



Machine-Level Programming III

15-213/14-513/15-513: Introduction to Computer Systems
5th Lecture, Sept 9, 2025

Mechanisms in Procedures

■ Passing control (last class)

- To beginning of procedure code
- Back to return point

■ Passing data

- Procedure arguments
- Return value

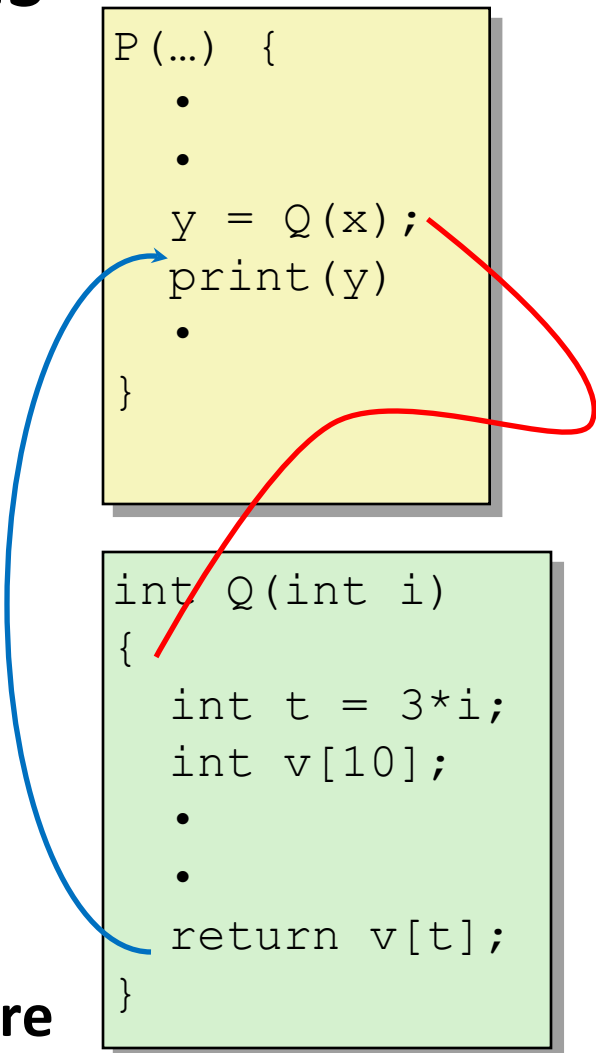
■ Memory management

- Allocate during procedure execution
- Deallocate upon return

■ Mechanisms all implemented with machine instructions

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

```
P (...) {  
    •  
    •  
    y = Q(x);  
    print(y)  
    •  
}
```



```
int Q(int i)  
{  
    int t = 3*i;  
    int v[10];  
    •  
    •  
    return v[t];  
}
```

Today

■ Procedure Calling Conventions (cont.)

CSAPP 3.7

- Passing data
- Managing local data

■ Arrays

CSAPP 3.8

- One-dimensional
- Multi-dimensional (nested)
- Multi-level

■ Structures

CSAPP 3.9

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

```
P (...) {  
    •  
    •  
    y = Q(x);  
    print(y)  
    •  
}
```

```
int Q(int i)  
{  
    int t = 3*i;  
    int v[10];  
    •  
    •  
    return v[t];  
}
```

Procedure Data Flow

Registers

Recall: registers have conventions

■ First 6 arguments

<code>%rdi</code>
<code>%rsi</code>
<code>%rdx</code>
<code>%rcx</code>
<code>%r8</code>
<code>%r9</code>

■ Return value

<code>%rax</code>

Stack

...
Arg <i>n</i>
...
Arg 8
Arg 7

■ 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: call    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:  ret                     # Return
```

Today

■ Procedure Calling Conventions (cont.)

- Passing data
- **Managing local data**

■ Arrays

- One-dimensional
- Multi-dimensional (nested)
- Multi-level

■ Structures

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

```
P (...) {  
    •  
    •  
    y = Q(x);  
    print(y)  
    •  
}
```

```
int Q(int i)  
{  
    int t = 3*i;  
    int v[10];  
    •  
    •  
    return v[t];  
}
```

Stack-Based Languages

■ Languages that support recursion

- e.g., C, Pascal, Java
- Code must be “*Reentrant*”
 - Multiple simultaneous instantiations of single procedure
- Need some place to store state of each instantiation
 - Arguments
 - Local variables
 - Return pointer

■ Stack discipline

- State for given procedure needed for limited time
 - From when called to when return
- Callee returns before caller does

■ Stack allocated in *Frames*

- state for single procedure instantiation

Call Chain Example

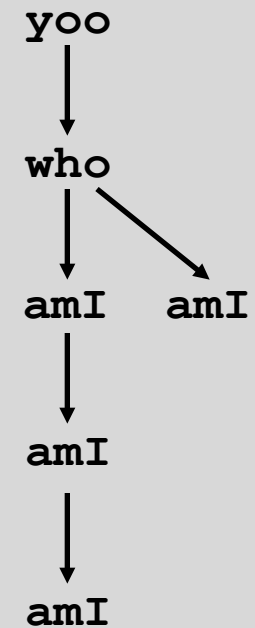
```
yoo (...)  
{  
  .  
  .  
  who ();  
  .  
  .  
}
```

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

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

Procedure amI () is recursive

Example Call Chain



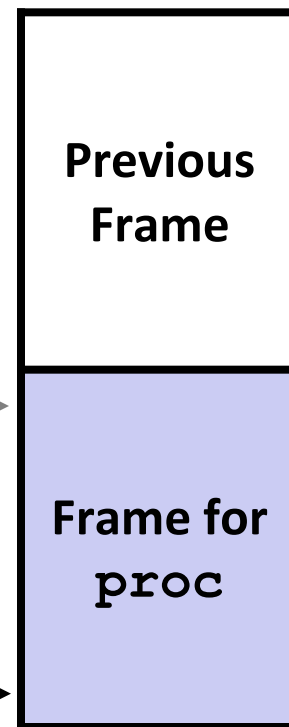
Stack Frames

■ Contents

- Return information
- Local storage (if needed)
- Temporary space (if needed)

Frame Pointer: `%rbp`
(Optional)

Stack Pointer: `%rsp`



■ Management

- Space allocated when enter procedure
 - “Set-up” code
 - Includes push by **call** instruction
- Deallocated when return
 - “Finish” code
 - Includes pop by **ret** instruction

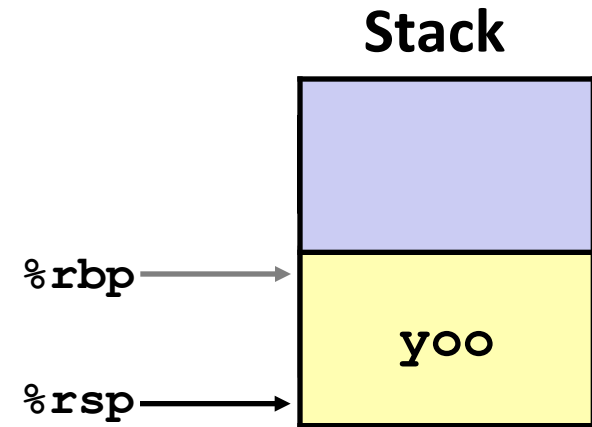

**Stack
“Top”**

Example

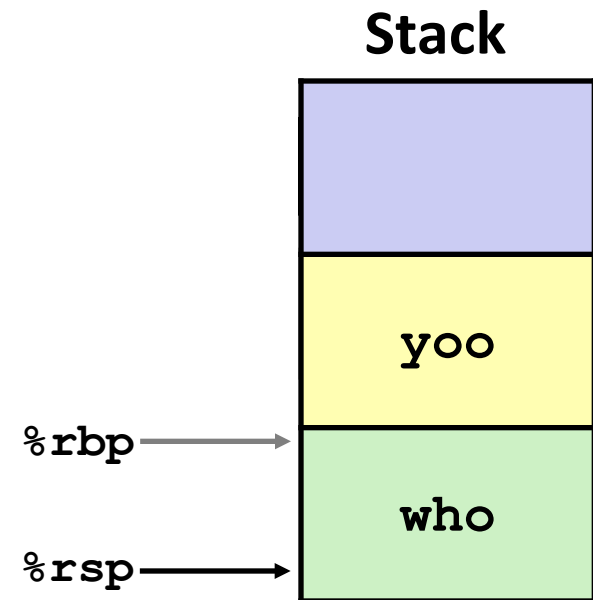
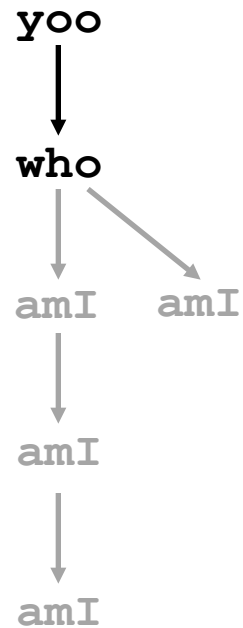
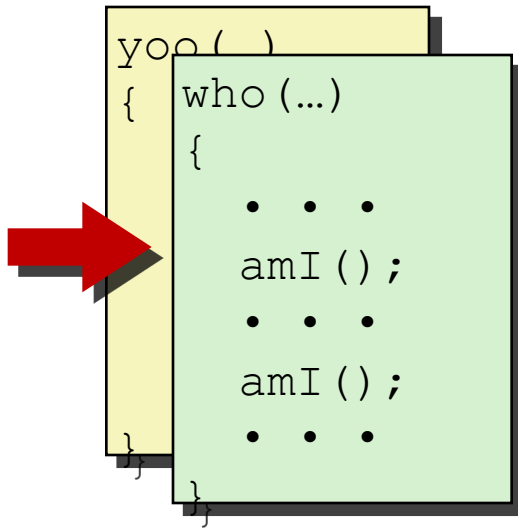


