Machine-Level Programming : Procedures and Data

Computer Systems
Friday, October 11, 2024

Review: Condition Codes

Single bit registers

CF Carry Flag (for unsigned) **SF** Sign Flag (for signed)

ZF Zero Flag **OF** Overflow Flag (for signed)

jX and SetX isntructions

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

SetX	Condition	Description
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~(SF^OF)&~ZF	Greater (Signed)
setge	~(SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF)|ZF	Less or Equal (Signed)
seta	~CF&~ZF	Above (unsigned)
setb	CF	Below (unsigned)

Review : Machine Level Programming

C Control

- if-then-else
- do-while
- while, for
- switch

Assembler Control

- Conditional jump
- Conditional move
- Indirect jump (via jump tables)
- Compiler generates code sequence to implement more complex control

Standard Techniques

- Loops converted to do-while or jump-to-middle form
- Large switch statements use jump tables
- Sparse switch statements may use decision trees (if-elseif-elseif-else)

Review : for loop (x86)

Example

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

Homework #4

Patch a binary

- Homework #04

  - Overview

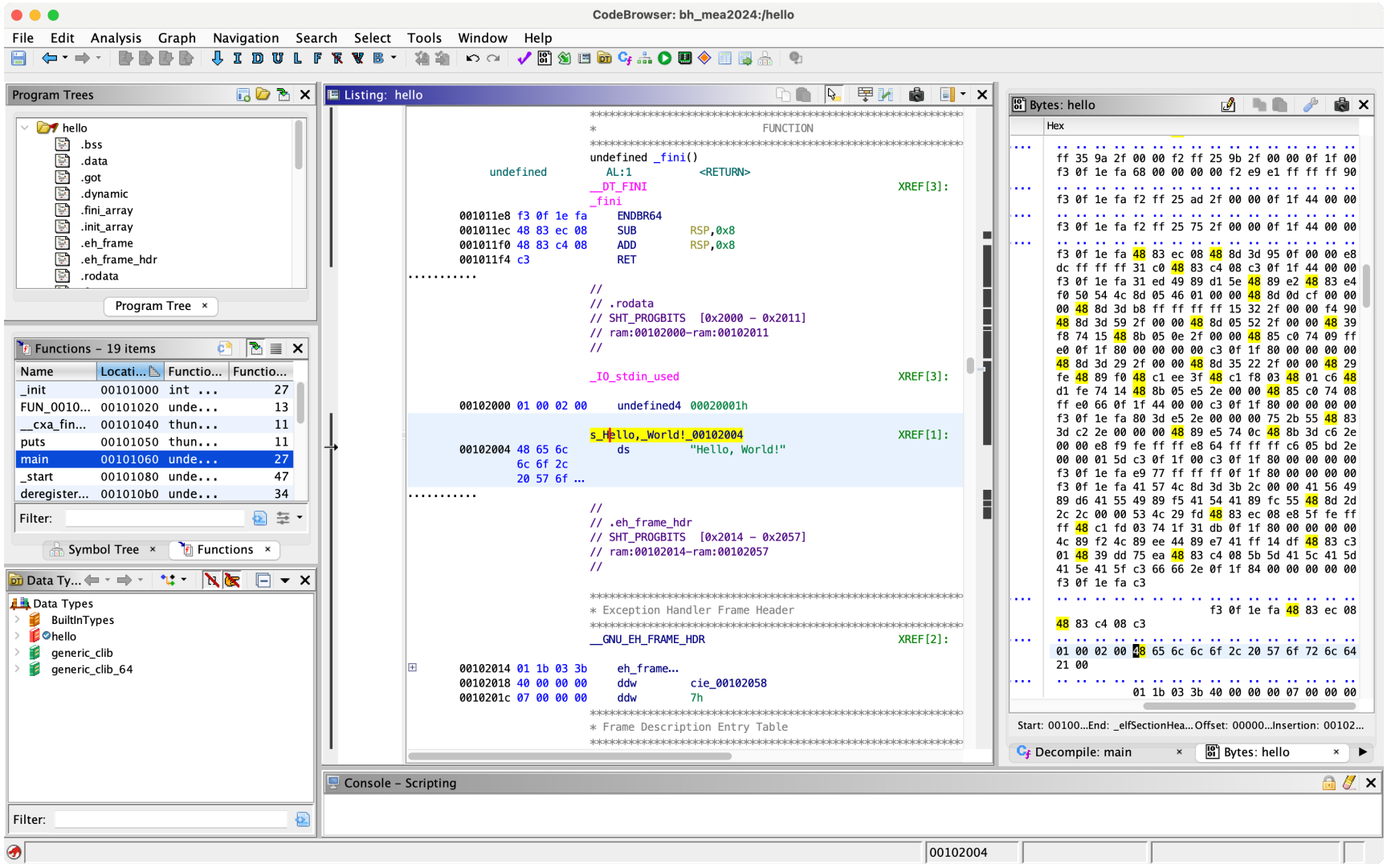
  - **Released date**: 10/4 (Fri.)
  - **Due date**: 10/11 (Fri.)
  - **Where to submit**: to e-class (http://eclass.seoultech.ac.kr)
    - Late submission is not allowed.
  - **Assigned score**: 1 points

    Decompile `hello` binary and change the string value to print your STUDENT ID

  - Submissions
    - Explain how to change the string in the bss section.
    - Captured images to show the result

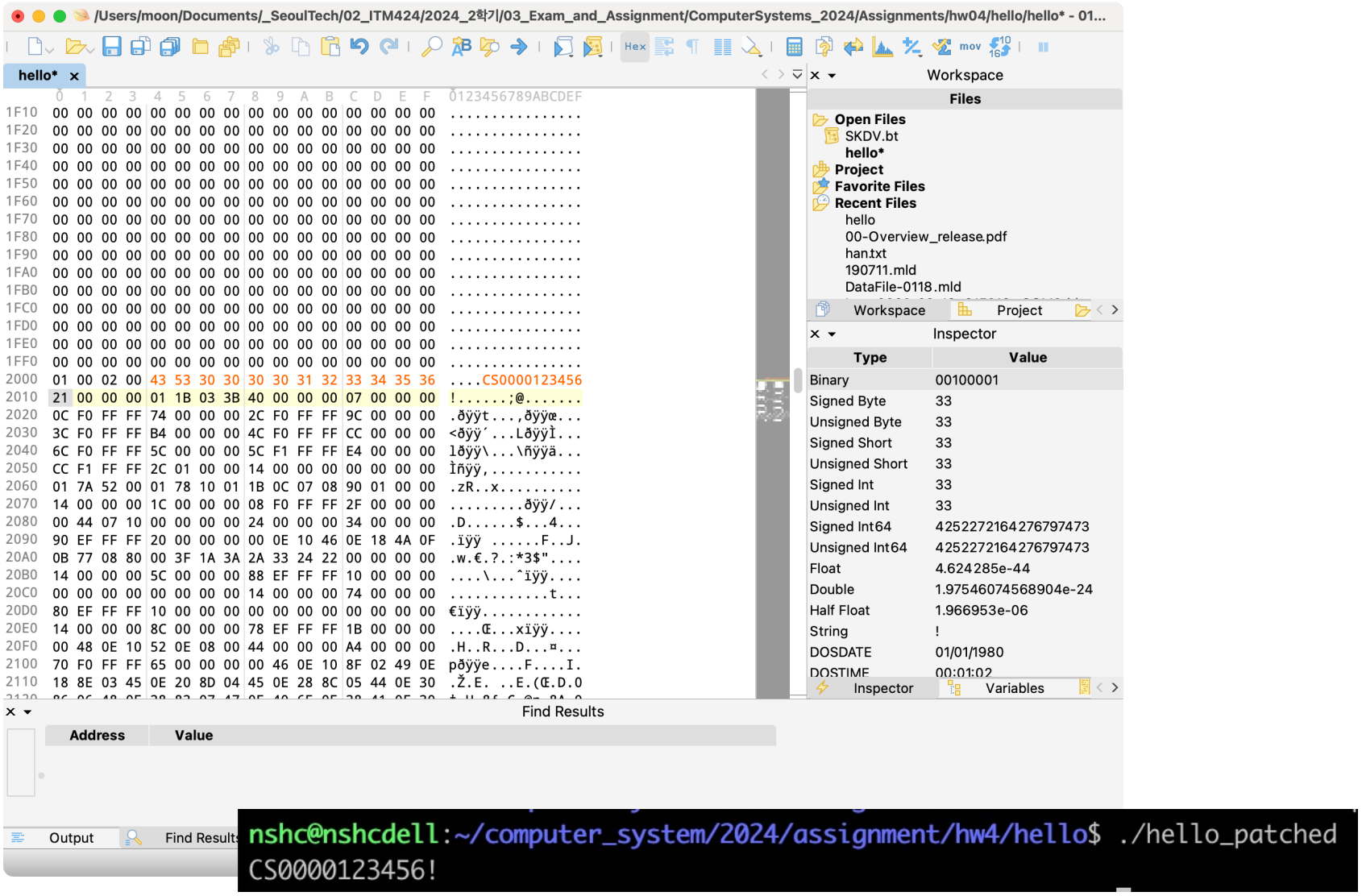
Homework #4

Patch a binary



Homework #4

Patch a binary



Today

Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data

Arrays

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

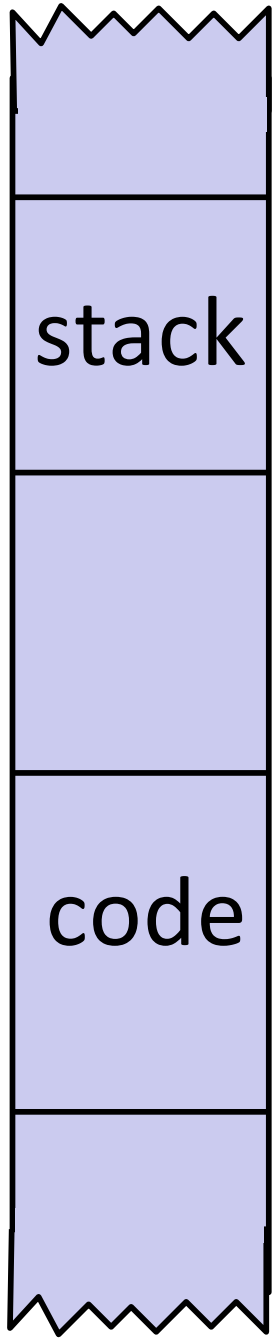
Structures

- Allocation
- Access
- Alignment

x86-64 Stack

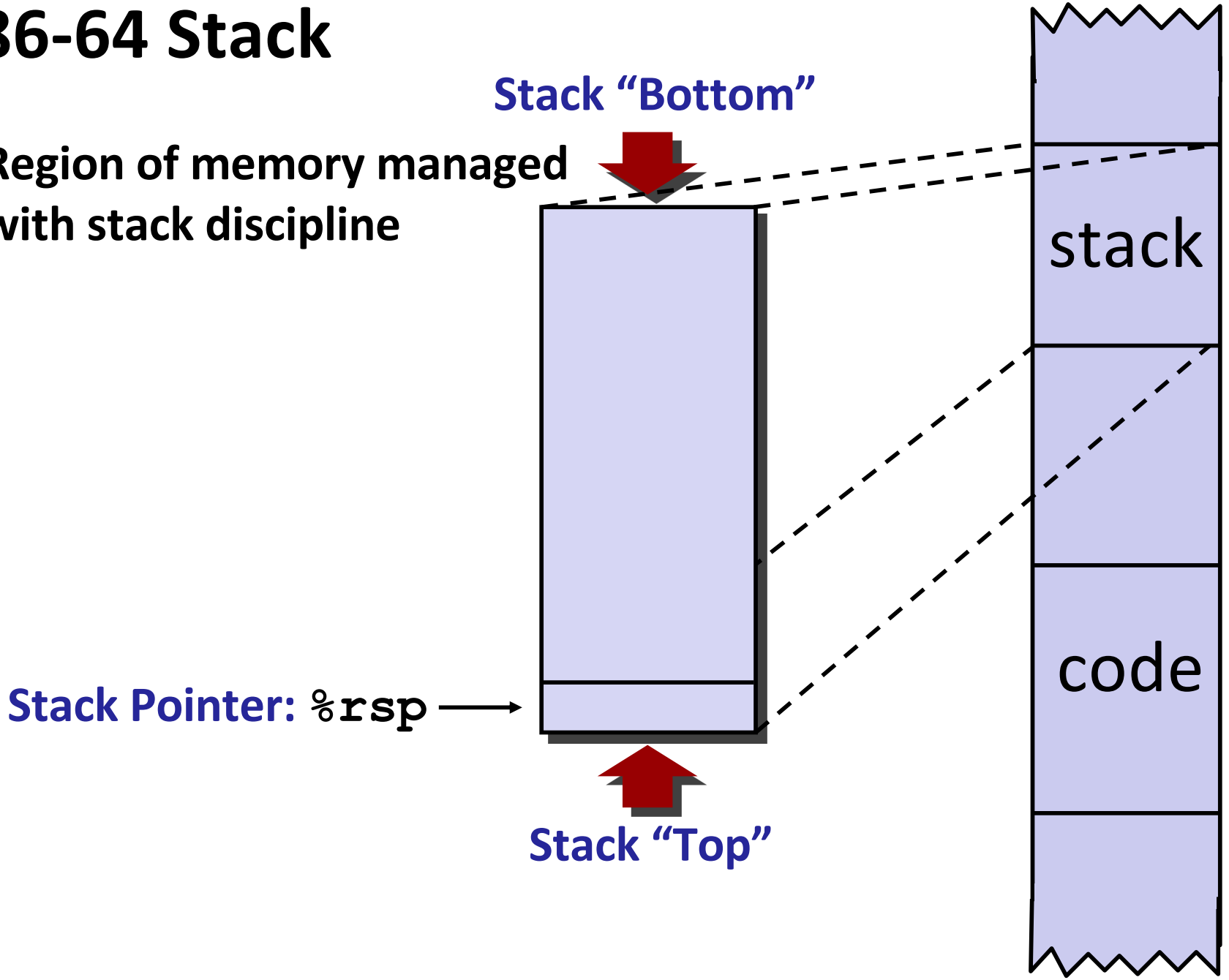
Region of memory managed with stack discipline

