# CS107, Lecture 14

## Calling: Callers, Callees, and Callbacks

Reading: B&O 3.1-3.4

This document is copyright (C) Stanford Computer Science, Adam Keppler and Joel Ramirez, licensed under Creative Commons Attribution 2.5 License. All rights reserved.
Based on slides created by Nick Troccoli and Chris Gregg.

# Warm-up: Reverse Engineering

```
int elem_arithmetic(int nums[], int y) {
    int z = nums[________] * ________;

    z -= ________;

    return ________;
}
----------
// nums in %rdi, y in %esi
elem_arithmetic:
  movl %esi, %eax
  imull 4(%rdi), %eax
  movslq %esi, %rsi
  subl (%rdi,%rsi,4), %eax
  lea 2(%rax, %rax), %eax
  ret
```

# Warm-up: Reverse Engineering

```
int elem_arithmetic(int nums[], int y) {
    int z = nums[1] * y;

    z -= ________;

    return ________;
}
----------
// nums in %rdi, y in %esi
elem_arithmetic:
  movl %esi, %eax            // copy y into %eax
  imull 4(%rdi), %eax        // multiply %eax by nums[1]
  movslq %esi, %rsi          // sign-extend %esi to %rsi
  subl (%rdi,%rsi,4), %eax
  lea 2(%rax, %rax), %eax
  ret
```

**Work through the last two blanks in groups and input your answer for the first blank on PollEv:**
pollev.com/cs107 or text CS107 to 22333 once to join.

# Warm-up: Reverse Engineering

```
int elem_arithmetic(int nums[], int y) {
    int z = nums[1] * y;

    z -= nums[y];

    return 2 * z + 2;
}
----------
// nums in %rdi, y in %esi
elem_arithmetic:
  movl %esi, %eax          // copy y into %eax
  imull 4(%rdi), %eax      // multiply %eax by nums[1]
  movslq %esi, %rsi        // sign-extend %esi to %rsi
  subl (%rdi,%rsi,4), %eax // subtract nums[y] from %eax
  lea 2(%rax, %rax), %eax  // multiply %rax by 2, and add 2
  ret
```

# `test` practice: What’s the C code?

```
0x400546 <test_func>     test   %edi,%edi
0x400548 <test_func+2>   jns    0x400550 <test_func+10>
0x40054a <test_func+4>   mov    $0xfeed,%eax
0x40054f <test_func+9>   retq
0x400550 <test_func+10>  mov    $0xaabbccdd,%eax
0x400555 <test_func+15>  retq
```

🤔

# test practice: What's the C code?

```
0x400546 <test_func>     test   %edi,%edi
0x400548 <test_func+2>   jns    0x400550 <test_func+10>
0x40054a <test_func+4>   mov    $0xfeed,%eax
0x40054f <test_func+9>   retq
0x400550 <test_func+10>  mov    $0xaabbccdd,%eax
0x400555 <test_func+15>  retq
```

```
int test_func(int x) {
    if (x < 0) {
        return 0xfeed;
    }
    return 0xaabbccdd;
}
```

(or anything like this)

# Practice: “Escape Room”

```
<escape_room+0>          lea    (%rdi,%rdi,1),%eax
<escape_room+3>          cmp    $0x5,%eax
<escape_room+6>          jg     0x114c <escape_room+19>
<escape_room+8>          cmp    $0x1,%edi
<escape_room+11>         je     0x1152 <escape_room+25>
<escape_room+13>         mov    $0x0,%eax
<escape_room+18>         retq
<escape_room+19>         mov    $0x1,%eax
<escape_room+24>         retq
<escape_room+25>         mov    $0x1,%eax
<escape_room+30>         retq
```

What must be passed to the escapeRoom function such that it returns true (1) and not false (0)?

You don’t have to reverse-engineer C code exactly!

# Practice: “Escape Room”

```
<escape_room+0>         lea     (%rdi,%rdi,1),%eax
<escape_room+3>         cmp     $0x5,%eax
<escape_room+6>         jg      0x114c <escape_room+19>
<escape_room+8>         cmp     $0x1,%edi
<escape_room+11>        je      0x1152 <escape_room+25>
<escape_room+13>        mov     $0x0,%eax
<escape_room+18>        retq
<escape_room+19>        mov     $0x1,%eax
<escape_room+24>        retq
<escape_room+25>        mov     $0x1,%eax
<escape_room+30>        retq
```

What must be passed to the escapeRoom function such that it returns true (1) and not false (0)?

**First param > 2 or == 1.**

# %rip Recap

- Machine code instructions live in main memory in the text/data segment.
- **%rip** is a special register that stores a number (an address in the text/data segment) that corresponds to the next instruction to execute.
- It marks our place in the program's instructions and advances us instruction by instruction through the program
- Special hardware handles the variable op-code sizes and correctly updates **%rip**.
- **jmp** instructions work by adjusting %rip by a specified amount.

# Calling Functions In Assembly

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

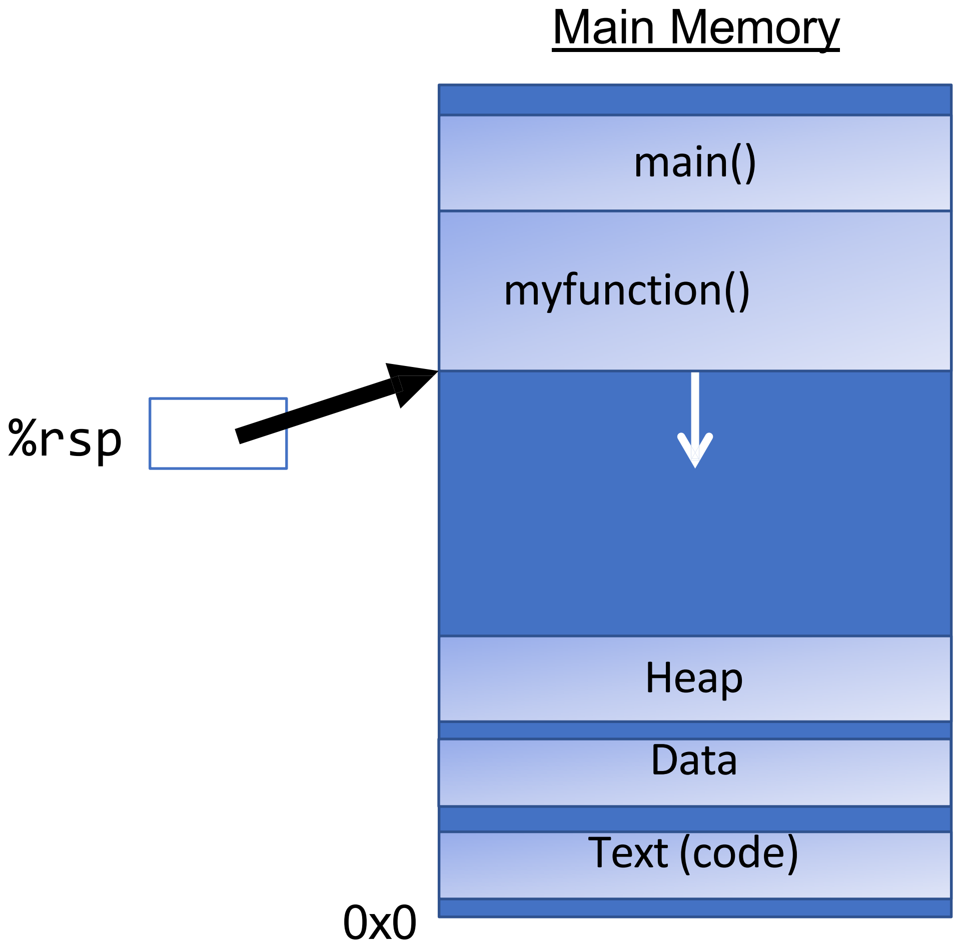
- **Pass Control** – %rip must be adjusted to execute the callee's instructions, and then resume the caller's instructions afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

How does assembly interact with the stack?

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

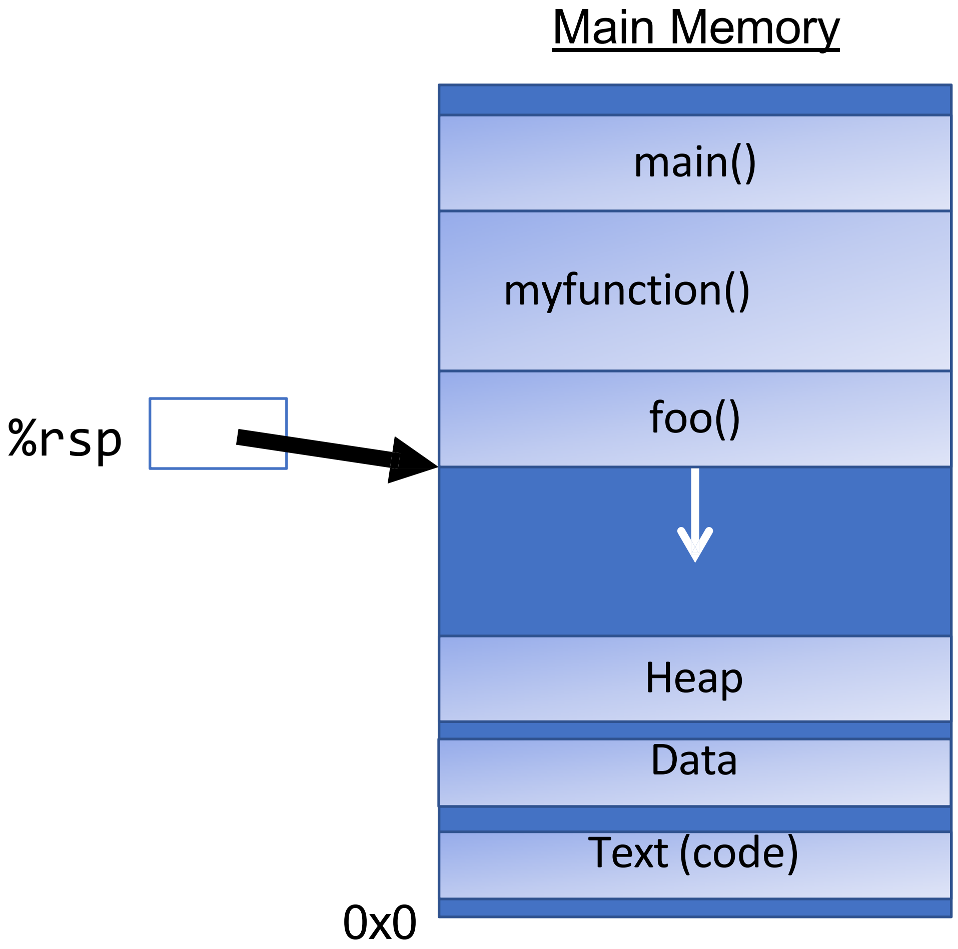
# %rsp

- **%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).



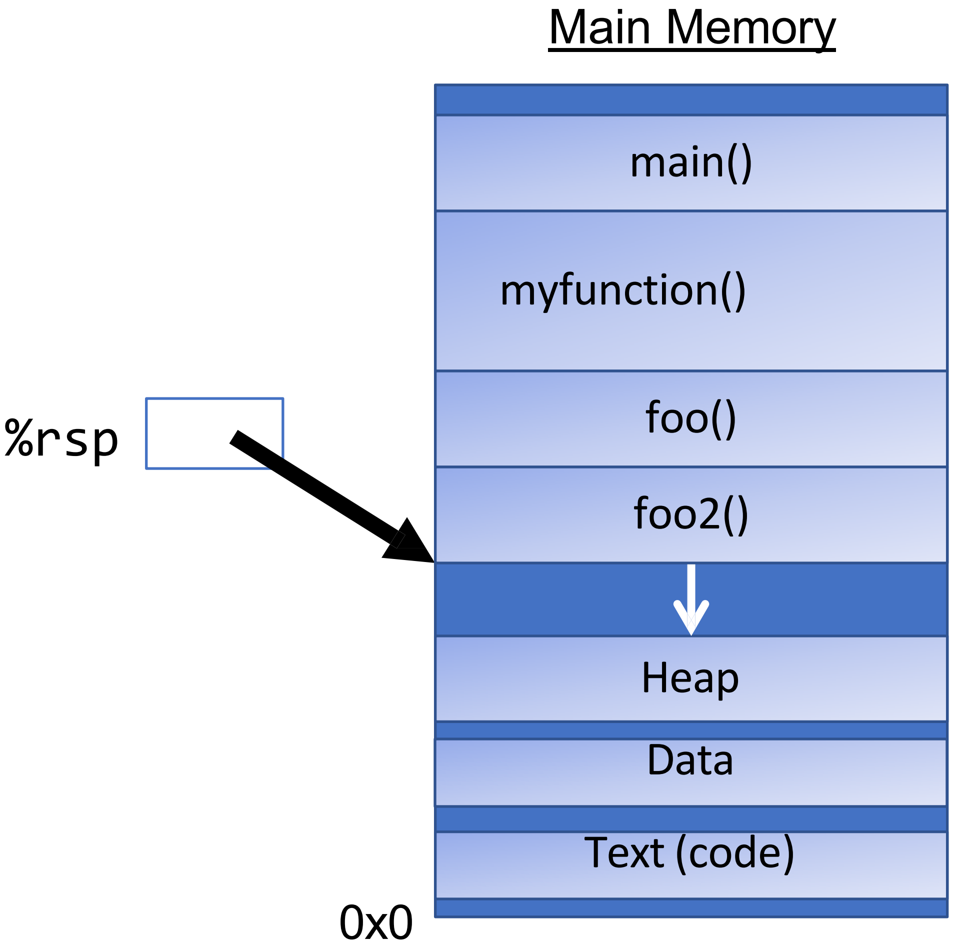
# %rsp

- **%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).



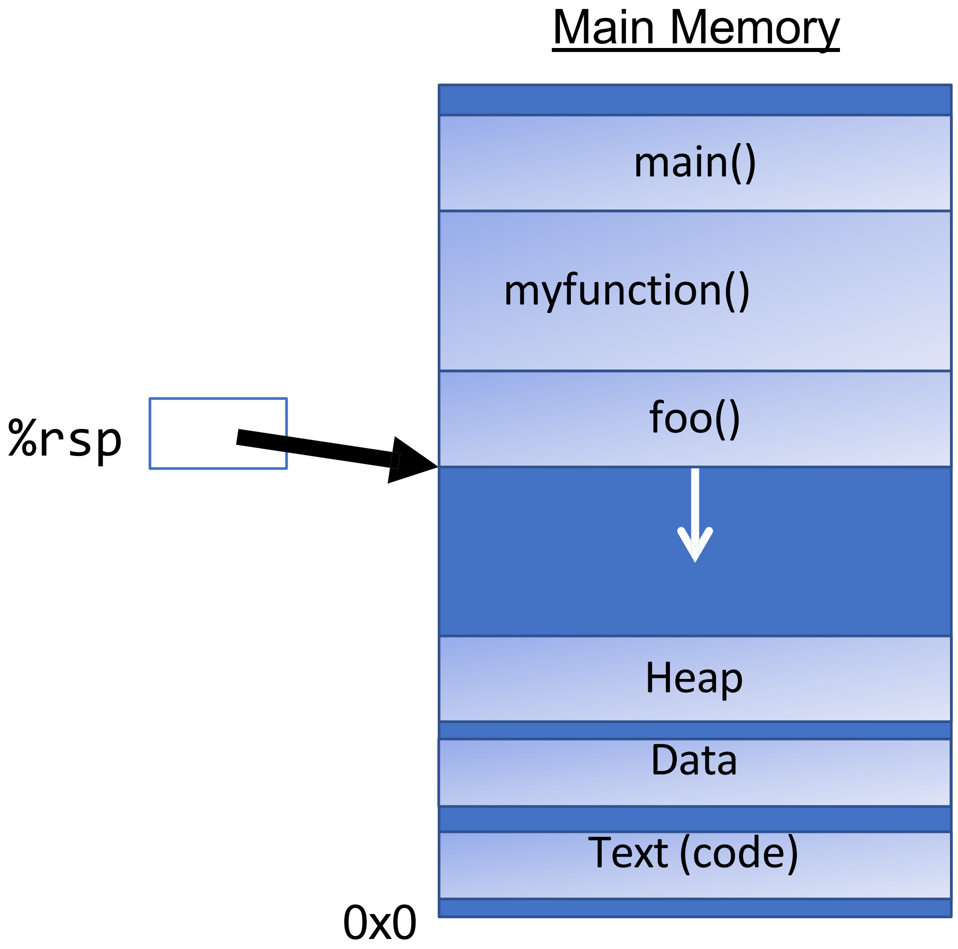
# %rsp

- **%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).



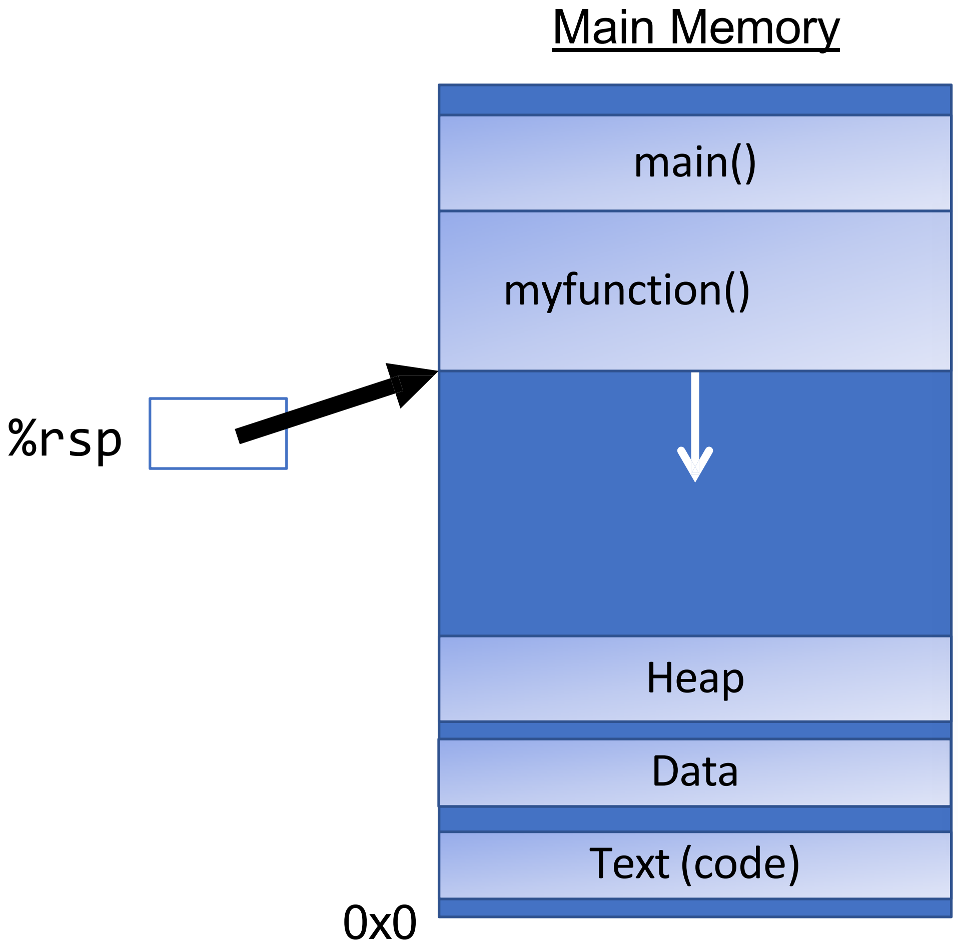
# %rsp

- **%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;`<br>`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]]`<br>`R[%rsp] ⟵ R[%rsp] + 8;` |