```
yoo (...)  
{  
  .  
  .  
  who ();  
  .  
  .  
}
```

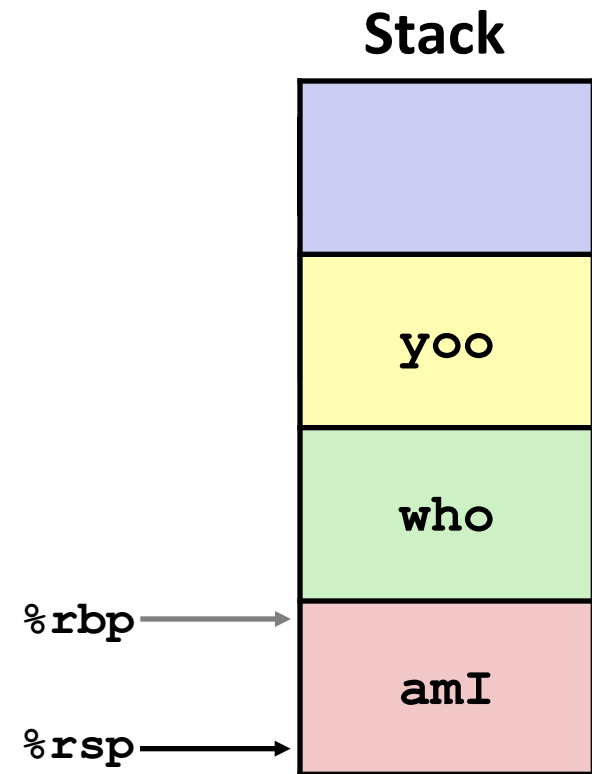
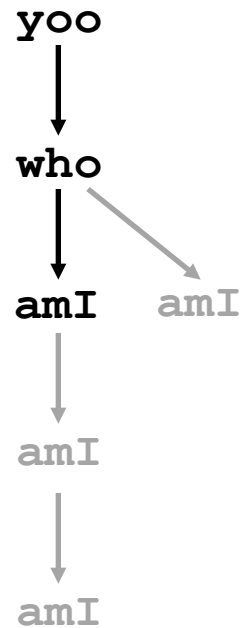
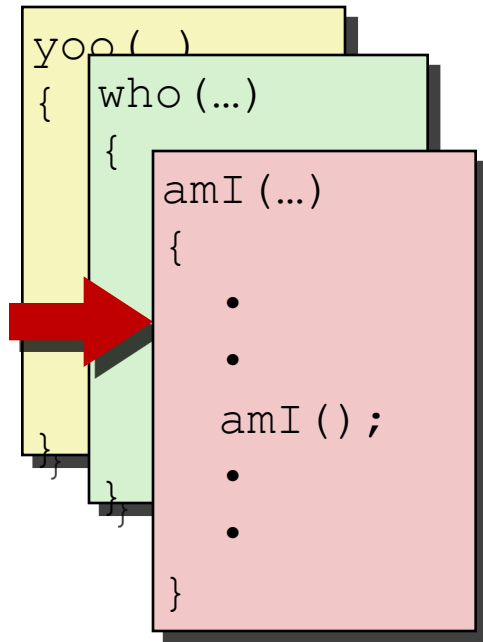
```
yoo  
  ↓  
who  
  ↓  ↘  
amI  amI  
  ↓  
amI  
  ↓  
amI
```



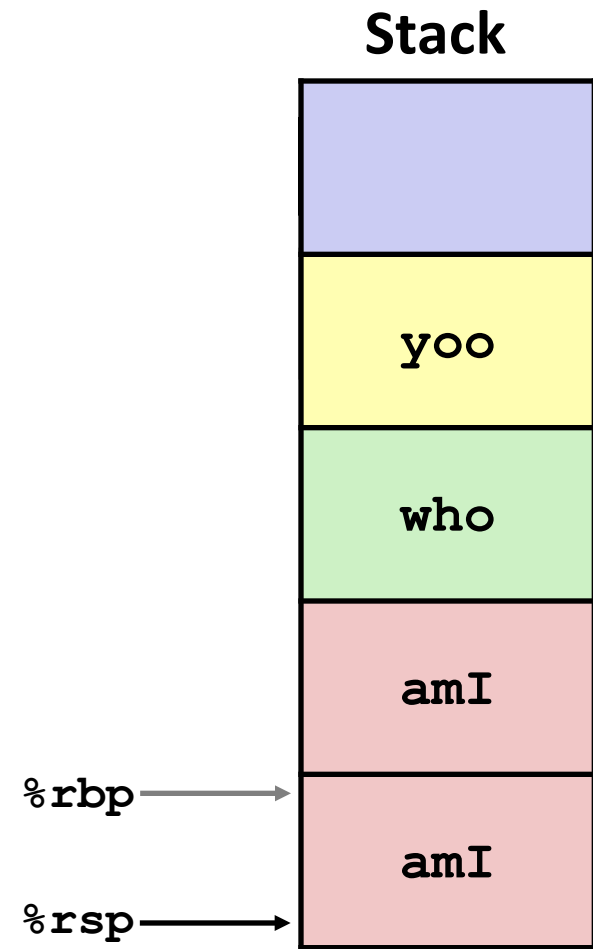
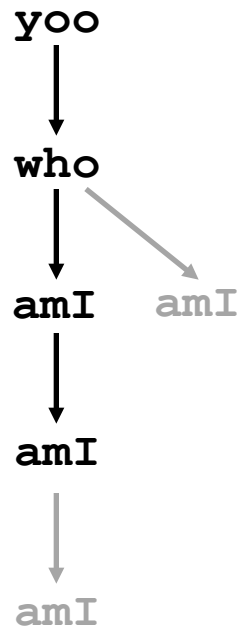
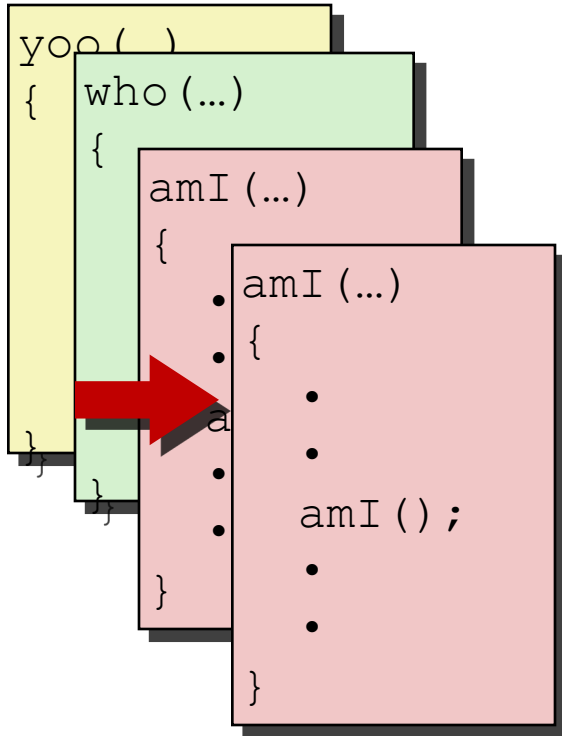
Example



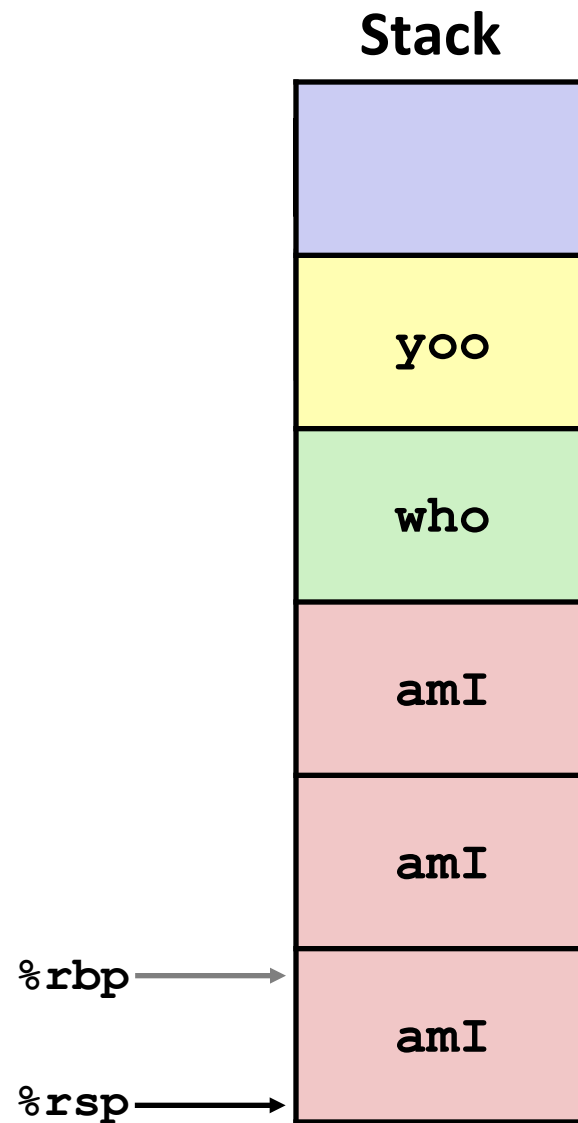
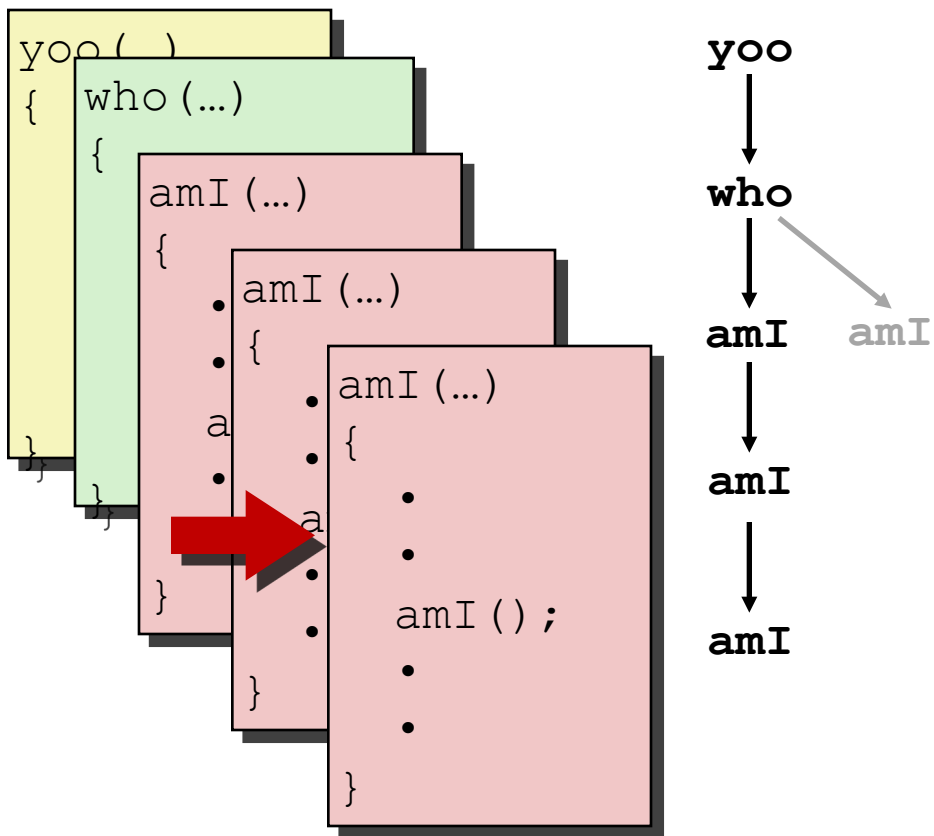
Example



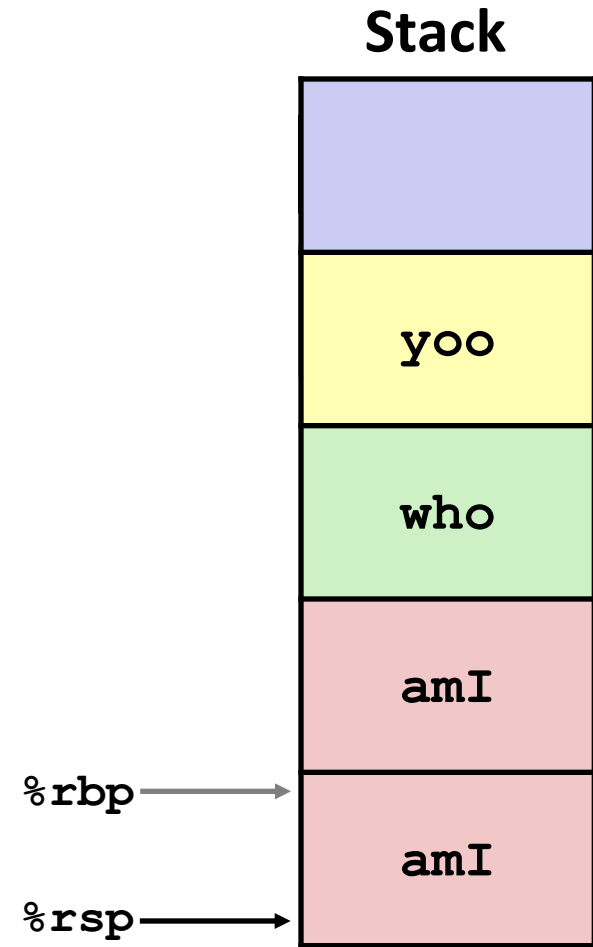
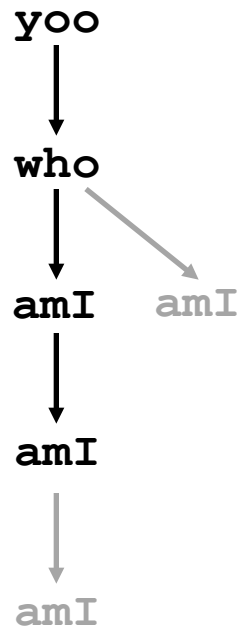
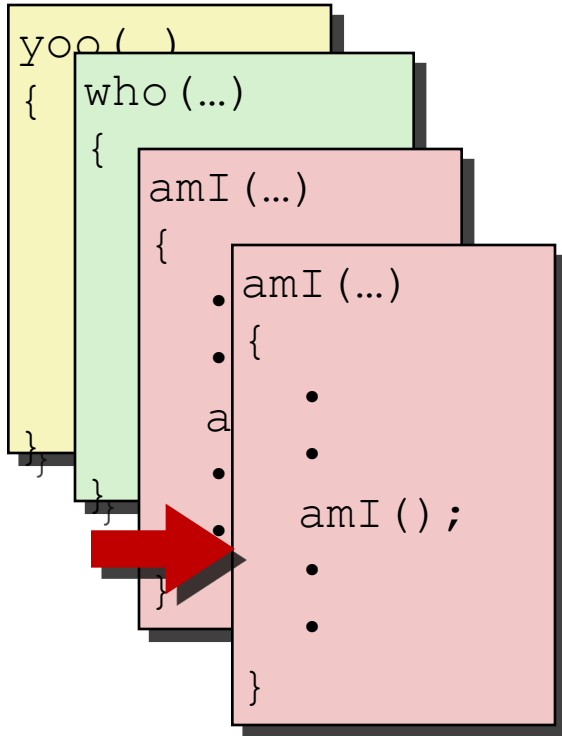
Example



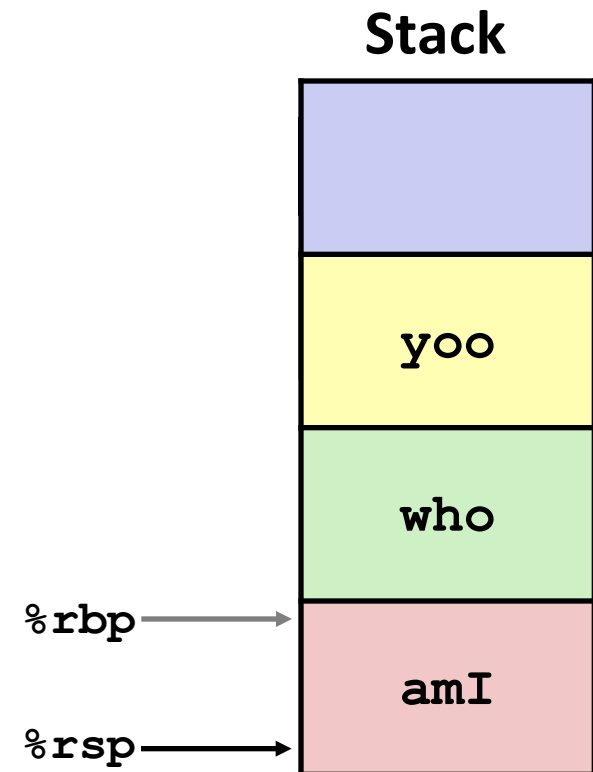
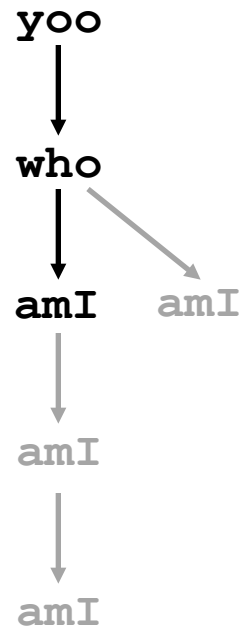
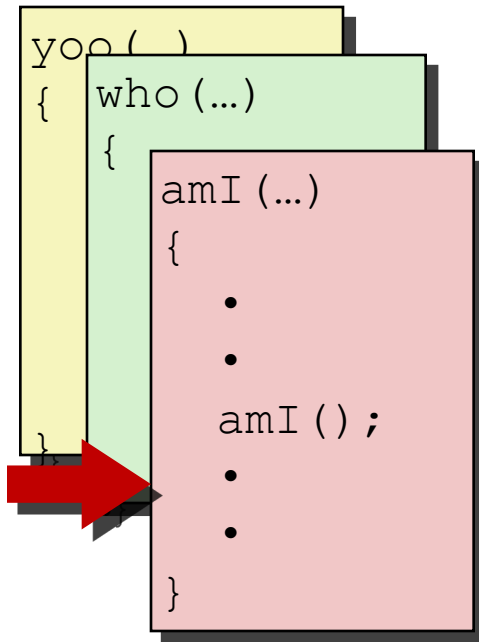
Example



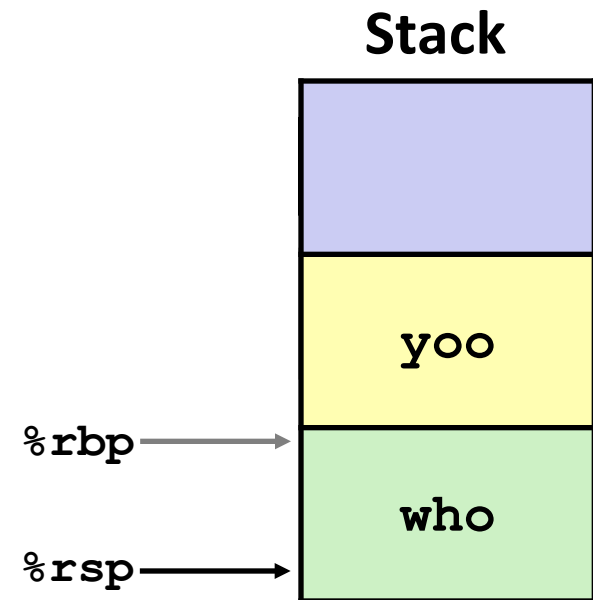
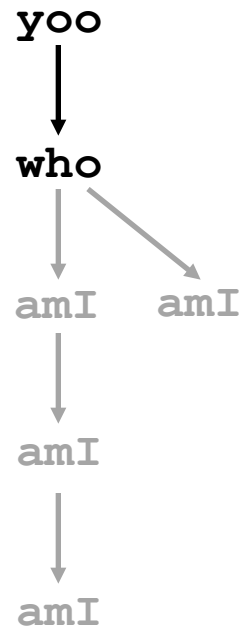
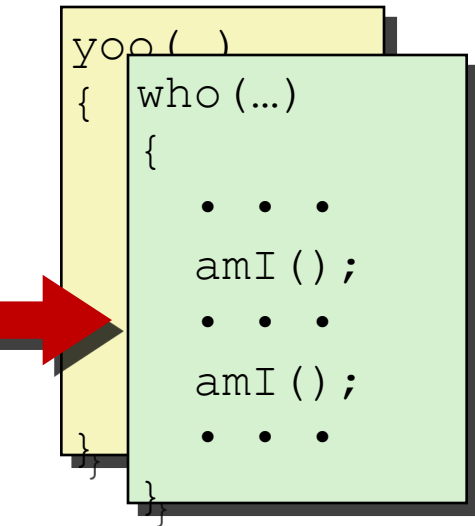
Example



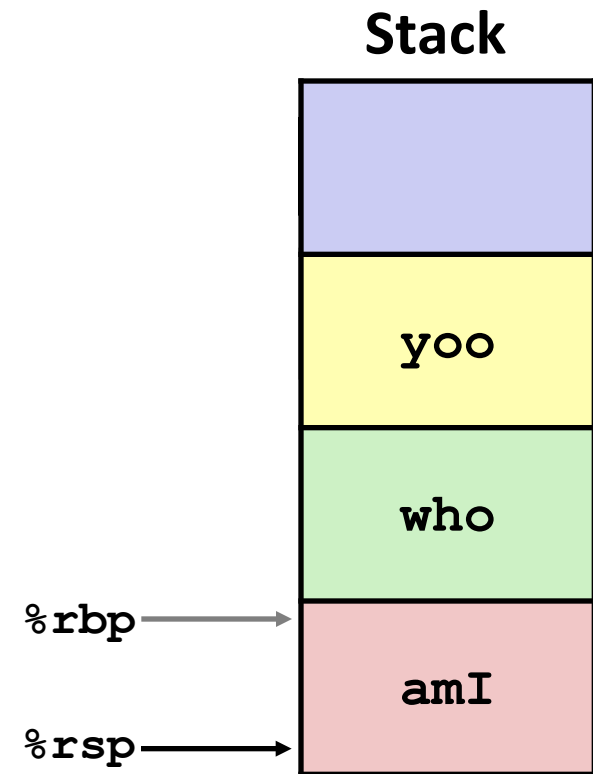
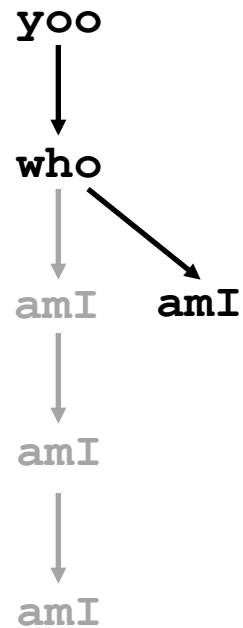
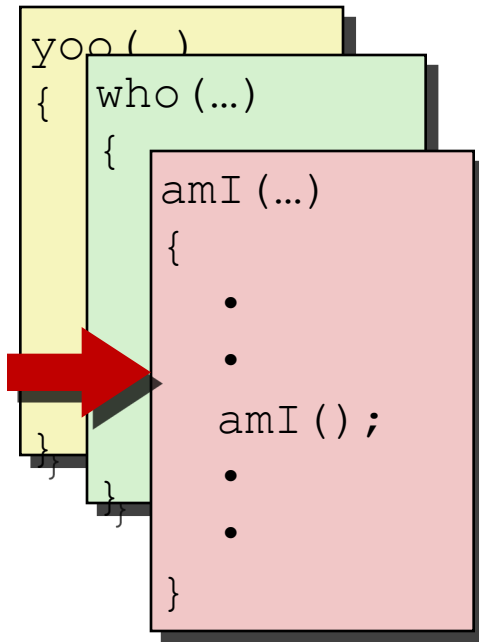
Example



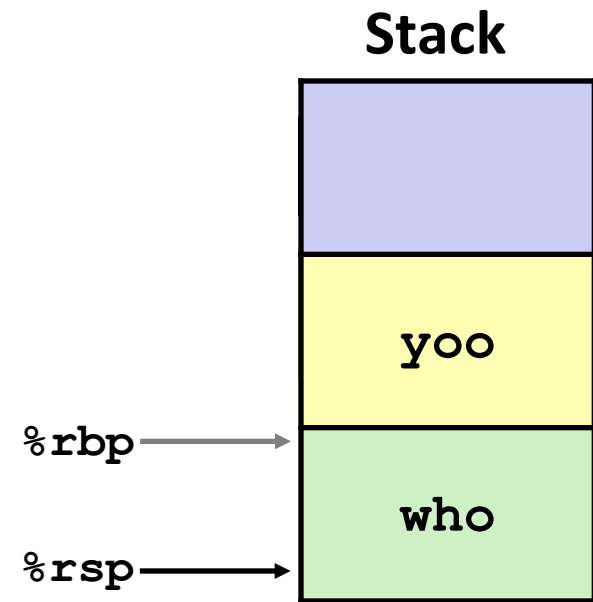
