

Lecture 9: Procedures & Functions

CS 540

George Mason University

Procedures/Functions

- Control Abstraction
 - call/return semantics, parameters, recursion
- Controlled Namespace
 - Scope (local/non-local), binding, addressing
- External Interface
 - separate compilation, libraries (not dealing with here)

Procedures as Control Abstractions

Relationship between caller and callee is asymmetric

- Control flow (call and return)
- Data flow (call and return): parameters and return values
- Recursion
- Variable addressing

The way this data/control flow is achieved is strictly defined and the given rules must be adhered to by the compiler

Data Structure: Call Graph

A **call graph** is a directed multi-graph where:

- the nodes are the procedures of the program and
- the edges represent calls between these procedures.

Used in optimization phase.

Acyclic → no recursion in the program

Can be computed **statically**.

Example

```
var a: array [0 .. 10] of integer;
```

```
procedure readarray
```

```
var i: integer
```

```
begin ... a[i] ... end
```

```
function partition(y,z: integer): integer
```

```
var i,j,x,v: integer
```

```
begin ... end
```

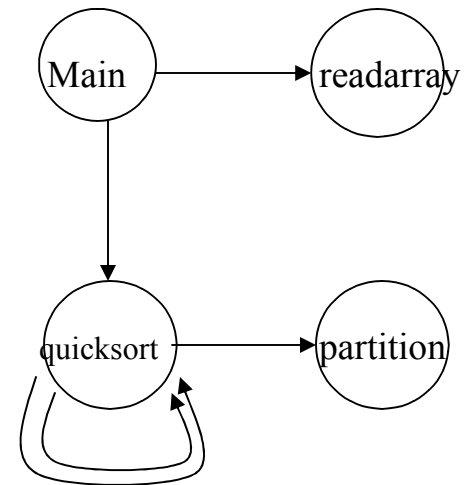
```
procedure quicksort(m,n: integer)
```

```
var i: integer
```

```
begin i := partition(m,n); quicksort(m,i-1); quicksort(i+1,n) end
```

```
procedure main
```

```
begin readarray(); quicksort(1,9); end
```

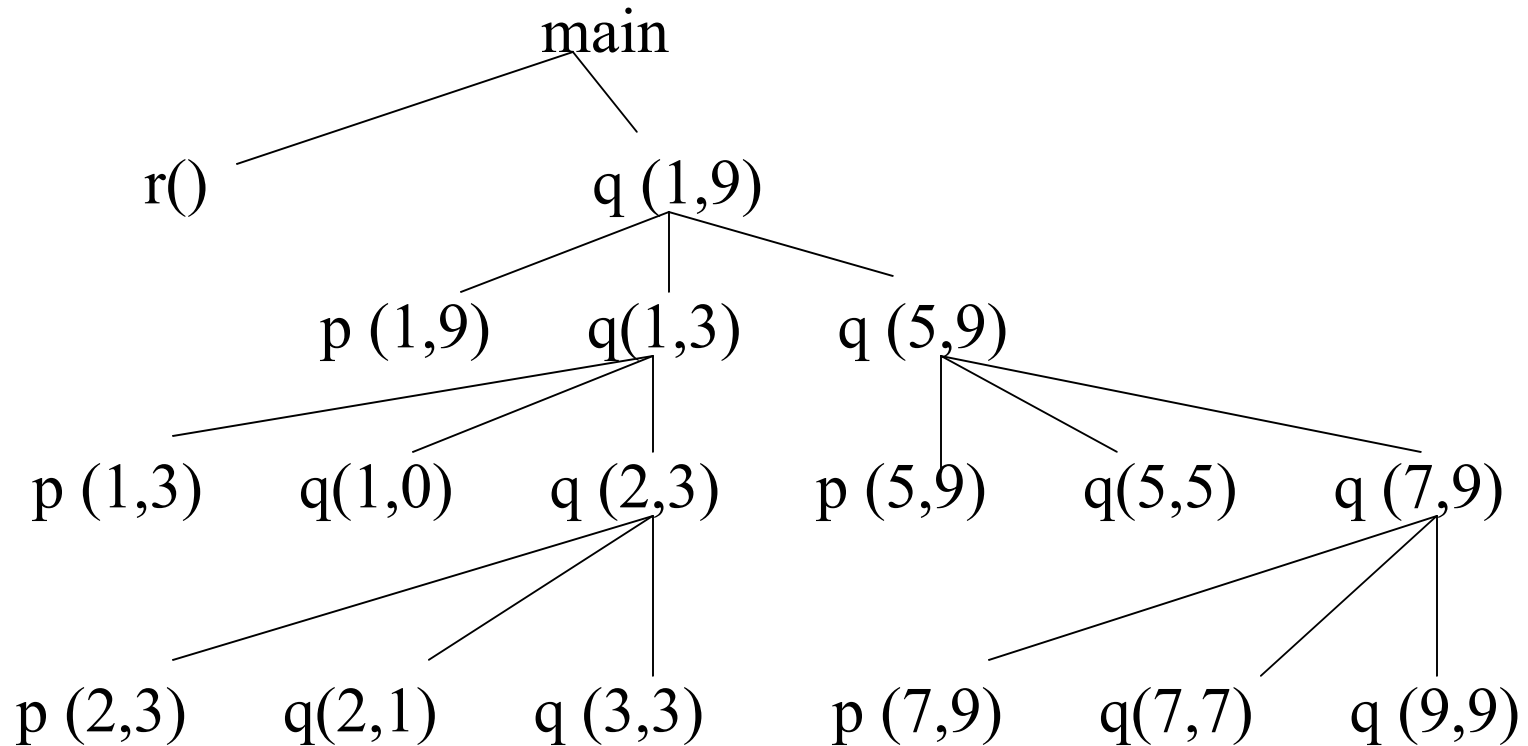


Data Structure: Call Tree

- A **call tree** is a tree where:
 - the nodes are the procedure activations of the program and
 - the edges represent calls between these procedure activations.
- Dynamic – typically different every time the program is run

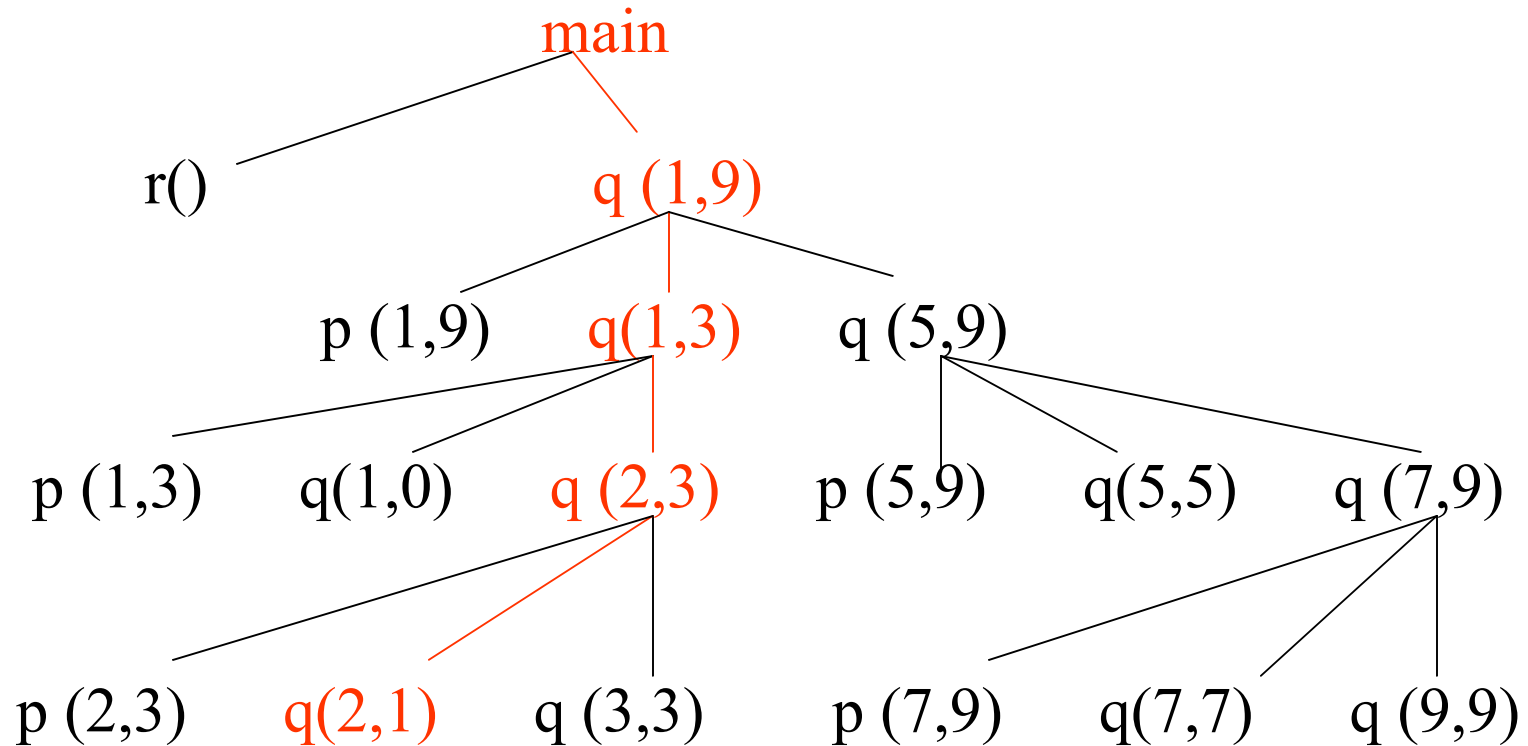
Run-time Control Flow

Call Tree - cannot be computed statically



Run-time Control Flow

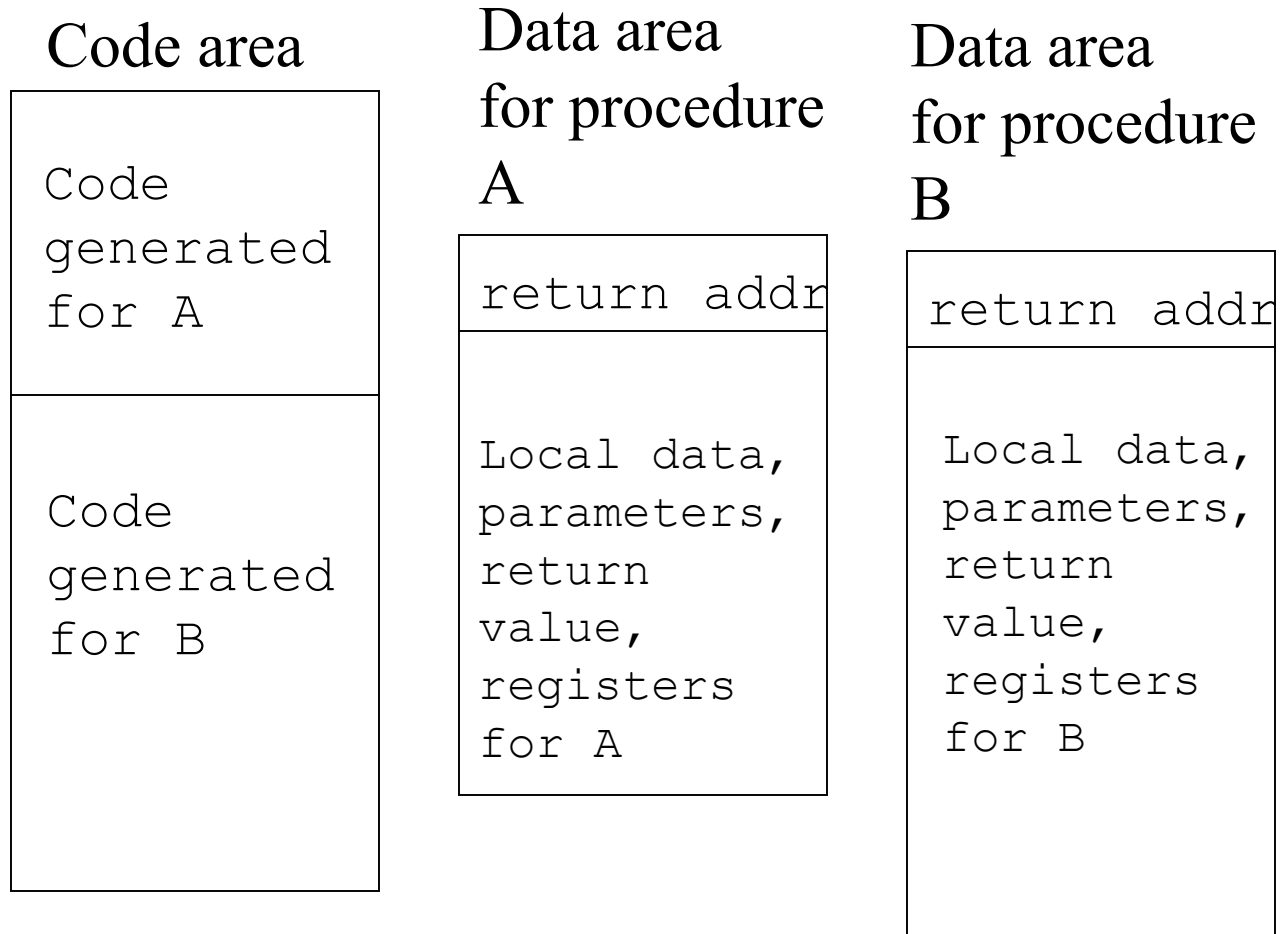
Paths in the call tree from root to some node represent a sequence of active calls at runtime



Static Allocation

- Historically, the first approach to solving the run-time control flow problem (Fortran)
- **All** space allocated at compile time → **No recursion**
 - Code area – machine instructions for each procedure
 - Static area –
 - single data area allocated for each procedure.
 - local vars, parameters, return value, saved registers
 - return address for each procedure.

