8 ee ff ff ff       callq  401126 <foo>
  401138: 89 c2                mov    %eax,%edx
  ...
```

Respond on PollEv: pollev.com/cs107
or text CS107 to 22333 once to join.

Practice: call and ret

In the assembly below, what will the value of %rip be and what value will be stored on top of the stack as a result of the **call** instruction executing?

```
000000000040112a <main>:
  40112a: 48 83 ec 08          sub    $0x8,%rsp
  40112e: bf 05 00 00 00       mov    $0x5,%edi
  401133: e8 ee ff ff ff       callq  401126 <foo>
  401138: 89 c2                mov    %eax,%edx
  ...
```

0x401138 stored on the stack (return address)**, 0x401126** put in %rip (address of foo's first instruction)

Calling Functions In Assembly

To call a function in assembly, we must do a few things:

- **Pass Control** – %rip must be adjusted to execute the function being called and then resume the caller function afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.

Lecture Plan

- **Calling Functions**
  - The Stack
  - Passing Control
  - **Passing Data**
  - Local Storage
- Register Restrictions

```
cp -r /afs/ir/class/cs107/lecture-code/lect20 .
```

Parameters and Return

- There are special registers that store parameters and the return value.
- To call a function, we must put any parameters we are passing into the correct registers. (%rdi, %rsi, %rdx, %rcx, %r8, %r9, in that order)
- Parameters beyond the first 6 are put on the stack.
- If the caller expects a return value, it looks in %rax after the callee completes.

Lecture Plan

- **Calling Functions**
  - The Stack
  - Passing Control
  - Passing Data
  - **Local Storage**
- Register Restrictions

```
cp -r /afs/ir/class/cs107/lecture-code/lect20 .
```

Calling Functions In Assembly

To call a function in assembly, we must do a few things:

- **Pass Control** – %rip must be adjusted to execute the function being called and then resume the caller function afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

Terminology: **caller** function calls the **callee** function.

Local Storage

- So far, we've often seen local variables stored directly in registers, rather than on the stack as we'd expect. This is for optimization reasons.
- There are **three** common reasons that local data must be in memory:
  - We've run out of registers
  - The '&' operator is used on it, so we must generate an address for it
  - They are arrays or structs (need to use address arithmetic)

Local Storage

```
long caller() {
    long arg1 = 534;
    long arg2 = 1057;
    long sum = swap_add(&arg1, &arg2);
    ...
}
```

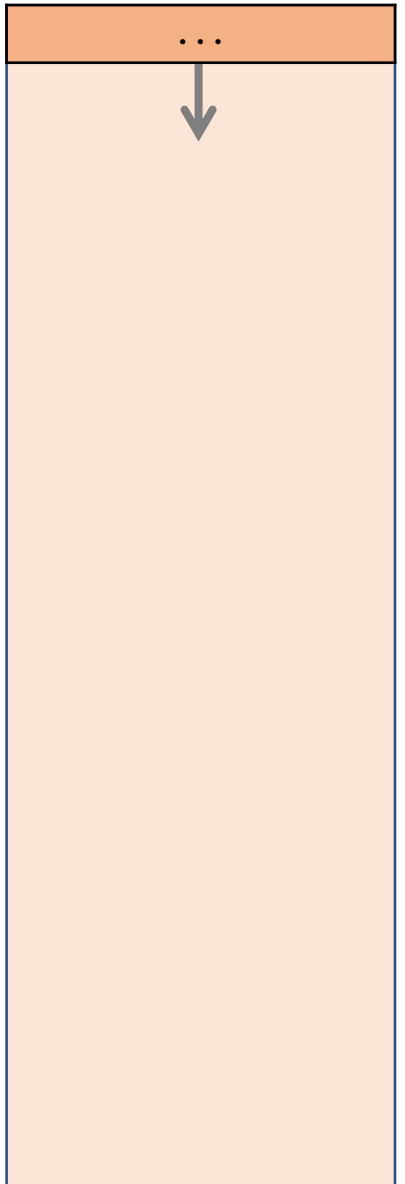
```
caller:
    sub    $0x10, %rsp           // 16 bytes for stack frame
    movq   $0x216, 0x8(%rsp)     // store 534 in arg1
    movq   $0x421, (%rsp)        // store 1057 in arg2
    mov    %rsp, %rsi            // compute &arg2 as second arg
    lea    0x8(%rsp), %rdi       // compute &arg1 as first arg
    callq swap_add               // call swap_add(&arg1, &arg2)
```

Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

main()



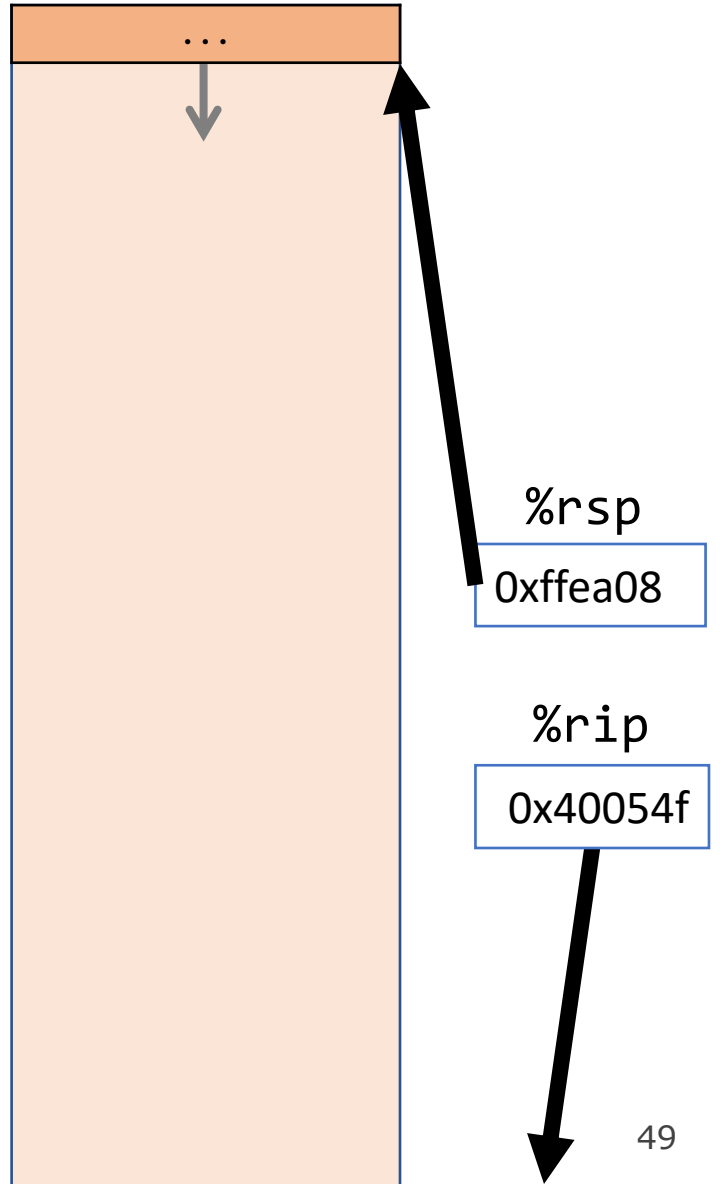
Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
             int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40054f <+0>:    sub    $0x18,%rsp
0x400553 <+4>:    movl   $0x1,0xc(%rsp)
0x40055b <+12>:   movl   $0x2,0x8(%rsp)
0x400563 <+20>:   movl   $0x3,0x4(%rsp)
0x40056b <+28>:   movl   $0x4,(%rsp)
```

