

# COMP201

## Computer Systems & Programming

Lecture #24 – x86-64 Procedures



**KOÇ**  
**UNIVERSITY**

Aykut Erdem // Koç University // Fall 2020

# Recap

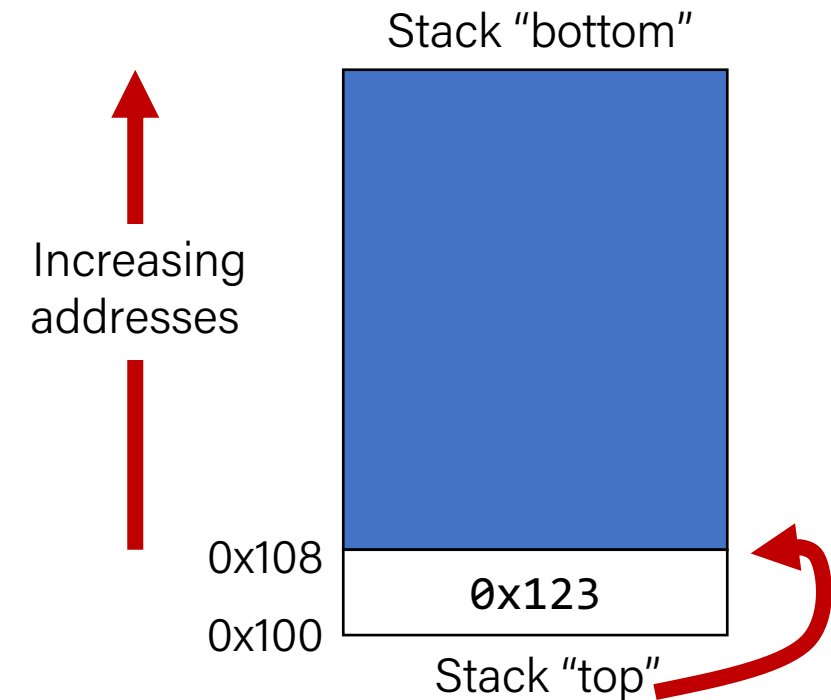
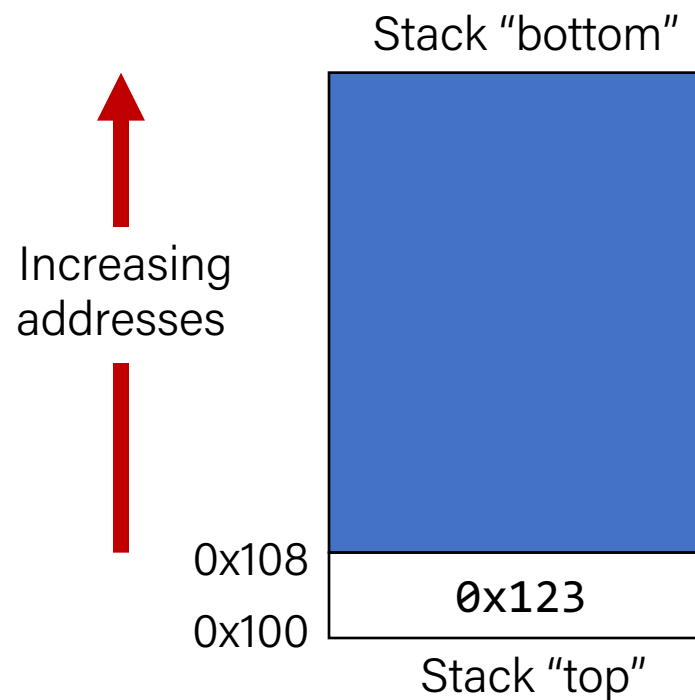
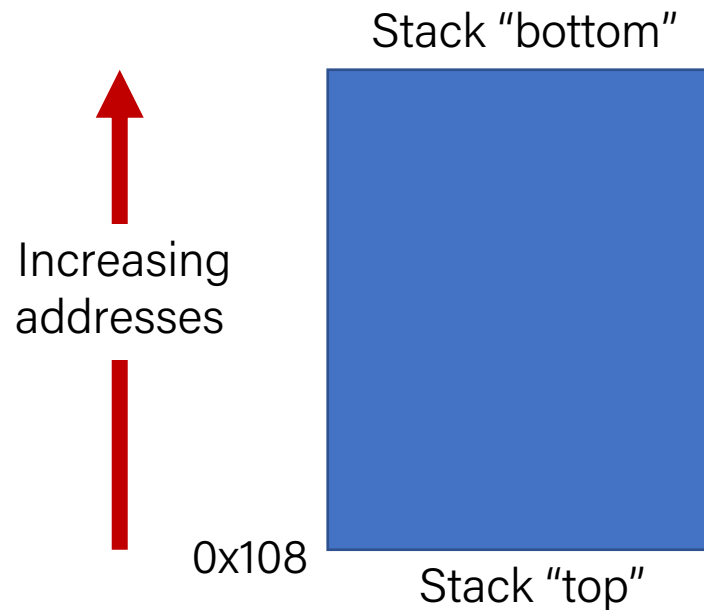
- Revisiting `%rip`
- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage

# Recap: The Stack

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

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

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



# Recap: 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.



# Recap: 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).

# Plan for Today

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

# Lecture Plan

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


# Parameters and Return

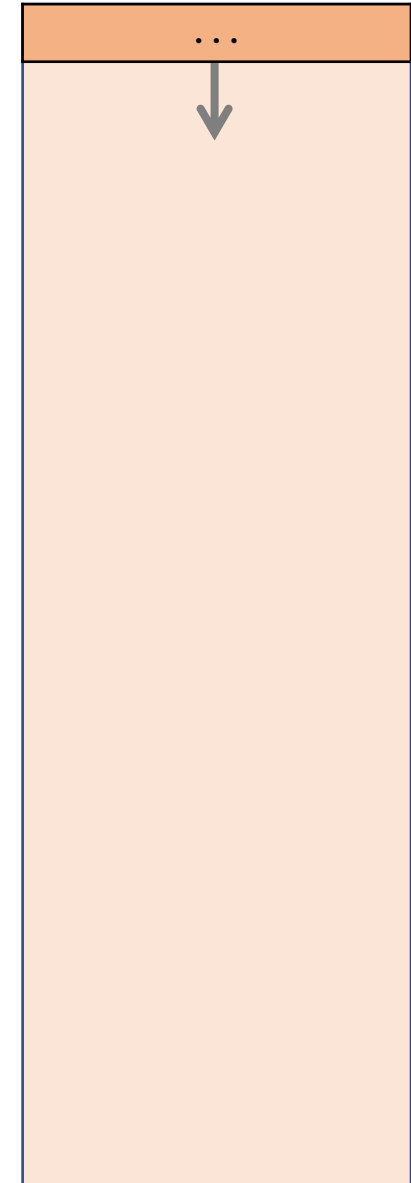
- There are special registers that store parameters and the return value.
- To call a function, we must put any parameters we are passing into the correct registers. (`%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`, in that order)
- Parameters beyond the first 6 are put on the stack.
- If the caller expects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

