CS107 Lecture 18
Assembly: Function Call

Reading: B&O 3.7

This document is copyright (C) Stanford Computer Science, licensed under Creative Commons Attribution 2.5 License. All rights reserved.
Based on slides created by Cynthia Lee, Chris Gregg, Jerry Cain, Lisa Yan, Nick Troccoli, and others.

Calling Functions In Assembly

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

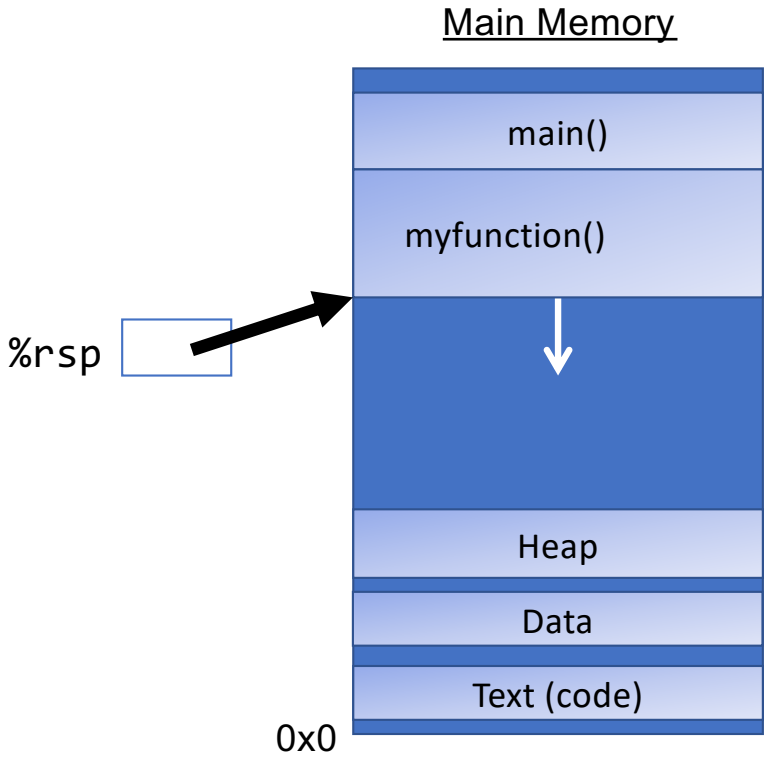
- **Transfer Control** – %rip must be adjusted to execute the callee's instructions and then resume the caller's instructions afterwards.
- **Transfer Data** – we must pass parameters and extract return values.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

How does assembly interact with the stack?

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

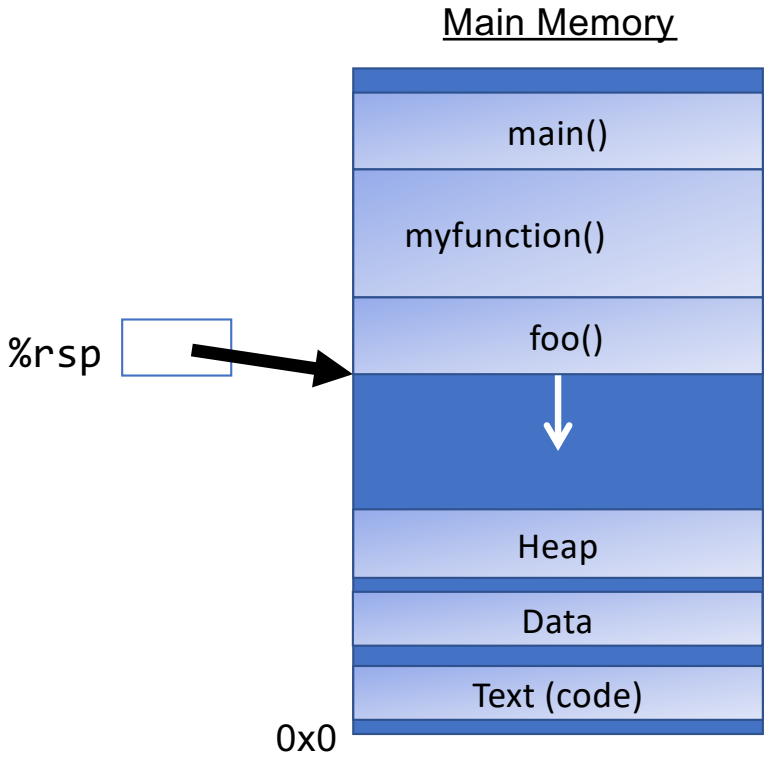
Tracking %rsp

- **%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).



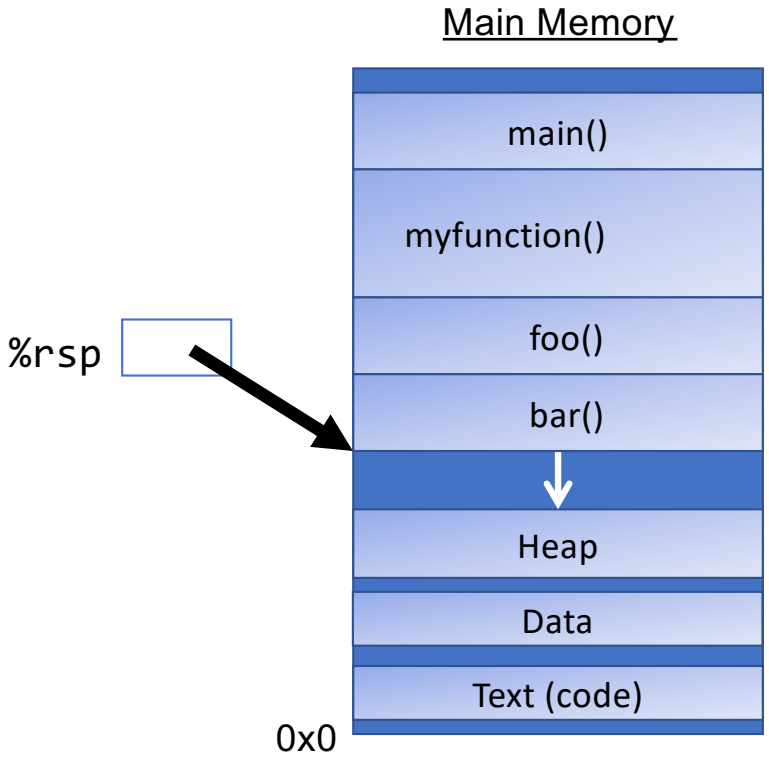
Tracking %rsp

- **%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).



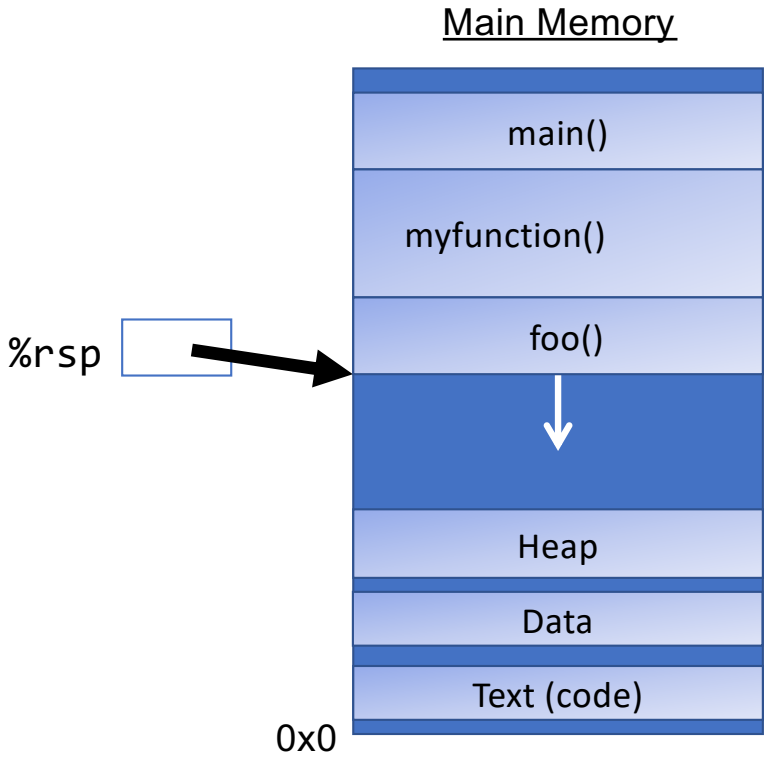
Tracking %rsp

- **%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).



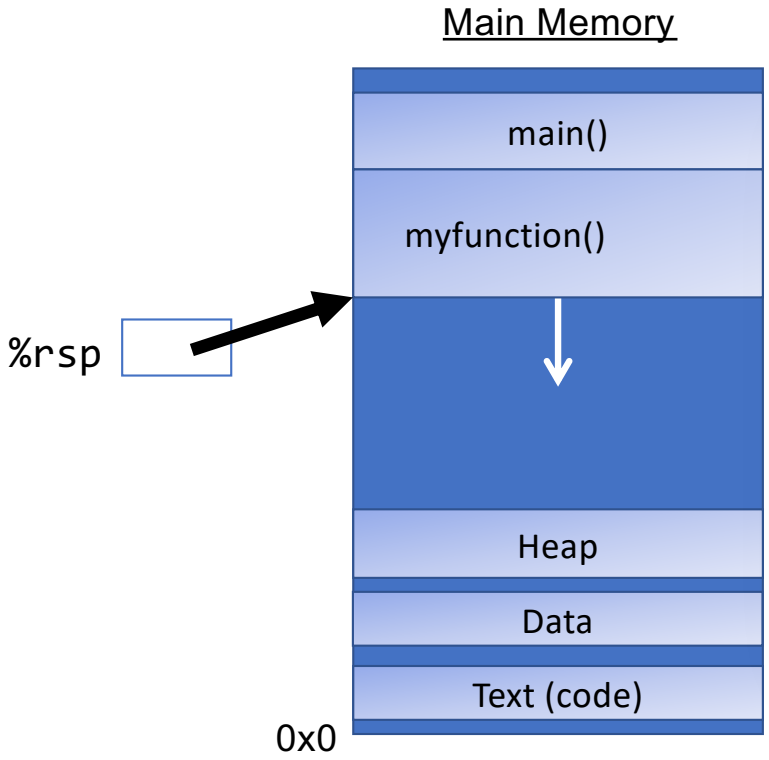
Tracking %rsp

- **%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).



Tracking %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.

Understanding push and pop

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

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

Understanding push and pop

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

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

Understanding push and pop

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

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

Understanding push and pop

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

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

- This behavior is mostly equivalent to the following, but **pushq** is shorter:

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

- Sometimes, you'll see instructions just explicitly decrement the stack pointer to make room for new local variables.

Understanding push and pop

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

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

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

Understanding push and pop

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

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

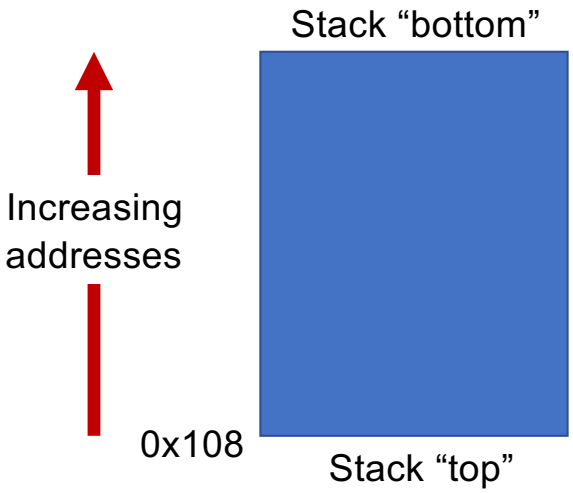
- This behavior is mostly equivalent to the following, but popq is shorter:

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

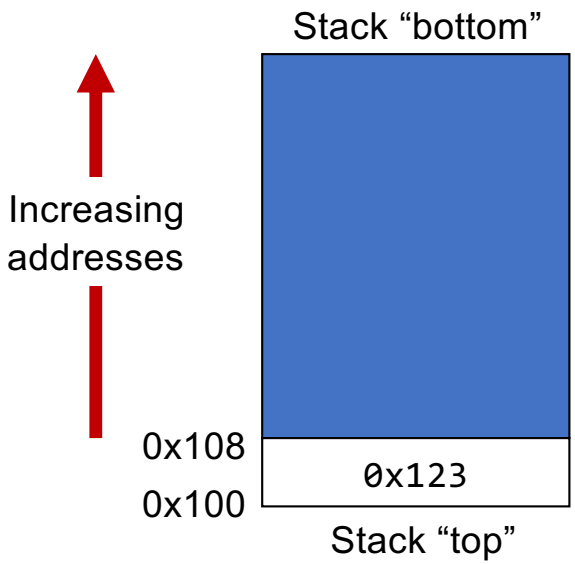
- Sometimes, you'll see instructions just explicitly increment the stack pointer to pop data.

