



CS107, Lecture 19

Assembly: Control Flow Wrap, Function Call Take I

Reading: B&O 3.7

Ed Discussion: <https://edstem.org/us/courses/65949/discussion/5634457>

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 executed
 - If n is large, right (alternative) is best b/c fewer instructions executed
- The compiler may emit a static instruction count that is longer than some alternative, but it may be more efficient if loop executes many times.
- Does the compiler *know* that a loop will execute many times? Of course not.
- What if our code has loops that always execute a small number of times? How do we know when gcc makes a bad decision?
 - (take EE108 and EE180!)

Condition Code-Dependent Instructions

There are three common instruction types that use condition codes:

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

set: Read condition codes

set instructions conditionally set a byte to 0 or 1.

- Reads current state of flags
- Destination is a single-byte register (e.g., %al) or single-byte memory location
- Leaves other bytes of register (e.g., everything else in %rax) alone
- Typically followed by movzbl to zero those other bytes

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

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

set: Read condition codes

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

cmov: Conditional move

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

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

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

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

cmov: Conditional move

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

Calling Functions In Assembly

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

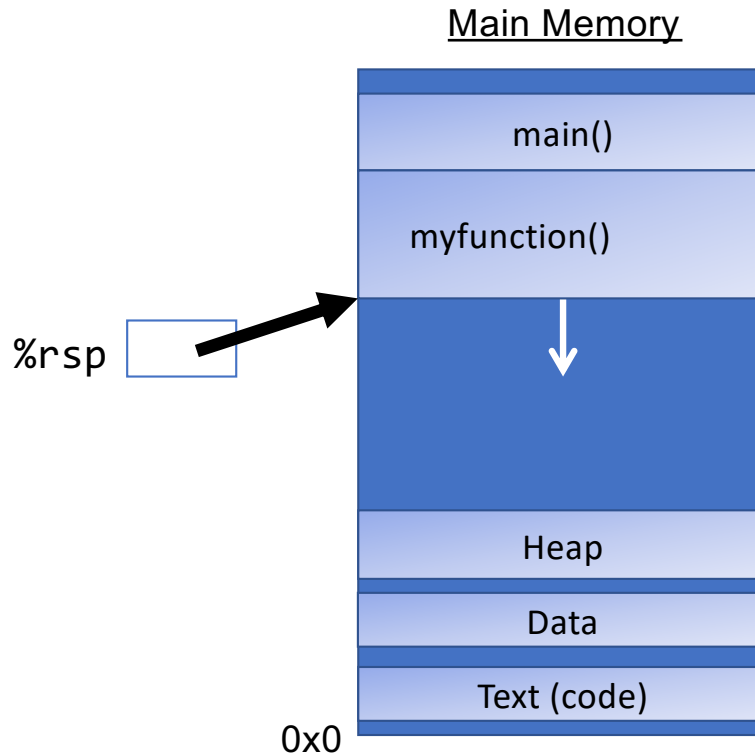
- **Transfer Control** – %rip must be adjusted to execute the callee's instructions, and then resume the caller's instructions afterwards.
- **Pass Data** – we must pass parameters and extract return values.
- **Manage Memory** – we must handle all of the callee's stack space needs.

How does assembly
interact with the stack?

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

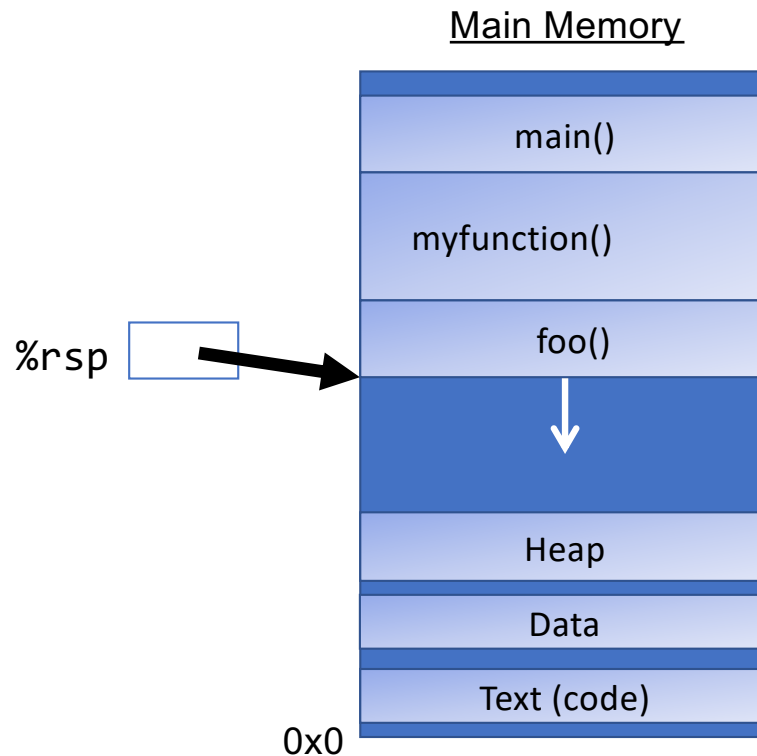
%rsp

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



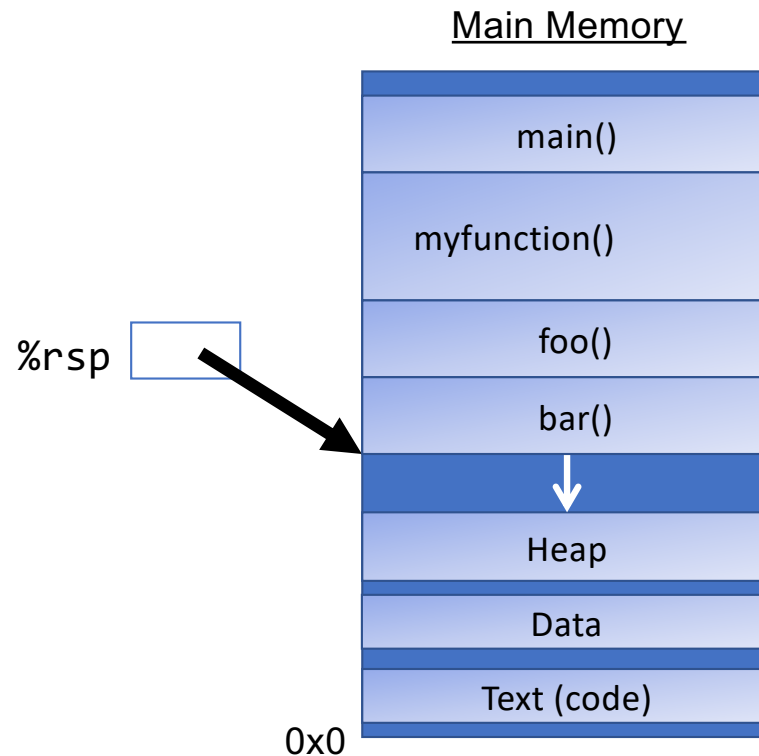
%rsp

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



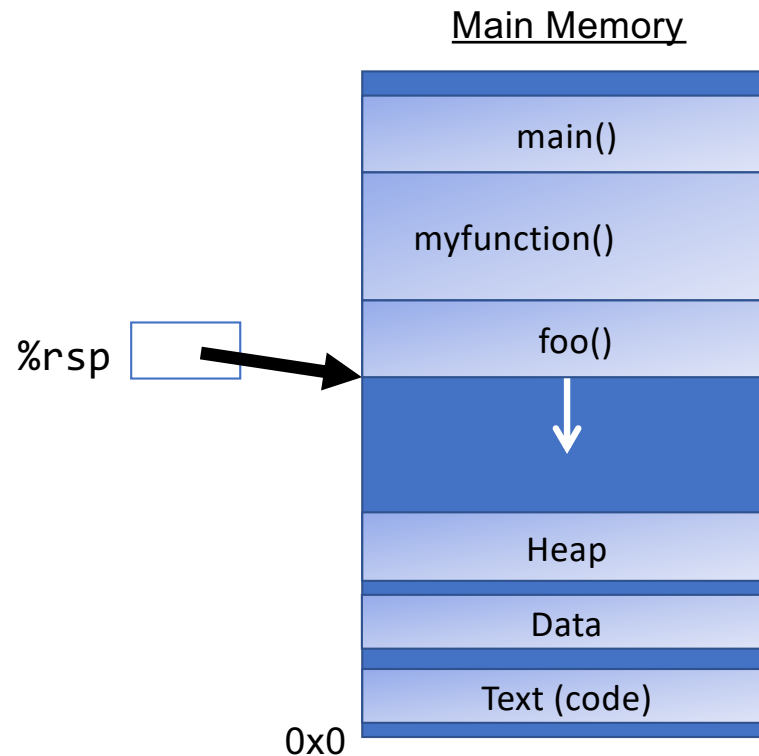
%rsp

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



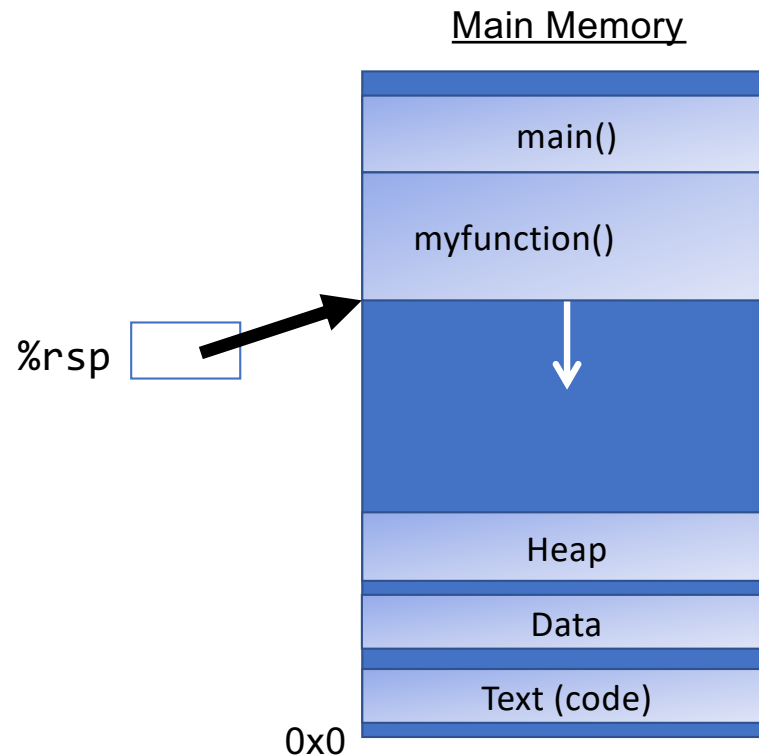
%rsp

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



%rsp

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



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

push

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

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

push

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

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

push

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

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

push

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

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

- This behavior is equivalent to the following, but **pushq** is a shorter instruction:
 subq \$8, %rsp
 movq S, (%rsp)
- Sometimes, you'll see instructions just explicitly decrement the stack pointer to make room for new local variables.