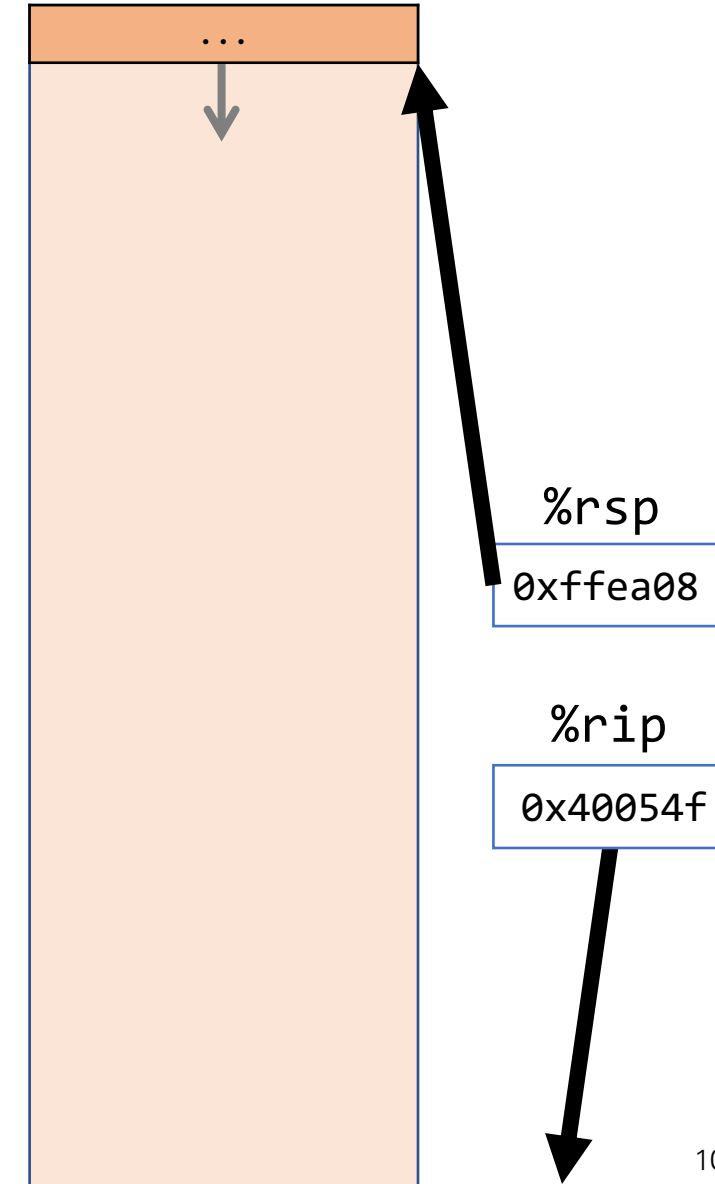


# Parameters and Return

```
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() 



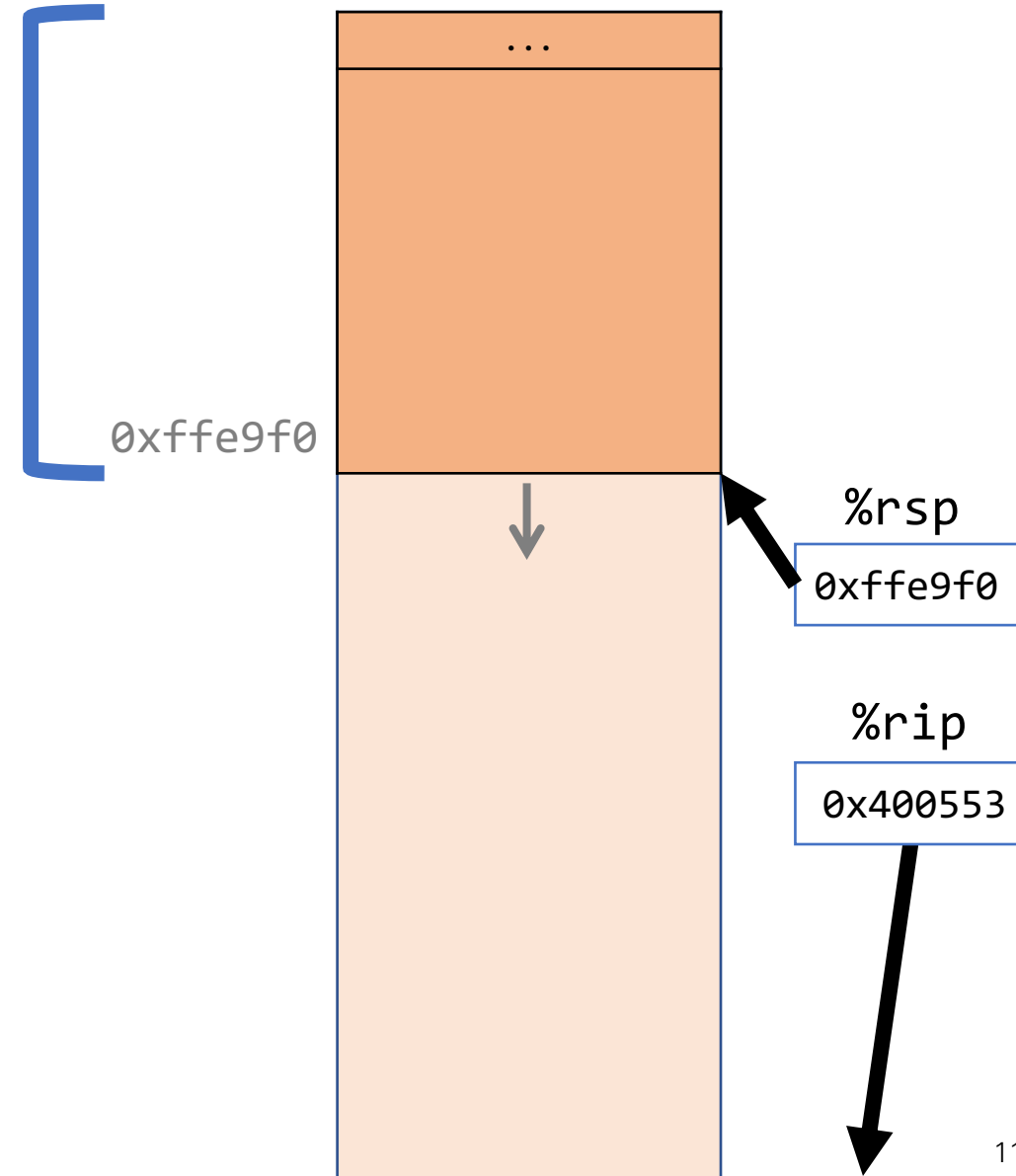
```
0x40054f <+0>:    sub    $0x18,%rsp
0x400553 <+4>:    movl   $0x1,0xc(%rsp)
0x40055b <+12>:   movl   $0x2,0x8(%rsp)
0x400563 <+20>:   movl   $0x3,0x4(%rsp)
0x40056b <+28>:   movl   $0x4,0x0(%rsp)
```

# 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()

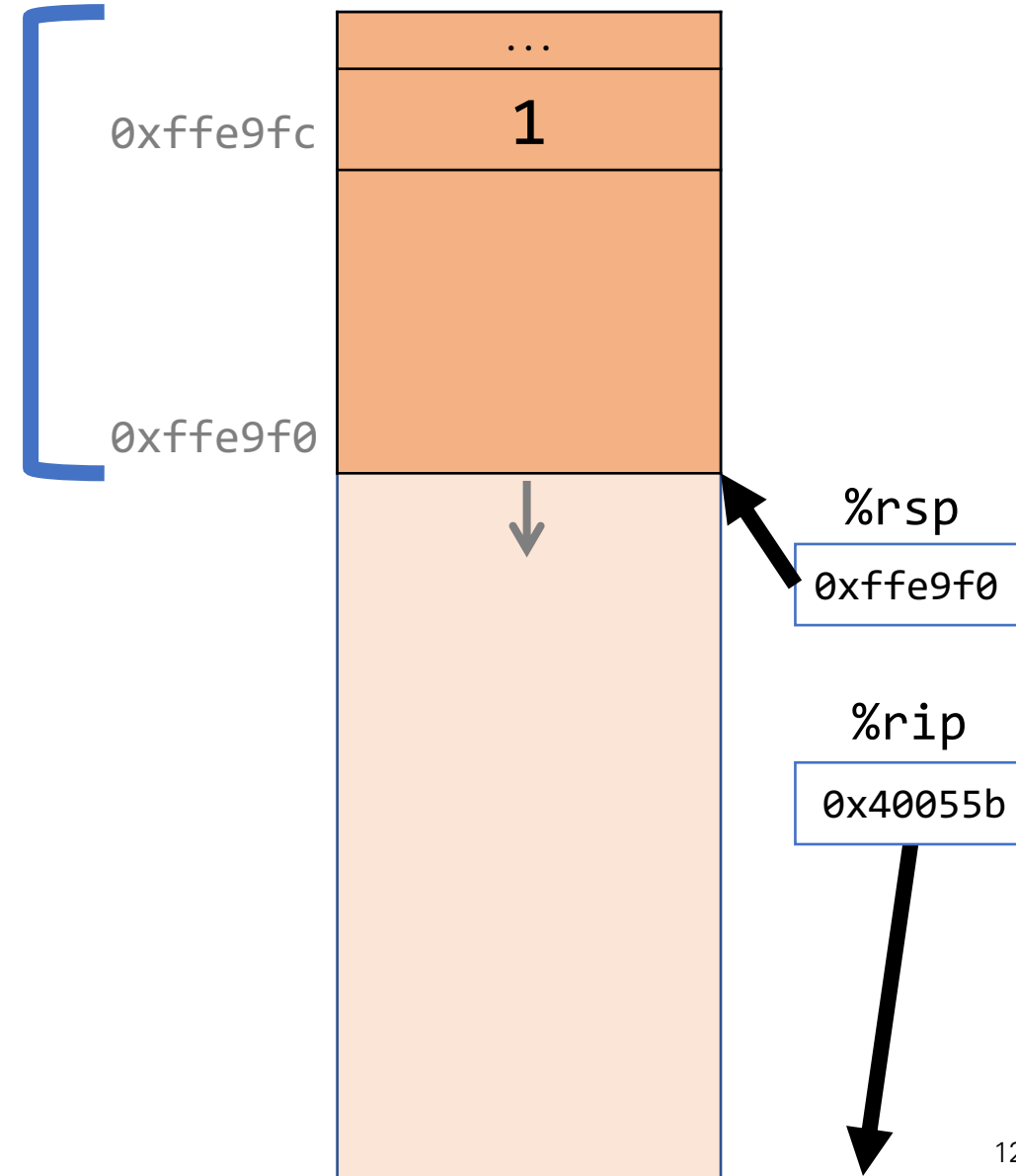


# Parameters and Return

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

```
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()

