Procedures

Mechanisms in Procedures

Passing control

- To beginning of procedure code
- Back to return point

Passing data

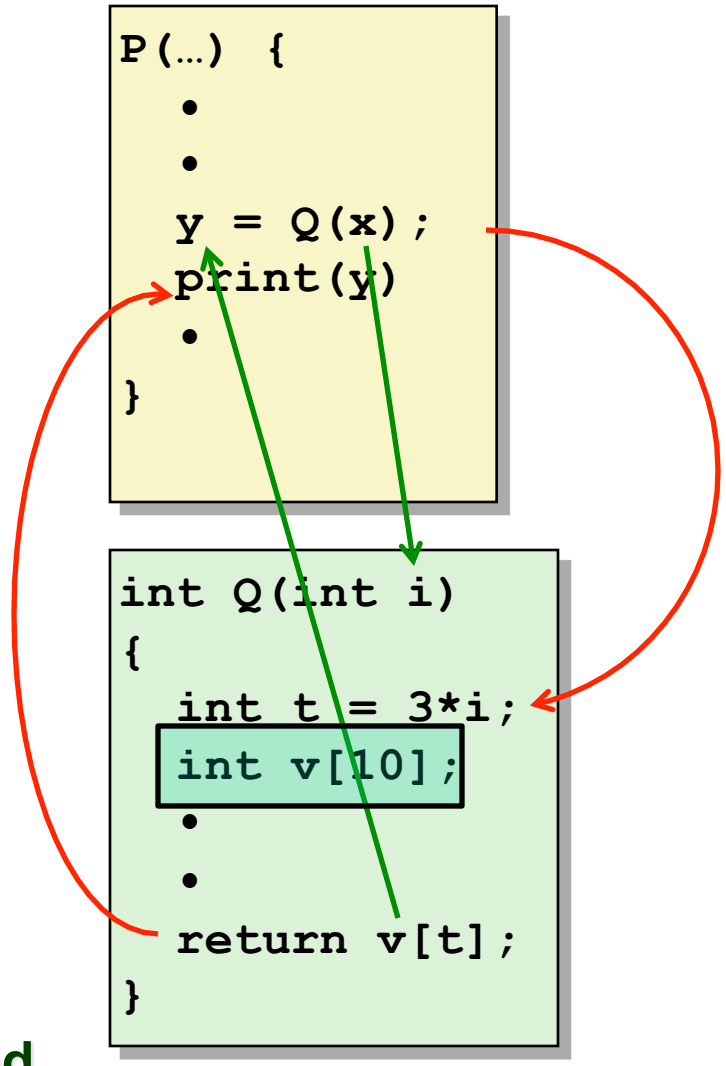
- Procedure arguments
- Return value

Memory management

- Allocate during procedure execution
- Deallocate upon return

Mechanisms all implemented with machine instructions

x86-64 implementation of a procedure uses only those mechanisms required



Procedures

A *Procedure* is a unit of code that we can call

- Depending on the programming language, it may be called a procedure, function, subroutine, or method
- A call is like a jump, except it can return.

The hardware provides machine instructions for this:

`call` *`label`*

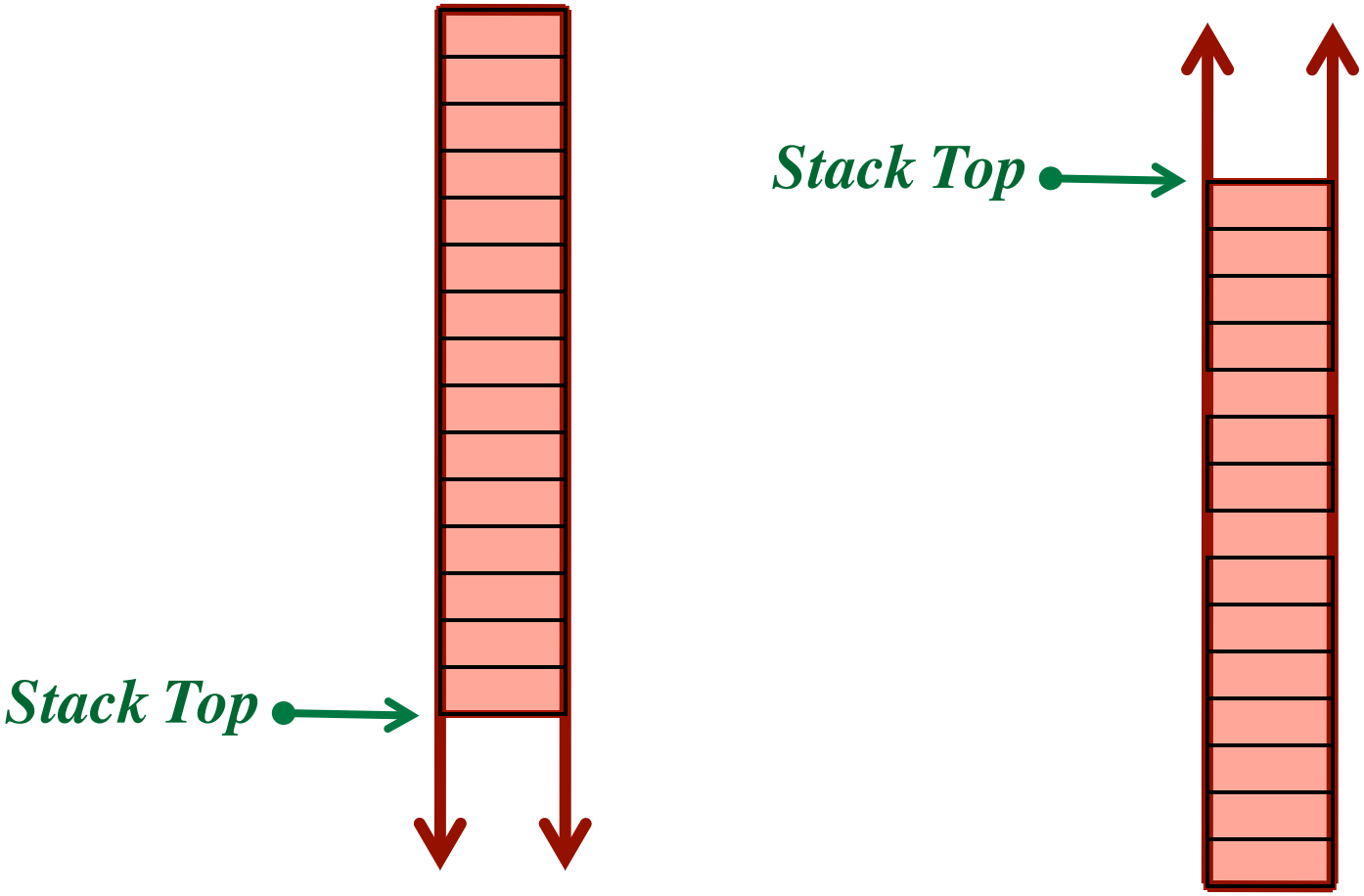
Push return address on stack; Jump to *`label`*

`ret`

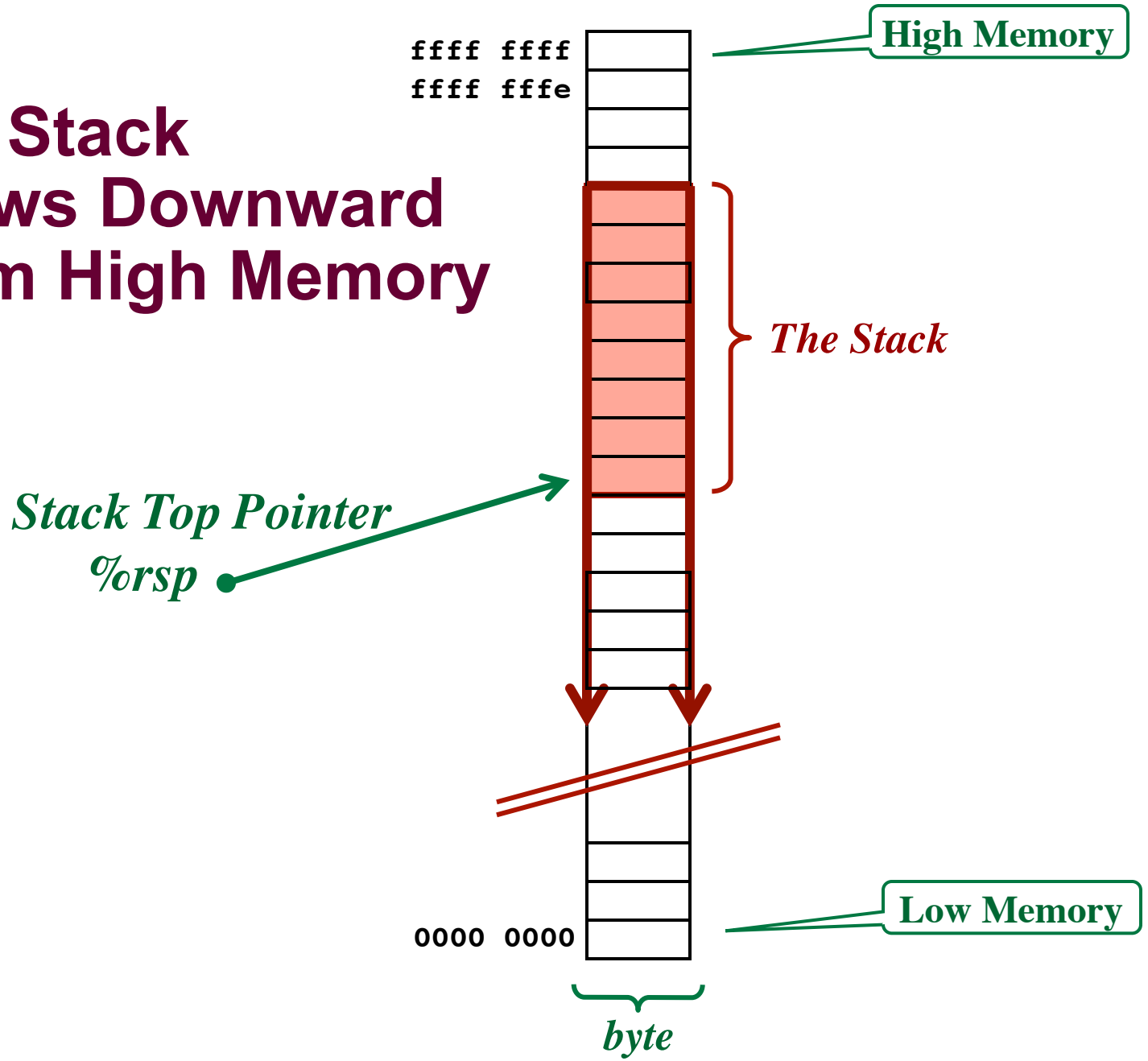
Pop address from stack; Jump to address

First of all, we have to understand how a stack works…

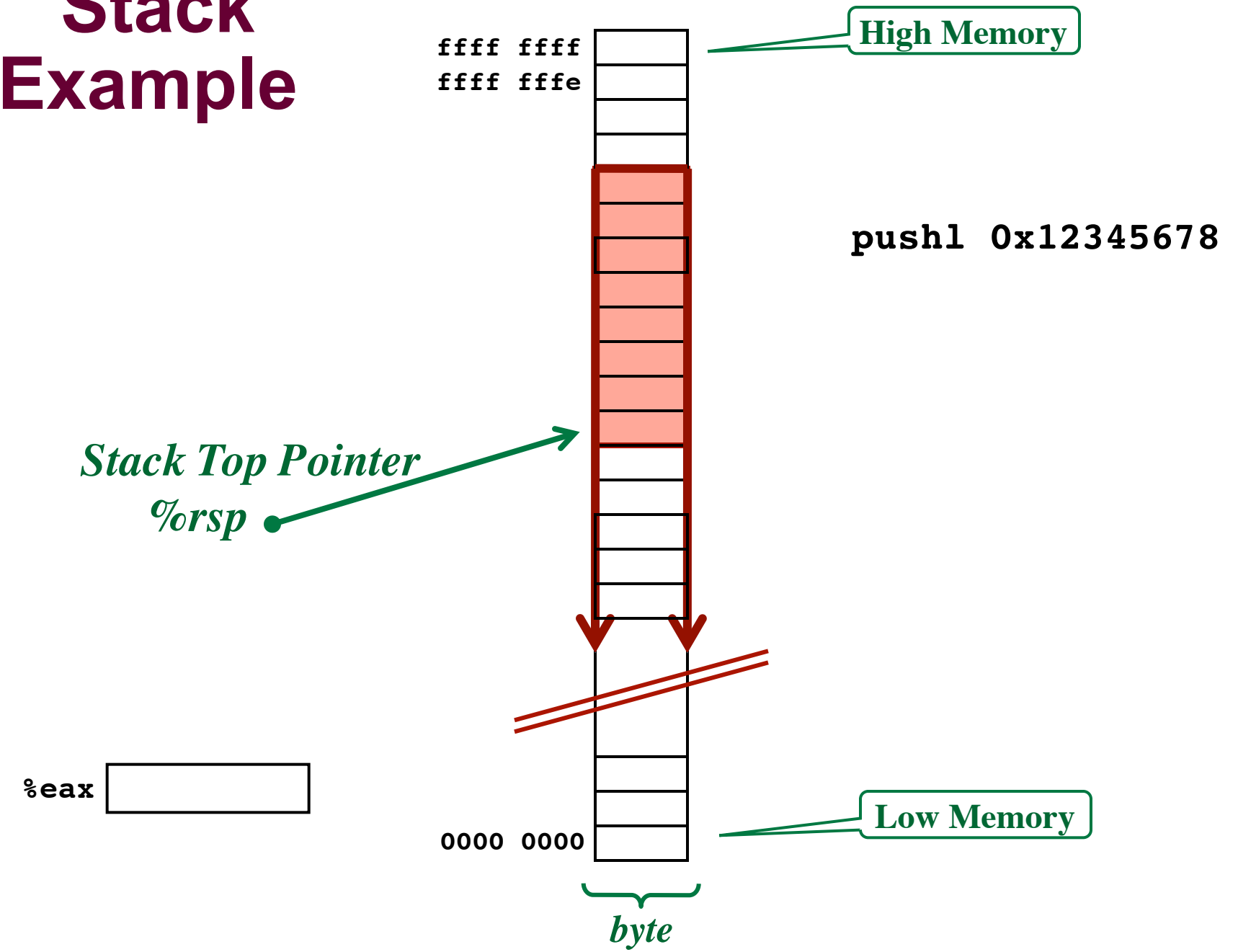
How shall we imagine a stack?



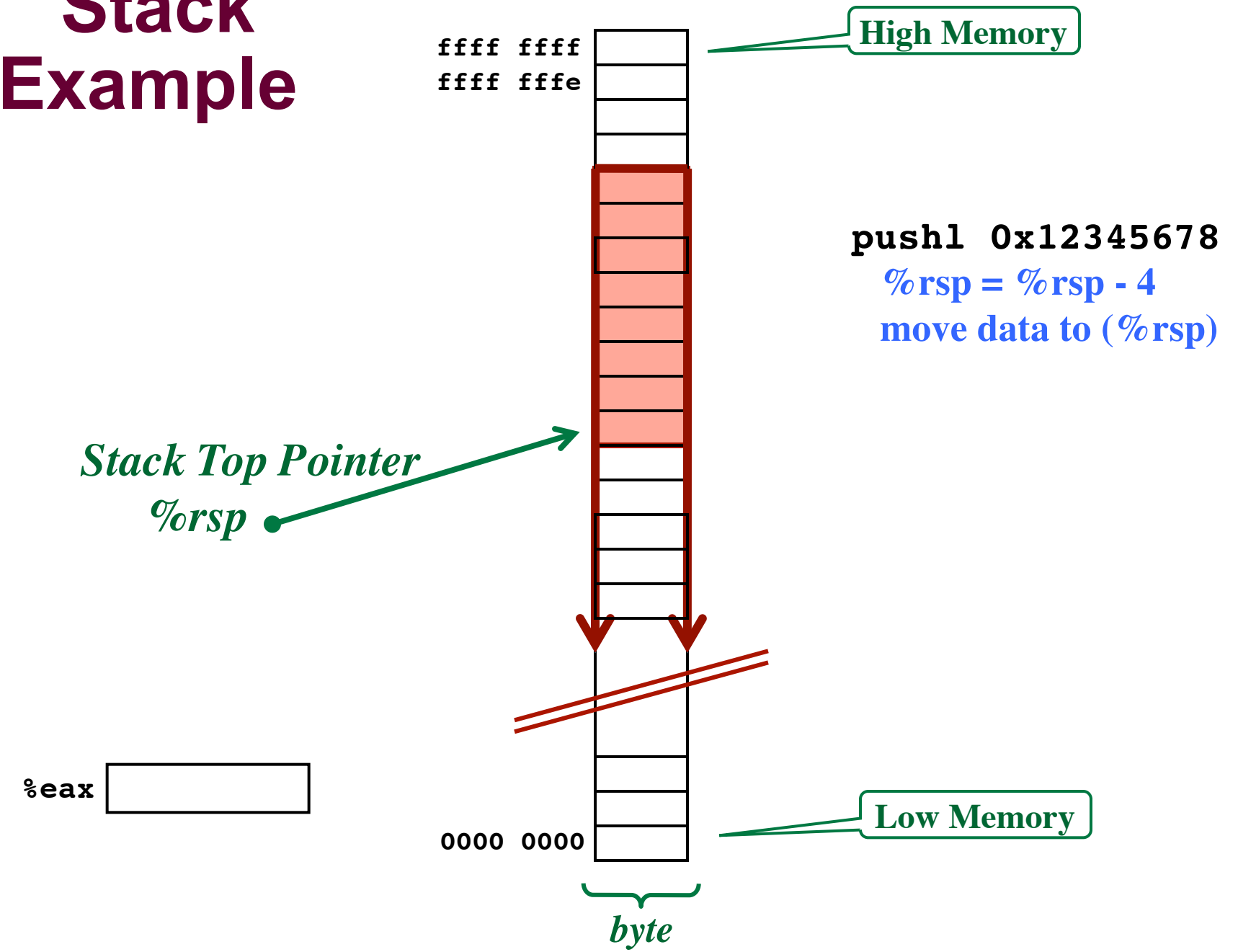
The Stack Grows Downward From High Memory