main() ⊂

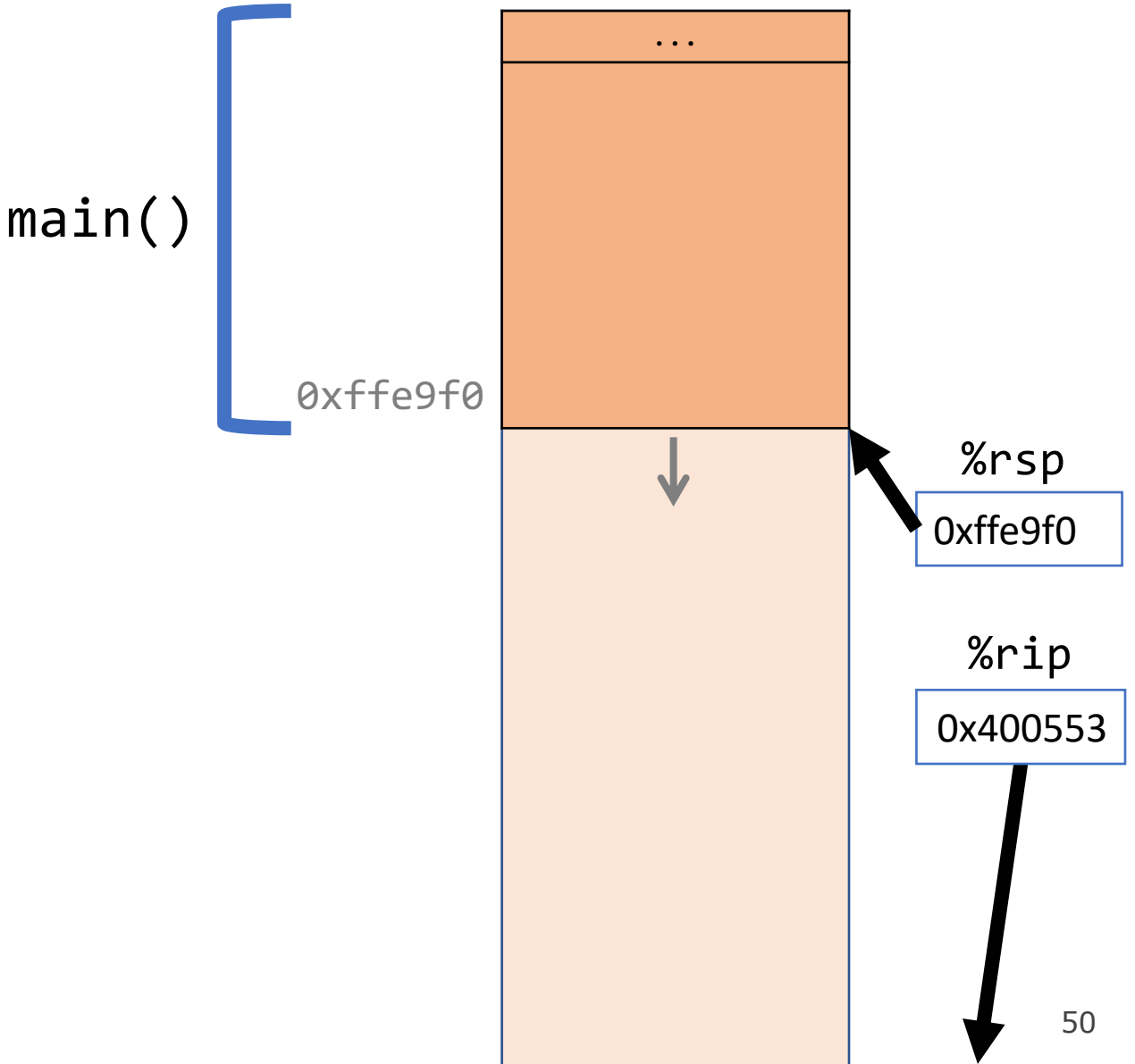


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40054f <+0>:    sub    $0x18,%rsp
0x400553 <+4>:    movl   $0x1,0xc(%rsp)
0x40055b <+12>:   movl   $0x2,0x8(%rsp)
0x400563 <+20>:   movl   $0x3,0x4(%rsp)
0x40056b <+28>:   movl   $0x4,(%rsp)
```

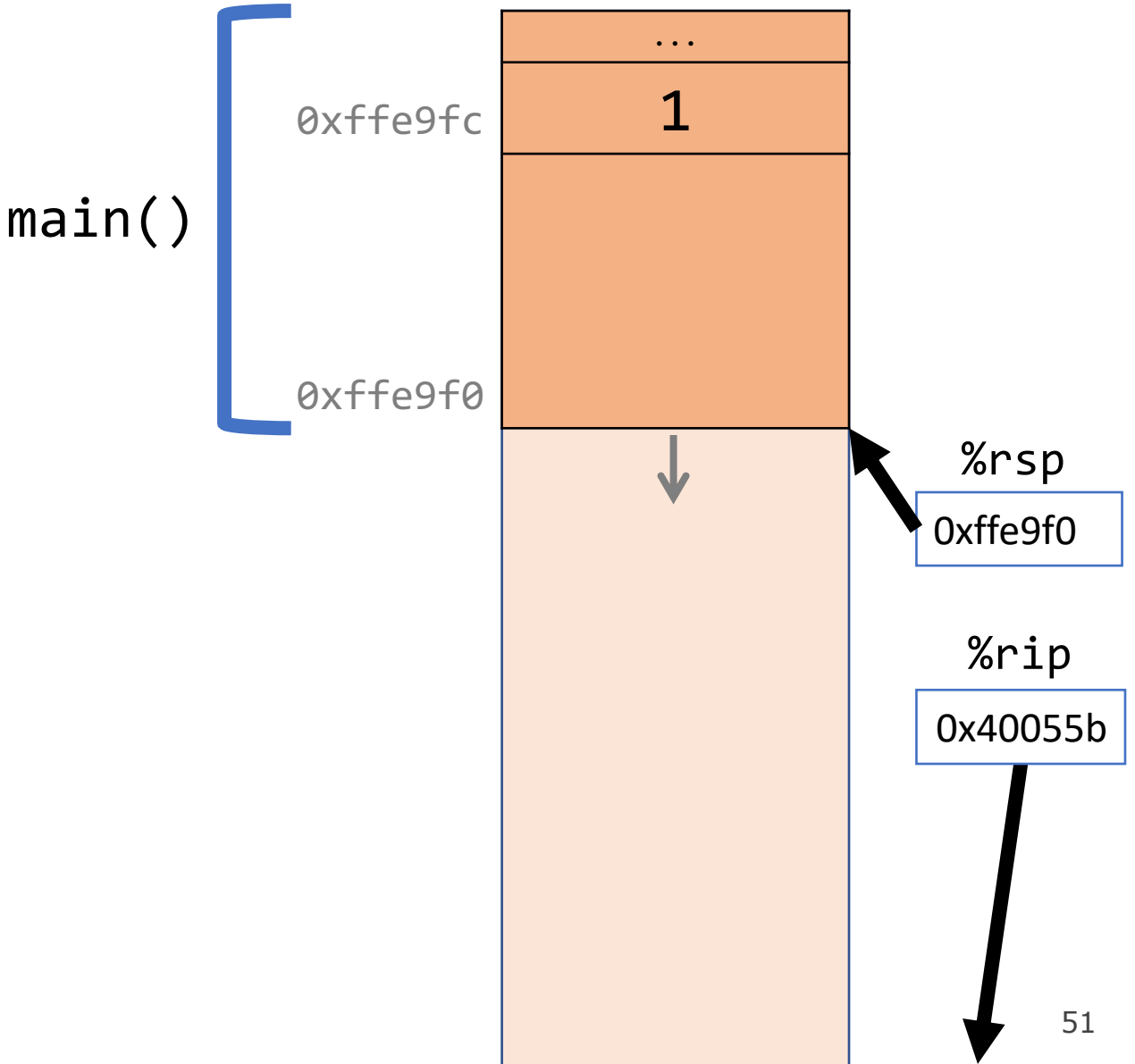


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40054f <+0>:    sub    $0x18,%rsp
0x400553 <+4>:    movl   $0x1,0xc(%rsp)
0x40055b <+12>:   movl   $0x2,0x8(%rsp)
0x400563 <+20>:   movl   $0x3,0x4(%rsp)
0x40056b <+28>:   movl   $0x4,(%rsp)
```

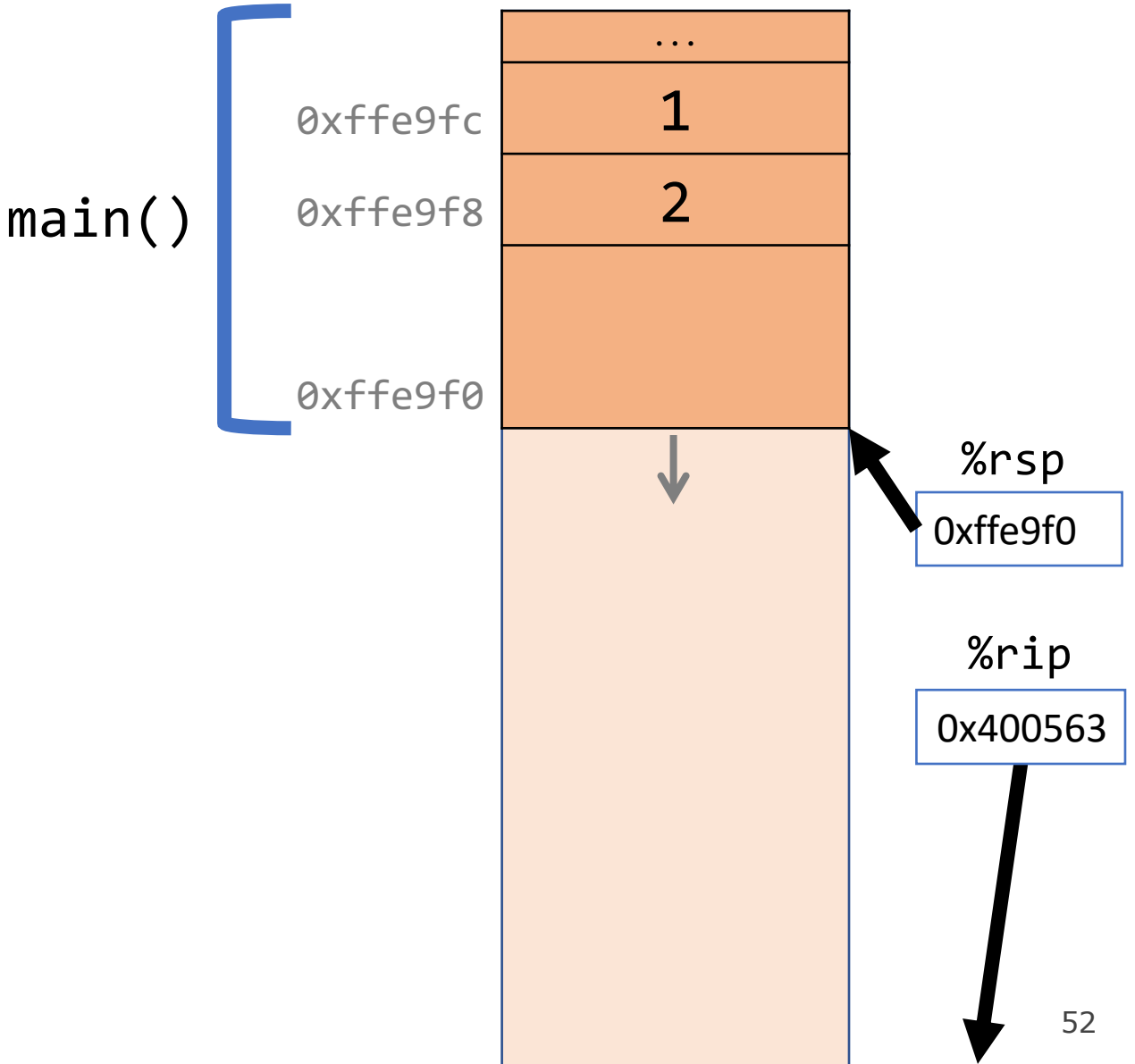


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40054f <+0>:    sub    $0x18,%rsp
0x400553 <+4>:    movl   $0x1,0xc(%rsp)
0x40055b <+12>:   movl   $0x2,0x8(%rsp)
0x400563 <+20>:   movl   $0x3,0x4(%rsp)
0x40056b <+28>:   movl   $0x4,(%rsp)
```

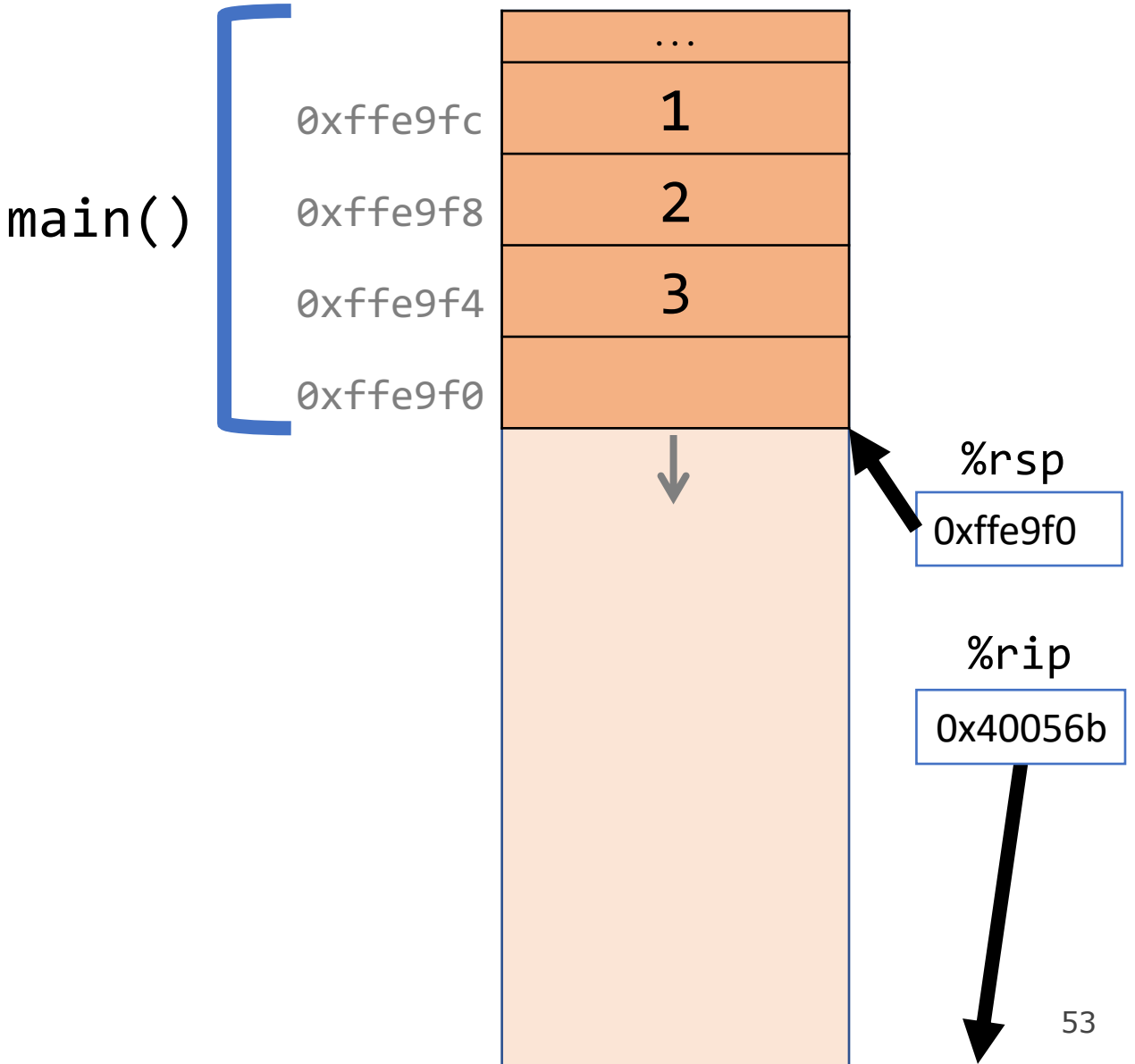


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400553 <+4>:    movl   $0x1,0xc(%rsp)
0x40055b <+12>:   movl   $0x2,0x8(%rsp)
0x400563 <+20>:   movl   $0x3,0x4(%rsp)
0x40056b <+28>:   movl   $0x4,(%rsp)
0x400572 <+35>:   pushq  $0x4
```

