Subprograms: Local Variables

ICS312

Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

Local Variables in Subprograms

- In all the examples we have seen so far, the subprograms were able to do their work using only registers
- But sometimes, a subprogram's needs are beyond the set of available registers and some data must be kept in memory
  - Just think of all subprograms you wrote that used more than 6 local variables (EAX, EBX, ECX, EDX, ESI, EDI)
- One possibility could be to declare a small .bss segment for each subprogram, to reserve memory space for all local variables
- Drawback #1: memory waste
  - This reserved memory consumes memory space for the entire duration of the execution even if the subprogram is only active for a tiny fraction of the execution time (or never!)
- Drawback #2: subprograms are not reentrant

Re-entrant subprogram

- A subprogram is active if it has been called but the RET instruction hasn't been executed yet
- A subprogram is reentrant if it can be called from anywhere in the program
- This implies that the program can call itself, directly or indirectly, which enables recursion
  - e.g., f calls g, which calls h, which calls f
- At a given time, two or more instances of a subprogram can be active
  - Two or more activation records for this subprogram on the stack
- If we store the local variables of a subprogram in the .bss segment, then there can only be one activation!
  - Otherwise activation #2 could corrupt the local variables of activation #1
  - In other words, multiple activations would share the same play pen
- Therefore, programs would not be reentrant and one cannot have recursive calls when subprograms have local variables!
  - In the previous set of lecture notes, the recursive program had no local variables, so we were "lucky"
- Having reentrant programs is so useful that we **must** have it

Local variables on the stack

- Since activation records on the stack are used to store relevant information pertaining to a subprogram, why not use them for storing the subprogram local variables?
- The standard approach is to store local variables right after the saved EBP value on the stack
  - This is simply done by subtracting some amount to the ESP pointer
- The local variables are then accessed as [EBP - 4], [EBP - 8], etc.
- Let's see this on an example

Local Variables Example

- Say we have a subprogram that takes 2 parameters, uses 3 local variables, and doesn't return any value
- The code of the subprogram is as follows:

```
func:
    push    ebp             ; save old EBP value
    mov     ebp, esp        ; set EBP
    sub     esp, 12         ; add space for 3 local variables
    ; subprogram body
    mov     esp, ebp        ; deallocate local variables
                            ; (could also be "add esp, 12")
    pop     ebp             ; restore old EBP value
    ret
```

- Let's look at the stack when the subprogram body begins

Local Variables Example

- Inside the body of the subprogram, parameters are referenced as:
  - [EBP+8]: 1st parameter
  - [EBP+12]: 2nd parameter
- and local variables are referenced as:
  - [EBP-4]: 1st local variable
  - [EBP-8]: 2nd local variable
  - [EBP-12]: 3rd local variable

EBP+12	2nd parameter
EBP+8	1st parameter
EBP+4	return address
EBP	saved EBP
EBP-4	1st local var
EBP-8	2nd local var
EBP-12	3rd local var
	(saved registers)

Local Variables Example

- Inside the body of the subprogram, parameters are referenced as:
  - [EBP+8]: 1st parameter
  - [EBP+12]: 2nd parameter
- and local variables are referenced as:
  - [EBP-4]: 1st local variable
  - [EBP-8]: 2nd local variable
  - [EBP-12]: 3rd local variable

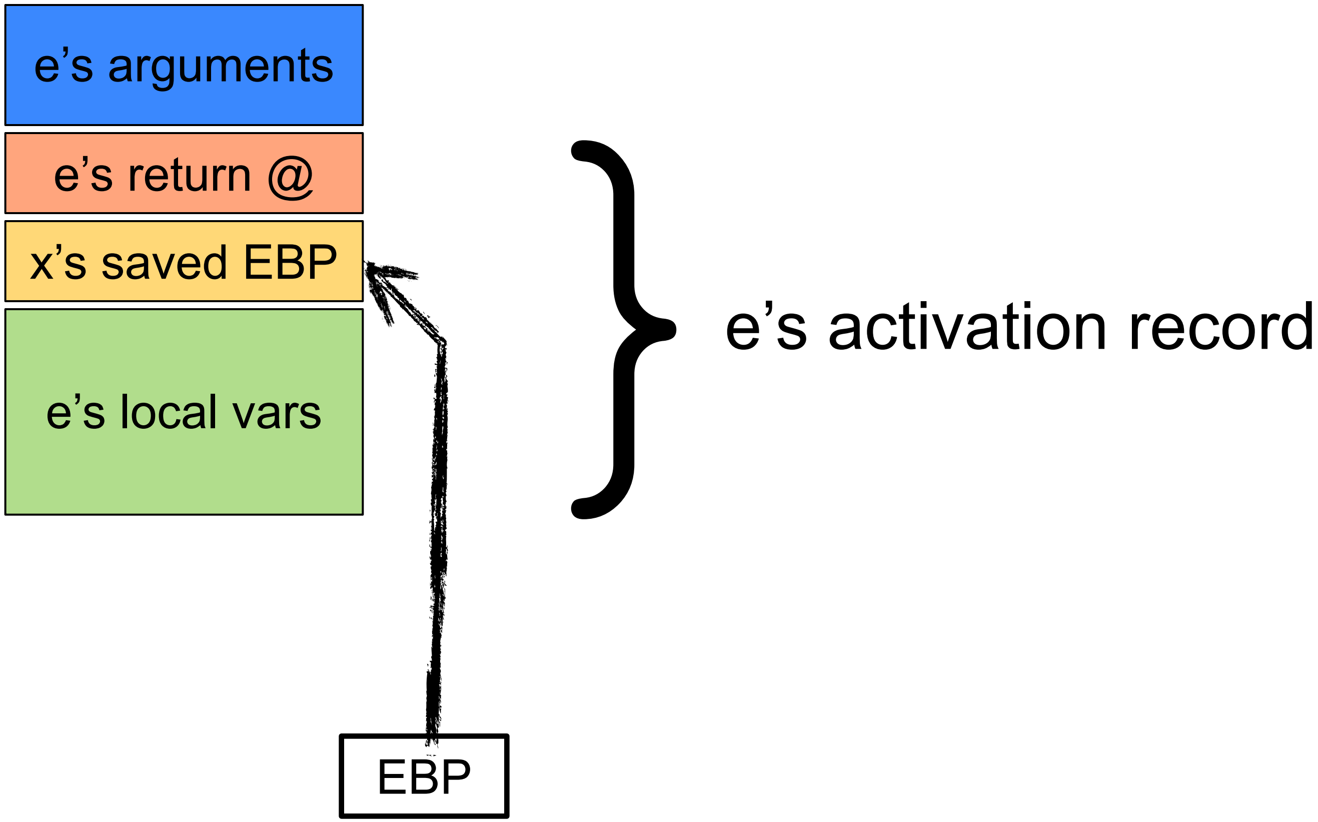
EBP+12	2nd parameter
EBP+8	1st parameter
EBP+4	return address
EBP	saved EBP
EBP-4	1st local var
EBP-8	2nd local var
EBP-12	3rd local var
	(saved registers)