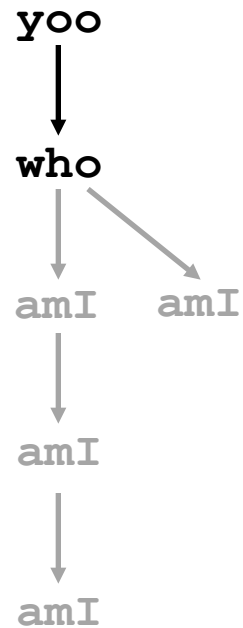
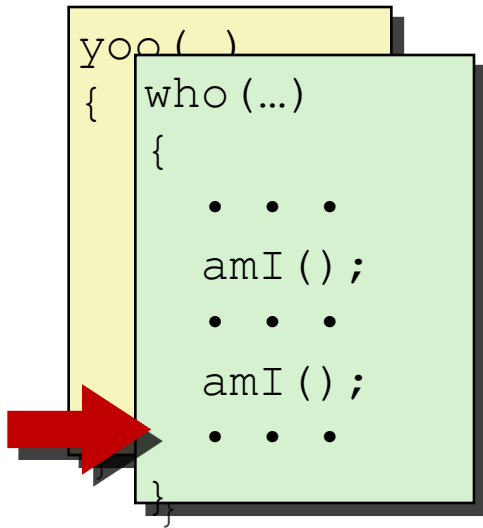
Example



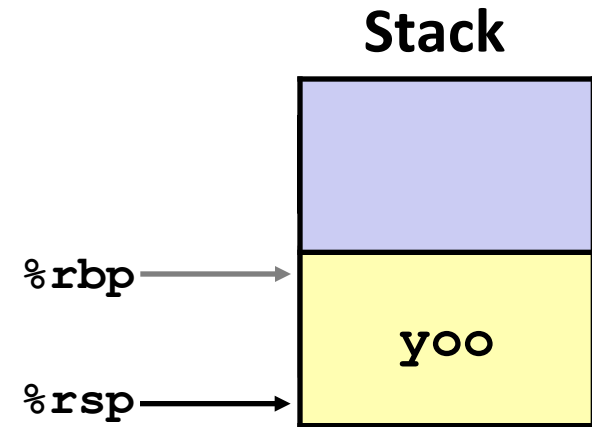
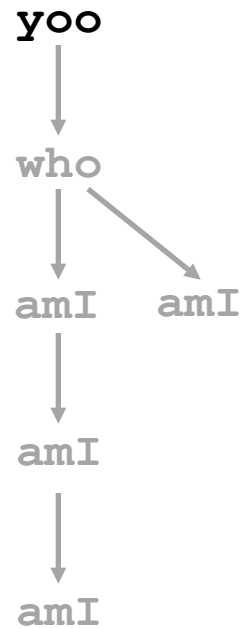
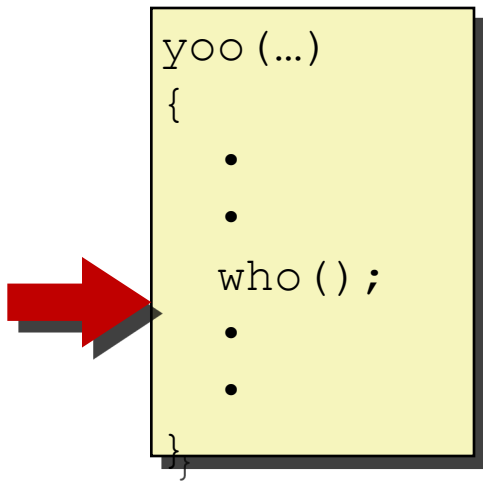
Example



Example



Example



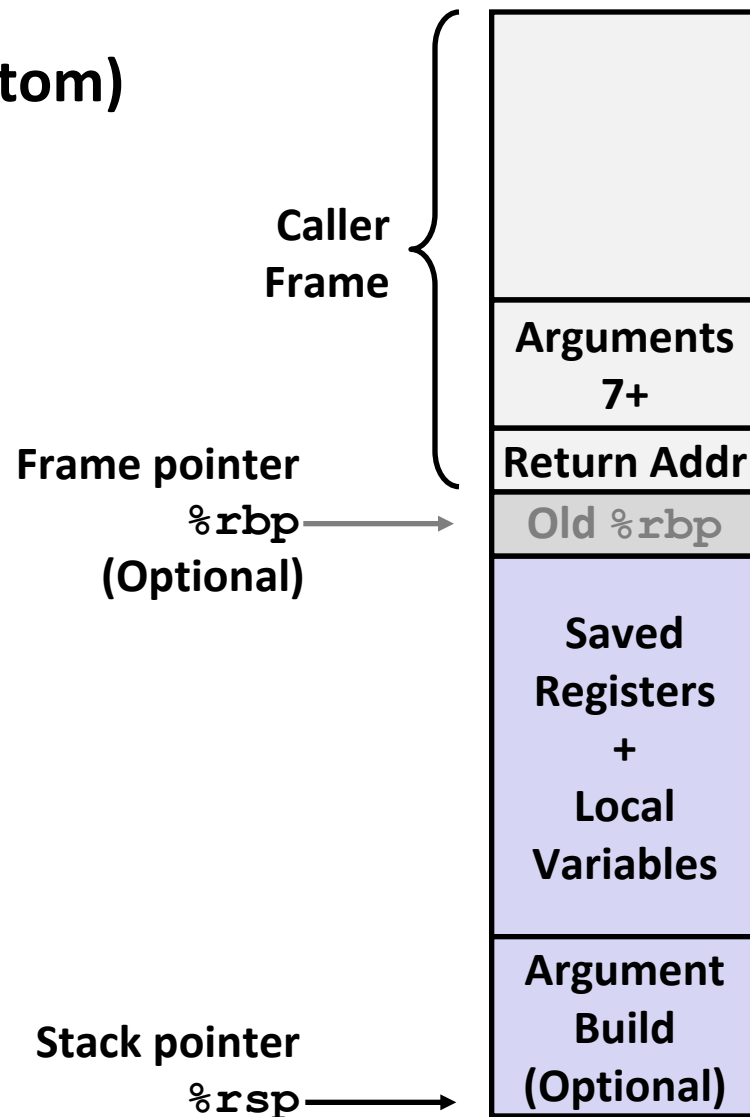
x86-64/Linux Stack Frame

■ Current Stack Frame (“Top” to Bottom)

- “Argument build:”
Parameters for function about to call
- Local variables
If can’t keep in registers
- Saved register context
- Old frame pointer (optional)

■ Caller Stack Frame

- Return address
 - Pushed by **call** instruction
- Arguments for this call



Example: `incr`

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

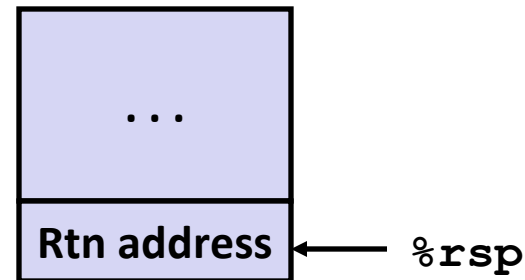
```
incr:  
    movq    (%rdi), %rax  
    addq    %rax, %rsi  
    movq    %rsi, (%rdi)  
    ret
```

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

Example: Calling `incr` #1

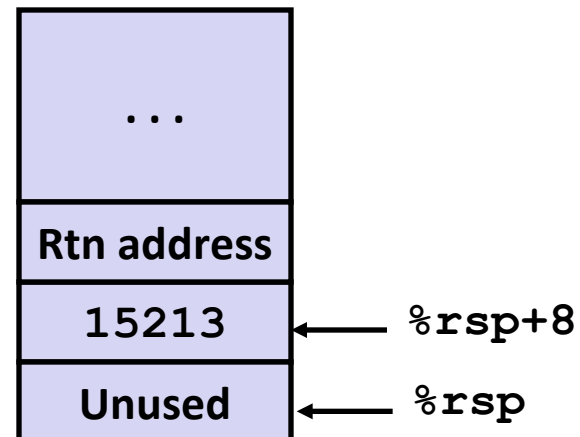
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Initial Stack Structure



