Machine-Level Programming II: Procedures and Data

Computer Systems
Friday, October 13, 2023

Today

Procedures

- **Stack Structure**
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Illustration of Recursion

Arrays

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

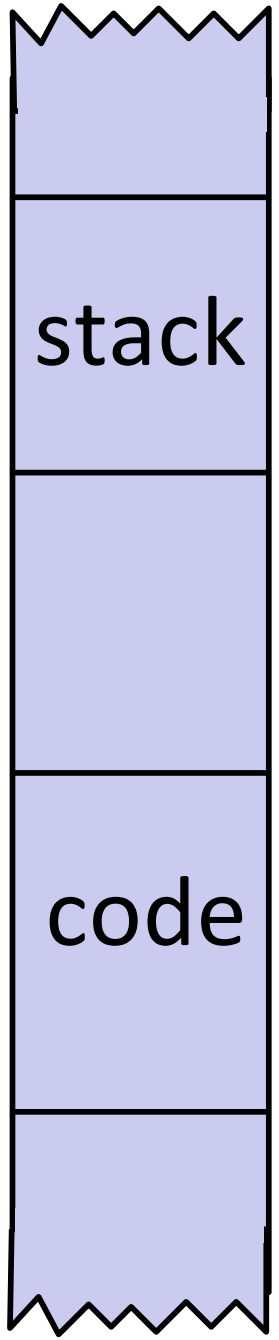
Structures

- Allocation
- Access
- Alignment

x86-64 Stack

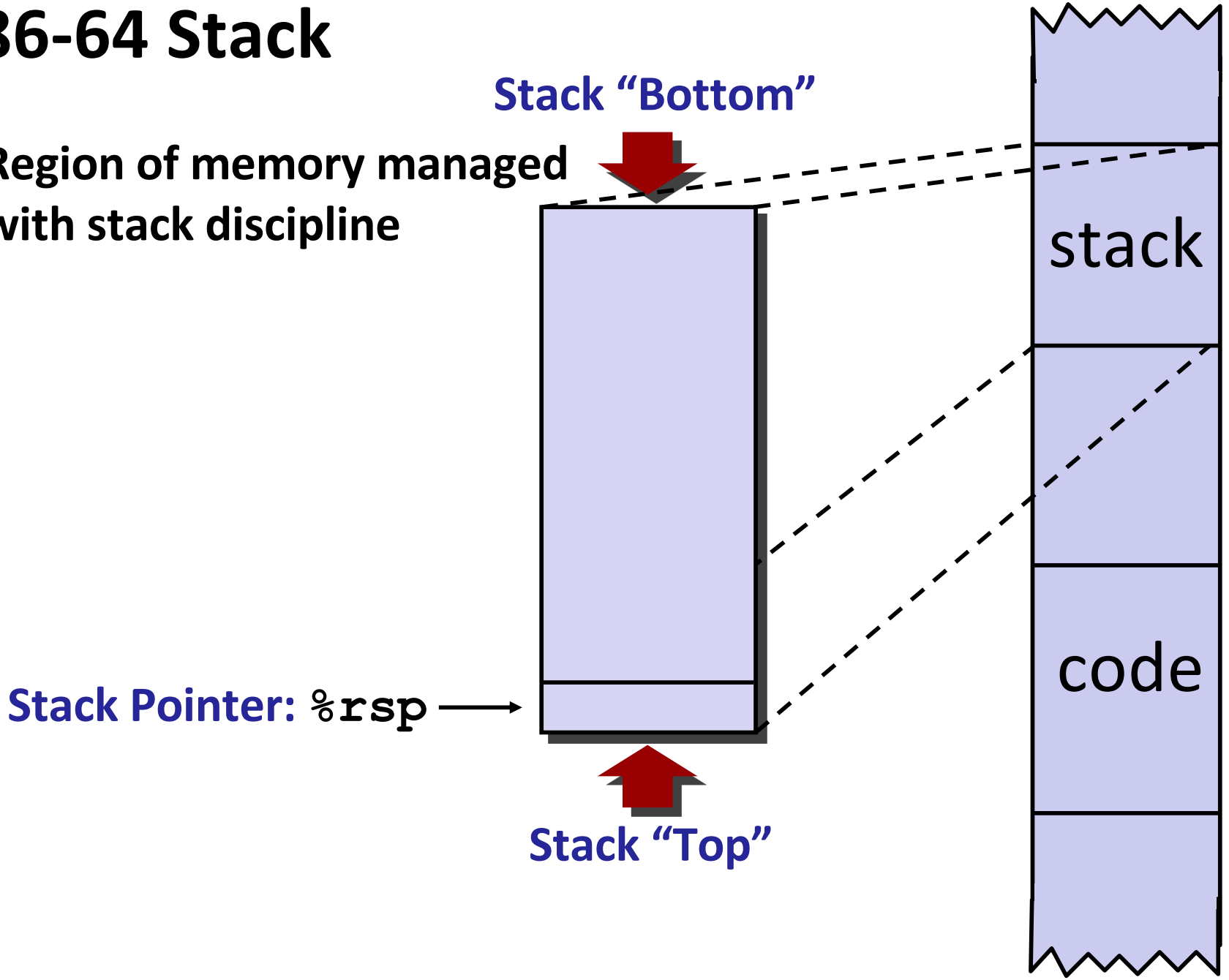
Region of memory managed with stack discipline

- Memory viewed as array of bytes.
- Different regions have different purposes.
- (Like ABI, a policy decision)



x86-64 Stack

Region of memory managed with stack discipline



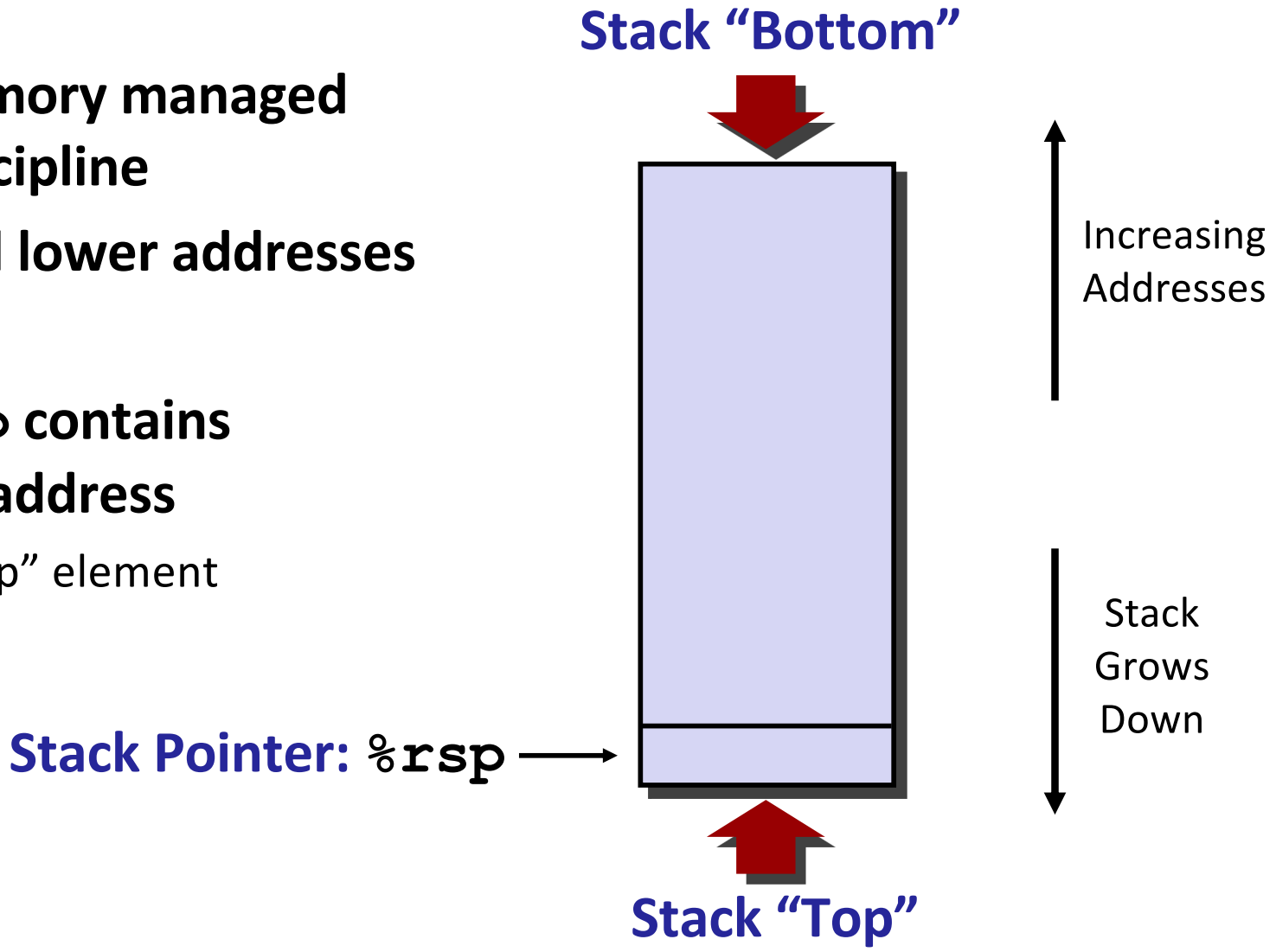
x86-64 Stack

Region of memory managed with stack discipline

Grows toward lower addresses

Register `%rsp` contains lowest stack address

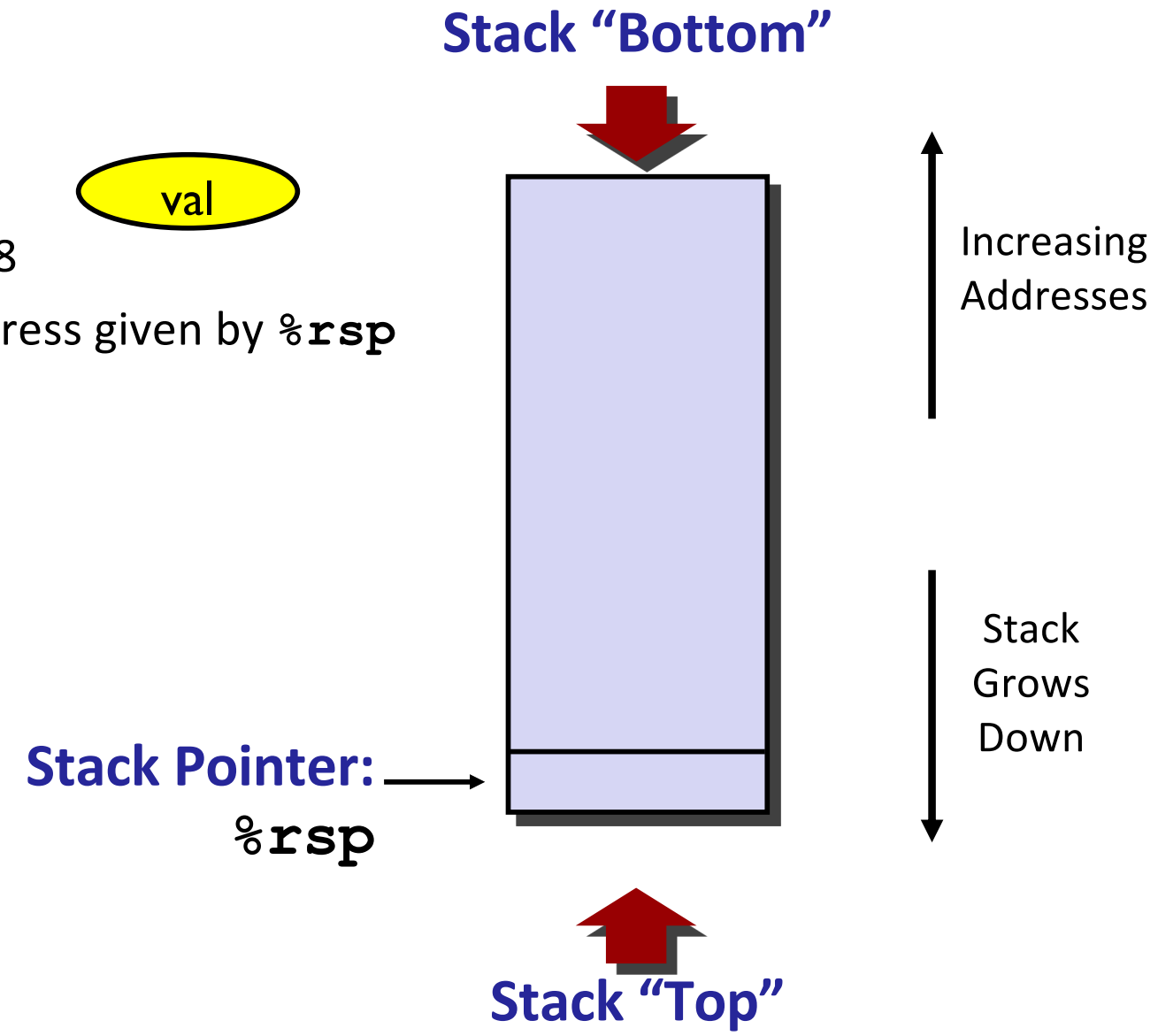
- address of "top" element



x86-64 Stack: Push

`pushq` *Src*

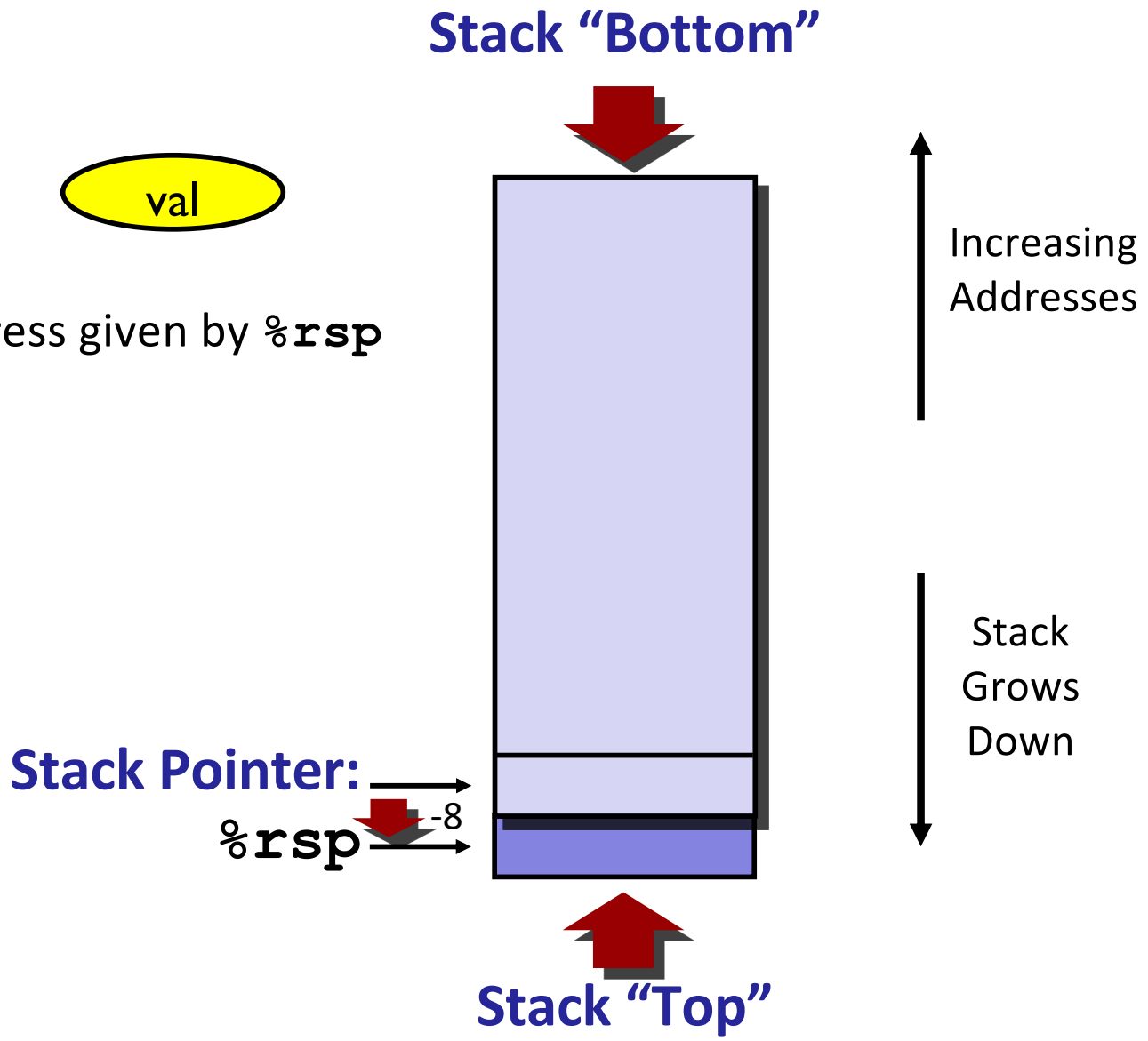
- Fetch operand at *Src*
- Decrement `%rsp` by 8
- Write operand at address given by `%rsp`



x86-64 Stack: Push

`pushq` *Src*

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

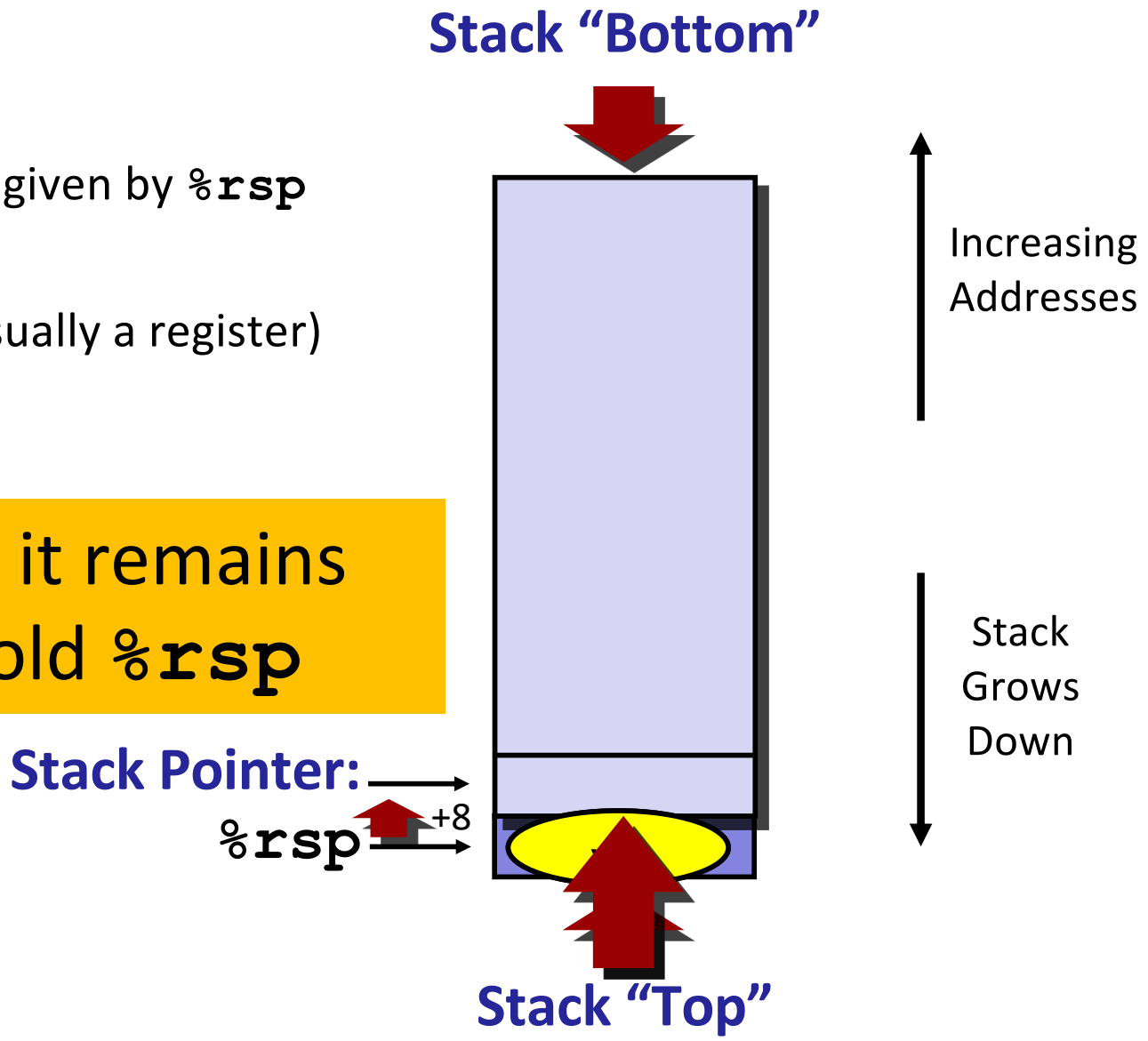


x86-64 Stack: Pop

`popq` *Dest*

- Read value at address given by **`%rsp`**
- Increment **`%rsp`** by 8
- Store value at Dest (usually a register)

Value is **copied**; it remains in memory at old **`%rsp`**



Today

Procedures

- Stack Structure
- Calling Conventions
  - **Passing control**
  - Passing data
  - Managing local data
- Illustration of Recursion

Arrays

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

Structures

- Allocation
- Access
- Alignment

Code 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: call   400550 <mult2>        # mult2(x,y)
  400549: mov    %rax,(%rbx)           # Save at dest
  40054c: pop    %rbx                  # Restore %rbx
  40054d: ret                          # 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:  ret                         # 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
- Example from disassembly

