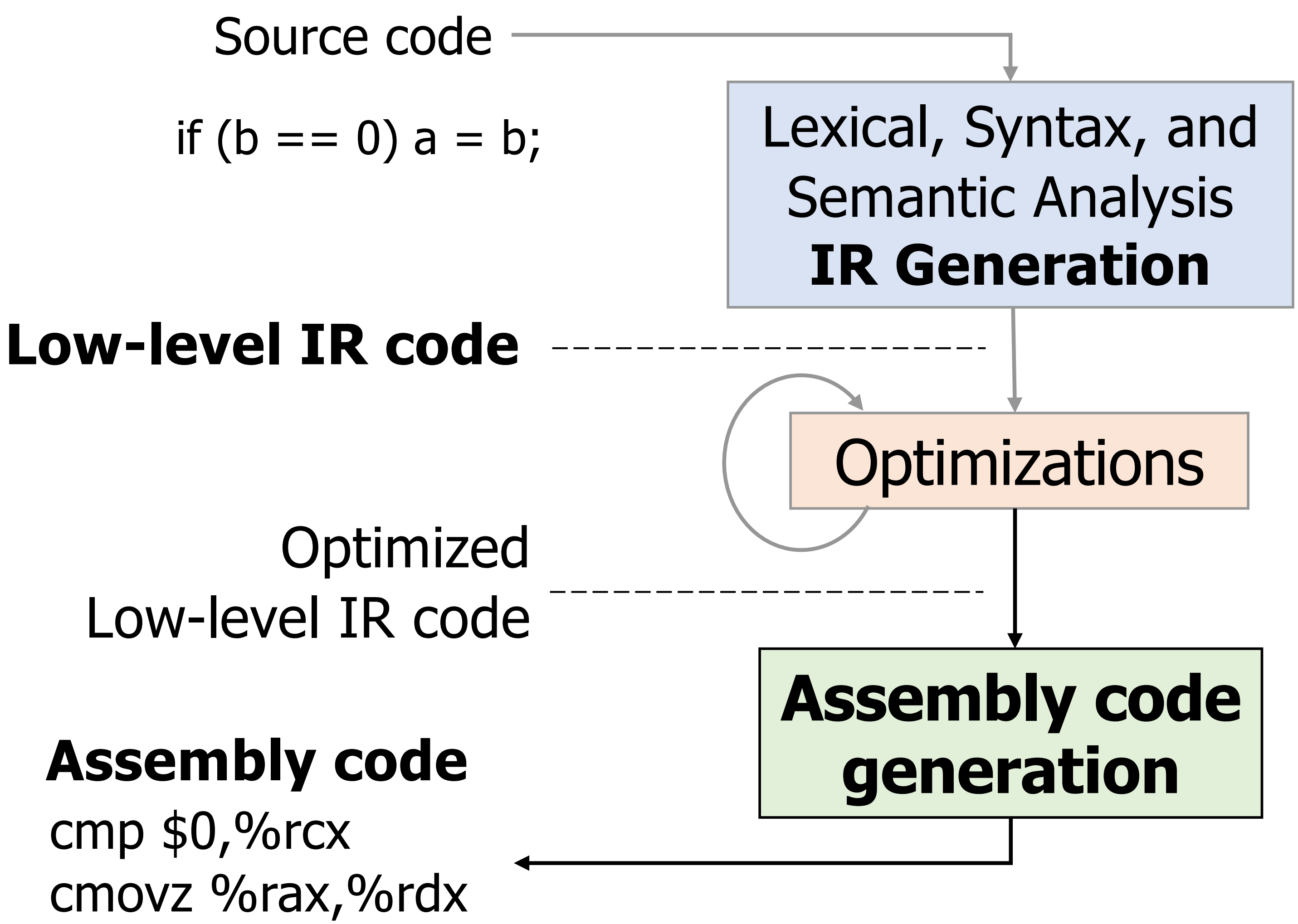
# Where We Are

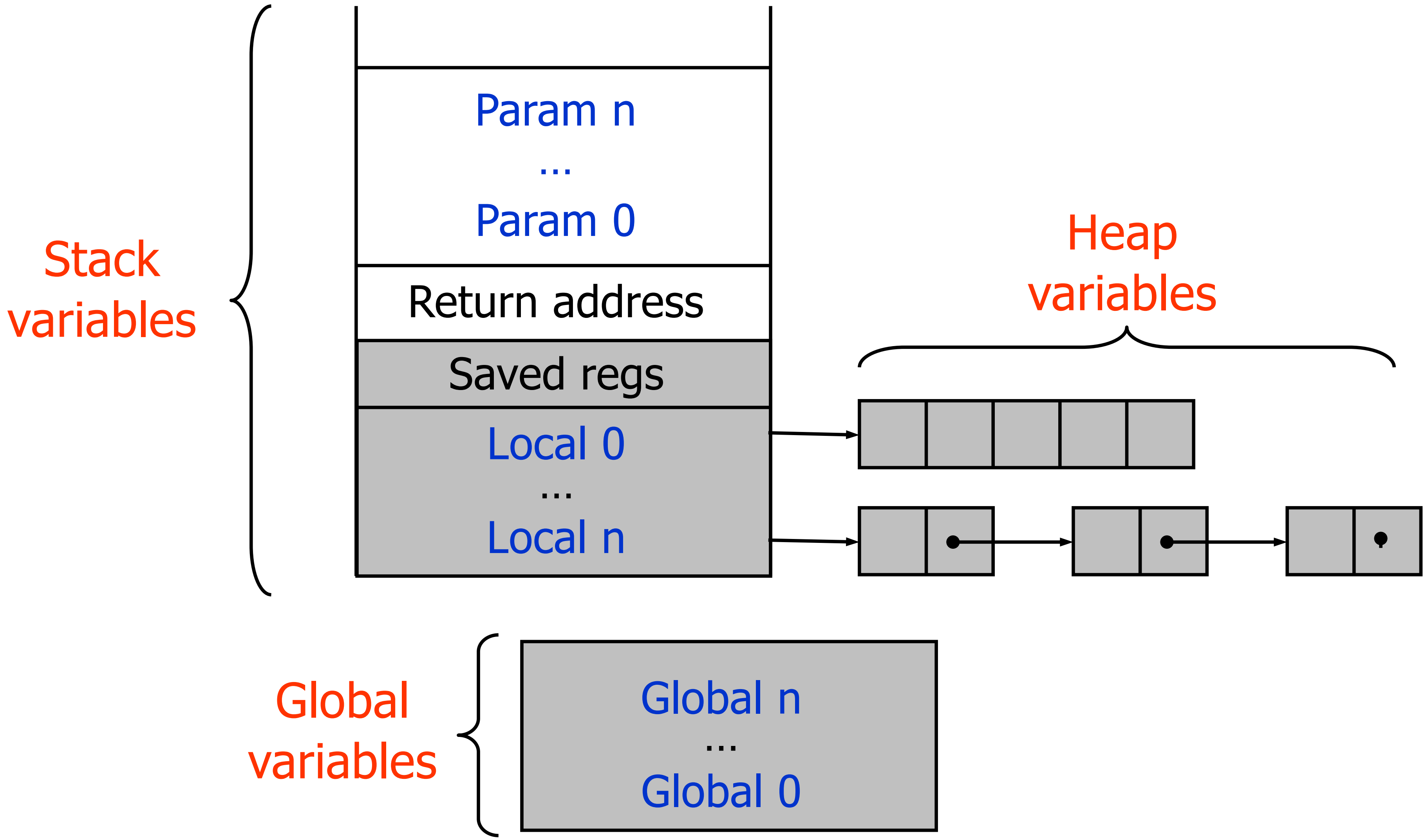

Source code
if (b == 0) a = b;
Lexical, Syntax, and Semantic Analysis
IR Generation
Low-level IR code
Optimizations
Optimized Low-level IR code
Assembly code generation
Assembly code
cmp $0,%rcx
cmovz %rax,%rdx

# Code Generation: IR to Assembly Translation

- IR code (TAC):
  - Variables (and temporaries)
  - No run-time stack
  - No calling sequences
  - Abstract set of instructions
- Translation to x86-64:
  - Calling sequences:
    - Translate function calls and returns
    - Manage run-time stack
  - Variables:
    - globals, locals, arguments, etc. assigned memory location
  - Instruction selection:
    - map sets of low level IR instructions to instructions in the target machine

```
t3 = a.x
t3 = t2 * t3
t0 = t1 + t2
r = t0
t4 = w + 1
```

