DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Compilers
Design and Implementation

Run-Time Environments

Activation Records
Procedure Linkage
Name Translation and Variable Access

Language Issues

- Scope of Declaration
  - What are the Properties of a Name?
  - Where is it Visible?

$$\text{name} \xrightarrow{\text{environment}} \text{storage} \xrightarrow{\text{state}} \text{value}$$

- Binding of Names & Values
  - Environment binds Names to Storage
  - State binds Storage to Values

- Static vs Dynamic

STATIC	DYNAMIC
procedure definition	procedure activations
name declaration	name bindings
declaration scope	lifetime of binding

Procedure Abstraction

- What is a Procedure?
  - Basic Unit of Abstraction and Program Reasoning

- Why do We use Them?
  - To allow us to build (very) large Programs
  - Conceptually, allows us to abstract from all the details

- How to Generate Code?
  - Storage Allocation (bindings and lifetime of local variables)
  - Scoping, *i.e.*, what is visible and where?
  - Control Transfer (Call and Return)

The Procedure as a Control Abstraction

Procedures have well-defined Control-Flow

The Algol-60 Procedure Call

- Invoked at a Call Site, with some set of *Actual Parameters*
- Control returns to Call Site, immediately after Invocation

```
…
s = p(10,t,u);
…
```

```
int p(a,b,c)
   int a, b, c;
{
  int   d;
  d = q(c,b);
  …
}
```

caller **callee**

The Procedure as a Control Abstraction

Procedures have well-defined Control-Flow

The Algol-60 Procedure Call

- Invoked at a Call Site, with some set of *Actual Parameters*
- Control returns to Call Site, immediately after Invocation

```
…
s = p(10,t,u);
…
```

```
int p(a,b,c)
   int a, b, c;
{
  int   d;
  d = q(c,b);
  ...
}
```

```
int q(x,y)
   int x,y;
{
   return x + y;
}
```

The Procedure as a Control Abstraction

Procedures have well-defined Control-Flow

The Algol-60 Procedure Call

- Invoked at a Call Site, with some set of *Actual Parameters*
- Control returns to Call Site, immediately after Invocation

```
…
s = p(10,t,u);
…
```

```
int p(a,b,c)
   int a, b, c;
{
  int   d;
  d = q(c,b);
  ...
}
```

```
int q(x,y)
   int x,y;
{
   return x + y;
}
```

The Procedure as a Control Abstraction

Procedures have well-defined Control-Flow

The Algol-60 Procedure Call

- Invoked at a Call Site, with some set of *Actual Parameters*
- Control returns to Call Site, immediately after Invocation

```
…
s = p(10,t,u);
…
```

```
int p(a,b,c)
   int a, b, c;
{
  int   d;
  d = q(c,b);
  …
}
```

```
int q(x,y)
   int x,y;
{
   return x + y;
}
```

The Procedure as a Control Abstraction

Procedures have well-defined Control-Flow

The Algol-60 Procedure Call

- Invoked at a Call Site, with some set of *Actual Parameters*
- Control returns to Call Site, immediately after Invocation

```
…
s = p(10,t,u);
…
```

```
int p(a,b,c)
   int a, b, c;
{
  int   d;
  d = q(c,b);
  …
}
```

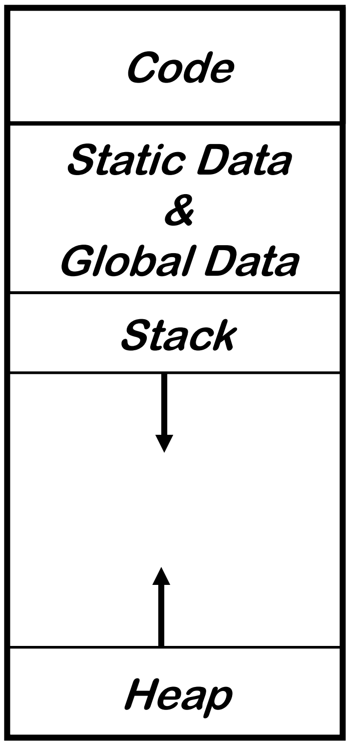
```
int q(x,y)
   int x,y;
{
   return x + y;
}
```

- Most Languages Allow Recursion

Compilation Issues

- How to Generate Code
  - Allocate/Deallocate Storage for Local Variables
  - Transfer of Arguments and Return Results
- How to Execute a Procedure
  - How to Access Local and Non-Local Variables
  - How to Communicate between Caller and Callee
  - How to Transfer Control between Caller and Callee
- The Role of the Symbol Table
  - Keep track of where Names are Defined and Declared
  - Scope and Lifetime

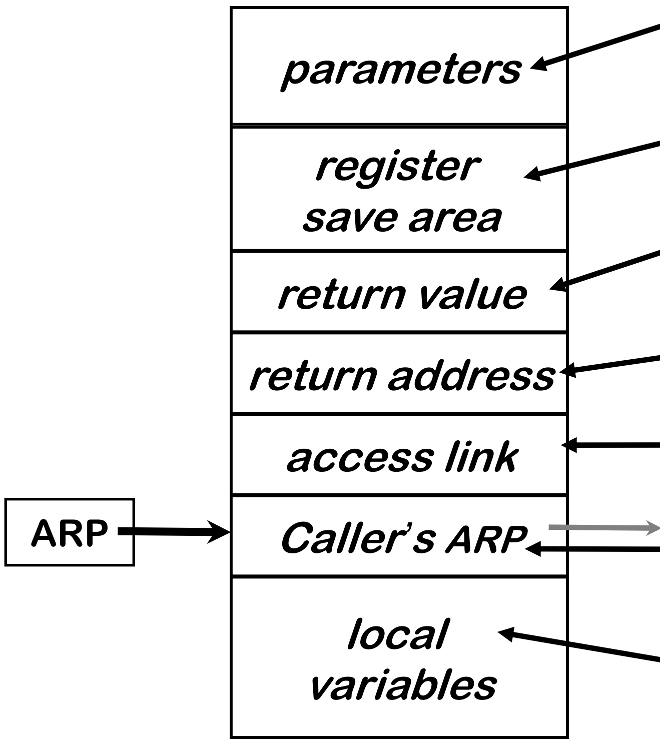
Run-Time Storage Organization



Classical Organization

- Code, Static, & Global Data have known size
  - Use symbolic labels in the code
- Heap & Stack grow and shrink over time
  - Stack used for Activation Records (AR)
  - Heap for Data (including AR) whose lifetime extends beyond activation.
- This is a virtual Address Space