pop

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

Instruction	Effect
popq D	$D \leftarrow M[R[\%rsp]]$ $R[\%rsp] \leftarrow R[\%rsp] + 8;$

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

pop

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

Instruction	Effect
popq D	$D \leftarrow M[R[\%rsp]]$ $R[\%rsp] \leftarrow R[\%rsp] + 8;$

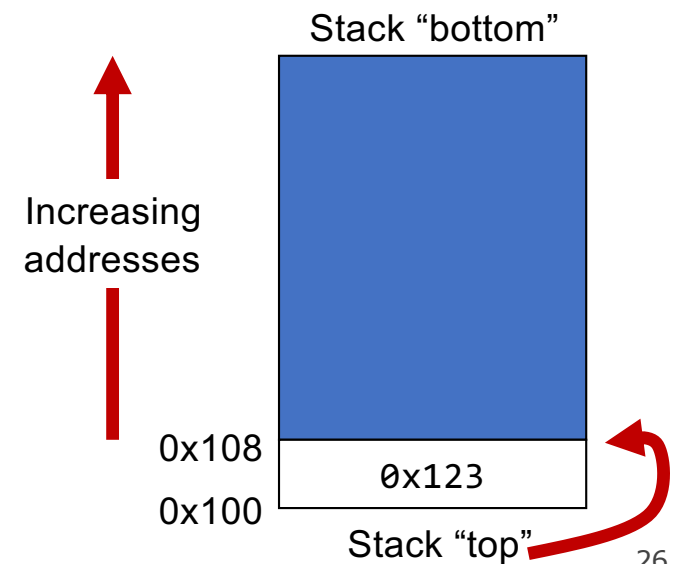
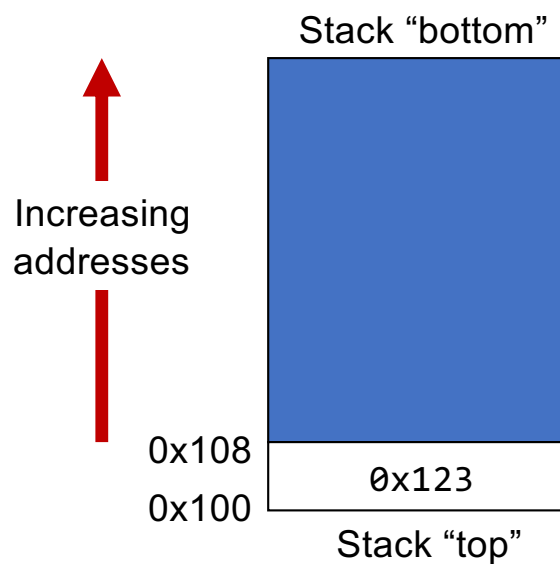
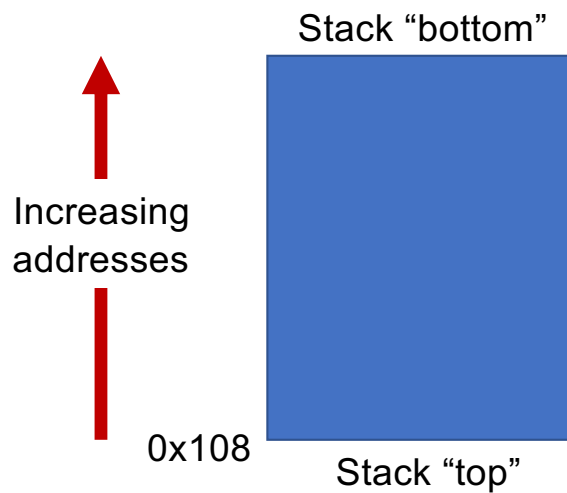
- This behavior is equivalent to the following, but popq is a shorter instruction:
movq (%rsp), D
addq \$8, %rsp
- Sometimes, you'll see instructions just explicitly increment the stack pointer to pop data.

Stack Example

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

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

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



Calling Functions In Assembly

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

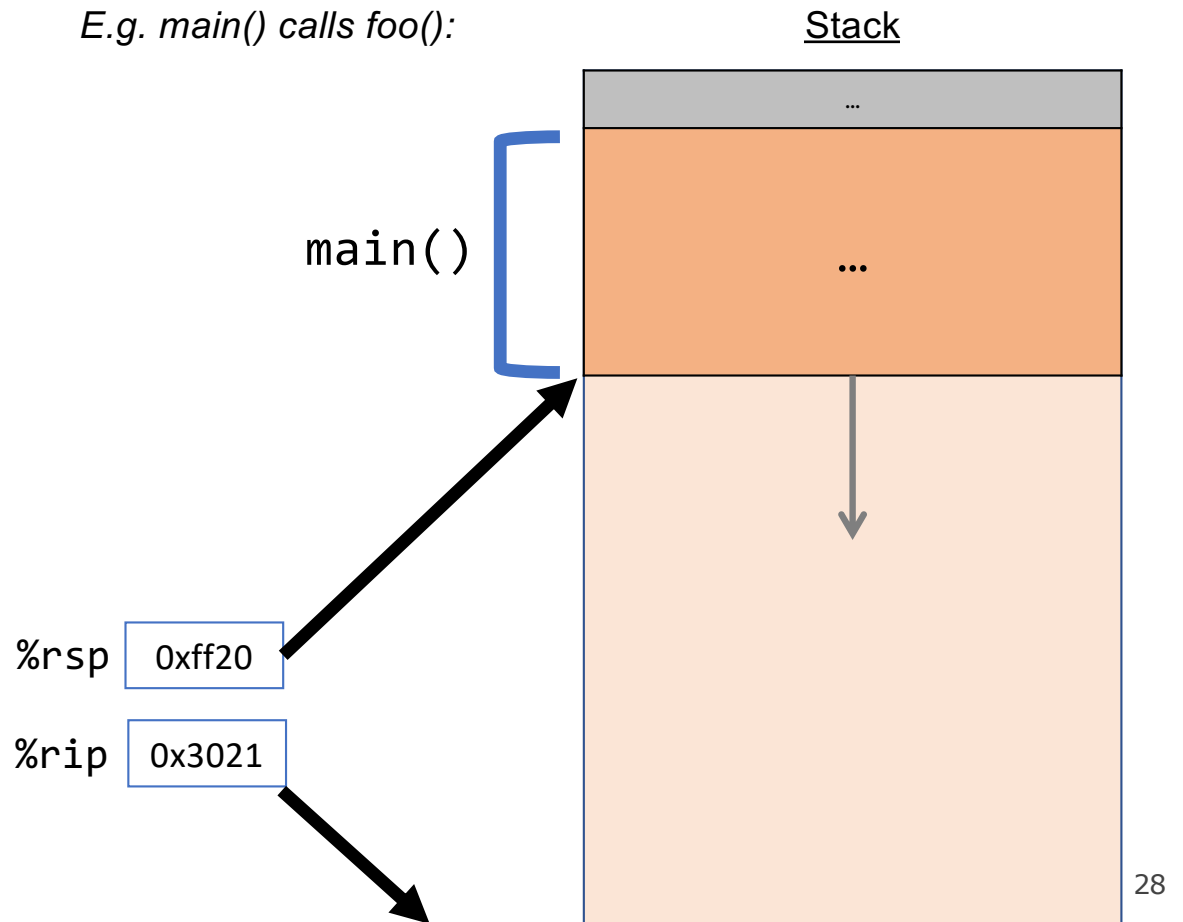
- **Pass Control** – **%rip** must be adjusted to execute the callee's instructions, and then resume the caller's instructions afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

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

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.

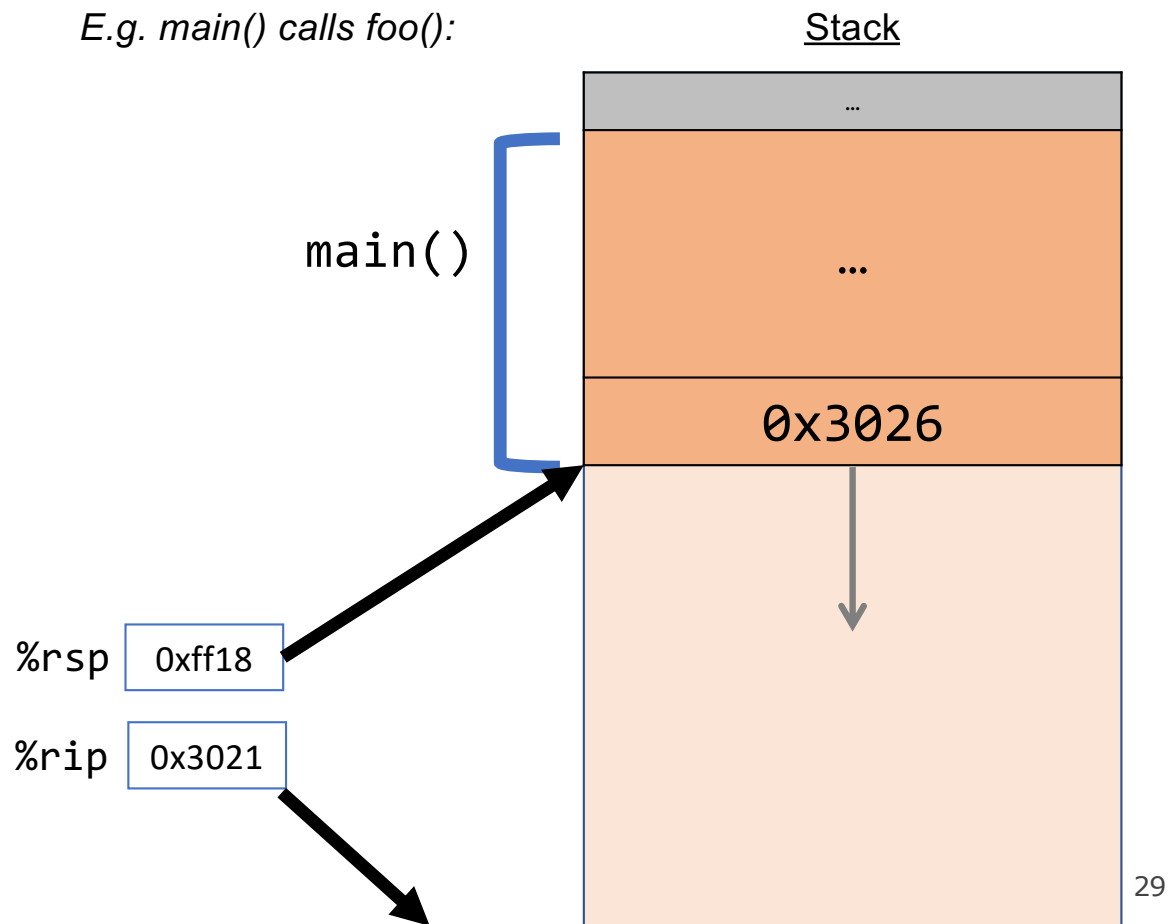
Solution: push the next value of `%rip` onto the stack. Then call the function. When it is finished, put this value back into `%rip` and continue executing.