Procedure return: `ret`

- Pop address from stack
- Jump to address

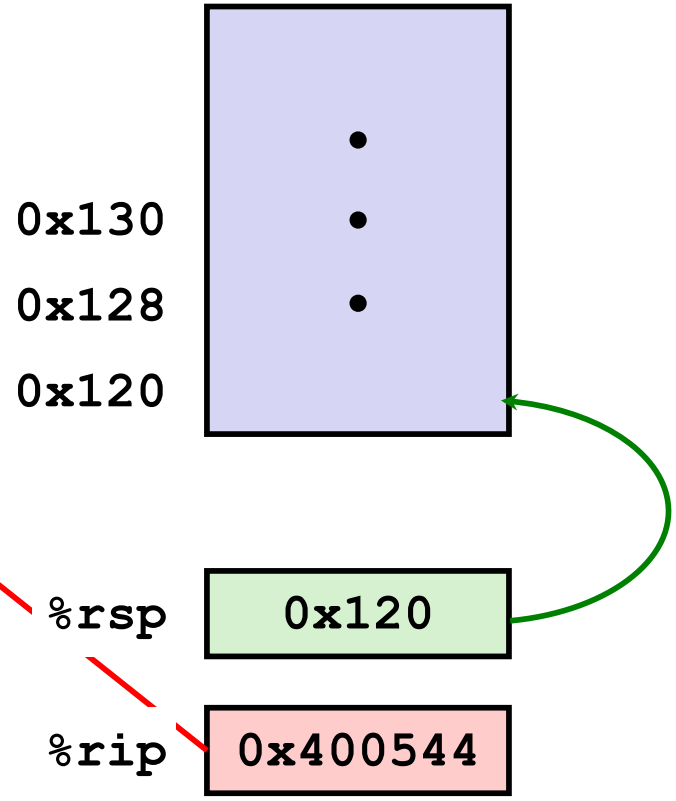
These instructions are sometimes printed with a `q` suffix

- This is just to remind you that you're looking at 64-bit code

Control Flow Example #1

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

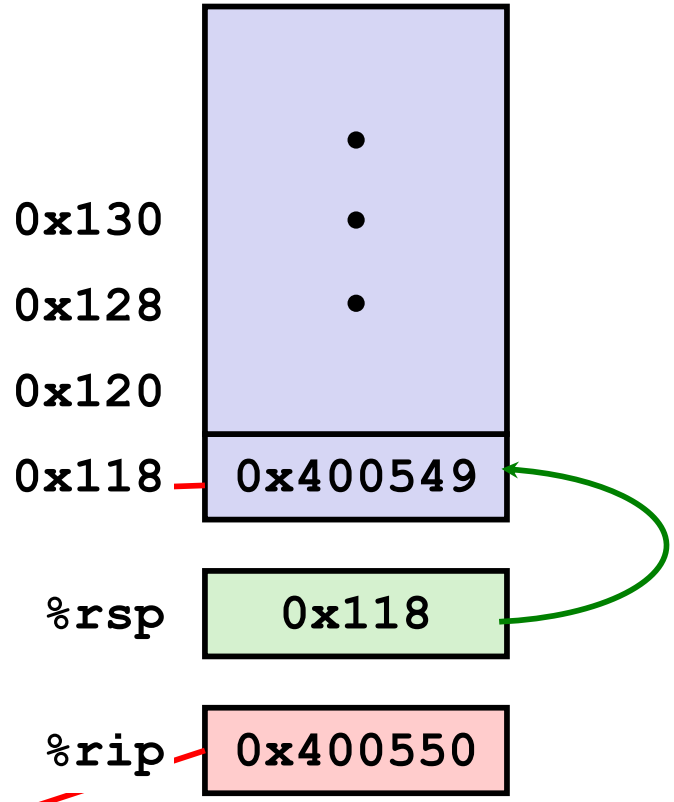
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  ret
```



Control Flow Example #2

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

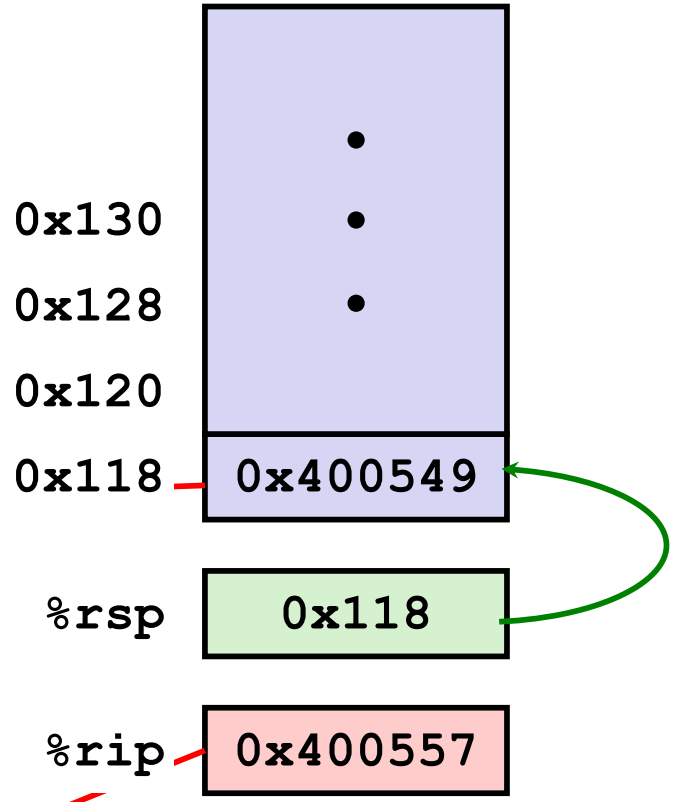
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  ret
```



Control Flow Example #3

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

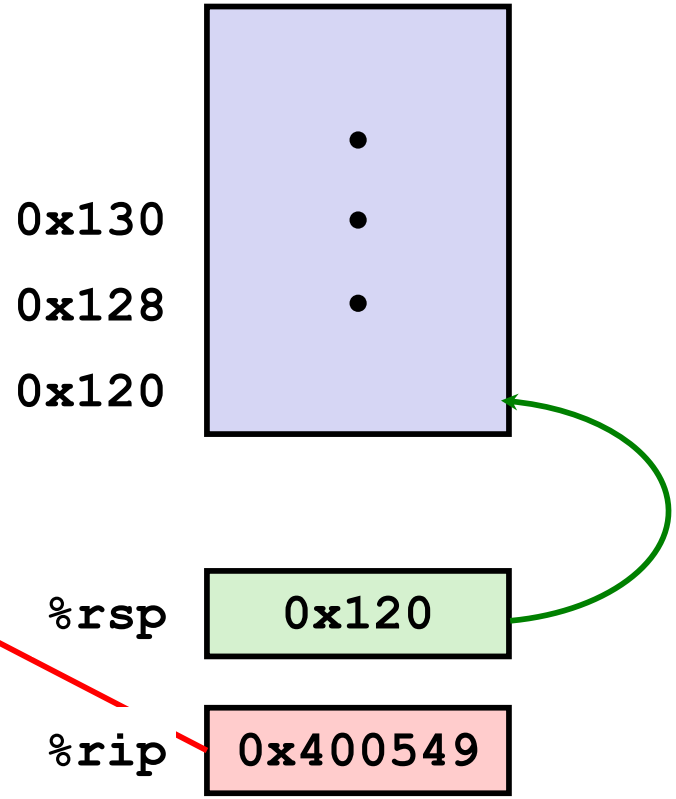
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  ret
```



Control Flow Example #4

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

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



Today

Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - **Passing data**
  - Managing local data
- Illustration of Recursion

Arrays

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

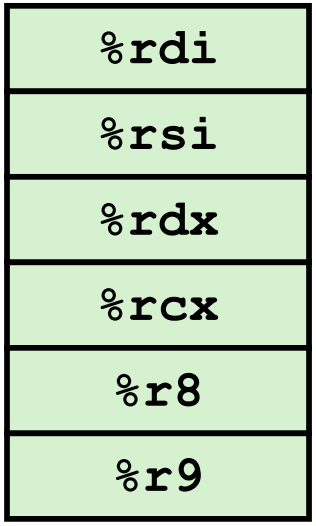
Structures

- Allocation
- Access
- Alignment

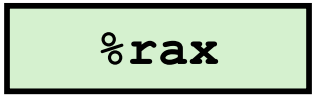
Procedure Data Flow

Registers

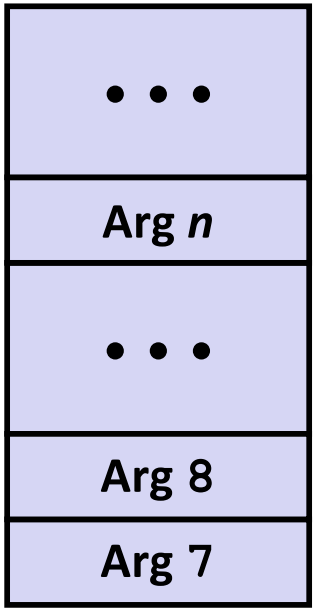
First 6 arguments



Return value



Stack



Only allocate stack space when needed

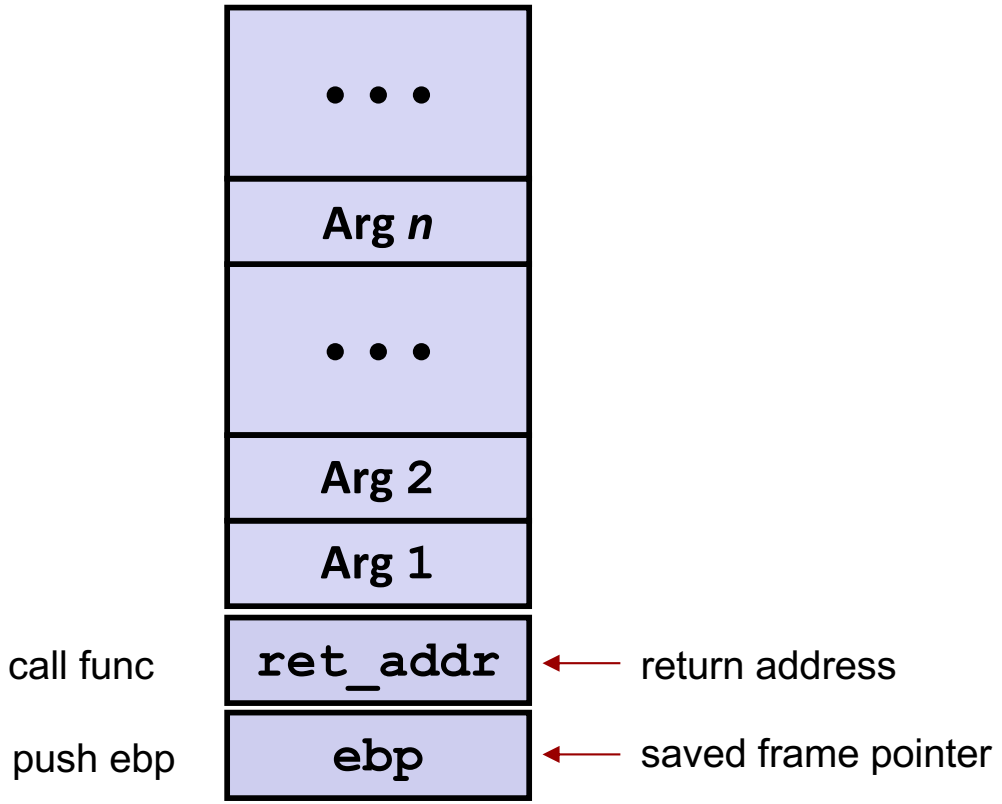
Procedure Data Flow (x86)

Registers

Return value

`%eax`

Stack



Procedure Data Flow (x86)

Prologue

```
push   %ebp
mov    %esp,%ebp
```

Epilogue

```
mov    %ebp,%esp
pop    %ebp
ret
```

or

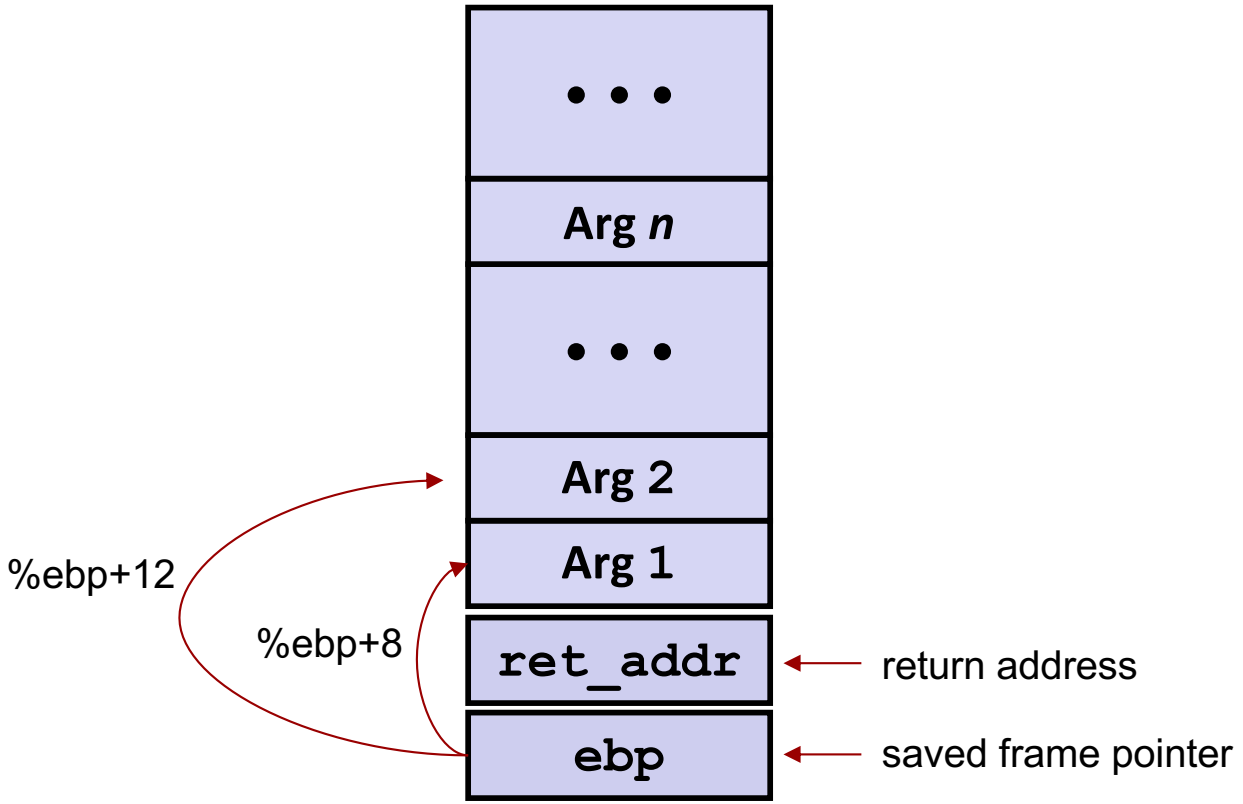
```
leave
ret
```

Procedure Data Flow (x86)

Prologue

```
push   %ebp
mov    %esp,%ebp
```

Stack



Procedure Data Flow (x86)

Example

```
#define M 9
#define N 15

int mat1[M][N];
int mat2[N][M];
int copy_element(int i, int j)
{
    mat1[i][j] = mat2[j][i];
}
```

```
copy_element:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%ecx
    movl 12(%ebp),%ebx
    movl %ecx,%edx
    leal (%ebx,%ebx,8),%eax
    sall $4,%edx
    sall $2,%eax
    subl %ecx,%edx
    movl mat2(%eax,%ecx,4),%eax
    sall $2,%edx
    movl %eax,mat1(%edx,%ebx,4)
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Procedure Data Flow (x86)

Example

```
int loop (int x, int y)
{
    int result;
    for (result=0; x>y; result++) {
        x--;
        y++;
    }
    result++;
}
```

```
loop:
        pushl %ebp
        movl %esp,%ebp
        movl 8(%ebp),%ecx
        movl 12(%ebp),%edx
        xorl %eax,%eax
        cmpl %edx,%ecx
        jle .L4
.L6:
        decl %ecx
        incl %edx
        incl %eax
        cmpl %edx,%ecx
        jg .L6
.L4:
        incl %eax
        movl %ebp,%esp
        popl %ebp
        ret
```

Procedure Data Flow (x86)

Example

```
/* copy string x to buf */
void foo(char *x) {
    int buf[1];
    strcpy((char *)buf, x);
}

void callfoo() {
    foo("abcdefghi");
}
```

Procedu

Example

```
/* copy stri
void foo(cha
    int buf
    strcpy(
}

void callfoo
    foo("abc
}
```

```
080484f4 <foo>:
080484f4:   55              pushl  %ebp
080484f5:   89 e5           movl   %esp,%ebp
080484f7:   83 ec 18        subl   $0x18,%esp
080484fa:   8b 45 08        movl   0x8(%ebp),%eax
080484fd:   83 c4 f8        addl   $0xfffffff8,%esp
08048500:   50              pushl  %eax
08048501:   8d 45 fc        leal   0xfffffffc(%ebp),%eax
08048504:   50              pushl  %eax
08048505:   e8 ba fe ff ff  call   80483c4 <strcpy>
0804850a:   89 ec           movl   %ebp,%esp
0804850c:   5d              popl   %ebp
0804850d:   c3              ret

08048510 <callfoo>:
08048510:   55              pushl  %ebp
08048511:   89 e5           movl   %esp,%ebp
08048513:   83 ec 08        subl   $0x8,%esp
08048516:   83 c4 f4        addl   $0xfffffff4,%esp
08048519:   68 9c 85 04 08  pushl  $0x804859c
0804851e:   e8 d1 ff ff ff  call   80484f4 <foo>
08048523:   89 ec           movl   %ebp,%esp
08048525:   5d              popl   %ebp
08048526:   c3              ret
```

Procedure Data Flow (x86)

Example

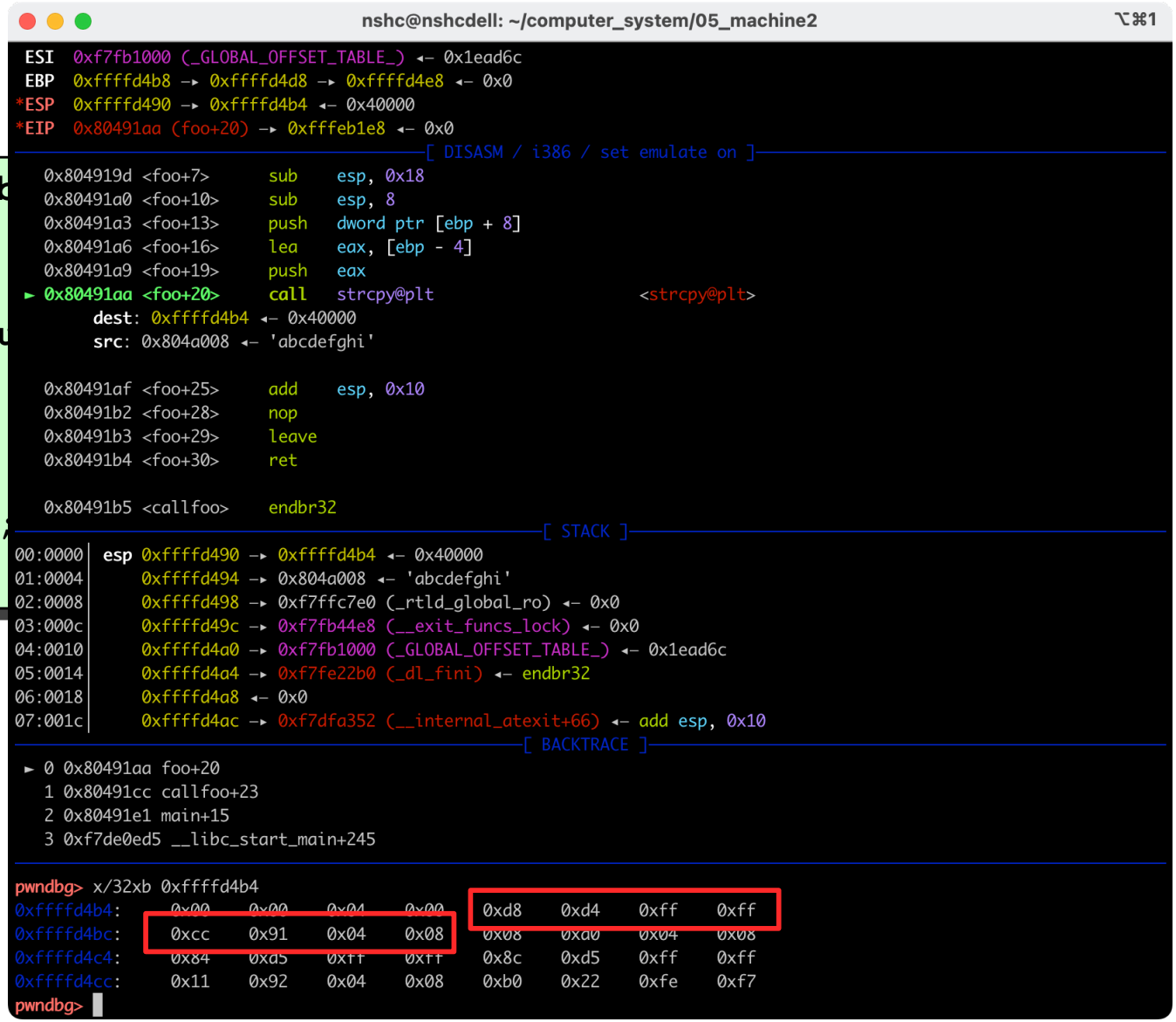
```
/* copy string x to
void foo(char *x) {
    int buf[1];
    strcpy((char *)b
}

void callfoo() {
    foo("abcdefghi")
}
```

```
nshc@nshcdell: ~/computer_system/05_machine2                    ⌥⌘1
EAX  0xffffd4b4 ◂— 0x40000
EBX  0x0
ECX  0x3e6fc73b
EDX  0xffffd514 ◂— 0x0
EDI  0xf7fb1000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1ead6c
ESI  0xf7fb1000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1ead6c
EBP  0xffffd4b8 —▸ 0xffffd4d8 —▸ 0xffffd4e8 ◂— 0x0
*ESP 0xffffd490 —▸ 0xffffd4b4 ◂— 0x40000
*EIP 0x80491aa (foo+20) —▸ 0xfffeb1e8 ◂— 0x0
[ DISASM / i386 / set emulate on ]
   0x804919d <foo+7>      sub    esp, 0x18
   0x80491a0 <foo+10>     sub    esp, 8
   0x80491a3 <foo+13>     push   dword ptr [ebp + 8]
   0x80491a6 <foo+16>     lea    eax, [ebp - 4]
   0x80491a9 <foo+19>     push   eax
 ► 0x80491aa <foo+20>     call   strcpy@plt                  <strcpy@plt>
        dest: 0xffffd4b4 ◂— 0x40000
        src: 0x804a008 ◂— 'abcdefghi'

   0x80491af <foo+25>     add    esp, 0x10
   0x80491b2 <foo+28>     nop
   0x80491b3 <foo+29>     leave
   0x80491b4 <foo+30>     ret

   0x80491b5 <callfoo>    endbr32
[ STACK ]
00:0000| esp 0xffffd490 —▸ 0xffffd4b4 ◂— 0x40000
01:0004|     0xffffd494 —▸ 0x804a008 ◂— 'abcdefghi'
02:0008|     0xffffd498 —▸ 0xf7ffc7e0 (_rtld_global_ro) ◂— 0x0
03:000c|     0xffffd49c —▸ 0xf7fb44e8 (__exit_funcs_lock) ◂— 0x0
04:0010|     0xffffd4a0 —▸ 0xf7fb1000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1ead6c
05:0014|     0xffffd4a4 —▸ 0xf7fe22b0 (_dl_fini) ◂— endbr32
06:0018|     0xffffd4a8 ◂— 0x0
07:001c|     0xffffd4ac —▸ 0xf7dfa352 (__internal_atexit+66) ◂— add esp, 0x10
[ BACKTRACE ]
 ► 0 0x80491aa foo+20
   1 0x80491cc callfoo+23
   2 0x80491e1 main+15
   3 0xf7de0ed5 __libc_start_main+245
pwndbg>
```