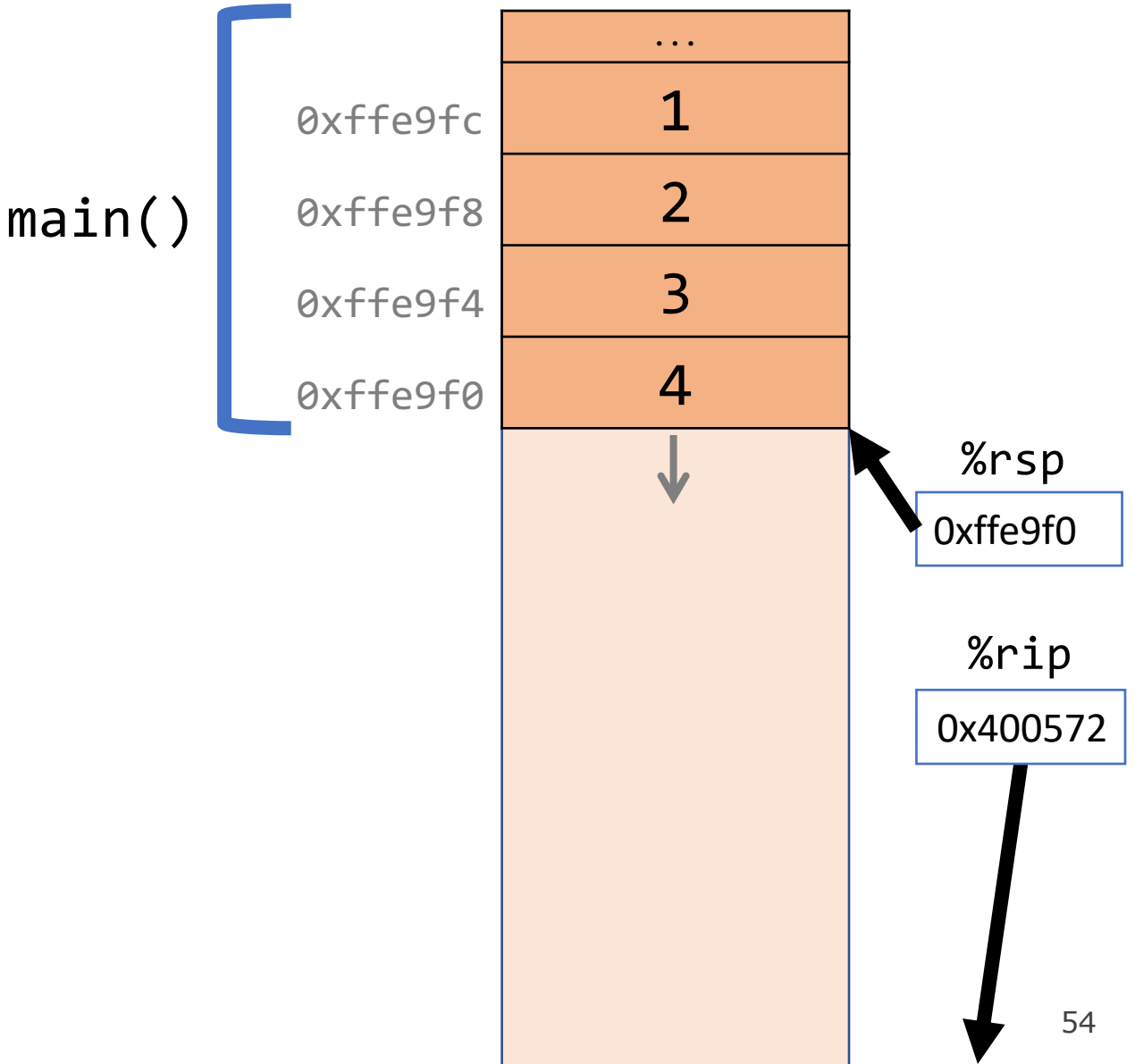


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40055b <+12>:    movl   $0x2,0x8(%rsp)
0x400563 <+20>:    movl   $0x3,0x4(%rsp)
0x40056b <+28>:    movl   $0x4,(%rsp)
0x400572 <+35>:    pushq  $0x4
0x400574 <+37>:    pushq  $0x3
```

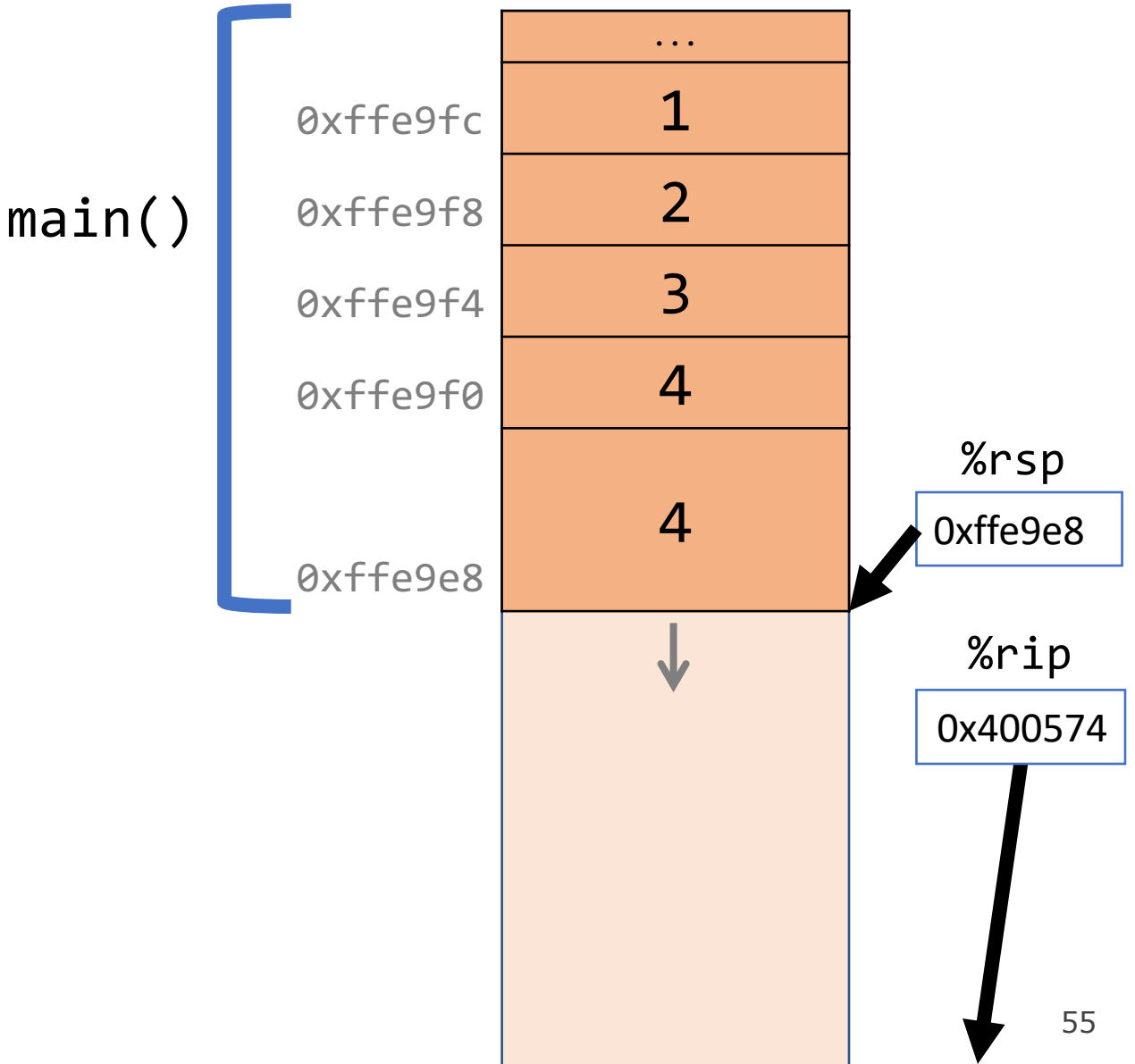


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400563 <+20>:    movl   $0x3,0x4(%rsp)
0x40056b <+28>:    movl   $0x4,(%rsp)
0x400572 <+35>:    pushq  $0x4
0x400574 <+37>:    pushq  $0x3
0x400576 <+39>:    mov    $0x2,%r9d
```