Activation Record Basics



Space for parameters to the called/current routine

Saved register contents

If function, space for the return value

Address to resume Caller

Help with non-local access

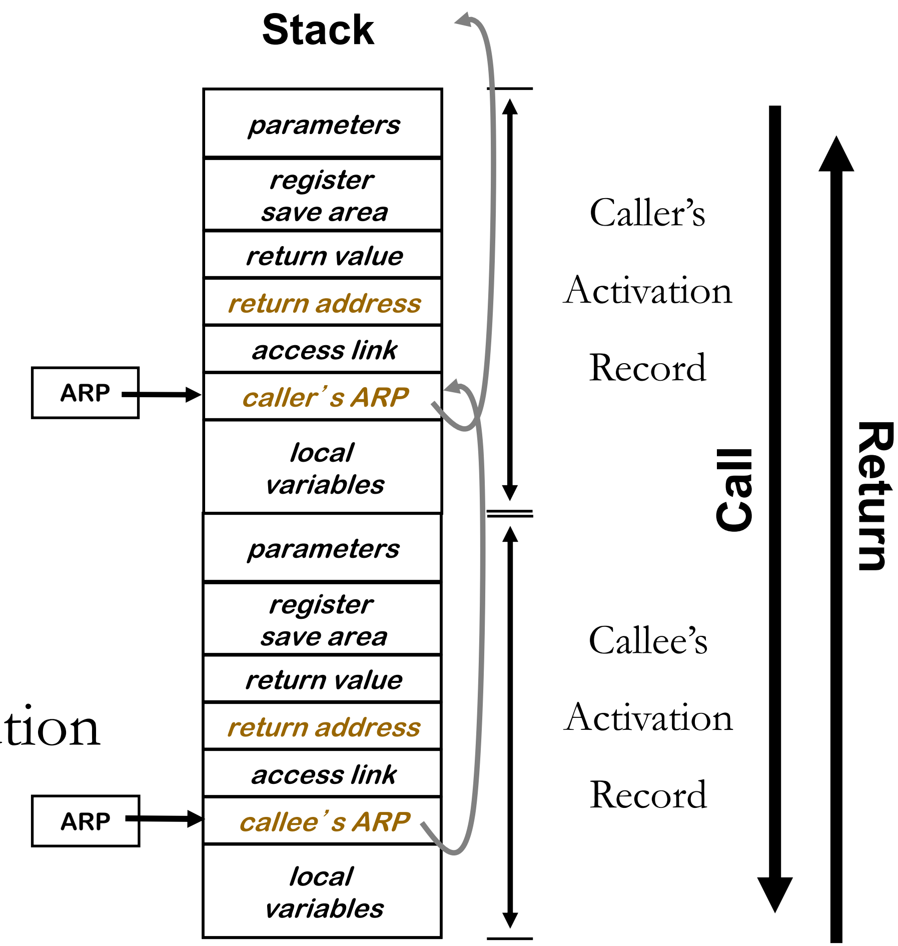
To restore Caller's AR on a return

Space for local values & variables (including spills)

One **AR** for each Invocation of a Procedure

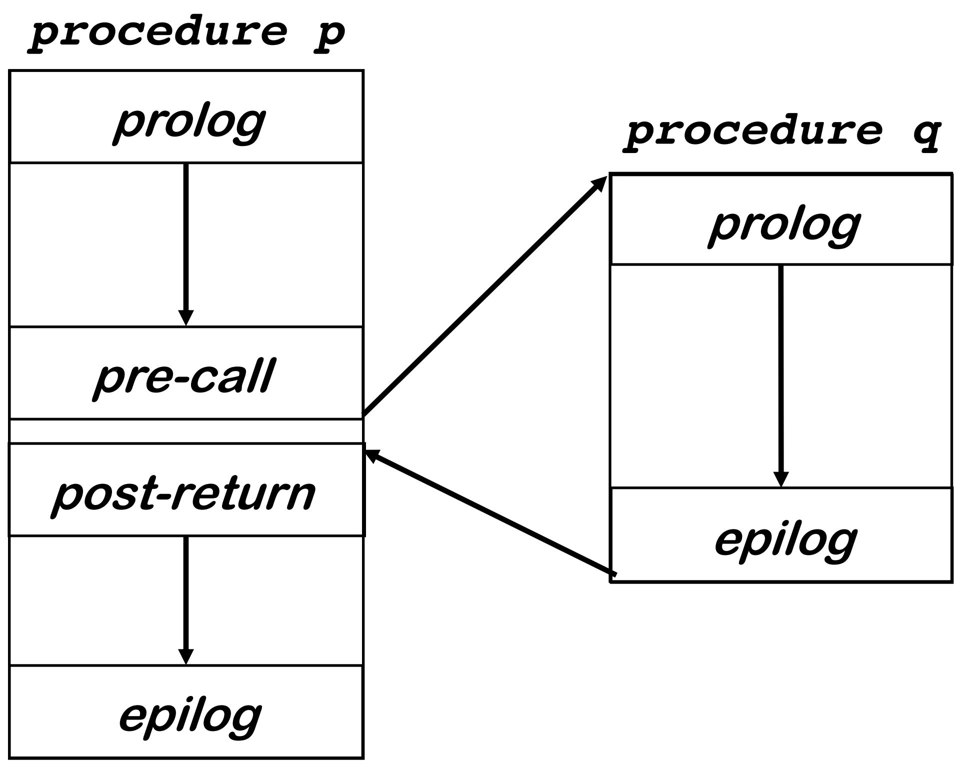
Activation Records on the Stack

- What Happens on a Call?
  - Passing of Arguments
  - Transfer of Control
- What Happens on a Return?
  - Recovery of Results (if any)
  - Transfer of Control
- Need to Save/Restore Execution Context?
  - ARP, PC, Access Link,
  - Register Values



Procedure Linkages

Standard procedure linkage



Procedure has

- standard **prolog**
- standard **epilog**

Each call involves a

- **pre-call** sequence
- **post-return** sequence

These are completely predictable from the Call Site as they depend on the number & type of the actual parameters

Procedure Linkages

Pre-Call Sequence

- Sets up Callee's basic AR
- Helps preserve its own environment

The Details

- Allocate Space for the Callee's AR
  - except space for local variables
- Evaluates each Parameter & Stores Value or Address
- Saves Return Address, caller's ARP into Callee's AR
- If Access Links are used
  - Find appropriate lexical ancestor & copy into Callee's **AR**
- Save any Caller-save Registers
  - Save into space in Caller's AR
- Jump to Address of Callee's prolog code

Procedure Linkages

Post-Return Sequence

- Finish restoring Caller's environment
- Place any value back where it belongs

The Details

- Copy return value from Callee's AR, if necessary
- Free the Callee's AR
- Restore any caller-save registers
- Restore any call-by-reference parameters to registers, if needed
  - Also copy back call-by-value/result parameters
- Continue execution after the call

Procedure Linkages

Prolog Code

- Finish setting up the Callee's environment
- Preserve parts of the Caller's environment that will be disturbed

The Details

- Preserve any Callee-save registers
- If *Display* is being used
  - Save display entry for current lexical level
  - Store current ARP into display for current lexical level
- Allocate Space for Local Data
  - Easiest scenario is to extend the AR
- Find any Static Data areas referenced in the Callee
- Handle any Local Variable Initializations

With heap allocated AR, may need to use a separate heap object for local variables

Procedure Linkages

Epilog Code

- Wind up the business of the Callee
- Start restoring the Caller's Environment

If ARs are stack allocated, this may not be necessary. (Caller can reset stack top to its pre-call value.)