Procedure Data Flow (x86)

Example

```
/* copy string x to b
void foo(char *x) {
    int buf[1];
    strcpy((char *)bu
}

void callfoo() {
    foo("abcdefghi");
}
```



Procedure Data Flow (x86)

Example

```
/* copy string x to b
void foo(char *x) {
    int buf[1];
    strcpy((char *)bu
}

void callfoo() {
    foo("abcdefghi");
}
```

```
nshc@nshcdell: ~/computer_system/05_machine2                    ⌥⌘1
*ECX  0x804a008 ◂— 'abcdefghi'
*EDX  0xffffd4b4 ◂— 'abcdefghi'
 EDI  0xf7fb1000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1ead6c
 ESI  0xf7fb1000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1ead6c
 EBP  0xffffd4b8 ◂— 'efghi'
 ESP  0xffffd490 —▸ 0xffffd4b4 ◂— 'abcdefghi'
*EIP  0x80491af (foo+25) ◂— add esp, 0x10
──────────────────[ DISASM / i386 / set emulate on ]──────────────────
   0x80491a0 <foo+10>       sub    esp, 8
   0x80491a3 <foo+13>       push   dword ptr [ebp + 8]
   0x80491a6 <foo+16>       lea    eax, [ebp - 4]
   0x80491a9 <foo+19>       push   eax
   0x80491aa <foo+20>       call   strcpy@plt                  <strcpy@plt>

 ► 0x80491af <foo+25>       add    esp, 0x10
   0x80491b2 <foo+28>       nop
   0x80491b3 <foo+29>       leave
   0x80491b4 <foo+30>       ret

   0x80491b5 <callfoo>      endbr32
   0x80491b9 <callfoo+4>    push   ebp
──────────────────────[ STACK ]──────────────────────
00:0000│ esp 0xffffd490 —▸ 0xffffd4b4 ◂— 'abcdefghi'
01:0004│     0xffffd494 —▸ 0x804a008 ◂— 'abcdefghi'
02:0008│     0xffffd498 —▸ 0xf7ffc7e0 (_rtld_global_ro) ◂— 0x0
03:000c│     0xffffd49c —▸ 0xf7fb44e8 (__exit_funcs_lock) ◂— 0x0
04:0010│     0xffffd4a0 —▸ 0xf7fb1000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1ead6c
05:0014│     0xffffd4a4 —▸ 0xf7fe22b0 (_dl_fini) ◂— endbr32
06:0018│     0xffffd4a8 ◂— 0x0
07:001c│     0xffffd4ac —▸ 0xf7dfa352 (__internal_atexit+66) ◂— add esp, 0x10
──────────────────[ BACKTRACE ]──────────────────
 ► 0 0x80491af foo+25
   1 0x8040069
   2 0x804a008
──────────────────────────────────────────────────
pwndbg> x/32xb 0xffffd4b4
0xffffd4b4:     0x61    0x62    0x63    0x64    0x65    0x66    0x67    0x68
0xffffd4bc:     0x69    0x00    0x04    0x08    0x08    0xa0    0x04    0x08
0xffffd4c4:     0x84    0xd5    0xff    0xff    0x8c    0xd5    0xff    0xff
0xffffd4cc:     0x11    0x92    0x04    0x08    0xb0    0x22    0xfe    0xf7
pwndbg>
```

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

Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - **Managing local data**
- Illustration of Recursion

Arrays

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

Structures

- Allocation
- Access
- Alignment

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

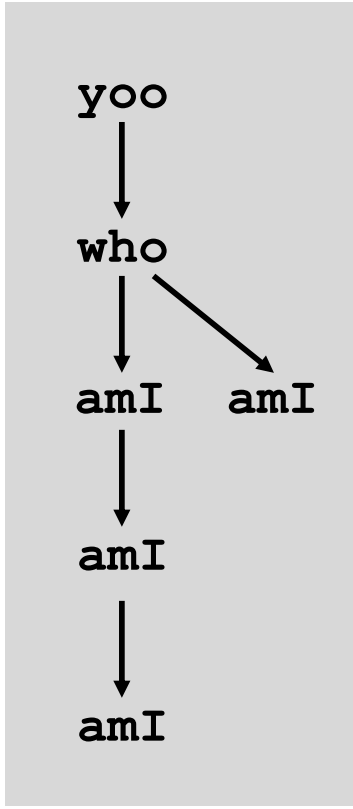
Call Chain Example

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

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

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

Example Call Chain

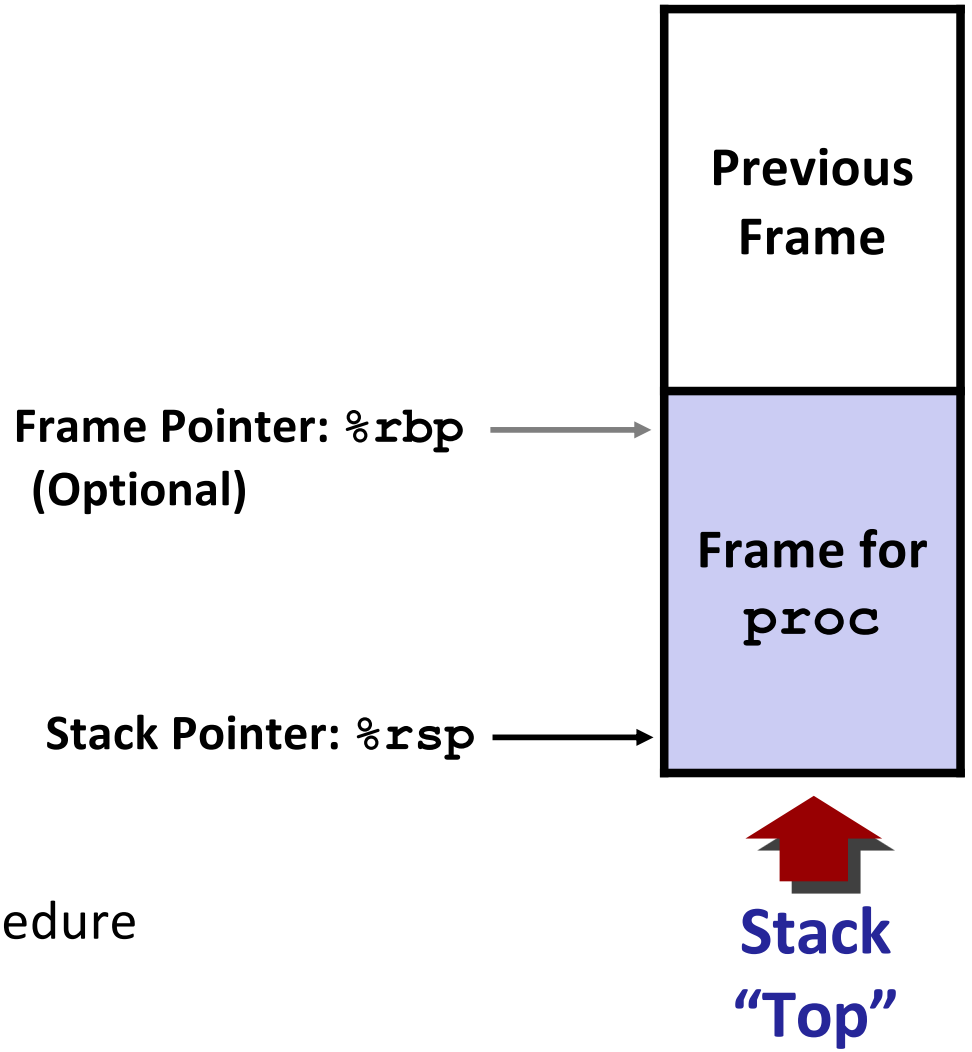


Procedure amI() is recursive

Stack Frames

Contents

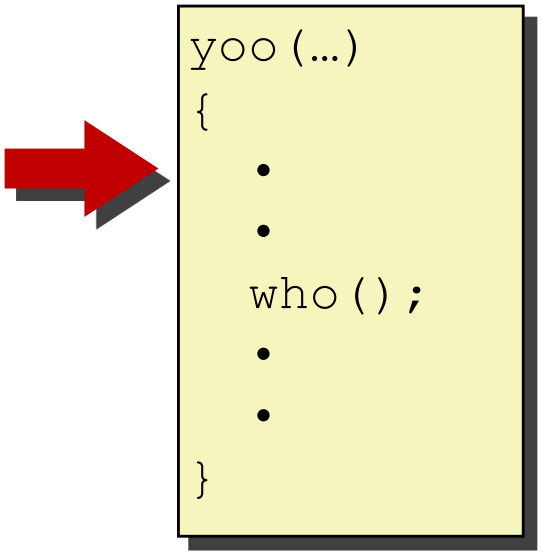
- Return information
- Local storage (if needed)
- Temporary space (if needed)