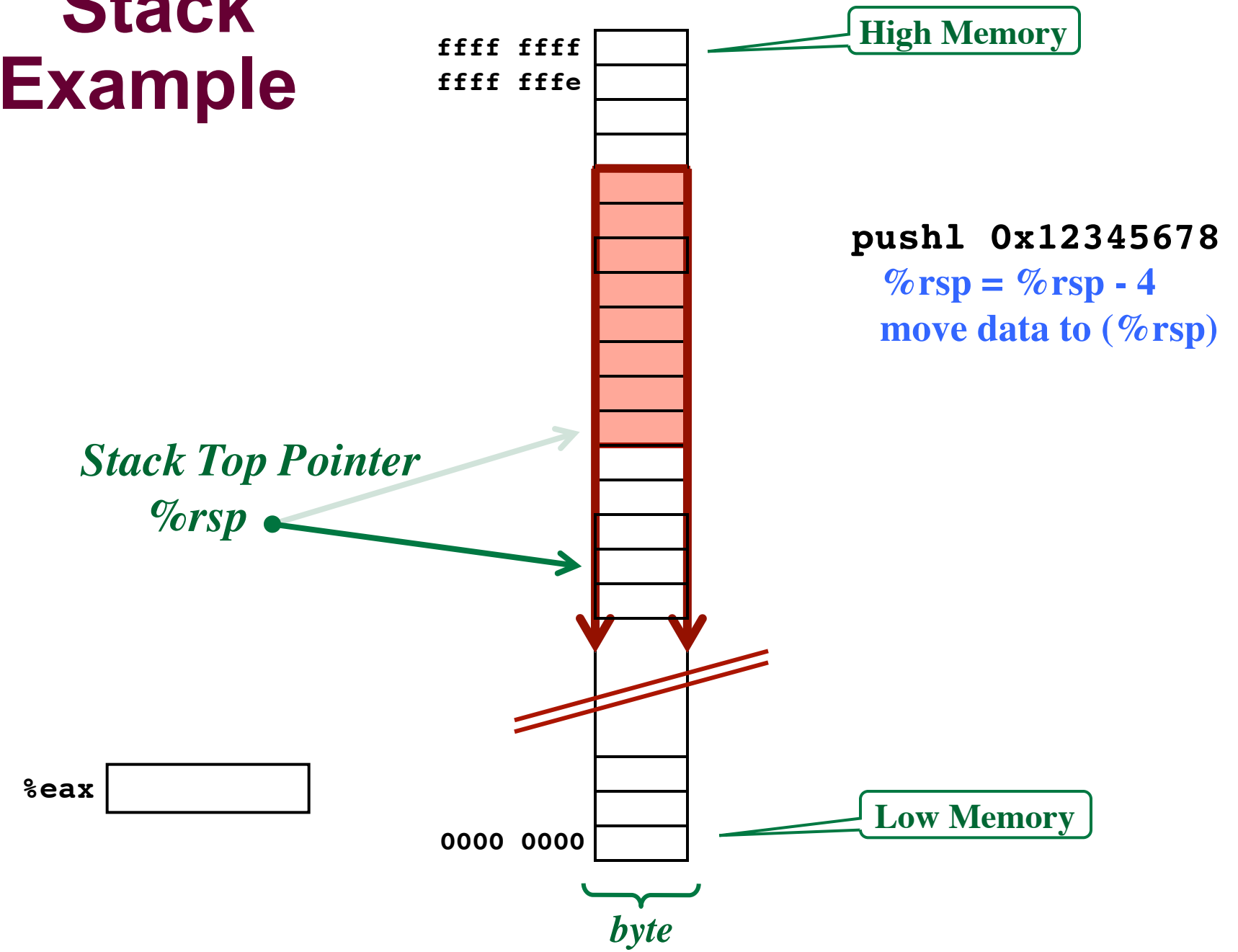
Stack Example



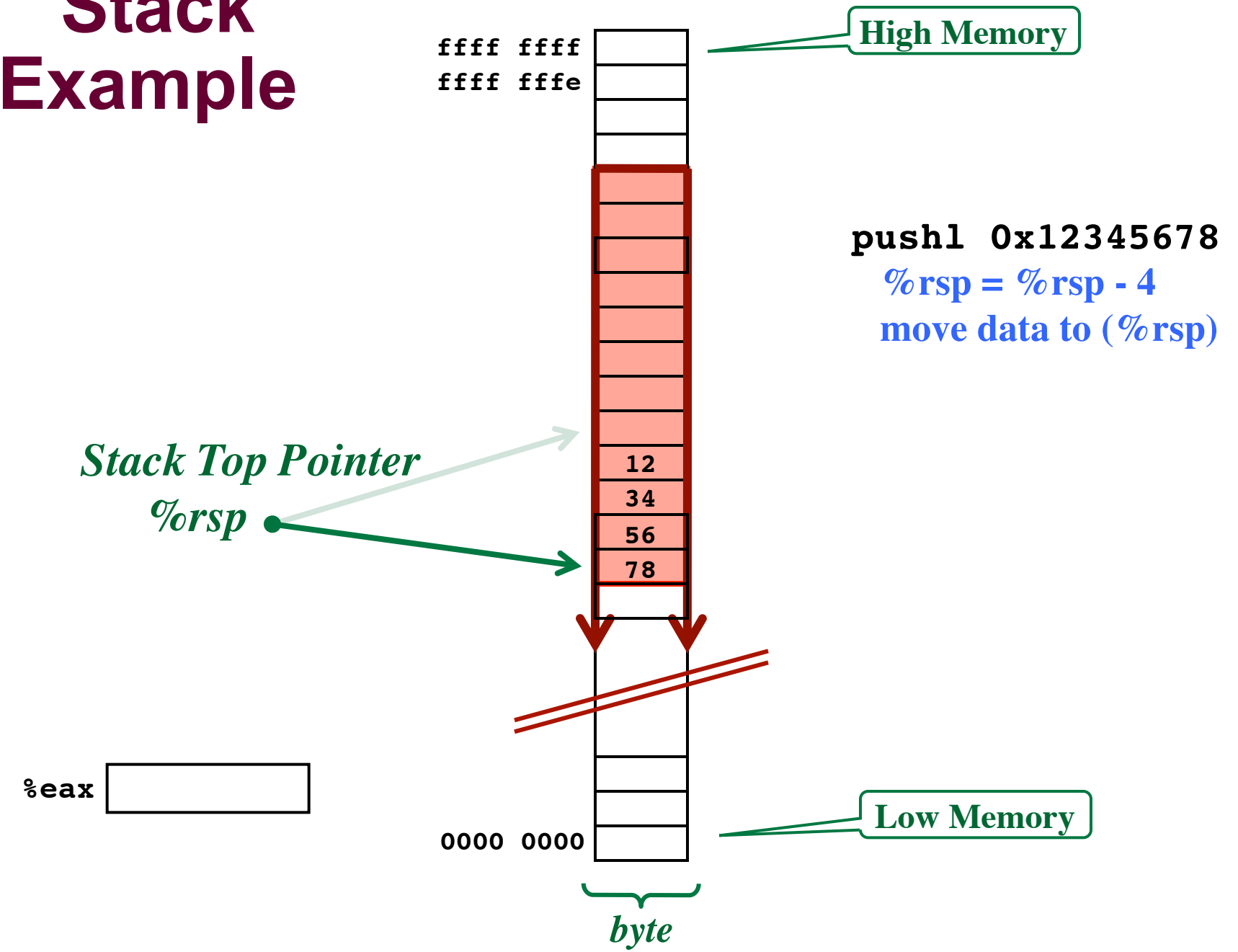
Stack Example



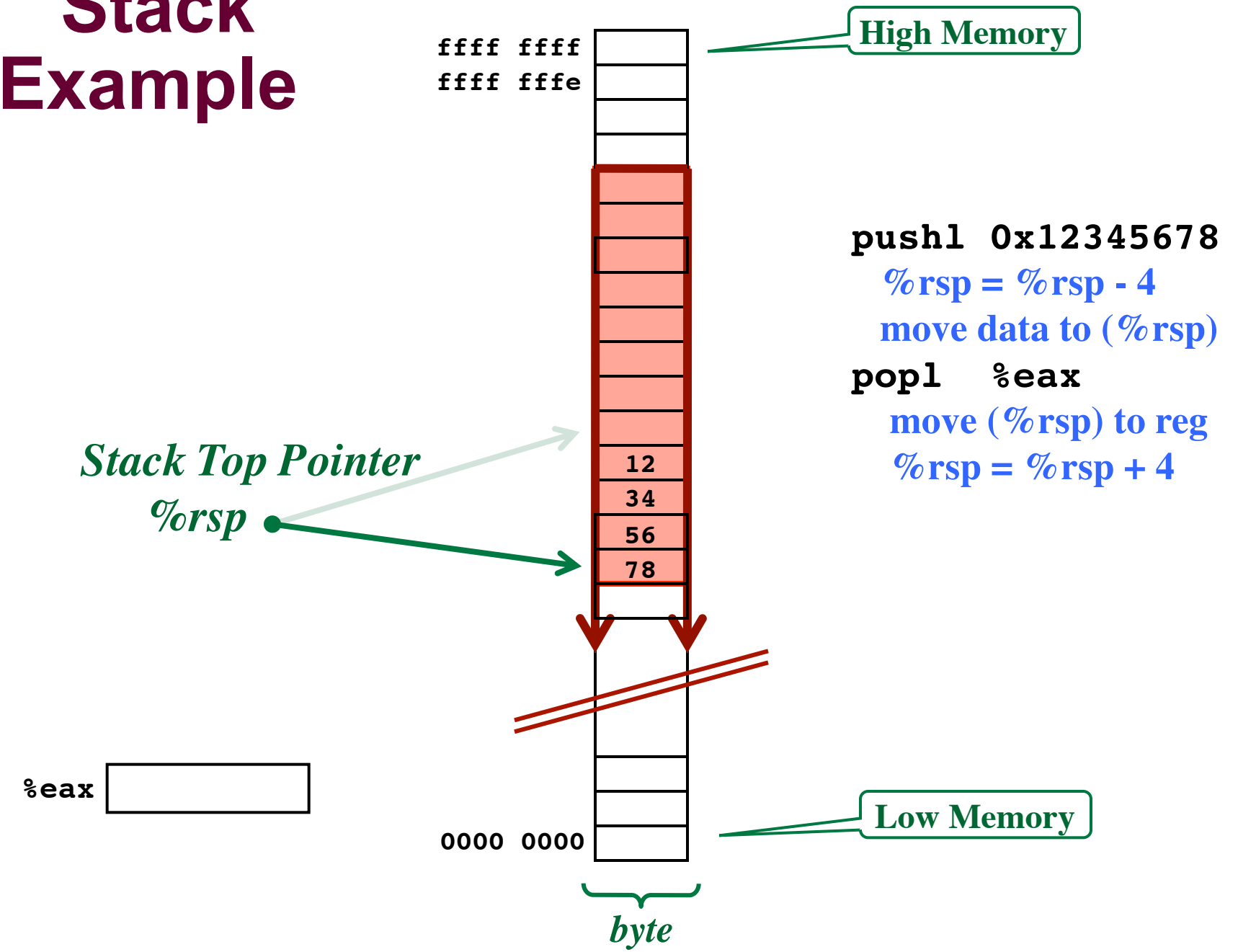
Stack Example



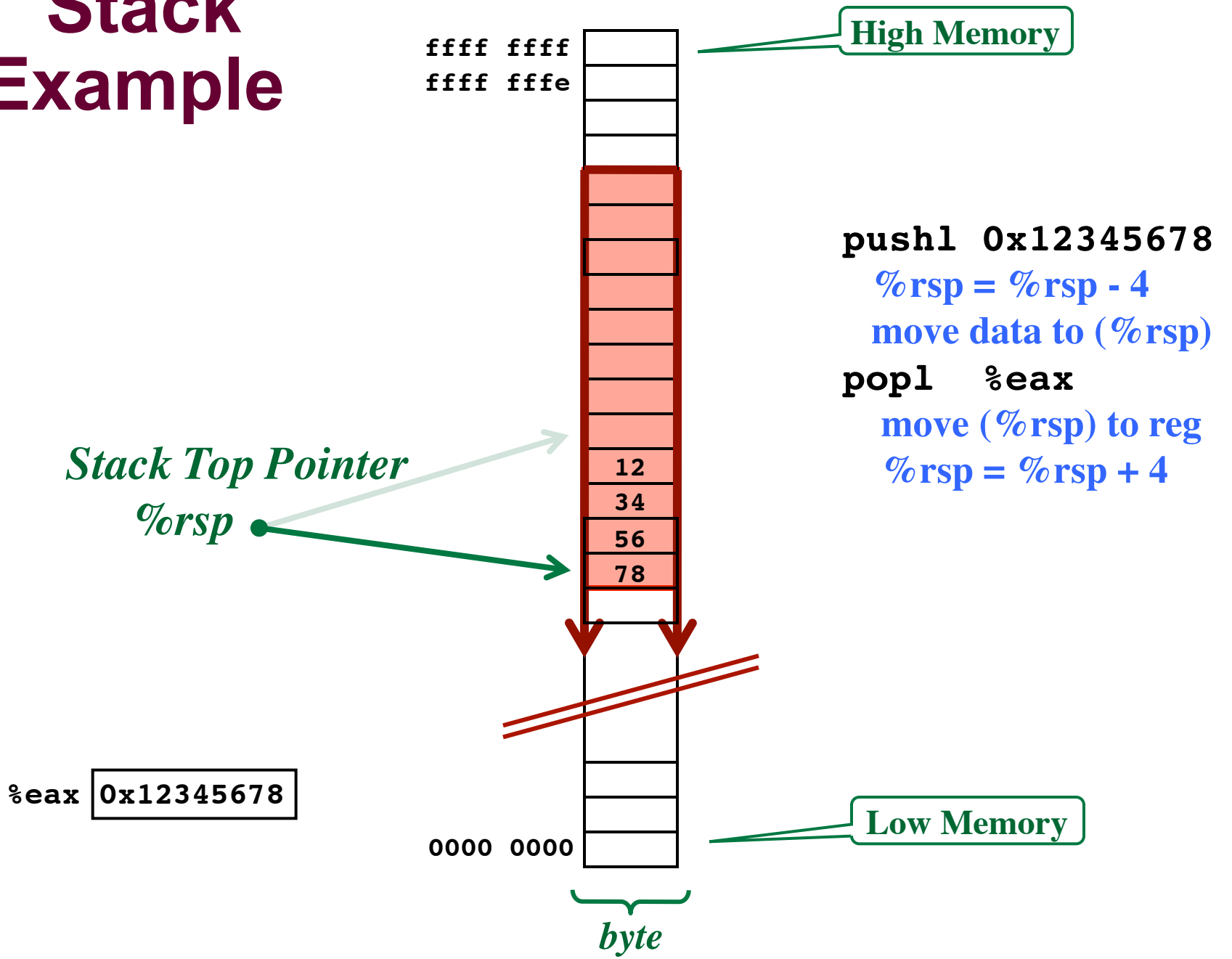
Stack Example



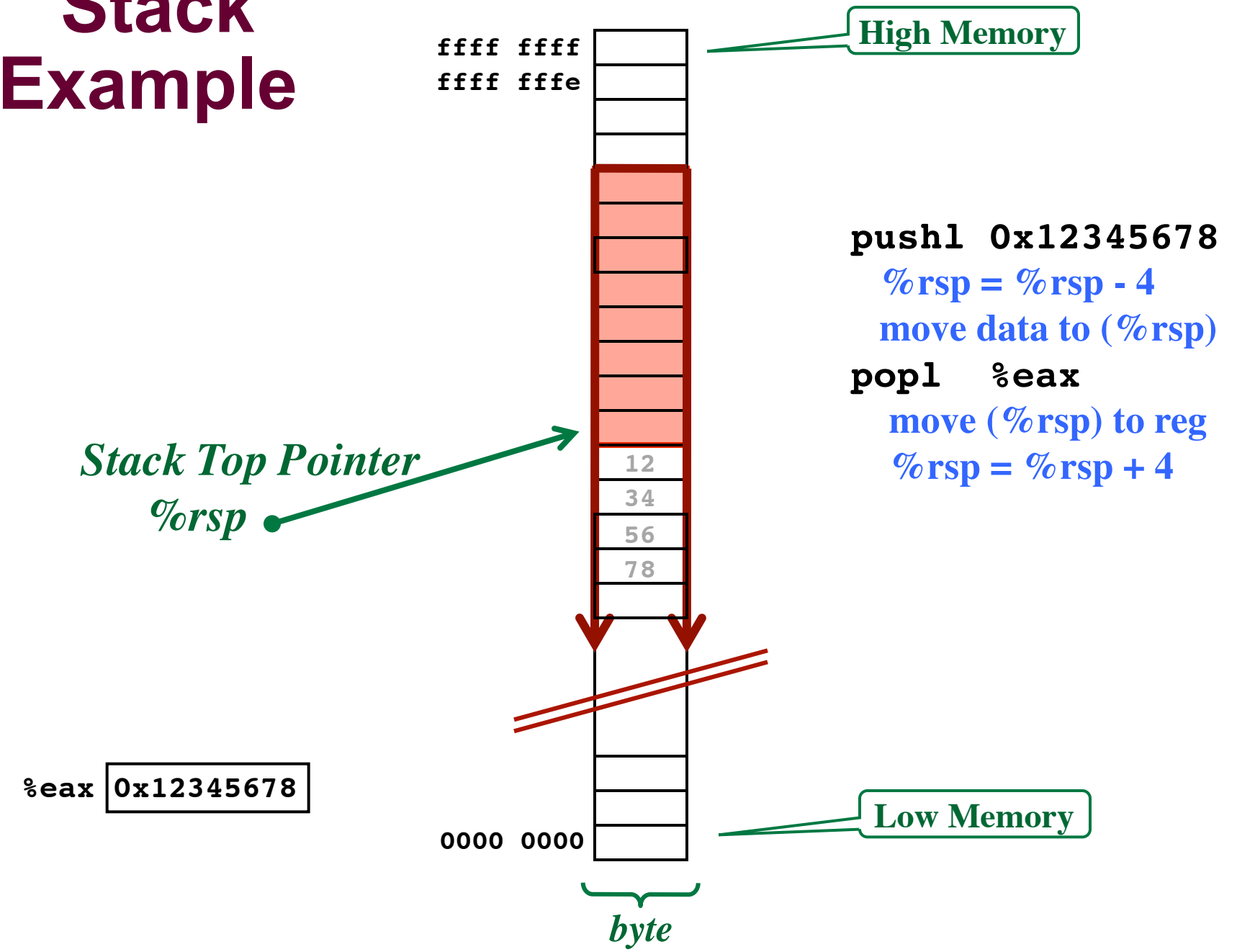
Stack Example



Stack Example



Stack Example




```
pushl 0x12345678
```
%rsp = %rsp - 4
move data to (%rsp)

```
popl  %eax
```
move (%rsp) to reg
%rsp = %rsp + 4

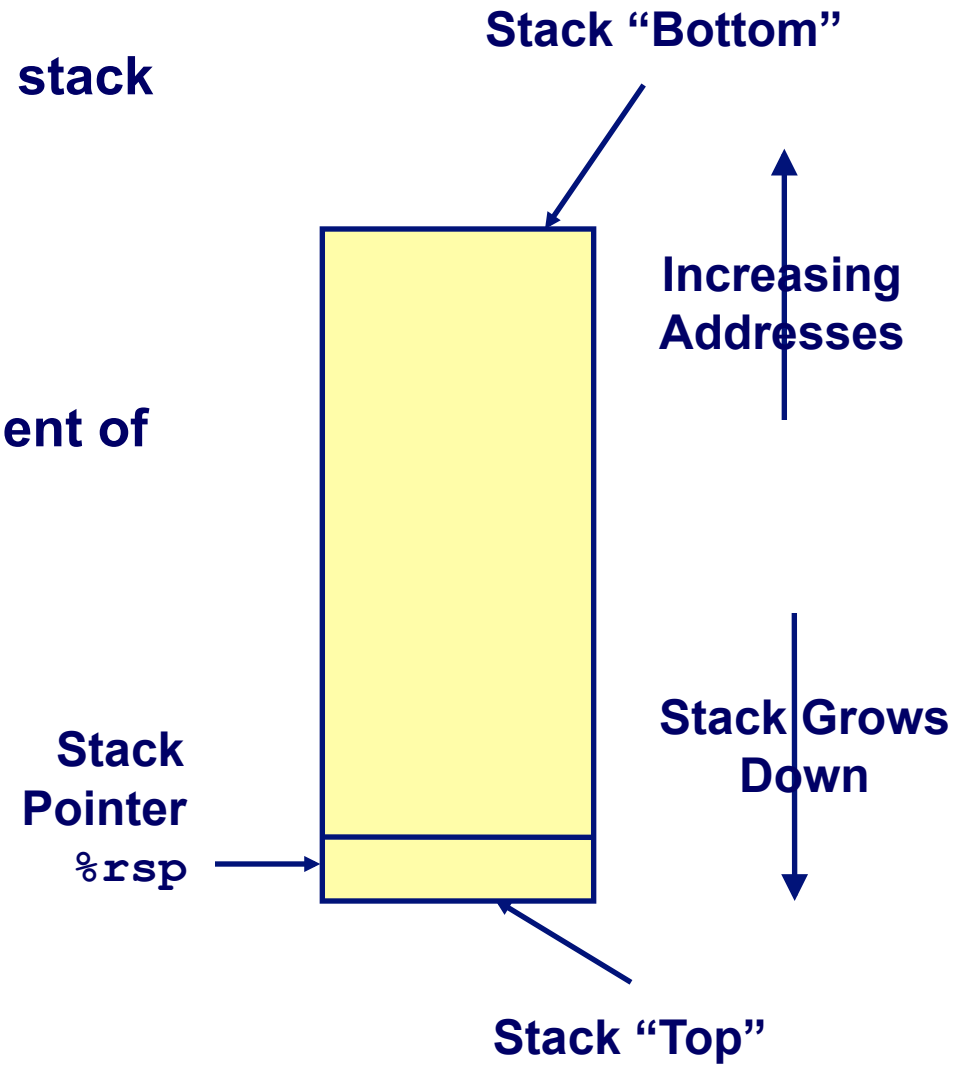
The x86-64 Stack

Region of memory managed with stack discipline

Grows toward lower addresses

Register %rsp indicates top element of stack

Top element has lowest address



Stack instructions (push and pop)

Stack manipulation is just data movement that updates the stack pointer register

- **Move data onto the stack (pushq)**
- **Move data off of the stack (popq)**

The stack is essential for implementing function calls

- **Function parameters**
- **Return address**
- **Prior stack frame information**
- **Local function variables**

Instruction	Effect
`pushq` *Source*	R[%rsp] ← R[%rsp] – 8; M[R[%rsp]] ← Source
`popq` *Dest*	Dest ← M[R[%rsp]]; R[%rsp] ← R[%rsp] + 8

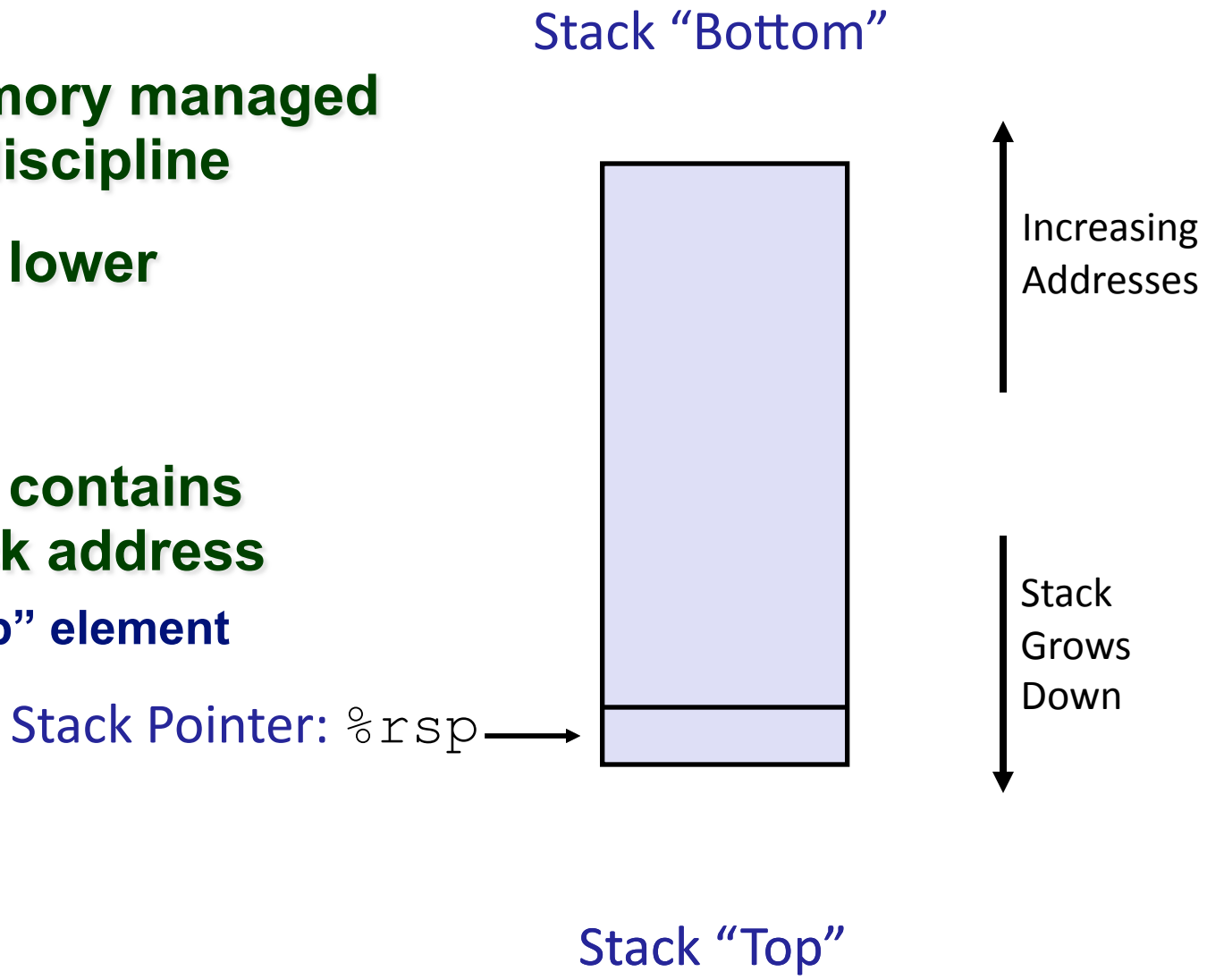
x86-64 Stack

Region of memory managed with stack discipline

Grows toward lower addresses

Register `%rsp` contains lowest stack address

address of "top" element

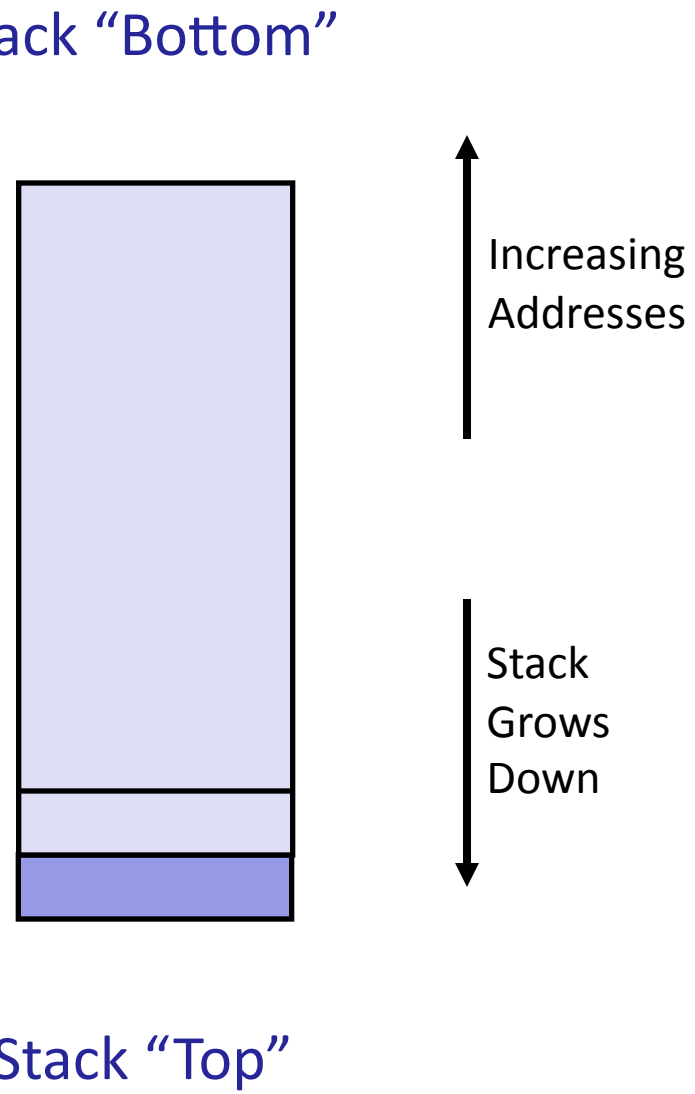


x86-64 Stack: Push

`pushq Src`

- **Fetch operand at Src**
- **Decrement `%rsp` by 8**
- **Write operand at address given by `%rsp`**

```
pushq %rax
    subq $8, %rsp
    movq %rax,(%rsp)
```



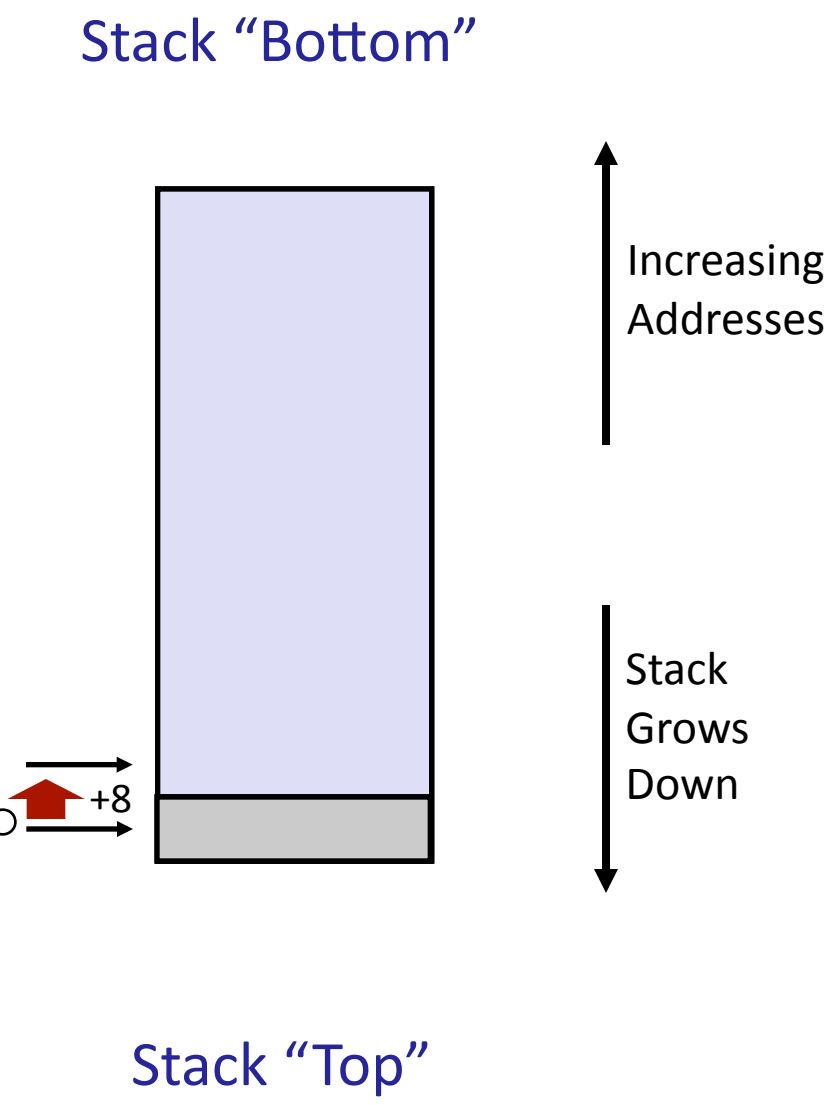
x86-64 Stack: Pop

`popq` Dest

- Read value at address given by `%rsp`
- Increment `%rsp` by 8
- Store value at Dest (must be register)

```
popq %rax
```

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



Examples

```
void multstore
  (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  400540: push   %rbx              # Save %rbx
  400541: mov    %rdx,%rbx         # Save dest
  400544: callq  400550 <mult2>    # mult2(x,y)
  400549: mov    %rax,(%rbx)       # Save at dest
  40054c: pop    %rbx              # Restore %rbx
  40054d: retq                     # Return
```

```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax    # a
  400553:  imul   %rsi,%rax    # a * b
  400557:  retq                # Return
```

Procedure Control Flow

Use stack to support procedure call and return

Procedure call: `call label`

- Push return address on stack
- Jump to label

Return address:

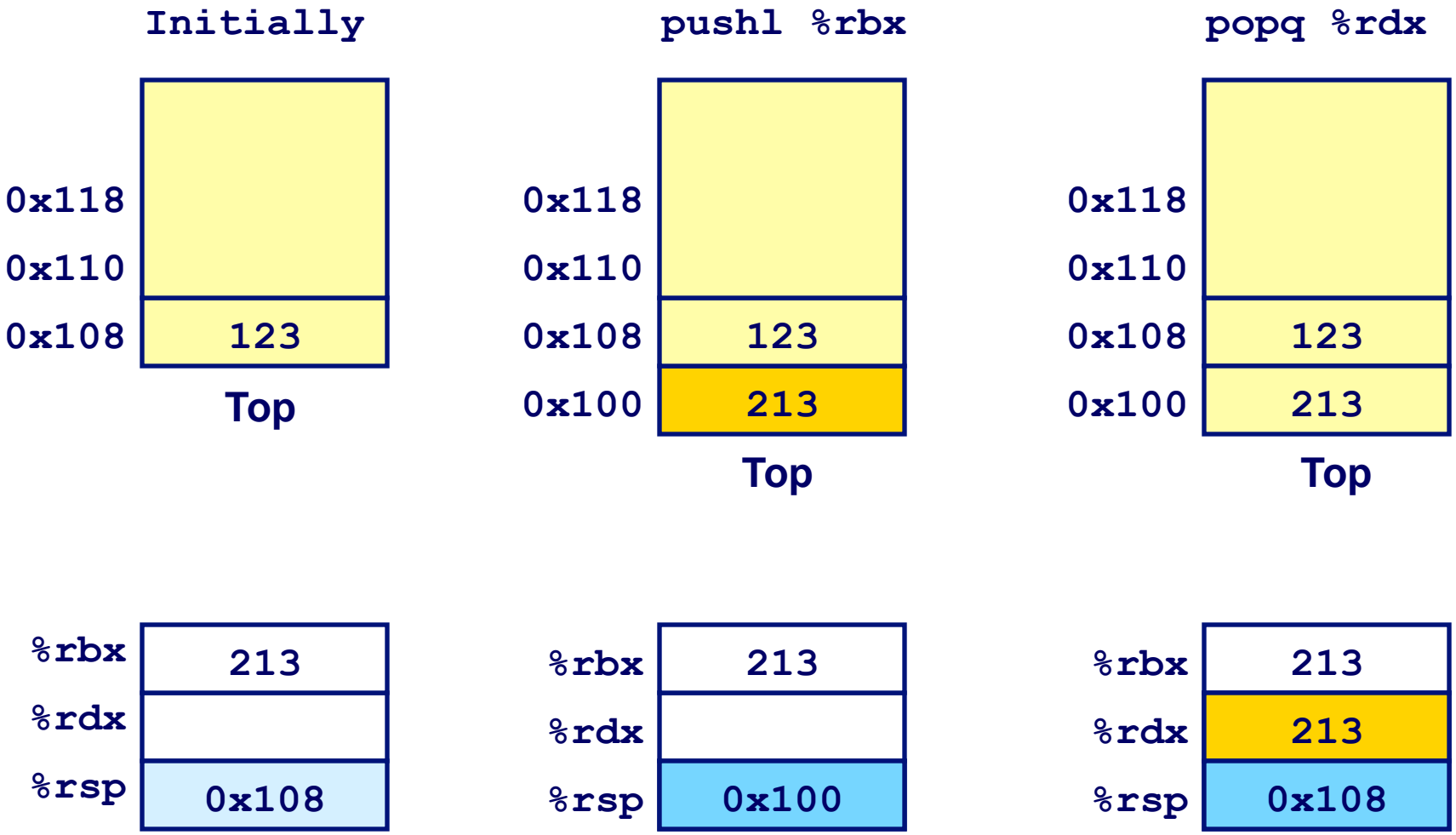
- Address of the next instruction right after call

Procedure return: `ret`

- Pop address from stack
- Jump to address

Stack Operation Examples

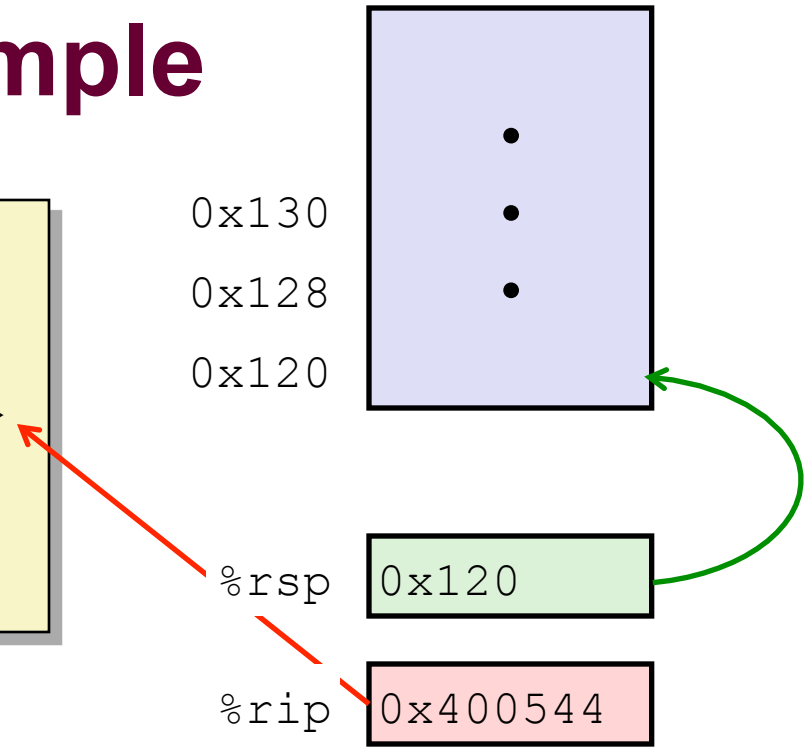
animation



Control Flow Example

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

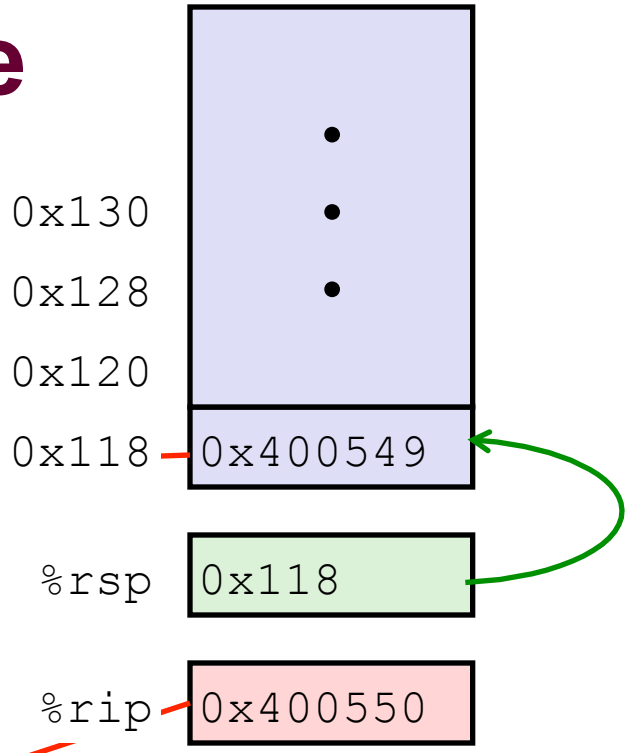
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

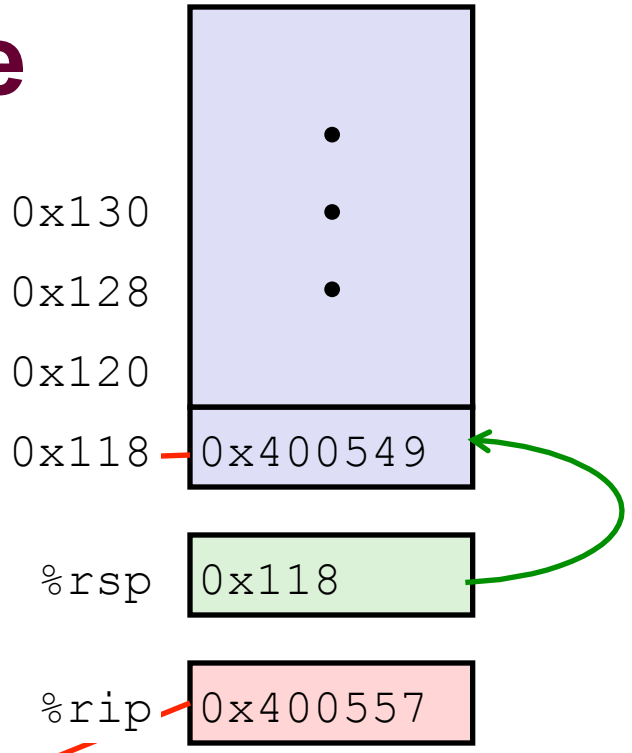
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