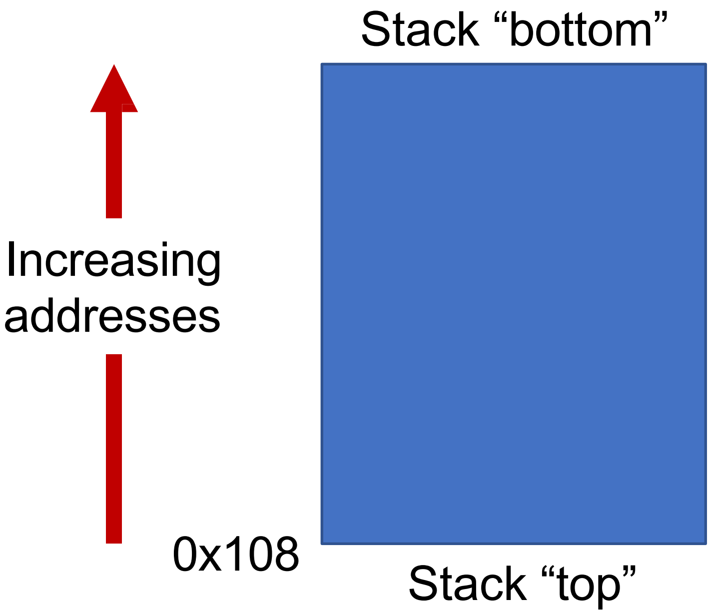
- This behavior is equivalent to the following, but popq is a shorter instruction:

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

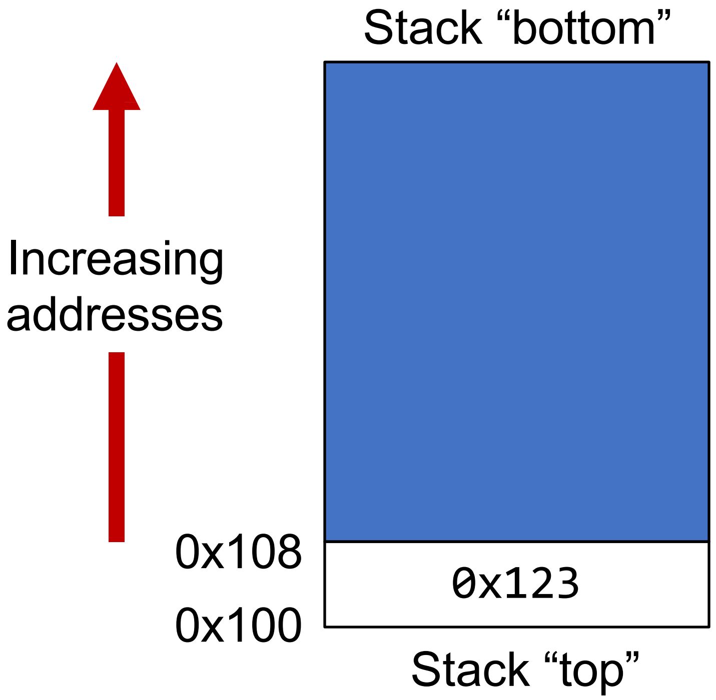
- Sometimes, you'll see instructions just explicitly increment the stack pointer to pop data.

# Stack Example

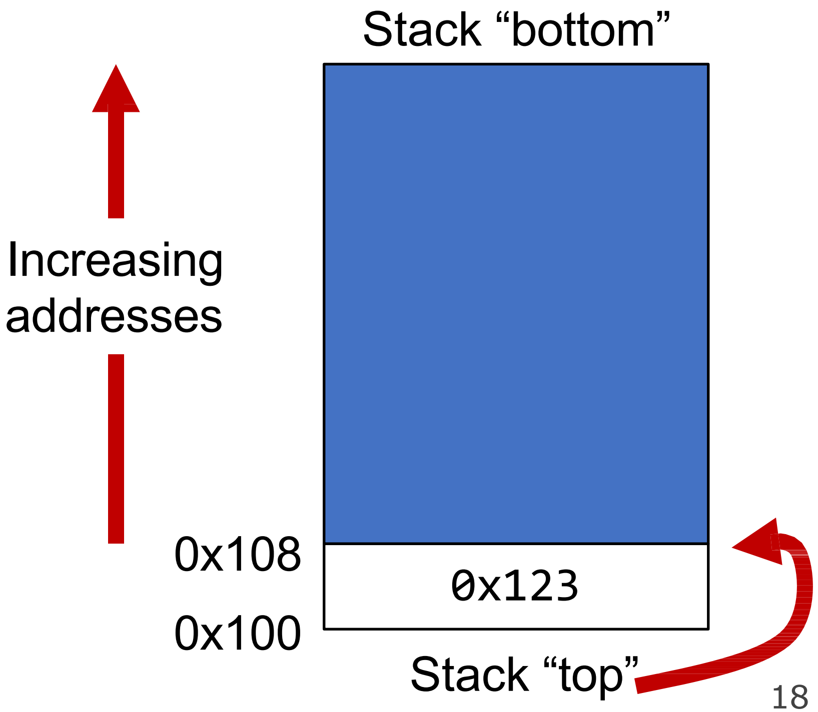
| Initially | |
|---|---|
| `%rax` | `0x123` |
| `%rdx` | `0` |
| `%rsp` | `0x108` |



| `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.

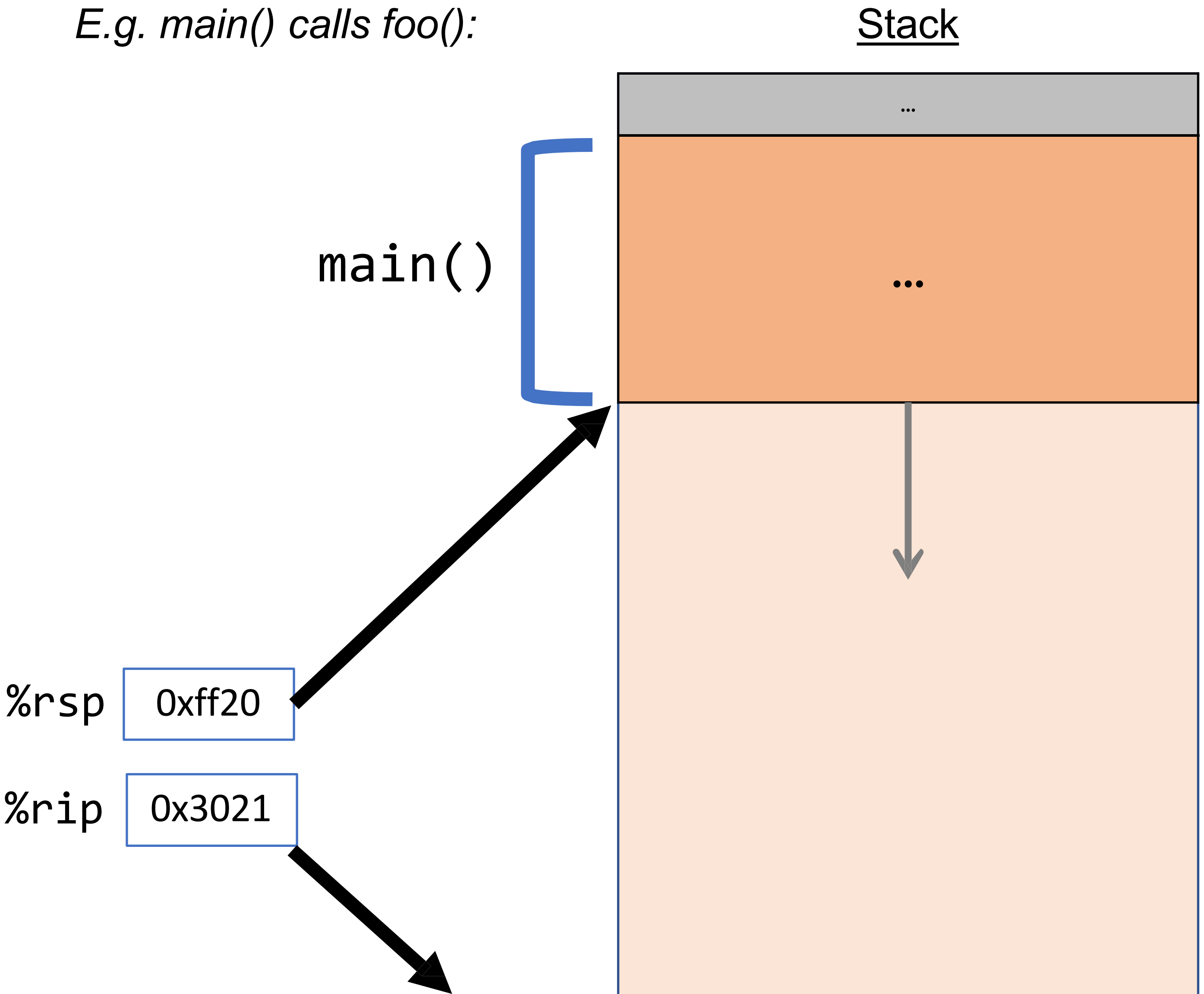
Terminology: **caller** function calls the **callee** function.

# 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.

*E.g. main() calls foo():*

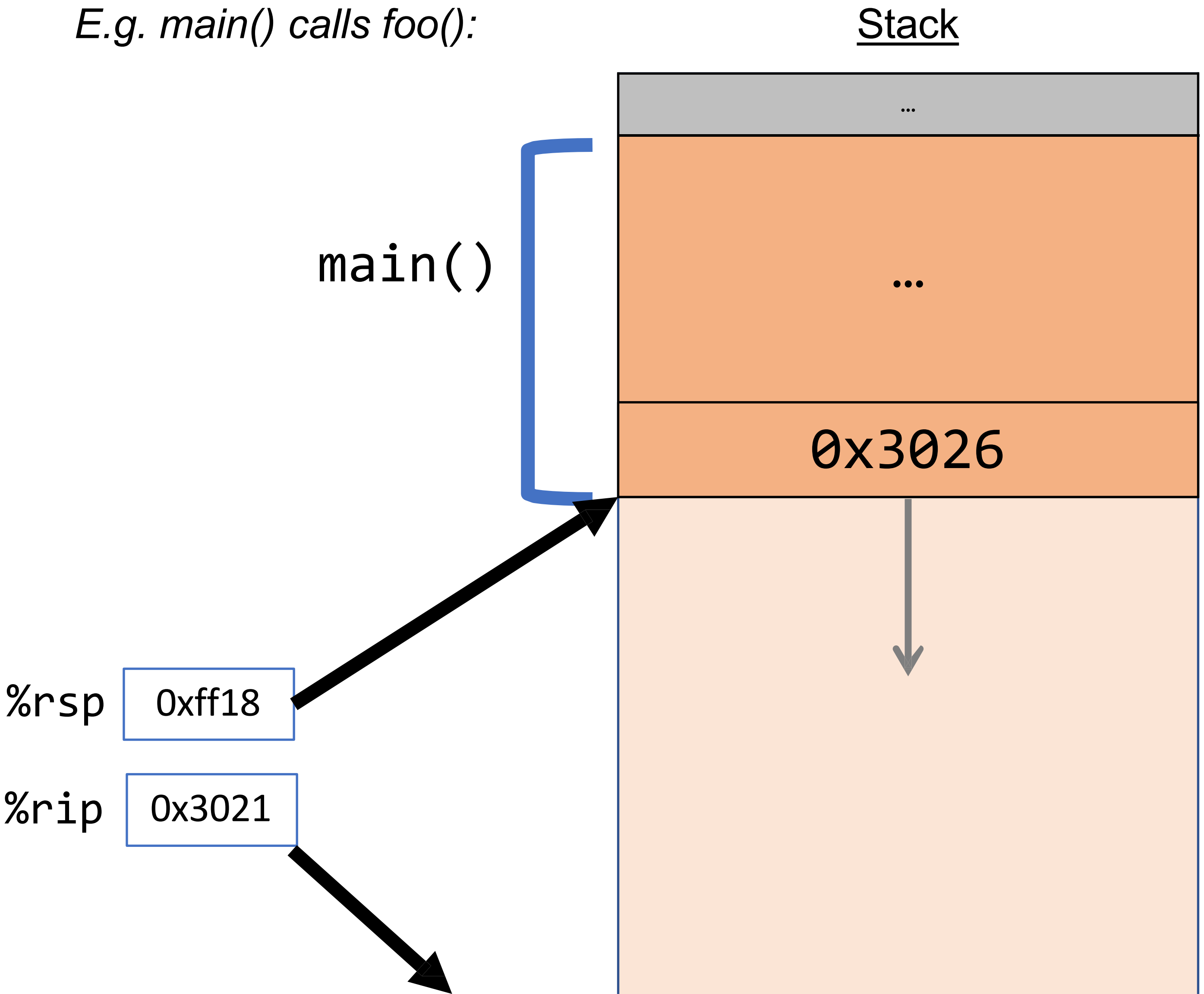


# Remembering Where We Left Off

**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.

*E.g. main() calls foo():*

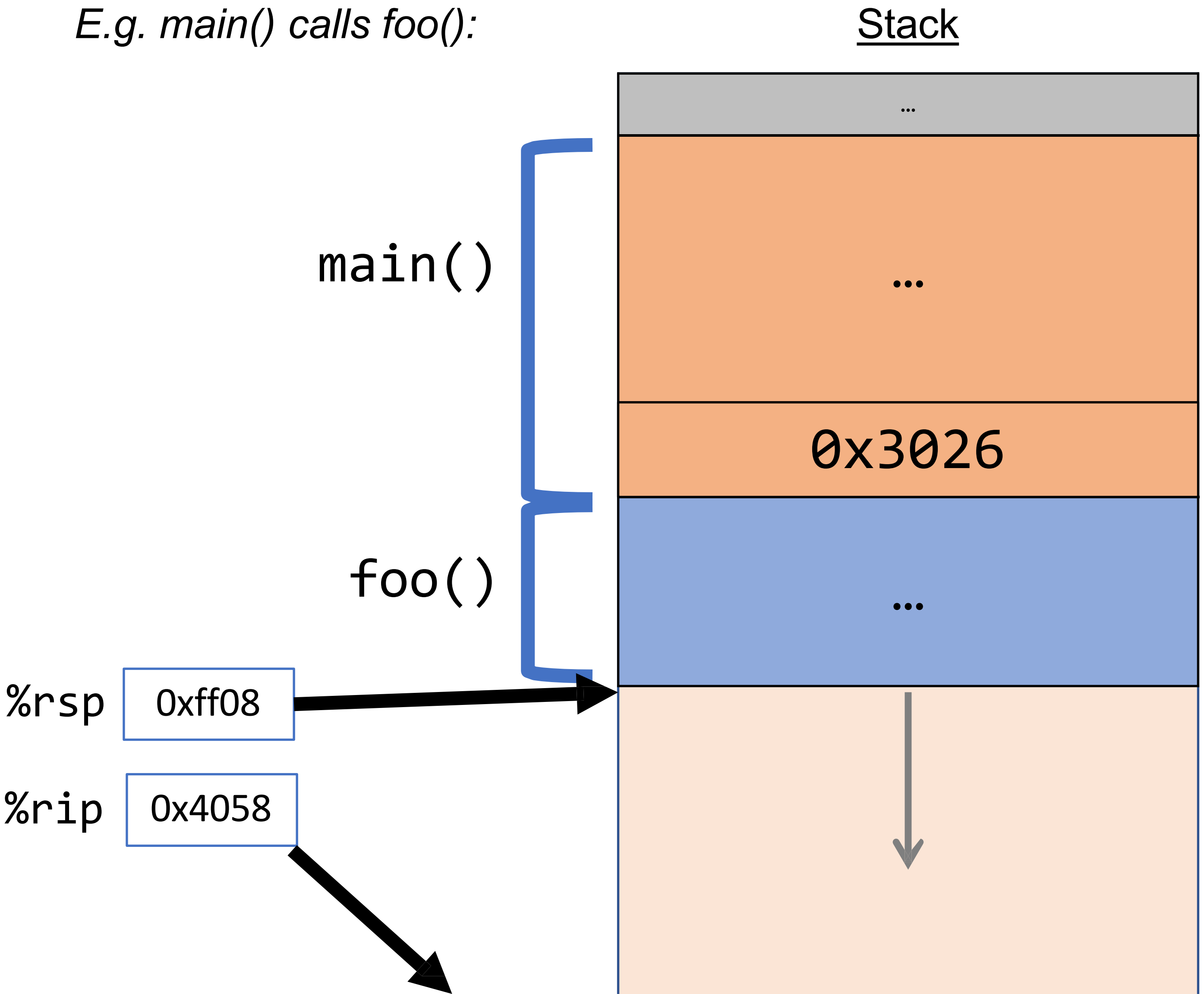


# Remembering Where We Left Off

**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.

*E.g. main() calls foo():*

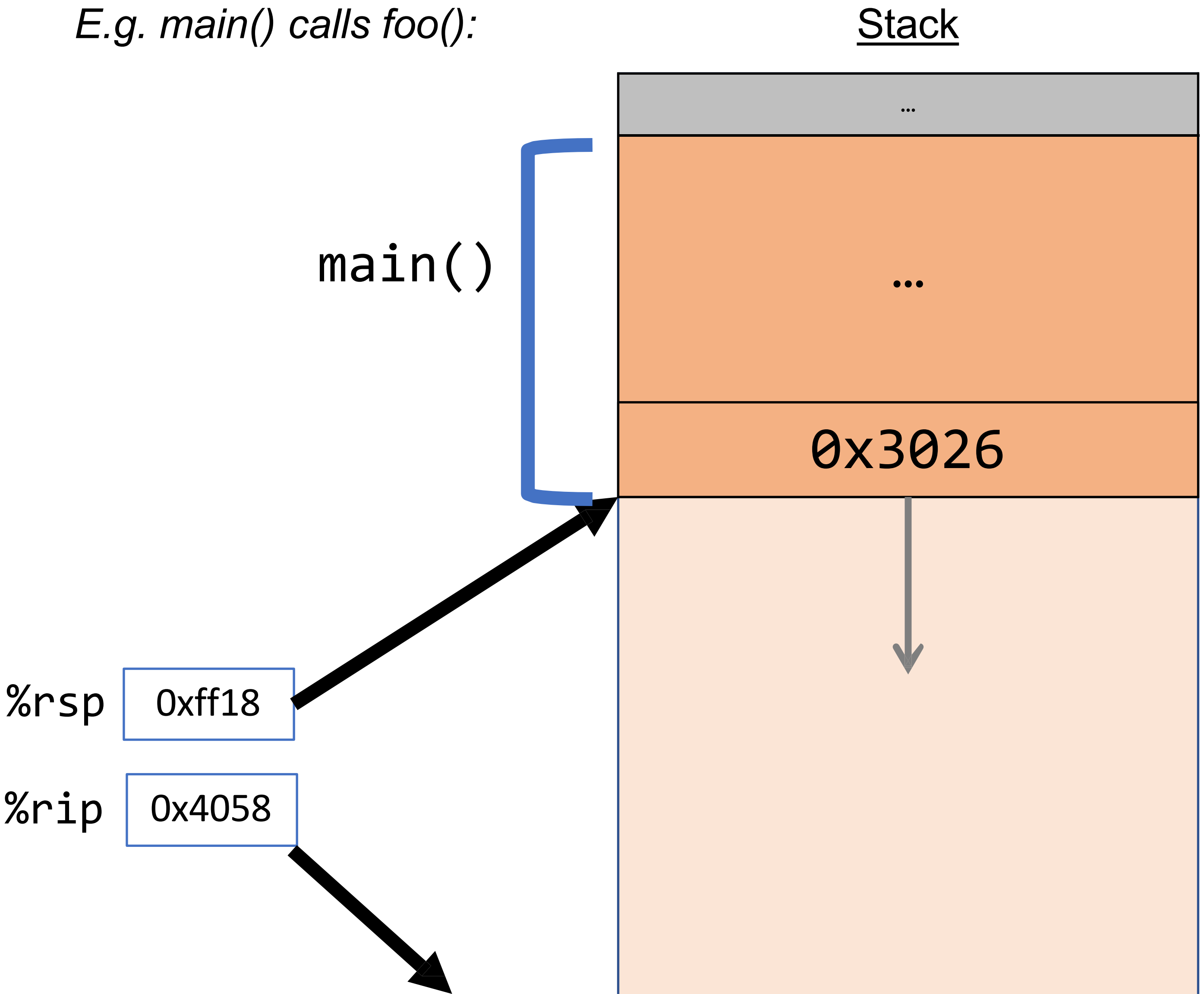


# Remembering Where We Left Off

**Problem:** %rip points to the next instruction to execute. To call a function, we must <u>remember</u> 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.