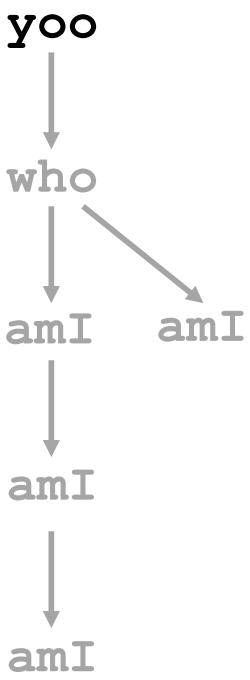


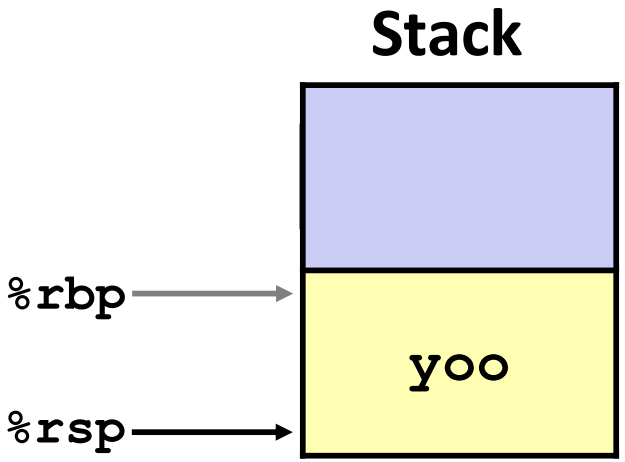
Management

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

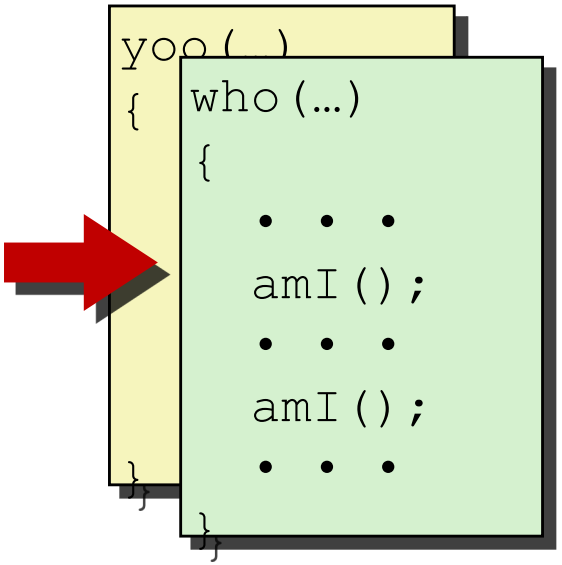
Example

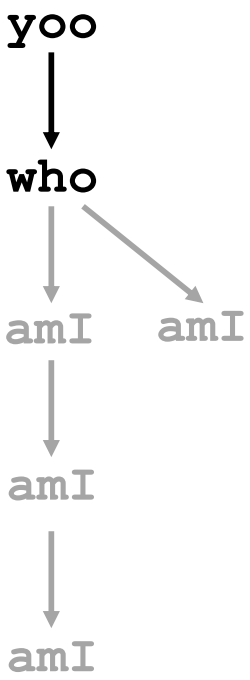


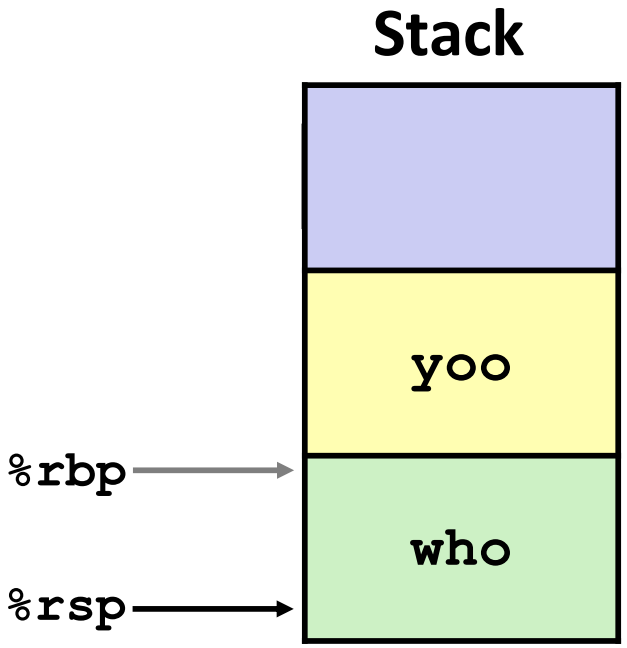




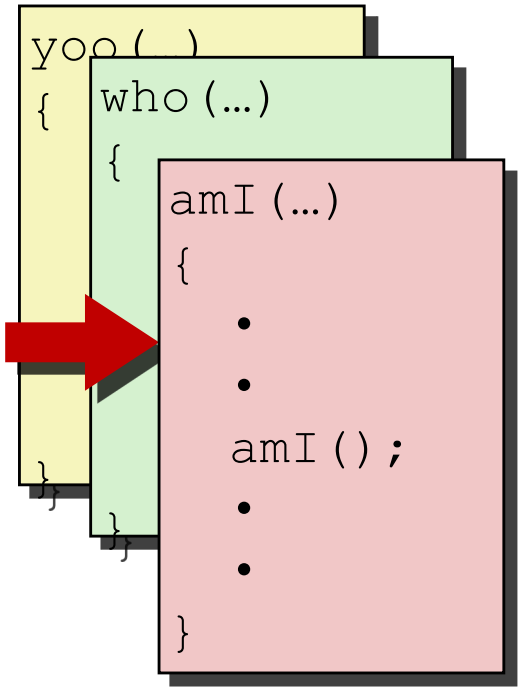
Example

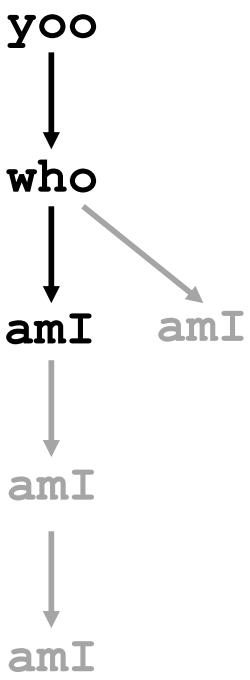


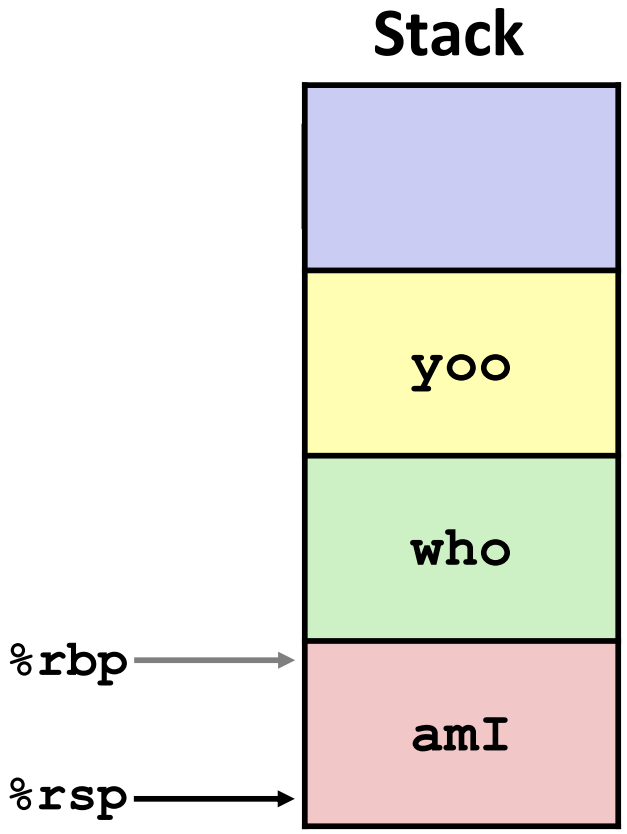




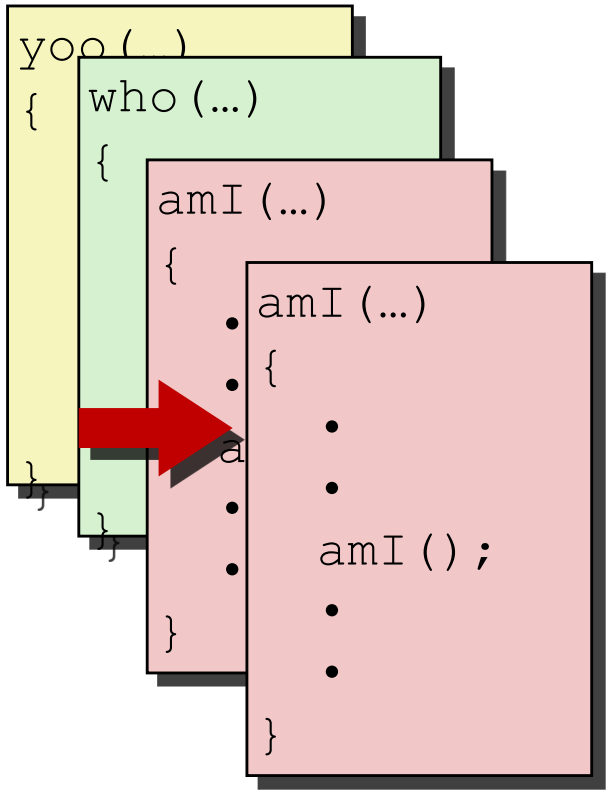
Example

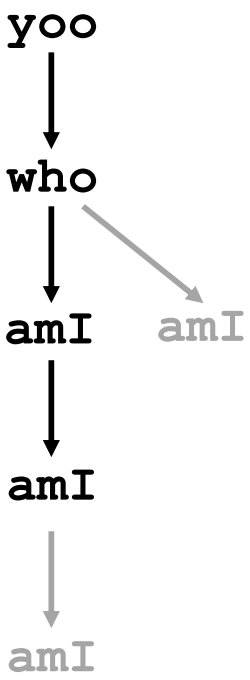

yoo(…)
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

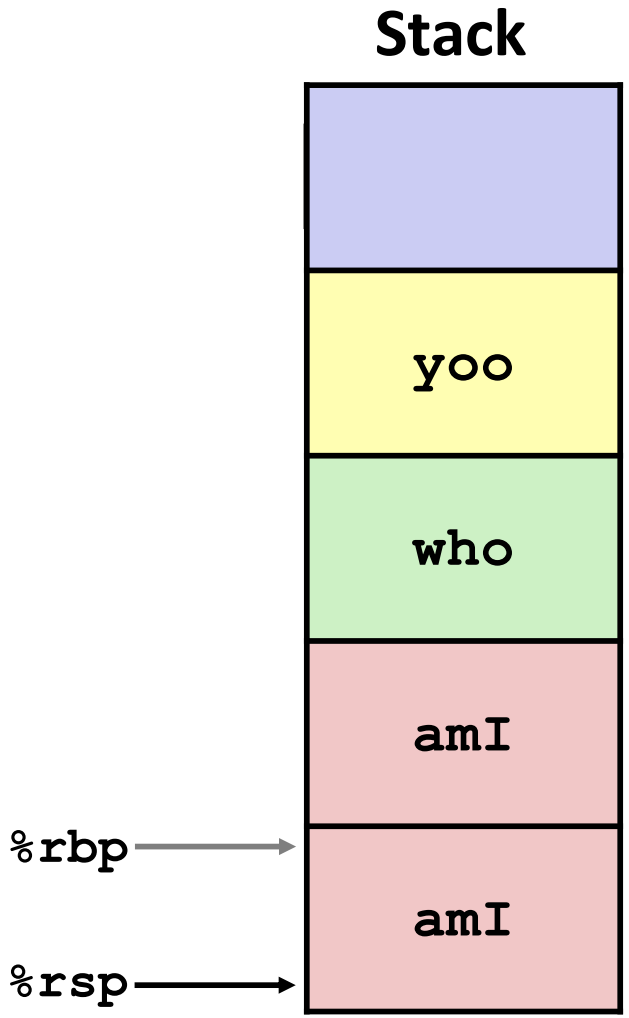

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

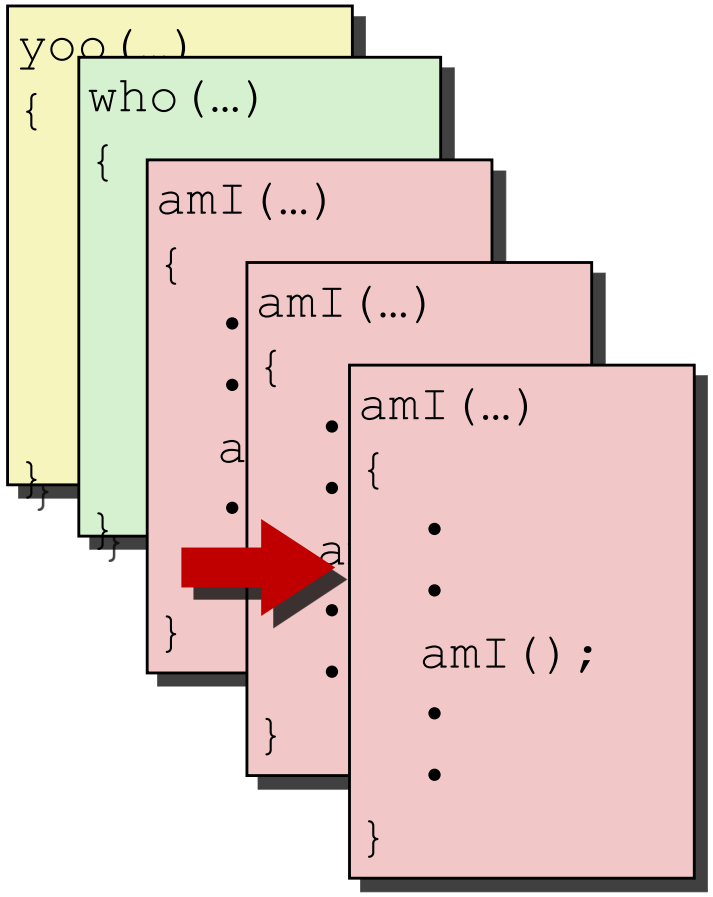
Example

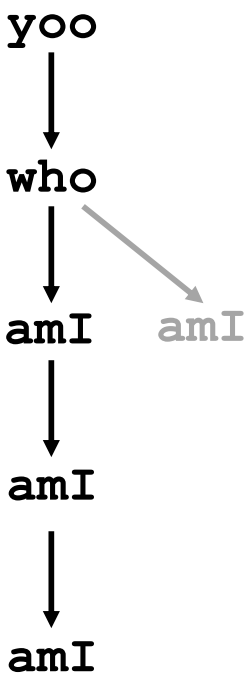

yoo(…)
{
who(…)
{
amI(…)
{
amI(…)
{
amI();
}

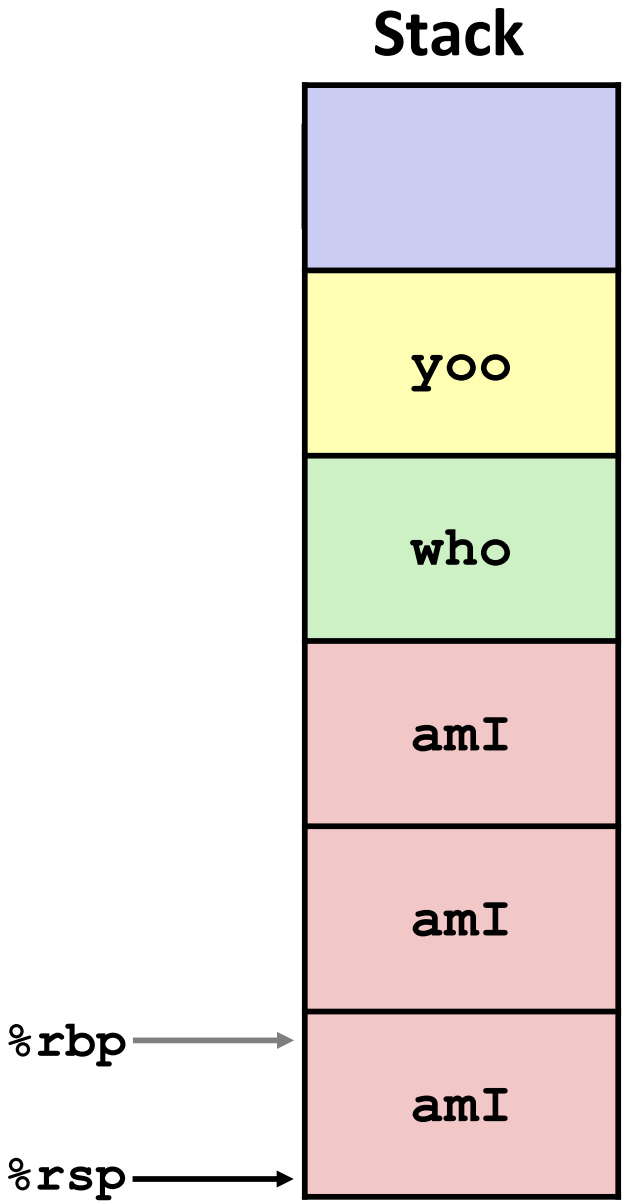

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

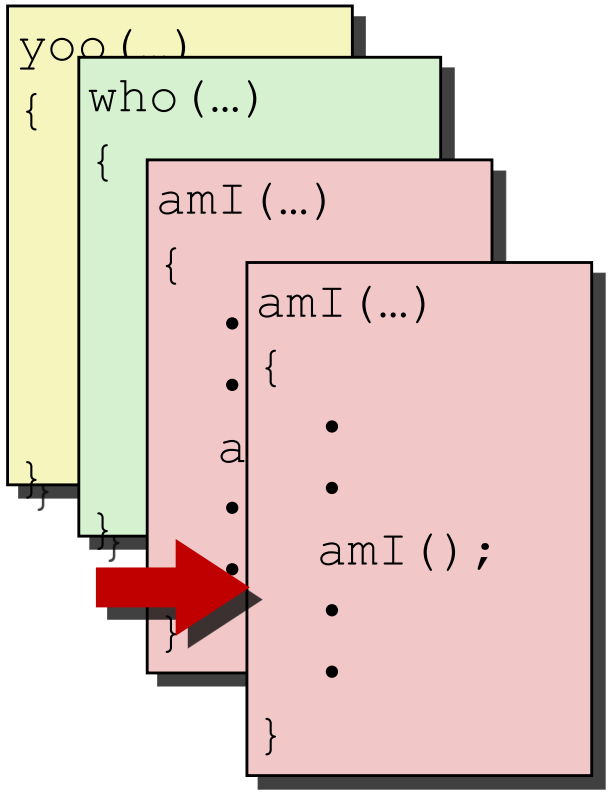
Example

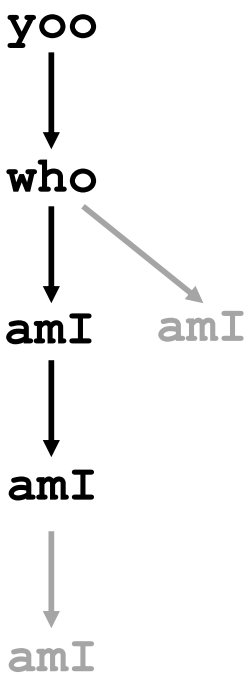

yoo(…)
{
who(…)
{
amI(…)
{
amI(…)
{
amI(…)
{
amI();
}

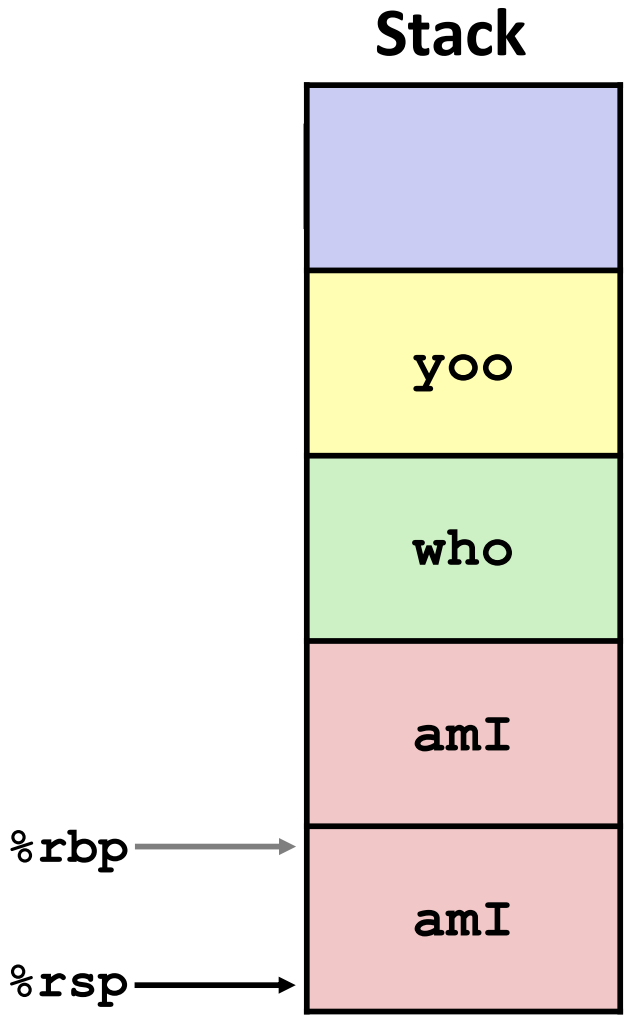

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
amI
%rbp
%rsp

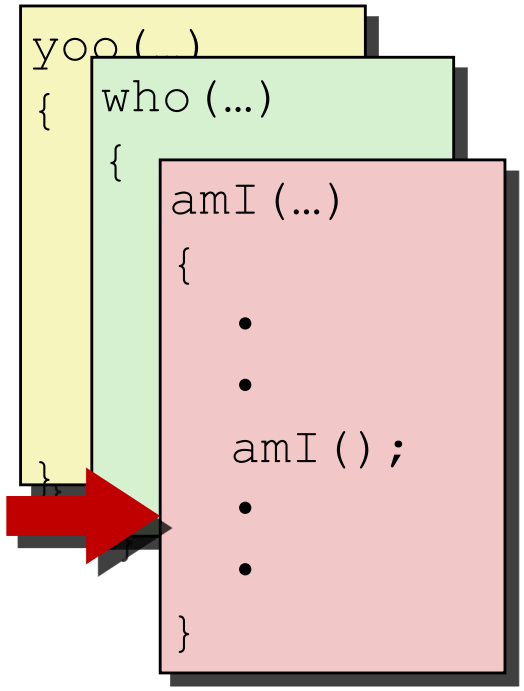
Example

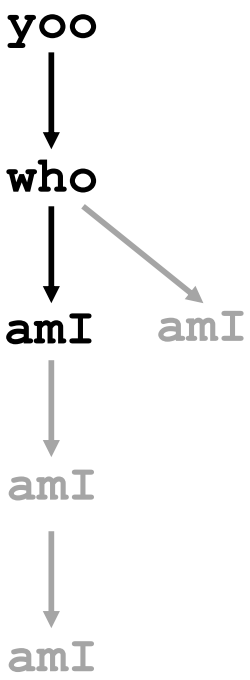

yoo(…)
{
who(…)
{
amI(…)
{
amI(…)
{
amI();
}

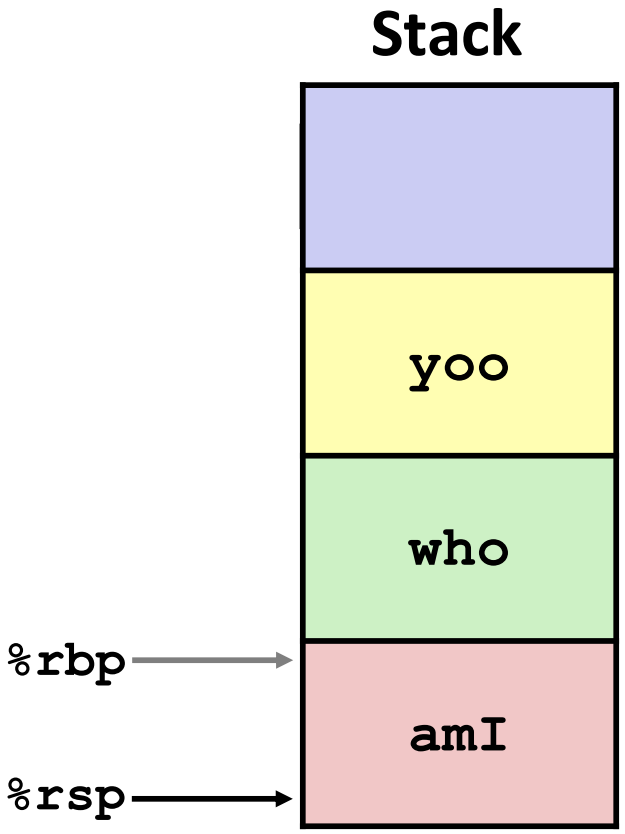

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

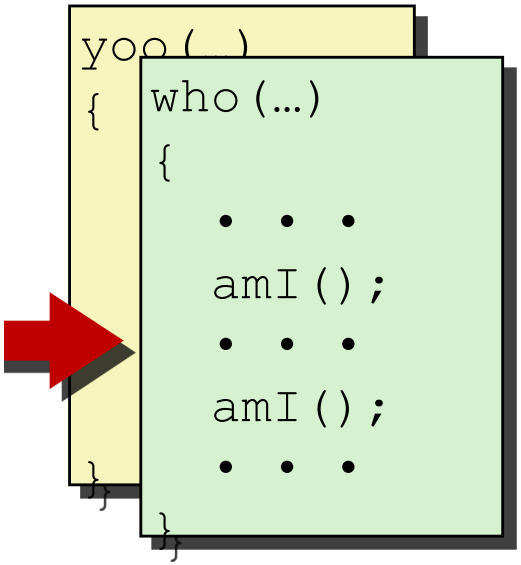
Example

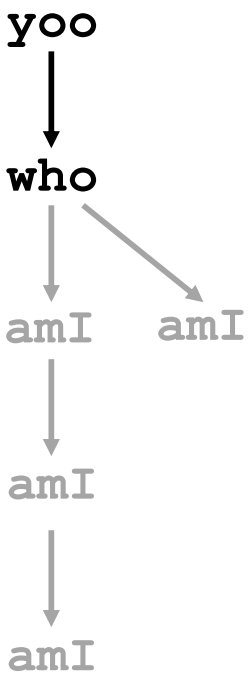

yoo(…)
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}

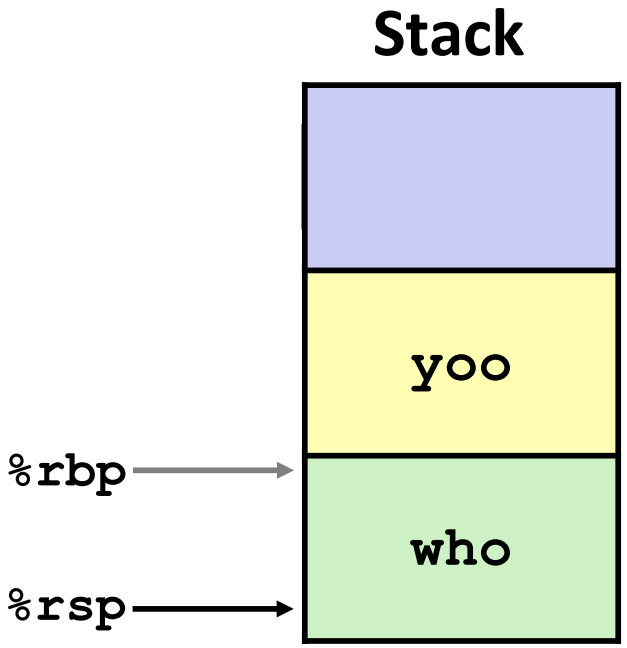

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

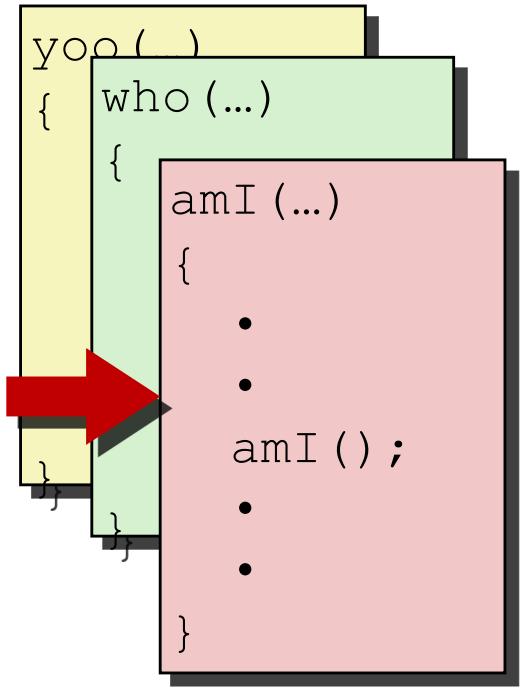
Example

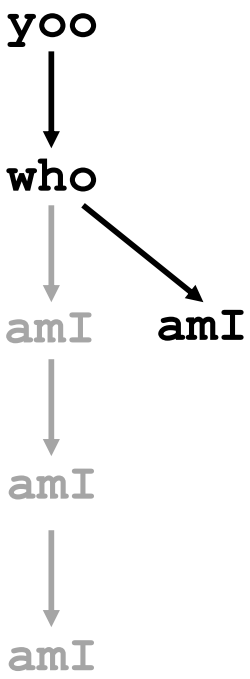


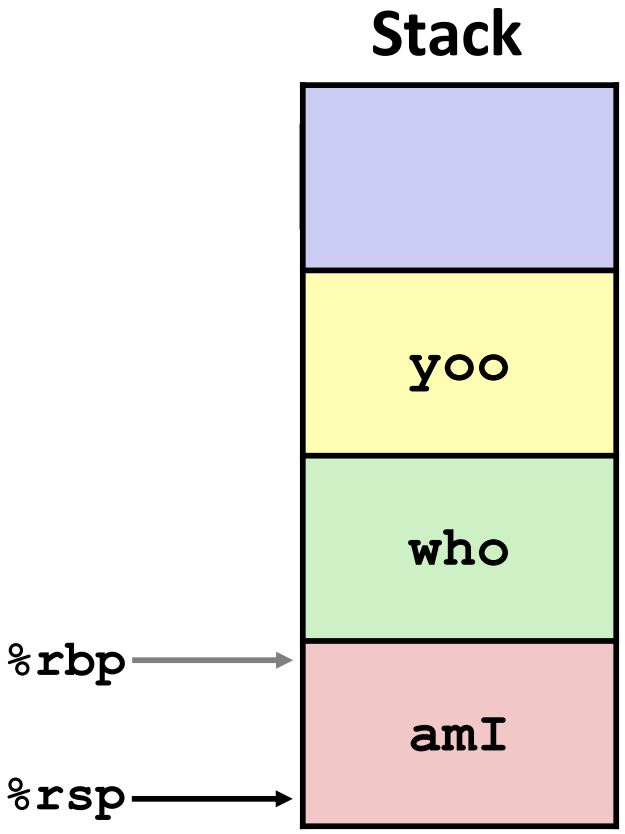




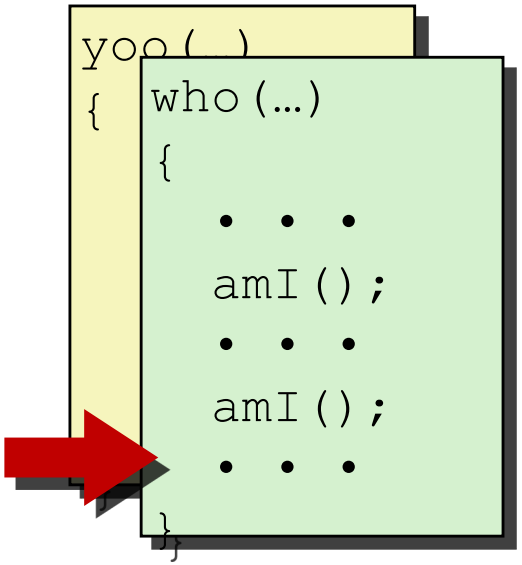
Example

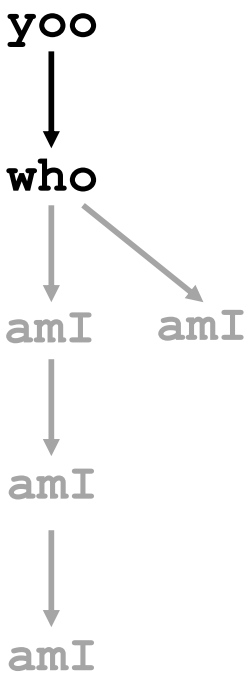

yoo( )
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

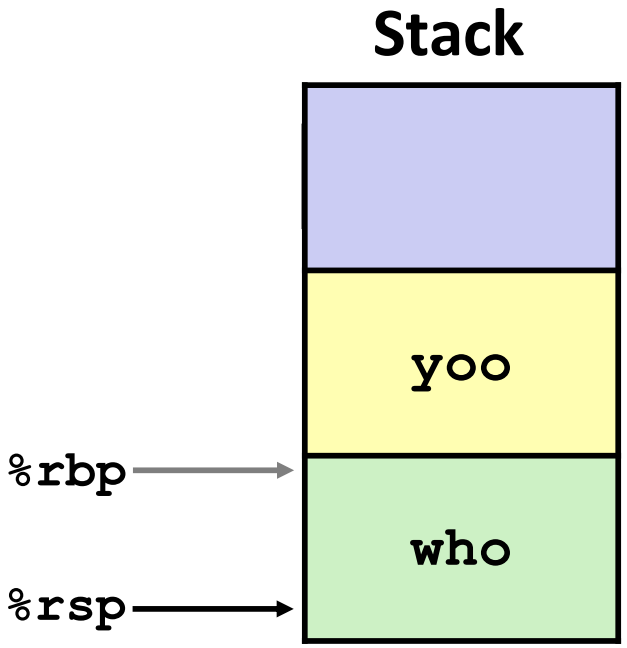

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

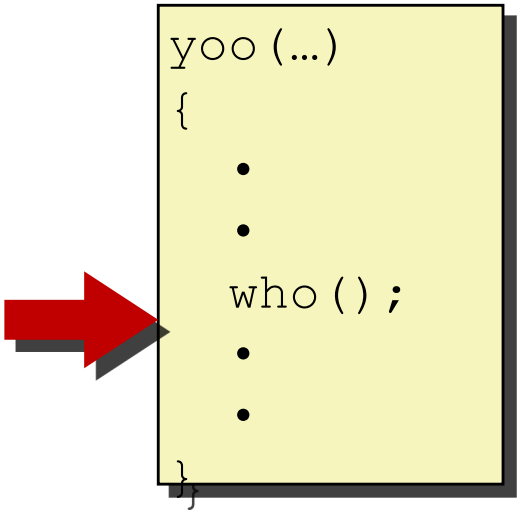
Example

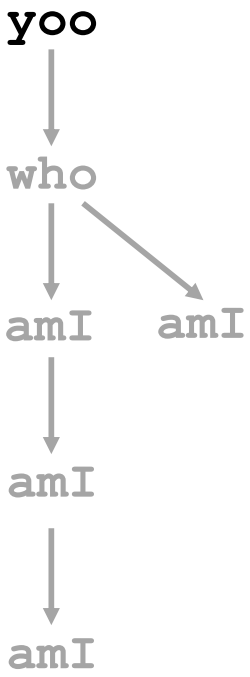


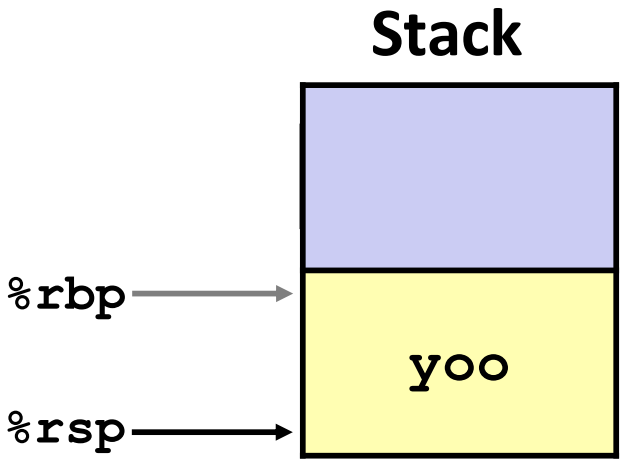




Example







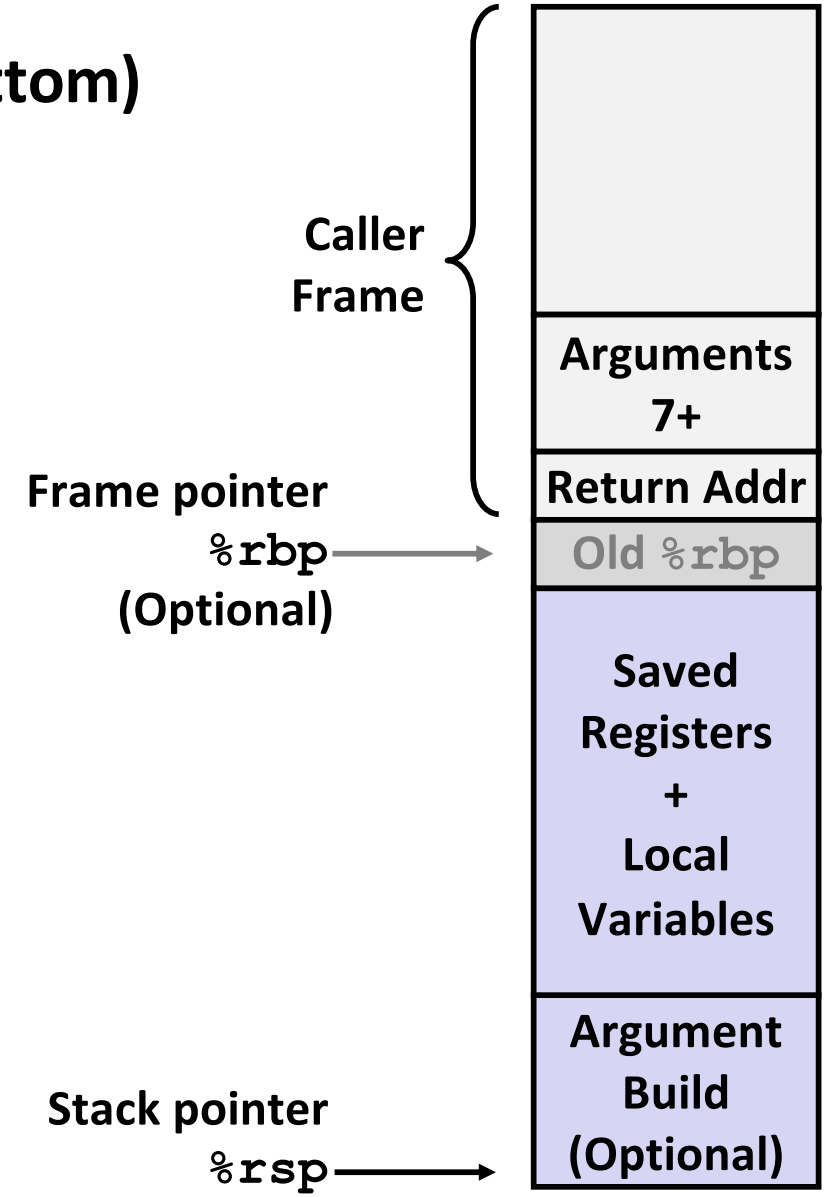
x86-64/Linux Stack Frame

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
