Machine Level Programming: Procedures and Arrays

CSCI 237: Computer Organization
12th Lecture, Mar 7, 2025

Jeannie Albrecht

Administrative Details

- Lab 2:
  - All phases due next week
  - You really should be finished with the first three phases at this point (the last two are a little trickier)
  - If you're finished, check out phase 6 and secret phase
  - If you finished that, too, try another bomb! ☺
- HW 3 due Friday

Last Time

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Register saving conventions

Today

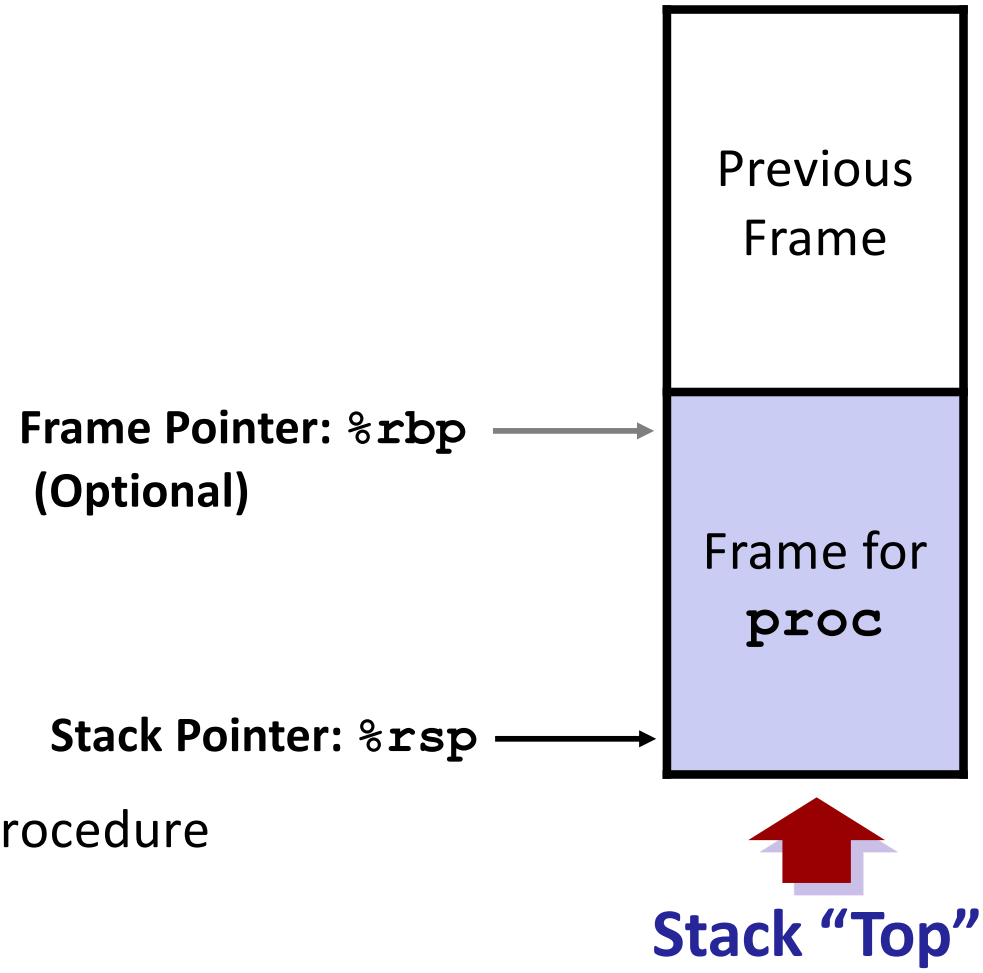
- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Register saving conventions
- Arrays (Ch 3.8)
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level

Stack-Based Languages

- Languages that support recursion
  - e.g., C, Java, Python
  - Code must be “reentrant”
    - Allows for multiple simultaneous instantiations of single procedure
  - Thus we need some place to store state (data) of each *instantiation*
    - Arguments, local variables, return address pointer
- Stack discipline
  - State (or data) for given procedure is only needed for limited time
    - From when procedure is called to when its return occurs
  - **Callee** (function being called) returns before **caller** (function calling) does
- Stack space allocated in frames
  - Stores state for single procedure instantiation

Stack Frames

- Contents
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)
- Management
  - Space allocated when entering procedure
    - "Set-up" code
    - Includes push by **`call`** instruction
  - Space deallocated when return occurs
    - "Finish" code
    - Includes pop by **`ret`** instruction



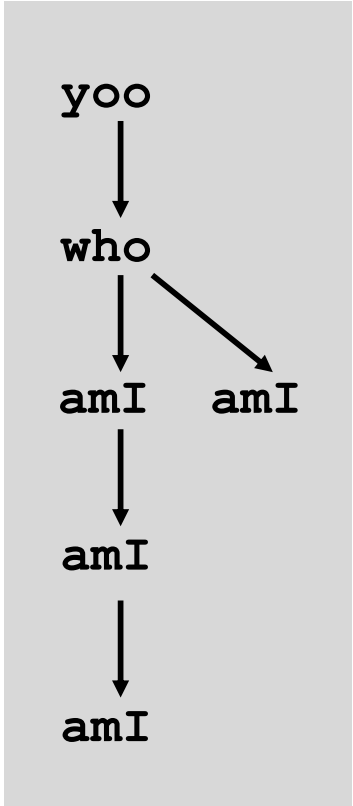
Call Chain Example

```
yoo(…) {
   •
   •
   who();
   •
   •
}
```

```
who(…) {
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…) {
   •
   •
   amI();
   •
   •
}
```

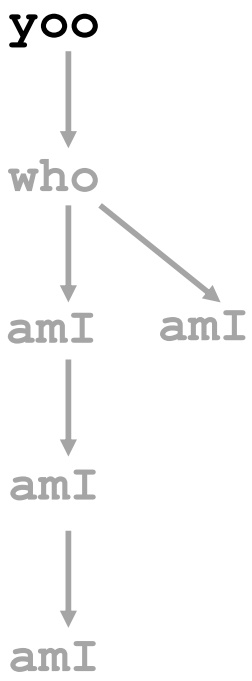
Example Call Chain

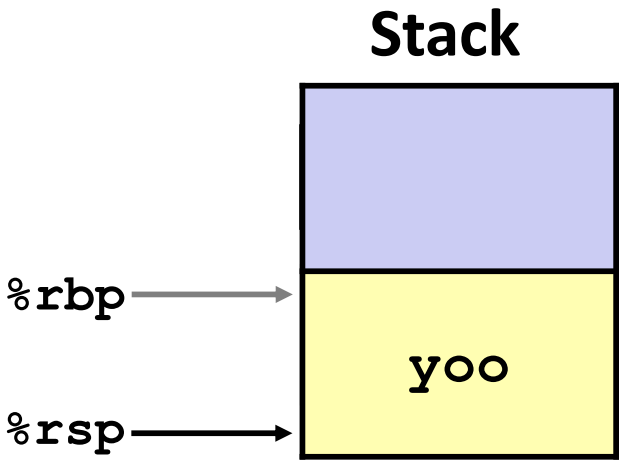


Note: Procedure `amI()` is recursive!

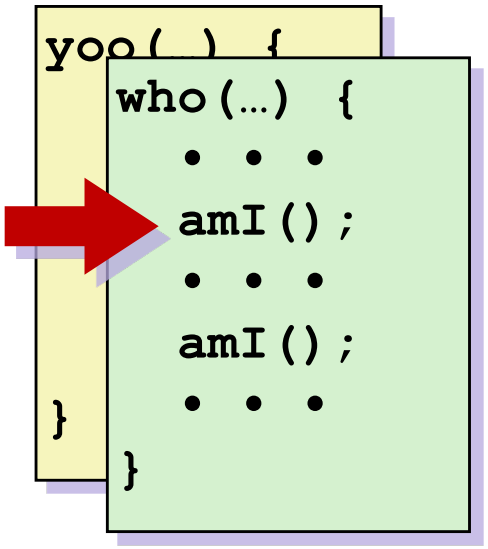
Example

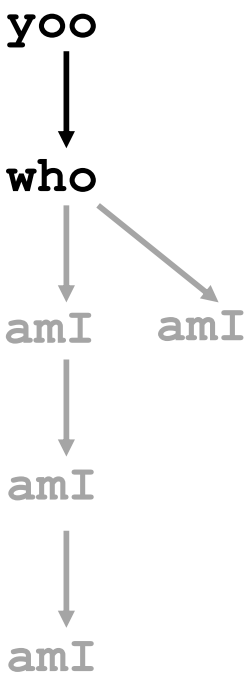
```
yoo(…)  {
    •
    •
    who();
    •
    •
}
```