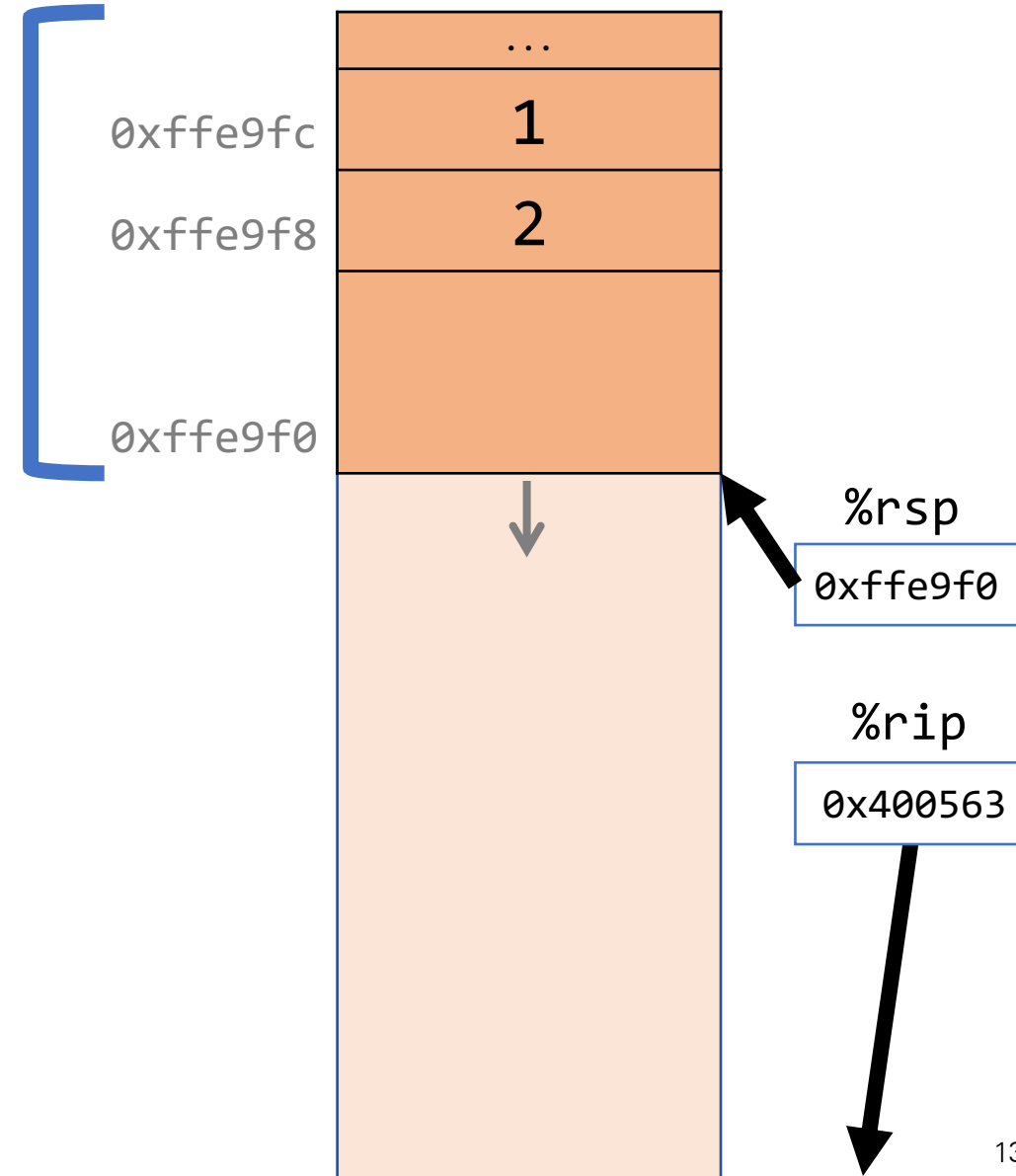


# Parameters and Return

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

```
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()

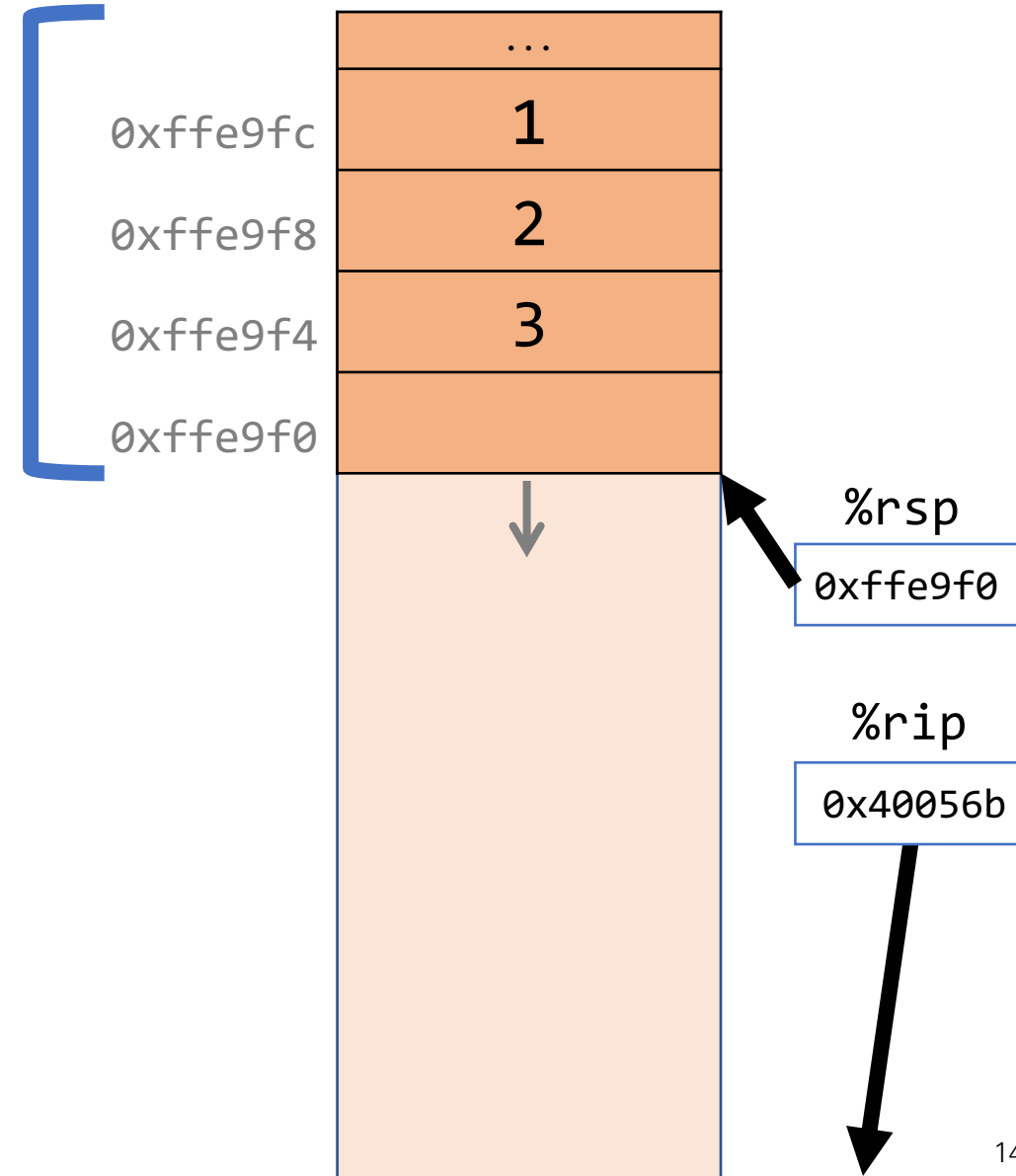


# Parameters and Return

```
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()



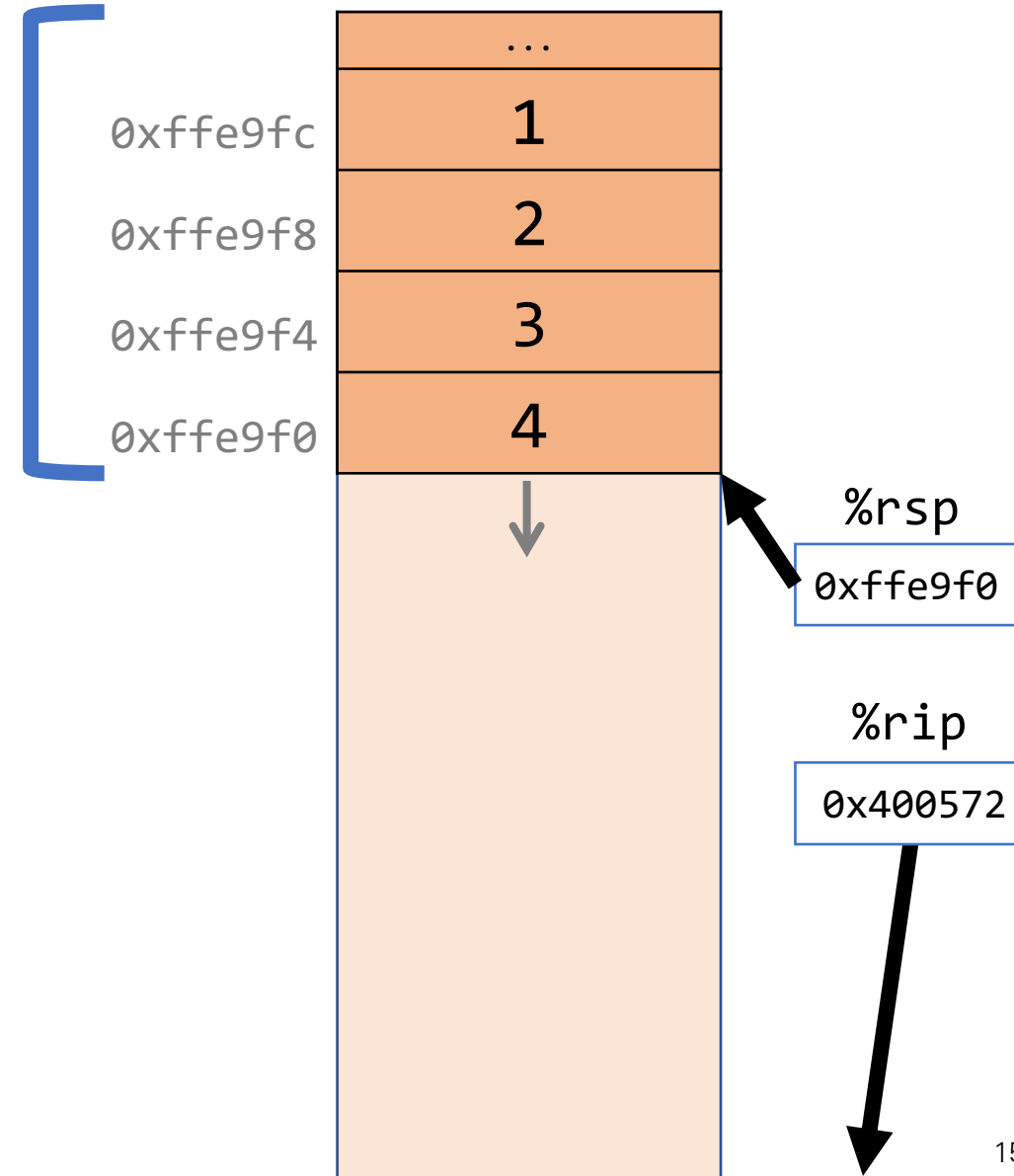


# Parameters and Return

```
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()



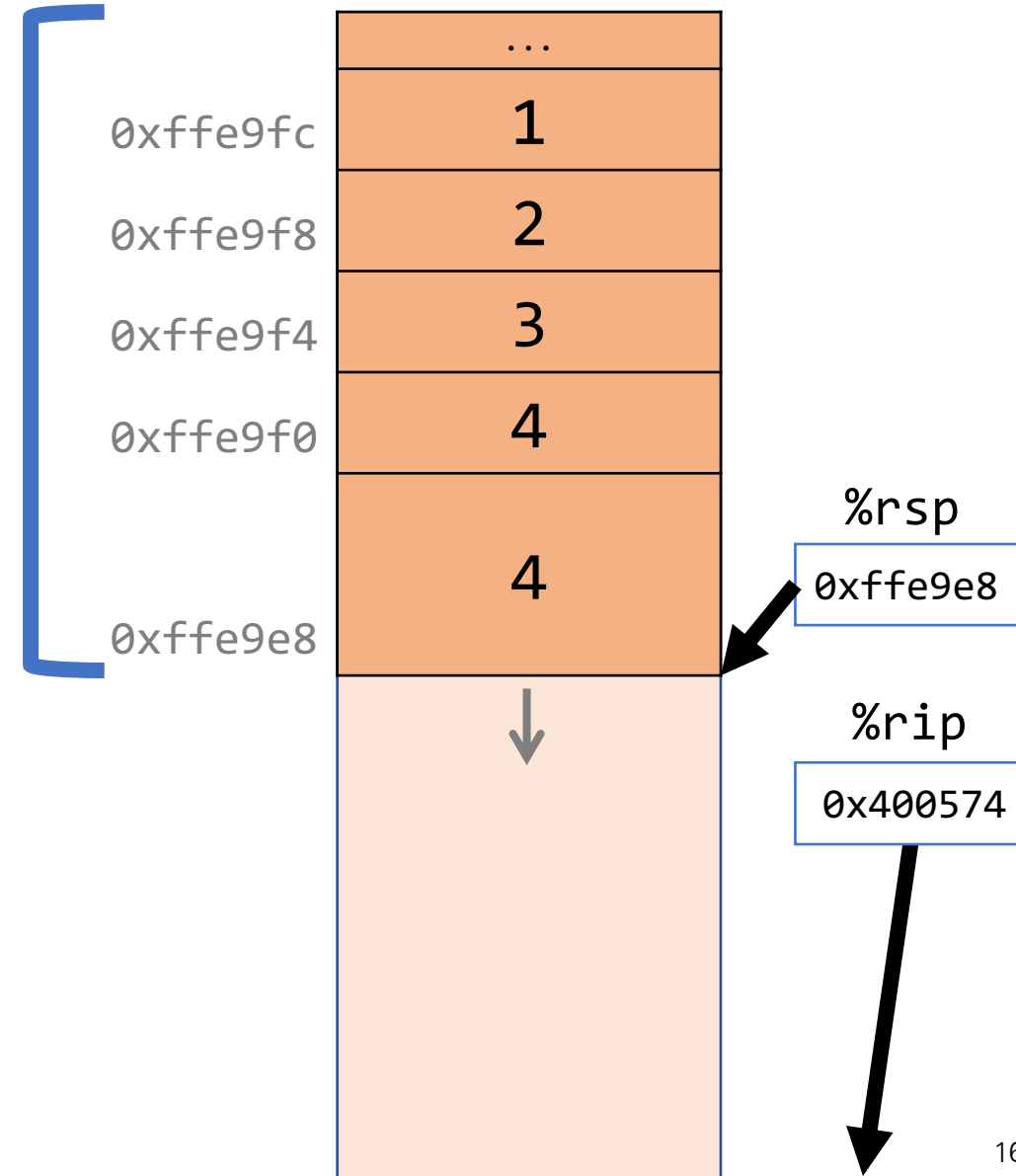
# Parameters and Return