```

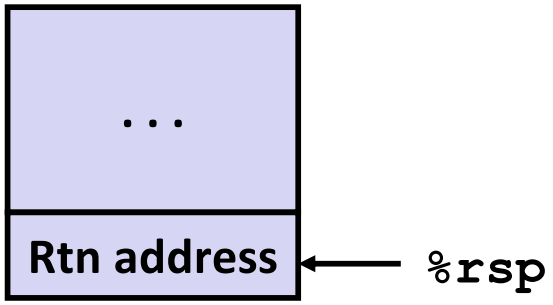
Register	Use(s)
`%rdi`	Argument **`p`**
`%rsi`	Argument **`val`**, **`y`**
`%rax`	**`x`**, Return value

Example: Calling `incr` #1

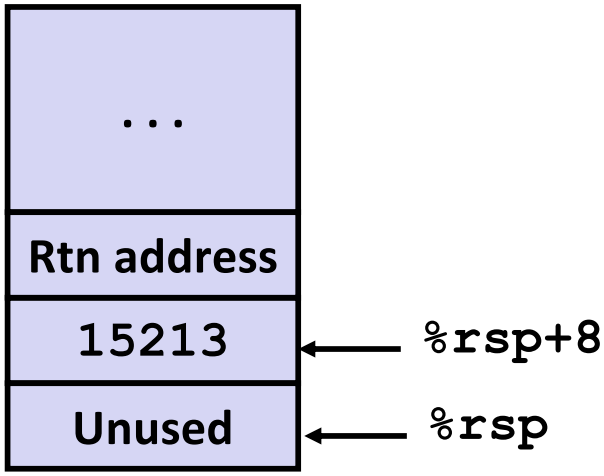
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
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

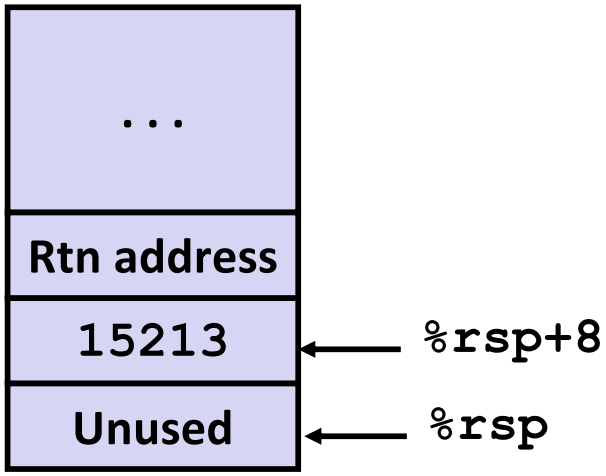


Example: Calling `incr` #2

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

```
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

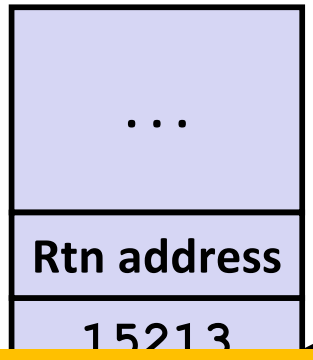


Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling `incr` #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.

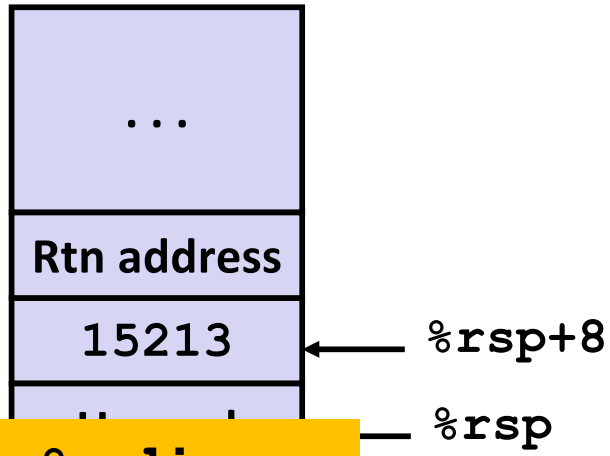
```
ca
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

%rdi	&v1
%rsi	3000

Example: Calling `incr` #2

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

Stack Structure




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

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

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

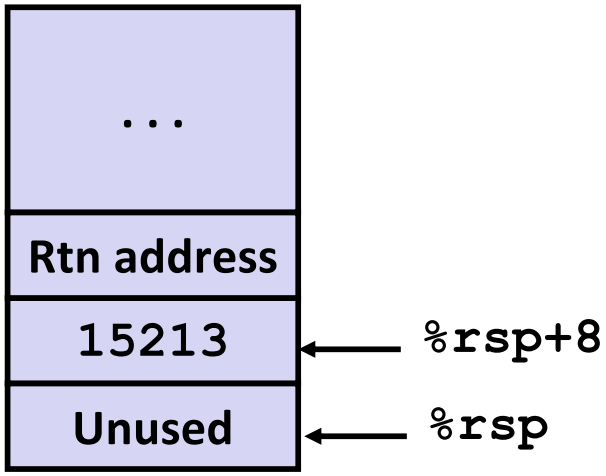
se(s)	
	v1
%rsi	3000

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



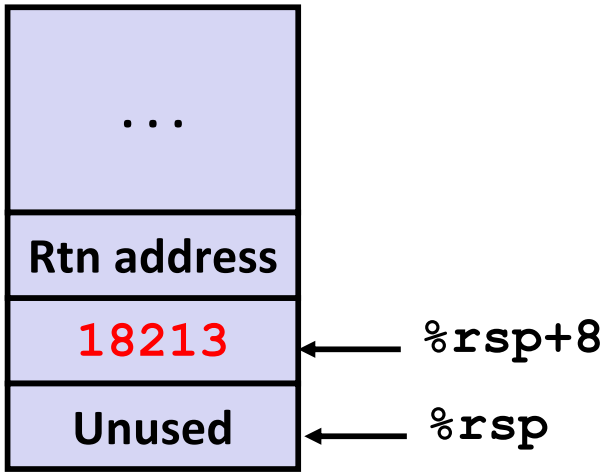
Register	Use(s)
%rdi	&v1
%rsi	3000

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

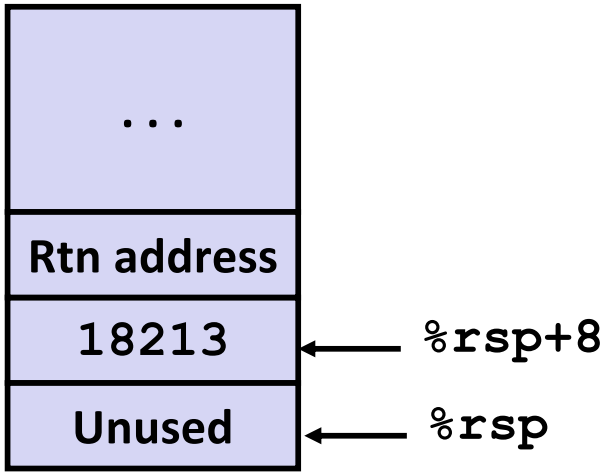


Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling `incr` #4

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

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
```

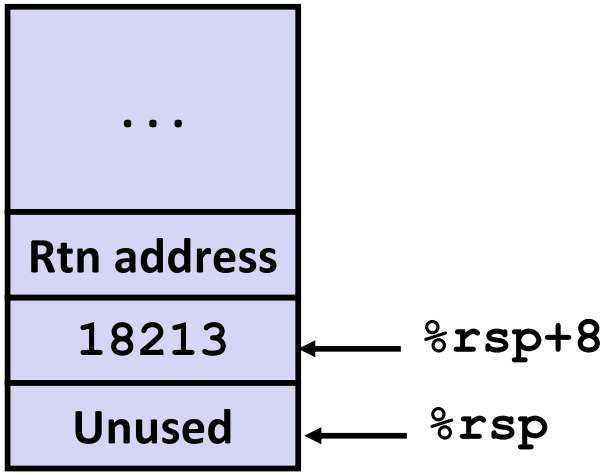
Register	Use(s)
%rax	Return value

Example: Calling `incr` #5a

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

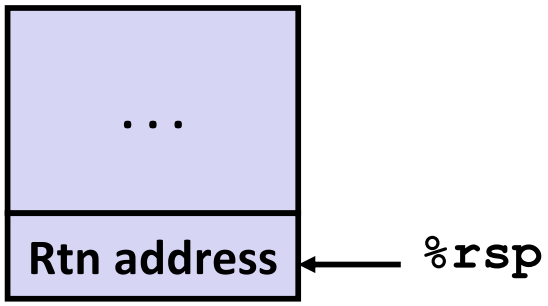
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



Register	Use(s)
%rax	Return value

Updated Stack Structure

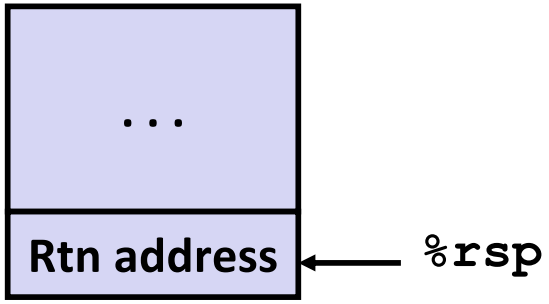


Example: Calling `incr` #5b

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

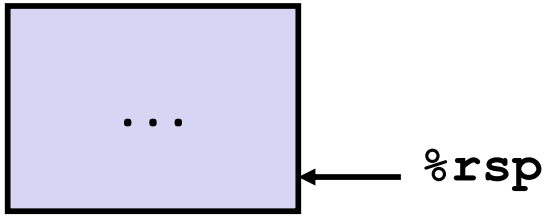
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Updated Stack Structure



Register	Use(s)
%rax	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`

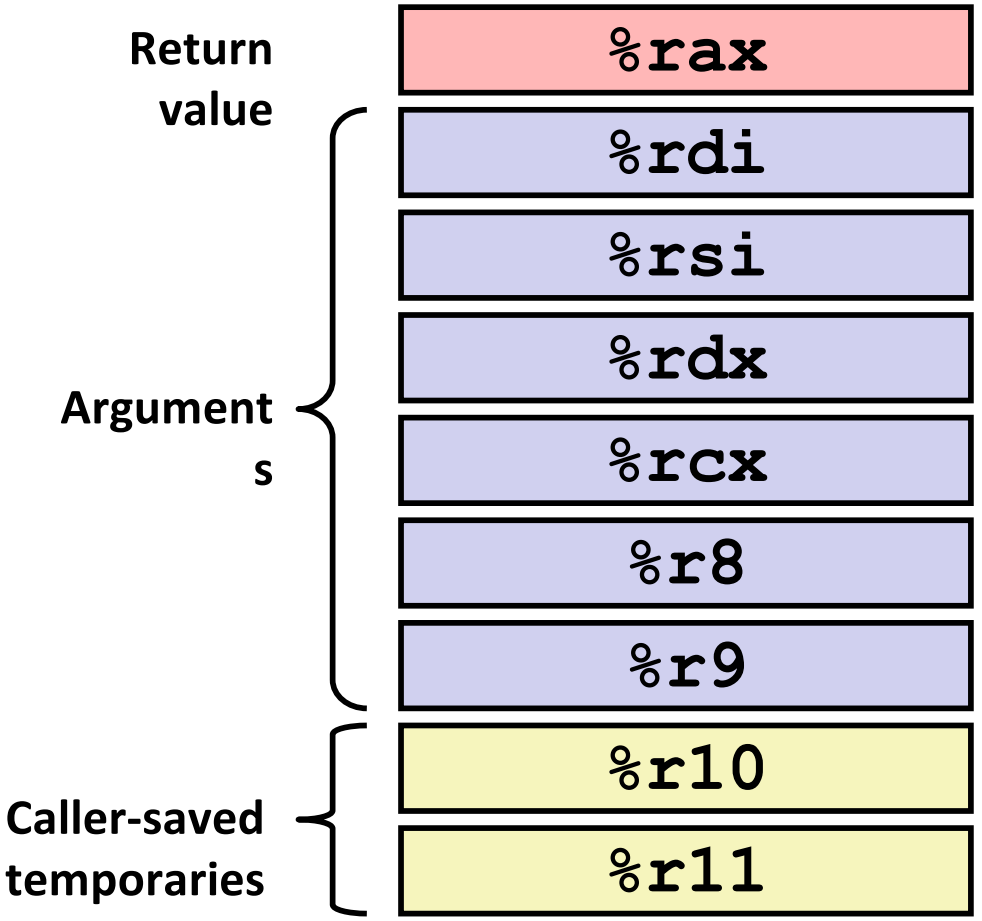
- Return value
- Also caller-saved
- Can be modified by procedure