Very important you have this picture in mind; you should be able to redraw it

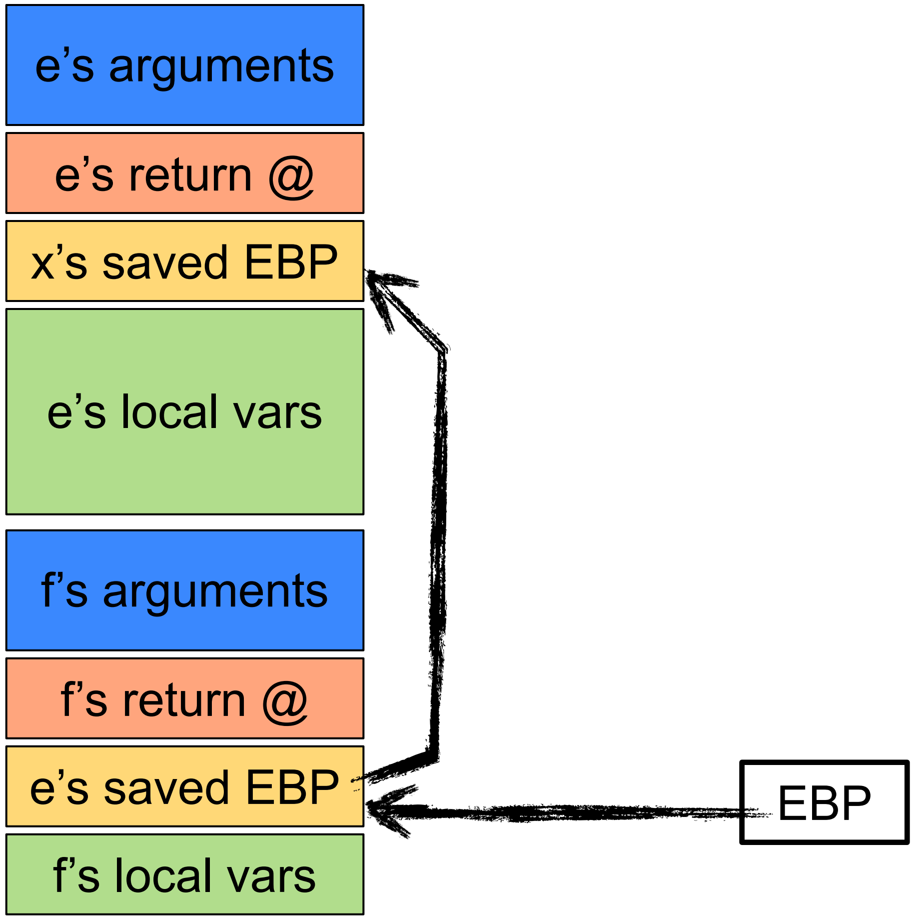
A “deep” stack

- Each call to a subprogram puts an activation record on the stack, saved EBP values and arguments
- Important: While a function is active, EBP always points to the saved EBP value saved for the function’s caller
  - EBP is the anchor point of the activation record (“B” stands for **Base** Pointer)
- We have seen this on a small example in the previous set of lecture notes
- Let’s look at a bigger example
  - But not with the corresponding assembly code
  - And not showing the “saved registers to avoid destruction of their values” on the stack

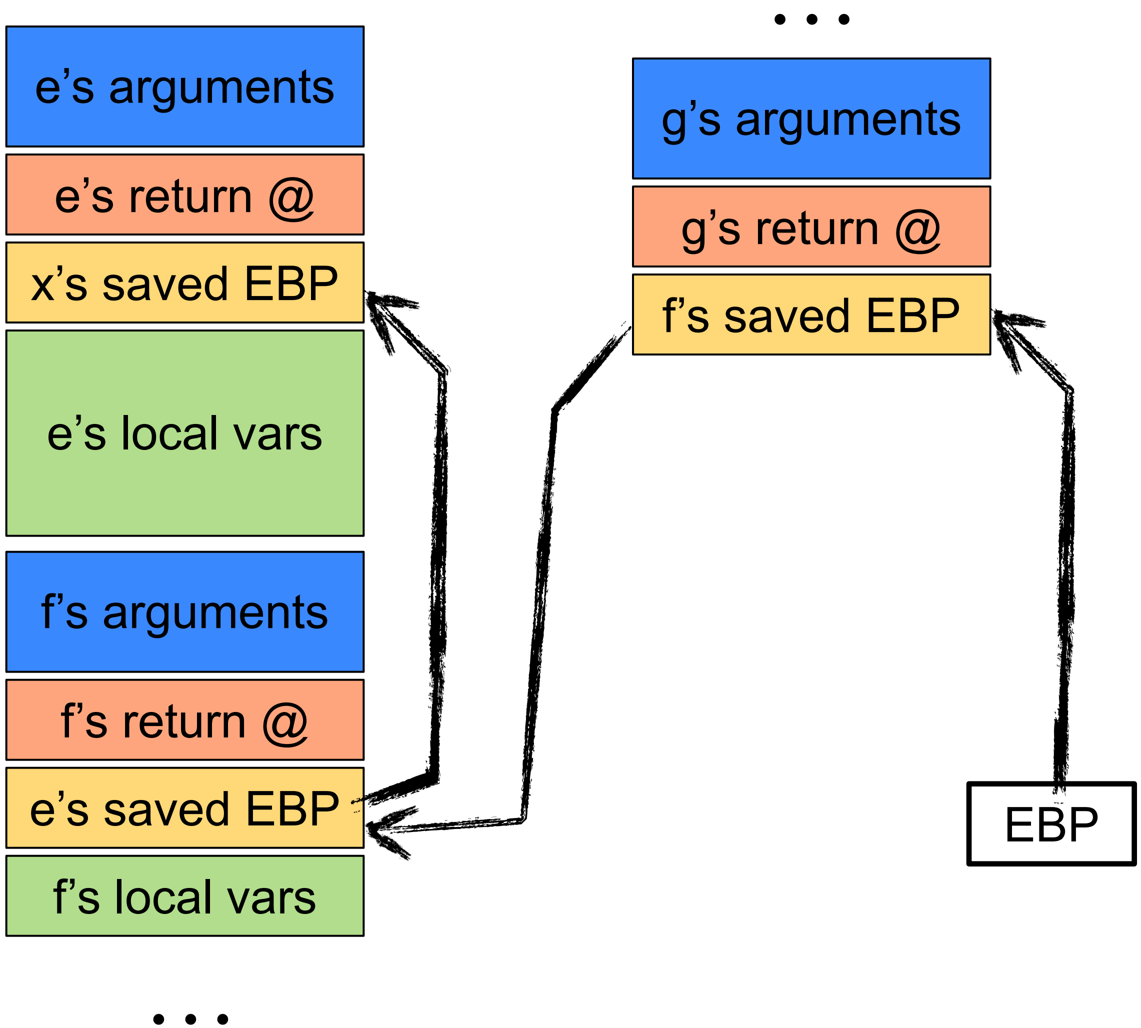
x() calls e() calls f() calls g() calls h()



