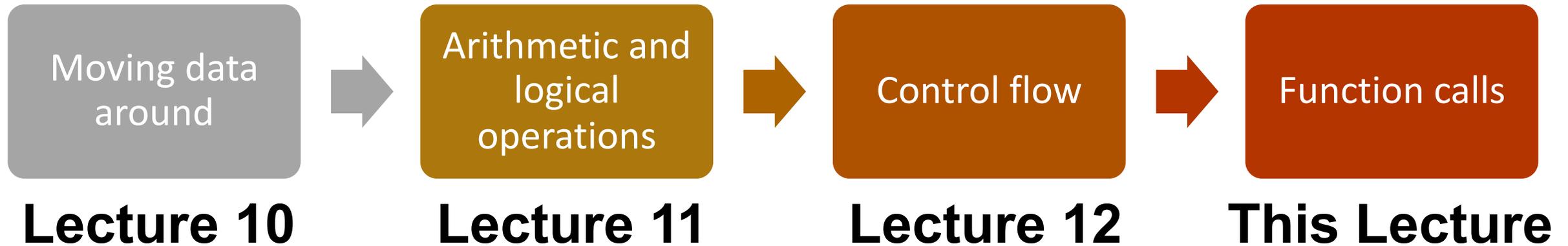


# **CS107, Lecture 13**

## **Assembly: Function Calls and the Runtime Stack**

Reading: B&O 3.7

# Learning Assembly



**Reference Sheet:** [cs107.stanford.edu/resources/x86-64-reference.pdf](https://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

- Revisiting %rip 5
- Calling Functions 19
  - The Stack 22
  - Passing Control 36
  - Passing Data 44
  - Local Storage 65
- Register Restrictions 69
- Pulling it all together: recursion example 78
- Optimizations 81
- Live session slides 93

```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

# Lecture Plan

- **Revisiting %rip** 5
- Calling Functions 19
  - The Stack 22
  - Passing Control 36
  - Passing Data 44
  - Local Storage 65
- Register Restrictions 69
- Pulling it all together: recursion example 78
- Optimizations 81
- Live session slides 93

```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

# %rip

- **%rip** is a special register that points to the next instruction to execute.
- **Let's dive deeper into how %rip works, and how jumps modify it.**

# %rip

```
void loop() {  
    int i = 0;  
    while (i < 100) {  
        i++;  
    }  
}
```

```
0x400570 <+0>:  b8 00 00 00 00  mov $0x0,%eax  
0x400575 <+5>:  eb 03                jmp 0x40057a <loop+10>  
0x400577 <+7>:  83 c0 01            add $0x1,%eax  
0x40057a <+10>: 83 f8 63            cmp $0x63,%eax  
0x40057d <+13>: 73 f8                jle 0x400577 <loop+7>  
0x40057f <+15>: f3 c3                repz retq
```

# %rip

```
void loop() {  
    int i = 0;  
    while (i < 100) {  
        i++;  
    }  
}
```

0x400570	<+0>:	b8 00 00 00 00	mov \$0x0,%eax
0x400575	<+5>:	eb 03	jmp 0x40057a <loop+10>
0x400577	<+7>:	83 c0 01	add \$0x1,%eax
0x40057a	<+10>:	83 f8 63	cmp \$0x63,%eax
0x40057d	<+13>:	73 f8	jle 0x400577 <loop+7>
0x40057f	<+15>:	f3 c3	repz retq

These are 0-based offsets in bytes for each instruction relative to the start of this function.

# %rip

```
void loop() {  
    int i = 0;  
    while (i < 100) {  
        i++;  
    }  
}
```

0x400570 <+0>:	b8 00 00 00 00	mov \$0x0,%eax
0x400575 <+5>:	eb 03	jmp 0x40057a <loop+10>
0x400577 <+7>:	83 c0 01	add \$0x1,%eax
0x40057a <+10>:	83 f8 63	cmp \$0x63,%eax
0x40057d <+13>:	73 f8	jle 0x400577 <loop+7>
0x40057f <+15>:	f3 c3	repz retq

These are bytes for the machine code instructions. Instructions are variable length.

# %rip

```
void loop() {  
    int i = 0;  
    while (i < 100) {  
        i++;  
    }  
}
```

```
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax  
0x400575 <+5>:  eb 03                jmp 0x40057a <loop+10>  
0x400577 <+7>:  83 c0 01            add $0x1,%eax  
0x40057a <+10>: 83 f8 63            cmp $0x63,%eax  
0x40057d <+13>: 73 f8                jle 0x400577 <loop+7>  
0x40057f <+15>: f3 c3                repz retq
```

# %rip

```
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:  eb 03                jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01            add $0x1,%eax
0x40057a <+10>:  83 f8 63            cmp $0x63,%eax
0x40057d <+13>:  73 f8                jle 0x400577 <loop+7>
0x40057f <+15>:  f3 c3                repz retq
```

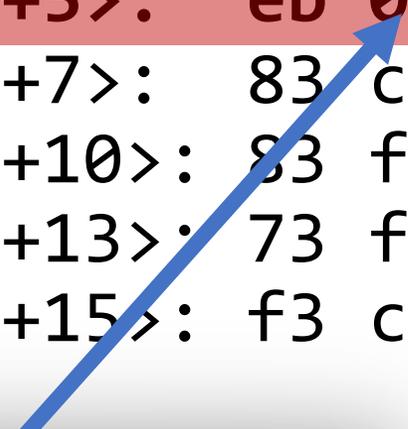
# %rip

```
0x400570 <+0>:  b8 00 00 00 00  mov $0x0,%eax
0x400575 <+5>:  eb 03                jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01            add $0x1,%eax
0x40057a <+10>:  83 f8 63            cmp $0x63,%eax
0x40057d <+13>:  73 f8                jle 0x400577 <loop+7>
0x40057f <+15>:  f3 c3                repz retq
```

**0xeb** means **jmp**.

# %rip

```
0x400570 <+0>:  b8 00 00 00 00 mov $0x0,%eax
0x400575 <+5>:  eb 03                jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01            add $0x1,%eax
0x40057a <+10>:  83 f8 63            cmp $0x63,%eax
0x40057d <+13>:  73 f8                jle 0x400577 <loop+7>
0x40057f <+15>:  f3 c3                repz retq
```

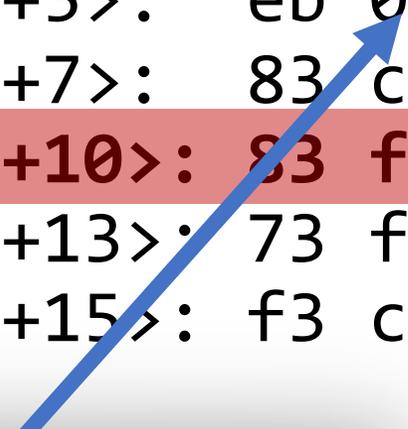


**0x03** is the number of instruction bytes to jump relative to %rip.

With no jump, %rip would advance to the next line. This **jmp** says to then go **3** bytes further!

# %rip

```
0x400570 <+0>:  b8 00 00 00 00  mov $0x0,%eax
0x400575 <+5>:  eb 03                jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01           add $0x1,%eax
0x40057a <+10>:  83 f8 63           cmp $0x63,%eax
0x40057d <+13>:  73 f8                jle 0x400577 <loop+7>
0x40057f <+15>:  f3 c3              repz retq
```



**0x03** is the number of instruction bytes to jump relative to %rip.

With no jump, %rip would advance to the next line. This **jmp** says to then go **3** bytes further!

# %rip

```
0x400570 <+0>:  b8 00 00 00 00  mov $0x0,%eax
0x400575 <+5>:  eb 03                jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01            add $0x1,%eax
0x40057a <+10>: 83 f8 63            cmp $0x63,%eax
0x40057d <+13>: 73 f8            jle 0x400577 <loop+7>
0x40057f <+15>:  f3 c3                repz retq
```

**0x73** means **jle**.

# %rip

```
0x400570 <+0>:  b8 00 00 00 00  mov $0x0,%eax
0x400575 <+5>:  eb 03                jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01            add $0x1,%eax
0x40057a <+10>: 83 f8 63            cmp $0x63,%eax
0x40057d <+13>: 73 f8                jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3                repz retq
```

**0xf8** is the number of instruction bytes to jump relative to %rip. This is -8 (in two's complement!).

With no jump, %rip would advance to the next line. This **jmp** says to then go **8** bytes back!

# %rip

```
0x400570 <+0>:  b8 00 00 00 00  mov $0x0,%eax
0x400575 <+5>:  eb 03                jmp 0x40057a <loop+10>
0x400577 <+7>:  83 c0 01            add $0x1,%eax
0x40057a <+10>: 83 f8 63            cmp $0x63,%eax
0x40057d <+13>: 73 f8                jle 0x400577 <loop+7>
0x40057f <+15>: f3 c3                repz retq
```



**0xf8** is the number of instruction bytes to jump relative to %rip. This is -8 (in two's complement!).

With no jump, %rip would advance to the next line. This **jmp** says to then go **8** bytes back!

# Summary: Instruction Pointer

- Machine code instructions live in main memory, just like stack and heap data.
- `%rip` is a register that stores a number (an address) of the next instruction to execute. It marks our place in the program's instructions.
- To advance to the next instruction, special hardware adds the size of the current instruction in bytes.
- **`jmp`** instructions work by adjusting `%rip` by a specified amount.

# Lecture Plan

- Revisiting %rip 5
- **Calling Functions** 19
  - The Stack 22
  - Passing Control 36
  - Passing Data 44
  - Local Storage 65
- Register Restrictions 69
- Pulling it all together: recursion example 78
- Optimizations 81
- Live session slides 93

```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

**How do we call functions in  
assembly?**

# 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.

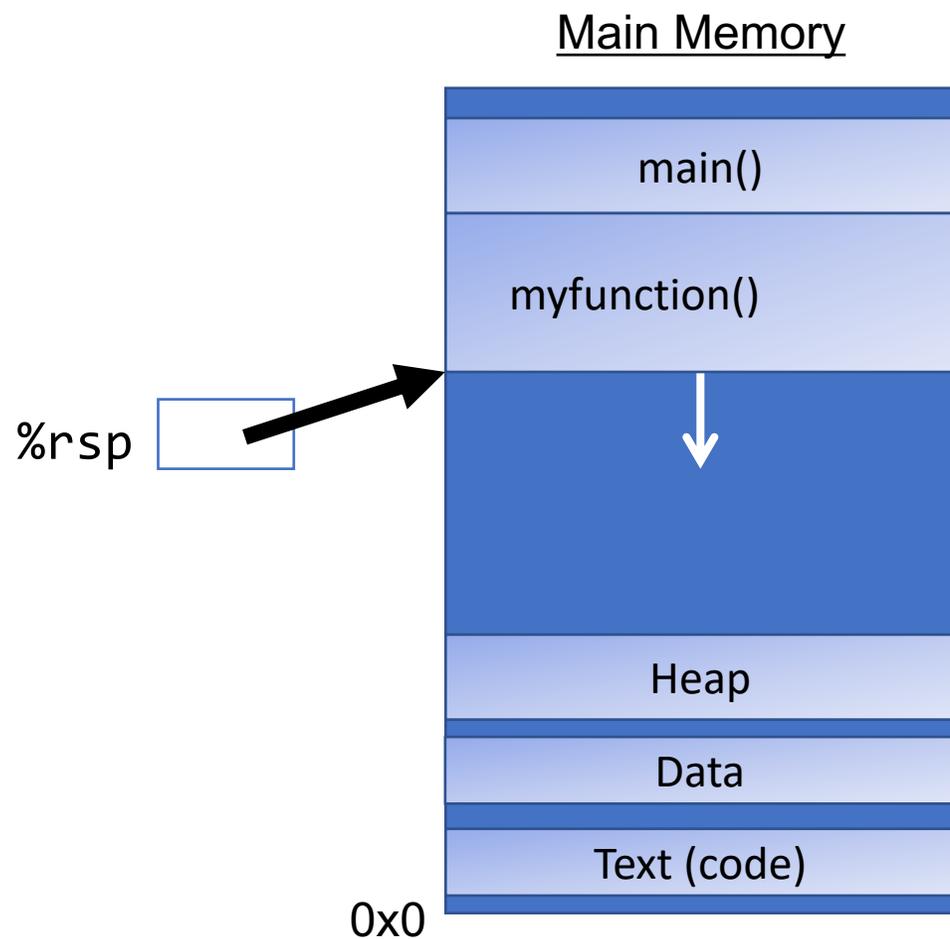
# Lecture Plan

- Revisiting %rip 5
- **Calling Functions** 19
  - **The Stack** 22
    - Passing Control 36
    - Passing Data 44
    - Local Storage 65
- Register Restrictions 69
- Pulling it all together: recursion example 78
- Optimizations 81
- Live session slides 93

```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

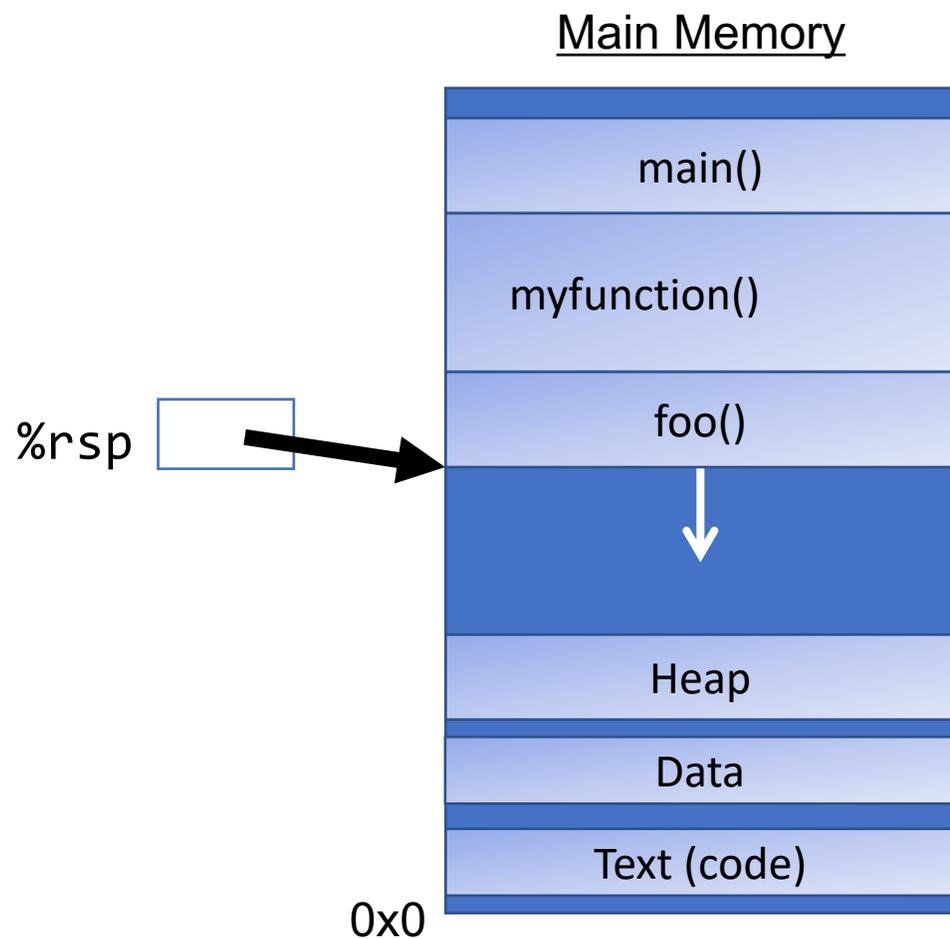
# %rsp

- **%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).



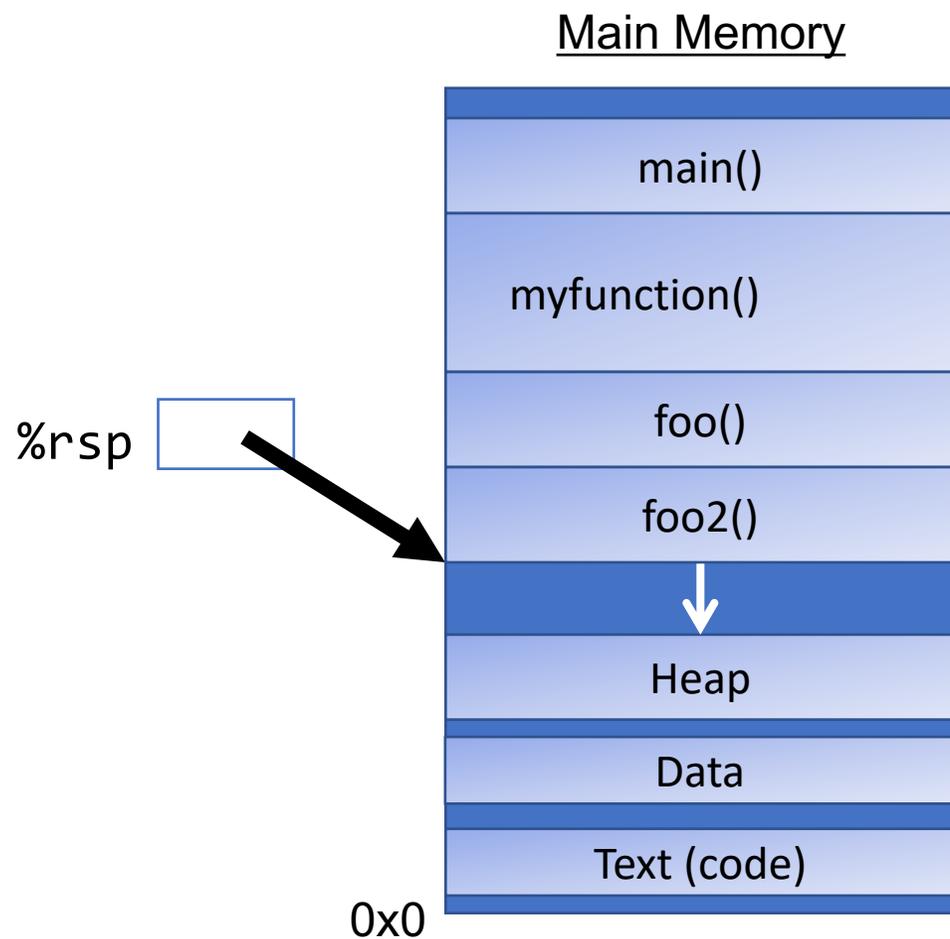
# %rsp

- **%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).



