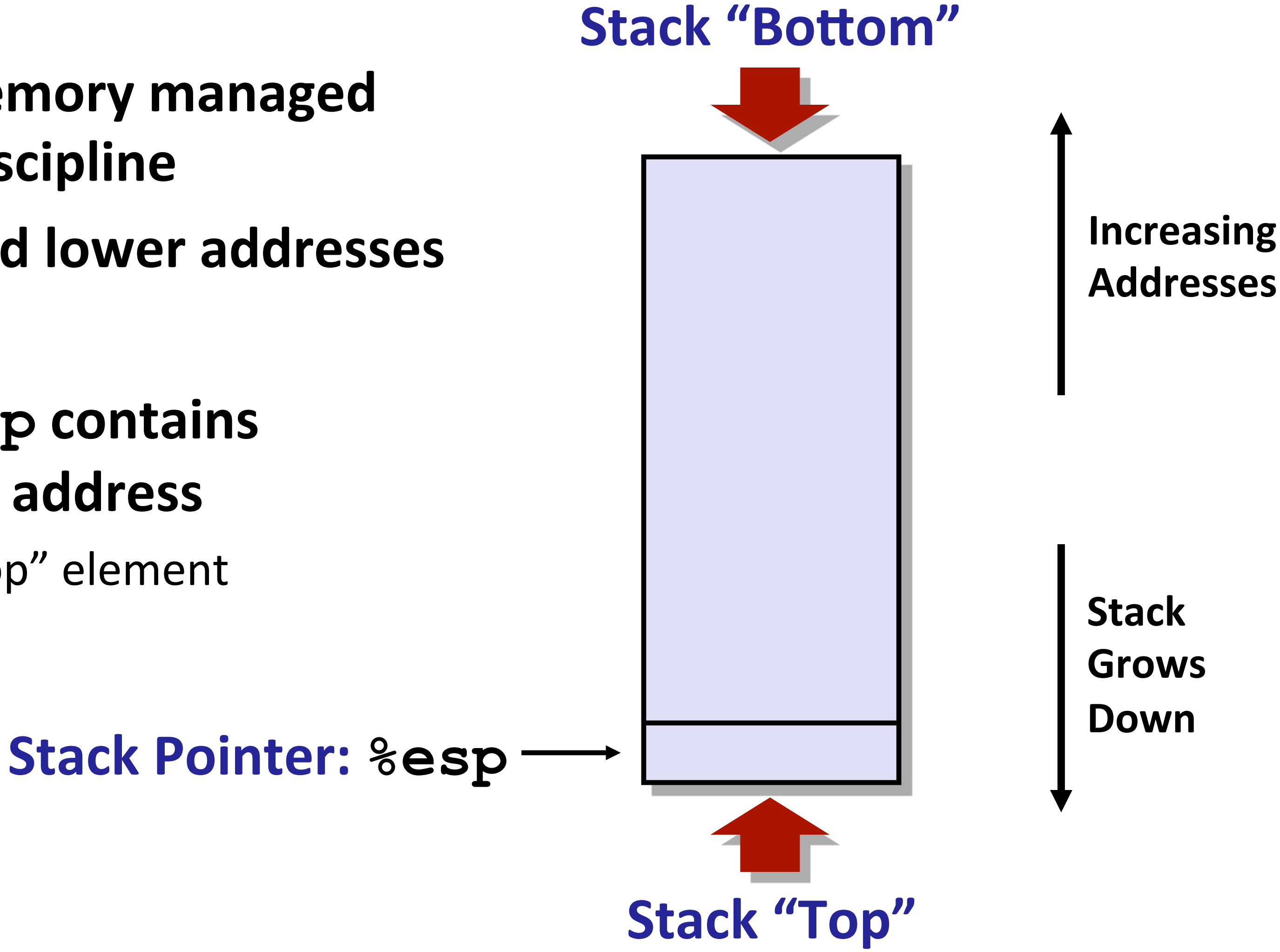
IA32 Procedures

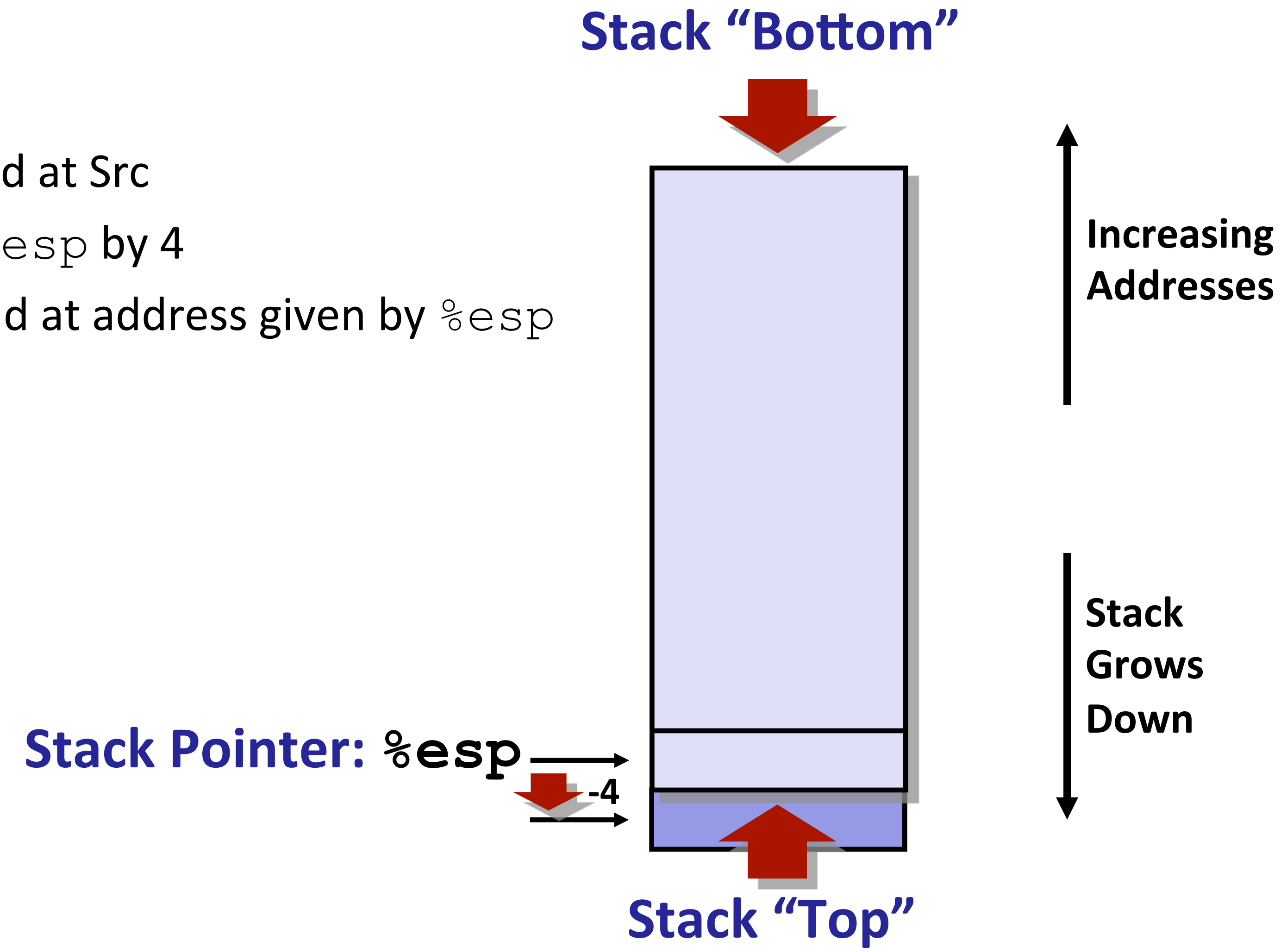
IA32 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%esp` contains lowest stack address
  - address of "top" element

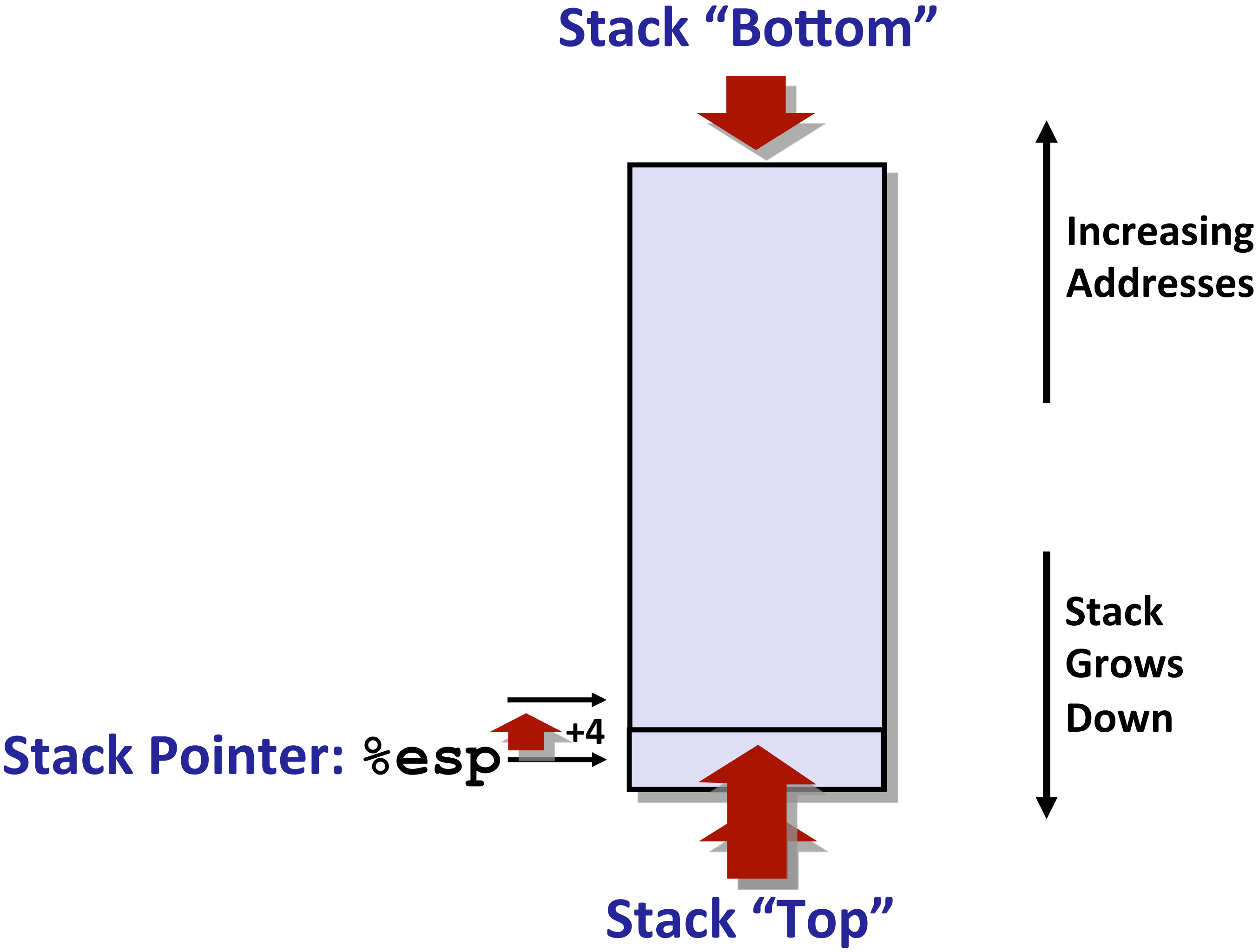


IA32 Stack: Push

- **`pushl` Src**
  - Fetch operand at Src
  - Decrement `%esp` by 4
  - Write operand at address given by `%esp`