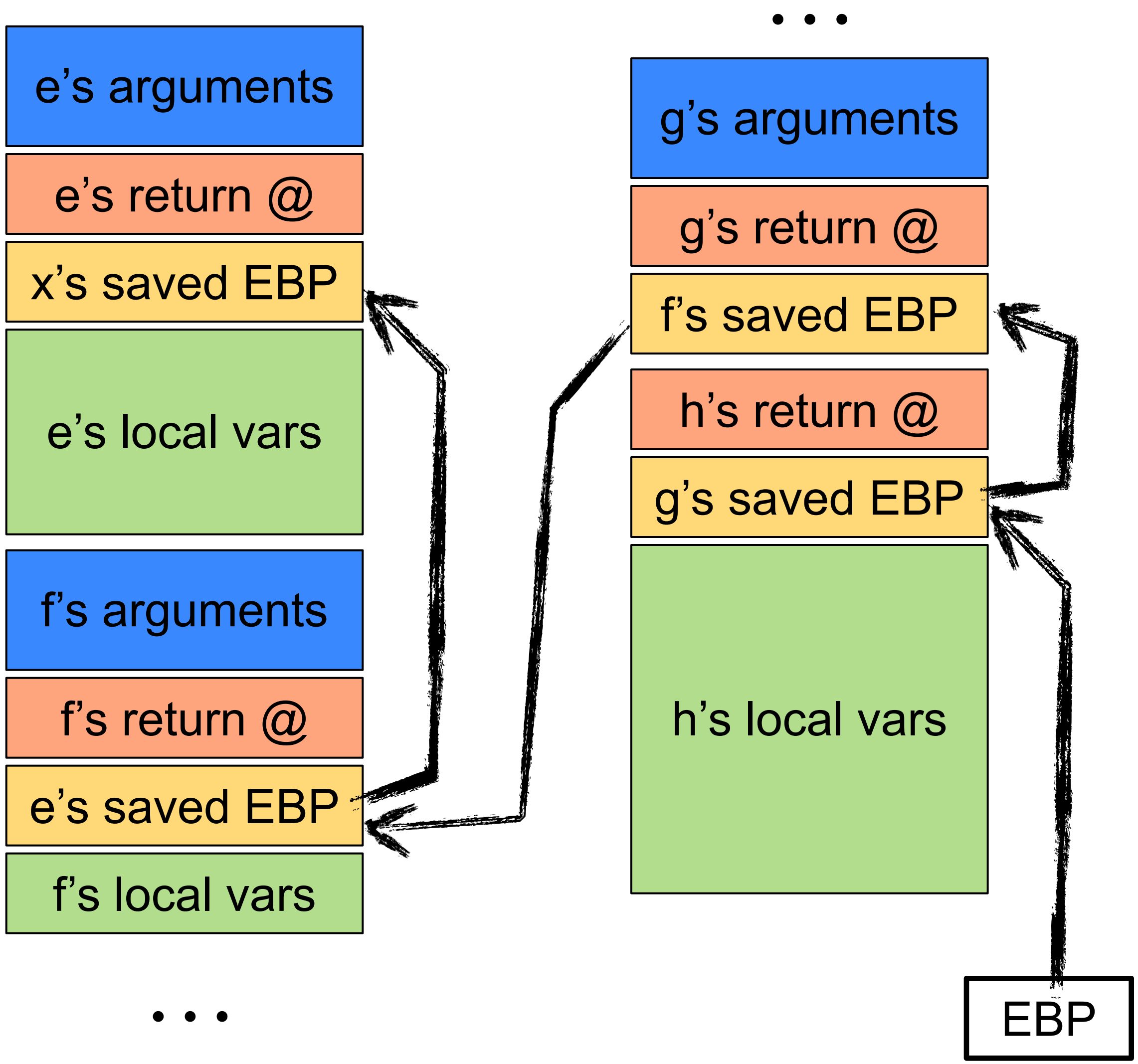
x() calls e() calls f() calls g() calls h()