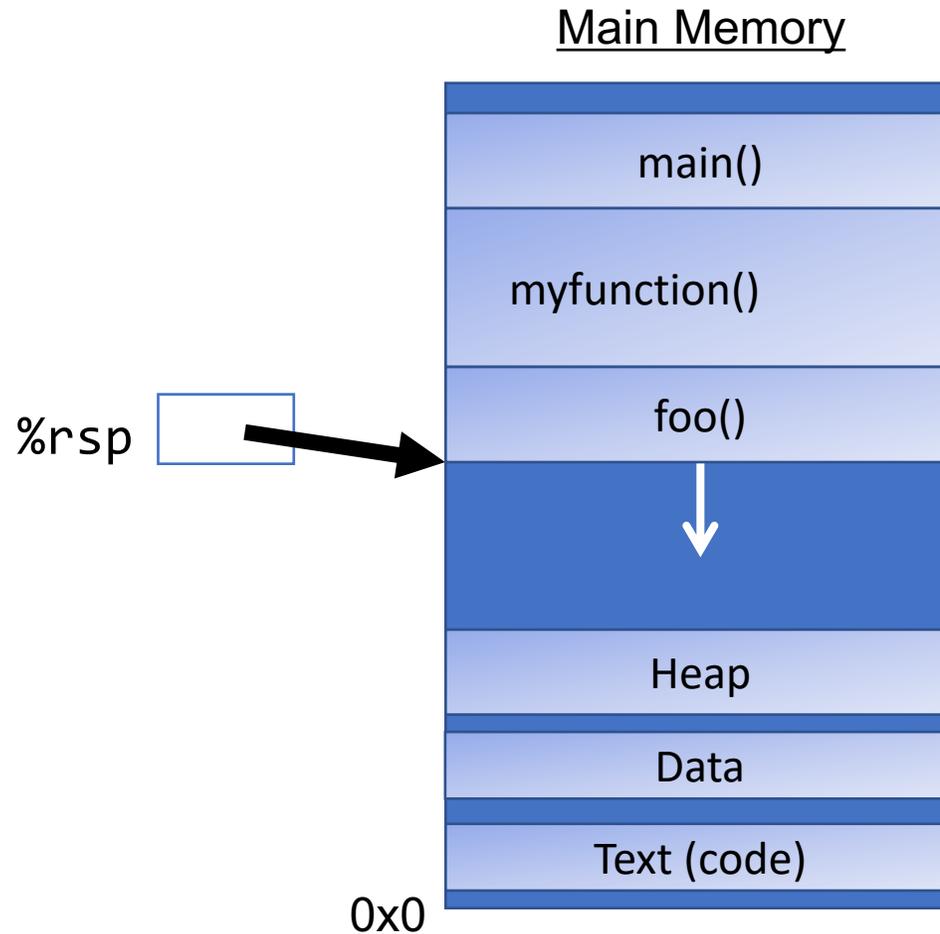
# %rsp

- **%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).



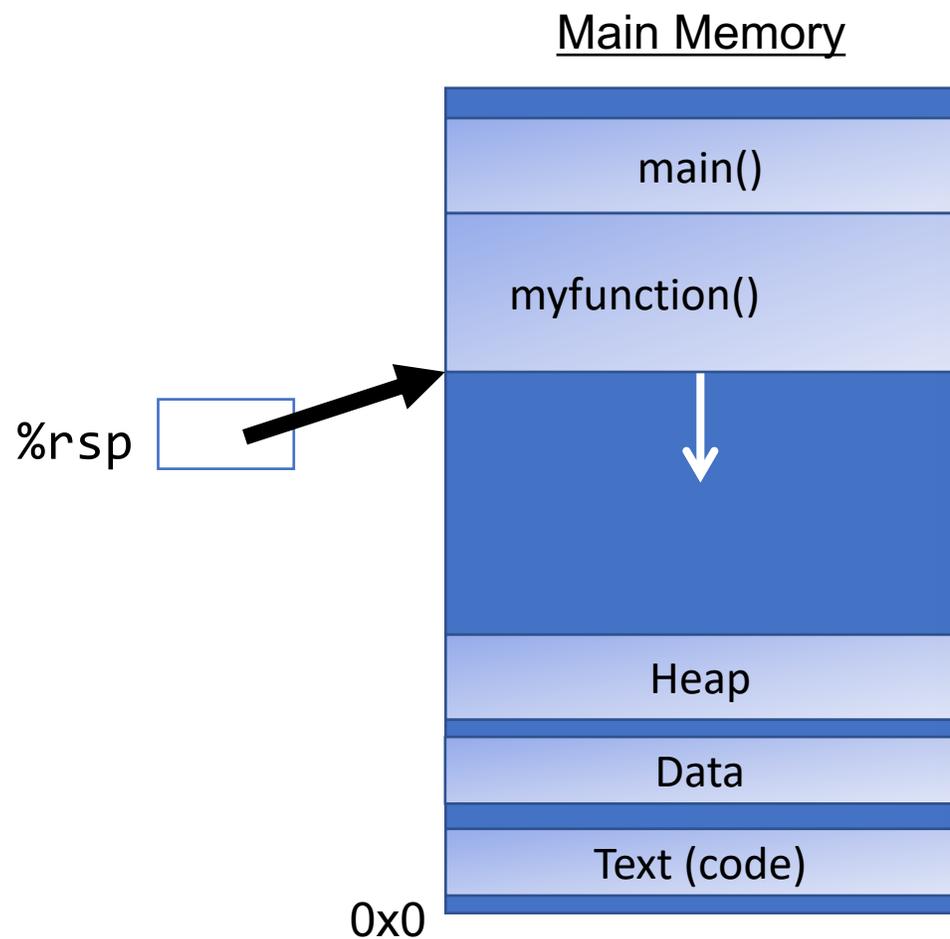
# %rsp

- **%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	$R[\%rsp] \leftarrow R[\%rsp] - 8;$ $M[R[\%rsp]] \leftarrow S$

# push

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

Instruction	Effect
pushq S	$R[\%rsp] \leftarrow R[\%rsp] - 8;$ $M[R[\%rsp]] \leftarrow S$

# push

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

Instruction	Effect
pushq S	$R[\%rsp] \leftarrow R[\%rsp] - 8;$ $M[R[\%rsp]] \leftarrow S$

# push

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

Instruction	Effect
<code>pushq S</code>	$R[\%rsp] \leftarrow R[\%rsp] - 8;$ $M[R[\%rsp]] \leftarrow S$

- 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	$D \leftarrow M[R[\%rsp]]$ $R[\%rsp] \leftarrow R[\%rsp] + 8;$

- **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 <i>D</i>	$D \leftarrow M[R[\%rsp]]$ $R[\%rsp] \leftarrow R[\%rsp] + 8;$

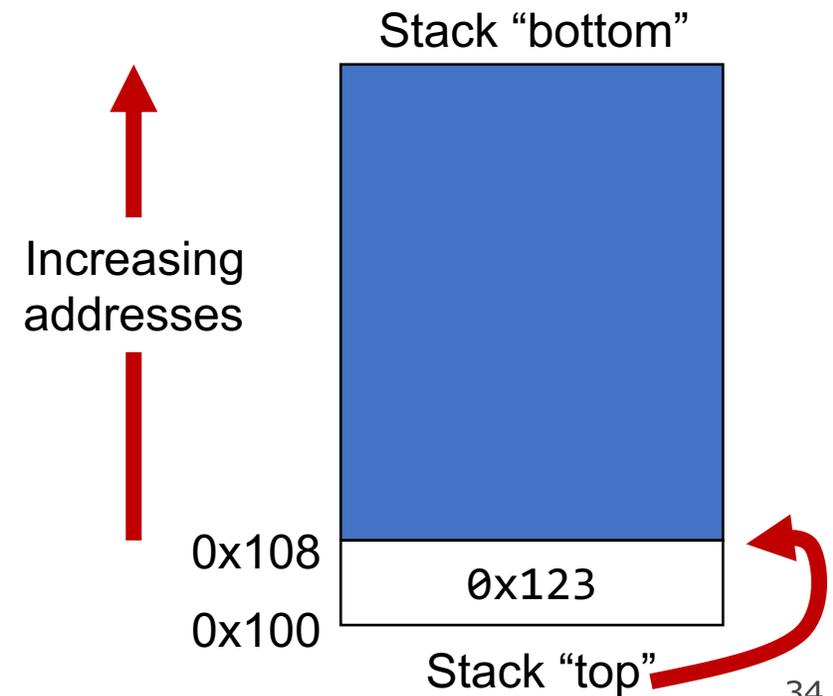
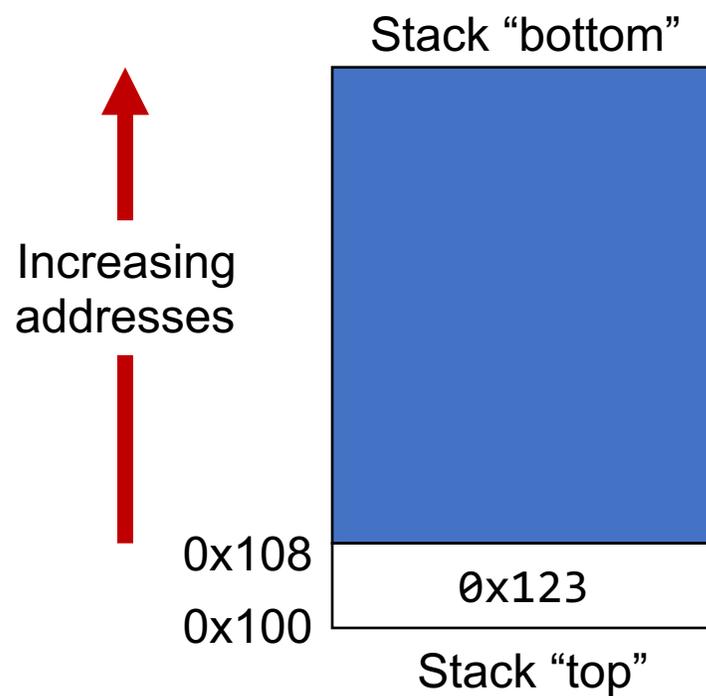
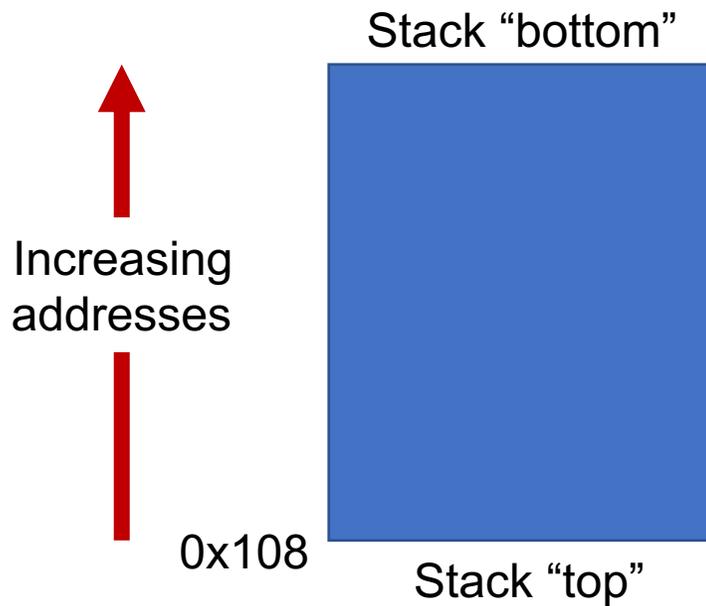
- This behavior is equivalent to the following, but **popq** is a shorter instruction:  
**movq (%rsp), *D***  
**addq \$8, %rsp**
- Sometimes, you'll see instructions just explicitly increment the stack pointer to pop data.