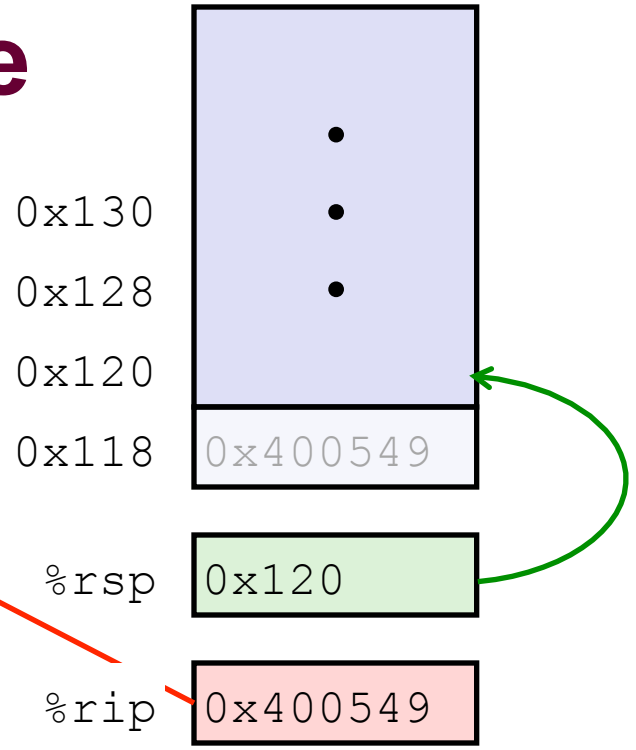
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

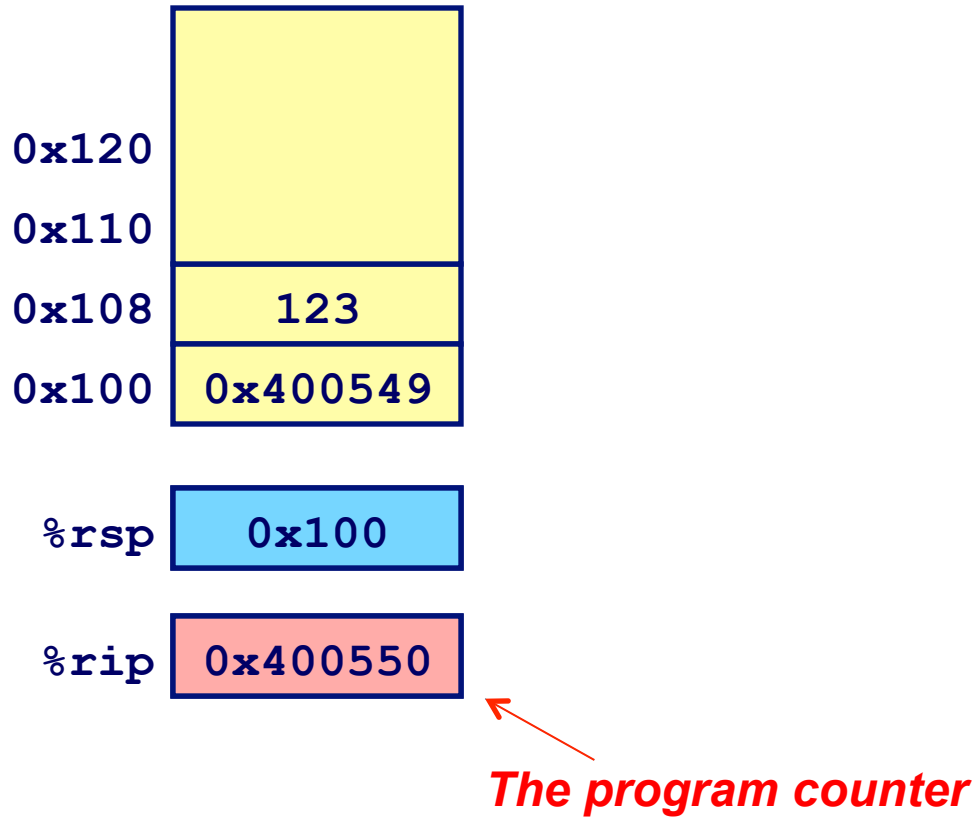
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Procedure Call Example

Calling code:

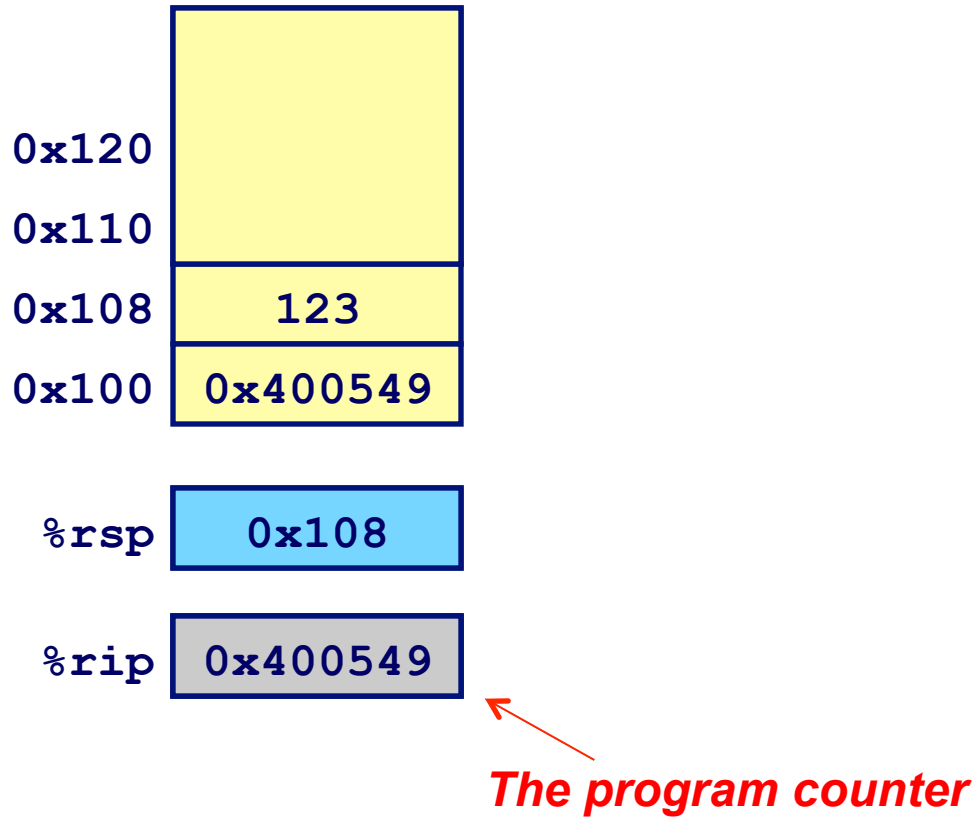
```
400544:   e8 3d 06 00 00   call   400550 <mult2>
400549:   50                       next instruction
```



Procedure Return Example

Within mult2:

```
400550:    48 89 fa    mov     %rdi,%rax
  •••                      •••
400577:    c3          ret
```



Procedure Control Flow

Procedure `main` calls procedure `f1`:

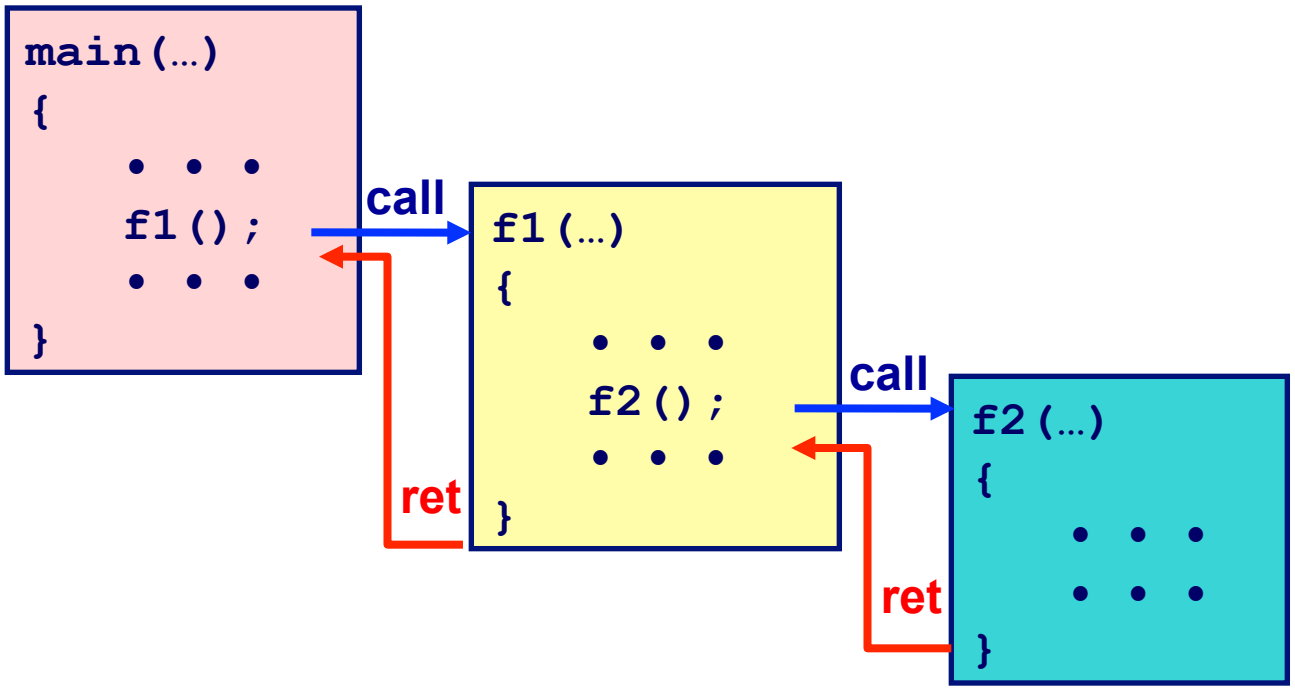
`main` is the *"caller"*, `f1` is the *"callee"*

CALL: Control is transferred to the *callee*

RETURN: Control is transferred back to the *caller*

Last-called, first-return (LIFO) order

→ Implemented via stack



Procedure calls and stack frames

How does the 'callee' know where to return later?

- Return address placed in a well-known location on stack within a "stack frame"

How are arguments passed to the 'callee'?

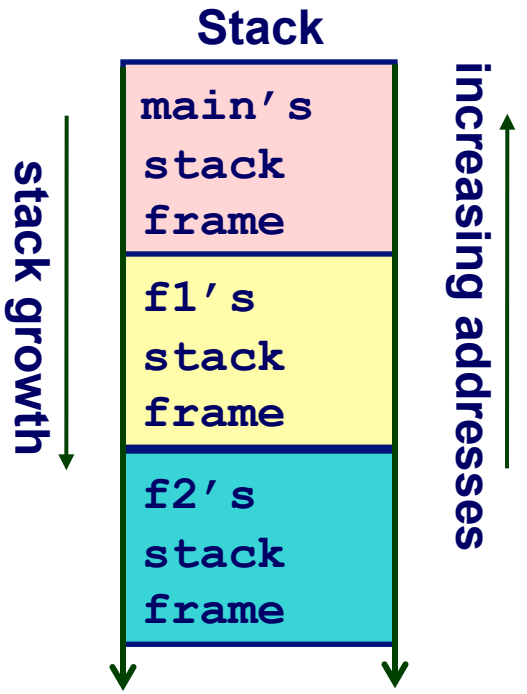
- Arguments placed in a well-known location on stack within a "stack frame"

Upon procedure invocation

- Stack frame created for the procedure
- Stack frame is pushed onto program stack

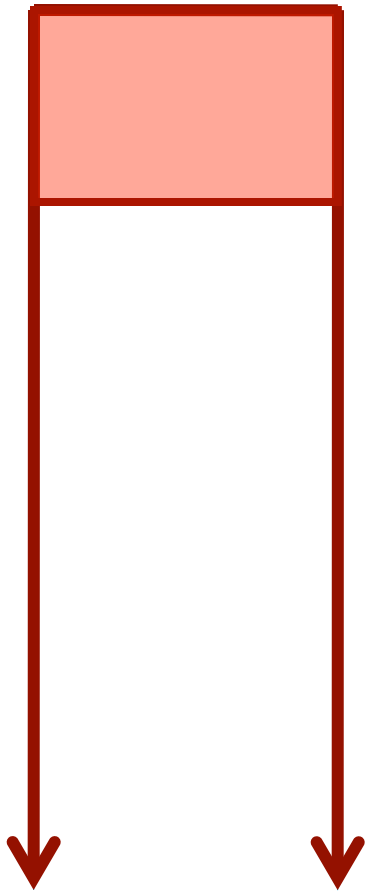
Upon procedure return

- Its frame is popped off of stack
- Caller's stack frame is recovered



Functions and Stack Frames

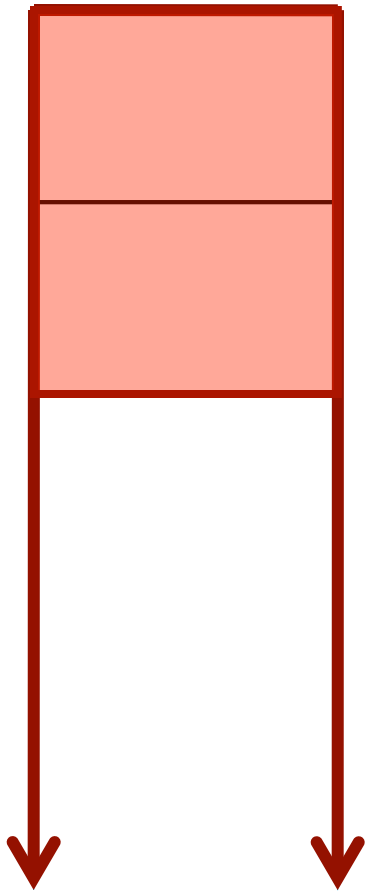
```
main() {
   … call f1()
}

f1() {
   … call f2()
}

f2() {
   … call f3()
}

f3 () {
    …
}
```



Frame of **main**

Functions and Stack Frames

```
main() {
   … call f1()
}

f1() {
   … call f2()
}

f2() {
   … call f3()
}

f3 () {
    …
}
```

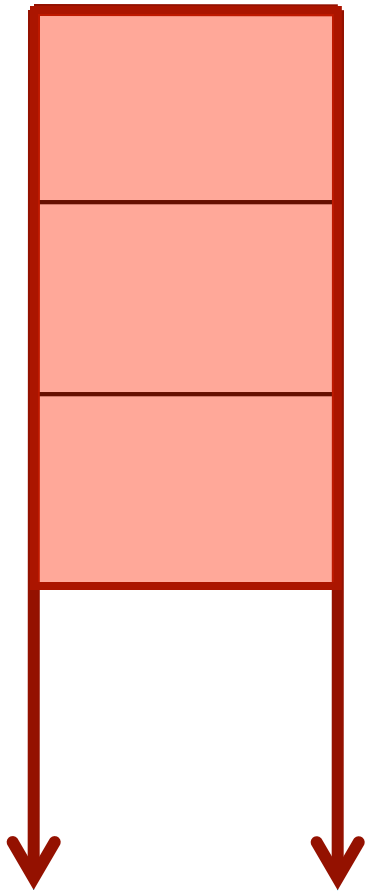


Frame of **`main`**

Frame of **`f1`**

Functions and Stack Frames

```
main() {
   … call f1()
}

f1() {
   … call f2()
}

f2() {
   … call f3()
}

f3 () {
    …
}
```



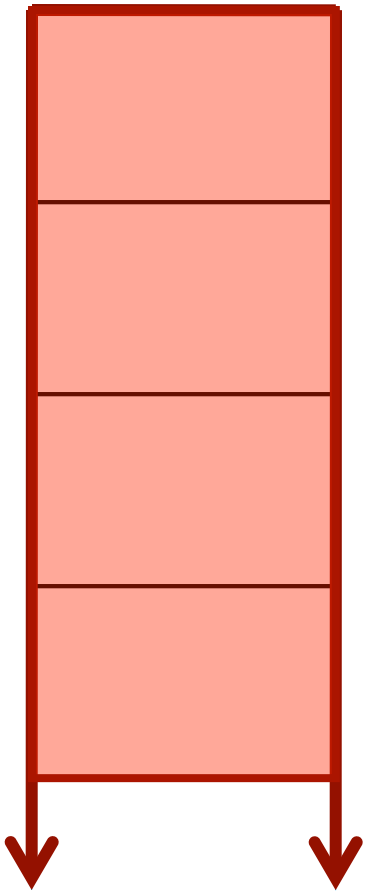
Frame of **main**

Frame of **f1**

Frame of **f2**

Functions and Stack Frames

```
main() {
   … call f1()
}

f1() {
   … call f2()
}

f2() {
   … call f3()
}

f3 () {
    …
}
```



Frame of **main**

Frame of **f1**

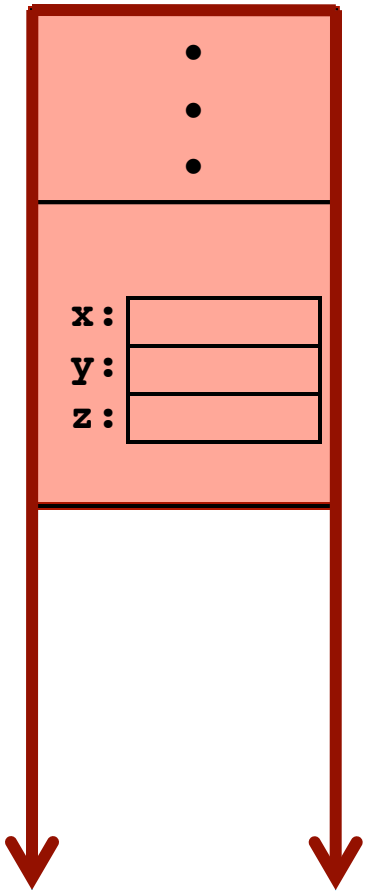
Frame of **f2**

Frame of **f3**

Functions and Stack Frames

```
f1() {
  int x,y,z;
  … call f2()
}
```

```
f2() {
  int a,b,c;
  … call f3()
}
```

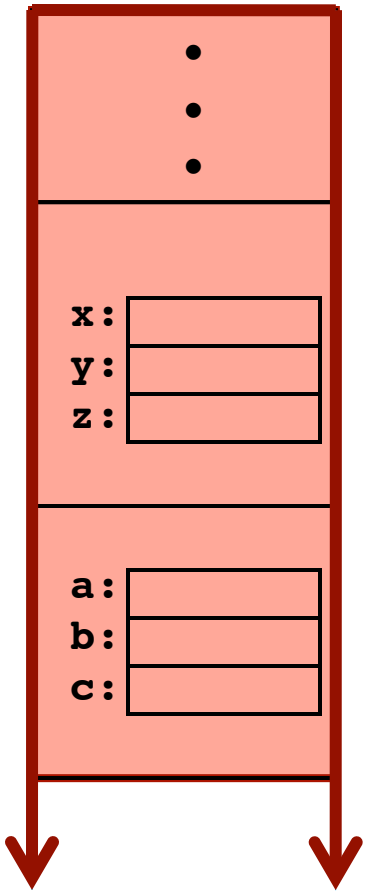


Frame of **f1**

Functions and Stack Frames

```
f1() {
   int x,y,z;
   … call f2()
}
```

```
f2() {
   int a,b,c;
   … call f3()
}
```



Frame of **f1**

Frame of **f2**

Where are the arguments???

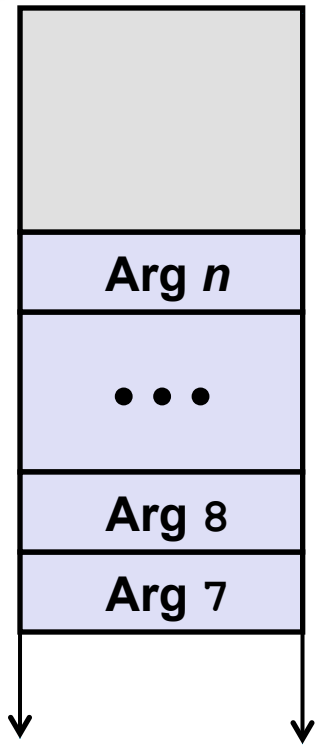
First 6 arguments
Registers

%rdi
%rsi
%rdx
%rcx
%r8
%r9

Return value

%rax

Stack



Only allocate stack space when needed

Passing Arguments

```
void multstore
  (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

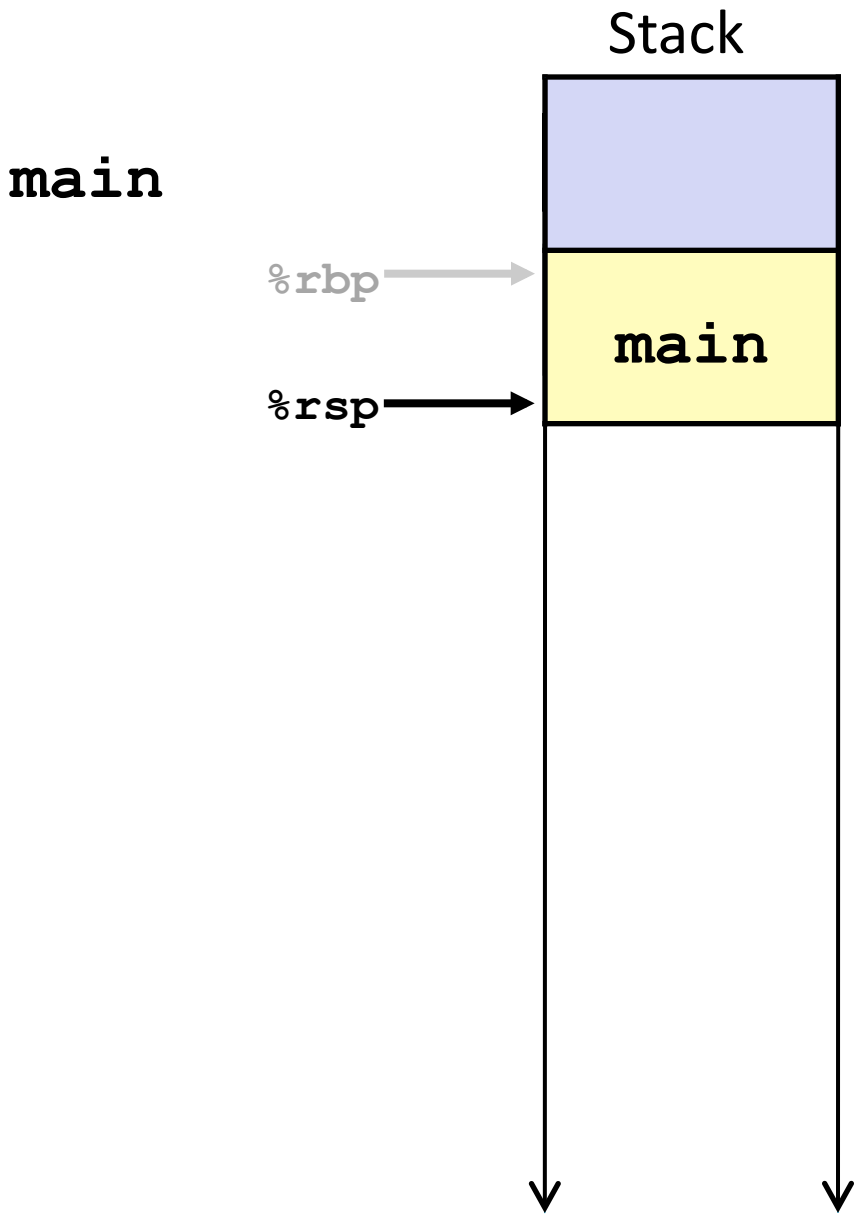
```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
  •••
  400541: mov    %rdx,%rbx          # Save dest
  400544: callq  400550 <mult2>     # mult2(x,y)
  # t in %rax
  400549: mov    %rax,(%rbx)        # Save at dest
  •••
```