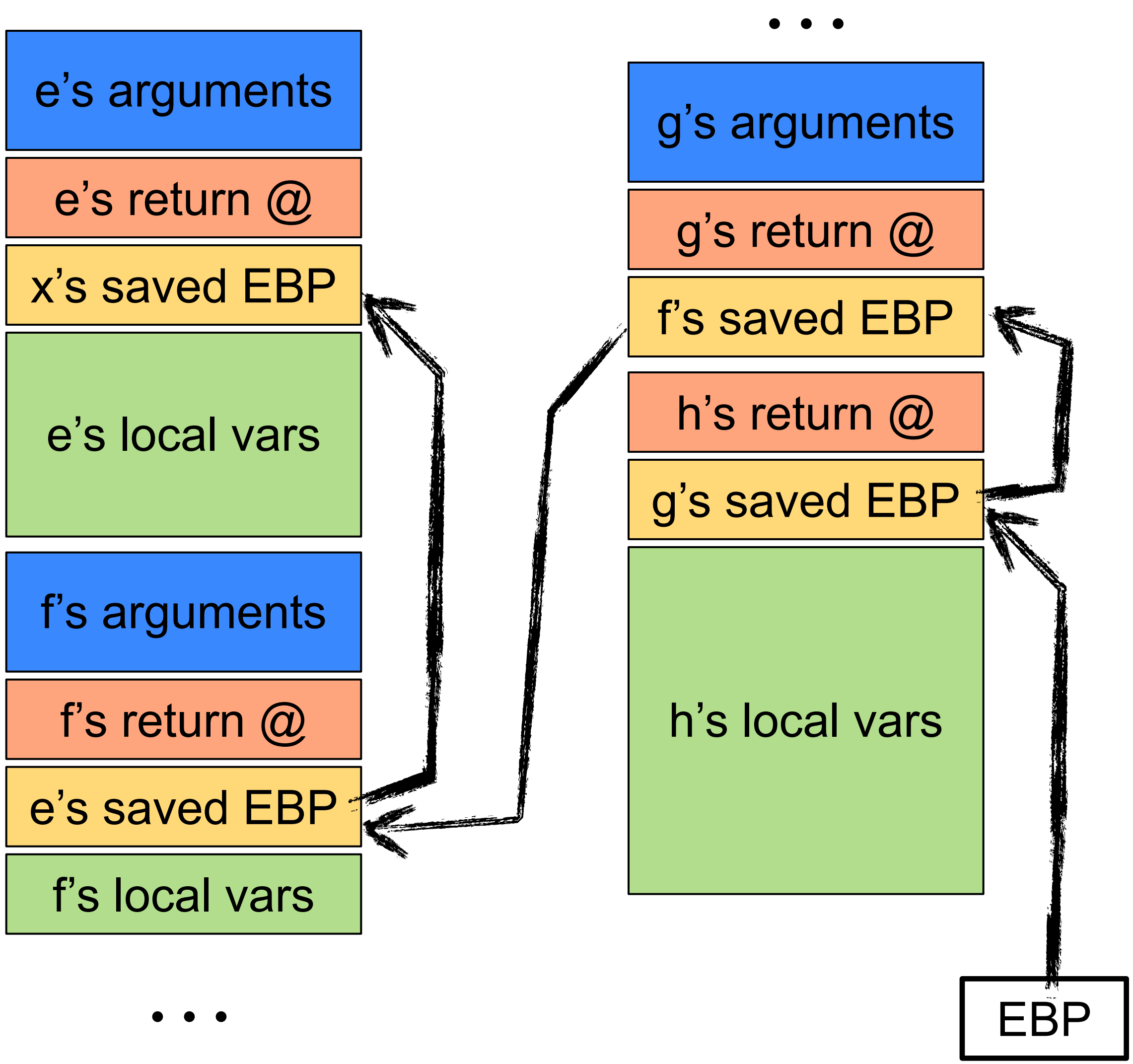
x() calls e() calls f() calls g() calls h()



x() calls e() calls f() calls g() calls h()