The Details

- Store Return Value? No, this happens on the return statement
- Restore Callee-save Registers
- Free space for Local Data, if necessary (on the Heap)
- Load Return Address from AR
- Restore caller's ARP
- Jump to the Return Address

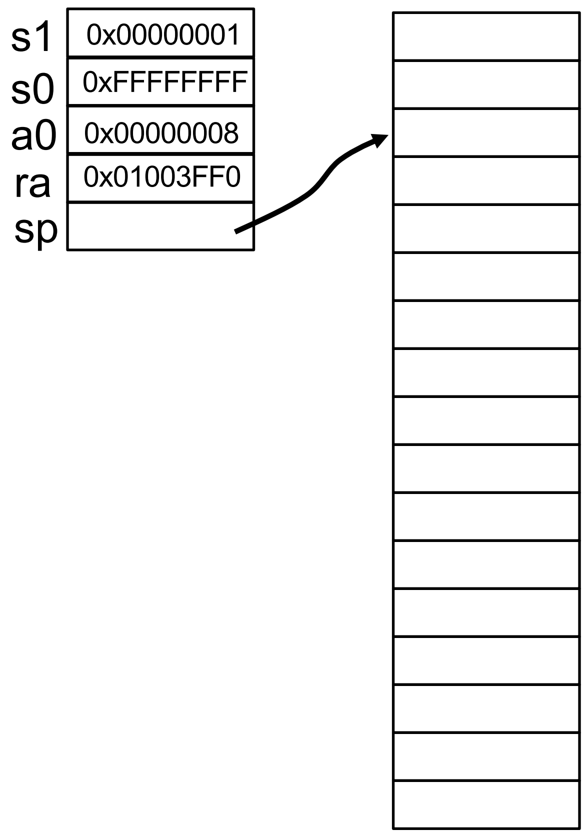
Caller-saved vs Callee-saved Registers

- **Caller-saved registers** (**volatile** registers, or **call-clobbered**)
  - Hold temporary quantities that need not be preserved across Calls.
  - Caller's responsibility to push these registers onto the stack or copy them somewhere else *if* it wants to restore this value after the call.
  - Expected callee to ***destroy*** temporary values in these registers…

- **Callee-saved registers** (**non-volatile** registers, or **call-preserved**)
  - Used to hold long-lived values that should be preserved across Calls.
  - Callee's responsibility to push then onto the stack or copy them somewhere else *if* it wants to restore this value after the call.
  - Expected callee to ***preserve*** (not destroy) temporary values in these registers…

Generated Assembly Code (some)

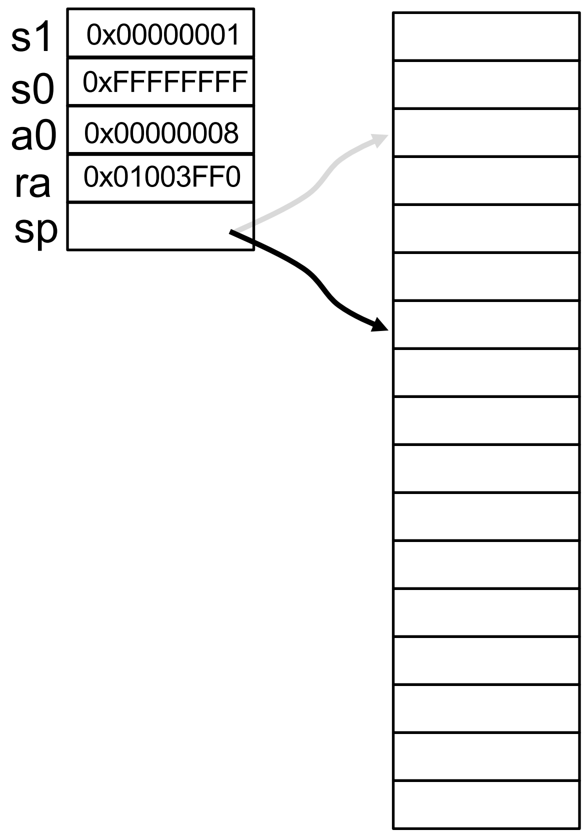
```
my_function:
   # Prologue
   addi sp, sp, -32
   sd    ra,  0(sp)
   sd    a0, 8(sp)
   sd    s0, 16(sp)
   sd    s1, 24(sp)

   # Epilogue
   ld     ra,  0(sp)
   ld     a0, 8(sp)
   ld     s0, 16(sp)
   ld     s1, 24(sp)
   addi sp, sp, 32
   ret
```



- Prologue:
  - "allocate" 32 bytes on the stack
  - save return address
  - save callee-saved registers

Generated Assembly Code (some)

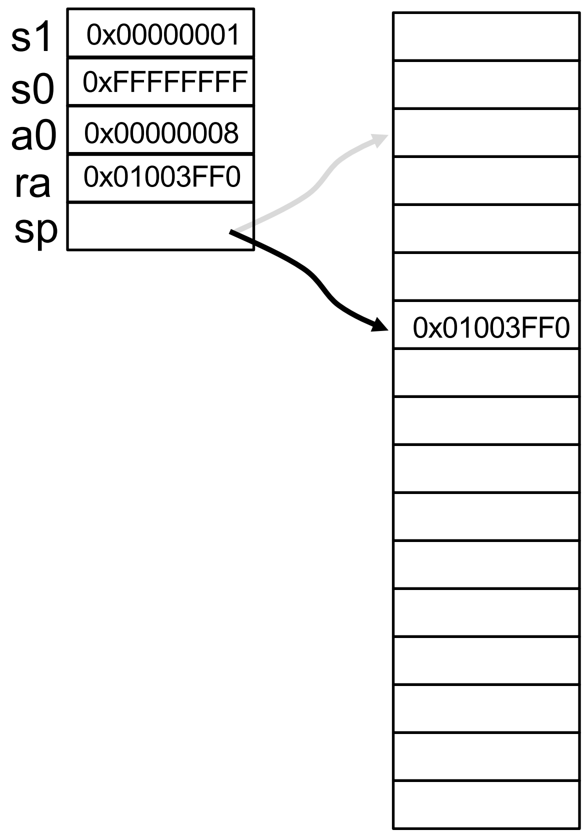
```
my_function:
   # Prologue
   addi sp, sp, -32
   sd   ra,  0(sp)
   sd   a0, 8(sp)
   sd   s0, 16(sp)
   sd   s1, 24(sp)

   # Epilogue
   ld   ra,  0(sp)
   ld   a0, 8(sp)
   ld   s0, 16(sp)
   ld   s1, 24(sp)
   addi sp, sp, 32
   ret
```



- Prologue:
  - “allocate” 32 bytes on the stack
  - save return address
  - save callee-saved registers

Generated Assembly Code (some)

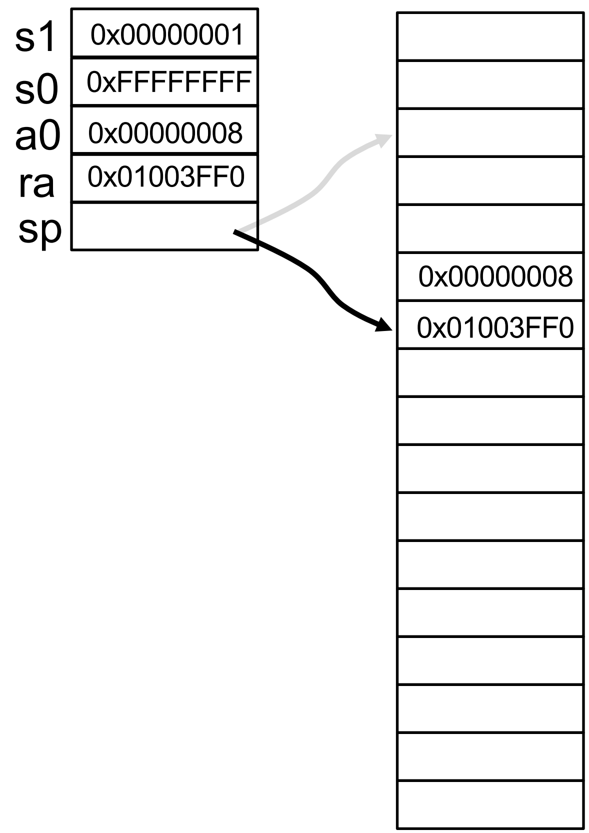
```
my_function:
    # Prologue
    addi sp, sp, -32
    sd   ra,  0(sp)
    sd   a0, 8(sp)
    sd   s0, 16(sp)
    sd   s1, 24(sp)

    # Epilogue
    ld   ra,  0(sp)
    ld   a0, 8(sp)
    ld   s0, 16(sp)
    ld   s1, 24(sp)
    addi sp, sp, 32
    ret
```