IA32 Stack: Pop



Procedure Control Flow

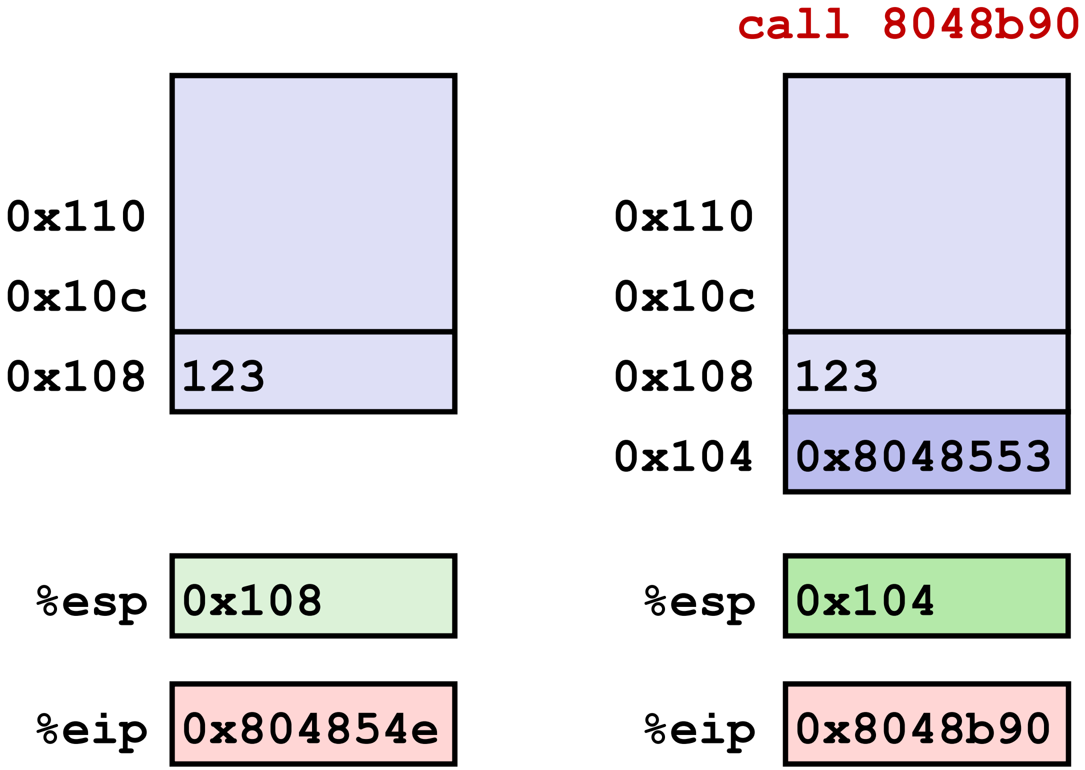
- **Use stack to support procedure call and return**
- **Procedure call: `call label`**
  - Push return address on stack
  - Jump to label
- **Return address:**
  - Address of the next instruction right after call
  - Example from disassembly

```
804854e:   e8 3d 06 00 00      call    8048b90
  <main>
8048553:   50                  pushl   %eax
```

  - Return address = `0x8048553`
- **Procedure return: `ret`**
  - Pop address from stack
  - Jump to address

Procedure Call Example

```
804854e:      e8 3d 06 00 00          call   8048b90 <main>
8048553:      50                      pushl  %eax
```



call 8048b90

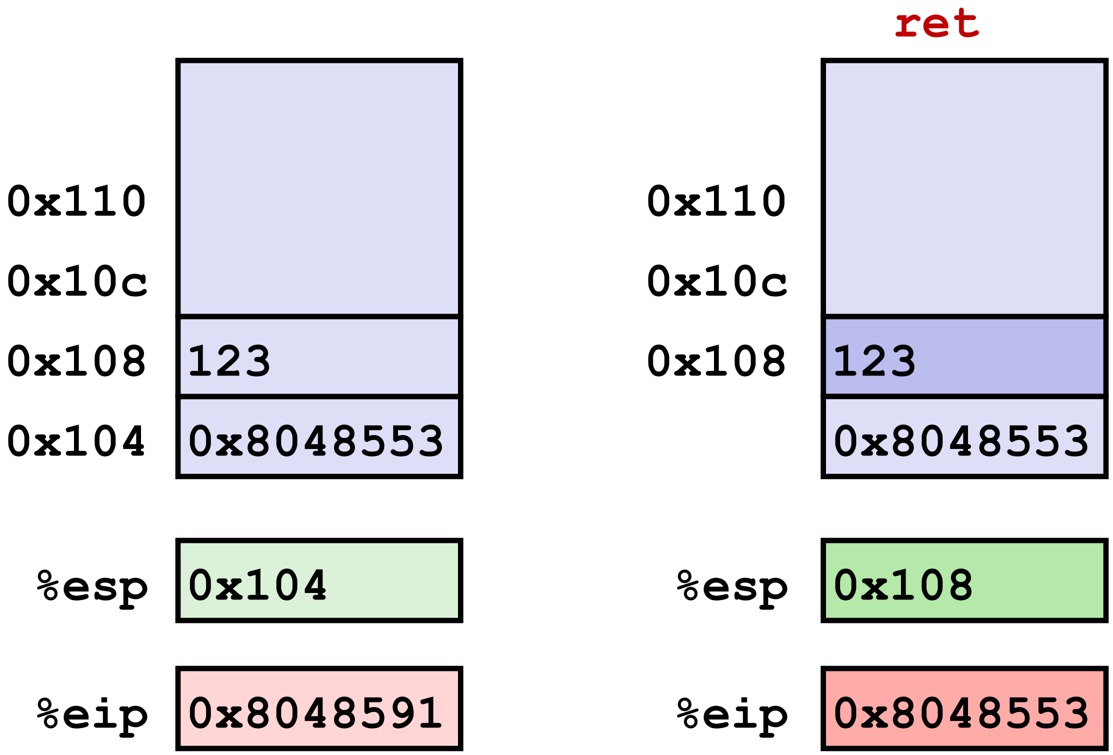
0x110	
0x10c	
0x108	123
%esp	0x108
%eip	0x804854e

0x110	
0x10c	
0x108	123
0x104	0x8048553
%esp	0x104
%eip	0x8048b90

`%eip`: program counter

Procedure Return Example

```
8048591:      c3                    ret
```



			ret
0x110		0x110	
0x10c		0x10c	
0x108	123	0x108	123
0x104	0x8048553		0x8048553
%esp	0x104	%esp	0x108
%eip	0x8048591	%eip	0x8048553

`%eip:` program counter

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

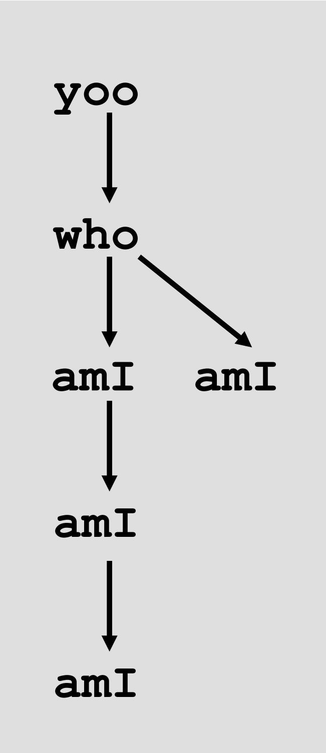
Call Chain Example

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

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

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

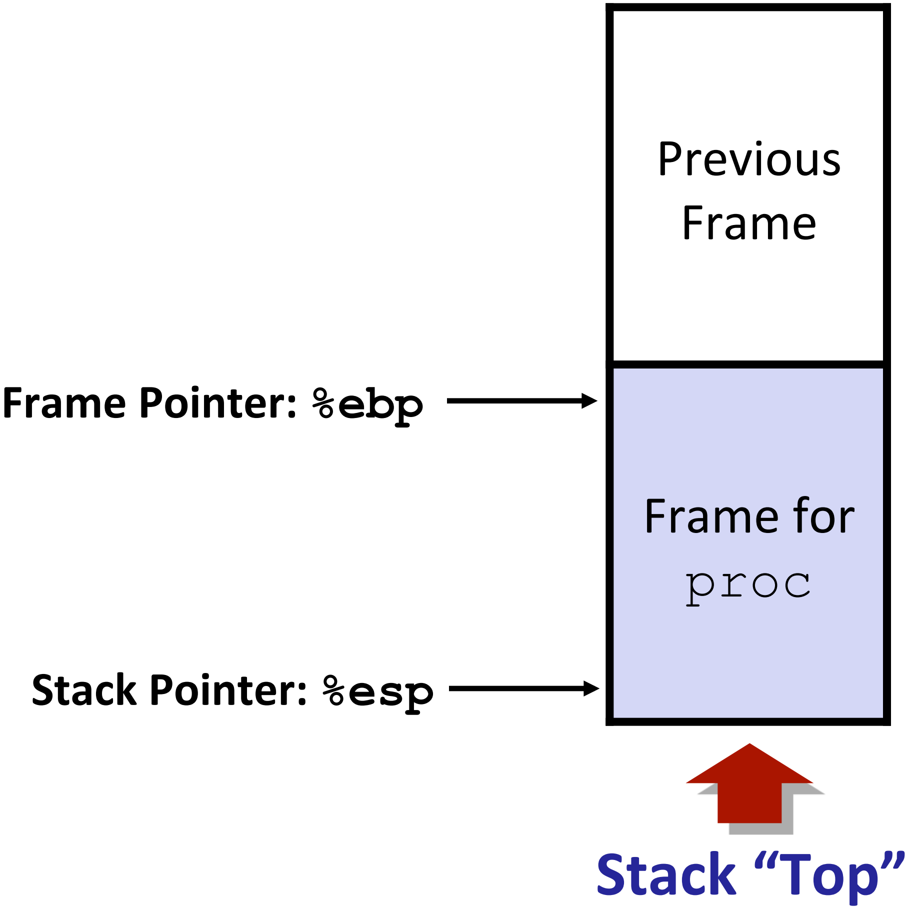
Example
Call Chain



Procedure `amI()` is recursive

Stack Frames

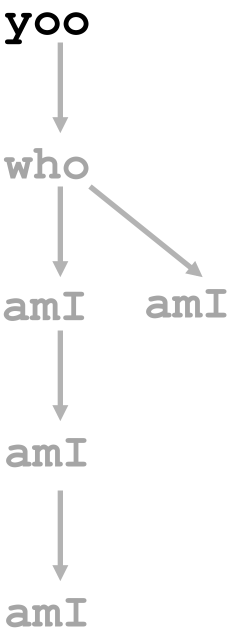
- **Contents**
  - Local variables
  - Return information
  - Temporary space

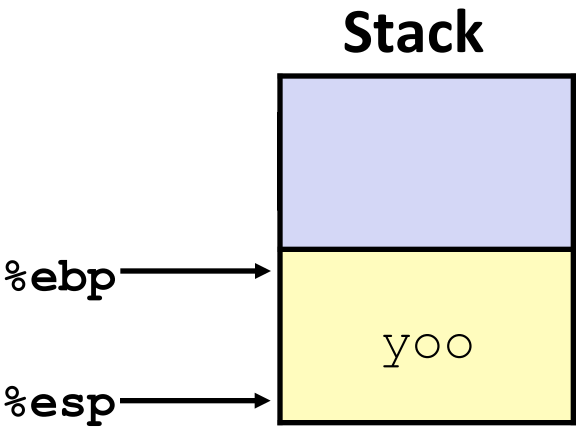


- **Management**
  - Space allocated when enter procedure
    - "Set-up" code
  - Deallocated when return
    - "Finish" code