```
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

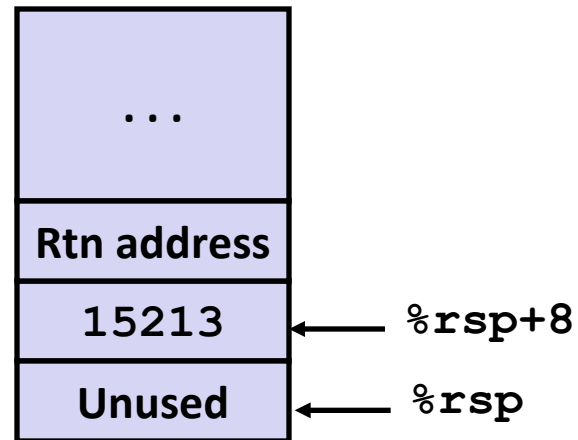


Example: Calling `incr` #2

```
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
```

Stack Structure

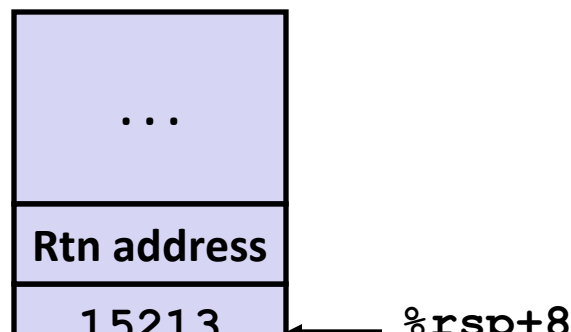


Register	Use(s)
<code>%rdi</code>	<code>&v1</code>
<code>%rsi</code>	3000

Example: Calling `incr` #2

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

Stack Structure



Aside 1: `movl $3000, %esi`

- Remember, `movl` -> `%exx` zeros out high order 32 bits.
- Why use `movl` instead of `movq`? 1 byte shorter.

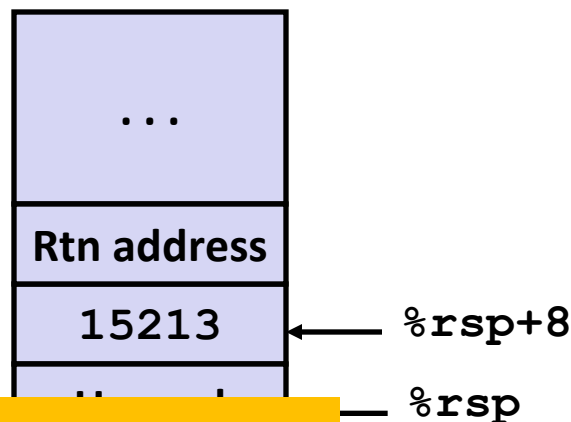
```
movl    $3000, %esi
leaq    8(%rsp), %rdi
call    incr
addq    8(%rsp), %rax
addq    $16, %rsp
ret
```

<code>%rdi</code>	<code>&v1</code>
<code>%rsi</code>	3000

Example: Calling `incr` #2

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

Stack Structure



Aside 2: `leaq 8(%rsp), %rdi`

- Computes `%rsp+8`
- Actually, used for what it is meant!

```
leaq    8(%rsp), %rdi
call    incr
addq    8(%rsp), %rax
addq    $16, %rsp
ret
```

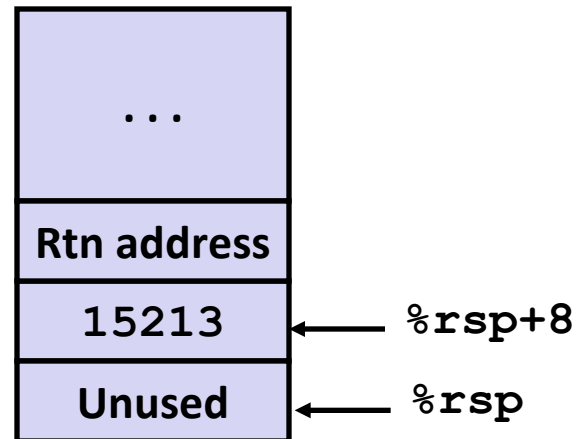
Case(s)	
<code>%rdi</code>	<code>v1</code>
<code>%rsi</code>	3000

Example: Calling `incr` #2

```
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
```

Stack Structure



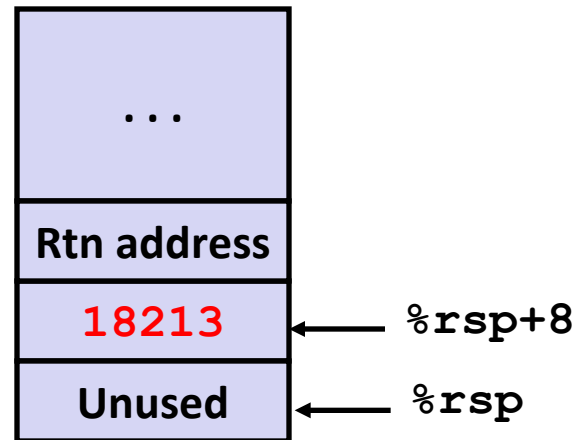
Register	Use(s)
<code>%rdi</code>	<code>&v1</code>
<code>%rsi</code>	3000

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
```

Stack Structure

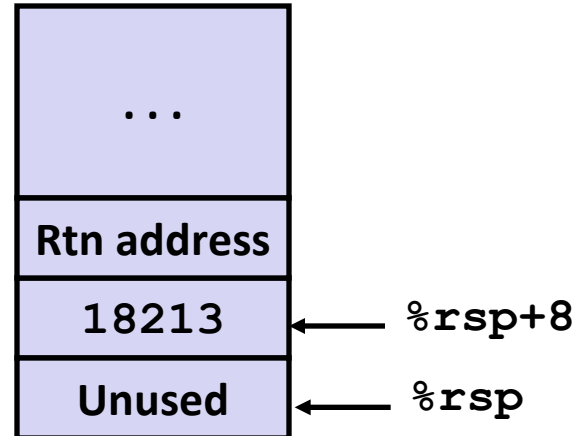


Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling `incr` #4

Stack Structure

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



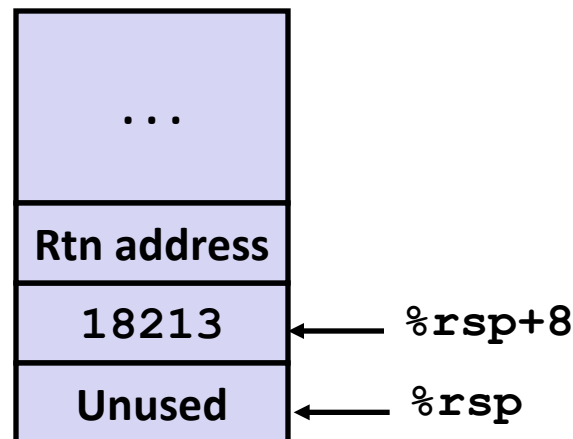
```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Register	Use(s)
%rax	Return value

Example: Calling `incr` #5a

Stack Structure

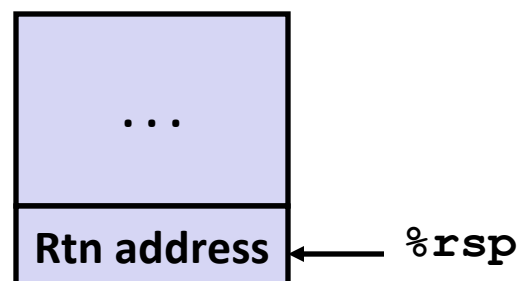
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```



```
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)
<code>%rax</code>	Return value

Updated Stack Structure

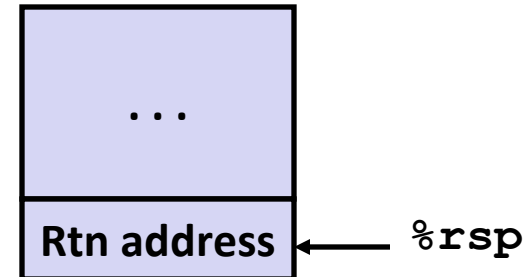


Example: Calling `incr` #5b

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

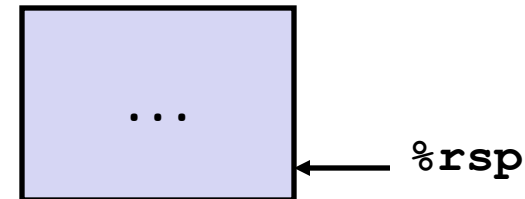
```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

Updated Stack Structure



Register	Use(s)
<code>%rax</code>	Return value

Final Stack Structure



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 → something should be done!
 - Need some 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” (aka “Call-Clobbered”)*
 - Caller saves temporary values in its frame before the call
- *“Callee Saved” (aka “Call-Preserved”)*
 - Callee saves temporary values in its frame before using
 - Callee restores them before returning to caller

x86-64 Linux Register Usage #1

■ **%rax**

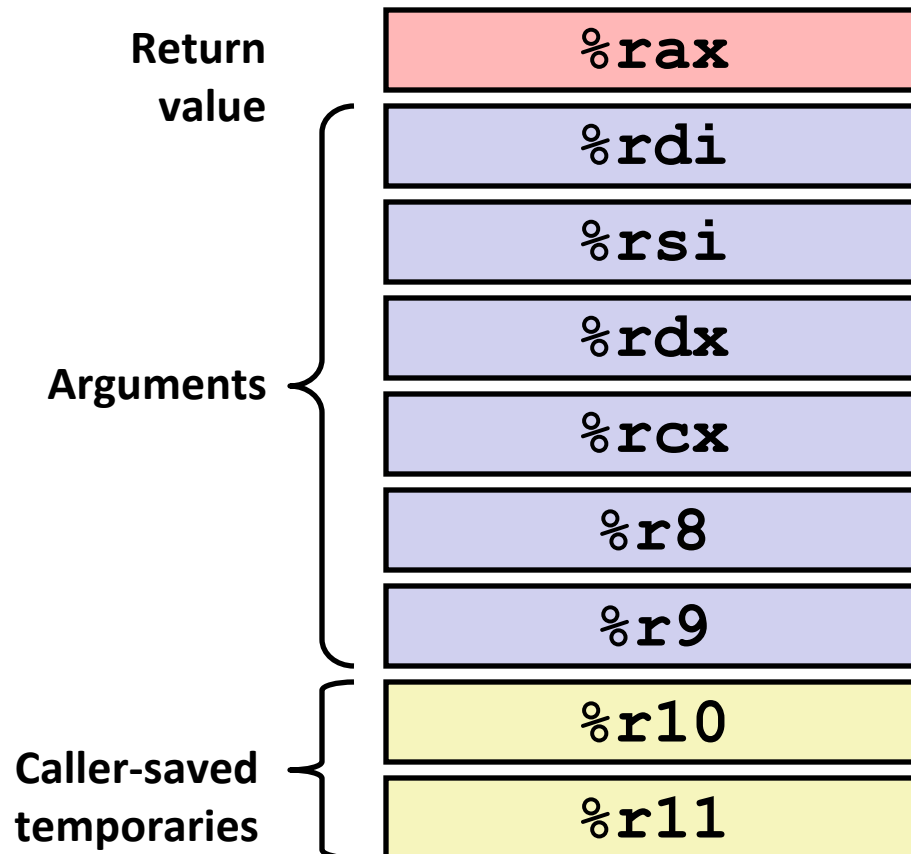
- Return value
- Also caller-saved
- Can be modified by procedure