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.

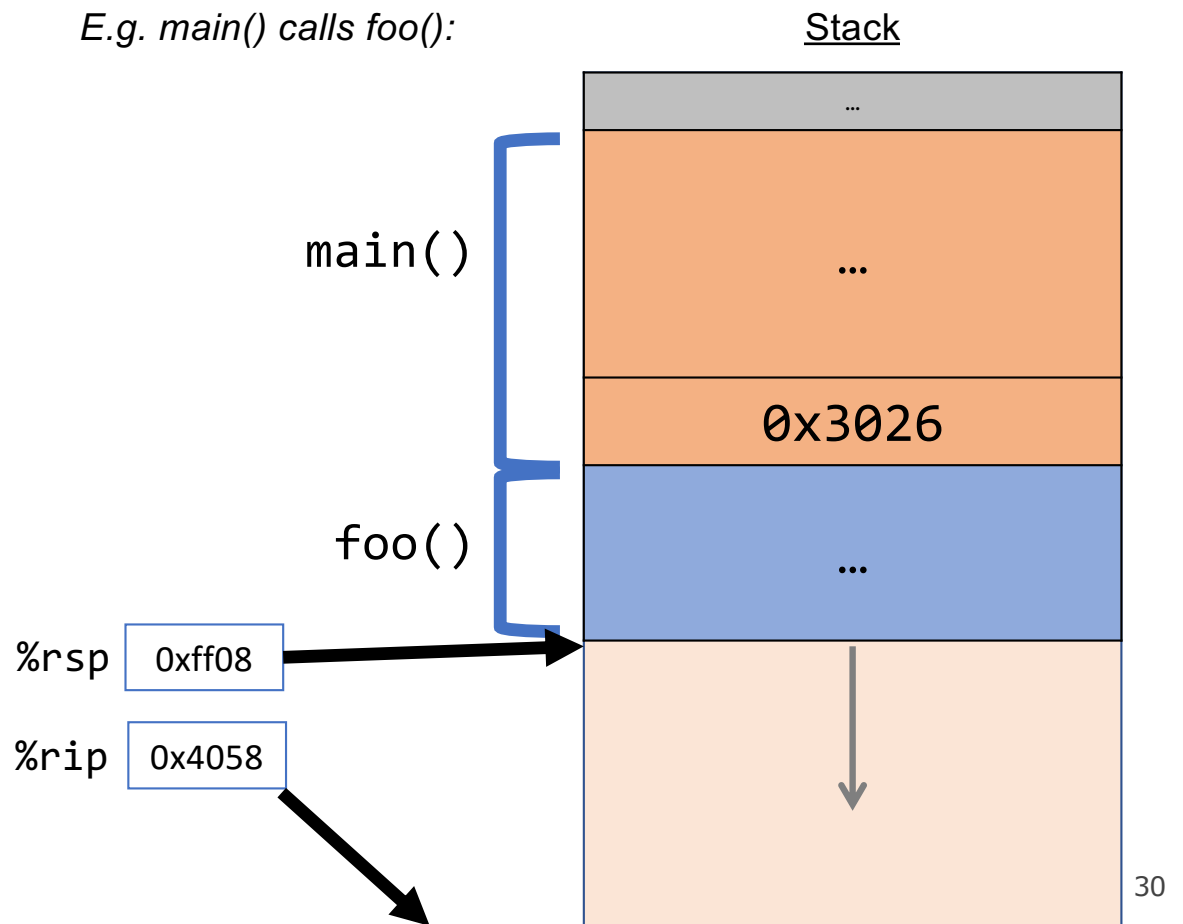
Solution: push the next value of `%rip` onto the stack. Then call the function. When it is finished, put this value back into `%rip` and continue executing.



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.

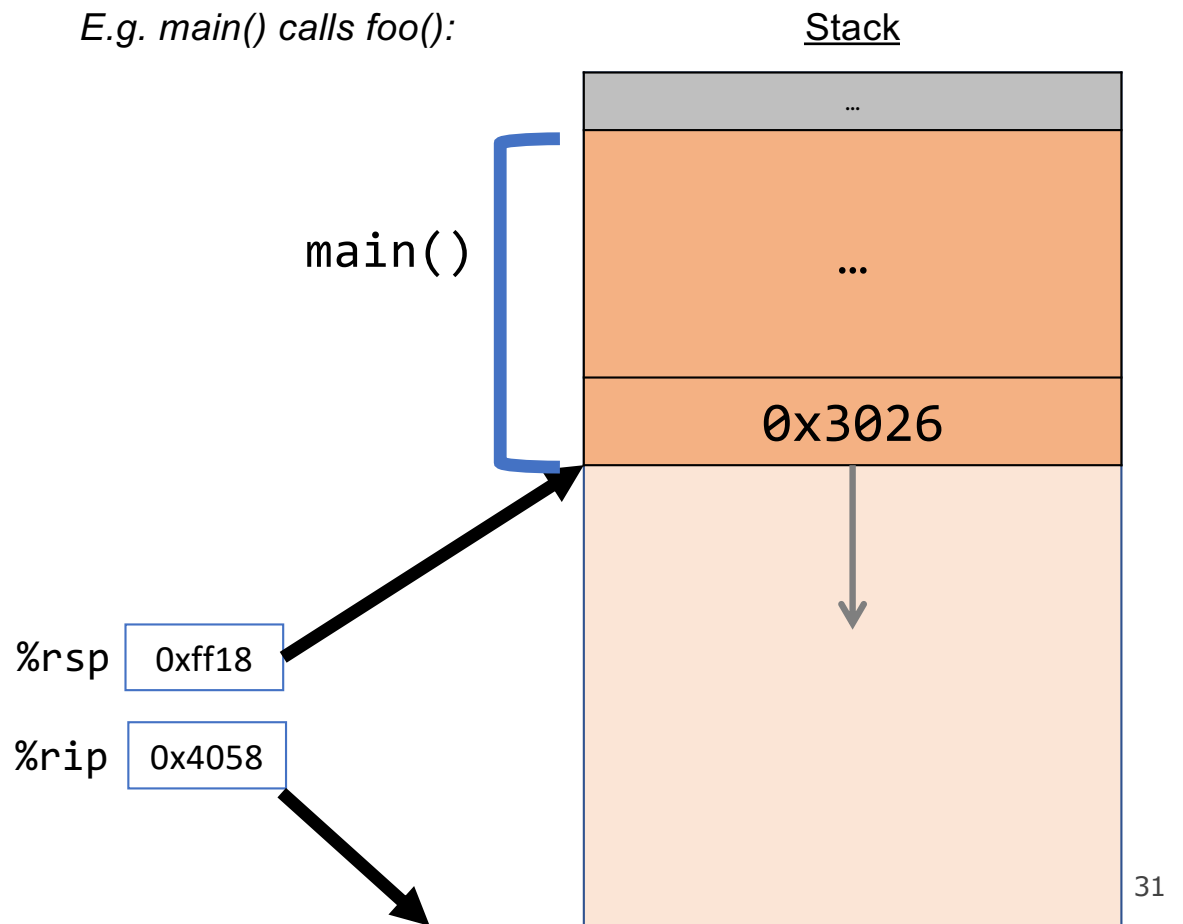
Solution: push the next value of `%rip` onto the stack. Then call the function. When it is finished, put this value back into `%rip` and continue executing.



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.