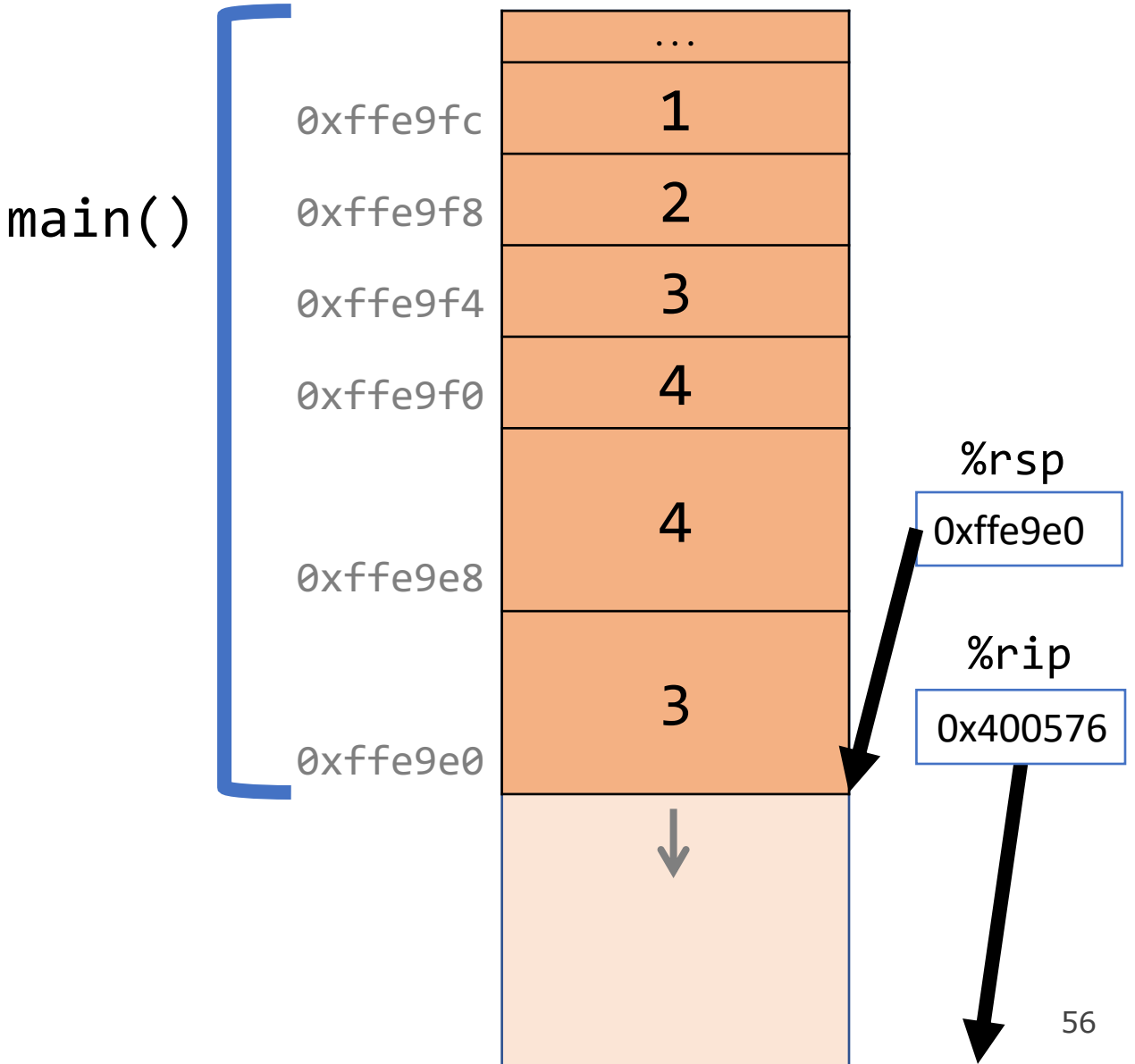


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40056b <+28>:   movl   $0x4,(%rsp)
0x400572 <+35>:   pushq  $0x4
0x400574 <+37>:   pushq  $0x3
0x400576 <+39>:   mov    $0x2,%r9d
0x40057c <+45>:   mov    $0x1,%r8d
```

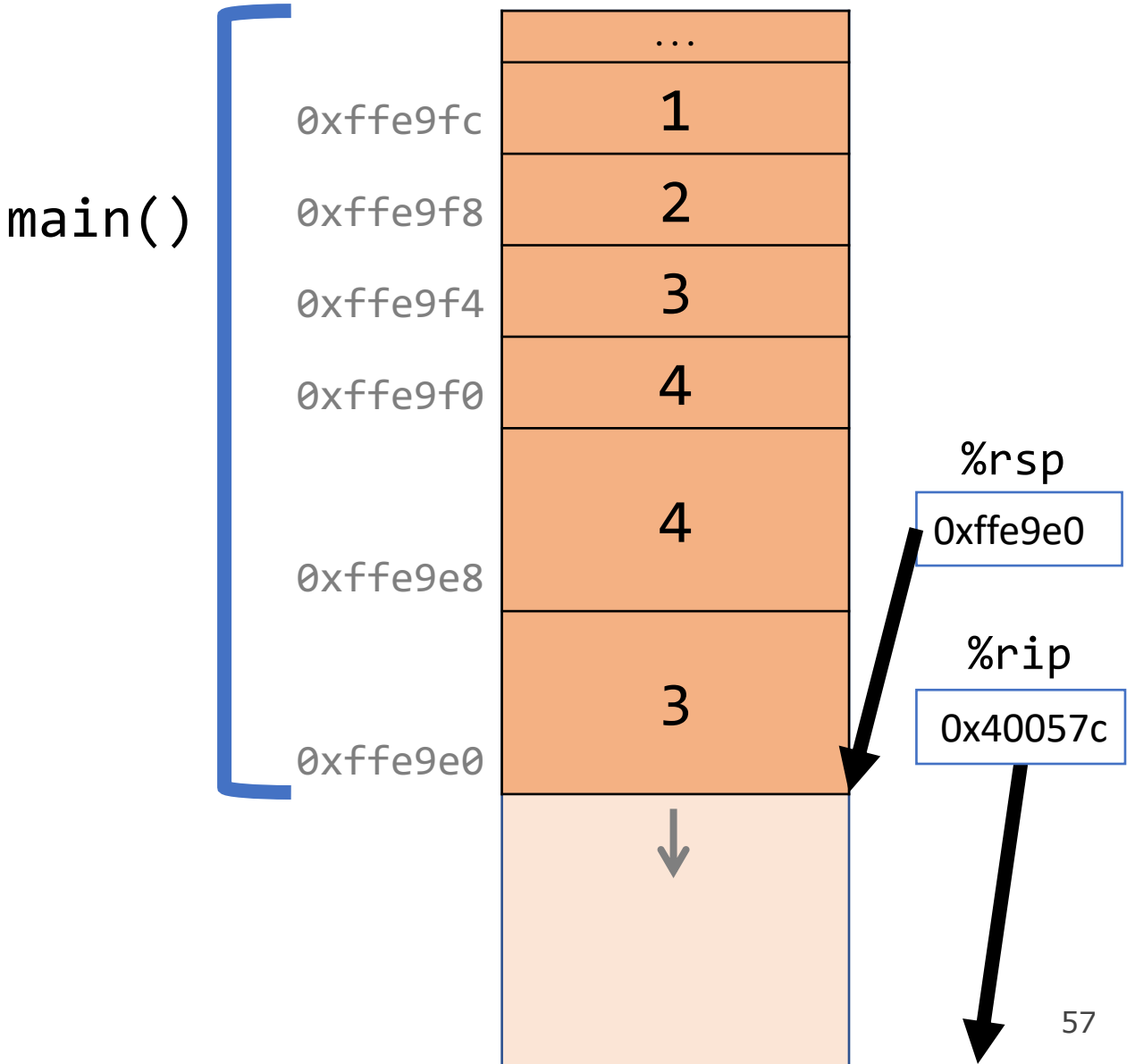


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400572 <+35>:   pushq  $0x4
0x400574 <+37>:   pushq  $0x3
0x400576 <+39>:   mov    $0x2,%r9d
0x40057c <+45>:   mov    $0x1,%r8d
0x400582 <+51>:   lea    0x10(%rsp),%rcx
```