■ **%rdi, ..., %r9**

- Arguments
- Also caller-saved
- Can be modified by procedure

■ **%r10, %r11**

- Caller-saved
- Can be modified by procedure



x86-64 Linux Register Usage #2

■ **%rbx, %r12, %r13, %r14**

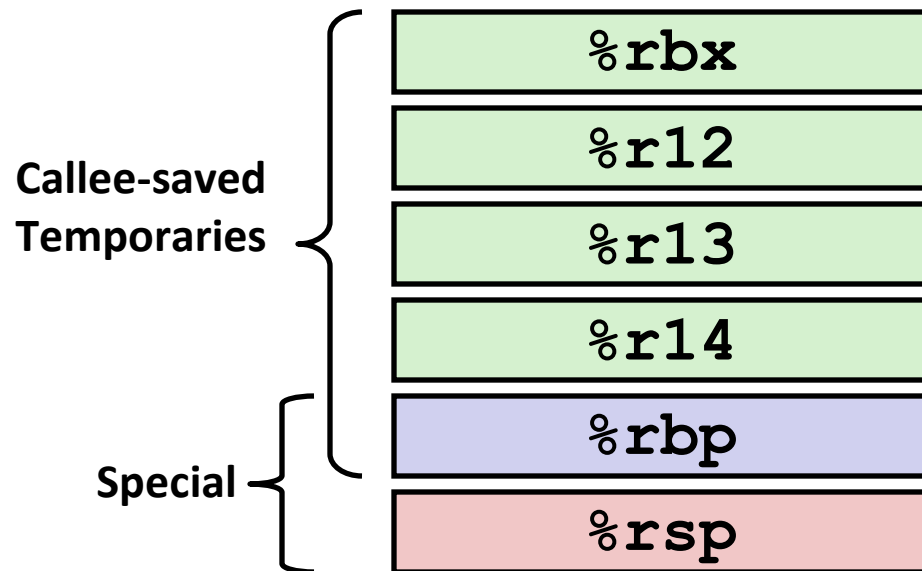
- Callee-saved
- Callee must save & restore

■ **%rbp**

- Callee-saved
- Callee must save & restore
- May be used as frame pointer
- Can mix & match

■ **%rsp**

- Special form of callee save
- Restored to original value upon exit from procedure



x86-64 Procedure Summary

■ Important Points

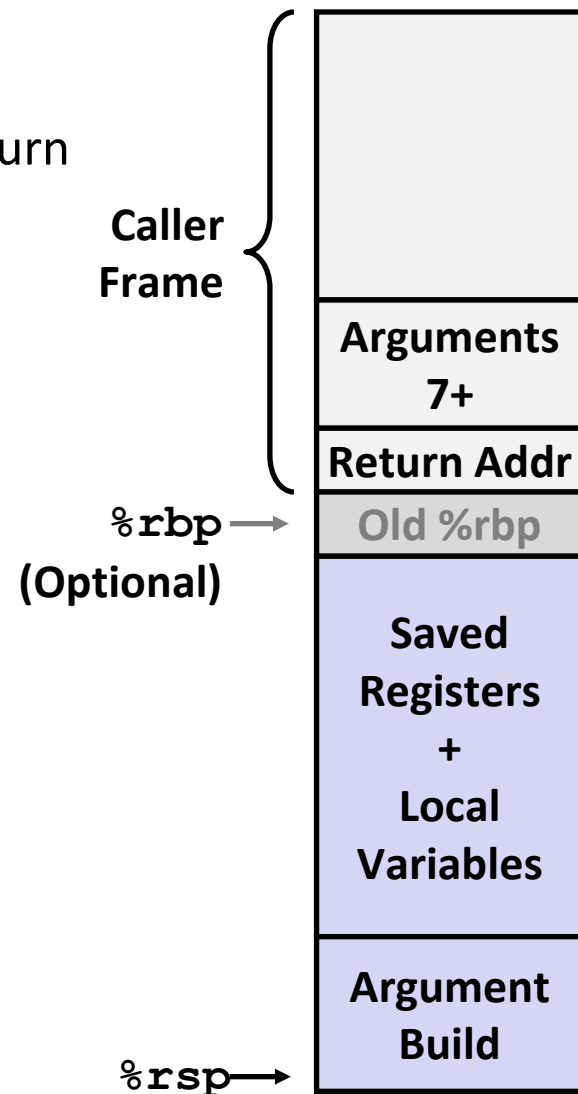
- Stack is the right data structure for procedure call/return
 - If P calls Q, then Q returns before P

■ 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



Quiz Time!

Check out:

<https://canvas.cmu.edu/courses/49105/quizzes/1500452>

Today

■ Procedure Calling Conventions (cont.)

- Passing data
- Managing local data

■ Arrays

- One-dimensional
- Multi-dimensional (nested)
- Multi-level

■ Structures

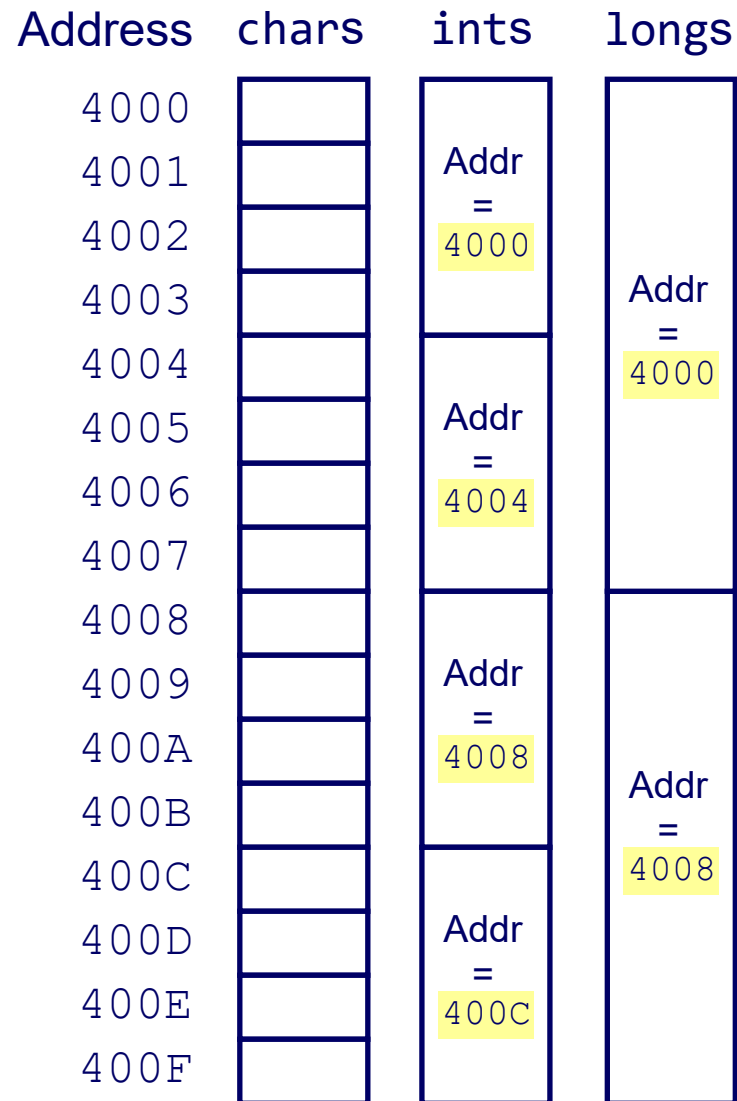
Reminder: Memory Organization

■ Memory locations do not have data types

- Types are implicit in how machine instructions *use* memory

■ Addresses specify byte locations

- Address of a larger datum is the address of its first byte
- Addresses of successive items differ by the item's size

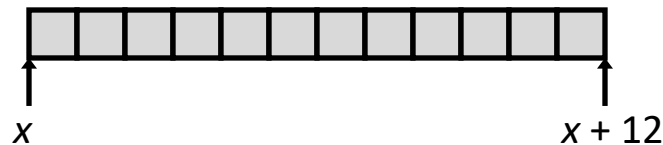


Array Allocation

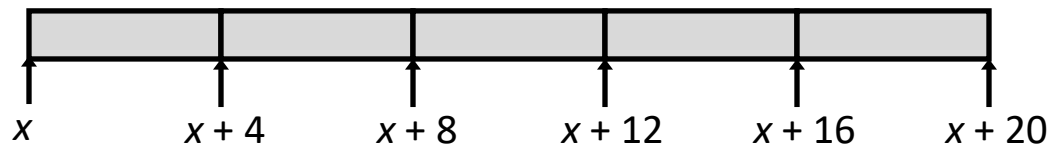
■ C declaration *Type* name [*Length*] ;

- Array of data type *Type* and length *Length*
- Contiguously allocated region of $Length * \text{sizeof}(Type)$ bytes in memory

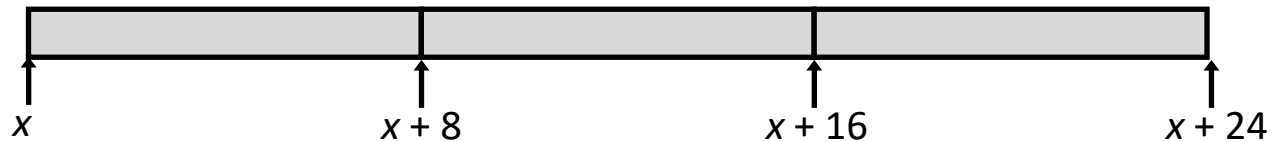
```
char string[12];
```



```
int val[5];
```



```
double a[3];
```



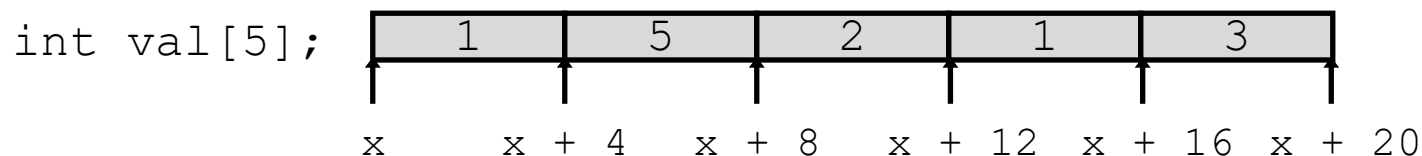
```
char *p[3];
```



Array Access

■ C declaration *Type name [Length] ;*

- Array of data type *Type* and length *Length*
- Identifier **name** acts like a pointer to array element 0

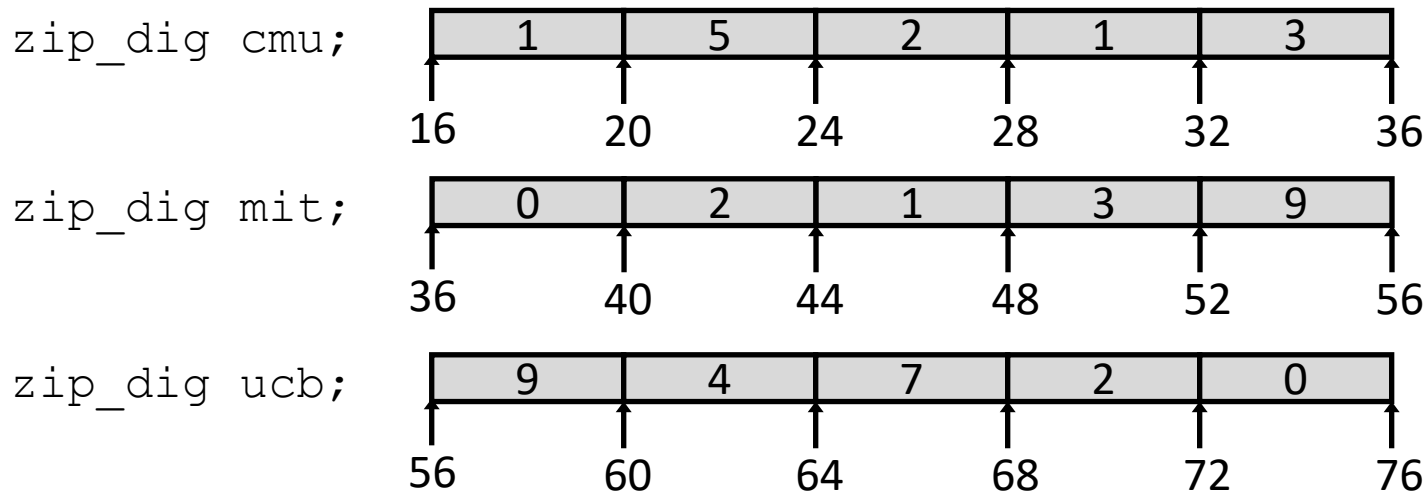


■ Expression	Type	Value	
<code>val[4]</code>	<code>int</code>	3	
<code>val[5]</code>	<code>int</code>	??	// access past end
<code>*(val+3)</code>	<code>int</code>	1	// same as <code>val[3]</code>
<code>val</code>	<code>int *</code>	<code>x</code>	
<code>val+1</code>	<code>int *</code>	<code>x + 4</code>	
<code>&val[2]</code>	<code>int *</code>	<code>x + 8</code>	// same as <code>val+2</code>
<code>val + i</code>	<code>int *</code>	<code>x + 4*i</code>	// same as <code>&val[i]</code>