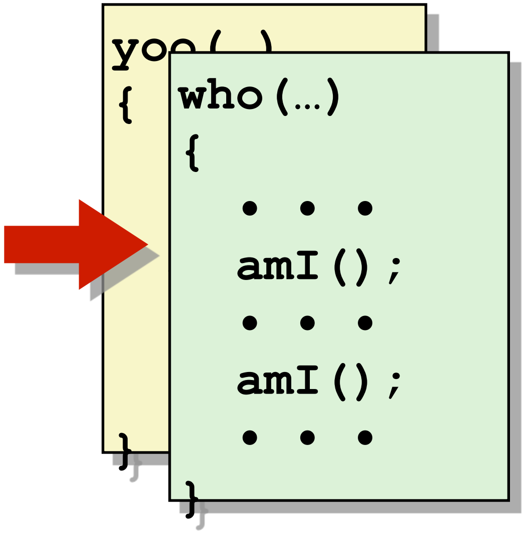
Example

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



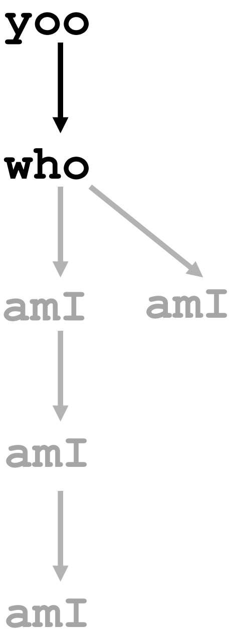


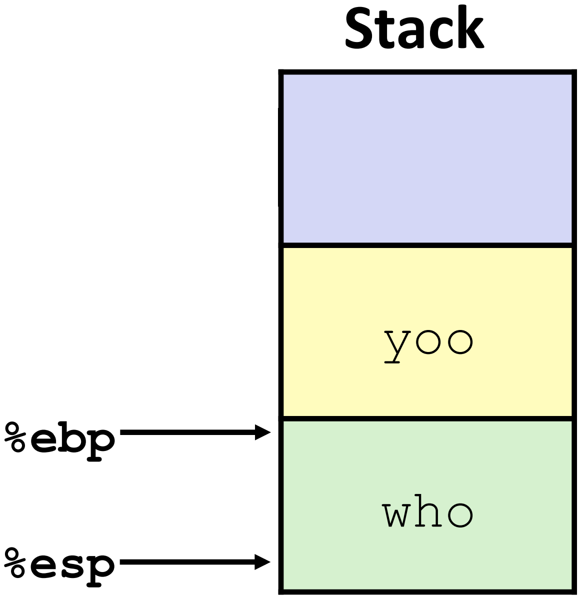
Example



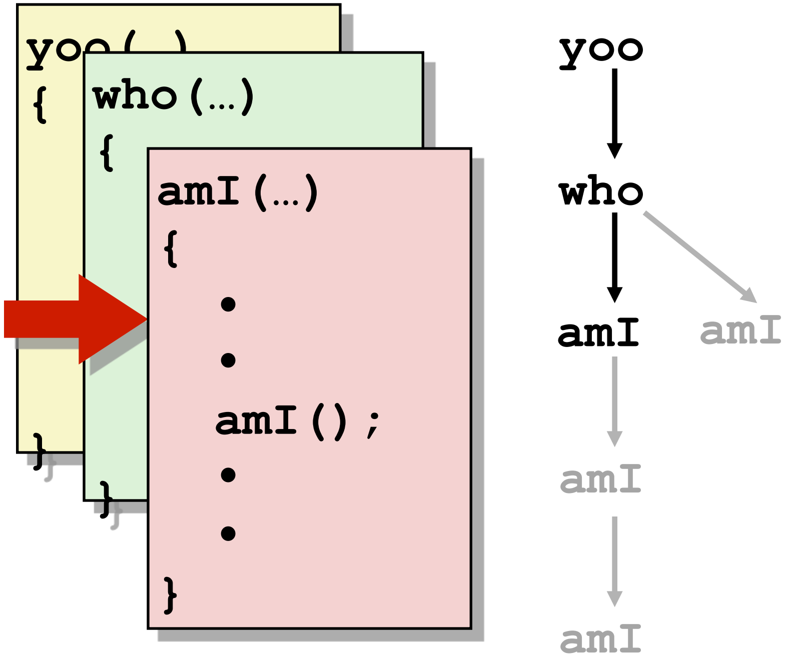
```
yoo(…)
{

}
```

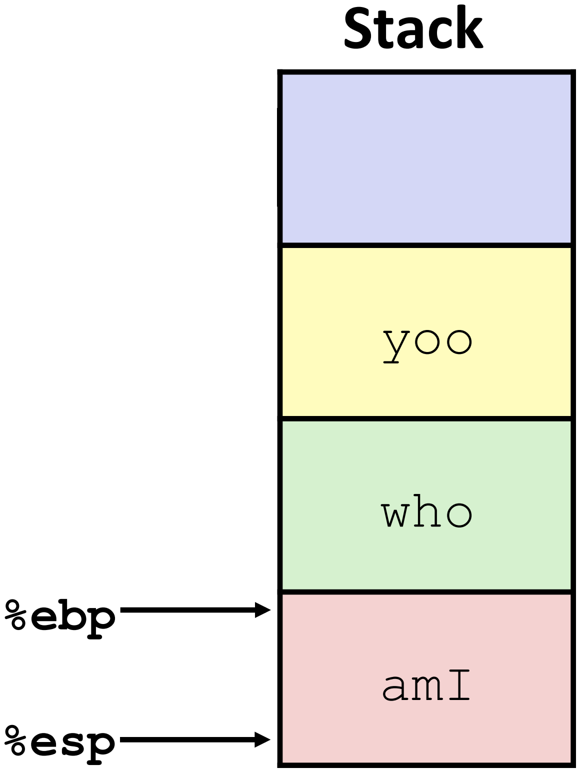
```
who(…)
{
  • • •
  amI();
  • • •
  amI();
  • • •
}
```



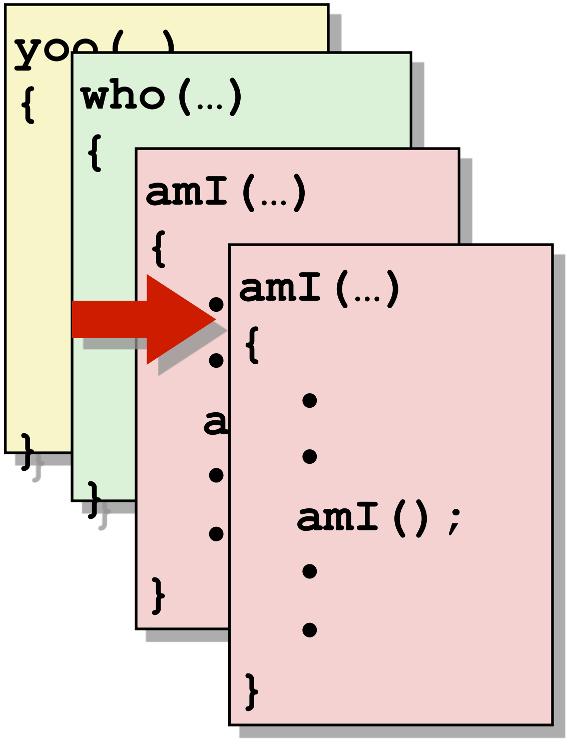


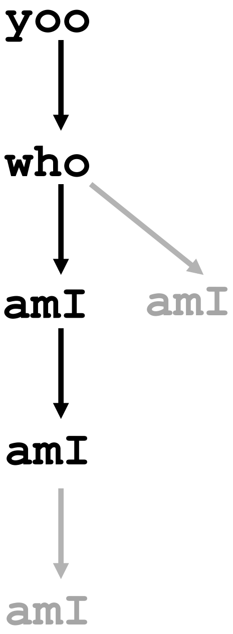
Example

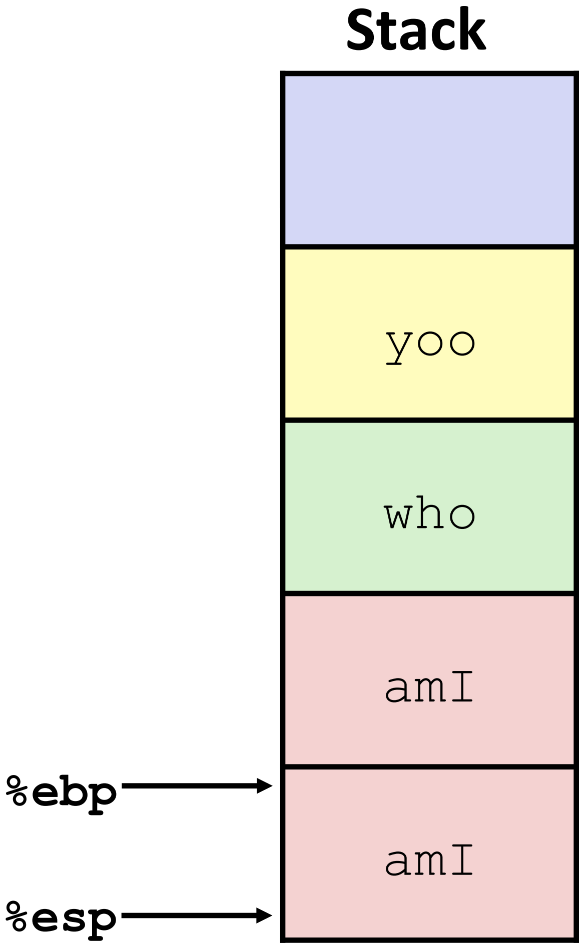

yoo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
amI();
•
•
}
yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%ebp
%esp

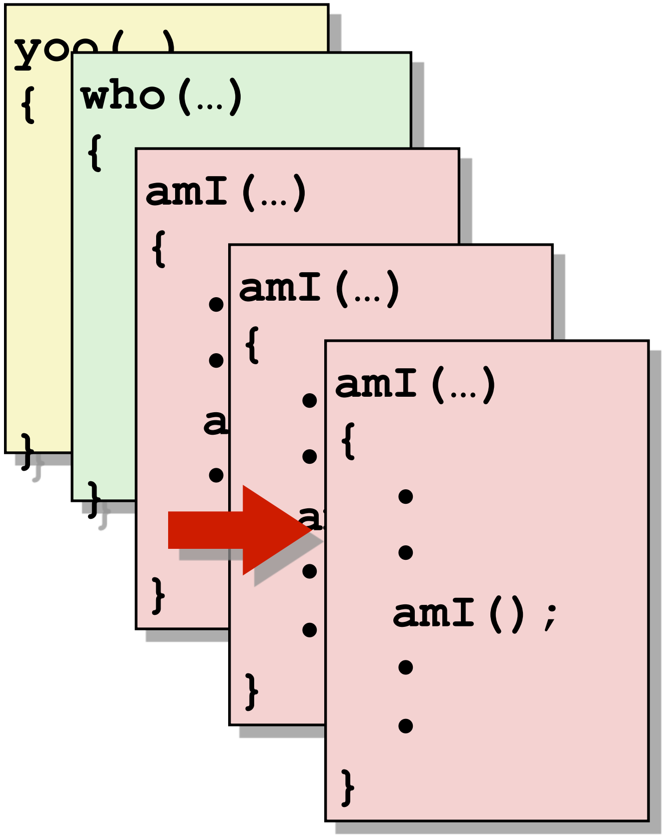
Example

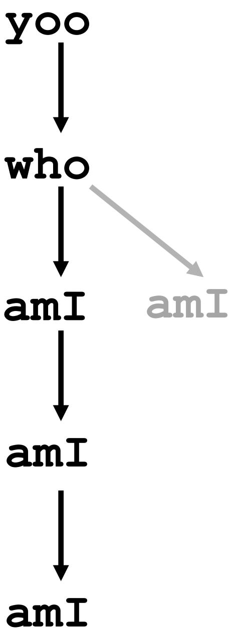

yoo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
a
•
•
}
amI(…)
{
•
•
amI();
•
•
}