*E.g. main() calls foo():*

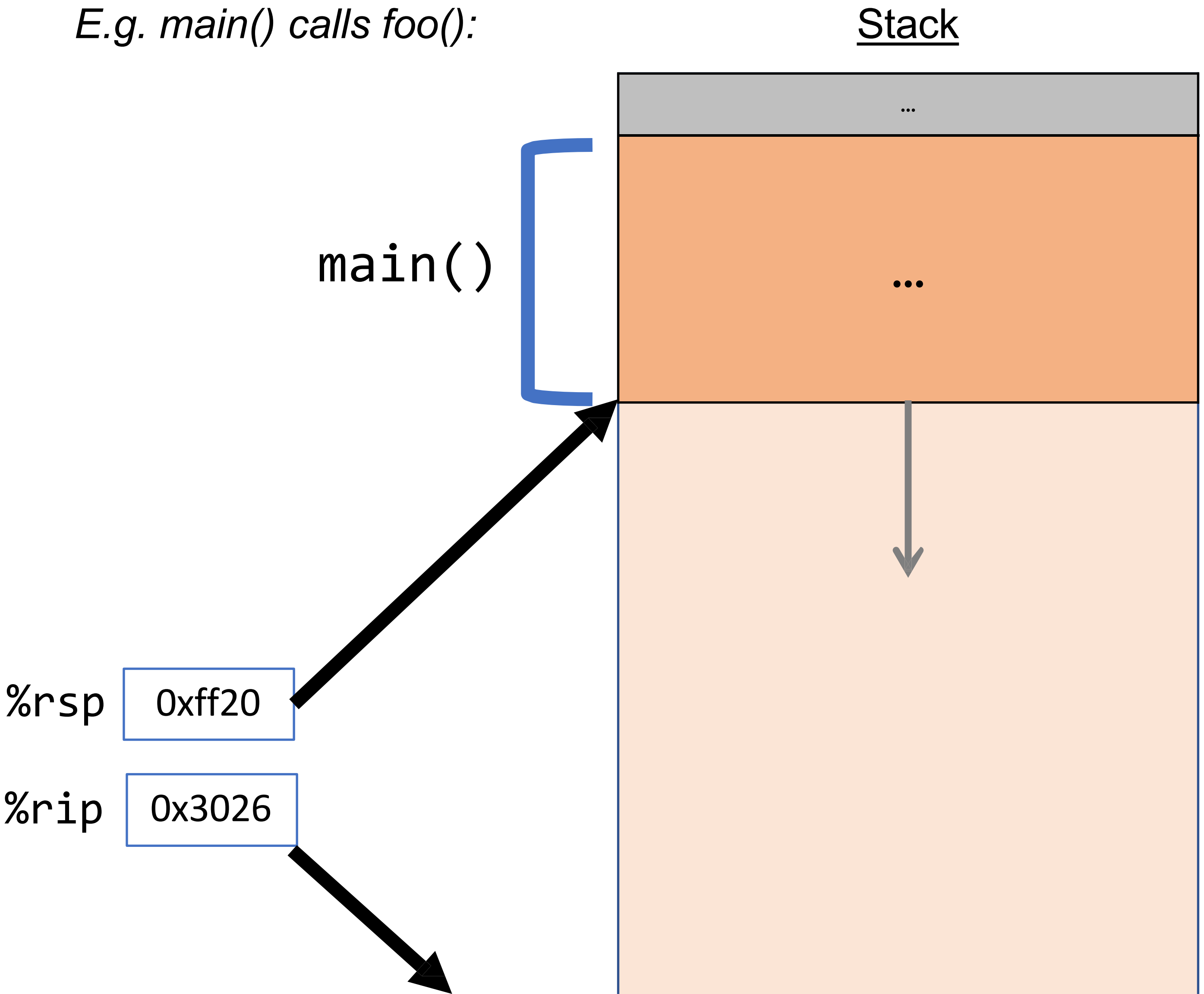


# Remembering Where We Left Off

**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.

*E.g. main() calls foo():*



# 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.

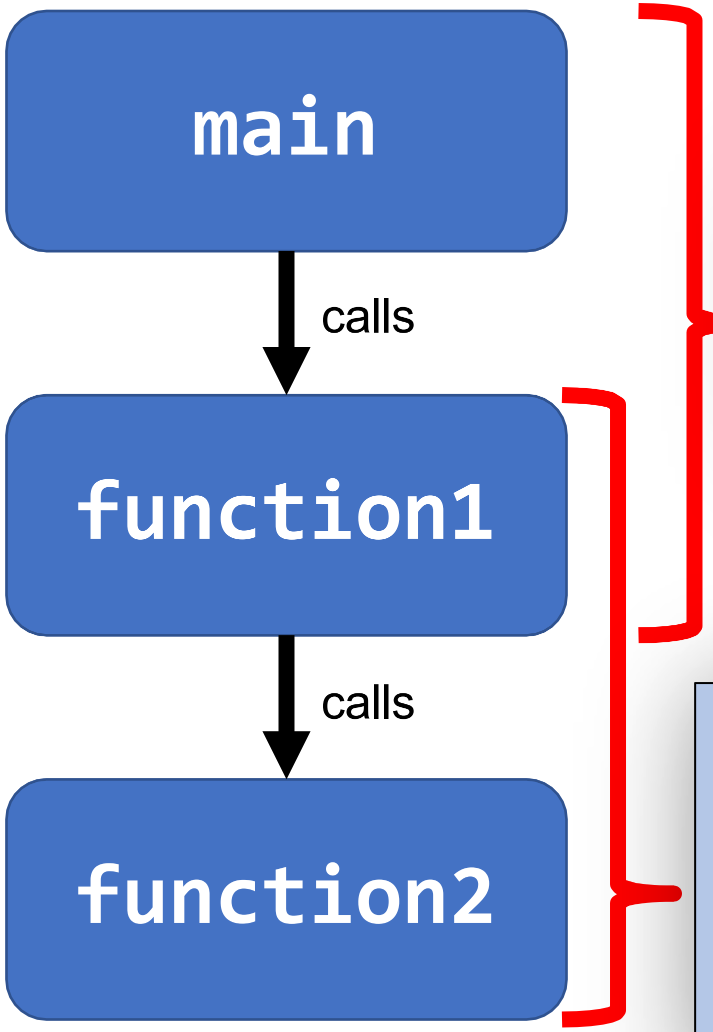
# Register Restrictions

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).



`main` is the caller, and `function1` is the callee.

`function1` is the caller, and `function2` is the callee.

# Register Restrictions

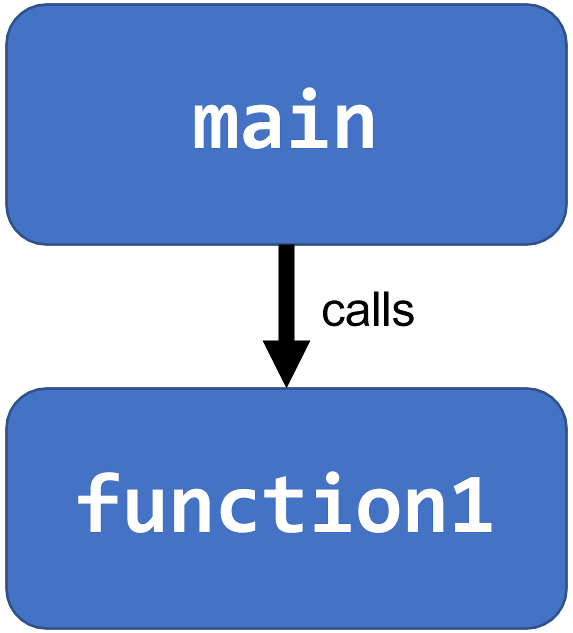
**Caller-Owned**

- 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.

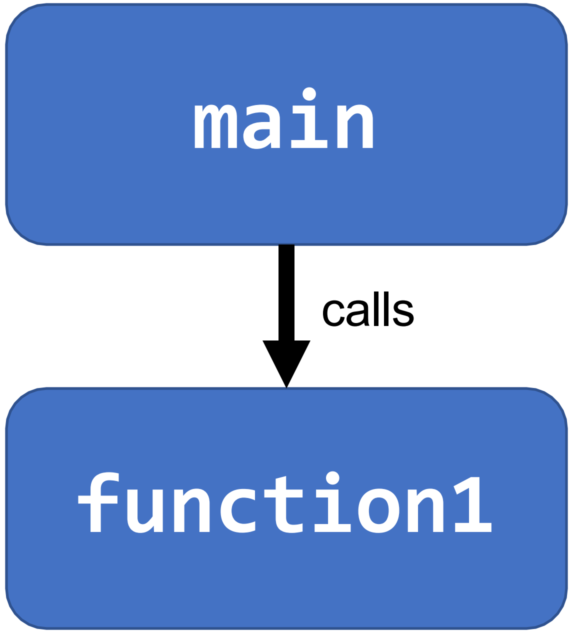
# Caller-Owned Registers



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

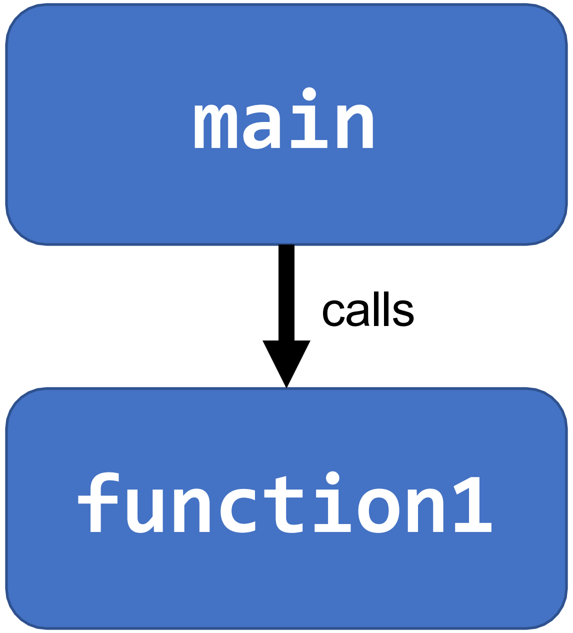
If function1 wants to use any caller-owned registers, it must save the existing values and restore them before returning.

# Caller-Owned Registers



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

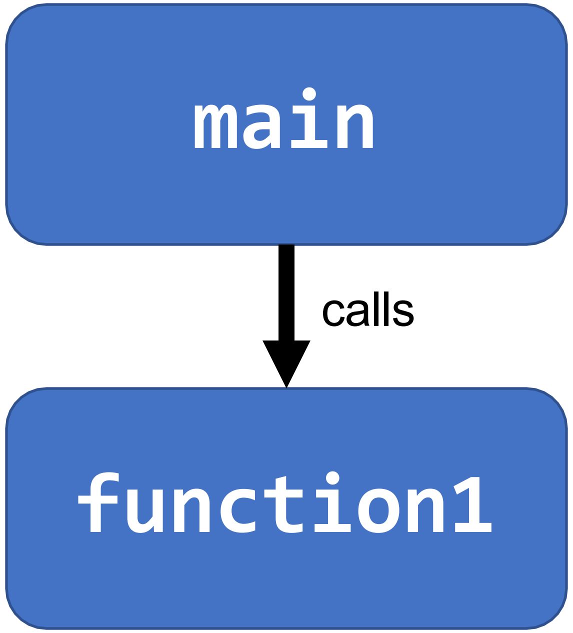
# Callee-Owned Registers



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

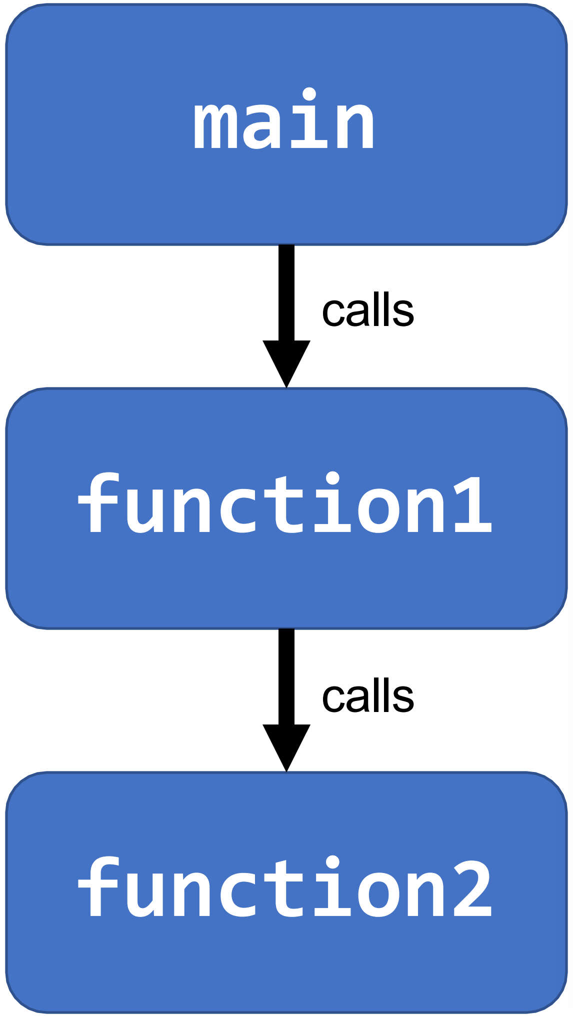
If function1 wants to use any callee-owned registers, it can do so without saving the existing values.

# 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.

# 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.

# Calling Functions In Assembly

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

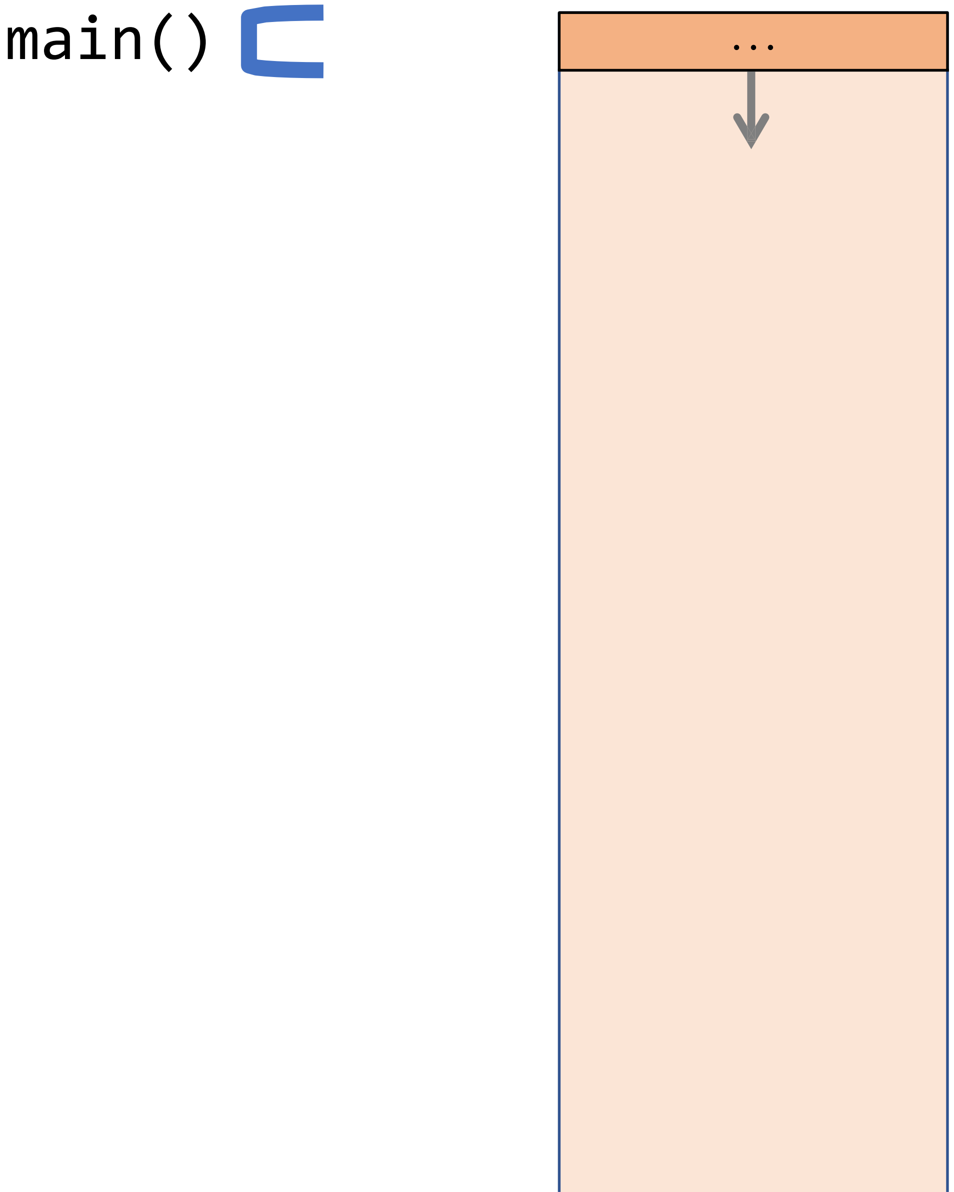
- **Pass Control** – %rip must be adjusted to execute the function being called and then resume the caller function afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