- 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



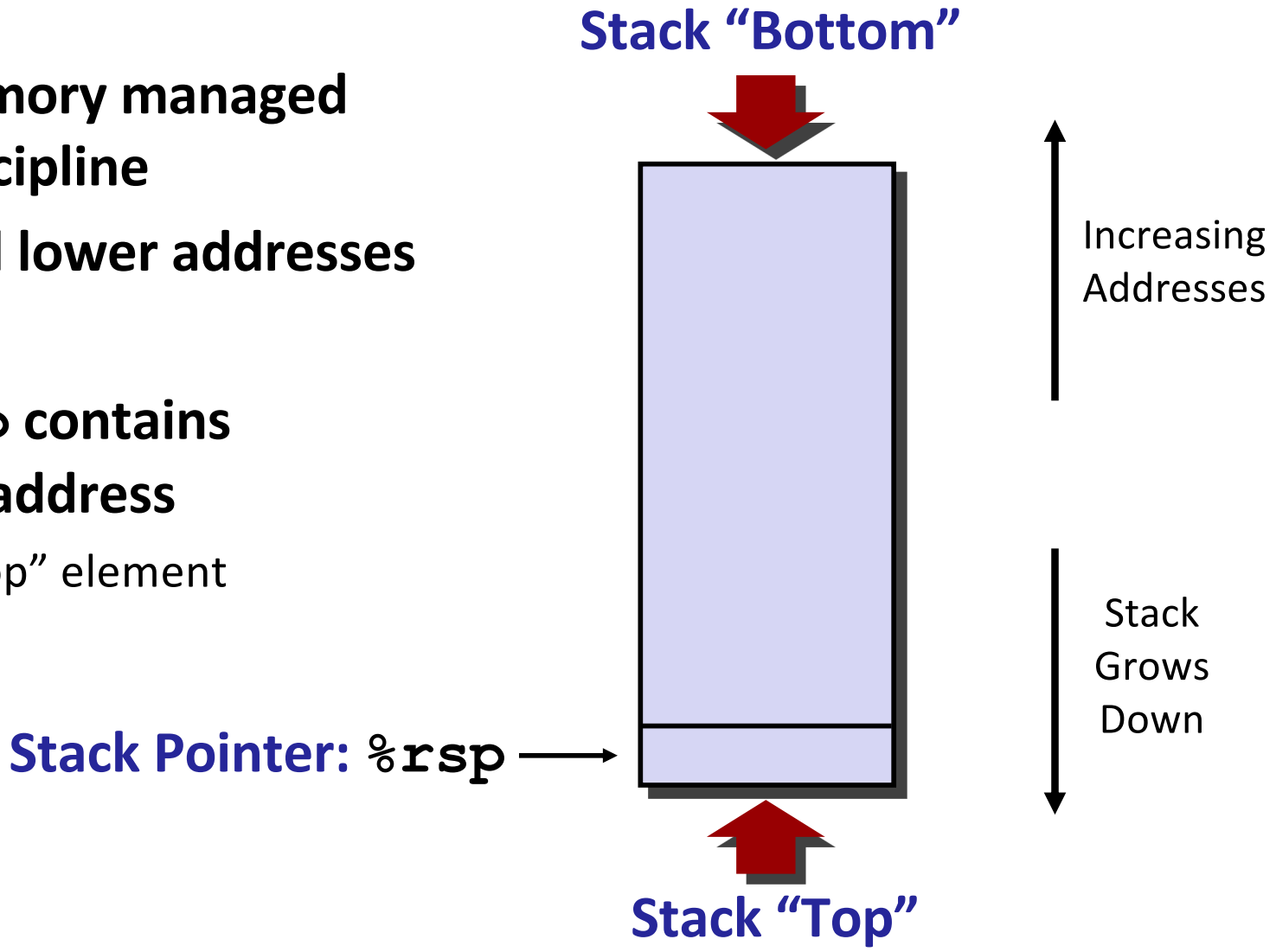
x86-64 Stack

Region of memory managed with stack discipline

Grows toward lower addresses

Register `%rsp` contains lowest stack address

address of "top" element



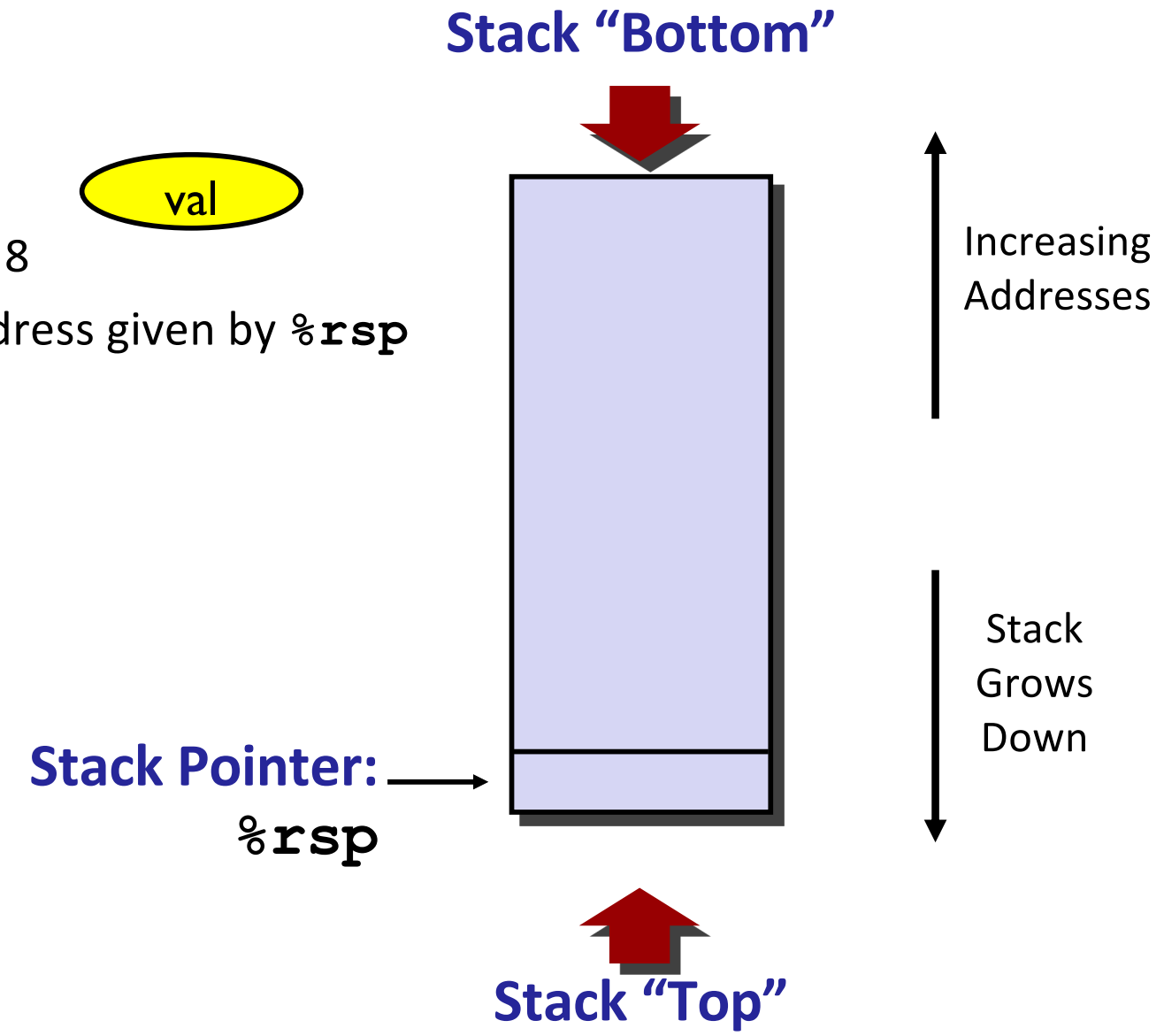
x86-64 Stack: Push

`pushq` *Src*

Fetch operand at *Src*

Decrement **`%rsp`** by 8

Write operand at address given by **`%rsp`**



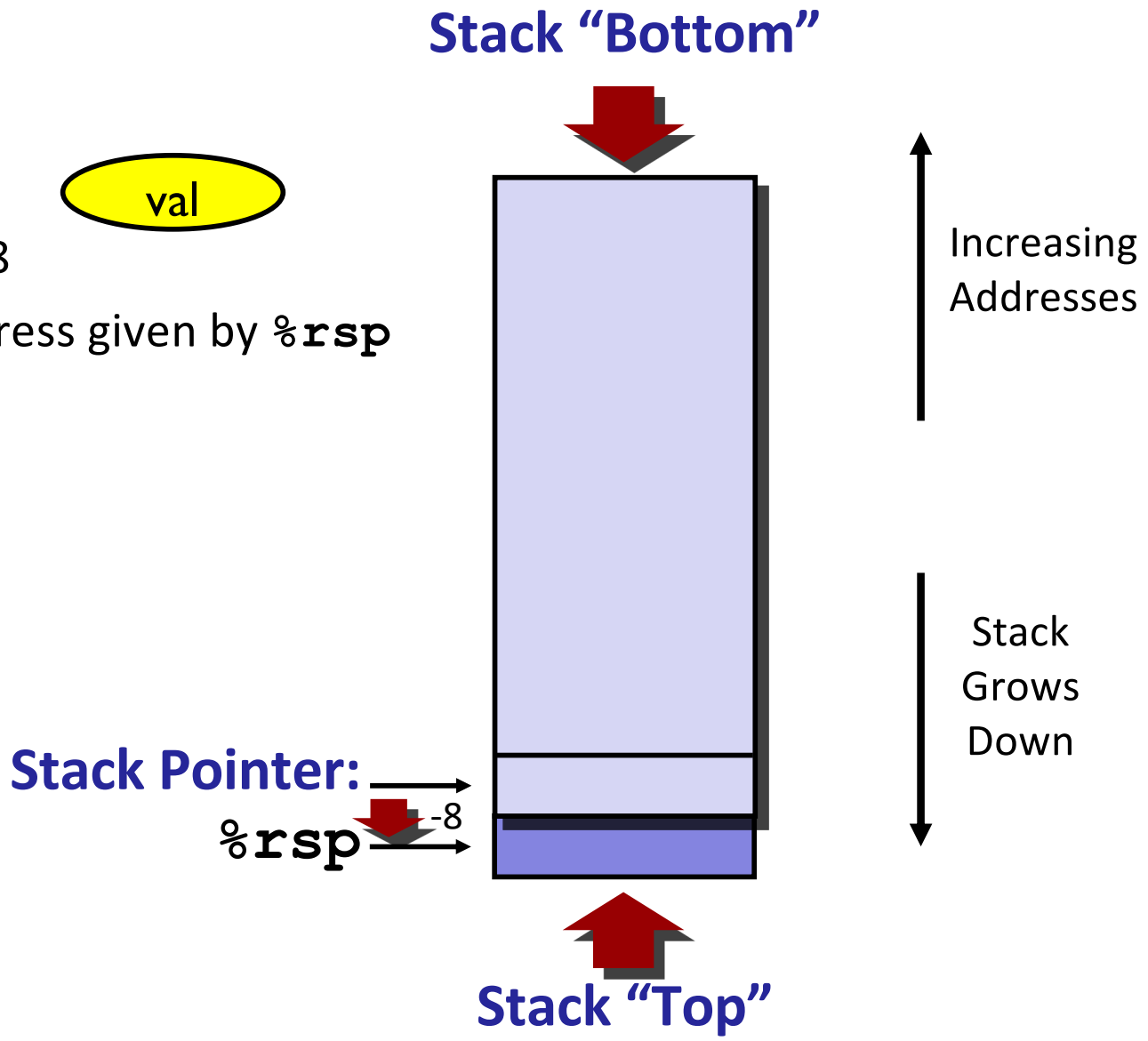
x86-64 Stack: Push

`pushq` *Src*

Fetch operand at *Src*

Decrement **`%rsp`** by 8

Write operand at address given by **`%rsp`**



x86-64 Stack: Pop

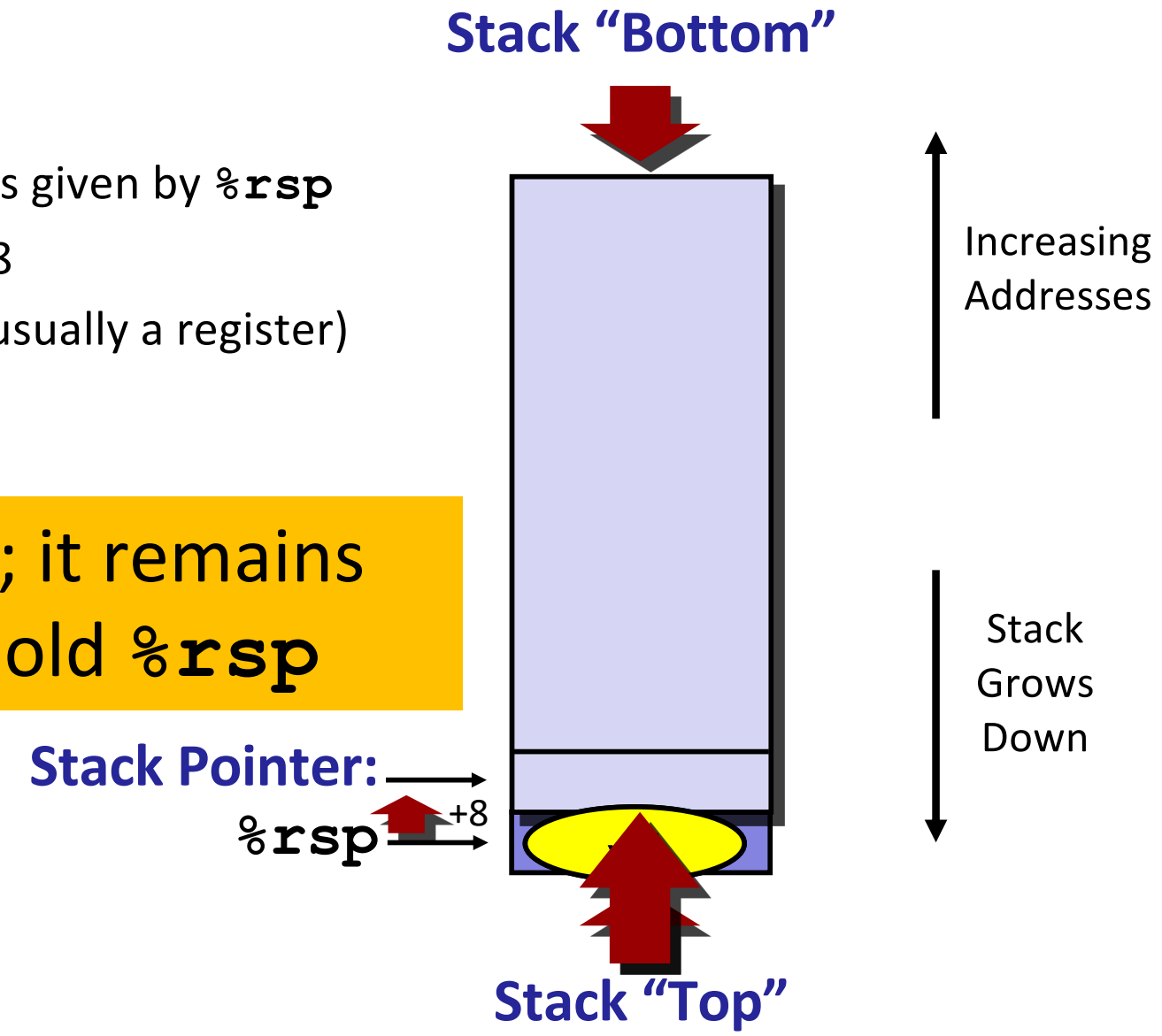
`popq` ***Dest***

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

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

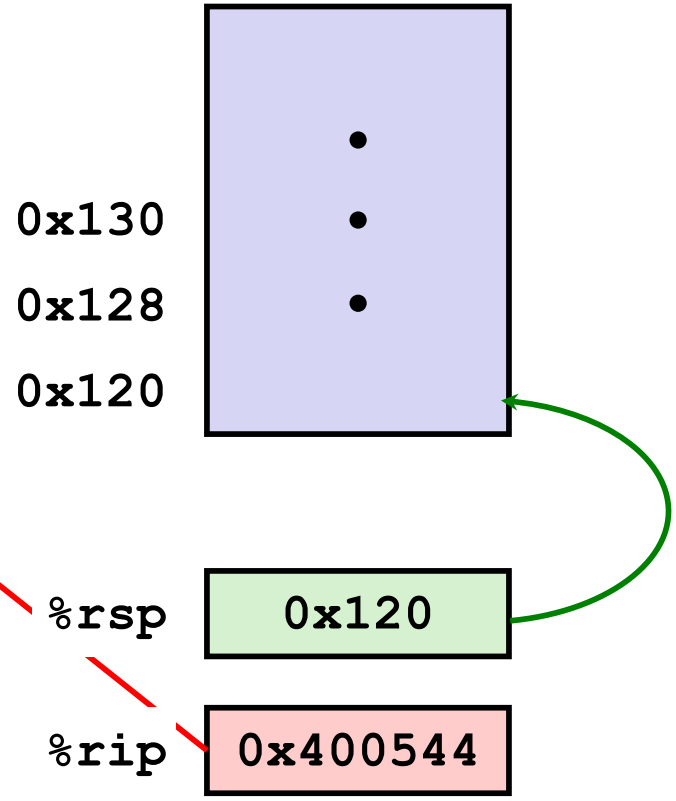
These instructions are sometimes printed with a `q` suffix

- 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)
  •
  •
```

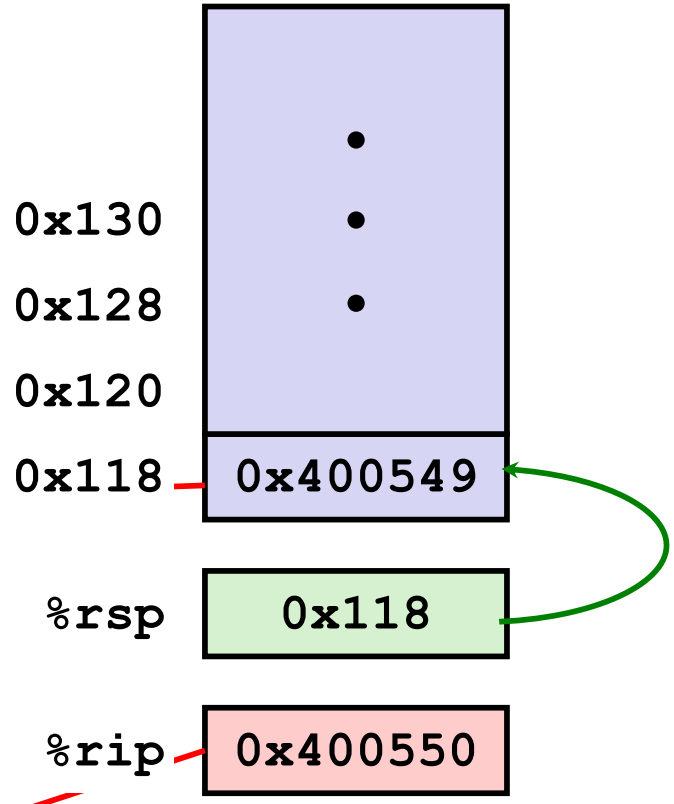
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  ret
```



Control Flow Example #2

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

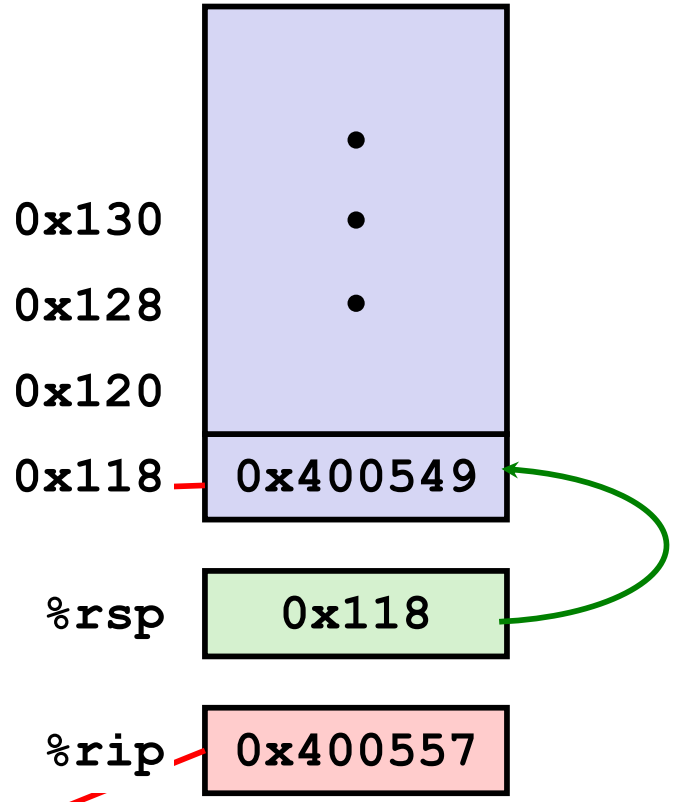
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  ret
```



Control Flow Example #3

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

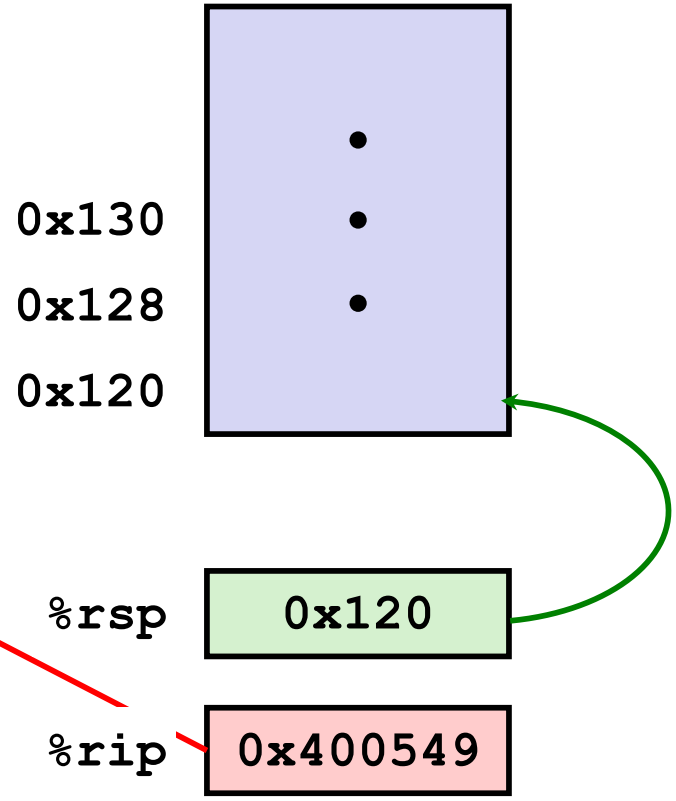
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  ret
```