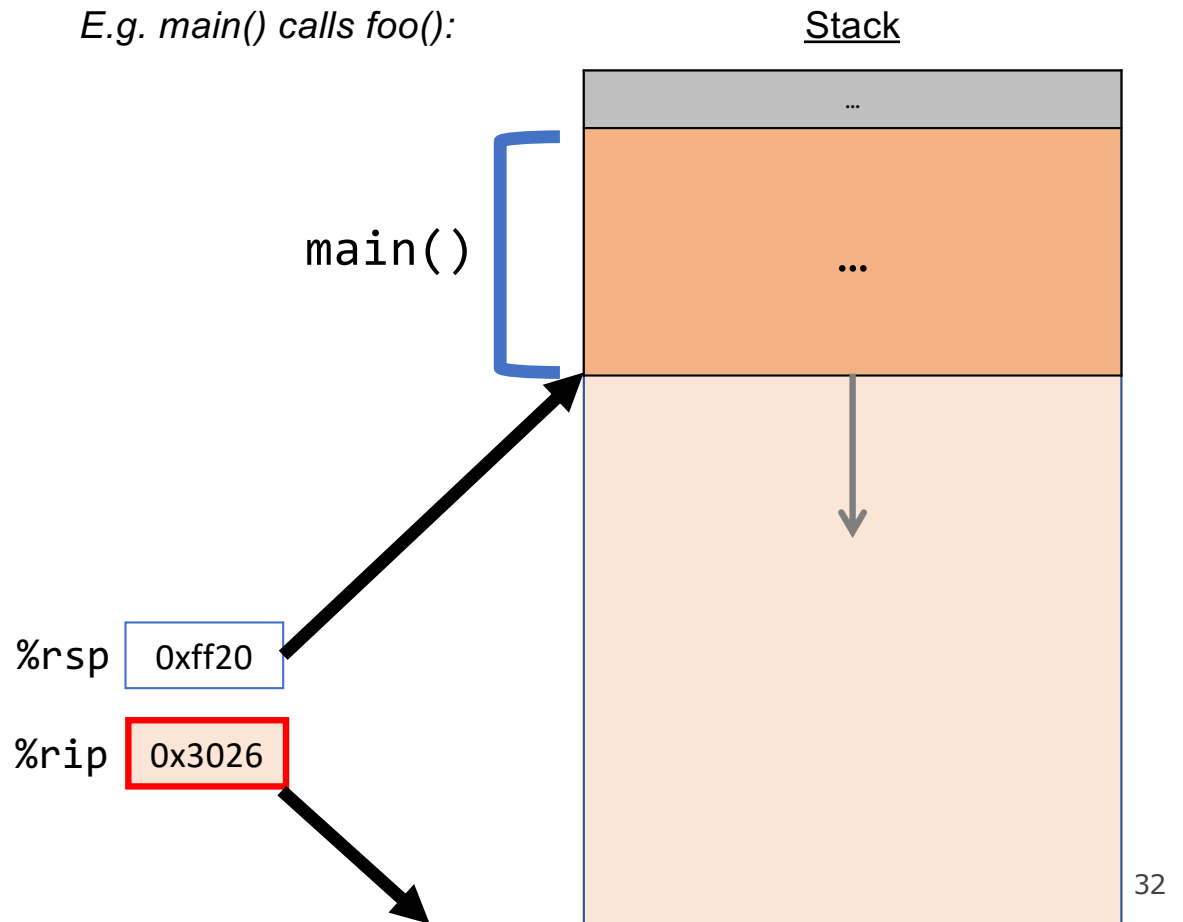
Solution: push the next value of `%rip` onto the stack. Then call the function. When it is finished, put this value back into `%rip` and continue executing.



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.



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 **return value**, which is the value returned by the function via **%rax** or a subset of it).

Registers

What does **call** do?

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

Registers

What does **ret** do?

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

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

Function Pointers

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

call Label

call *Operand

- Why would we use **call** with a register instead of hardcoding the function name in the assembly? *When would we not know the function to call until we run the code?*
- Function pointers! e.g., qsort – qsort calls a function 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.
 - Registers aren't protected against function call, so any variables or important partial results stored in register must be flushed out to the stack.
 - The & operator is applied to a variable, so we need an address for it
 - The variables themselves are arrays or structs and we should anticipate the need for pointer arithmetic.

Local Storage

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

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

Parameters and Return

```
int main(int argc, char *argv[]) {  
    int i1 = 1;  
    int i2 = 2;  
    int i3 = 3;  
    int i4 = 4;  
    int result = func(&i1, &i2, &i3, &i4,  
                     i1, i2, i3, i4);  
    ...  
}  
  
int func(int *p1, int *p2, int *p3, int *p4,  
         int v1, int v2, int v3, int v4) {  
    ...  
}
```

main() 



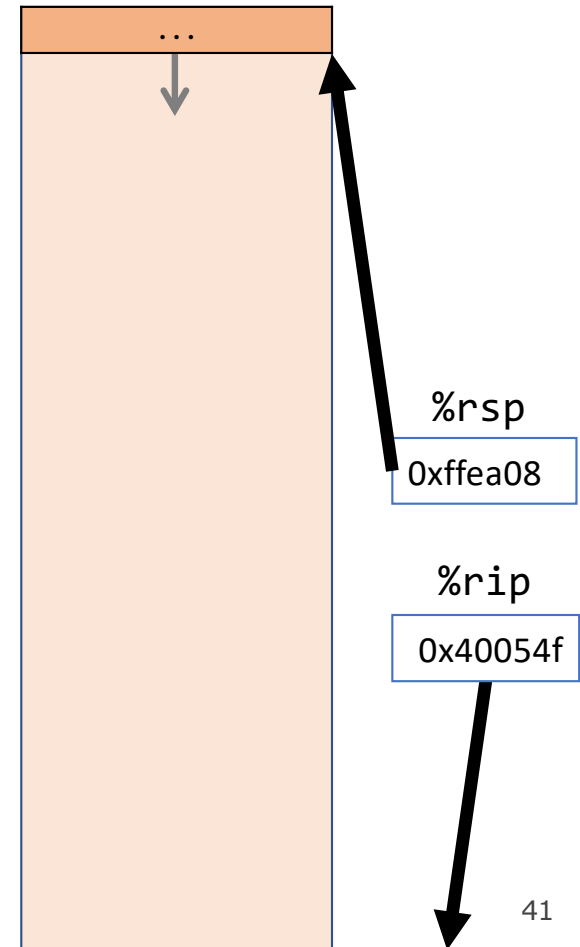
Parameters and Return

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

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

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

main() 



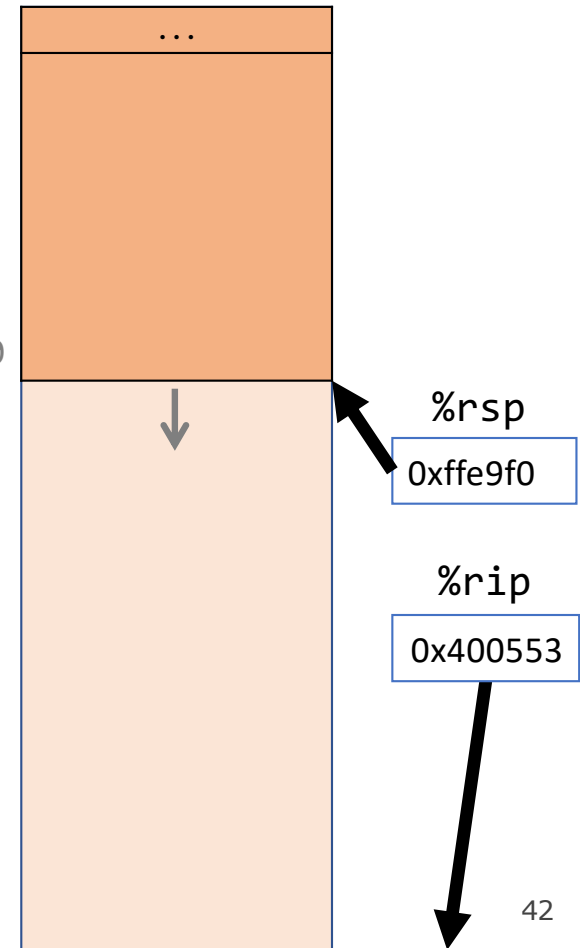
Parameters and Return

```
int main(int argc, char *argv[]) {  
    int i1 = 1;  
    int i2 = 2;  
    int i3 = 3;  
    int i4 = 4;  
    int result = func(&i1, &i2, &i3, &i4,  
                     i1, i2, i3, i4);  
    ...  
}  
  
int func(int *p1, int *p2, int *p3, int *p4,  
         int v1, int v2, int v3, int v4) {  
    ...  
}
```

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

main()