Static Allocation



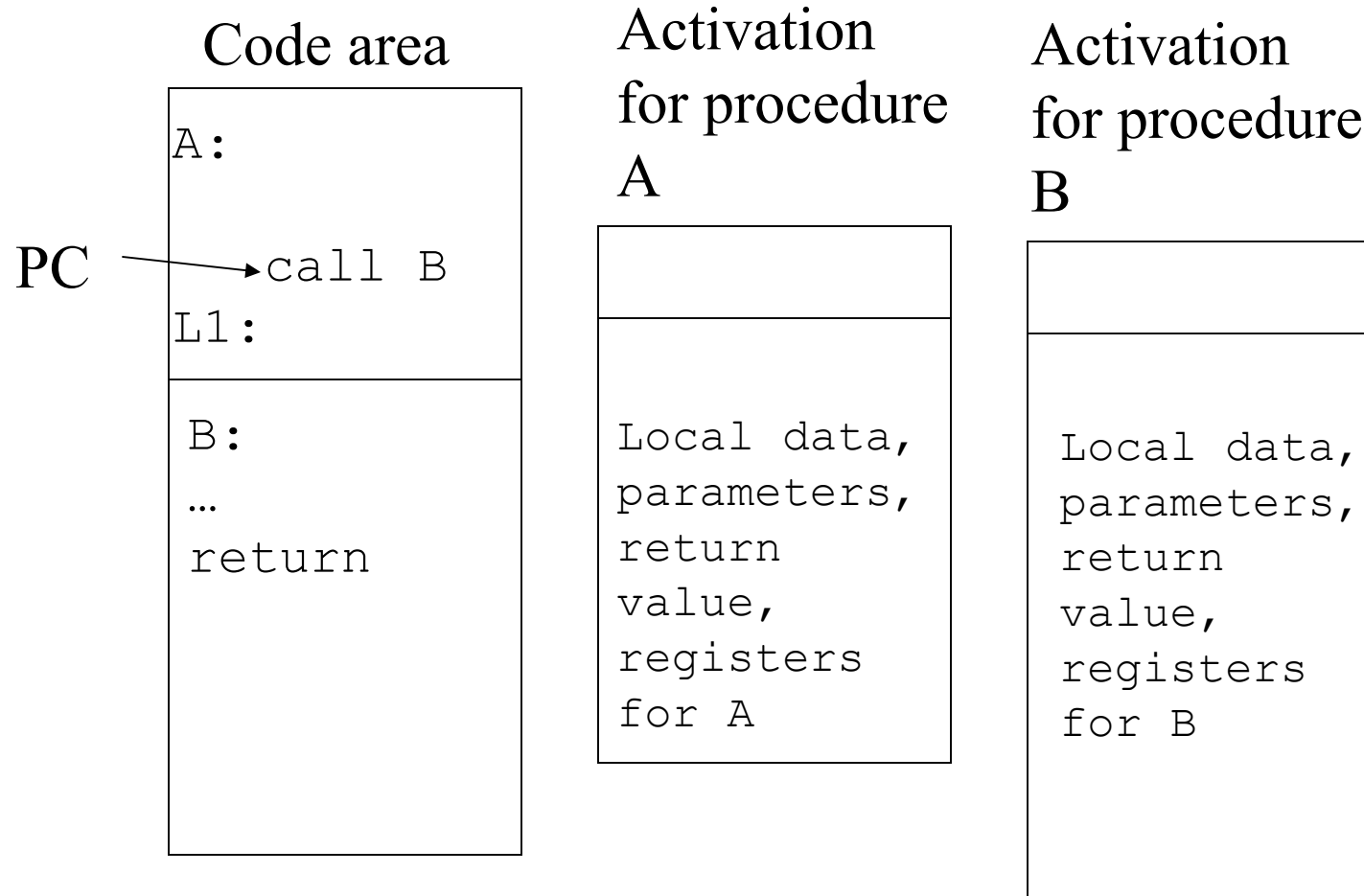
Call/Return processing in Static Allocation

- When A calls B:
 - in A: evaluate and save actual parameters, save any registers and status data needed, save RA, finally update the program counter (PC) to B's code
 - In B: deal with parameters and RA (if needed)
- When the call returns
 - in B: save return value, update PC to value in RA
 - in A: get return value, restore any saved registers or status data

Save options: data areas (A or B), registers

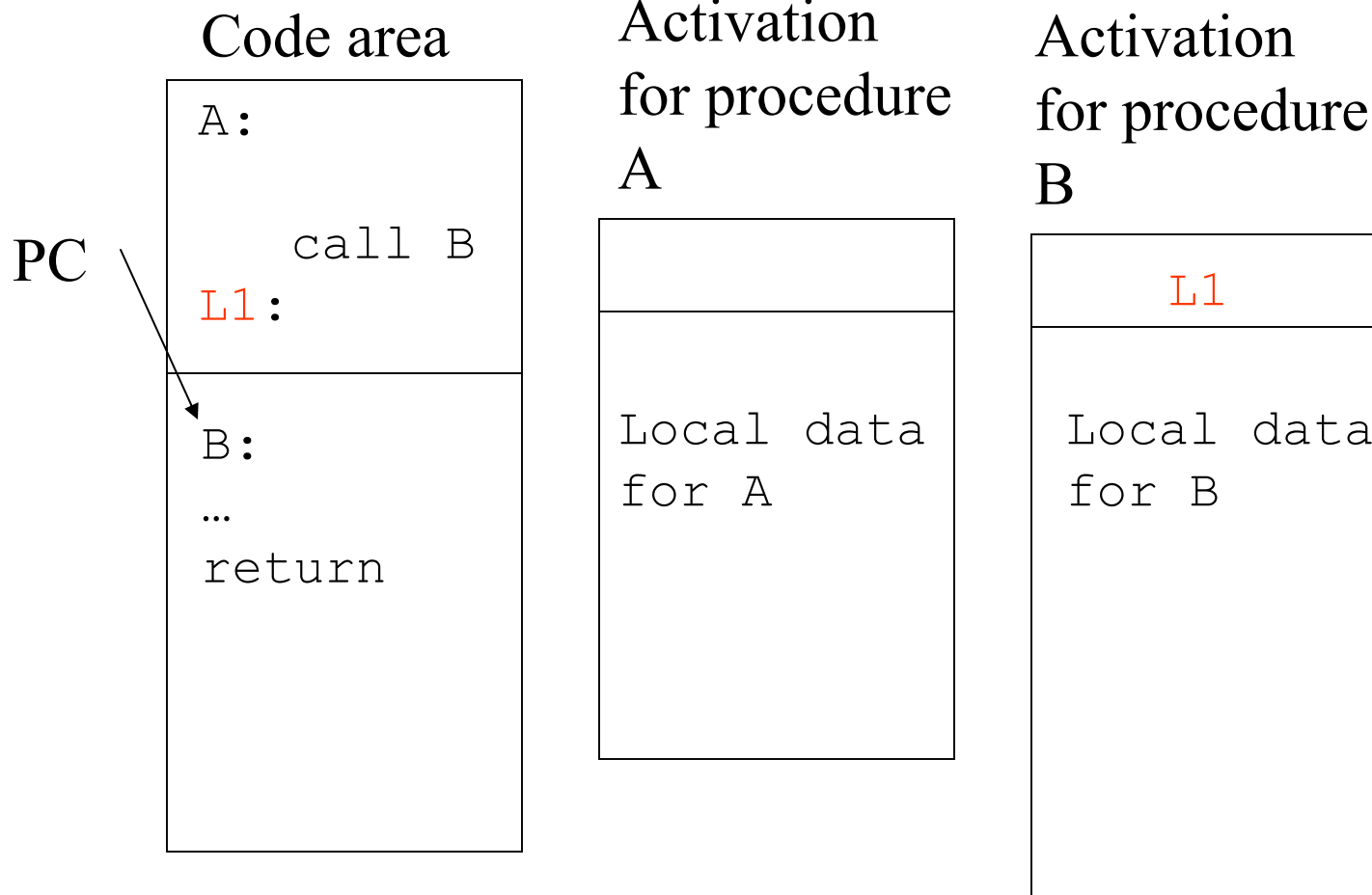
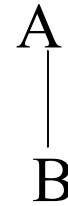
Static Allocation

Call tree:
A



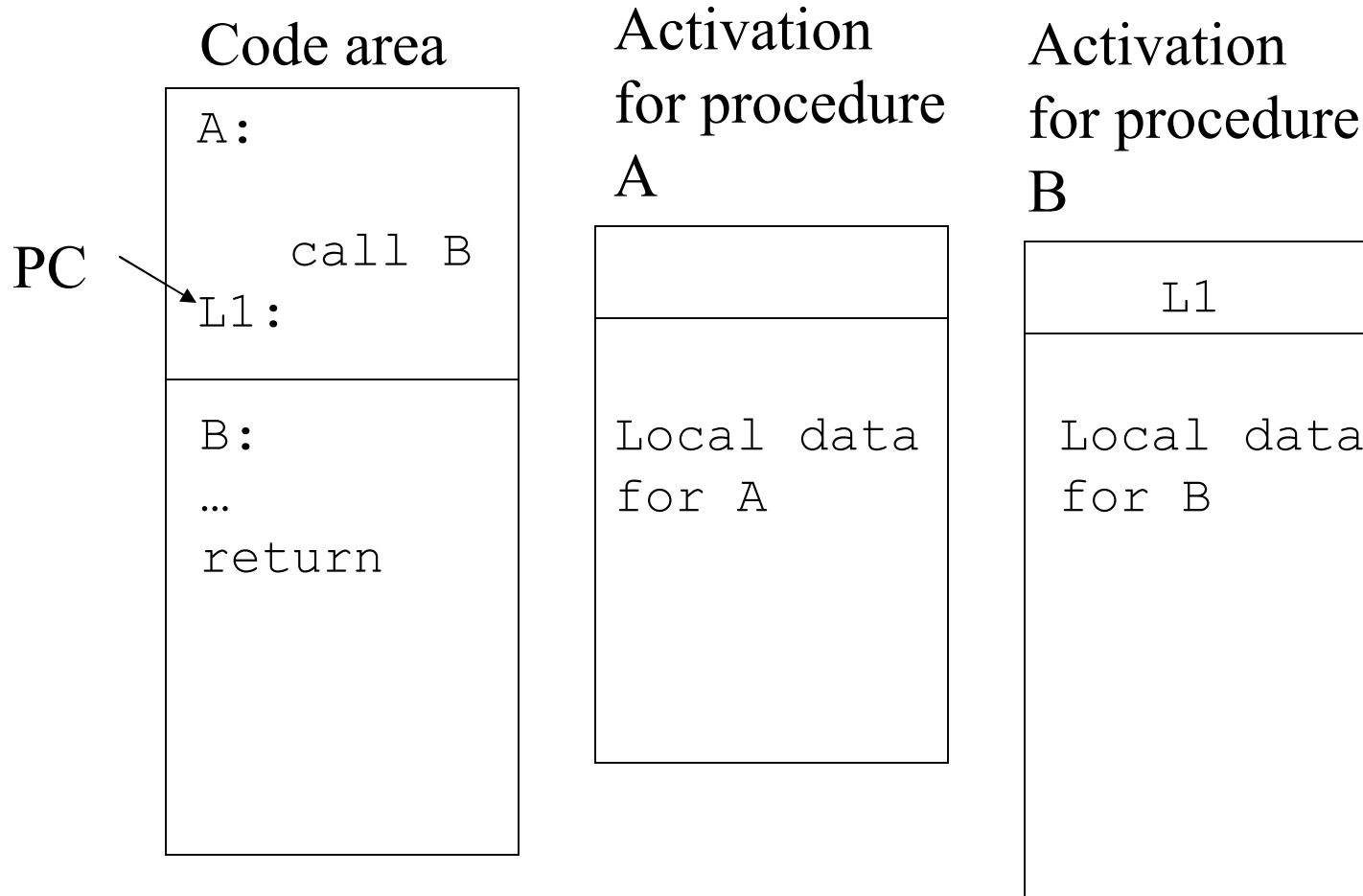
Static Allocation

Call tree:



Static Allocation

Call tree:
A



Managing Control Flow: Spim

1. Set aside extra space for return address and local info for each function.

```
FN_RA:          .word    0  
FN_Regs:      .space   48  
                # save 8 $t + 4 $a registers  
FN_Params:    .word   ...
```

Managing Control Flow: Spim

2. Add code at the point of the call:

```
sw $t0, FN1_Regs
sw $t1, FN1_Regs+4 ...
```

```
move $a0, $t ...
```

```
jal function
```

```
lw $t0, FN1_Regs
lw $t1, FN1_Regs +4 ...
```

```
move $tx, $v0
```

Save and restore local registers in use

Pass in parameters and get the return value

jal – save the address of the next statement in \$ra and then jump to the given label

Managing Control Flow: Spim

3. Add code in the called function

- Prologue:

```
sw $ra, FN2_RA  
sw $a0, FN2_Param1 ...
```

jal put return
address in \$ra

- Epilogue

```
move $v0, $ty  
lw $tx, FN2_RA  
jr $tx
```

If there is a
return value...

result := f(a,bb);

```
.data
main_Regs: .word 0,0,0,0,0,0,0,0
main_RA:   .word 0
.text
lw $t0,a
move $a0,$t0
lw $t0,bb
move $a1,$t0
sw $t0,main_Regs
sw $t1,main_Regs+4
sw $t2,main_Regs+8
...
jal label_f
lw $t0,main_Regs
lw $t1,main_Regs+4
lw $t2,main_Regs+8
move $t0,$v0
sw $t0,result
```

```
int f(int x, int y) { ... return max; }
```

```
.data  
f_RA: .word 0 # return addr
```

```
.text
```

```
label_f:
```

```
sw $ra, f_RA
```

```
.data
```

```
x: .word 0 # param 1
```

```
.text
```

```
sw $a0, x
```

```
.data
```

```
y: .word 0 # param 2
```

```
.text
```

```
sw $a1, y
```