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

# 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) {
    …
}
```

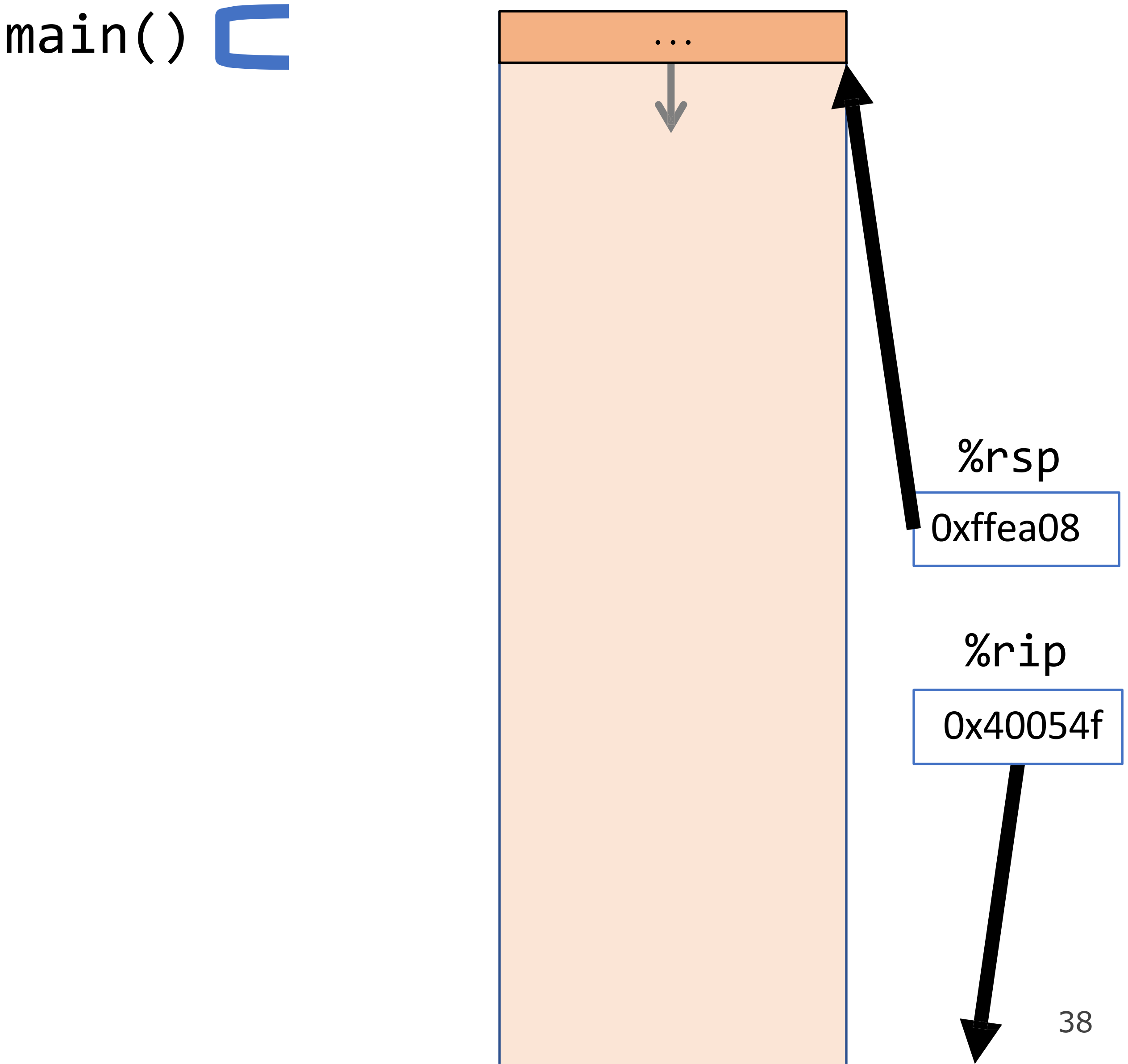


# Parameters and Return

```
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)
```

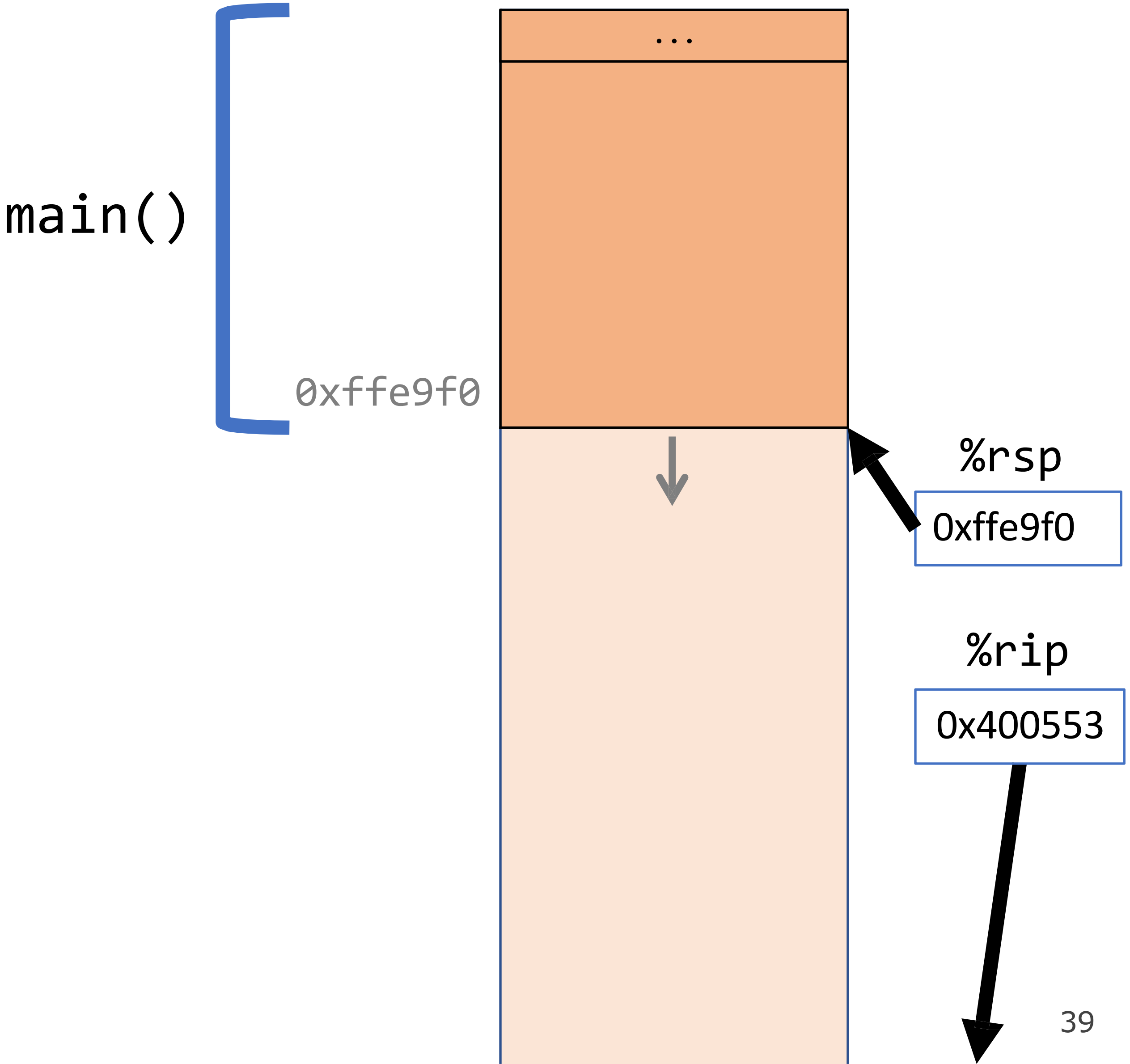


# Parameters and Return

```
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)
```

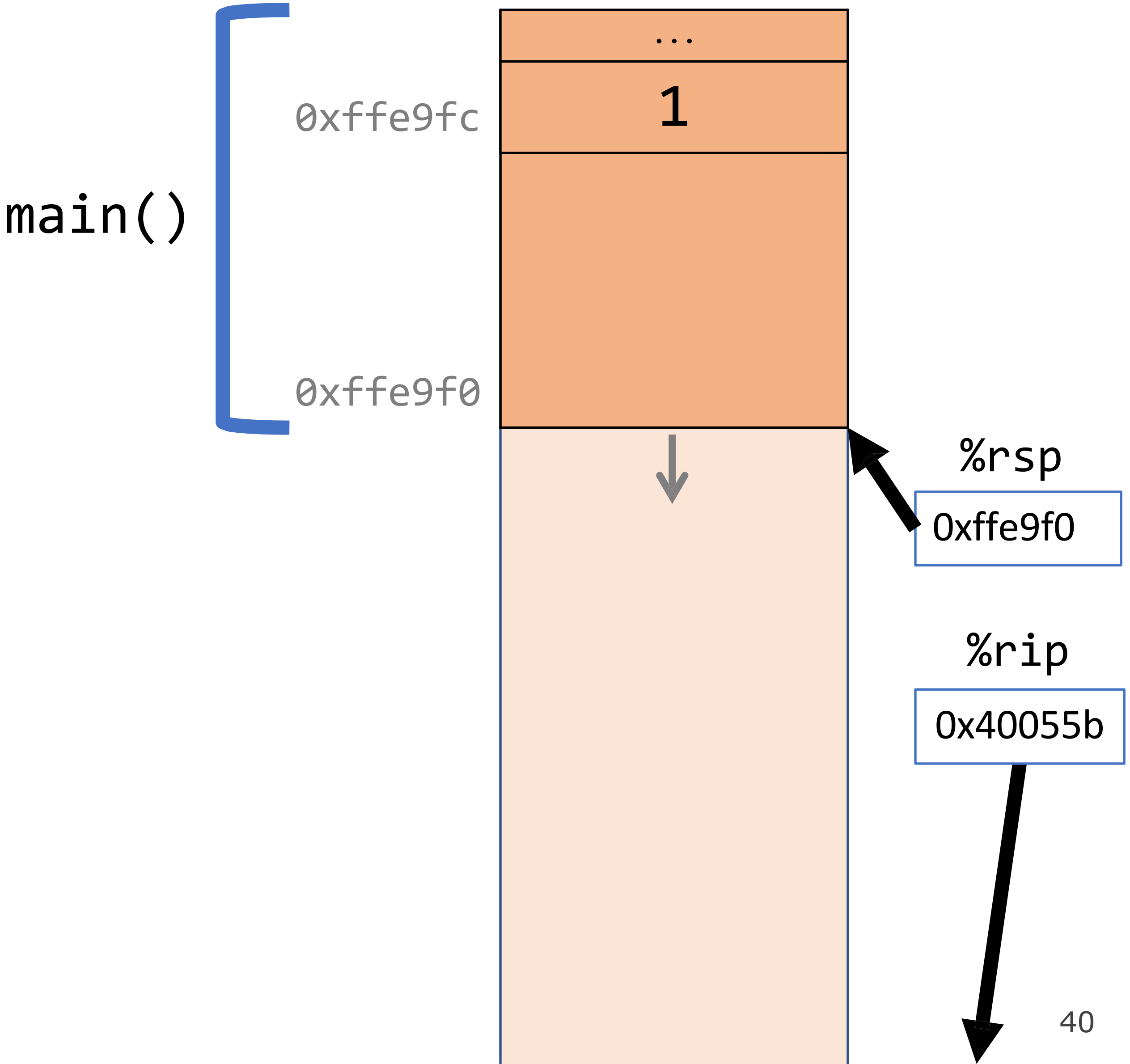


# Parameters and Return

```
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)
```

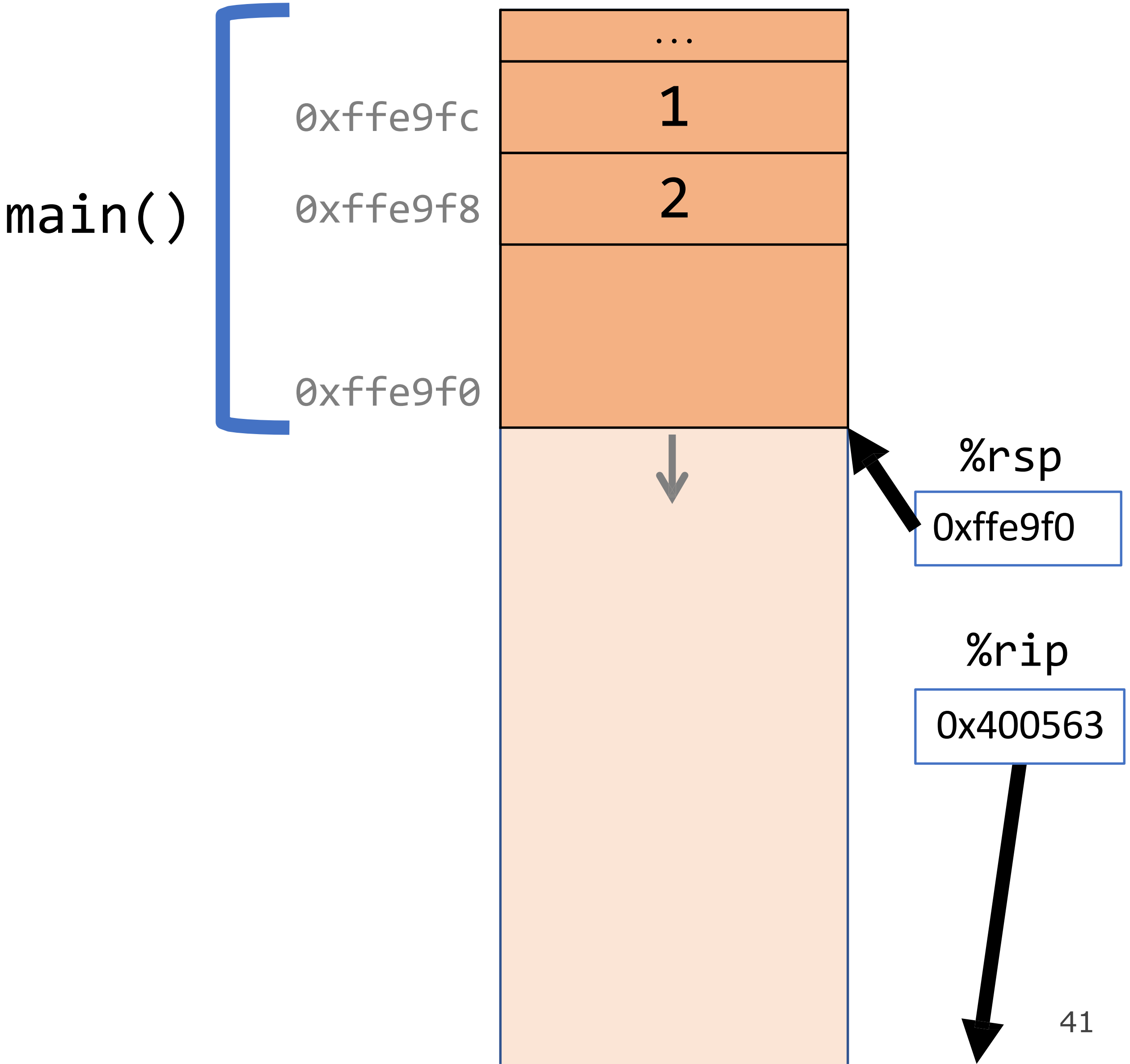


# Parameters and Return

```
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) {
    …
}
```

```
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)
```

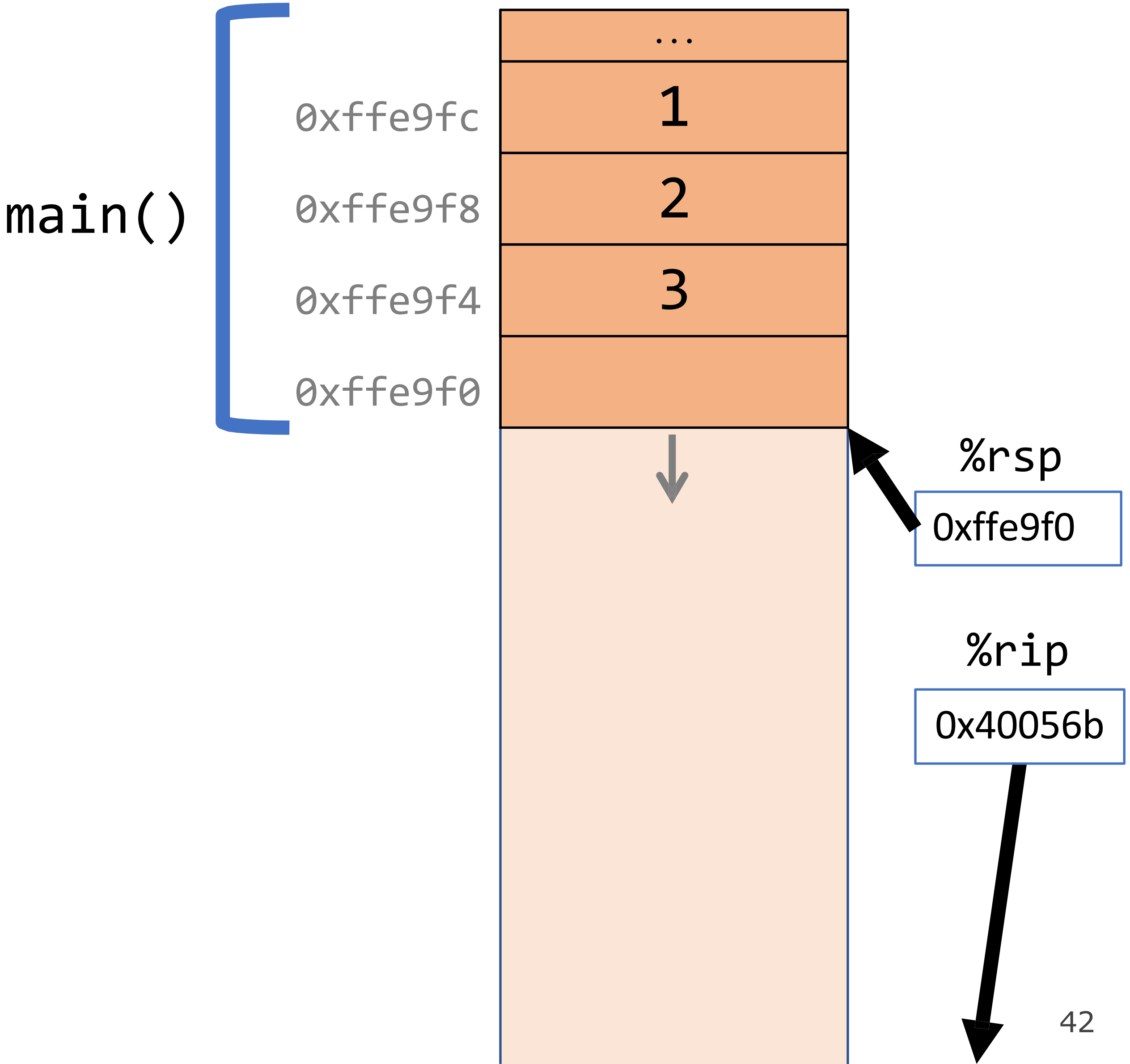


# Parameters and Return