```
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>: mov     $0x2,%rax
```

main()



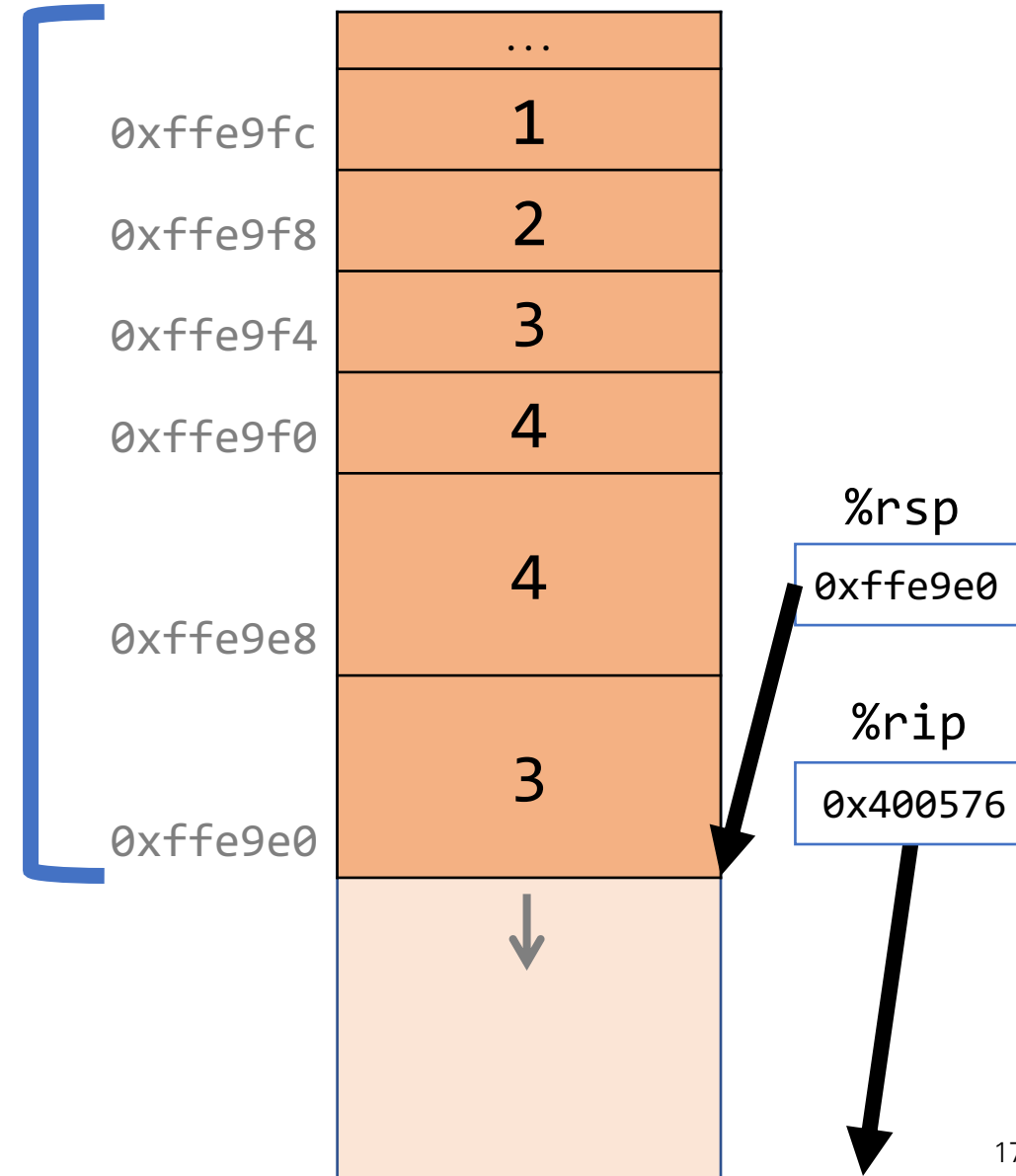
# Parameters and Return

```
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()

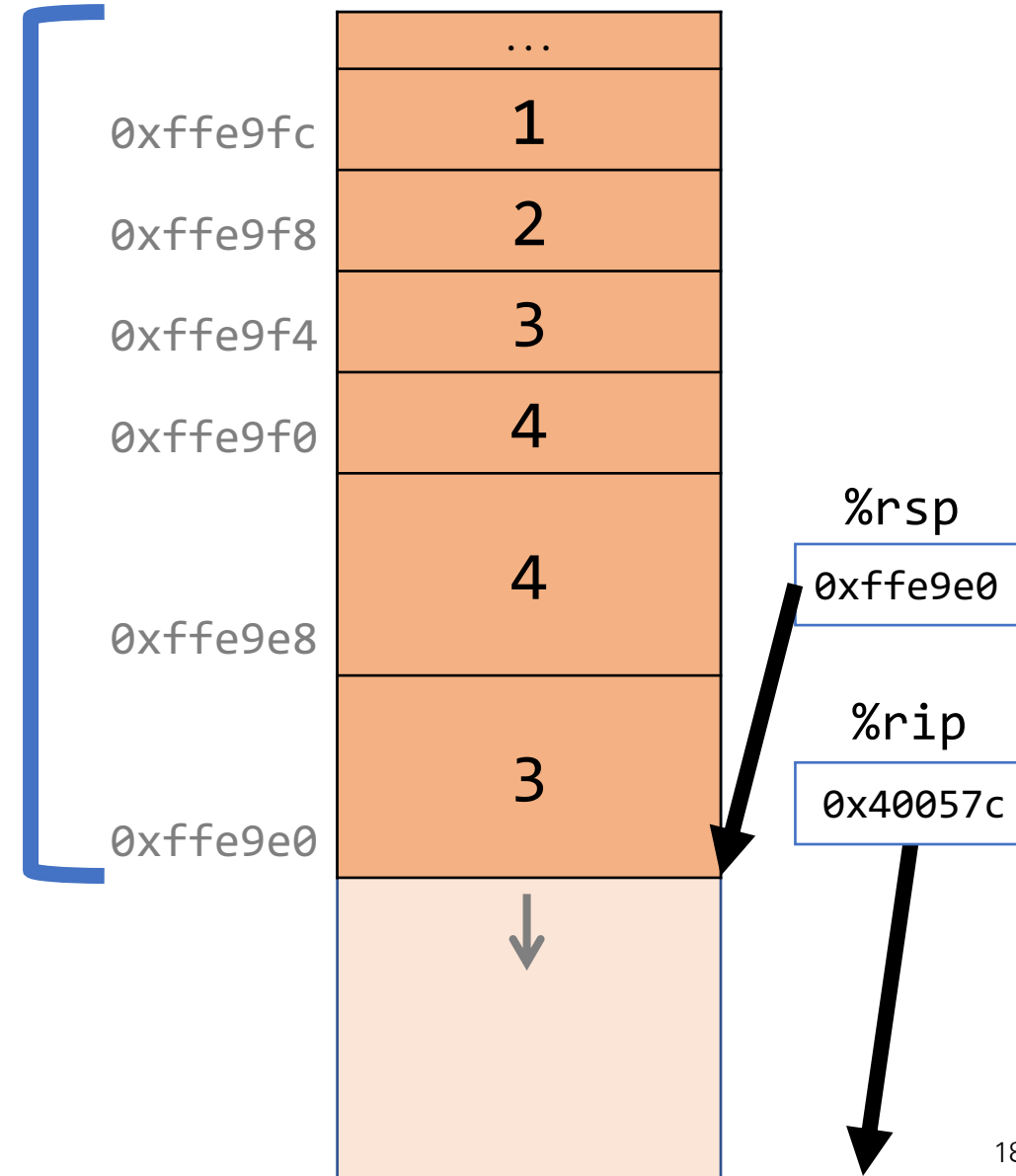


# Parameters and Return

```
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()

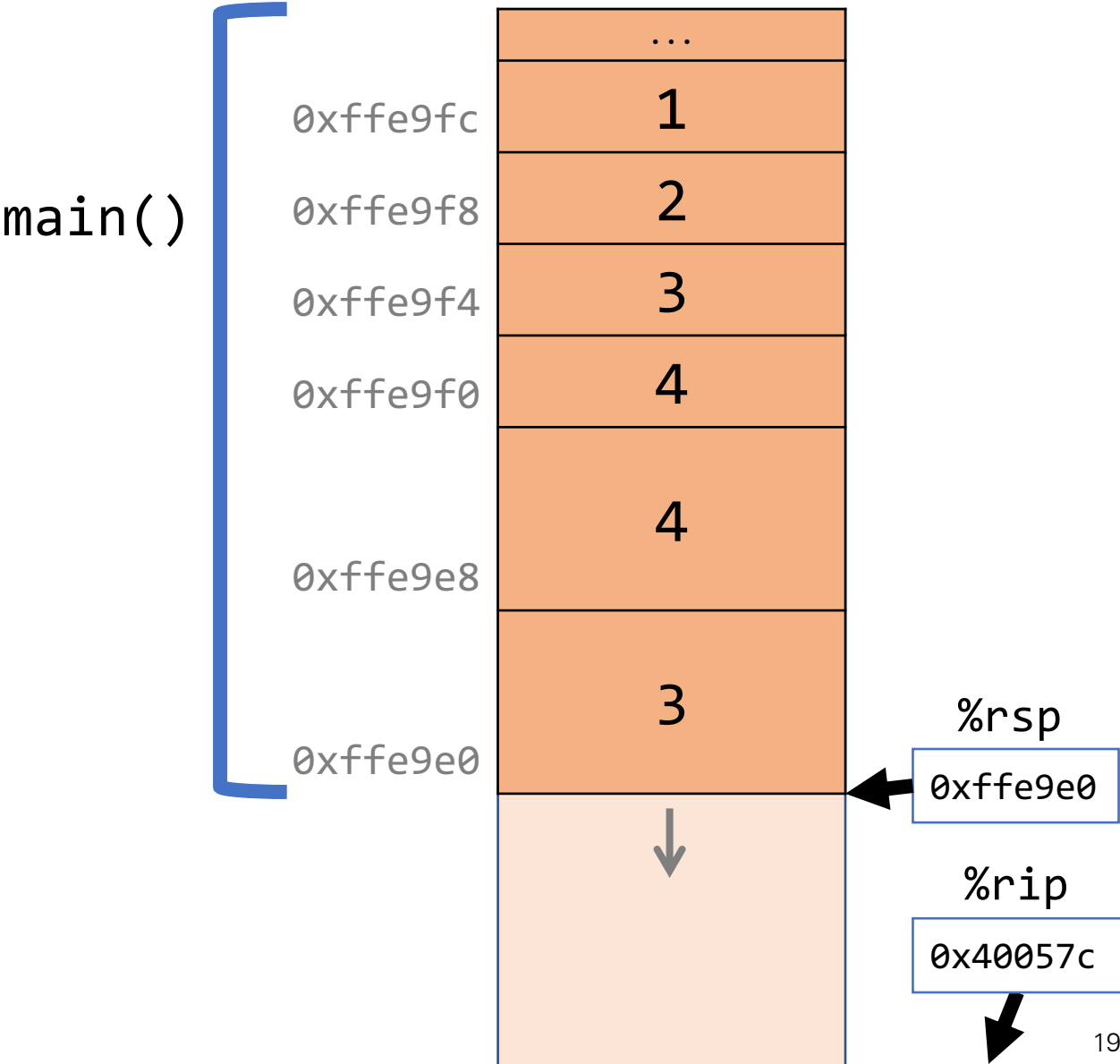


# Parameters and Return