# 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

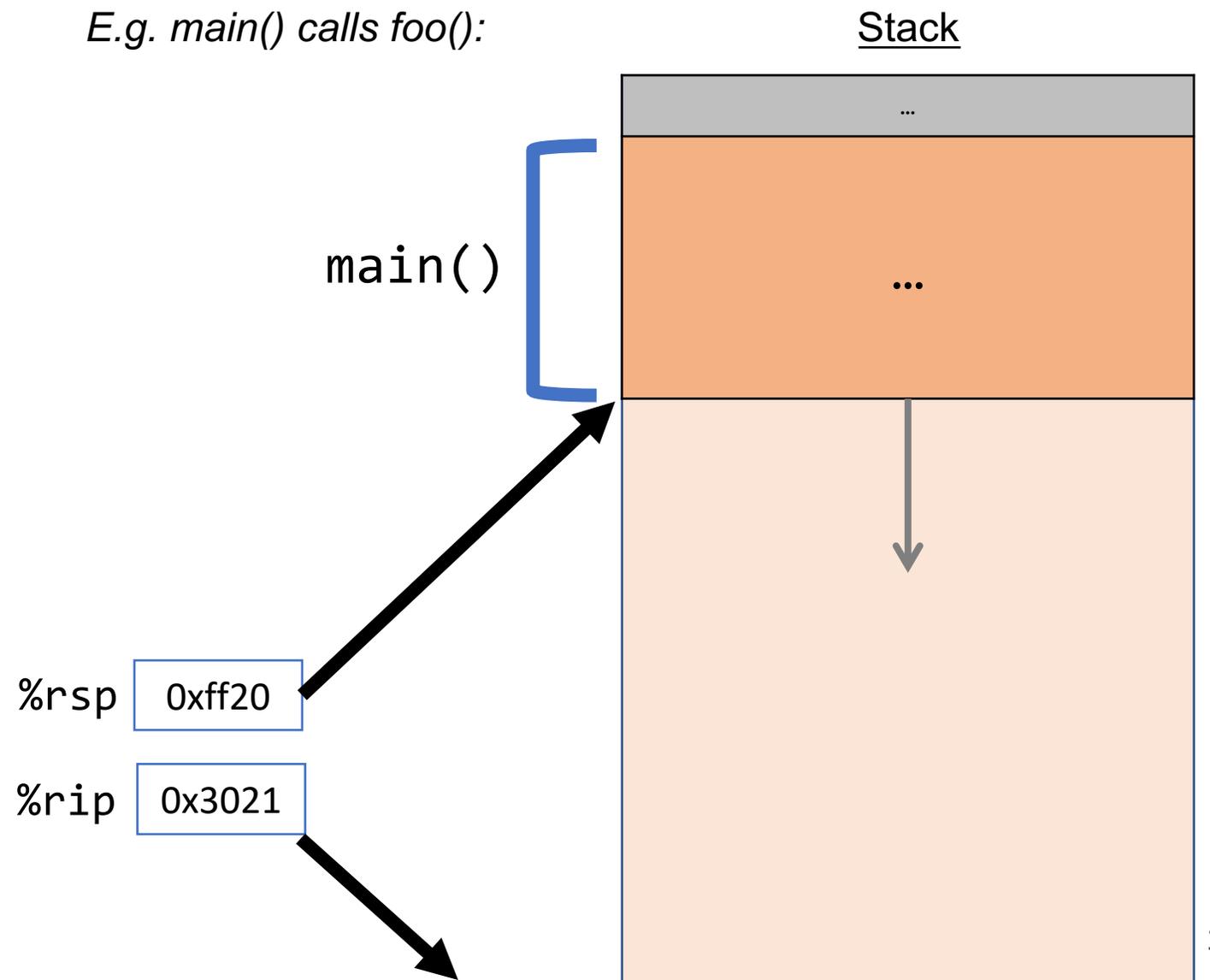
- Revisiting %rip 5
- **Calling Functions** 19
  - The Stack 22
  - **Passing Control** 36
  - Passing Data 44
  - Local Storage 65
- Register Restrictions 69
- Pulling it all together: recursion example 78
- Optimizations 81
- Live session slides 93

```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

# 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.

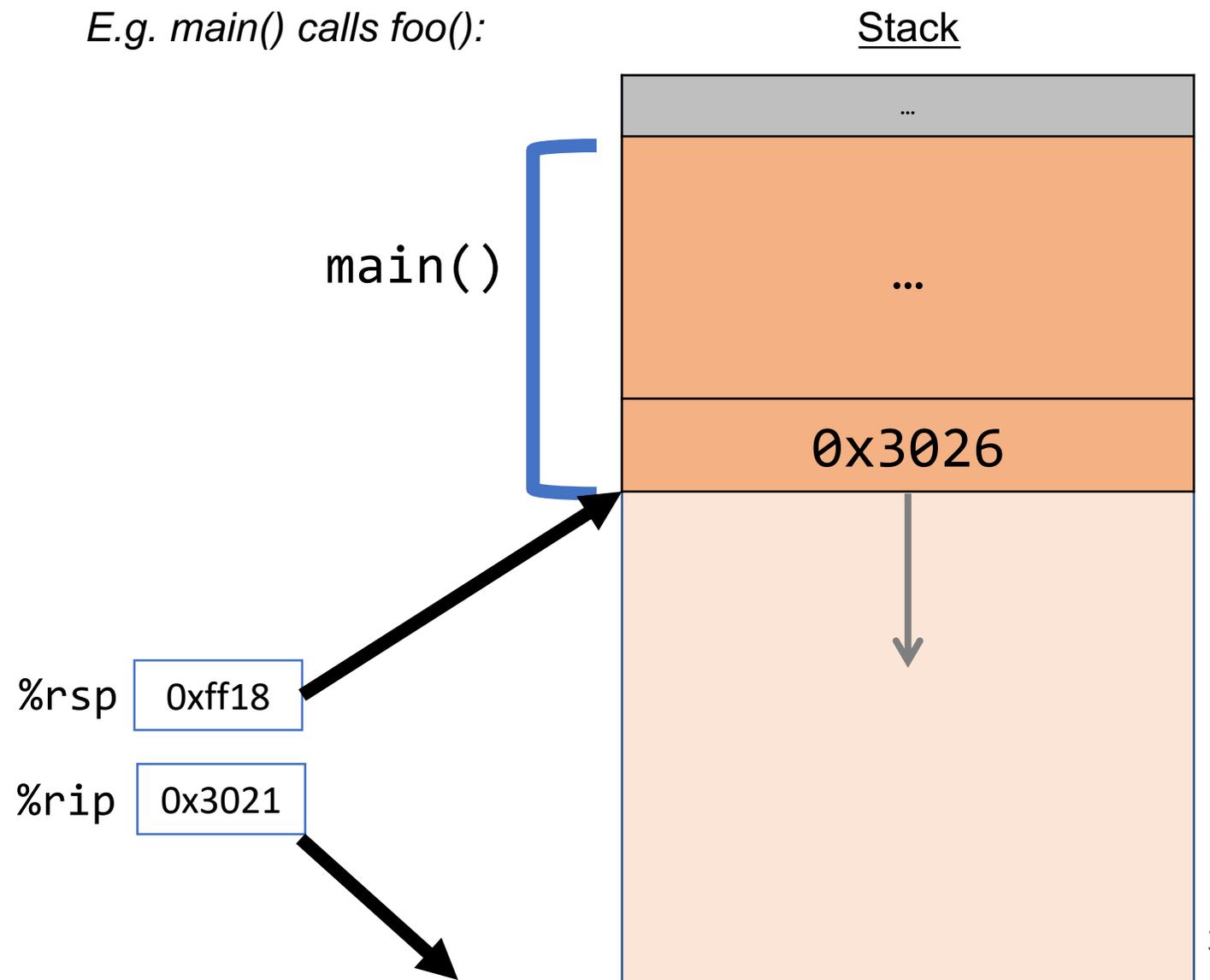
**Solution:** push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.



# 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.

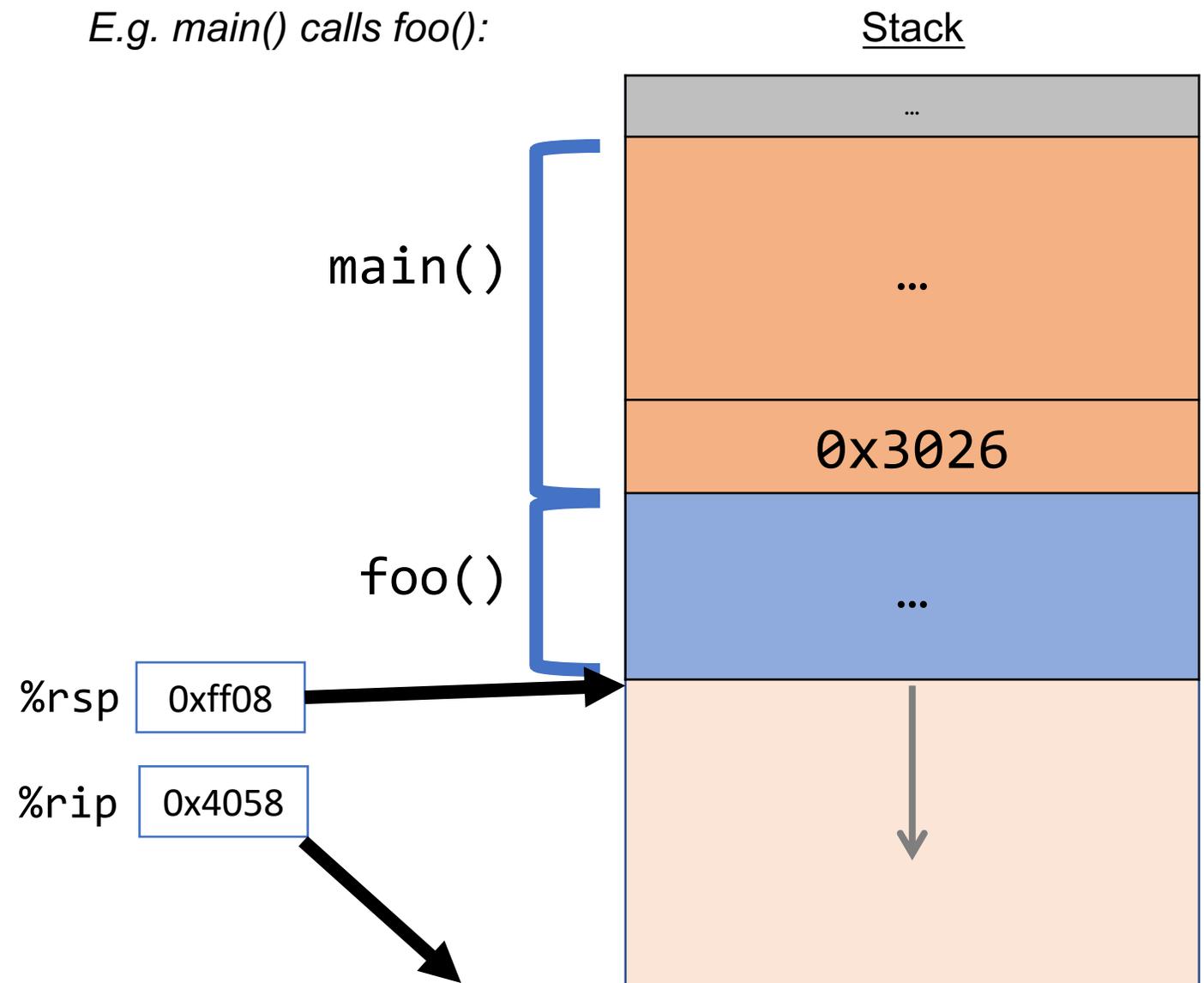
**Solution:** push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.



# 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.

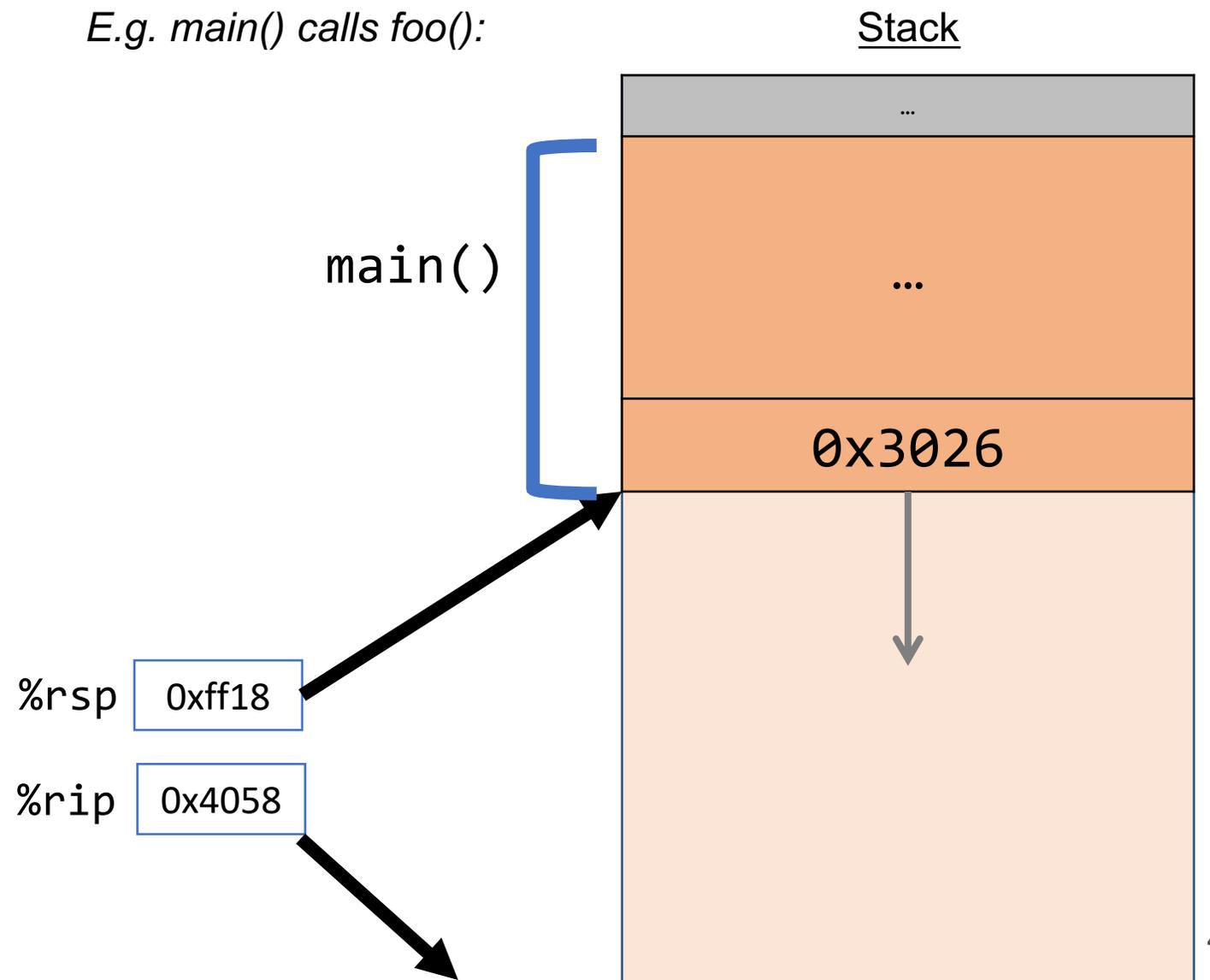
**Solution:** push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.



# 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.

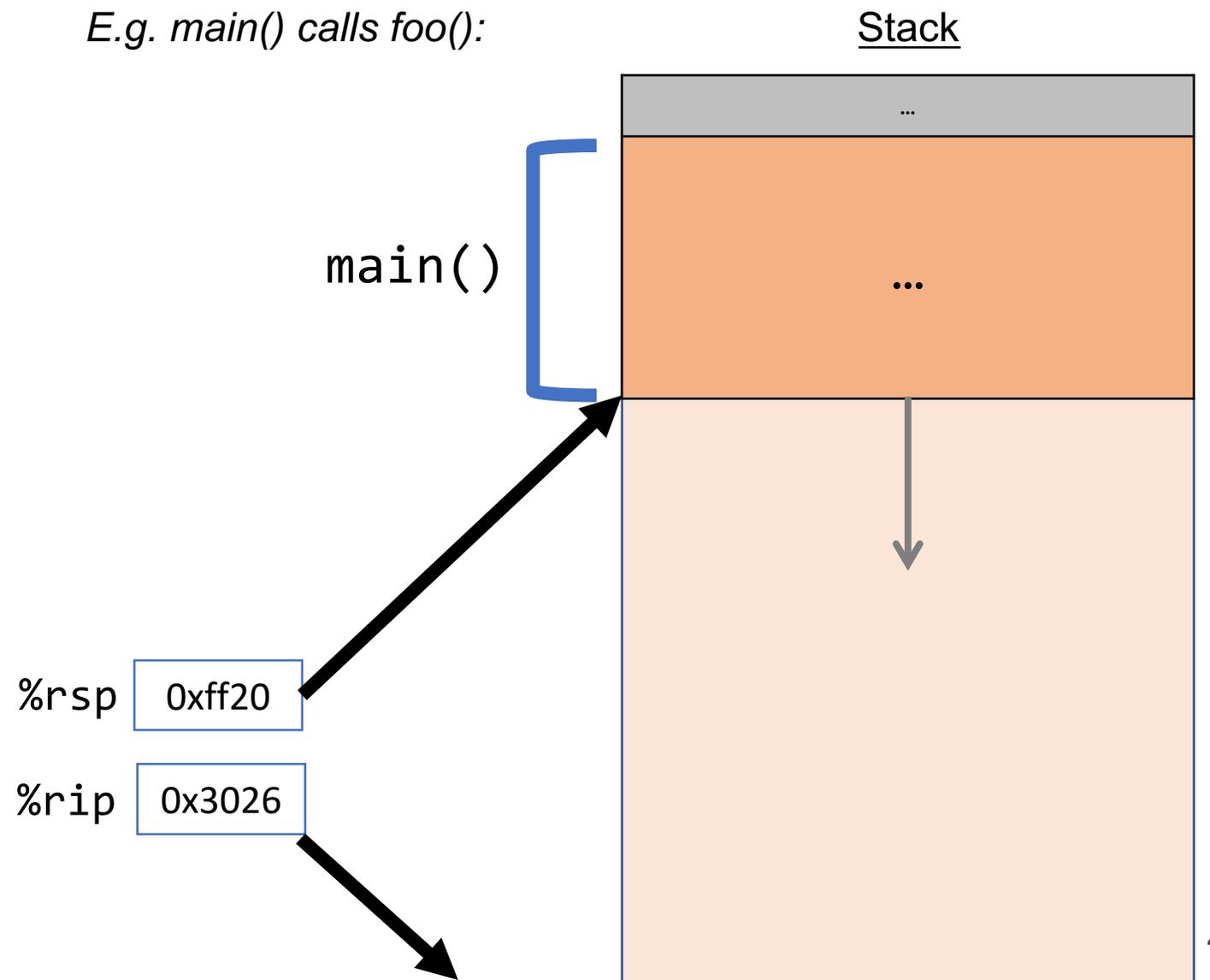
**Solution:** push the next value of %rip onto the stack. Then call the function. When it is finished, put this value back into %rip and continue executing.



# 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).

# 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

- Revisiting %rip 5
- **Calling Functions** 19
  - The Stack 22
  - Passing Control 36
  - **Passing Data** 44
  - Local Storage 65
- Register Restrictions 69
- Pulling it all together: recursion example 78
- Optimizations 81
- Live session slides 93

```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

# Parameters and Return

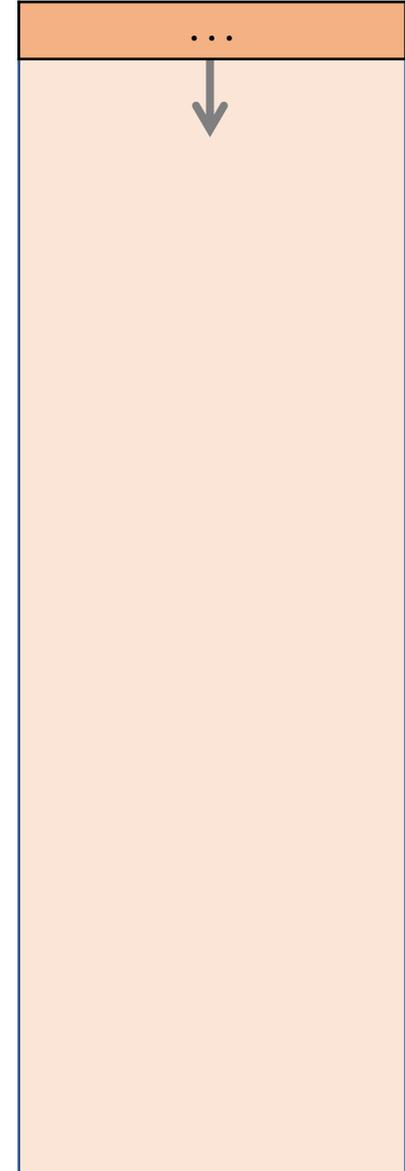
- There are special registers that store parameters and the return value.
- To call a function, we must put any parameters we are passing into the correct registers. (%rdi, %rsi, %rdx, %rcx, %r8, %r9, in that order)
- Parameters beyond the first 6 are put on the stack.
- If the caller expects a return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

# 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() 

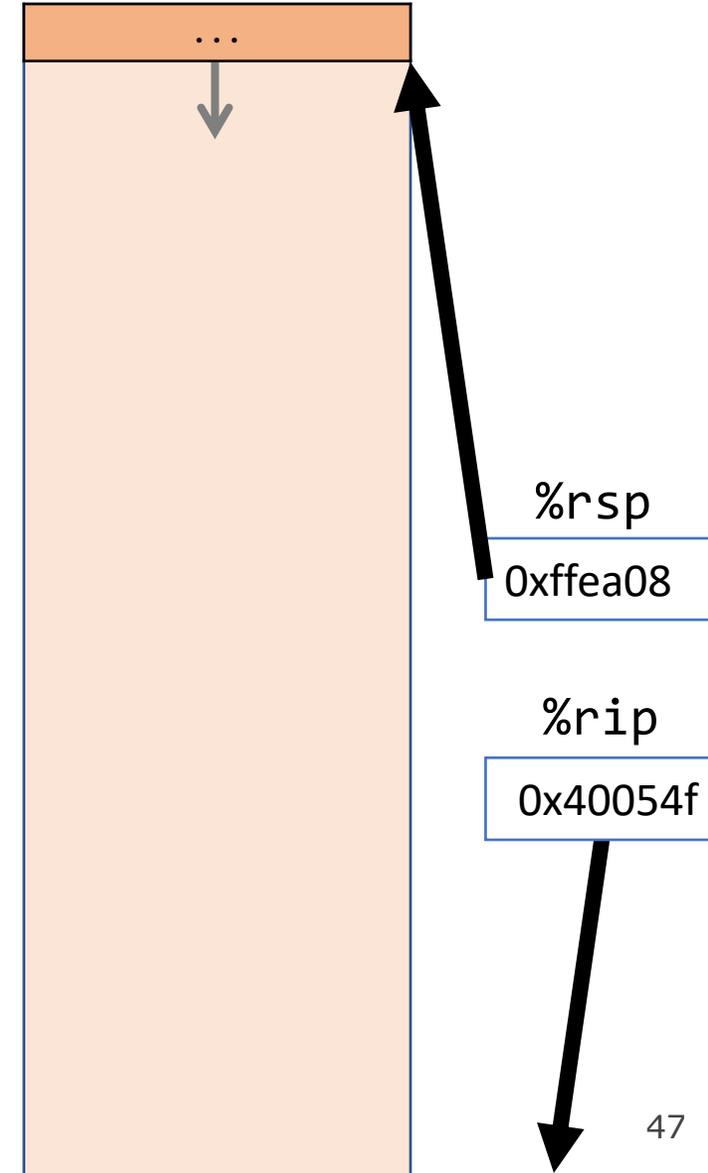


# Parameters and Return

```
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) {
    ...
}
```

main() 