# Big Picture: Memory Layout



# x86-64 stack frames

**x86-64/Linux ABI**

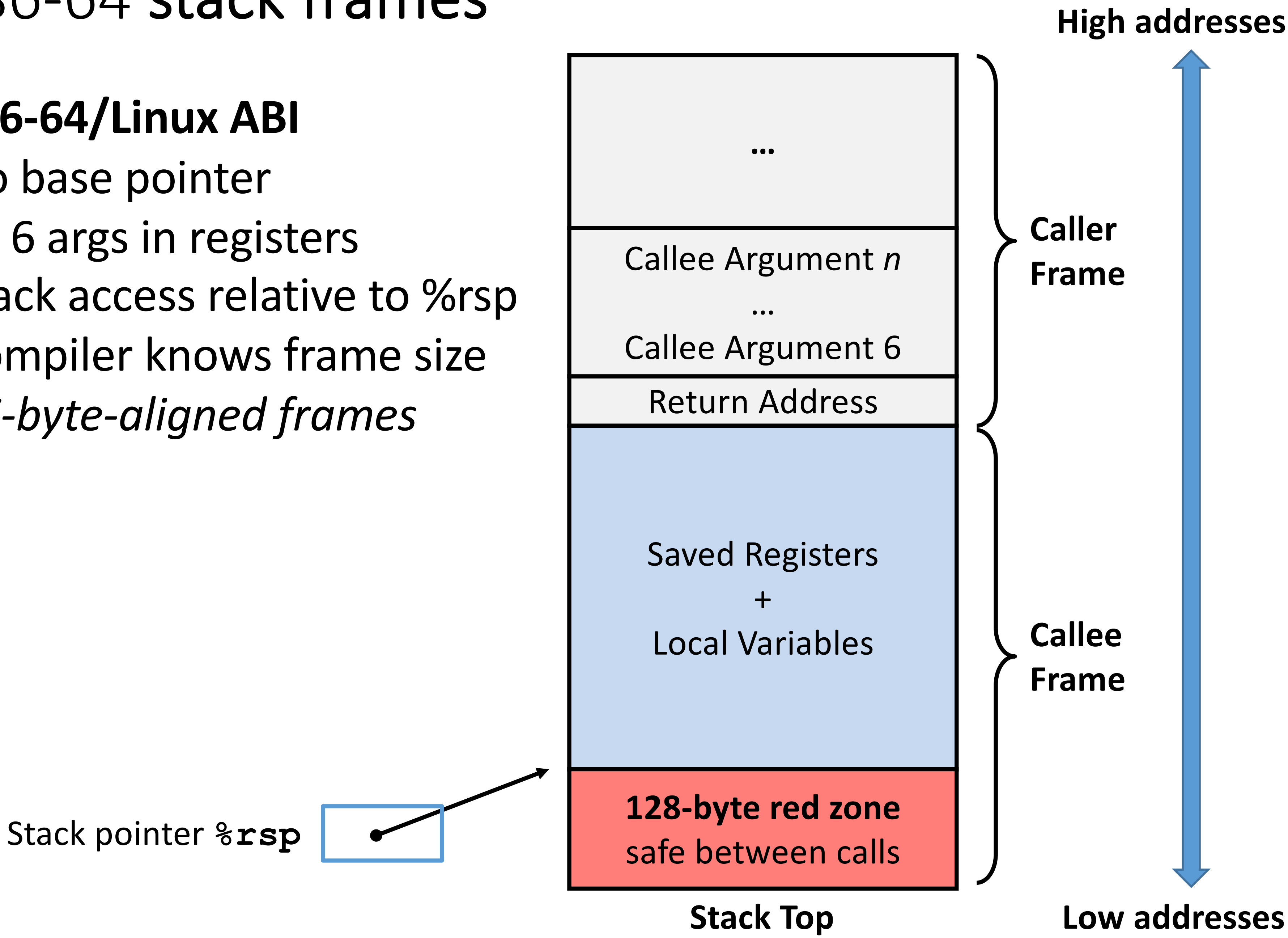
No base pointer

1st 6 args in registers

Stack access relative to %rsp

Compiler knows frame size

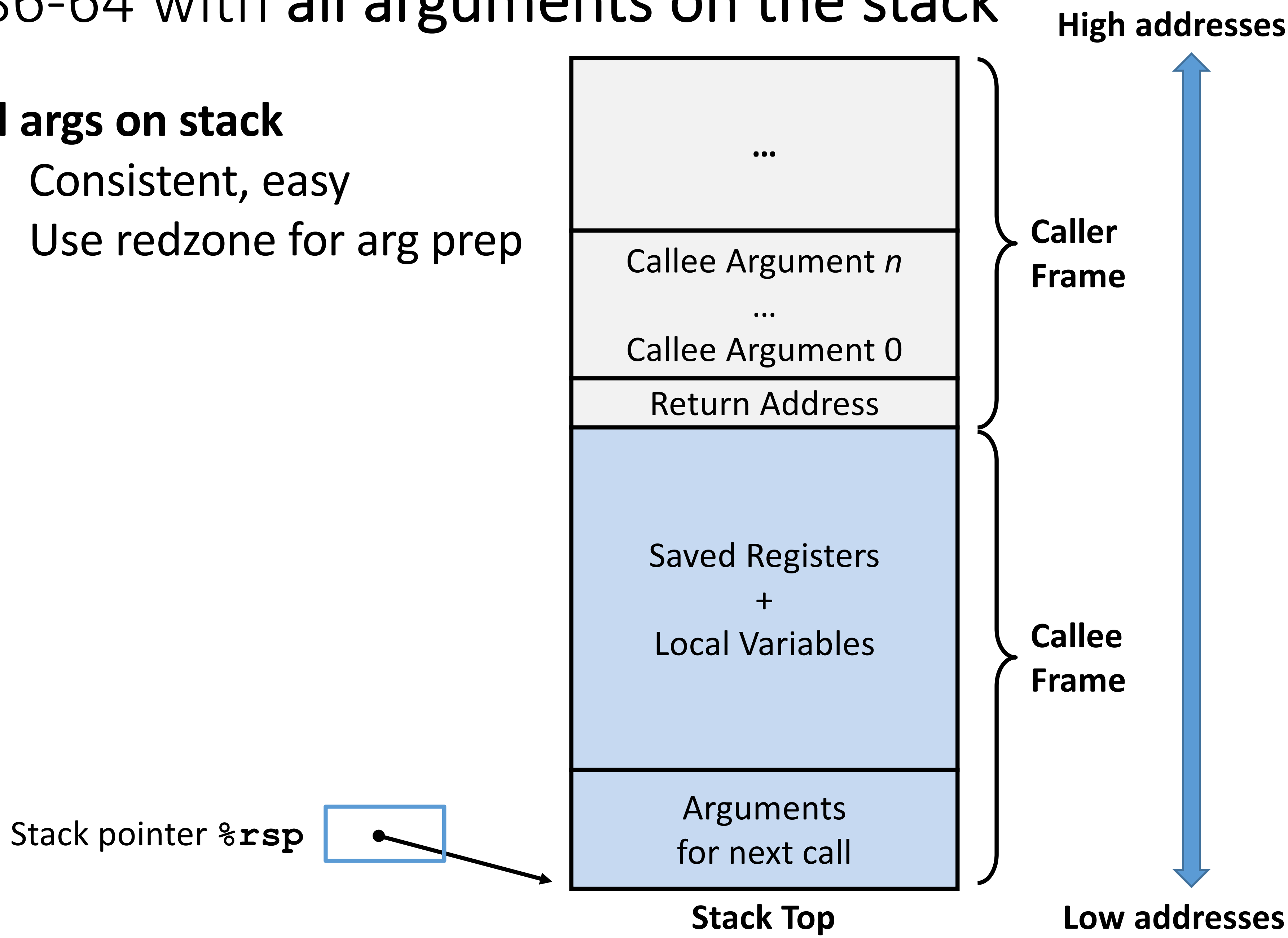
*16-byte-aligned frames*



# x86-64 with **all arguments on the stack**

**All args on stack**

- Consistent, easy
- Use redzone for arg prep



# x86-64/Linux ABI: register conventions

| | | | |
|---|---|---|---|
| `%rax` | Return value | `%r8` | Argument #5 |
| `%rbx` | Callee saved | `%r9` | Argument #6 |
| `%rcx` | Argument #4 | `%r10` | Caller saved |
| `%rdx` | Argument #3 | `%r11` | Caller Saved |
| `%rsi` | Argument #2 | `%r12` | Callee saved |
| `%rdi` | Argument #1 | `%r13` | Callee saved |
| `%rsp` | Stack pointer | `%r14` | Callee saved |
| `%rbp` | Callee saved | `%r15` | Callee saved |