Control Flow Example #4

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

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



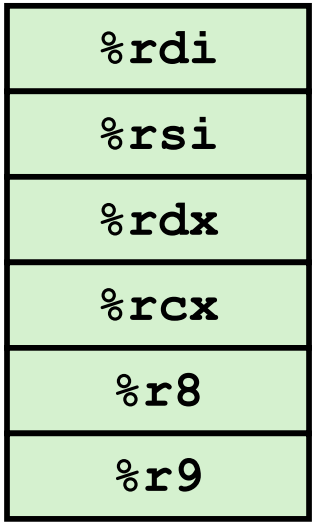
Today

- **Procedures**
  - Stack Structure
  - Calling Conventions
    - Passing control
    - **Passing data**
    - Managing local data
- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Allocation
  - Access
  - Alignment

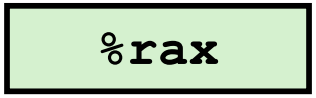
Procedure Data Flow

Registers

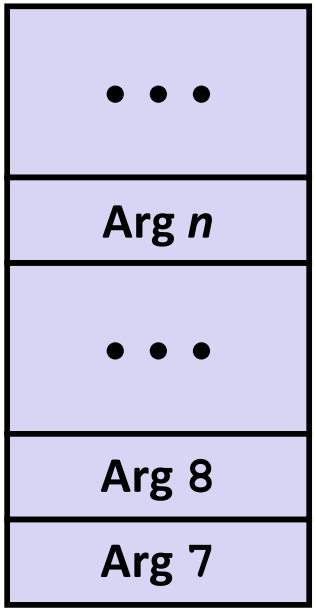
First 6 arguments



Return value



Stack



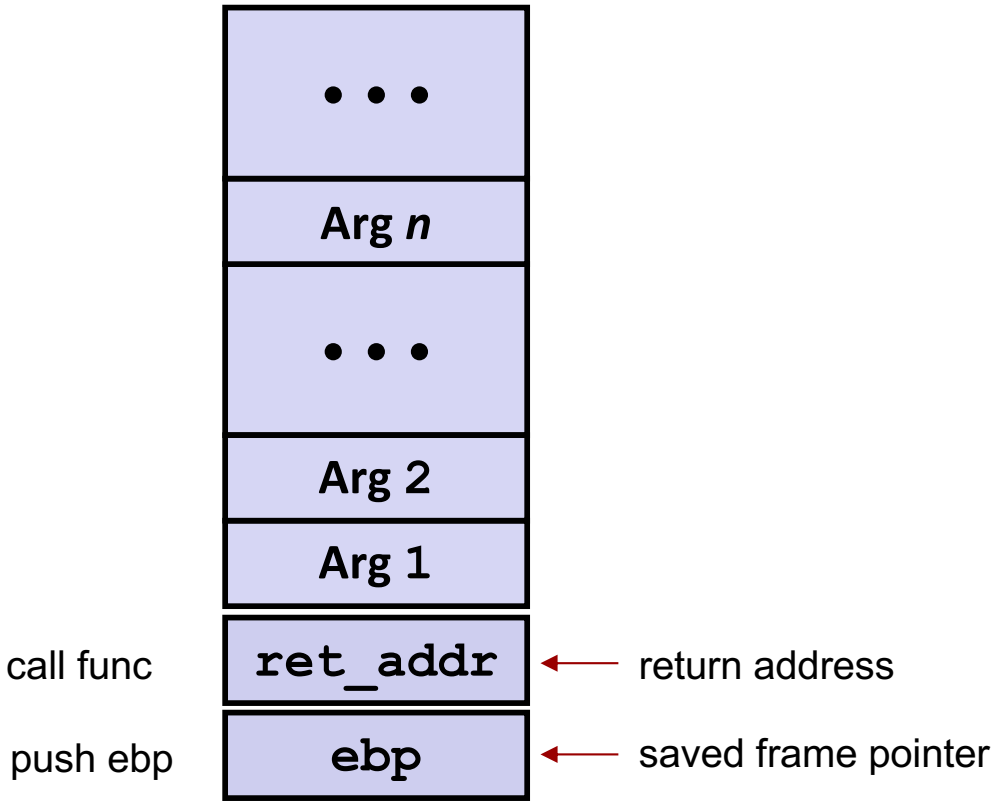
Only allocate stack space when needed

Procedure Data Flow (x86)

Registers

Return value

Stack



Procedure Data Flow (x86)

Prologue

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

Epilogue

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

or

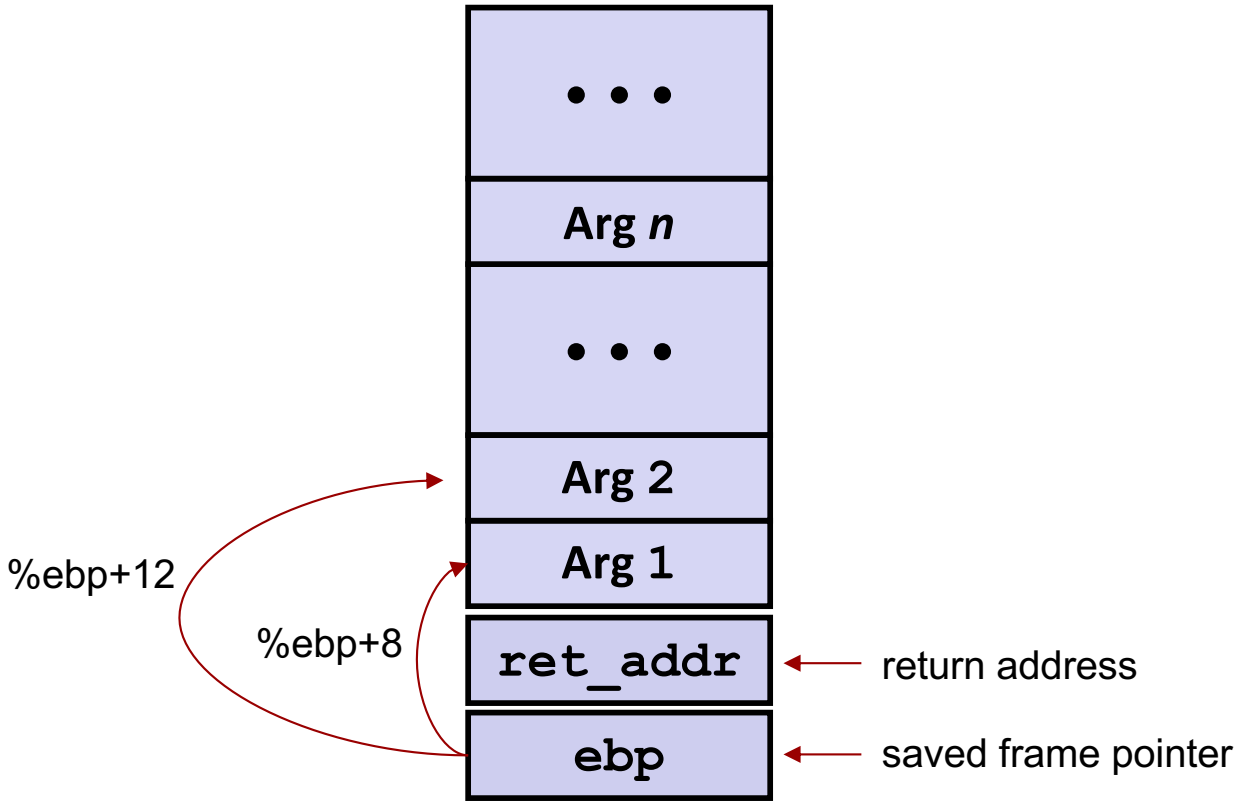
```
leave
ret
```

Procedure Data Flow (x86)

Prologue

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

Stack



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
```

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

void callfoo() {
    fo
}
```

/home/public/05_machine2

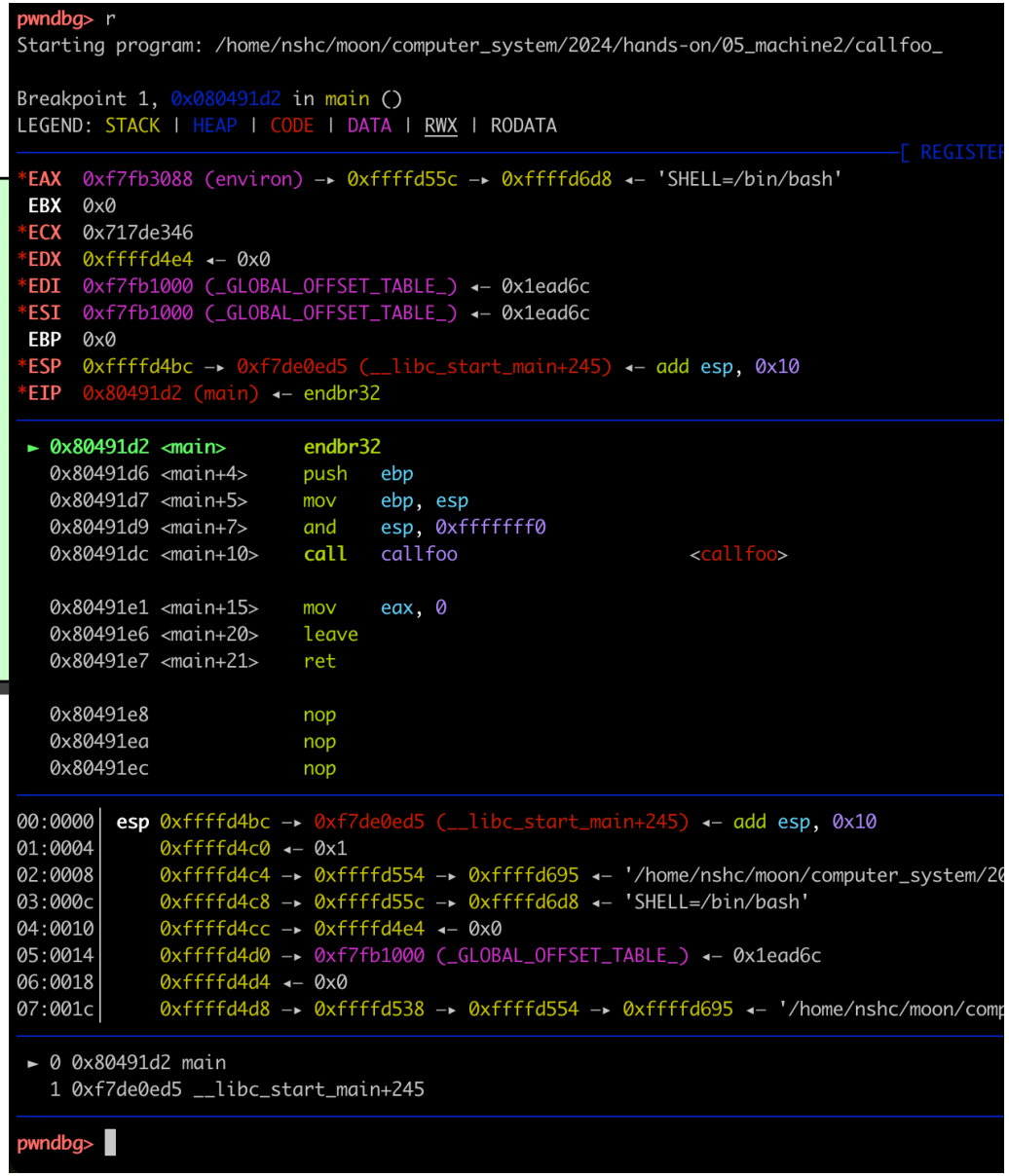
```
nshc@nshcdell:~/computer_system/2024/hands-on/05_machine2$ gdb ./callfoo_ -q
pwndbg: loaded 147 pwndbg commands and 47 shell commands. Type pwndbg [--shell | --all] [filter] for a list.
pwndbg: created $rebase, $ida GDB functions (can be used with print/break)
Reading symbols from ./callfoo_...
(No debugging symbols found in ./callfoo_)
------- tip of the day (disable with set show-tips off) -------
Use Pwndbg's config and theme commands to tune its configuration and theme colors!
pwndbg> b main
Breakpoint 1 at 0x80491d2
pwndbg>
```

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");
}
```

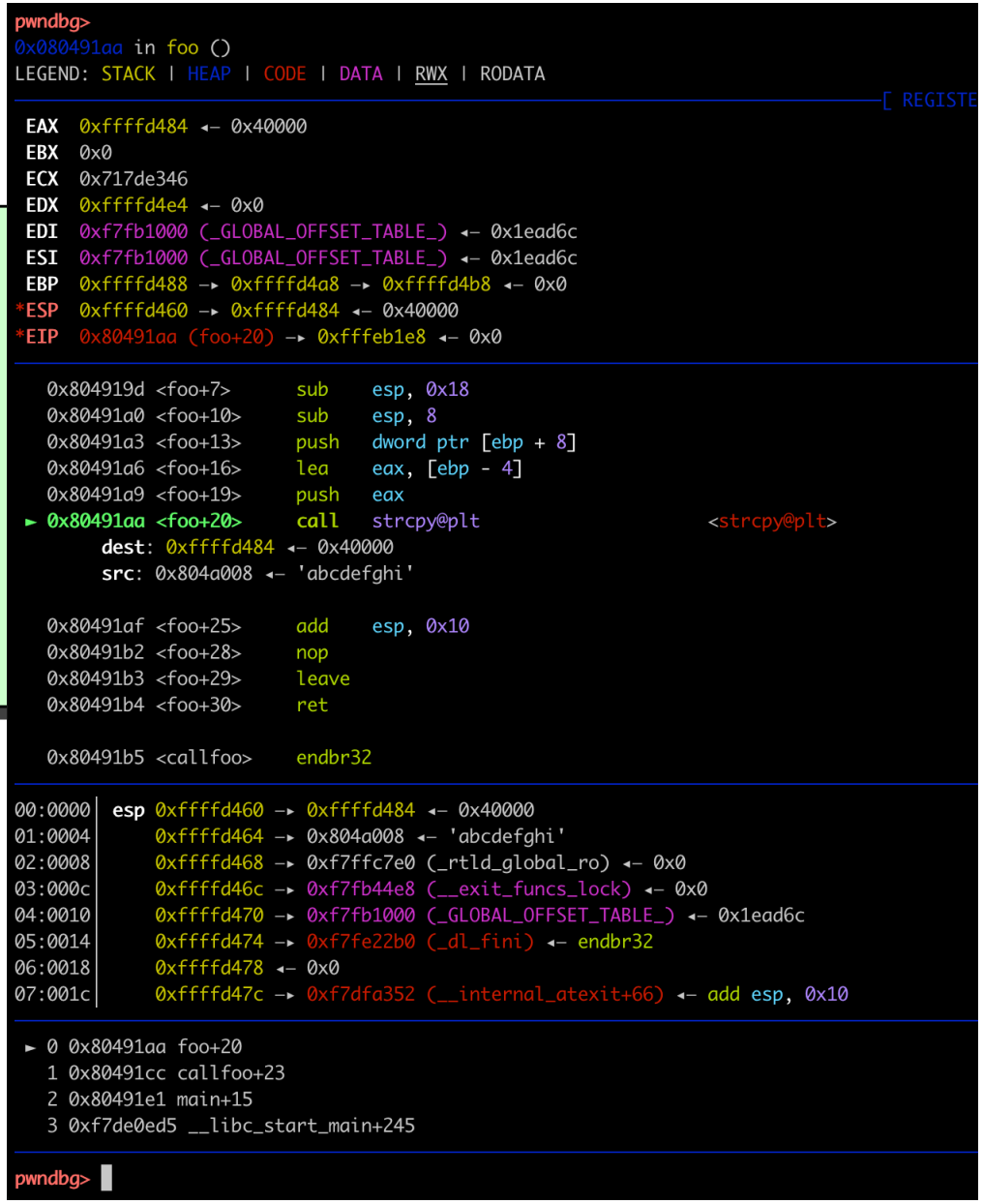


Procedure Data Flow (x86)

Example

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

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



Procedure Data Flow

Example

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

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

```
pwndbg> x/4xw 0xffffd484
0xffffd484:     0x00040000      0xffffd4a8      0x080491cc      0x0804a008
pwndbg> ni
0x080491af in foo ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[
 EAX  0xffffd484 ◂— 'abcdefghi'
 EBX  0x0
*ECX  0x804a008 ◂— 'abcdefghi'
*EDX  0xffffd484 ◂— 'abcdefghi'
 EDI  0xf7fb1000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1ead6c
 ESI  0xf7fb1000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1ead6c
 EBP  0xffffd488 ◂— 'efghi'
 ESP  0xffffd460 —▸ 0xffffd484 ◂— 'abcdefghi'
*EIP  0x80491af (foo+25) ◂— add esp, 0x10

   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

00:0000| esp 0xffffd460 —▸ 0xffffd484 ◂— 'abcdefghi'
01:0004|     0xffffd464 —▸ 0x804a008 ◂— 'abcdefghi'
02:0008|     0xffffd468 —▸ 0xf7ffc7e0 (_rtld_global_ro) ◂— 0x0
03:000c|     0xffffd46c —▸ 0xf7fb44e8 (__exit_funcs_lock) ◂— 0x0
04:0010|     0xffffd470 —▸ 0xf7fb1000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1ead6c
05:0014|     0xffffd474 —▸ 0xf7fe22b0 (_dl_fini) ◂— endbr32
06:0018|     0xffffd478 ◂— 0x0
07:001c|     0xffffd47c —▸ 0xf7dfa352 (__internal_atexit+66) ◂— add esp, 0x10

 ► 0 0x80491af foo+25
   1 0x8040069
   2 0x804a008

pwndbg> x/4xw 0xffffd484
0xffffd484:     0x64636261      0x68676665      0x08040069      0x0804a008
pwndbg>
```

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");
}
```

```
buf[0] = 0x________________
buf[1] = 0x________________
buf[2] = 0x________________
```