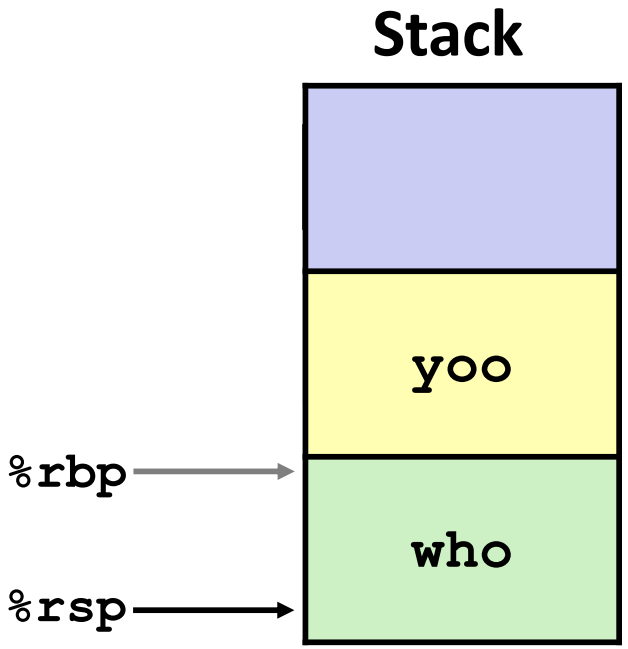




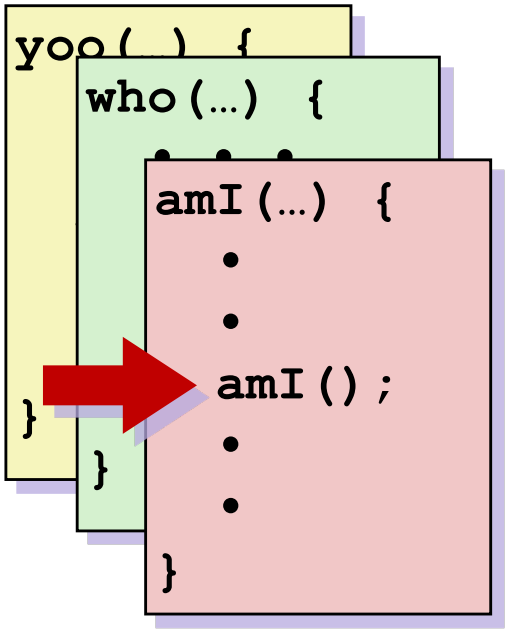
Example

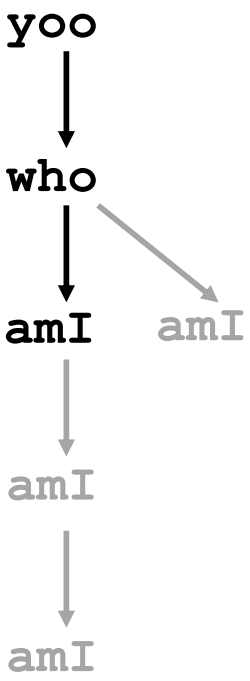


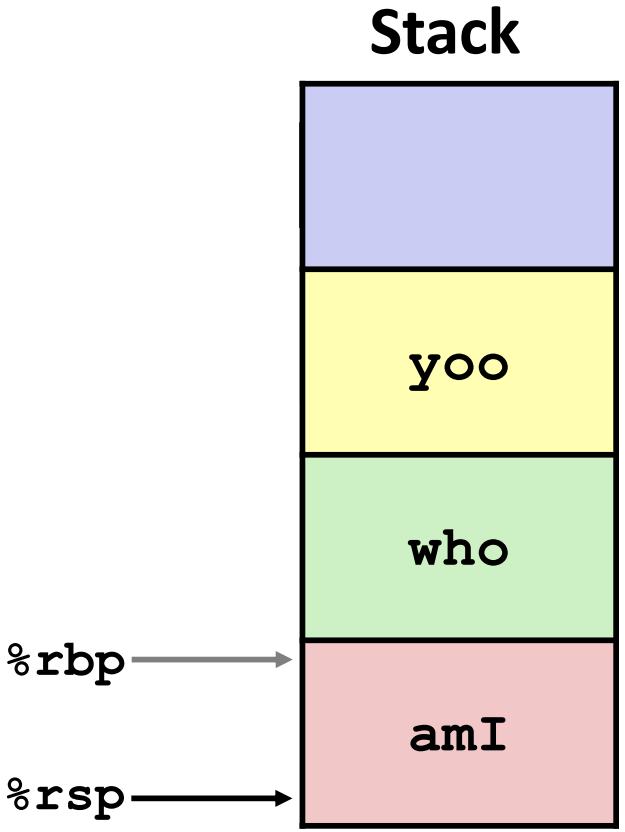




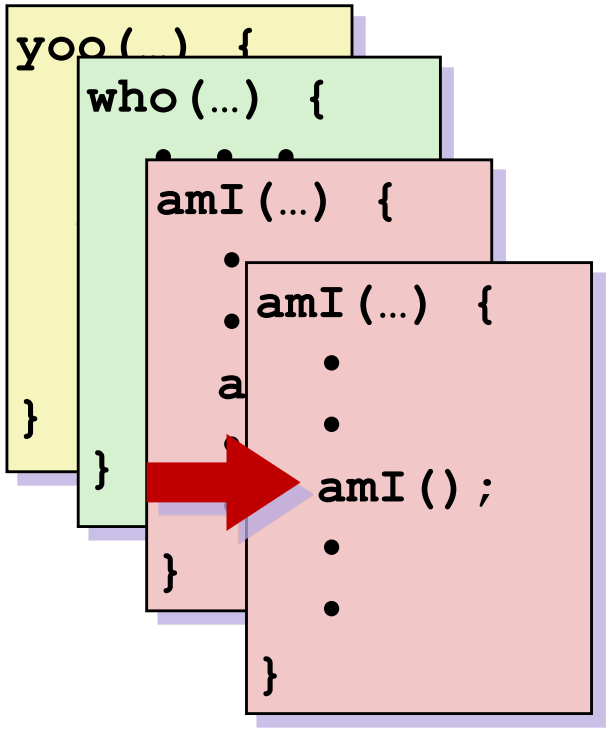
Example

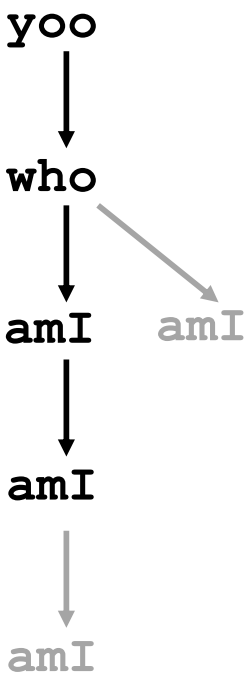


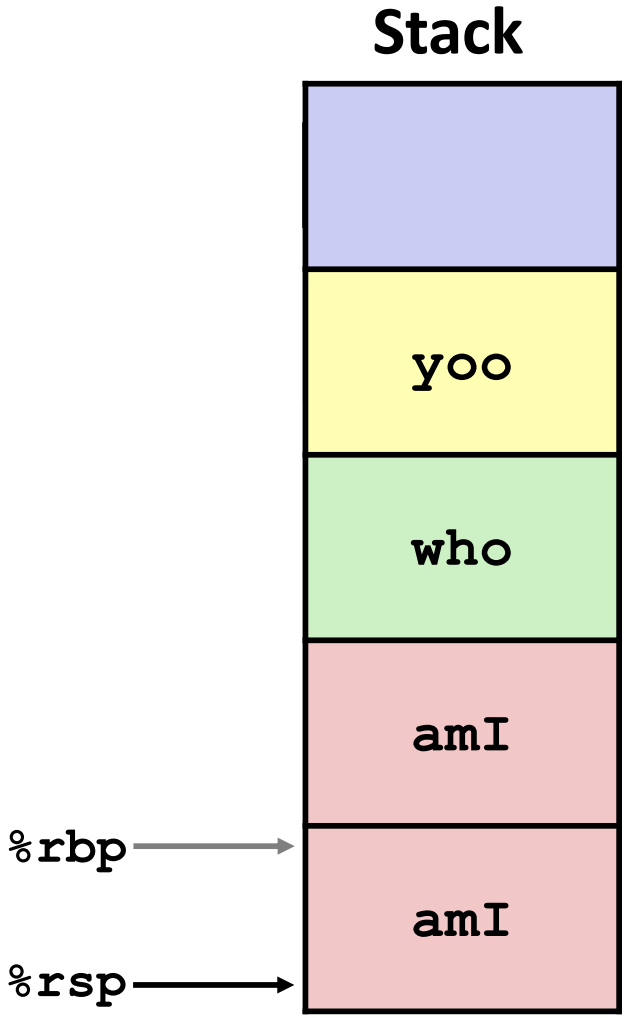




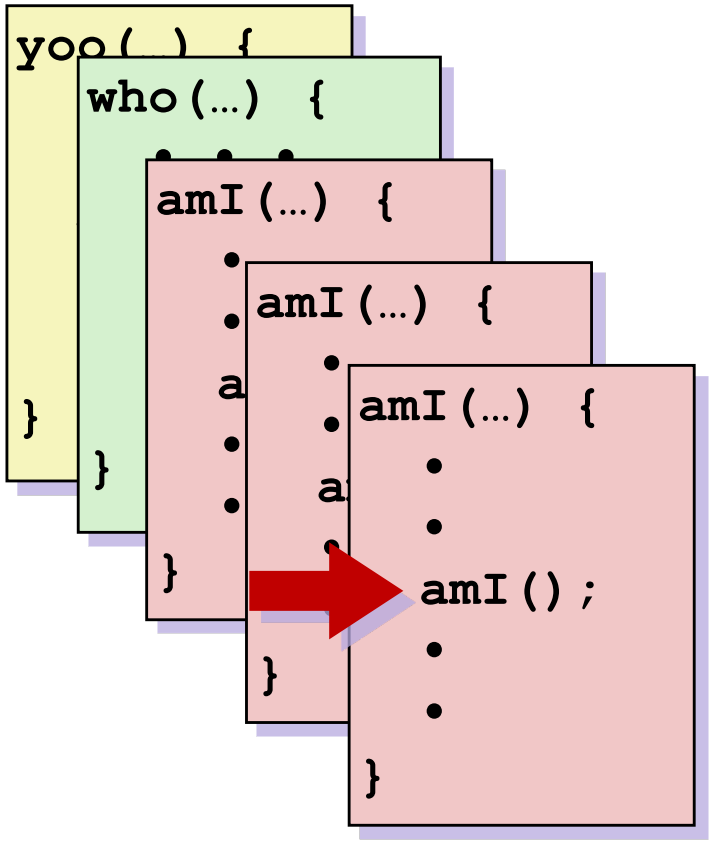
Example

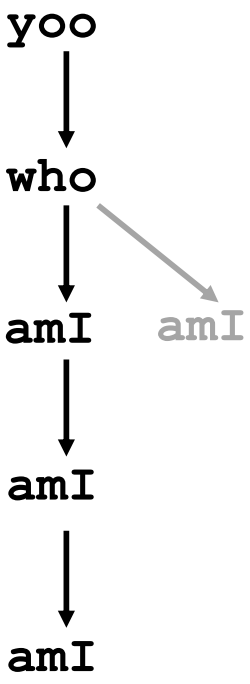

who(…) {
amI(…) {
amI(…) {
amI();
}

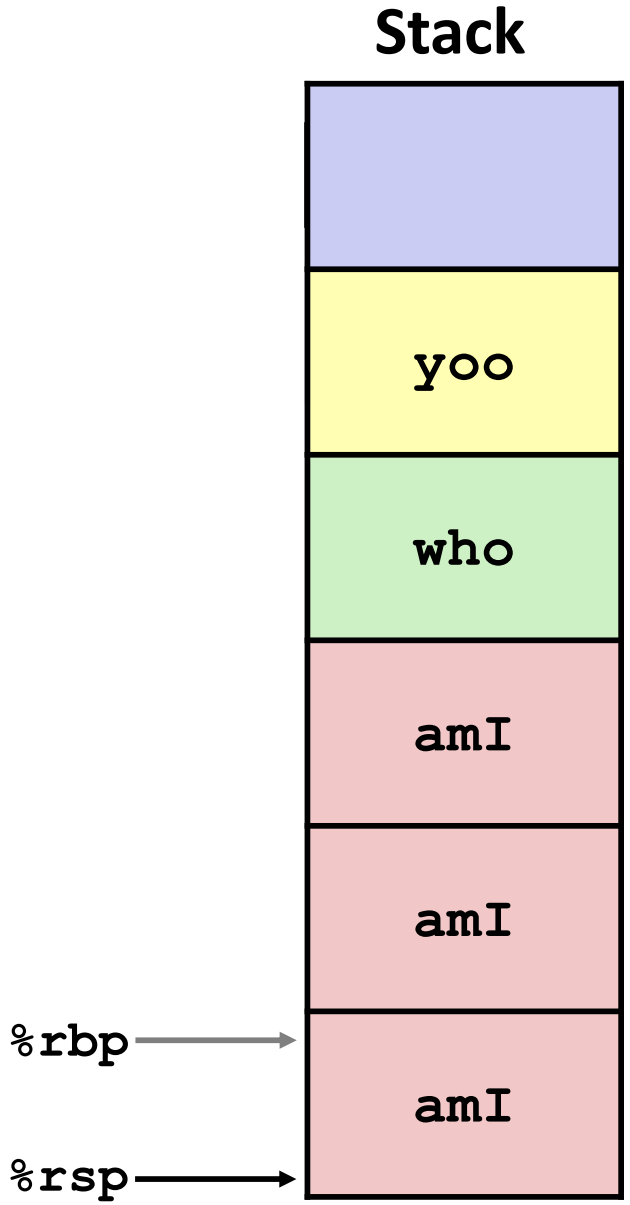

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

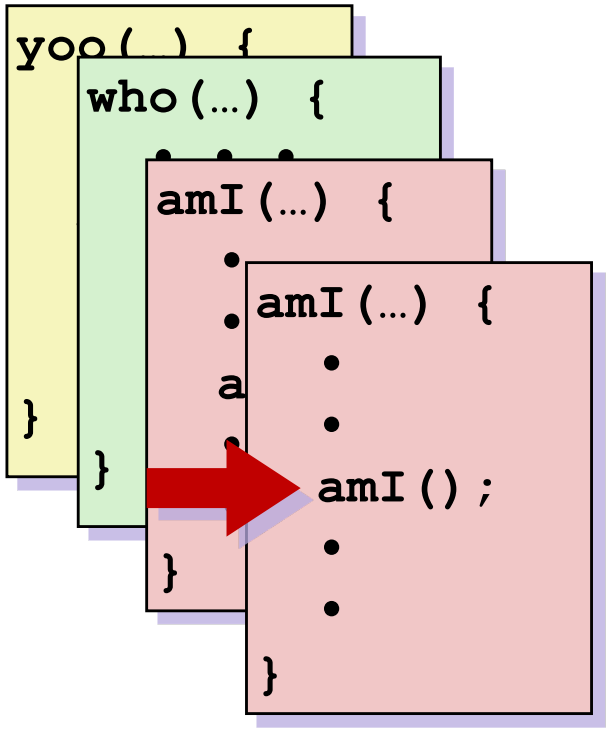
Example

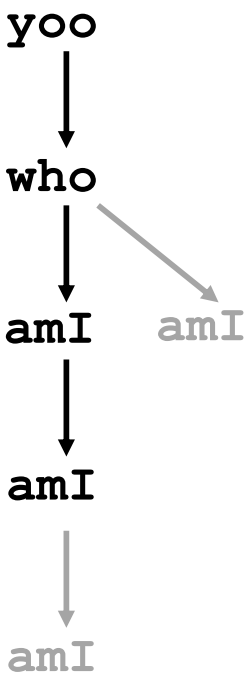