- Prologue:
  - "allocate" 32 bytes on the stack
  - save return address
  - save callee-saved registers

Generated Assembly Code (some)

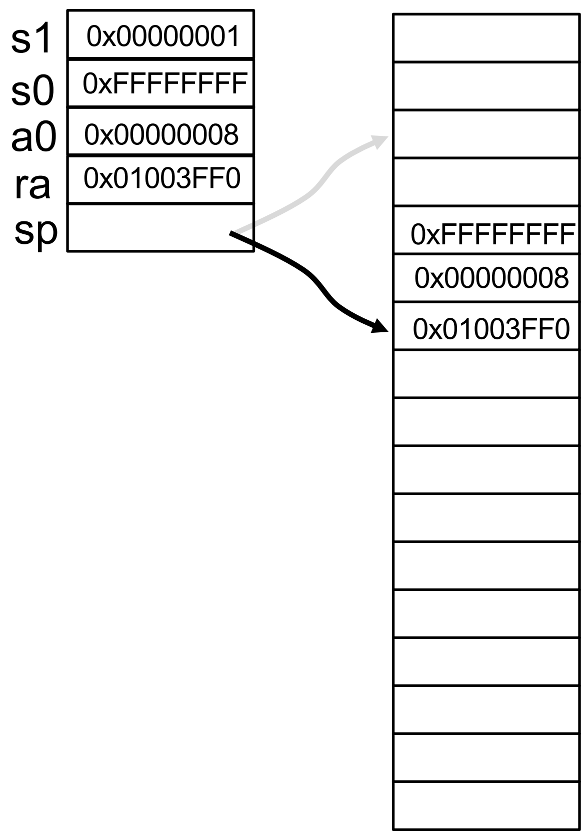
```
my_function:
   # Prologue
   addi sp, sp, -32
   sd   ra,  0(sp)
   sd   a0, 8(sp)
   sd   s0, 16(sp)
   sd   s1, 24(sp)

   # Epilogue
   ld   ra,  0(sp)
   ld   a0, 8(sp)
   ld   s0, 16(sp)
   ld   s1, 24(sp)
   addi sp, sp, 32
   ret
```



- Prologue:
  - "allocate" 32 bytes on the stack
  - save return address
  - save callee-saved registers

Generated Assembly Code (some)

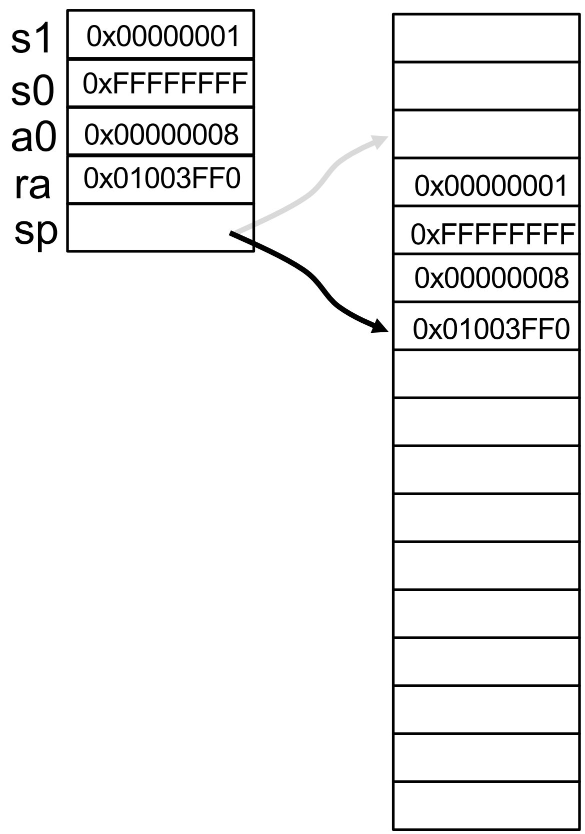
```
my_function:
   # Prologue
   addi sp, sp, -32
   sd    ra,  0(sp)
   sd    a0, 8(sp)
   sd    s0, 16(sp)
   sd    s1, 24(sp)

   # Epilogue
   ld    ra,  0(sp)
   ld    a0, 8(sp)
   ld    s0, 16(sp)
   ld    s1, 24(sp)
   addi sp, sp, 32
   ret
```



- Prologue:
  - "allocate" 32 bytes on the stack
  - save return address
  - save callee-saved registers

Generated Assembly Code (some)

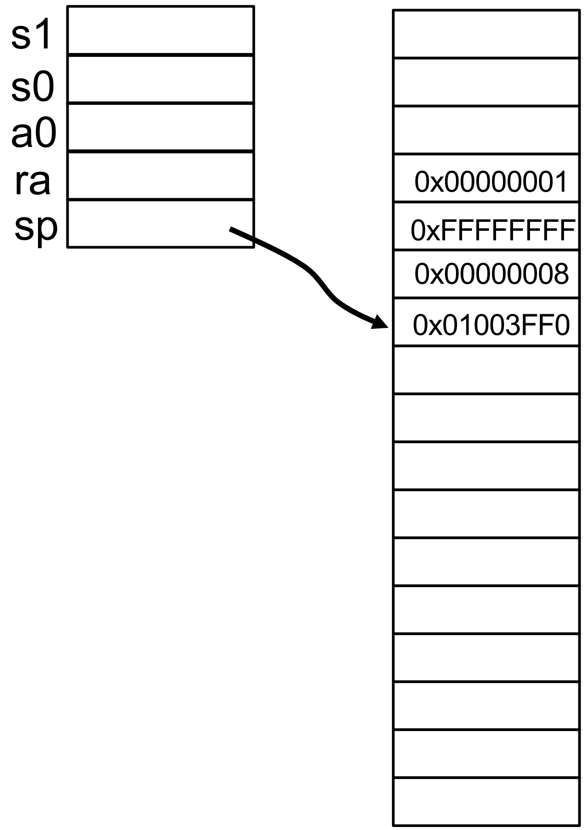
```
my_function:
  # Prologue
  addi sp, sp, -32
  sd   ra,  0(sp)
  sd   a0, 8(sp)
  sd   s0, 16(sp)
  sd   s1, 24(sp)

  # Epilogue
  ld   ra,  0(sp)
  ld   a0, 8(sp)
  ld   s0, 16(sp)
  ld   s1, 24(sp)
  addi sp, sp, 32
  ret
```



- Prologue:
  - "allocate" 32 bytes on the stack
  - save return address
  - save callee-saved registers

Generated Assembly Code (some)