Array Example

```
#define ZLEN 5
typedef int zip_dig[ZLEN];

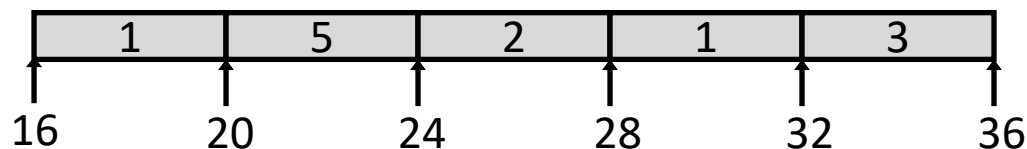
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



- Declaration “`zip_dig cmu`” equivalent to “`int cmu[5]`”
- Example arrays were allocated in successive 20 byte blocks
 - Not guaranteed to happen in general

Array Accessing Example

zip_dig cmu;



```
int get_digit
(zip_dig z, int digit)
{
    return z[digit];
}
```

x86-64

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

- Register `%rdi` contains starting address of array
- Register `%rsi` contains array index
- Desired digit at `%rdi + 4*%rsi`
- Use memory reference `(%rdi,%rsi,4)`

Array Loop Example

```
void zincr(zip_dig z) {  
    size_t i;  
    for (i = 0; i < ZLEN; 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  
rep; ret
```

Array Loop Example

```
void zincr(zip_dig z) {  
    size_t i;  
    for (i = 0; i < ZLEN; 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  
rep; ret
```


Multidimensional (Nested) Arrays

Declaration

`T A[R][C];`

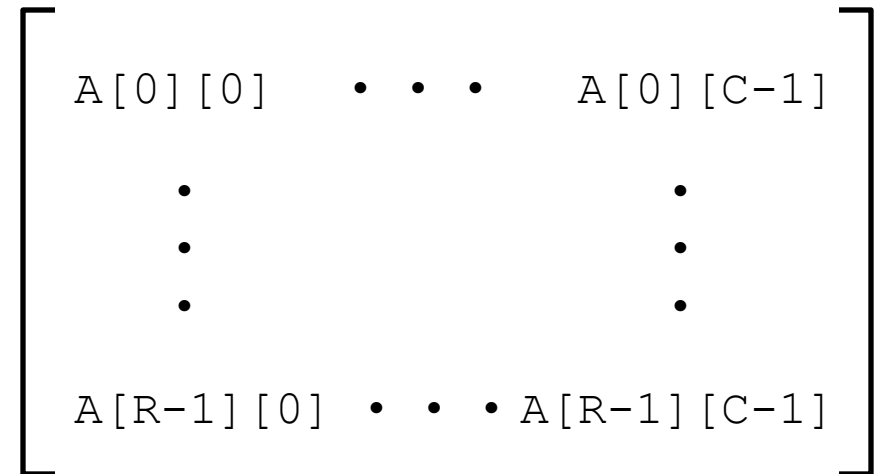
- 2D array of data type T
- R rows, C columns

Array Size

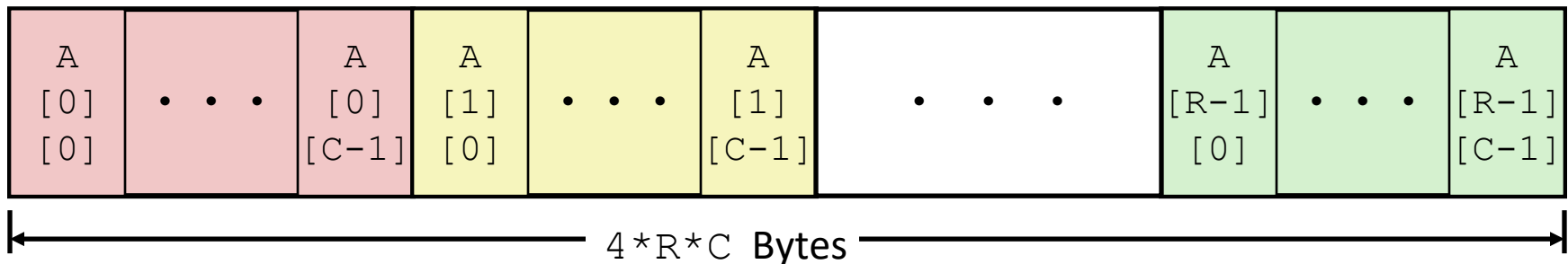
- $R * C * \text{sizeof}(T)$ bytes

Arrangement

- Row-Major Ordering



`int A[R][C];`

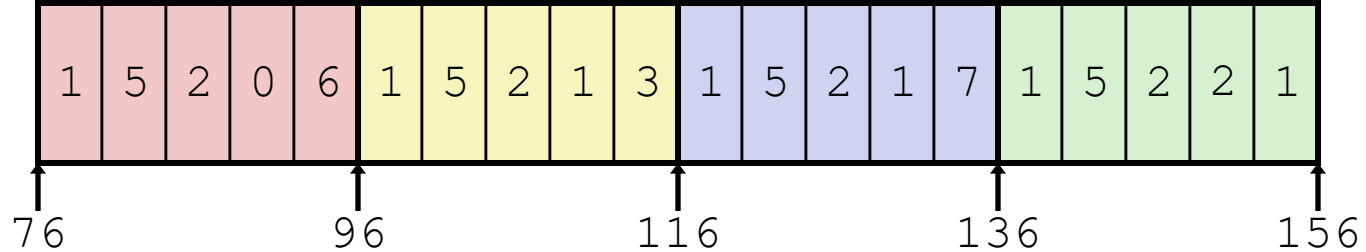


Nested Array Example

```
#define PCOUNT 4
typedef int zip_dig[5];

zip_dig pgh[PCOUNT] =
    {{1, 5, 2, 0, 6 },
     {1, 5, 2, 1, 3 },
     {1, 5, 2, 1, 7 },
     {1, 5, 2, 2, 1 }};
```

zip_dig
pgh[4];



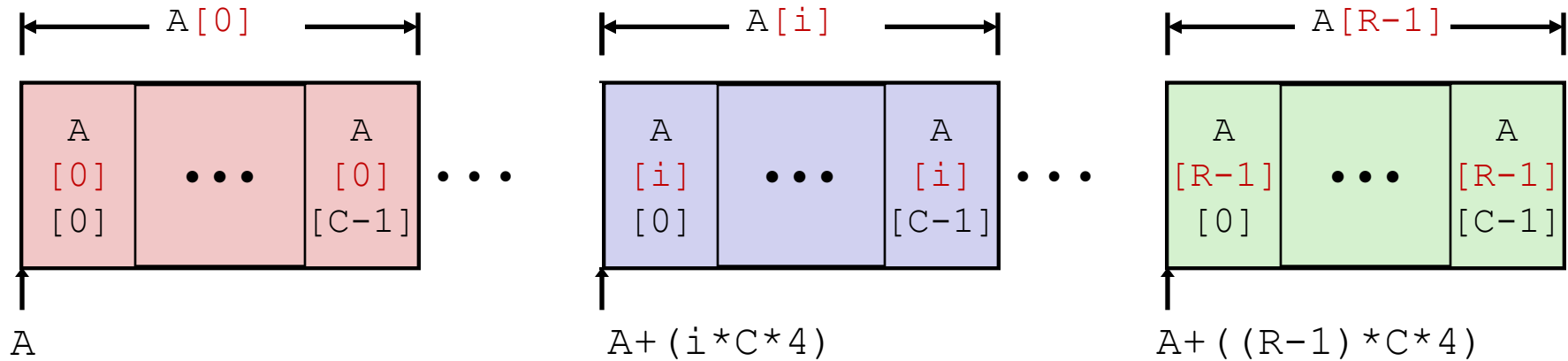
- **“zip_dig pgh[4]” equivalent to “int pgh[4][5]”**
 - Variable **pgh**: array of 4 elements, allocated contiguously
 - Each element is an array of 5 **int**’s, allocated contiguously
- **“Row-Major” ordering of all elements in memory**

Nested Array Row Access

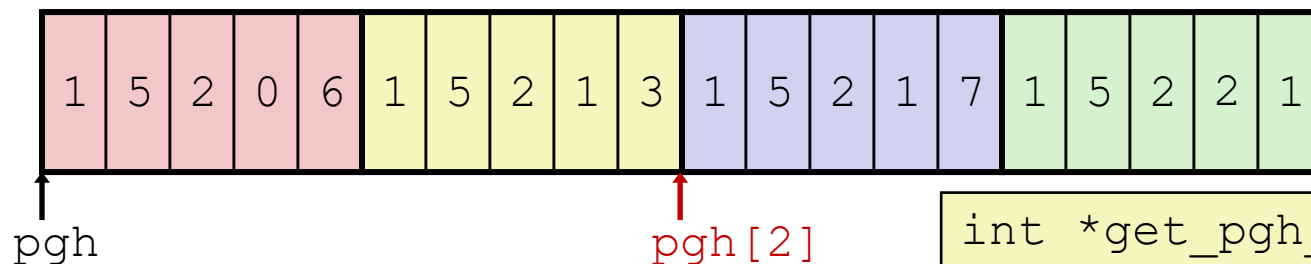
■ Row Vectors

- $A[i]$ is array of C elements of type T
- Starting address $A + i * (C * \text{sizeof}(T))$

```
int A[R][C];
```



Nested Array Row Access Code



```
int *get_pgh_zip(int index)
{
    return pgh[index];
}
```

```
# %rdi = index
leaq (%rdi,%rdi,4),%rax # 5 * index
leaq pgh(,%rax,4),%rax  # pgh + (20 * index)
```

■ Row Vector

- `pgh[index]` is array of 5 `int`'s
- Starting address **`pgh+20*index`**

■ Machine Code

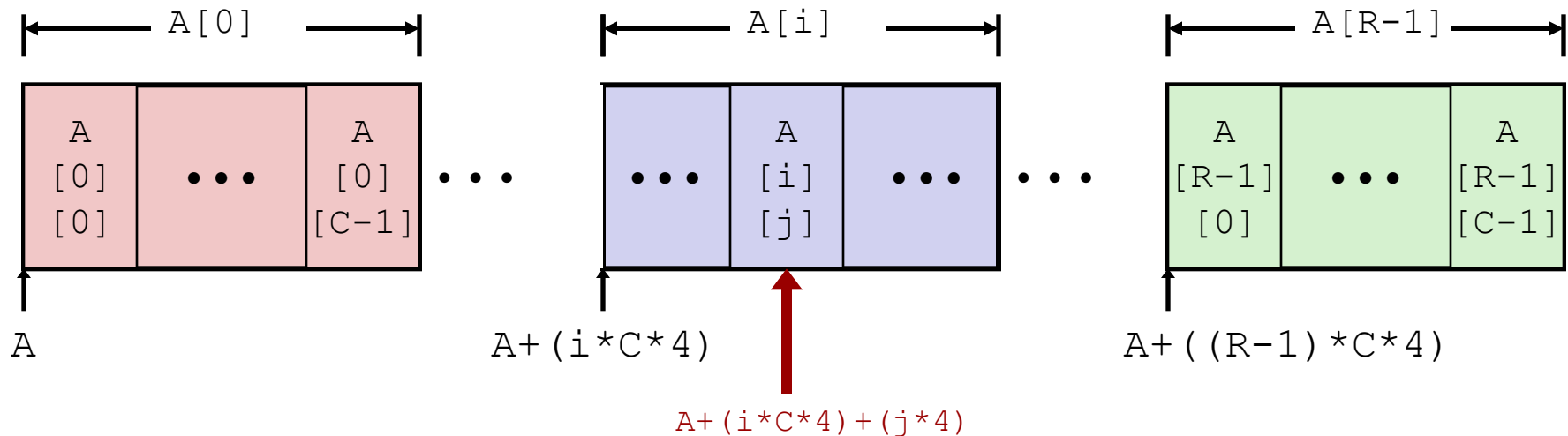
- Computes and returns address
- Compute as **`pgh + 4*(index+4*index)`**

Nested Array Element Access

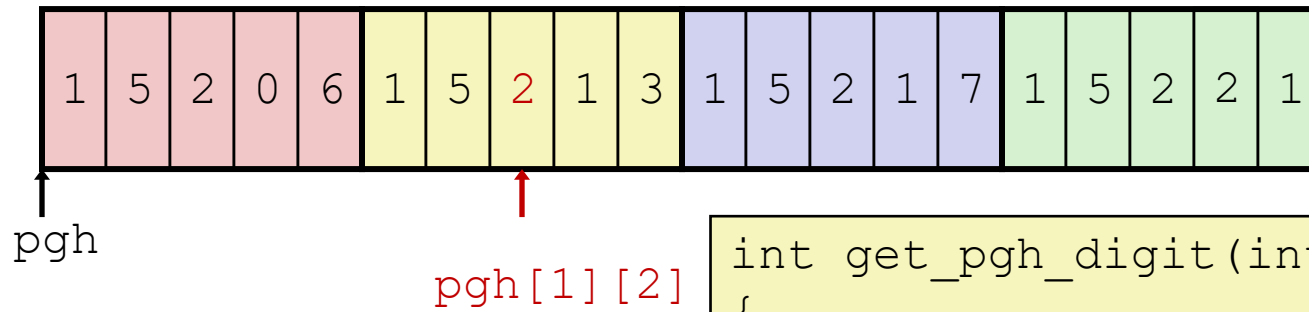
■ Array Elements

- $A[i][j]$ is element of type T , which requires K bytes
- Address $A + i * (C * K) + j * K$
 $= A + (i * C + j) * K$