```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550:  mov    %rdi,%rax   # a
  400553:  imul   %rsi,%rax   # a * b
  # s in %rax
  400557:  retq               # Return
```

Example

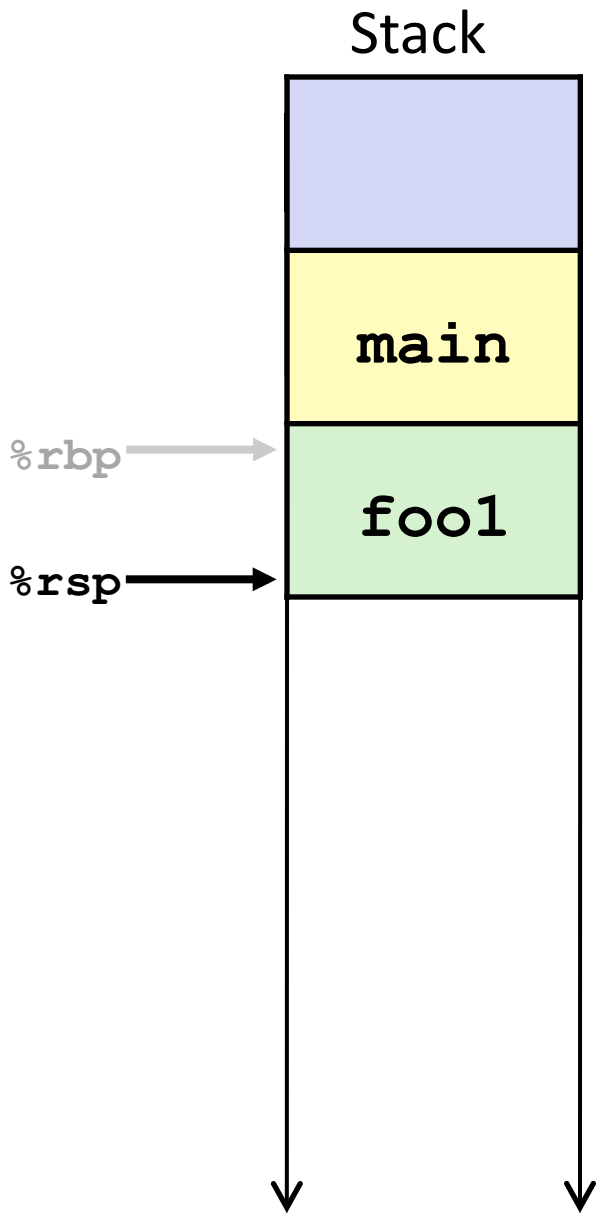
```
main(…)
{
   •
   •
   foo1();
   •
   •
}
```



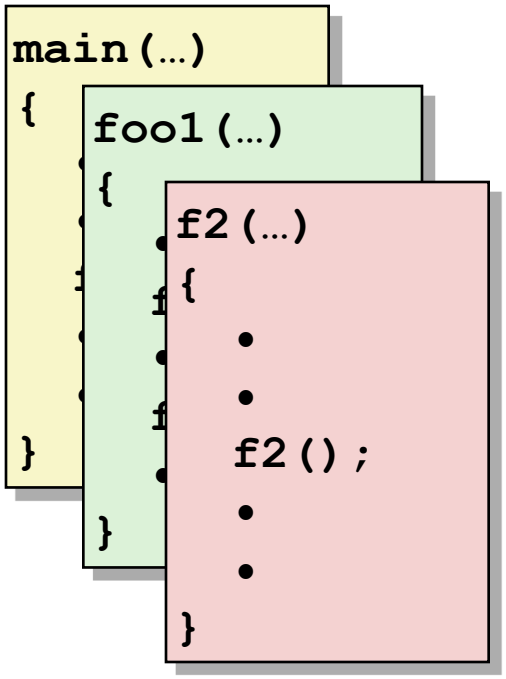
Example

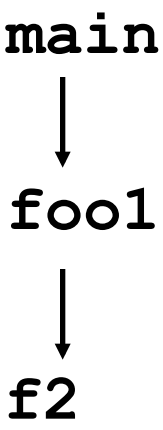
```
main(…)
{
    foo1(…)
    {
       • • •
       f2();
       • • •
       f2();
       • • •
    }
}
```

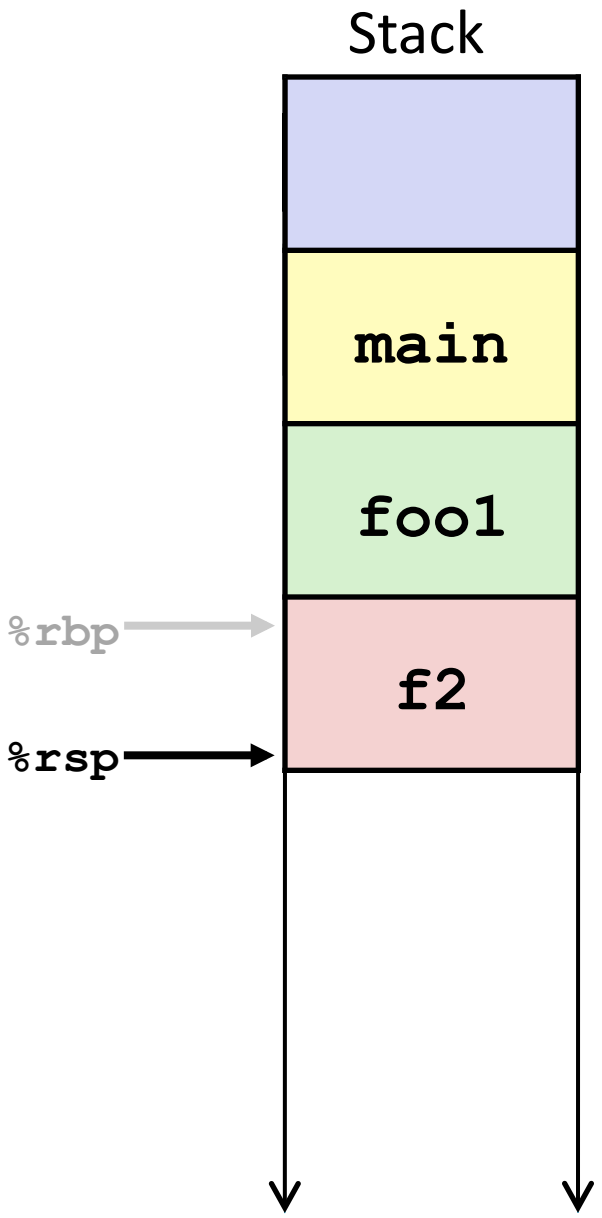
```
main
  |
  v
foo1
```



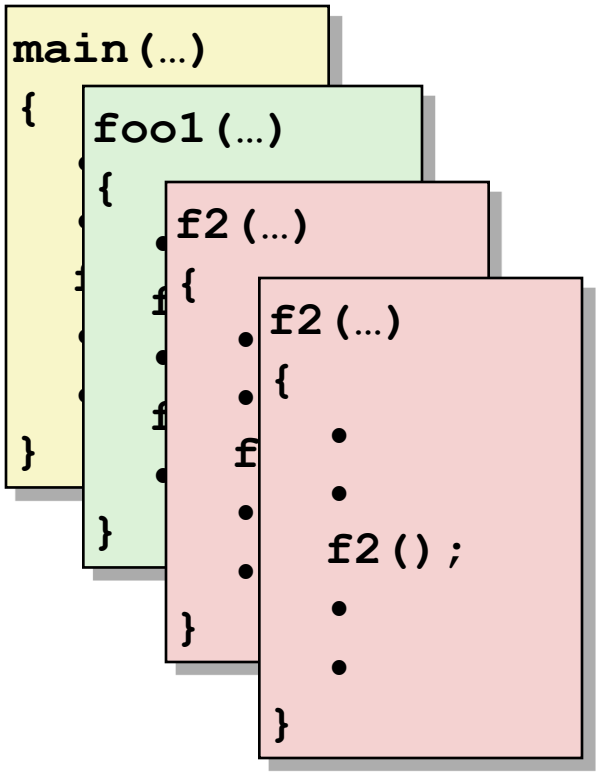
Example

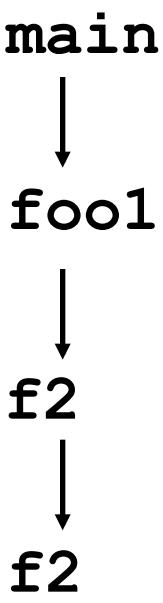


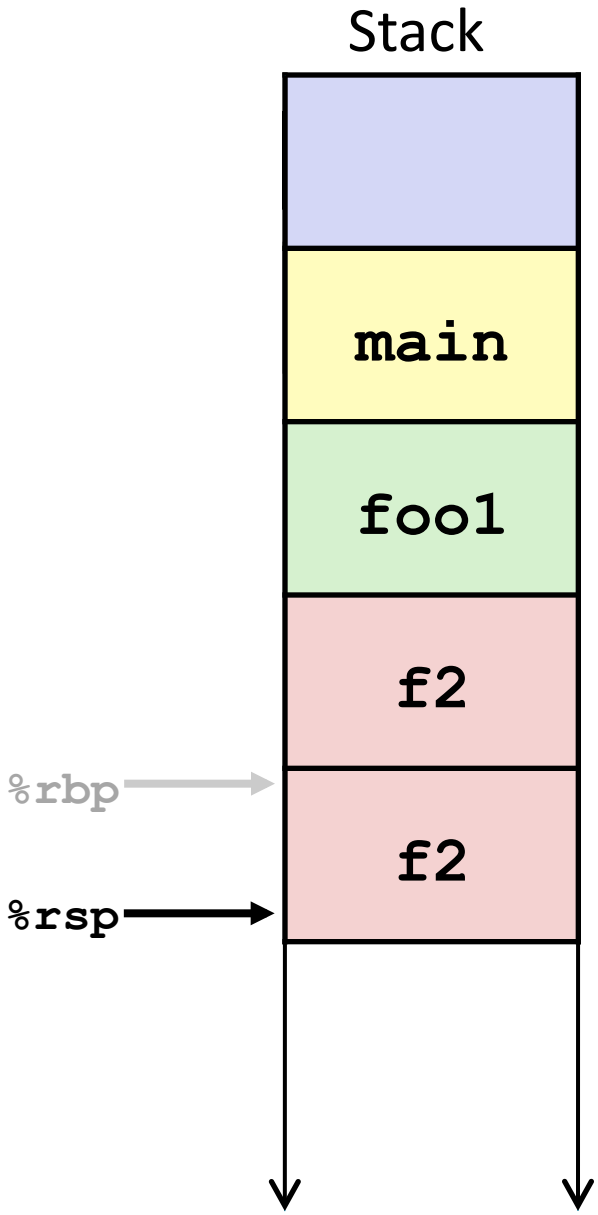




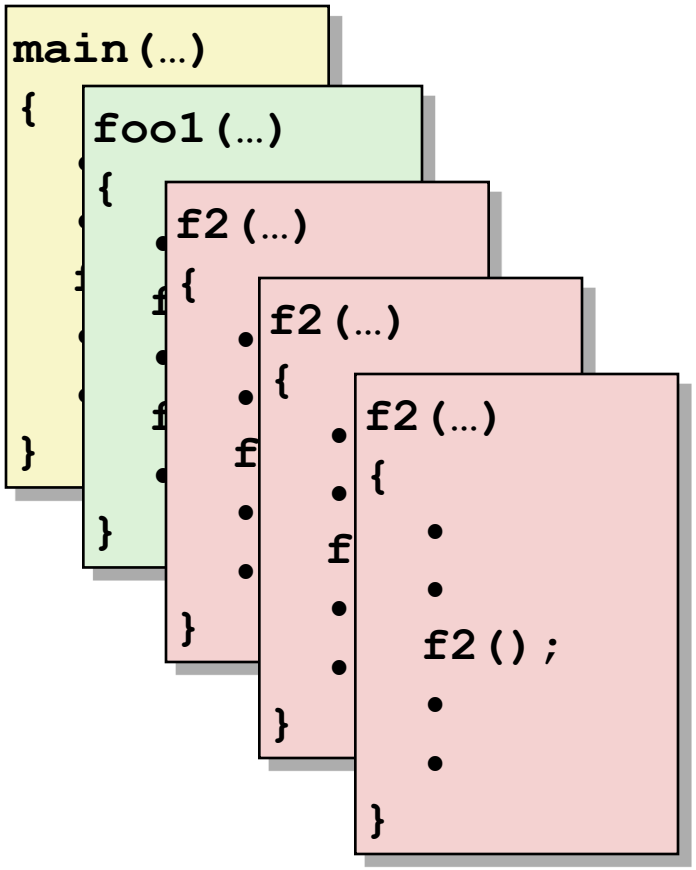
Example

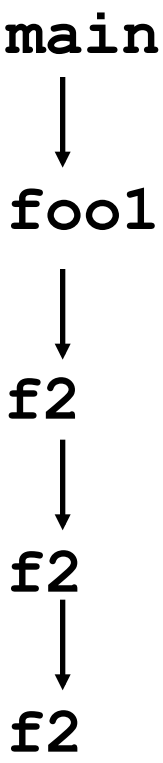


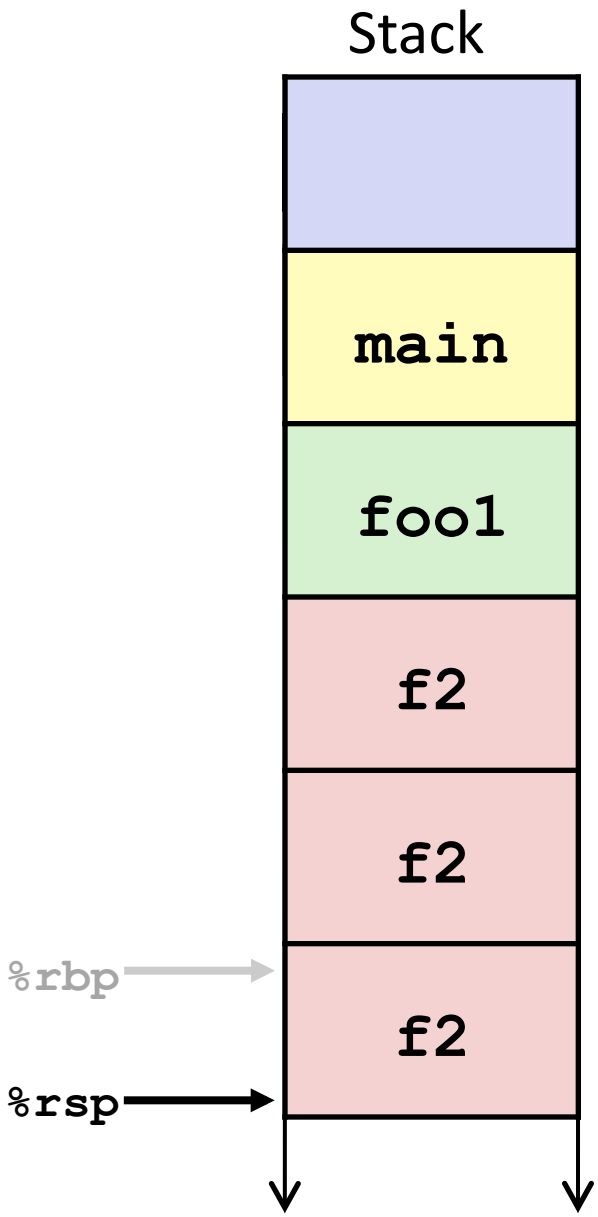




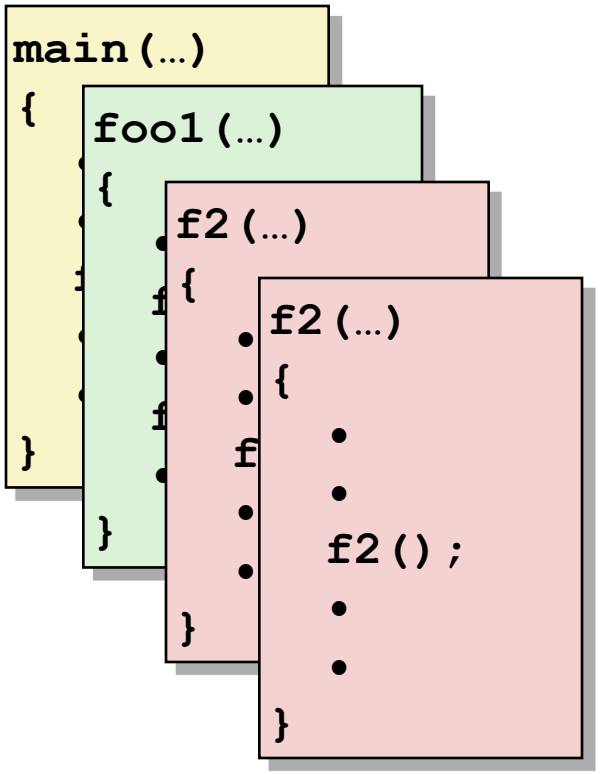
Example

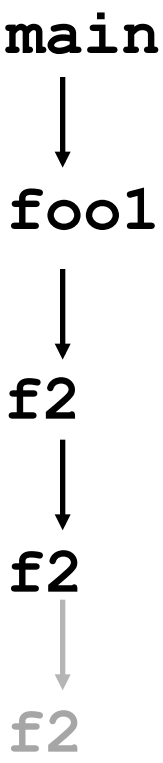

main(…)
{
foo1(…)
{
f2(…)
{
f2(…)
{
f2(…)
{
f2();
}

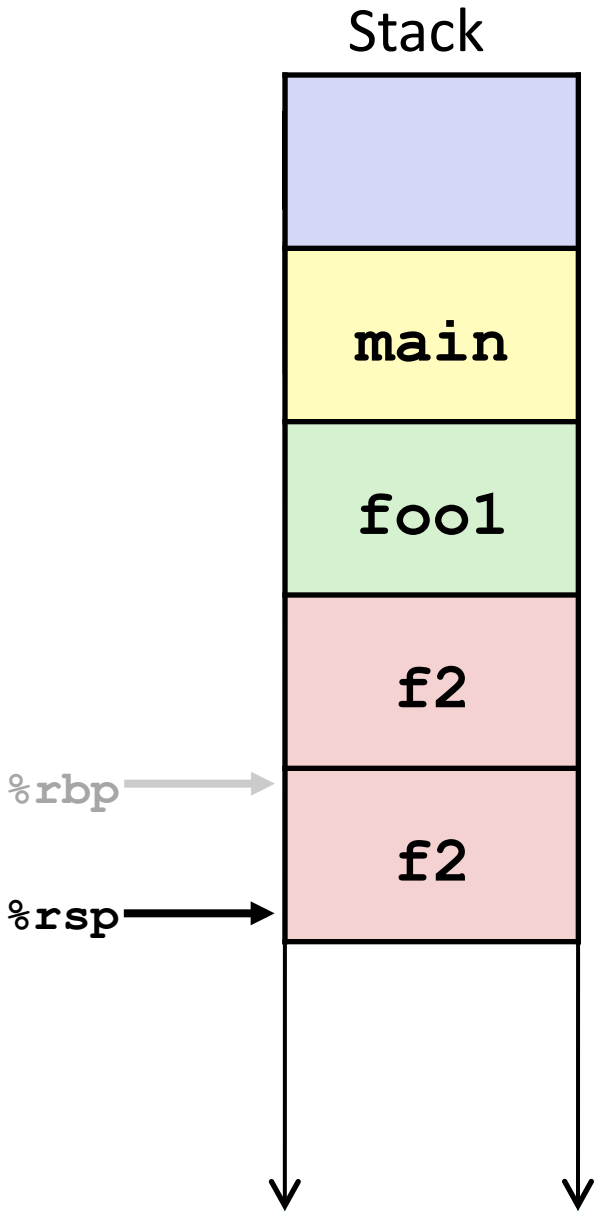

main
foo1
f2
f2
f2


Stack
main
foo1
f2
f2
f2
%rbp
%rsp

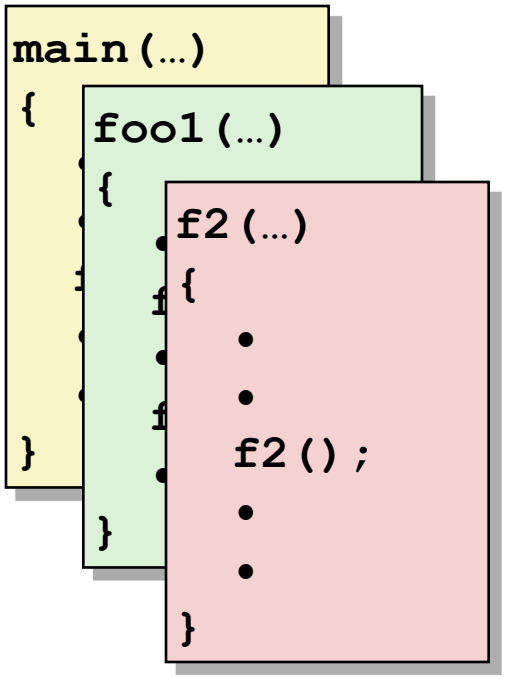
Example

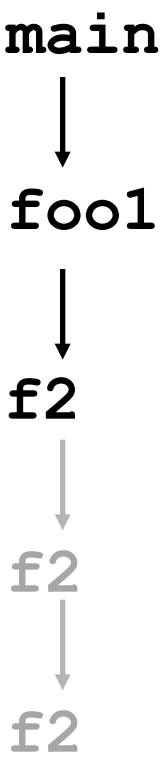

main(…)
{
foo1(…)
{
f2(…)
{
f2(…)
{
f2();
}

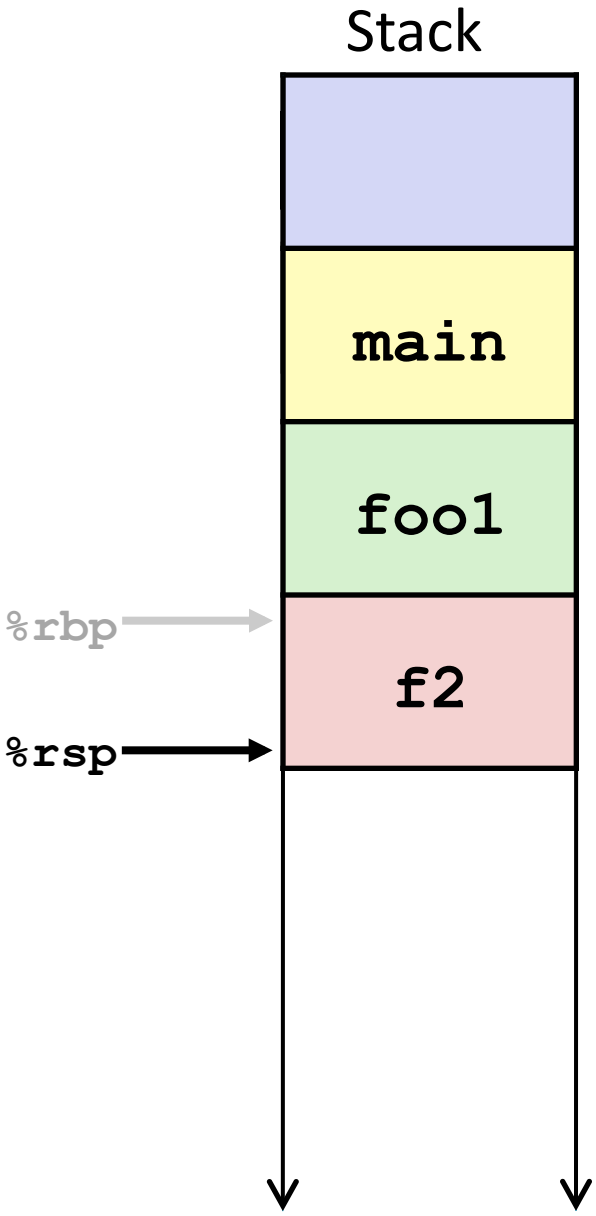

main
foo1
f2
f2
f2


Stack
main
foo1
f2
f2
%rbp
%rsp

Example

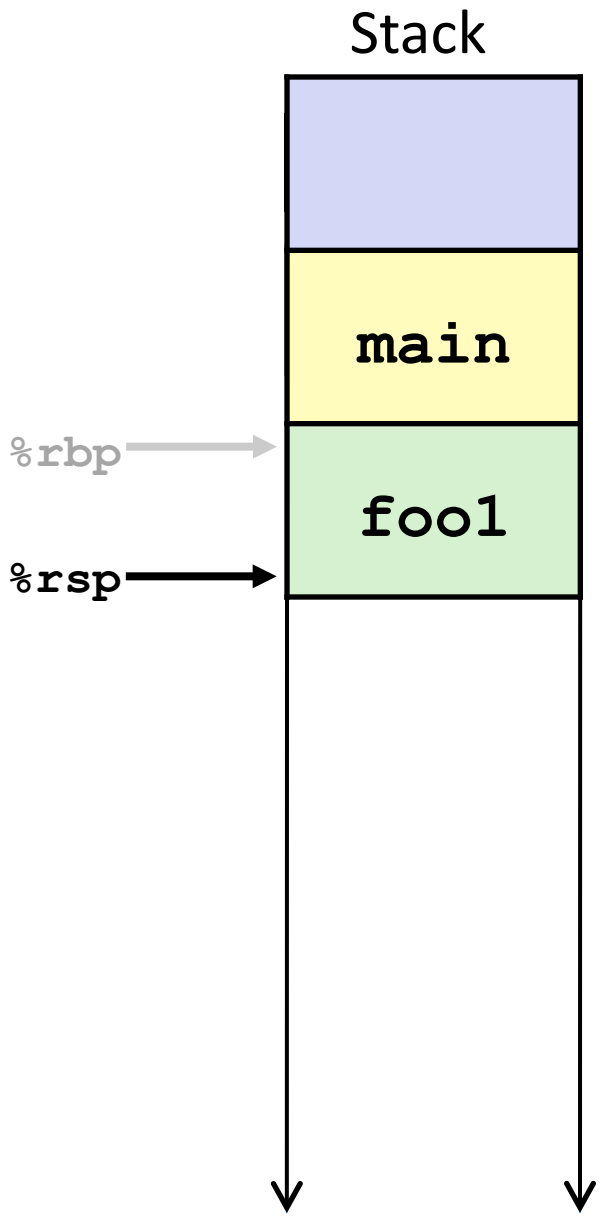




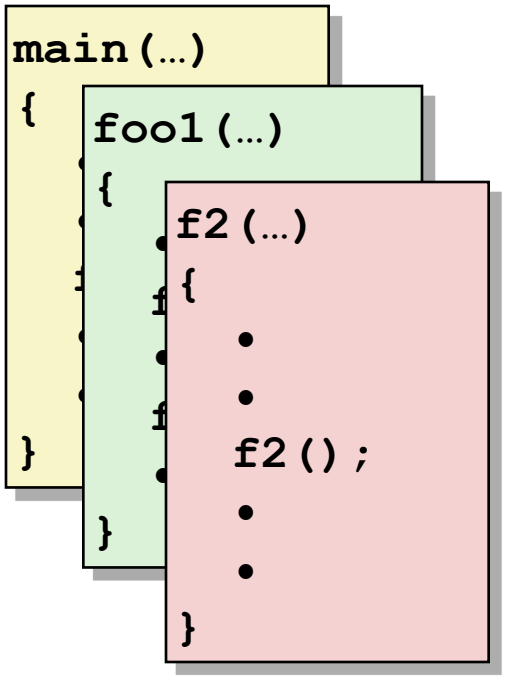


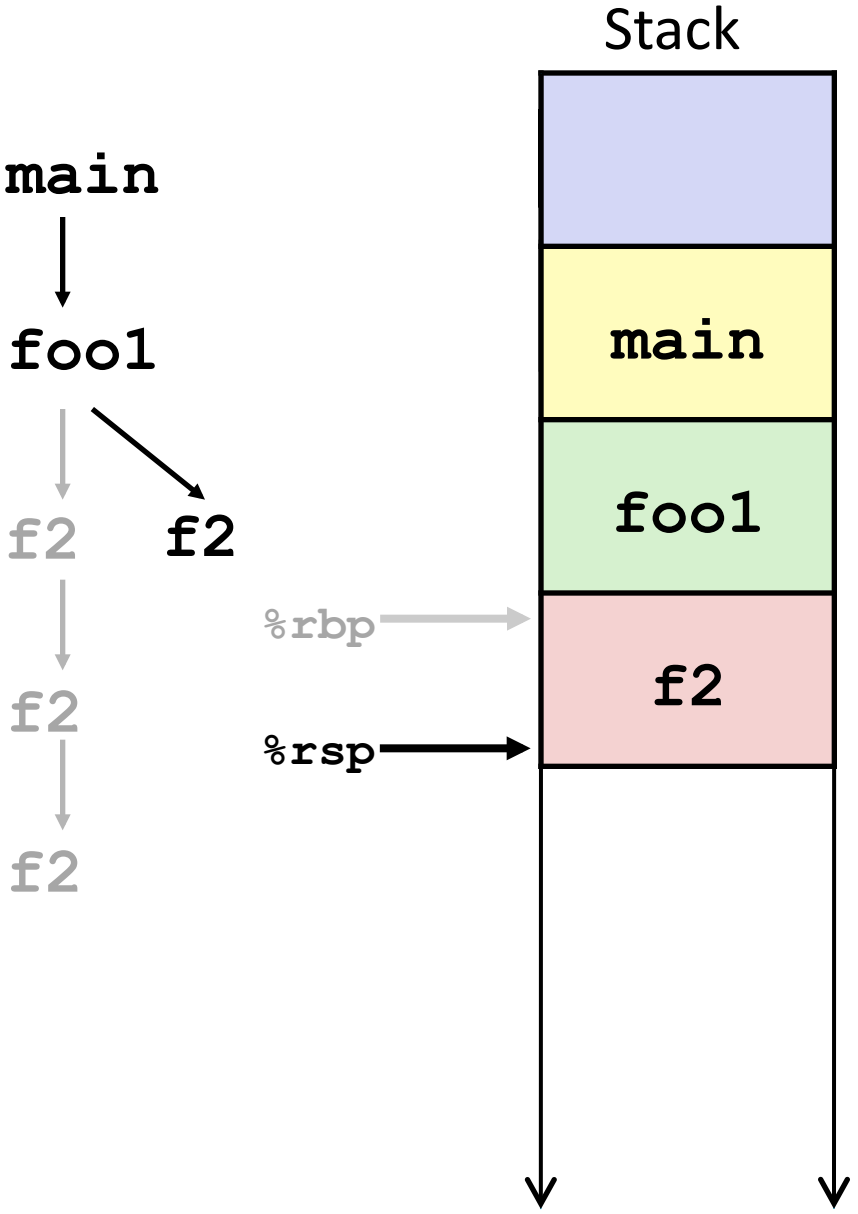
Example

```
main(…)
{
    foo1(…)
    {
      • • •
      f2();
      • • •
      f2();
      • • •
    }
}
```



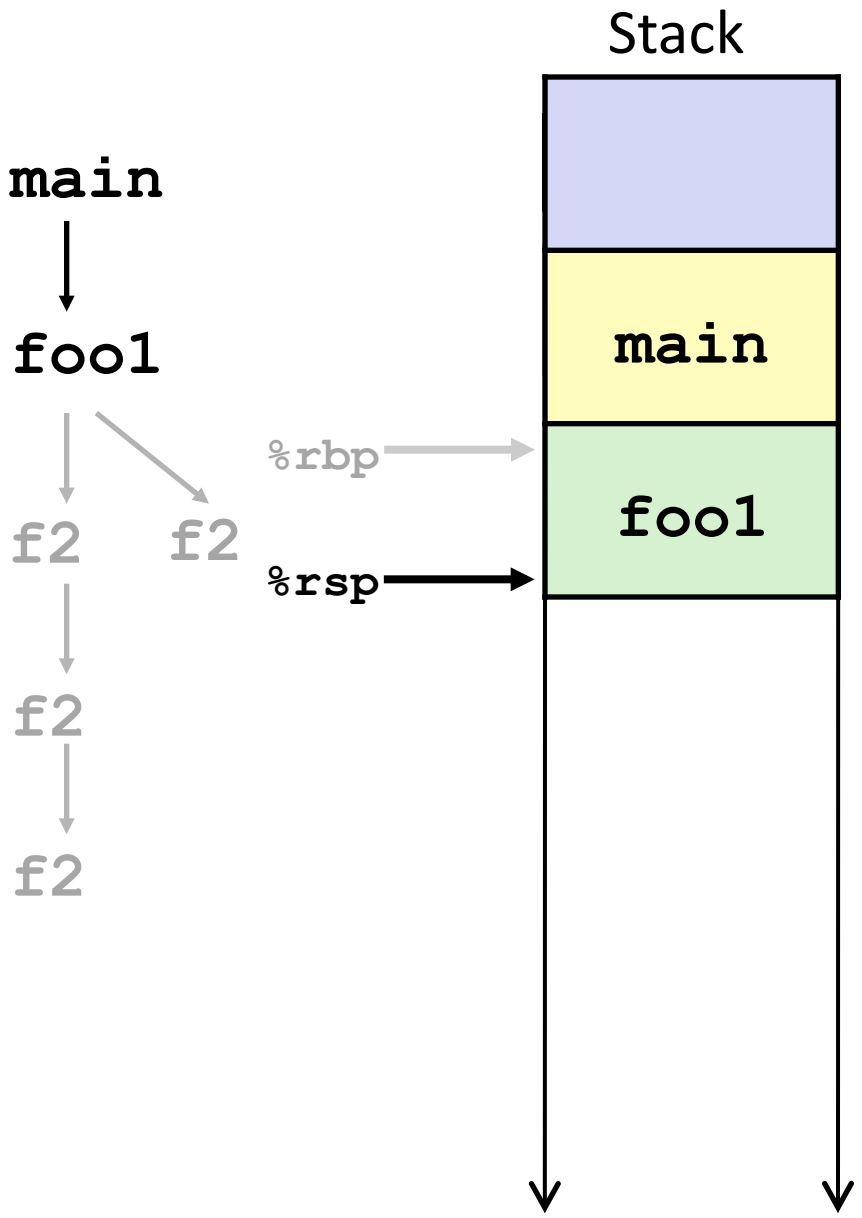
Example





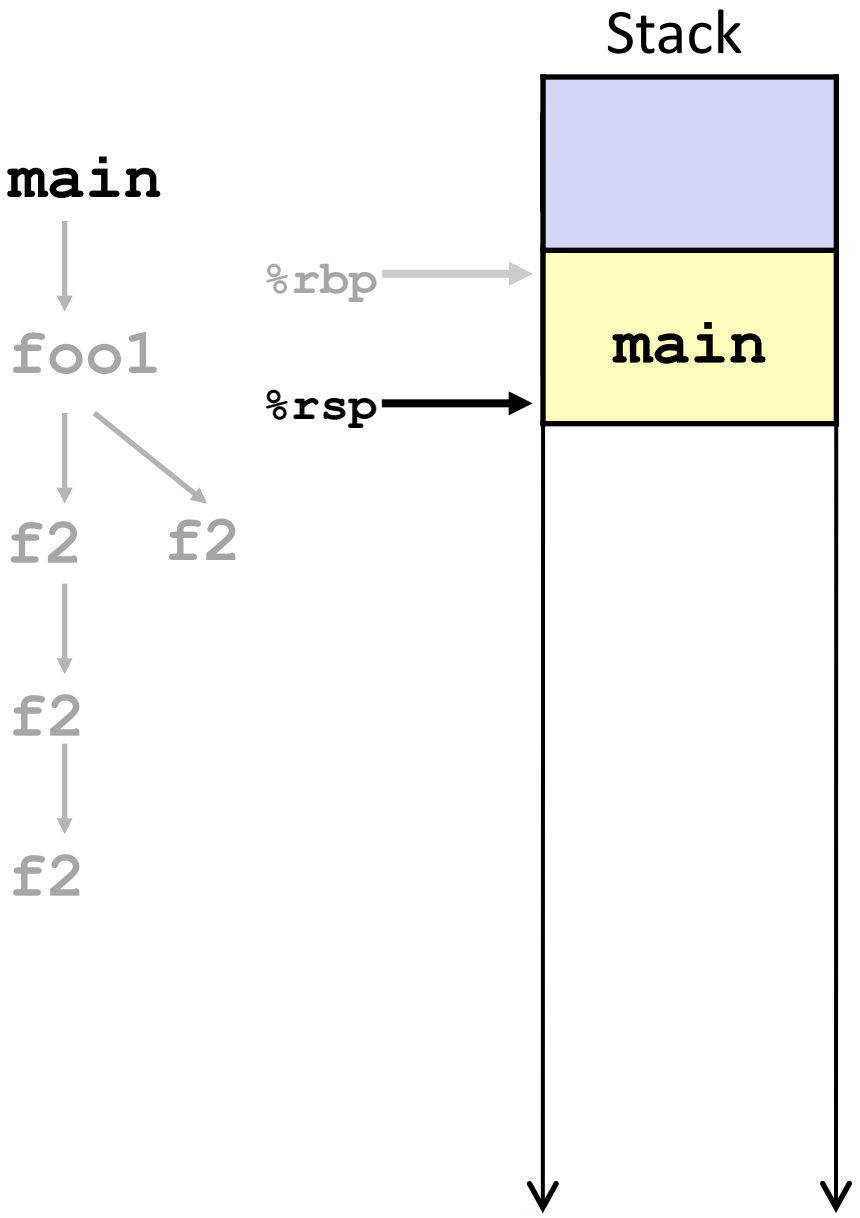
Example