`%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`

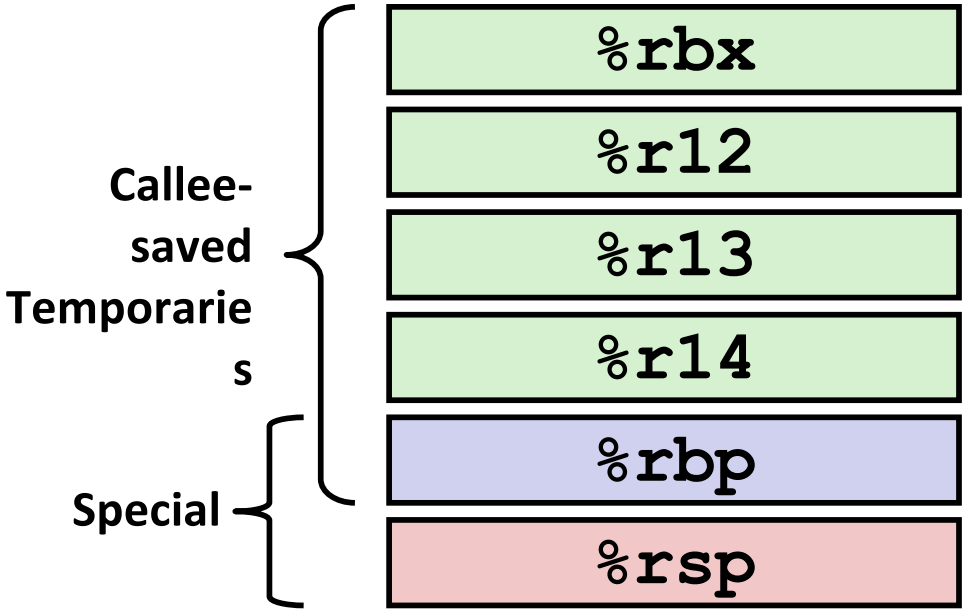
- Callee-saved
- Callee must save & restore

`%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



Today

Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- **Illustration of Recursion**

Arrays

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

Structures

- Allocation
- Access
- Alignment

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
```

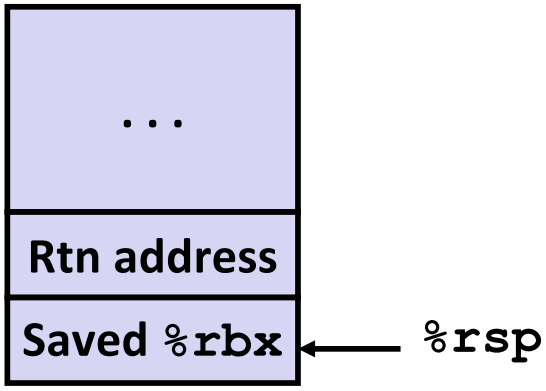
Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

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
```

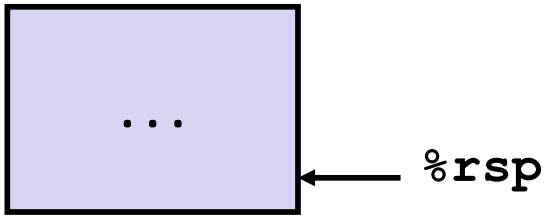
Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

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

x86-64 Procedure Summary

Important Points

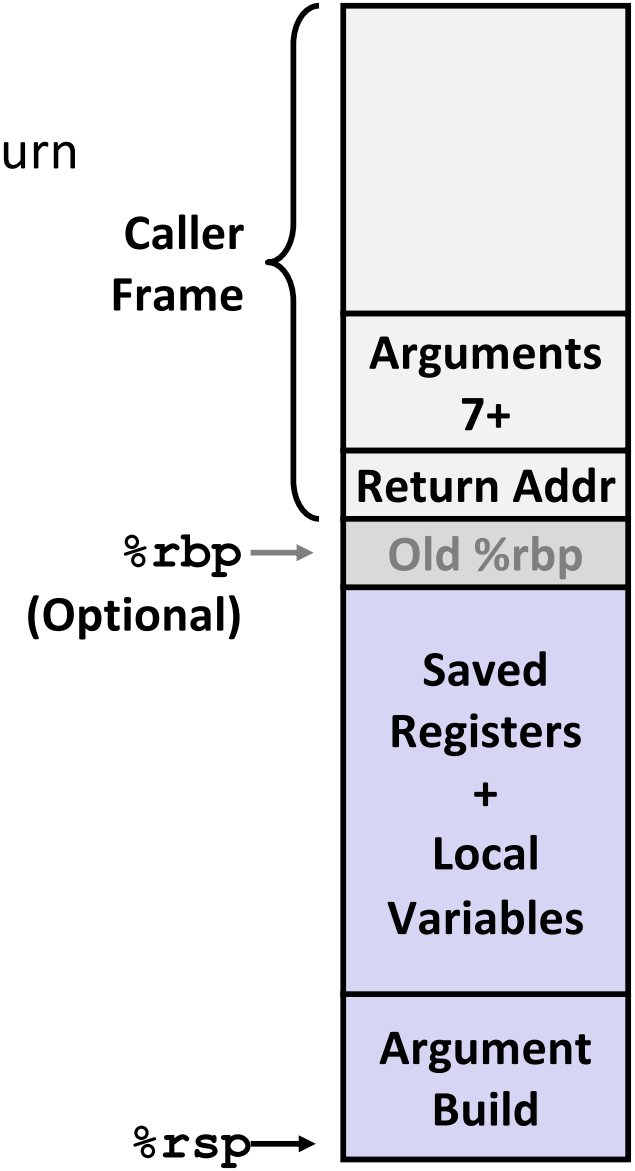
- Stack is the right data structure for procedure call/return
  - If P calls Q, then Q returns before P

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**



Today

Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Illustration of Recursion

Arrays

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

Structures

- Allocation
- Access
- Alignment

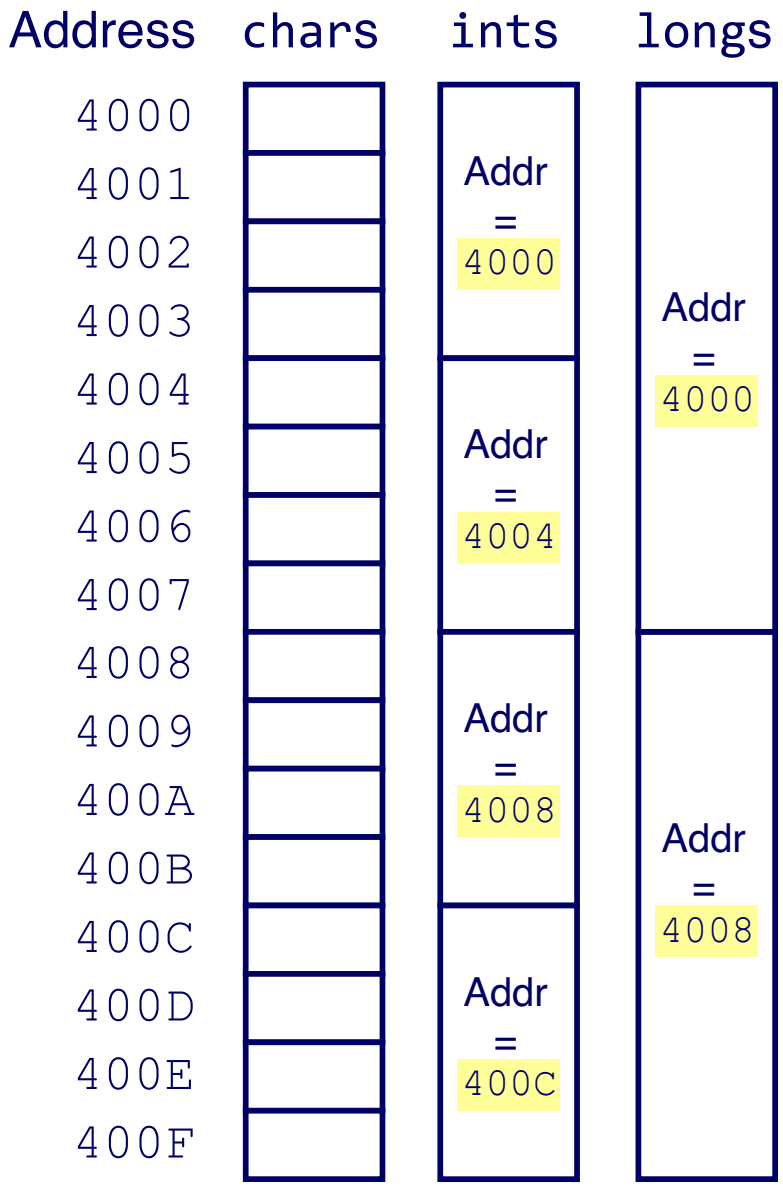
Reminder: Memory Organization

Memory locations do not have data types

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

Addresses specify byte locations

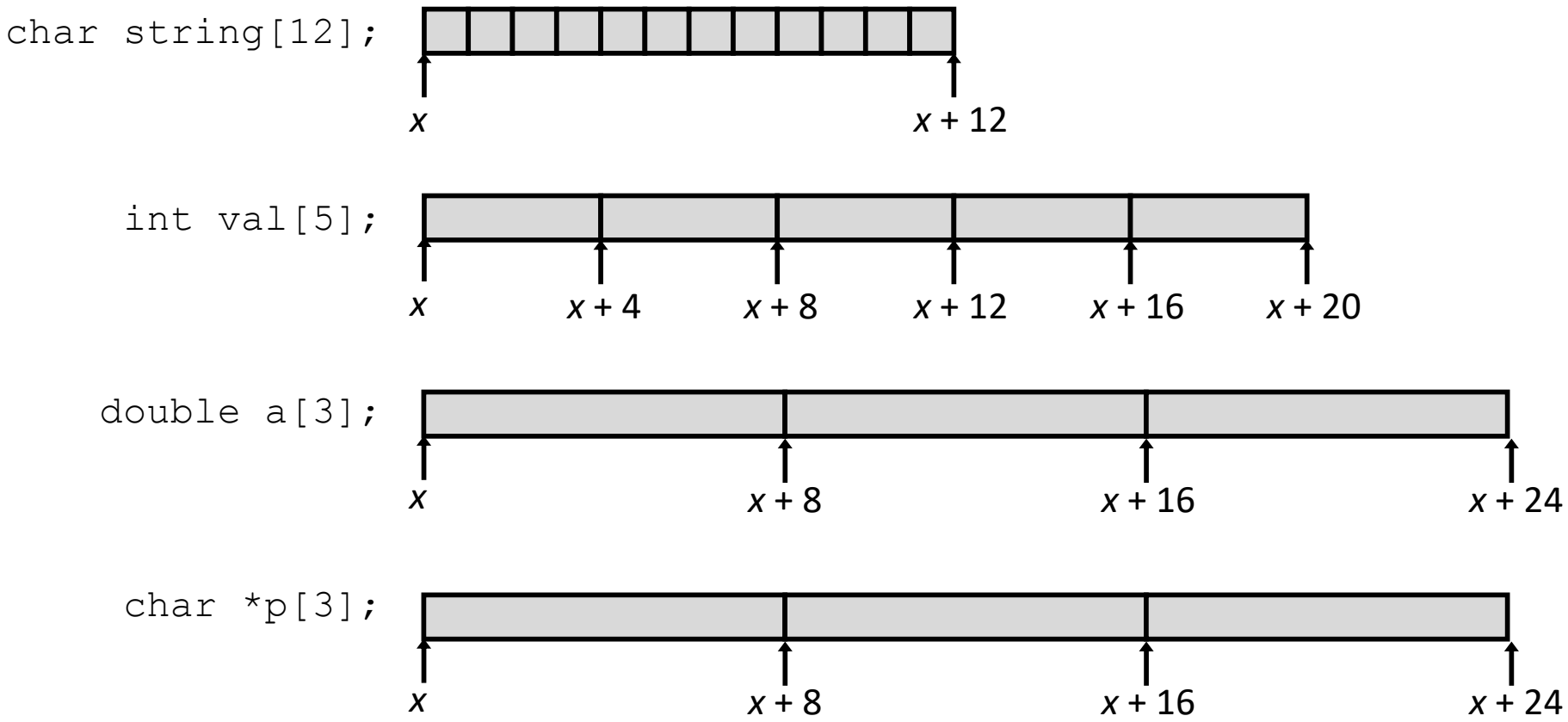
- Address of a larger datum is the address of its first byte
- Addresses of successive items differ by the item's size



Array Allocation

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

- Array of data type *Type* and length *Length*
- Contiguously allocated region of *Length* * `sizeof(`*Type*`)` bytes in memory

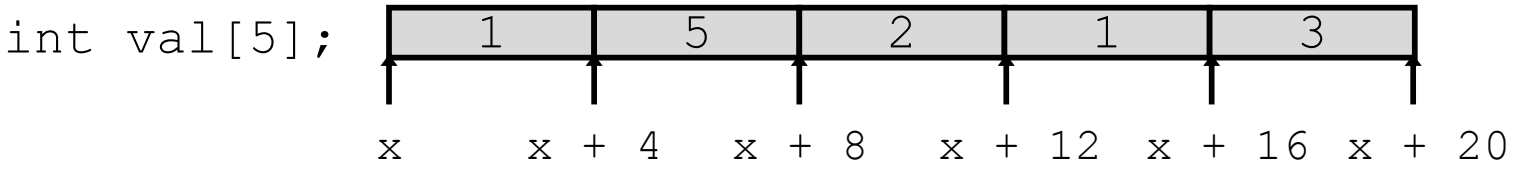


Array Access

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

- Array of data type *Type* and length *Length*

- Identifier **name** acts like[1] a pointer to array element 0

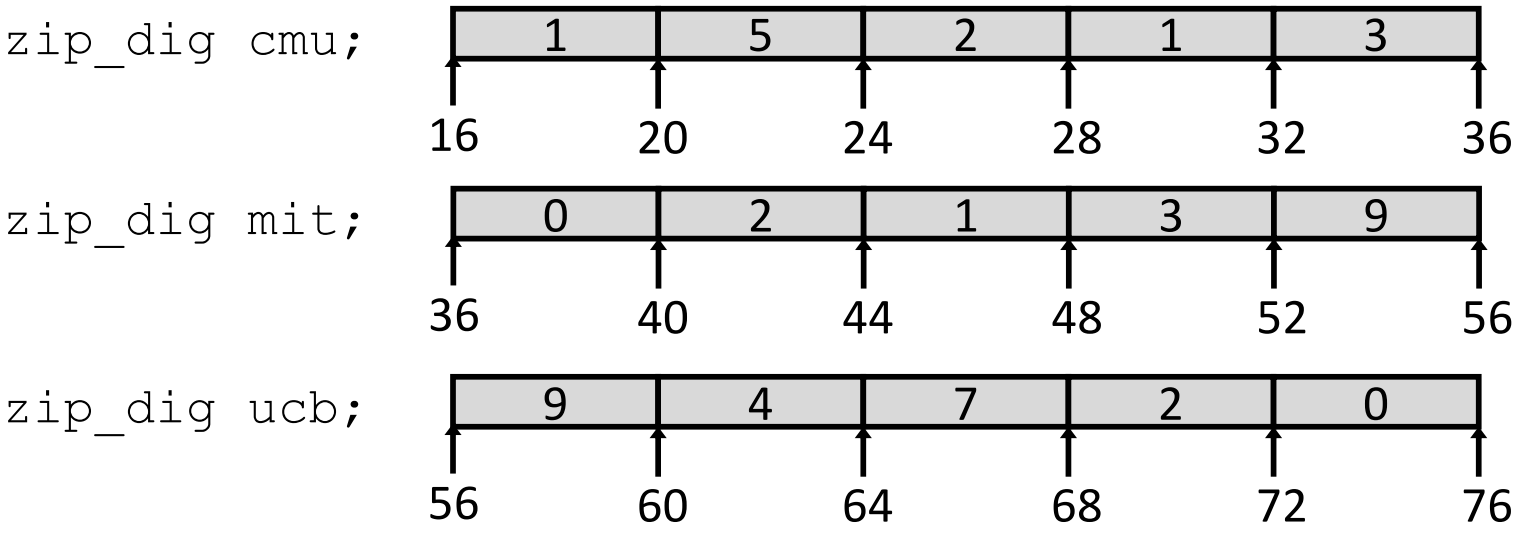


Expression	Type	Value	
`val[4]`	`int`	`3`	
`val[5]`	`int`	`??`	`// access past end`
`*(val+3)`	`int`	`1`	`// same as val[3]`
`val`	`int *`	`x`	
`val+1`	`int *`	`x + 4`	
`&val[2]`	`int *`	`x + 8`	`// same as val+2`
`val + i`	`int *`	`x + 4*i`	`// same as &val[i]`

[1] in most contexts (but not all)

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 };
```



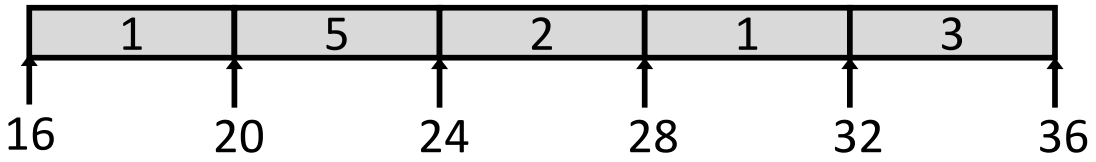
Declaration "`zip_dig cmu`" equivalent to "`int cmu[5]`"

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
```

Understanding Pointers & Arrays #1

Decl	A1 , A2			*A1 , *A2		
	Comp	Bad	Size	Comp	Bad	Size
`int A1[3]`						
`int *A2`						

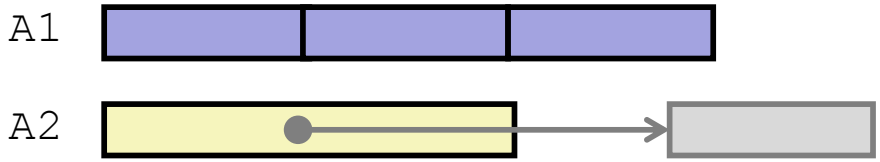
Comp: Compiles (Y/N)

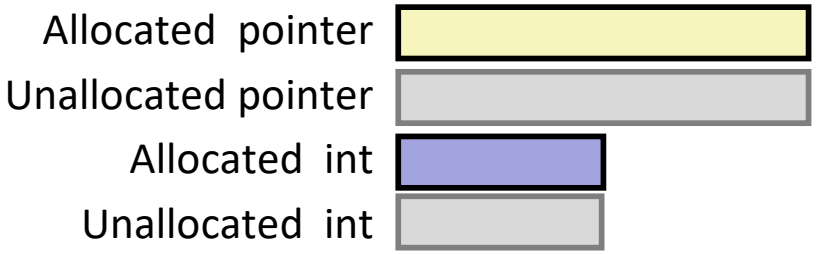
Bad: Possible bad pointer reference (Y/N)

Size: Value returned by `sizeof`

Understanding Pointers & Arrays #1

Decl	A1 , A2			*A1 , *A2		
	Comp	Bad	Size	Comp	Bad	Size
`int A1[3]`						
`int *A2`						





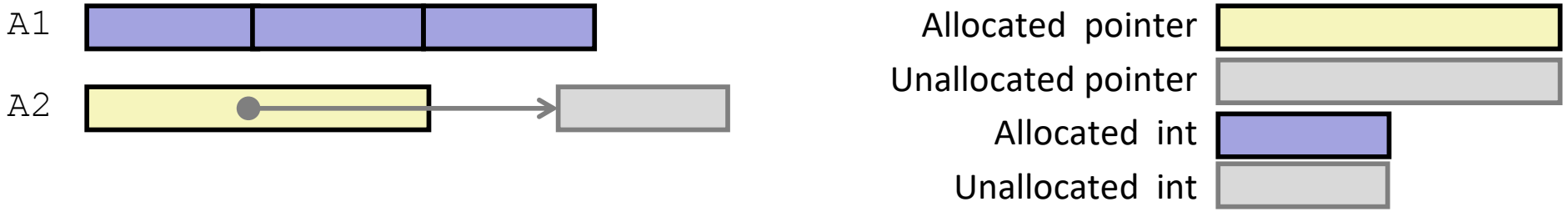
Comp: Compiles (Y/N)

Bad: Possible bad pointer reference (Y/N)

Size: Value returned by `sizeof`

Understanding Pointers & Arrays #1

Decl	A1 , A2			*A1 , *A2		
	Comp	Bad	Size	Comp	Bad	Size
`int A1[3]`	Y	N	12	Y	N	4
`int *A2`	Y	N	8	Y	Y	4



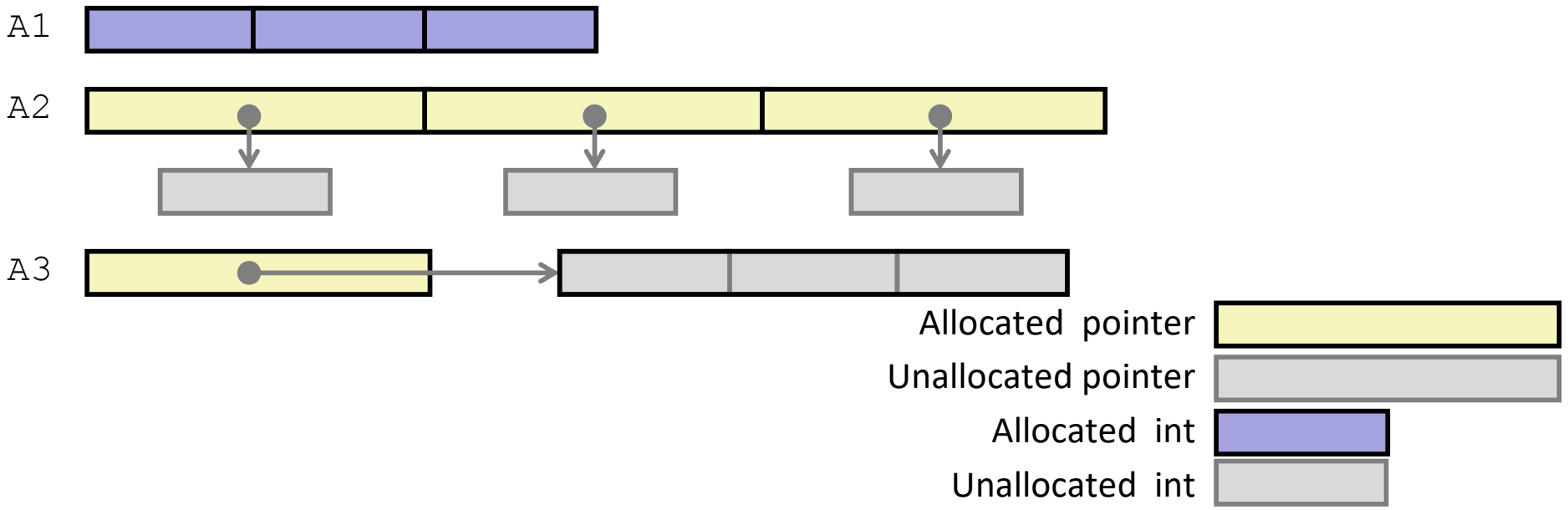
Comp: Compiles (Y/N)