```
int A[R][C];
```



Nested Array Element Access Code



```
int get_pgh_digit(int index, int dig)
{
    return pgh[index][dig];
}
```

```
leaq    (%rdi,%rdi,4), %rax    # 5*index
addl    %rax, %rsi             # 5*index+dig
movl    pgh(,%rsi,4), %eax     # M[pgh + 4*(5*index+dig)]
```

■ Array Elements

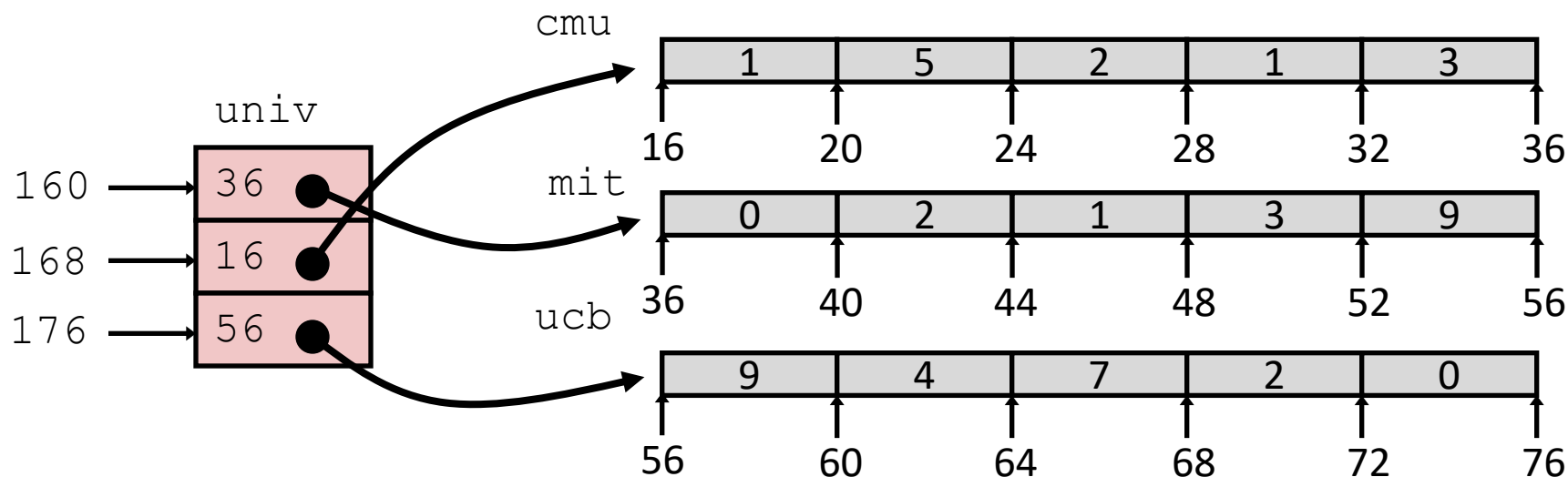
- `pgh[index][dig]` is `int`
- Address: $\text{pgh} + 20 \cdot \text{index} + 4 \cdot \text{dig}$
 $= \text{pgh} + 4 \cdot (5 \cdot \text{index} + \text{dig})$

Multi-Level Array Example

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

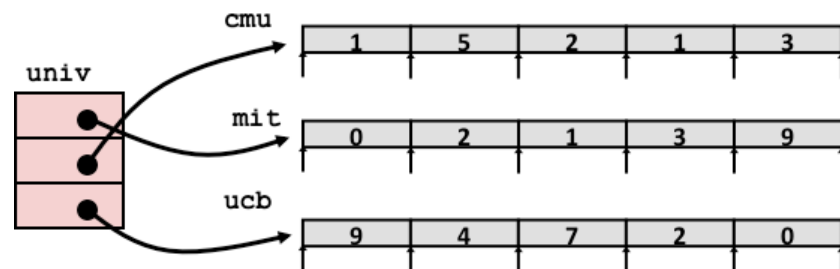
```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

- Variable `univ` denotes array of 3 elements
- Each element is a pointer
 - 8 bytes
- Each pointer points to array of `int`'s



Element Access in Multi-Level Array

```
int get_univ_digit
(size_t index, size_t digit)
{
    return univ[index][digit];
}
```



```
salq    $2, %rsi          # 4*digit
addq    univ(,%rdi,8), %rsi # p = univ[index] + 4*digit
movl    (%rsi), %eax       # return *p
ret
```

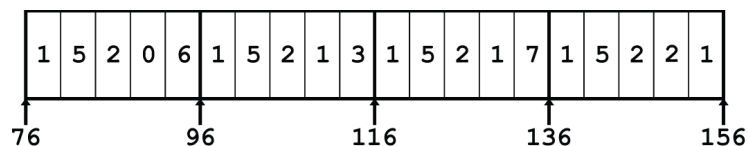
■ Computation

- Element access **Mem[Mem[univ+8*index]+4*digit]**
- Must do two memory reads
 - First get pointer to row array
 - Then access element within array

Array Element Accesses

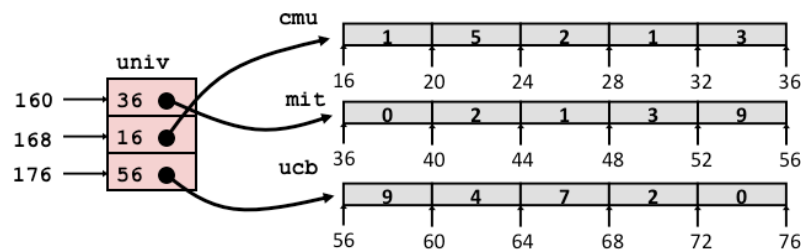
Nested array

```
int get_pgh_digit
(size_t index, size_t digit)
{
    return pgh[index][digit];
}
```



Multi-level array

```
int get_univ_digit
(size_t index, size_t digit)
{
    return univ[index][digit];
}
```



Accesses looks similar in C, but address computations very different:

$\text{Mem}[\text{pgh} + 20 * \text{index} + 4 * \text{digit}]$ $\text{Mem}[\text{Mem}[\text{univ} + 8 * \text{index}] + 4 * \text{digit}]$

$N \times N$ Matrix Code

■ Fixed dimensions

- Known value of N at compile time

```
#define N 16
typedef int fix_matrix[N][N];
/* Get element A[i][j] */
int fix_ele(fix_matrix A,
            size_t i, size_t j)
{
    return A[i][j];
}
```

■ Variable dimensions

- Unknown value of N at compile time

```
/* Get element A[i][j] */
int var_ele(size_t n, int A[n][n],
            size_t i, size_t j) {
    return A[i][j];
}
```

Assembly for Fixed Dimensions

■ Array Elements

- `int A[16][16];`
- Address $A + i * (C * K) + j * K$
- $C = 16, K = 4$
- No need for integer multiplication (for power of 2)

```
/* Get element A[i][j] */  
int fix_ele(fix_matrix A, size_t i, size_t j) {  
    return A[i][j];  
}
```

```
# A in %rdi, i in %rsi, j in %rdx  
salq    $6, %rsi           # 64*i  
addq    %rsi, %rdi          # A + 64*i  
movl    (%rdi,%rdx,4), %eax # Mem[A + 64*i + 4*j]  
ret
```

Assembly for Variable Dimensions

■ Array Elements

- `size_t n; int A[n][n];`
- Address $A + i * (C * K) + j * K$
- $C = n, K = 4$
- Must perform integer multiplication

```
/* Get element A[i][j] */
int var_ele(size_t n, int A[n][n], size_t i, size_t j)
{
    return A[i][j];
}
```

```
# n in %rdi, A in %rsi, i in %rdx, j in %rcx
imulq    %rdx, %rdi          # n*i
leaq     (%rsi,%rdi,4), %rax  # A + 4*n*i
movl     (%rax,%rcx,4), %eax  # Mem[A + 4*n*i + 4*j]
ret
```

Today

■ Procedure Calling Conventions (cont.)

- Passing data
- Managing local data

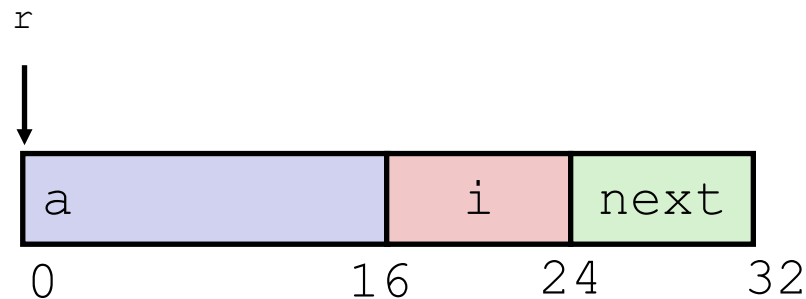
■ Arrays

- One-dimensional
- Multi-dimensional (nested)
- Multi-level

■ Structures

Structure Representation

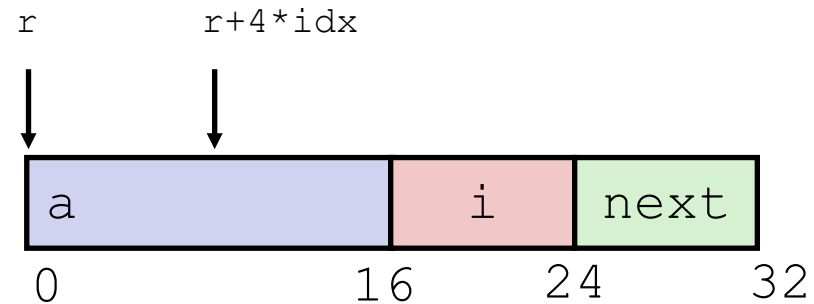
```
struct rec {  
    int a[4];  
    size_t i;  
    struct rec *next;  
};
```



- **Structure represented as block of memory**
 - Big enough to hold all the fields
- **Fields ordered according to declaration**
 - Even if another ordering could be more compact
- **Compiler determines overall size + positions of fields**
 - In assembly, we see only offsets, not field names

Generating Pointer to Structure Member

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



■ Generating Pointer to Array Element

- Offset of each structure member determined at compile time
- Compute as **`r + 4*idx`**

```
int *get_ap
(struct rec *r, size_t idx)
{
    return &r->a[idx];
}
```

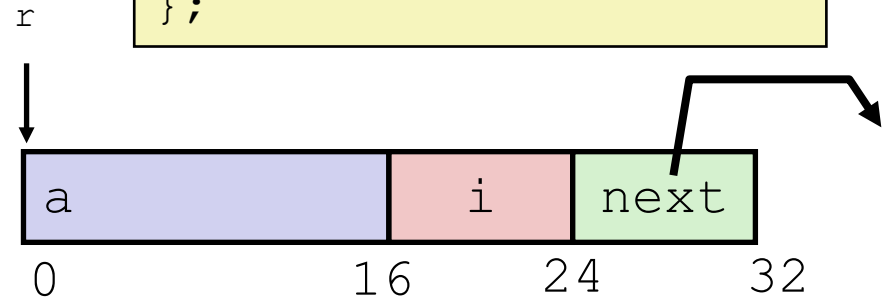
```
# r in %rdi, idx in %rsi
leaq (%rdi,%rsi,4), %rax
ret
```

Following Linked List

■ C Code

```
long length(struct rec*r) {
    long len = 0L;
    while (r) {
        len ++;
        r = r->next;
    }
    return len;
}
```

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



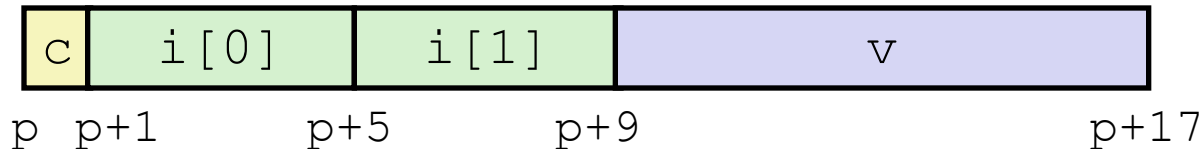
Register	Value
<code>%rdi</code>	<code>r</code>
<code>%rax</code>	<code>len</code>