```
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
```

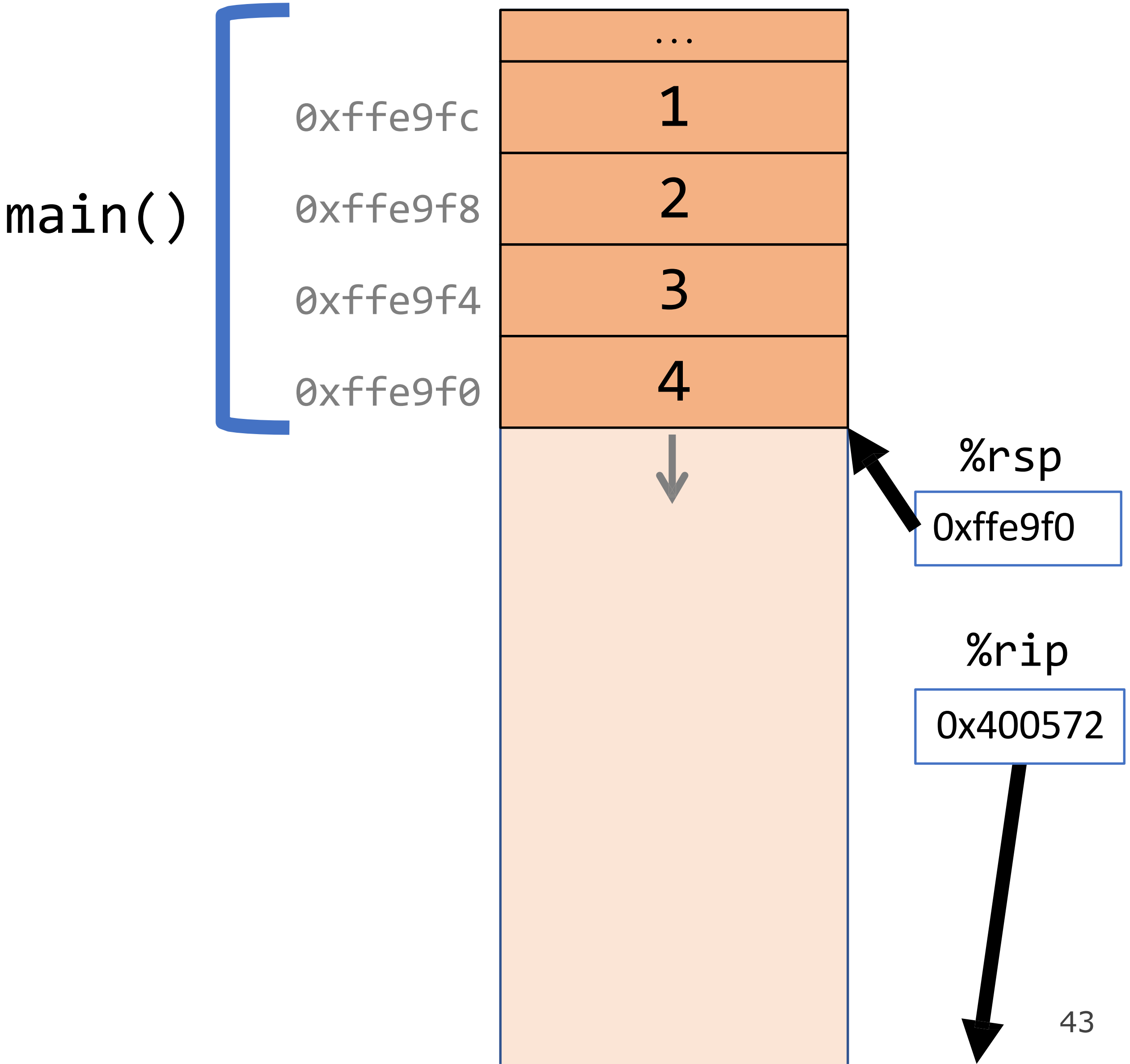


# Parameters and Return

```
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
```

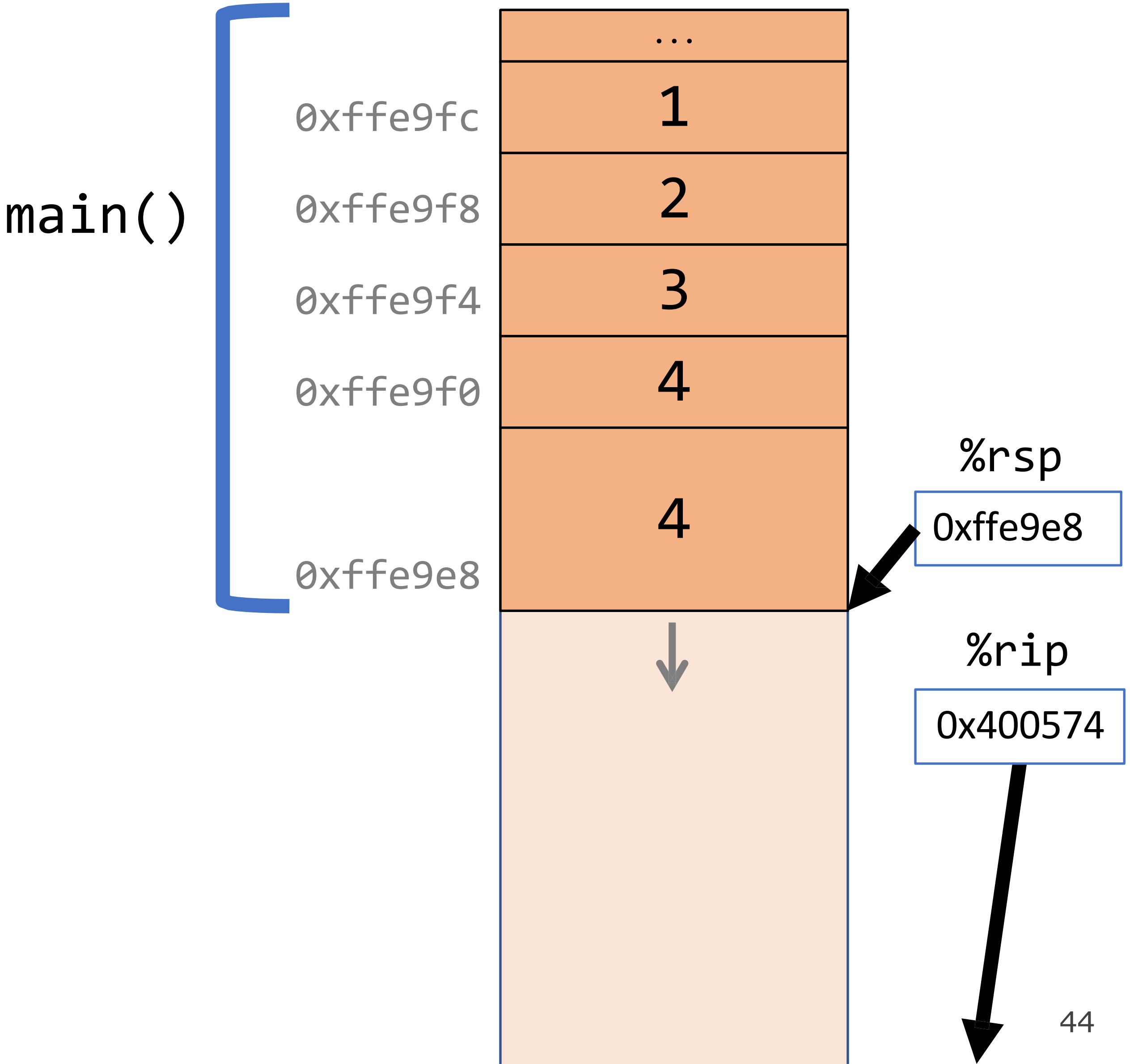


# Parameters and Return

```
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
```

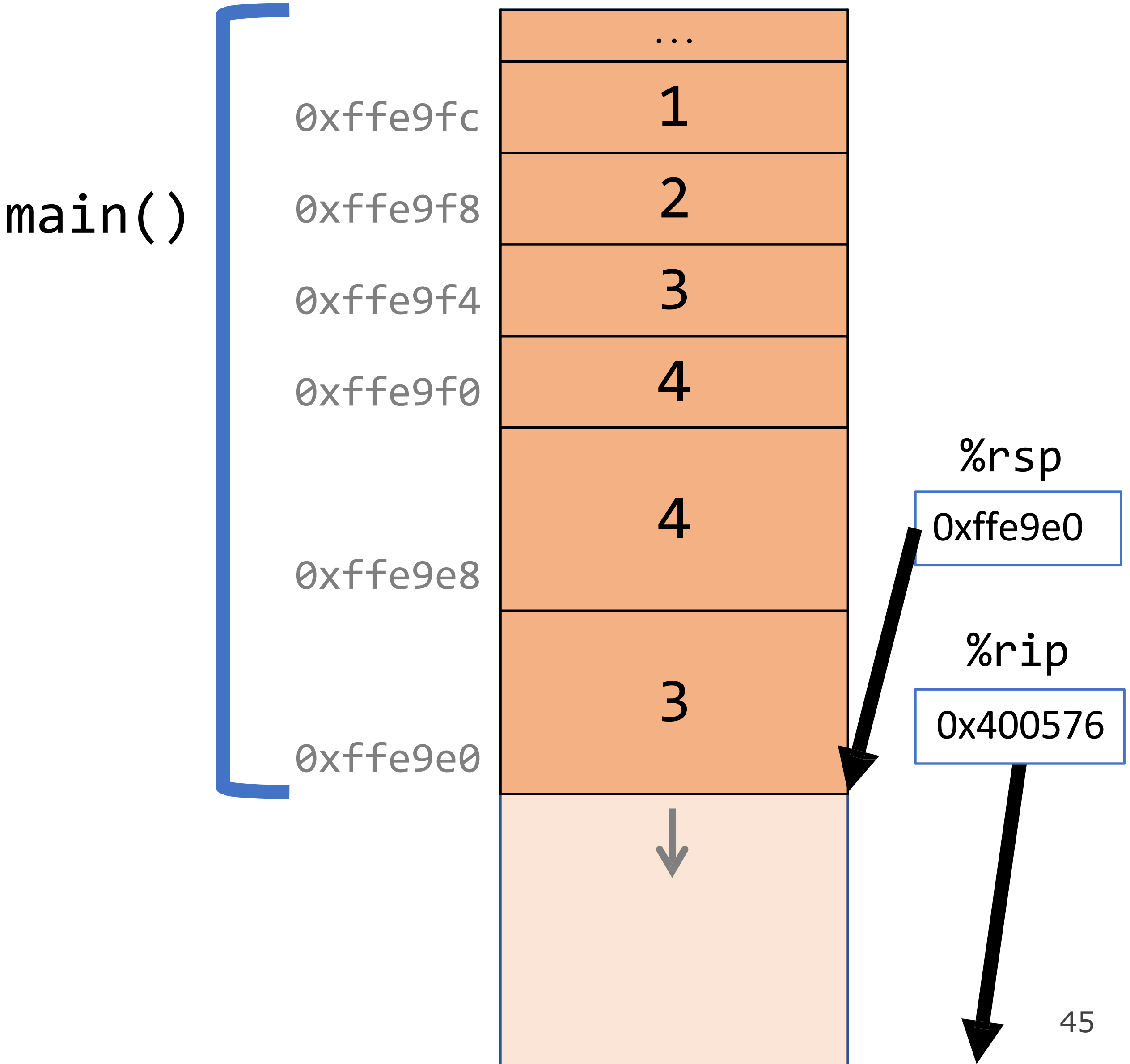


# Parameters and Return

```
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
```

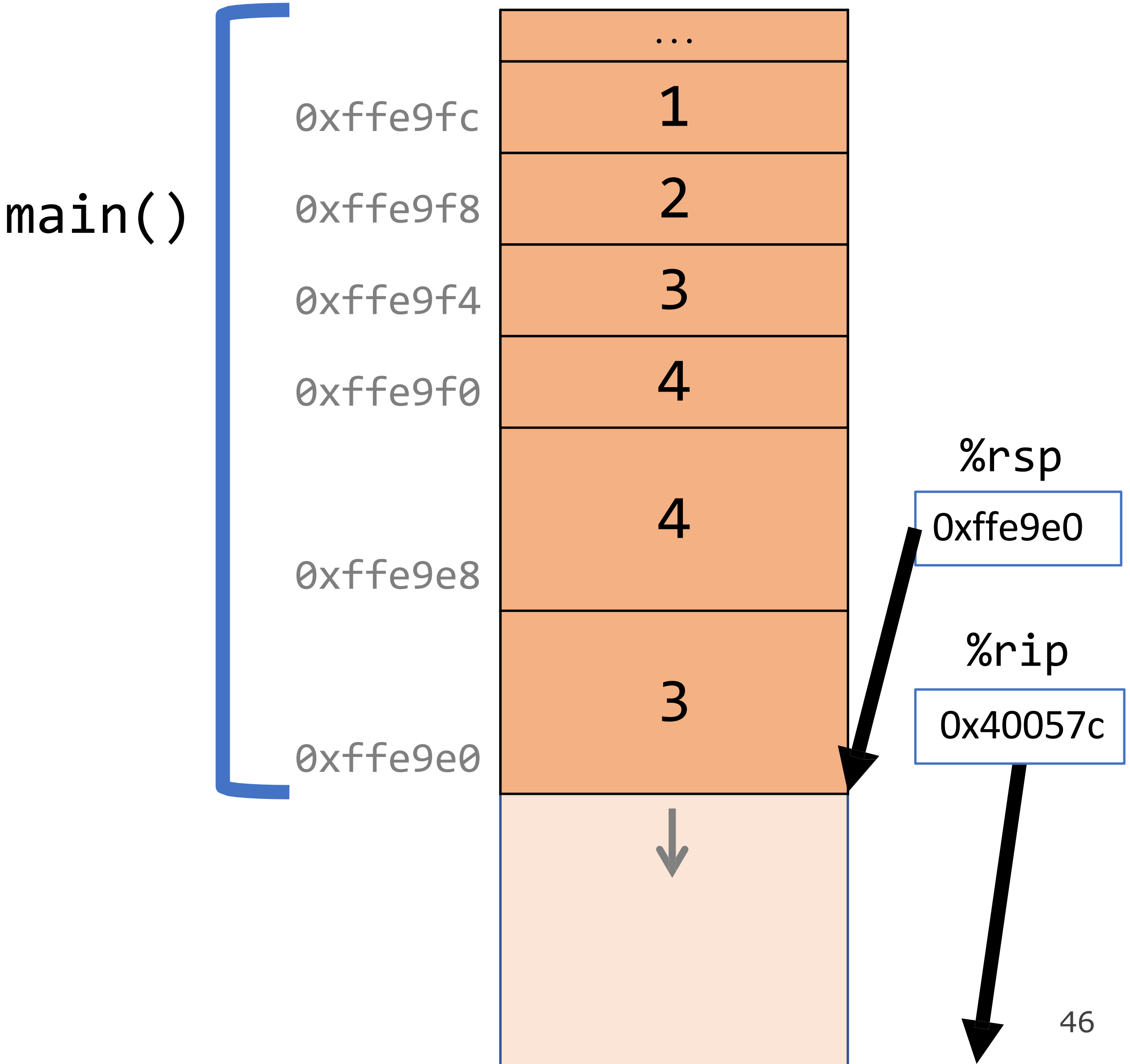


# Parameters and Return

```
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
```

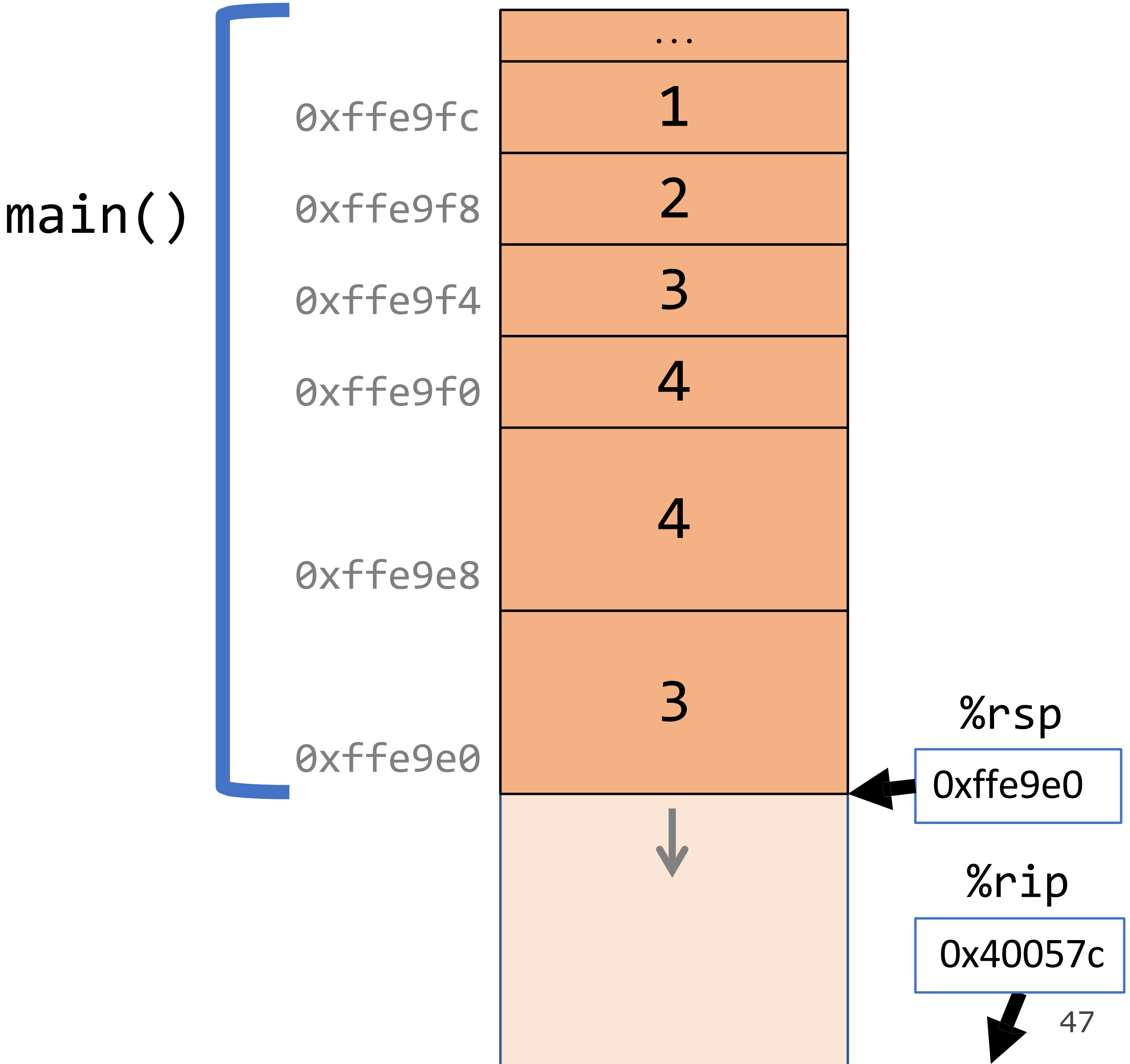


# Parameters and Return

```
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
```

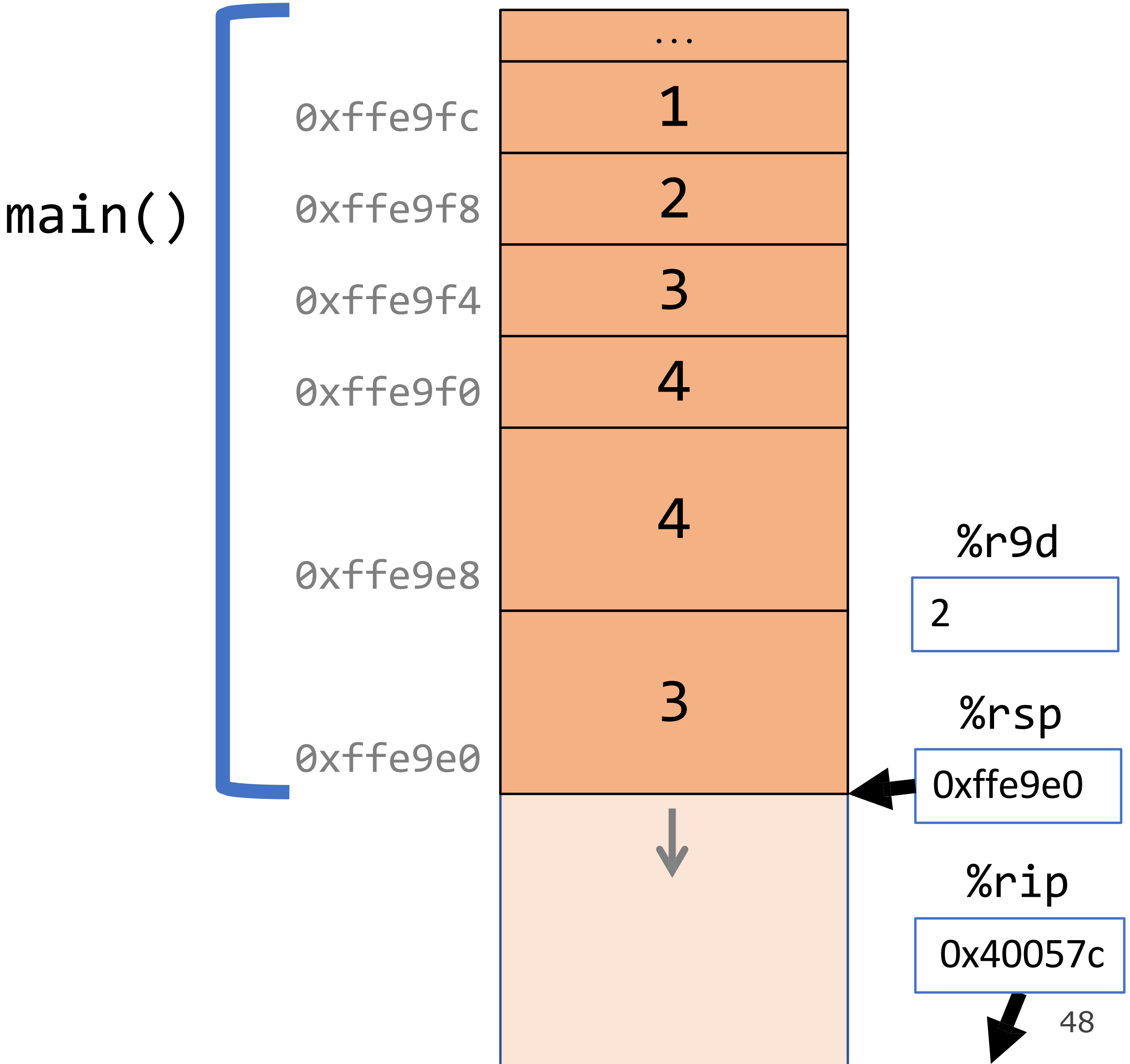


# Parameters and Return