body of f

```
lw $v0, max # return val
```

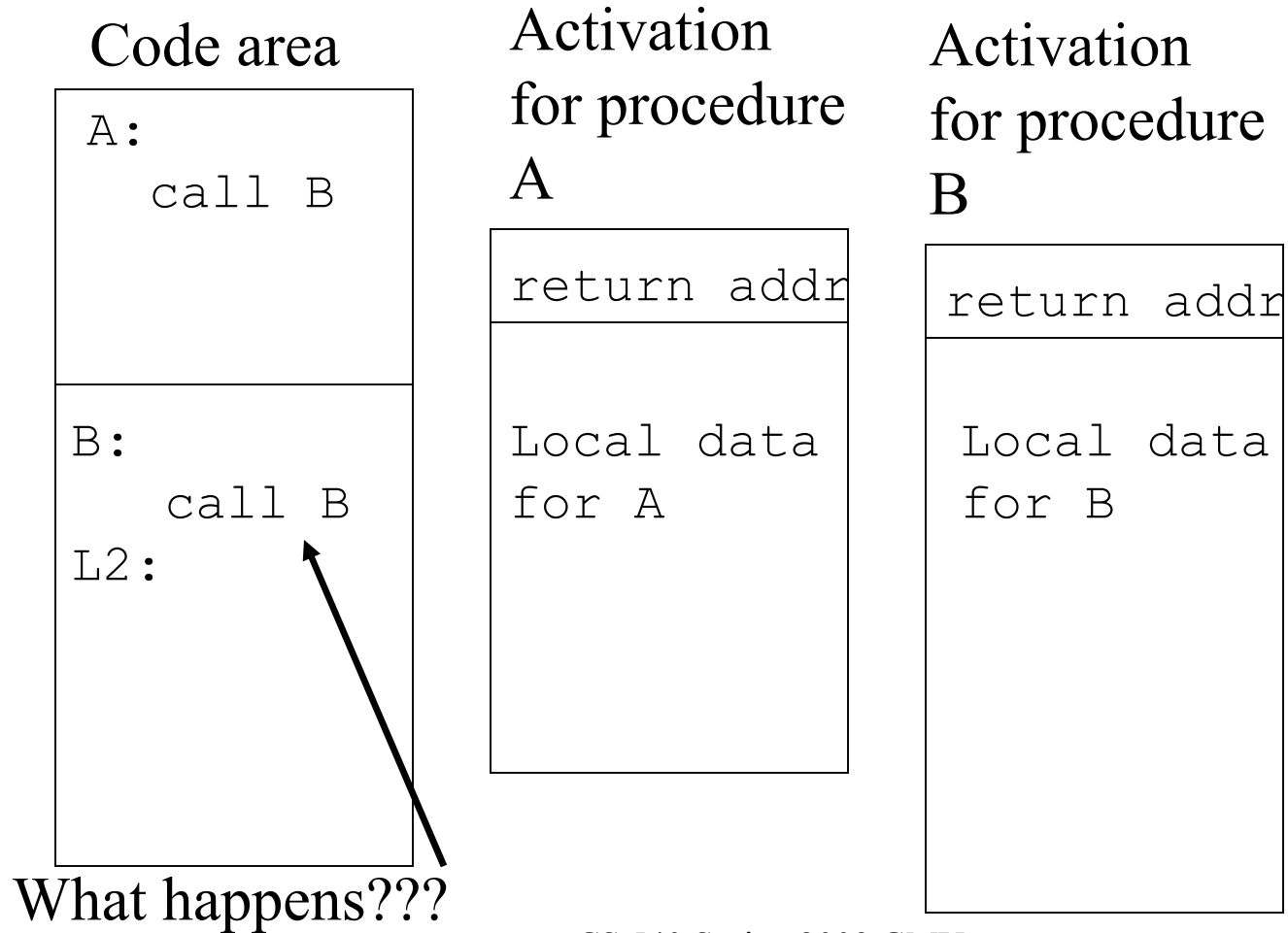
```
lw $t0, f_RA
```

```
jr $t0
```

Runtime Addressing in Static Allocation

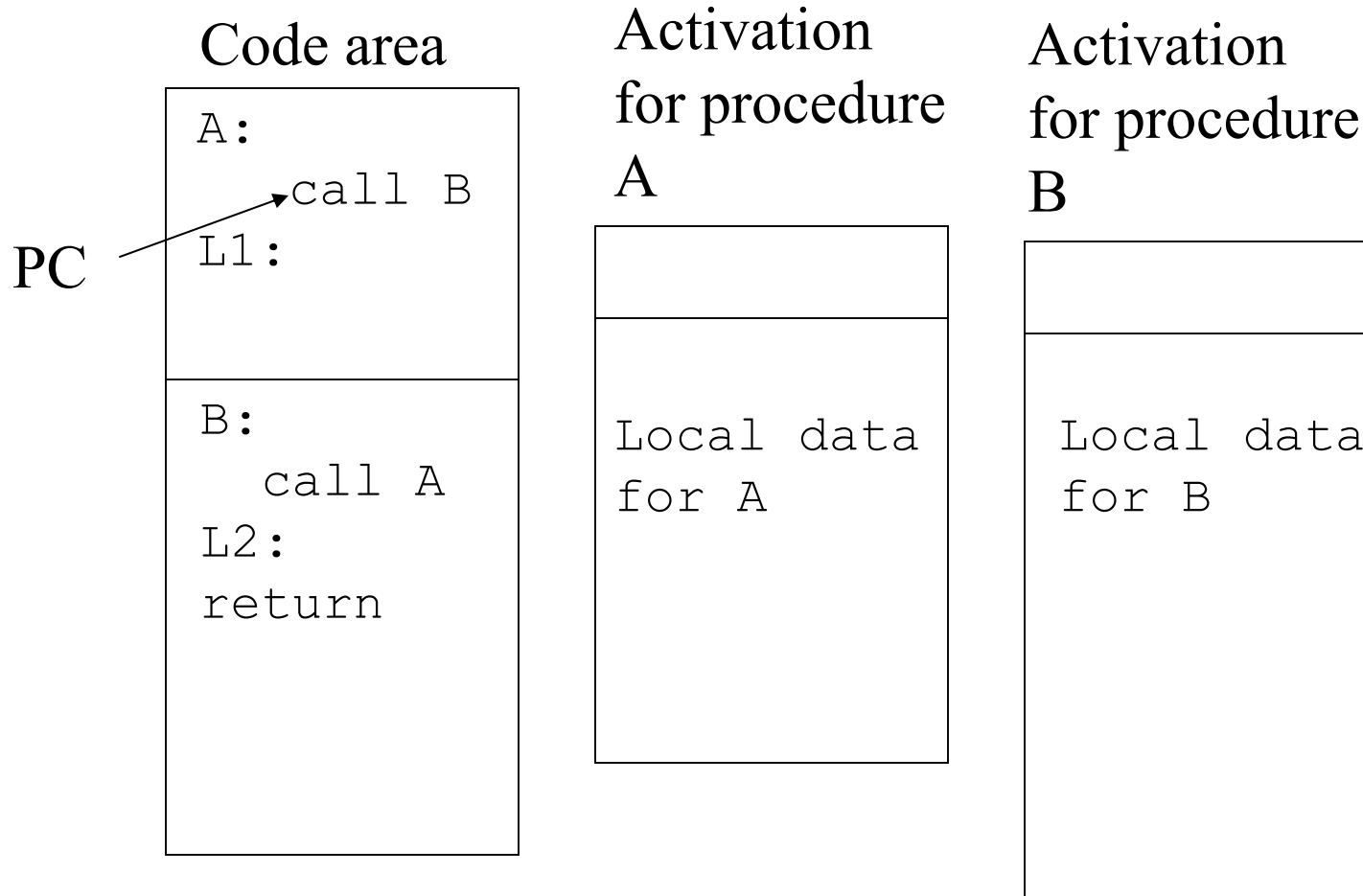
- Variable addresses hard-coded, usually as offset from data area where variable is declared.
 - $\text{addr}(x) = \text{start of } x\text{'s local scope} + x\text{'s offset}$
- In Spim, we are going to save the local variable using a label and when needed, we can use **lw** to get the value (or **la** to get the address).

Static Allocation: Recursion?



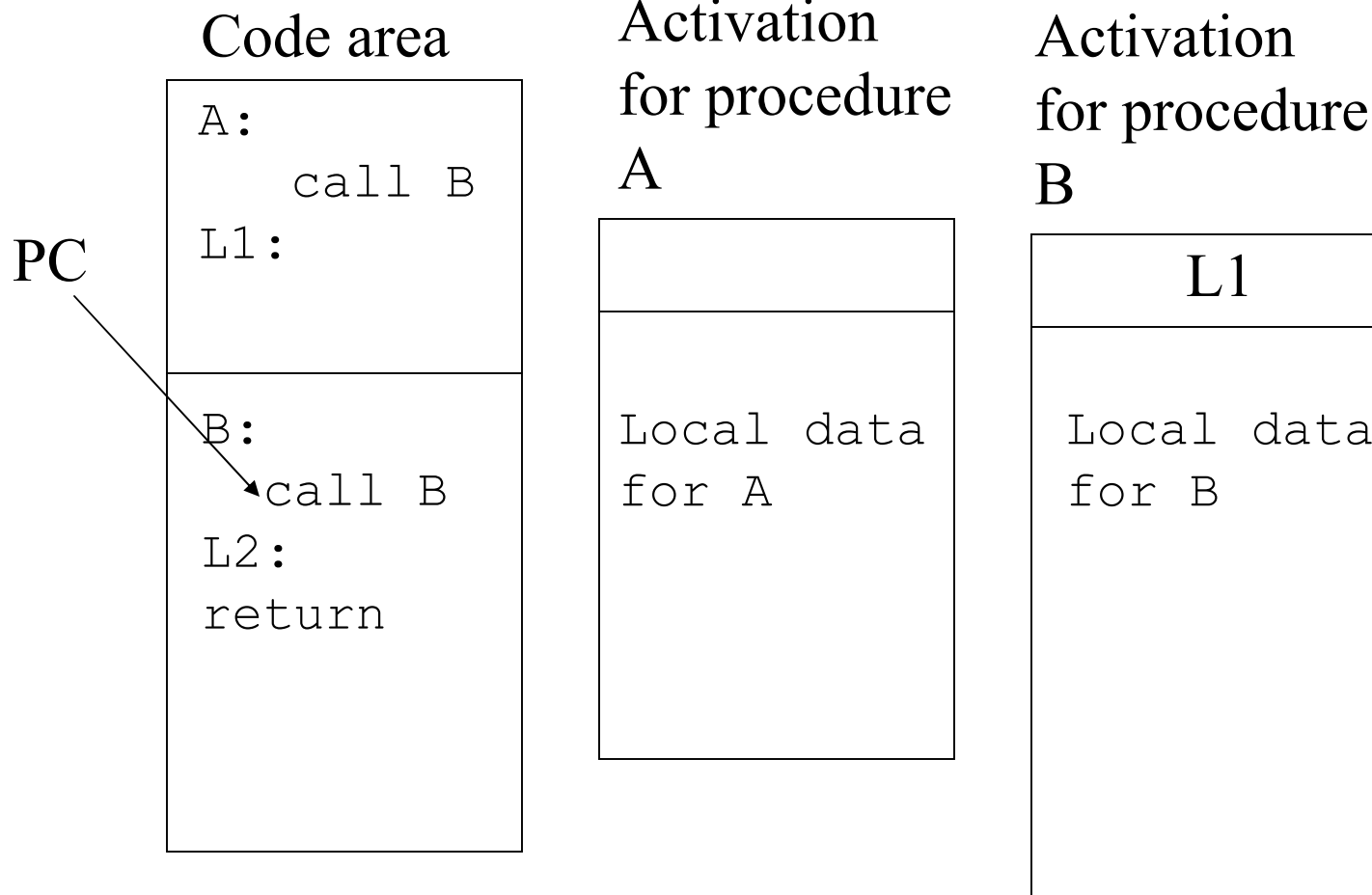
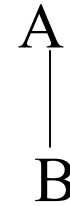
Static Allocation

Call tree:
A



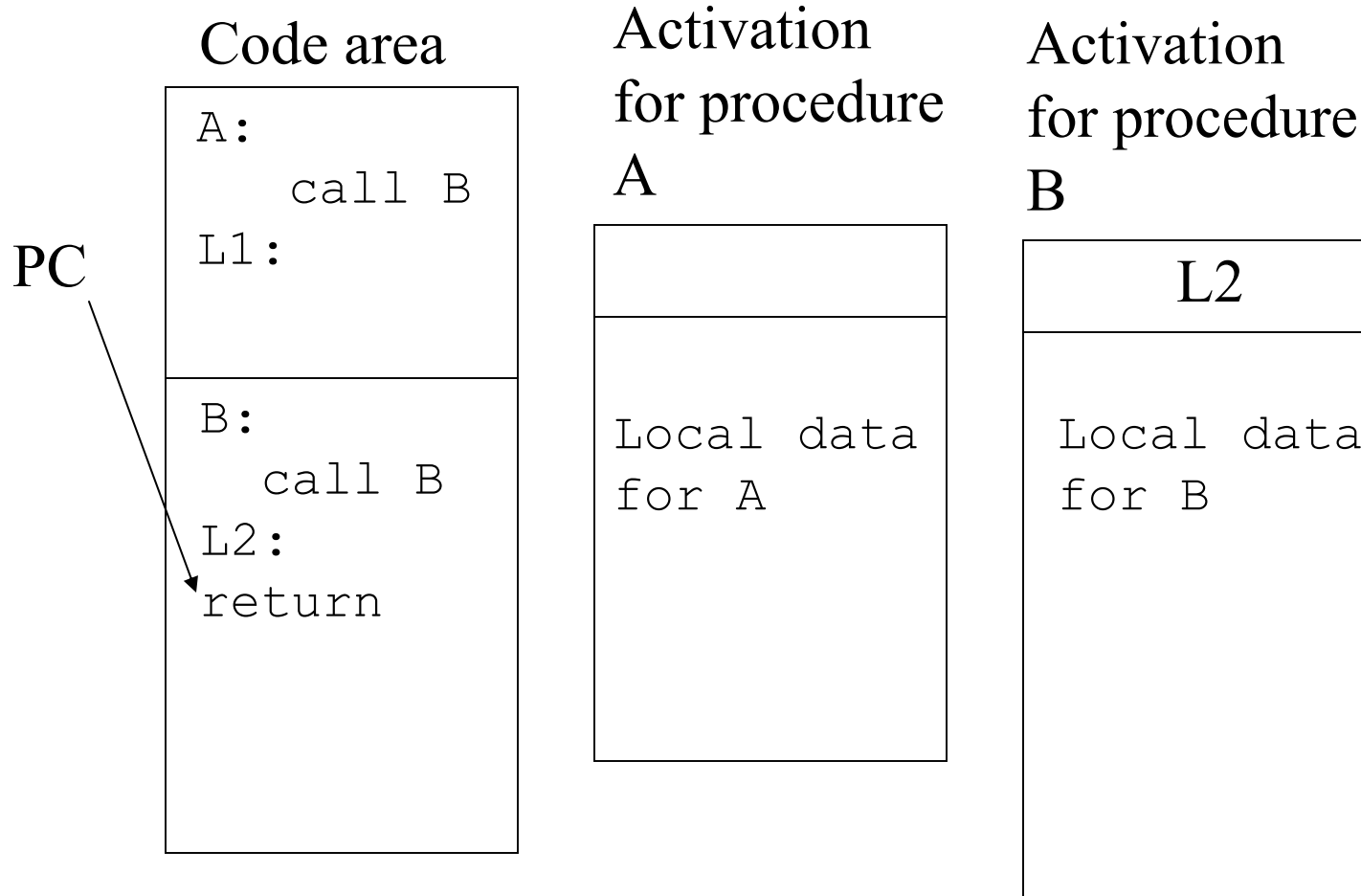
Static Allocation

Call tree:



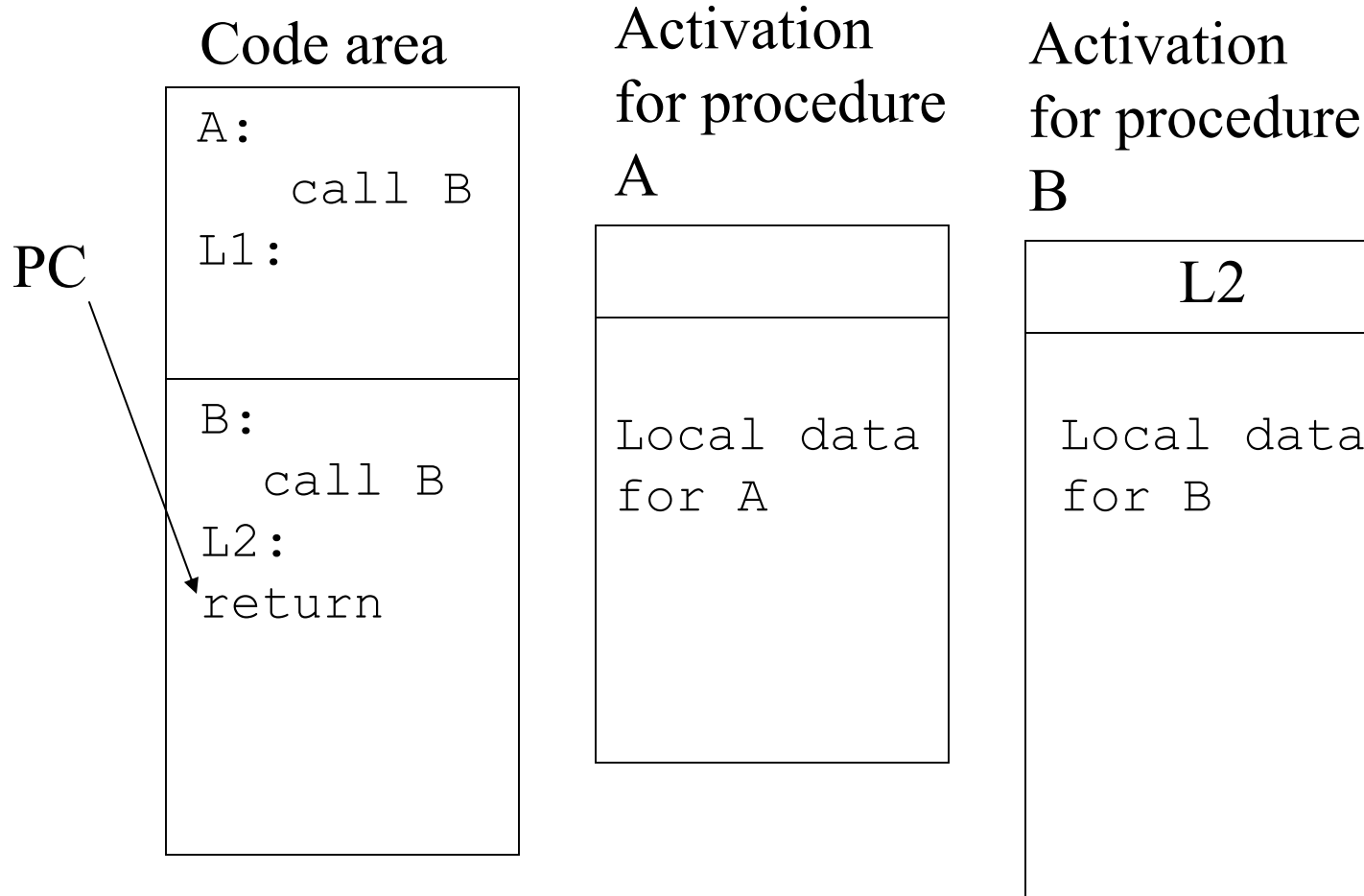
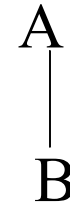
Static Allocation

Call tree:



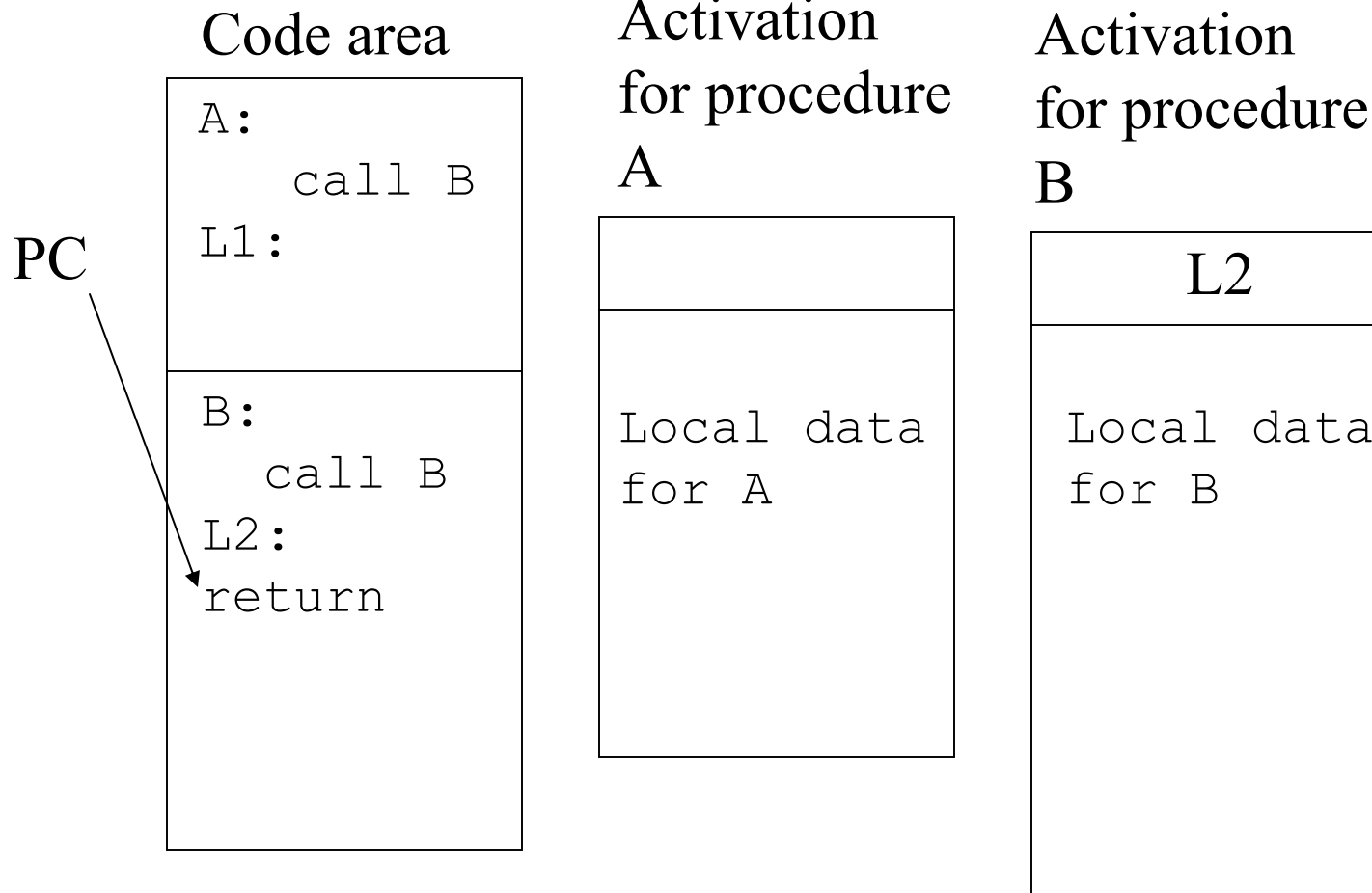
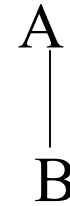
Static Allocation

Call tree:



Static Allocation

Call tree:



We've lost the L1 label so we can't get back to A

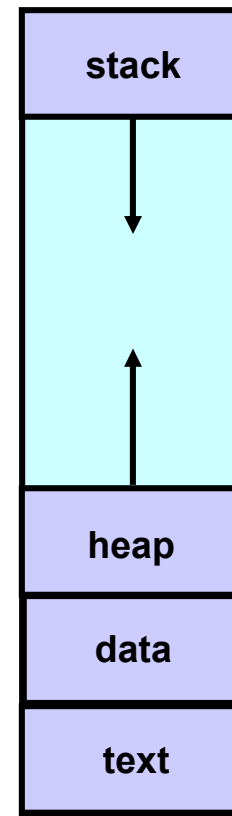
Stack Allocation

Need a different approach to handle recursion.

- Code area – machine code for procedures
- Static data – often not associated with procedures
- Stack – runtime information
 - Activation records – allocated at call time onto a runtime stack. Holds return addresses, local information
 - Dynamic – can grow and shrink

Process Address Space

- Each process has its own *address space*:
 - Text section (text segment) contains the executable code
 - Data section (data segment) contains the global variables
 - Stack contains temporary data (local variables, return addresses..)
 - Heap, which contains memory that is dynamically allocated at run-time.



Activation Records (frame)

Information needed by a single instance of a procedure.

- Local data
 - Parameter storage
 - Return value storage
- Saved registers
- Control links for stack
- Return address

Activation Records

Different procedures/functions will have different size activation records.

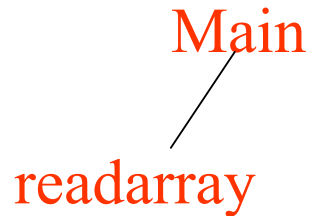
Activation record size can be determined at compile time.

At call time, we push a new activation on the runtime stack.

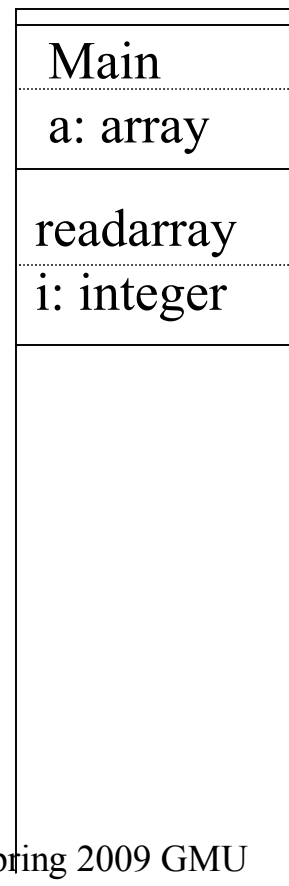
At call termination time, we pop the activation off the stack.

Stack Allocation - 1

Call Tree

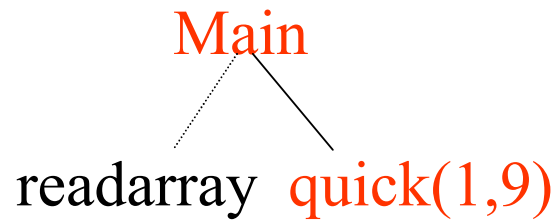


Stack (growing downward)

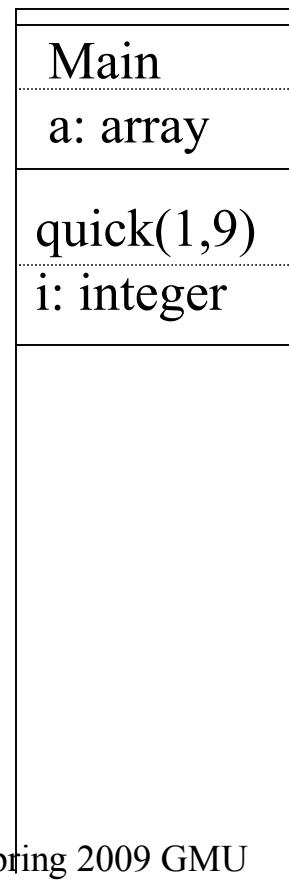


Stack Allocation - 2

Call Tree

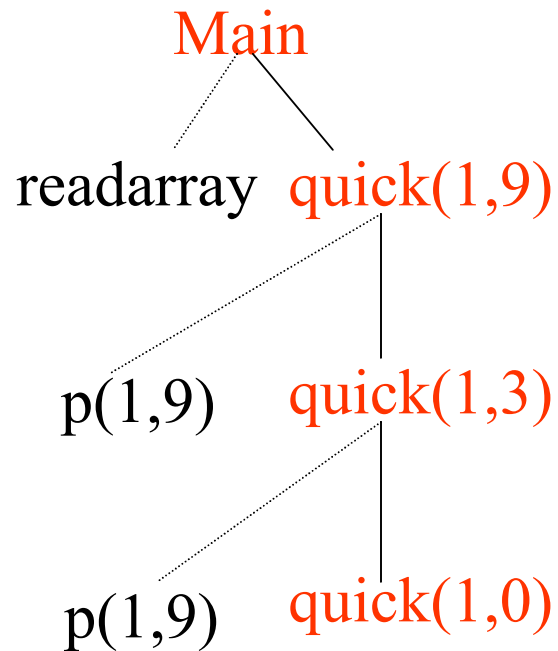


Stack (growing downward)

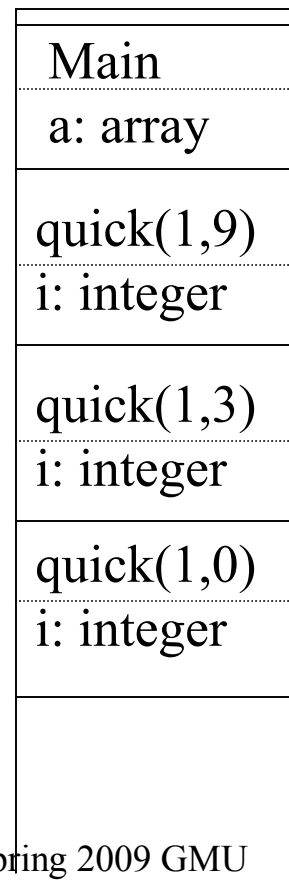


Stack Allocation - 3

Call Tree



Stack (growing downward)



Call Processing: Caller

- **Create new (callee) activation record on stack.**
- Evaluate actual parameters and place them on stack (or register).
- Save registers and other status data
- Store a return address, **dynamic link (= current stack pointer)**, and **static link information** into callee activation then
- **Make stack pointer (SP) point at new activation.**
- Update the program counter (PC) to the code area for the called procedure.

Added at the point of the call

Call Processing: Callee

- Initialize local data, including moving parameters (if needed)
- Save return address if needed
- Stack maintenance
- Begin local execution

Added at the start of the function

Return Processing: Callee

- Place return value (if any) in activation.
- Stack maintenance
- Restore PC (from saved RA).

Added at the ‘return’ point(s) of the function

Return Processing: Caller

- Restore the stack pointer
- Restore registers and status
- Copy the return value (if any) from activation
- Continue local execution

Added after the point of the call

Spim Example: Activation

| Offset | Data | Size |
|--------|----------------|------|
| 0 | Return address | 4 |
| 4 | old frame ptr | 4 |
| 8 | \$t registers | 32 |
| 40 | \$a registers | 16 |
| 56 | local vars | ? |
| ... | | |

Spim Example

Assume a function `f` with two integer parameters named `x` and `y` that are passed by value. `f` returns a single integer. Also in function `f` are two local variables `a` and `b`.

Function `f`'s prolog :

```
sw $ra, 0($sp) # save return address
sw $a0, 56($sp) # parameter x
sw $a1, 60($sp) # parameter y
```

NOTE: local variables `a` and `b` can be stored at offsets 64 and 68 respectively.