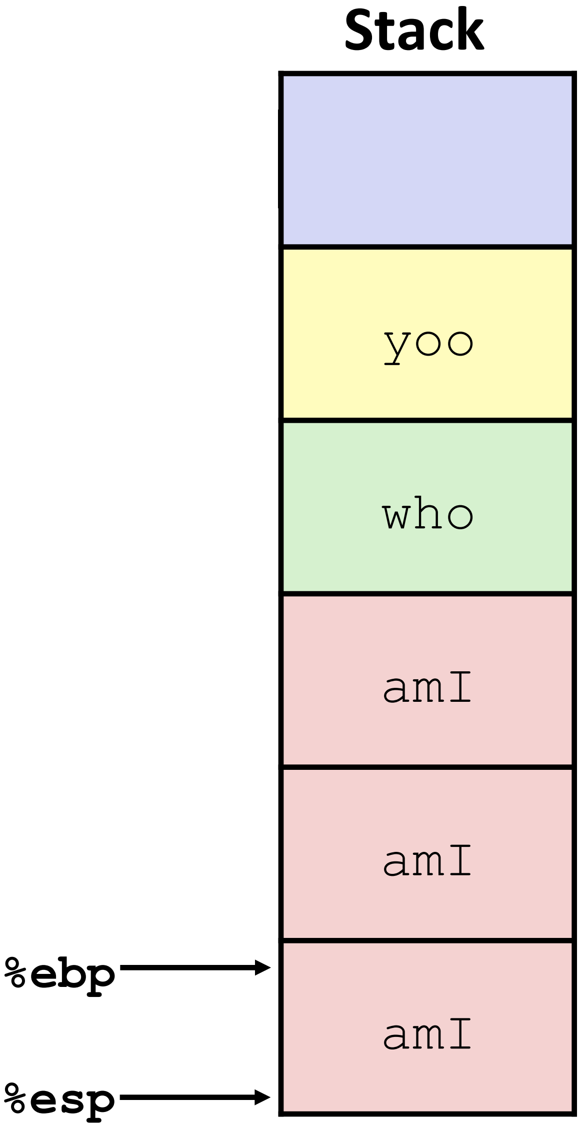

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%ebp
%esp

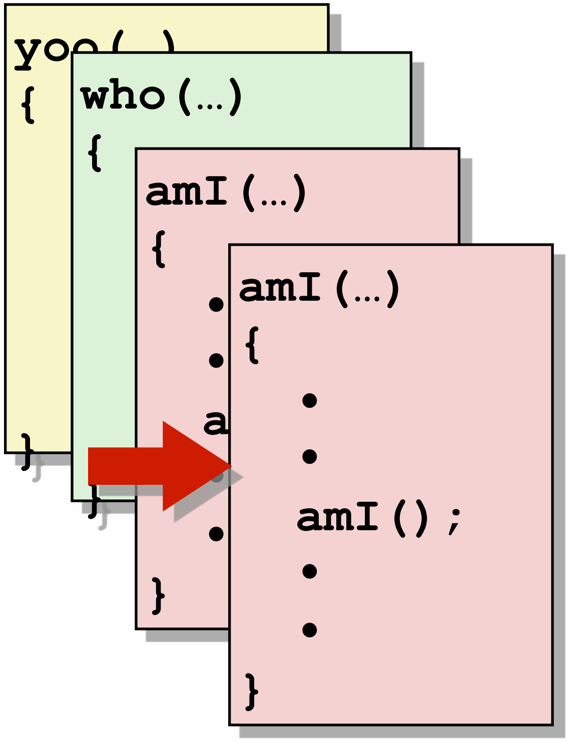
Example

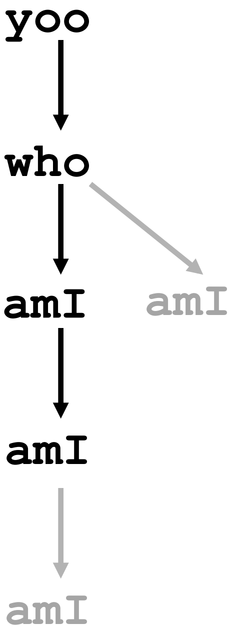

yoo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
a
•
}
amI(…)
{
•
•
a
•
•
}
amI(…)
{
•
•
amI();
•
•
}

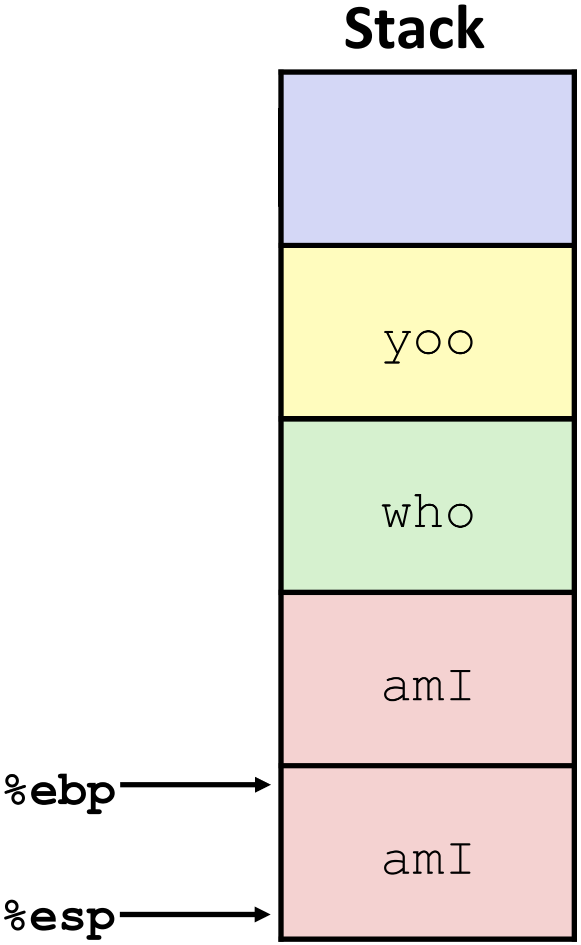

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
amI
%ebp
%esp

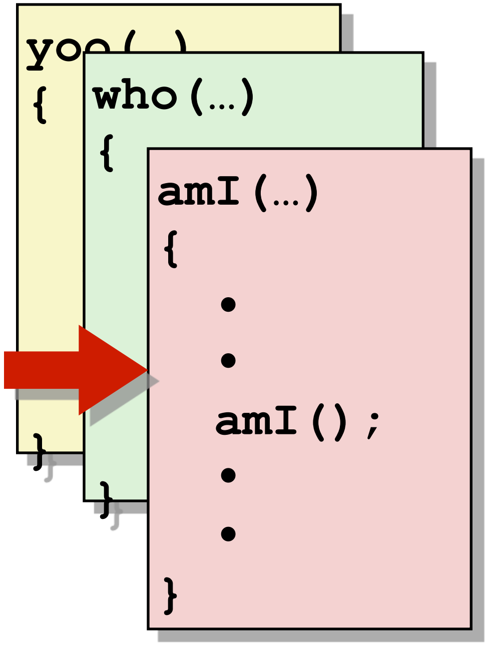
Example

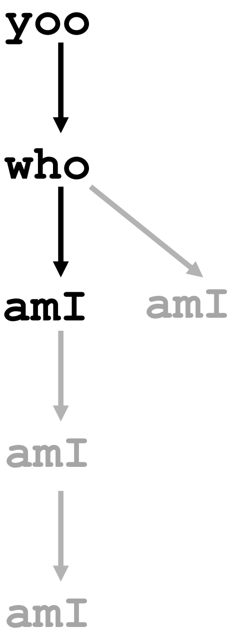

yoo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
a
•
}
amI(…)
{
•
•
amI();
•
•
}