who(…) {
amI(…) {
amI(…) {
amI(…) {
amI();
}

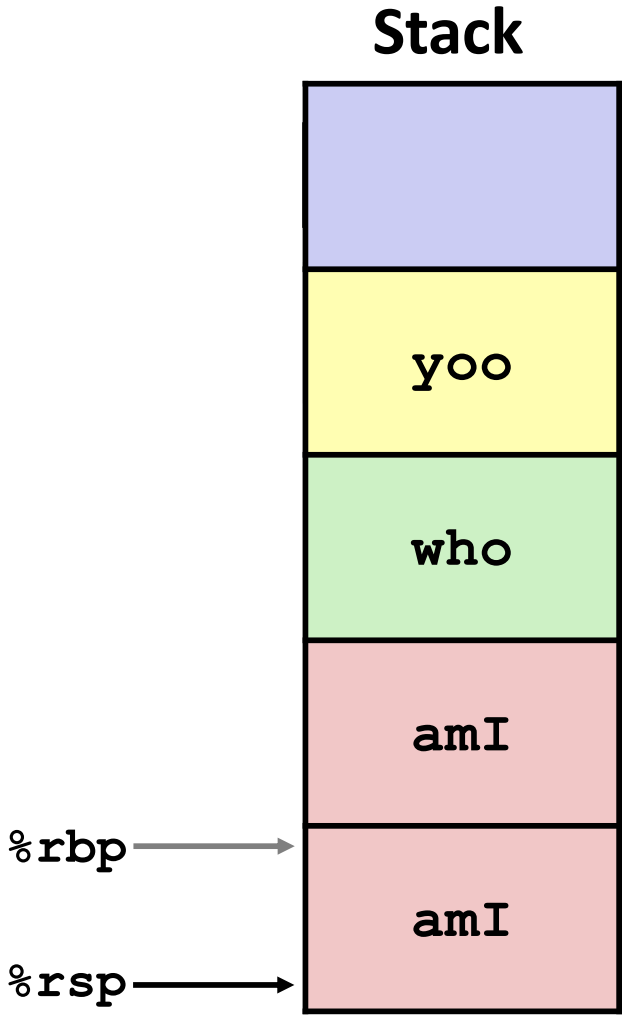

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
amI
%rbp
%rsp

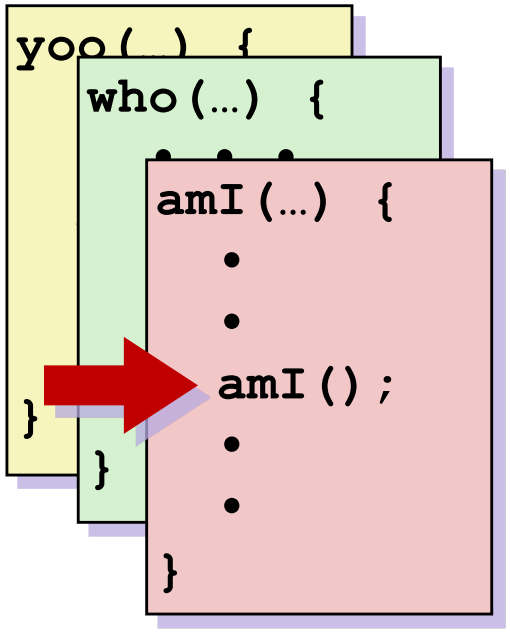
Example

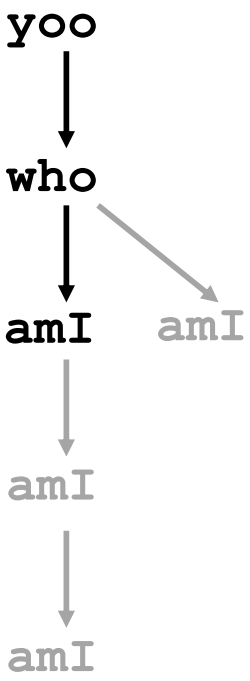

who(…) {
amI(…) {
amI(…) {
amI();
}

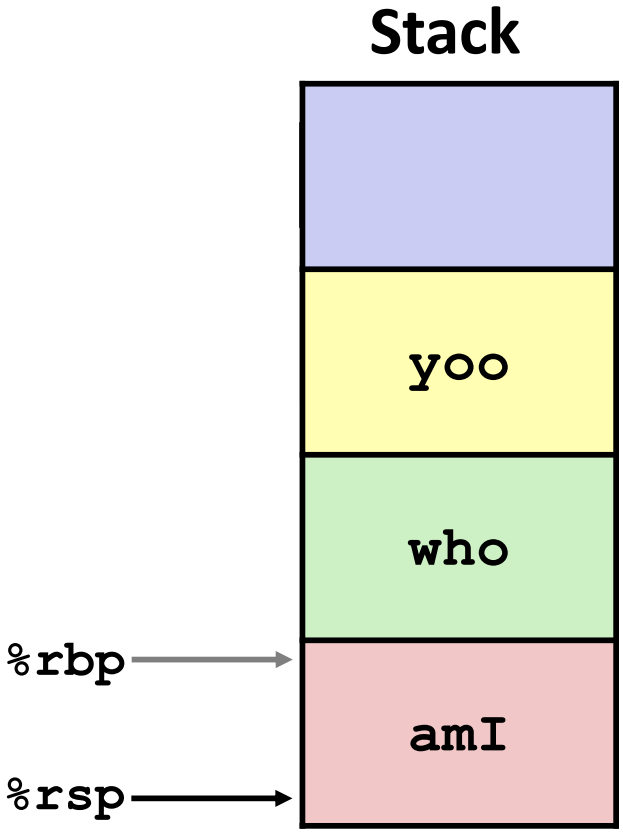

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

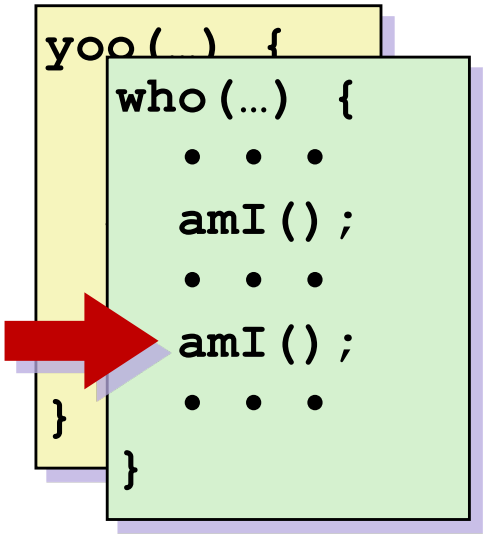
Example

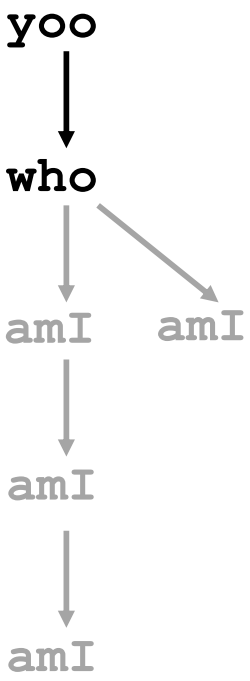


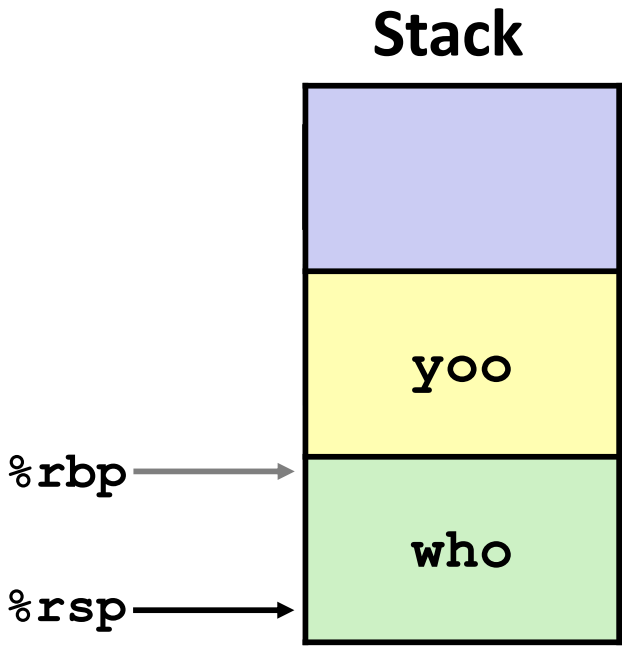




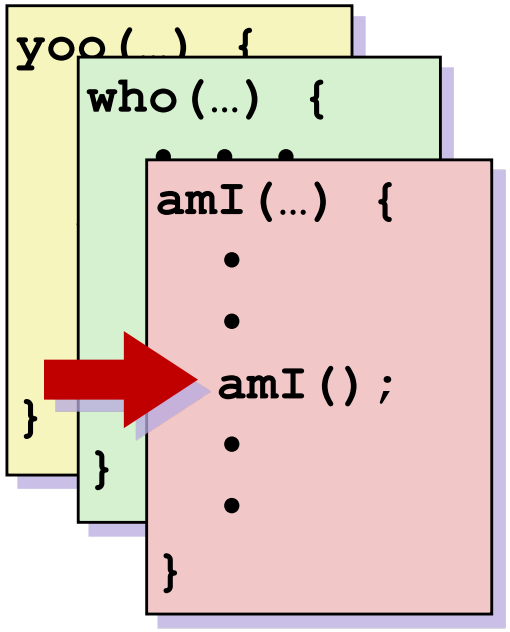
Example

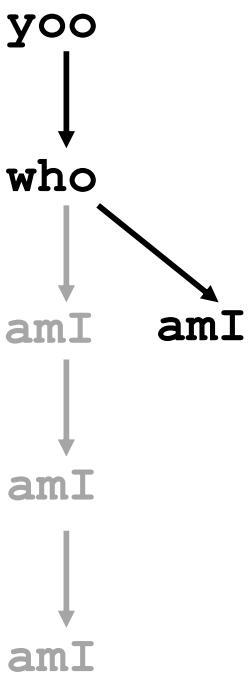