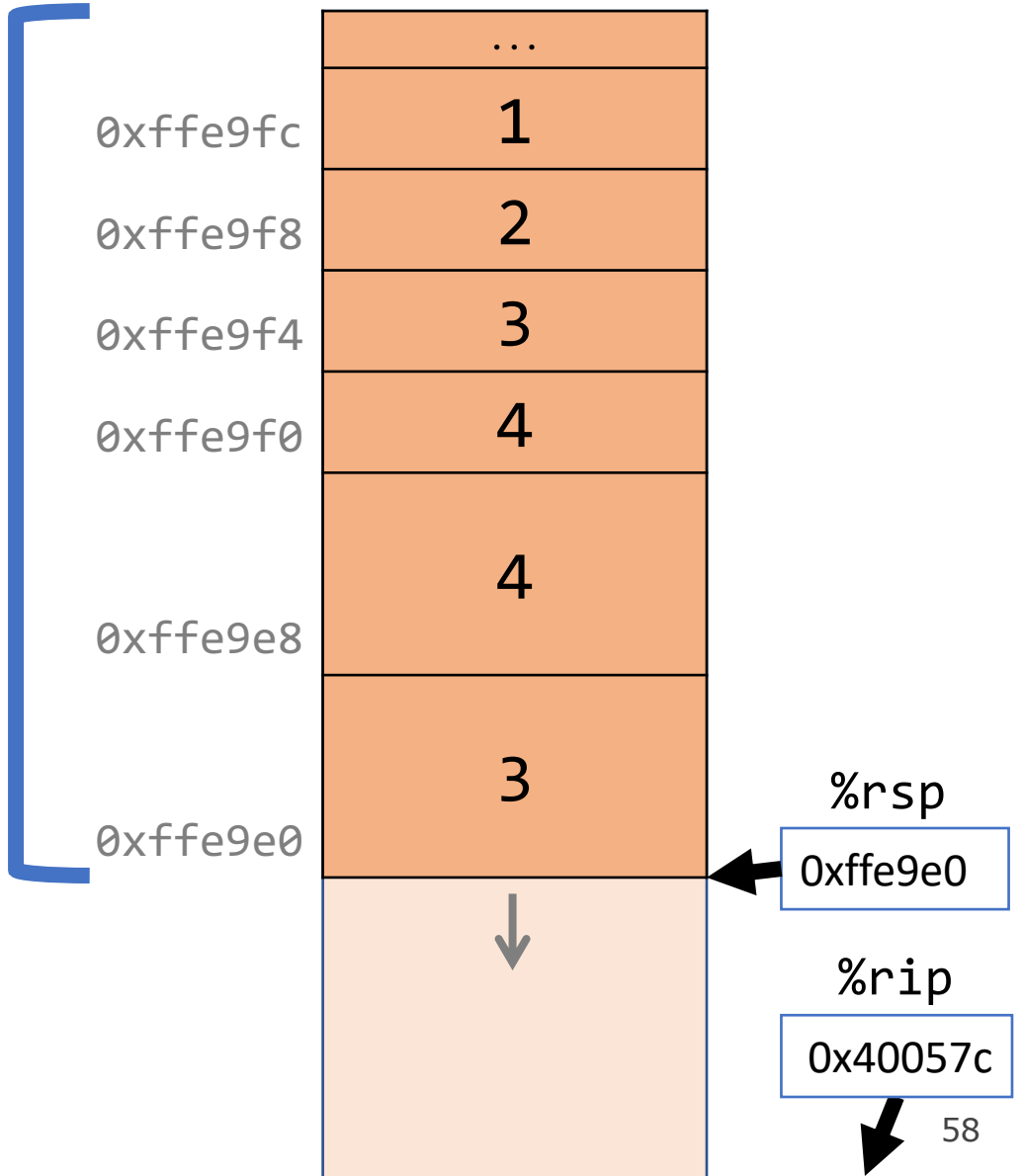


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400572 <+35>:    pushq   $0x4
0x400574 <+37>:    pushq   $0x3
0x400576 <+39>:    mov     $0x2,%r9d
0x40057c <+45>:    mov     $0x1,%r8d
0x400582 <+51>:    lea     0x10(%rsp),%rcx
```



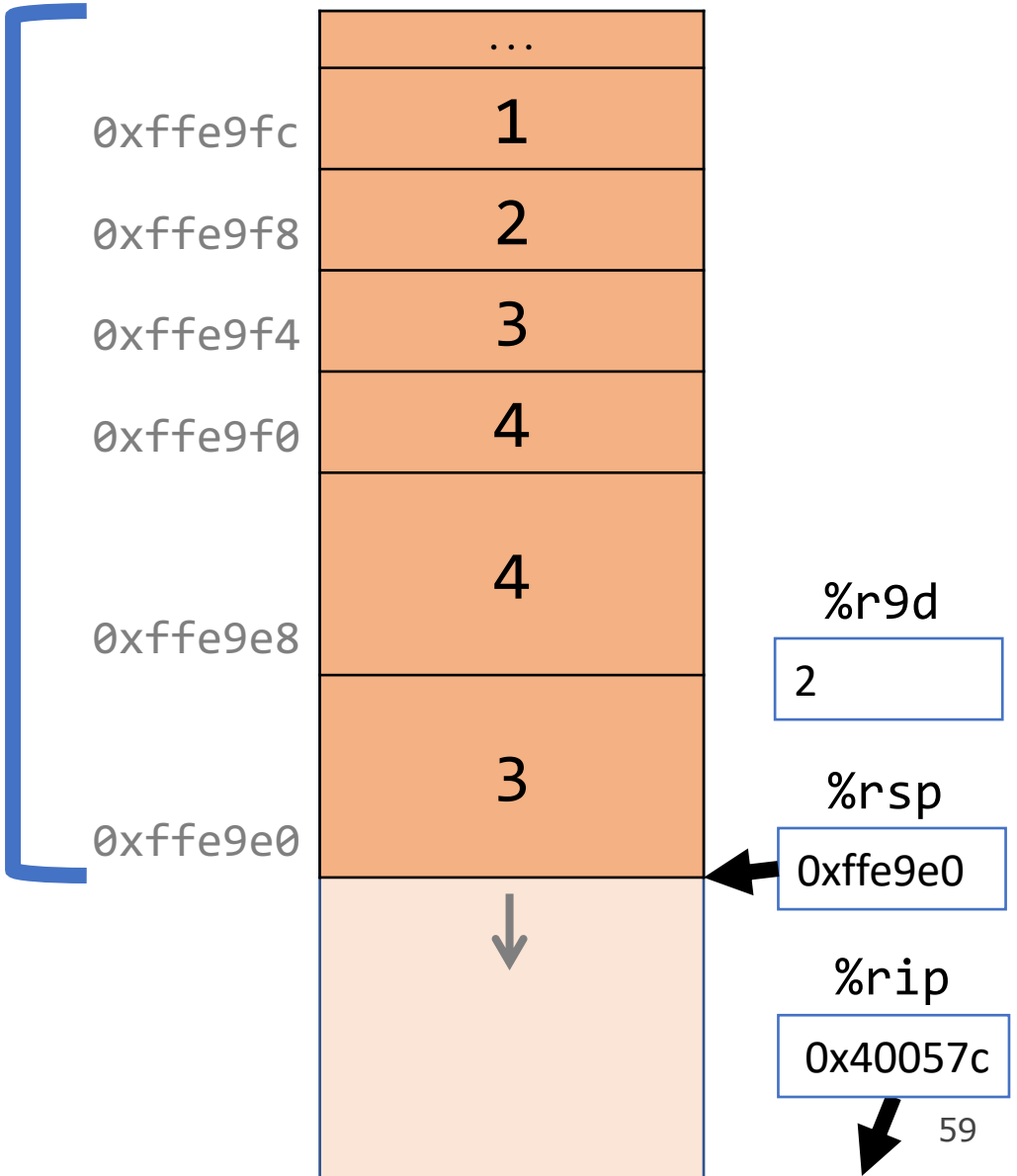
Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400572 <+35>:    pushq   $0x4
0x400574 <+37>:    pushq   $0x3
0x400576 <+39>:    mov     $0x2,%r9d
0x40057c <+45>:    mov     $0x1,%r8d
0x400582 <+51>:    lea     0x10(%rsp),%rcx
```

main()

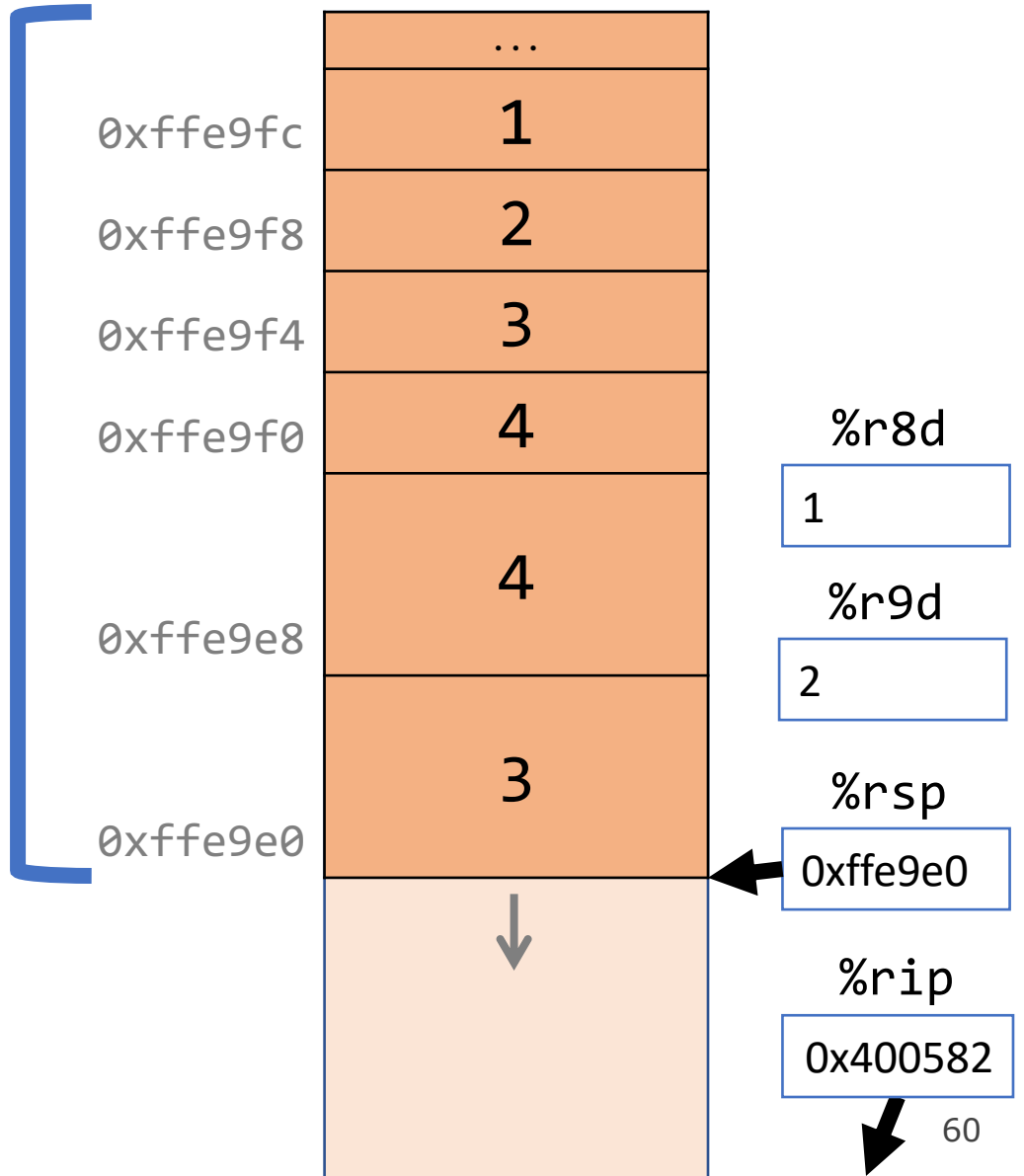


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400574 <+37>:    pushq   $0x3
0x400576 <+39>:    mov     $0x2,%r9d
0x40057c <+45>:    mov     $0x1,%r8d
0x400582 <+51>:    lea     0x10(%rsp),%rcx
0x400587 <+56>:    lea     0x14(%rsp),%rdx
```

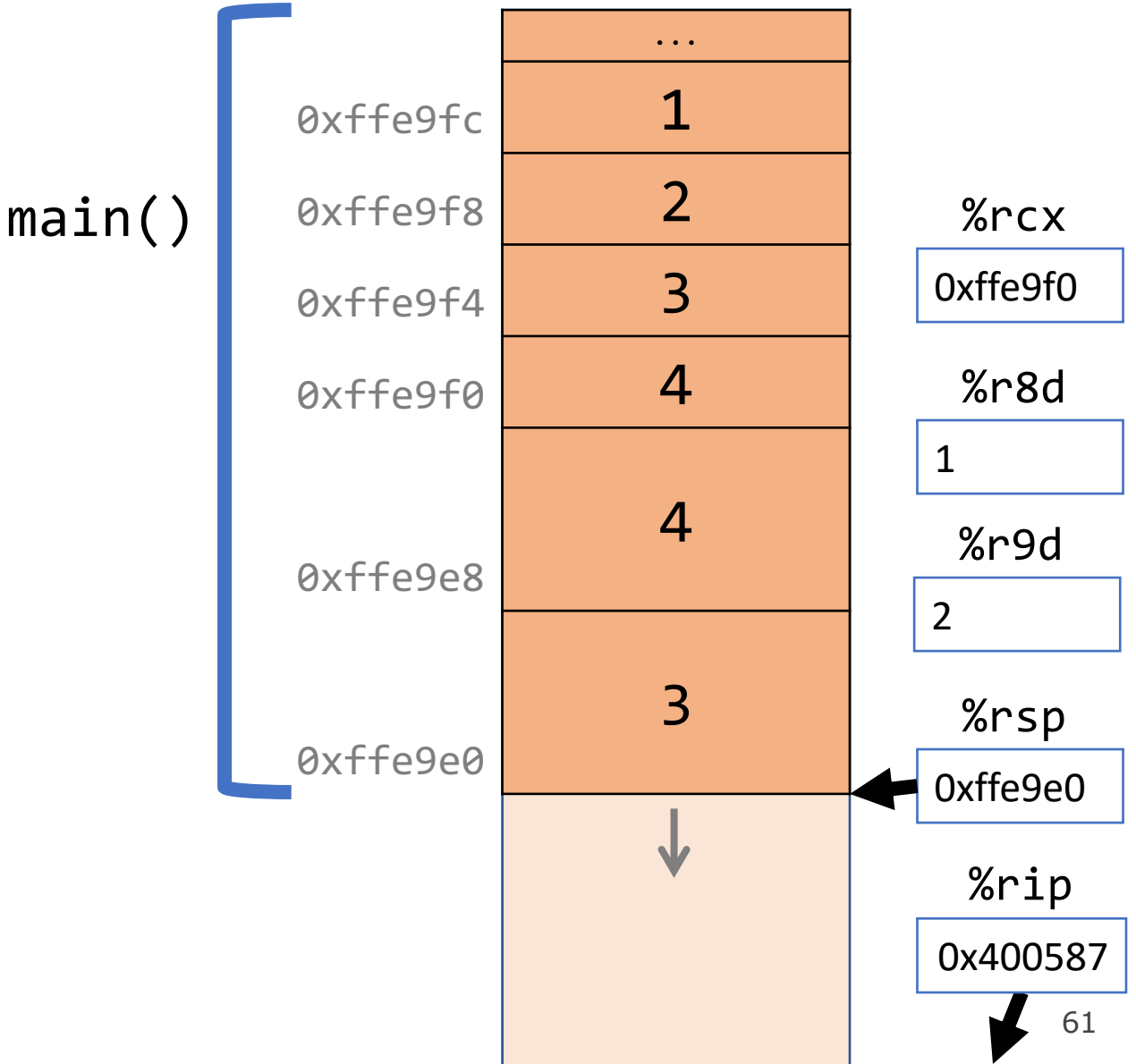


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400576 <+39>:    mov    $0x2,%r9d
0x40057c <+45>:    mov    $0x1,%r8d
0x400582 <+51>:    lea    0x10(%rsp),%rcx
0x400587 <+56>:    lea    0x14(%rsp),%rdx
0x40058c <+61>:    lea    0x18(%rsp),%rsi
```