yoo
who
amI
amI
amI
amI

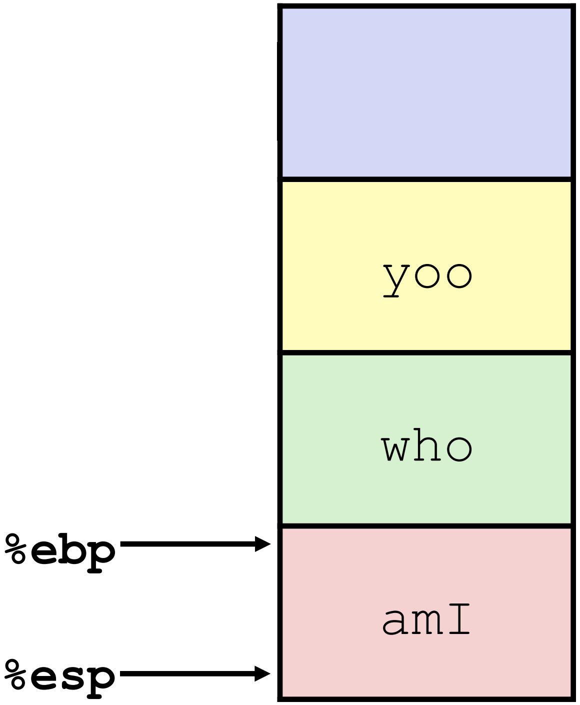

Stack
yoo
who
amI
amI
%ebp
%esp

Example

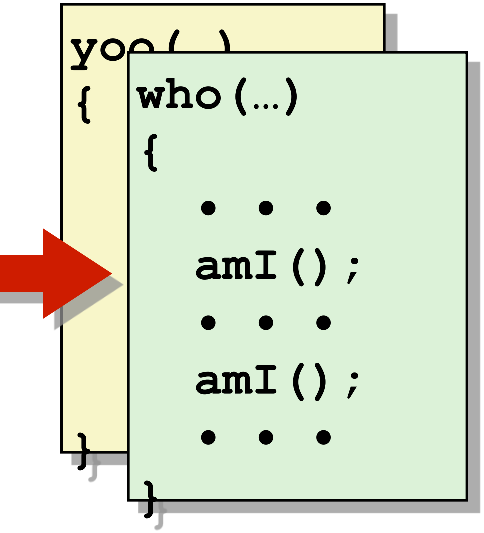




Stack

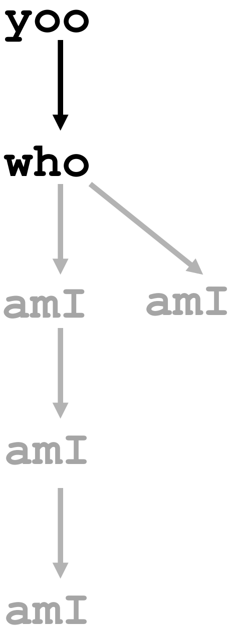


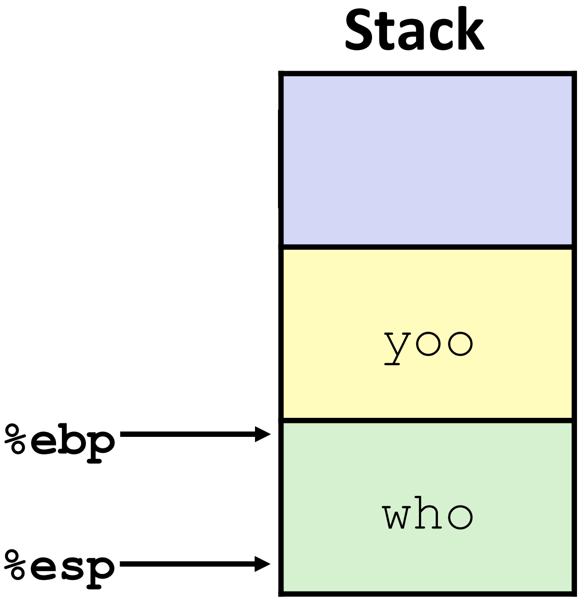
Example



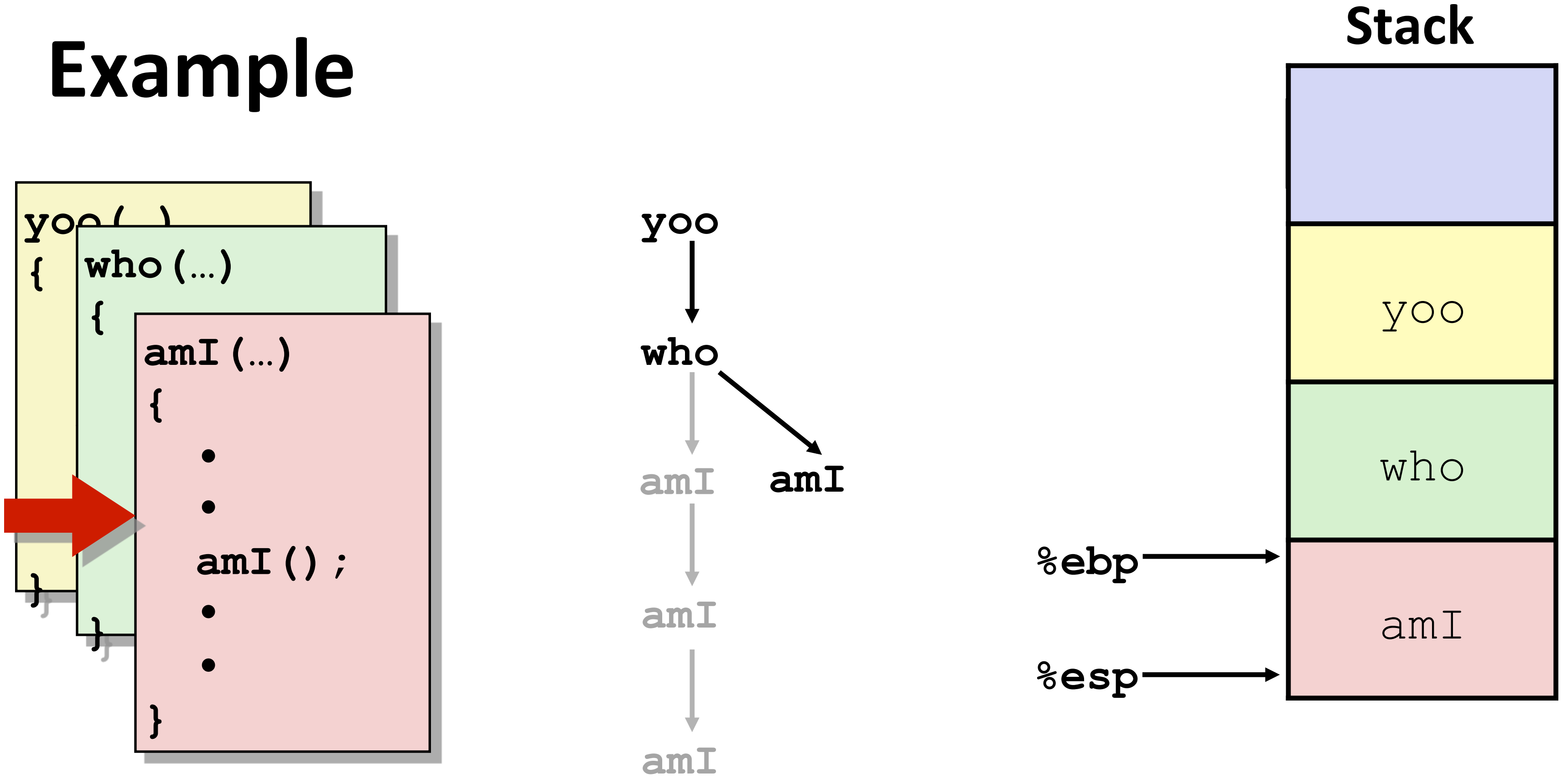
```
yoo(…)
{
  
}
```

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





Example



```
yoo(…)
{
}
```

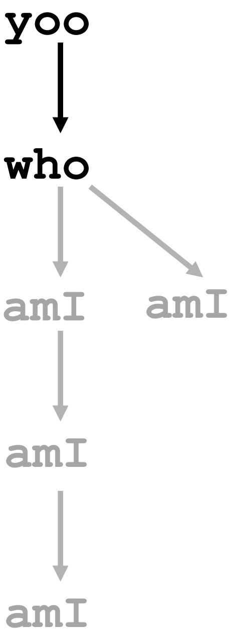
```
who(…)
{
}
```

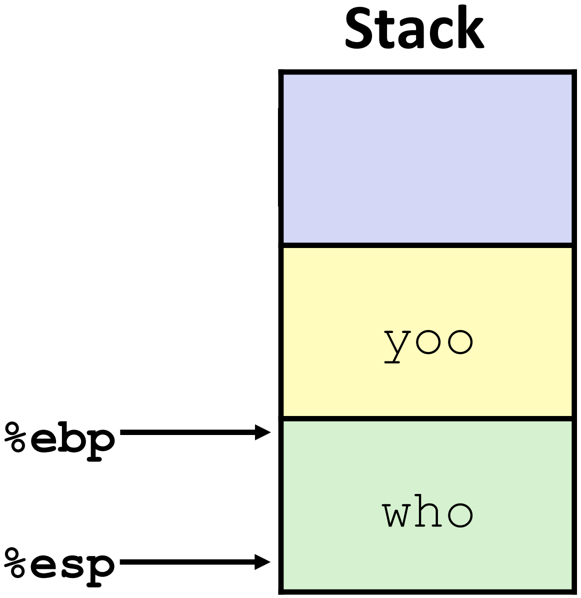
```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

Example

```
yoo(…)
{
}
```

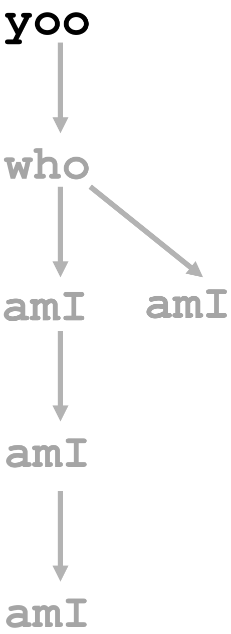
```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

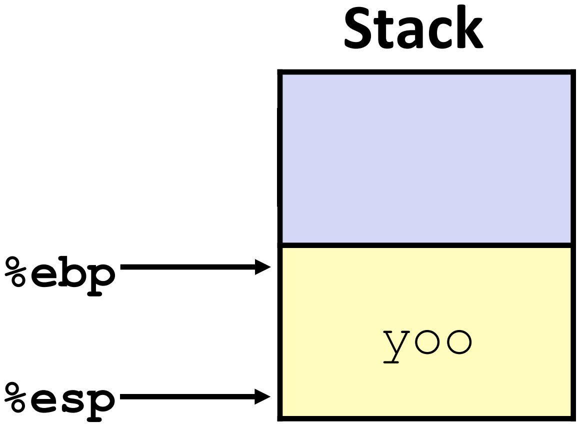




Example

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

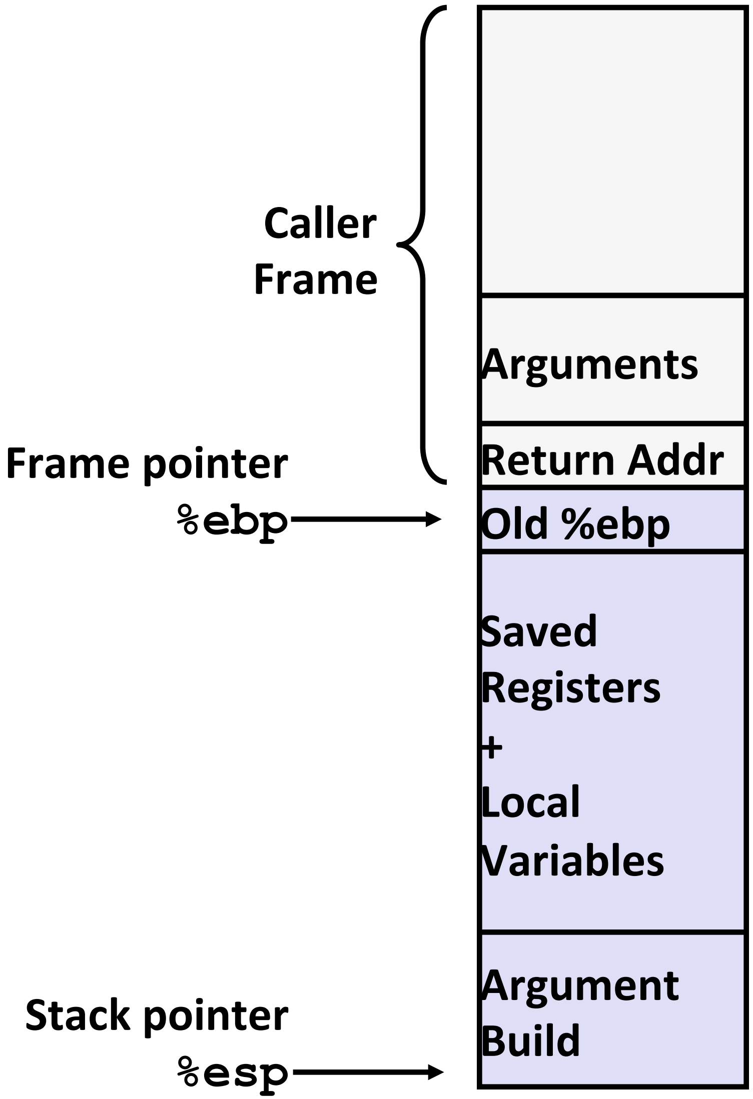




IA32/Linux Stack Frame

- **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

- **Caller Stack Frame**
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call



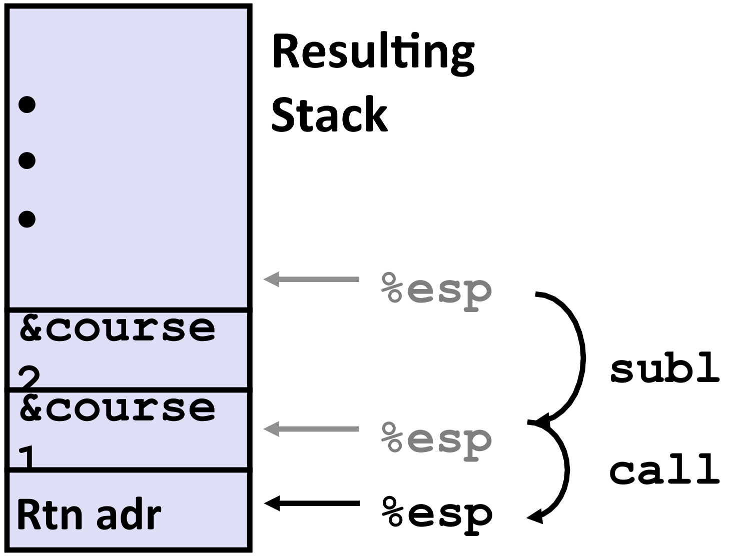
Revisiting swap

```
int course1 = 15213;
int course2 = 18243;

void call_swap() {
  swap(&course1, &course2);
}
```

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

Calling swap from call_swap

```
call_swap:
    • • •
    subl    $8, %esp
    movl    $course2, 4(%esp)
    movl    $course1, (%esp)
    call    swap
    • • •
```