```
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
```

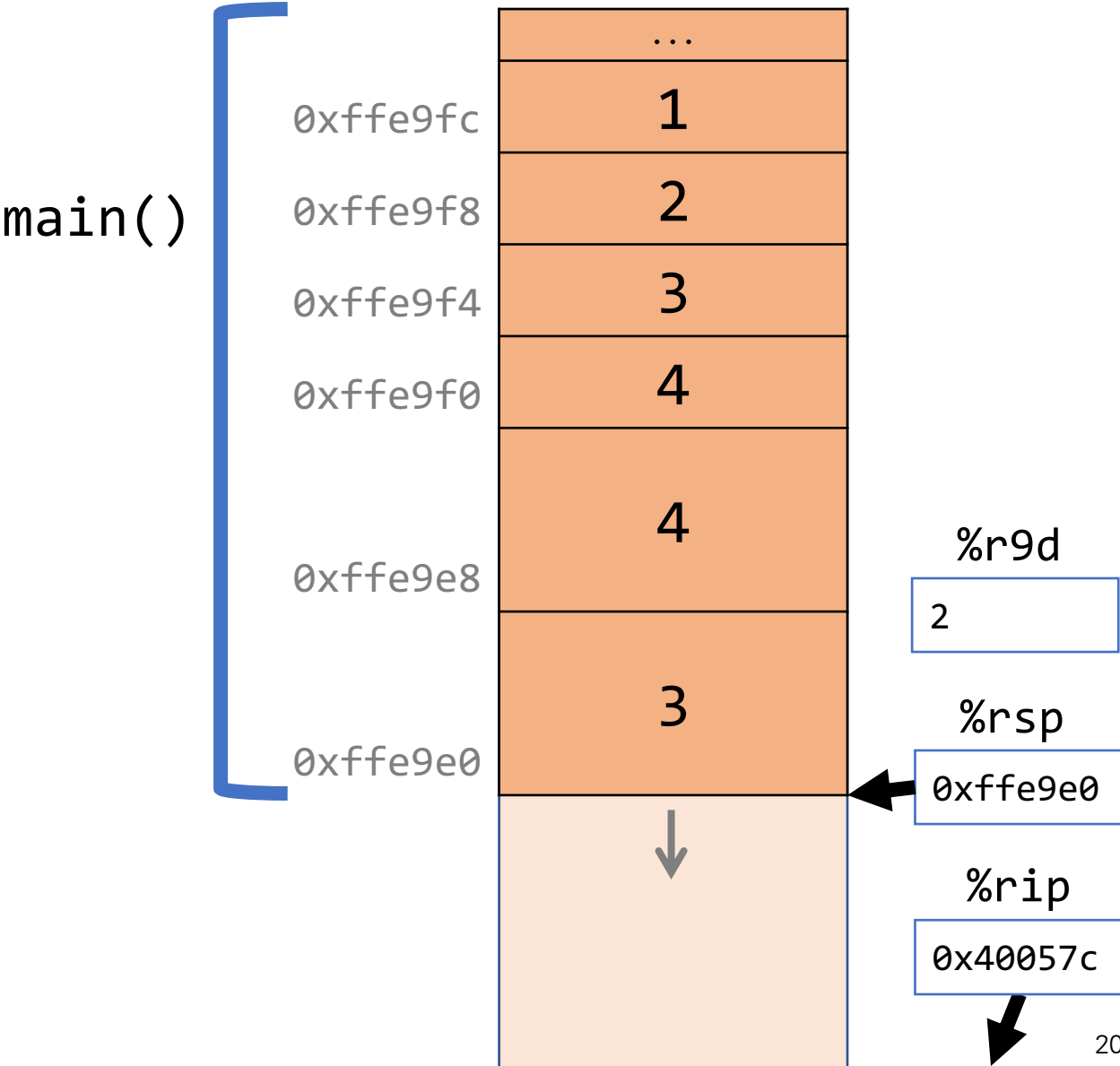


# Parameters and Return

```
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
```



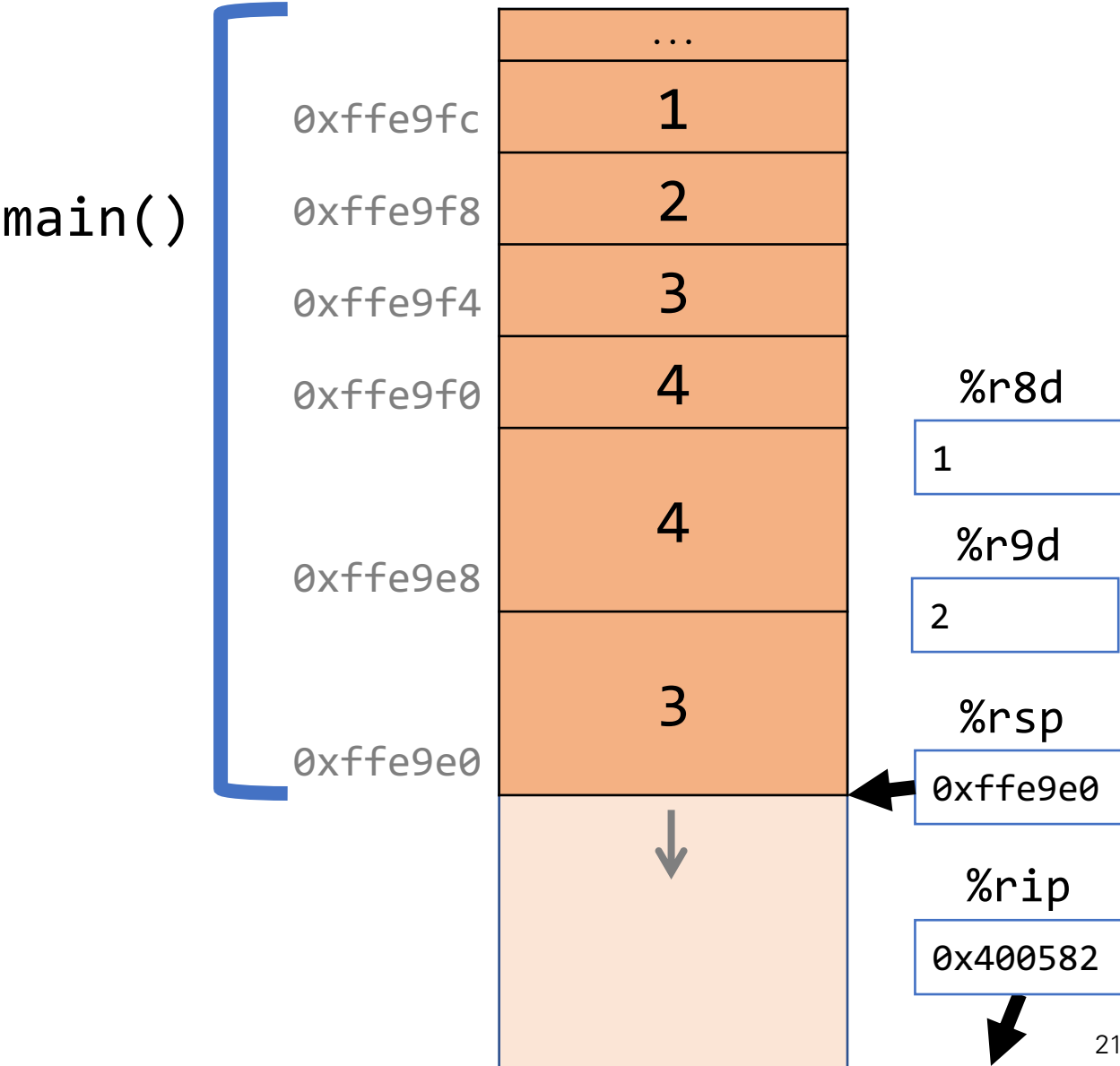


# Parameters and Return

```
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
```

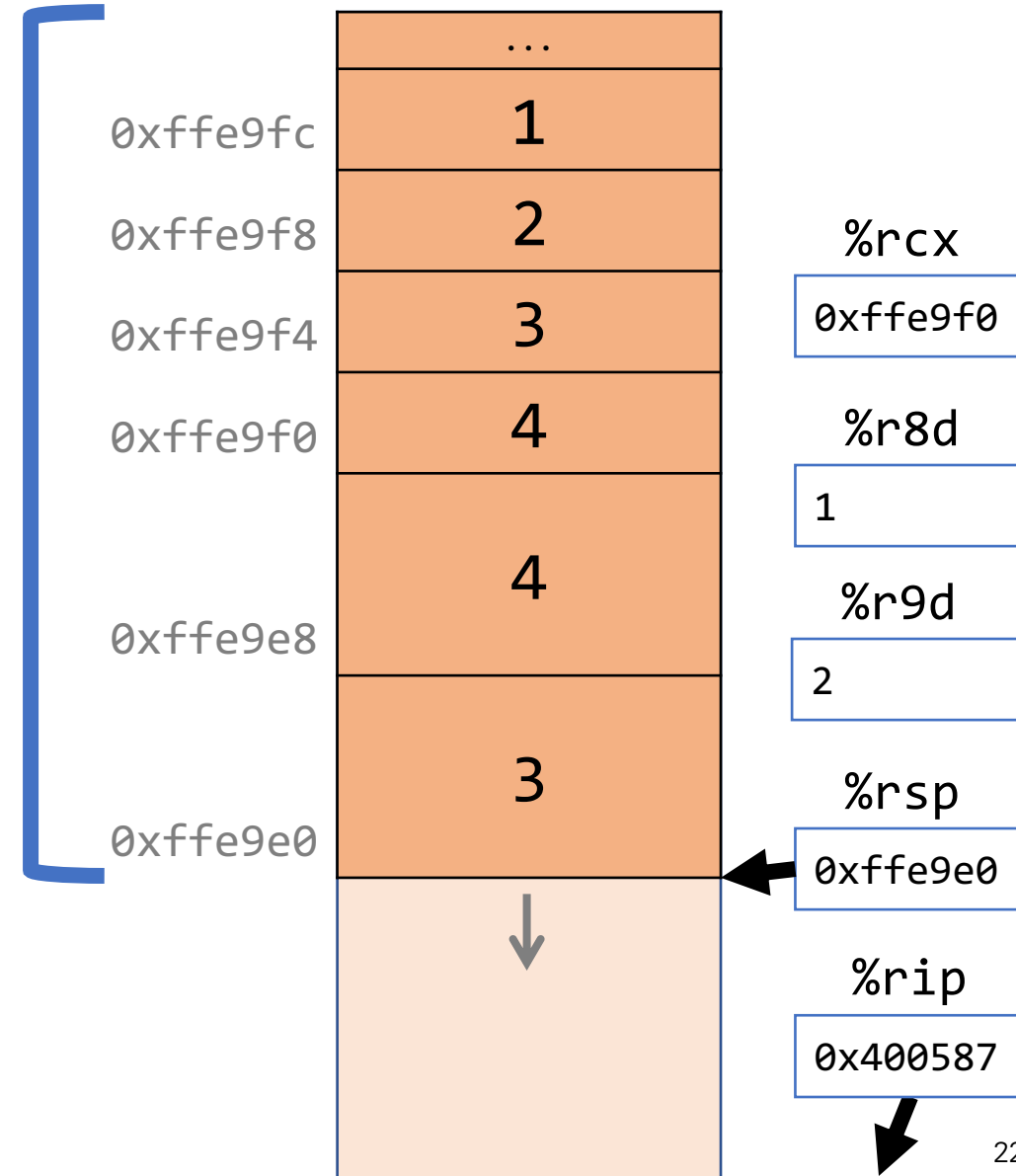


# Parameters and Return