```
main(…)
{
    foo1(…)
    {
       • • •
       f2();
       • • •
       f2();
       • • •
    }
}
```



Example

```
main(…)
{
   •
   •
   foo1();
   •
   •
}
```



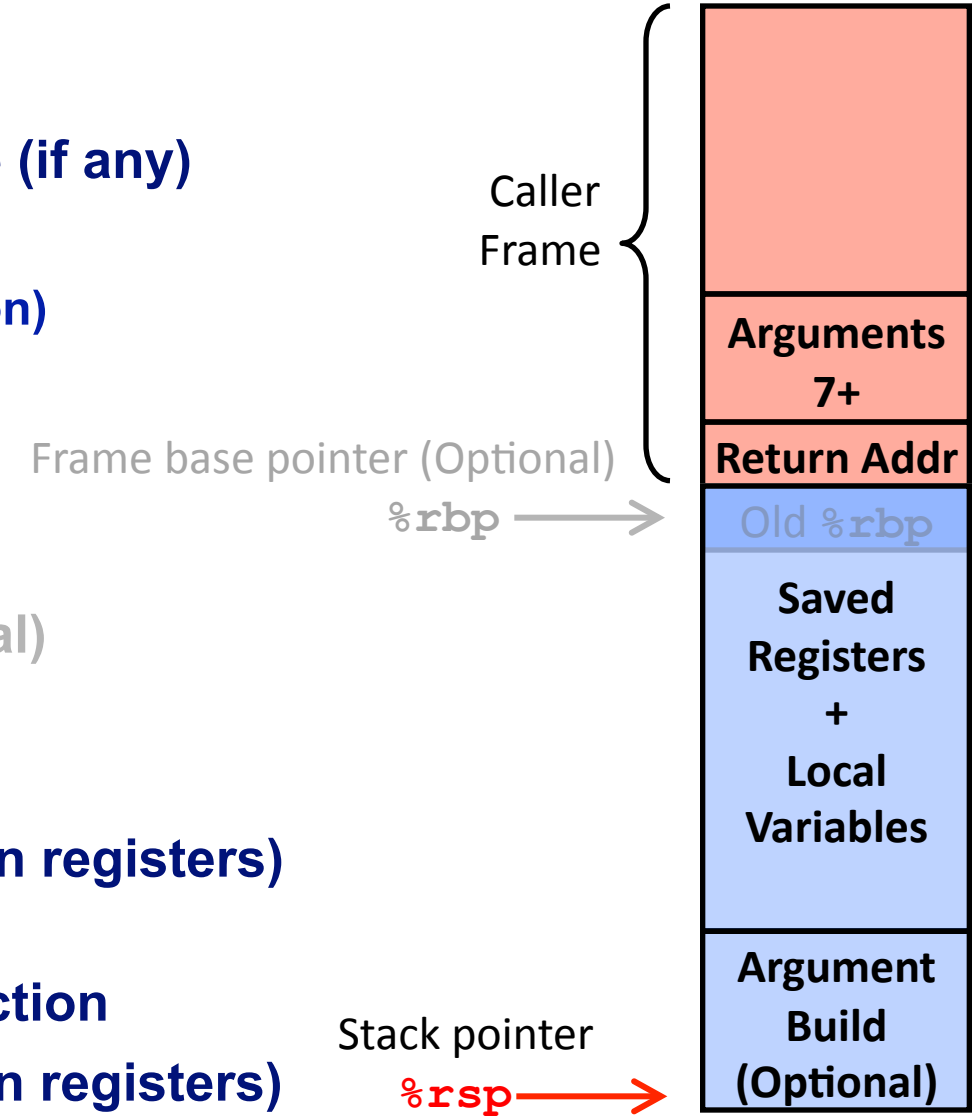
x86-64/Linux Stack Frames

Caller Stack Frame

- Extra Arguments to callee (if any)
- Return address
  (Pushed by `call` instruction)

Callee Stack Frame

- Old frame pointer (optional)
- Saved registers
- Local variables
  (if they can't be kept in registers)
- Argument build area
  to the next called function
  (if they can't be kept in registers)



Example: `incr`

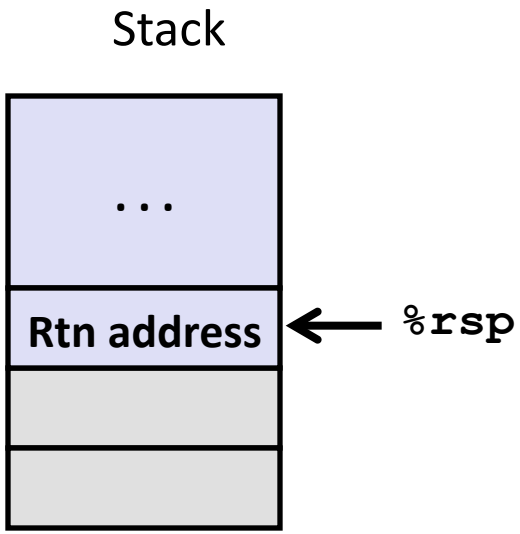
```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

Register	Use(s)
%rdi	Argument p
%rsi	Argument val, y
%rax	x, Return value

```
incr:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

Calling `incr`

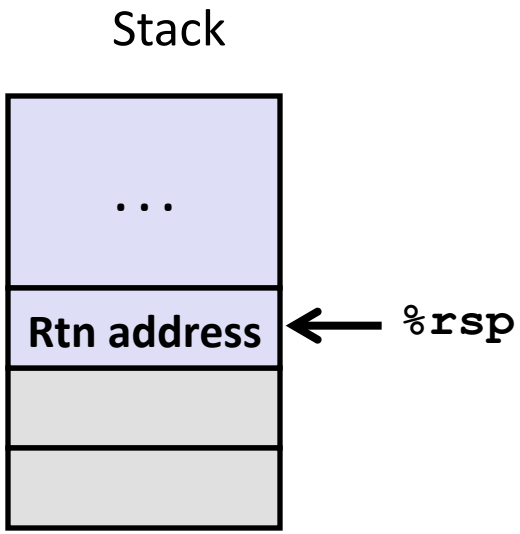
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```




```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Calling `incr`

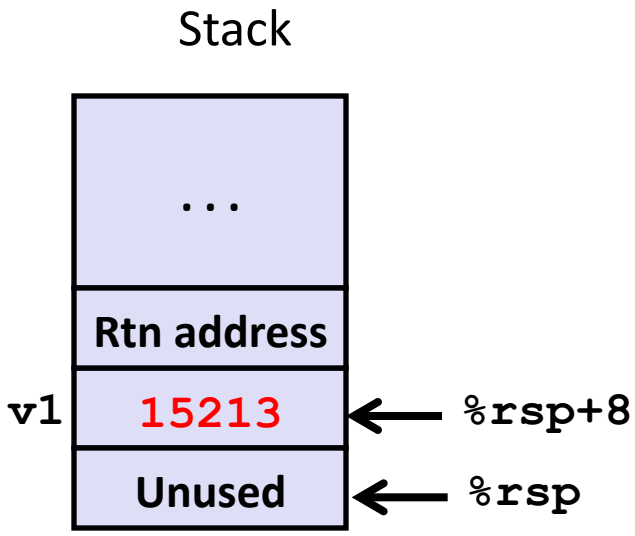
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```




```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Calling `incr`

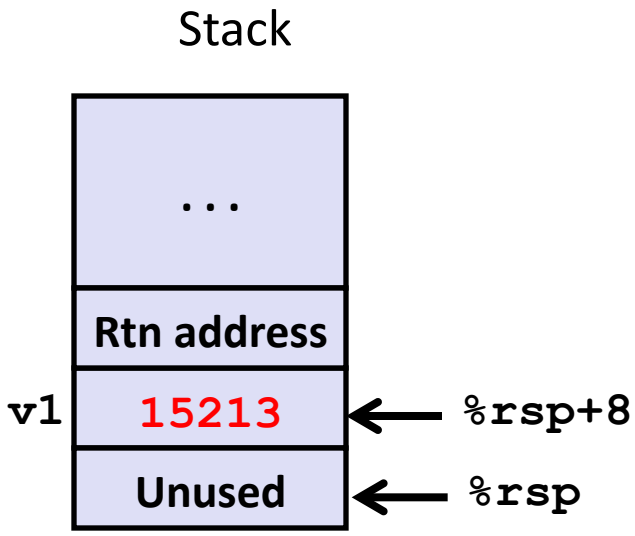
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```




```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Calling `incr`

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

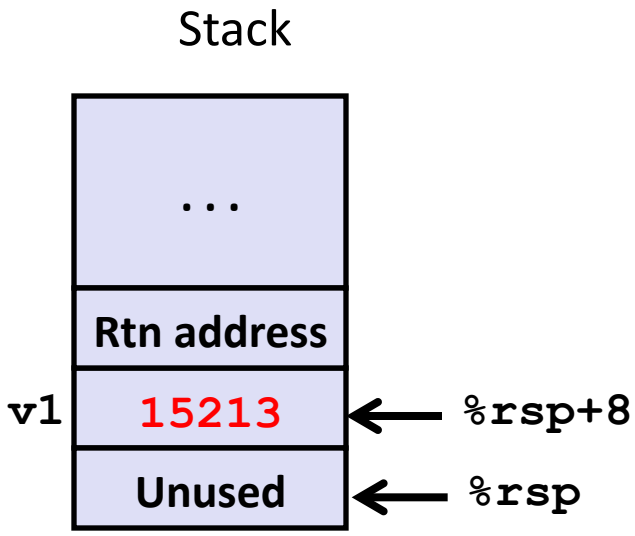



```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Register	Use(s)
%rdi	&v1
%rsi	3000

Calling `incr`

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

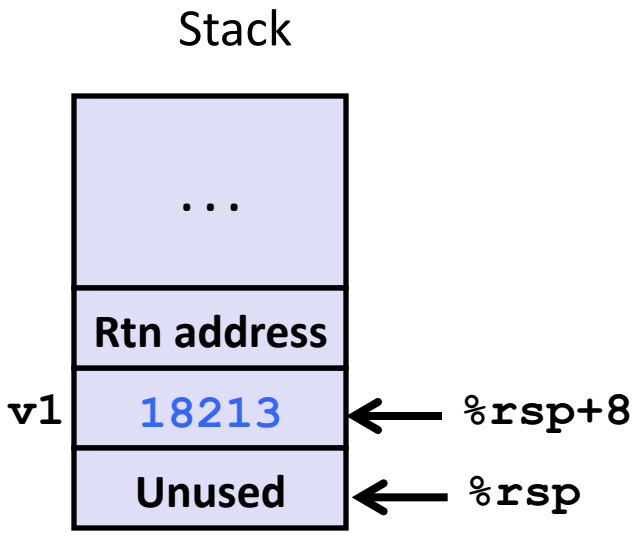



```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Register	Use(s)
%rdi	&v1
%rsi	3000

Calling `incr`

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

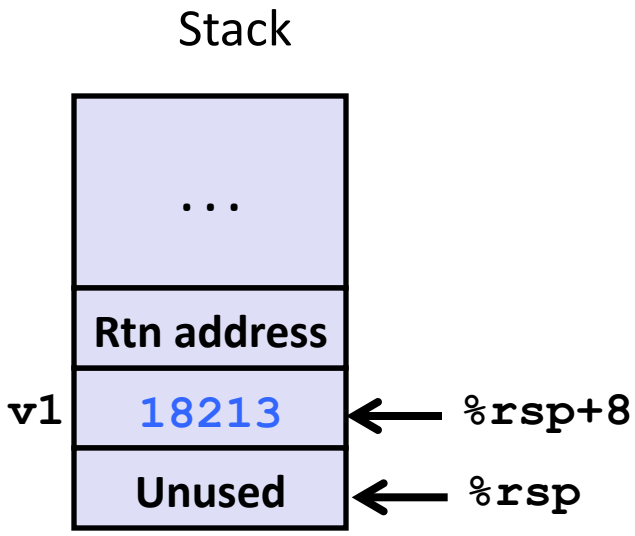



```
call_incr:
   subq     $16, %rsp
   movq     $15213, 8(%rsp)
   movl     $3000, %esi
   leaq     8(%rsp), %rdi
   call     incr
   addq     8(%rsp), %rax
   addq     $16, %rsp
   ret
```

Register	Use(s)
%rdi	&v1
%rsi	3000

Calling `incr`

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

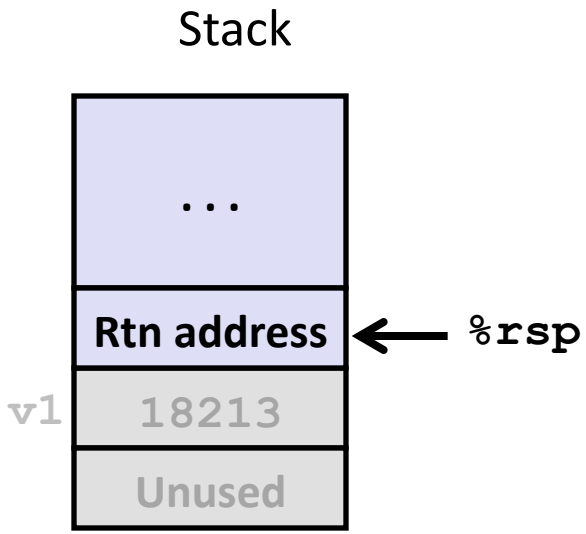



```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Register	Use(s)
%rax	Return value

Calling `incr`

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

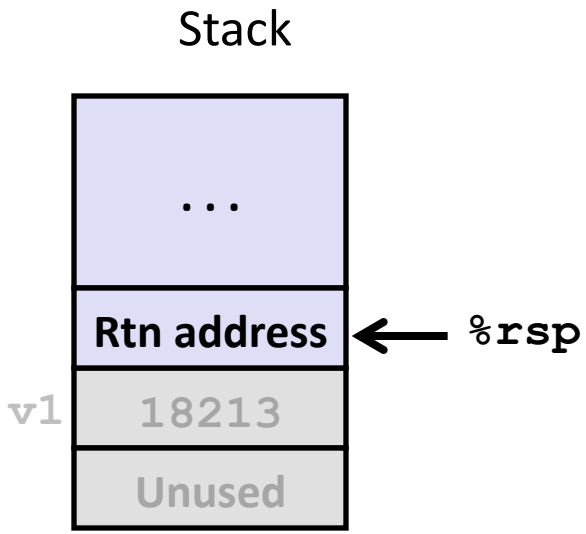



```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Register	Use(s)
%rax	Return value

Calling `incr`

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

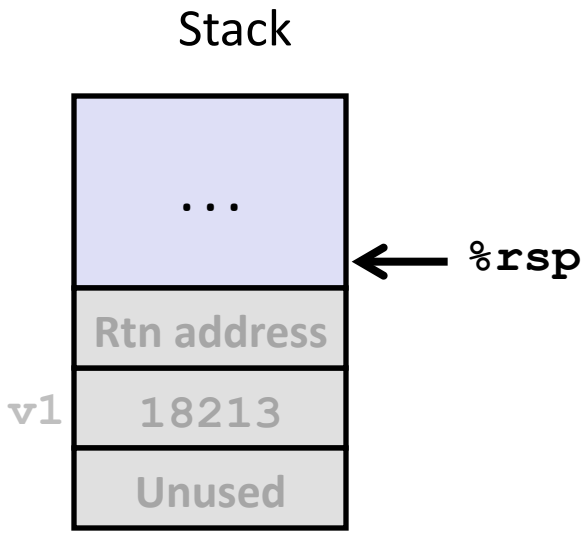



```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Register	Use(s)
%rax	Return value

Calling `incr`

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```




```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Register	Use(s)
%rax	Return value

Register Saving Conventions

When procedure `yoo` calls `who`:

- `yoo` is the caller
- `who` is the callee

Can register be used for temporary storage?

```
yoo:
     • • •
    movq   $15213, %rdx
    call   who
    addq   %rdx, %rax
     • • •
    ret
```

```
who:
     • • •
    subq   $18213, %rdx
     • • •
    ret
```

Contents of register `%rdx` overwritten by `who`

This could be trouble → We need coordination!

Register Saving Conventions

When procedure `yoo` calls `who`:

- `yoo` is the caller
- `who` is the callee

Can register be used for temporary storage?

Conventions

- "Caller Saved"
  - Caller saves temporary values in its frame before the call
- "Callee Saved"
  - Callee saves temporary values in its frame before using
  - Callee restores them before returning to caller

x86-64 Linux Register Conventions

Caller-saved

Can be modified by the function

%rax

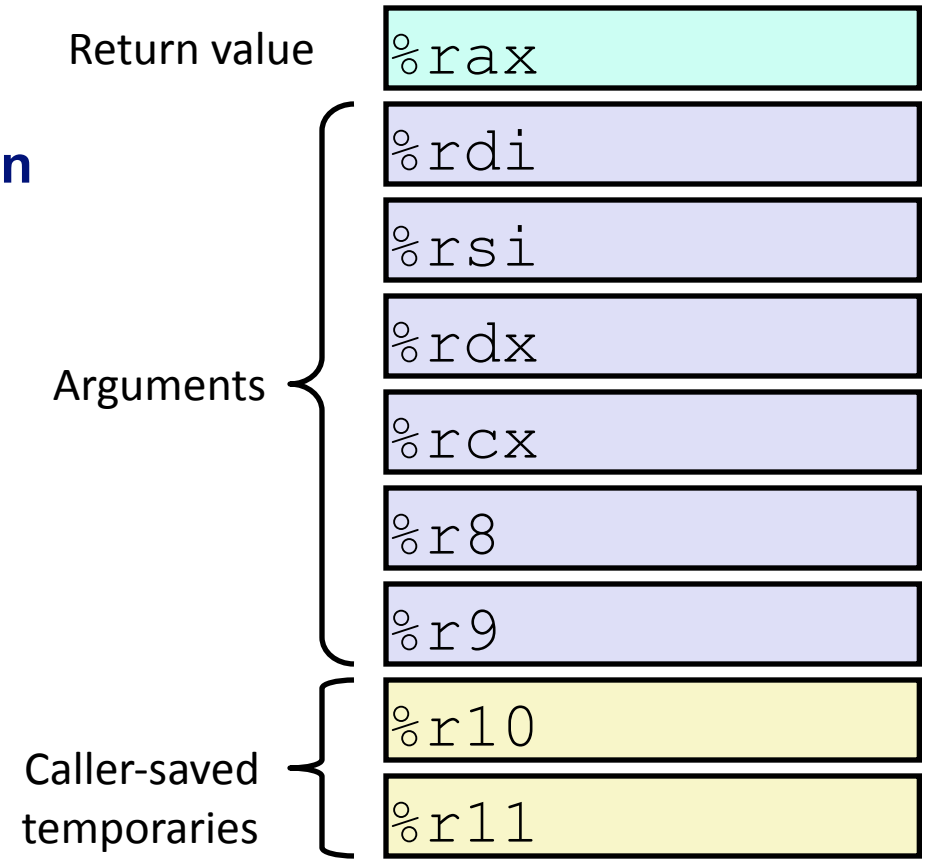
Return value

%rdi, ..., %r9

Arguments

%r10, %r11

temp "work" regs



x86-64 Linux Register Conventions

Callee-saved

Function must save & restore

`%rbx`, `%r12`, `%r13`, `%r14`

`%rbp`

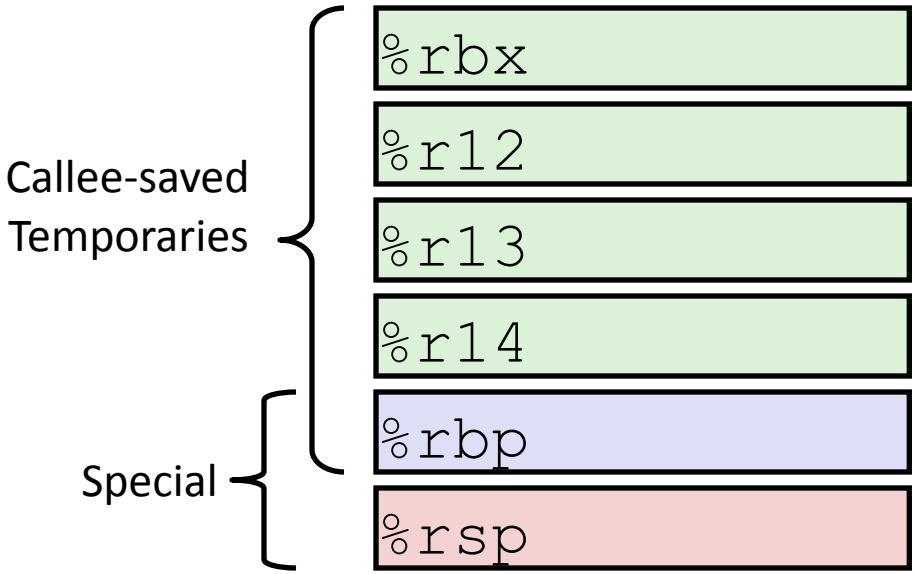
May be used as frame pointer

Can mix & match

`%rsp`

Special form of callee save

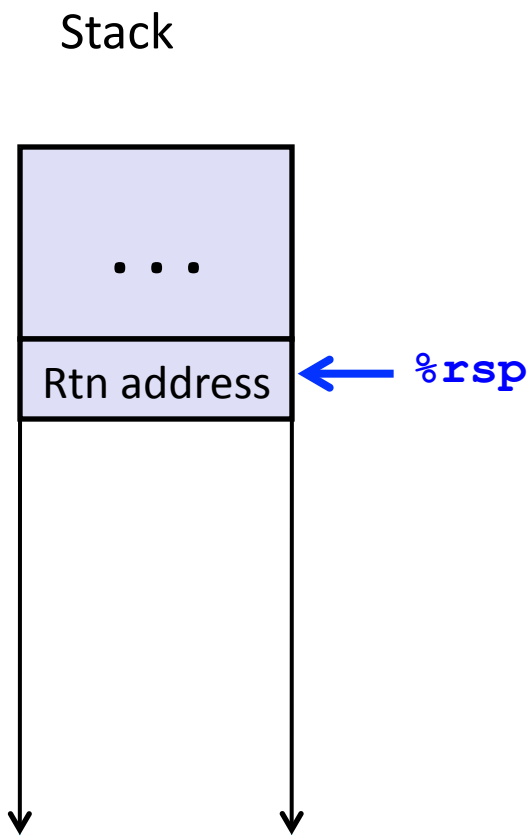
Restored to original value upon exit from procedure



Callee-Saved Example

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

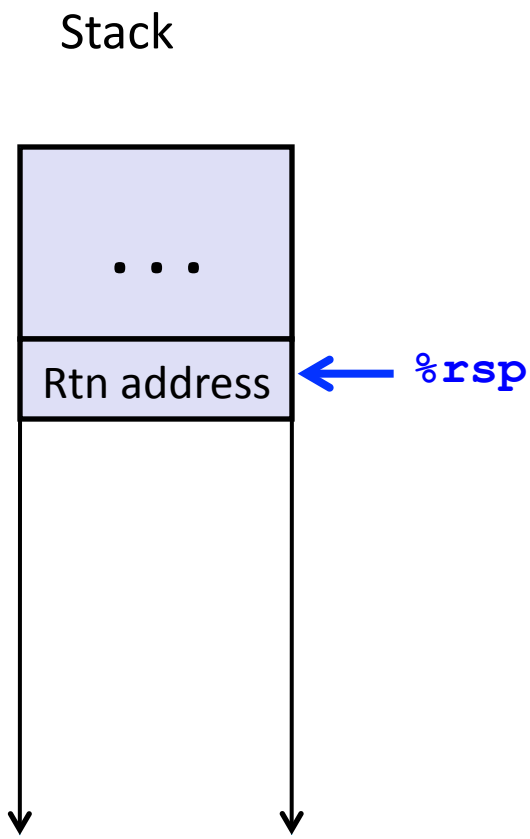
```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```



Callee-Saved Example

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

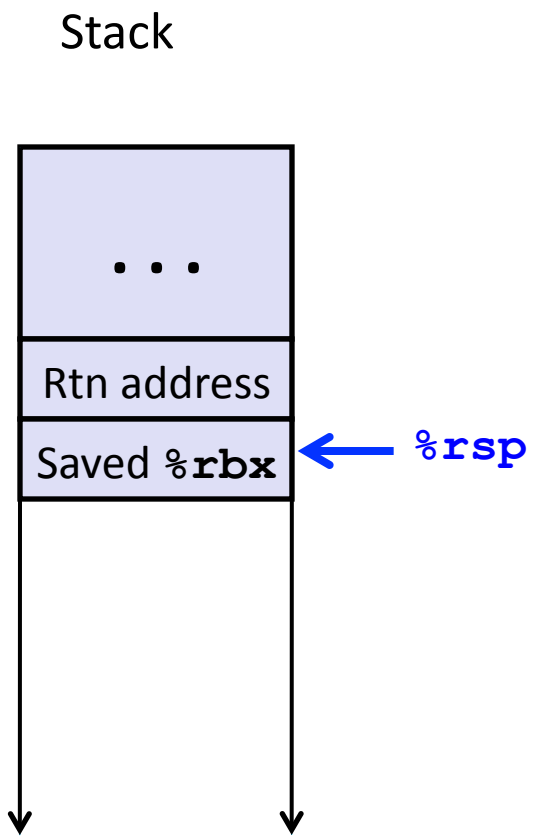
```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```



Callee-Saved Example

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

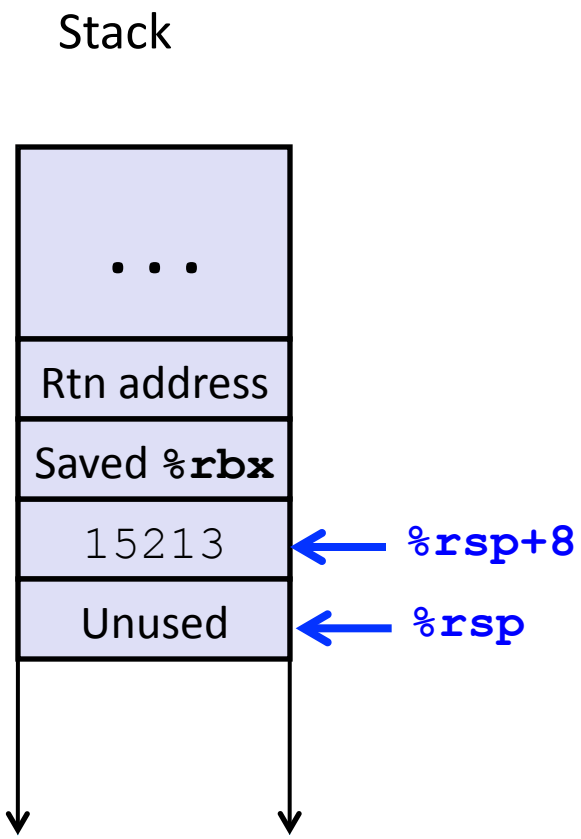
```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```



Callee-Saved Example

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

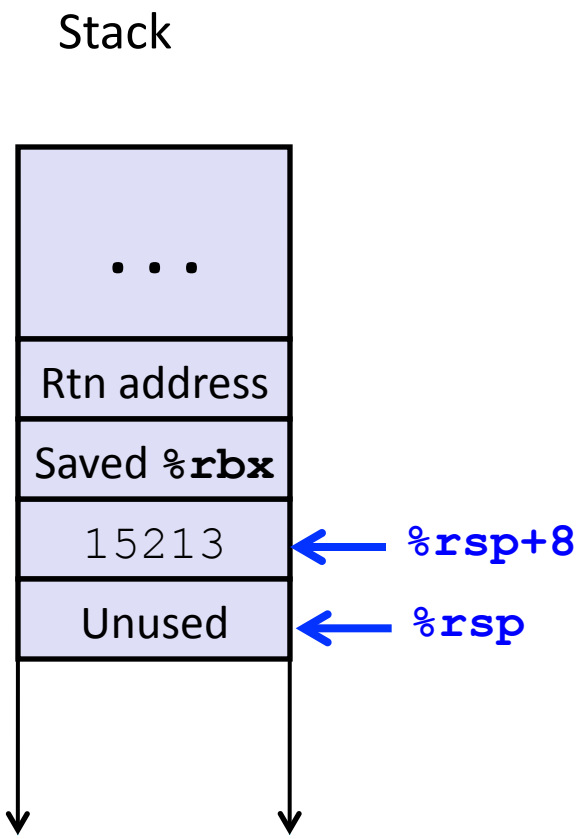
```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```