0xffe9f0

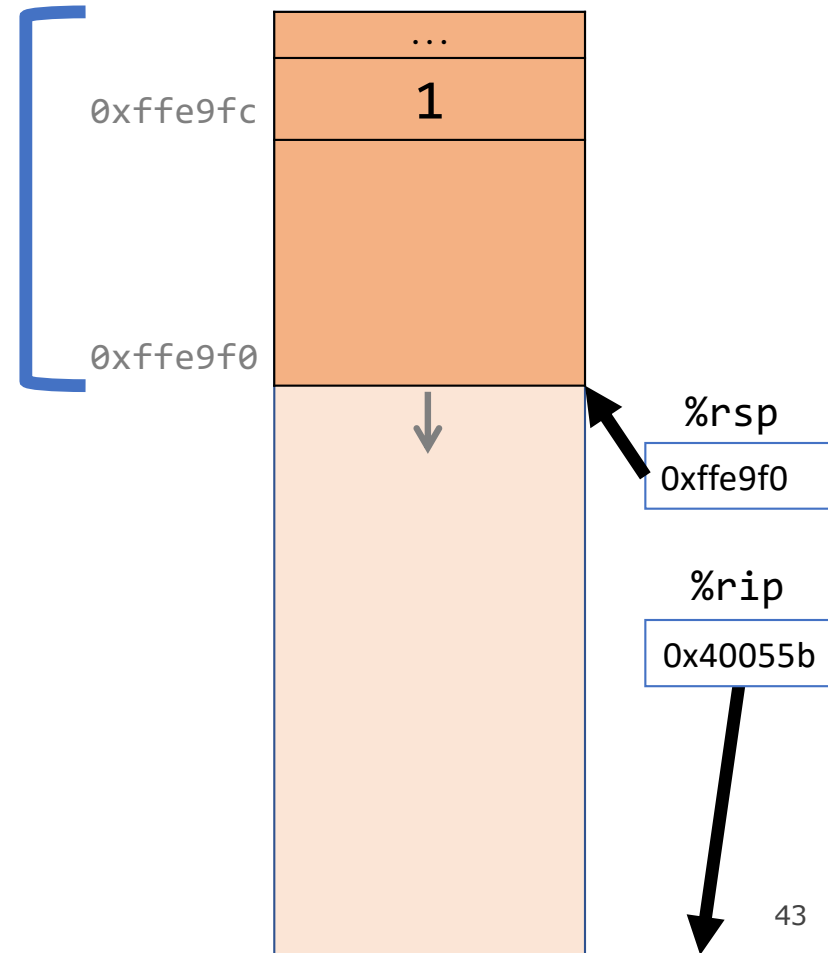


Parameters and Return

```
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()

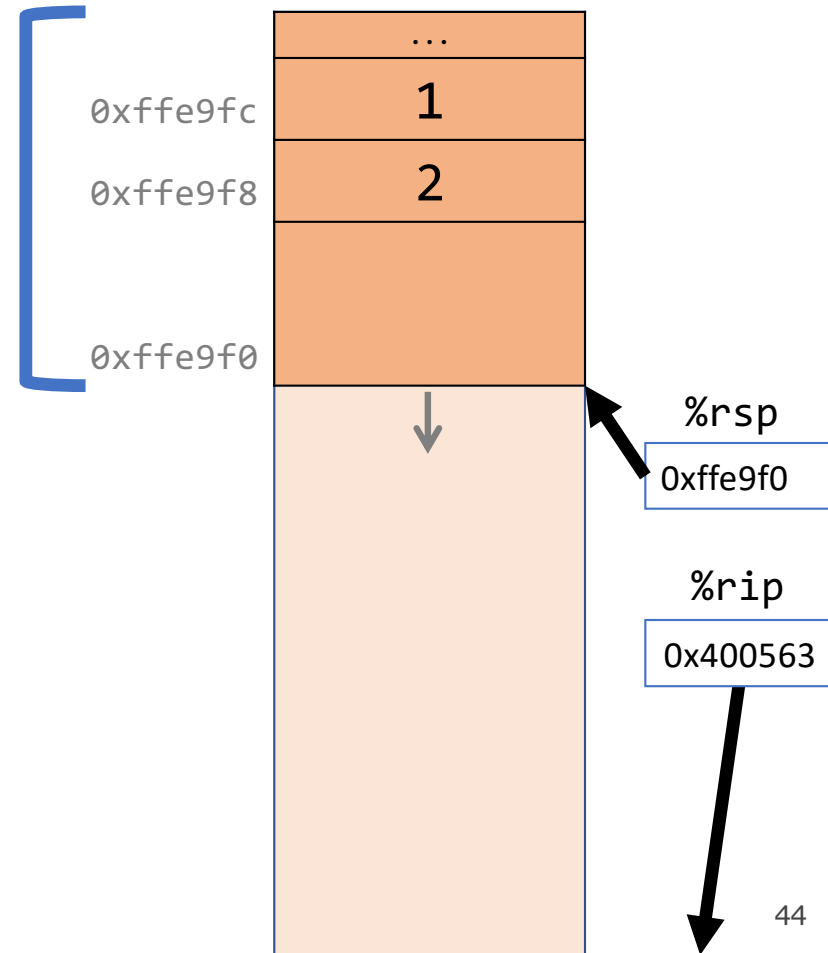


Parameters and Return

```
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()

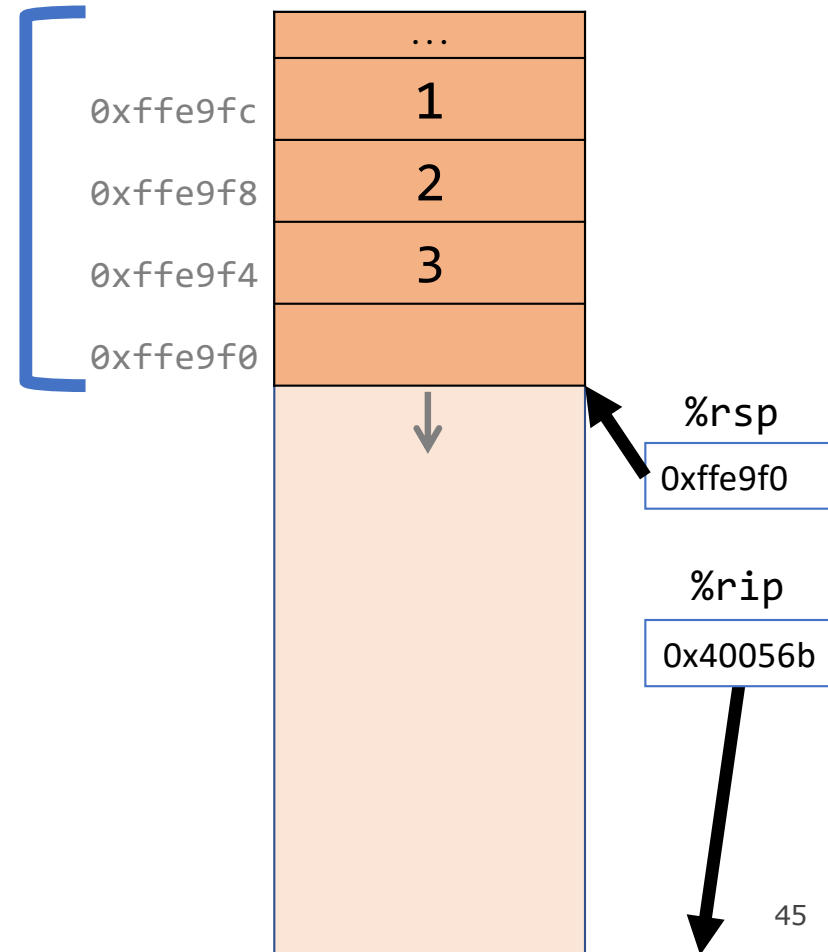


Parameters and Return

```
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()

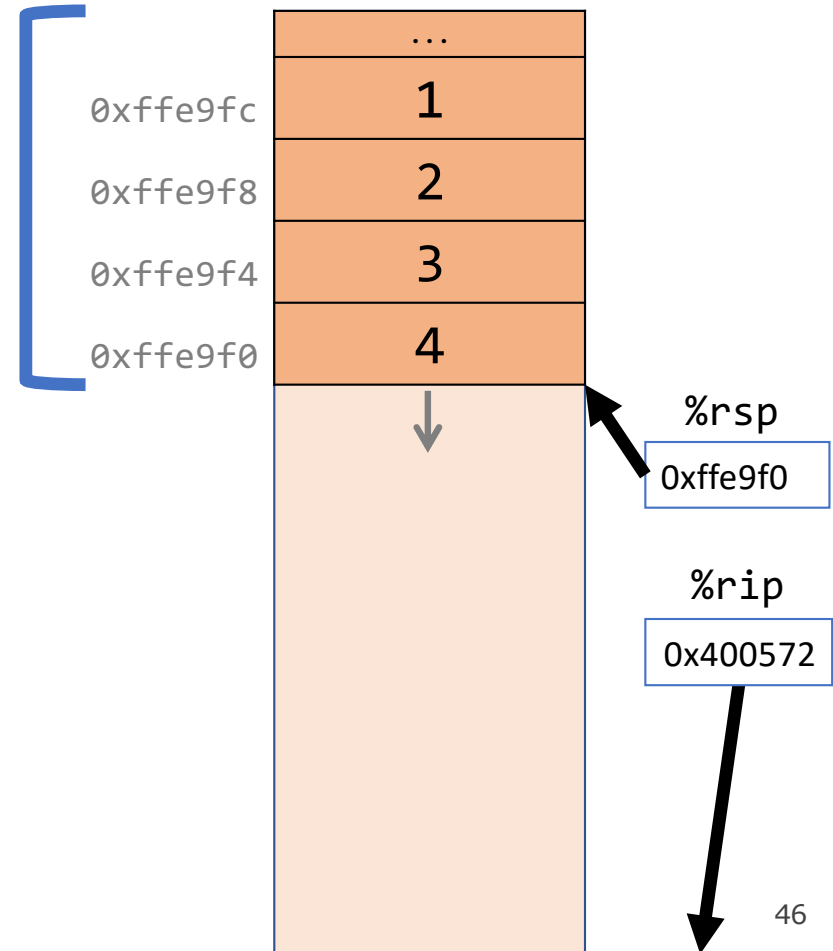


Parameters and Return

```
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()



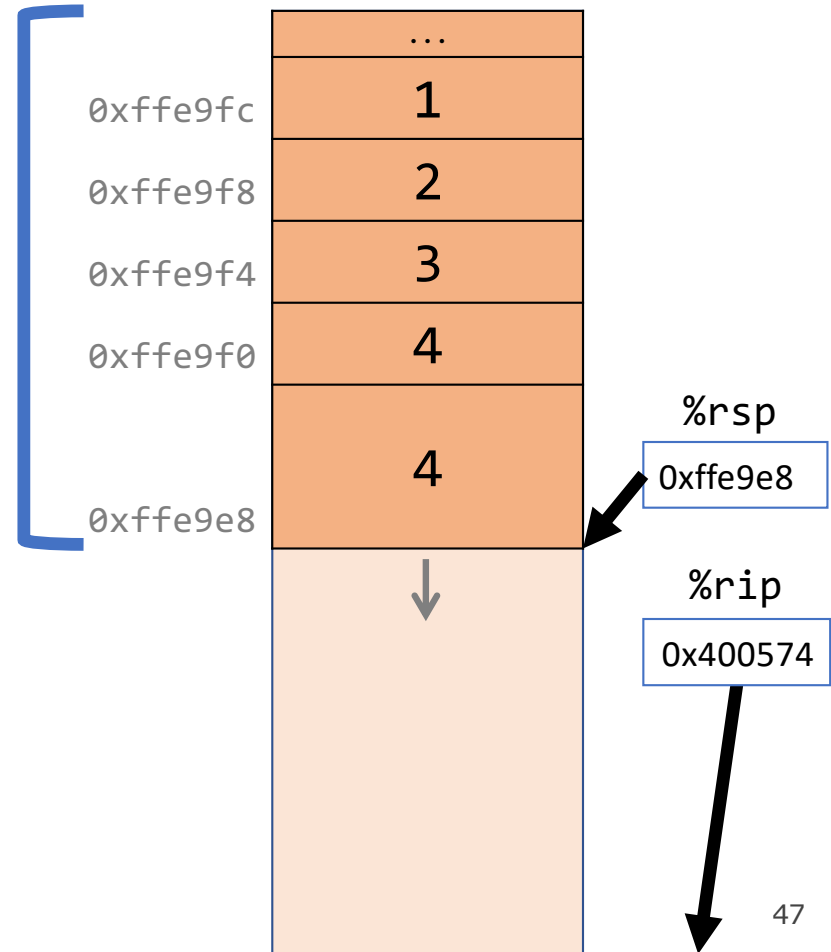
Parameters and Return

```
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()

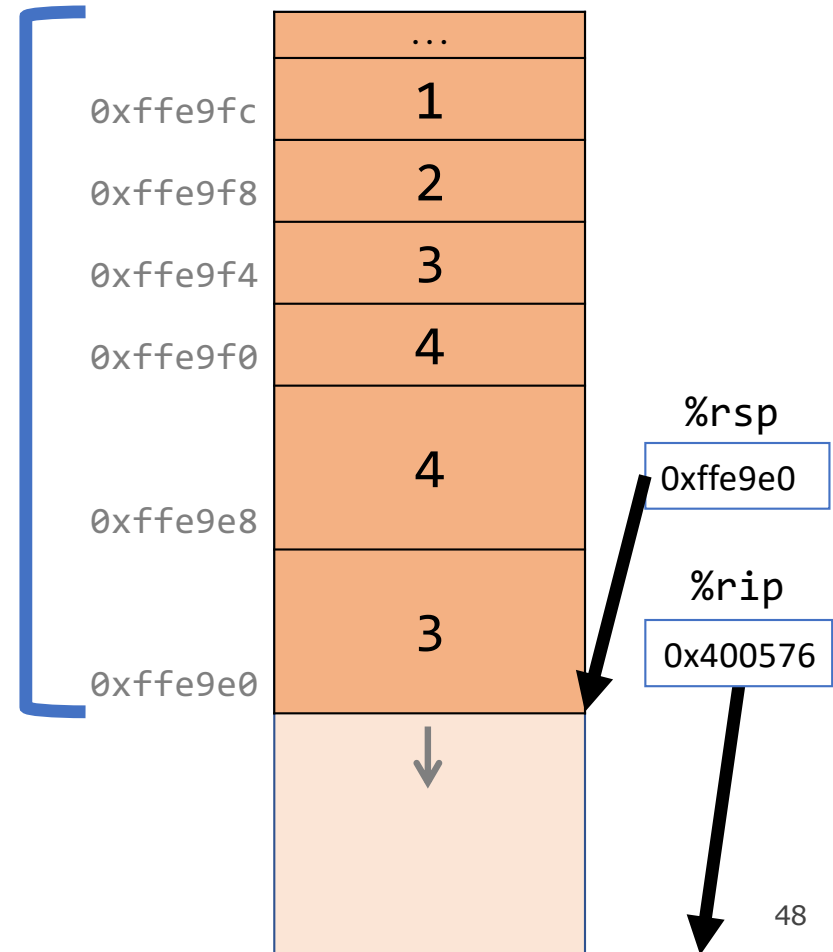


Parameters and Return