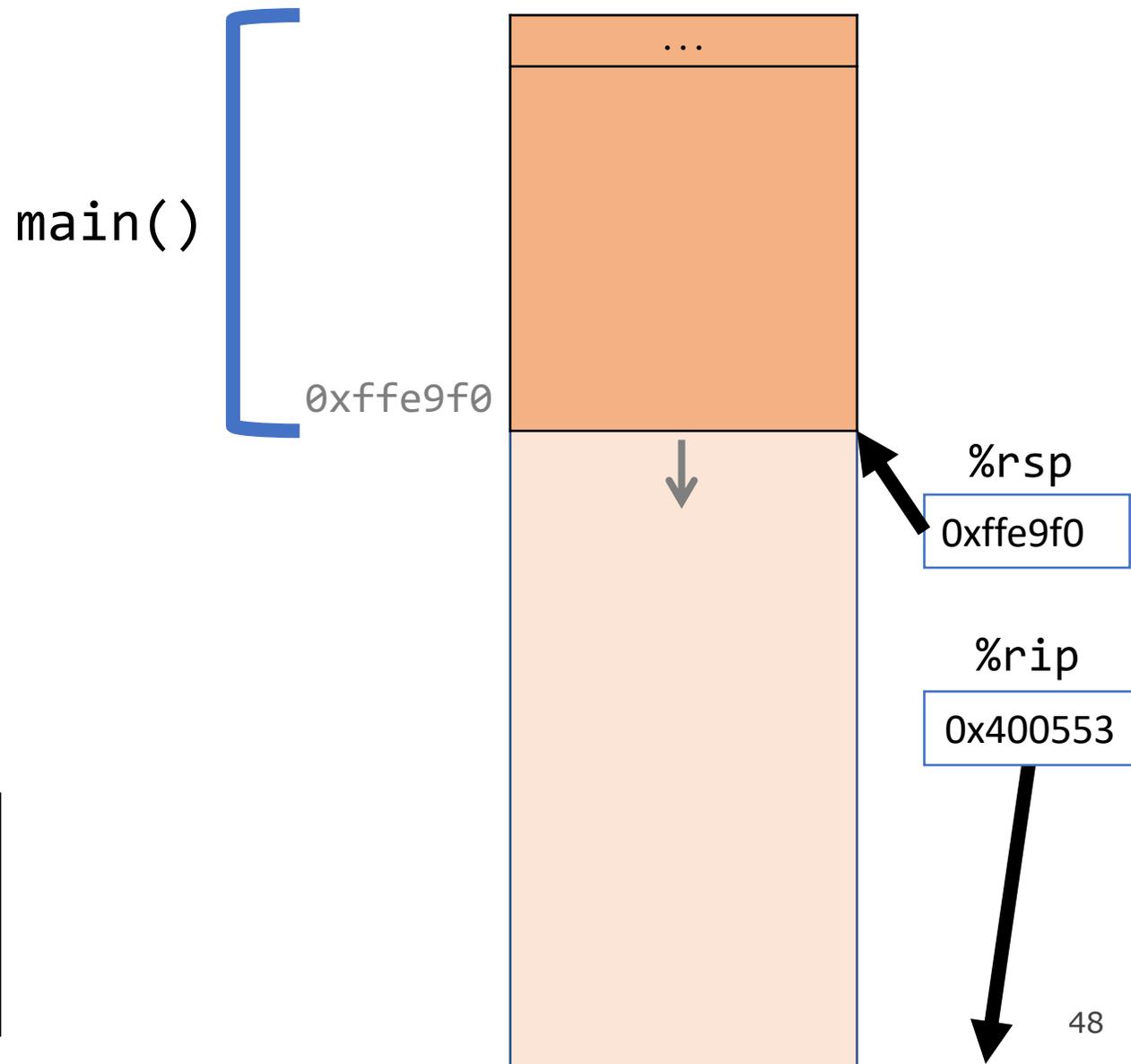


```
0x40054f <+0>:    sub    $0x18,%rsp
0x400553 <+4>:    movl   $0x1,0xc(%rsp)
0x40055b <+12>:   movl   $0x2,0x8(%rsp)
0x400563 <+20>:   movl   $0x3,0x4(%rsp)
0x40056b <+28>:   movl   $0x4,(%rsp)
```

# 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) {  
    ...  
}
```

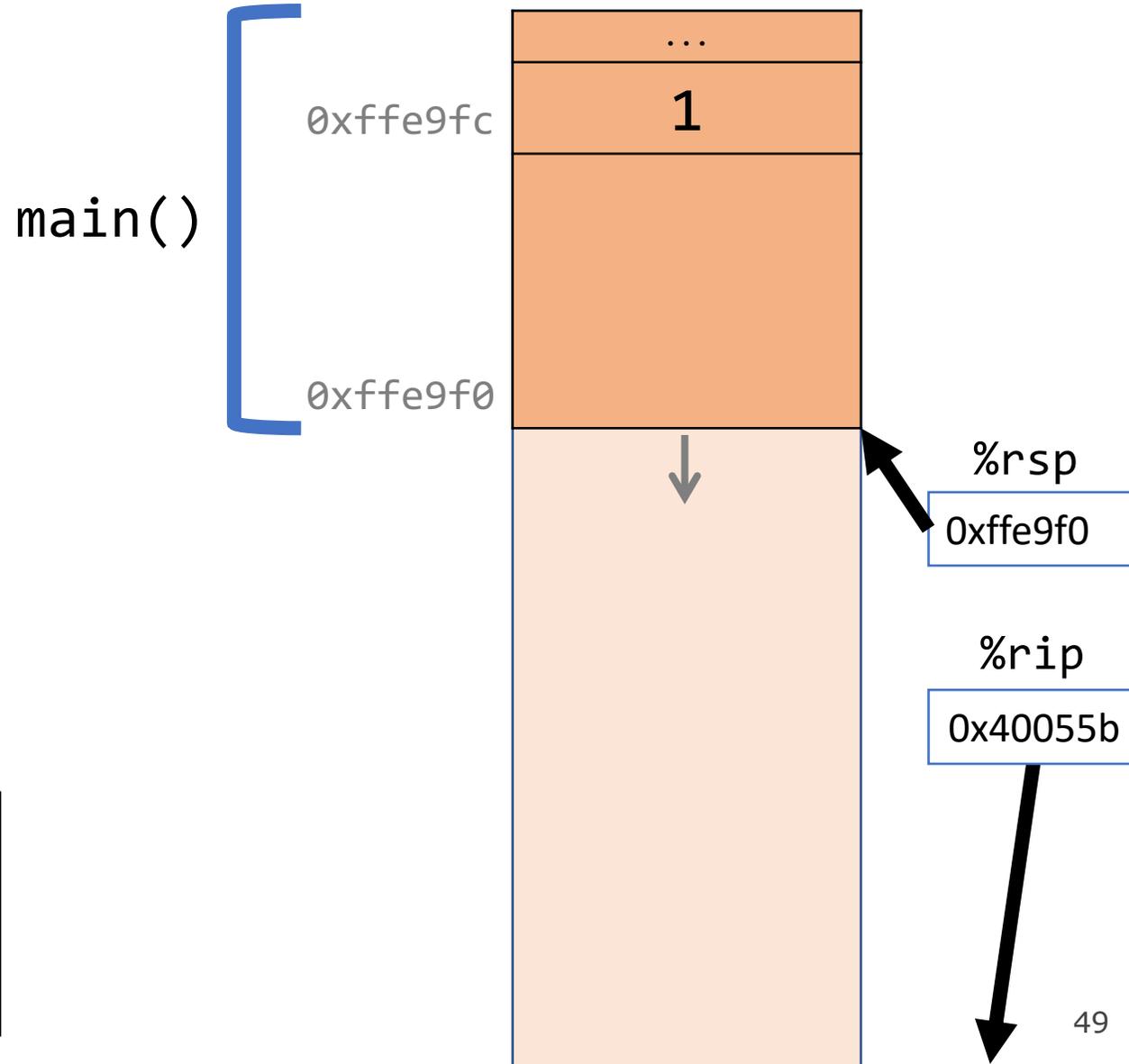
```
0x40054f <+0>:    sub    $0x18,%rsp  
0x400553 <+4>:    movl   $0x1,0xc(%rsp)  
0x40055b <+12>:   movl   $0x2,0x8(%rsp)  
0x400563 <+20>:   movl   $0x3,0x4(%rsp)  
0x40056b <+28>:   movl   $0x4,0x0(%rsp)
```



# 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   $0x4,0x0(%rsp)
```

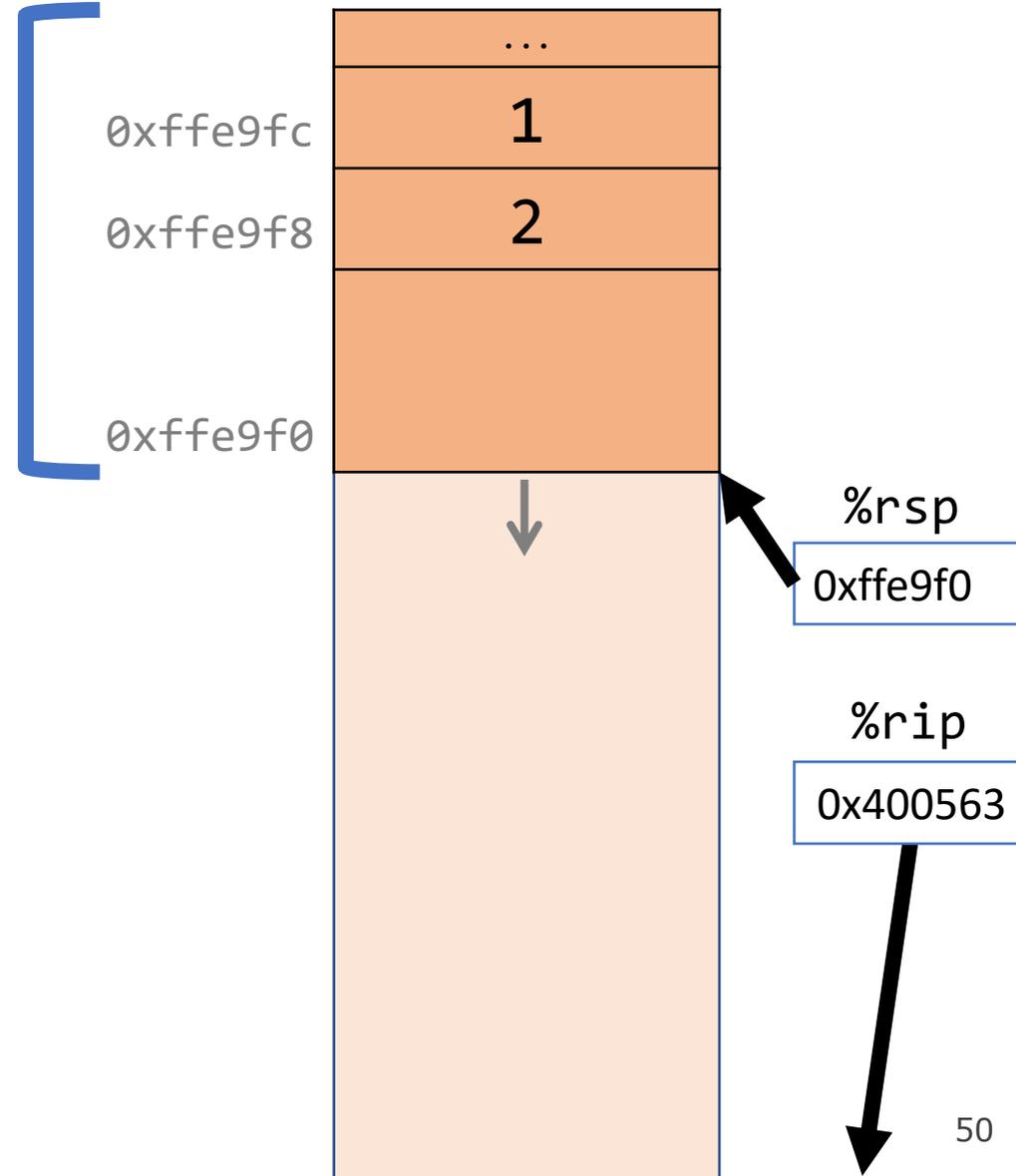


# Parameters and Return

```
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   $0x4,0x0(%rsp)
```

main()

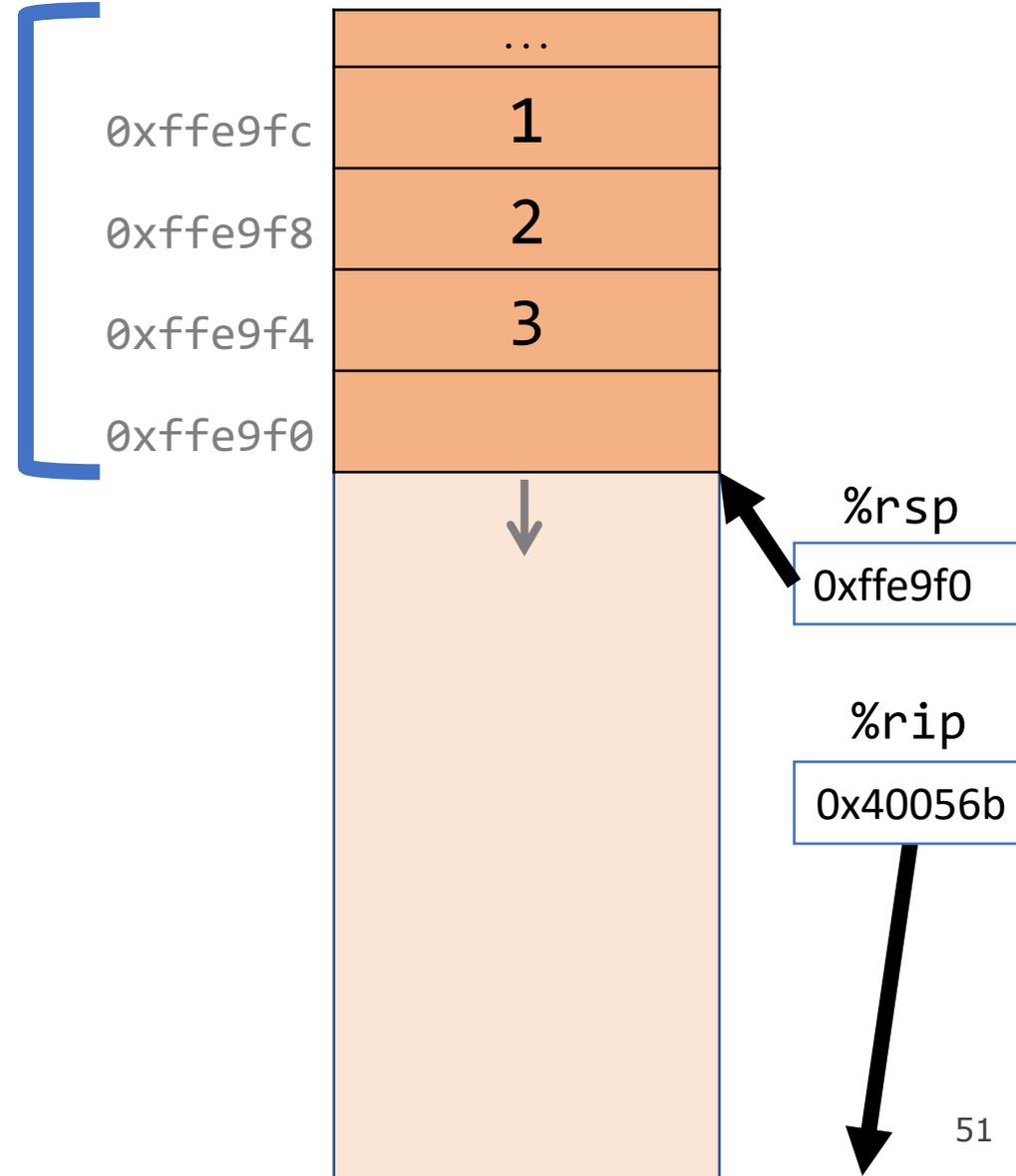


# Parameters and Return

```
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) {  
    ...  
}
```

```
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>:   pusha  $0x4
```

main()



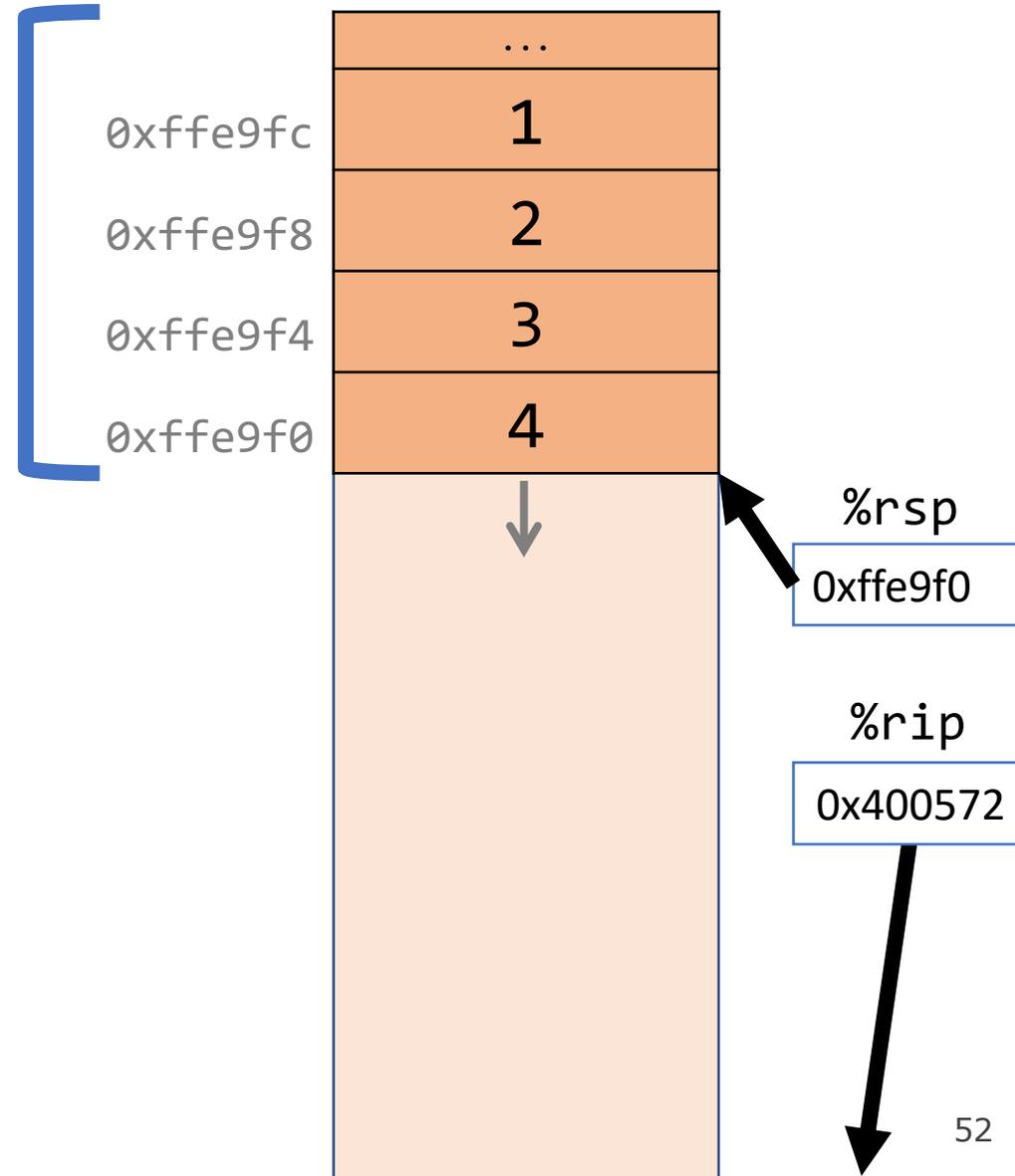
# Parameters and Return

```
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) {
    ...
}
```