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");
}
```

```
buf[0] = 0x64636261
buf[1] = 0x68676665  saved ebp
buf[2] = 0x08040069  return address
```

Today

- **Procedures**
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - **Managing local data**
- 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

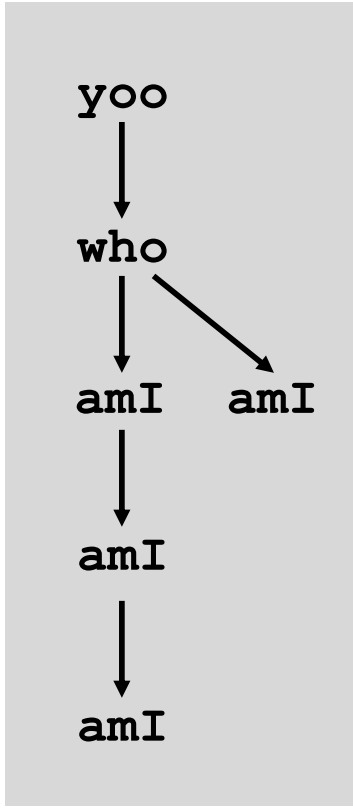
Call Chain Example

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

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

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

Example Call Chain



Procedure amI() is recursive

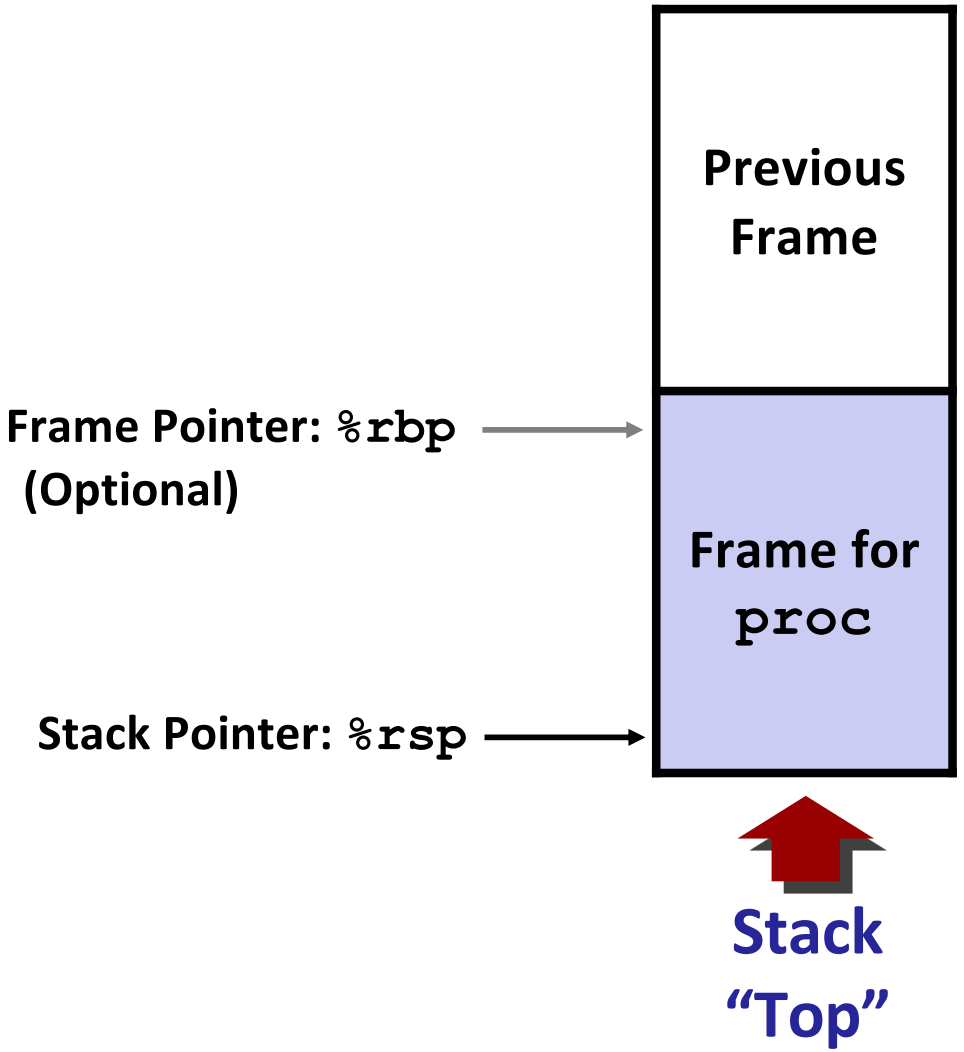
Stack Frames

Contents

Return information

Local storage (if needed)

Temporary space (if needed)



Management

Space allocated when enter procedure

"Set-up" code

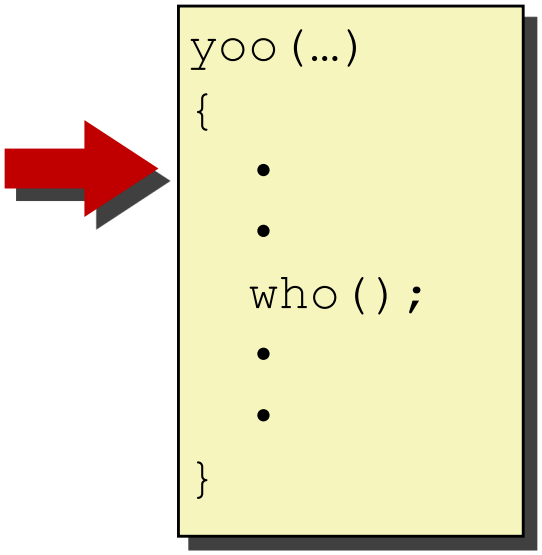
Includes push by **`call`** instruction

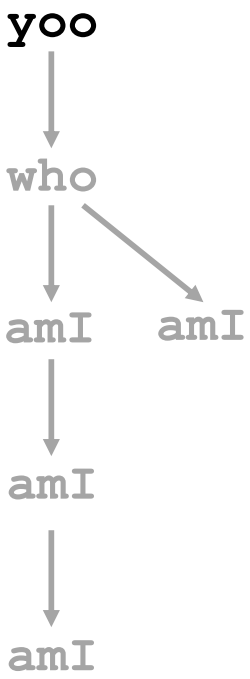
Deallocated when return

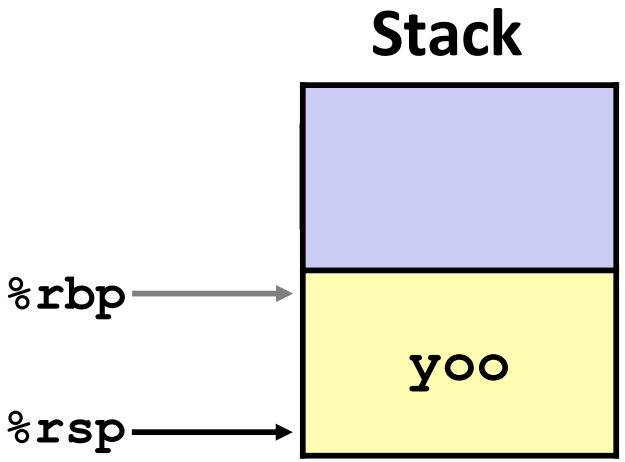
"Finish" code

Includes pop by **`ret`** instruction

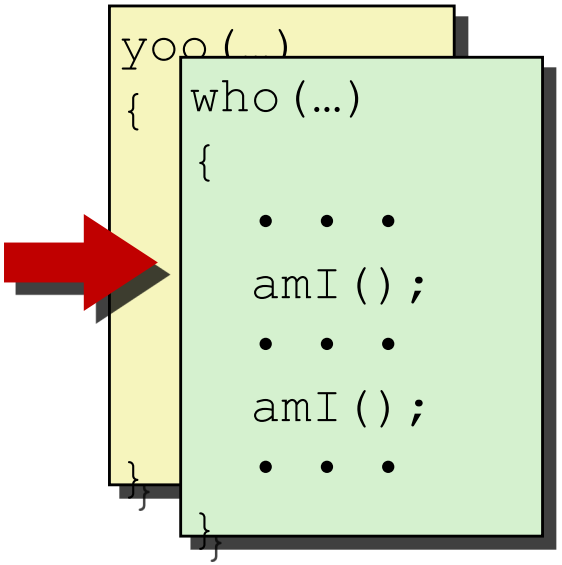
Example

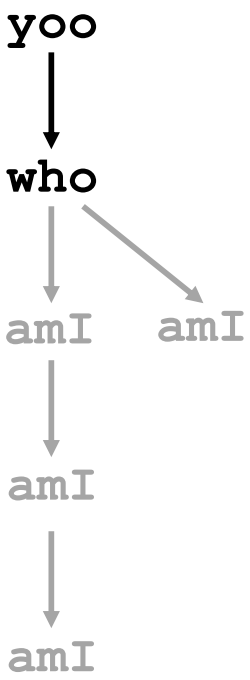


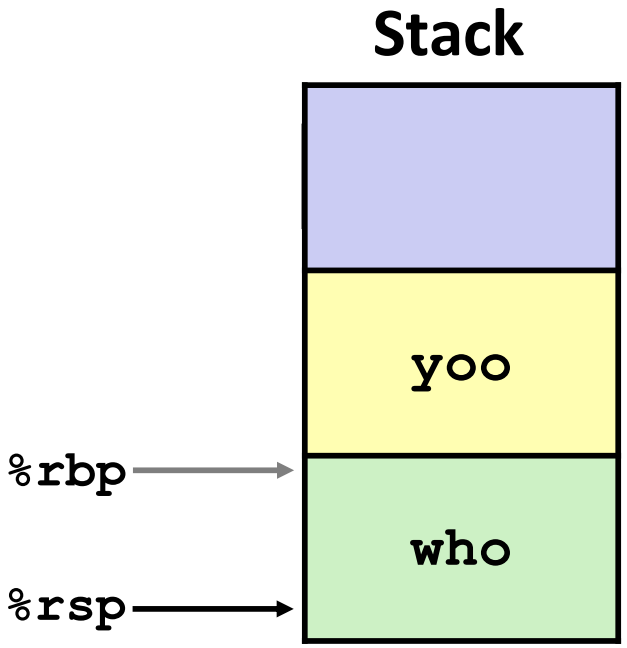




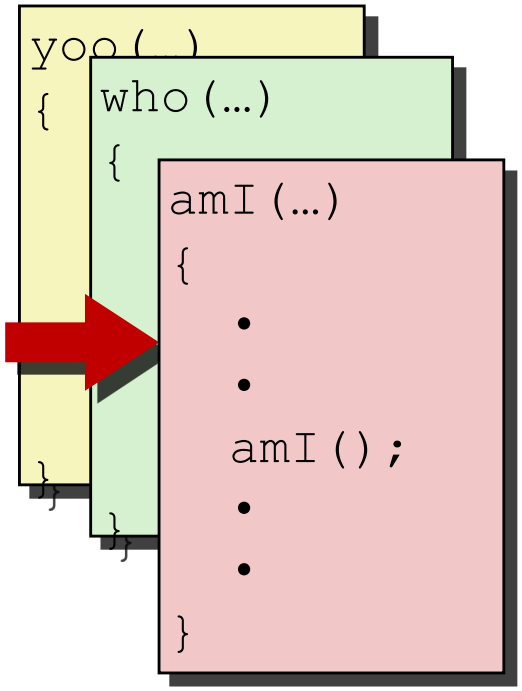
Example