Bad: Possible bad pointer reference (Y/N)

Size: Value returned by `sizeof`

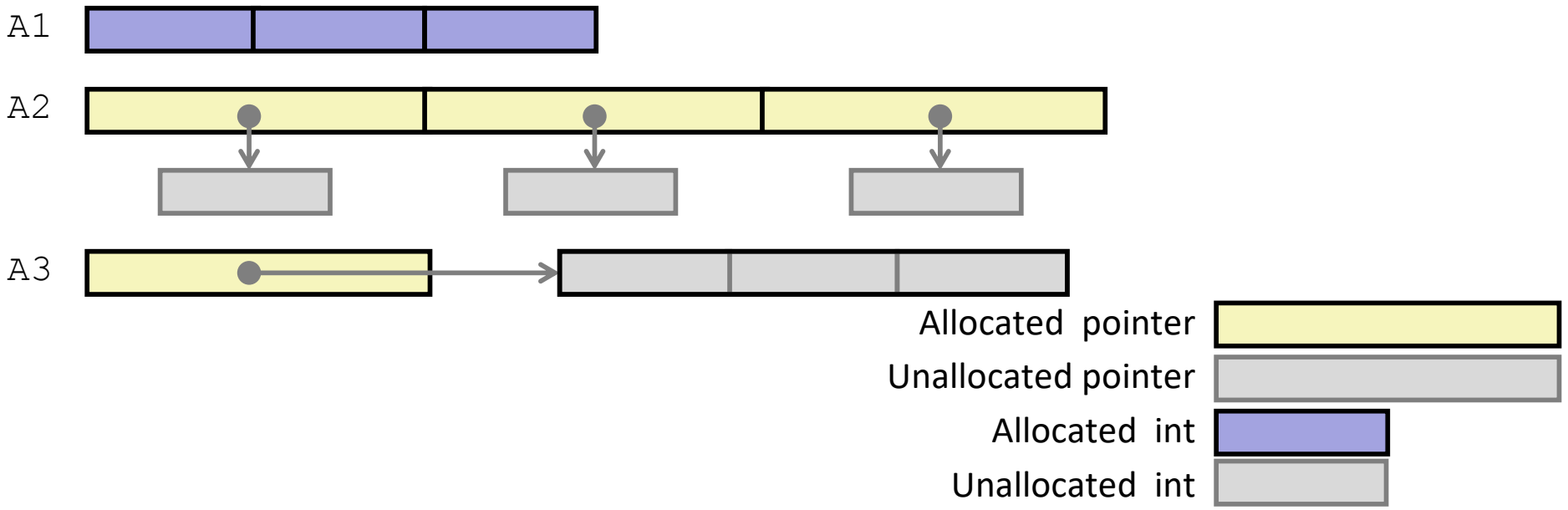
Understanding Pointers & Arrays #2

Decl	*An*			**An*			***An*		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`									
`int *A2[3]`									
`int (*A3)[3]`									



Understanding Pointers & Arrays #2

Decl	*An*			**An*			***An*		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`	Y	N	12	Y	N	4	N	–	–
`int *A2[3]`	Y	N	24	Y	N	8	Y	Y	4
`int (*A3)[3]`	Y	N	8	Y	Y	12	Y	Y	4



Multidimensional (Nested) Arrays

Declaration

T **A**[*R*][*C*];

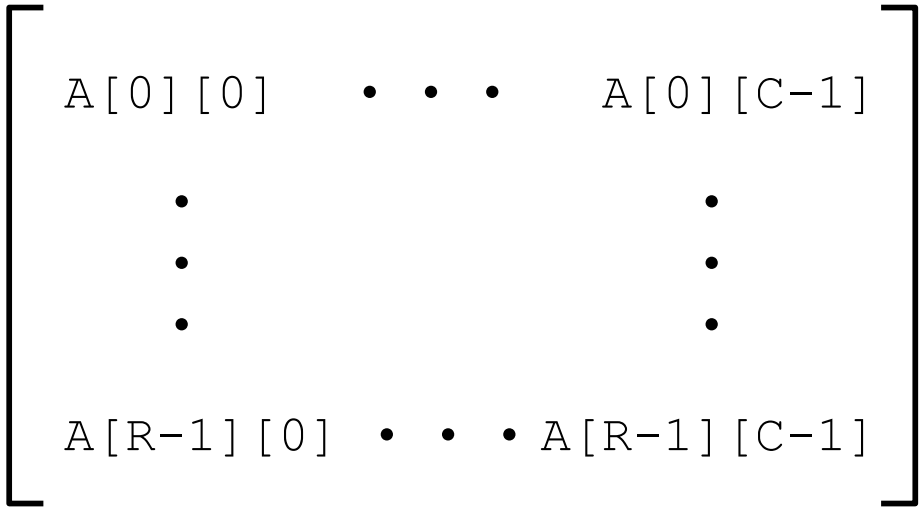
- 2D array of data type *T*
- *R* rows, *C* columns

Array Size

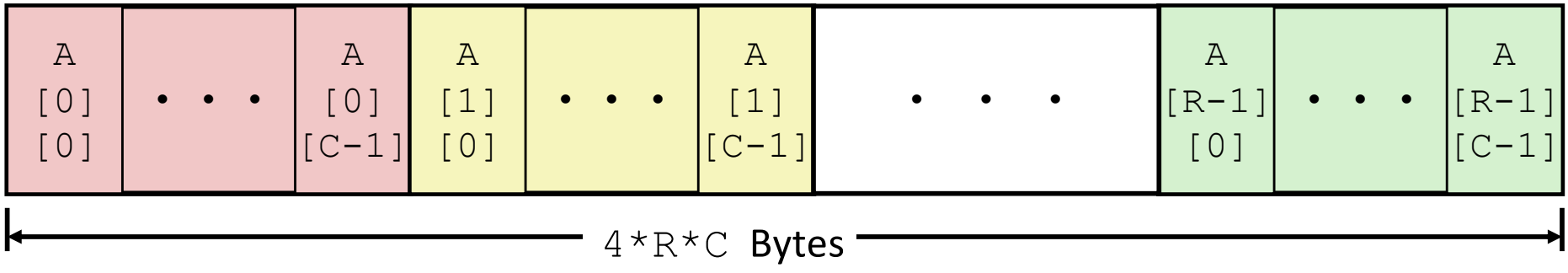
- *R* * *C* * **sizeof**(*T*) bytes

Arrangement

- Row-Major Ordering

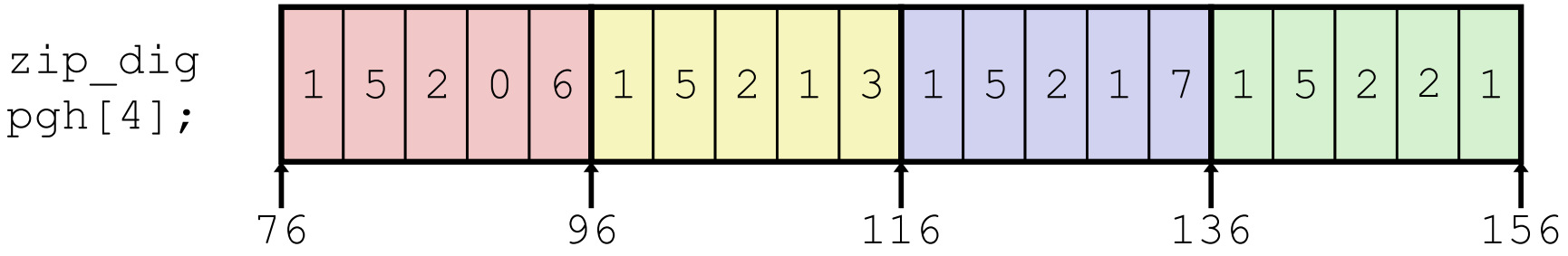


int A[R][C];



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]`" 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

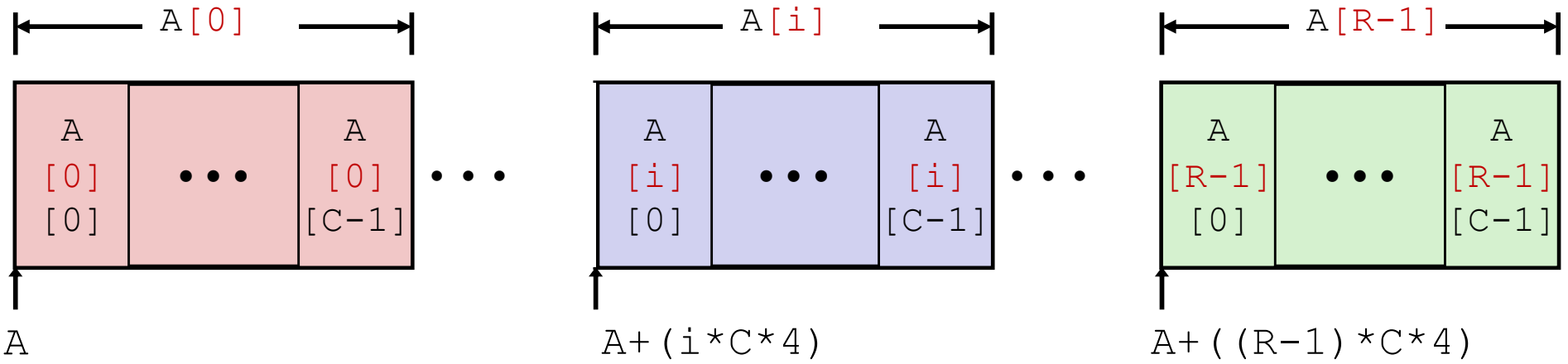
"Row-Major" ordering of all elements in memory

Nested Array Row Access

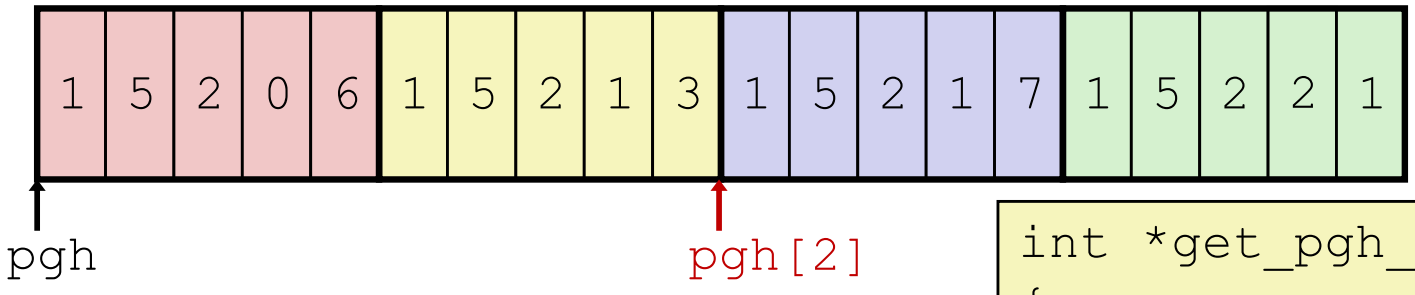
Row Vectors

- **`A[i]`** is array of *C* elements of type *T*
- Starting address **`A + i * (C * 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

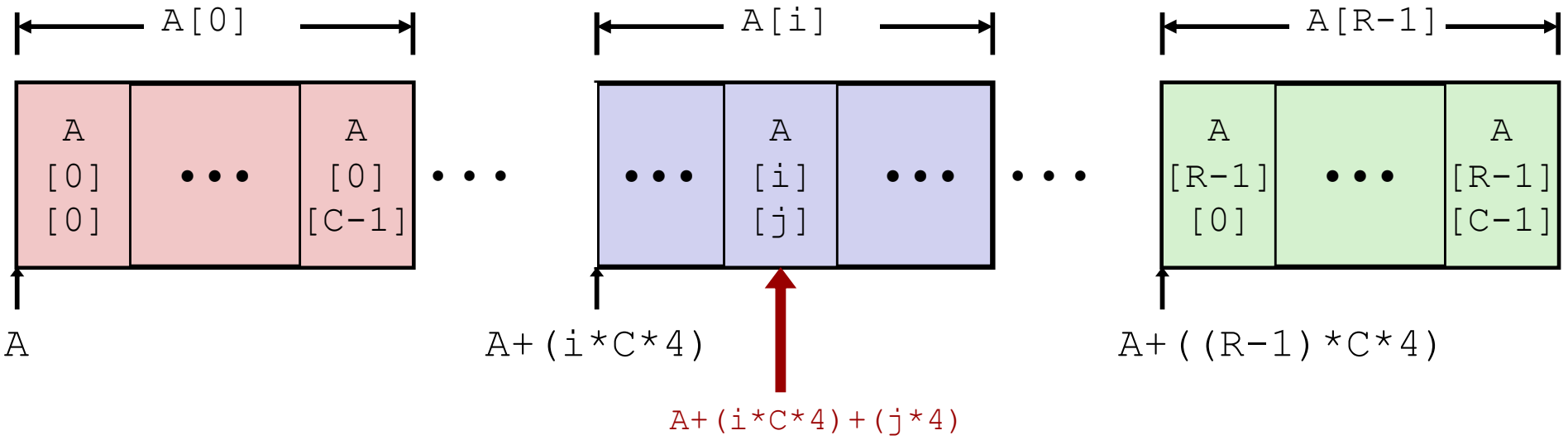
- Computes and returns address
- Compute as **`pgh + 4*(index+4*index)`**

Nested Array Element Access

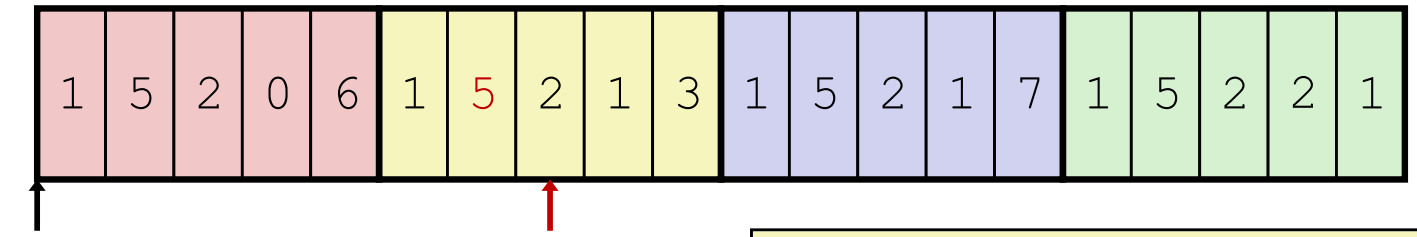
Array Elements

- **`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: **`pgh + 20*index + 4*dig`**

  `= pgh + 4*(5*index + 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 };
```

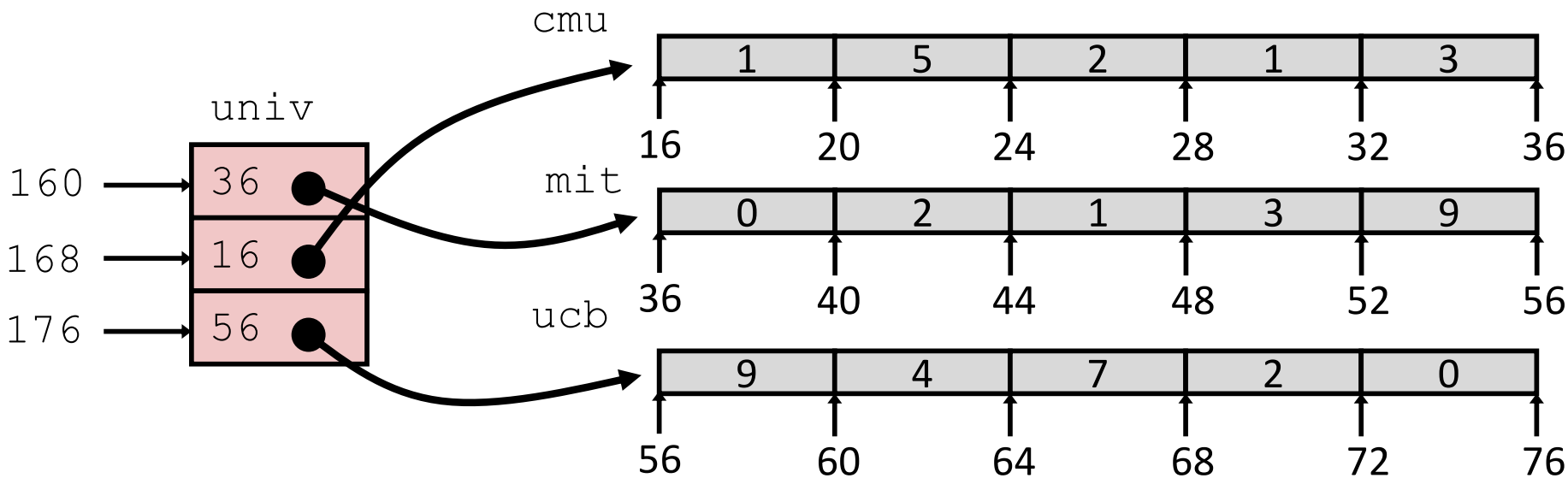
```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

Variable `univ` denotes array of 3 elements

Each element is a pointer

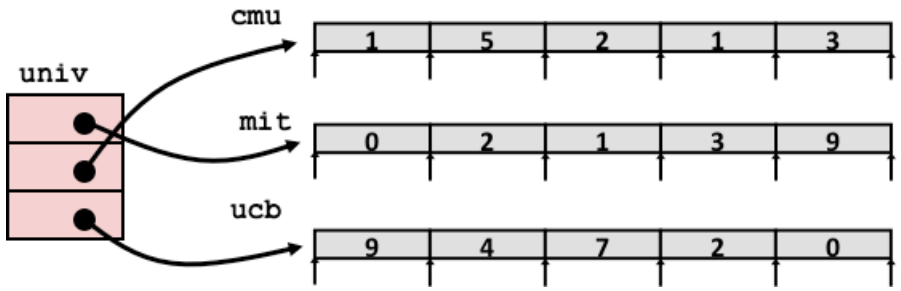
- 8 bytes

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

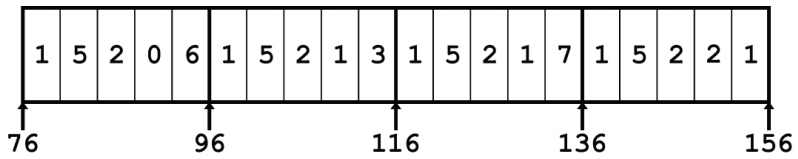
Array Element Accesses

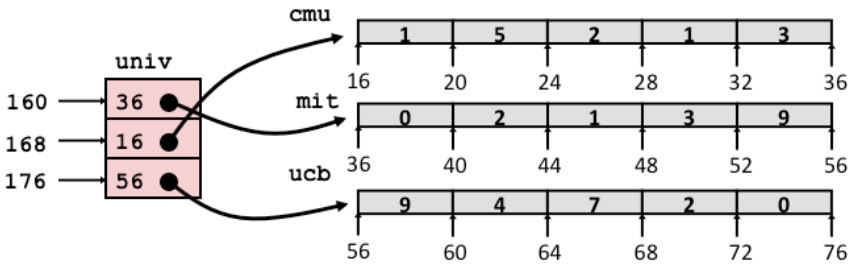
Nested array

```
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:

`Mem[pgh+20*index+4*digit]` `Mem[Mem[univ+8*index]+4*digit]`

N X *N* Matrix Code

Fixed dimensions

- Know 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, explicit indexing

- Traditional way to implement dynamic arrays

```
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element A[i][j] */
int vec_ele(size_t n, int *A,
            size_t i, size_t j)
{
  return A[IDX(n,i,j)];
}
```

Variable dimensions, implicit indexing

- Not in K&R; added to language in 1999

```
/* 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];
}
```

16 X 16 Matrix Access

- **Array Elements**
  - `int A[16][16];`
  - Address `A + i * (C * K) + j * K`
  - C = 16, K = 4

```
/* 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
```

n X *n* Matrix Access

- **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
```