■ Loop assembly code

```
.L11:                                # loop:
    addq    $1, %rax                 # len ++
    movq    24(%rdi), %rdi           # r = Mem[r+24]
    testq   %rdi, %rdi              # Test r
    jne     .L11                     # If != 0, goto loop
```


Structures & Alignment

■ Unaligned Data

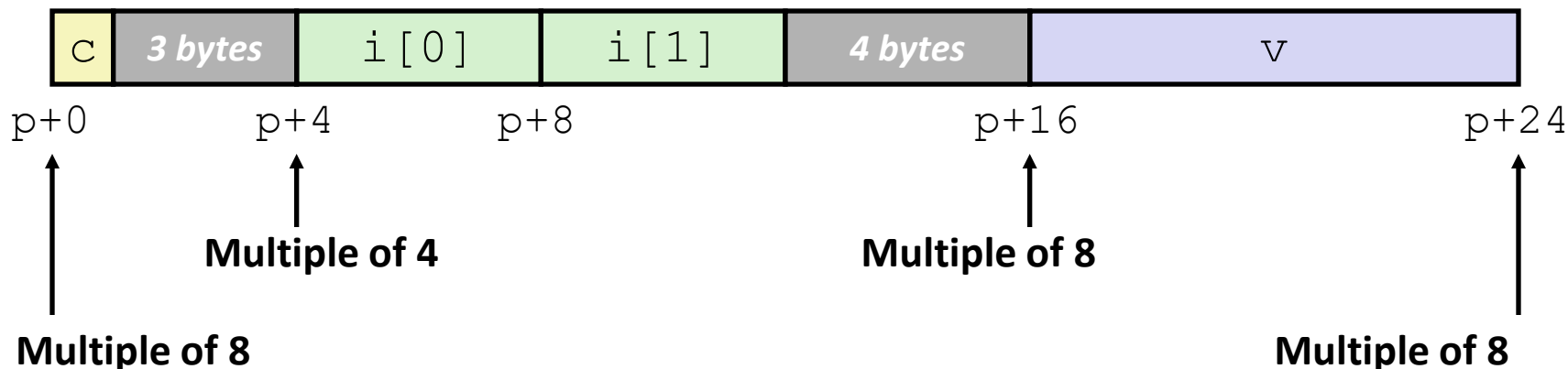


```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

■ Aligned Data

- Primitive data type requires ***B*** bytes implies
Address must be multiple of ***B***

**Compiler inserts gaps
to ensure alignment**



Specific Cases of Alignment (x86-64)

■ 1 byte: `char`, ...

- no restrictions on address

■ 2 bytes: `short`, ...

- lowest 1 bit of address must be 0_2

■ 4 bytes: `int`, `float`, ...

- lowest 2 bits of address must be 00_2

■ 8 bytes: `double`, `long`, `char *`, ...

- lowest 3 bits of address must be 000_2

Motivation for Aligning Data

- Memory accessed by (aligned) chunks of 4 or 8 bytes
- Inefficient to load or store datum that spans cache lines (64 bytes)
- Virtual memory trickier when datum spans 2 pages (4 KB pages)

Satisfying Alignment with Structures

■ Within structure:

- Must satisfy each element's alignment requirement

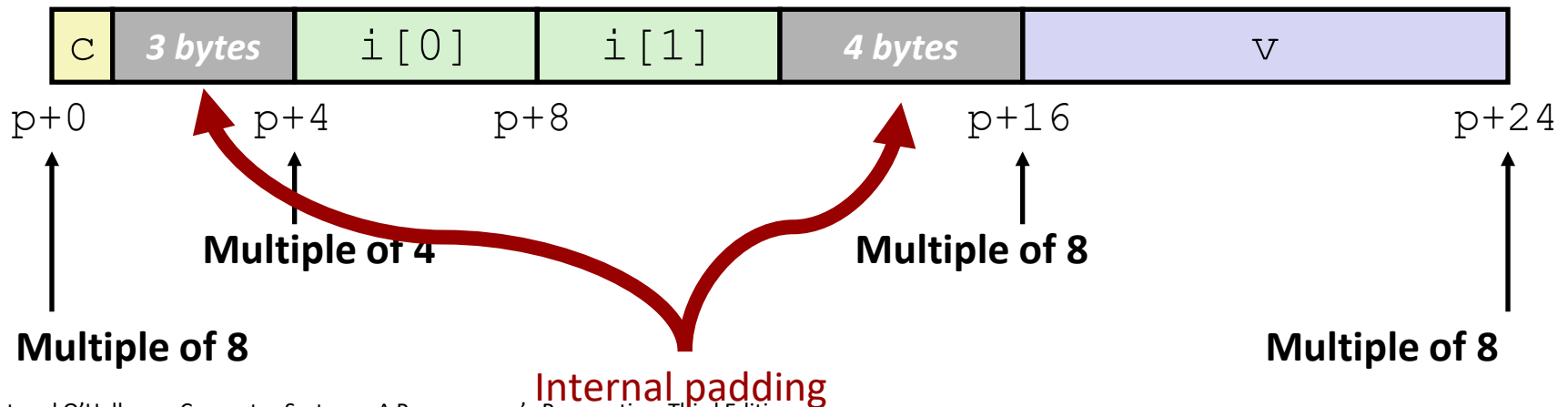
■ Overall structure placement

- Each structure has alignment requirement **K**
 - **K** = Largest alignment of any element
- Initial address & structure length must be multiples of **K**

```
struct S1 {  
    char c;  
    int i[2];  
    double v;  
} *p;
```

■ Example:

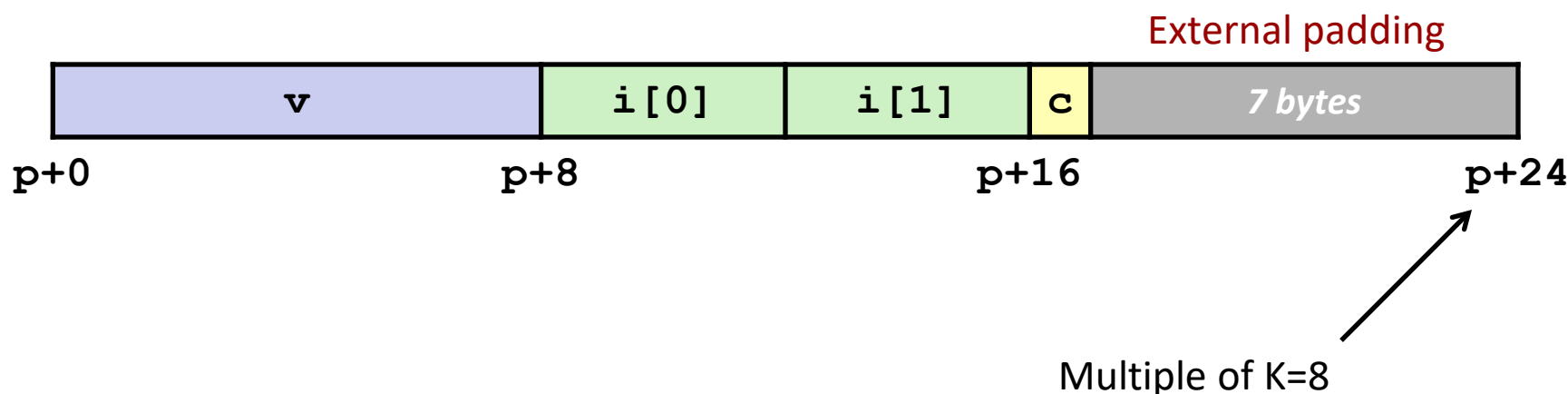
- **K** = 8, due to **double** element. NOTE: $K < \text{sizeof}(\text{struct S1})$



Meeting Overall Alignment Requirement

- For largest alignment requirement K
- Overall structure must be multiple of K

```
struct S2 {  
    double v;  
    int i[2];  
    char c;  
} *p;
```



Summary

■ Procedure Calling Conventions (cont.)

CSAPP 3.7

- Passing data
- Managing local data

■ Arrays

CSAPP 3.8

- One-dimensional
- Multi-dimensional (nested)
- Multi-level

■ Structures

CSAPP 3.9

Additional Slides

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
```

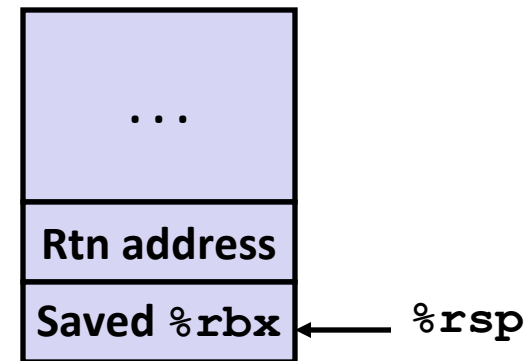
Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

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	Rec. argument
%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
```

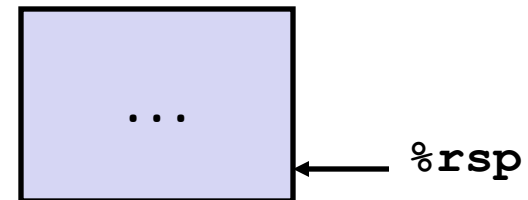
Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

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 in Lecture 9)
- 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

Bonus Content: Tail Recursive

■ Recursive functions can be “tail recursive”

- There is no computation after the function call
- Many compilers / languages can optimize these functions

```
long popcount_t(unsigned long x,  
                long s)  
{  
    if (x == 0) return s;  
    long b = x & 1UL;  
    return popcount_t(x >>1,  
                      s + b);  
}
```

```
popcount_t:  
    movq    %rsi, %rax  
    testq   %rdi, %rdi  
    je      .L18  
.L15:  
    movq    %rdi, %rdx  
    andl    $1, %edx  
    addq    %rdx, %rax  
    shrq    %rdi  
    jne     .L15  
.L18:  
    ret
```

Example: Array Access

```
#include <stdio.h>
#define ZLEN 5
#define PCOUNT 4
typedef int zip_dig[ZLEN];

int main(int argc, char** argv) {
    zip_dig pgh[PCOUNT] =
        {{1, 5, 2, 0, 6},
         {1, 5, 2, 1, 3 },
         {1, 5, 2, 1, 7 },
         {1, 5, 2, 2, 1 }};
    int *linear_zip = (int *) pgh;
    int *zip2 = (int *) pgh[2];
    int result =
        pgh[0][0] +
        linear_zip[7] +
        *(linear_zip + 8) +
        zip2[1];
    printf("result: %d\n", result);
    return 0;
}
```

```
linux> ./array
```


Example: Array Access

```
#include <stdio.h>
#define ZLEN 5
#define PCOUNT 4
typedef int zip_dig[ZLEN];

int main(int argc, char** argv) {
    zip_dig pgh[PCOUNT] =
        {{1, 5, 2, 0, 6},
         {1, 5, 2, 1, 3 },
         {1, 5, 2, 1, 7 },
         {1, 5, 2, 2, 1 }};
    int *linear_zip = (int *) pgh;
    int *zip2 = (int *) pgh[2];
    int result =
        pgh[0][0] +
        linear_zip[7] +
        *(linear_zip + 8) +
        zip2[1];
    printf("result: %d\n", result);
    return 0;
}
```

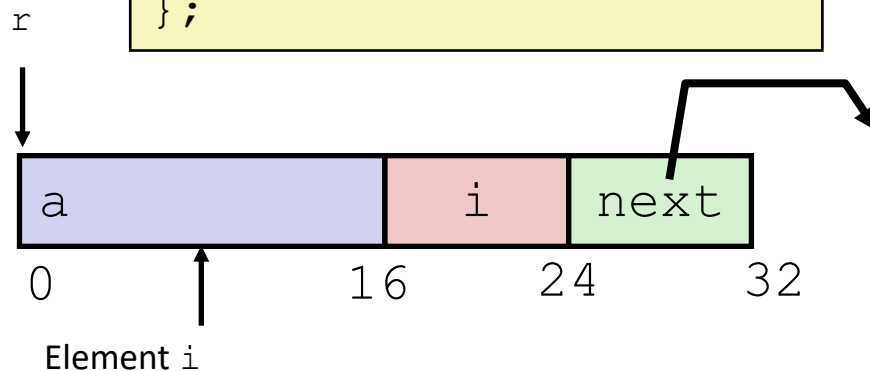
```
linux> ./array
result: 9
```

Following Linked List #2

■ C Code

```
void set_val
(struct rec *r, int val)
{
    while (r) {
        size_t i = r->i;
        // No bounds check
        r->a[r->i] = val;
        r = r->next;
    }
}
```

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



Register	Value
<code>%rdi</code>	<code>r</code>
<code>%rsi</code>	<code>val</code>

```
.L11:                                # loop:
    movq    16(%rdi), %rax           # i = Mem[r+16]
    movl    %esi, (%rdi,%rax,4)     # Mem[r+4*i] = val
    movq    24(%rdi), %rdi          # r = Mem[r+24]
    testq   %rdi, %rdi              # Test r
    jne     .L11                    # if !=0 goto loop
```