- Function `f`'s epilog:

```
move $v0, $t1
lw $t1, 0($sp)
jr $t1
```

At f(a,b)

```
# saving registers
```

```
sw $t0,8($sp)
```

```
sw $t1,12($sp)
```

```
...
```

```
sw $a0,40($sp)
```

```
...
```

```
# get actual parameters
```

```
lw $a0,a_global
```

```
lw $a1,b_global
```

```
#stack maintenance
```

```
sw $fp,4($sp)
```

```
subu $sp,$sp,104
```

```
addiu $fp,$sp,100 # set frame ptr
```

```
jal outputnums
```

```
move $t0,$v0 # grab return value
```

```
addiu $sp,$sp,104 # reset sp
```

```
lw $fp,4($sp) # reset frame pointer
```

```
lw $t0,8($sp) # restoring registers
```

```
lw $t1,12($sp)
```

```
...
```

```
lw $a0,40($sp)
```

```
...
```

<http://cs.gmu.edu/~white/CS540/Slides/Semantic/runtime.html>

Runtime Addressing

- Given a variable reference in the code, how can we find the correct instance of that variable?
- Things are trickier – variables can live on the stack.
- Tied to issues of scope

Types of Scoping

- **Static** – scope of a variable determined from the source code. Scope A is enclosed in scope B if A's source code is nested inside B's source code.
- **Dynamic** – current call tree determines the relevant declaration of a variable use.

Static Scoping: Most Closely Nested Rule

The scope of a particular declaration is given by the most closely nested rule

- The scope of a variable declared in block B, includes B.
- If x is not declared in block B, then an occurrence of x in B is in the scope of a declaration of x in some enclosing block A, such that A has a declaration of x and A is more closely nested around B than any other block with a declaration of x.

Example Program

Program main;

a,b,c: real;

procedure sub1(a: real);

d: int;

procedure sub2(c: int);

d: real;

body of sub2

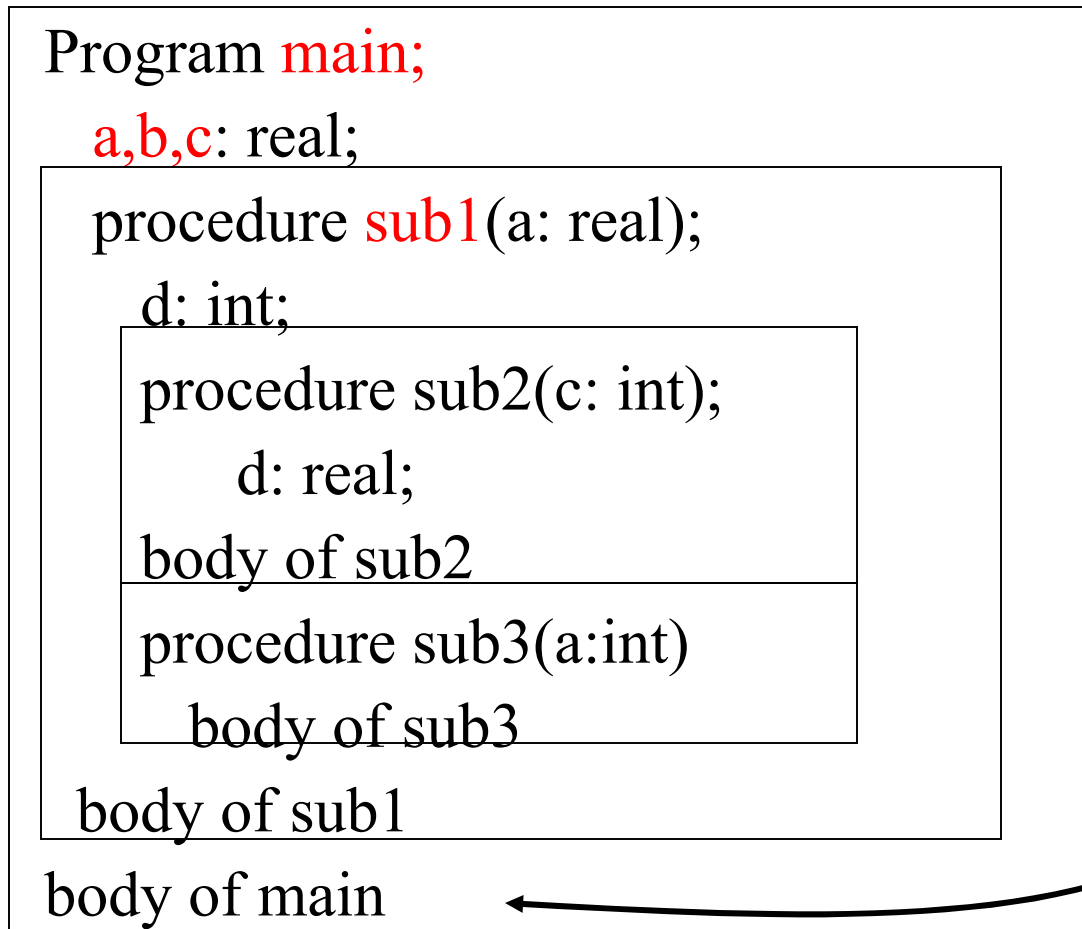
procedure sub3(a:int)

body of sub3

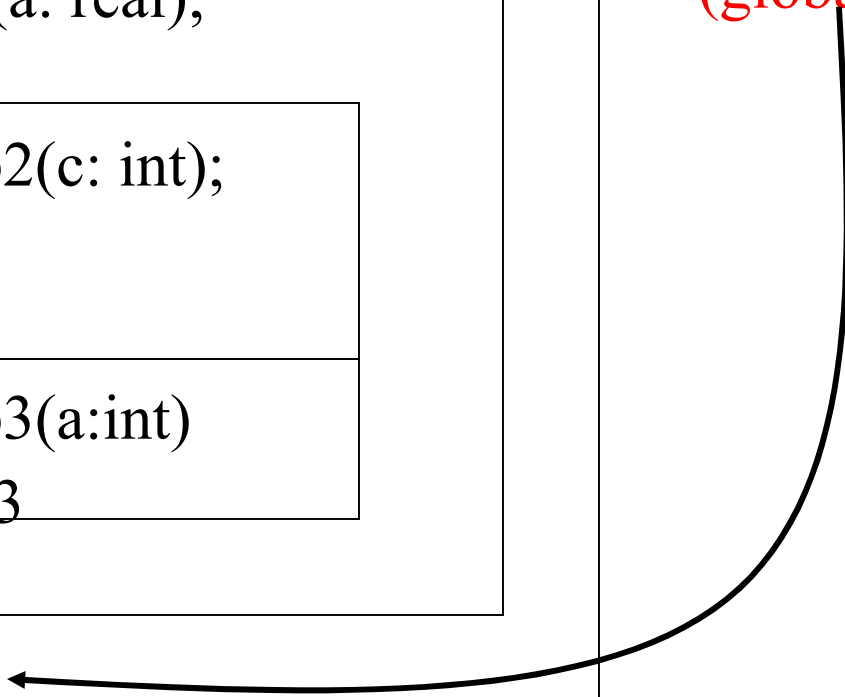
body of sub1

body of main

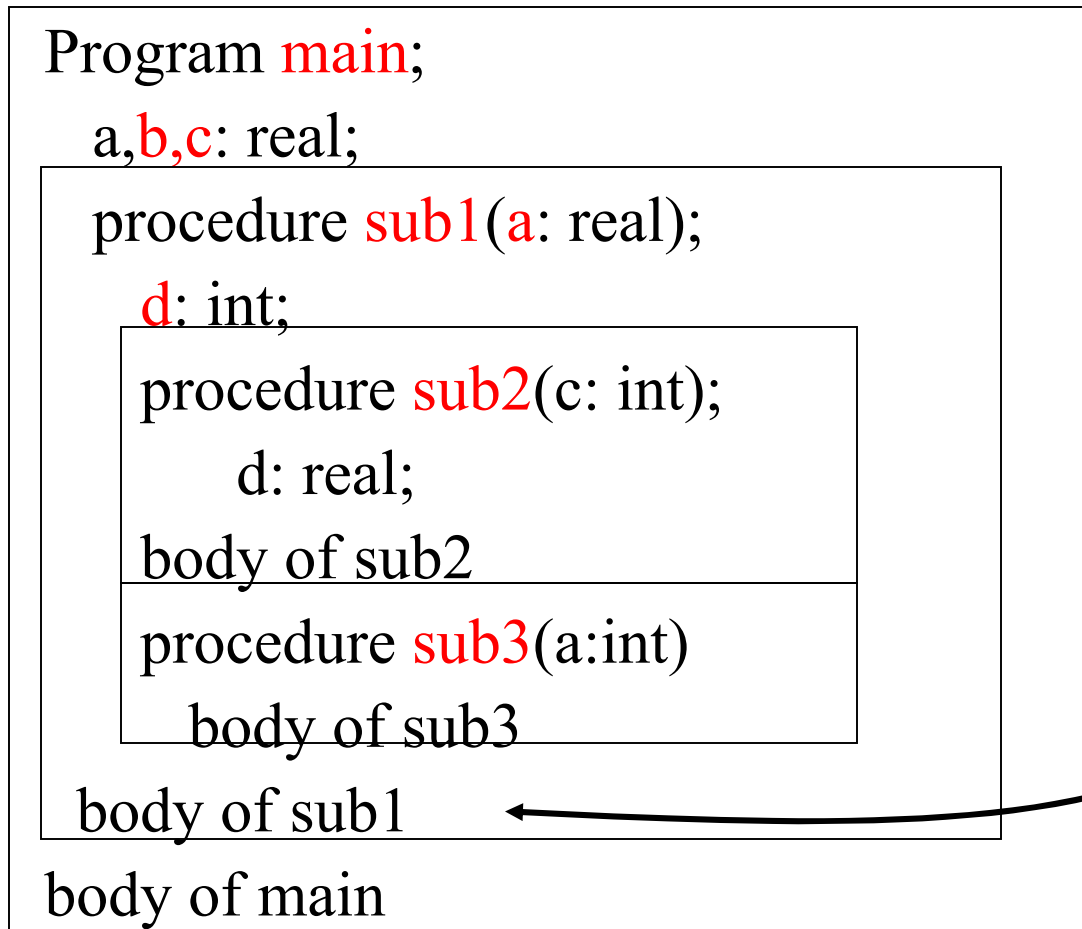
Example Program: Static



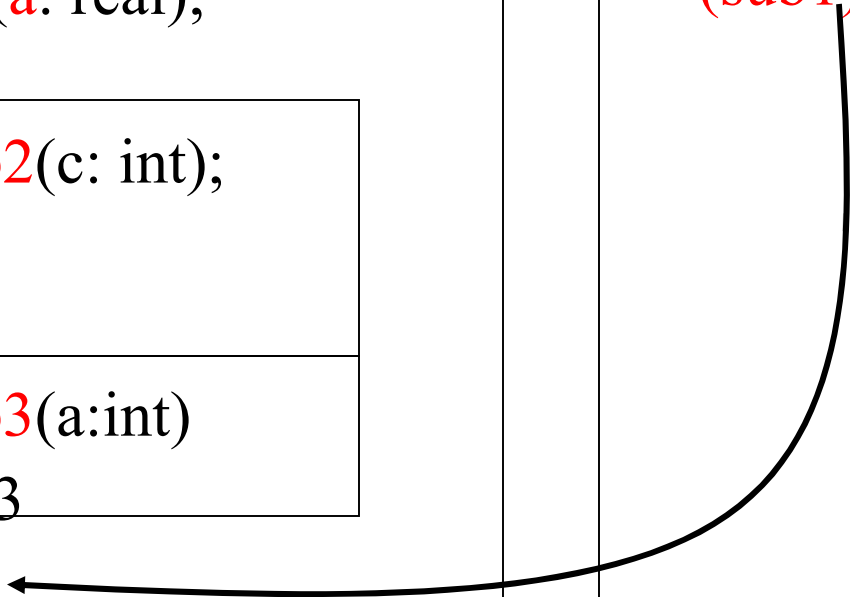
What is visible
at this point
(globally)?



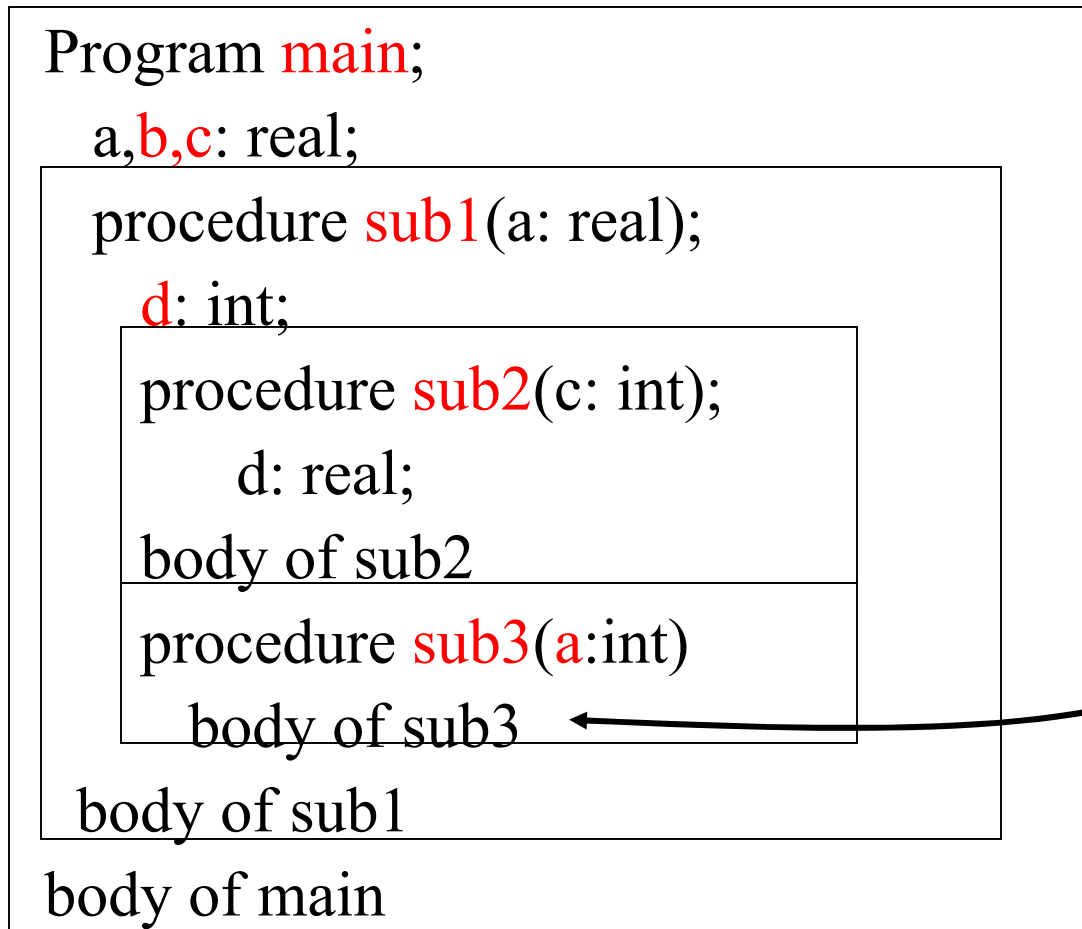
Example Program: Static



What is visible
at this point
(sub1)?