push and pop Etude

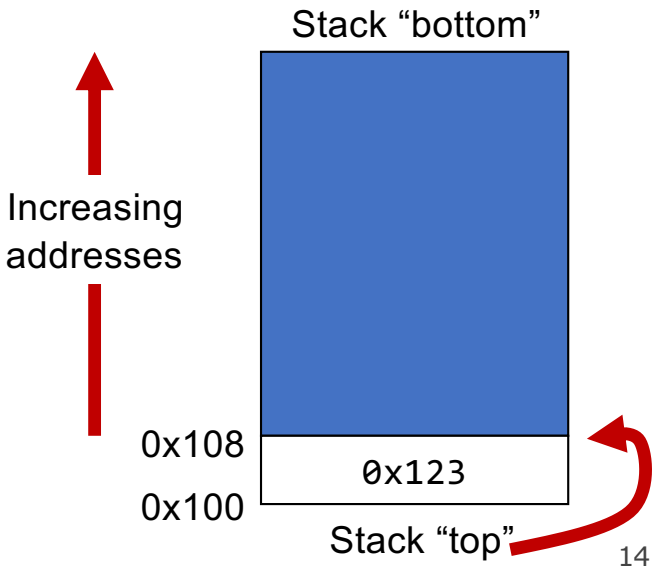
Initially	
%rax	0x123
%rdx	0
%rsp	0x108



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:

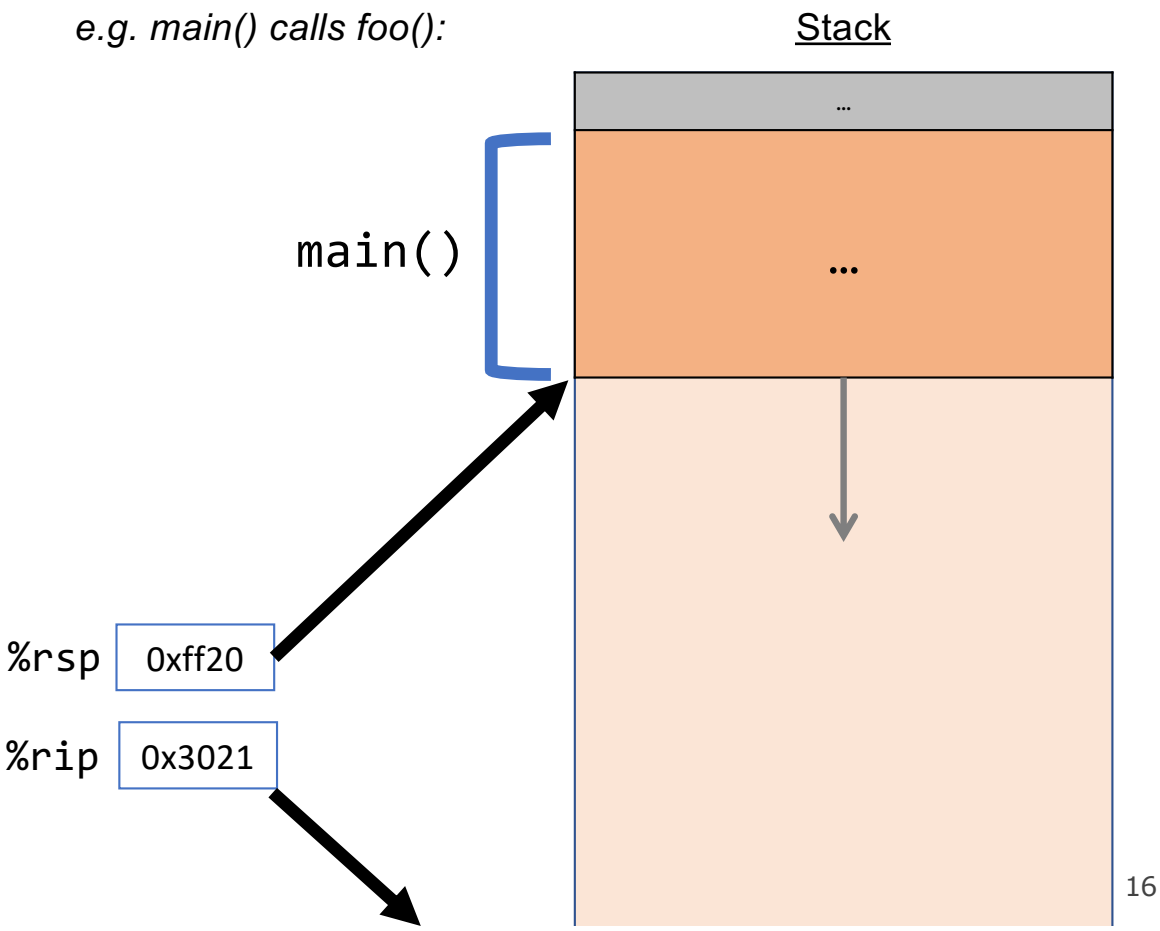
- **Transfer Control** – %rip must be adjusted to execute the callee's instructions and then resume the caller's instructions afterwards.
- **Transfer Data –** we must pass parameters and extract return **values.**
- **Manage Memory** – we must handle any space needs of the callee on the stack.

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 that instruction address for later.

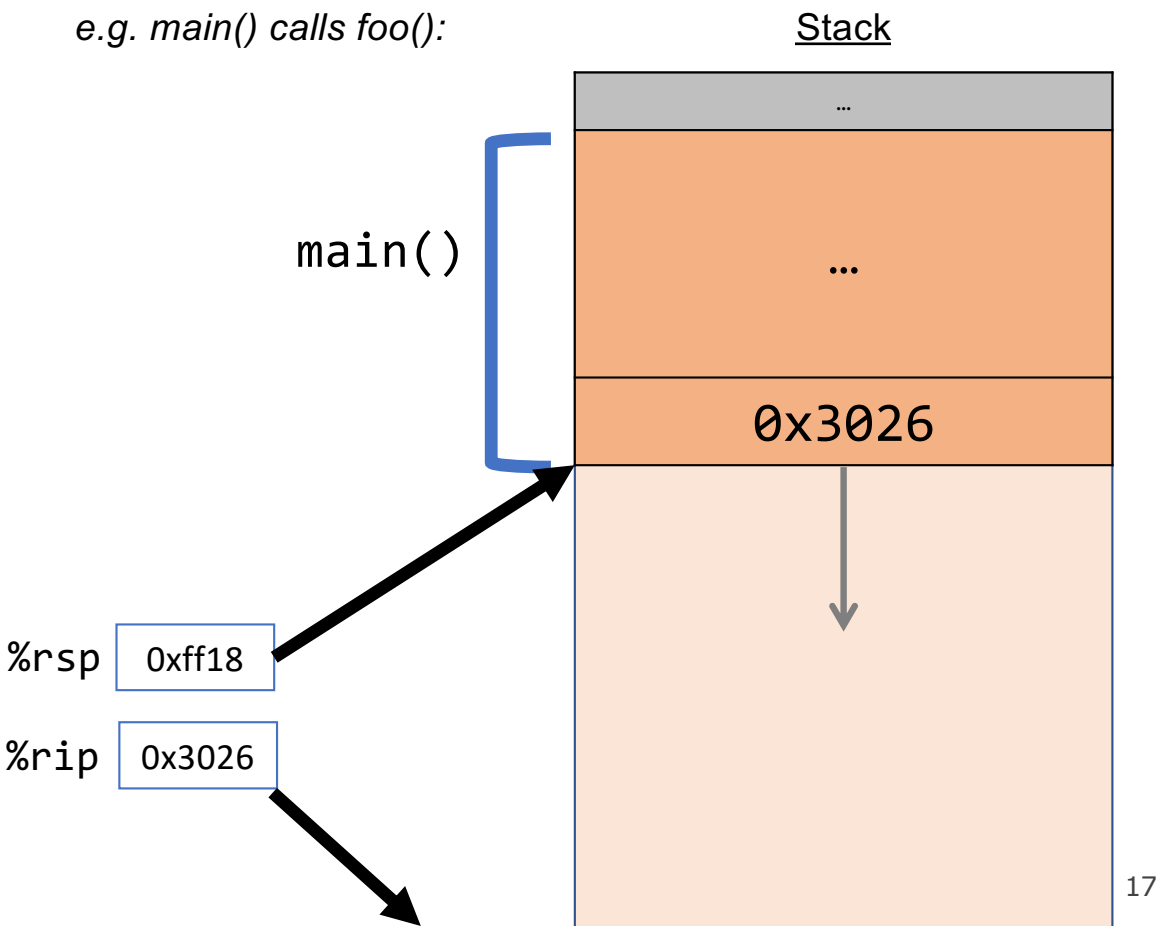
Solution: push the next value of **%rip** onto the stack. Then call the function. When it is finished, put this value back into **%rip** and continue executing as if never interrupted by the function call.



Remembering Where We Left Off

Problem: %rip points to the next instruction to execute. To call a function, we must remember that instruction address for later.

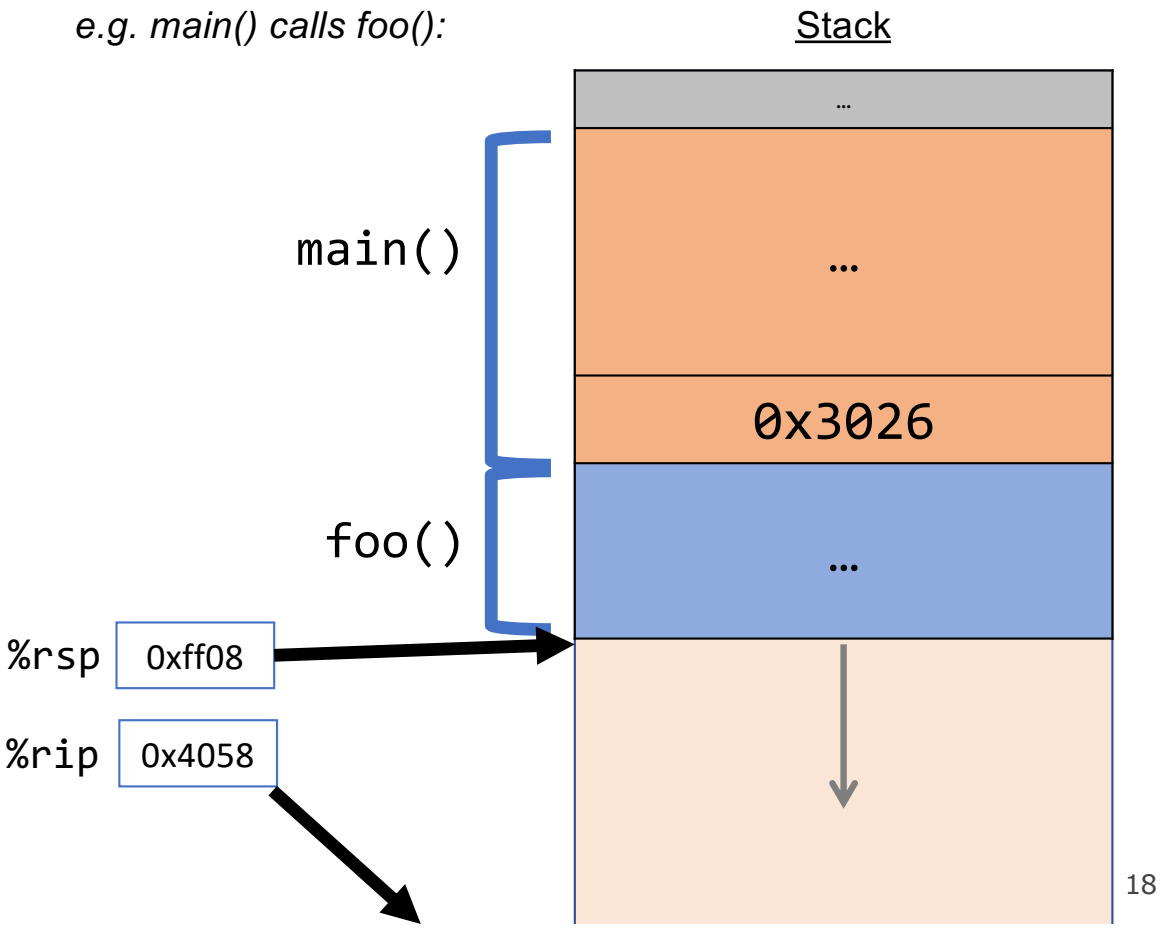
Solution: push the next value of **%rip** onto the stack. Then call the function. When it is finished, put this value back into **%rip** and continue executing as if never interrupted by the function call.



Remembering Where We Left Off

Problem: %rip points to the next instruction to execute. To call a function, we must remember that instruction address for later.