Callee-Saved Example

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

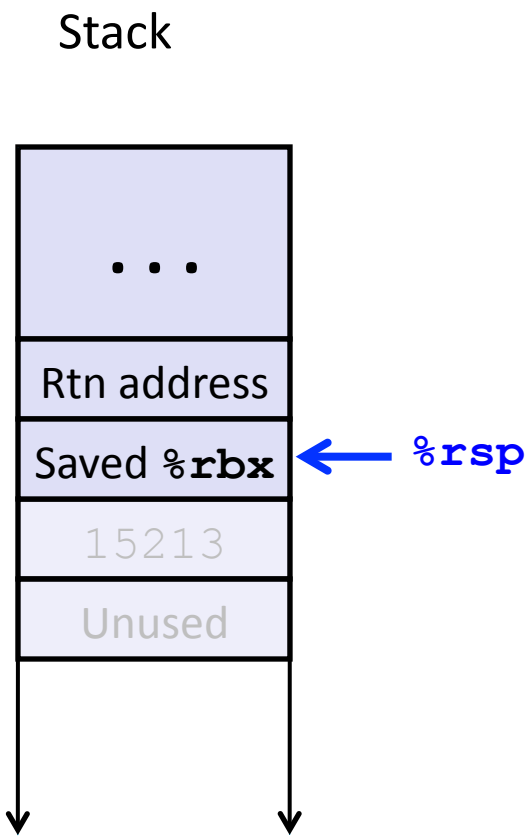
```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```



Callee-Saved Example

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

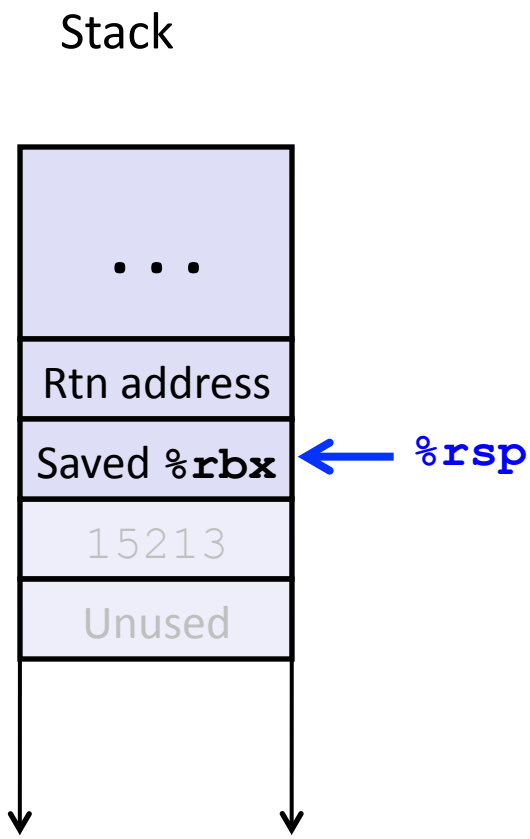
```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```



Callee-Saved Example

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

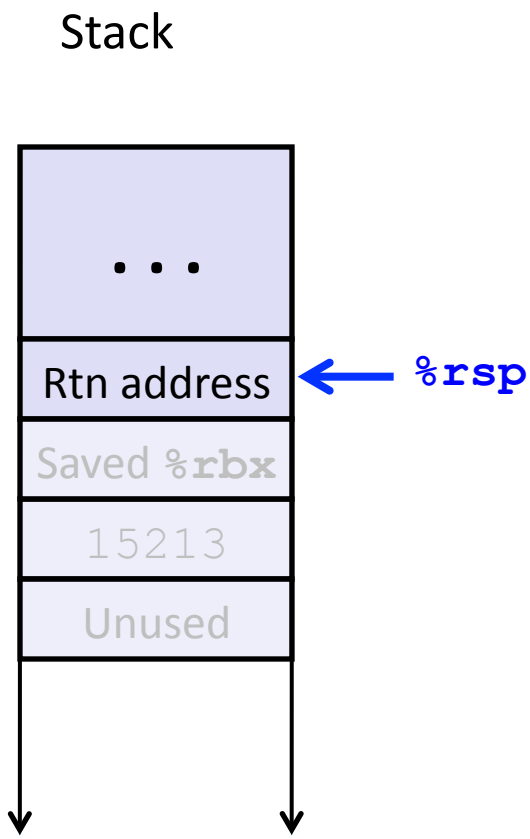
```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```



Callee-Saved Example

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```



Recursive Function

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Recursive Function Terminal Case

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

Recursive Function Register Save

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rdi	x	Argument

Recursive Function Call Setup

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rdi	x >> 1	Argument to recursive call
%rbx	x & 1	Callee-saved

Recursive Function Call

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Recursive call return value	

Recursive Function Result

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

Recursive Function Completion

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rax	Return value	Return value

Observations About Recursion

Handled Without Special Consideration

- Stack frames mean that each function call has private storage
  - Saved registers & local variables
  - Saved return pointer
- Register saving conventions prevent one function call from corrupting another's data
  - Unless the C code explicitly does so (e.g., buffer overflow)
- Stack discipline follows call / return pattern
  - If P calls Q, then Q returns before P
  - Last-In, First-Out

Also works for mutual recursion

- P calls Q; Q calls P

x86-64 Procedure Summary

Important Points

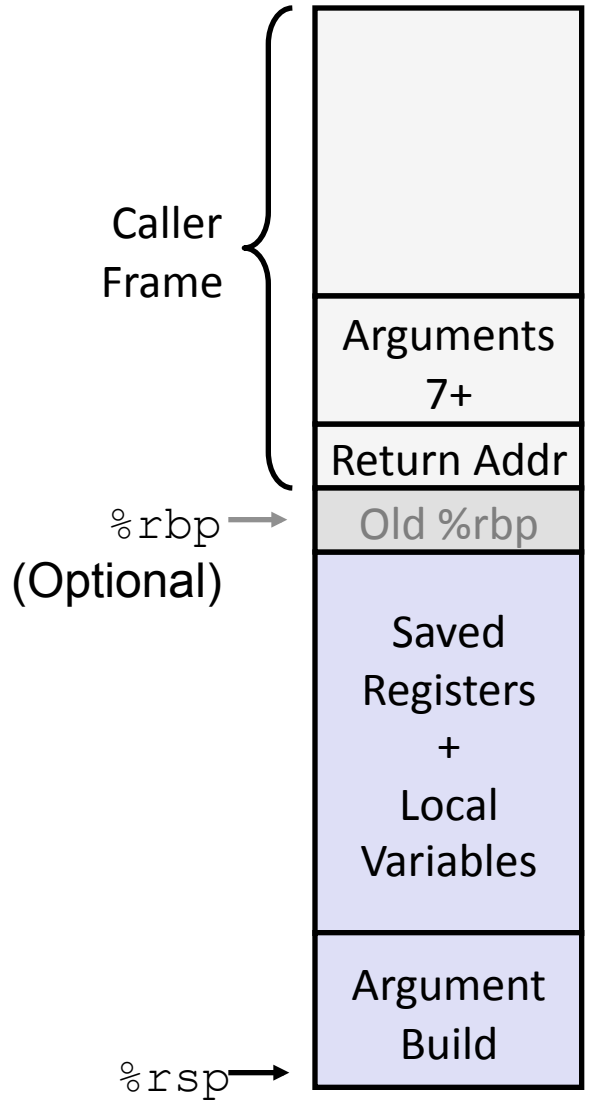
- Stack is the right data structure for procedure call / return
  - If P calls Q, then Q returns before P

Recursion (& mutual recursion) handled by normal calling conventions

- Can safely store values in local stack frame and in callee-saved registers
- Put function arguments at top of stack
- Result return in `%rax`

Pointers are addresses of values

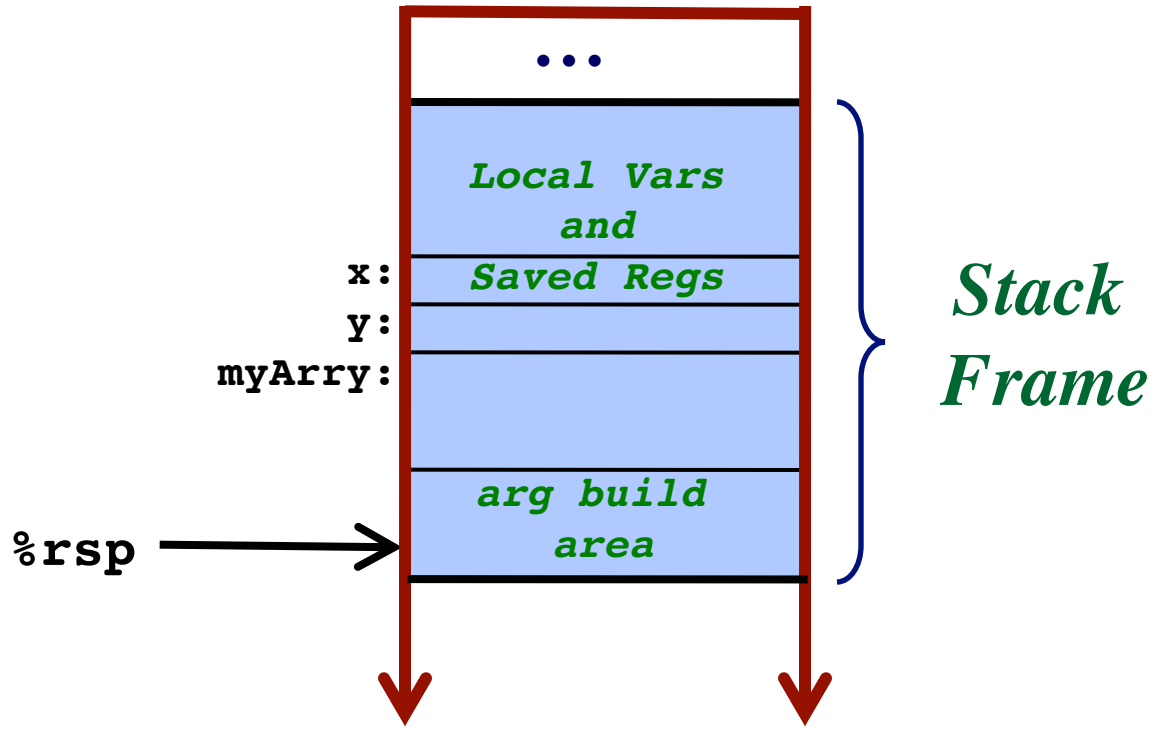
- On stack or global



Dynamically-Sized Stack Frames

```
f2(int n, …) {
   char myArry[n];
   …
}
```

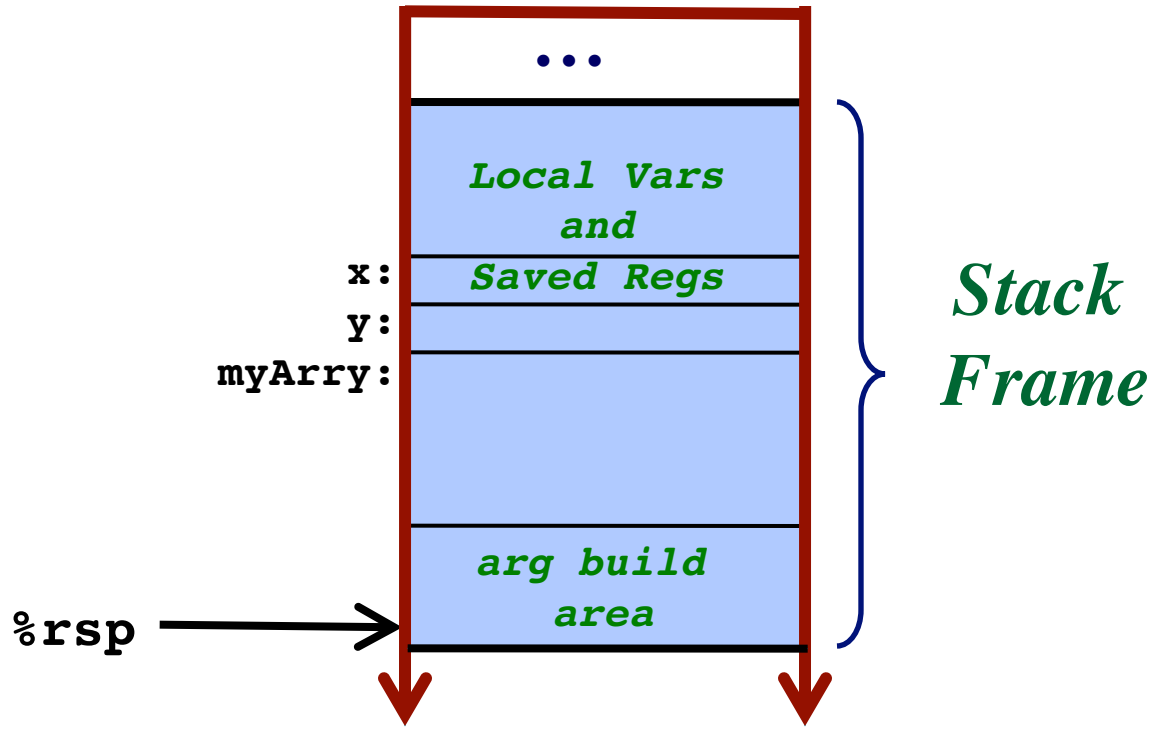
Local variables go into the frame.
But we don't know how big they are!
`%rsp` will be adjusted by ???
How to access things in the frame?



Dynamically-Sized Stack Frames

```
f2(int n, …) {
   char myArry[n];
   …
}
```

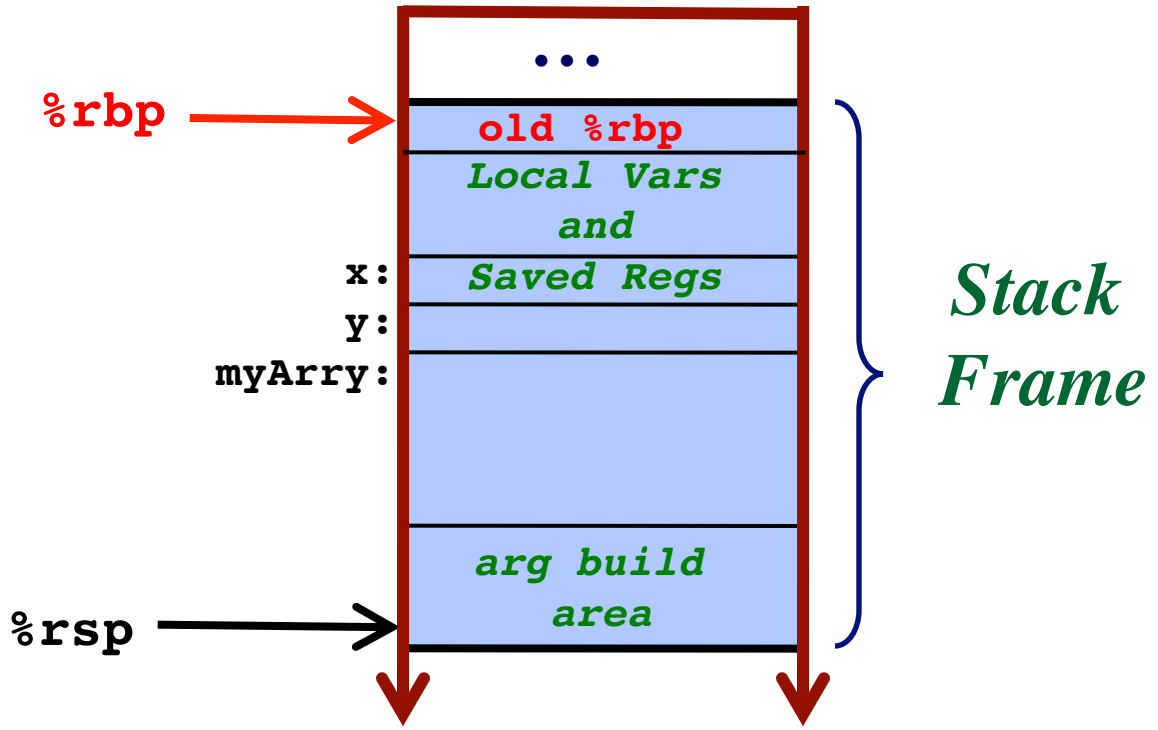
Local variables go into the frame.
But we don't know how big they are!
`%rsp` will be adjusted by ???
How to access things in the frame?



Dynamically-Sized Stack Frames

```
f2(int n, …) {
   char myArry[n];
   …
}
```

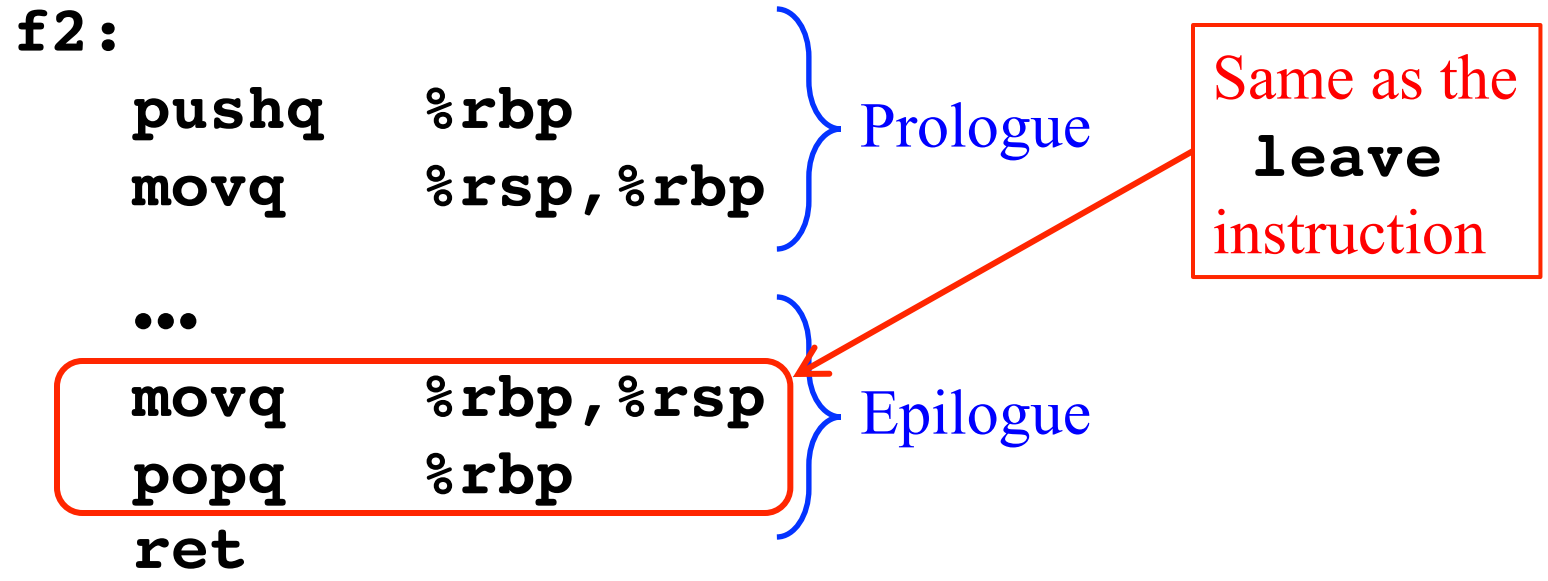
Local variables go into the frame.
But we don't know how big they are!
%rsp will be adjusted by ???
How to access things in the frame?
Use a frame Base Pointer: **%rbp**



Dynamically-Sized Stack Frames

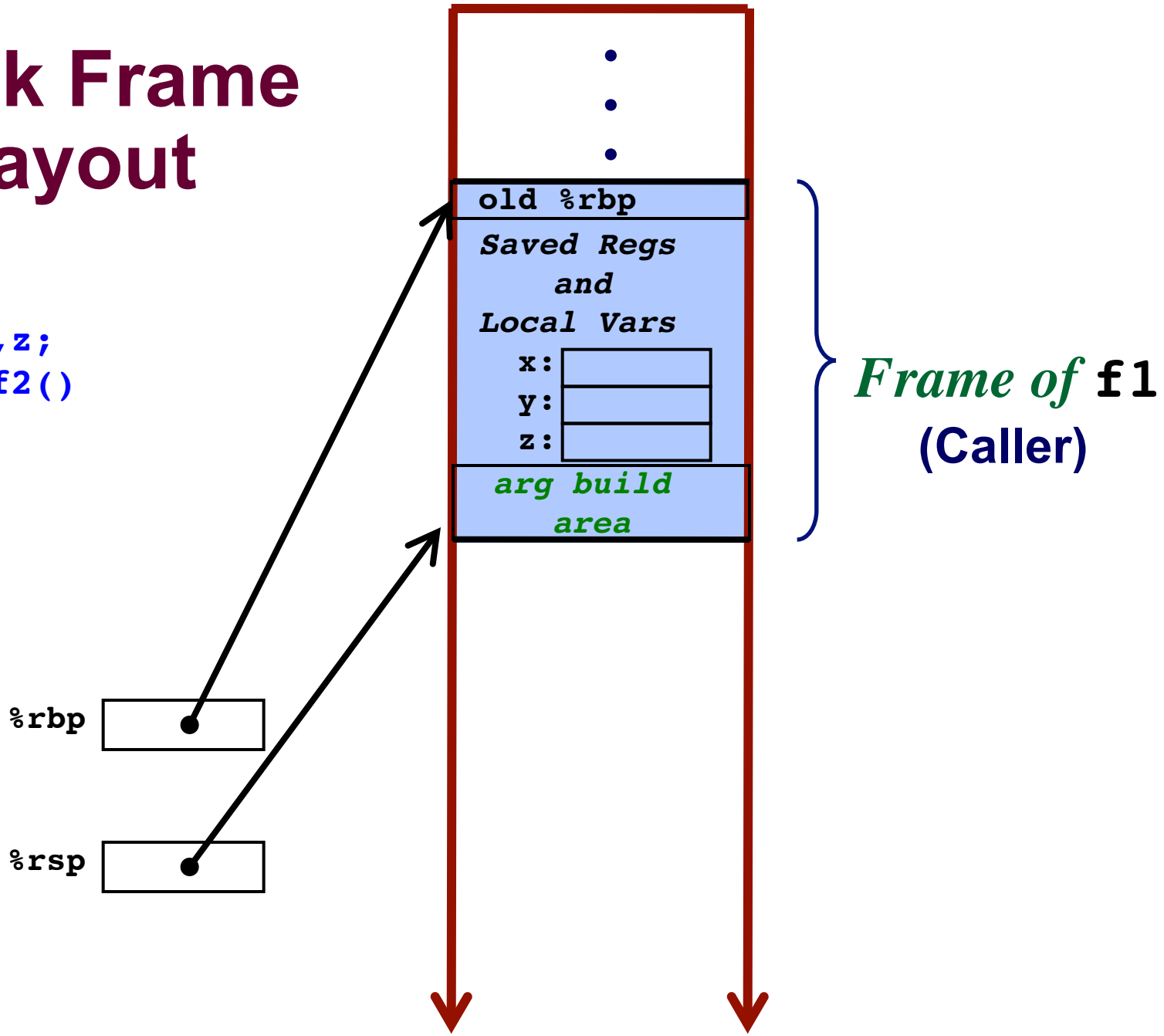
```
f2(int n, …) {
   char myArry[n];
   …
}
```

Local variables go into the frame.
But we don't know how big they are!
%rsp will be adjusted by ???
How to access things in the frame?
Use a frame Base Pointer: **%rbp**



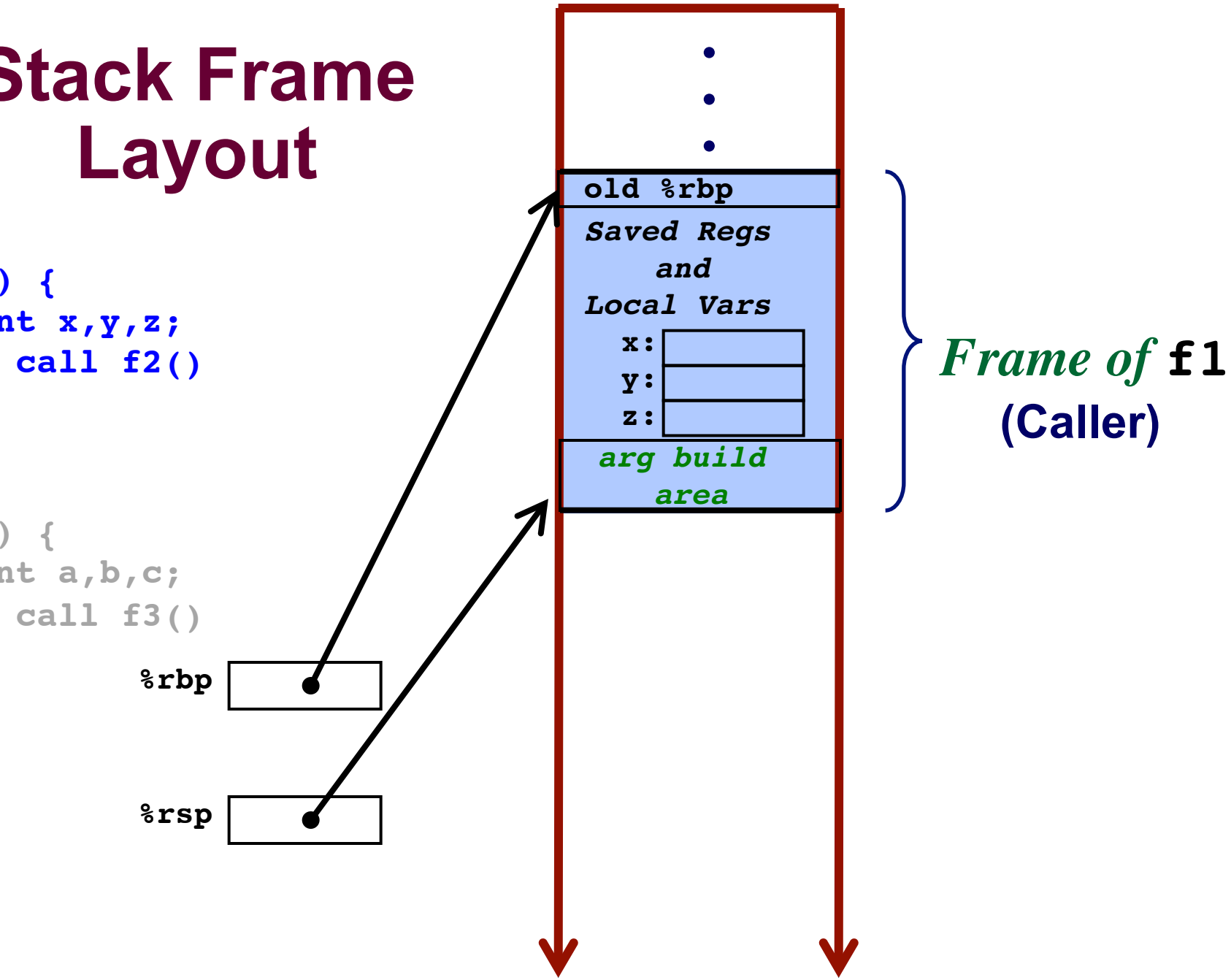
Stack Frame Layout