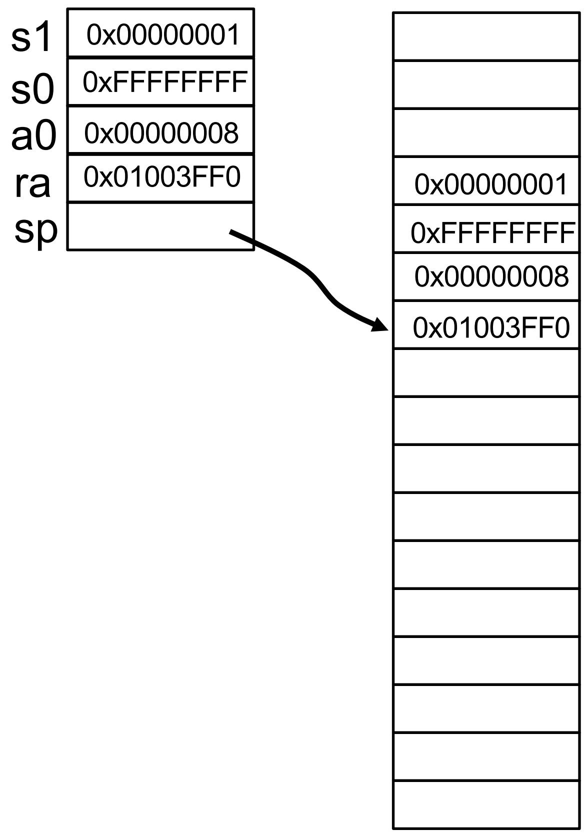
```
my_function:
  # Prologue
  addi sp, sp, -32
  sd   ra,  0(sp)
  sd   a0, 8(sp)
  sd   s0, 16(sp)
  sd   s1, 24(sp)

  # Epilogue
  ld   ra,  0(sp)
  ld   a0, 8(sp)
  ld   s0, 16(sp)
  ld   s1, 24(sp)
  addi sp, sp, 32
  ret
```



- Prologue:
  - "allocate" 32 bytes on the stack
  - save return address
  - save callee-saved registers
- Epilogue:
  - restore return address
  - restore callee-saved registers
  - "deallocate" 32 bytes off the stack

Generated Assembly Code (some)

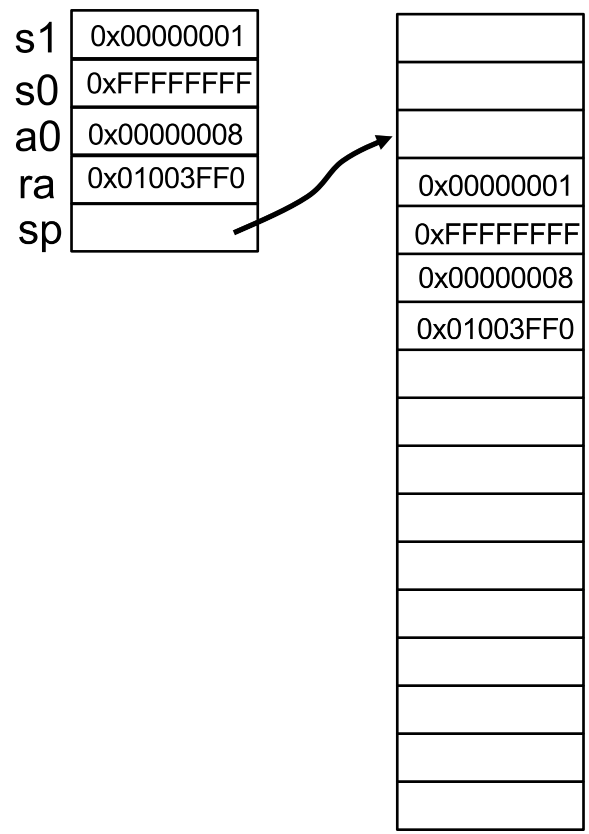
```
my_function:
   # Prologue
   addi sp, sp, -32
   sd   ra,  0(sp)
   sd   a0, 8(sp)
   sd   s0, 16(sp)
   sd   s1, 24(sp)

   # Epilogue
   ld   ra,  0(sp)
   ld   a0, 8(sp)
   ld   s0, 16(sp)
   ld   s1, 24(sp)
   addi sp, sp, 32
   ret
```



- Prologue:
  - "allocate" 32 bytes on the stack
  - save return address
  - save callee-saved registers
- Epilogue:
  - restore return address
  - restore callee-saved registers
  - "deallocate" 32 bytes off the stack

Generated Assembly Code (some)

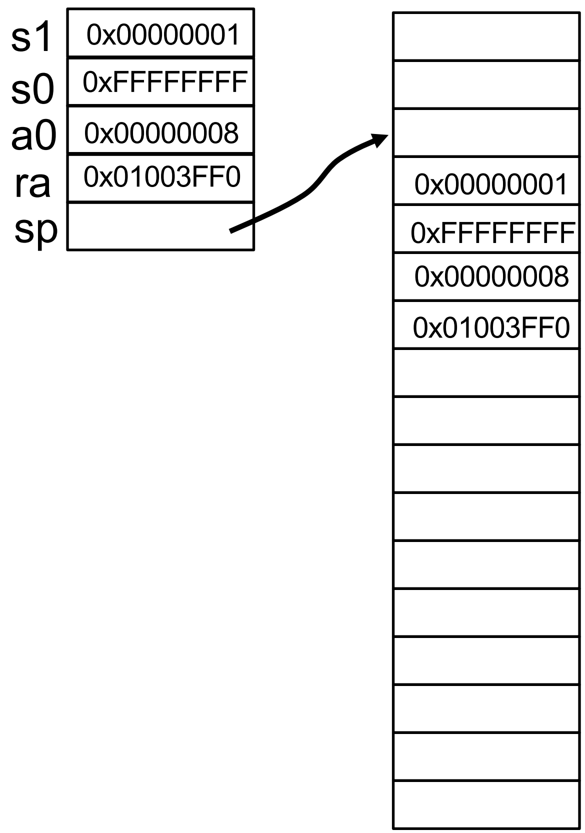
```
my_function:
   # Prologue
   addi sp, sp, -32
   sd    ra,  0(sp)
   sd    a0, 8(sp)
   sd    s0, 16(sp)
   sd    s1, 24(sp)

   # Epilogue
   ld     ra,  0(sp)
   ld     a0, 8(sp)
   ld     s0, 16(sp)
   ld     s1, 24(sp)
   addi sp, sp, 32
   ret
```



- Prologue:
  - "allocate" 32 bytes on the stack
  - save return address
  - save callee-saved registers
- Epilogue:
  - restore return address
  - restore callee-saved registers
  - "deallocate" 32 bytes off the stack

Generated Assembly Code (some)

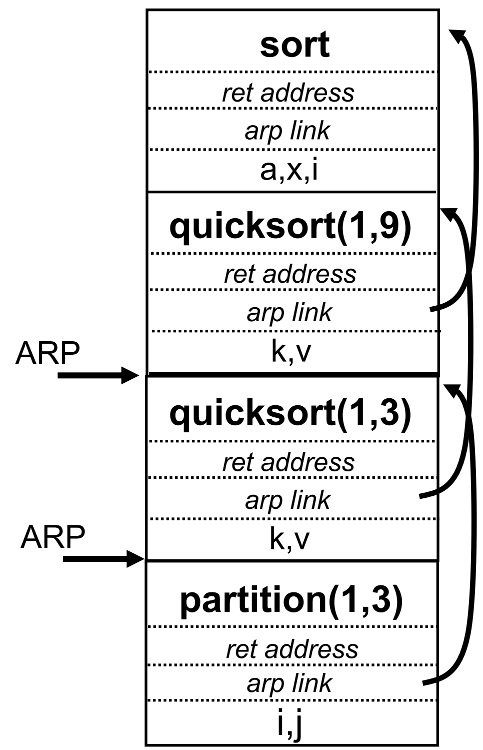
```
my_function:
   # Prologue
   addi sp, sp, -32
   sd    ra,  0(sp)
   sd    a0, 8(sp)
   sd    s0, 16(sp)
   sd    s1, 24(sp)

   # Epilogue
   ld    ra,  0(sp)
   ld    a0, 8(sp)
   ld    s0, 16(sp)
   ld    s1, 24(sp)
   addi sp, sp, 32
   ret
```