```
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
```

main()

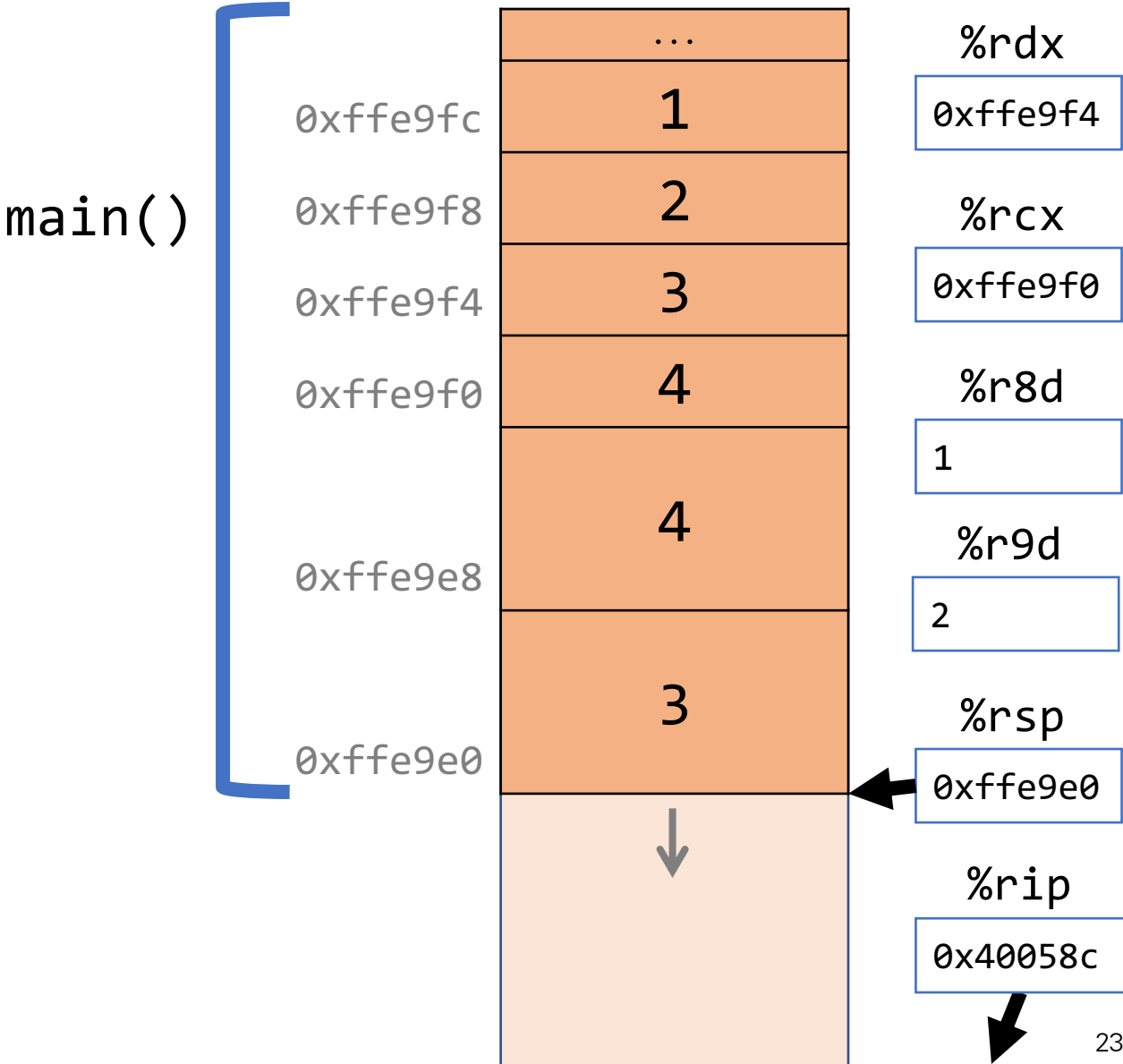


# Parameters and Return

```
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
```



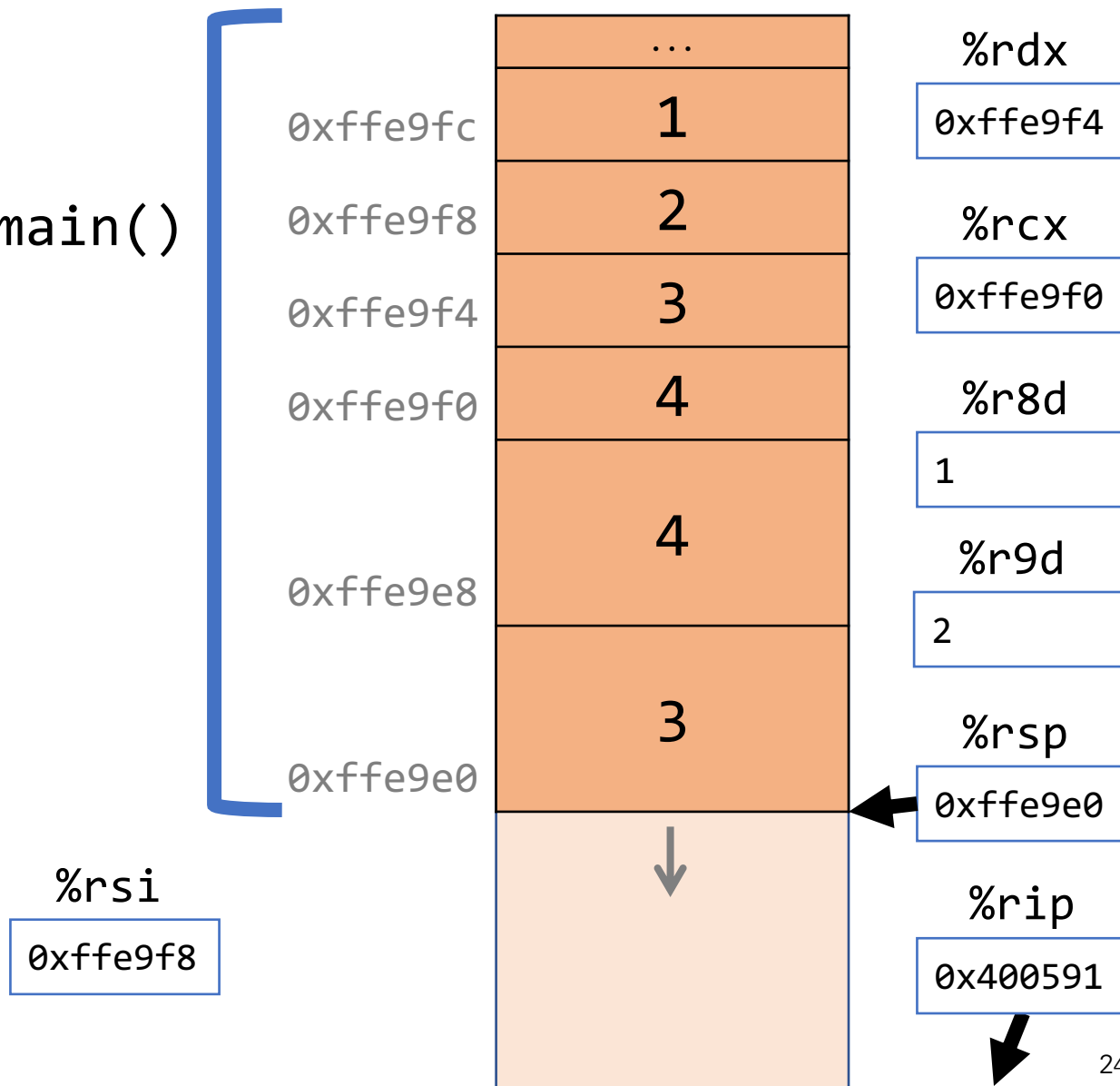
# Parameters and Return