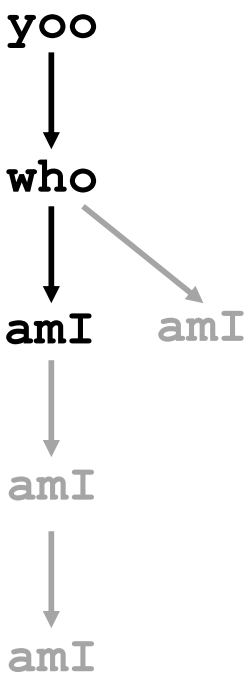


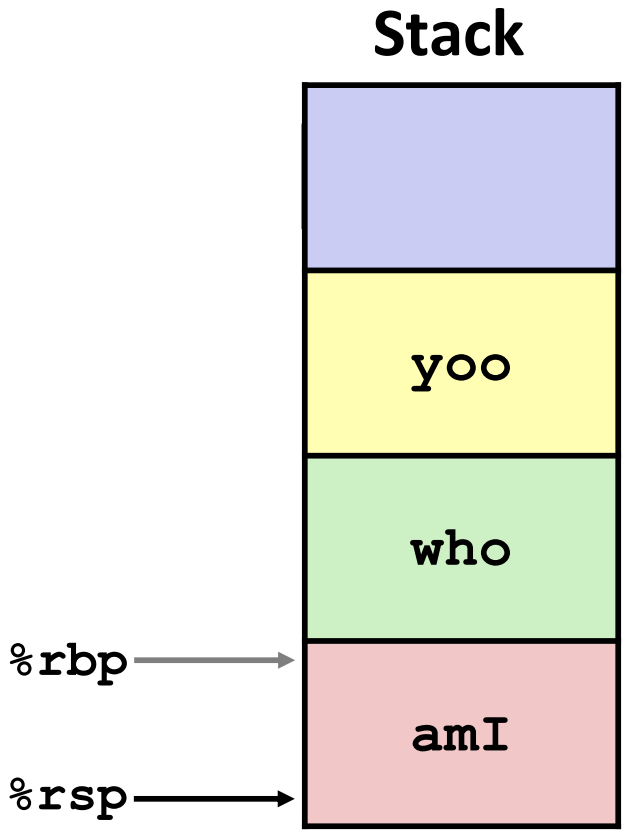




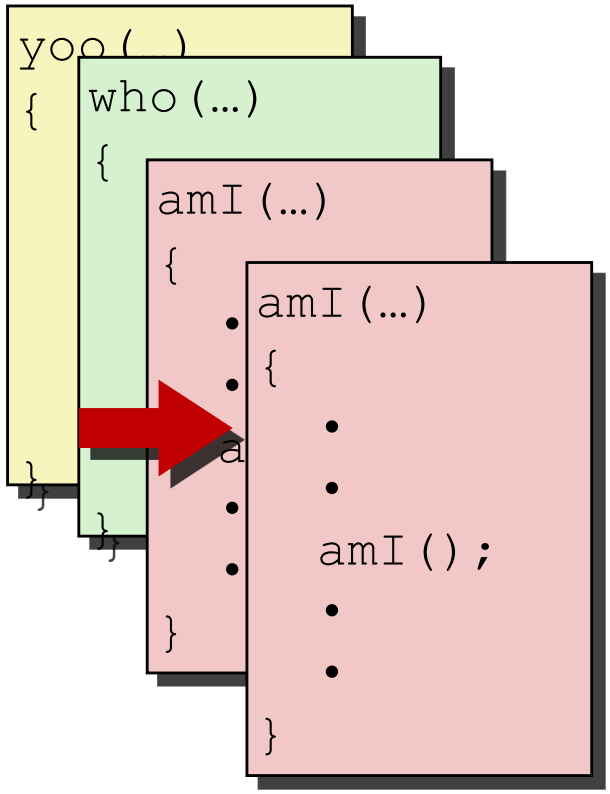
Example

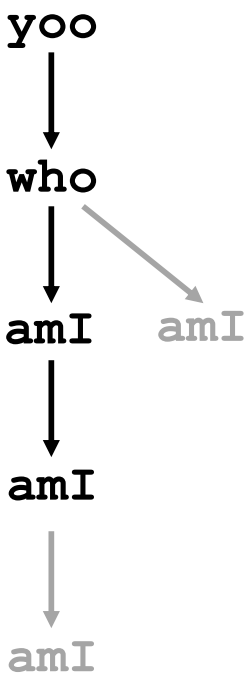

yoo(…)
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

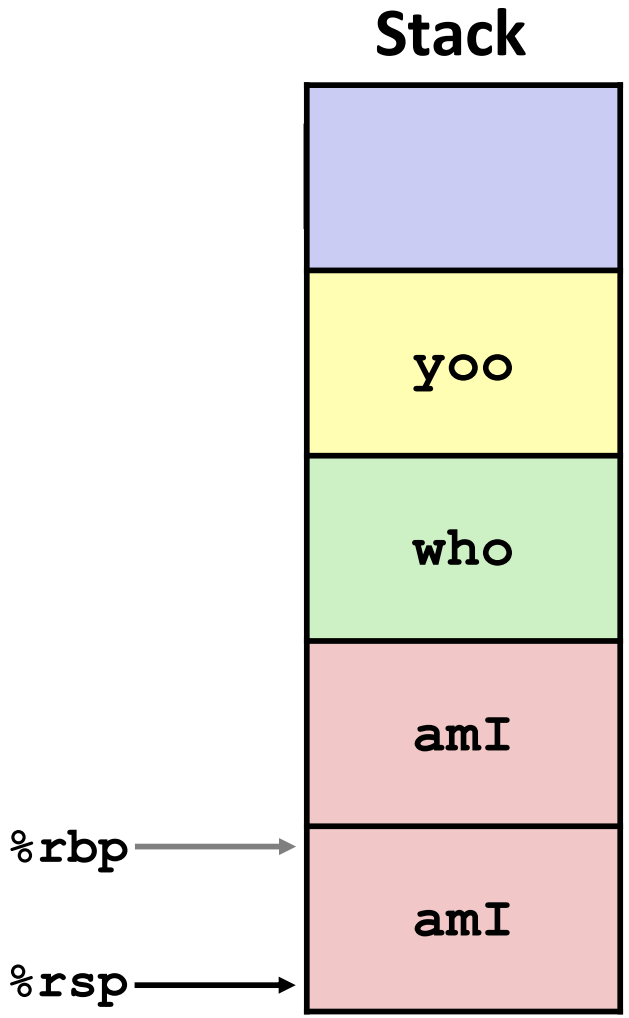

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

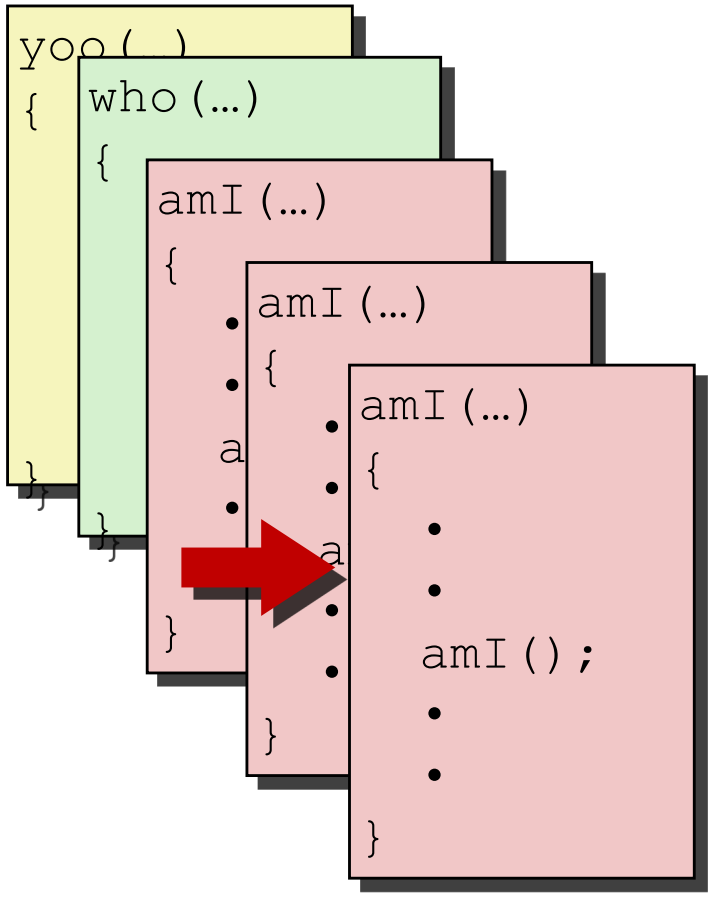
Example

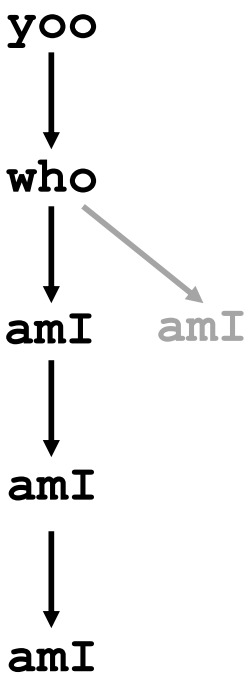

yoo(…)
{
who(…)
{
amI(…)
{
amI(…)
{
amI();
}

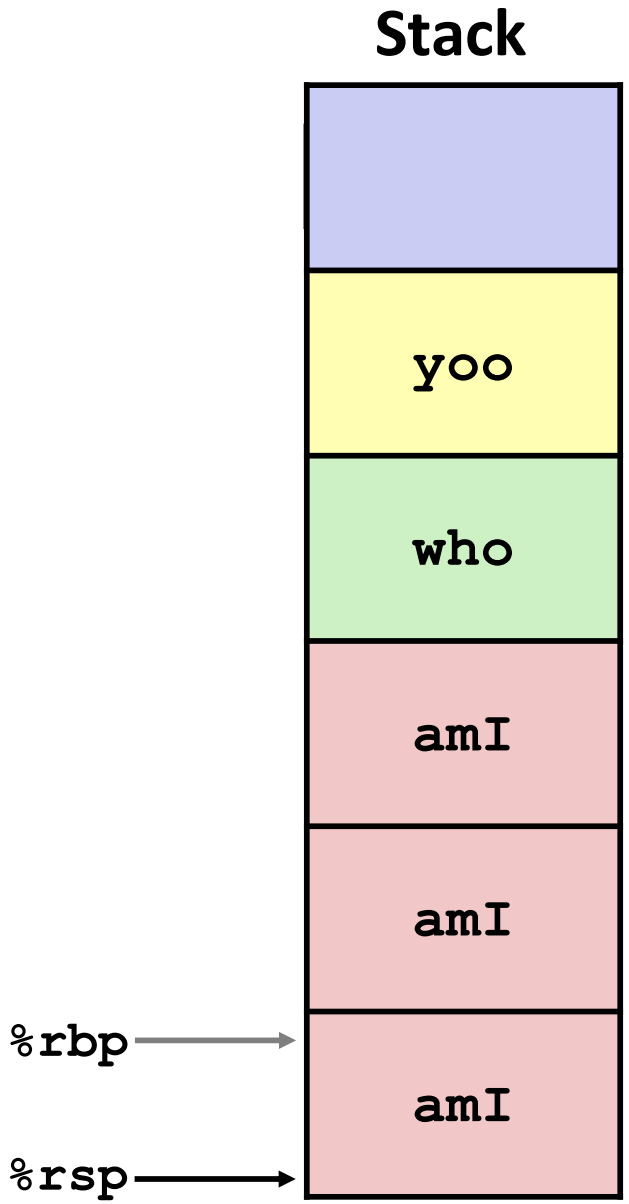

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

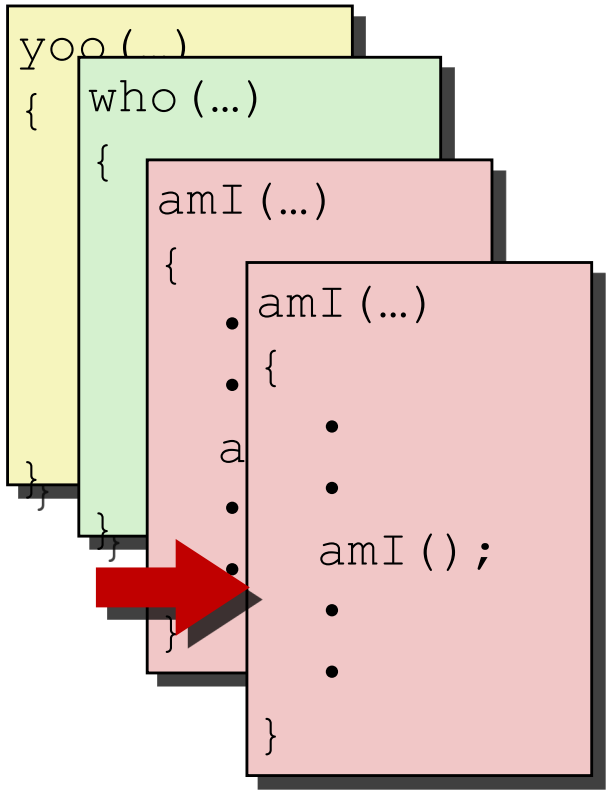
Example

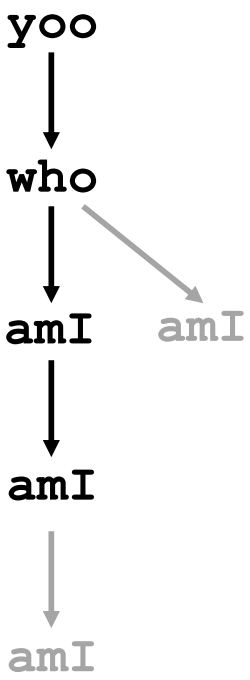

yoo(…)
{
who(…)
{
amI(…)
{
amI(…)
{
amI(…)
{
amI();
}

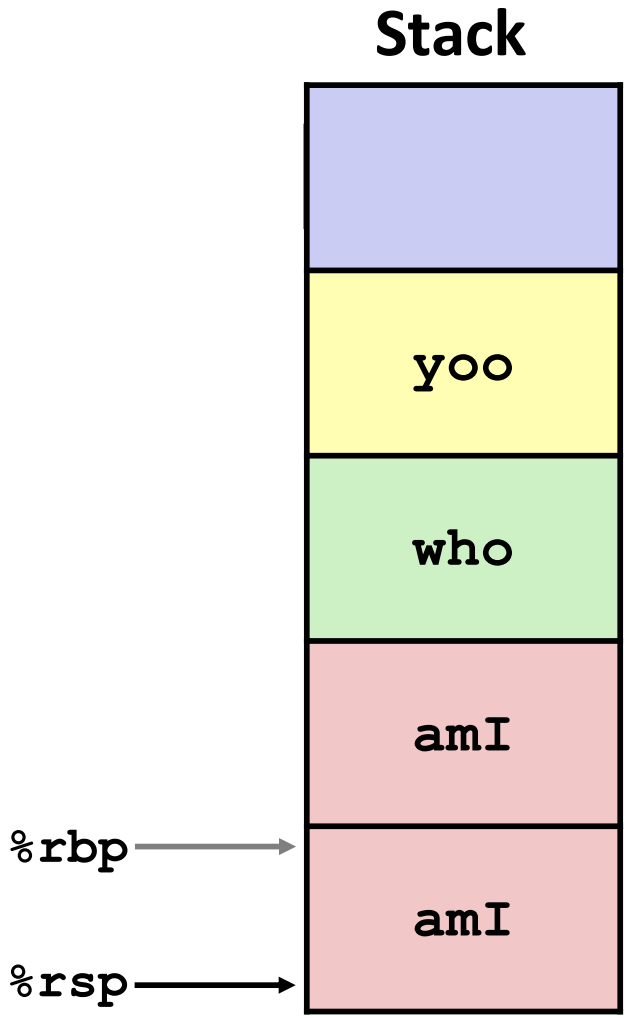

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
amI
%rbp
%rsp

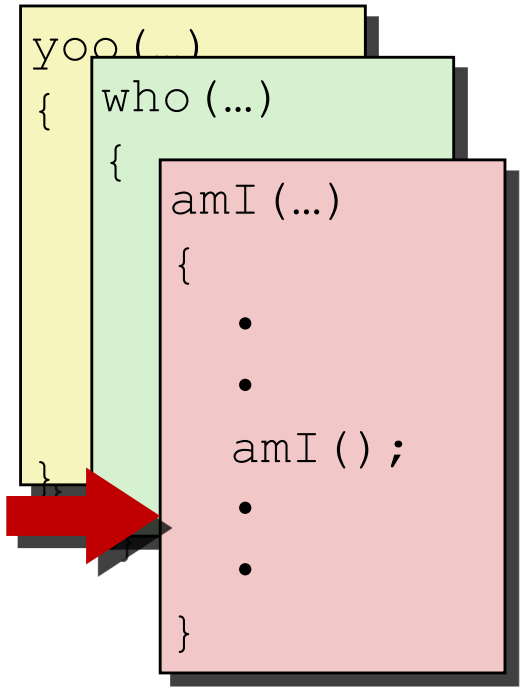
Example

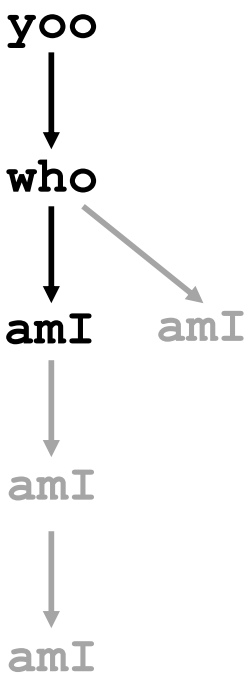

yoo(…)
{
who(…)
{
amI(…)
{
amI(…)
{
amI();
}

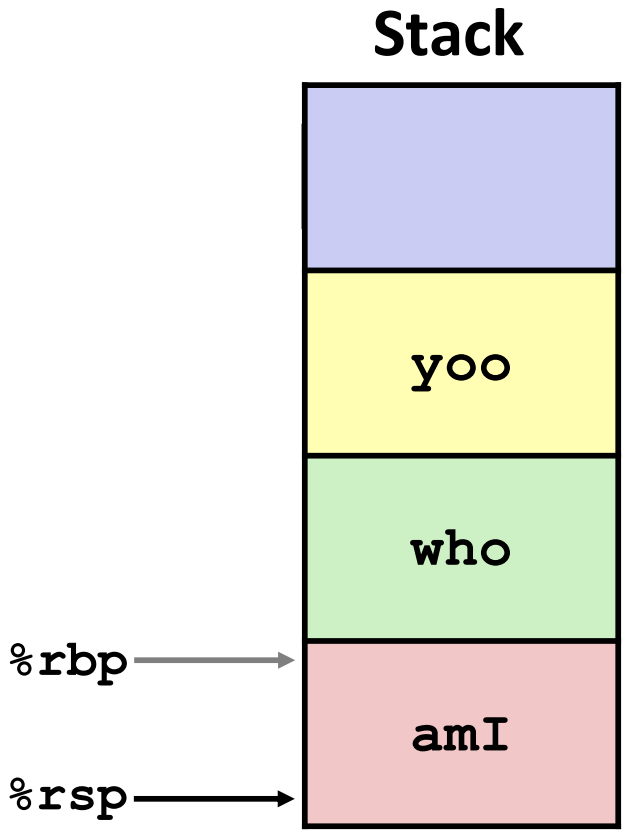

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

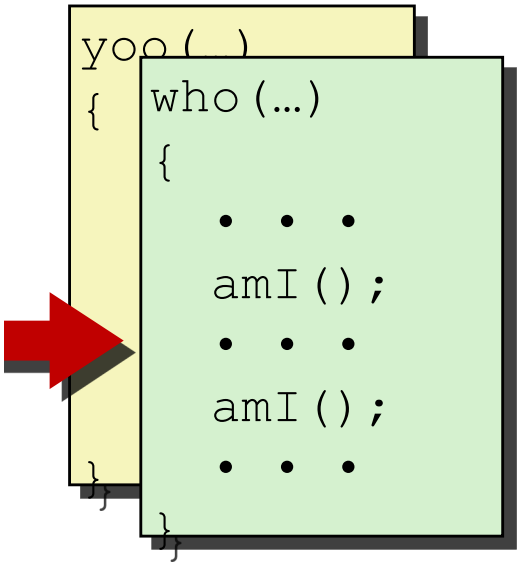
Example

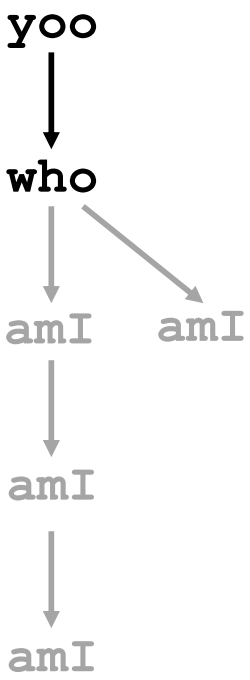

yoo( )
{
who(…)
{
amI(…)
{
amI();
}

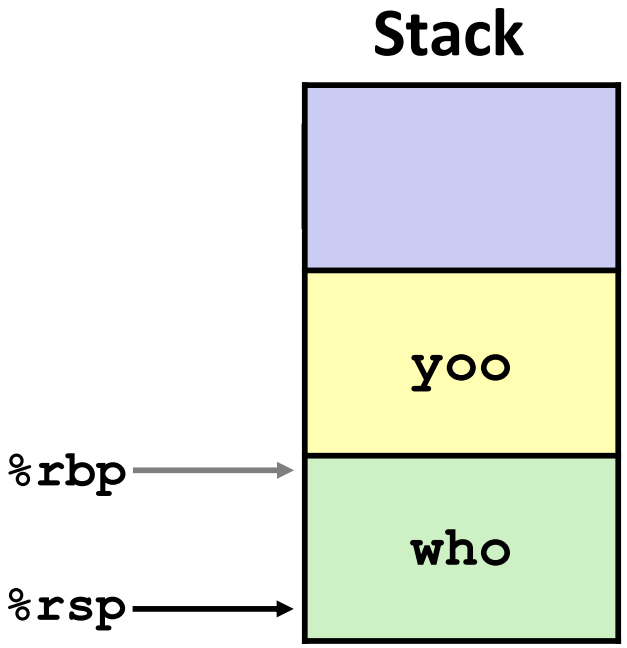

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