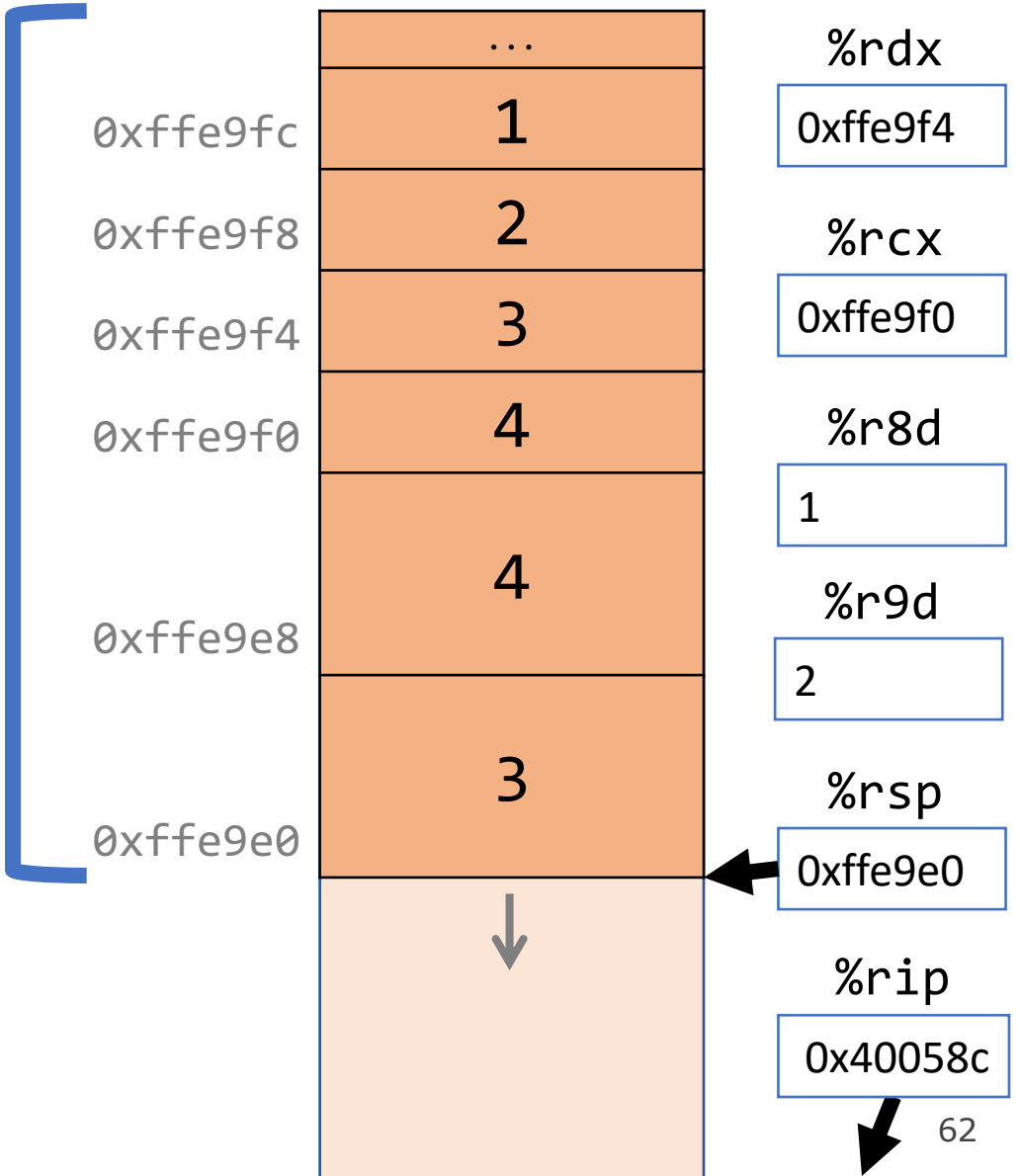
Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40057c <+45>:   mov    $0x1,%r8d
0x400582 <+51>:   lea    0x10(%rsp),%rcx
0x400587 <+56>:   lea    0x14(%rsp),%rdx
0x40058c <+61>:   lea    0x18(%rsp),%rsi
0x400591 <+66>:   lea    0x1c(%rsp),%rdi
```

main()

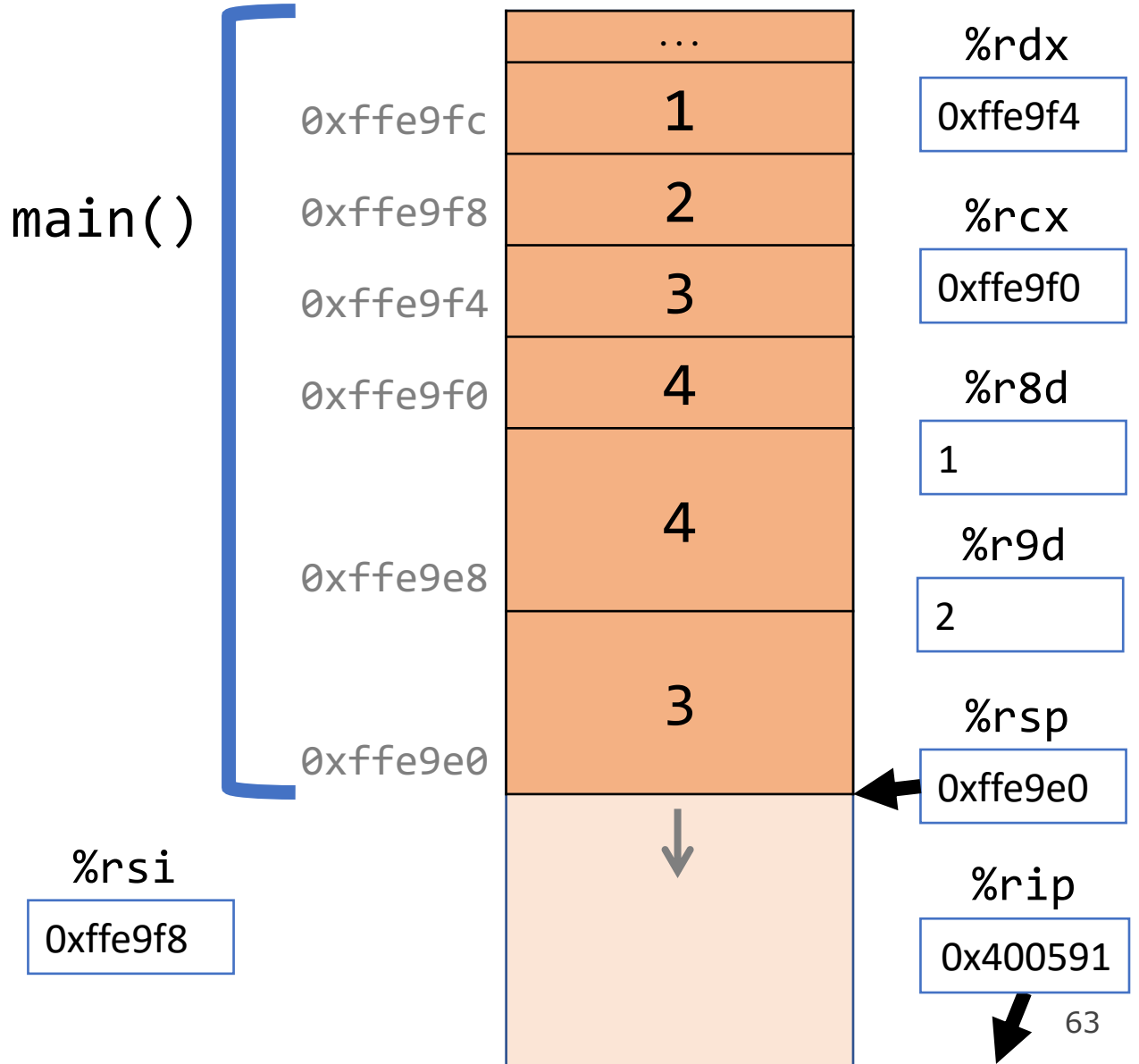


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400582 <+51>:    lea    0x10(%rsp),%rcx
0x400587 <+56>:    lea    0x14(%rsp),%rdx
0x40058c <+61>:    lea    0x18(%rsp),%rsi
0x400591 <+66>:    lea    0x1c(%rsp),%rdi
0x400596 <+71>:    callq  0x401238 <func>
```