```
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>
```

main()

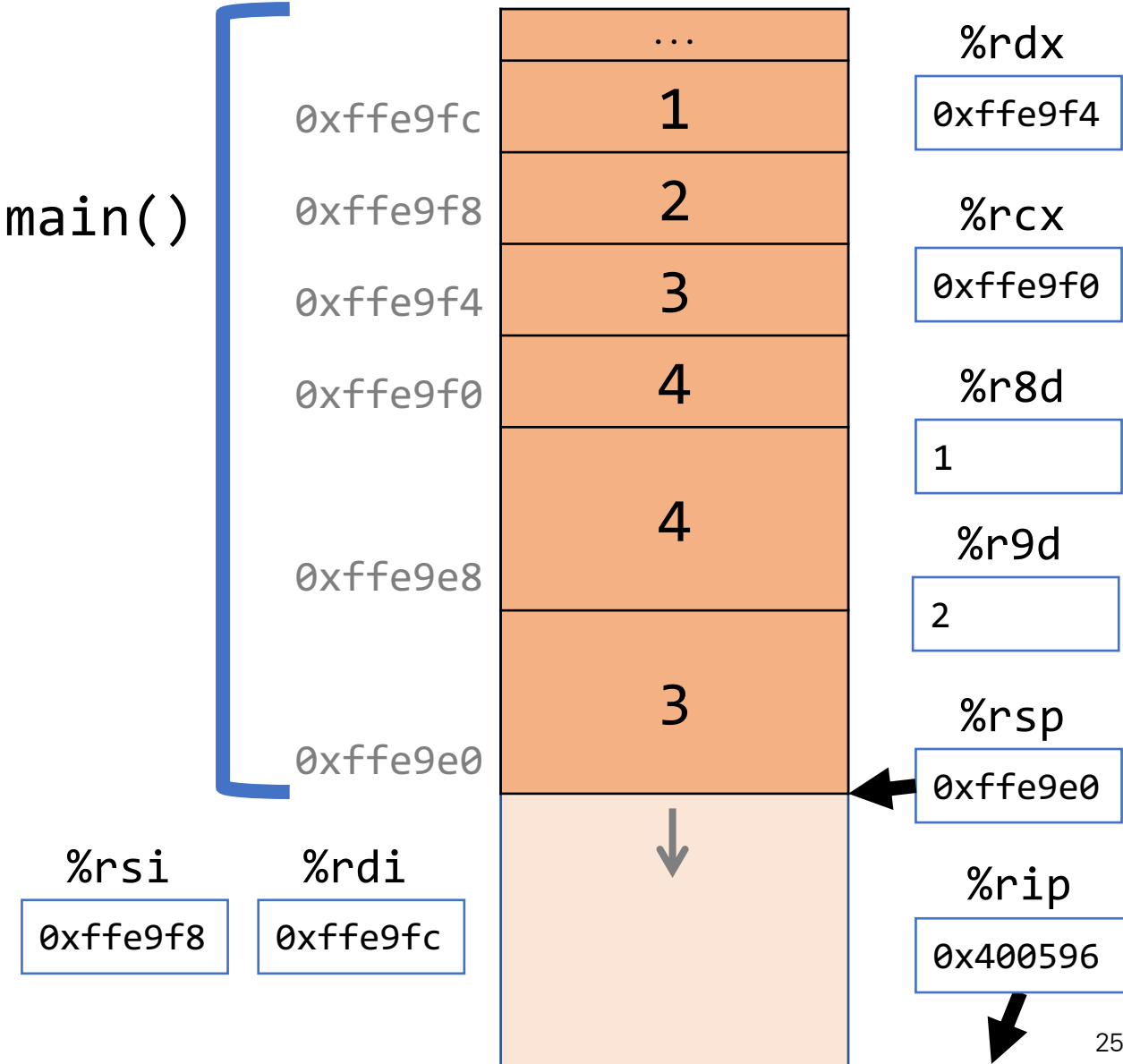


# Parameters and Return

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

```
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
```

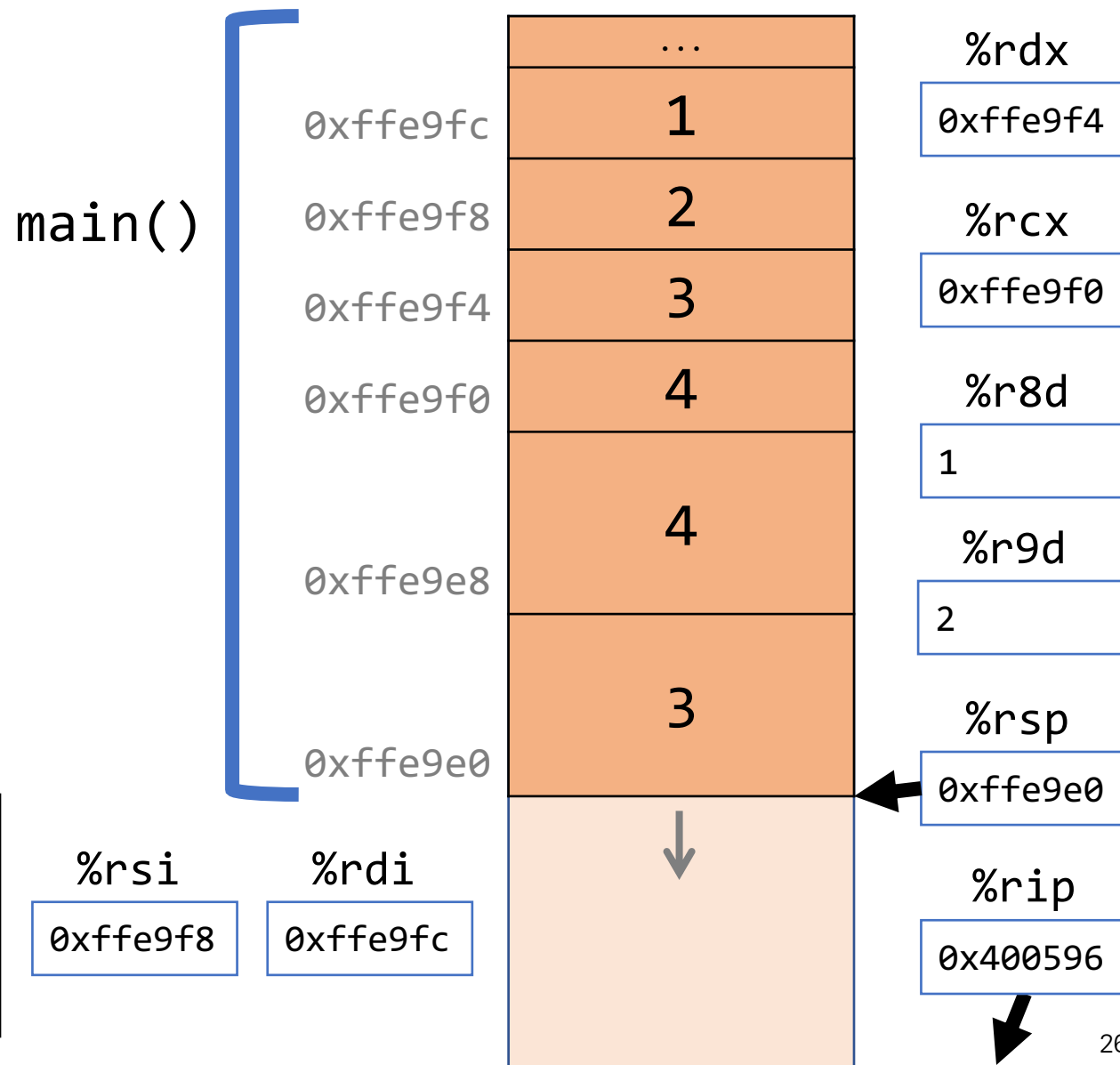


# Parameters and Return

```
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
...
```



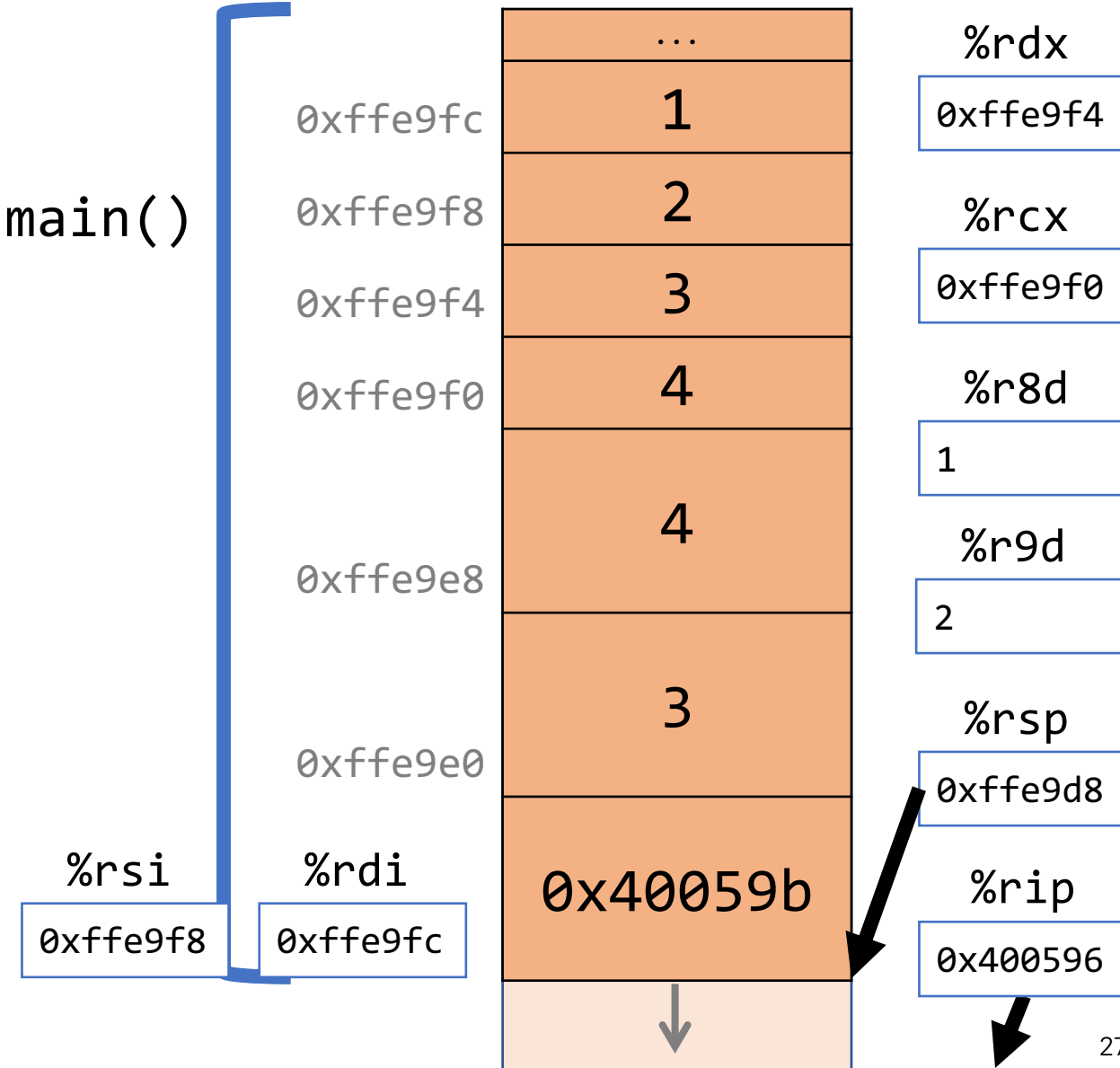


# Parameters and Return