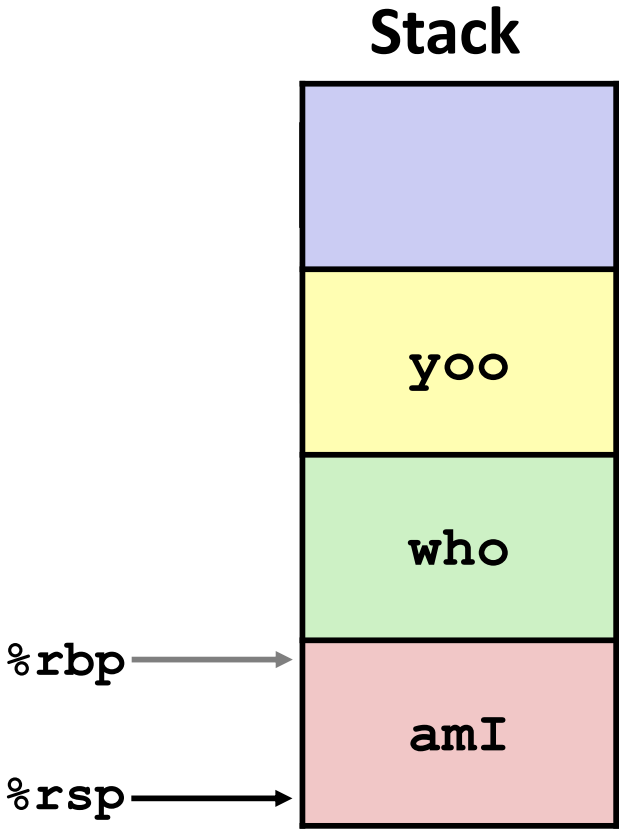




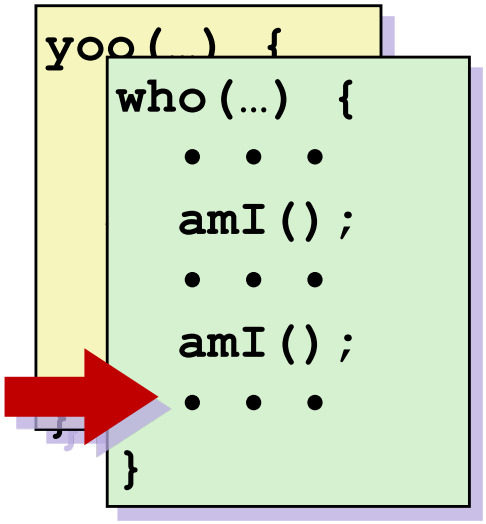
Example

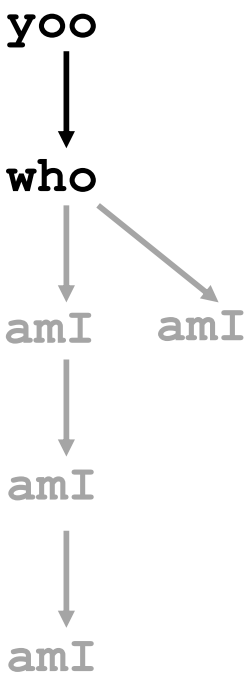


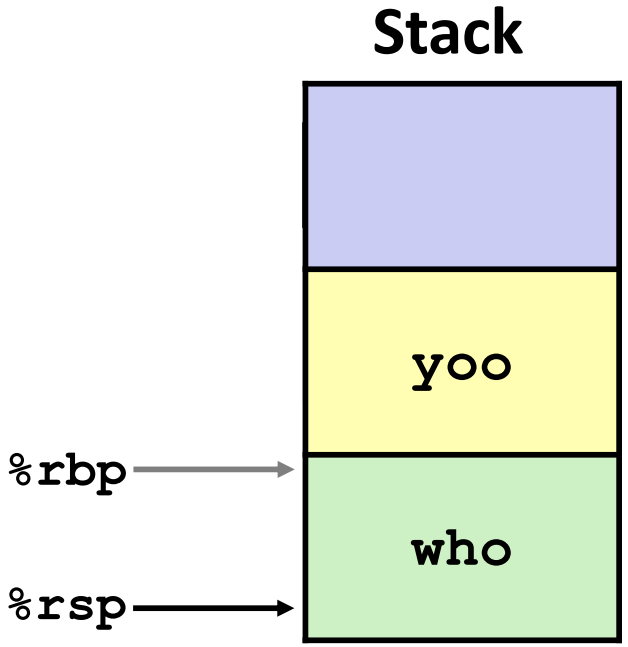




Example

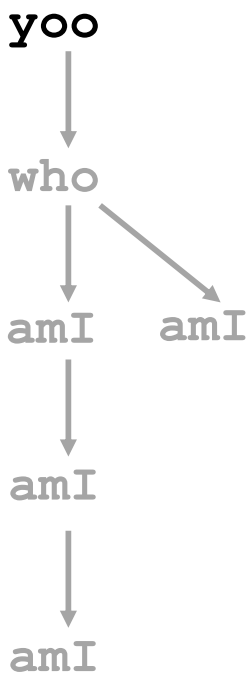


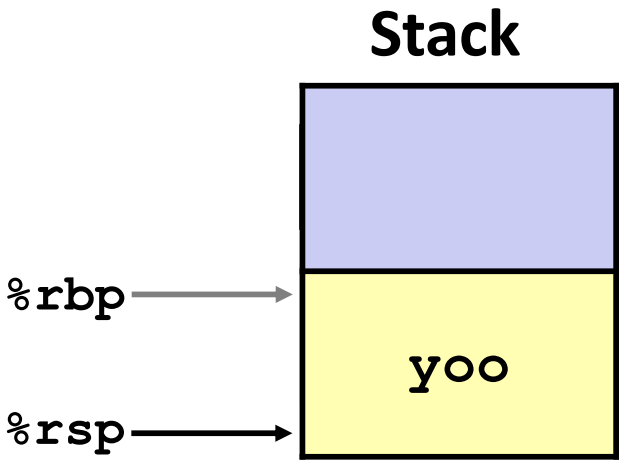




Example

```
yoo(…)  {
   •
   •
   who();
   •
   •
}
```

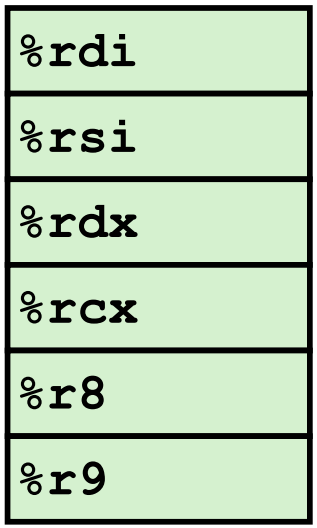




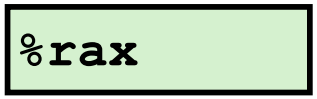
Procedure Data Flow

Registers

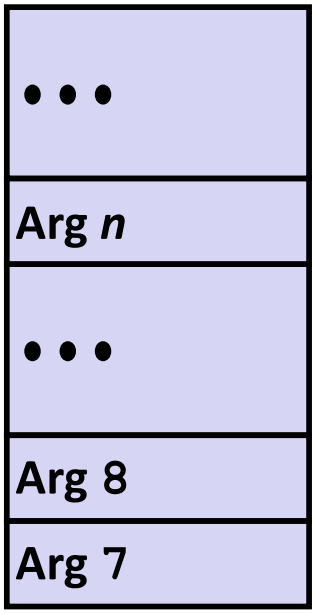
- First 6 arguments



- Return value



Stack



- Use registers or stack to "pass" data between functions
- But only allocate stack space when needed!