```
f1() {
   int x,y,z;
   … call f2()
}
```



Stack Frame Layout

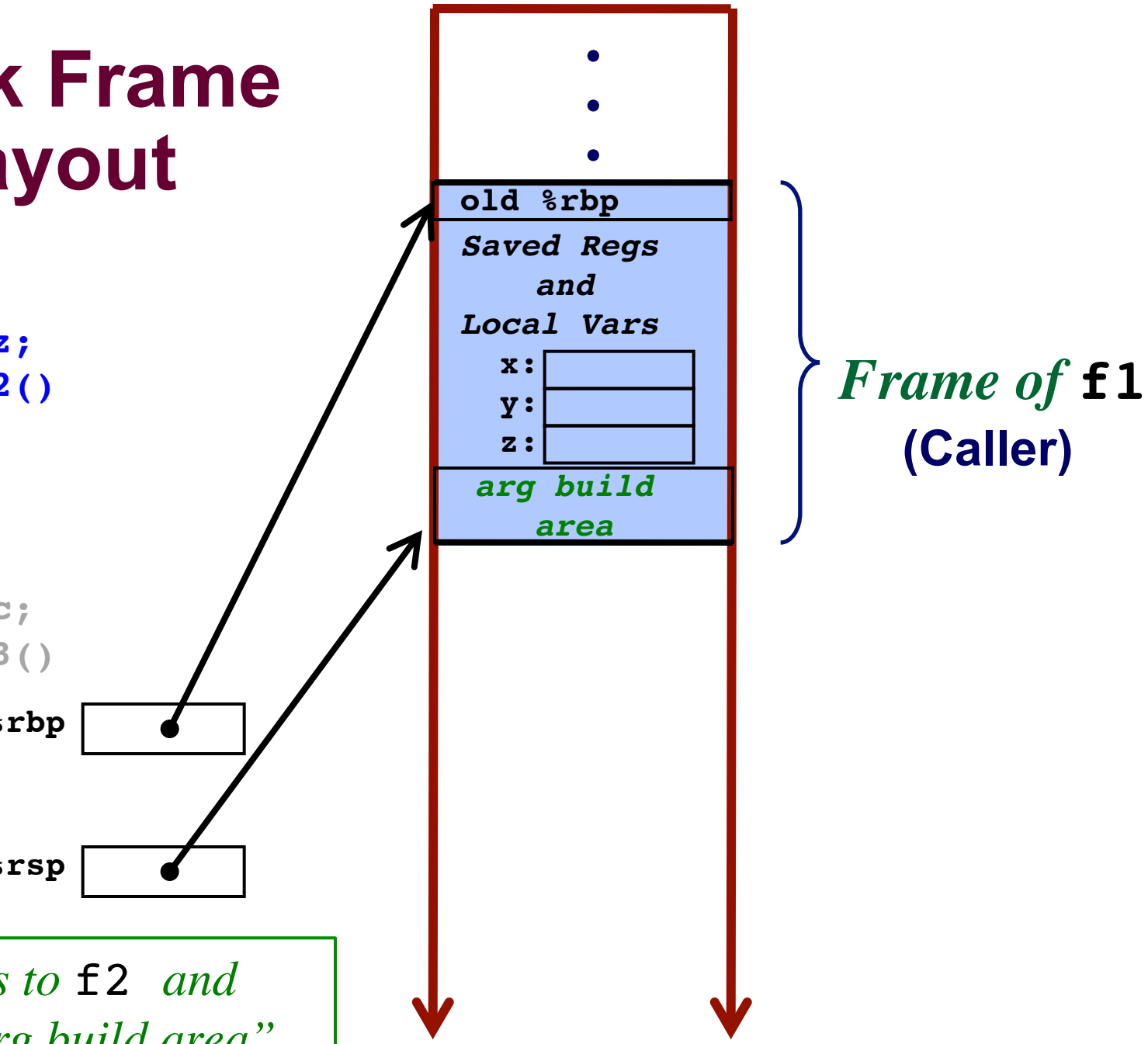
```
f1() {
  int x,y,z;
  … call f2()
}

f2() {
  int a,b,c;
  … call f3()
}
```



Stack Frame Layout

```
f1() {
  int x,y,z;
  … call f2()
}
```

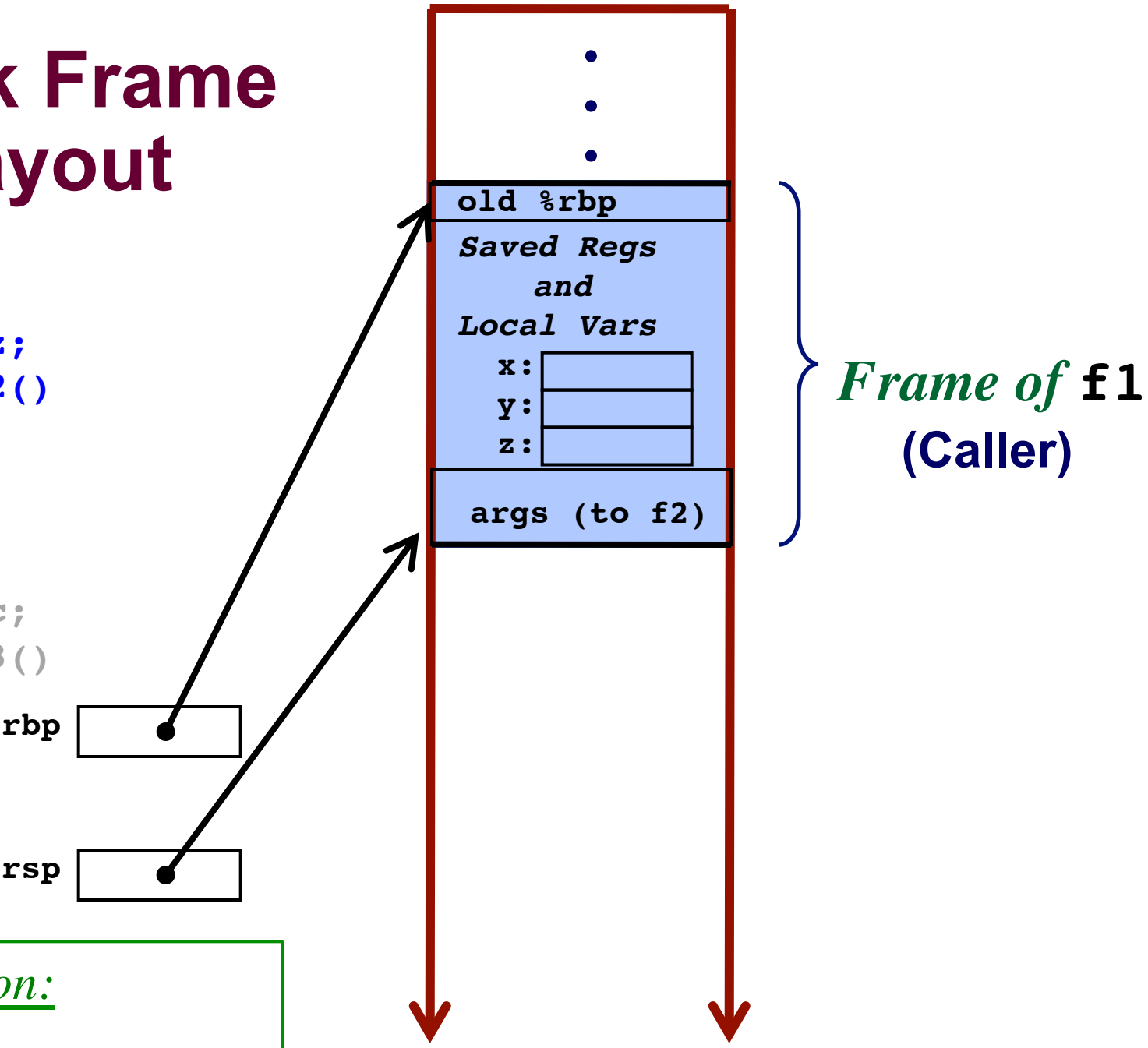
```
f2() {
  int a,b,c;
  … call f3()
}
```



Compute args to `f2` *and move into "arg build area"*

Stack Frame Layout

```
f1() {
  int x,y,z;
  … call f2()
}

f2() {
  int a,b,c;
  … call f3()
}
```

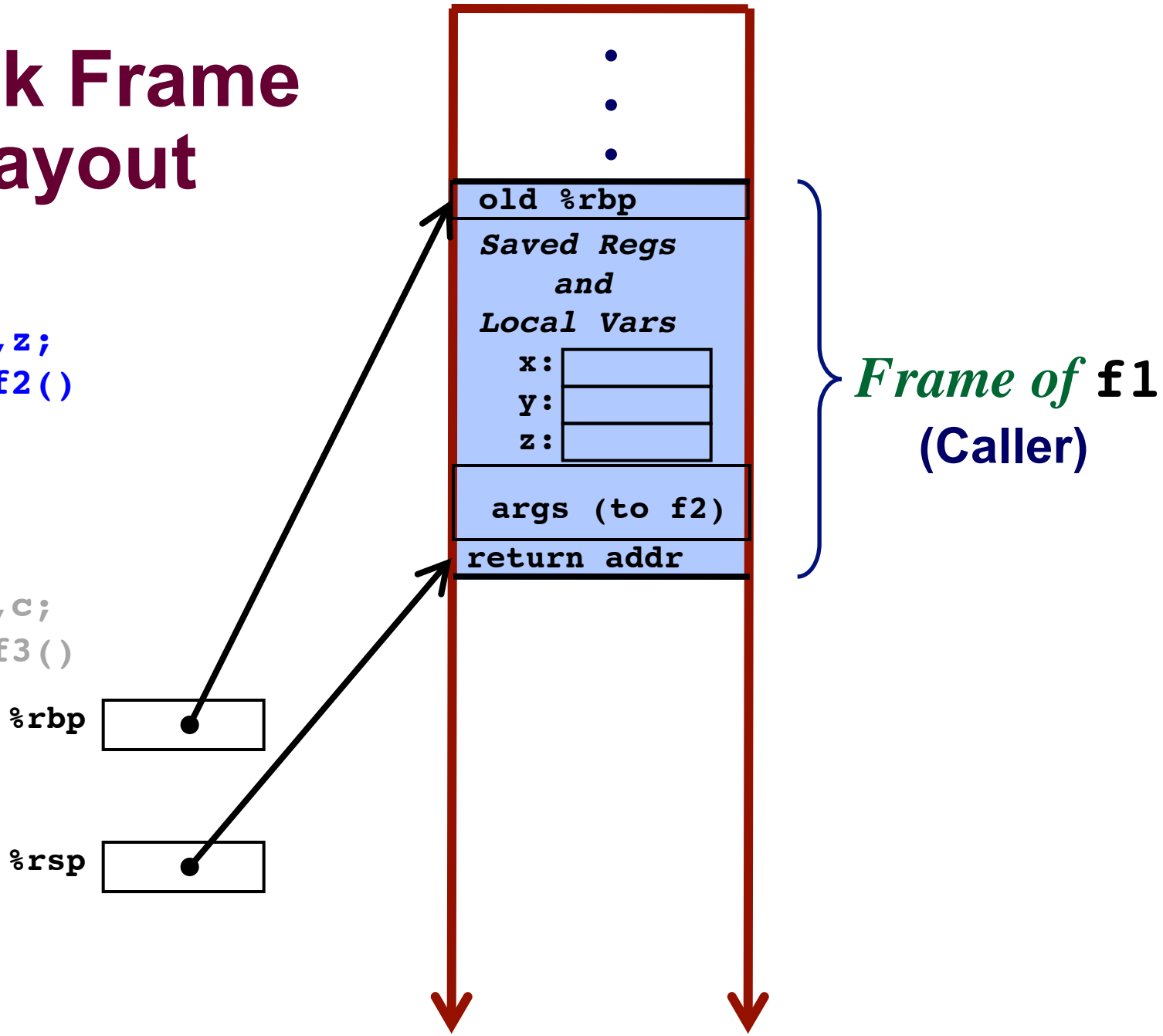


next instruction:

```
call   f2
```

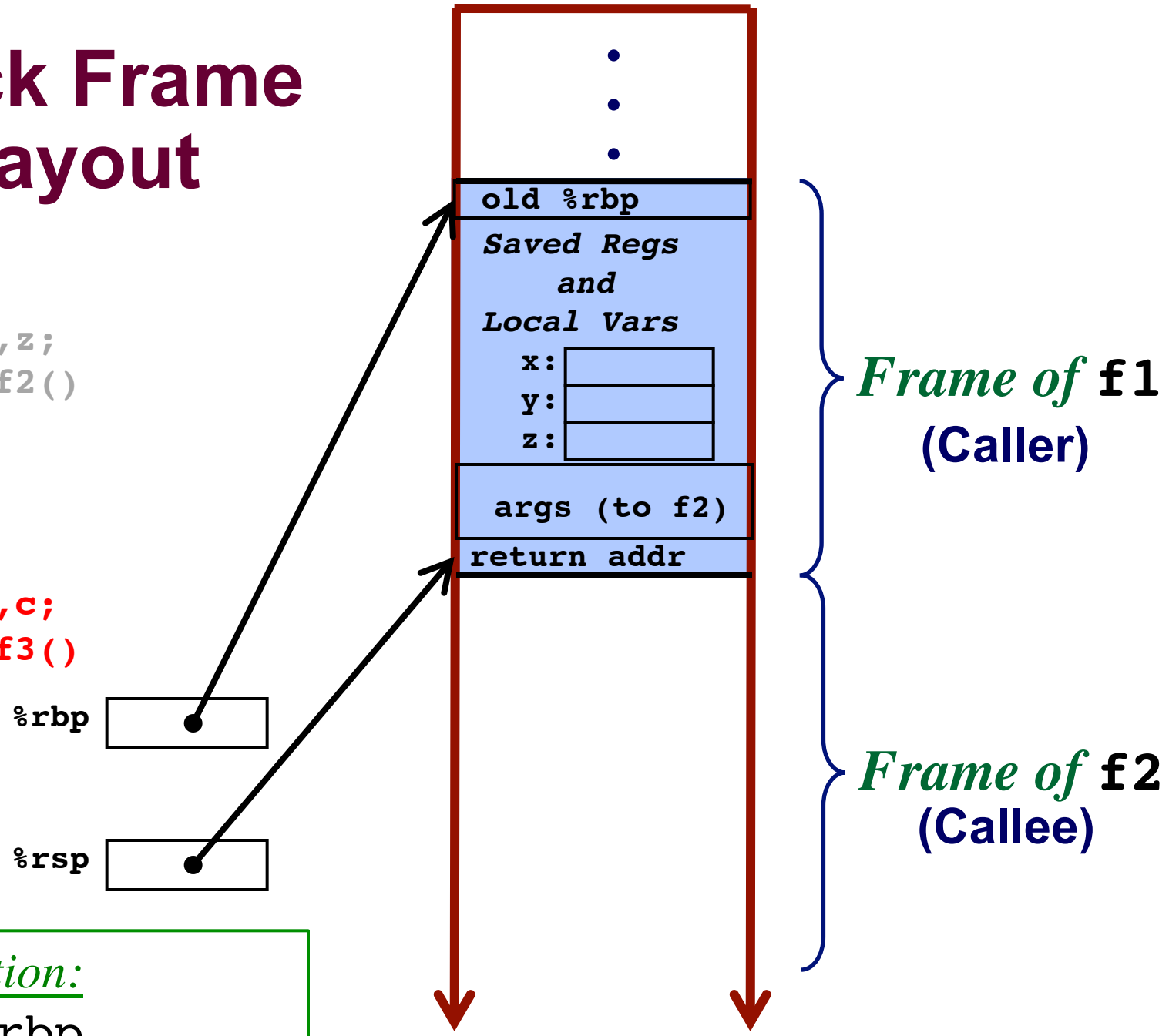
Stack Frame Layout

```
f1() {
  int x,y,z;
  … call f2()
}

f2() {
  int a,b,c;
  … call f3()
}
```



Stack Frame Layout

```
f1() {
   int x,y,z;
   … call f2()
}

f2() {
   int a,b,c;
   … call f3()
}
```

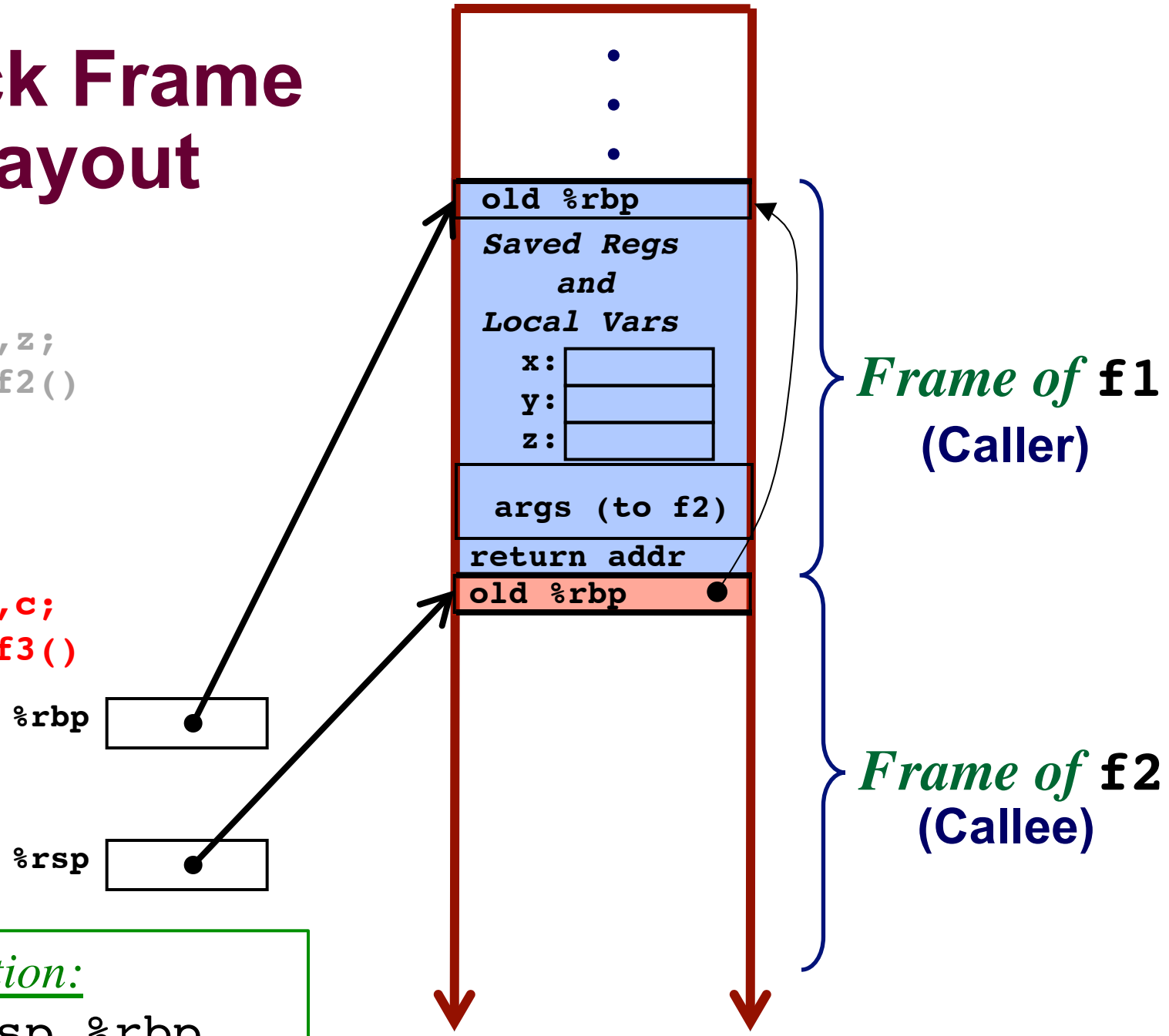


next instruction:

```
pushq   %rbp
```

Stack Frame Layout

```
f1() {
  int x,y,z;
  … call f2()
}

f2() {
  int a,b,c;
  … call f3()
}
```



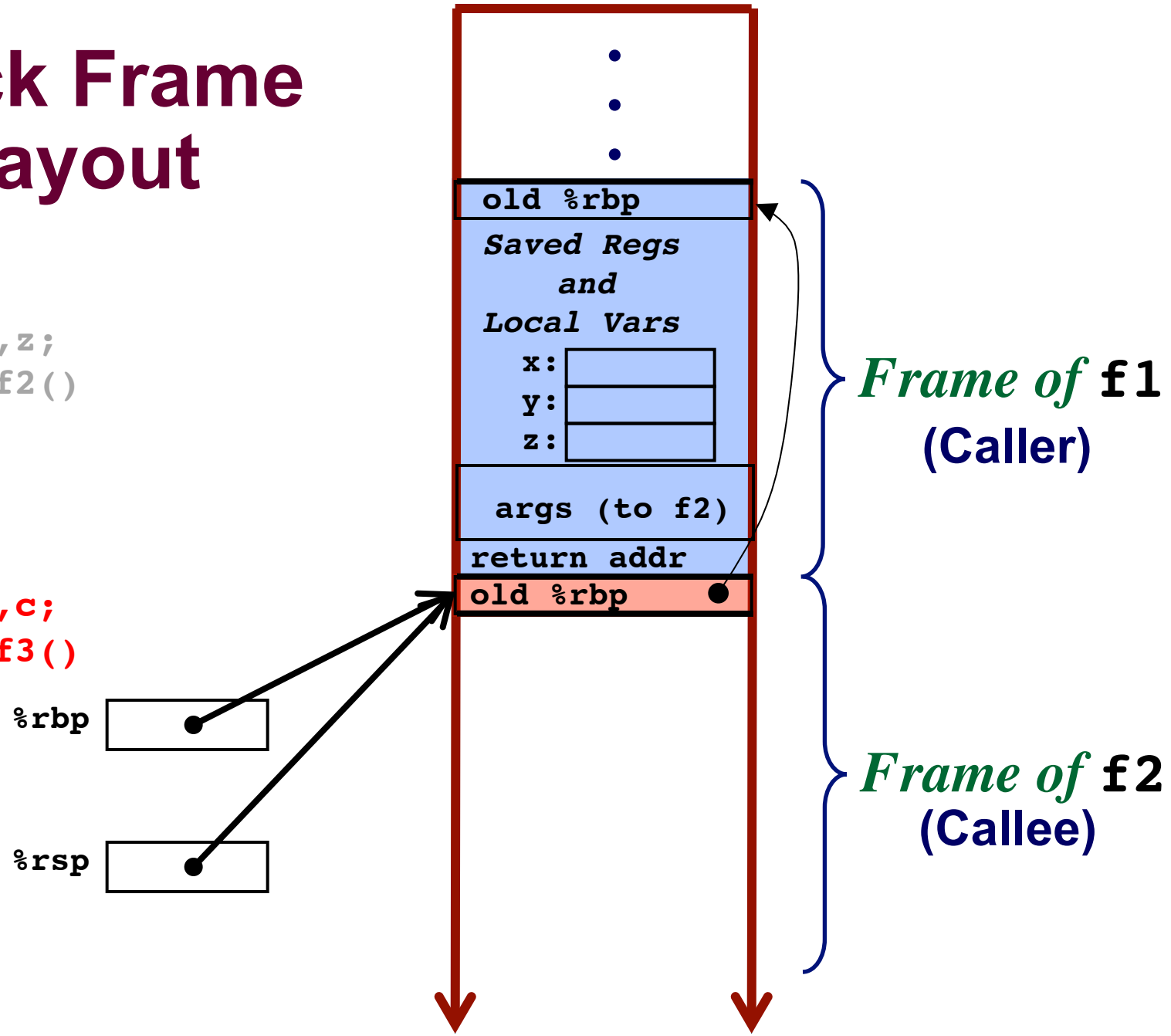
next instruction:

```
movq   %rsp,%rbp
```

Stack Frame Layout

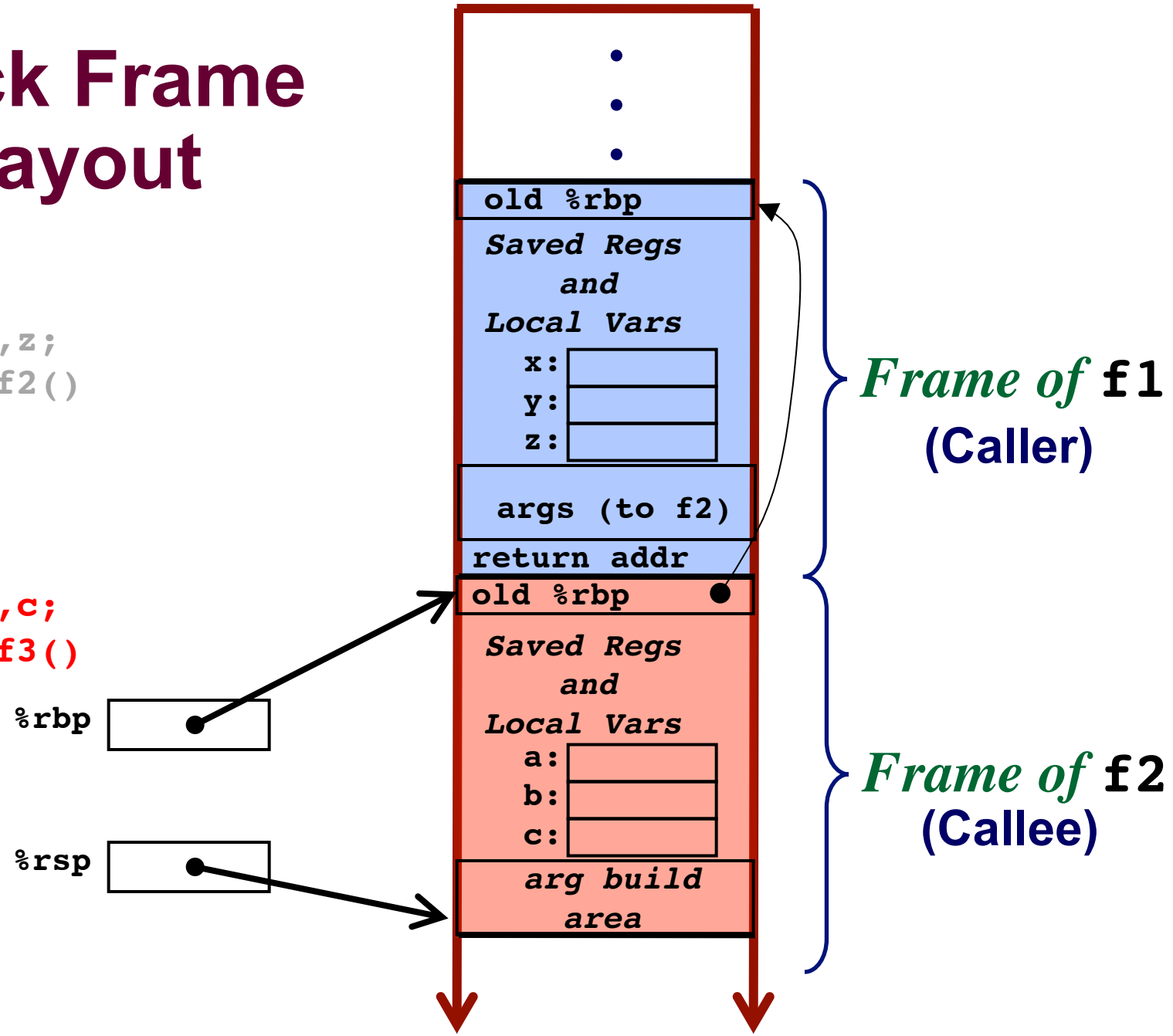
```
f1() {
  int x,y,z;
  … call f2()
}
```