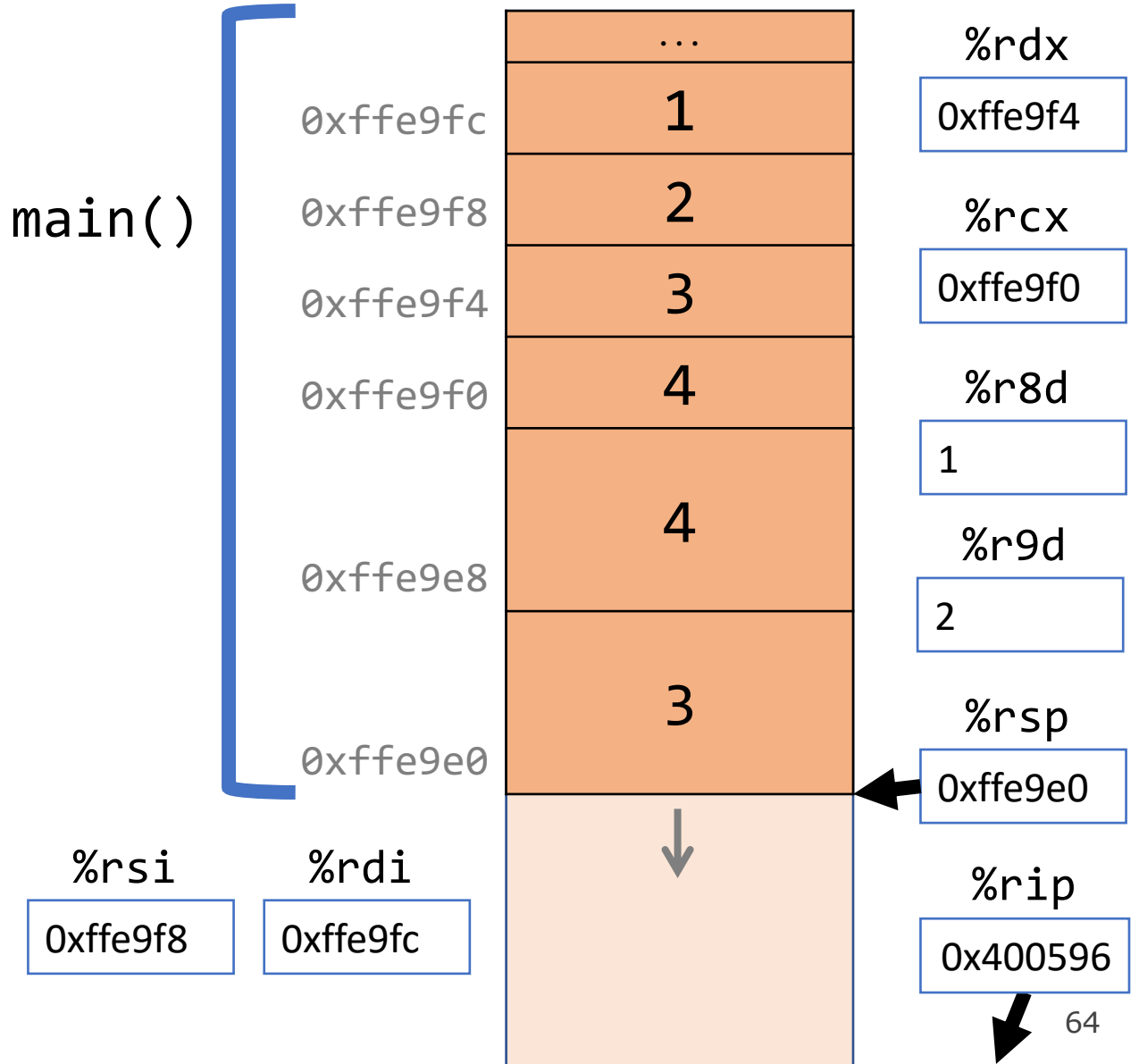


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x400587 <+56>:   lea    0x14(%rsp),%rdx
0x40058c <+61>:   lea    0x18(%rsp),%rsi
0x400591 <+66>:   lea    0x1c(%rsp),%rdi
0x400596 <+71>:   callq  0x401238 <func>
0x40059b <+76>:   add    $0x10,%rsp
```

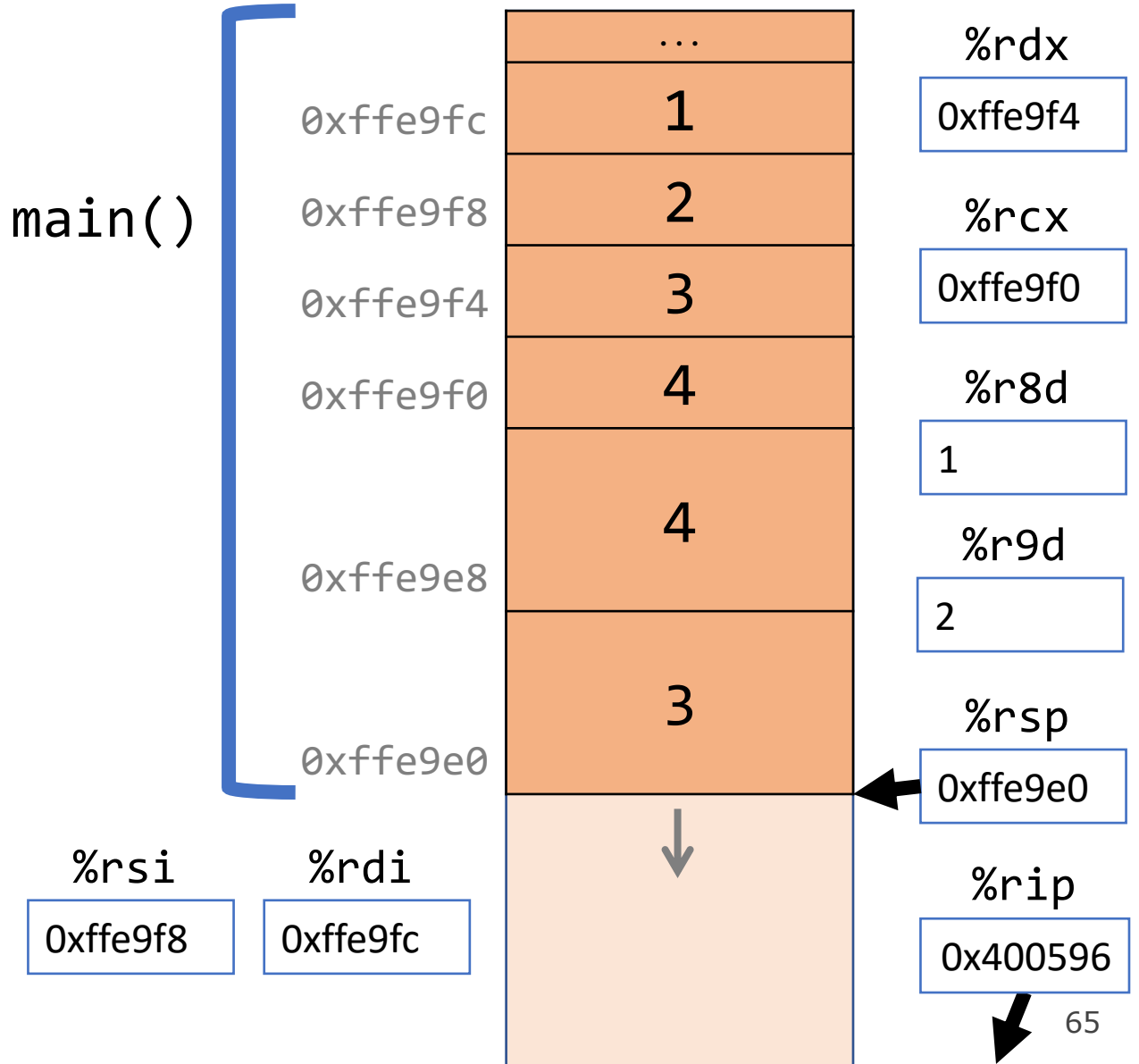


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40058c <+61>:    lea    0x18(%rsp),%rsi
0x400591 <+66>:    lea    0x1c(%rsp),%rdi
0x400596 <+71>:    callq  0x401238 <func>
0x40059b <+76>:    add    $0x10,%rsp
…
```

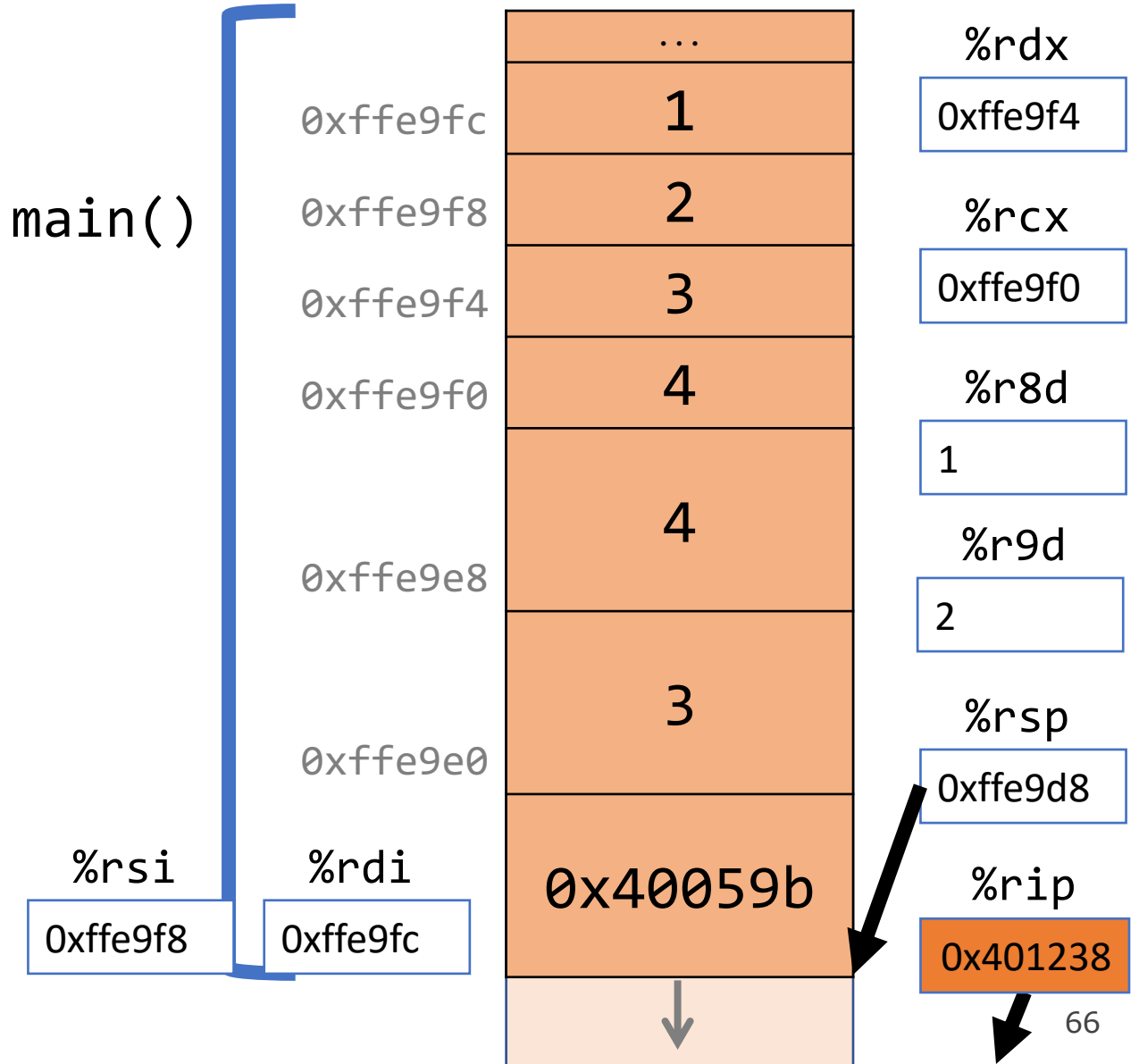


Parameters and Return

```
int main(int argc, char *argv[]) {
    int i1 = 1;
    int i2 = 2;
    int i3 = 3;
    int i4 = 4;
    int result = func(&i1, &i2, &i3, &i4,
                      i1, i2, i3, i4);
    …
}

int func(int *p1, int *p2, int *p3, int *p4,
         int v1, int v2, int v3, int v4) {
    …
}
```

```
0x40058c <+61>:    lea    0x18(%rsp),%rsi
0x400591 <+66>:    lea    0x1c(%rsp),%rdi
0x400596 <+71>:    callq  0x401238 <func>
0x40059b <+76>:    add    $0x10,%rsp
…
```



Lecture Plan

- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage
- **Register Restrictions**

```
cp -r /afs/ir/class/cs107/lecture-code/lect20 .
```

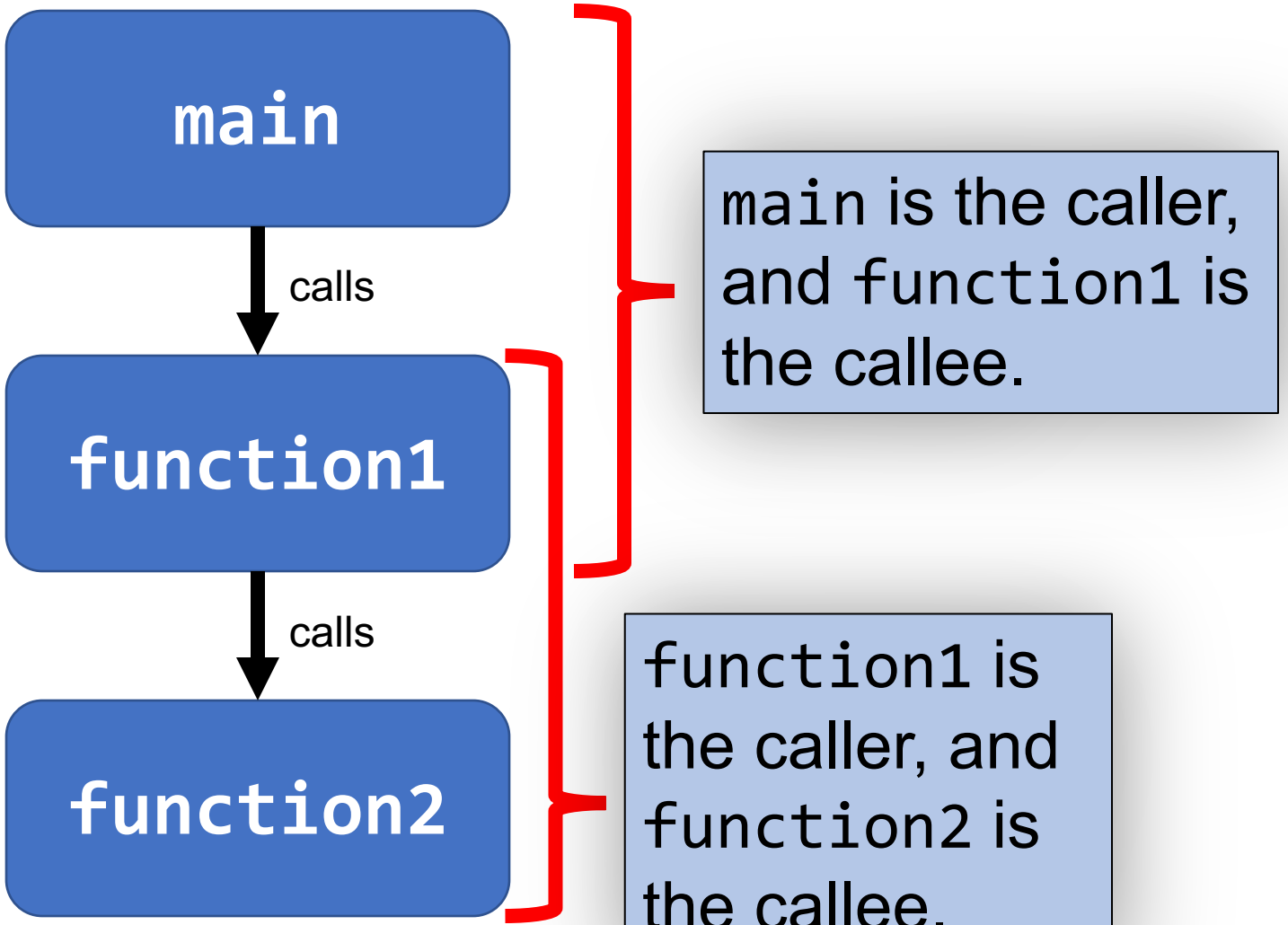
Register Restrictions

There is only one copy of registers for all functions.