```
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()

| | |
|---|---|
| | ... |
| 0xffe9fc | 1 |
| 0xffe9f8 | 2 |
| 0xffe9f4 | 3 |
| 0xffe9f0 | 4 |
| 0xffe9e8 | 4 |
| 0xffe9e0 | 3 |

%r9d: 2

%rsp: 0xffe9e0

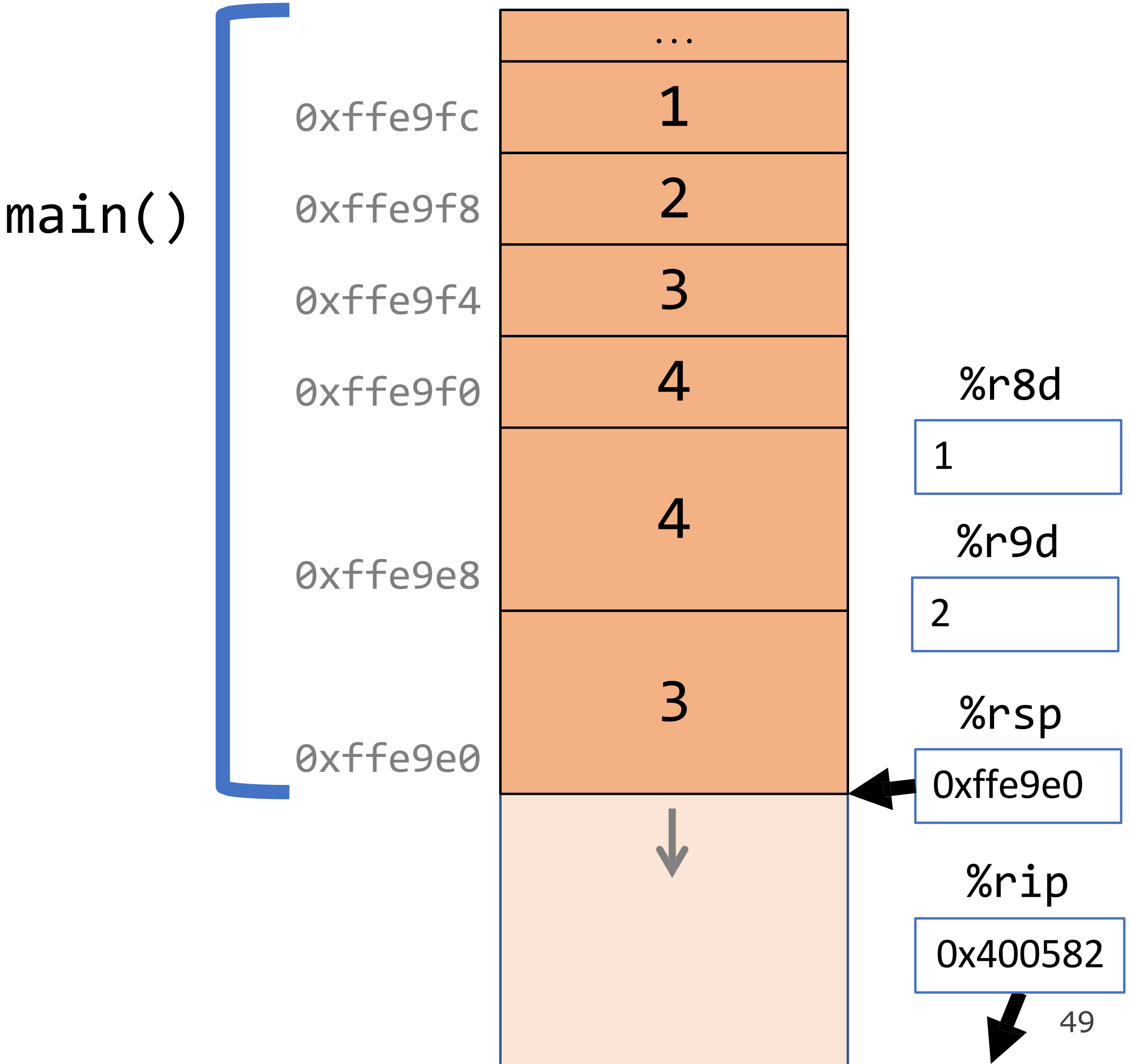
%rip: 0x40057c

# 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()

| | |
|---|---|
| | … |
| 0xffe9fc | 1 |
| 0xffe9f8 | 2 |
| 0xffe9f4 | 3 |
| 0xffe9f0 | 4 |
| 0xffe9e8 | 4 |
| 0xffe9e0 | 3 |

%r8d
1

%r9d
2

%rsp
0xffe9e0

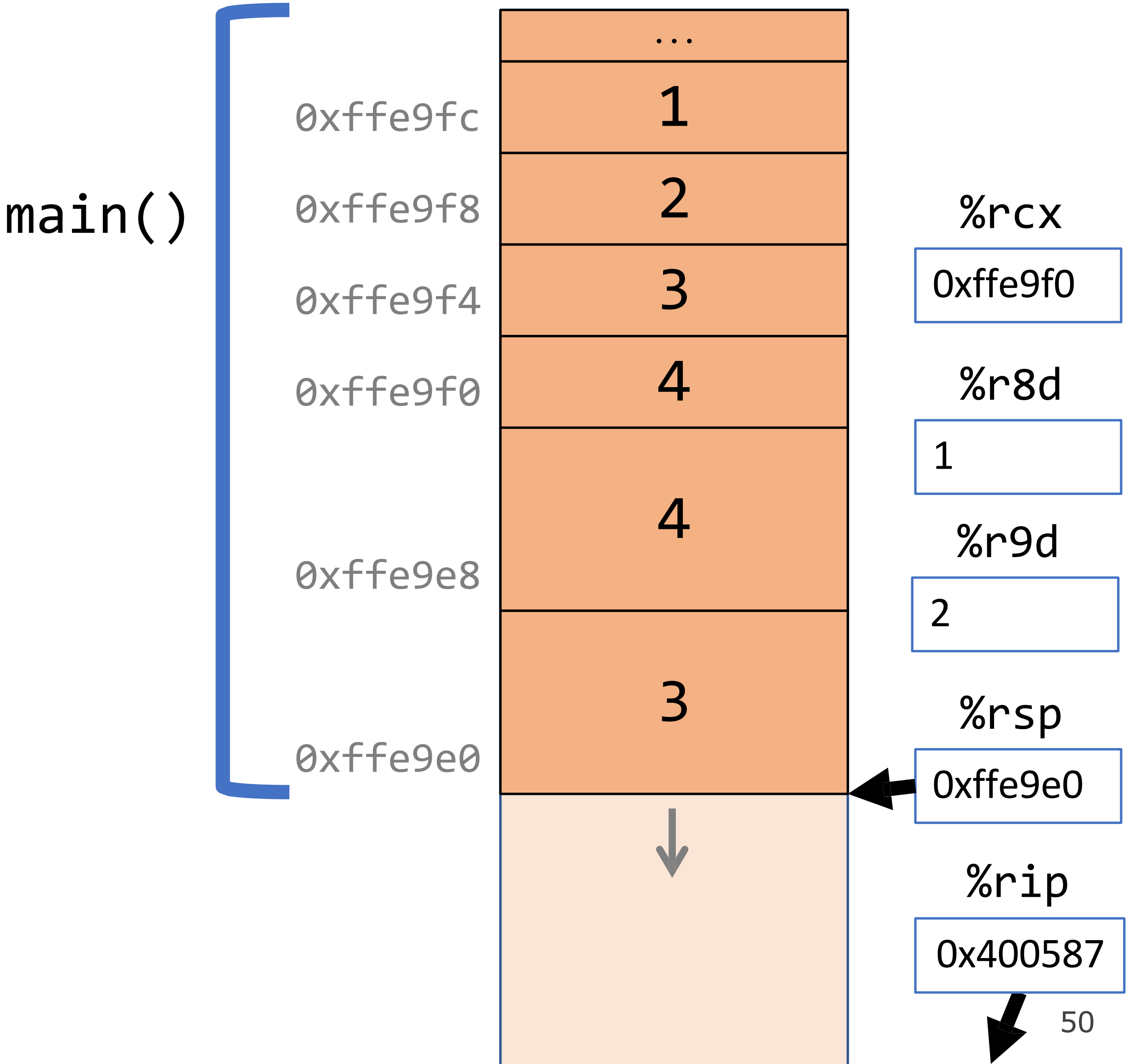
%rip
0x400582

# 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
```

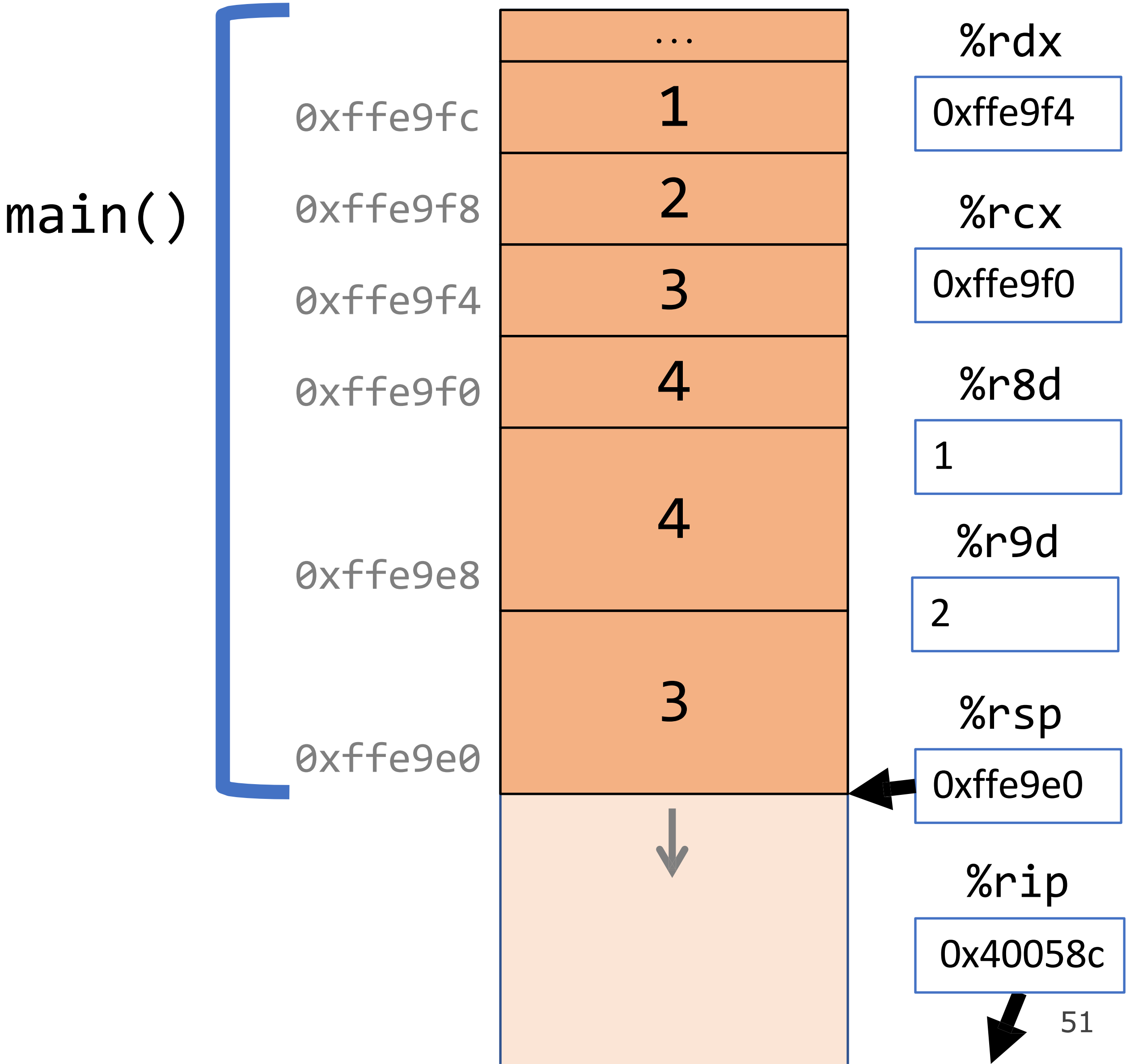


# Parameters and Return

```
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()

| | |
|---|---|
| | … |
| 0xffe9fc | 1 |
| 0xffe9f8 | 2 |
| 0xffe9f4 | 3 |
| 0xffe9f0 | 4 |
| 0xffe9e8 | 4 |
| 0xffe9e0 | 3 |

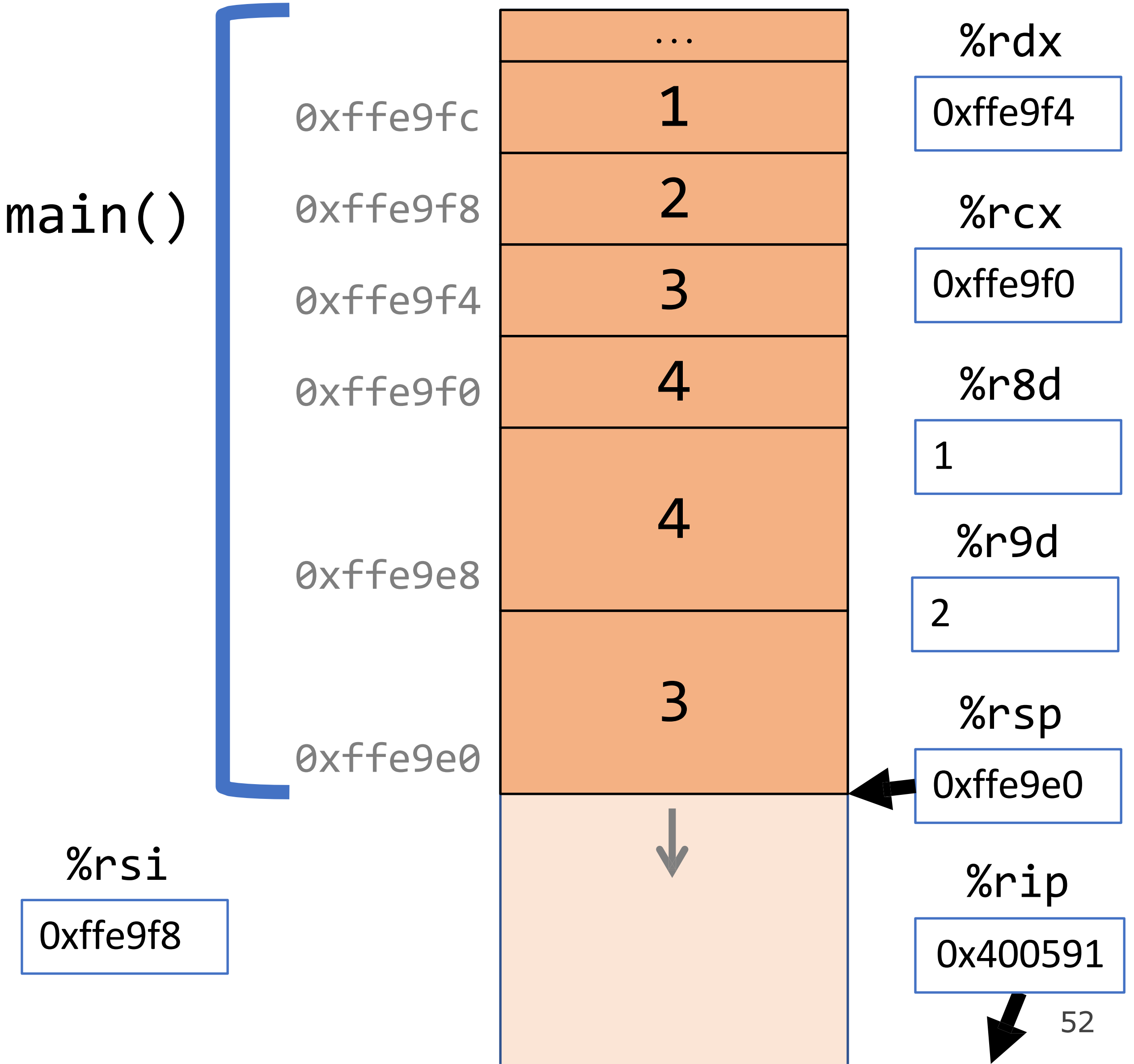
| | |
|---|---|
| %rdx | 0xffe9f4 |
| %rcx | 0xffe9f0 |
| %r8d | 1 |
| %r9d | 2 |
| %rsp | 0xffe9e0 |
| %rip | 0x40058c |

# 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()

| | |
|---|---|
| | … |
| 0xffe9fc | 1 |
| 0xffe9f8 | 2 |
| 0xffe9f4 | 3 |
| 0xffe9f0 | 4 |
| 0xffe9e8 | 4 |
| 0xffe9e0 | 3 |