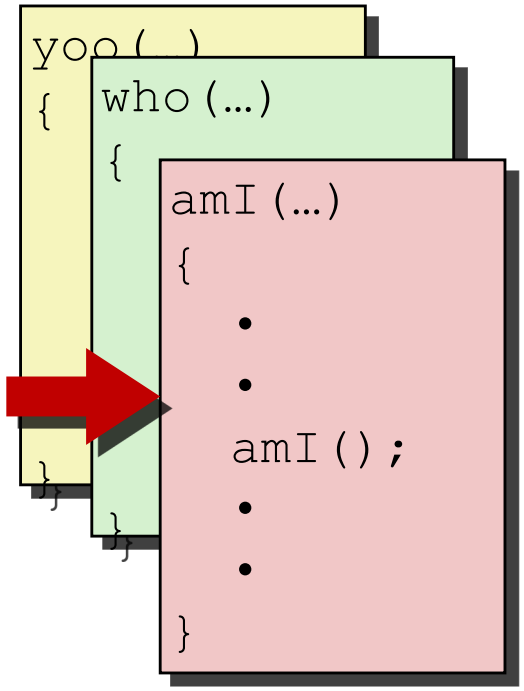
Example

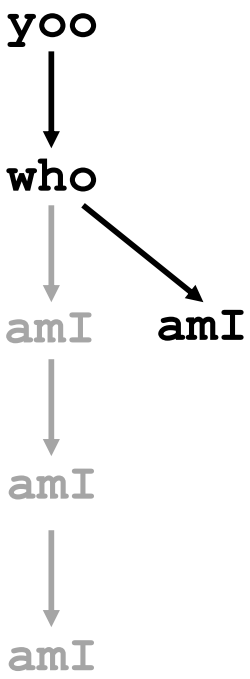


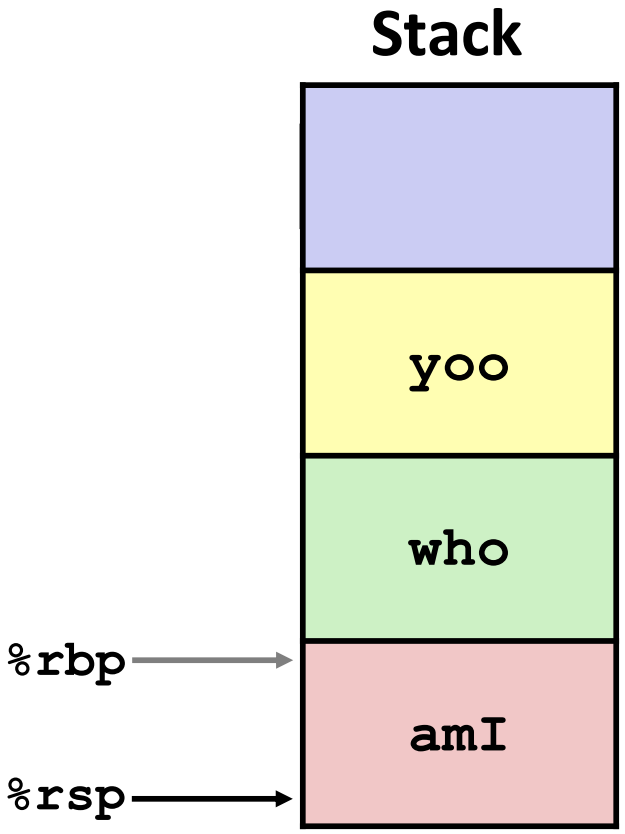




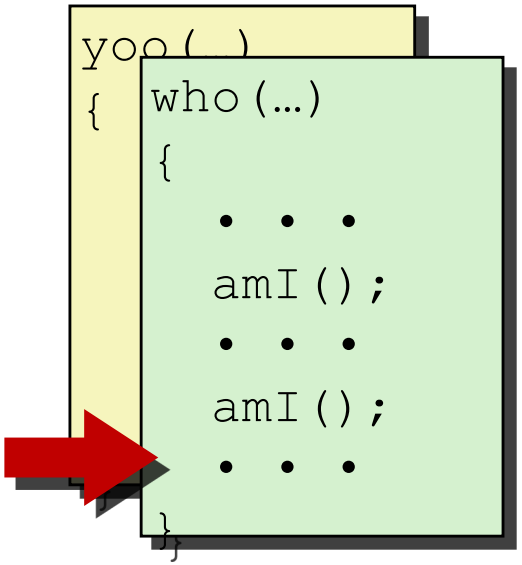
Example

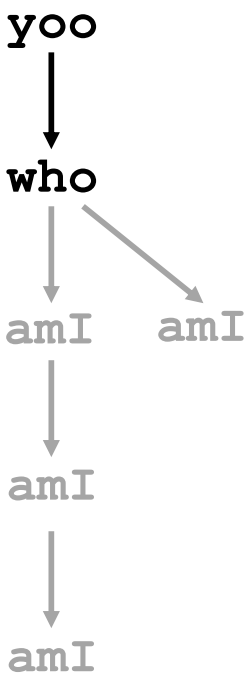

yoo( )
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

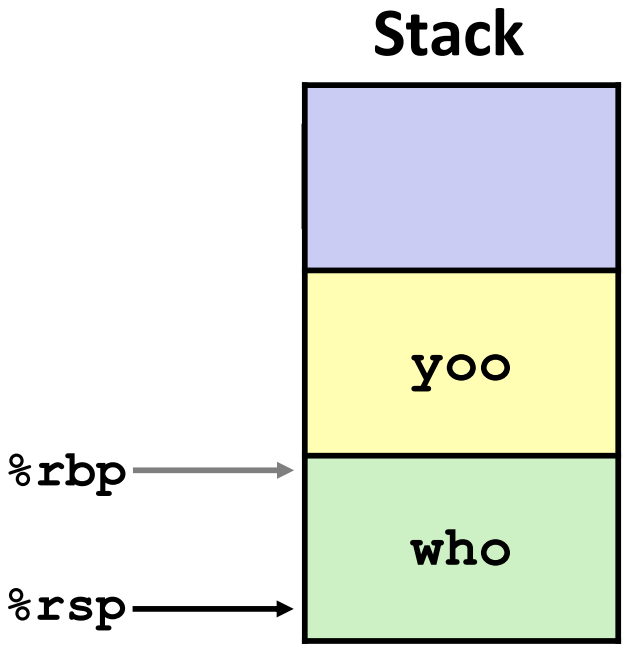

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

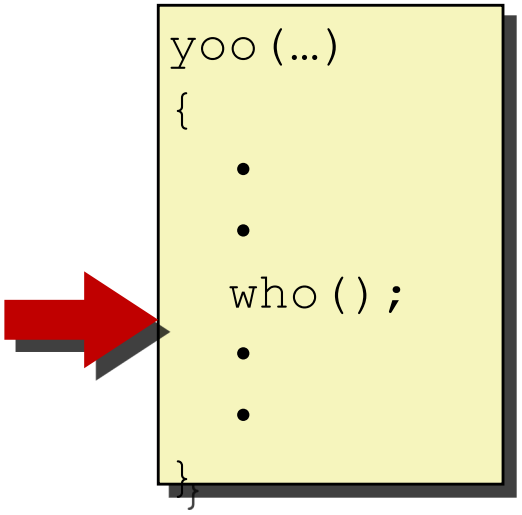
Example

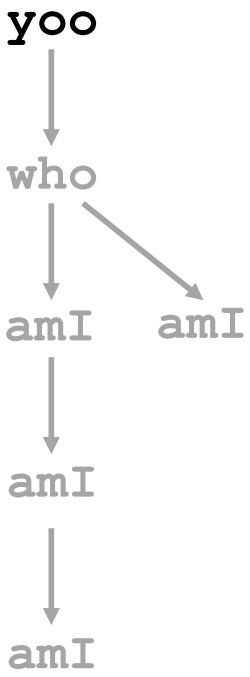


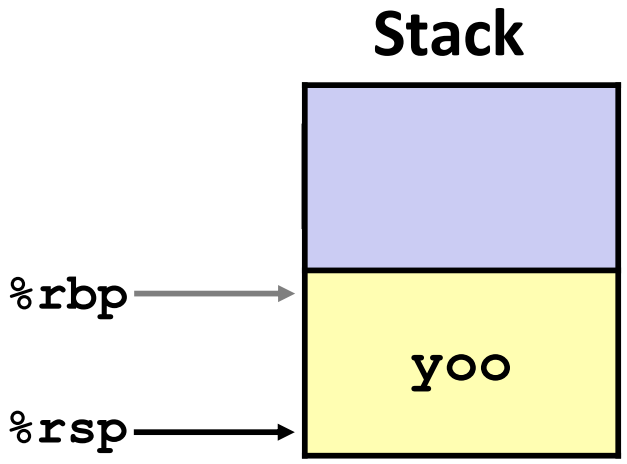




Example







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

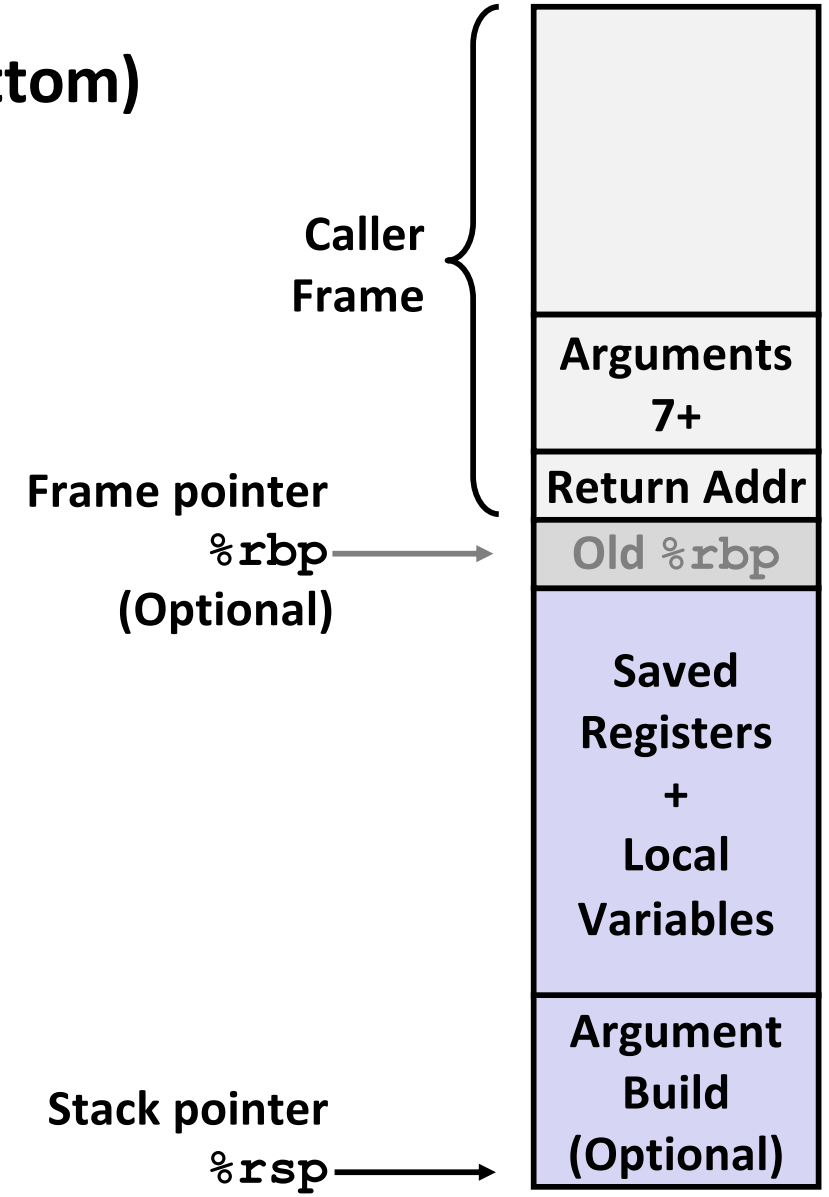
Saved register context

Old frame pointer (optional)

Caller Stack Frame

Return address
Pushed by `call` instruction

Arguments for this call



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`

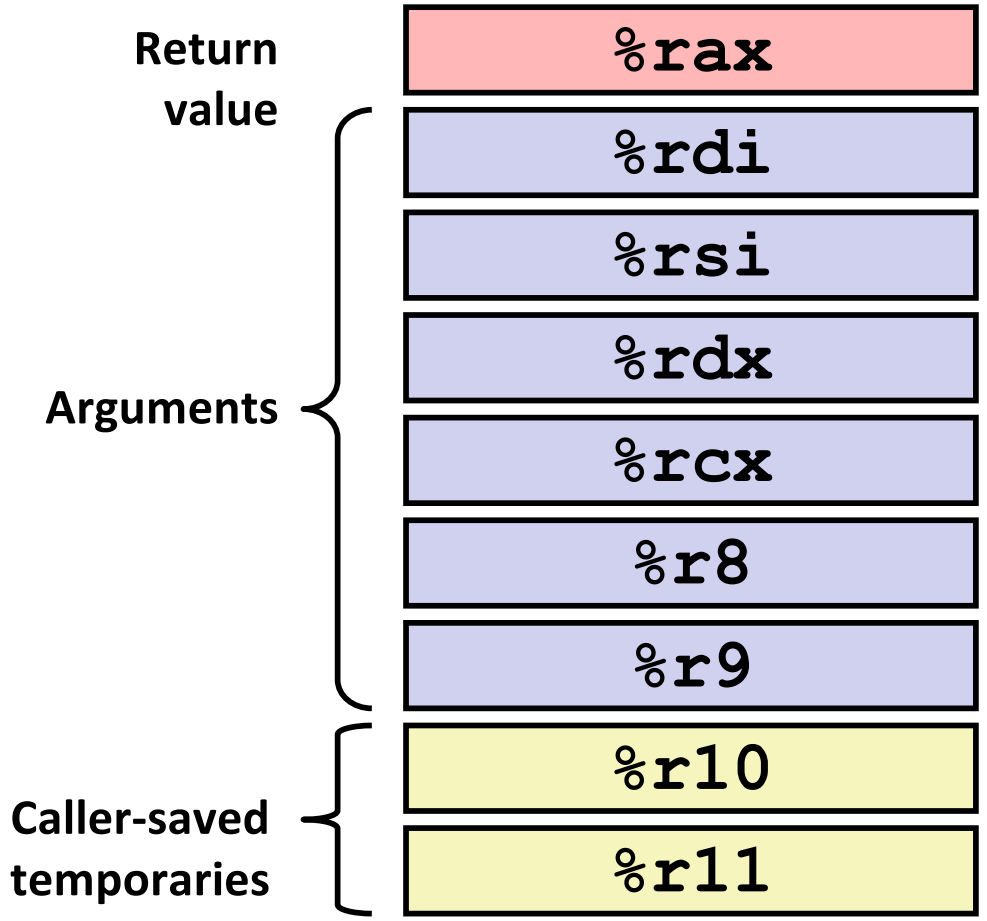
- Return value
- Also caller-saved
- Can be modified by procedure

`%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`

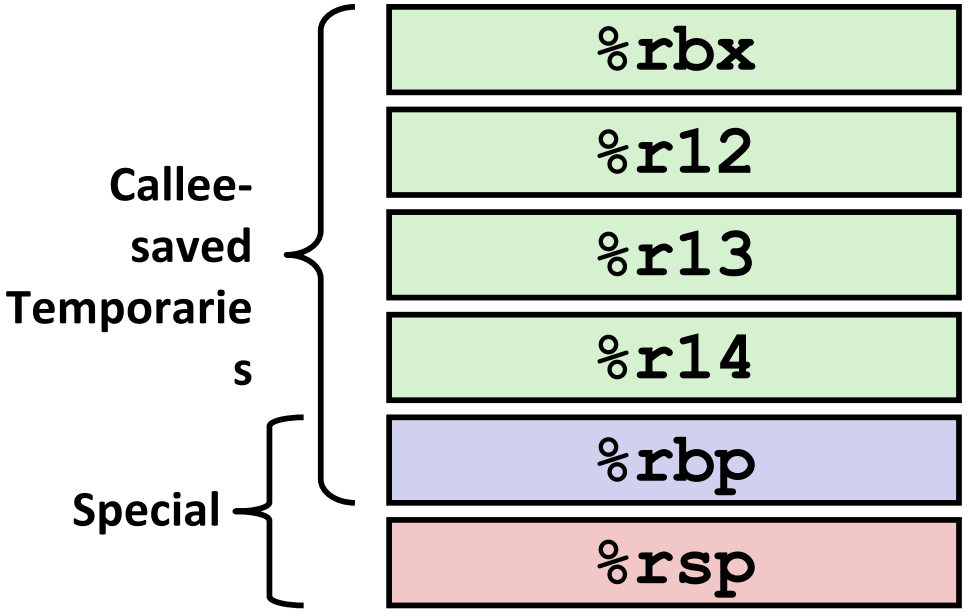
- Callee-saved
- Callee must save & restore

`%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



Summary : 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];
}
```