Data Flow Examples

```
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: `incr`

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

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

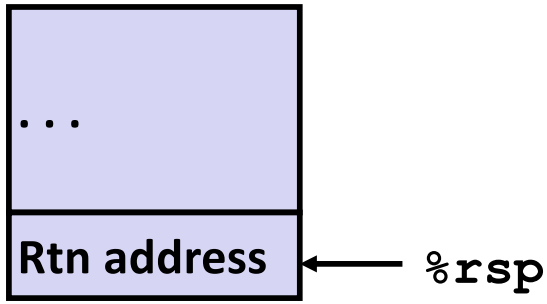
Register	Use(s)
`%rdi`	Argument **`p`**
`%rsi`	Argument **`val`**, **`y`**
`%rax`	**`x`**, Return value

Example: Calling **`incr`** #1

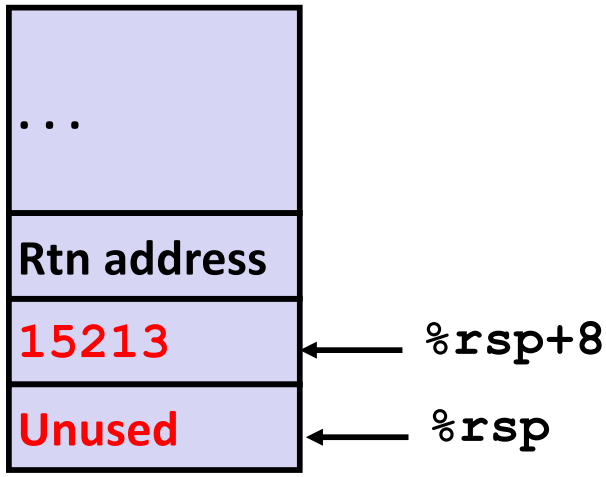
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
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
```

Initial Stack Structure



Resulting Stack Structure



Example: Calling **`incr`** #2

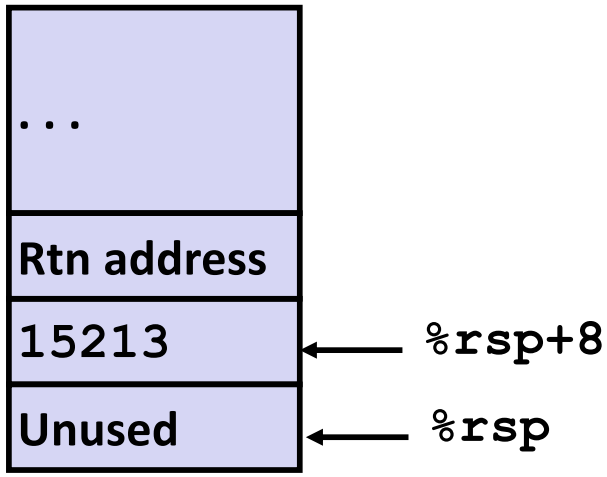
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

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

Register	Use(s)
%rdi	&v1 -> %rsp+8
%rsi	3000

```
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
```

Resulting Stack Structure



Example: Calling `incr` #3

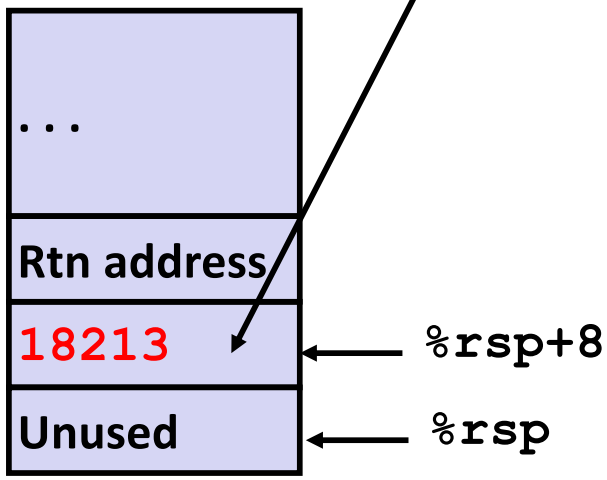
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
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
```

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

Register	Use(s)
%rdi	&v1
%rsi	3000, 18213
%rax	15213

Resulting Stack Structure



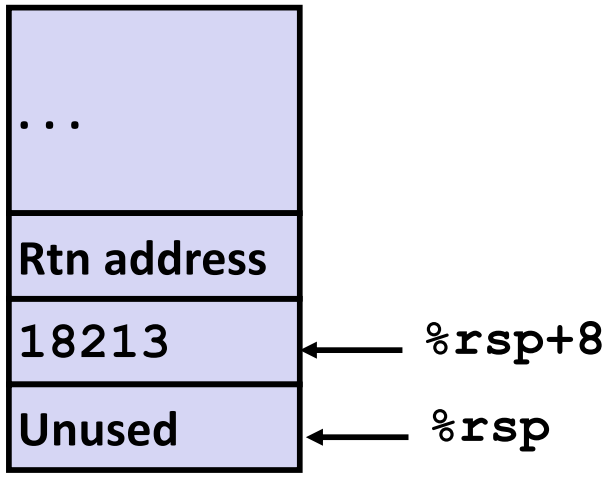
Example: Calling `incr` #4

```
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	33426 (Return value)

Resulting Stack Structure

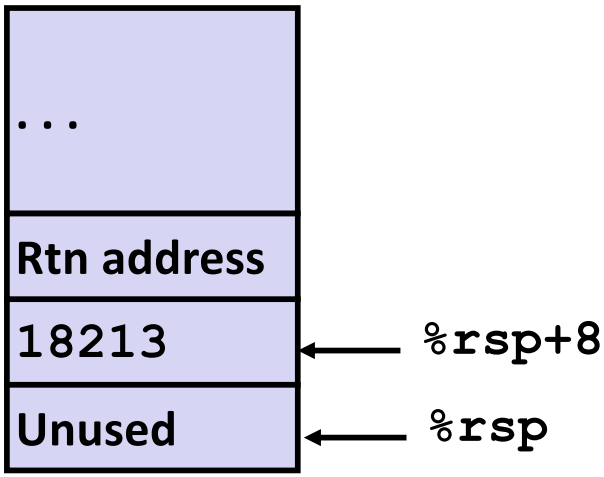


Example: Calling `incr` #5

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

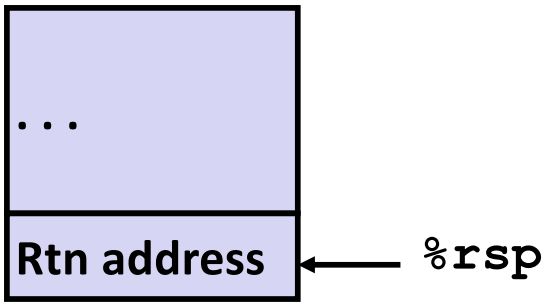
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Previous Stack Structure



Register	Use(s)
%rax	33426 (Return value)

Updated Stack Structure

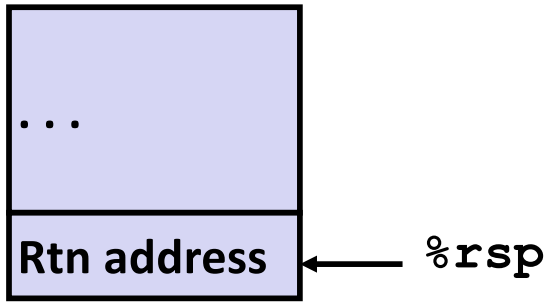


Example: Calling `incr` #6

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

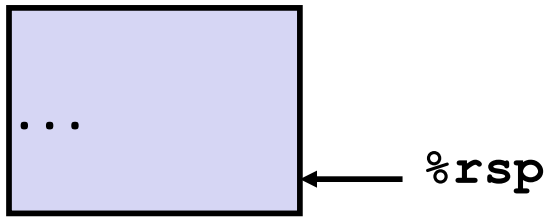
```
call_incr:
   subq     $16, %rsp
   movq     $15213, 8(%rsp)
   movl     $3000, %esi
   leaq     8(%rsp), %rdi
   call     incr
   addq     8(%rsp), %rax
   addq     $16, %rsp
   ret
```

Previous Stack Structure



Register	Use(s)
%rax	33426 (Return value)

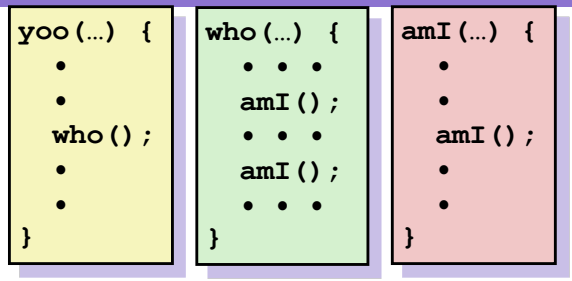
Final Stack Structure



Today

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Register saving conventions
- Arrays (Ch 3.8)
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level