```
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

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

# 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)
```

# Question Break



# Lecture Plan

- Revisiting `%rip`
- Calling Functions
- Register Restrictions
- Pulling it all together: recursion example

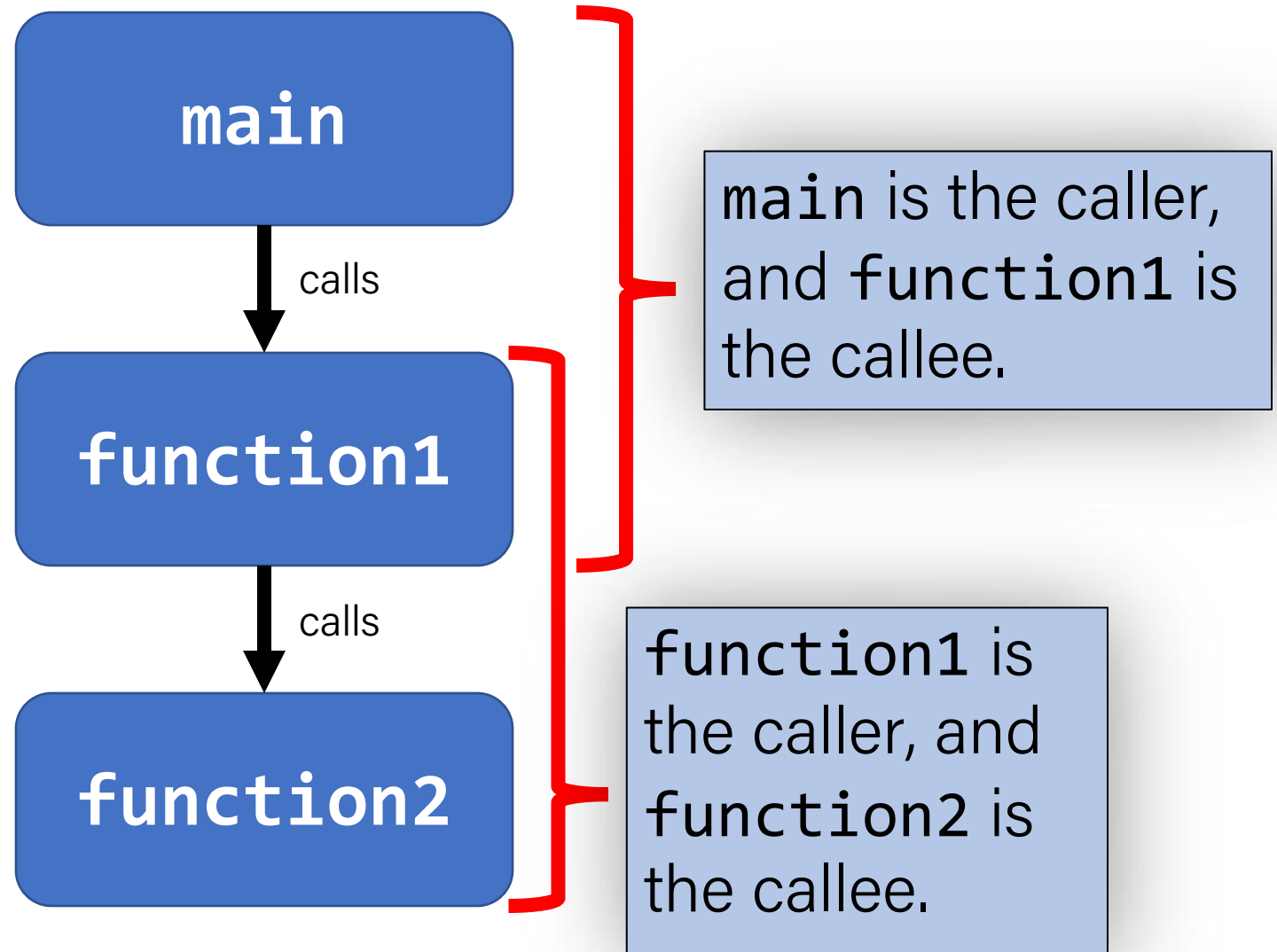
# Register Restrictions

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

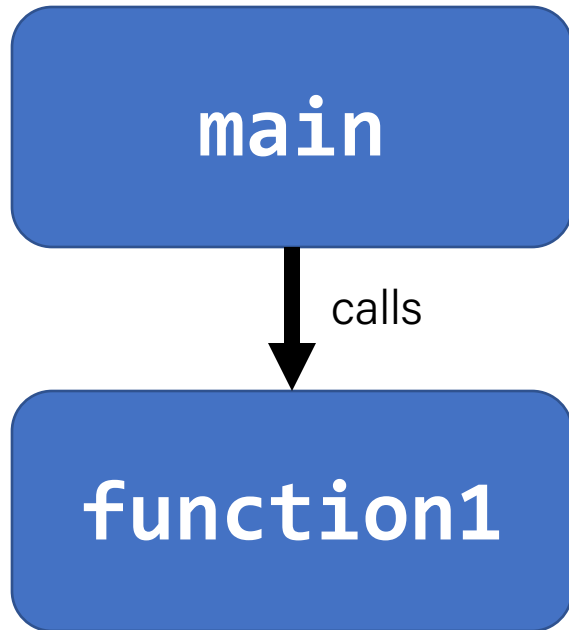
## Caller-Owned

- 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.

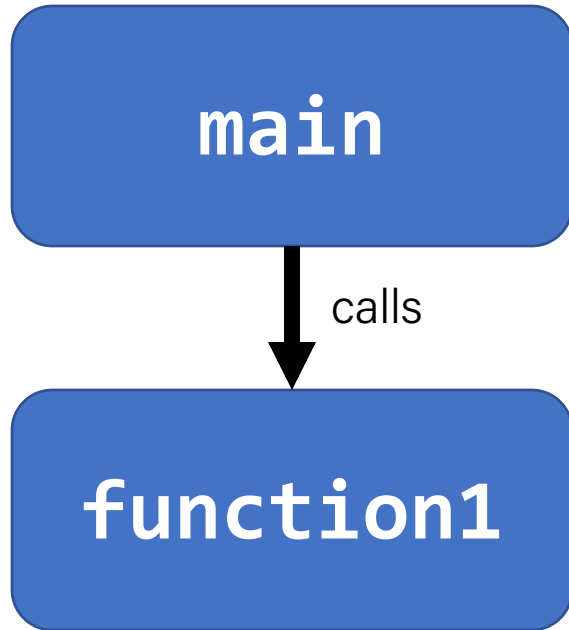
# Caller-Owned Registers



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

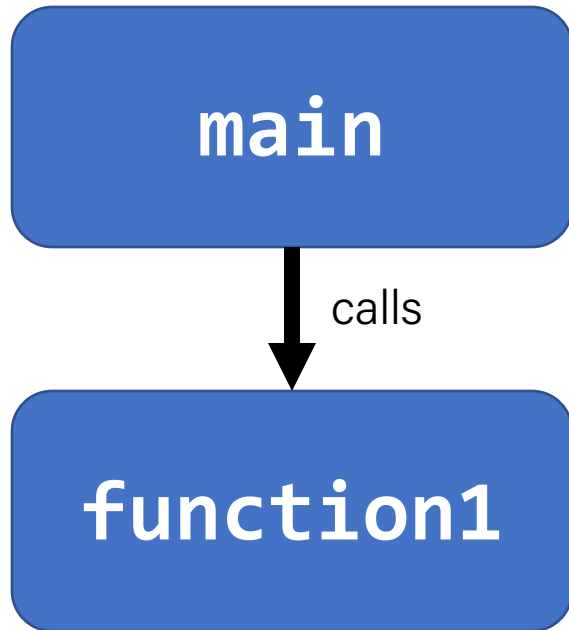
If `function1` wants to use any caller-owned registers, it must save the existing values and restore them before returning.

# Caller-Owned Registers



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

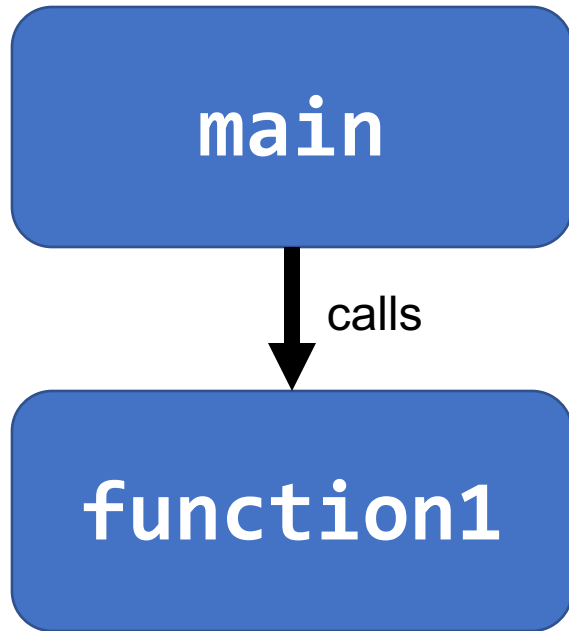
# Callee-Owned Registers



`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.

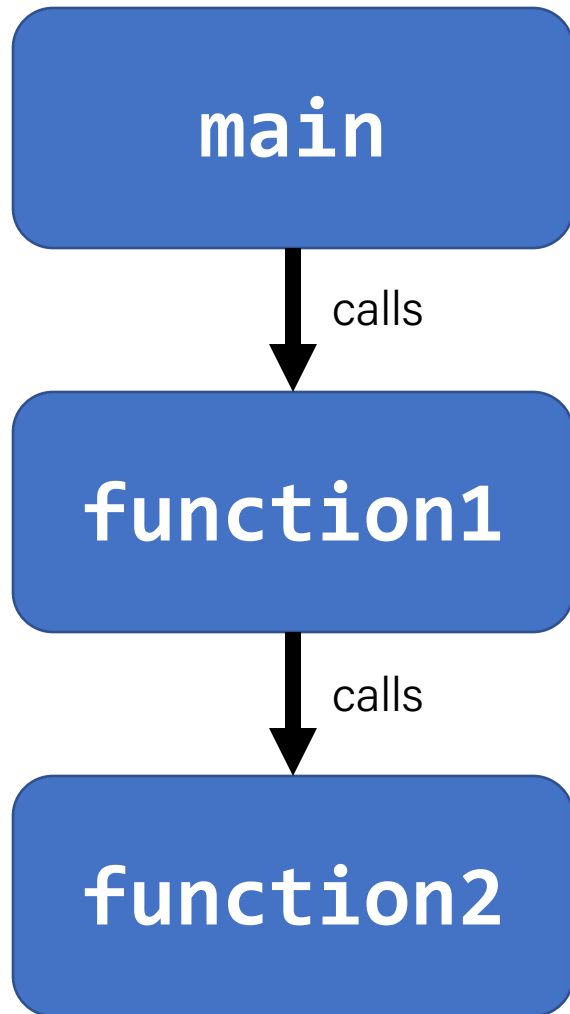
# Callee-Owned Registers



```
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.

# Question Break

# Lecture Plan

- Revisiting `%rip`
- Calling Functions
- Register Restrictions
- Pulling it all together: recursion example

# 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!

<https://godbolt.org/z/f43dz1>



`factorial.c` and `factorial`

# Extra Practice – Escape Room 2

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



escape\_room

# 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

# Recap

- Revisiting `%rip`
- Calling Functions
- Register Restrictions
- Pulling it all together: recursion example

**Next time:** Data and Stack Frames