**Only %rsp is special-purpose.**

# roostc calling convention

- Always follow **x86-64/Linux register save convention**.
- To interface with **external code** (LIB), use:
  - x86-64/Linux calling convention.
- To interface with other **roostc-generated code**, use one of:
  1. frame pointer and stack pointer, all args on stack
     - Easy, more work to convert if you convert to #3 later.
  2. stack pointer only, all args on stack (recommended)
     - Moderately easy, easier to convert to #3 later.
  3. x86-64/Linux calling convention
     - Harder, requires more register allocation work, more efficient, **only use this later if you have time.**

# Simple stack frame maintenance: stack args

- Compute offsets for each variable
  - Count args, local vars, temporaries
  - Assign each an offset relative to %rsp (top of stack frame).
  - Args are fixed, everything else flexible.
- Avoid using callee-save registers – no saving required.
- Prologue at start of function:

```
subq $24, %rsp    # space for 3 locals (or 2 plus alignment)
```

- Epilogue/cleanup at end of function:

```
addq $24, %rsp
retq
```

- Many improvements possible, but a good starting point.

# Simple template-based code generation

```
a = p+q   →   movq 16(%rsp), %rax
              addq 8(%rsp), %rax
              movq %rax, 24(%rsp)
```

- Need to consider many language constructs:
  - Operations: arithmetic, logic, comparisons
  - Accesses to local variables, global variables
  - Array accesses, field accesses
  - Control flow: conditional and unconditional jumps
  - Method calls, dynamic dispatch
  - Dynamic allocation (new)
  - Run-time checks

# Division

```
movq …, %rcx # divisor, can't be %rax or %rdx
movq …, %rax # dividend
cqto         # sign-extend %rax into %rdx:%rax
idivq %rcx   # divide %rdx:%rax by %rcx
             # quotient in %rax
             # remainder in %rdx
```

# String Literals

```
.rodata
      ...
      .align 8
      .quad 13
strlit3:
      .ascii "Hello, World!"
      ...
.text
      ...
    # t4 = "Hello, World!"
    # Works on both LLVM/macOS and GCC/Linux:
      leaq strlit3(%rip), %rax    # GCC only: movq $strlit3, %rax
      movq %rax, 8(%rsp)
    # Roost.println(t4);
    # external call must use x86-64/Linux ABI calling convention
      movq 8(%rsp), %rdi    # load t4 as arg 1
      callq __LIB_println
```

Method vectors/vtables and vtable pointer initialization will be similar.

# `cmpq` and `testq`

`cmpq %rcx,%rax` computes `%rax - %rcx`,
sets CF, OF SF, ZF, discards result

`testq %rax,%rcx` computes `%rax & %rcx`,
sets SF, ZF, discards result

Flags/condition codes:

- CF: carry flag, 1 iff carry out
- OF: overflow flag, 1 iff signed overflow
- SF: sign flag, 1 iff result's MSB=1
- ZF: zero flag, 1 iff result=0

Common pattern to test for 0 or <0: `testq %rax, %rax`

# `jmp` and `j`*CC*

**Always jump**

**Jump iff *condition***

| *jCC* | Condition | Jump iff ... |
|---|---|---|
| `jmp` | `1` | **Unconditional** |
| `je, jz` | `ZF` | **Equal / Zero** |
| `jne, jnz` | `~ZF` | **Not Equal / Not Zero** |
| `jg` | `~(SF^OF)&~ZF` | **Greater** (Signed) |
| `jge` | `~(SF^OF)` | **Greater or Equal** (Signed) |
| `jl` | `(SF^OF)` | **Less** (Signed) |
| `jle` | `(SF^OF)\|ZF` | **Less or Equal** (Signed) |
| `js` | **`SF`** | **Negative** |
| `jns` | **`~SF`** | **Nonnegative** |
| `ja` | `~CF&~ZF` | Above (unsigned) |
| `jb` | `CF` | Below (unsigned) |