Register Saving Conventions



- Recall recursive example. When procedure **`yoo`** calls **`who`**:
  - **`yoo`** is the **caller** (function calling the next function)
  - **`who`** is the **callee** (function being called)
- Can registers safely be used for temporary storage of data?

```
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 problematic → something should be done
    - Need some coordination between caller and callee

Register Saving Conventions

```
yoo(...) {
   •
   •
   who();
   •
   •
}
```

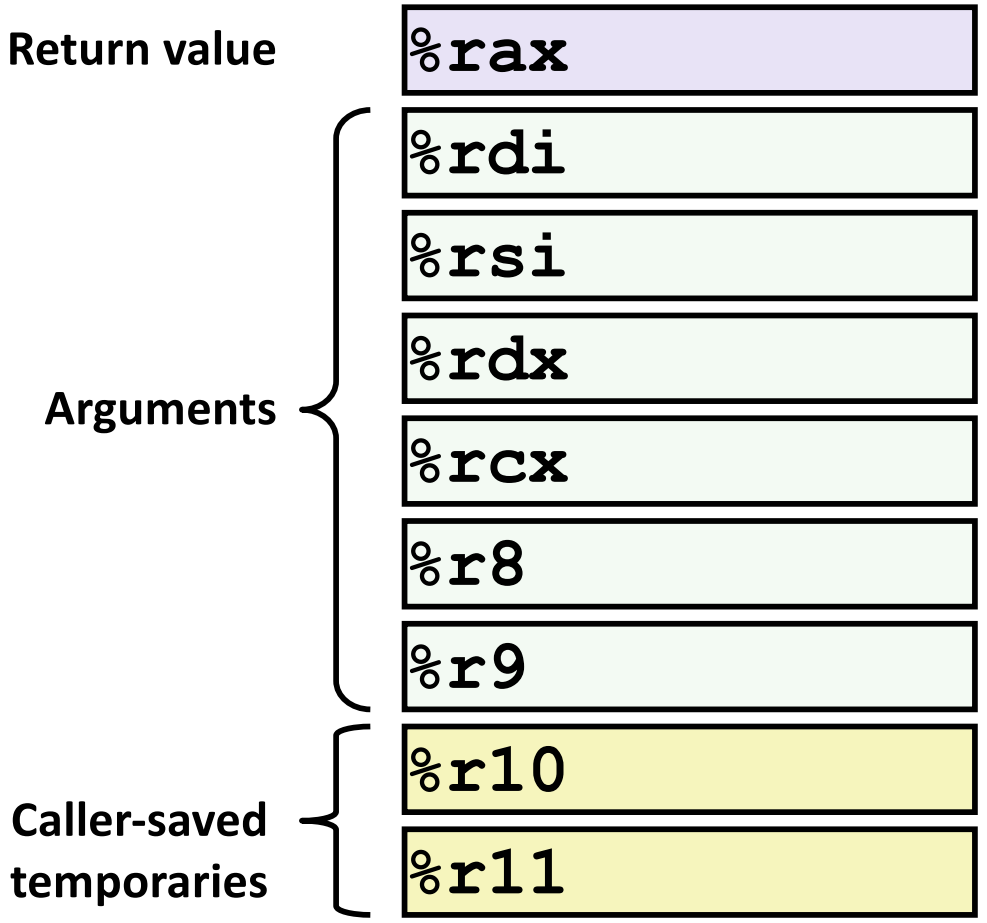
```
who(...) {
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(...) {
   •
   •
   amI();
   •
   •
}
```

- Recall recursive example. When procedure **`yoo`** calls **`who`**:
  - **`yoo`** is the **caller** (function calling the next function)
  - **`who`** is the **callee** (function being called)
- Can registers safely be used for temporary storage of data?
- Conventions
  - "Caller Saved"
    - Caller saves temporary values in its stack frame before the function call
  - "Callee Saved"
    - Callee saves temporary values in its frame before using
    - Callee restores them before returning to caller function

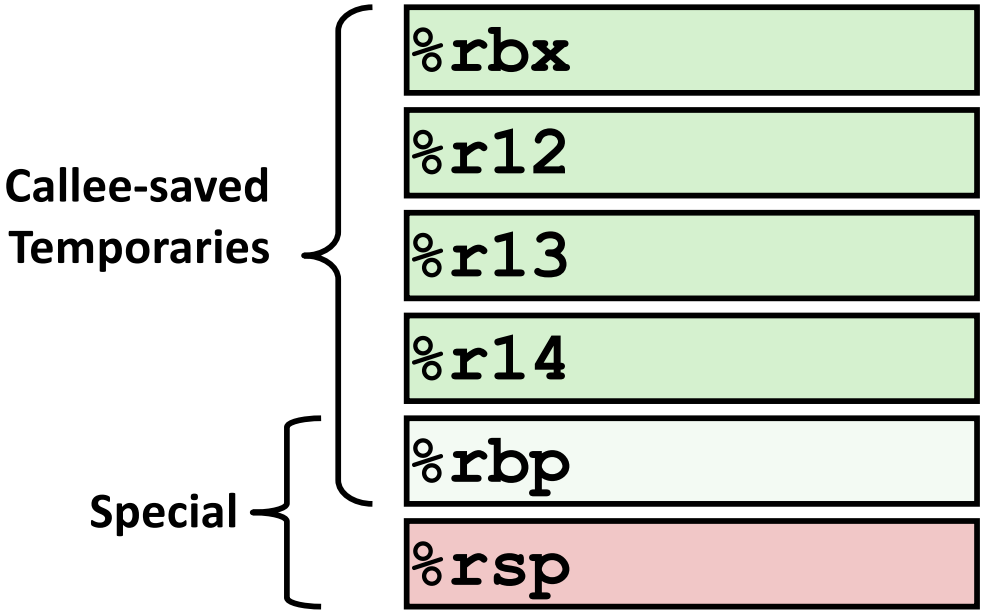
x86-64 Linux Register Usage #1: Caller-saved

- **`%rax`**
  - Return value
  - Caller-saved
  - Can be modified by procedure
- **`%rdi`, …, `%r9`**
  - Arguments
  - Caller-saved
  - Can be modified by procedure
- **`%r10`, `%r11`**
  - Caller-saved
  - Can be modified by procedure



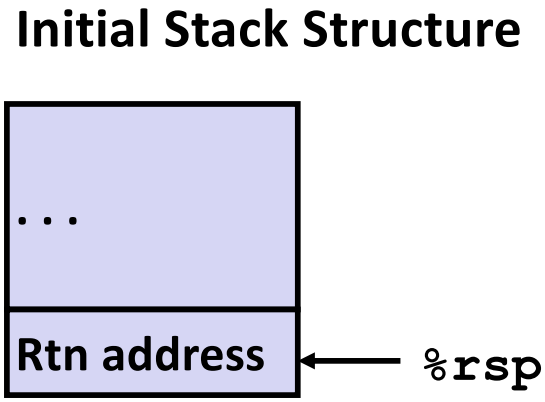
x86-64 Linux Register Usage #2: Callee-saved

- **`%rbx`, `%r12`, `%r13`, `%r14`**
  - Callee-saved
  - Callee must save & restore
- **`%rbp`**
  - Callee-saved
  - Callee must save & restore
  - May be used as frame pointer
- **`%rsp`**
  - Special form of callee-saved
  - Restored to original value upon exit from procedure



Callee-Saved Example #1

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



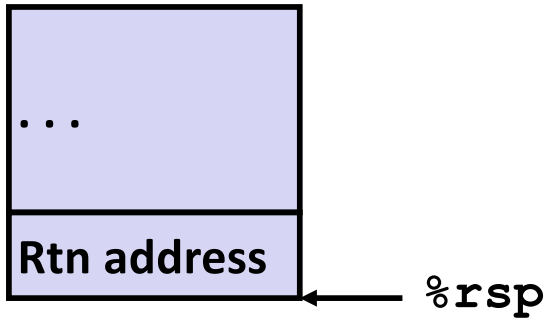
- x stored in register `%rdi`.
- `%rdi` is a *caller-saved* register.
- Must put `&v1` in `%rdi` before the call to `incr`.
- But we still need x after the call to `incr`
- Where should we put x, so we can use it after the call to `incr`?
  - Save on stack or in a *callee-saved* register
  - Let's look at the latter scenario (save in register)

Callee-Saved Example #2

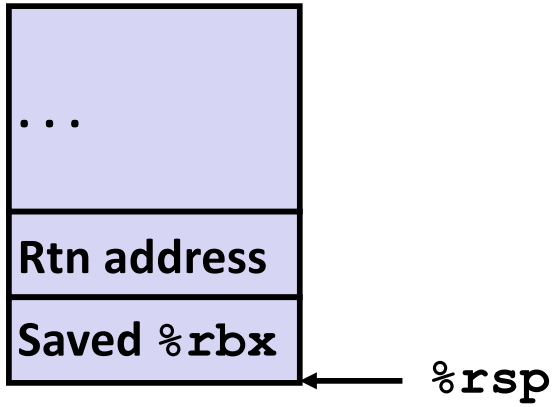
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx  #callee-saved
  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
```