%rsi: 0xffe9f8

%rdx: 0xffe9f4

%rcx: 0xffe9f0

%r8d: 1

%r9d: 2

%rsp: 0xffe9e0

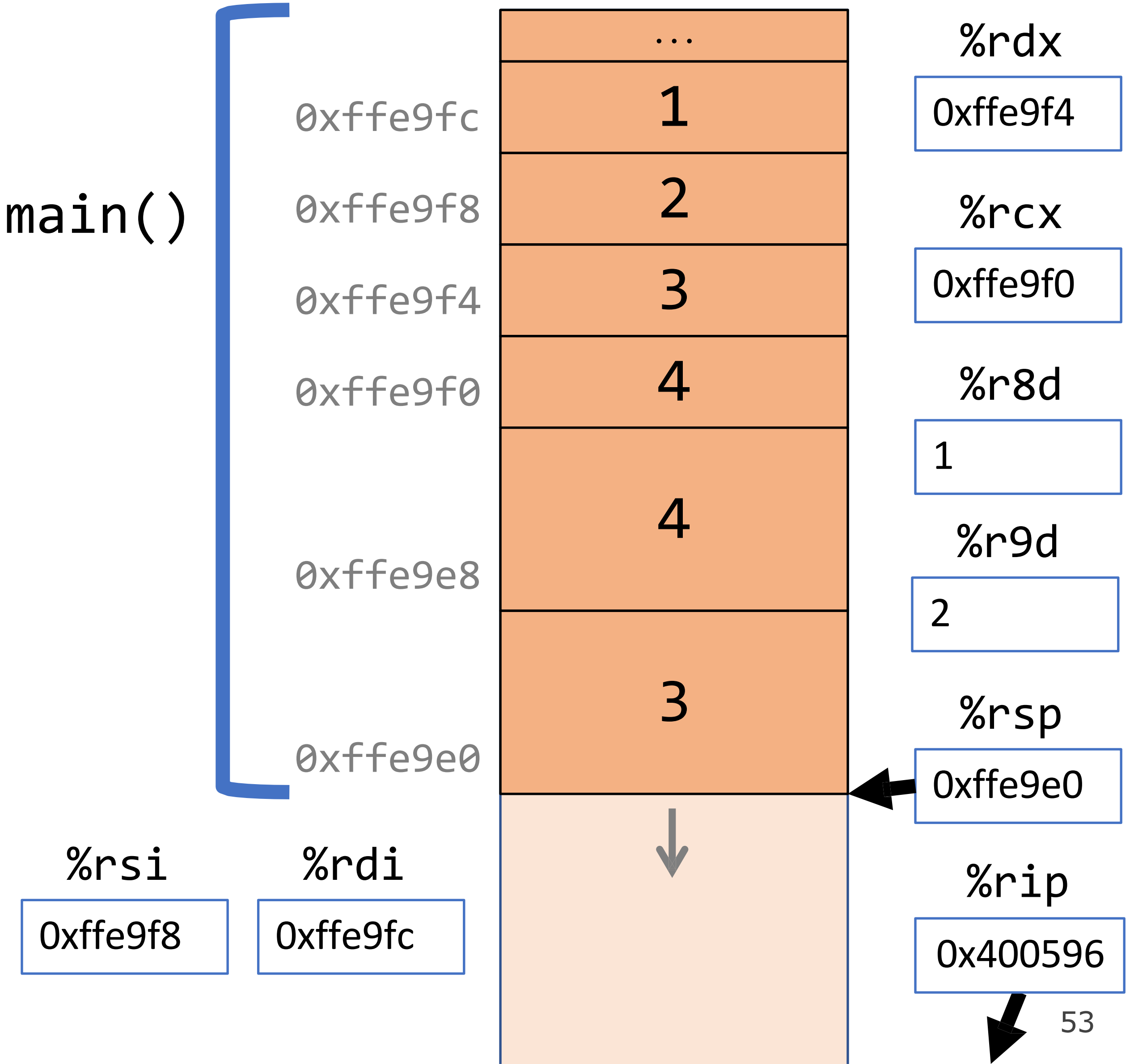
%rip: 0x400591

# 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
```

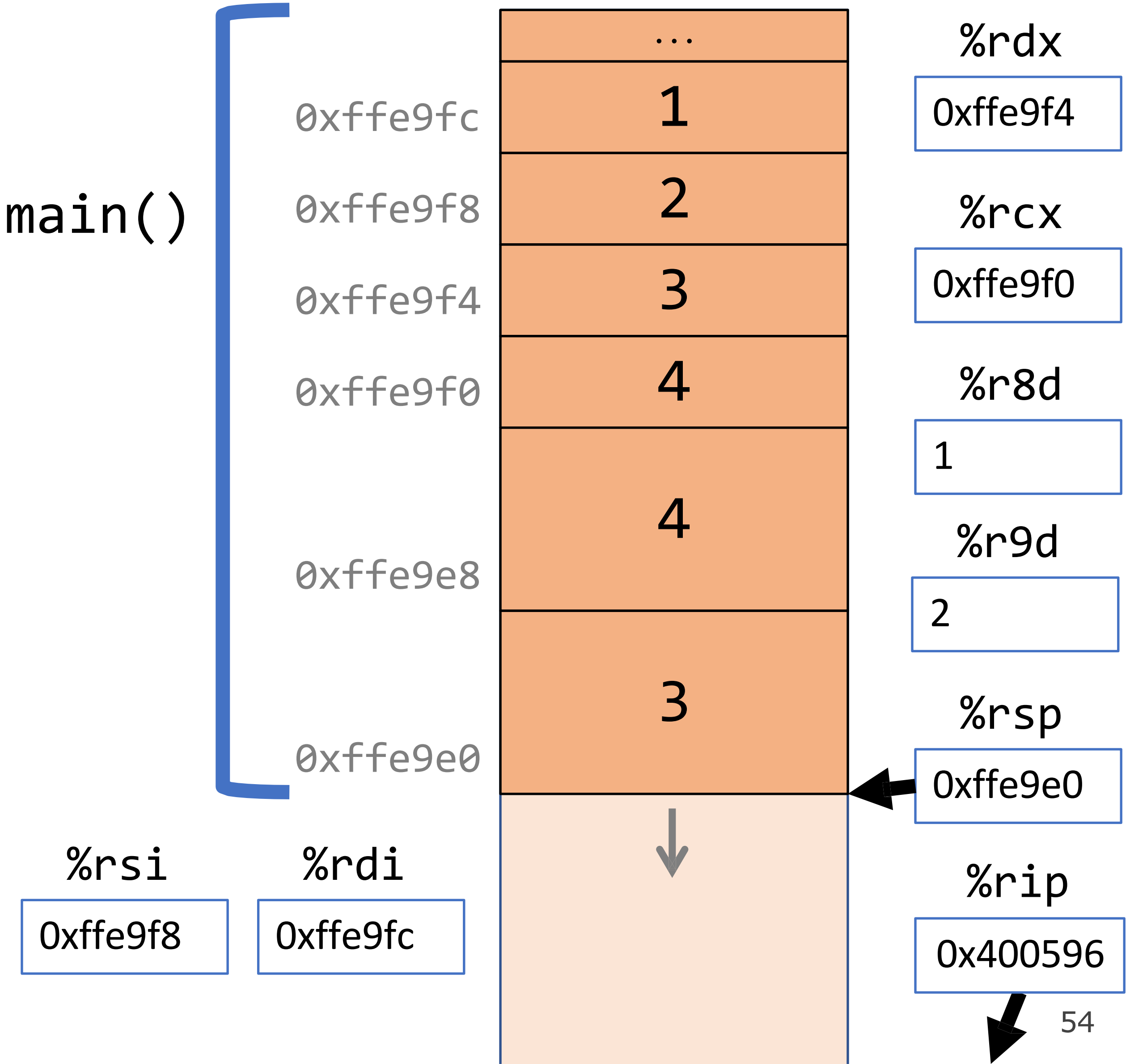


# Parameters and Return

```
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
…
```

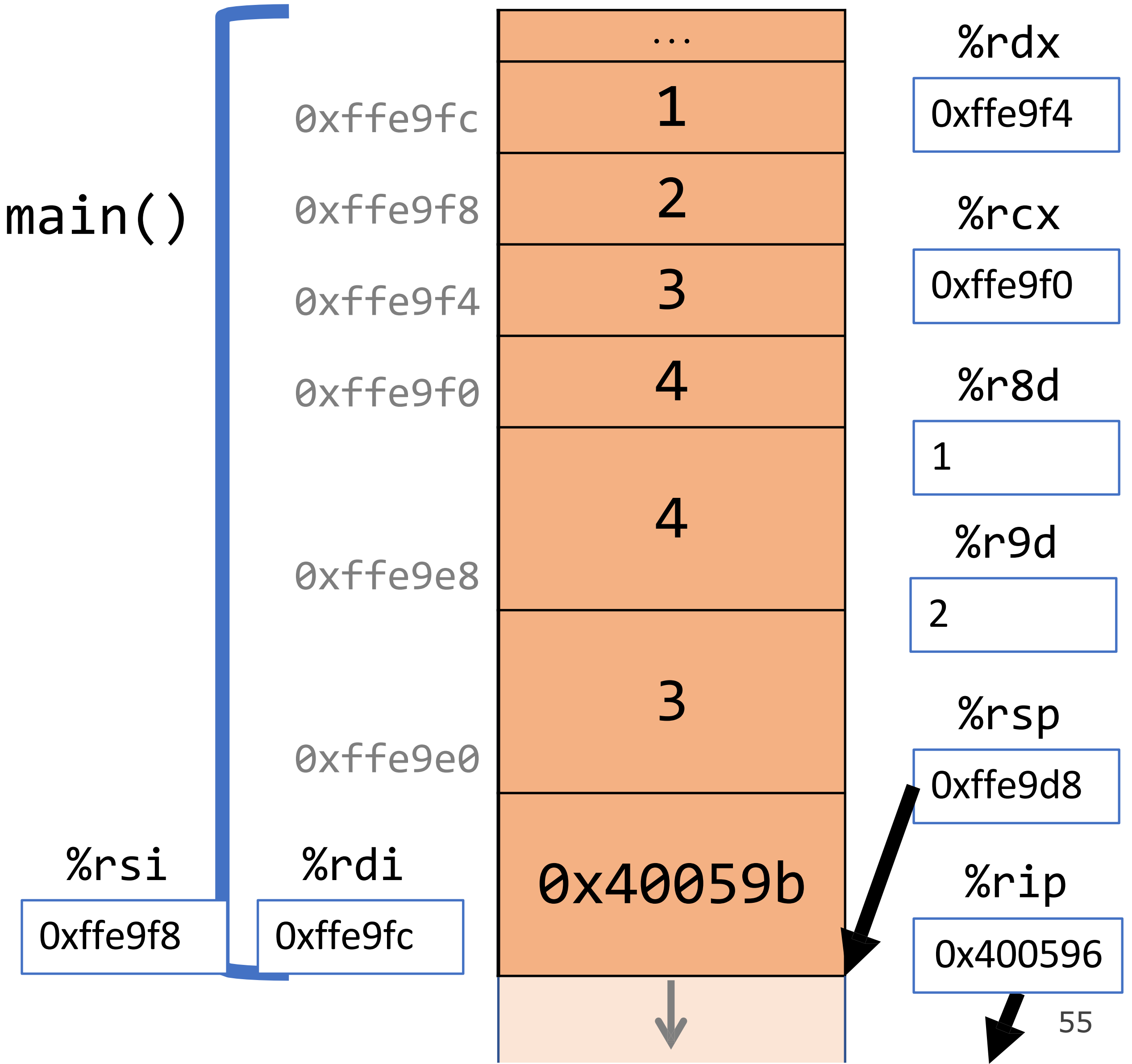


# Parameters and Return

```
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
…
```



main()

| | |
|---|---|
| | … |
| 0xffe9fc | 1 |
| 0xffe9f8 | 2 |
| 0xffe9f4 | 3 |
| 0xffe9f0 | 4 |
| 0xffe9e8 | 4 |
| 0xffe9e0 | 3 |
| | 0x40059b |

%rsi: 0xffe9f8

%rdi: 0xffe9fc

%rdx: 0xffe9f4

%rcx: 0xffe9f0

%r8d: 1

%r9d: 2

%rsp: 0xffe9d8

%rip: 0x400596

# Local Storage

- So far, we’ve often seen local variables stored directly in registers, rather than on the stack.

- 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)

# Data Alignment

- Computer systems often put restrictions on the allowable addresses for primitive data types, requiring that the address for some objects must be a multiple of some value *K* (normally 2, 4, or 8).
- These ***alignment restrictions*** simplify the design of the hardware.
- For example, suppose that a processor always fetches 8 bytes from the memory system, and an address must be a multiple of 8. If we can guarantee that any `double` will be aligned to have its address as a multiple of 8, then we can read or write the values with a single memory access.
- For x86-64, Intel recommends the following alignments for best performance:

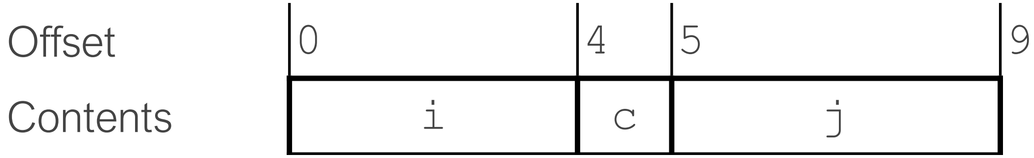
| *K* | Types |
|---|---|
| 1 | `char` |
| 2 | `short` |
| 4 | `int, float` |
| 8 | `long, double, char *` |

# Data Alignment

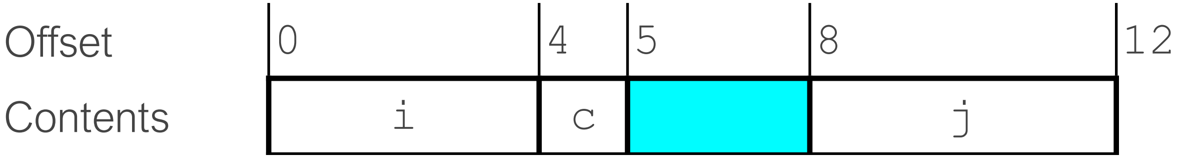
- The compiler enforces alignment by making sure that every data type is organized in such a way that every field within the struct satisfies the alignment restrictions.
- For example, let's look at the following struct:

```
struct S1 {
    int i;
    char c;
    int j;
};
```

- If the compiler used a minimal allocation:



- This would make it impossible to align fields `i` (offset 0) and `j` (offset 5). Instead, the compiler inserts a 3-byte gap between fields `c` and `j`:



- So, don't be surprised if your structs have a `sizeof()` that is larger than you expect!

# GCC Optimizations

# Optimization

Most of what you need to do with optimization can be summarized by:

1) If doing something seldom and only on small inputs, do whatever is simplest to code, understand, and debug
2) If doing things a lot, or on big inputs, make the primary algorithm's Big-O cost reasonable
3) **Let gcc do its magic from there**
4) Optimize explicitly as a last resort

# 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).

# GCC For Loop Output

**GCC Common For Loop Output**

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

**Possible Alternative**

```
Initialization
Jump to test
Body
Update
Test
Jump to body if success
```

# GCC For Loop Output

**GCC Common For Loop Output**

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

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

# GCC For Loop Output

**GCC Common For Loop Output**

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

**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
...
```

# GCC For Loop Output

**GCC Common For Loop Output**

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

**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
...
```

# GCC For Loop Output

```
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
...
```

**Possible Alternative**

```
Initialization
Jump to test
Body
Update
Test
Jump to body if success
```

# GCC For Loop Output

```
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
...
```

**Possible Alternative**

```
Initialization
Jump to test
Body
Update
Test
Jump to body if success
```

# GCC For Loop Output

**GCC Common For Loop Output**

```
Initialization
Test
Jump past loop if passes
Body
Update
Jump to test
```

**Possible Alternative**

```
Initialization
Jump to test
Body
Update
Test
Jump to body if success
```

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, left (GCC common output) is best b/c fewer instructions
  - If n is large, right (alternative) 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.

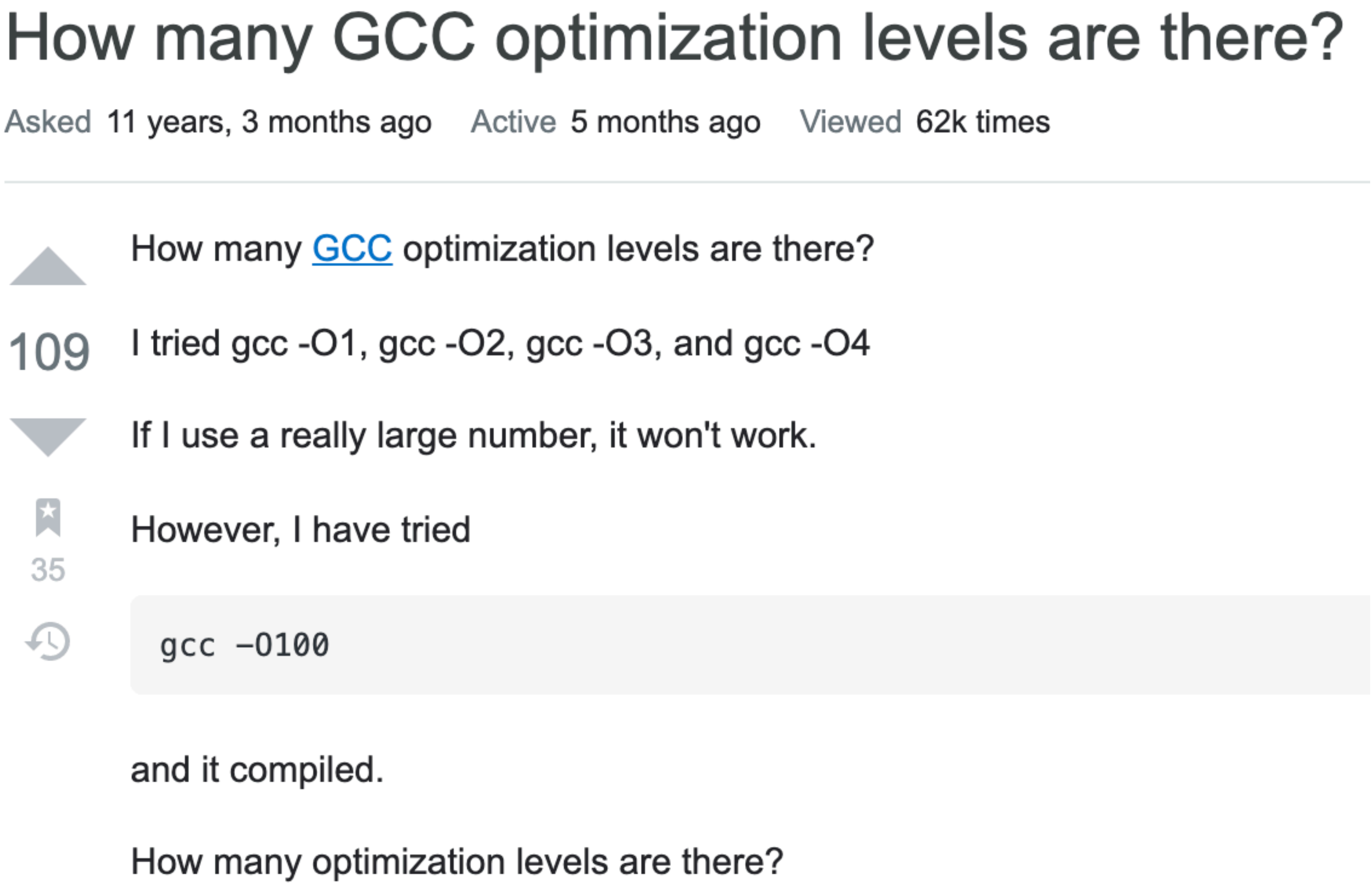
# GCC Optimization

- Today, we'll be comparing two levels of optimization in the gcc compiler:
  - `gcc –O0  // mostly just literal translation of C`
  - `gcc –O2  // enable nearly all reasonable optimizations`
  - (we also use –Og, like –O0 but more debugging friendly)

- There are other custom and more aggressive levels of optimization, e.g.:
  - `-O3     //more aggressive than O2, trade size for speed`
  - `-Os     //optimize for size`
  - `-Ofast  //disregard standards compliance (!!)`

- Exhaustive list of gcc optimization-related flags:
  - https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

# Compiler optimizations



## How many GCC optimization levels are there?

Asked 11 years, 3 months ago Active 5 months ago Viewed 62k times

109

35

How many GCC optimization levels are there?

I tried gcc -O1, gcc -O2, gcc -O3, and gcc -O4

If I use a really large number, it won't work.

However, I have tried

```
gcc -O100
```

and it compiled.

How many optimization levels are there?

Gcc supports numbers up to 3. Anything above is interpreted as 3

https://stackoverflow.com/questions/1778538/how-many-gcc-optimization-levels-are-there

# GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling

# Constant Folding

**Constant Folding** pre-calculates constants at compile-time where possible.