Resulting Stack

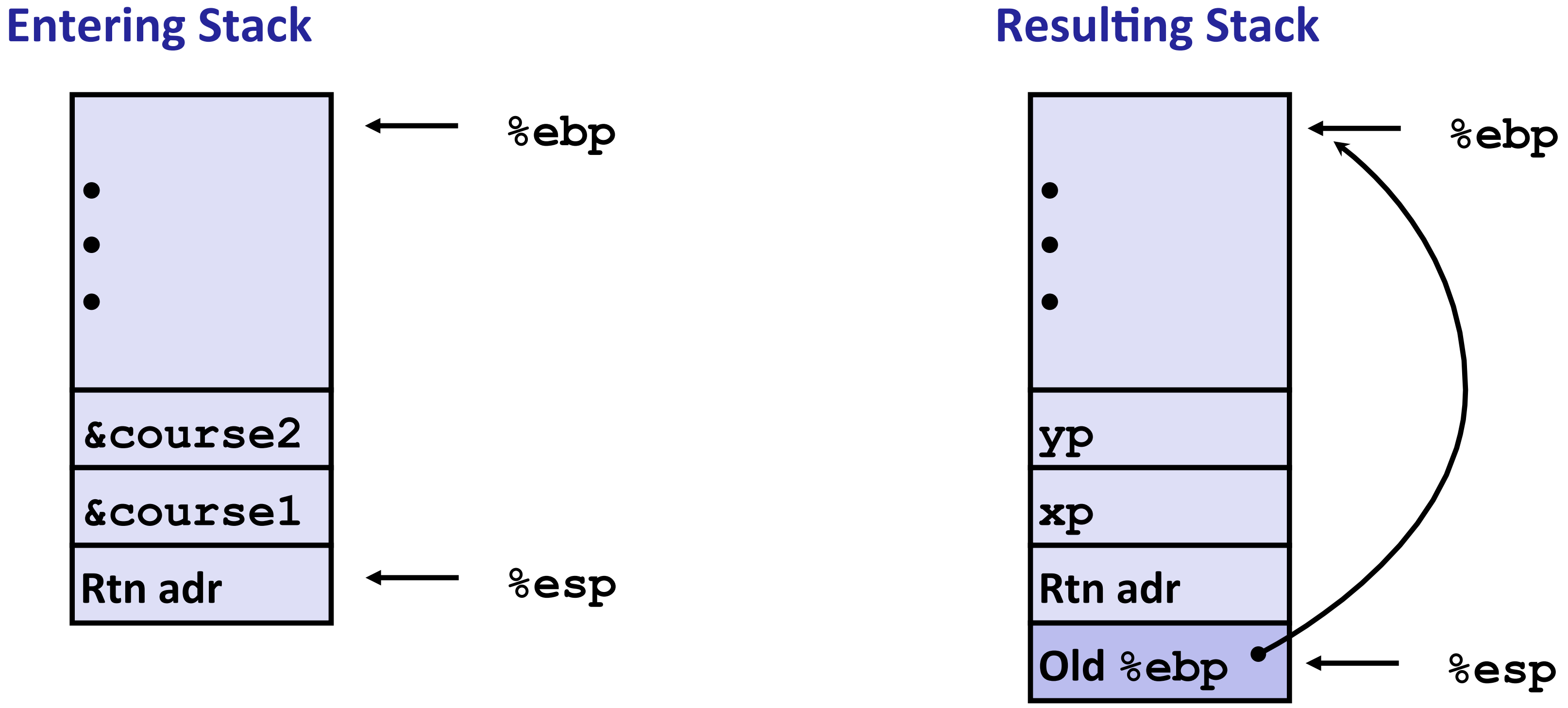
•
•
•
&course
2
&course
1
Rtn adr
%esp
%esp
%esp
subl
call

Revisiting swap

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

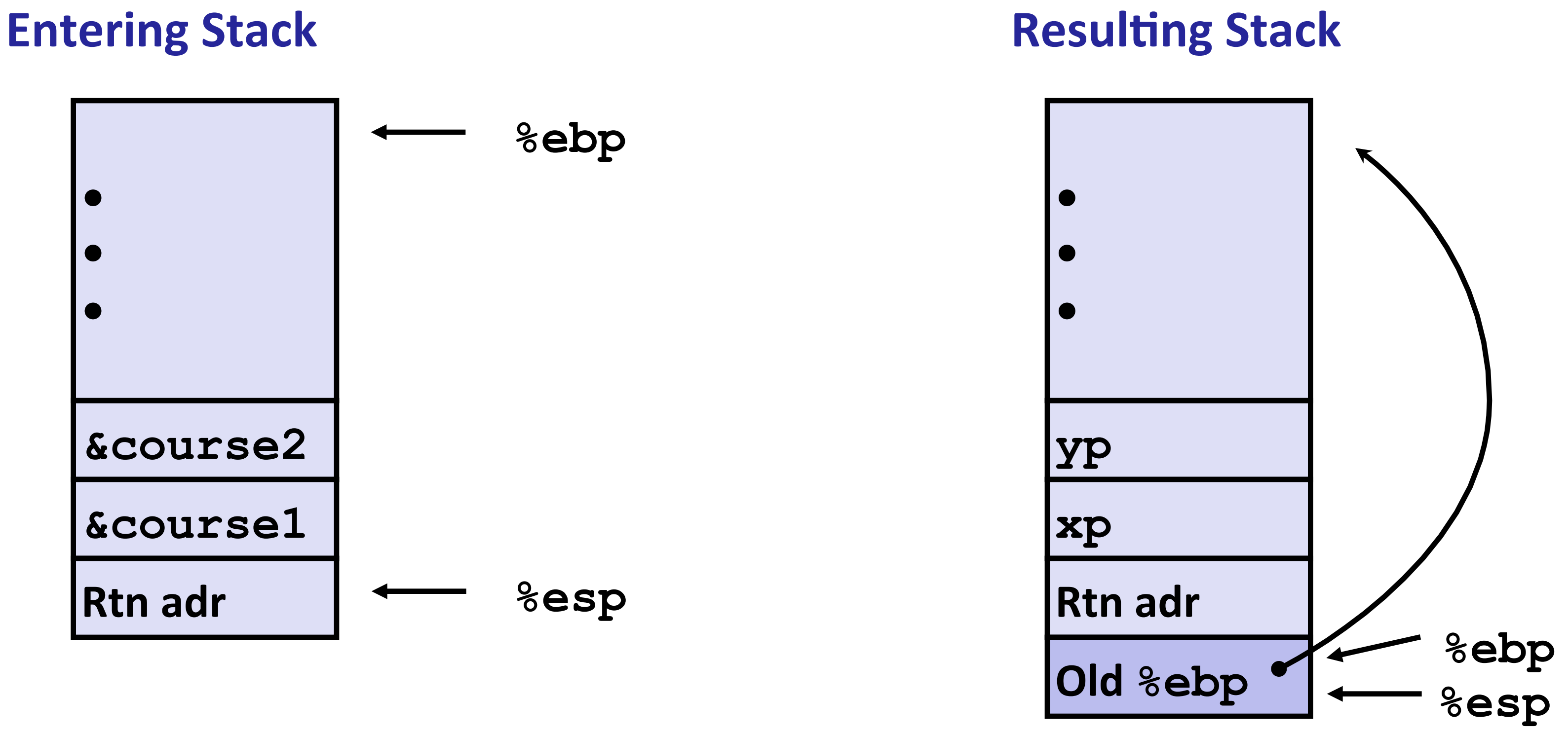
```
swap:
   pushl %ebp                    
   movl  %esp, %ebp               Set Up
   pushl %ebx                    

   movl  8(%ebp), %edx
   movl  12(%ebp), %ecx
   movl  (%edx), %ebx             Body
   movl  (%ecx), %eax
   movl  %eax, (%edx)
   movl  %ebx, (%ecx)

   popl  %ebx
   popl  %ebp                     Finish
   ret
```

swap Setup #1



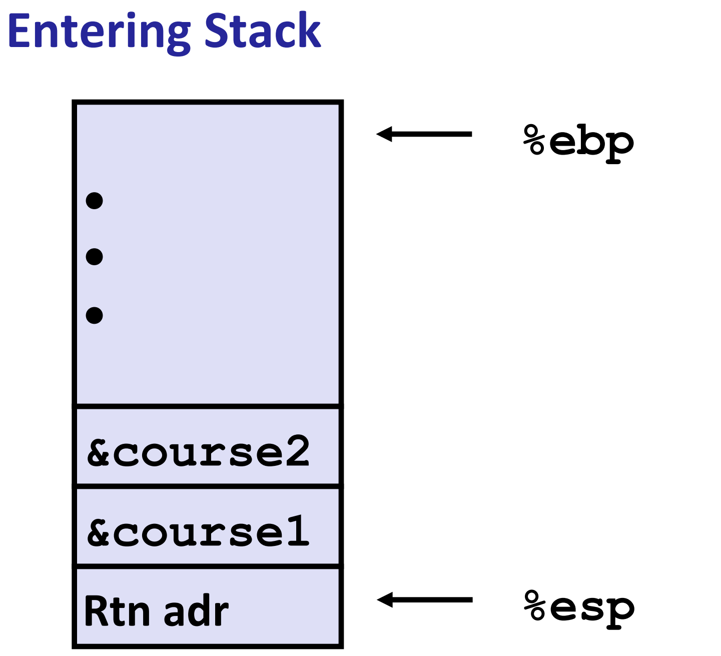
```
swap:
    pushl %ebp
    movl  %esp,%ebp
    pushl %ebx
```

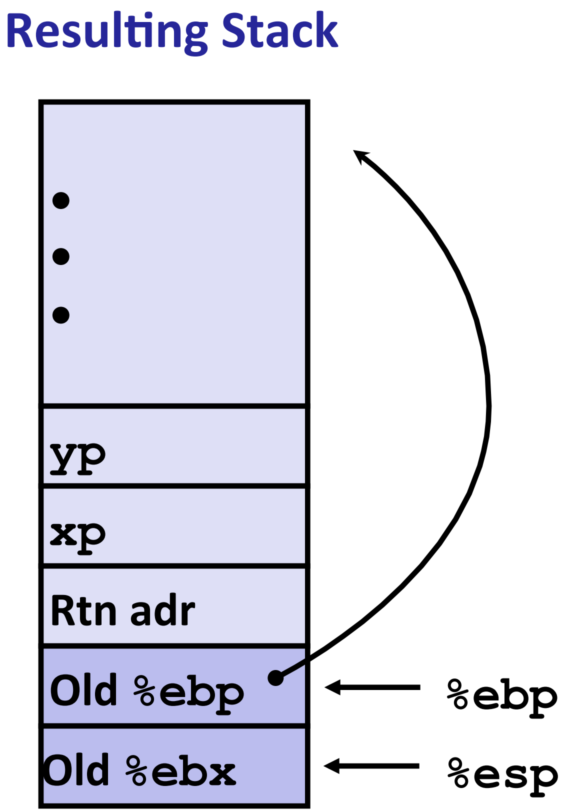
swap Setup #2



```
swap:
    pushl  %ebp
    movl  %esp,%ebp
    pushl  %ebx
```