```
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()

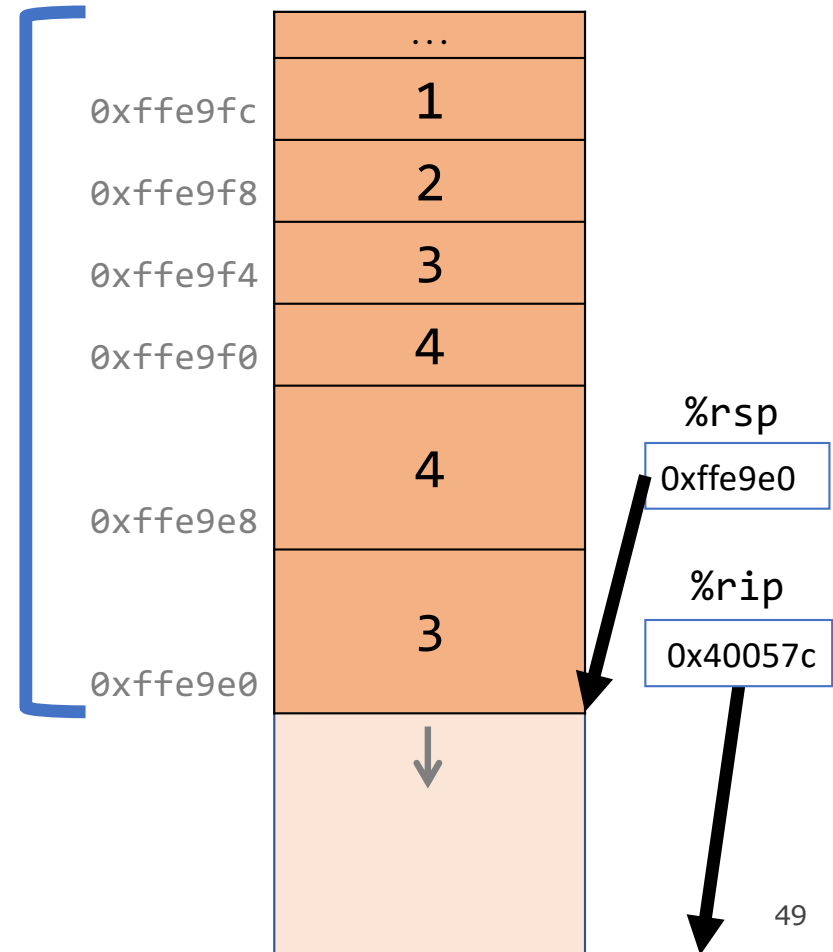


Parameters and Return

```
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()



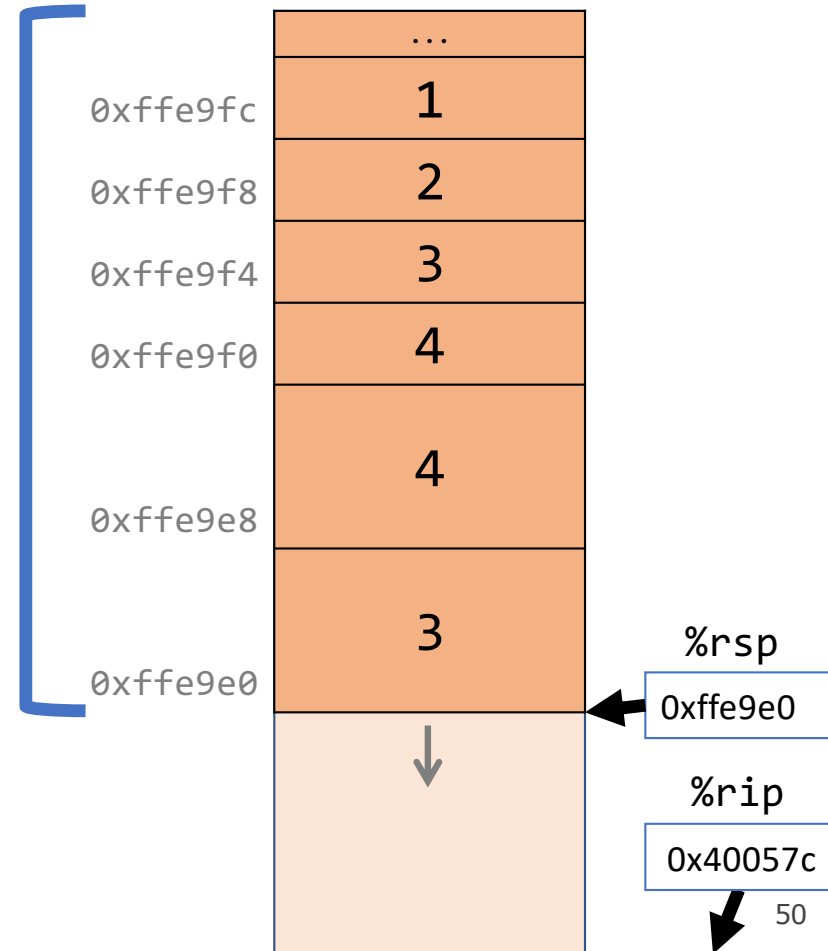
Parameters and Return

```
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
```

main()



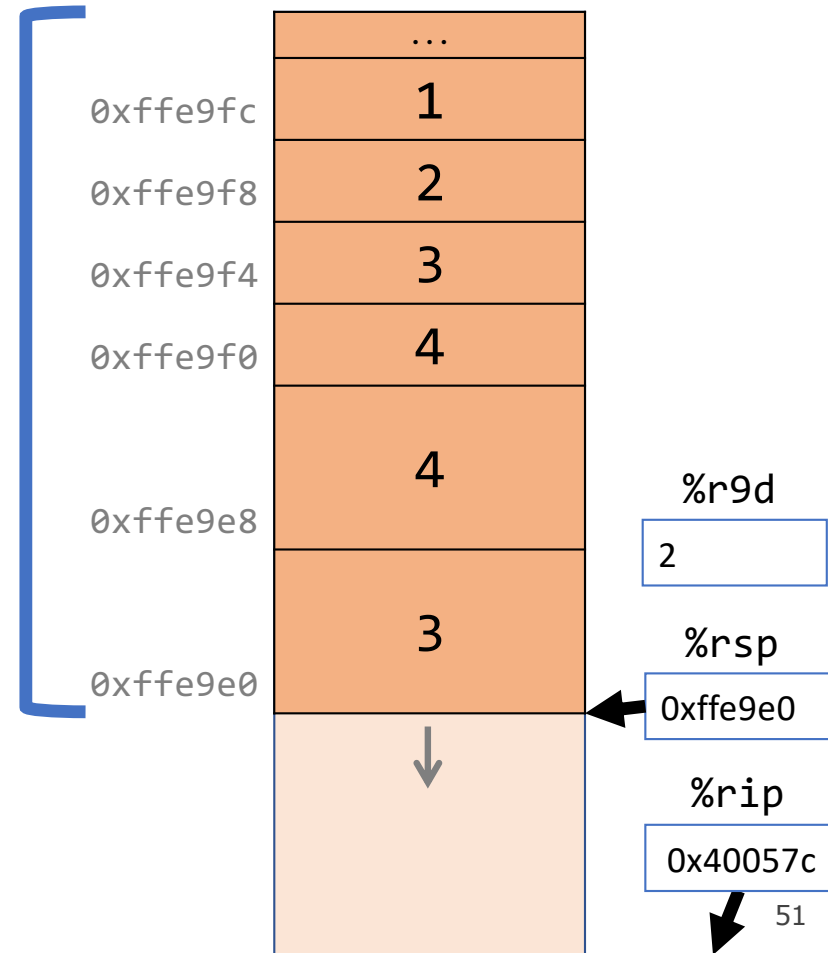
Parameters and Return

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

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

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

main()

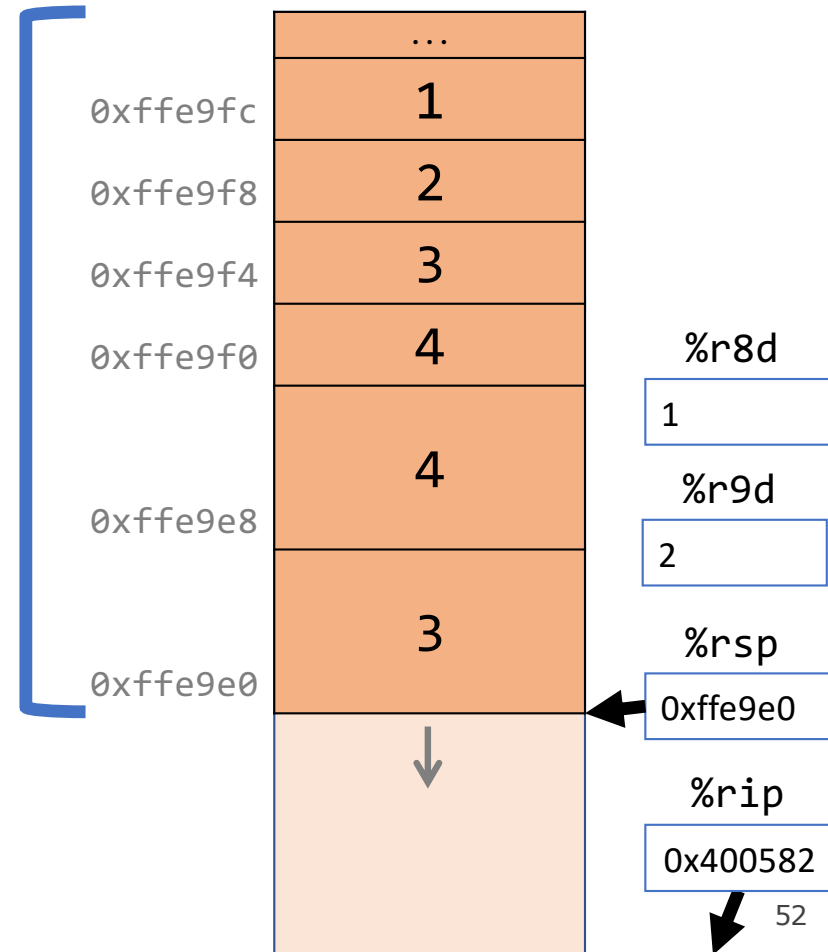


Parameters and Return