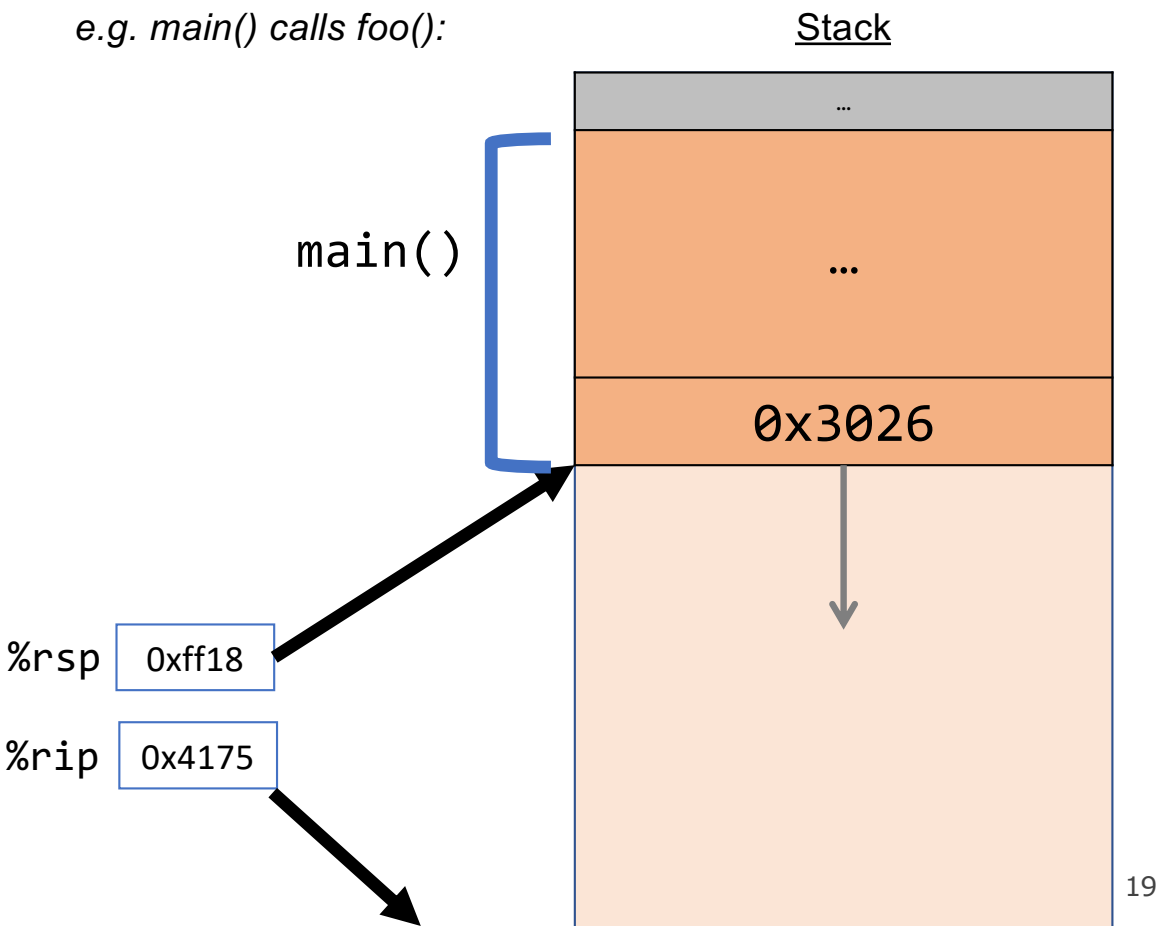
Solution: push the next value of **%rip** onto the stack. Then call the function. When it is finished, put this value back into **%rip** and continue executing as if never interrupted by the function call.



Remembering Where We Left Off

Problem: %rip points to the next instruction to execute. To call a function, we must remember that instruction address for later.

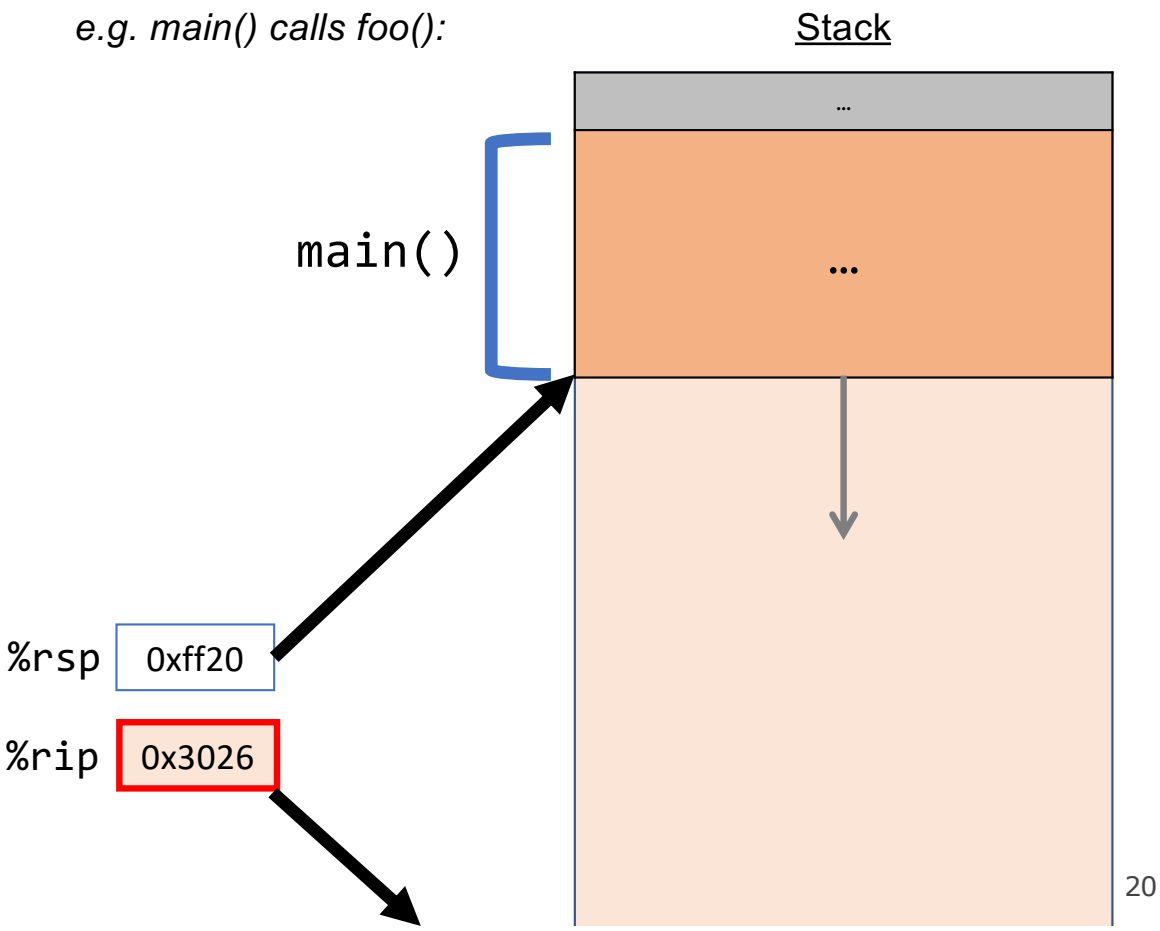
Solution: push the next value of **%rip** onto the stack. Then call the function. When it is finished, put this value back into **%rip** and continue executing as if never interrupted by the function call.



Remembering Where We Left Off

Problem: %rip points to the next instruction to execute. To call a function, we must remember that instruction address for later.

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 as if never interrupted by the function call.



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** is called the **return address.** It is the address of the instruction where execution would have continued had flow not been interrupted by the function call. (Don't confuse this with the **return value**, which is the value returned by the function via **%rax** or a subset of it).

Call and Return

What does **call** do?

call pushes the next instruction address onto the stack and overwrites %rip to address another function's very first instruction.

Call and Return

What does **ret** do?

ret pops off the 8 bytes from the top of the stack and places them in %rip, thereby resuming execution in the caller.

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

Function Pointers

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

```
call Label
call *Operand
```

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

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 placed directly on the stack.
- If the caller expects a return value, it looks in **%rax** after the callee completes.

Local Storage

- So far, all local variables have been stored directly in registers.
- There are **four** common reasons that a local variable must be stored in memory instead of a register:
  - We've simply run out of registers—we only have 16, some of which are special-purpose.
  - Not all registers are protected against function calls, so any variables or important partial results stored in registers sometimes need to be flushed out to the stack.
  - The & operator is applied to a variable, so we need a true address for it.
  - The variables themselves are arrays or structs and are better stored on the stack as an aggregate of memory.

Local Storage

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

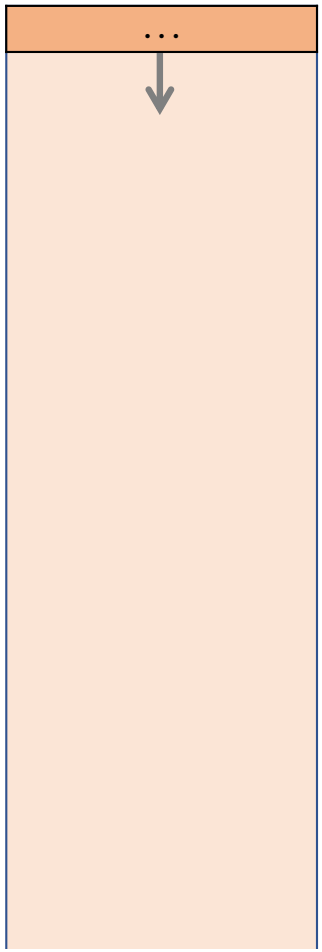
```
caller:
    sub    $0x18, %rsp          // 16 for locals, 8 to align
    movq   $0x216, 0x8(%rsp)    // store 534 in arg1
    movq   $0x421, (%rsp)       // store 1057 in arg2
    mov    %rsp, %rsi           // compute &arg2 as second arg
    lea    0x8(%rsp), %rdi      // compute &arg1 as first arg
    callq swap_add              // call swap_add(&arg1, &arg2)
```

Parameters and Return

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

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

main()



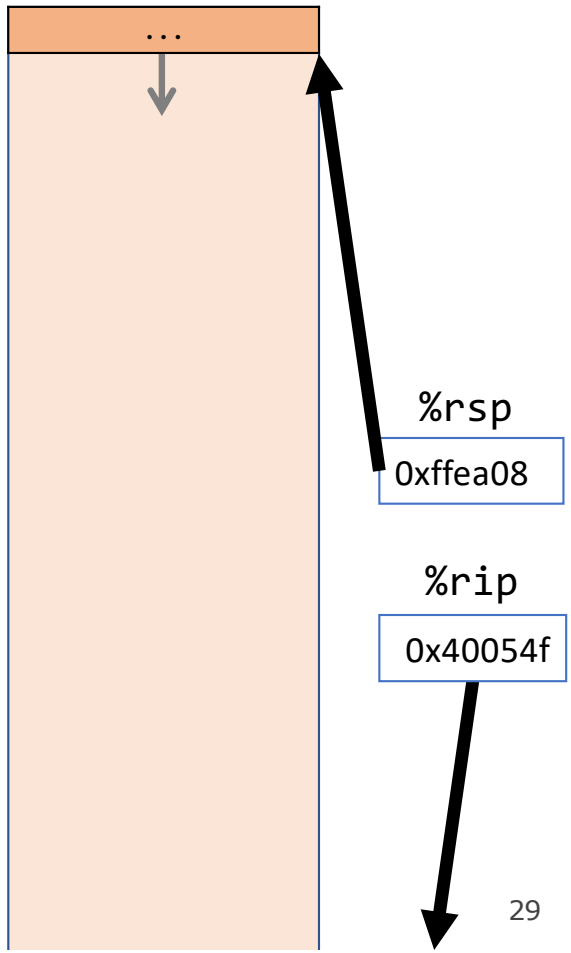
Parameters and Return

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

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

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

main()