Initial Stack Structure



Resulting Stack Structure

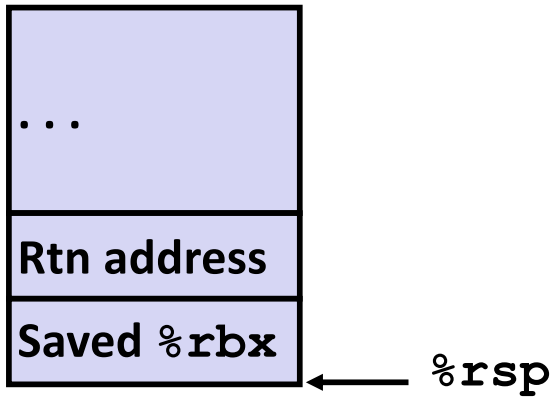


Callee-Saved Example #3

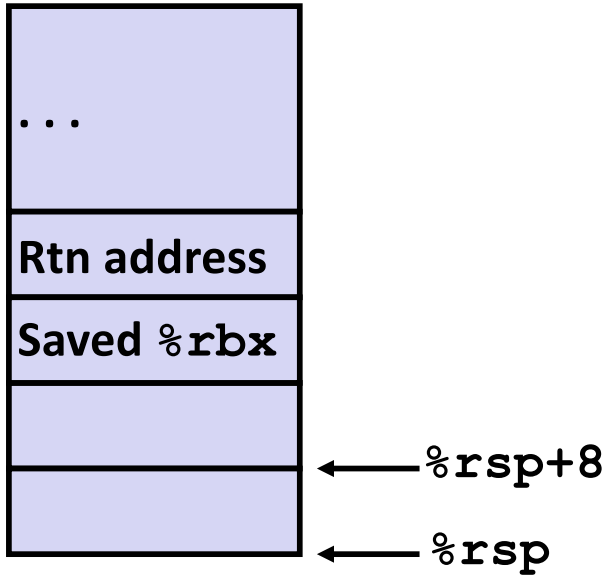
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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
```

Previous Stack Structure



Resulting Stack Structure

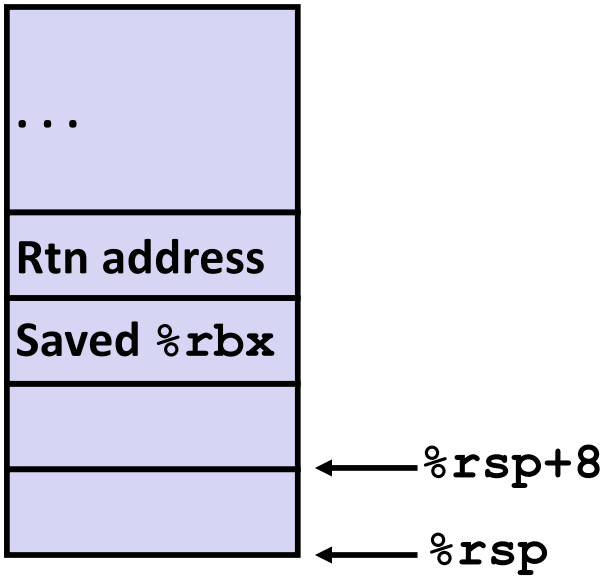


Callee-Saved Example #4

```
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
```

Stack Structure



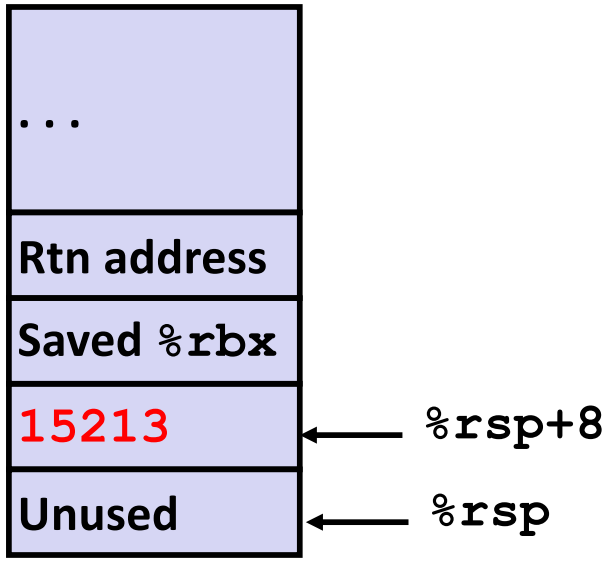
- x now saved in `%rbx`, a *callee-saved* register

Callee-Saved Example #5

```
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
```

Stack Structure



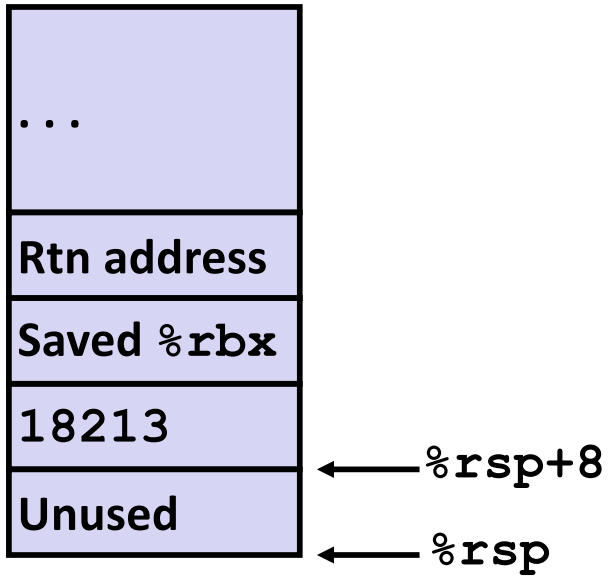
- x now saved in `%rbx`, a *callee-saved* register

Callee-Saved Example #6

```
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
```

Stack Structure



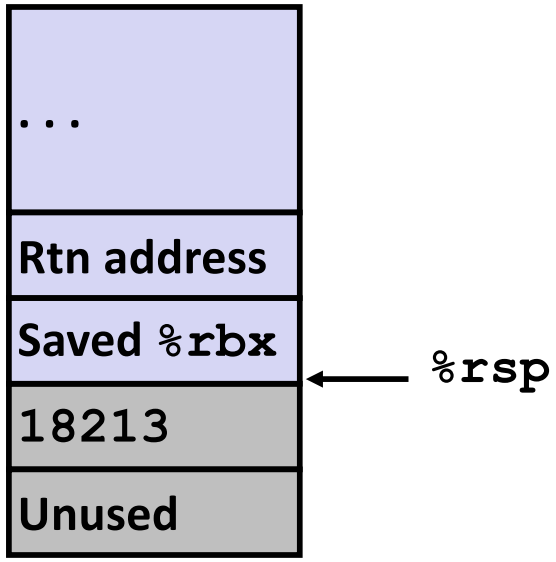
- After `incr`, x is still safely saved in `%rbx`
- Return result from `incr` in `%rax`

Callee-Saved Example #7

```
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
```

Stack Structure

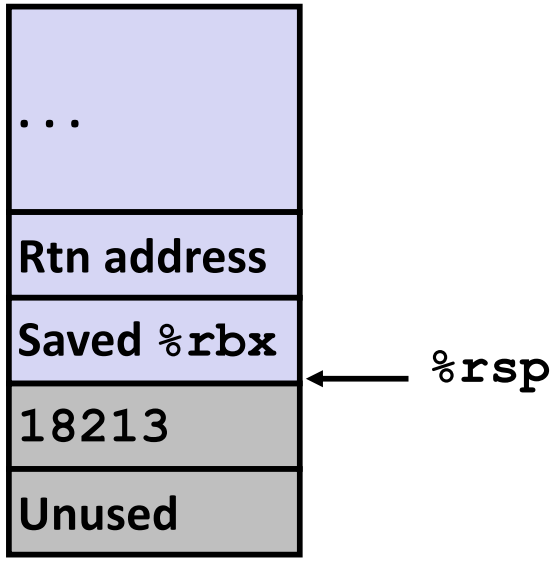


Callee-Saved Example #8

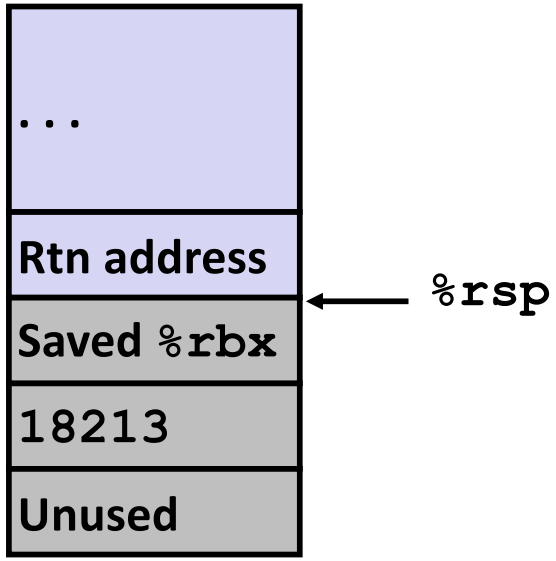
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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 #restore %rbx
  ret
```

Previous Stack Structure



Final Stack Structure

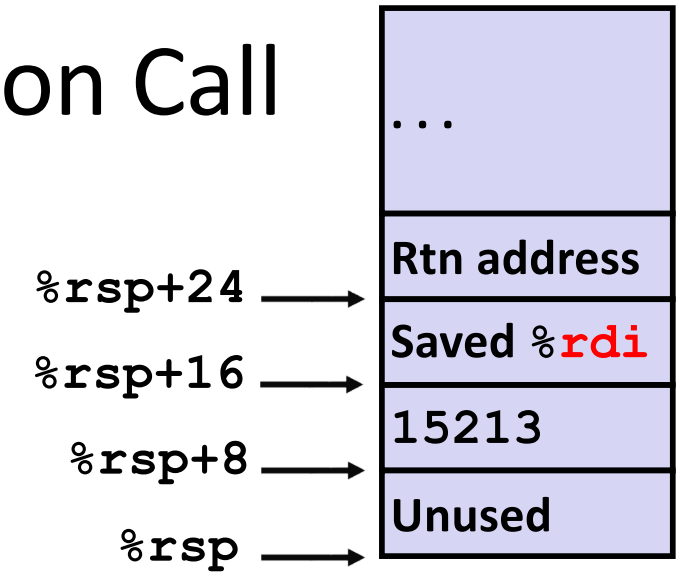


Summary: Saving Values Before Function Calls

- If there is a value in a **caller-saved** register X before a function call:
  - If the value is **not needed** after the function call, don't do anything!
  - If the value is **needed** after the function call, you need to **either**:
    - Save on stack
      - Store value in X on to the stack and then restore value to X from the stack after function call
    - Save in callee-saved register
      - Store value in **callee-saved** register Y to stack, copy value in X to Y, use Y after the function call returns, restore original value of Y by popping value from stack to Y *before* returning (see %rbx in previous example)

Saving Values Before Function Call

```
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
```

Save x in register

```
call_incr2:
  pushq   %rdi
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  movq    16(%rsp), %rdi
  addq    %rdi, %rax
  addq    $24, %rsp
  ret
```