```
int main(int argc, char *argv[]) {  
    int i1 = 1;  
    int i2 = 2;  
    int i3 = 3;  
    int i4 = 4;  
    int result = func(&i1, &i2, &i3, &i4,  
                     i1, i2, i3, i4);  
    ...  
}  
  
int func(int *p1, int *p2, int *p3, int *p4,  
         int v1, int v2, int v3, int v4) {  
    ...  
}
```

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

main()



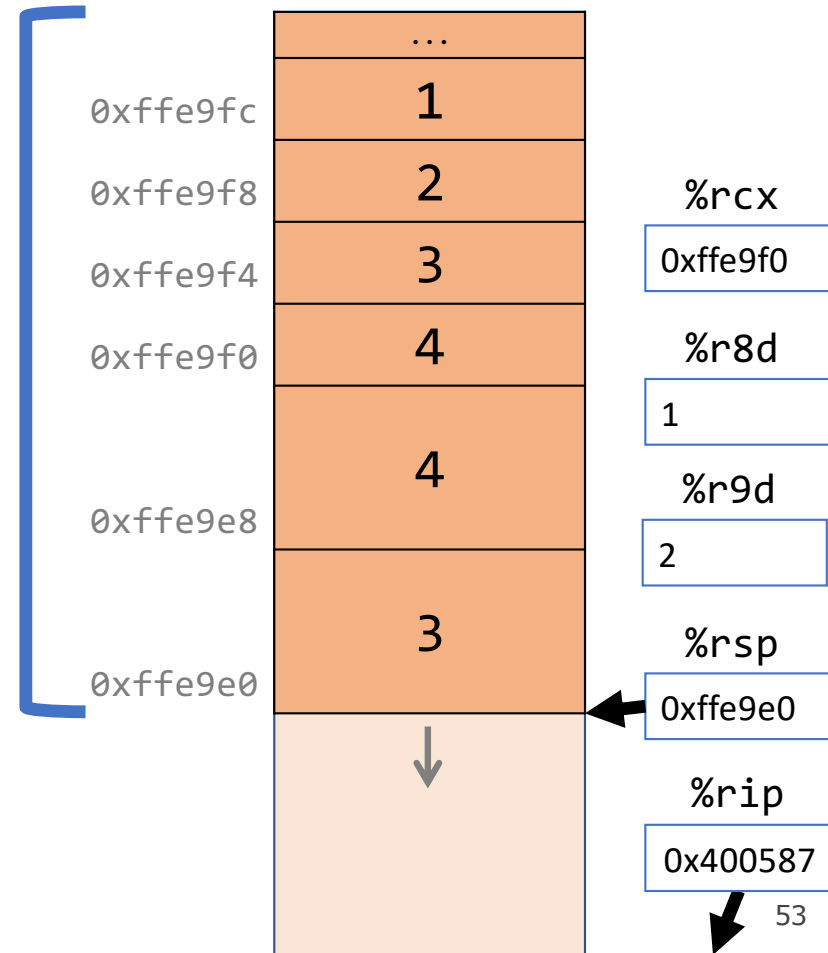
Parameters and Return

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

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

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

main()



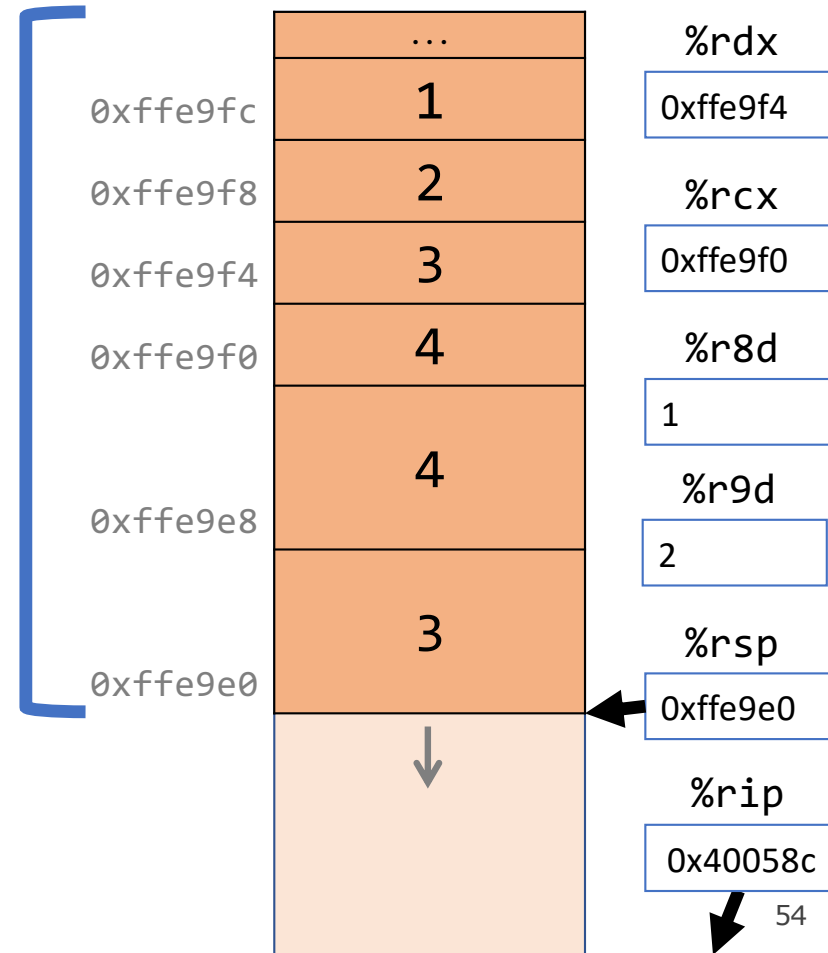
Parameters and Return

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

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

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

main()



Parameters and Return

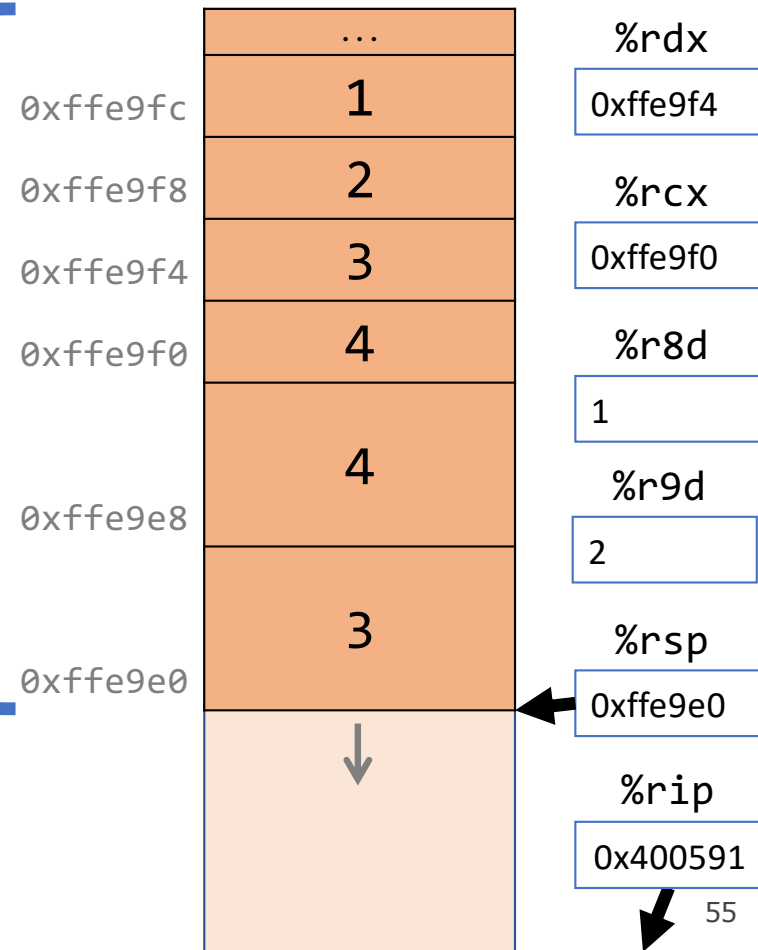
```
int main(int argc, char *argv[]) {  
    int i1 = 1;  
    int i2 = 2;  
    int i3 = 3;  
    int i4 = 4;  
    int result = func(&i1, &i2, &i3, &i4,  
                     i1, i2, i3, i4);  
    ...  
}  
  
int func(int *p1, int *p2, int *p3, int *p4,  
        int v1, int v2, int v3, int v4) {  
    ...  
}
```