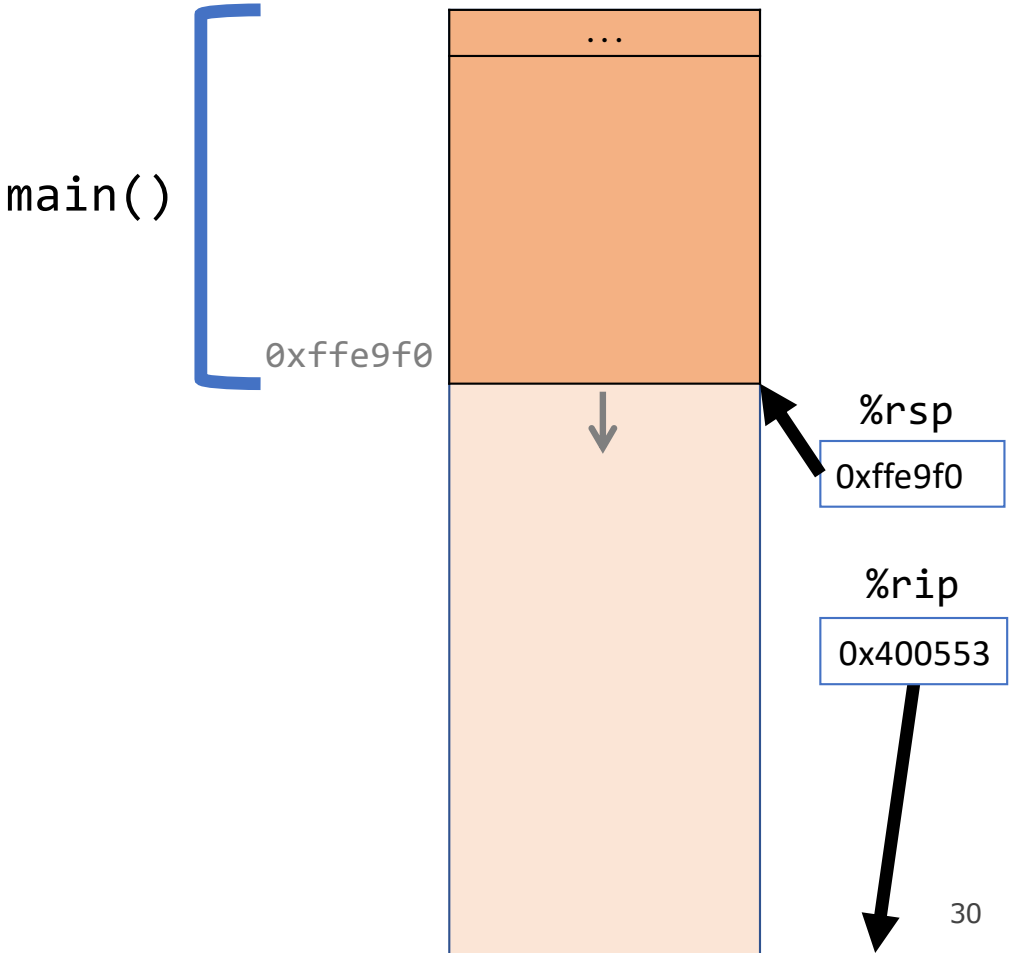


Parameters and Return

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

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

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

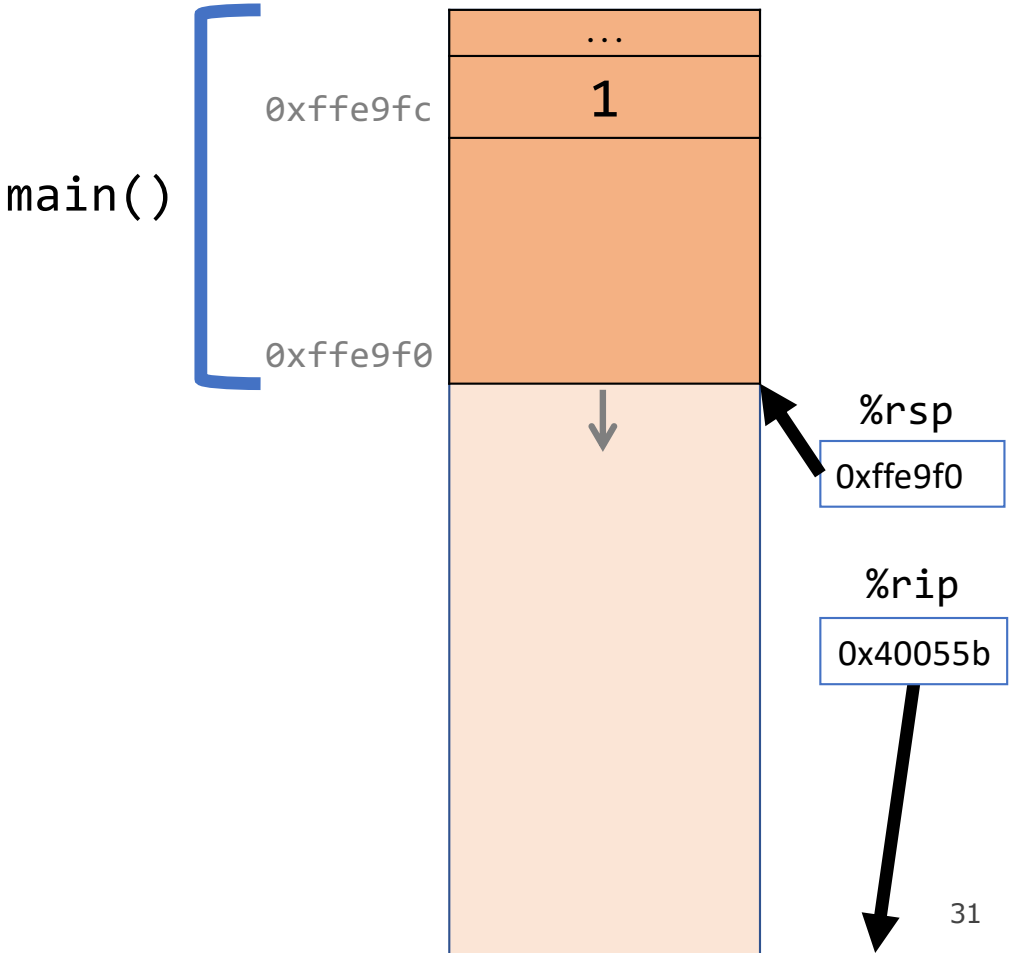


Parameters and Return

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

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

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

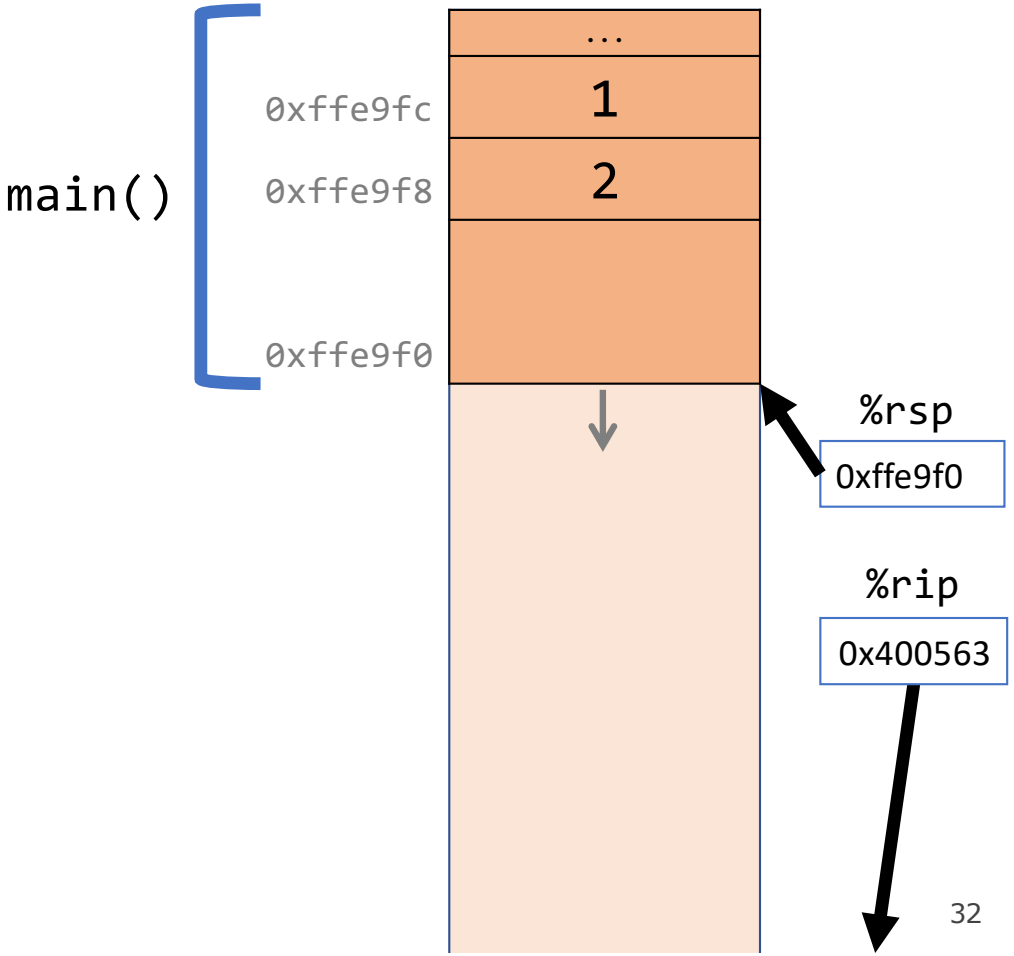


Parameters and Return

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

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

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

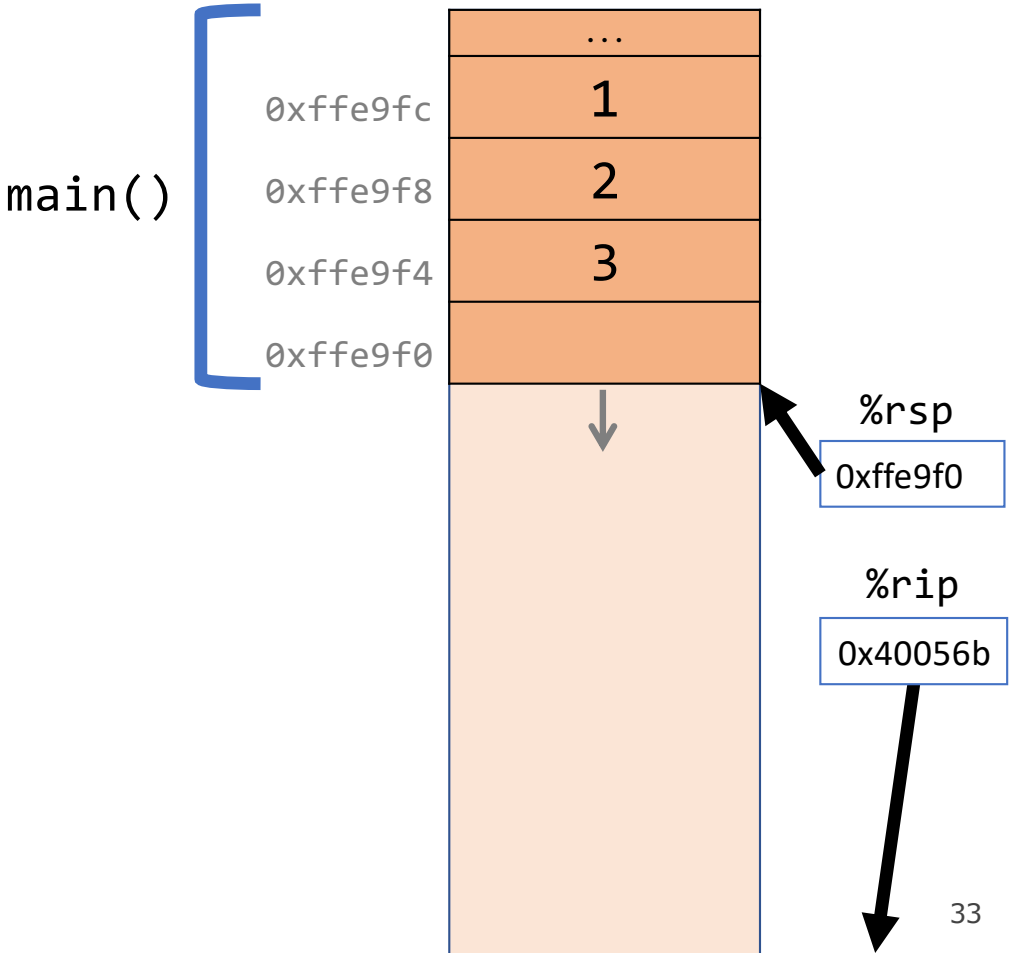


Parameters and Return

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

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

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

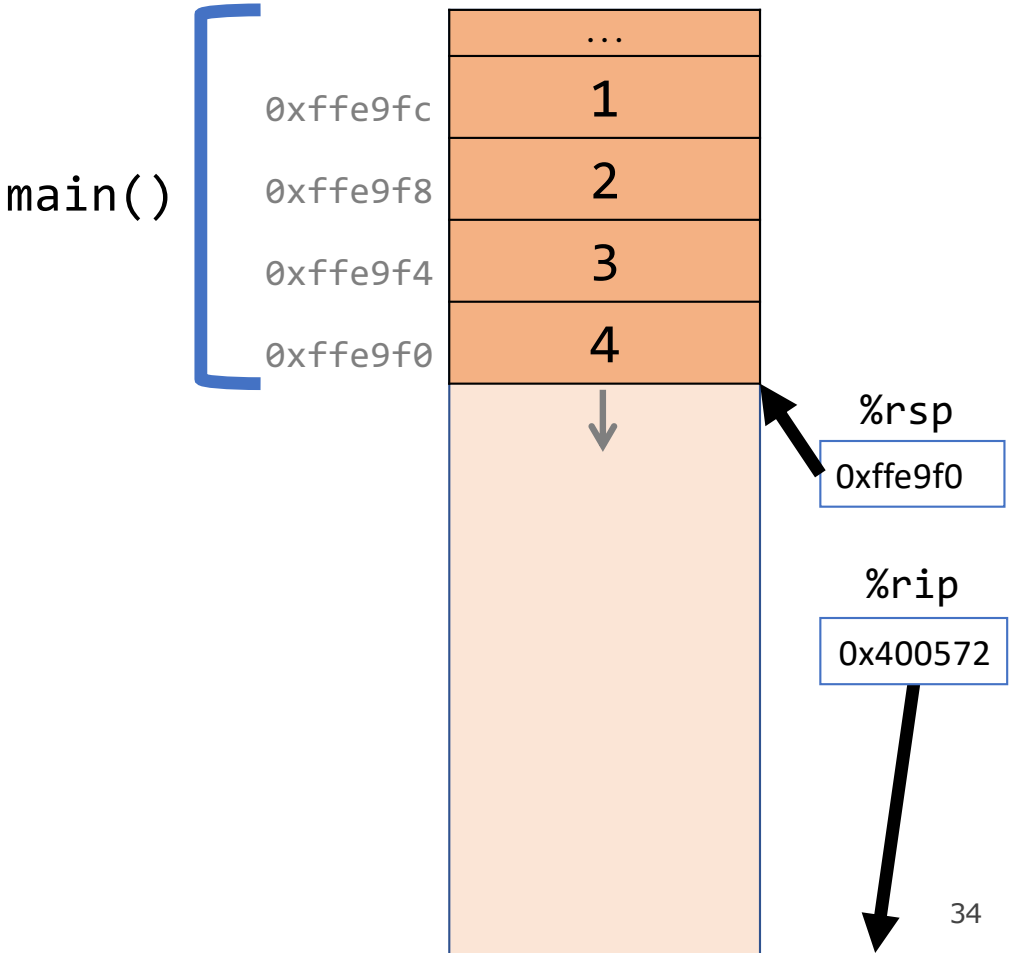


Parameters and Return

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