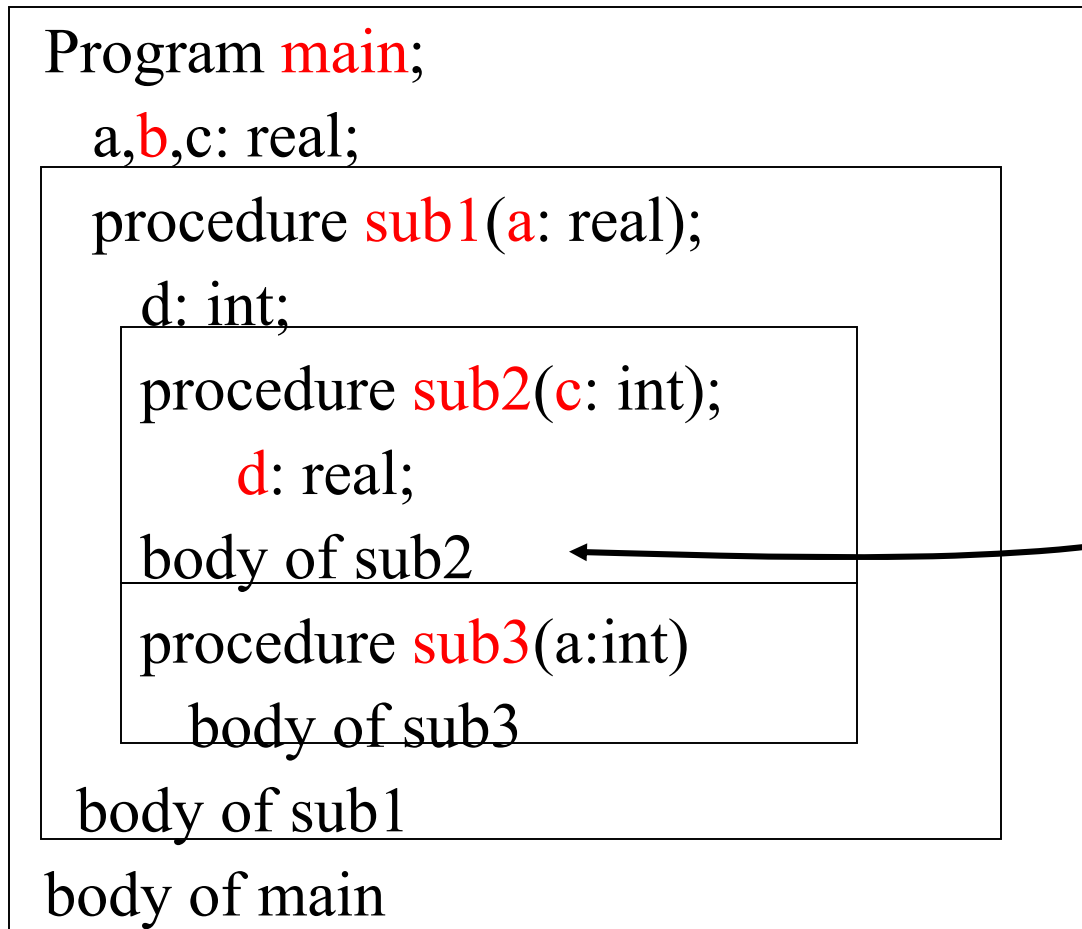
Example Program: Static



What is visible
at this point
(sub3)?



Example Program: Static



What is visible
at this point
(sub2)?



Variables from Example

| Procedure | Enclosing | Local:addr = offset | Non-local: addr =(scope,offset) |
|------------------|------------------|--------------------------------|--|
| main | - | a:0,b:1,c:2 | - |
| sub1 | main | a:0,d:1 | b:(main,1),c:(main,2) |
| sub2 | sub1 | c:0,d:1 | b:(main,1) a:(sub1,0) |
| sub3 | sub1 | a:0 | b:(main,1),c:(main,2) d:(sub1,1) |

Control Links in Stack Allocation

- Dynamic – points to caller's activation (old frame pointer)
- Static (access) link – points to enclosing scope

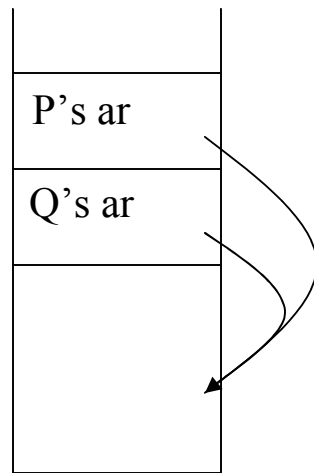
Static Chain Maintenance

- How to set the static link?
 - Let P_{sd} be the `static_depth` of P, and Q_{sd} be the `static_depth` of Q
 - Assume Q calls P
 - There are three possible cases:
 1. $Q_{sd} = P_{sd}$
 2. $Q_{sd} < P_{sd}$
 3. $Q_{sd} > P_{sd}$

Static Chain Maintenance: Q calls P

$Q_{sd} = P_{sd}$ - They are at same static depth

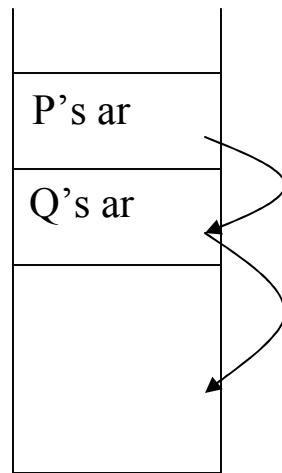
P's static link should be the same as Q's since they must occur in same enclosing scope – Q copies its link to P



Static Chain Maintenance: Q calls P

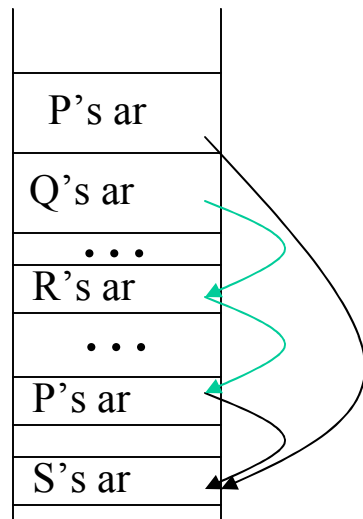
$Q_{sd} < P_{sd}$ - P must be enclosed directly in Q

P's static link should point at Q's activation

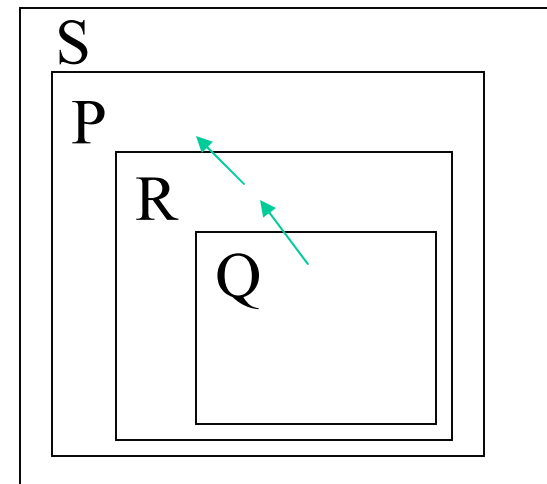


Static Chain Maintenance: Q calls P

$Q_{sd} > P_{sd}$ - Q is n levels down in the nesting –
must follow Q's static chain n levels and copy
that pointer



Suppose n is 2



Runtime Addressing in Stack Allocation

- At runtime, we can't know where the relevant activation record holding the variable exists on the stack
- Use static (access) links to enable quick location
 - $\text{addr}(x) = \# \text{ static links} + x\text{'s offset}$
 - Local: (0,offset)
 - Immediately enclosing scope: (1,offset)

Example Program

Program main;

```
procedure sub1(a: int,b:int);
```

```
  procedure sub2(c: int);
```

```
    if c > 0 call sub2(c-1)
```

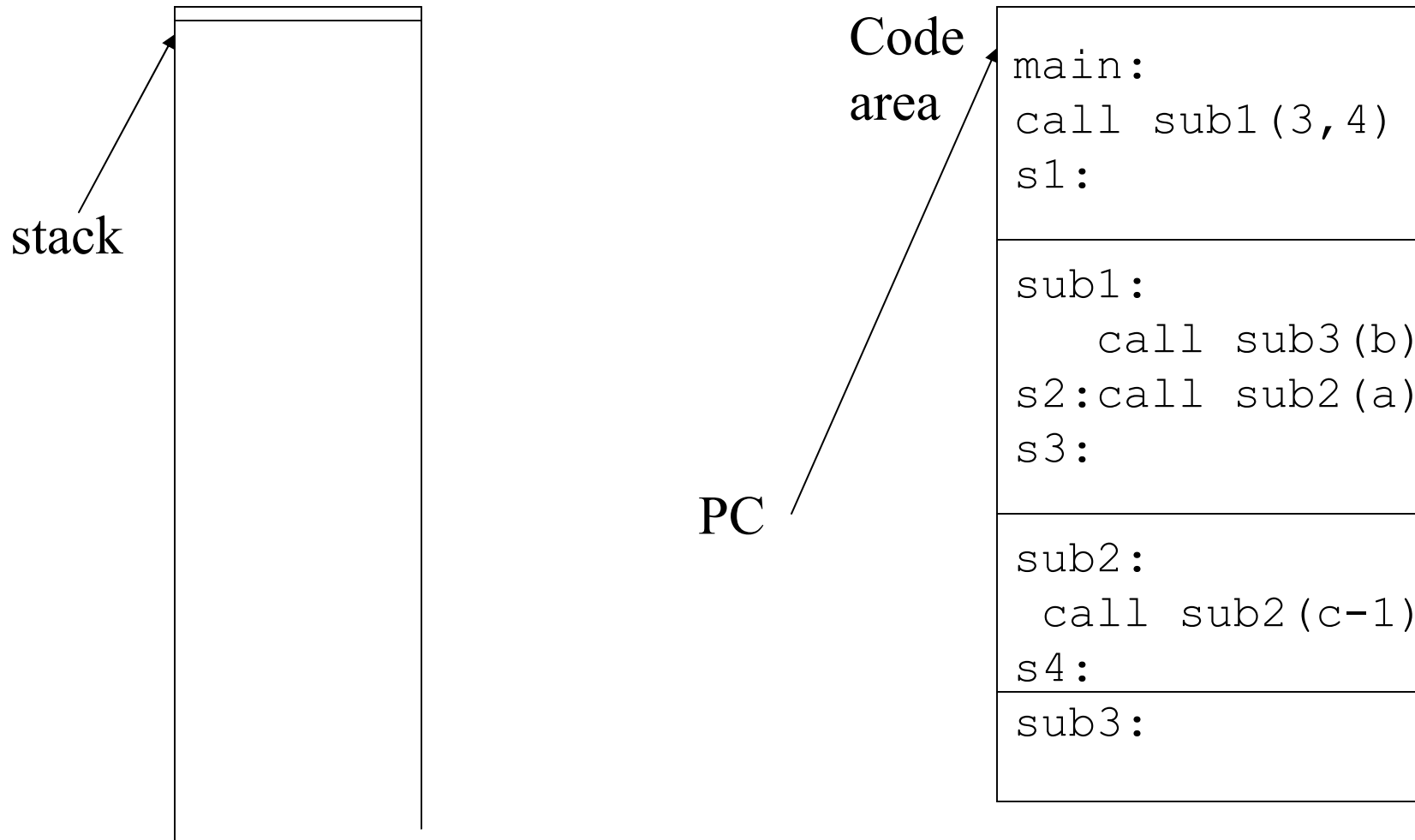
```
  procedure sub3()
```

```
    body of sub3
```

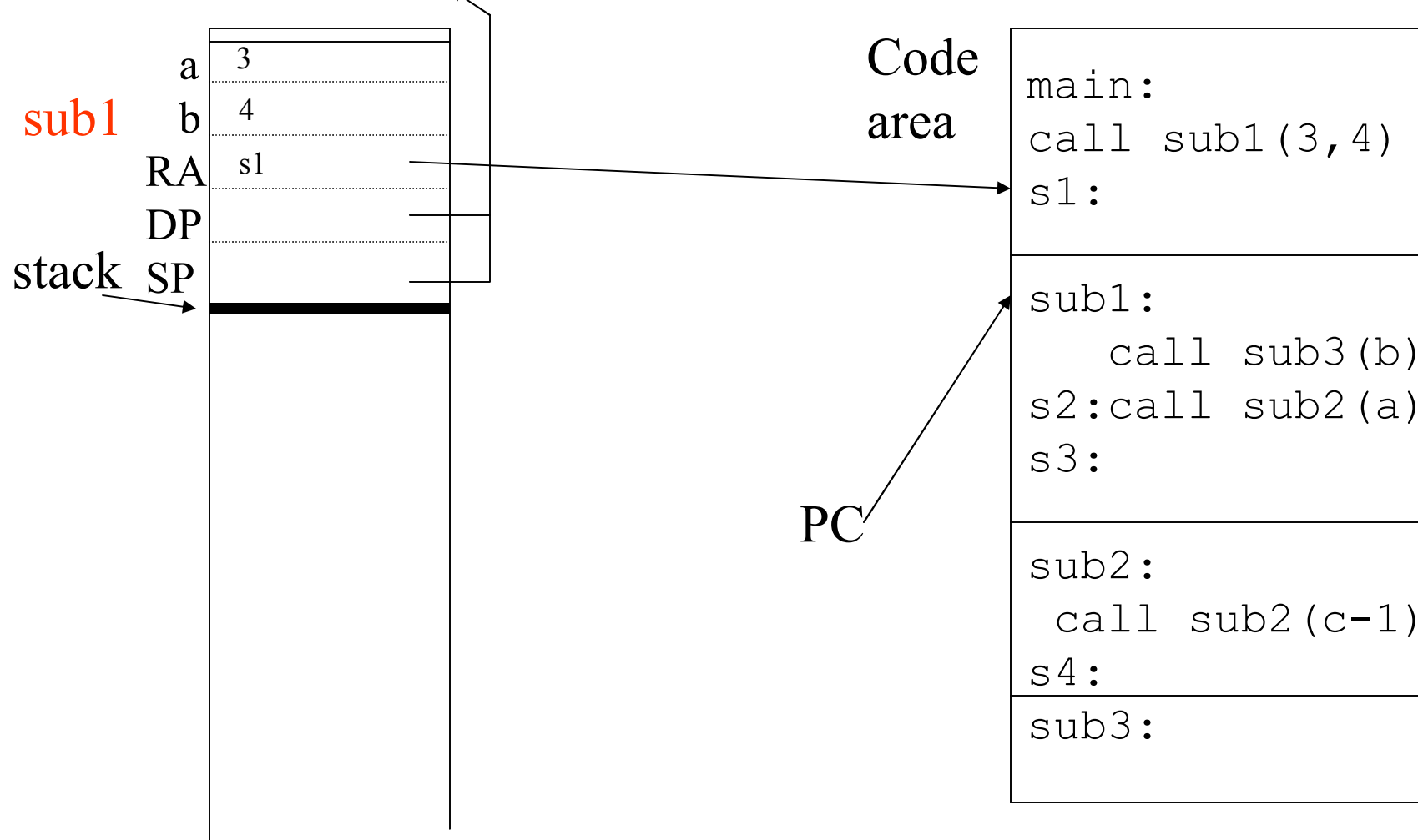
```
  call sub3(b); call sub2(a);
```

```
call sub1(3,4);
```