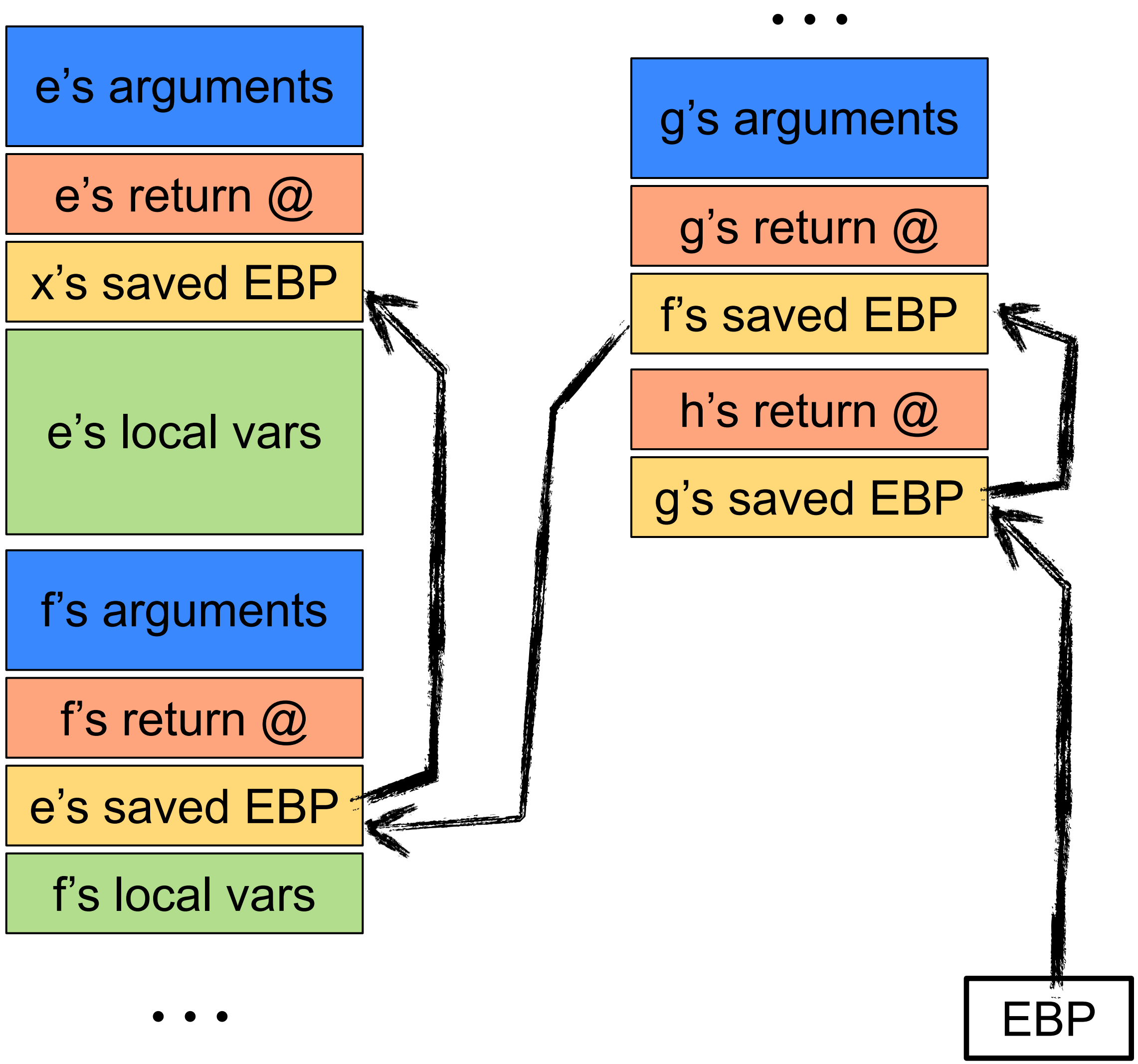


x() calls e() calls f() calls g() calls h()



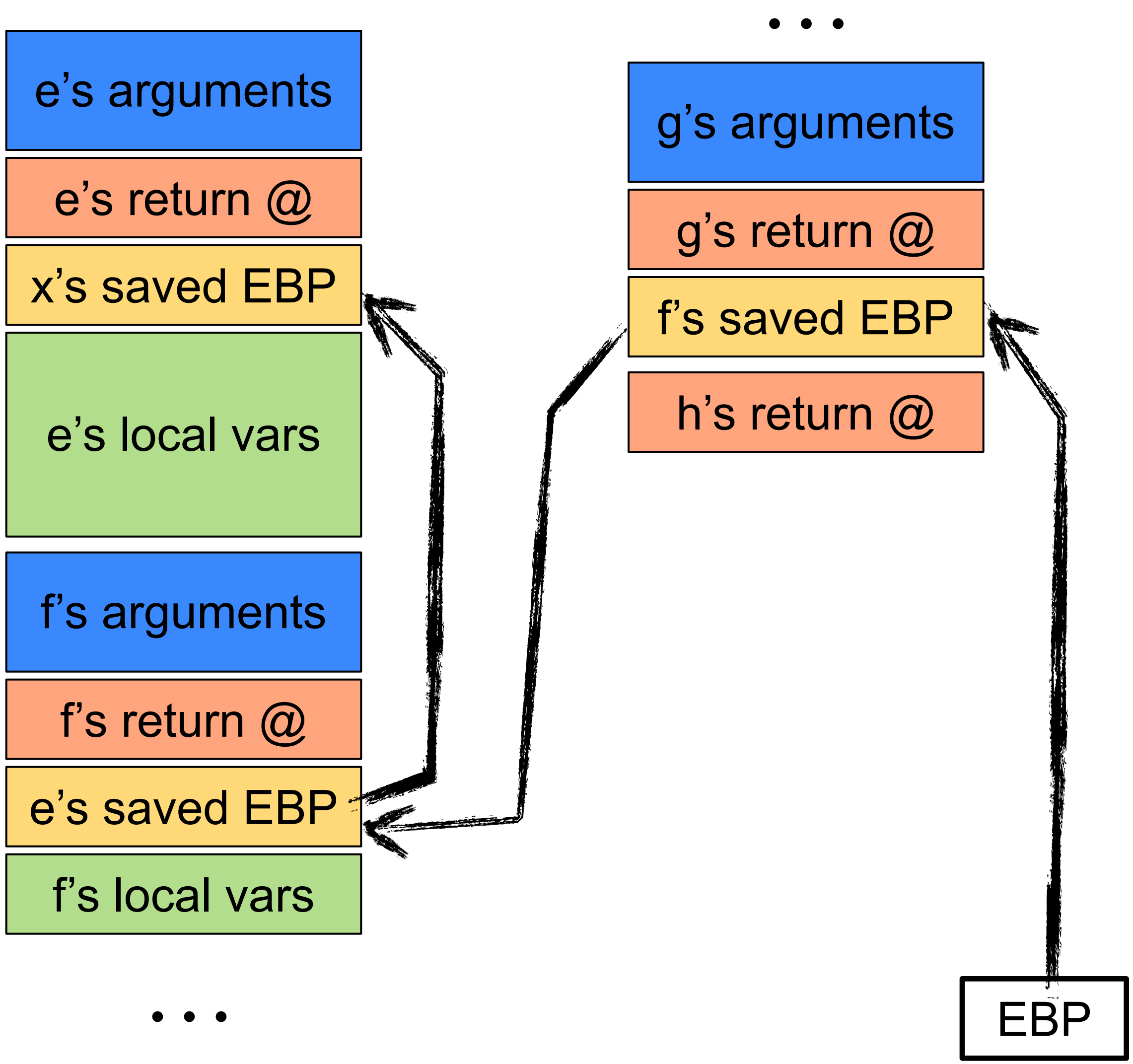
- The saved EBPs provide links between the activation records
- The current EBP is for the current function
- Let's see what happens when h returns

x() calls e() calls f() calls g() calls h()