```
0x400582 <+51>: lea    0x10(%rsp),%rcx  
0x400587 <+56>: lea    0x14(%rsp),%rdx  
0x40058c <+61>: lea    0x18(%rsp),%rsi  
0x400591 <+66>: lea    0x1c(%rsp),%rdi  
0x400596 <+71>: callq 0x400546 <func>
```

main()

%rsi

0xffe9f8

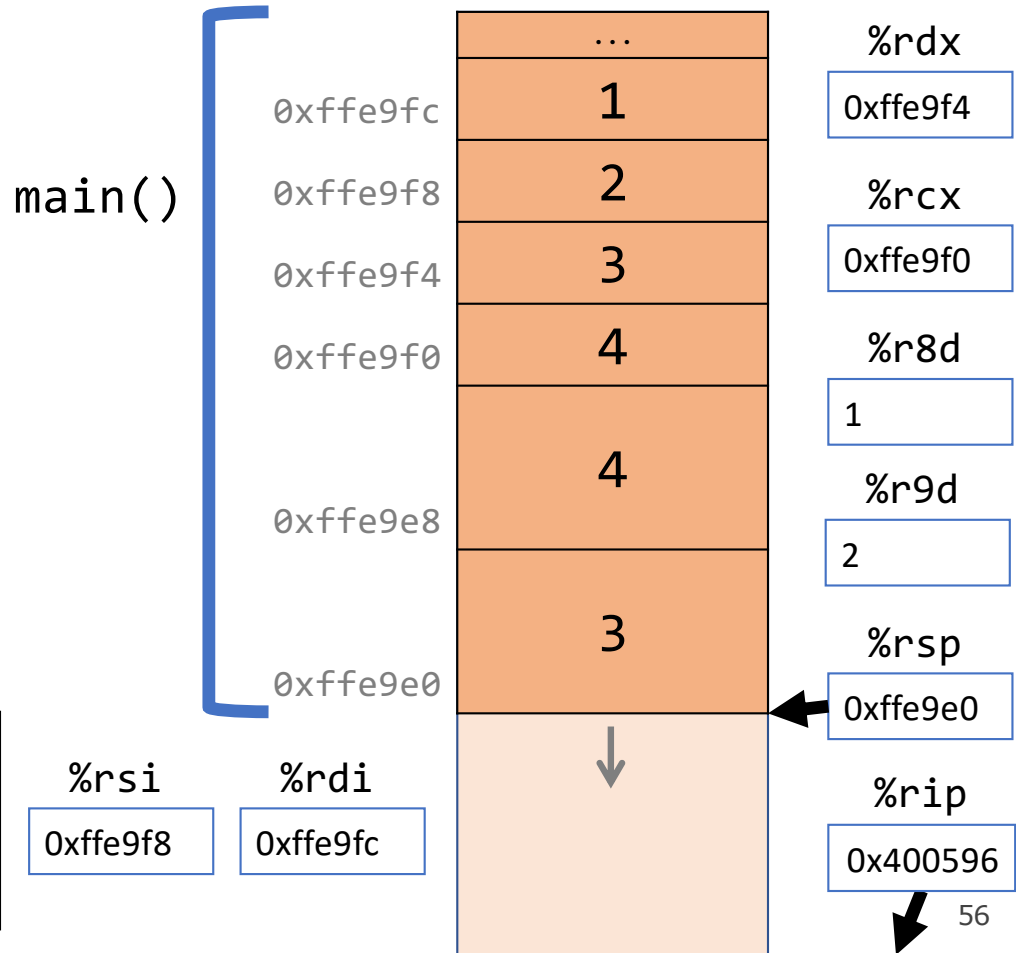


Parameters and Return

```
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
```

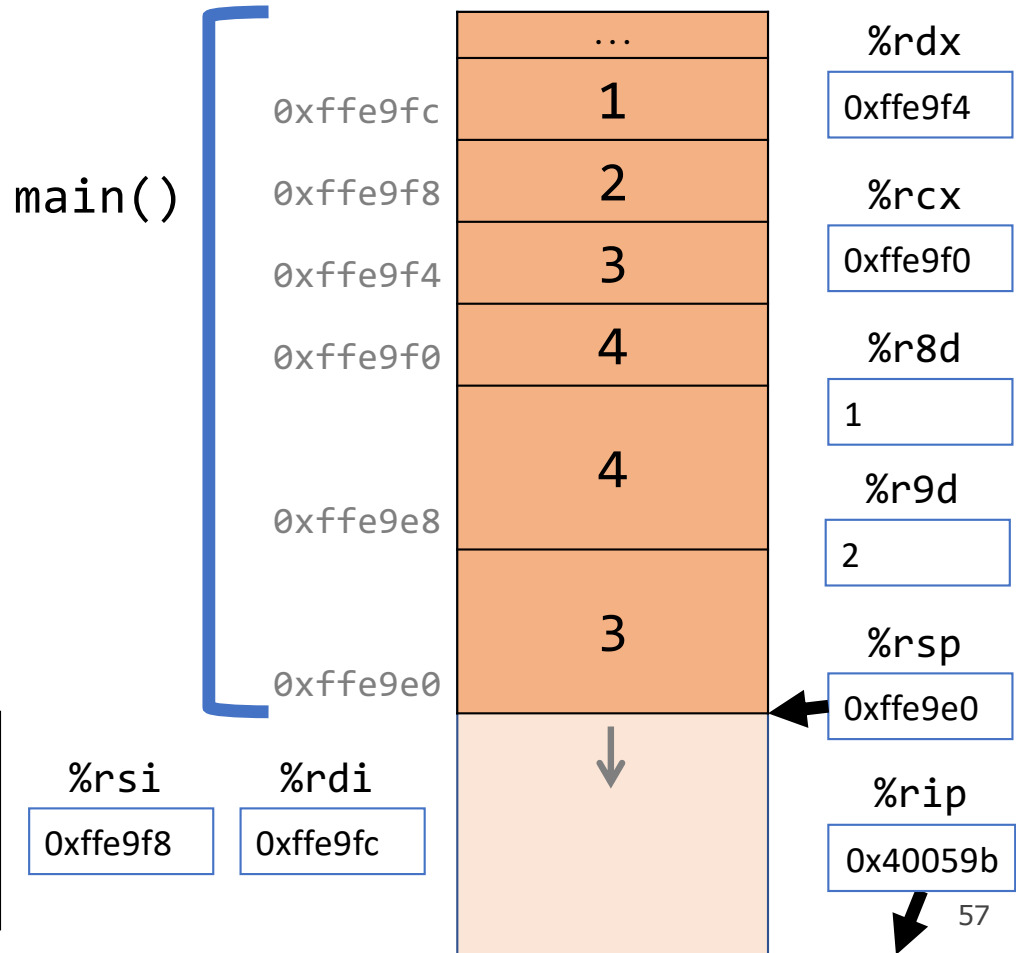


Parameters and Return

```
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
...
```

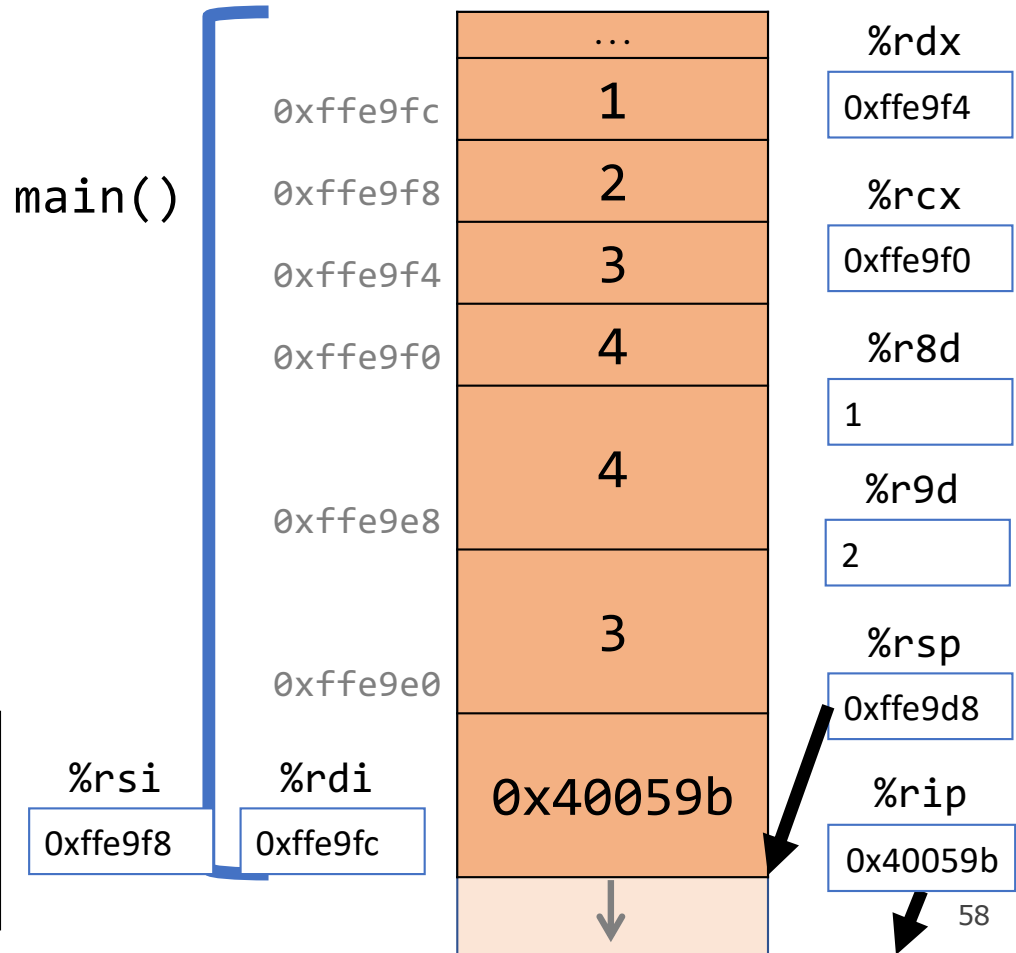


Parameters and Return

```
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  
...
```