- **Problem:** what if ***funcA*** is building up a value in register %r10, and calls ***funcB*** in the middle, which also has instructions that modify %r10? ***funcA***'s value will be overwritten!
- **Solution:** make some "rules of the road" that callers and callees must follow when using registers so they do not interfere with one another.
- These rules define two types of registers: **caller-owned** and **callee-owned**

Caller/Callee

Caller/callee is terminology that refers to a pair of functions. A single function may be both a caller and callee simultaneously (e.g. `function1` at right).



Register Restrictions

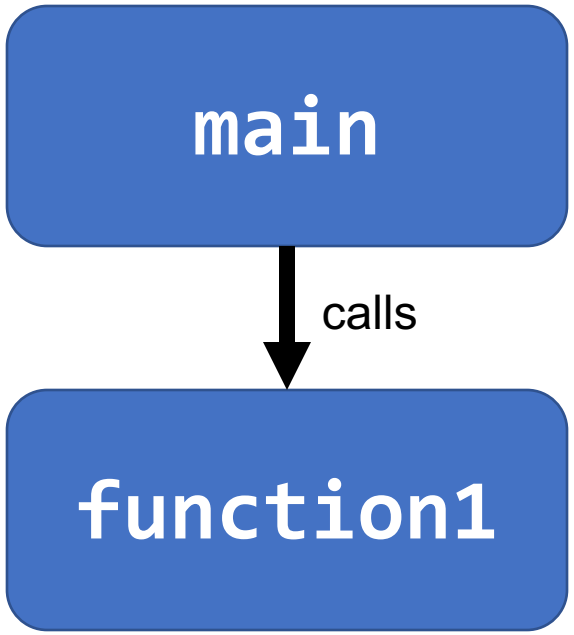
Caller-Owned

- Callee must *save* the existing value and *restore* it when done.
- Caller can store values and assume they will be preserved across function calls.

Callee-Owned

- Callee does not need to save the existing value.
- Caller's values could be overwritten by a callee! The caller may consider saving values elsewhere before calling functions.

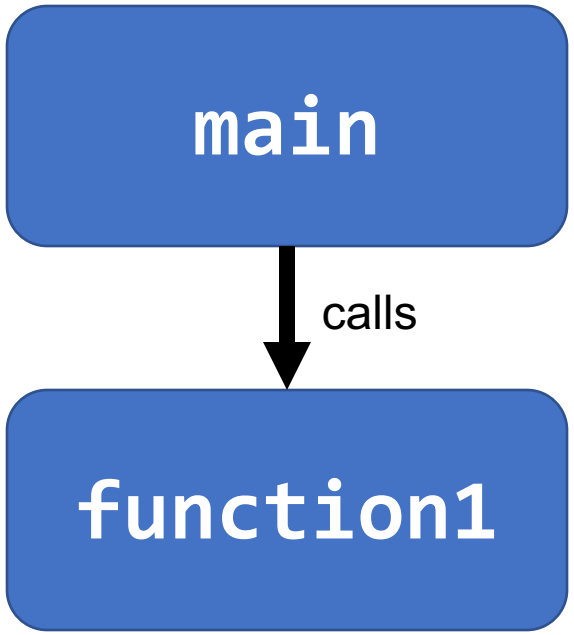
Caller-Owned Registers



`main` can use caller-owned registers and know that `function1` will not permanently modify their values.

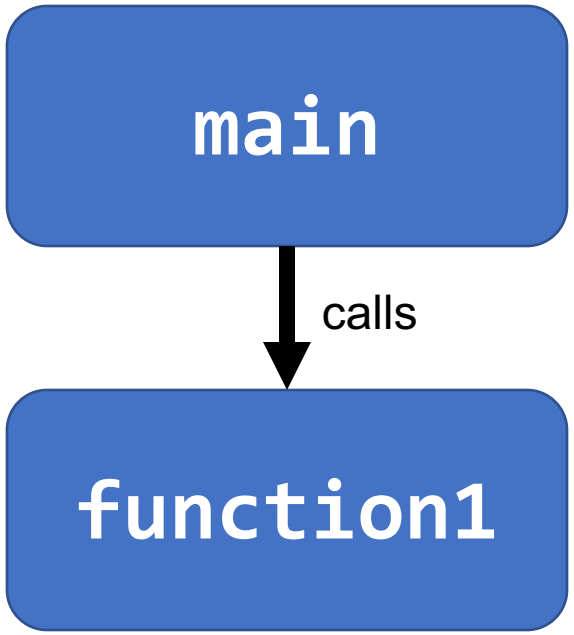
If function1 wants to use any caller-owned registers, it must save the existing values and restore them before returning.

Caller-Owned Registers




```
function1:
  push %rbp
  push %rbx
  ...
  pop %rbx
  pop %rbp
  retq
```

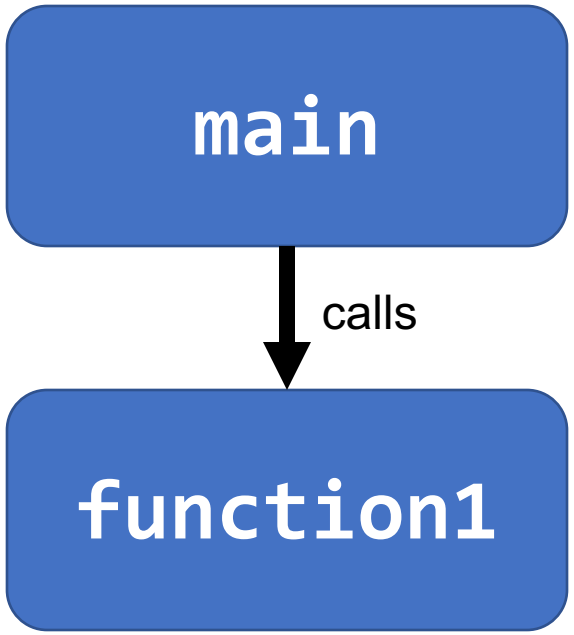
Callee-Owned Registers



`main` can use callee-owned registers but calling `function1` may permanently modify their values.

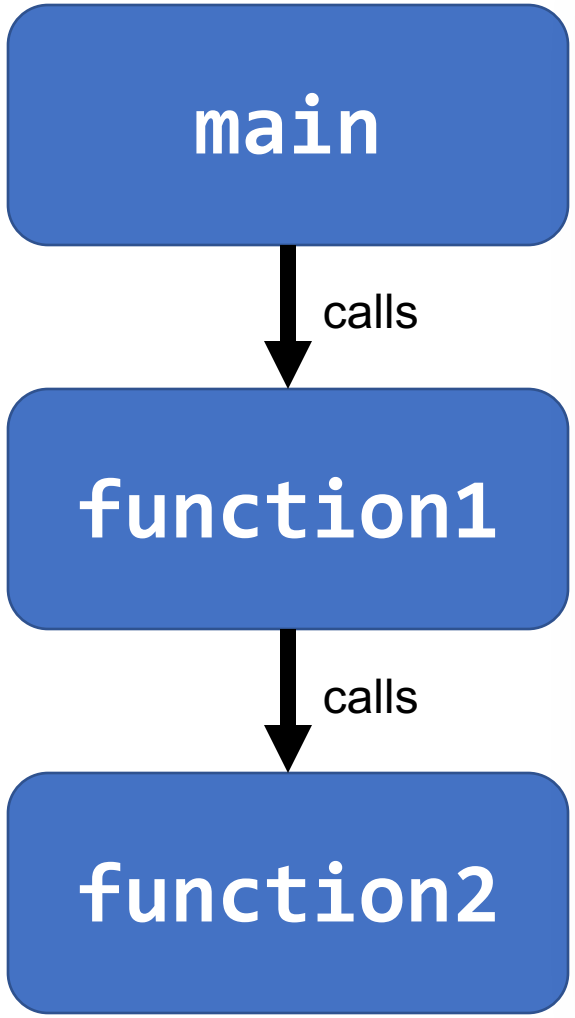
If function1 wants to use any callee-owned registers, it can do so without saving the existing values.

Callee-Owned Registers




```
main:
   ...
   push %r10
   push %r11
   callq function1
   pop %r11
   pop %r10
   ...
```

A Day In the Life of `function1`



Caller-owned registers:

- **`function1`** must save/restore existing values of any it wants to use.
- **`function1`** can assume that calling **`function2`** will not permanently change their values.

Callee-owned registers:

- **`function1`** does not need to save/restore existing values of any it wants to use.
- calling **`function2`** may permanently change their values.

Our First Assembly

```
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}
```

We're done with all our assembly lectures! Now we can fully understand what's going on in the assembly below, including how someone would call sum_array in assembly and what the **ret** instruction does.

```
0000000000401136 <sum_array>:
  401136 <+0>:  mov    $0x0,%eax
  40113b <+5>:  mov    $0x0,%edx
  401140 <+10>: cmp    %esi,%eax
  401142 <+12>: jge    0x40114f <sum_array+25>
  401144 <+14>: movslq %eax,%rcx
  401147 <+17>: add    (%rdi,%rcx,4),%edx
  40114a <+20>: add    $0x1,%eax
  40114d <+23>: jmp    0x401140 <sum_array+10>
  40114f <+25>: mov    %edx,%eax
  401151 <+27>: retq
```

Recap

- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage
- Register Restrictions

Lecture 20 takeaway: Function calls rely on the special %rip and %rsp registers to execute another function's instructions and make stack space. We rely on special registers to pass parameters and the return value between functions. And there are caller and callee owned registers to manage use across functions.

Extra Practice

Extra Practice – Escape Room 2

https://godbolt.org/z/8e31fG4r5

`escape_room`

Escape room assembly code

```
000000000000115b <escape_room>:
    115b:  48 83 ec 08           sub    $0x8,%rsp
    115f:  ba 0a 00 00 00        mov    $0xa,%edx
    1164:  be 00 00 00 00        mov    $0x0,%esi
    1169:  e8 d2 fe ff ff        callq  1040 <strtol@plt>
    116e:  48 89 c7              mov    %rax,%rdi
    1171:  e8 d3 ff ff ff        callq  1149 <transform>
    1176:  a8 01                 test   $0x1,%al
    1178:  74 0a                 je     1184 <escape_room+0x29>
    117a:  b8 00 00 00 00        mov    $0x0,%eax
    117f:  48 83 c4 08           add    $0x8,%rsp
    1183:  c3                    retq
    1184:  b8 01 00 00 00        mov    $0x1,%eax
    1189:  eb f4                 jmp    117f <escape_room+0x24>
```

Escape room assembly code

```
0000000000001149 <transform>:
    1149:  8d 04 bd 00 00 00 00  lea    0x0(,%rdi,4),%eax
    1150:  8d 50 01              lea    0x1(%rax),%edx
    1153:  83 fa 32              cmp    $0x32,%edx
    1156:  7f 02                 jg     115a <transform+0x11>
    1158:  89 d0                 mov    %edx,%eax
    115a:  c3                    retq
```