Mechanisms in Procedures

Passing control

To beginning of procedure code

Back to return point

Passing data

Procedure arguments

Return value

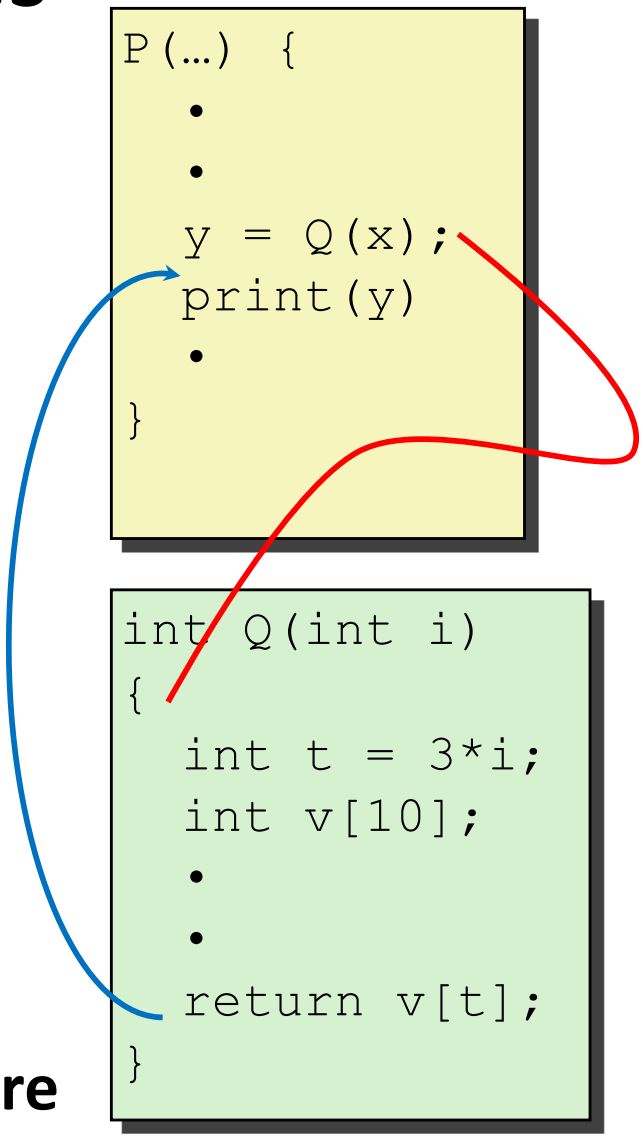
Memory management

Allocate during procedure execution

Deallocate upon return

Mechanisms all implemented with machine instructions

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



Mechanisms in Procedures

Passing control

- To beginning of procedure code
- Back to return point

Passing data

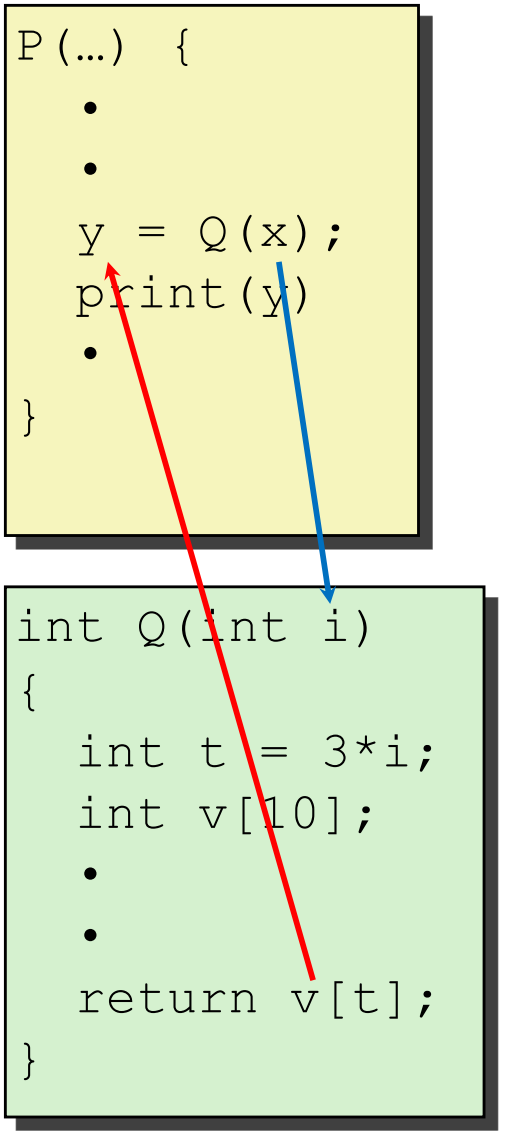
- Procedure arguments
- Return value

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



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];
}
```

Today

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
- **Arrays**
  - **One-dimensional**
  - **Multi-dimensional (nested)**
  - **Multi-level**
- Structures
  - Allocation
  - Access
  - Alignment

Reminder: Memory Organization

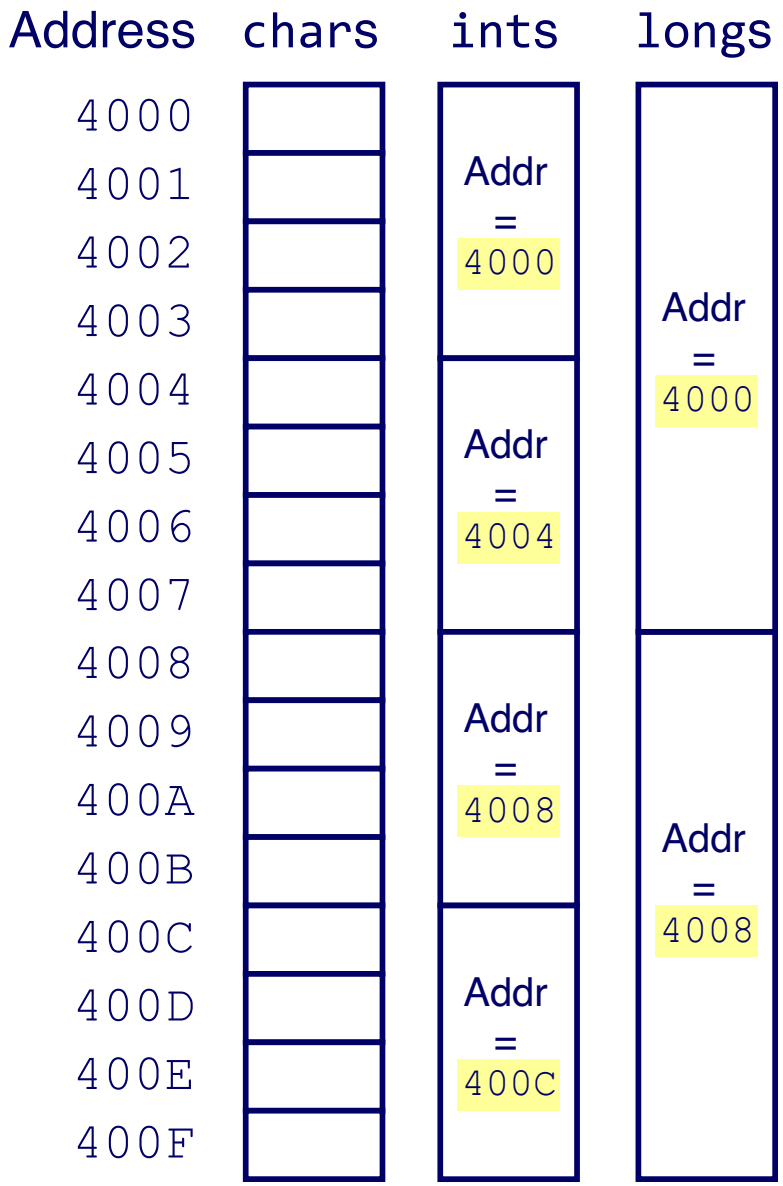
Memory locations do not have data types

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

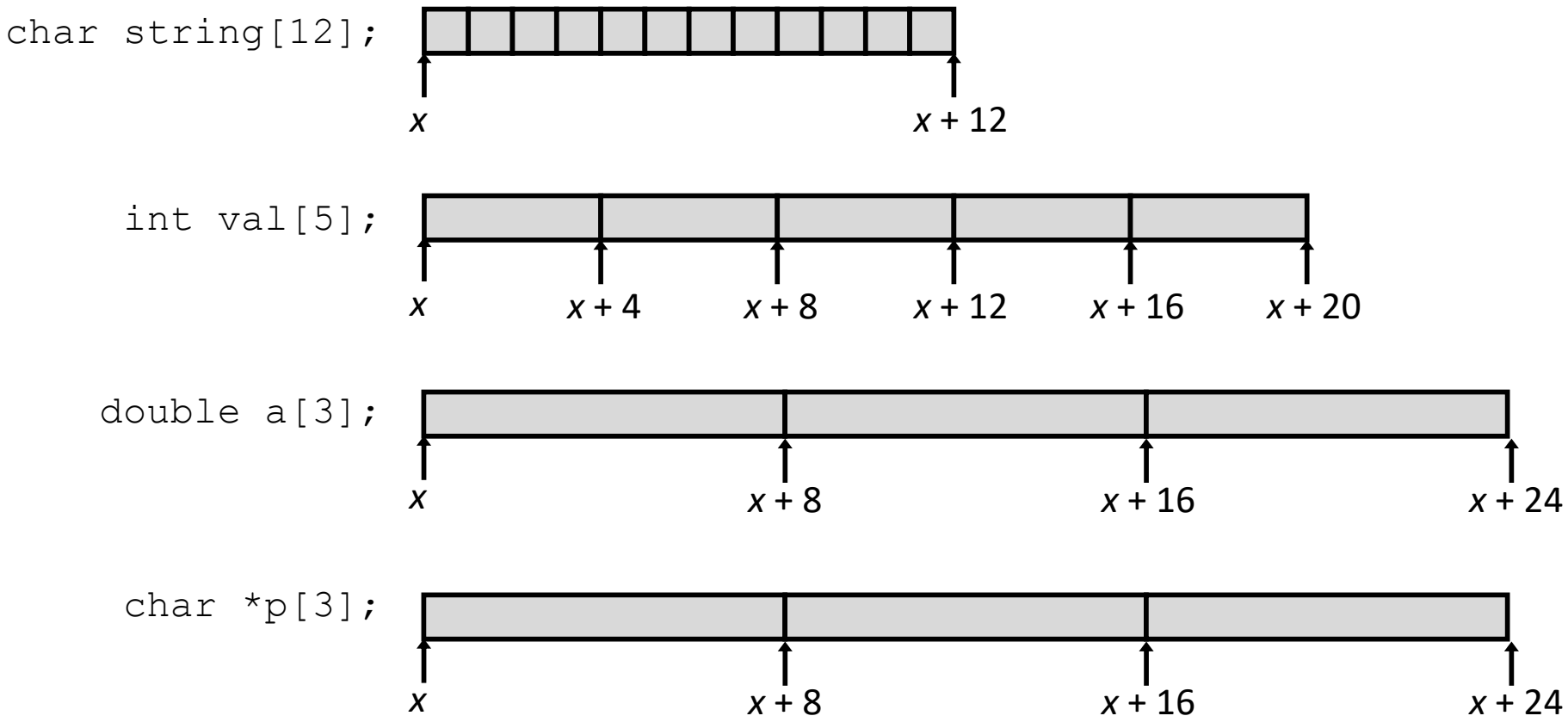


Array Allocation

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

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

Contiguously allocated region of *Length* * **sizeof**(*Type*) bytes in memory

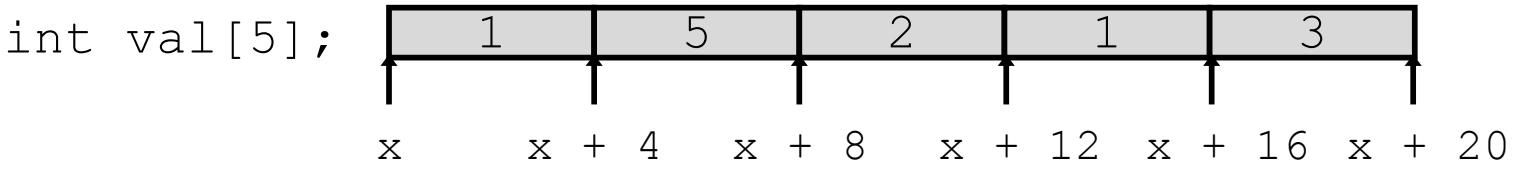


Array Access

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

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

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



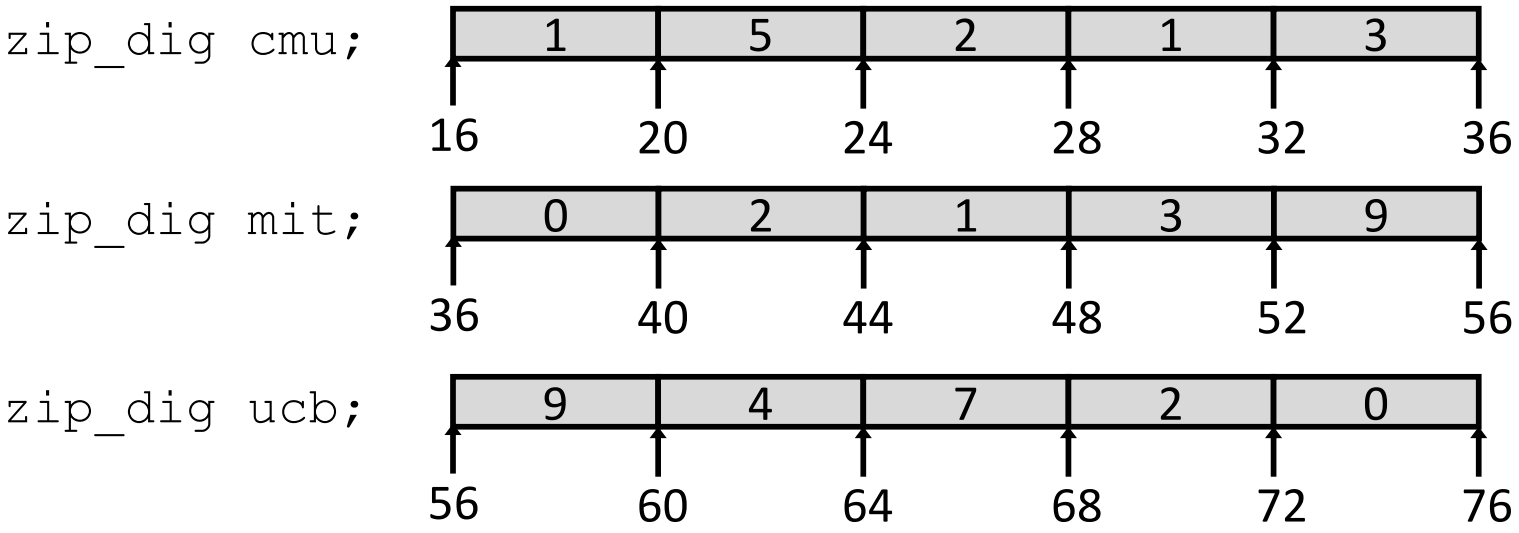
Expression	Type	Value	
`val[4]`	`int`	`3`	
`val[5]`	`int`	`??`	`// access past end`
`*(val+3)`	`int`	`1`	`// same as val[3]`
`val`	`int *`	`x`	
`val+1`	`int *`	`x + 4`	
`&val[2]`	`int *`	`x + 8`	`// same as val+2`
`val + i`	`int *`	`x + 4*i`	`// same as &val[i]`

[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 };
```



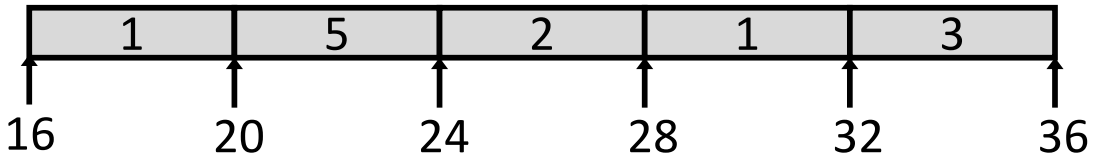
Declaration "`zip_dig cmu`" equivalent to "`int cmu[5]`"

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`						

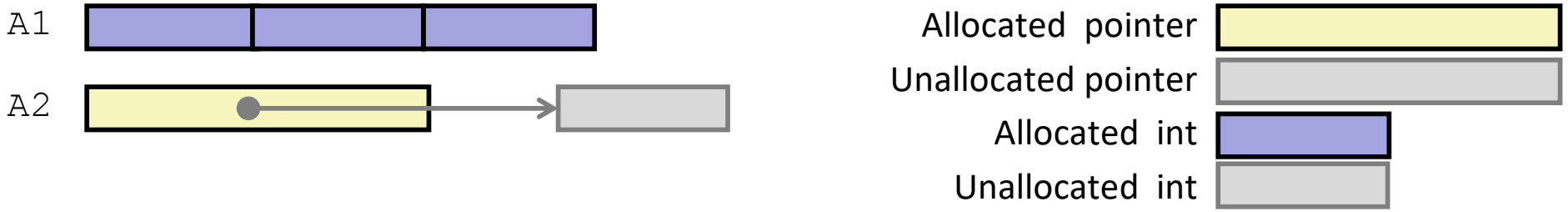
Comp: Compiles (Y/N)

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`						



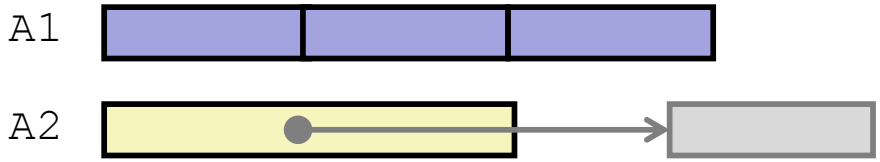
Comp: Compiles (Y/N)

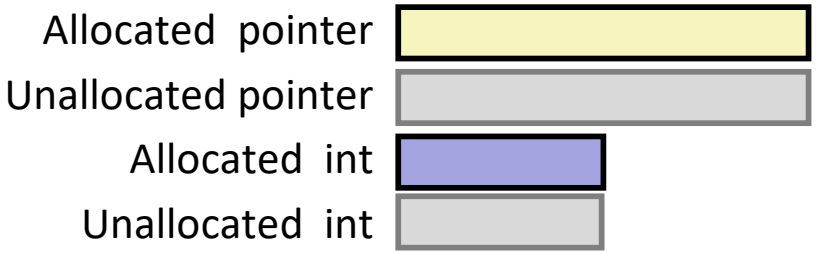
Bad: Possible bad pointer reference (Y/N)

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





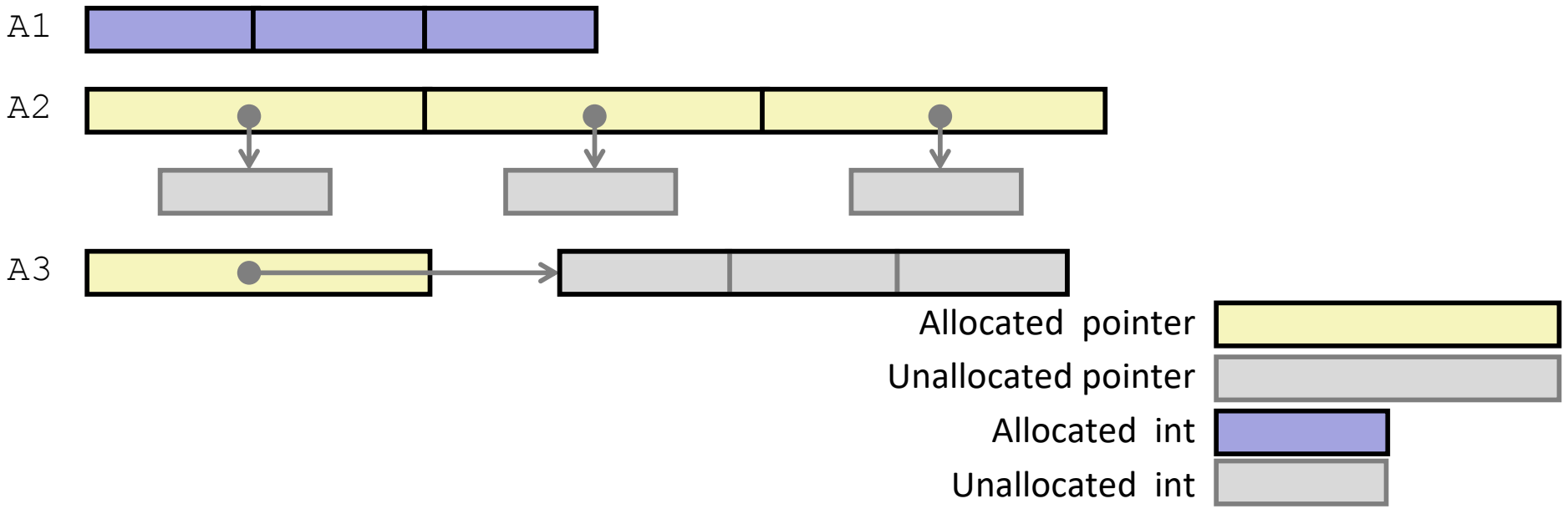
Comp: Compiles (Y/N)

Bad: Possible bad pointer reference (Y/N)