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

main()



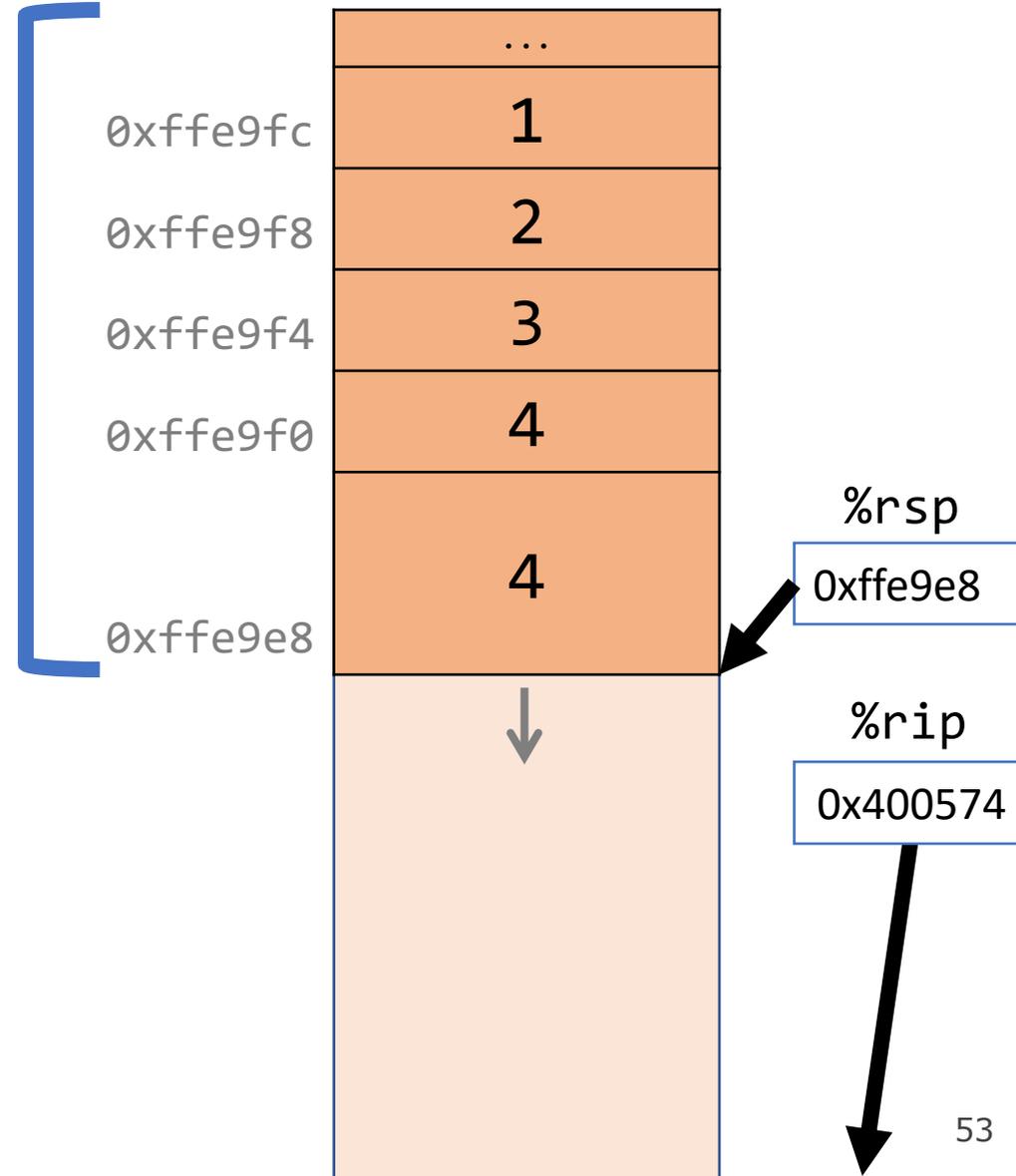
# Parameters and Return

```
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) {
    ...
}
```

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

main()



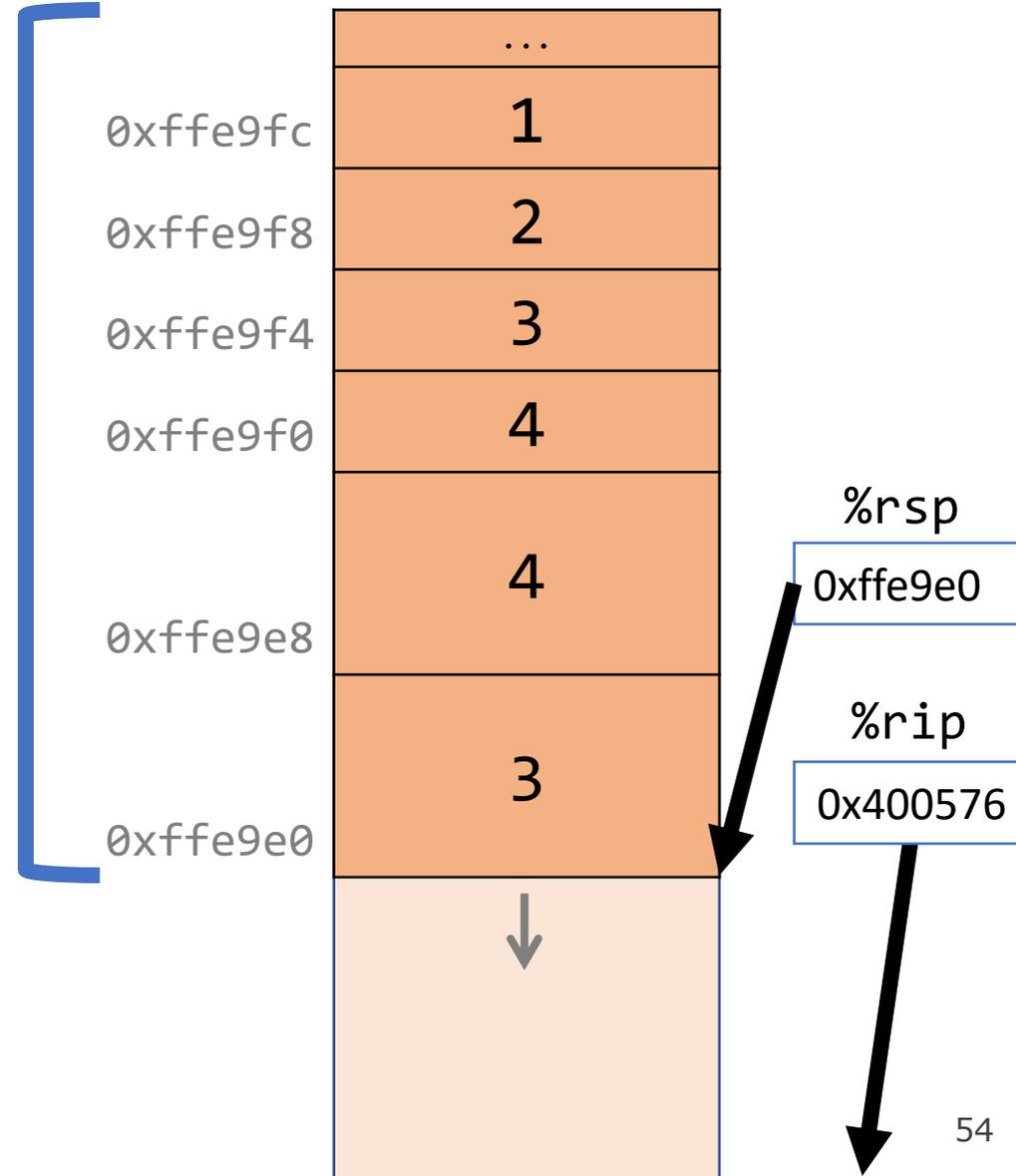
# Parameters and Return

```
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) {
    ...
}
```

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

main()

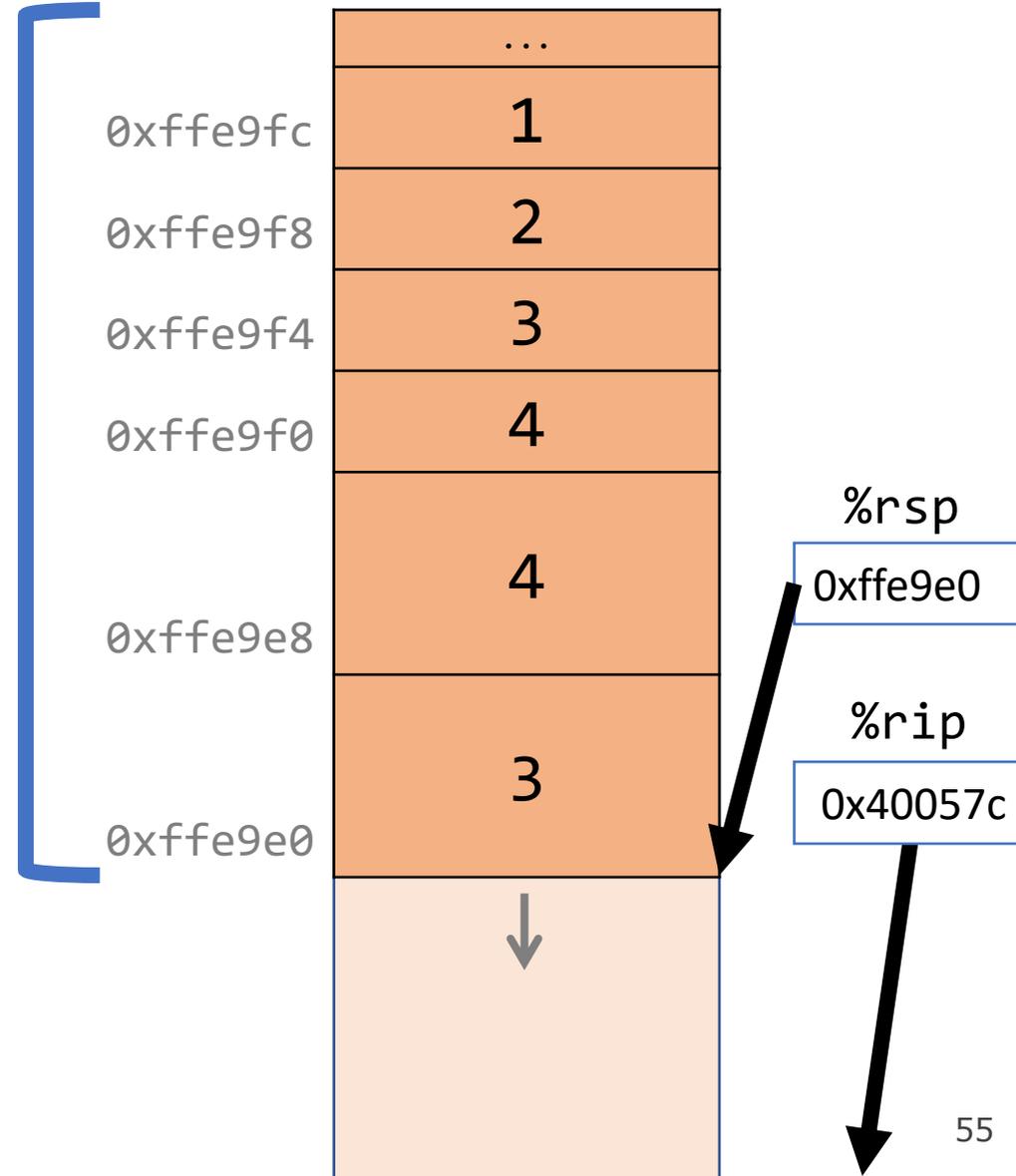


# Parameters and Return

```
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) {  
    ...  
}
```

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

main()



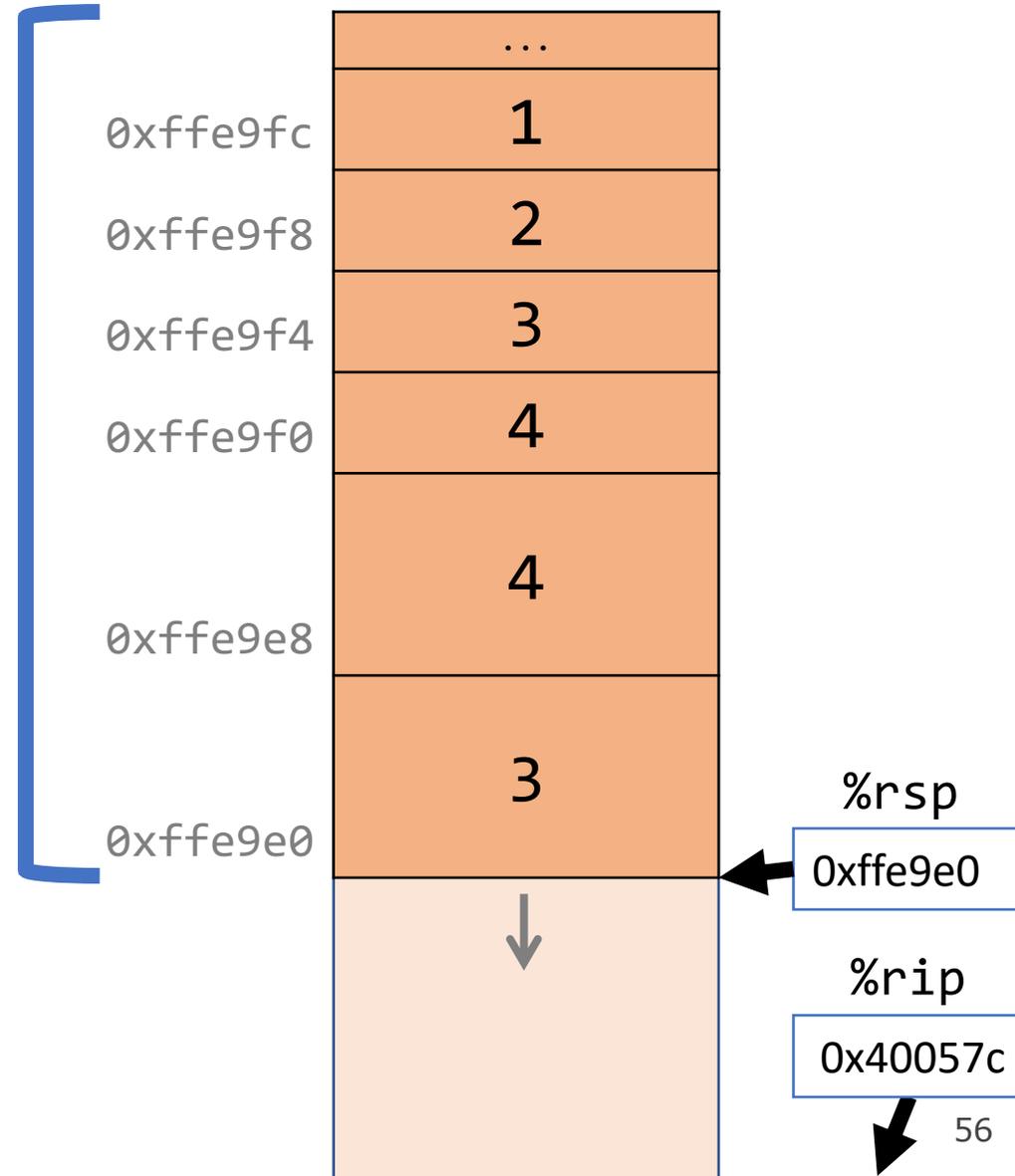
# Parameters and Return

```
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) {
    ...
}
```

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

main()



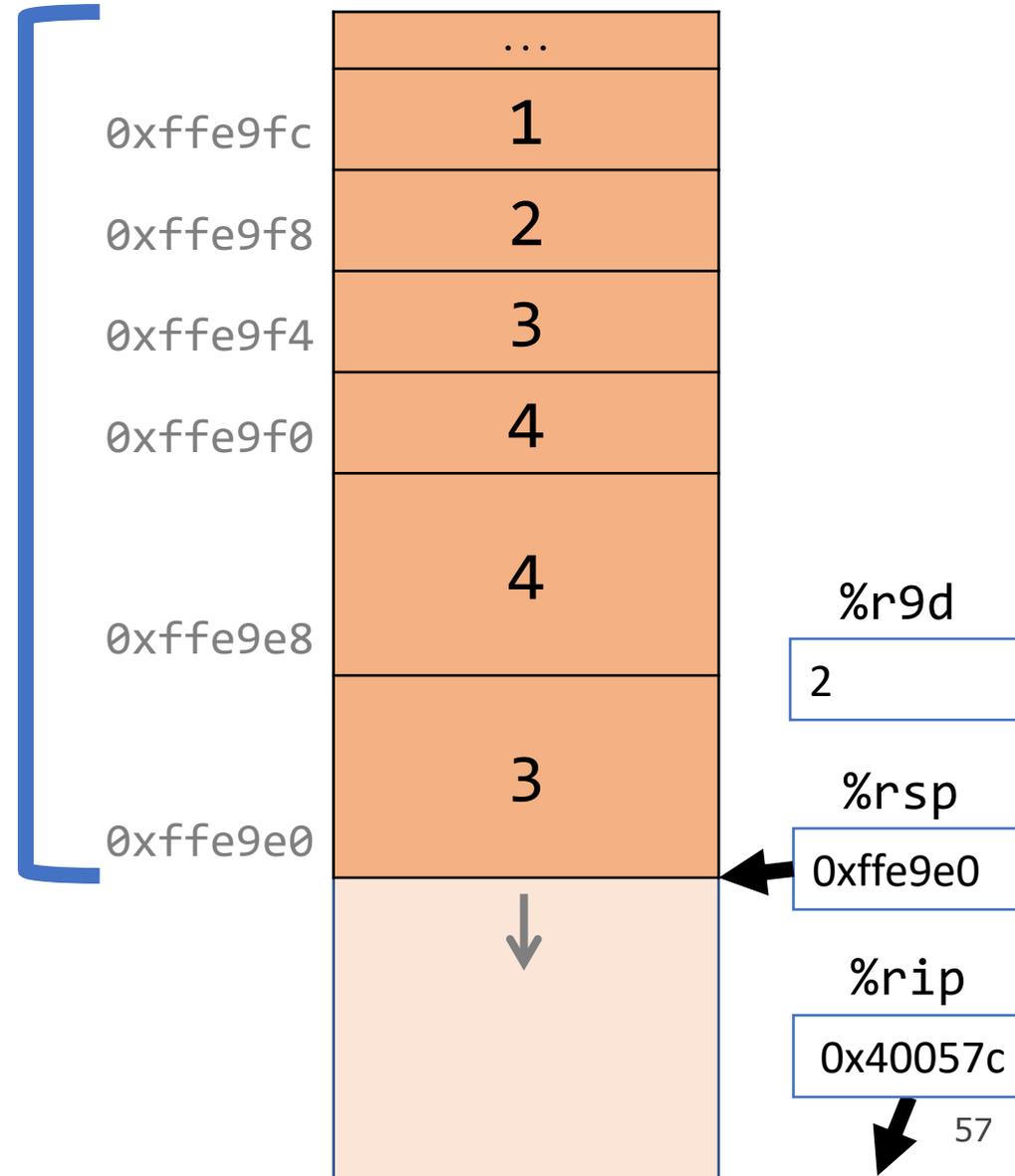
# Parameters and Return

```
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) {
    ...
}
```