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

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

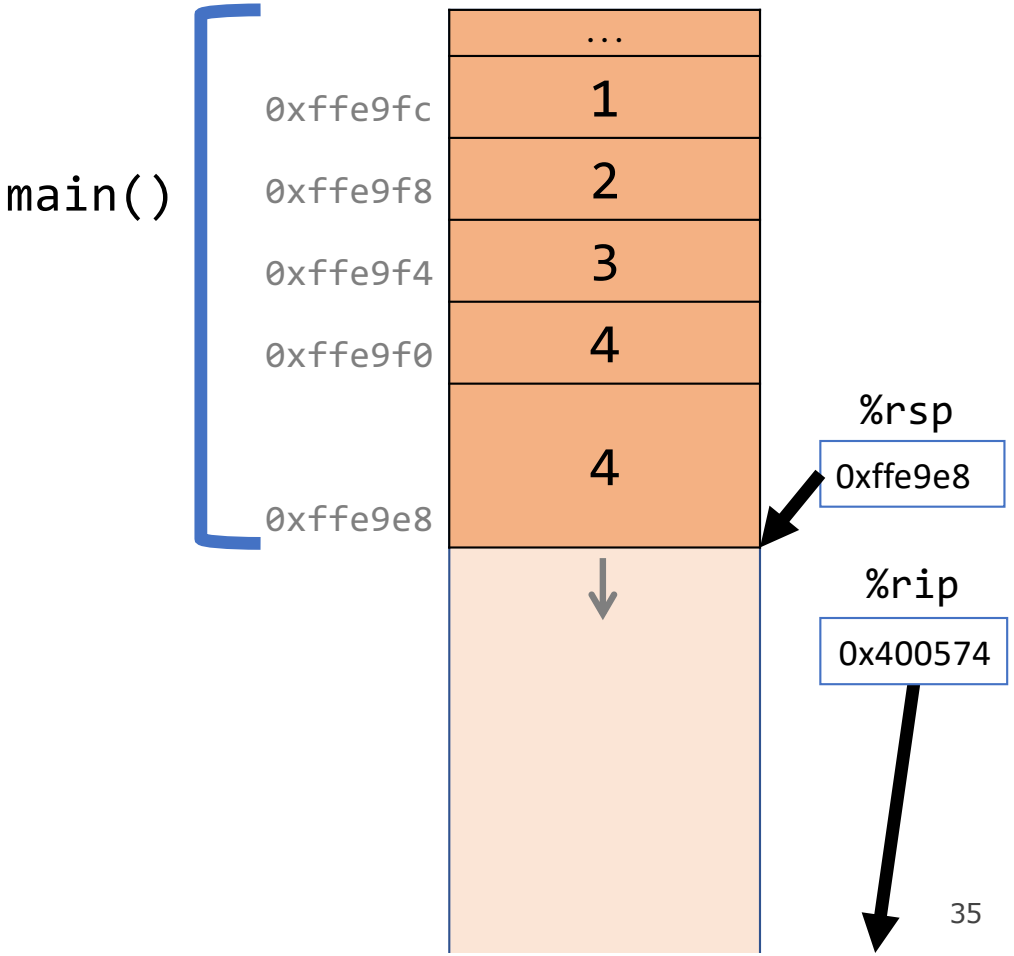


Parameters and Return

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

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

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

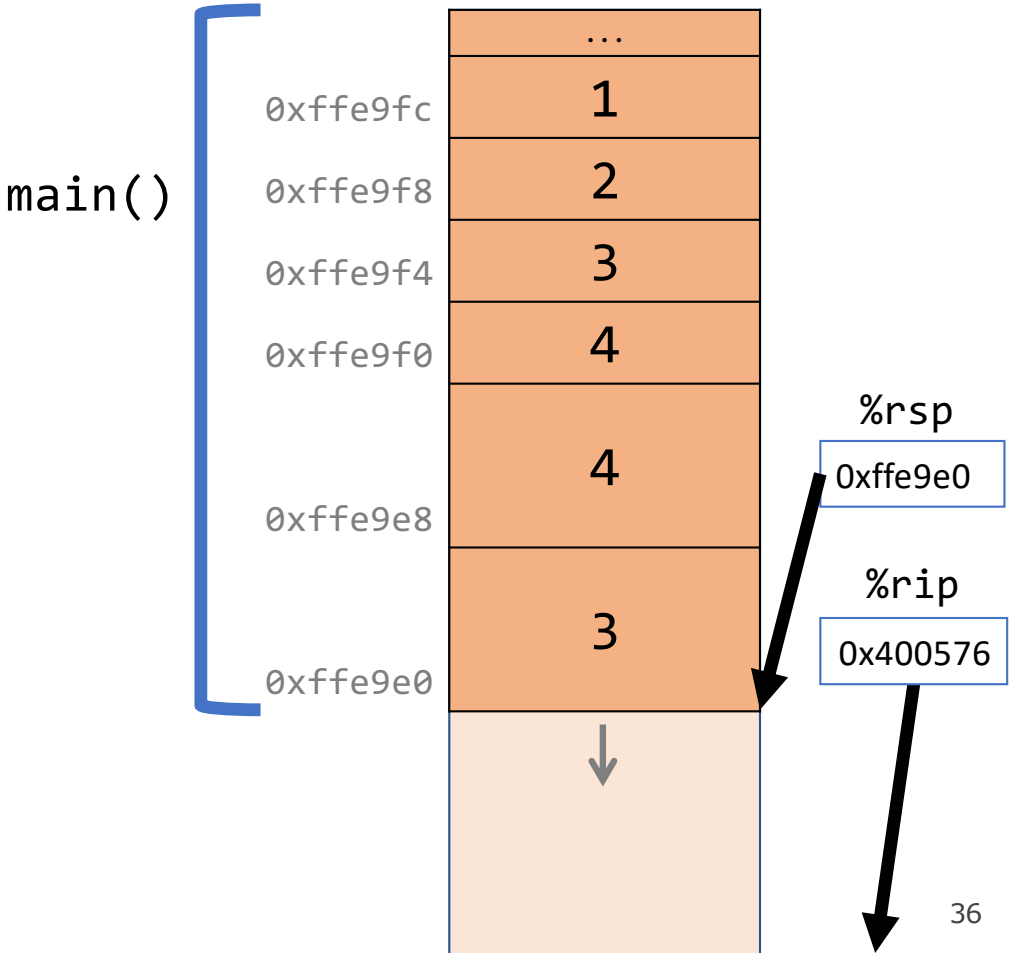


Parameters and Return

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

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

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

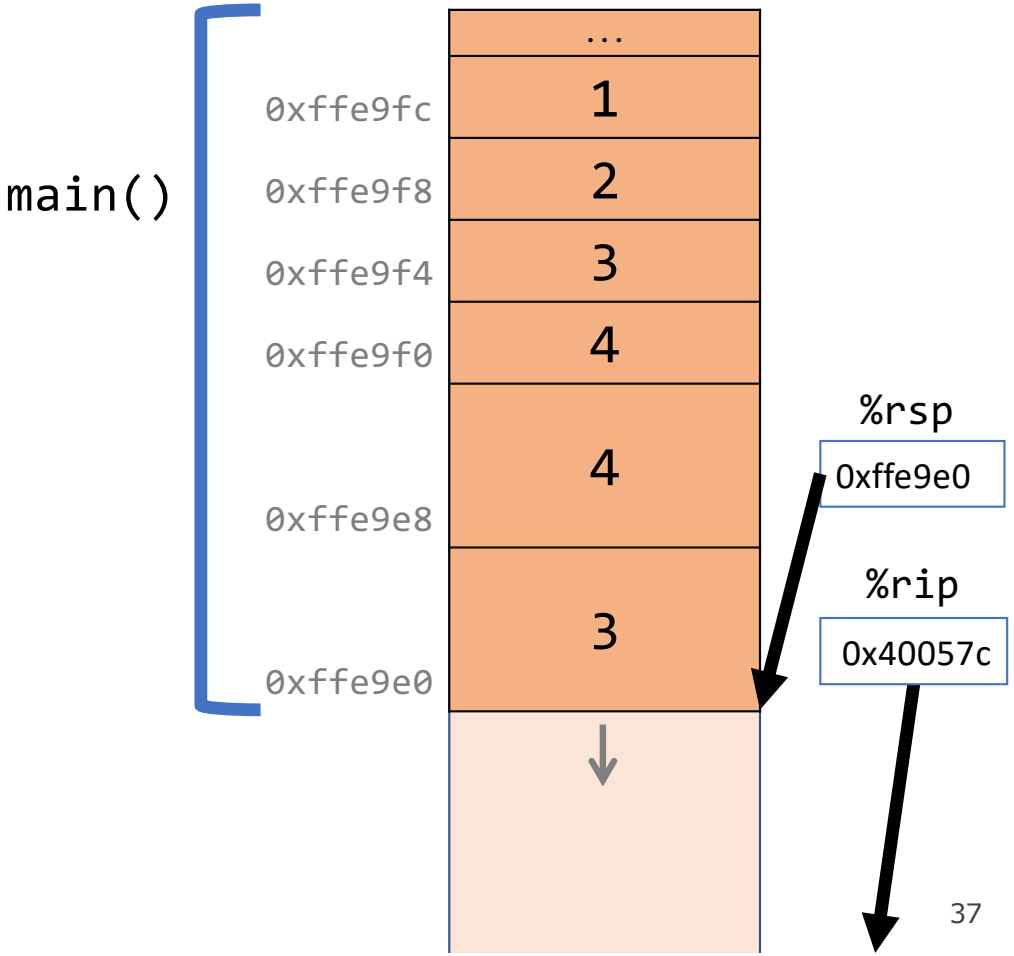


Parameters and Return

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

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

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



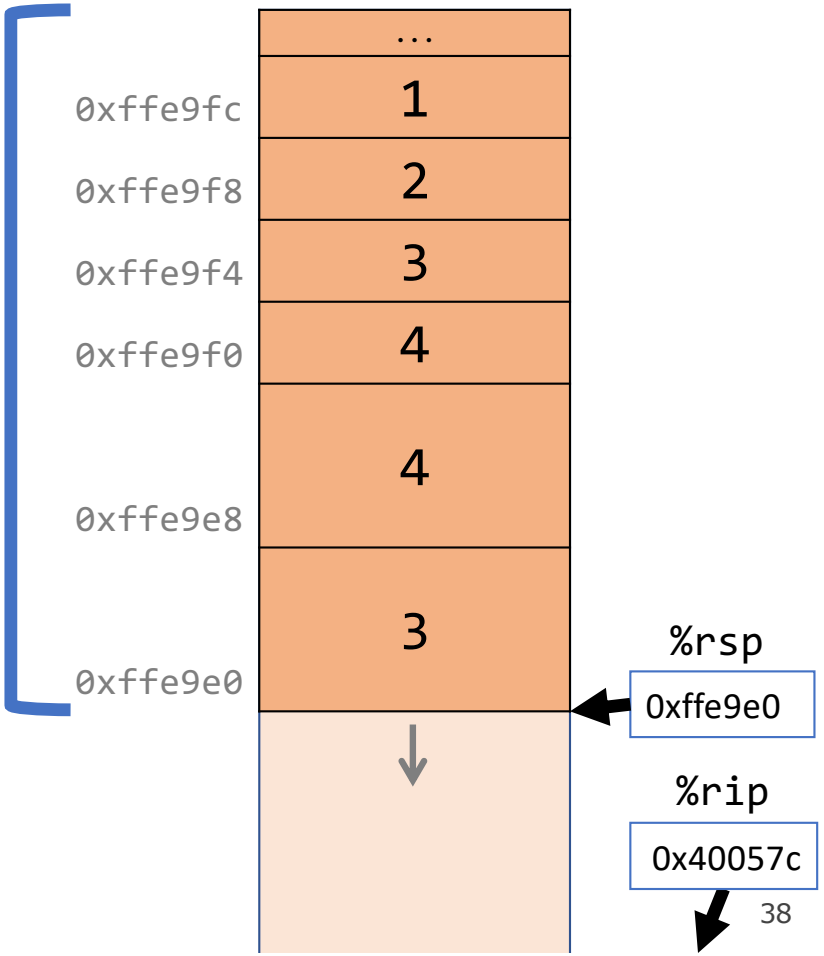
Parameters and Return

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

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

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

main()