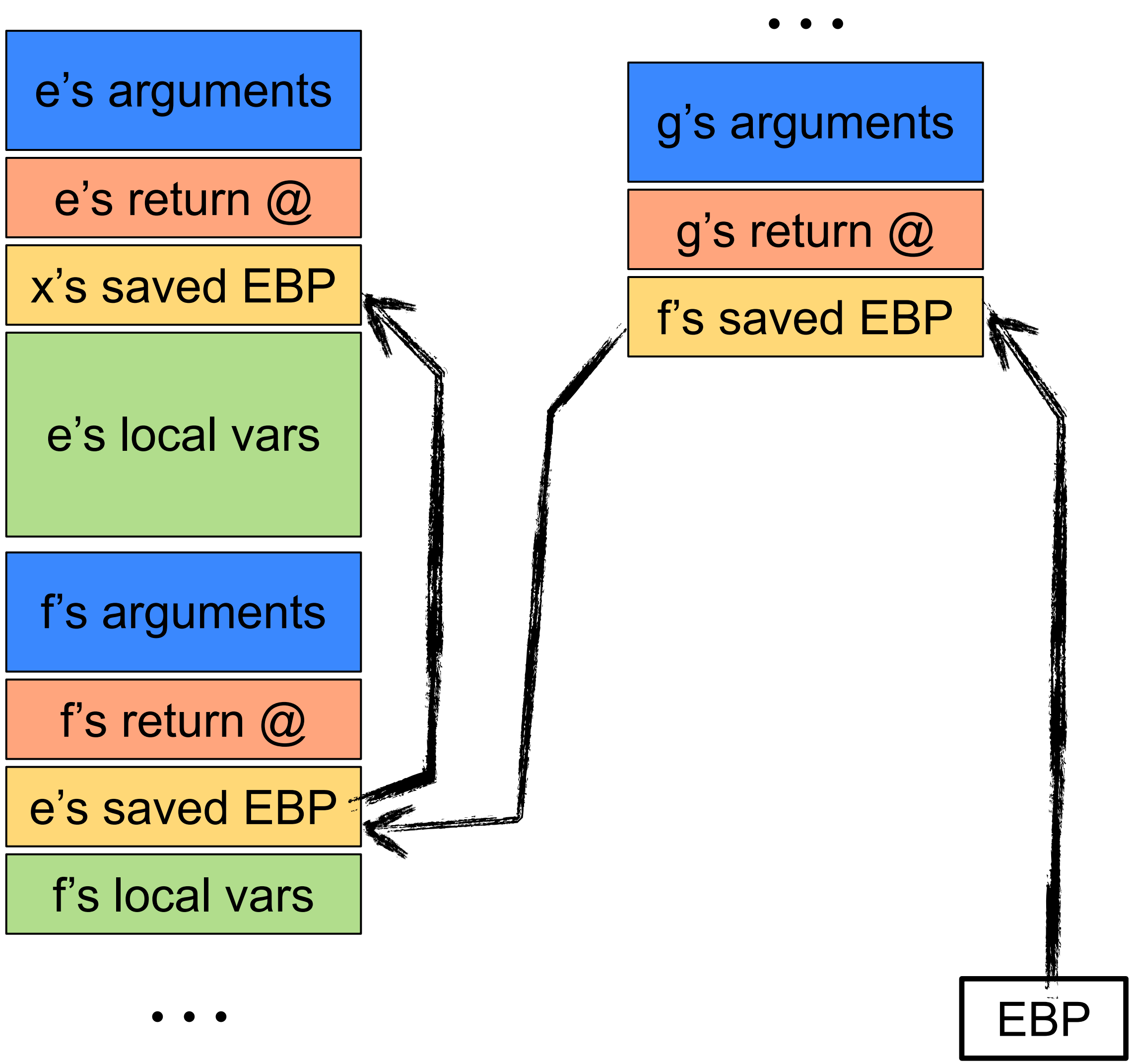
- When h returns
  - mov ESP, EBP

x() calls e() calls f() calls g() calls h()



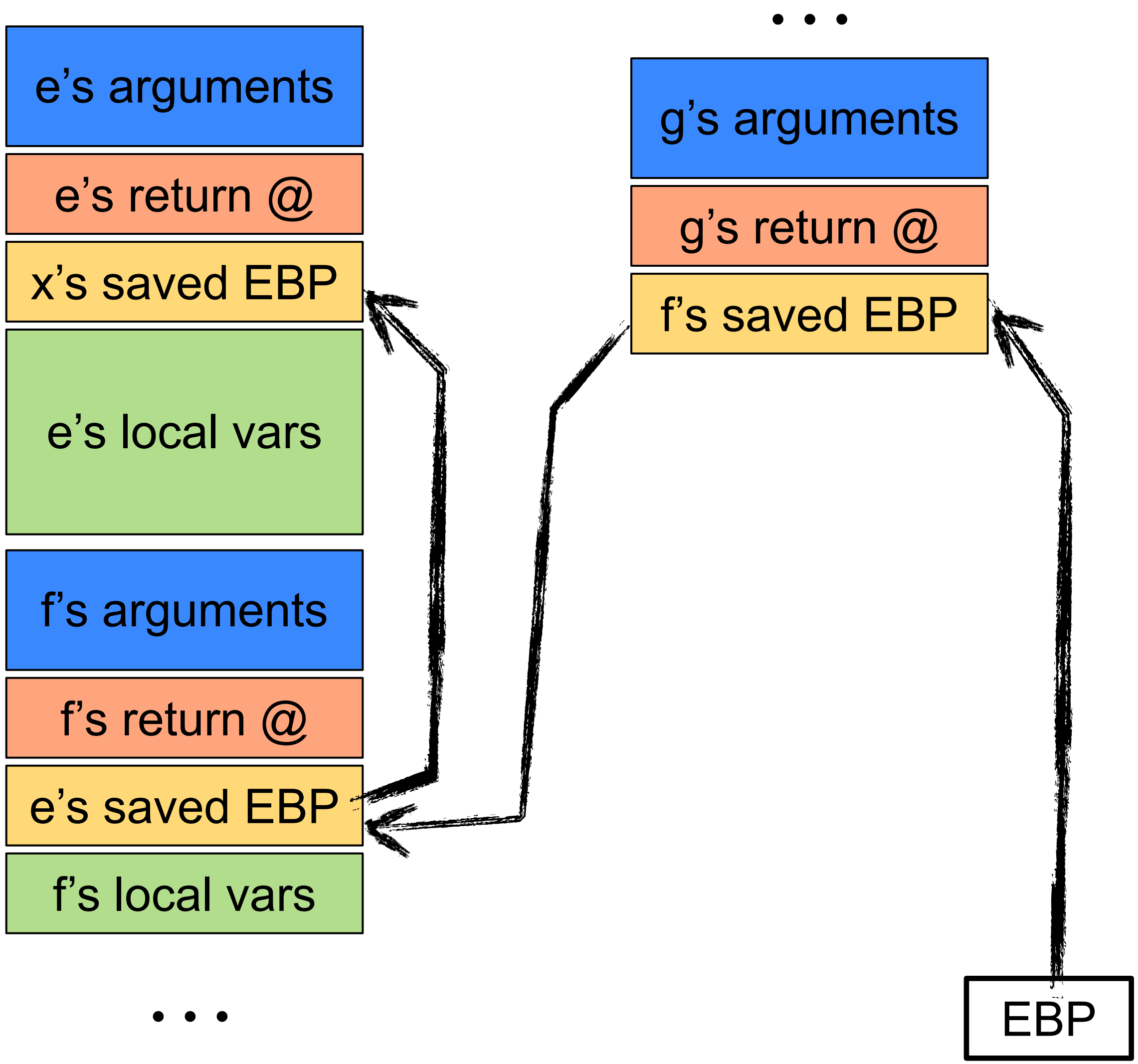
- When h returns
  - mov ESP, EBP
  - pop EBP