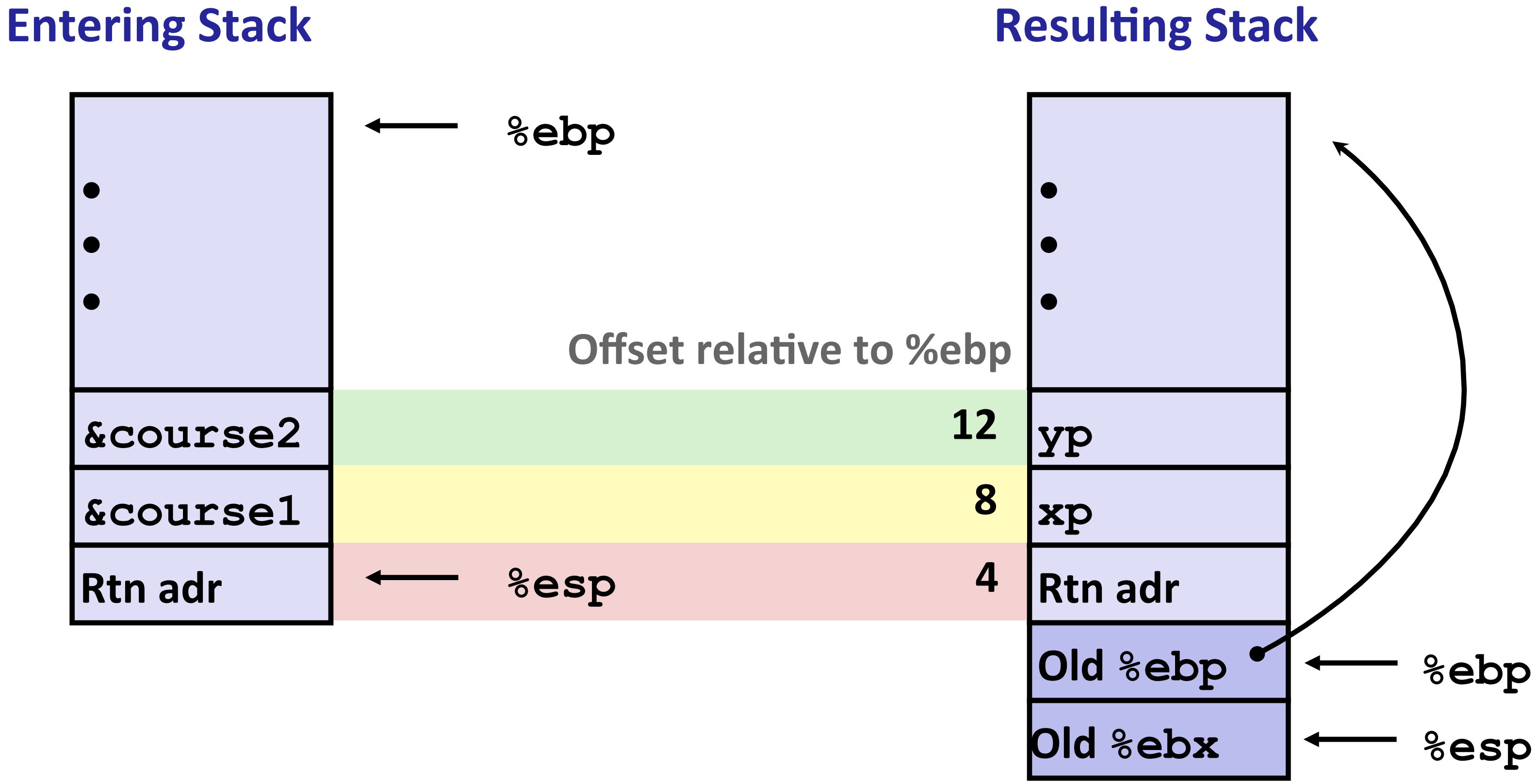
swap Setup #3





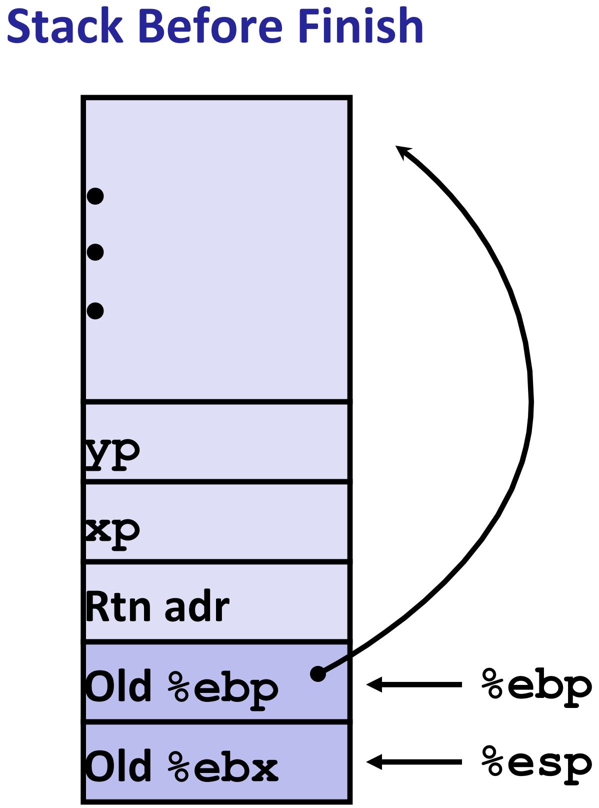
```
swap:
    pushl  %ebp
    movl  %esp,%ebp
    pushl  %ebx
```

swap Body

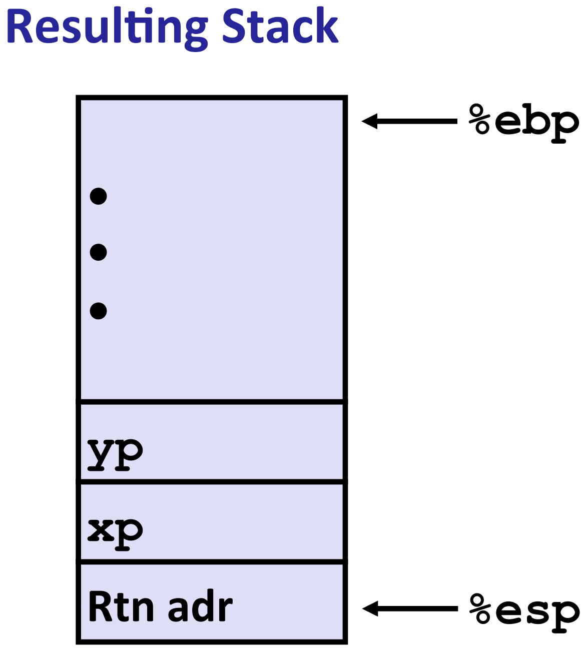


```
movl 8(%ebp),%edx    # get xp
movl 12(%ebp),%ecx   # get yp
. . .
```

swap Finish



```
popl     %ebx
popl     %ebp
```



- Observation
  - Saved and restored register %ebx
  - Not so for %eax, %ecx, %edx

Disassembled swap

```
08048384 <swap>:
 8048384:   55                              push    %ebp
 8048385:   89 e5                           mov     %esp,%ebp
 8048387:   53                              push    %ebx
 8048388:   8b 55 08                        mov     0x8(%ebp),%edx
 804838b:   8b 4d 0c                        mov     0xc(%ebp),%ecx
 804838e:   8b 1a                           mov     (%edx),%ebx
 8048390:   8b 01                           mov     (%ecx),%eax
 8048392:   89 02                           mov     %eax,(%edx)
 8048394:   89 19                           mov     %ebx,(%ecx)
 8048396:   5b                              pop     %ebx
 8048397:   5d                              pop     %ebp
 8048398:   c3                              ret
```

Calling Code

```
 80483b4:   movl    $0x8049658,0x4(%esp) # Copy &course2
 80483bc:   movl    $0x8049654,(%esp)    # Copy &course1
 80483c3:   call    8048384 <swap>       # Call swap
 80483c8:   leave                        # Prepare to return
 80483c9:   ret                          # Return
```

Today

- Switch statements
- **IA 32 Procedures**
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers

Register Saving Conventions

- **When procedure yoo calls who:**
  - yoo is the caller
  - who is the callee

- **Can register be used for temporary storage?**

```
yoo:
    • • •
    movl $15213, %edx
    call who
    addl %edx, %eax
    • • •
    ret
```

```
who:
    • • •
    movl 8(%ebp), %edx
    addl $18243, %edx
    • • •
    ret
```

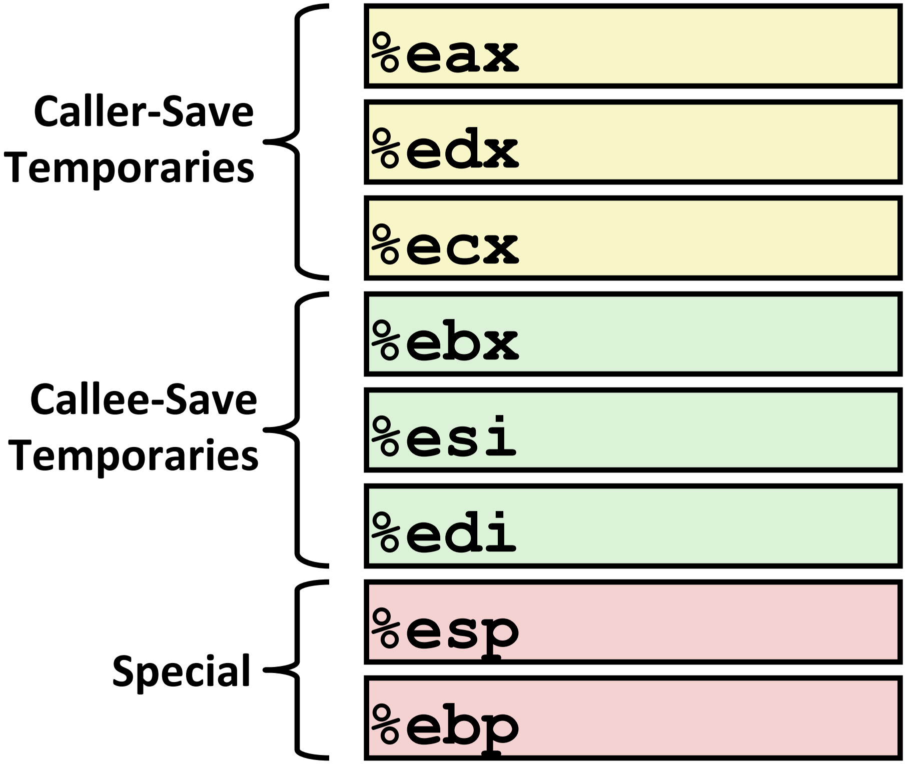
  - Contents of register %edx overwritten by who
  - This could be trouble ➙ something should be done!
    - Need some coordination

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 Save"
    - Caller saves temporary values in its frame before the call
  - "Callee Save"
    - Callee saves temporary values in its frame before using

IA32/Linux+Windows Register Usage

- **`%eax`, `%edx`, `%ecx`**
  - Caller saves prior to call if values are used later
- **`%eax`**
  - also used to return integer value
- **`%ebx`, `%esi`, `%edi`**
  - Callee saves if wants to use them
- **`%esp`, `%ebp`**
  - special form of callee save
  - Restored to original values upon exit from procedure



Today

- **Switch statements**
- **IA 32 Procedures**
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers

Recursive Function

```
/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
     (x & 1) + pcount_r(x >> 1);
}
```

- **Registers**
  - %eax, %edx used without first saving
  - %ebx used, but saved at beginning & restored at end

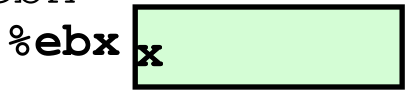
```
pcount_r:
    pushl  %ebp
    movl   %esp, %ebp
    pushl  %ebx
    subl   $4, %esp
    movl   8(%ebp), %ebx
    movl   $0, %eax
    testl  %ebx, %ebx
    je   .L3
    movl   %ebx, %eax
    shrl   %eax
    movl   %eax, (%esp)
    call   pcount_r
    movl   %ebx, %edx
    andl   $1, %edx
    leal   (%edx,%eax), %eax
.L3:
    addl   $4, %esp
    popl   %ebx
    popl   %ebp
    ret
```

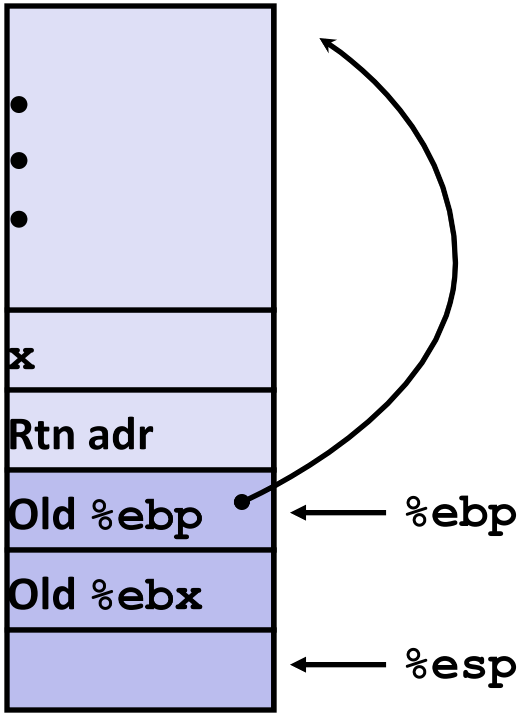
Recursive Call #1

```
/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}
```

```
pcount_r:
    pushl %ebp
    movl  %esp, %ebp
    pushl %ebx
    subl  $4, %esp
    movl  8(%ebp), %ebx
     • • •
```

- **Actions**
  - Save old value of `%ebx` on stack
  - Allocate space for argument to recursive call
  - Store x in `%ebx`



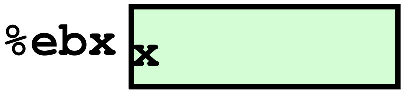


Recursive Call #2

```
/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}
```

```
    • • •
   movl   $0, %eax
   testl %ebx, %ebx
   je   .L3
    • • •
.L3:
    • • •
   ret
```