Save x on stack

Today

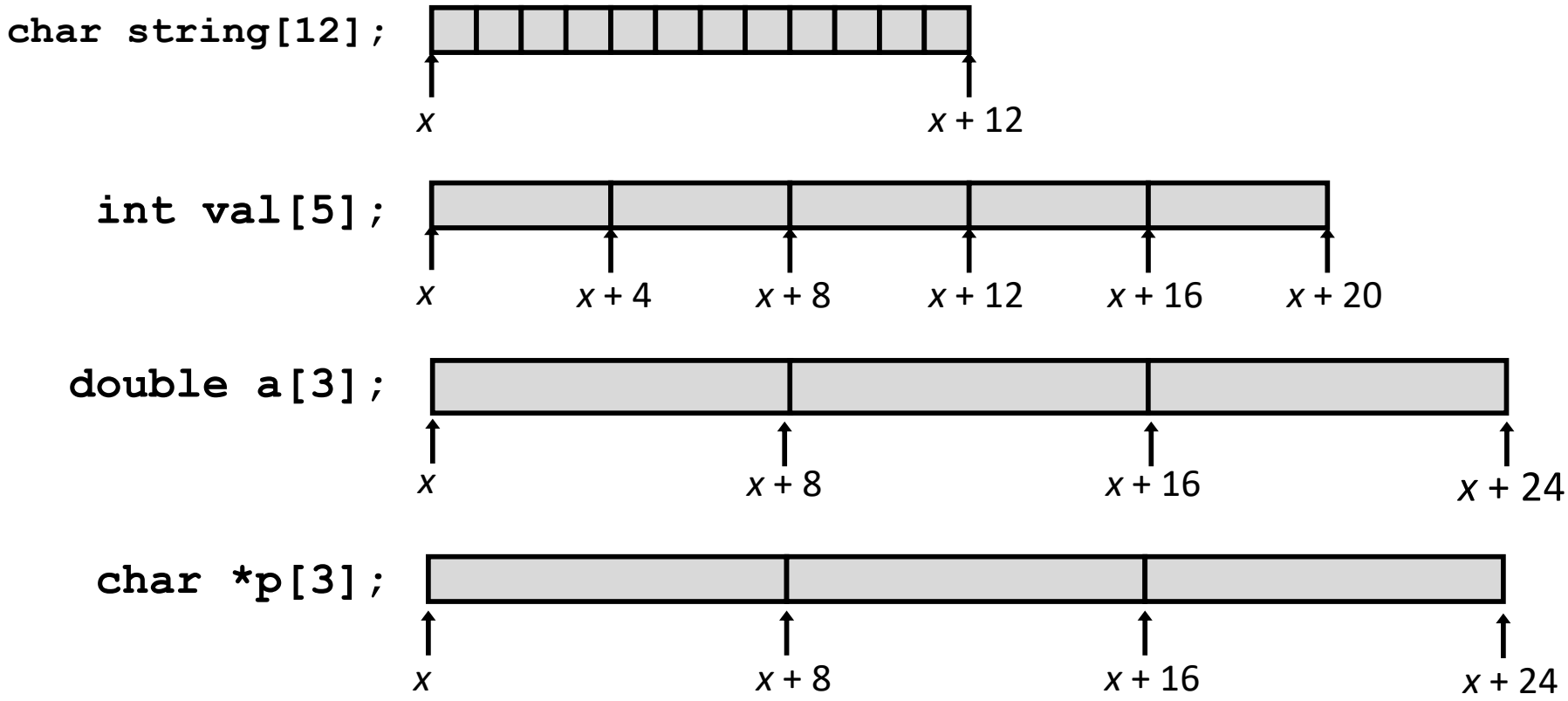
- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Register saving conventions
- Arrays (Ch 3.8)
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level

Ch 3.8 - Array Allocation

- Basic Principle

  T **`A[`***L***`];`**

  - Array A of data type *T* and length *L*
  - **Contiguously** allocated region of *L* * **`sizeof`**`(`*T*`)` bytes in memory
  - *x* is an address

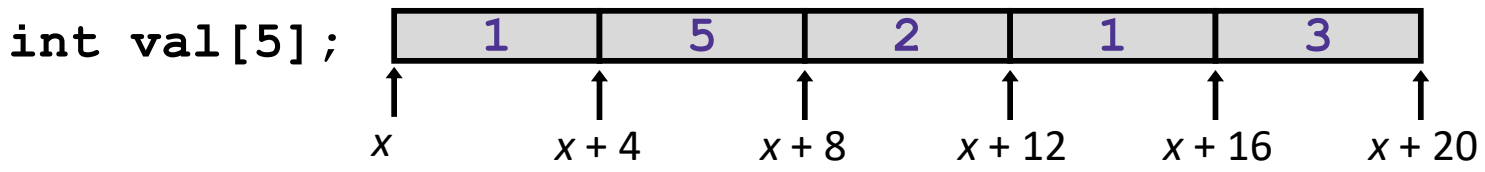


Array Access

- Basic Principle

  T **A**[*L*];

  - Array A of data type *T* and length *L*
  - Recall: Identifier **A** can be used as a pointer to array element 0: Type *T**

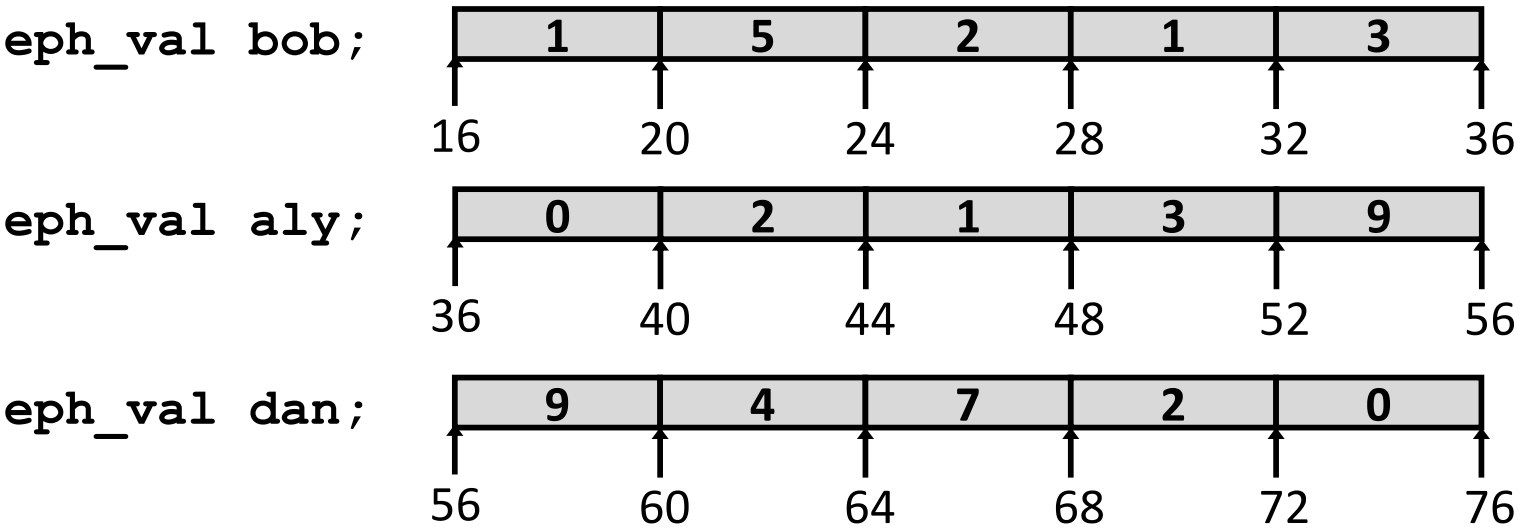


- Reference Type Value

Reference	Type	Value	
`val[4]`	`int`	`3`	
`val`	`int*`	`x`	
`val+1`	`int*`	`x + 4`	
`&val[2]`	`int*`	`x + 8`	
`val[5]`	`int`	`??`	
`*(val+1)`	`int`	`5`	`// val[1]`
`val + i`	`int*`	`x + 4 * i`	`// &val[i]`

Array Example

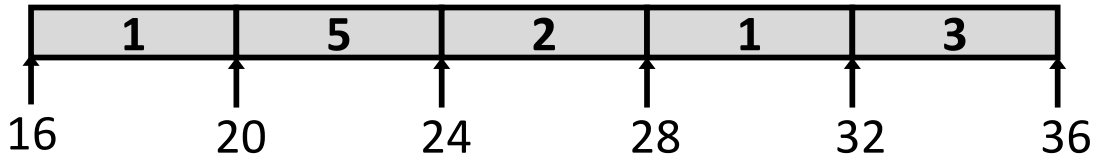
```
#define LEN 5
typedef int eph_val[LEN];

eph_val bob = { 1, 5, 2, 1, 3 };
eph_val aly = { 0, 2, 1, 3, 9 };
eph_val dan = { 9, 4, 7, 2, 0 };
```



- Declaration "`eph_val bob`" equivalent to "`int bob[5]`"
- Notes: Example arrays were allocated in *successive* 20 byte blocks in this scenario
  - Not guaranteed to happen in general

Array Accessing Example

```
eph_val bob;
```




```
int get_value
   (eph_val z, int value) {
   return z[value];
}
```

x86-64

```
# %rdi = z
# %rsi = value
movl (%rdi,%rsi,4), %eax  # z[value]
```

- Register `%rdi` contains starting address of array
- Register `%rsi` contains array index
- Desired value at `%rdi + 4*%rsi`
- Use memory reference `(%rdi,%rsi,4)`
- `D(base, index, S)` = `D + base + index*S`

Array Loop Example

```
#define LEN 5
typedef int eph_val[LEN];
```

```
void zincr(eph_val z) {
  size_t i;
  for (i = 0; i < LEN; i++)
    z[i]++;
}
```

```
  # %rdi = z
  movl    $0, %eax
  jmp     .L3
.L4:
  addl    $1, (%rdi,%rax,4)
  addq    $1, %rax
.L3:
  cmpq    $4, %rax
  jbe     .L4
  ret
```

Array Loop Example

```
#define LEN 5
typedef int eph_val[LEN];
```

```
void zincr(eph_val z) {
  size_t i;
  for (i = 0; i < LEN; i++)
    z[i]++;
}
```

```
  # %rdi = z
  movl    $0, %eax              #   i = 0
  jmp     .L3                   #   goto middle
.L4:                            # loop:
  addl    $1, (%rdi,%rax,4)     #   z[i]++
  addq    $1, %rax              #   i++
.L3:                            # middle:
  cmpq    $4, %rax              #   i:4
  jbe     .L4                   #   if <=, goto loop
  ret
```