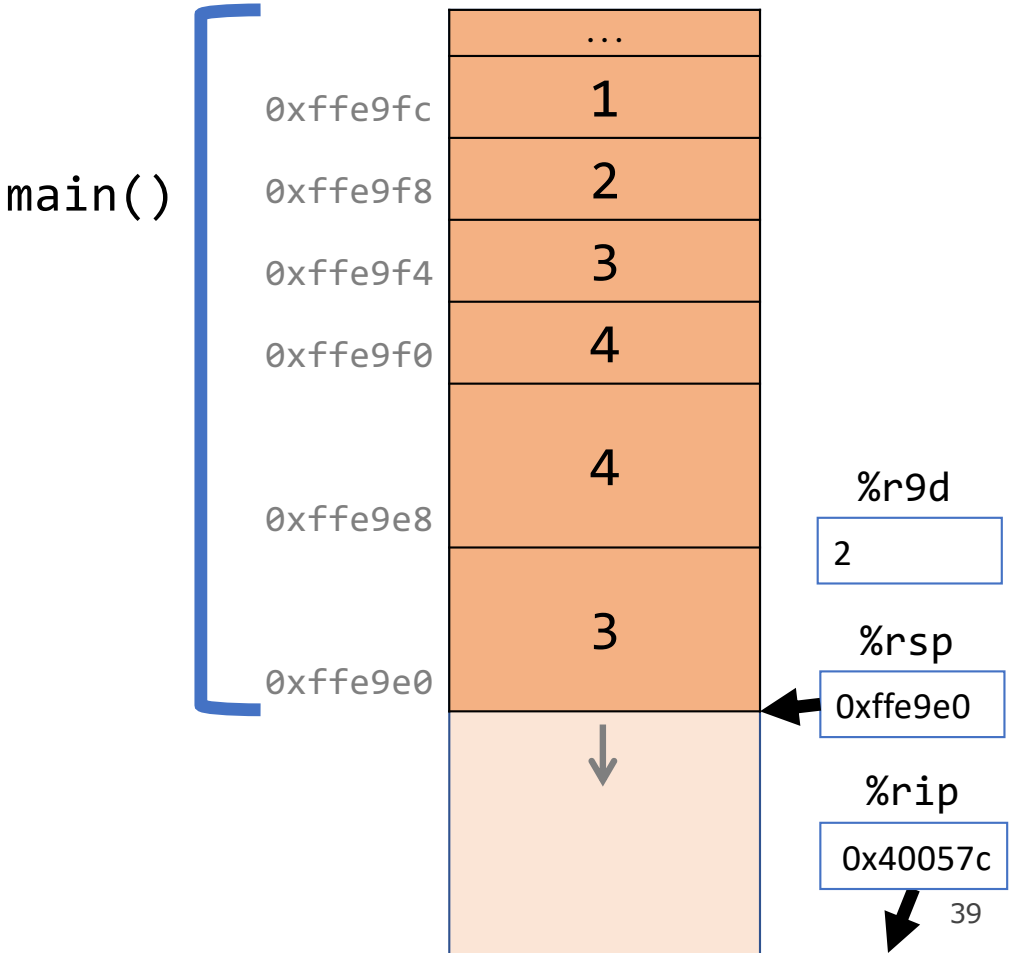


Parameters and Return

```
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>:    lea    0x10(%rsp),%rcx
```

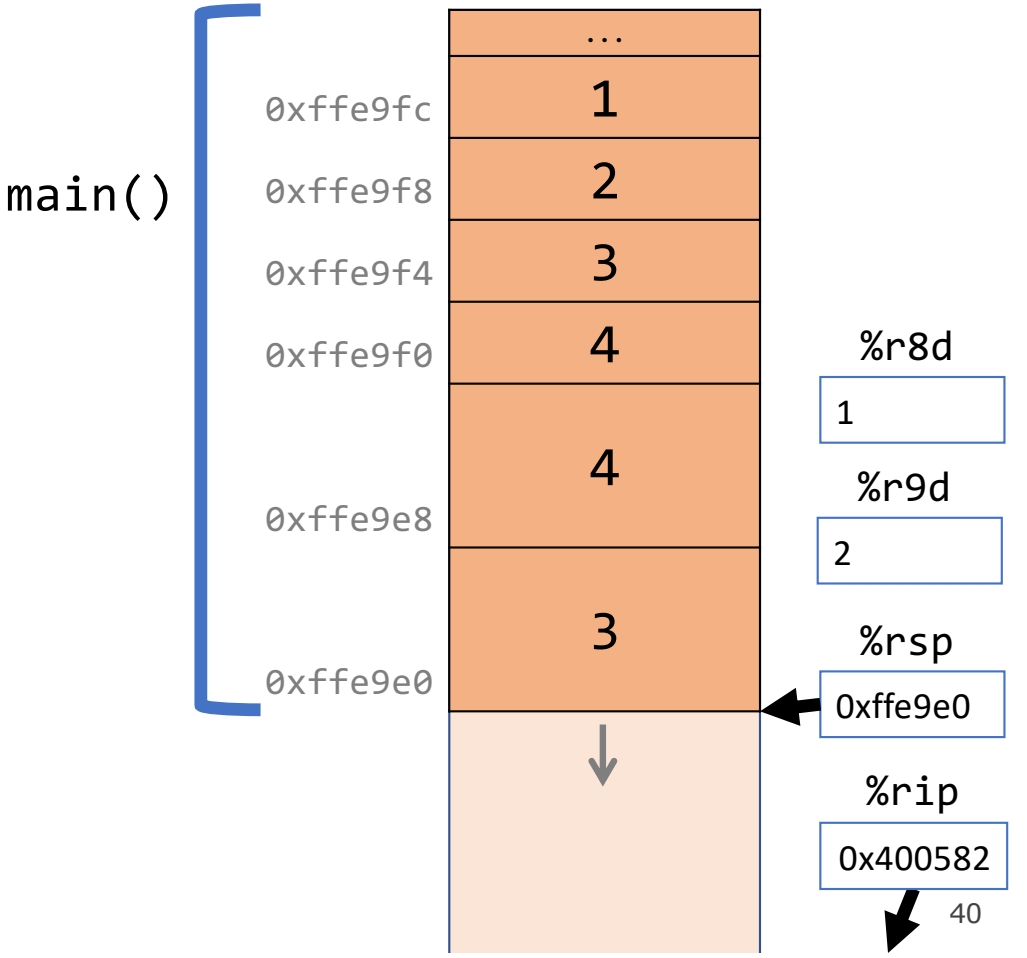


Parameters and Return

```
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) {
    …
}
```