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

main()



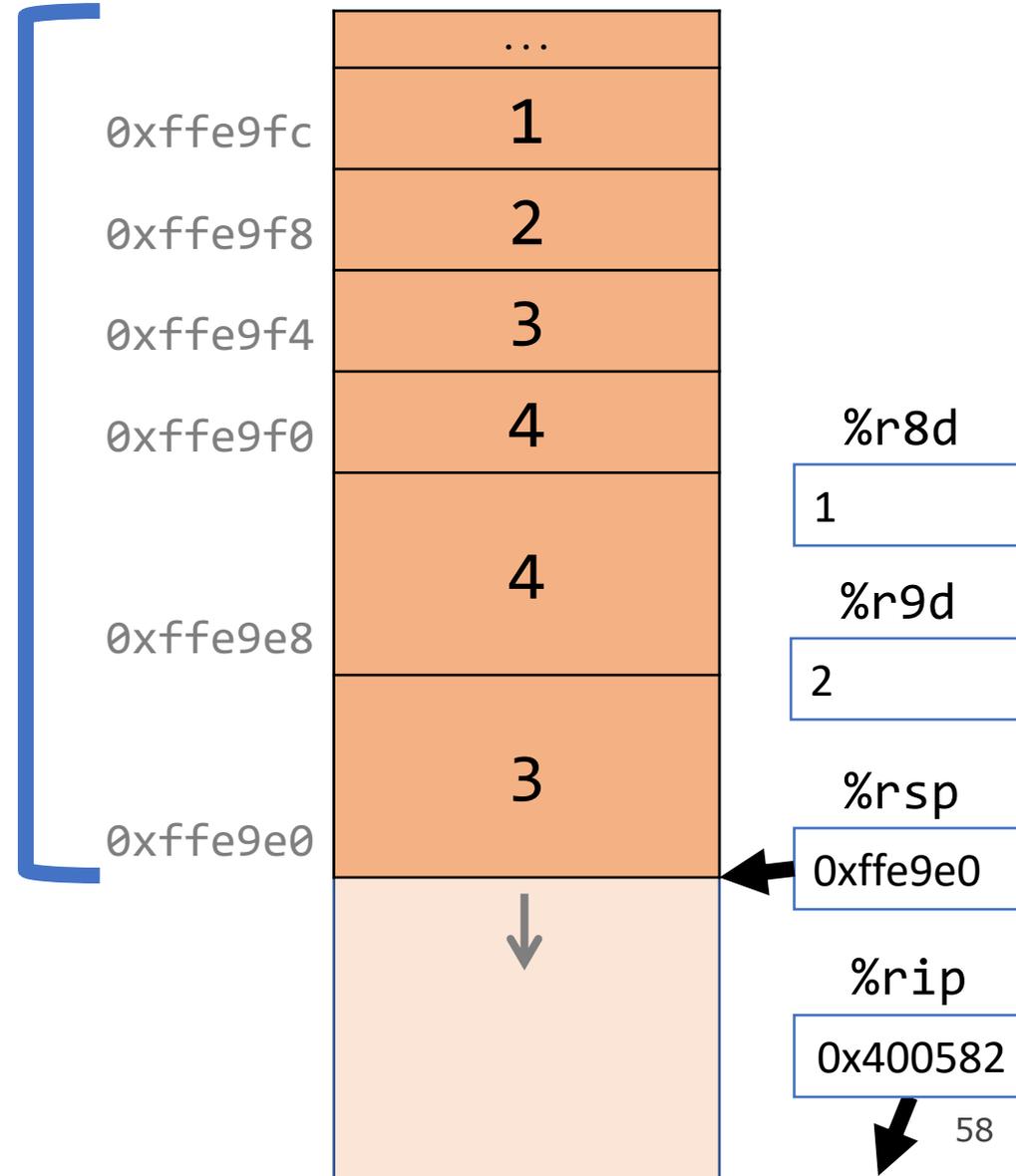
# Parameters and Return

```
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
```

main()



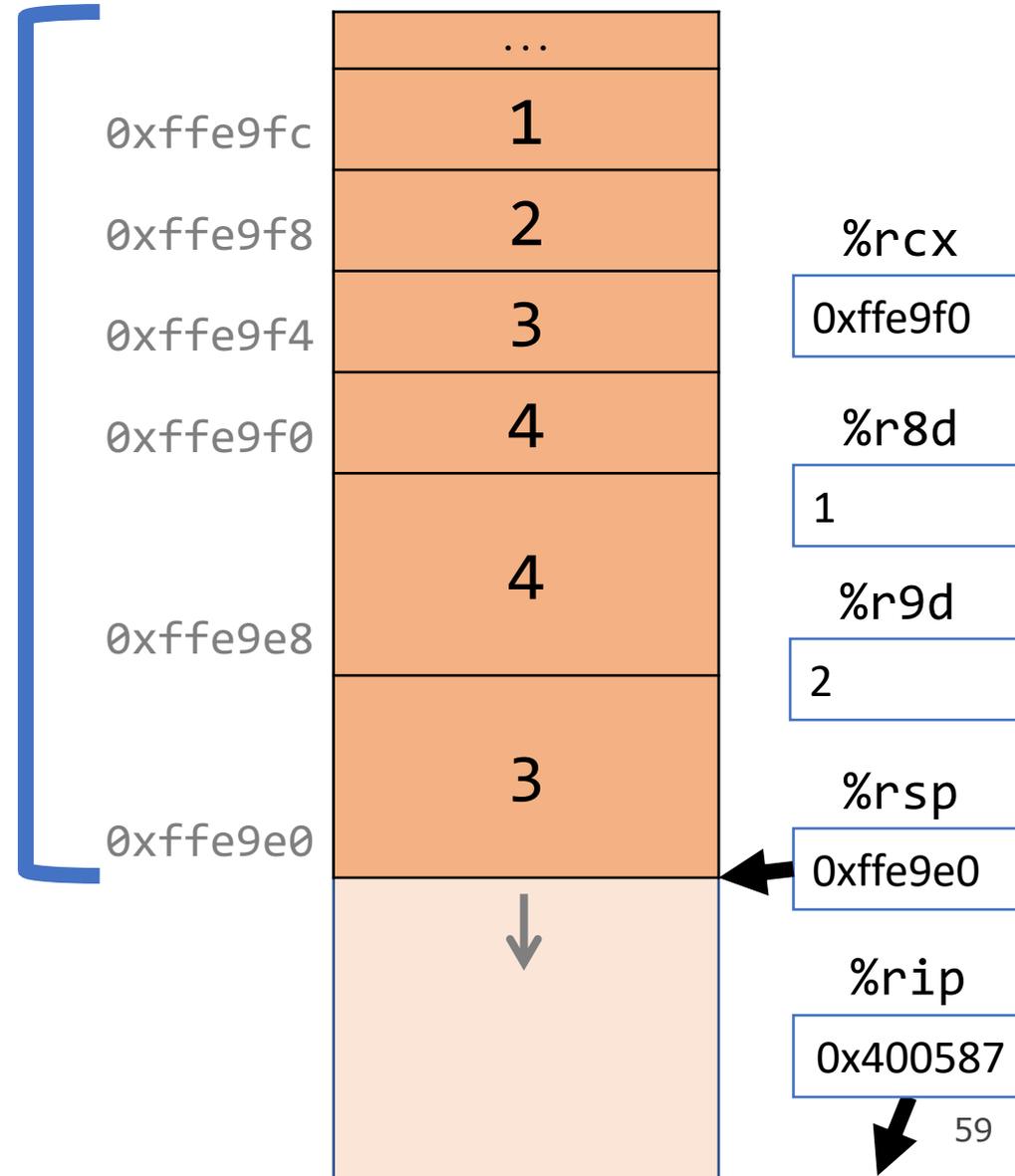
# Parameters and Return

```
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) {
    ...
}
```

```
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
```

main()



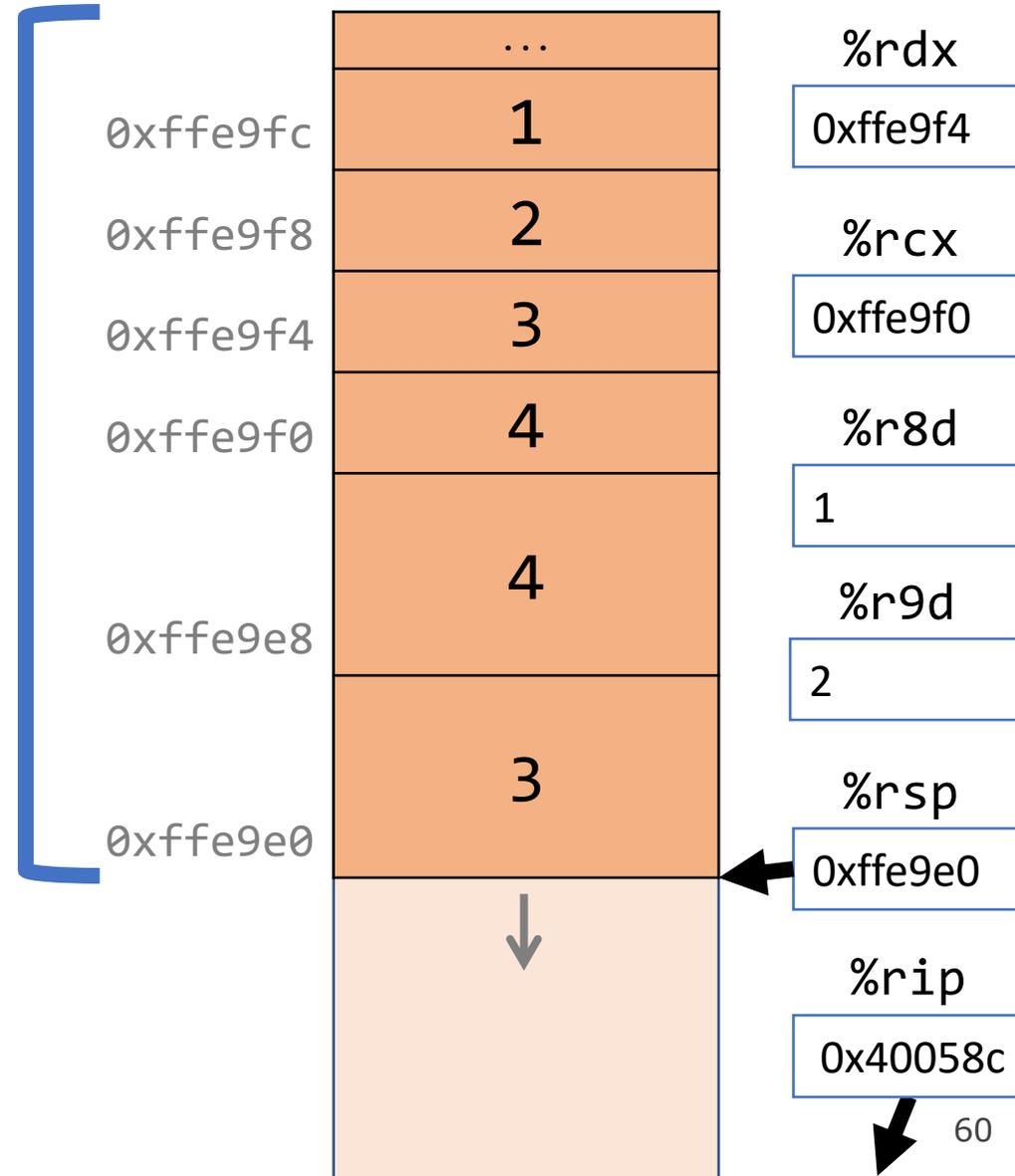
# Parameters and Return

```
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) {
    ...
}
```

```
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);
    ...
}

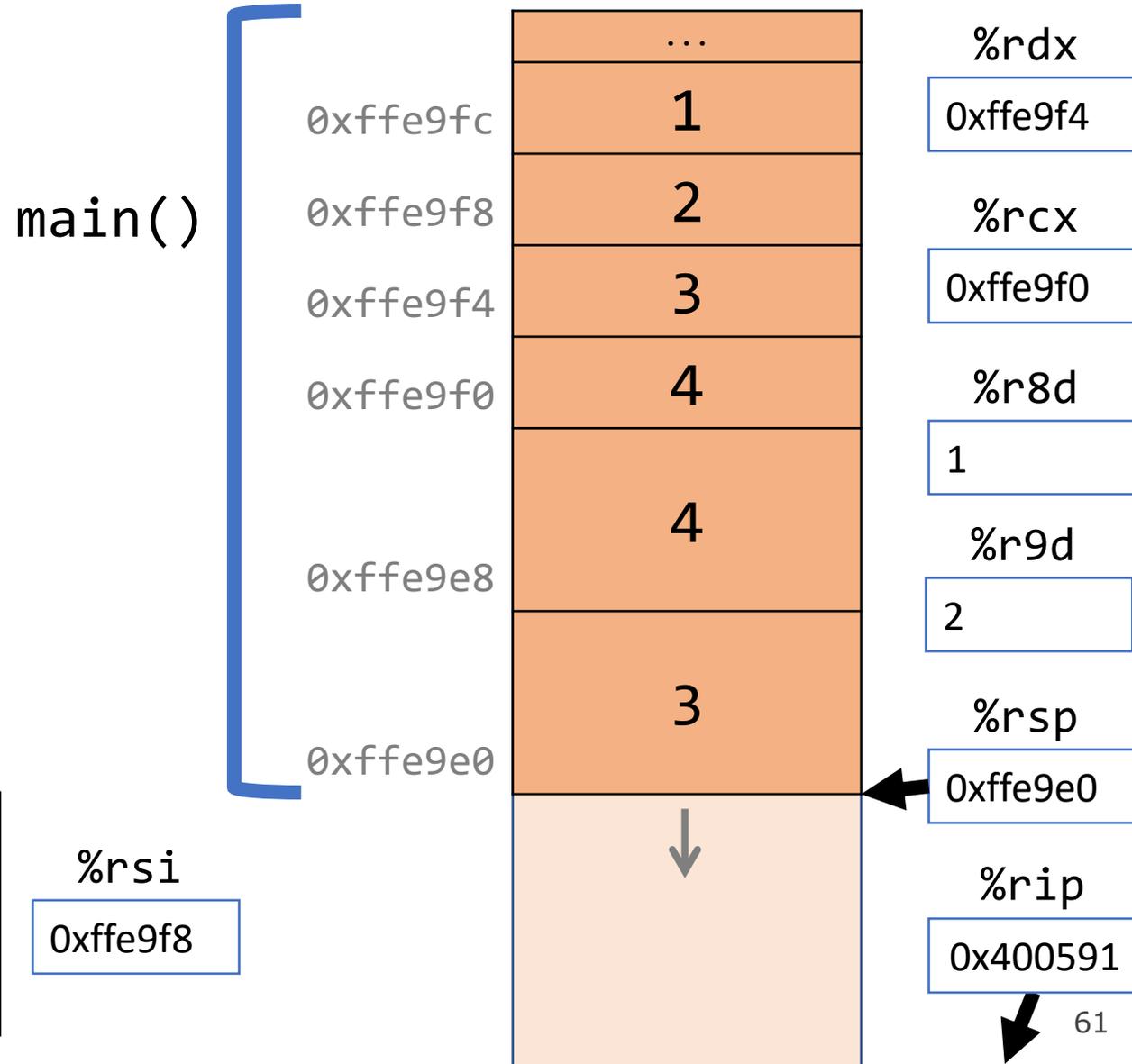
int func(int *p1, int *p2, int *p3, int *p4,
        int v1, int v2, int v3, int v4) {
    ...
}