Size: Value returned by `sizeof`

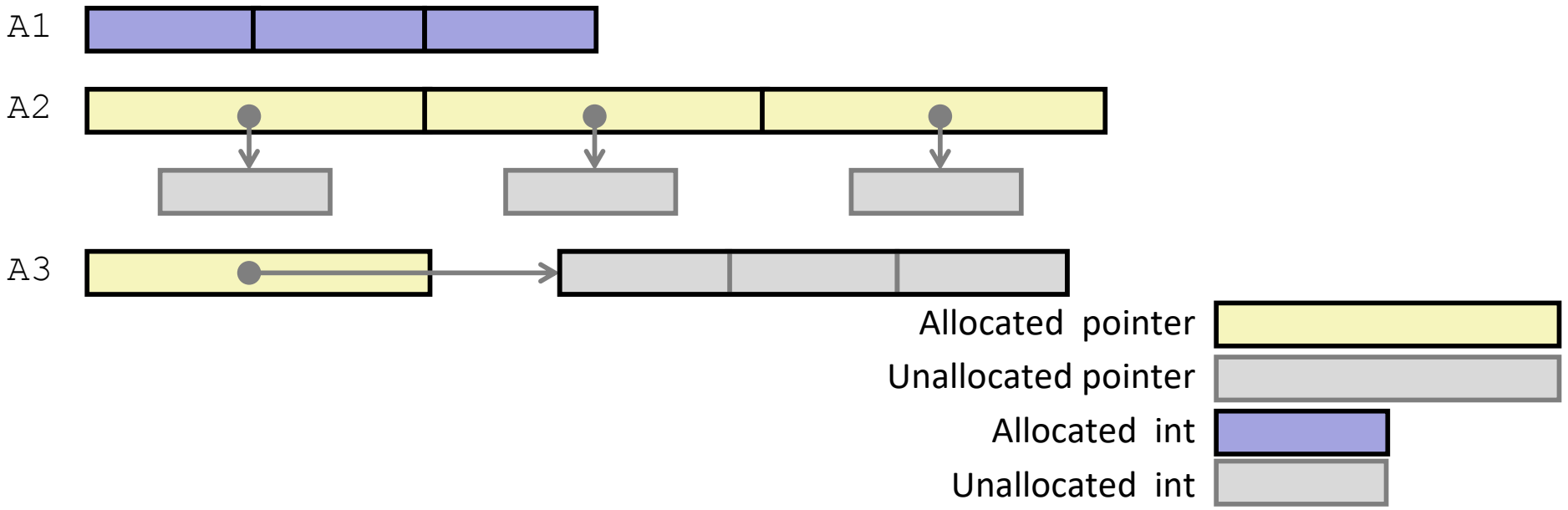
Understanding Pointers & Arrays #2

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*

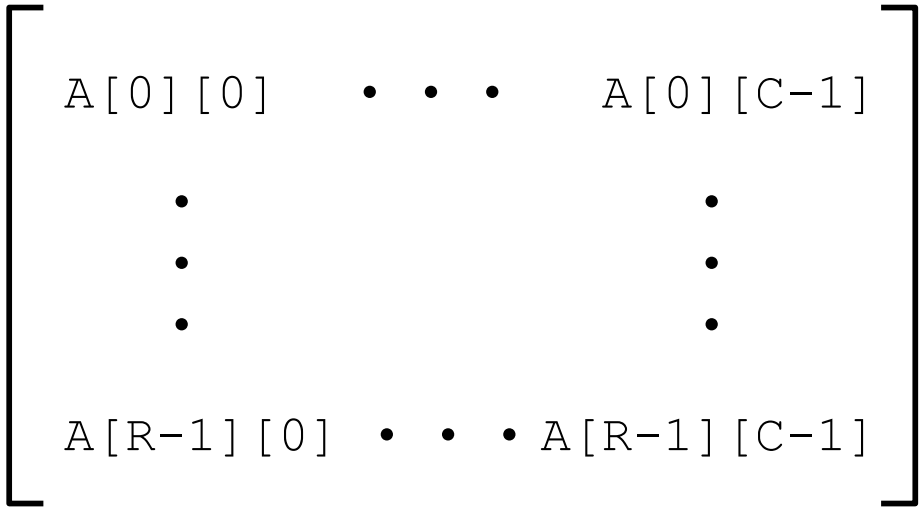
R rows, *C* columns

Array Size

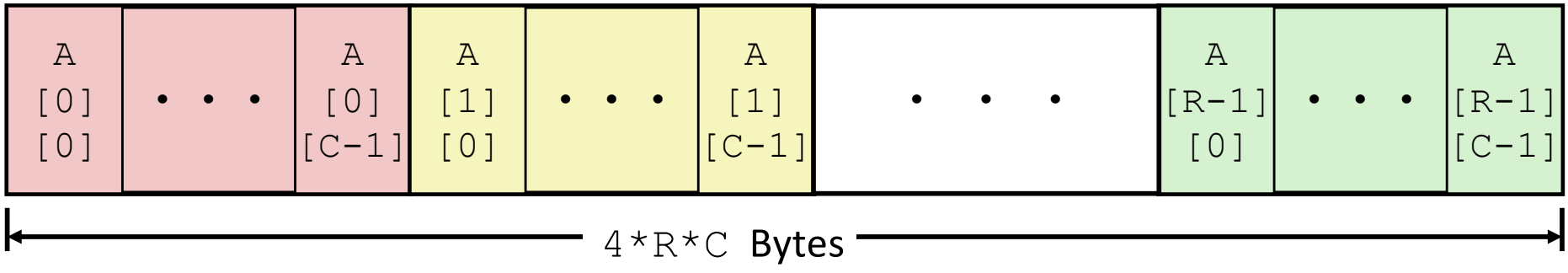
R * *C* * **sizeof**(*T*) bytes

Arrangement

Row-Major Ordering



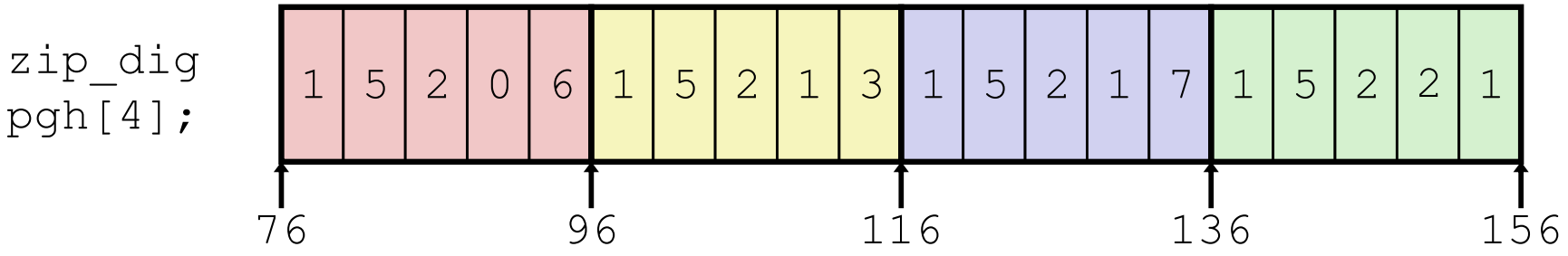

```
int A[R][C];
```



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

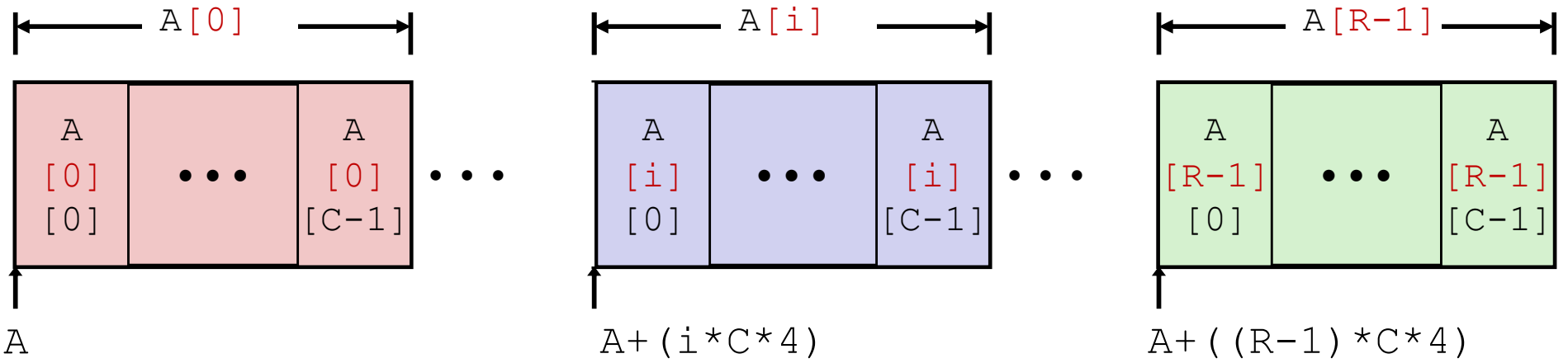
Nested Array Row Access

Row Vectors

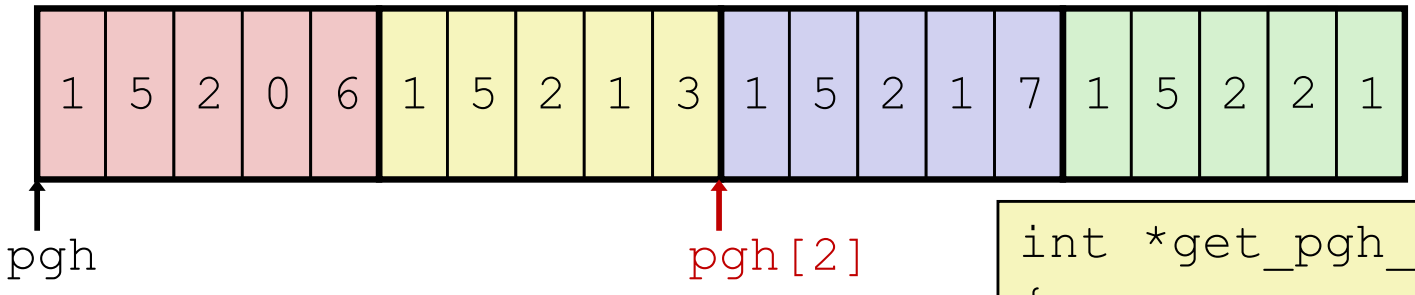
- **`A[i]`** is array of *C* elements of type *T*

  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

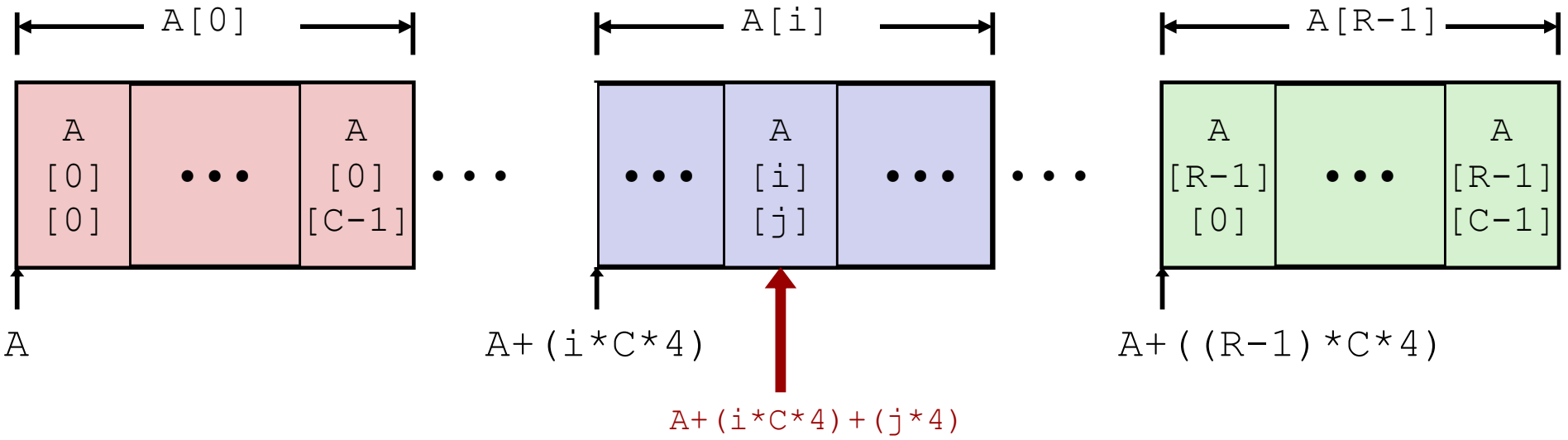
Array Elements

- **`A[i][j]`** is element of type *T*, which requires *K* bytes

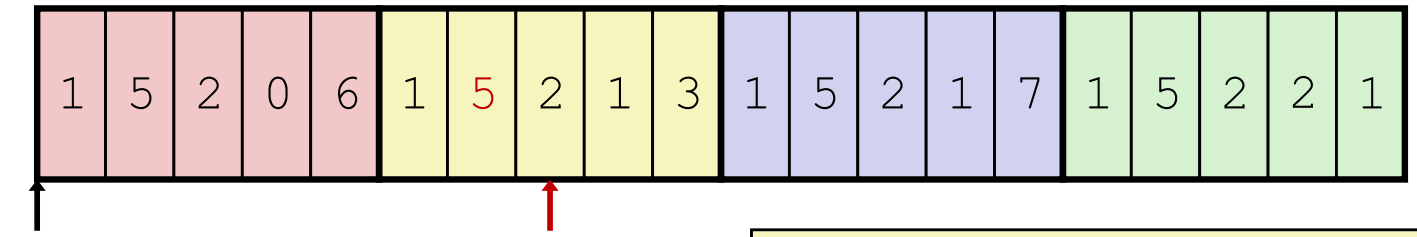
  Address **`A + i * (C * K) + j * K`**

  `= 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)`

Example: copy_element

```
#define M
#define N

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
```

Example: copy_element

```
#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];
}
```

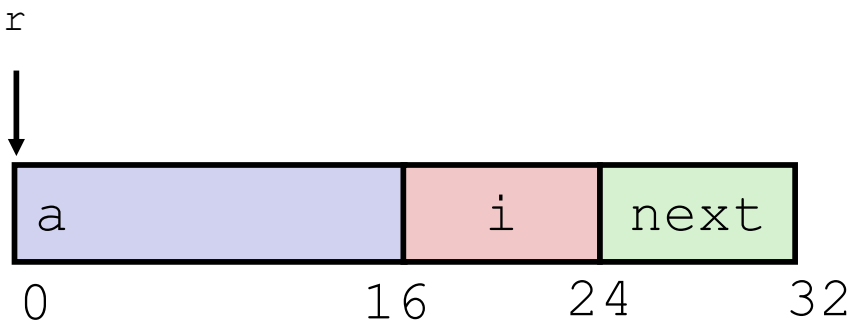
```
copy_element:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%ecx
    movl 12(%ebp),%ebx
    movl %ecx,%edx
    leal (%ebx,%ebx,8),%eax
    sall $4,%edx
    sall $2,%eax
    subl %ecx,%edx
    movl mat2(%eax,%ecx,4),%eax
    sall $2,%edx
    movl %eax,mat1(%edx,%ebx,4)
    movl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

Today

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
- 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

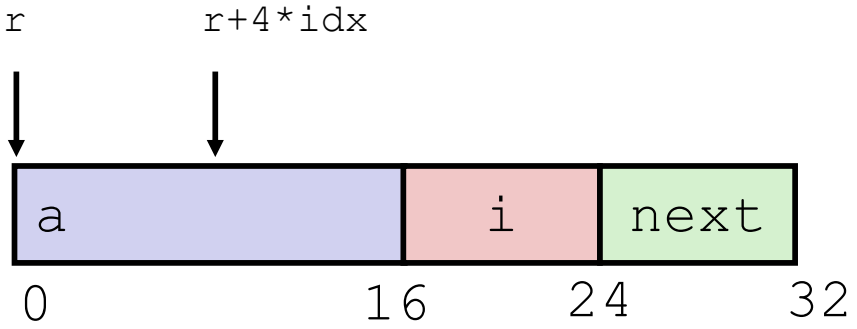
- Even if another ordering could be more compact

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

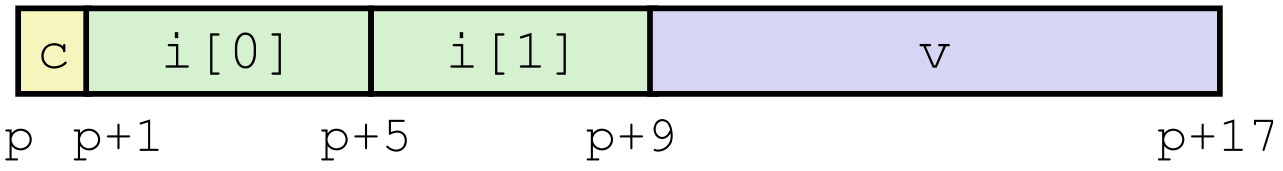
- Offset of each structure member determined at compile time
- Compute as **`r + 4*idx`**

```
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
```

Structures & Alignment

Unaligned Data

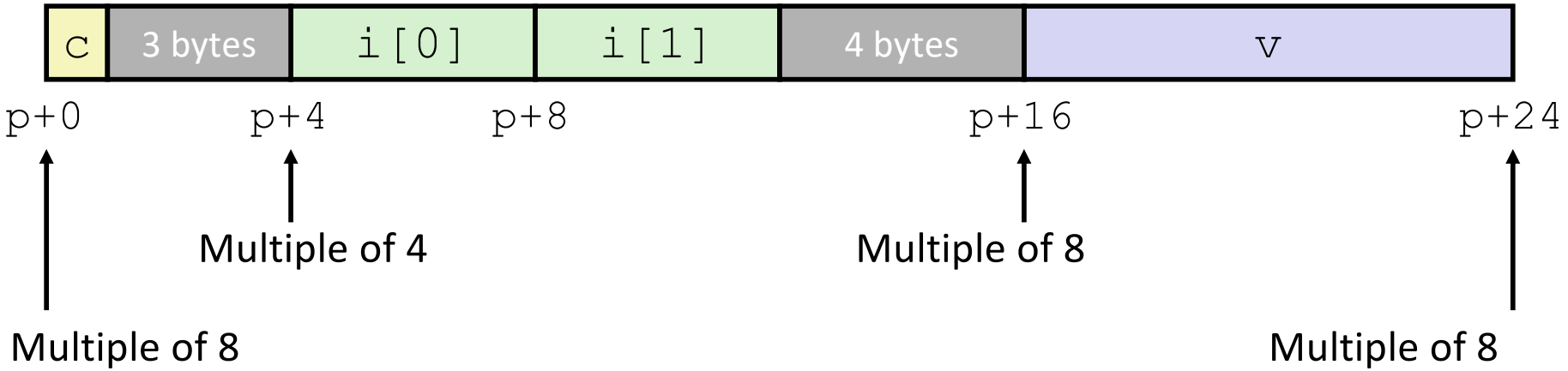



```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

Aligned Data

Primitive data type requires B bytes implies

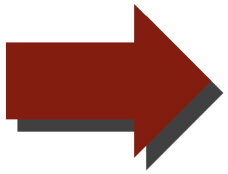
Address must be multiple of B



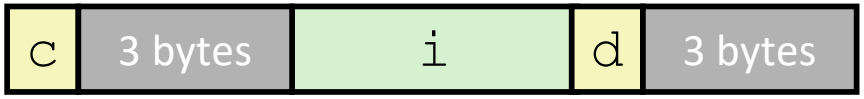
Saving Space

Put large data types first

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

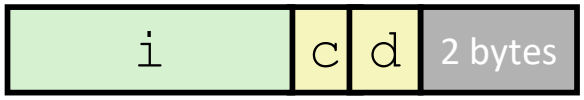


```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```



12 bytes

Effect (largest alignment requirement K=4)



8 bytes