```
f2() {
  int a,b,c;
  … call f3()
}
```



Stack Frame Layout

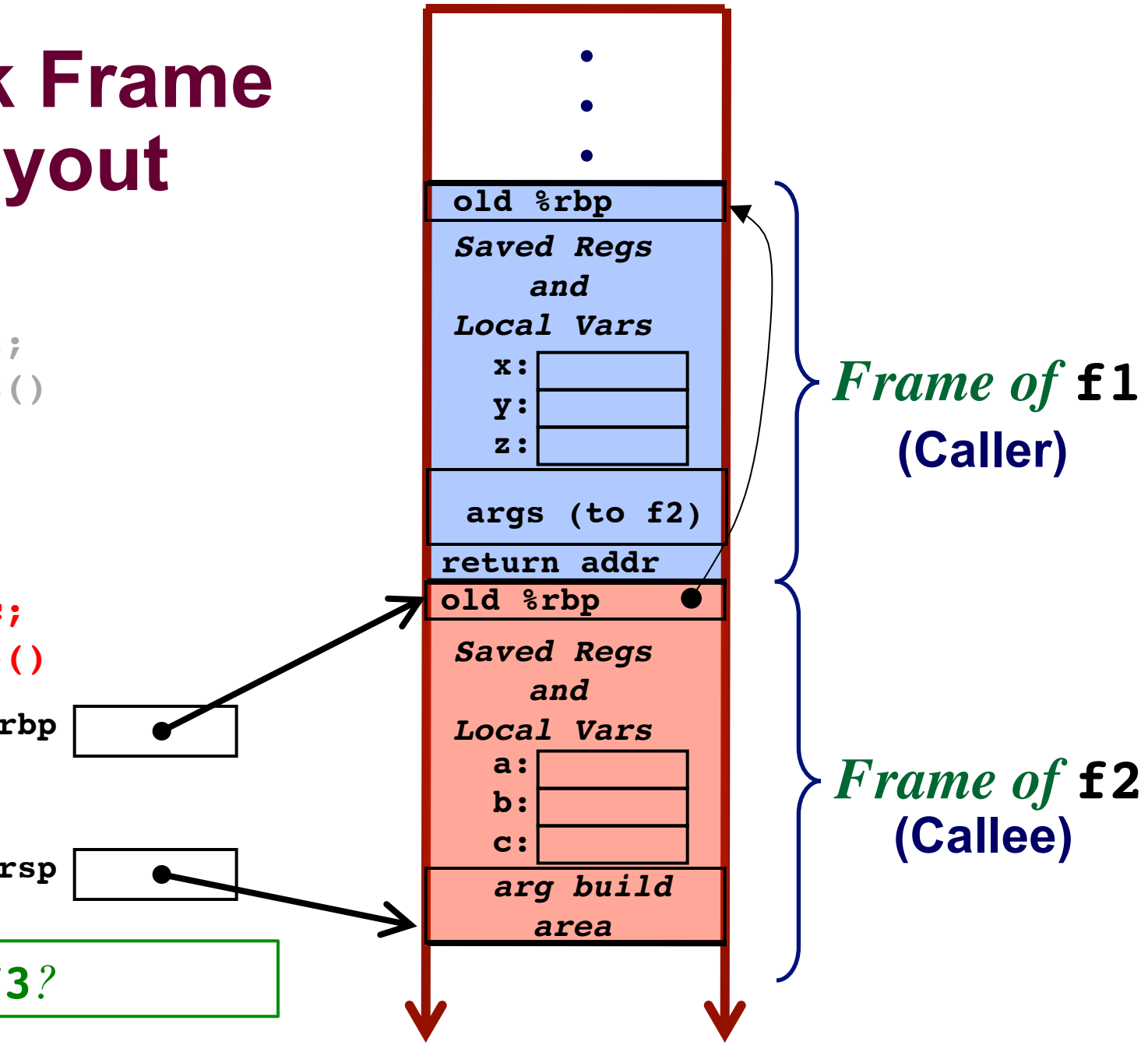
```
f1() {
   int x,y,z;
   … call f2()
}

f2() {
   int a,b,c;
   … call f3()
}
```



Stack Frame Layout

```
f1() {
   int x,y,z;
   … call f2()
}
```

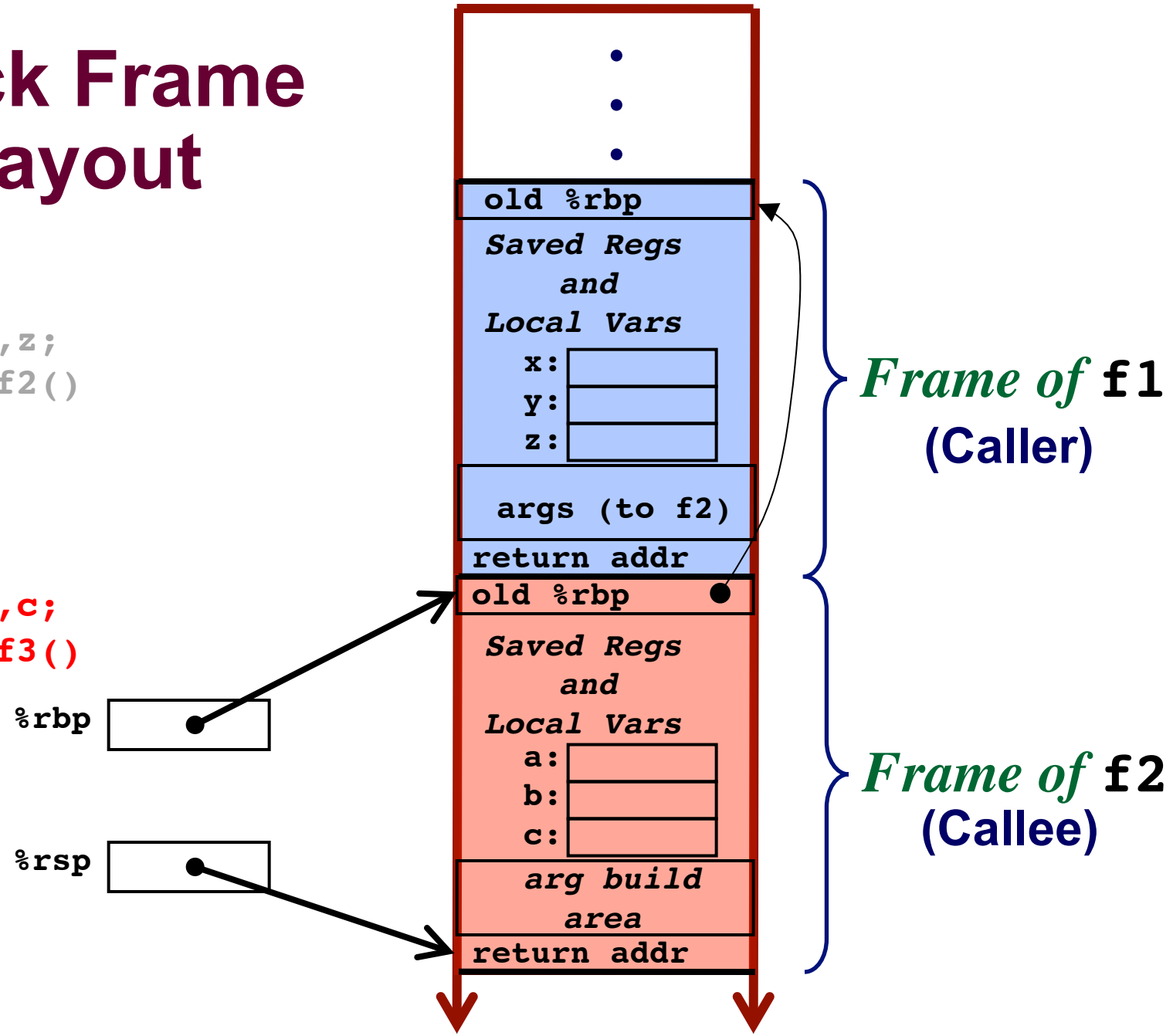
```
f2() {
   int a,b,c;
   … call f3()
}
```



Want to call **f3**?

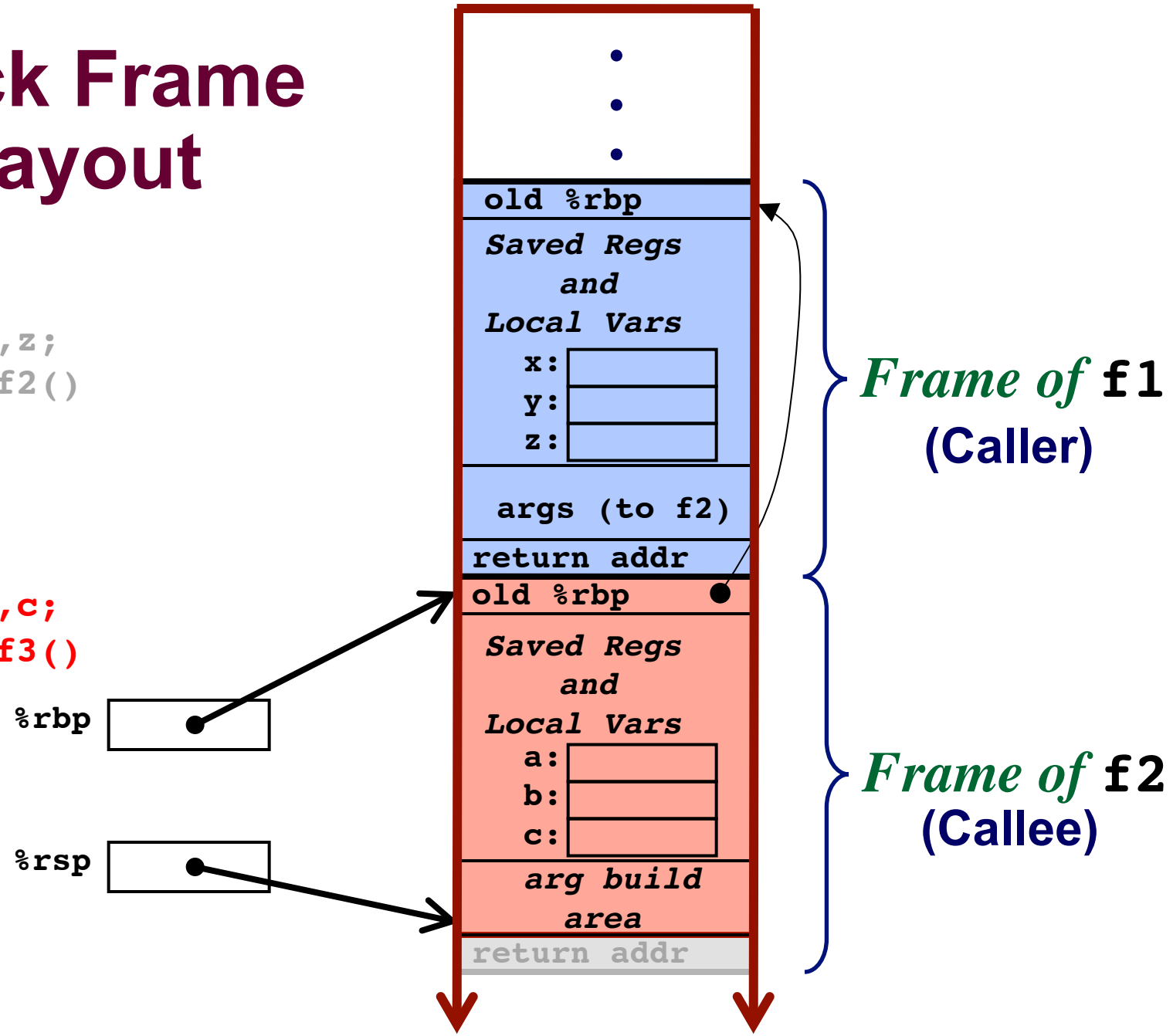
Stack Frame Layout

```
f1() {
  int x,y,z;
  … call f2()
}

f2() {
  int a,b,c;
  … call f3()
}
```



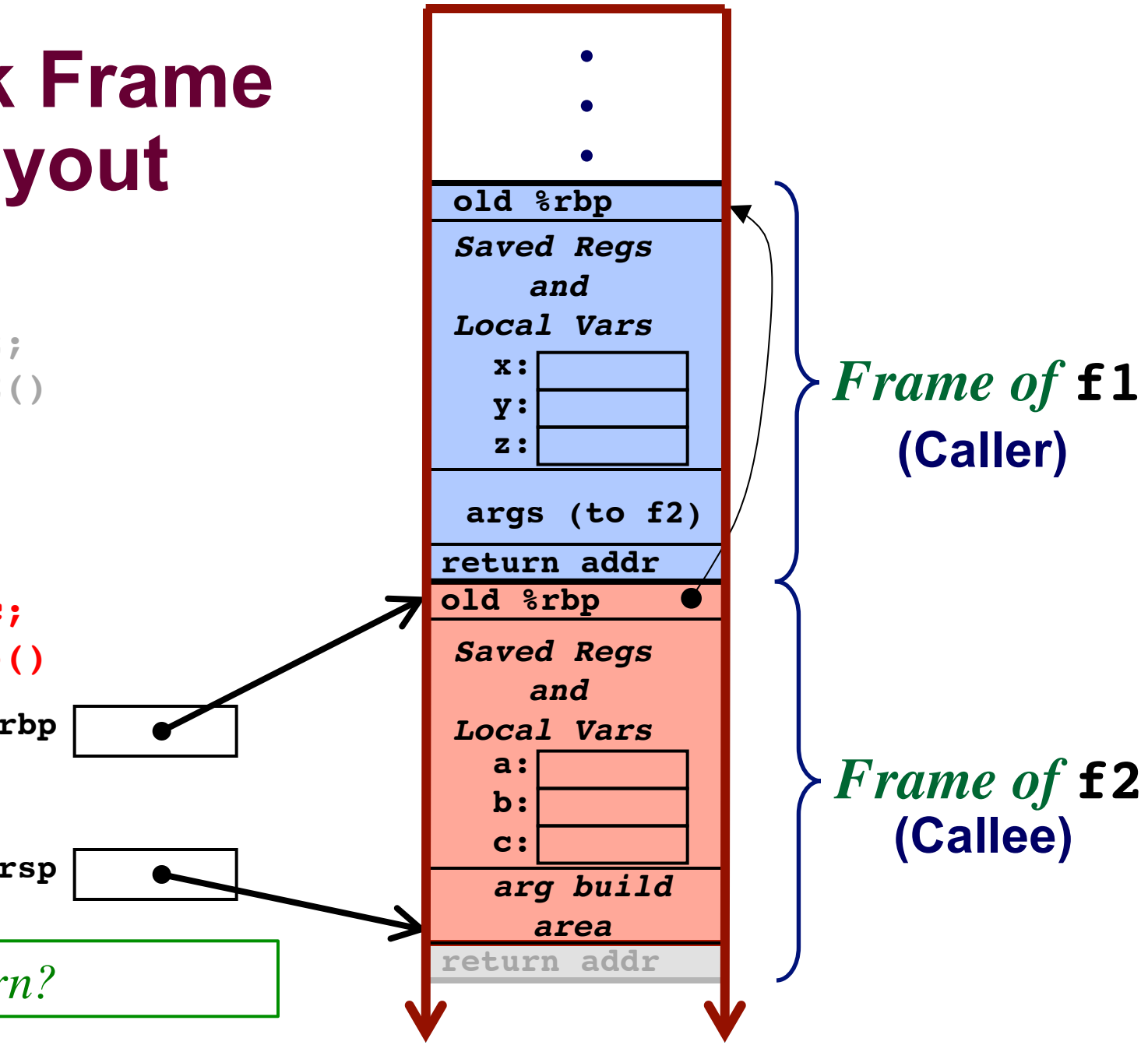
Stack Frame Layout

```
f1() {
   int x,y,z;
   … call f2()
}

f2() {
   int a,b,c;
   … call f3()
}
```



Stack Frame Layout

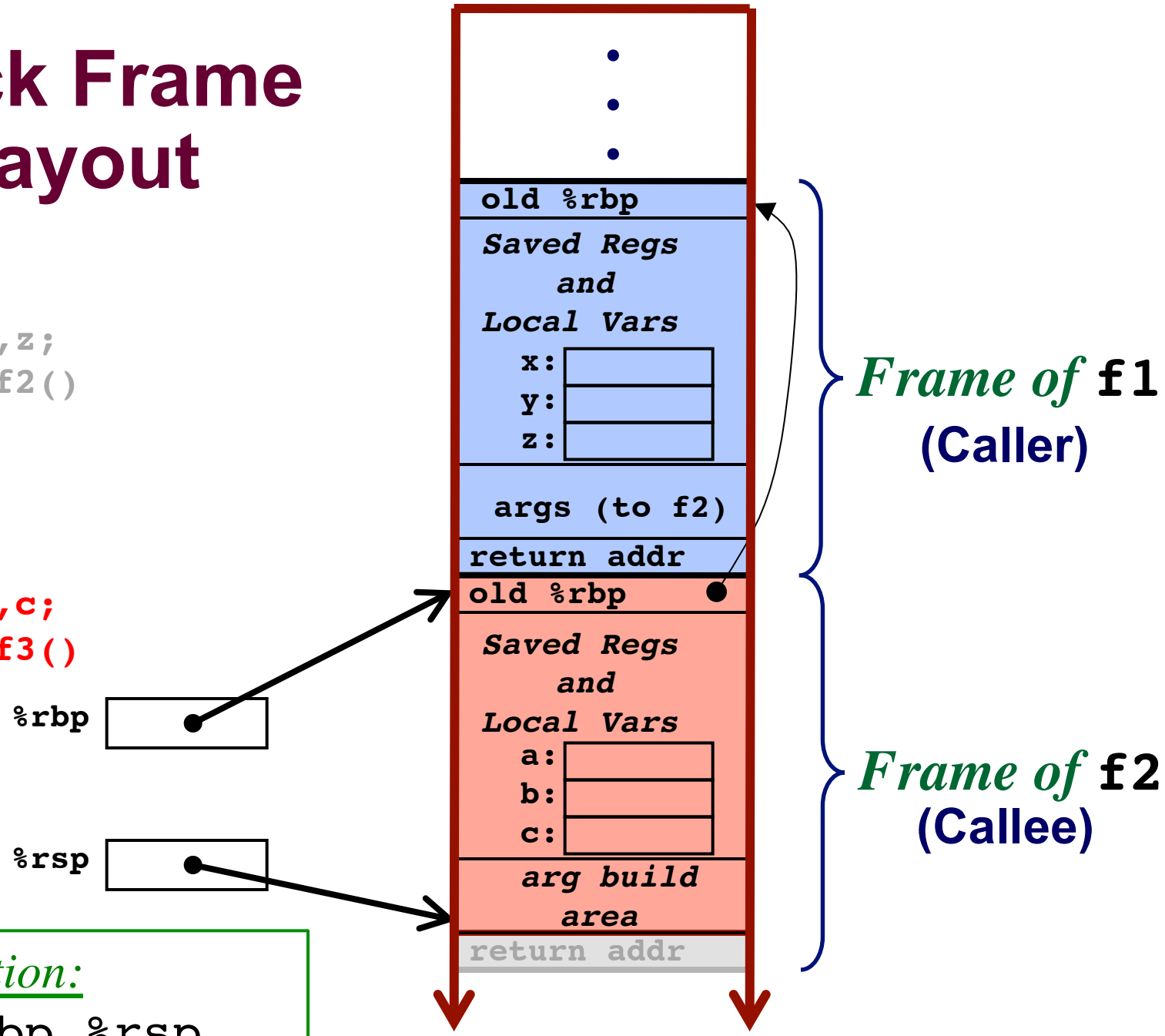
```
f1() {
   int x,y,z;
   … call f2()
}

f2() {
   int a,b,c;
   … call f3()
}
```



Ready to return?

Stack Frame Layout

```
f1() {
   int x,y,z;
   … call f2()
}

f2() {
   int a,b,c;
   … call f3()
}
```



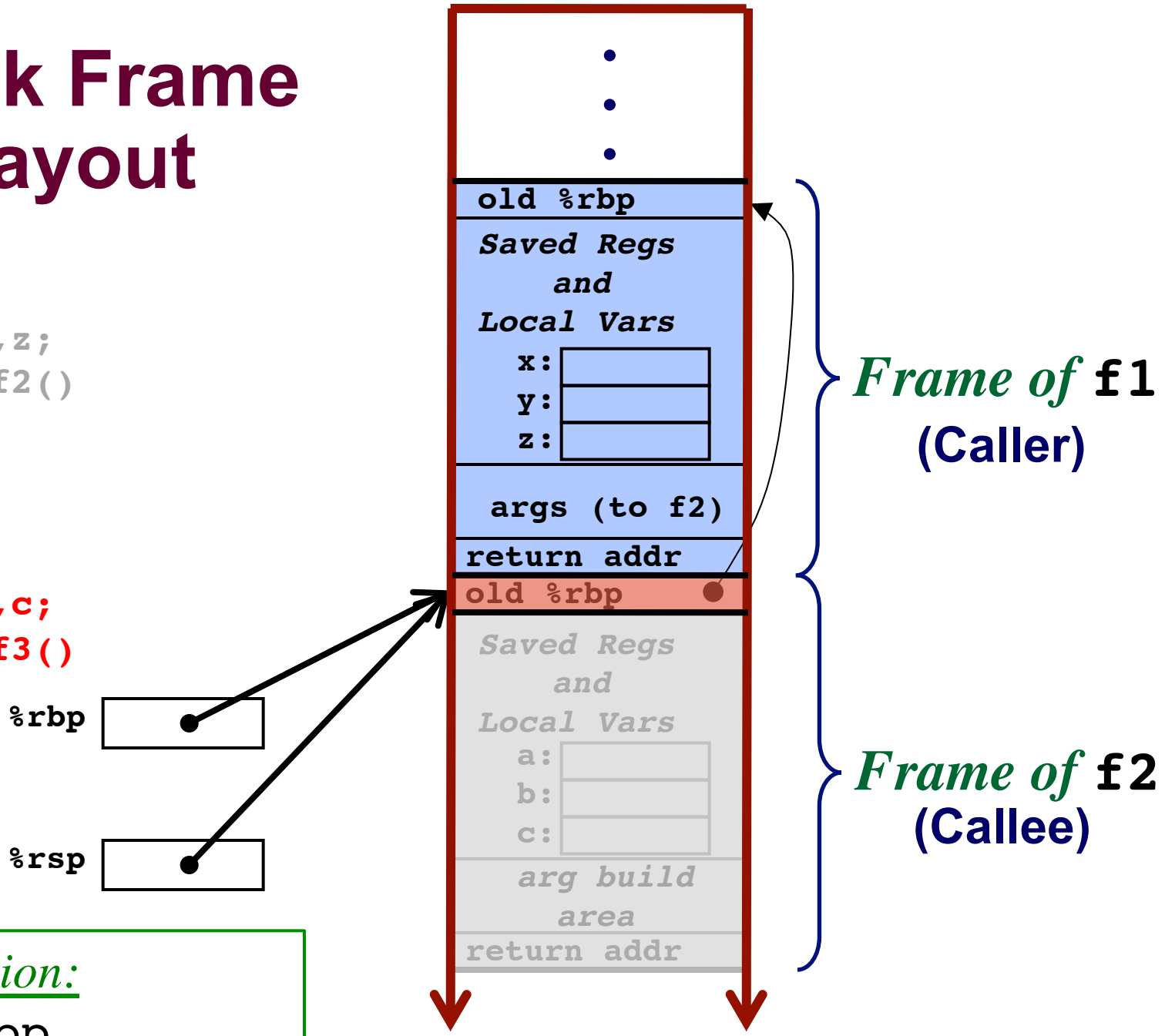
next instruction:

```
movq   %rbp,%rsp
```

Stack Frame Layout

```
f1() {
   int x,y,z;
   … call f2()
}
```

```
f2() {
   int a,b,c;
   … call f3()
}
```

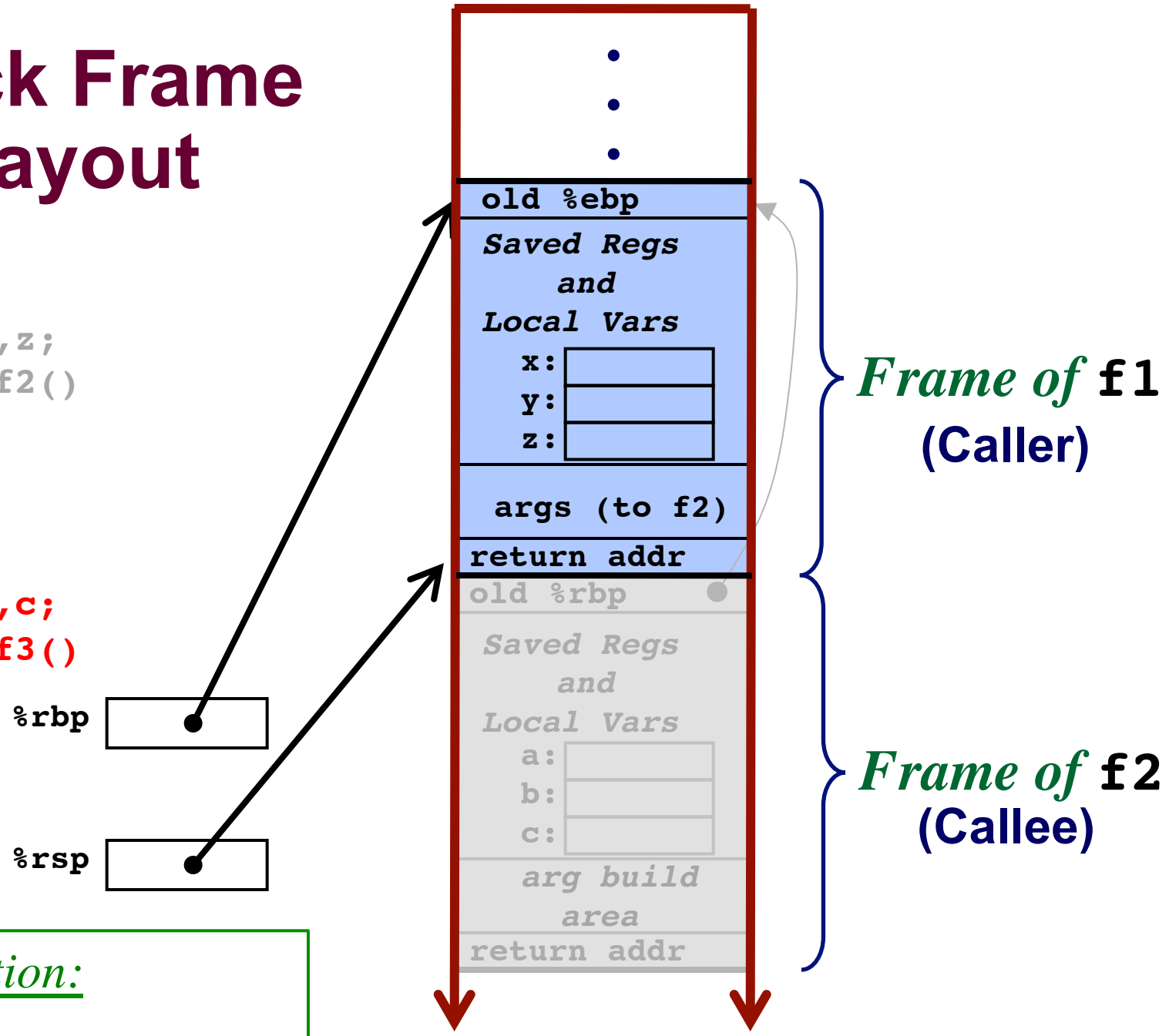


next instruction:

```
popq   %rbp
```

Stack Frame Layout

```
f1() {
   int x,y,z;
   … call f2()
}

f2() {
   int a,b,c;
   … call f3()
}
```

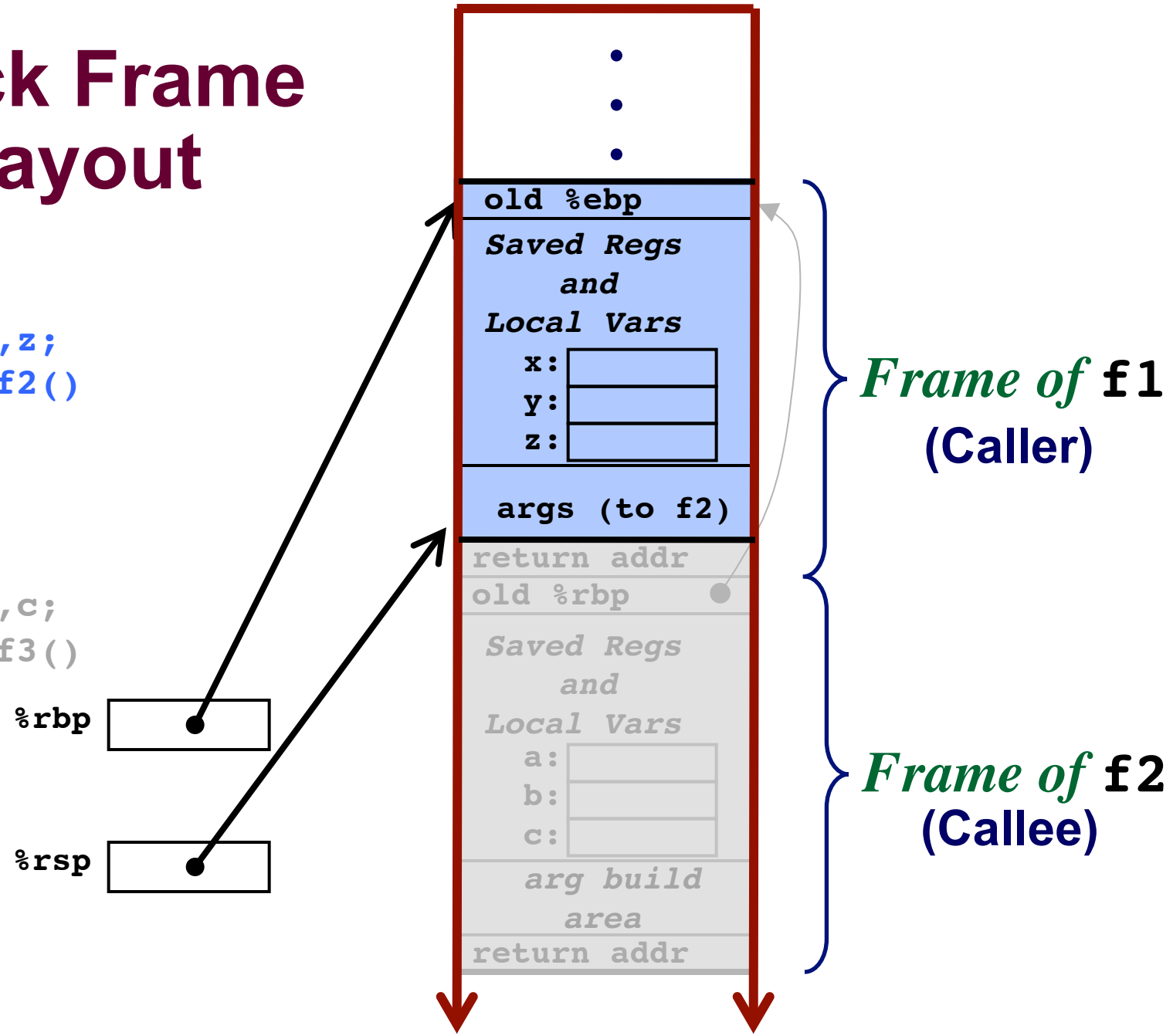


next instruction:

`ret`

Stack Frame Layout

```
f1() {
  int x,y,z;
  … call f2()
}

f2() {
  int a,b,c;
  … call f3()
}
```



What's the matter with this?

```
int *foo(.. .)
{
    int *val;
    ...
    *val = 34;
    ...
}
```

Will the C compiler allow this?

At runtime?

What happens?

What's the matter with this?

```
int *foo(.. .)
{
    int *val;
    ...
    *val = 34;
    ...
}
```

Will the C compiler allow this?

A warning is given.

At runtime?

Attempts to modify a "random" location.

What happens?

Depends on what address "val" happens to contain.

- In a valid page
- Not in a valid page

Leads to a program crash (segfault)? (We hope!)

Program keeps going with bad data!!!

What's the matter with this?

```
int *foo(.. .)
{
    int x;
    ...
    return &x;
}
```

Will the C compiler allow this?

What happens?

What if the pointer it returns is dereferenced?

What's the matter with this?

```
int *foo(.. .)
{
    int x;
    ...
    return &x;
}
```

Will the C compiler allow this?

A warning is given

What happens?

Returns an address that is no longer part of the stack

What if the pointer it returns is dereferenced?

Reads/writes a random location

- Possibly in a new frame
- Possibly beyond the stack top

Leads to a program crash? (We hope!)

Program keeps going with bad data!!!