- Prologue:
  - "allocate" 32 bytes on the stack
  - save return address
  - save callee-saved registers
- Epilogue:
  - restore return address
  - restore callee-saved registers
  - "deallocate" 32 bytes off the stack
- Return:
  - ret ≡ jalr x0, x1, 0
  - where x1 is ra and x0 is zero
  - So loads pc with ra …

Simplified Example

```
program sort(input, output);
var a: array [0..10] of integer;
     x, i: integer;
procedure readarray;
var i : integer;
begin … a… end { readarray } ;
procedure exchange(i, j: integer);
   begin
     x := a[i]; a[i] := a[j]; a[j] := x;
   end { exchange } ;
procedure quicksort(m, n : integer);
  var k, v: integer;
  function partition(y, z : integer) : integer;
   var i, j : integer;
   begin … a …
           … v …
            … exchange(i,j); …
   end { partition }
  begin … end { quicksort }
begin … end { sort }
```

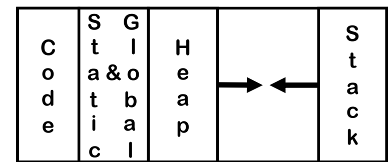


Activation Record Details

Where do Activation Records live?

- If lifetime of AR matches lifetime of invocation, *AND*
- If code normally executes a "return"

⇒ Keep ARs on a stack



Yes! This stack.

- If a procedure can outlive its caller, *OR*
- If it can return an object that can reference its execution state

⇒ ARs <u>must</u> be kept in the Heap

- If a procedure makes no Calls

⇒ AR can be allocated statically

Efficiency prefers Static, Stack, then Heap

Activation Record Details

How does the Compiled Code finds the Variables?

- They are at known offsets from the AR pointer
- Code nesting level and AR offset within a procedure
  - **Level** specifies an ARP, offset is the constant (*later*…)

Variable-Length Data

- If AR can be extended, put it after Local Variables
- Leave a pointer at a known offset from ARP
- Otherwise, put variable-length data on the Heap

Initializing Local Variables

- Must generate explicit Code to Store the Values
- Among the procedure's first actions

Storage for Blocks within a Single Procedure

```
B0: {
        int a, b, c
B1:   {
          int v, b, x, w
B2:       {
              int x, y, z
              ….
          }
B3:       {
              int x, a, v
              …
          }
          …
      }
      …
    }
```

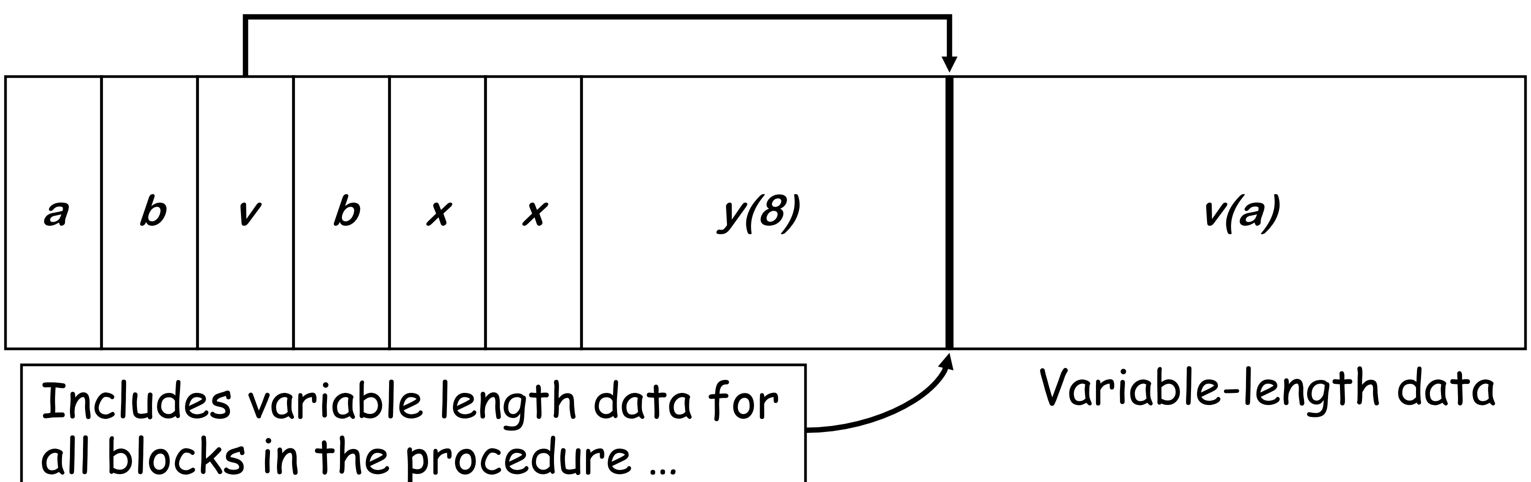
- Fixed-length data can always be at a constant offset from the beginning of a procedure
  - In our example, the *a* declared at level 0 will always be the first data element, stored at byte 0 in the fixed-length data area
  - The ***x*** declared at level 1 will always be the sixth data item, stored at byte 20 in the fixed data area
  - The ***x*** declared at level 2 will always be the eighth data item, stored at byte 28 in the fixed data area
  - But what about the *a* declared in the second block at level 2?

Variable-Length Data

```
B0: {
        int a, b
        … assign value to a
B1:     {
            int v(a), b, x
B2:         {
                int x, y(8)
                ….
            }
        }
    }
```

Arrays

- → If size is fixed at compile time, store in fixed-length data area
- → If size is variable, store **descriptor** in fixed length area, with pointer to variable length area
- → Variable-length data area is assigned at the end of the fixed length area for block in which it is allocated



Translating Local Names

How does the compiler represent a specific instance of *x* ?

- Name is translated into a ***static coordinate***
  - < *level,offset* > pair
  - "*level*" is lexical nesting level of the procedure
  - "*offset*" is *unique* within that scope
- Subsequent code will use the static coordinate to generate addresses and references
- "*level*" is a function of the table in which *x* is found
  - Stored in the entry for each *x*
- "*offset*" must be assigned and stored in the Symbol Table
  - Assigned at Compile time
  - Known at Compile time
  - Used to Generate code that executes at run-time