# `set`*CC* and `movzbq`

```
# t7 = t4 <= t9
movq 72(%rsp), %rdx    # %rdx = t9
cmpq 32(%rsp), %rdx    # set flags: t9 - t4
setle %al              # set byte to 0x00 or 0x01
                       # based on condition le: <=
                       # as in %rdx <= 32(%rsp)
movzbq %al, %rax       # move, zero-extend byte to quad
                       # (Extend to 64 bits.)
movq %rax, 56(%rsp)    # t7 = result
```

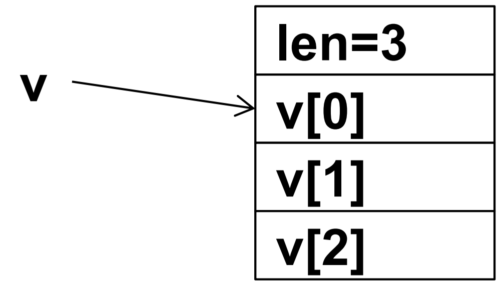
**Set has all the same flavors as conditional jump.**

# Accessing Heap Data

- Heap data allocated with new (Java) or malloc (C/C++)
  - Allocation function returns address of allocated heap data
  - Access heap data through that reference

- Array accesses in Java
  - access **`a[i]`** requires:
    - computing address of element: **`a + i * size`**
    - accessing memory at that address
  - Example:
    - assume size of array elements is 8 bytes, and local variables a, i (offsets –8, -16)

```
a[i] = 1  ➡  mov -8(%rbp), %rdx         (load a)
             mov -16(%rbp), %rcx        (load i)
             mov $1, (%rdx,%rcx,8)      (store into the heap)
```




# Run-time Checks

- Check if array/object references are non-null
- Check if array index is within bounds
  - if v holds the address of an array, insert array bounds checking code for v before each load `_ = v[i]` or store `v[i] = _`
  - Array length is stored just before array elements:

```
cmp $0, -24(%rbp)            (compare i to 0)
jl ArrayBoundsError23        (test lower bound)
mov -16(%rbp), %rcx          (load v into %ecx)
mov -8(%rcx), %rcx           (load array length into %ecx)
cmp -24(%rbp), %rcx          (compare i to array length)
jle ArrayBoundsError23       (test upper bound)
 ...
```

# Field Offsets

- Offsets of fields from beginning of object known statically

```
struct Shape {
  field ll : Point /* 8 */
  field ur : Point /* 16 */
}
```

| |
|---|
| ll: Point |
| ur: Point |

# Method Arguments

- Receiver is (implicit) argument to method

```
struct A {
  method f(x: i64,
           y: i64 ) -> unit {
}
```

compile as

```
fn f(self: A,
     x: i64,
     y: i64) -> unit {
}
```

# Code Generation: Calls

- Pre-function-call code:
  - Save registers
  - Push parameters
  - call function by its label

- Pre-method call:
  - Save registers
  - Push parameters
  - *Push receiver object reference*
  - *Lookup method in vtable*

# Example call: using declared fn name

```
fn foo(x: i64, y:i64) -> i64 { ... }

...

      foo(2,3)
```

```
movq $3, -8(%rsp)
movq $2, -16(%rsp)
subq  $16, %rsp
callq foo
addq  $16, %rsp
```

**Direct call because function name is known statically.**

**Uses redzone to setup stack args instead of push.**

# Example call: using expression of fn type

```
fn foo(x: i64, y:i64) -> i64 { ... }
...
  let f:(x: i64, y:i64) -> i64 = foo in {
```

```
f(2,3)
```

```
movq $3, -8(%rsp)
movq $2, -16(%rsp)
movq 24(%rsp), %rax
subq  $16, %rsp
callq *%rax
addq  $16, %rsp
```

**Indirect call because function address is only known dynamically.**

# Code Generation: Library Calls

- Use x86-64/Linux ABI calling convention

- Warning: library functions
  may modify caller save registers

```
# Roost.printi64(301)
movq  $301, %rdi
callq __LIB_printi64


# Roost.random(301)
movq $301, %rdi
call  __LIB_random
movq %rax, -32(%rbp)
```

# Code Generation: Allocation

- Heap allocation: o = new C()
  - Allocate heap space for object

```
movq  $24, %rdi # 3 fields
callq __LIB_allocateObject
# reference in %rax
```