x() calls e() calls f() calls g() calls h()



- When h returns
  - mov ESP, EBP
  - pop EBP
  - pop return address

x() calls e() calls f() calls g() calls h()



- We are now in a "clean" state, where g is the active subprogram
- The EBP register and its saved values provide the crucial link between activation records
- If EBP values get corrupted, then all is lost

ENTER and LEAVE

- We always have the same *prologue* and the same *epilogue*

```
push    ebp         ; save old EBP value
mov     ebp, esp    ; set EBP
sub     esp, X      ; reserve X=4*N bytes for N local vars
```

```
mov     esp, ebp    ; remove space for local vars
pop     ebp         ; restore old EBP value
ret                 ; return
```

ENTER and LEAVE

- There are two convenient functions: ENTER and LEAVE

```
push    ebp          ; save old EBP value
mov     ebp, esp     ; set EBP
sub     esp, X       ; reserve X=4*N bytes for N local vars
```

equivalent to

```
enter   X, 0
```

```
mov     esp, ebp     ; remove space for local vars
pop     ebp          ; restore old EBP value
ret                  ; return
```

equivalent to

```
leave
ret
```

Recall the NASM Skeleton

```
    ; include directives

segment .data
    ; DX directives

segment .bss
    ; RESX directives

segment .text
        global asm_main
    asm_main:
        enter   0,0
        pusha
        ; Your program here
        popa
        mov     eax, 0
        leave
        ret
```

Prologue and epilogue of asm_main

We Finally Understand the Skeleton

```
    ; include directives

segment .data
    ; DX directives

segment .bss
    ; RESX directives

segment .text
        global asm_main
    asm_main:
        enter   0,0         ; Save EBP, reserve 0 bytes for local variables
        pusha               ; Save ALL registers
        ; Your program here
        popa                ; Restore ALL registers
        mov     eax, 0      ; Set the return value to 0
        leave               ; Restore EBP, remove space for local variables
        ret                 ; Pop the return address and jump to it
```

Knowing your stack

- At this point it should be clear that it is very important to understand how the stack works and how to use it
- When programming in assembly you should always have a mental picture of the stack
  - Something you don’t do when using a high-level programming language typically
  - As always, abstractions are great, but having no idea how they are implemented can be problematic when hunting bugs
    - Basic example: “running out of stack space”
- It’s typically a good idea to be consistent
  - Compilers are consistent by design

A Full Example

- Let's write the assembly code equivalent to the following C/Java function

```
int   f(int num) {   // computes Fibonacci numbers
      int x, sum;
      if (num == 0) return 0;
      if (num == 1) return 1;
      x = f(num-1);
      sum = x + f(num-2)
      return sum;
}
```

- Let's write a "straight" translation, without optimizing variables away, just for demonstration purposes
- Let's do it live… (even though the next slides have one version of the code)

A Full Example (main program)

```
%include "asm_io.inc"

segment .data
    msg1        db      "Enter n: ", 0
    msg2        db      "The result is: ", 0

    . . . ; declaration of asm_main and setup

    mov     eax, msg1           ; eax = address of msg1
    call    print_string        ; print msg1
    call    read_int            ; get an integer from the keyboard (in EAX)
    push    eax                 ; put the integer on the stack (parameter #1)
    call    f                   ; call f
    pop     ebx                 ; remove the parameter from the stack
    mov     ebx, eax            ; save the value returned by f
    mov     eax, msg2           ; eax = address of msg2
    call    print_string        ; print msg2
    mov     eax, ebx            ; eax = sum
    call    print_int           ; print the sum
    call    print_nl            ; print a new line

    . . . ; clean up
```

A Full Example (function f)

```
;       FUNCTION: f
;       Takes one parameter:  an integer
;       eax = return value
segment .text
f:      enter  8,0          ; num in [ebp+8]
                            ; local var x in [ebp-4],
                            ; local var sum in [ebp-8]
        push   ebx          ; save ebx
        push   ecx          ; save ecx
        push   edx          ; save edx

        mov    eax, [ebp+8]         ; eax = num
        sub    eax, 2               ; eax -= 2
        jns    next         ; if not <0, goto next
        add    eax, 2               ; eax += 2
        jmp    end
next:
        mov    eax, [ebp+8]    ; eax = num
        add    eax, -1         ; eax -= 1
        push   eax          ; put (num -1) on stack
        call   f            ; call f (recursively)
        add    esp, 4       ; remove (num-1) from stack
        mov    [ebp-4], eax    ; put the returned value in x
        mov    eax, [ebp+8]    ; eax = num
        add    eax, -2      ; eax -= 2
        push   eax          ; put (num -2) on stack
        call   f            ; call f (recursively),
                            ; the return value is in eax
        add    esp, 4       ; remove (num-1) from stack
        add    eax, [ebp-4]    ; eax += x

end:
        pop    edx          ; restore ebx
        pop    ecx          ; restore ecx
        pop    ebx          ; restore edx
        leave               ; clean up the stack
        ret                 ; return
```