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
```

Today

Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Illustration of Recursion

Arrays

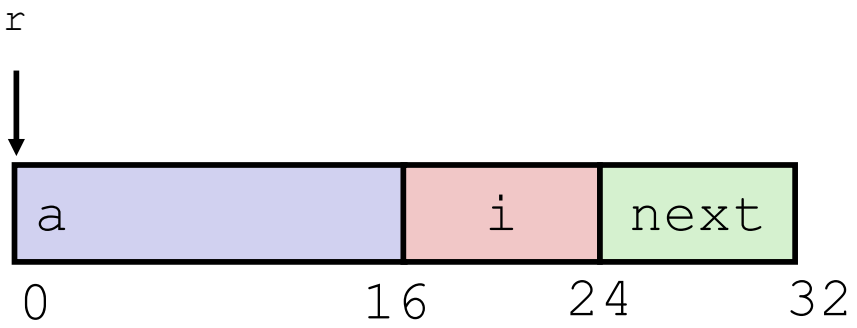
- One-dimensional
- Multi-dimensional (nested)
- Multi-level

Structures

- **Allocation**
- **Access**
- **Alignment**

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

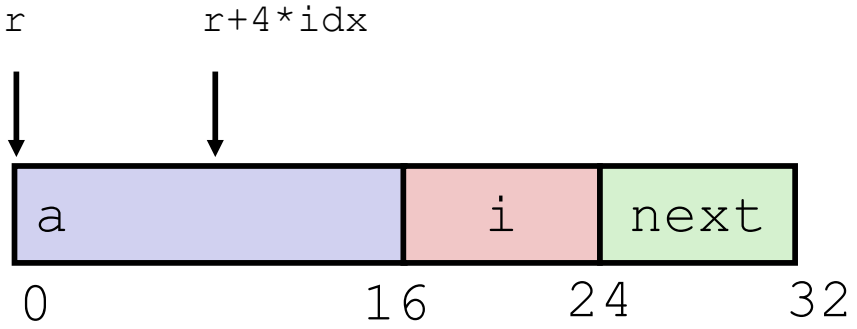
- Even if another ordering could be more compact

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];
}
```

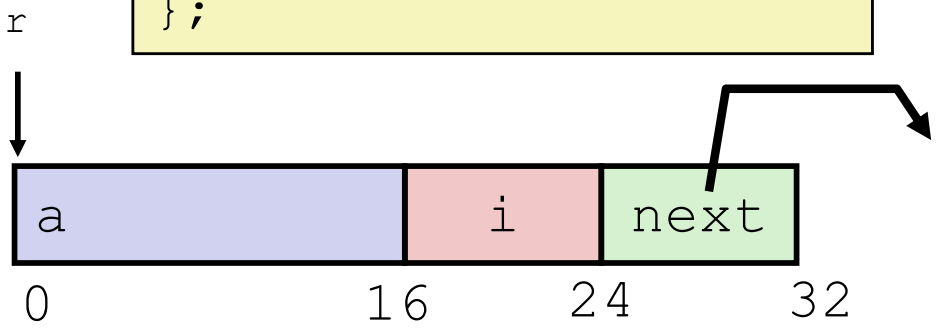
```
  # r in %rdi, idx in %rsi
  leaq  (%rdi,%rsi,4), %rax
  ret
```

Following Linked List #1

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

C Code

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



Register	Value
%rdi	r
%rax	len

Loop assembly code

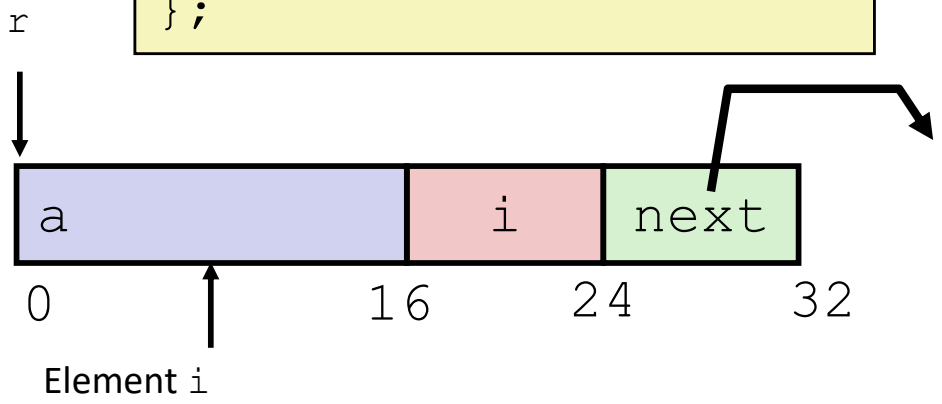
```
.L11:                           # loop:
  addq    $1, %rax              #   len ++
  movq    24(%rdi), %rdi        #   r = Mem[r+24]
  testq   %rdi, %rdi            #   Test r
  jne     .L11                  #   If != 0, goto loop
```

Following Linked List #2

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

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;
  }
}
```

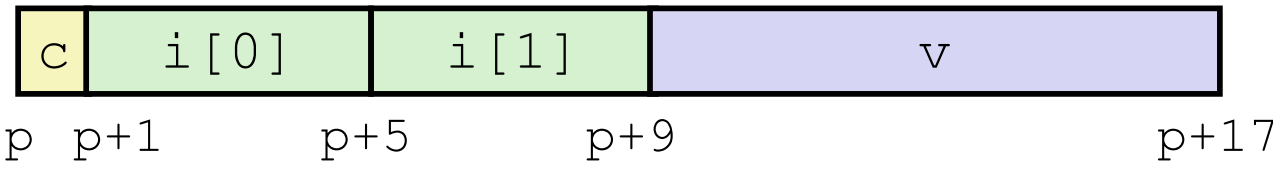


Register	Value
%rdi	r
%rsi	val

```
.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
```

Structures & Alignment

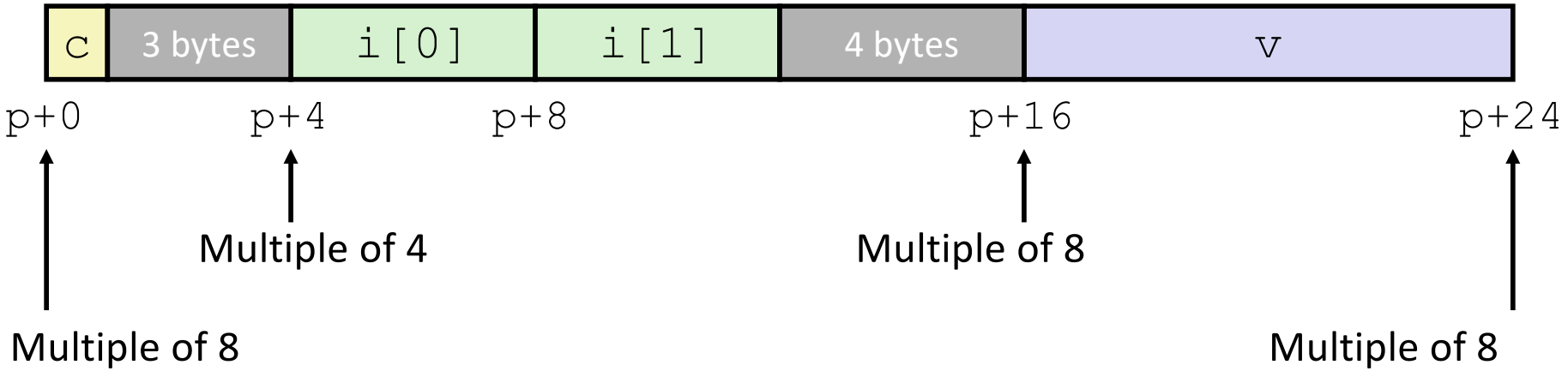
Unaligned Data




```
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



Alignment Principles

Aligned Data

- Primitive data type requires B bytes
- Address must be multiple of B
- Required on some machines; advised on x86-64

Motivation for Aligning Data

- Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
  - Inefficient to load or store datum that spans cache lines (64 bytes). Intel states should avoid crossing 16 byte boundaries.

    [Cache lines will be discussed in Lecture 10.]
  - Virtual memory trickier when datum spans 2 pages (4 KB pages)

    [Virtual memory pages will be discussed in Lecture 17.]

Compiler

- Inserts gaps in structure to ensure correct alignment of fields

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

Satisfying Alignment with Structures

Within structure:

- Must satisfy each element's alignment requirement

Overall structure placement

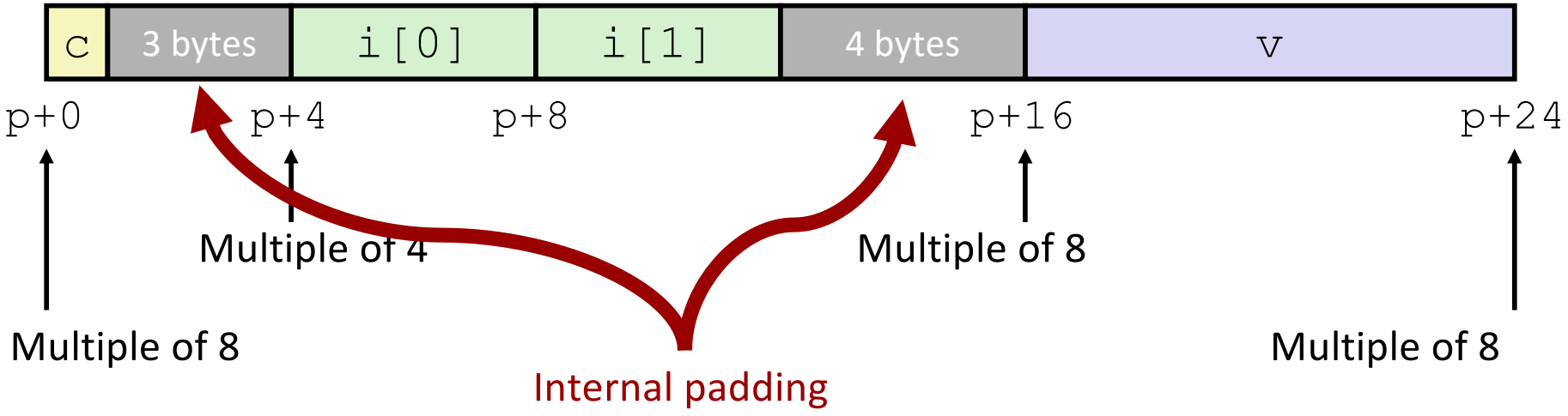
- Each structure has alignment requirement K
  - K = Largest alignment of any element
- Initial address & structure length must be multiples of K

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

Example:

- **K** = 8, due to **`double`** element

  NOTE: K < sizeof(struct S1)

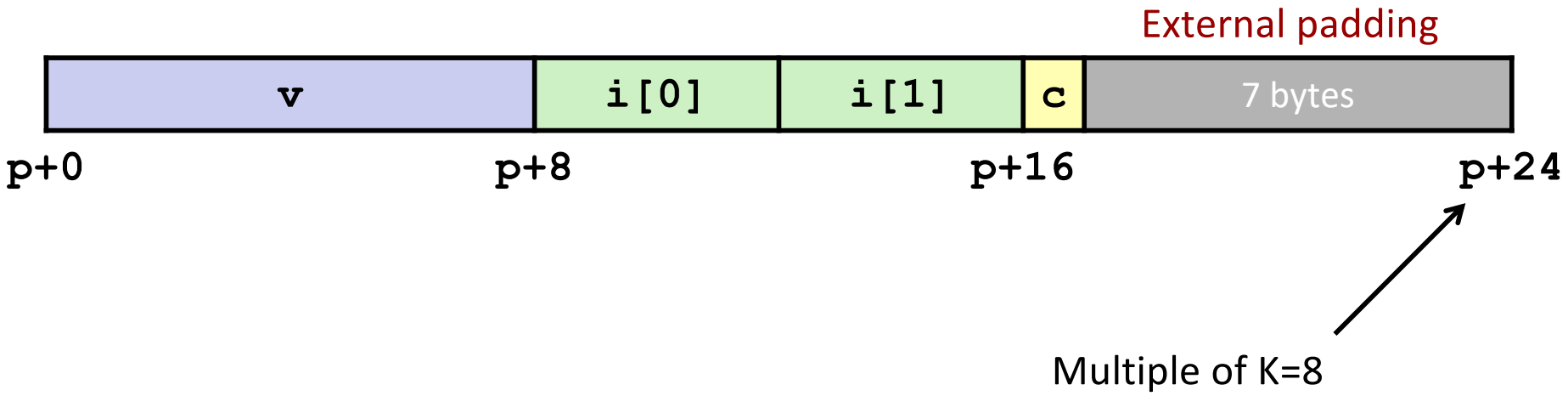


Meeting Overall Alignment Requirement

For largest alignment requirement K

Overall structure must be multiple of K

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



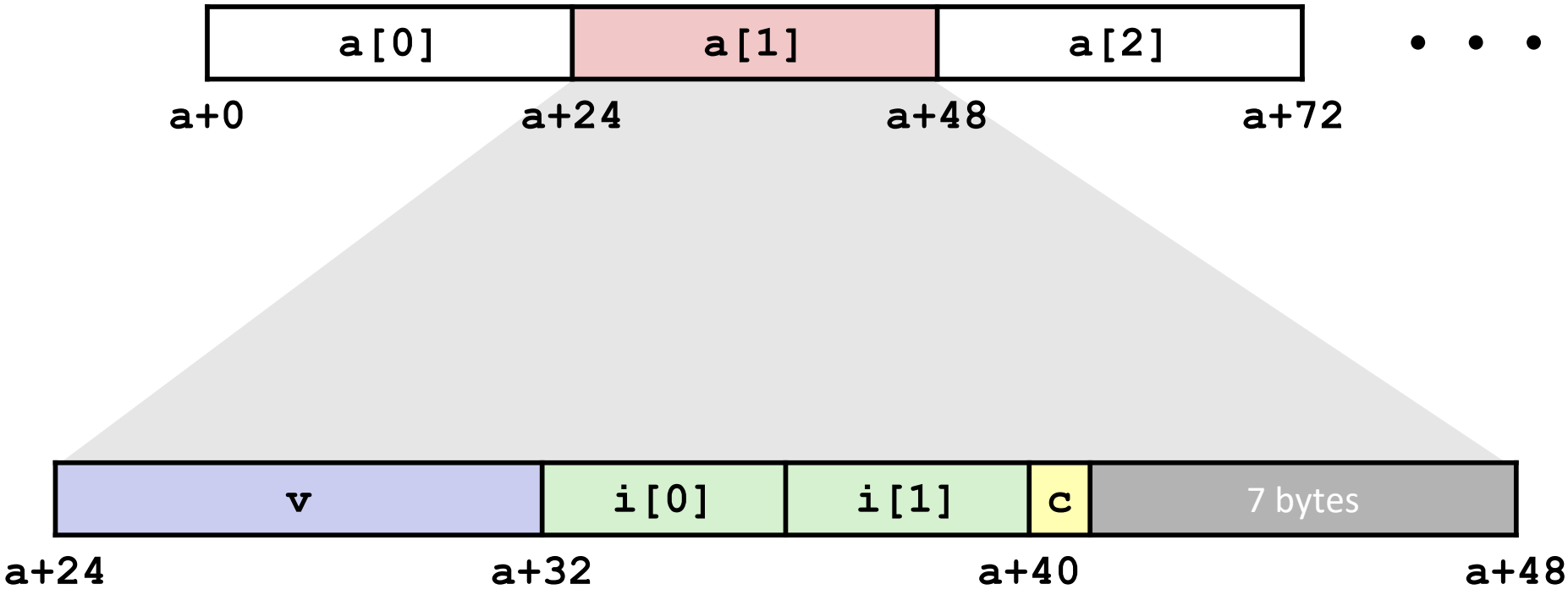
Arrays of Structures

No padding in between array elements

Overall structure length multiple of K

Satisfy alignment requirement
for every element

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Accessing Array Elements

```
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```

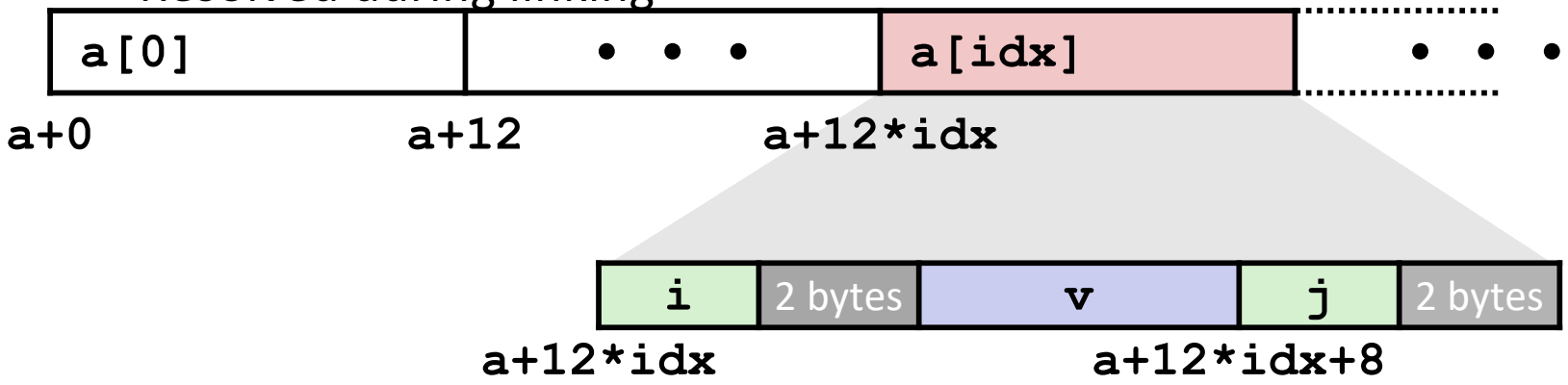
Compute array offset 12*idx

- `sizeof(S3)`, including alignment spacers

Element `j` is at offset 8 within structure

Assembler gives offset `a+8`

- Resolved during linking



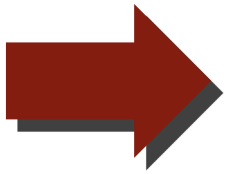

```
short get_j(int idx)
{
  return a[idx].j;
}
```

```
# %rdi = idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(,%rax,4),%eax
```

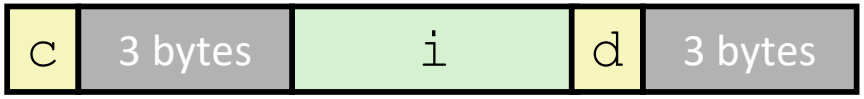
Saving Space

Put large data types first

```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

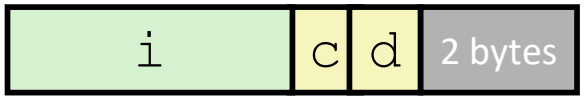


```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```



12 bytes

Effect (largest alignment requirement K=4)



8 bytes

Summary

Arrays

- Elements packed into contiguous region of memory
- Use index arithmetic to locate individual elements

Structures

- Elements packed into single region of memory
- Access using offsets determined by compiler
- Possible require internal and external padding to ensure alignment

Combinations

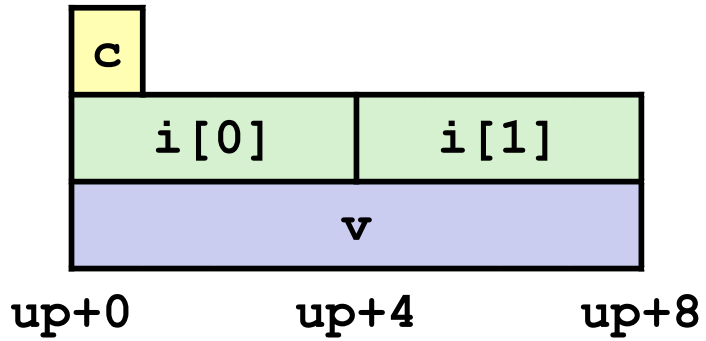
- Can nest structure and array code arbitrarily

Union Allocation

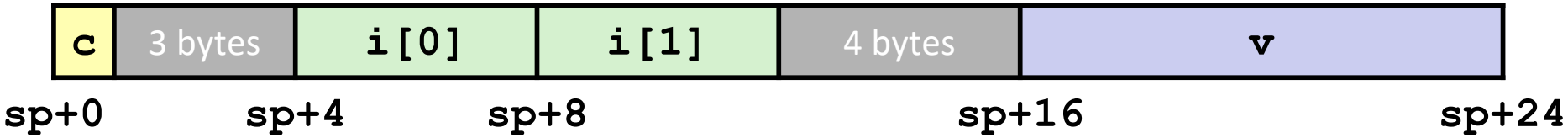
Allocate according to largest element

Can only use one field at a time

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```

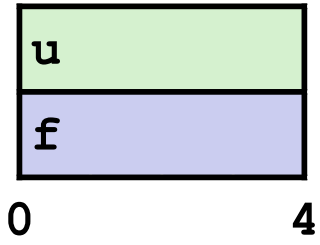



```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



Using Union to Access Bit Patterns

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```




```
float bit2float(unsigned u)
{
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

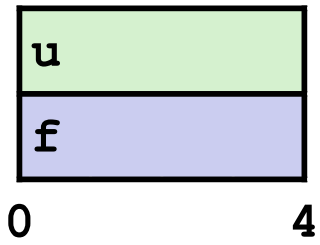
Same as **`(float) u`** ?

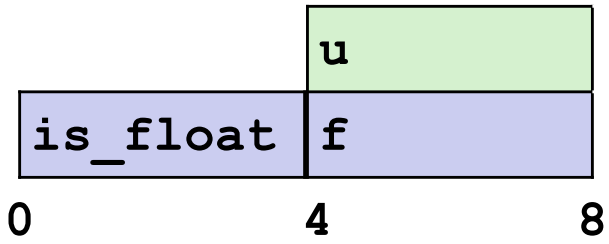
```
unsigned float2bit(float f)
{
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```

Same as **`(unsigned) f`** ?

Using Unions as Sum Types

```
typedef union {
  float f;
  unsigned u;
} num_t;

typedef struct {
  bool is_float;
  num_t val;
} value_t;
```





(technically is_float only takes 1 byte and then there's 3 bytes of padding)