- **Actions**
  - If x == 0, return
    - with `%eax` set to 0



%ebx x

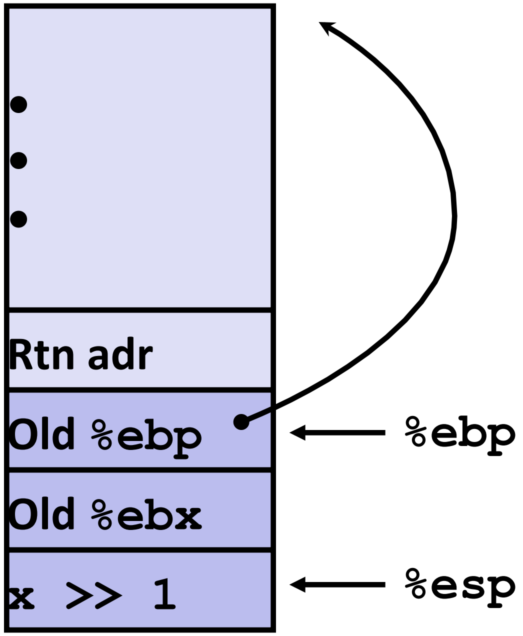
Recursive Call #3

```
/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}
```

```
 • • •
movl   %ebx, %eax
shrl   %eax
movl   %eax, (%esp)
call   pcount_r
 • • •
```

- **Actions**
  - Store x >> 1 on stack
  - Make recursive call
- **Effect**
  - %eax set to function result
  - %ebx still has value of x

%ebx x

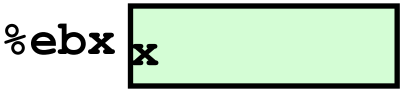

•
•
•
Rtn adr
Old %ebp
Old %ebx
x >> 1
%ebp
%esp

Recursive Call #4

```
/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}
```

```
  • • •
movl    %ebx, %edx
andl    $1, %edx
leal   (%edx,%eax), %eax
  • • •
```

- **Assume**
  - %eax holds value from recursive call
  - %ebx holds x
- **Actions**
  - Compute (x & 1) + computed value
- **Effect**
  - %eax set to function result

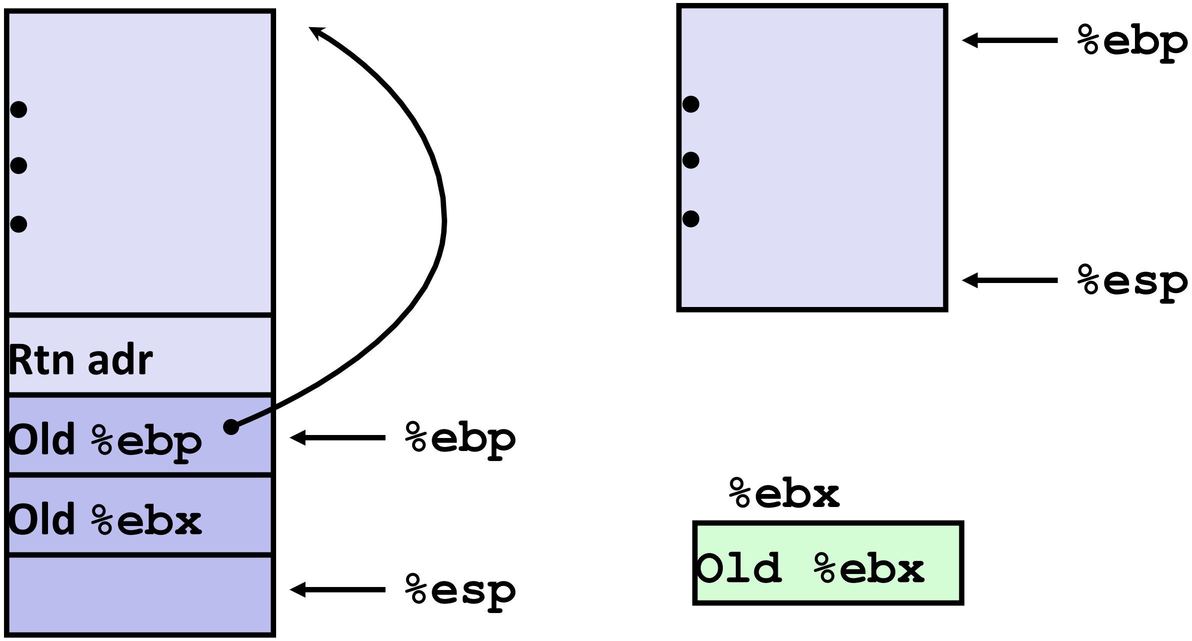


Recursive Call #5

```
/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}
```

```
      • • •
L3:
      addl$4, %esp
      popl%ebx
      popl%ebp
      ret
```

- **Actions**
  - Restore values of %ebx and %ebp
  - Restore %esp



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
  - 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

Pointer Code

Generating Pointer

```
/* Compute x + 3 */
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
```

Referencing Pointer

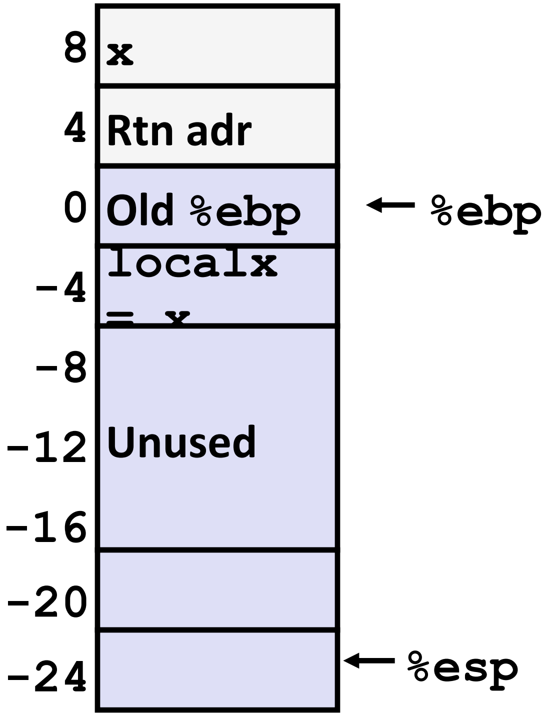
```
/* Increment value by k */
void incrk(int *ip, int k) {
  *ip += k;
}
```

- `add3` creates pointer and passes it to `incrk`

Creating and Initializing Local Variable

```
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
```

- **Variable localx must be stored on stack**
  - Because: Need to create pointer to it
- **Compute pointer as -4(%ebp)**



First part of add3

```
add3:
  pushl%ebp
  movl %esp, %ebp
  subl $24, %esp      # Alloc. 24 bytes
  movl 8(%ebp), %eax
  movl %eax, -4(%ebp)# Set localx to x
```

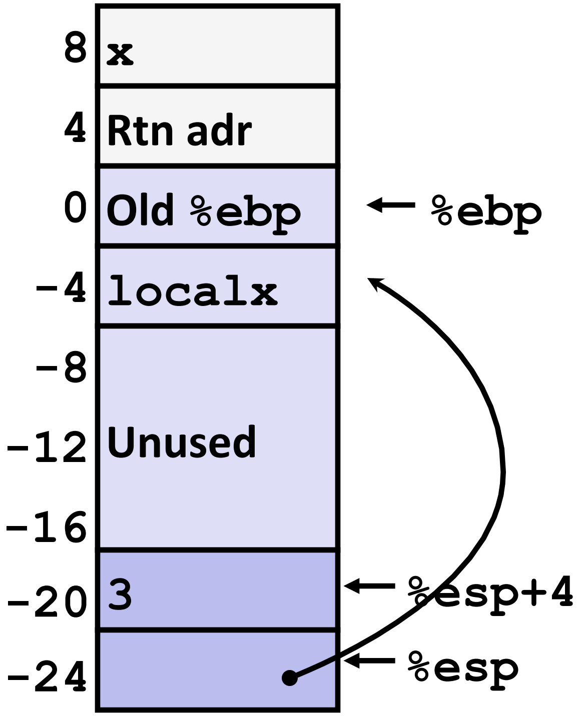
Creating Pointer as Argument

```
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
```

- Use leal instruction to compute address of localx

Middle part of add3

```
  movl $3, 4(%esp)    # 2nd arg = 3
  leal -4(%ebp), %eax# &localx
  movl %eax, (%esp)   # 1st arg = &localx
  call incrk
```



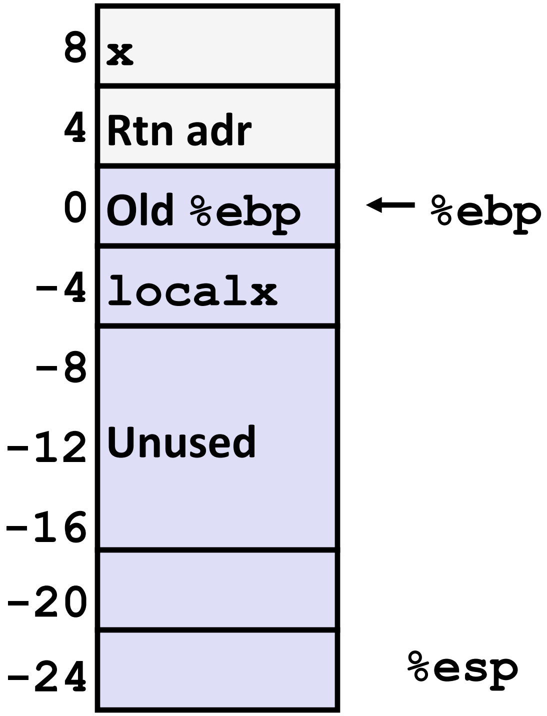
Retrieving local variable

```
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
```

- **Retrieve localx from stack as return value**

Final part of add3

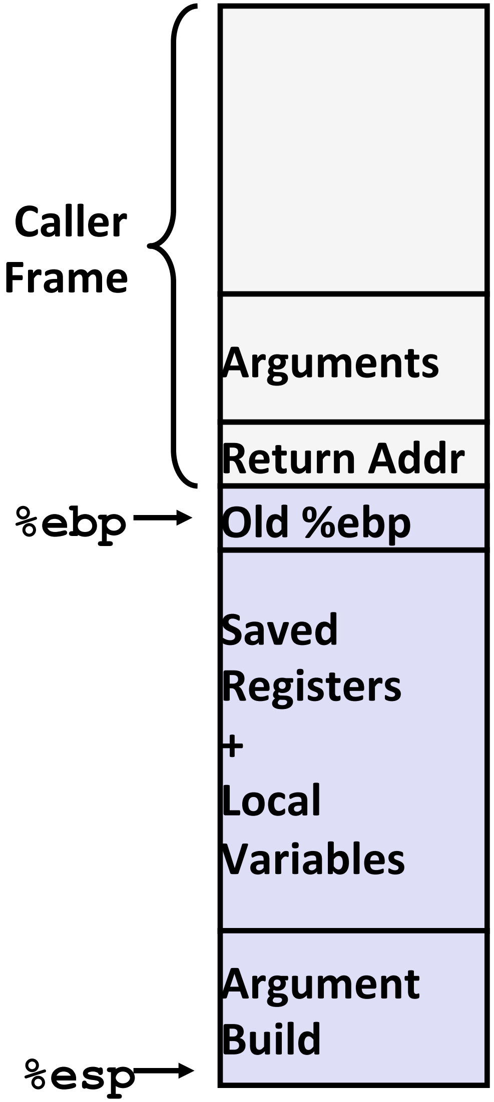
```
  movl -4(%ebp), %eax # Return val= localx
  leave
  ret
```



8	x	
4	Rtn adr	
0	Old %ebp	← %ebp
-4	localx	
-8	Unused	
-12		
-16		
-20		
-24		%esp

IA 32 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 `%eax`
- **`Pointers are addresses of values`**
  - `On stack or global`


Caller Frame
Arguments
Return Addr
%ebp
Old %ebp
Saved Registers + Local Variables
Argument Build
%esp