```

```

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 0x400546 <func>

```

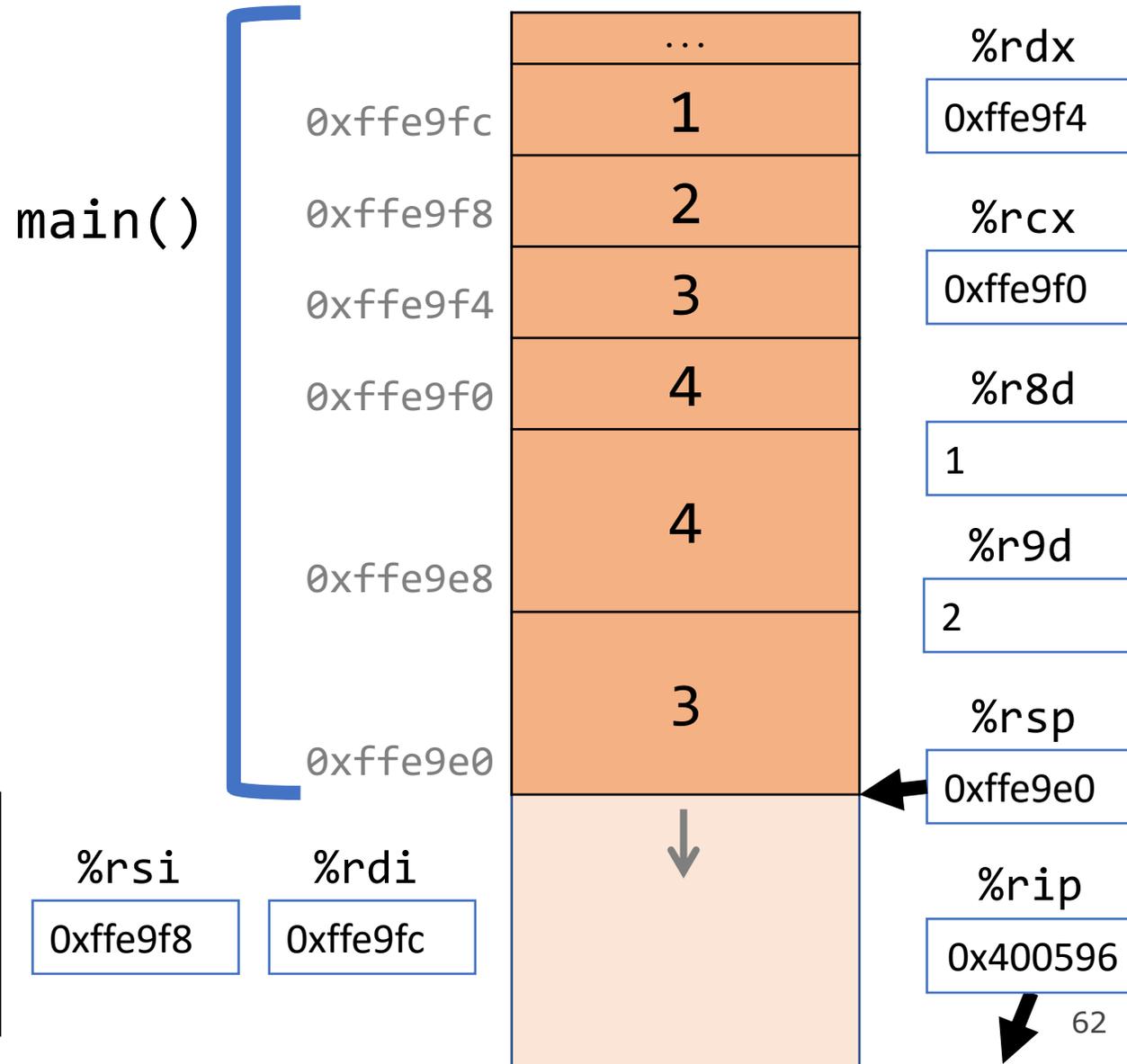


# Parameters and Return

```
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  0x400546 <func>
0x40059b <+76>: add    $0x10,%rsp
```



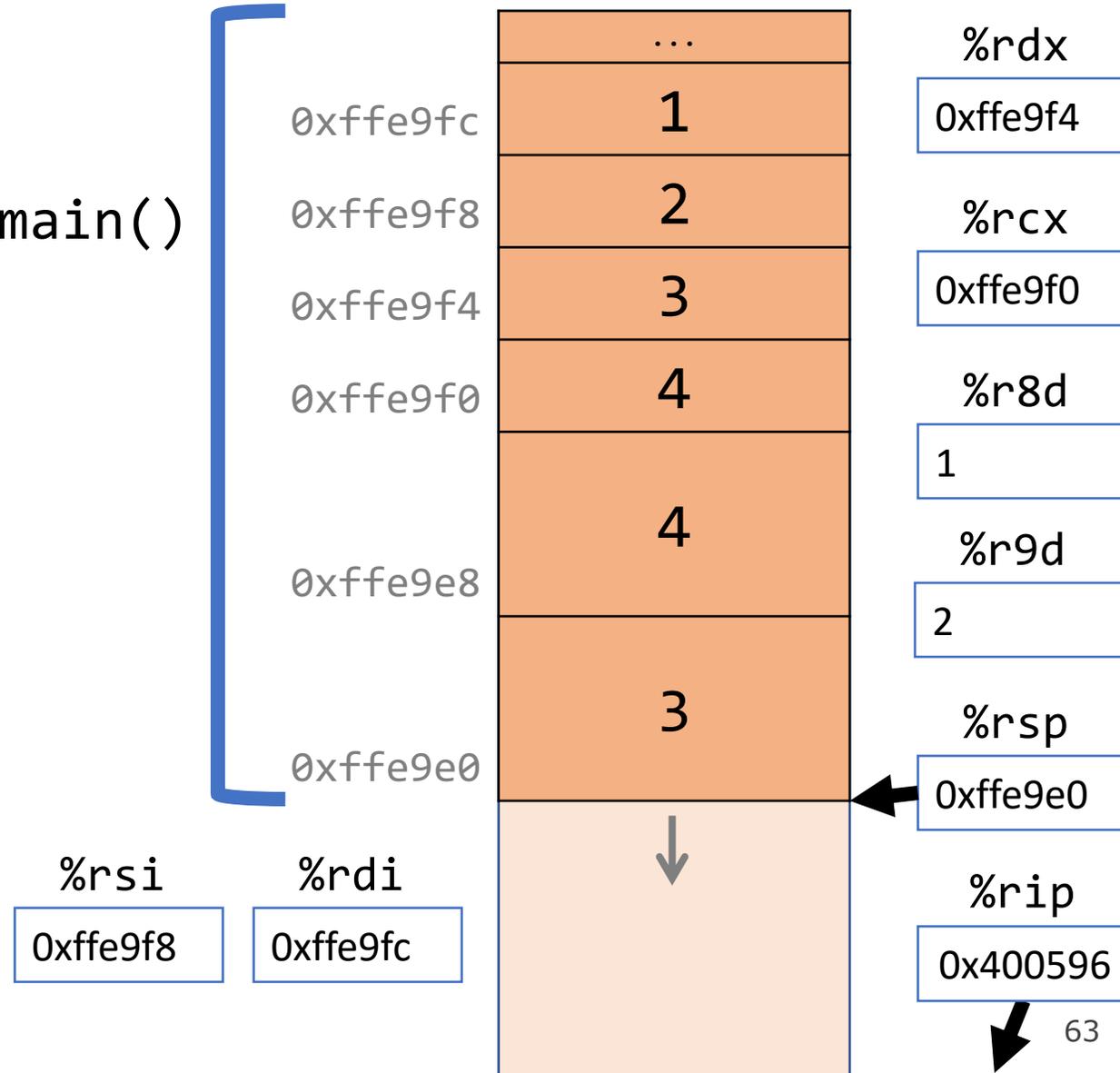
# Parameters and Return

```
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 0x400546 <func>
0x40059b <+76>: add    $0x10,%rsp
...
```

main()

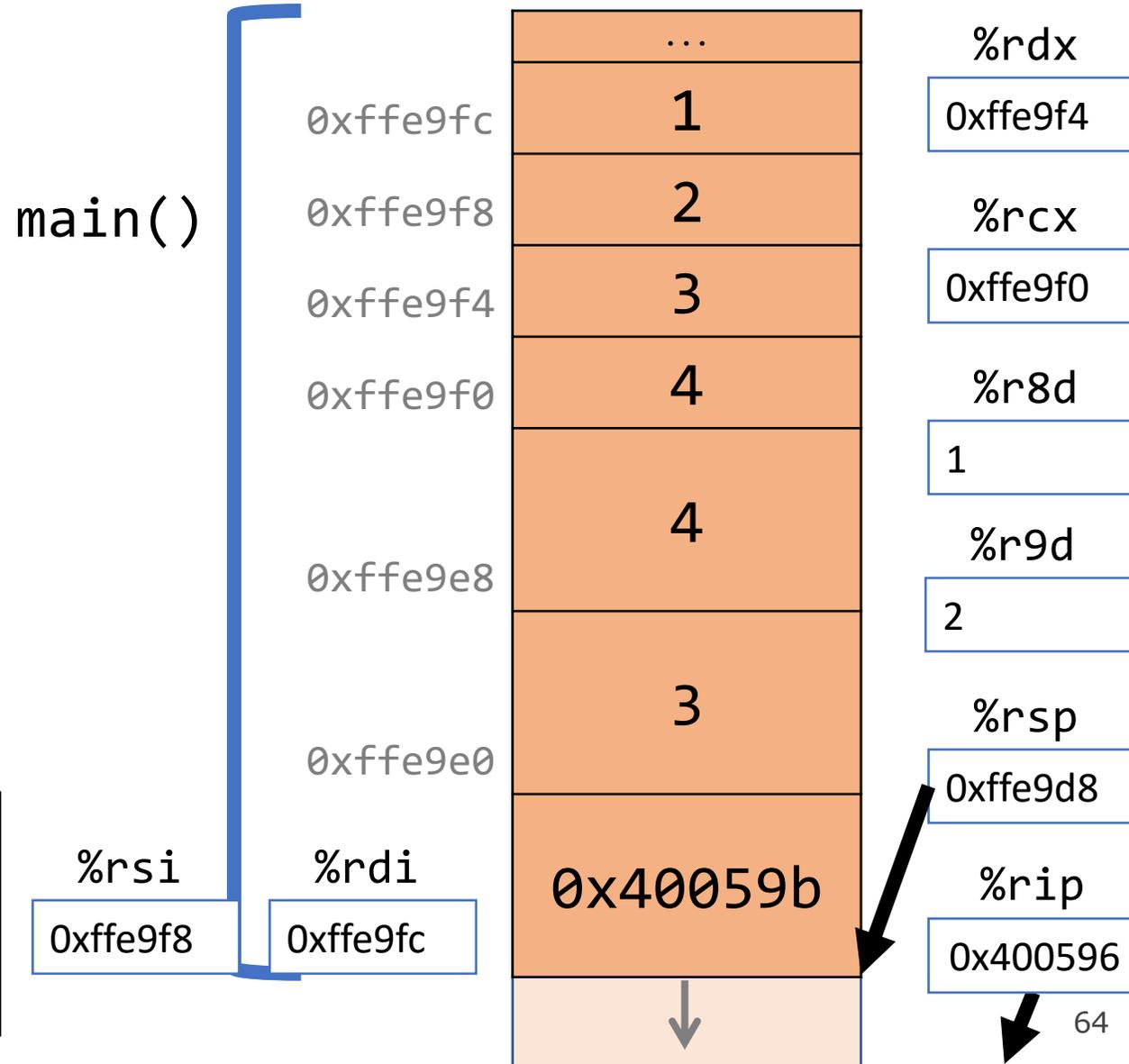


# Parameters and Return

```
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) {
    ...
}
```

```
0x40058c <+61>: lea    0x18(%rsp),%rsi
0x400591 <+66>: lea    0x1c(%rsp),%rdi
0x400596 <+71>: callq 0x400546 <func>
0x40059b <+76>: add    $0x10,%rsp
...
```



# Lecture Plan

- Revisiting %rip 5
- **Calling Functions** 19
  - The Stack 22
  - Passing Control 36
  - Passing Data 44
  - **Local Storage** 65
- Register Restrictions 69
- Pulling it all together: recursion example 78
- Optimizations 81
- Live session slides 93

```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

# 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:  
    subq $0x10, %rsp           // 16 bytes for stack frame  
    movq $0x216, (%rsp)       // store 534 in arg1  
    movq $0x421, 8(%rsp)      // store 1057 in arg2  
    leaq 8(%rsp), %rsi        // compute &arg2 as second arg  
    movq %rsp, %rdi           // compute &arg1 as first arg  
    call swap_add             // call swap_add(&arg1, &arg2)
```

# Lecture Plan

- Revisiting %rip 5
- Calling Functions 19
  - The Stack 22
  - Passing Control 36
  - Passing Data 44
  - Local Storage 65
- **Register Restrictions** 69
- Pulling it all together: recursion example 78
- Optimizations 81

```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

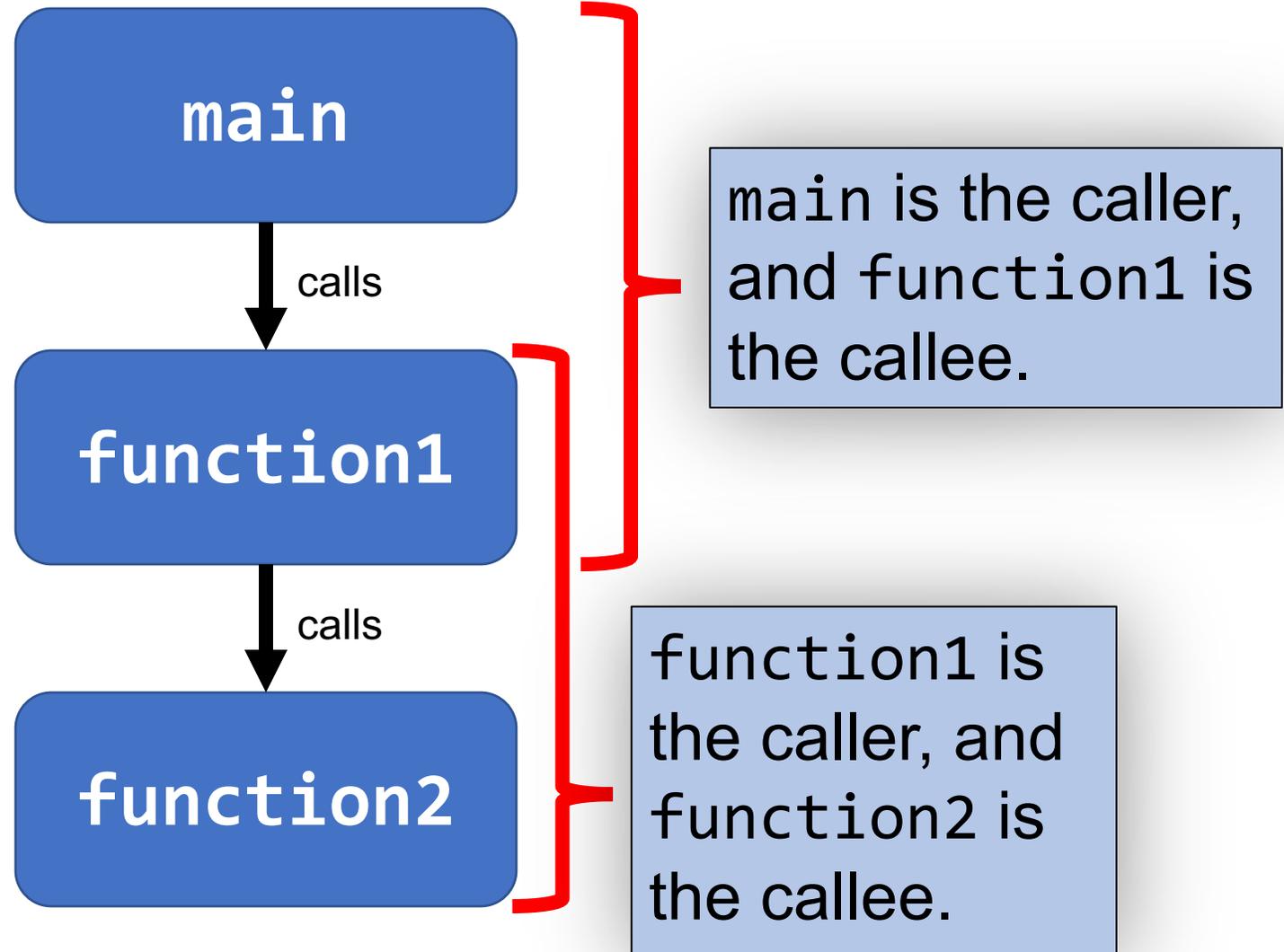
# Register Restrictions

There is only one copy of registers for all programs and 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

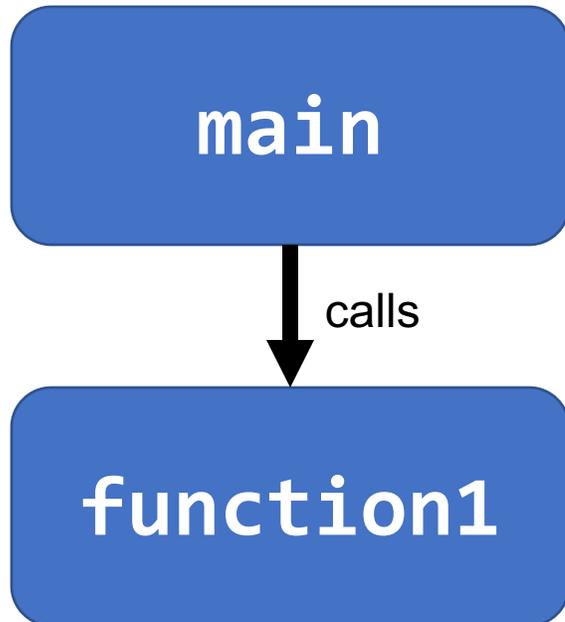
## Caller-Owned

- 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.

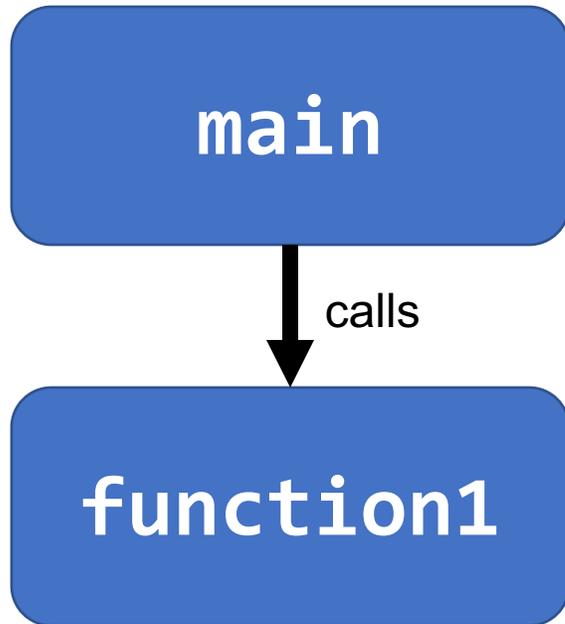
# Caller-Owned Registers



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

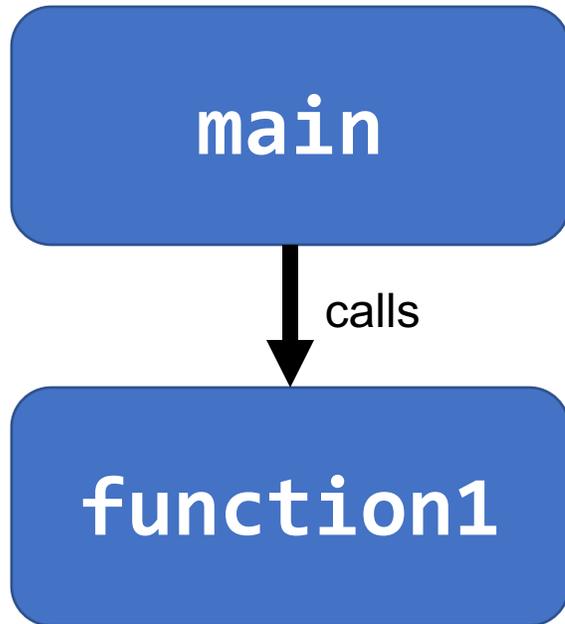
If function1 wants to use any caller-owned registers, it must save the existing values and restore them before returning.

# Caller-Owned Registers



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

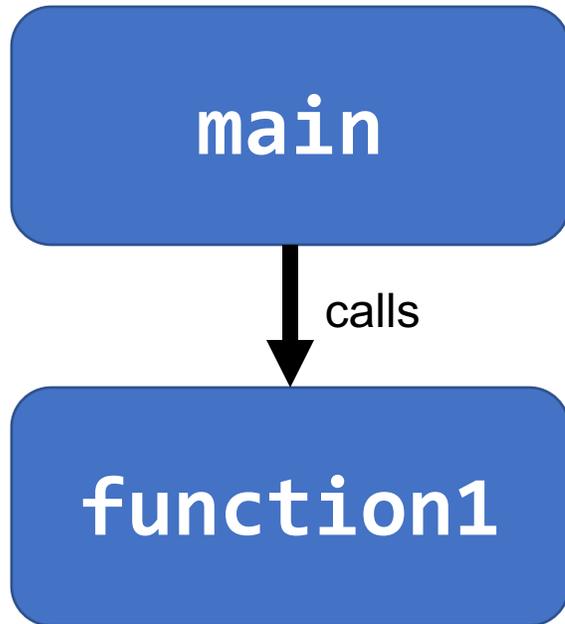
# Callee-Owned Registers



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

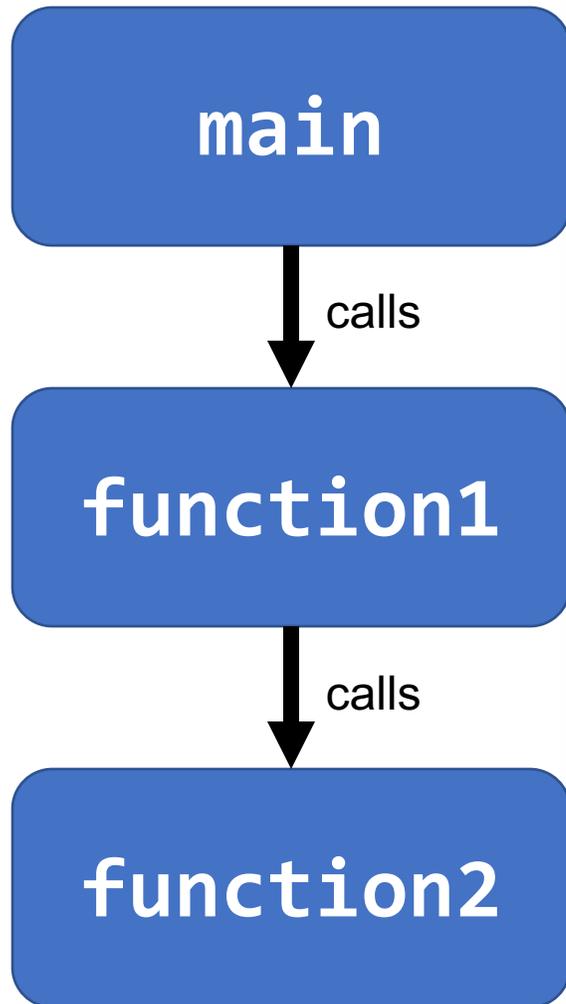
If function1 wants to use any callee-owned registers, it can do so without saving the existing values.

# 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.

# Lecture Plan

- Revisiting %rip 5
- Calling Functions 19
  - The Stack 22
  - Passing Control 36
  - Passing Data 44
  - Local Storage 65
- Register Restrictions 69
- **Pulling it all together: recursion example** 78
- Optimizations 81
- Live session slides 93