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

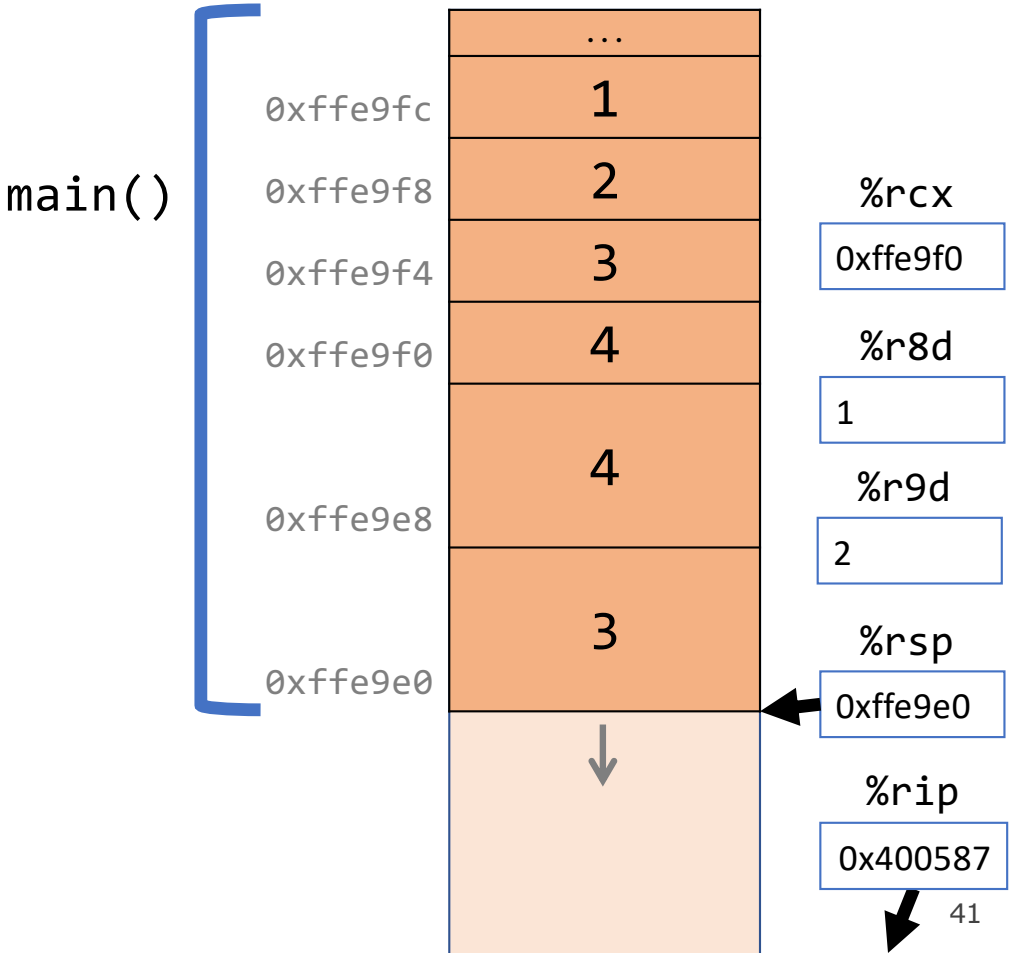


Parameters and Return

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

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

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

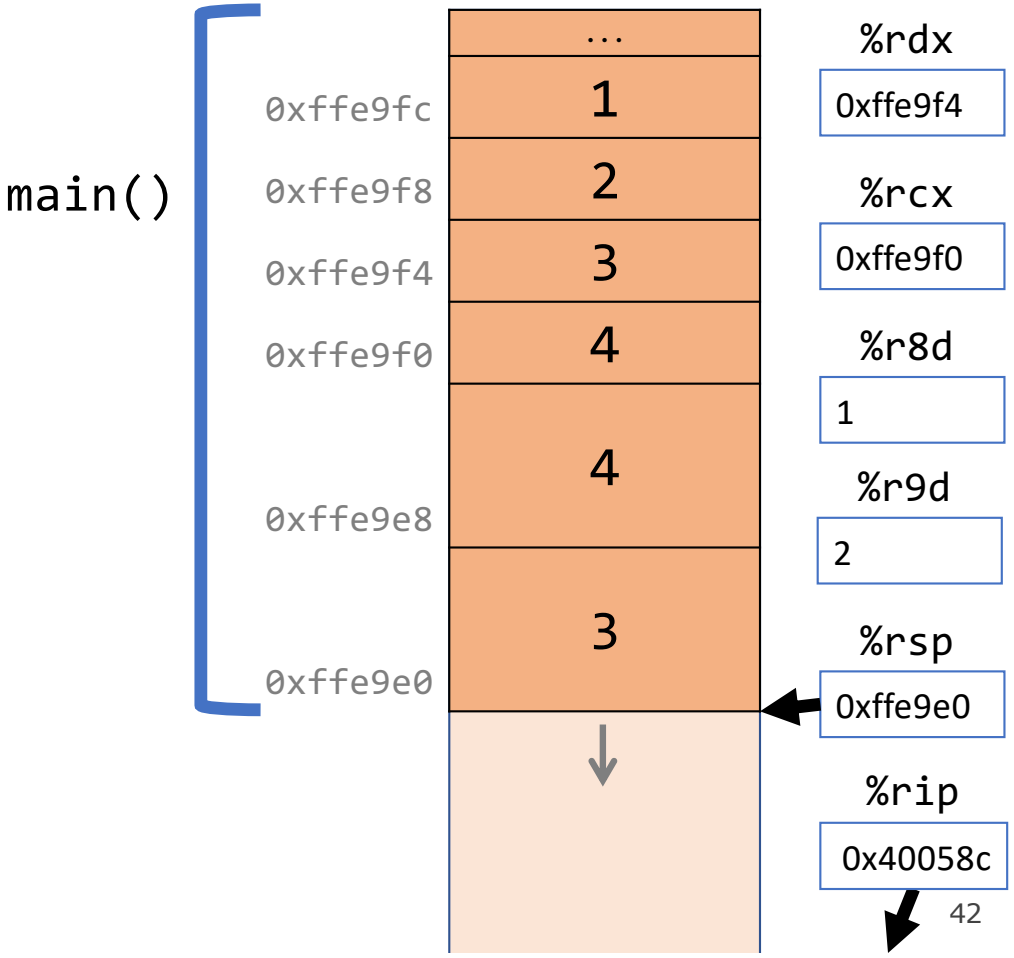


Parameters and Return

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

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

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

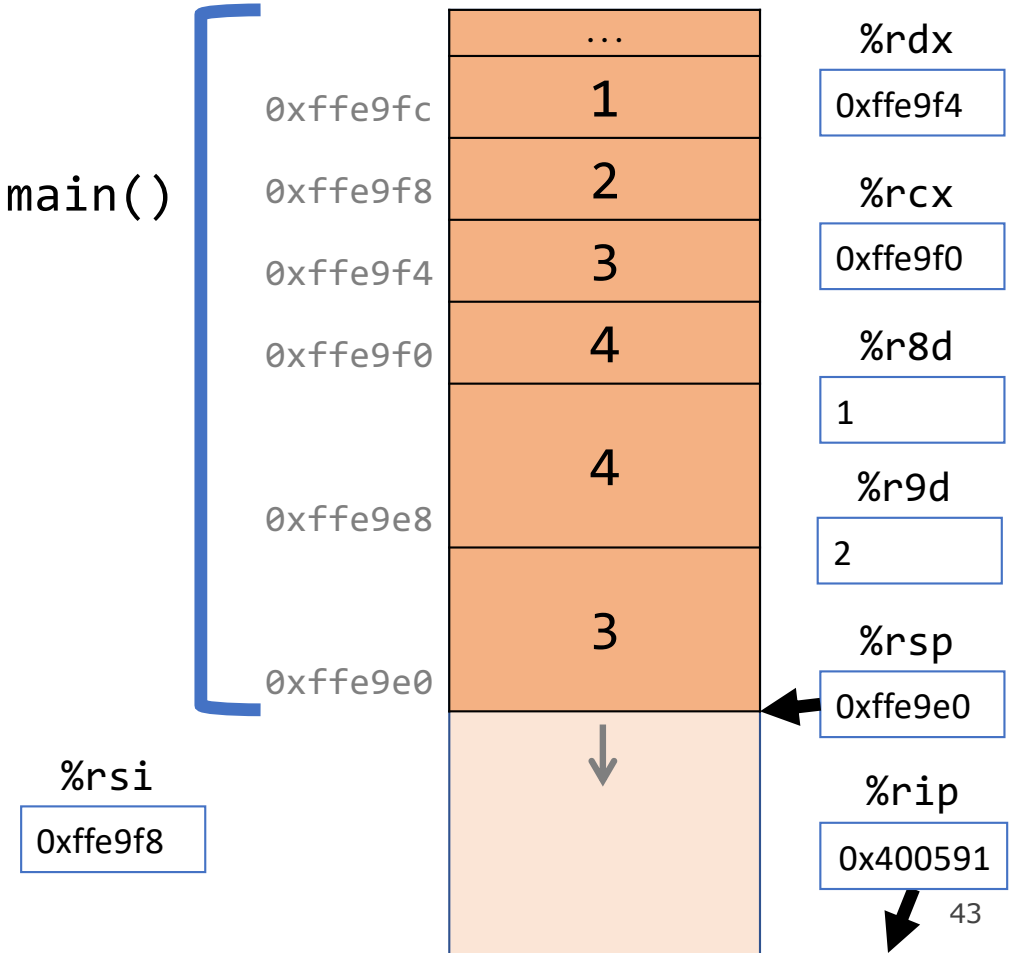


Parameters and Return

```
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>
```

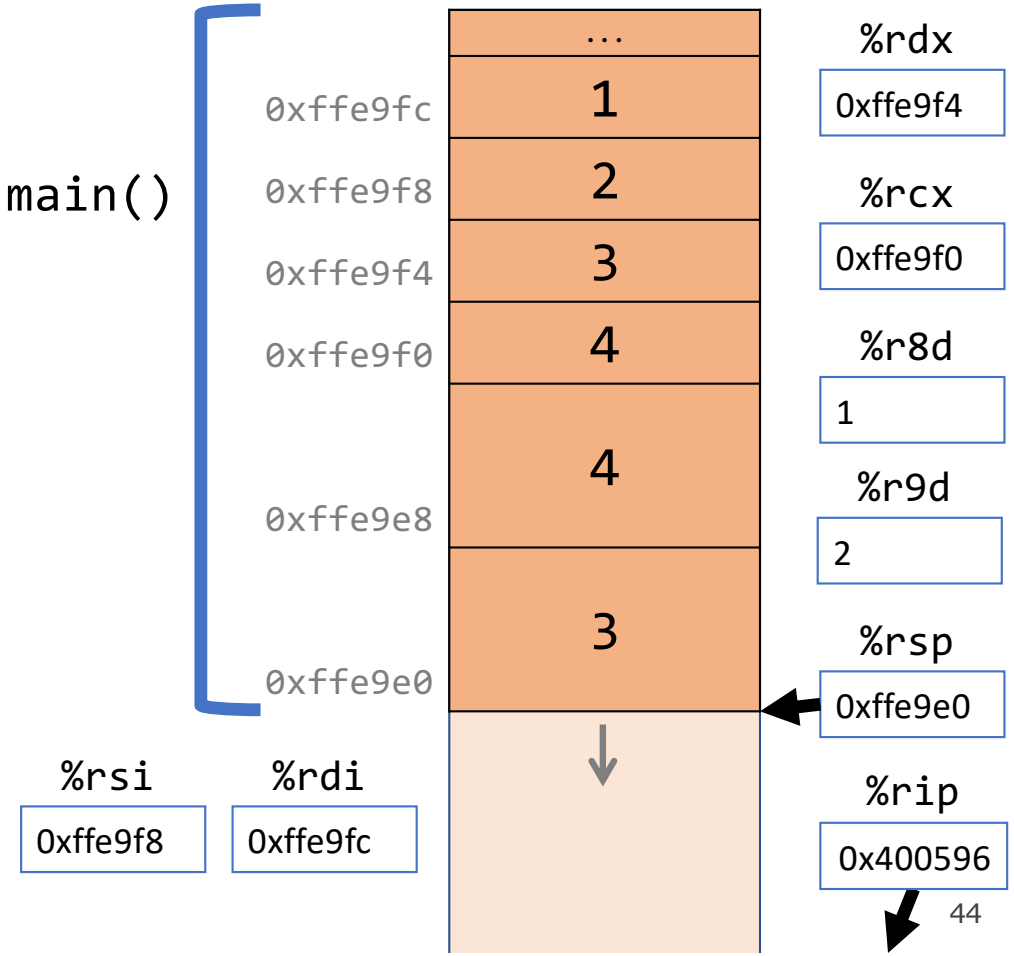


Parameters and Return

```
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
```

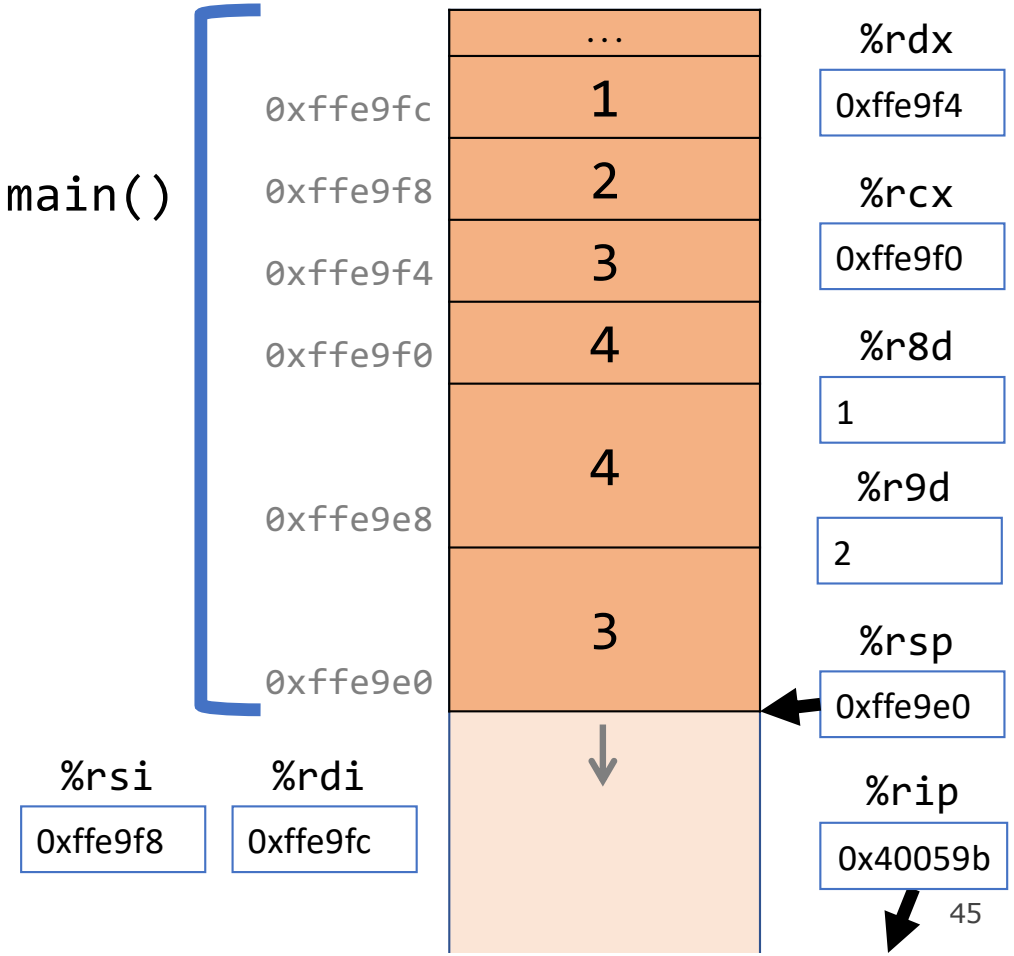


Parameters and Return

```
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
…
```

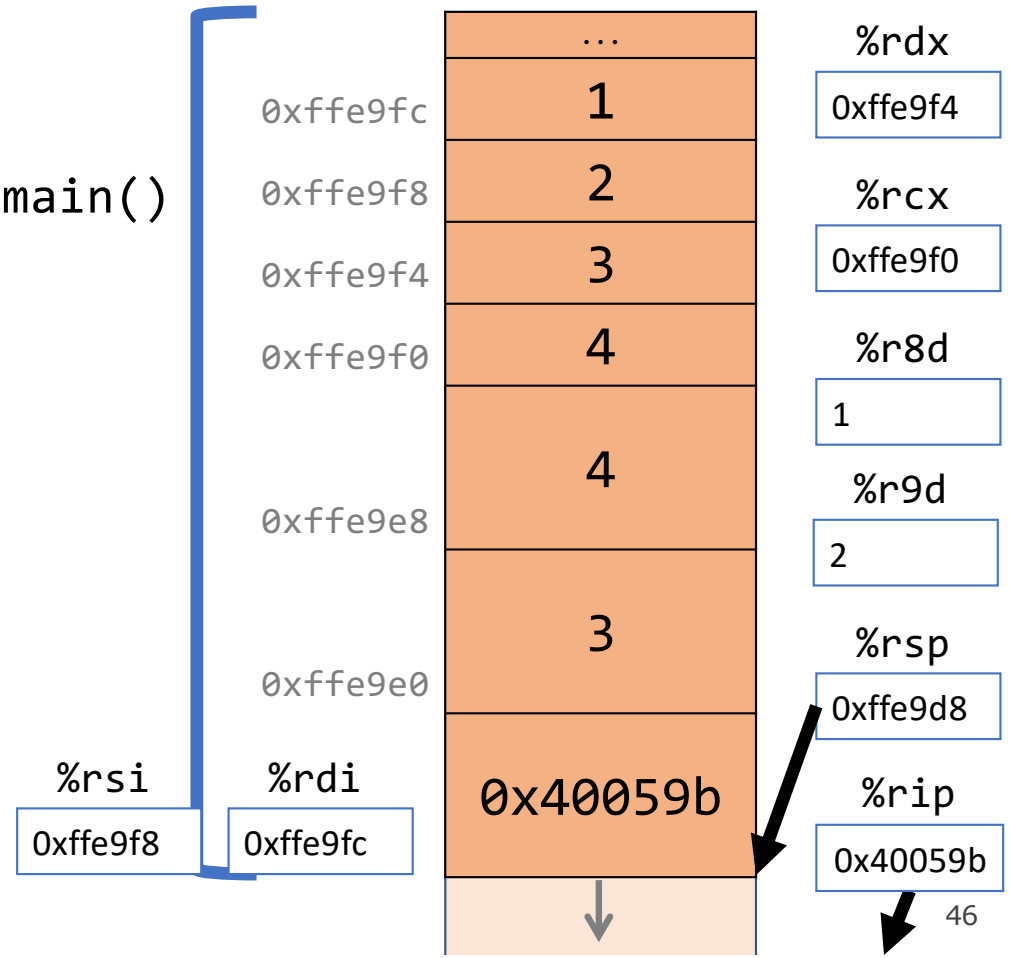


Parameters and Return

```
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
…
```