```
int seconds = 60 * 60 * 24 * n_days;
```

# Constant Folding

```
int fold(int param) {
    char arr[5];
    int a = 0x107;
    int b = a * sizeof(arr);
    int c = sqrt(2.0);
    return a * param + (a + 0x15 / c + strlen("Hello") * b - 0x37) / 4;
}
```

# Constant Folding: Before (-O0)

```
00000000000011b9 <fold>:
    11b9:   55                      push   %rbp
    11ba:   48 89 e5                mov    %rsp,%rbp
    11bd:   41 54                   push   %r12
    11bf:   53                      push   %rbx
    11c0:   48 83 ec 30             sub    $0x30,%rsp
    11c4:   89 7d cc                mov    %edi,-0x34(%rbp)
    11c7:   c7 45 ec 07 01 00 00    movl   $0x107,-0x14(%rbp)
    11ce:   8b 45 ec                mov    -0x14(%rbp),%eax
    11d1:   48 98                   cltq
    11d3:   89 c2                   mov    %eax,%edx
    11d5:   89 d0                   mov    %edx,%eax
    11d7:   c1 e0 02                shl    $0x2,%eax
    11da:   01 d0                   add    %edx,%eax
    11dc:   89 45 e8                mov    %eax,-0x18(%rbp)
    11df:   48 8b 05 2a 0e 00 00    mov    0xe2a(%rip),%rax        # 2010 <_IO_stdin_used+0x10>
    11e6:   66 48 0f 6e c0          movq   %rax,%xmm0
    11eb:   e8 b0 fe ff ff          callq  10a0 <sqrt@plt>
    11f0:   f2 0f 2c c0             cvttsd2si %xmm0,%eax
    11f4:   89 45 e4                mov    %eax,-0x1c(%rbp)
    11f7:   8b 45 ec                mov    -0x14(%rbp),%eax
    11fa:   0f af 45 cc             imul   -0x34(%rbp),%eax
    11fe:   41 89 c4                mov    %eax,%r12d
    1201:   b8 15 00 00 00          mov    $0x15,%eax
    1206:   99                      cltd
    1207:   f7 7d e4                idivl  -0x1c(%rbp)
    120a:   89 c2                   mov    %eax,%edx
    120c:   8b 45 ec                mov    -0x14(%rbp),%eax
    120f:   01 d0                   add    %edx,%eax
    1211:   48 63 d8                movslq %eax,%rbx
    1214:   48 8d 3d ed 0d 00 00    lea    0xded(%rip),%rdi        # 2008 <_IO_stdin_used+0x8>
    121b:   e8 20 fe ff ff          callq  1040 <strlen@plt>
    1220:   8b 55 e8                mov    -0x18(%rbp),%edx
    1223:   48 63 d2                movslq %edx,%rdx
    1226:   48 0f af c2             imul   %rdx,%rax
    122a:   48 01 d8                add    %rbx,%rax
    122d:   48 83 e8 37             sub    $0x37,%rax
    1231:   48 c1 e8 02             shr    $0x2,%rax
    1235:   44 01 e0                add    %r12d,%eax
    1238:   48 83 c4 30             add    $0x30,%rsp
    123c:   5b                      pop    %rbx
    123d:   41 5c                   pop    %r12
    123f:   5d                      pop    %rbp
    1240:   c3                      retq
```

# Constant Folding: After (-O2)

```
00000000000011b0 <fold>:
    11b0:    69 c7 07 01 00 00       imul   $0x107,%edi,%eax
    11b6:    05 a5 06 00 00          add    $0x6a5,%eax
    11bb:    c3                      retq
```

What is the consequence of this for you as a programmer?  What should you do differently or the same knowing that compilers can do this for you?

# GCC Optimizations

- Constant Folding
- **Common Sub-expression Elimination**
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling

# Common Sub-Expression Elimination

**Common Sub-Expression Elimination** prevents the recalculation of the same thing many times by doing it once and saving the result.

```
int a = (param2 + 0x107);
int b = param1 * (param2 + 0x107) + a;
return a * (param2 + 0x107) + b * (param2 + 0x107);
```

# Common Sub-Expression Elimination

**Common Sub-Expression Elimination** prevents the recalculation of the same thing many times by doing it once and saving the result.

```
int a = (param2 + 0x107);
int b = param1 * (param2 + 0x107) + a;
return a * (param2 + 0x107) + b * (param2 + 0x107);
// = 2 * a * a + param1 * a * a
```

```
0000000000011b0 <subexp>:  // param1 in %edi, param2 in %esi
    11b0:  lea    0x107(%rsi),%eax       // %eax stores a
    11b6:  imul   %eax,%edi              // param1 * a
    11b9:  lea    (%rdi,%rax,2),%esi     // 2 * a + param1 * a
    11bc:  imul   %esi,%eax              // a * (2 * a + param1 * a)
    11bf:  retq
```

# Common Sub-Expression Elimination

*Why should we bother saving repeated calculations in variables if the compiler has common subexpression elimination?*

- The compiler may not always be able to optimize every instance. Plus, it can help reduce redundancy!
- Makes code more readable!

# GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- **Dead Code**
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling

# Dead Code

**Dead code elimination** removes code that doesn't serve a purpose:

```
if (param1 < param2 && param1 > param2) {
    printf("This test can never be true!\n");
}

// Empty for loop
for (int i = 0; i < 1000; i++);

// If/else that does the same operation in both cases
if (param1 == param2) {
    param1++;
} else {
    param1++;
}

// If/else that more trickily does the same operation in both cases
if (param1 == 0) {
    return 0;
} else {
    return param1;
}
```

# Dead Code: Before (-O0)

```
00000000000011a9 <dead_code>:
    11a9:	55                   	push   %rbp
    11aa:	48 89 e5             	mov    %rsp,%rbp
    11ad:	48 83 ec 20          	sub    $0x20,%rsp
    11b1:	89 7d ec             	mov    %edi,-0x14(%rbp)
    11b4:	89 75 e8             	mov    %esi,-0x18(%rbp)
    11b7:	8b 45 ec             	mov    -0x14(%rbp),%eax
    11ba:	3b 45 e8             	cmp    -0x18(%rbp),%eax
    11bd:	7d 19                	jge    11d8 <dead_code+0x2f>
    11bf:	8b 45 ec             	mov    -0x14(%rbp),%eax
    11c2:	3b 45 e8             	cmp    -0x18(%rbp),%eax
    11c5:	7e 11                	jle    11d8 <dead_code+0x2f>
    11c7:	48 8d 3d 36 0e 00 00 	lea    0xe36(%rip),%rdi        # 2004 <_IO_stdin_used+0x4>
    11ce:	b8 00 00 00 00       	mov    $0x0,%eax
    11d3:	e8 68 fe ff ff       	callq  1040 <printf@plt>
    11d8:	c7 45 fc 00 00 00 00 	movl   $0x0,-0x4(%rbp)
    11df:	eb 04                	jmp    11e5 <dead_code+0x3c>
    11e1:	83 45 fc 01          	addl   $0x1,-0x4(%rbp)
    11e5:	81 7d fc e7 03 00 00 	cmpl   $0x3e7,-0x4(%rbp)
    11ec:	7e f3                	jle    11e1 <dead_code+0x38>
    11ee:	8b 45 ec             	mov    -0x14(%rbp),%eax
    11f1:	3b 45 e8             	cmp    -0x18(%rbp),%eax
    11f4:	75 06                	jne    11fc <dead_code+0x53>
    11f6:	83 45 ec 01          	addl   $0x1,-0x14(%rbp)
    11fa:	eb 04                	jmp    1200 <dead_code+0x57>
    11fc:	83 45 ec 01          	addl   $0x1,-0x14(%rbp)
    1200:	83 7d ec 00          	cmpl   $0x0,-0x14(%rbp)
    1204:	75 07                	jne    120d <dead_code+0x64>
    1206:	b8 00 00 00 00       	mov    $0x0,%eax
    120b:	eb 03                	jmp    1210 <dead_code+0x67>
    120d:	8b 45 ec             	mov    -0x14(%rbp),%eax
    1210:	c9                   	leaveq
    1211:	c3                   	retq
```

# Dead Code: After (-O2)

```
0000000000011b0 <dead_code>:
    11b0:    8d 47 01                        lea    0x1(%rdi),%eax
    11b3:    c3                              retq
```

# GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- **Strength Reduction**
- Code Motion
- Tail Recursion
- Loop Unrolling

# Strength Reduction

**Strength reduction** changes divide to multiply, multiply to add/shift, and mod to AND to avoid using instructions that cost many cycles (multiply and divide).

```
int a = param2 * 32;
int b = a * 7;
int c = b / 2;
int d = param2 % 2;

for (int i = 0; i <= param2; i++) {
    c += param1[i] + 0x107 * i;
}
return c + d;
```

# Shifting into Shifts

- `int a = param2 * 32;`

  Becomes:

- `int a = param2 * 32;`

- `int b = a * 7;`

  Becomes:

- `int b = a + (a << 2) + (a << 1);`

- `int c = b / 2;`

  Becomes

- `int c = b >> 1`

# GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- **Code Motion**
- Tail Recursion
- Loop Unrolling

# Code Motion

**Code motion** moves code outside of a loop if possible.

```
for (int i = 0; i < n; i++) {
  sum += arr[i] + foo * (bar + 3);
}
```

Common subexpression elimination deals with expressions that appear multiple times in the code.  Here, the expression appears once, but is calculated each loop iteration, even though none of its values change during the loop.

# Code Motion

**Code motion** moves code outside of a loop if possible.

```
int temp = foo * (bar + 3);
for (int i = 0; i < n; i++) {
  sum += arr[i] + temp;
}
```

Moving it out of the loop allows the computation to happen only once.

# Practice: GCC Optimization

```
int char_sum(char *s) {
    int sum = 0;
    for (size_t i = 0; i < strlen(s); i++) {
        sum += s[i];
    }
    return sum;
}
```

What is the bottleneck?  What (if anything) can GCC do?

# Practice: GCC Optimization

```
int char_sum(char *s) {
    int sum = 0;
    for (size_t i = 0; i < strlen(s); i++) {
        sum += s[i];
    }
    return sum;
}
```

What is the bottleneck?  What (if anything) can GCC do?

**strlen is called every loop iteration – code motion can pull it out of the loop**

# Tail Recursion

**Tail recursion** is an example of where GCC can identify recursive patterns that can be more efficiently implemented iteratively.

```
long factorial(int n) {
  if (n <= 1) {
      return 1;
  }
  else return n * factorial(n - 1);
}
```

# Tail recursion example: Lab6 bonus

Recall the factorial problem from assembly lectures:

```
unsigned int factorial(unsigned int n) {
    if (n <= 1) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

What happens with **`factorial(-1)`**?

- Infinite recursion → Literal stack overflow!
- Compiled with `-0g`!

# Factorial: -Og vs –O2

```
401146 <+0>:  cmp    $0x1,%edi
401149 <+3>:  jbe    0x40115b <factorial+21>
40114b <+5>:  push   %rbx
40114c <+6>:  mov    %edi,%ebx
40114e <+8>:  lea    -0x1(%rdi),%edi
401151 <+11>:callq   0x401146 <factorial>
401156 <+16>:imul    %ebx,%eax
401159 <+19>:pop     %rbx
40115a <+20>:retq
40115b <+21>:mov     $0x1,%eax
401160 <+26>:retq
```

-O2:

- What happened?
- Did the compiler “fix” the infinite recursion?

```
4011e0 <+0>:  mov    $0x1,%eax
4011e5 <+5>:  cmp    $0x1,%edi
4011e8 <+8>:  jbe    0x4011fd <factorial+29>
4011ea <+10>:nopw    0x0(%rax,%rax,1)
4011f0 <+16>:mov     %edi,%edx
4011f2 <+18>:sub     $0x1,%edi
4011f5 <+21>:imul    %edx,%eax
4011f8 <+24>:cmp     $0x1,%edi
4011fb <+27>:jne     0x4011f0 <factorial+16>
4011fd <+29>:retq
```

# Breaking Down the –O2

```
4011e0 <+0>:  mov    $0x1,%eax                      # Initialize %eax with 1.
4011e5 <+5>:  cmp    $0x1,%edi                      # Compare input value (%edi) with 1.
4011e8 <+8>:  jbe    0x4011fd <factorial+29>        # If input <= 1 (unsigned check), jump to return.
4011ea <+10>: nopw   0x0(%rax,%rax,1)               # No operation (probably for alignment).
4011f0 <+16>: mov    %edi,%edx                      # Copy current value of %edi to %edx.
4011f2 <+18>: sub    $0x1,%edi                      # Decrement %edi.
4011f5 <+21>: imul   %edx,%eax                      # Multiply %eax by %edx and store result in %eax.
4011f8 <+24>: cmp    $0x1,%edi                      # Compare decremented value of %edi with 1.
4011fb <+27>: jne    0x4011f0 <factorial+16>        # If %edi is not 1, repeat the multiplication.
4011fd <+29>: retq                                  # Return with the result in %eax.
```

-02:

- Recursive -> Iterative
- No Stack Overflow, Saves Memory and Operations

# GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- **Loop Unrolling**

# Loop Unrolling

**Loop Unrolling:** Do **n** loop iterations' worth of work per actual loop iteration, so we save ourselves from doing the loop overhead (test and jump) every time, and instead incur overhead only every n-th time.

```
for (int i = 0; i <= n - 4; i += 4) {
     sum += arr[i];
     sum += arr[i + 1];
     sum += arr[i + 2];
     sum += arr[i + 3];
} // after the loop handle any leftovers
```

# Some Extra Reading

# Key GDB Tips For Assembly

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

```
(gdb) x/4g $rsp
0x7fffffffe870: 0x0000000000000005      0x0000000000400559
0x7fffffffe880: 0x0000000000000000      0x0000000000400575
```

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

```
(gdb) display/4w $rsp
1: x/4xw $rsp
0x7fffffffe8a8:
0xf7a2d830      0x00007fff      0x00000000      0x00000000
```

# Key GDB Tips For Assembly

- `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`

# gdb tips

| | |
|---|---|
| `layout split` (ctrl-x a: exit, ctrl-l: resize) | View C, assembly, and gdb (lab5) |
| `info reg` | Print all registers |
| `p $eax` | Print register value |
| `p $eflags` | Print all condition codes currently set |
| `b *0x400546` | Set breakpoint at assembly instruction |
| `b *0x400550 if $eax > 98` | Set **conditional breakpoint** |
| `ni` | Next assembly instruction |
| `si` | Step into assembly instruction (will step into function calls) |

# gdb tips

| | |
|---|---|
| `p/x $rdi` | Print register value in hex |
| `p/t $rsi` | Print register value in binary |
| | |
| `x $rdi` | Examine the byte stored at this address |
| `x/4bx $rdi` | Examine 4 bytes starting at this address |
| `x/4wx $rdi` | Examine 4 ints starting at this address |

# Array Allocation and Access

- Arrays in C map in a fairly straightforward way to X86 assembly code, thanks to the addressing modes available in instructions.
- When we perform pointer arithmetic, the assembly code that is produced will have address computations built into them.
- Optimizing compilers are ***very*** good at simplifying the address computations (in lab you will see another optimizing compiler benefit in the form of division — if the compiler can avoid dividing, it will!). Because of the transformations, compiler-generated assembly for arrays often doesn't look like what you are expecting.
- Consider the following form of a data type *T* and integer constant *N:*

*T* `A[`*N*`]`

- The starting location is designated as $x_A$
- The declaration allocates *N* `* sizeof(`*T*`)` bytes, and gives us an identifier that we can use as a pointer (but it isn't a pointer!), with a value of $x_A$.

# Array Allocation and Access

- Example:

```
char     A[12];
char    *B[8];
int      C[6];
double  *D[5]
```

| Array | Element Size | Total Size | Start address | Element i |
|---|---|---|---|---|
| A | 1 | 12 | $x_A$ | $x_A + i$ |
| B | 8 | 64 | $x_B$ | $x_B + 8i$ |
| C | 4 | 24 | $x_C$ | $x_C + 4i$ |
| D | 8 | 40 | $x_D$ | $x_D + 8i$ |

- The memory referencing operations in x86-64 are designed to simplify array access. Suppose we wanted to access `C[3]` above. If the address of C is in register `%rdx`, and `3` is in register `%rcx`
- The following copies `C[3]` into `%eax`,

```
movl (%rdx,%rcx,4), %eax
```

# Pointer Arithmetic

- C allows arithmetic on pointers, where the computed value is calculated according to the size of the data type referenced by the pointer.
- The array reference `A[i]` is identical to `*(A+i)`
- Example: if the address of array E is in `%rdx`, and the integer index, `i`, is in `%rcx`, the following are some expressions involving E:

| Expression | Type | Value | Assembly Code |
|---|---|---|---|
| `E` | int * | $x_E$ | `movq %rdx, %rax` |
| | int | | |
| `E[0]` | int | $M[x_E]$ | `movl (%rdx), %eax` |
| `E[i]` | int | $M[x_E+4i]$ | `movl (%rdx,%rcx,4) %eax` |
| `&E[2]` | int * | $x_E+8$ | `leaq 8(%rdx), %rax` |
| `E+i-1` | int * | $x_E+4i-4$ | `leaq -4(%rdx,%rcx,4), %rax` |
| `*(E+i-3)` | int | $M[x_E+4i-12]$ | `movl -12(%rdx,%rcx,4) %eax` |
| `&E[i]-E` | long | i | `movq %rcx,%rax` |

# Pointer Arithmetic

- Practice: $x_S$ is the address of a `short` integer array, S, stored in `%rdx`, and a long integer index, `i`, is stored in register `%rcx`.
- For each of the following expressions, give its type, a formula for its value, and an assembly-code implementation. The result should be stored in `%rax` if it is a pointer, and the result should be in register `%ax` if it has a data type `short`.

| Expression | Type | Value | Assembly Code |
|---|---|---|---|
| `S+1` | `short *` | $x_S + 2$ | `leaq 2(%rdx),%rax` |
| `S[3]` | `short` | $M[x_S + 6]$ | `movw 6(%rdx),%ax` |
| `&S[i]` | `short *` | $x_S + 2i$ | `leaq (%rdx,%rcx,2),%rax` |
| `S[4*i+1]` | `short` | $M[x_S + 8i + 2]$ | `movw 2(%rdx,%rcx,8),%ax` |
| `S+i-5` | `short *` | $x_S + 2i - 10$ | `leaq -10(%rdx,%rcx,2),%rax` |

# References and Advanced

- References:
  - Stanford guide to x86-64: https://web.stanford.edu/class/cs107/guide/x86-64.html
  - CS107 one-page of x86-64: https://web.stanford.edu/class/cs107/resources/onepage_x86-64.pdf
  - gdbtui: https://beej.us/guide/bggdb/
  - More gdbtui: https://sourceware.org/gdb/onlinedocs/gdb/TUI.html
  - Compiler explorer: https://gcc.godbolt.org
- Advanced Reading:
  - Stack frame layout on x86-64: https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64
  - x86-64 Intel Software Developer manual: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf
  - history of x86 instructions: https://en.wikipedia.org/wiki/X86_instruction_listings
  - x86-64 Wikipedia: https://en.wikipedia.org/wiki/X86-64