```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

# Example: Recursion

- Let's look at an example of recursion at the assembly level.
- We'll use everything we've learned about registers, the stack, function calls, parameters, and assembly instructions!
- We'll also see how helpful GDB can be when tracing through assembly.



factorial.c and factorial

# 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.

**00000000004005b6 <sum\_array>:**

4005b6:	ba 00 00 00 00	mov	\$0x0,%edx
4005bb:	b8 00 00 00 00	mov	\$0x0,%eax
4005c0:	eb 09	jmp	4005cb <sum_array+0x15>
4005c2:	48 63 ca	movslq	%edx,%rcx
4005c5:	03 04 8f	add	(%rdi,%rcx,4),%eax
4005c8:	83 c2 01	add	\$0x1,%edx
4005cb:	39 f2	cmp	%esi,%edx
4005cd:	7c f3	j1	4005c2 <sum_array+0xc>
4005cf:	f3 c3	repz	retq

# Lecture Plan

- Revisiting %rip 5
- Calling Functions 19
  - The Stack 22
  - Passing Control 36
  - Passing Data 44
  - Local Storage 65
- Register Restrictions 69
- Pulling it all together: recursion example 78
- **Optimizations** 81
- Live session slides 93

```
cp -r /afs/ir/class/cs107/lecture-code/lect13 .
```

# Optimizations you'll see

## **nop**

- **nop/nopl** are “no-op” instructions – they do nothing!
- Intent: Make functions align on address boundaries that are nice multiples of 8.
- “Sometimes, doing nothing is how to be most productive” – Philosopher Nick

## **mov %ebx,%ebx**

- Zeros out the top 32 register bits (because a mov on an e-register zeros out rest of 64 bits).

## **xor %ebx,%ebx**

- Optimizes for performance as well as code size (read more [here](#)):

```
b8 00 00 00 00
31 c0
```

```
mov $0x0,%eax
xor %eax,%eax
```

# GCC For Loop Output

## GCC For Loop Output

Initialization

Jump to test

Body

Update

Test

Jump to body if success

## Possible Alternative?

Initialization

Test

Jump past loop if fails

Body

Update

Jump to test

# GCC For Loop Output

## GCC For Loop Output

Initialization

Jump to test

Body

Update

Test

Jump to body if success

```
for (int i = 0; i < n; i++)           // n = 100
```

# GCC For Loop Output

## GCC For Loop Output

Initialization

Jump to test

Body

Update

Test

Jump to body if success

```
for (int i = 0; i < n; i++)           // n = 100
```

Initialization

Jump to test

Test

Jump to body

Body

Update

Test

Jump to body

Body

Update

Test

Jump to body

...

# GCC For Loop Output

## GCC For Loop Output

Initialization

Jump to test

Body

Update

Test

Jump to body if success

```
for (int i = 0; i < n; i++)           // n = 100
```

```
Initialization
```

```
Jump to test
```

```
Test
```

```
Jump to body
```

```
Body
```

```
Update
```

```
Test
```

```
Jump to body
```

```
Body
```

```
Update
```

```
Test
```

```
Jump to body
```

```
...
```

# GCC For Loop Output

```
for (int i = 0; i < n; i++)           // n = 100
```

Initialization

Test

No jump

Body

Update

Jump to test

Test

No jump

Body

Update

Jump to test

...

**Possible Alternative?**

Initialization

Test

Jump past loop if fails

Body

Update

Jump to test

# GCC For Loop Output

```
for (int i = 0; i < n; i++)           // n = 100
```

Initialization

Test

No jump

Body

Update

Jump to test

Test

No jump

Body

Update

Jump to test

...

**Possible Alternative?**

Initialization

Test

Jump past loop if fails

Body

Update

Jump to test

# GCC For Loop Output

## GCC For Loop Output

Initialization

Jump to test

Body

Update

Test

Jump to body if success

## Possible Alternative?

Initialization

Test

Jump past loop if fails

Body

Update

Jump to test

Which instructions are better when  $n = 0$ ?  $n = 1000$ ?

```
for (int i = 0; i < n; i++)
```

# Optimizing Instruction Counts

- Both versions have the same **static instruction count** (# of written instructions).
- But they have different **dynamic instruction counts** (# of executed instructions when program is run).
  - If  $n = 0$ , right (possible alternative) is best b/c fewer instructions
  - If  $n$  is large, left (gcc is best) is best b/c fewer instructions
- The compiler may emit a static instruction count that is several times longer than an alternative, but it may be more efficient if loop executes many times.
- Does the compiler *know* that a loop will execute many times? (in general, no)
- So what if our code had loops that always execute a small number of times? How do we know when gcc makes a bad decision?
- (take EE108, EE180, CS316 for more!)

# Optimizations

- **Conditional Moves** can sometimes eliminate “branches” (jumps), which are particularly inefficient on modern computer hardware.
- Processors try to *predict* the future execution of instructions for maximum performance. This is difficult to do with jumps.

# Recap

- Revisiting %rip
- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage
- Register Restrictions
- Pulling it all together: recursion example
- Optimizations

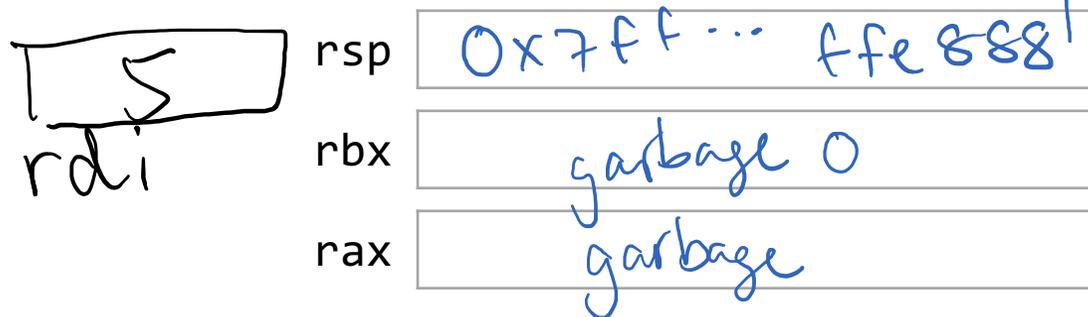
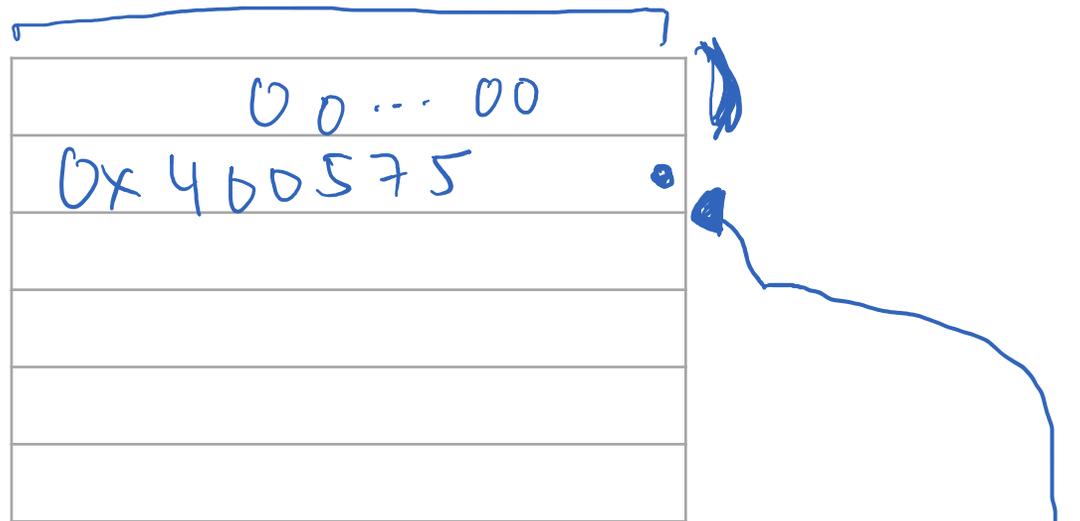
*That's it for assembly!* **Next time:** managing the heap

# Live Session Slides

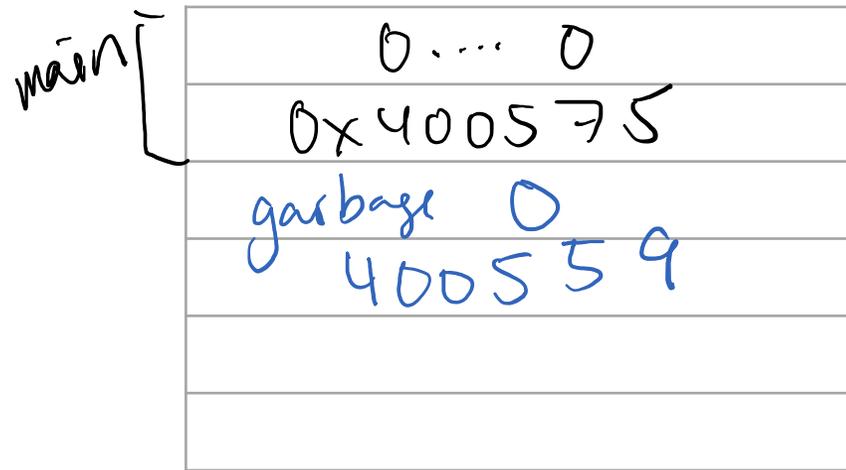
Post any questions you have to today's lecture thread on the discussion forum!

# Factorial review

- At the beginning of the **first** factorial call (factorial(5)), what does the stack look like? *8B*



- At the beginning of the **second** factorial call (factorial(4)), what does the stack look like?



# Useful gdb (really)

- Examine 4 giant words (8 bytes) on the stack:

```
(gdb) x/4g $rsp
```

```
0x7fffffffefe870: 0x0000000000000005          0x000000000000400559
```

```
0x7fffffffefe880: 0x0000000000000000          0x000000000000400575
```

- display/undisplay (prints out things every time you step/next)

```
(gdb) display/4w $rsp
```

```
1: x/4xw $rsp
```

```
0x7fffffffefe8a8:
```

```
0xf7a2d830
```

```
0x00007fff
```

```
0x00000000
```

```
0x00000000
```

*break .....*  
Continue  
exit

*next*  
Step  
finish

ni  
Si

*break ""  
delete 1*

# Useful gdb (really)

- `stepi/finish`: step into current function call/`return to caller`:

`(gdb) finish`

- Set register values during the run

`(gdb) p $rdi = $rdi + 1`

(Might be useful to write down the original value of `$rdi` somewhere)



- Tui things

- `refresh`

- `focus cmd` – use up/down arrows on gdb command line (vs `focus asm`, `focus regs`)

- `layout regs`, `layout asm`

# Extra Practice – Escape Room 2

main calls escape\_room

success: ret 1

[https://godbolt.org/z/jch\\_D2](https://godbolt.org/z/jch_D2)

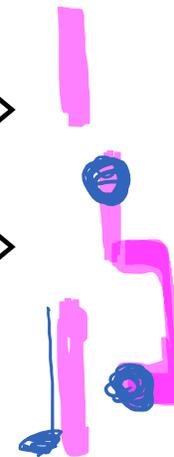


escape\_room

# Escape room assembly code

```
0000000000400579 <escape_room>:
400579:   sub    $0x8,%rsp
40057d:   mov    $0xa,%edx
400582:   mov    $0x0,%esi
400587:   callq  400450 <strtol@plt>
40058c:   mov    %eax,%edi
40058e:   callq  400566 <transform>
400593:   test   $0x1,%al
400595:   jne    40059e <escape_room+0x25>
400597:   mov    $0x1,%eax
40059c:   jmp    4005a3 <escape_room+0x2a>
40059e:   mov    $0x0,%eax
4005a3:   add    $0x8,%rsp
4005a7:   retq
```

*\$rdi*



*cmp jne \$4,%rdi*  
*%rdi - \$4 != 0*

*test jne \$4,%rdi*  
*\$4 & %rdi*

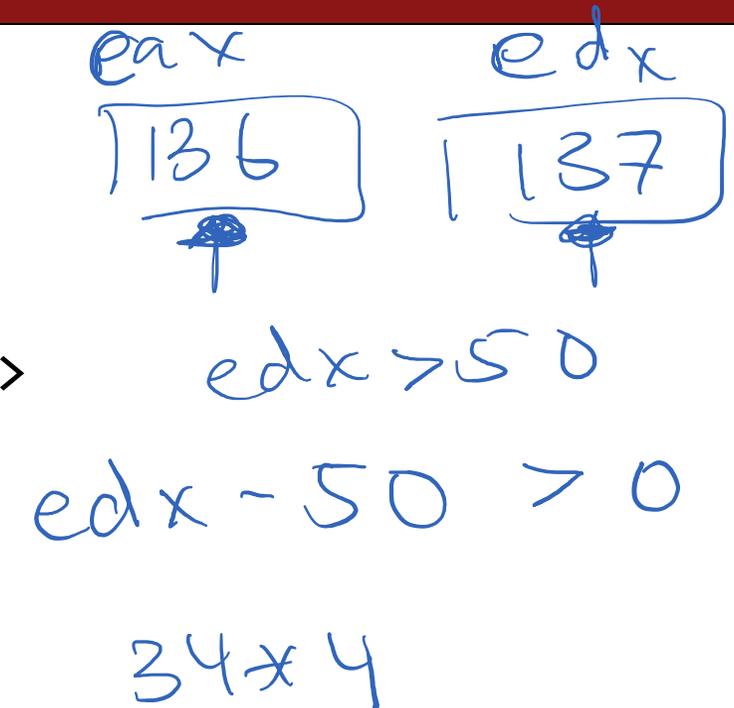
*8 0 0 0 1 0 0*  
*0 0 0 0 0 0*

*\$1 & %al*

*8 0 0 0 0 1*  
*0 0 0 0 0 0*

# Escape room assembly code

```
0000000000400566 <transform>:  
400566:    lea    0x0(,%rdi,4),%eax  
40056d:    lea    0x1(%rax),%edx  
400570:    cmp    $0x32,%edx  
400573:    jg     400577 <transform+0x11>  
400575:    mov    %edx,%eax  
400577:    repz  retq
```



- <https://stackoverflow.com/questions/64276500/purpose-of-sub-0x8-rsp-although-there-is-no-local-variable-to-store>
- <https://www.freebsd.org/cgi/man.cgi>