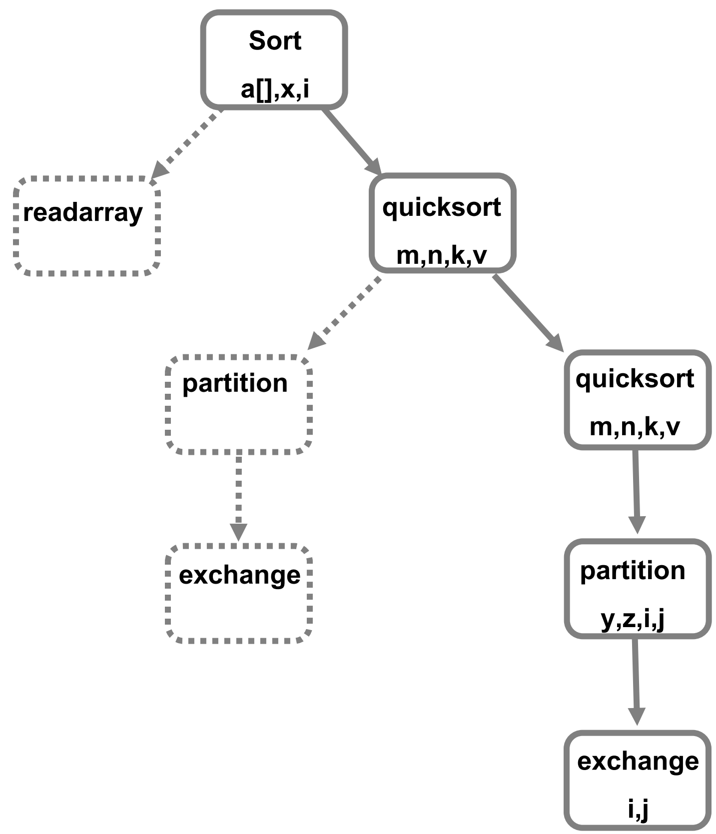
Scoping Rules

- Scoping
  - Define which instance each name refers to
- Lexical Scoping
  - Look at the source text of the code
  - Determine the closest (nesting structure) name
  - Ex. FORTRAN, C, Pascal.
- Dynamic Scoping
  - Check at Run-Time the closest variable with the same name
  - Ex. Scheme, Lisp, Miranda, etc.

Lexical Scoping Example

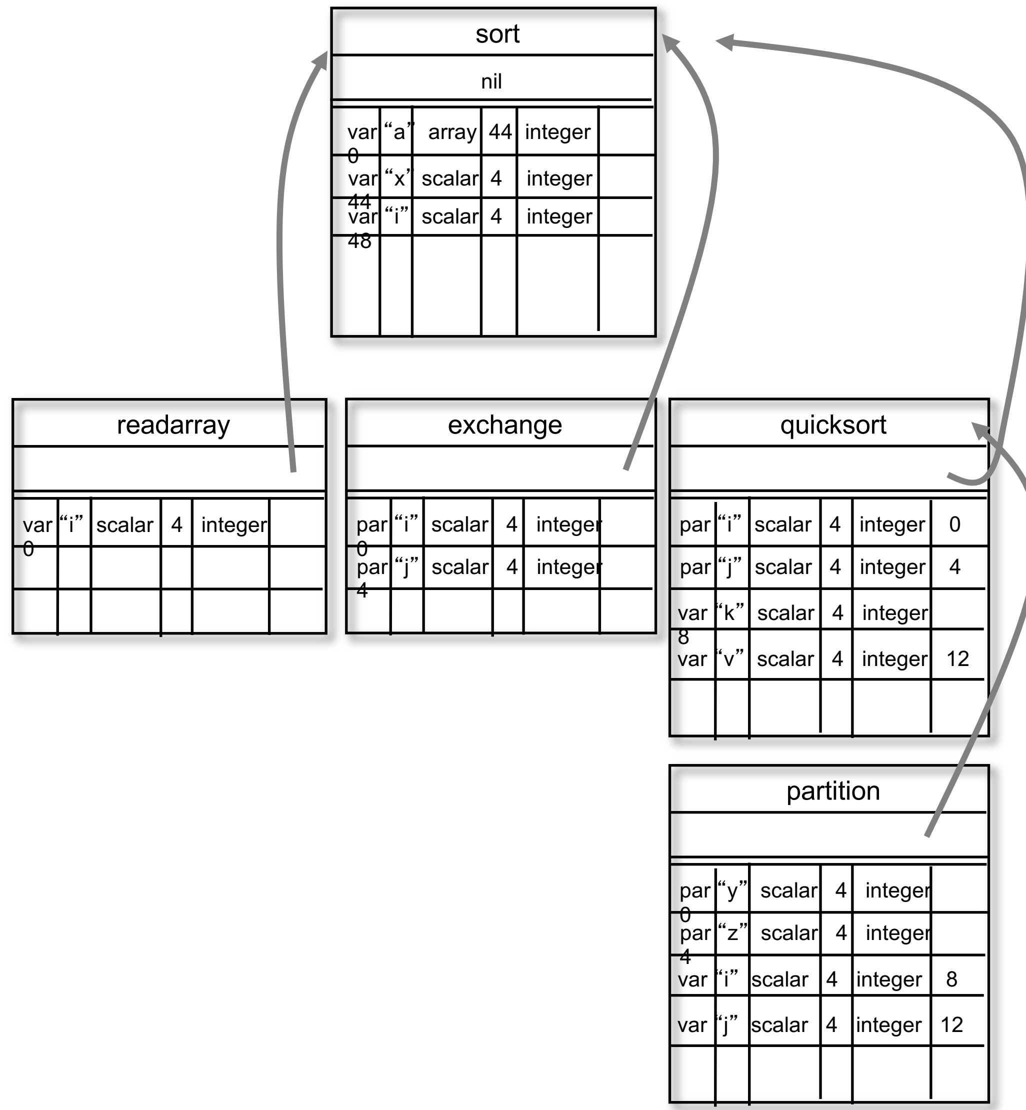
```
program sort(input, output);
var a: array [0..10] of integer;
    x, i: integer;
procedure readarray;
var i : integer;
begin … a… end { readarray } ;
procedure exchange(i, j: integer);
   begin
      x := a[i]; a[i] := a[j]; a[j] := x;
   end { exchange } ;
procedure quicksort(m, n : integer);
  var k, v: integer;
  function partition(y, z : integer) : integer;
    var i, j : integer;
    begin … a …
          … v …
          … exchange(i,j); …
    end { partition }
  begin … end { quicksort }
begin … end { sort }
```

Call Tree



Nested Procedures & Symbol Tables

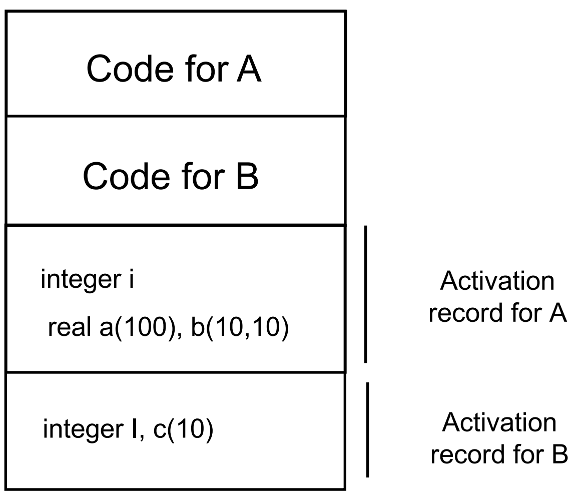
```
program sort(input, output);
var a: array [0..10] of integer;
      x, i: integer;
procedure readarray;
var i : integer;
begin … a… end { readarray } ;
procedure exchange(i, j: integer);
    begin
      x := a[i]; a[i] := a[j]; a[j] := x;
    end { exchange } ;
procedure quicksort(m, n : integer);
  var k, v: integer;
  function partition(y, z : integer) : integer;
    var i, j : integer;
    begin … a …
            … v …
             … exchange(i,j); …
    end { partition }
  begin … end { quicksort }
begin … end { sort }
```



Static Allocation

```
subroutine A()
  integer i
  real a(100), b(10,10)
    do 100 i=1, 10
      a(i*10) = b(i,10)
100: continue
end
```

```
subroutine B()
  integer i, c(10)
    do 200 i=1, 10
      c(i) = 0
200: continue
end
```



- Local variables are bound to fixed location in storage
  - Values can be retained across procedure call (static)
  - Save PC in AR but no need for stack
- Limitations:
  - Fixed size variables only
  - Does not support recursion
  - No dynamic memory allocation
- Advantages:
  - Simplified code generation