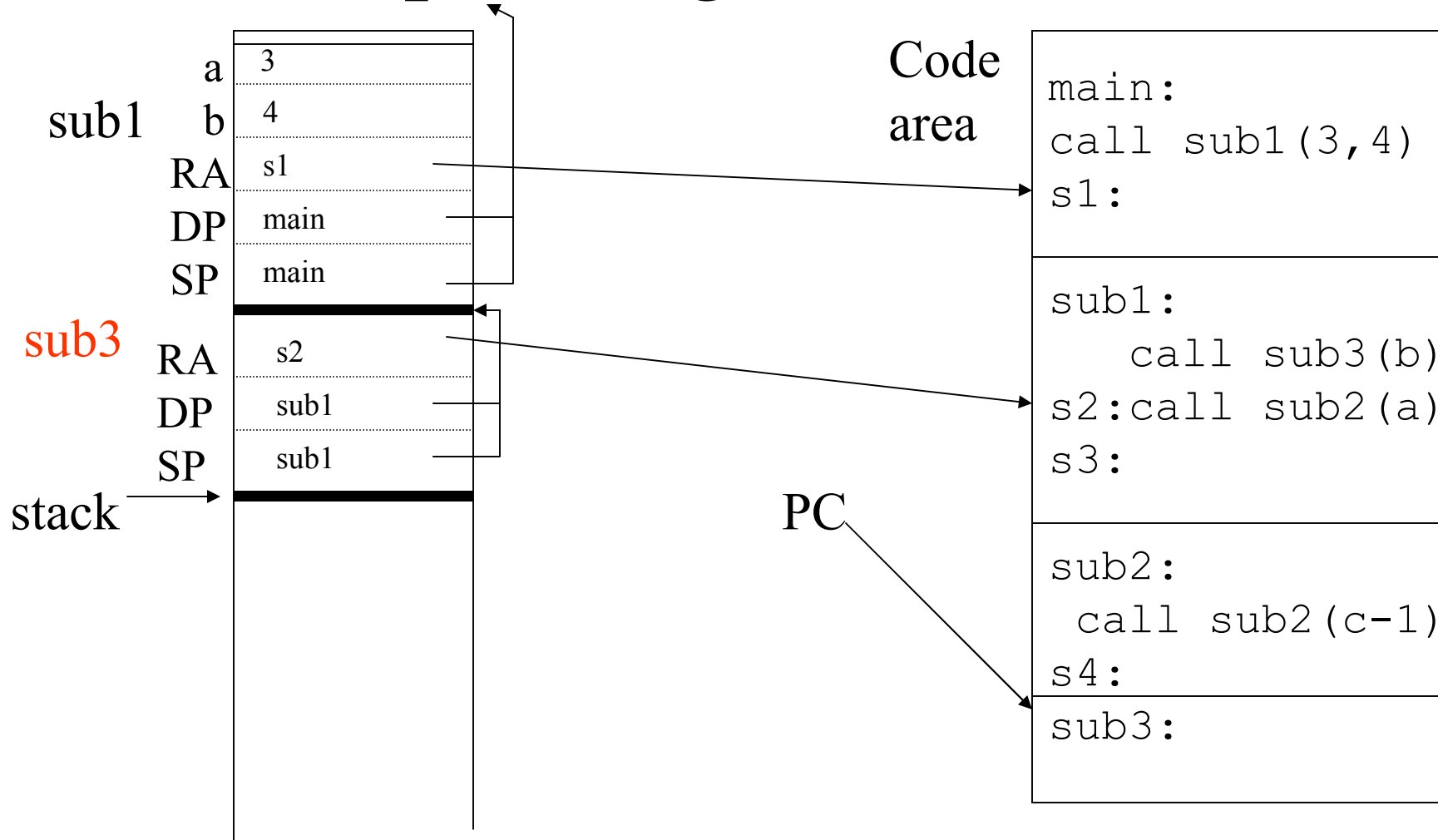
Example Program at runtime 1



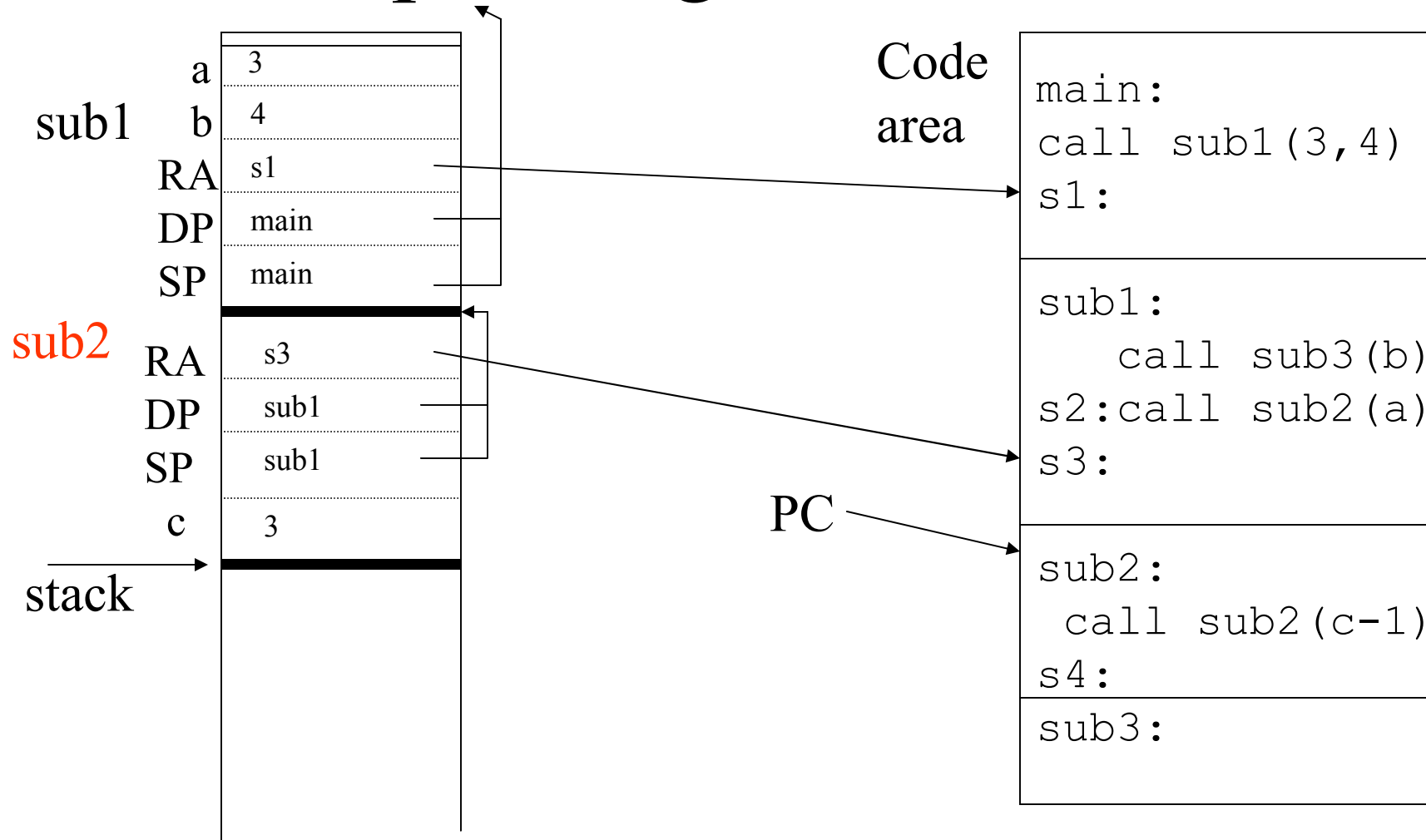
Example Program at runtime 2



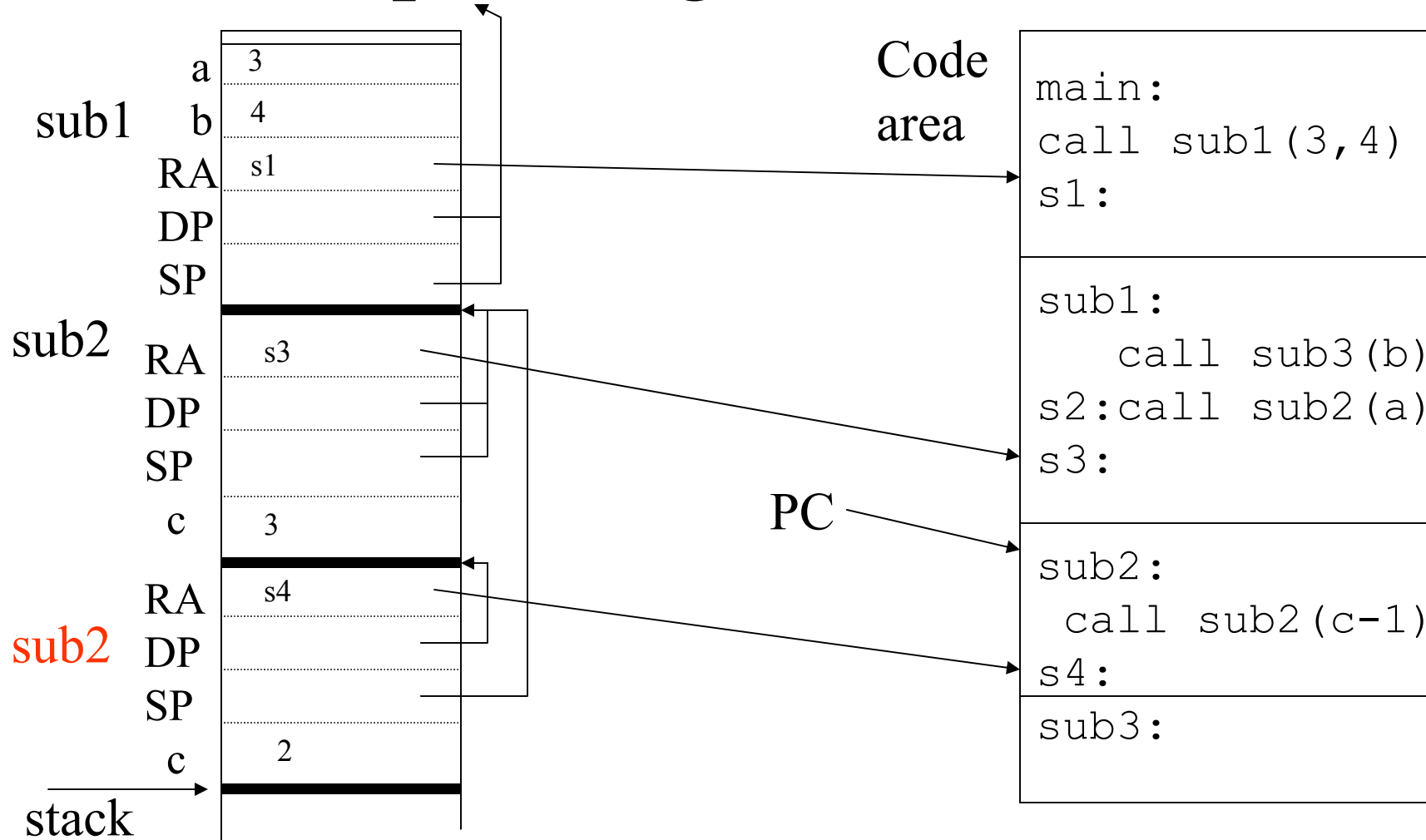
Example Program at runtime 3



Example Program at runtime 4



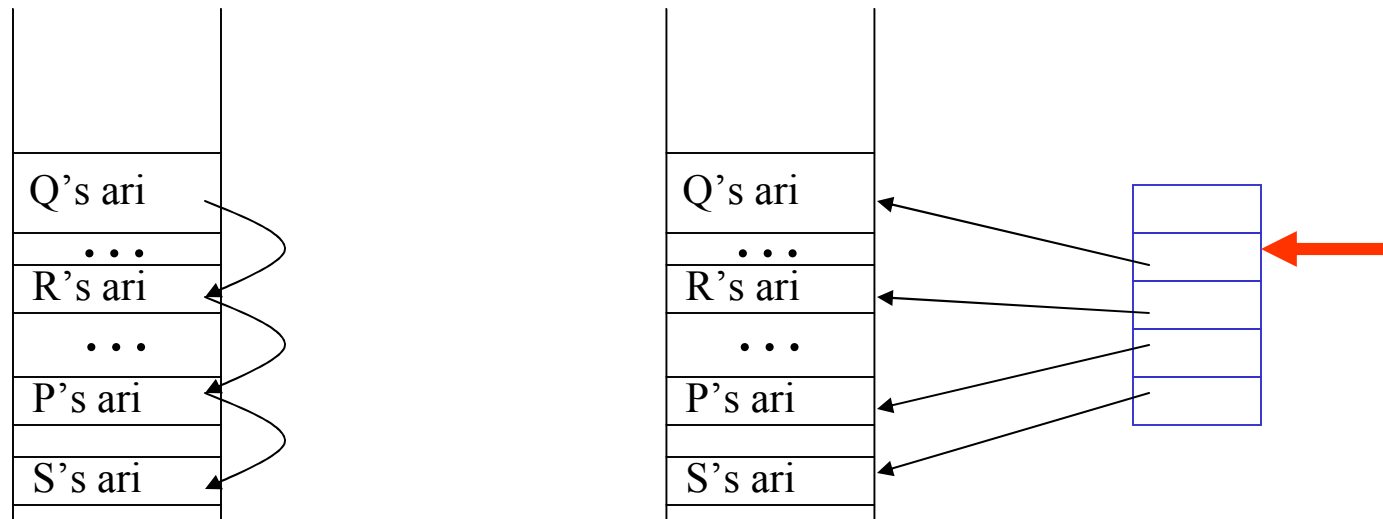
Example Program at runtime 5



Display

Alternate representation for access information.