Interfacing Assembly and C

- Section 4.7 of the book talks about interfacing C and assembly
- We have seen most of this content already, but let's talk about the issue of saving registers on the stack
- By convention, C assumes that a subprogram (e.g., the one you're writing in assembly), will not destroy values in EBX, ESI, EDI, EBP, CS, DS, SS, and ES
- So, if you write an assembly subprogram for others to use, make sure you save these on the stack and restore them
  - We've already said we save EBP
- Example: I know my subprogram uses EBX (as on page 86)

```
enter    4,0            ; prologue (1 32-bit local var)
push     ebx            ; save EBX
. . .
pop ebx                 ; restore EBX
leave                   ; epilogue
ret                     ; return
```

High-level code

- Even though we do assembly in this course, we can now draw stacks for high-level code
- Example:

```
main() {
   int a = 10;
   f(a, 2*a);
}

f(int x, int y) {
  int z = x*3;
  int t;
  // what is the stack here?
}
```

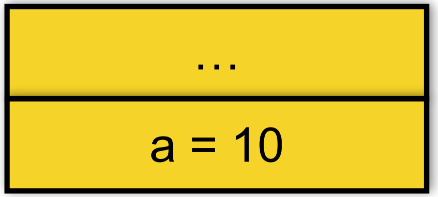
High-level code Example

```
main() {
   int a = 10;
   f(a, 2*a);
}

f(int x, int y) {
  int z = x*3;
  int t;
  // what is the stack here?
}
```

...

High-level code Example

```
main() {
   int a = 10;
   f(a, 2*a);
}

f(int x, int y) {
  int z = x*3;
  int t;
  // what is the stack here?
}
```



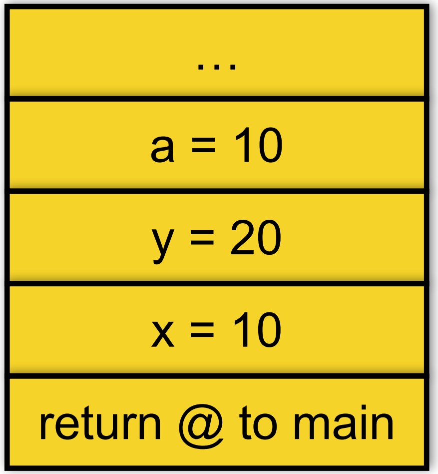
High-level code Example

```
main() {
   int a = 10;
   f(a, 2*a);
}

f(int x, int y) {
  int z = x*3;
  int t;
  // what is the stack here?
}
```

…
a = 10
y = 20
x = 10

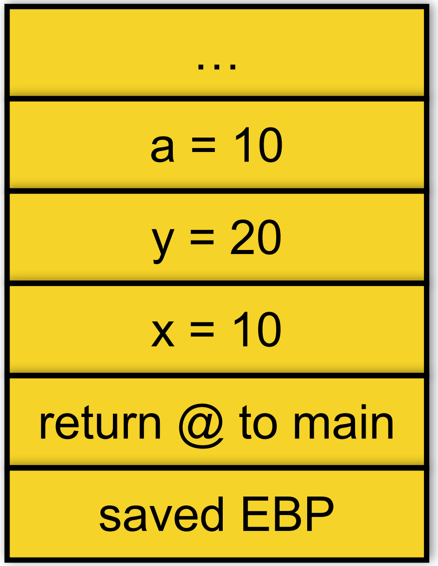
High-level code Example

```
main() {
   int a = 10;
   f(a, 2*a);
}

f(int x, int y) {
  int z = x*3;
  int t;
  // what is the stack here?
}
```



High-level code Example

```
main() {
   int a = 10;
   f(a, 2*a);
}

f(int x, int y) {
  int z = x*3;
  int t;
  // what is the stack here?
}
```



High-level code Example

```
main() {
   int a = 10;
   f(a, 2*a);
}

f(int x, int y) {
  int z = x*3;
  int t;
  // what is the stack here?
}
```

…
a = 10
y = 20
x = 10
return @ to main
saved EBP
z = 30
t = ?

In-class Exercise

- Draw the stack for:

```
main() {
   f(2, 5);
}

f(int x, int y) {
  int z = y * 2;
  if (x == 1) {
    z = 10;
    // what is the stack here?
  } else {
    f(x-1, z+1);
  }
}
```

Really hard to do as a Zoom poll! let's do it together instead…

Solution

```
main() {
   f(2, 5);
}

f(int x, int y) {
  int z = y * 2;
  if (x == 1) {
    z = 10;
    // what is the stack here?
  } else {
    f(x-1, z+1);
  }
}
```

…
y = 5
x = 2
return @ to main
saved EBP
z = 10
y = 11
x = 1
return @ to f
saved EBP
z = ~~22~~ 10

We are done! (with the stack)

- At this point you know everything that's can be on the stack, and why it's there
- Details vary depending on compilers
  - So if you disassemble compiled code, you may find out that things are weirdly out-of-order, and extra things are on the stack, etc.
- But the principles remain
- To demonstrate that it's all real, let's write a piece of C code that "spies" on the stack to discover local variable values of its callers……

Spying on the stack is useful

- This is what your debugger does!
- When you debug a compiled program (using your IDE debugger, using low-tech gdb, etc.) you can always go “up” and “down” the stack to check all local variables
- This is basically jumping back and forth between activation records
  - And we can do that because activation records are linked by saved EBP pointers (see a few slides back)
- The small program with just wrote is a horrible version of what a debugger does
  - And because we didn’t compile with -g, we lost so-called “debugging information”
  - e.g., we no don’t know variable names in the source code

Conclusion

- When programming one always faces trade-offs between program readability and program performance
- With by-hand assembly programming, the programmer can make fine-tuned decisions for these trade-offs
  - e.g., for a particular function I decide to not save all registers because I _know_ that it won't corrupt them, thus saving time
  - e.g., I know that I can reuse some register value that was modified in a subprogram to do some clever optimization
- Some of these optimizations can only be done by a human who understands what the program does
  - Many optimizations can be done by a compiler

- We'll have an in-class practice quiz next week on this module
- We now can look at Homework Assignments #7 and #8
- Onward to "Buffer Overflow"…