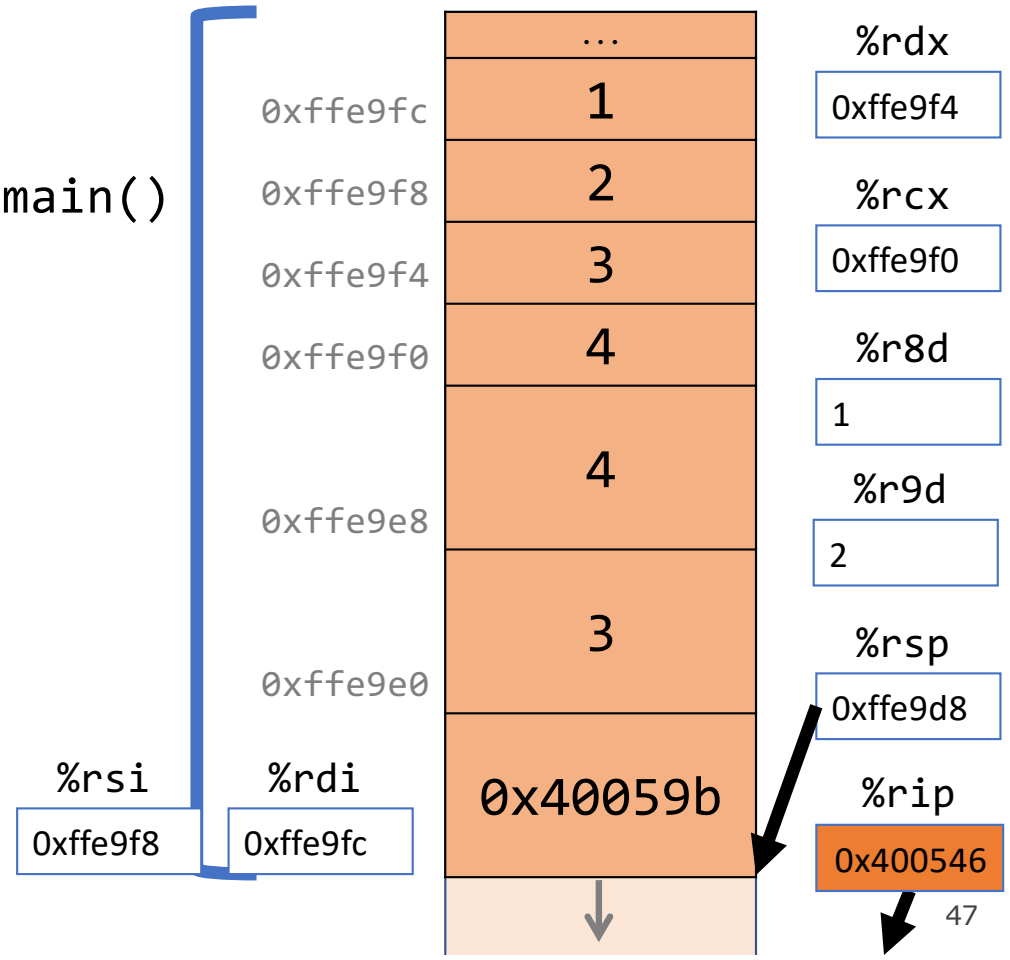


Parameters and Return

```
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
…
```



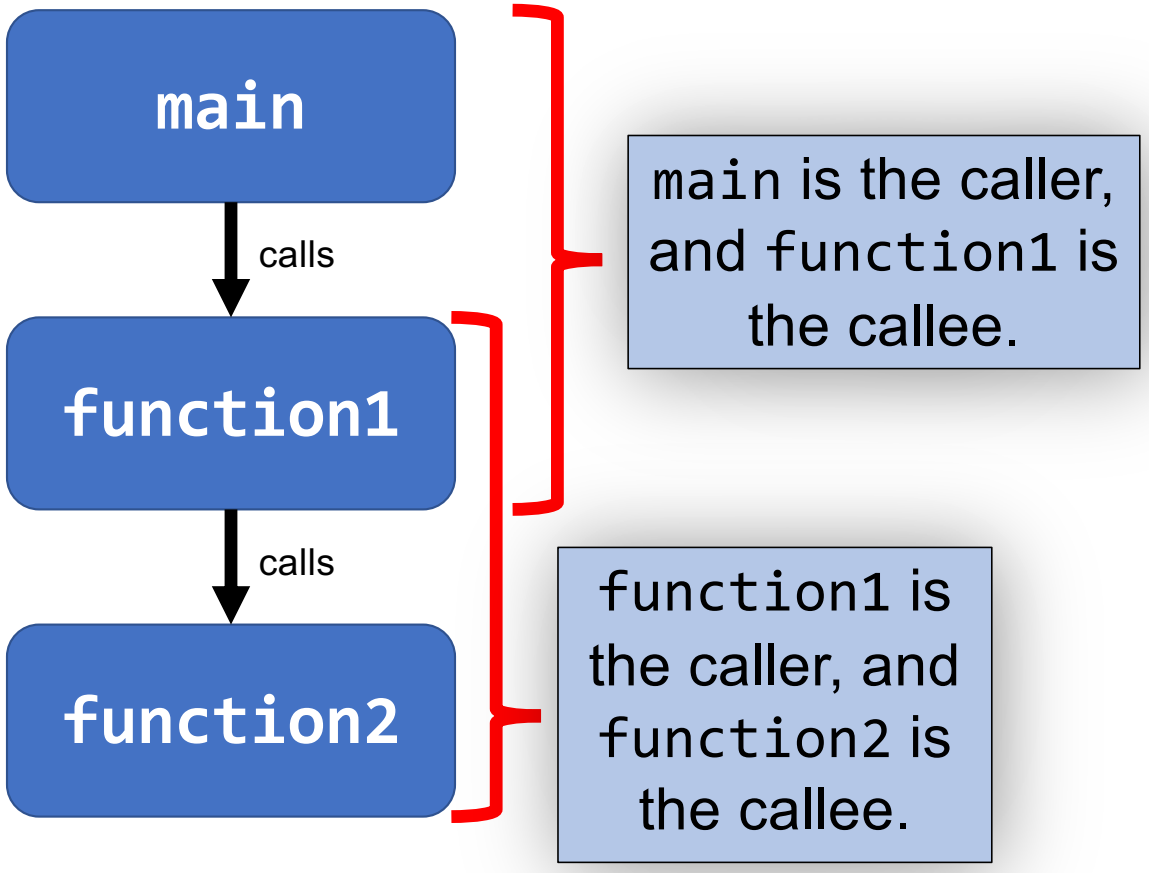
Register Restrictions

There is only one copy of registers for all functions in a running program.

- **Problem:** what if ***funcA*** is building up a value in register %r10, and calls ***funcB*** in the middle, which itself has instructions that modify %r10? ***funcA***'s value will be destroyed!
- **Solution:** lay down 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

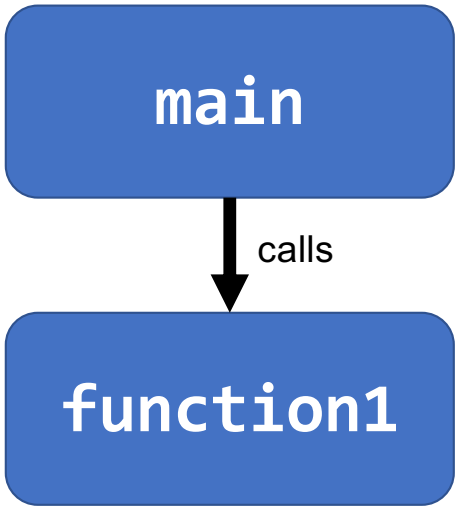
Caller-Owned

- Callee must *save* the existing value and *restore* it when done.
- Caller can store values in them and assume they'll 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.

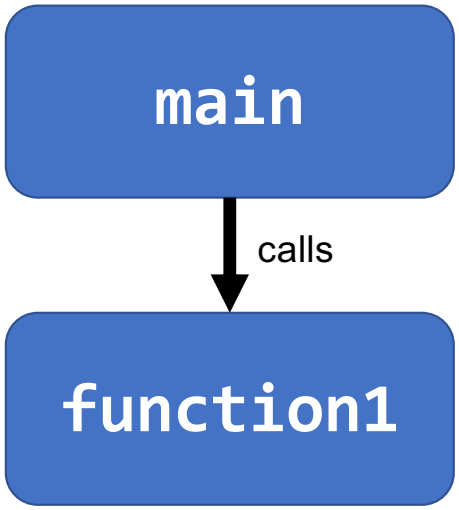
Caller-Owned Registers



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

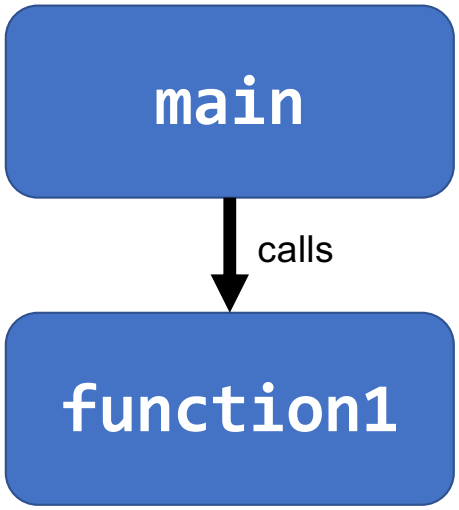
If function1 wants to use any caller-owned registers, it must save the existing values and restore them before returning.

Caller-Owned Registers




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

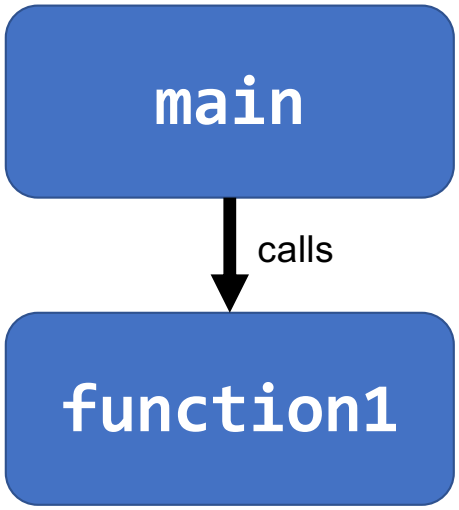
Callee-Owned Registers



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

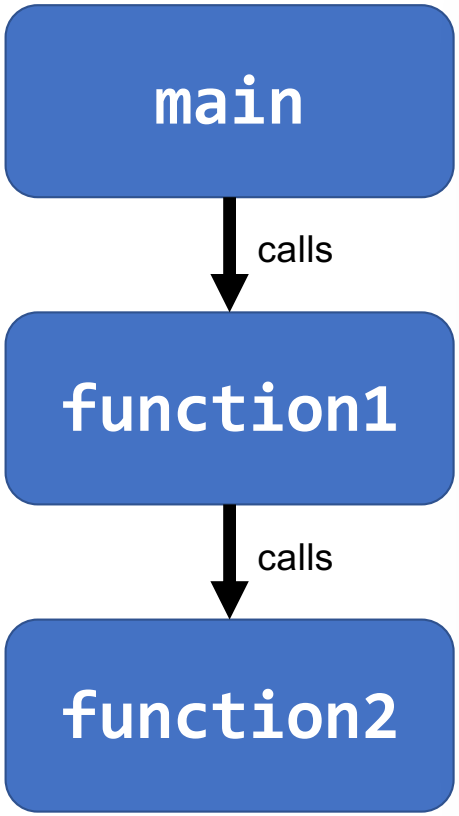
If function1 wants to use any callee-owned registers, it can do so without saving the existing values.

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

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.



`rfact.c and rfact`

Our First Assembly

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

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

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