- The current static chain kept in an array – entry n points to the activation record at level n in the static chain.

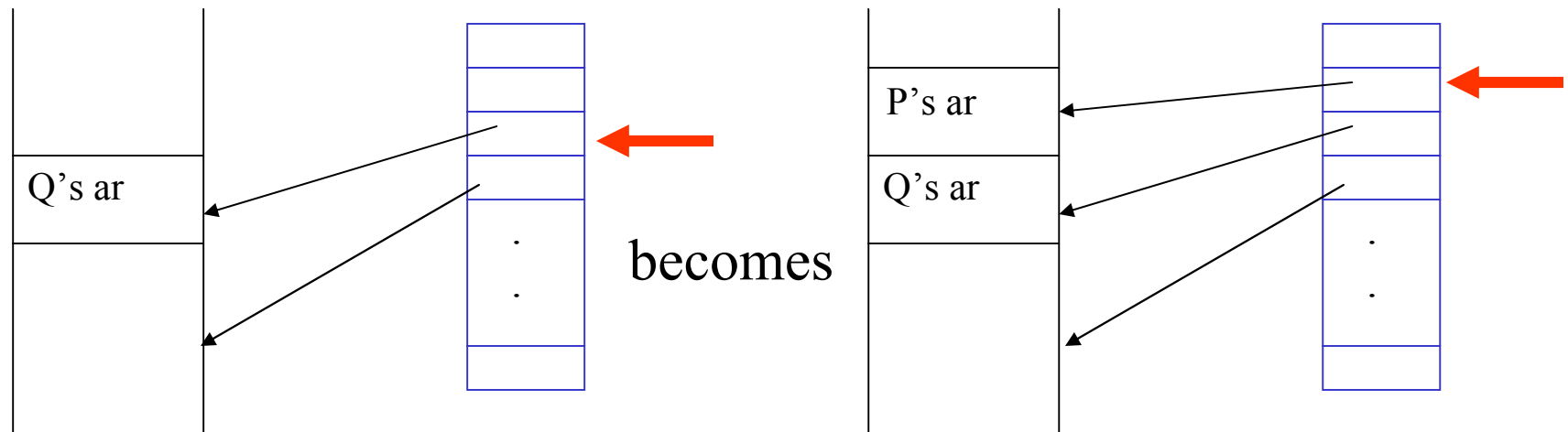


Display Maintenance

For Q calls P ...

$Q_{sd} < P_{sd}$ - P must be enclosed directly in Q

If entry n points at Q, entry n+1 points at P



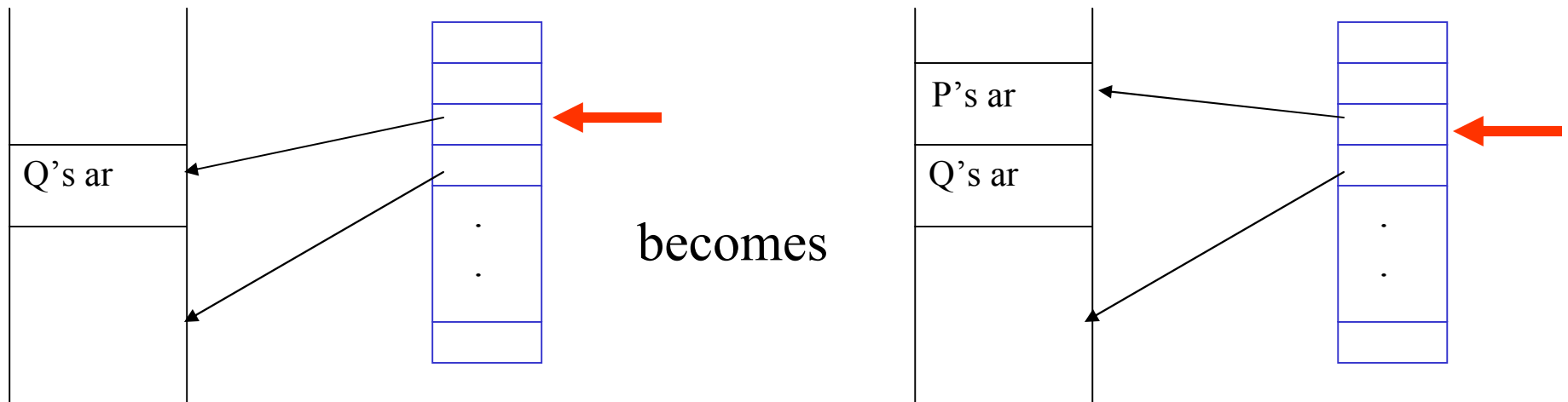
Display Maintenance

For Q calls P ...

$Q_{sd} = P_{sd}$ - They are at same static depth

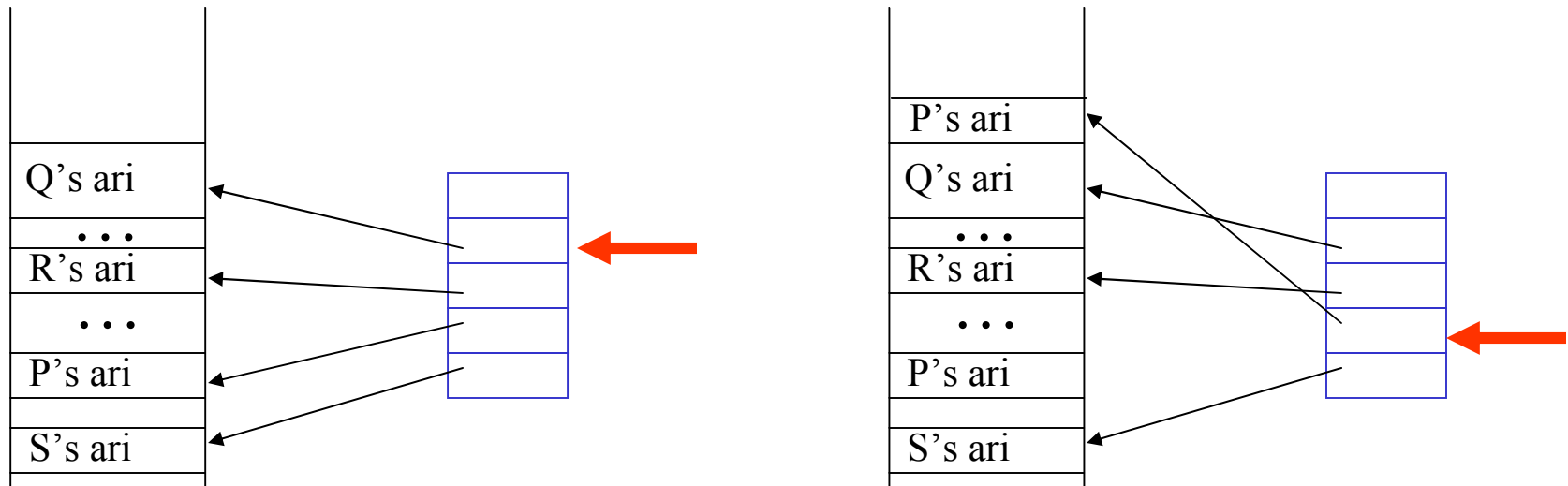
Update entry n to point to P

What happens when P ends? Must save the old entry n in P's ar to make things work properly.



Display Maintenance

$Q_{sd} > P_{sd}$ - Q is n levels down in the nesting –
update that pointer n levels down



Display

- Advantages: faster addressing
- Disadvantages: additional data structure to store and maintain.

Parameter Passing

Various approaches to passing data into and out of a procedure via parameters

1. **Call-by-value** – data is copied at the callee into activation and any item changes do not affect values in the caller.
2. **Call-by-reference** – pointer to data is placed in the callee activation and any changes made by the callee are indirect references to the actual value in the caller.
3. **Call-by-value-result** (copy-restore) – hybrid of call-by-value and call-by-reference. Data copied at the callee. During the call, changes do not affect the actual parameter. After the call, the actual value is updated.
4. **Call-by-name** – the actual parameter is in-line substituted into the called procedure. This means it is not evaluated until it is used.

Call-by-value

```
var a,b : integer
```

```
  procedure s (x,y : integer);
```

```
    var t: integer;
```

```
    begin t := x; x := y; y := t; end;
```

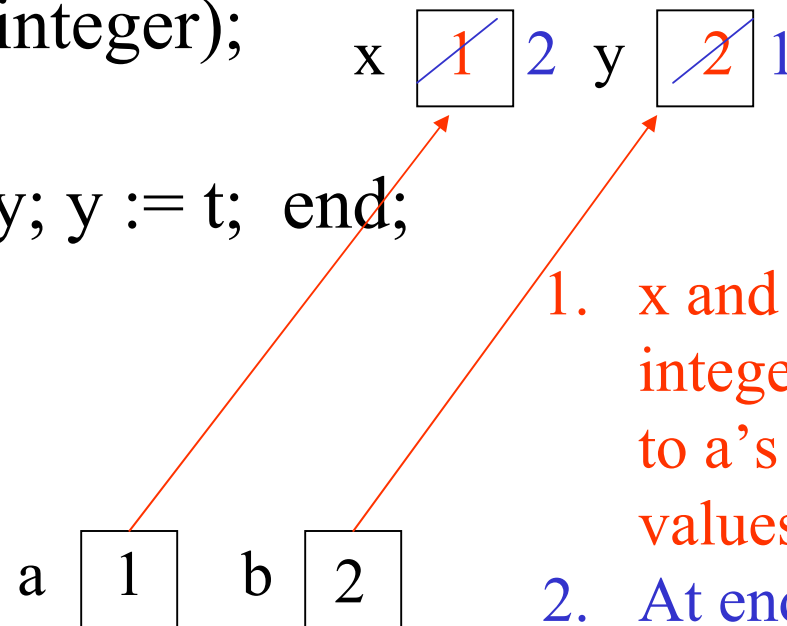
```
a := 1; b := 2;
```

```
s (a,b);
```

```
write ('a = ',a);
```

```
write ('b = ',b);
```

```
end.
```

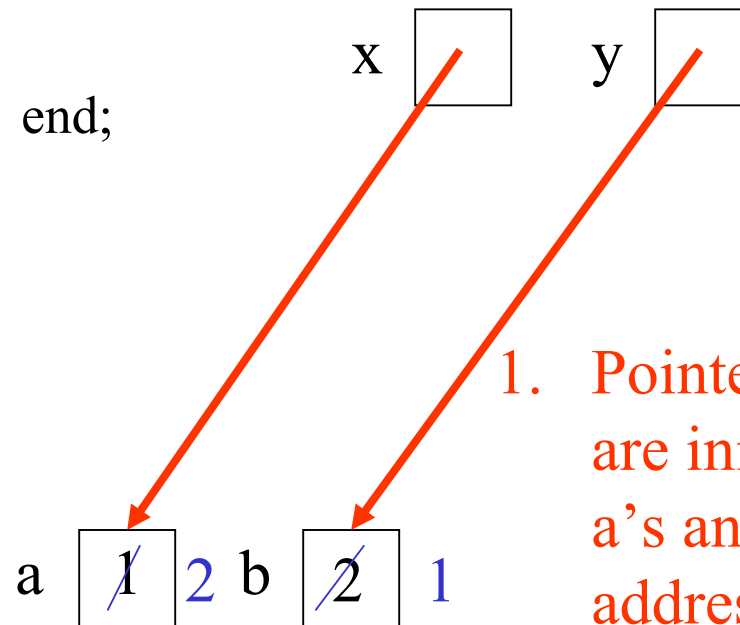


1. x and y are integers initialized to a's and b's values
2. At end of s, x = 2 and y = 1
3. No change to a and b on return

Call-by-reference

```
var a,b : integer
  procedure s (x,y : integer);
    var t: integer;
    begin t := x; x := y; y := t; end;
begin
a := 1; b := 2;
s (a,b);
write ('a = ',a);
write ('b = ',b);
end.
```

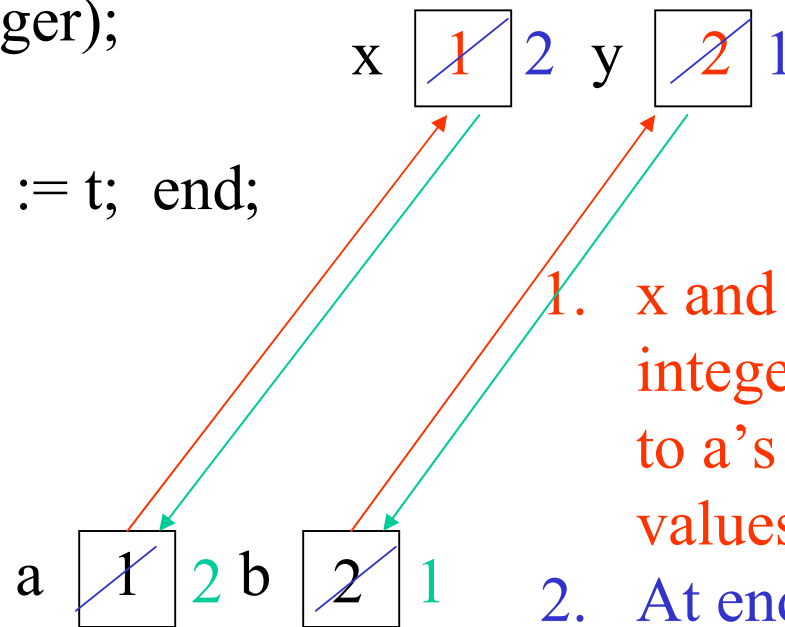
begin



1. Pointers x and y are initialized to a's and b's addresses
2. At end of s, x (and a) = 2 and y (and b) = 1

Call-by-value/result

```
var a,b : integer
  procedure s (x,y : integer);
    var t: integer;
    begin t := x; x := y; y := t; end;
begin
a := 1; b := 2;
s (a,b);
write ('a = ',a);
write ('b = ',b);
end.
```



1. x and y are integers initialized to a's and b's values
2. At end of s, x = 2 and y = 1
3. At return, a is given x's value and b is given y's value

Call-by-name

```
var a,b : integer
  procedure s (x,y : integer);
    var t: integer;
    begin t := x; x := y; y := t; end;
begin
a := 1; b := 2;
s (a,b);
write ('a = ',a);
write ('b = ',b);
end.
```

```
procedure s
var t: integer;
begin t := a; a := b; b := t; end;
```

begin

a ~~1~~ 2 b ~~2~~ 1

Call-by-value-result vs. Call-by-reference

```
var a: integer
  procedure foo(x: integer);
    begin a := a + 1; x := x + 1; end;
```

```
begin
```

```
a := 1;
```

```
foo(a);
```

```
write ('a = ',a);
```

```
end.
```

| | Value-result | reference |
|----------|--------------|-----------|
| write(a) | 2 | 3 |