Lexical Scopes Without Nested Procedures

```
program sort(input, output);
  var a: array [0..10] of integer;
        x: integer;
procedure readarray;
  var i : integer;
  begin … a… end { readarray } ;
procedure exchange(i, j: integer);
    begin
      x := a[i]; a[i] := a[j]; a[j] := x;
    end { exchange } ;
 function partition(y, z : integer) : integer;
    var i, j : integer;
    begin … a …
               … exchange(i,j); …
    end { partition }
procedure quicksort(m, n : integer);
  var k, v: integer;
begin … end { quicksort }
begin … end { sort }
```

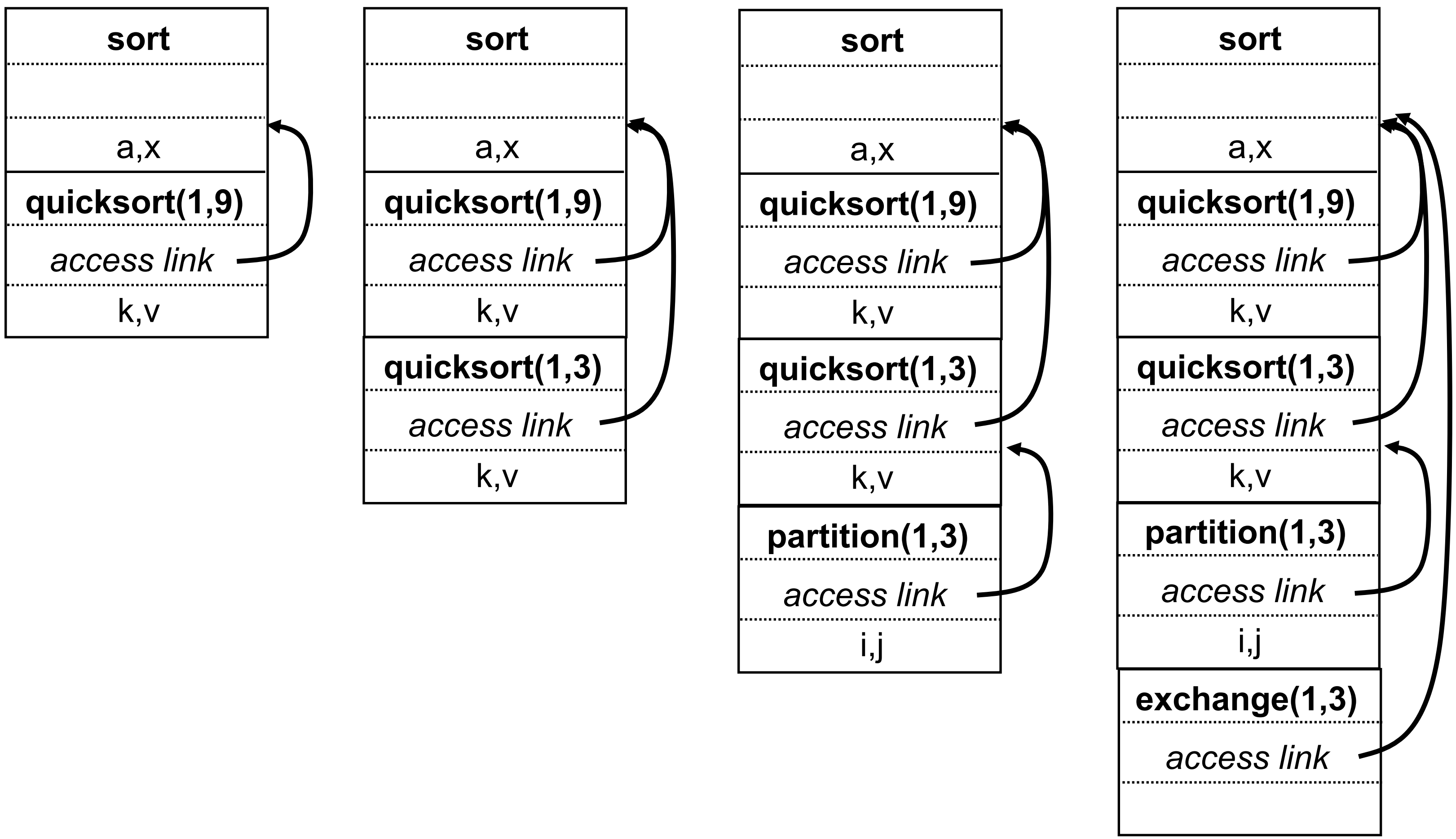
- Easy location of variables
  - Either local, i.e., in the AR
  - Global, i.e. at specified global offset
- Why Do with Need a Stack?

Lexical Scope With Nested Procedures

```
program sort(input, output);
  var a: array [0..10] of integer;
      x: integer;
procedure  readarray;
  var i : integer;
  begin … a… end { readarray } ;
procedure exchange(i, j: integer);
    begin
      x := a[i]; a[i] := a[j]; a[j] := x;
    end { exchange } ;
procedure  quicksort(m, n : integer);
  var k, v: integer;
  function partition(y, z : integer) : integer;
    var i, j : integer;
    begin … a …
          … v …
           … exchange(i,j); …
    end { partition }
    begin … end { quicksort }
begin … end { sort }
```

- Problem!
  - Now `quicksort` might have to access `a`, `x` at `sort` …
  - Also, `partition` needs to access `k`, `v` at `quicksort`
  - But which one?
- Need to keep Track of Depth
  - (static) Nesting Depth
    - `sort` at depth 1
    - `readarray`, `quicksort` at depth 2
    - `partition` at depth 3
  - Implementation
    - Link Chasing in AR
    - Reflect Nesting Structure During Calls
    - Display indexed by depth

Activation Records on the Stack



Lexical Scoping Example

```
program sort(input, output);
var a: array [0..10] of integer;
var x, i: integer;
procedure readarray;
var i : integer;
begin … a… end { readarray } ;
procedure exchange(i, j: integer);
  begin
   x := a[i]; a[i] := a[j]; a[j] := x;
  end { exchange } ;
procedure quicksort(m, n : integer);
 var k, v: integer;
 function partition(y, z : integer) : integer;
  var i, j : integer;
  begin … a …
      … v …
       … exchange(i,j); …
  end { partition }
 begin … end { quicksort }
begin … end { sort }
```

Access Links and How to Use Them

- Suppose procedure p at lexical nesting depth n_p refers to non-local variable *a* at depth $n_q \leq n_p$, then *a* can be found:
    1. Follow n_p - n_q access links from AR of p
    2. Access the variable at offset *a* in 'that' AR
- Example:
    - `partition` code at depth = 3 refers to *v* and *a* at depth 2 and 1 for which the code should traverse 1 and 2 access links respectively.
- As (n_p - n_q) can be computed at compile-time this tracing "method" is always feasible.
- What happens if $n_q > n_p$?

Access Links and How to Use Them

- Suppose procedure **p** at lexical nesting depth n_p refers to non-local variable ***a*** at depth $n_q \leq n_p$, then ***a*** can be found:
  1. Follow n_p - n_q access links from AR of **p**
  2. Access the variable at offset *a* in 'that' AR
- Example:
  - `partition` code at depth = 3 refers to *v* and *a* at depth 2 and 1 for which the code should traverse 1 and 2 access links respectively.
- As (n_p - n_q) can be computed at compile-time this tracing "method" is always feasible.
- What happens if $n_q > n_p$?
  - Variable a is not visible! The compiler will never allow this access.

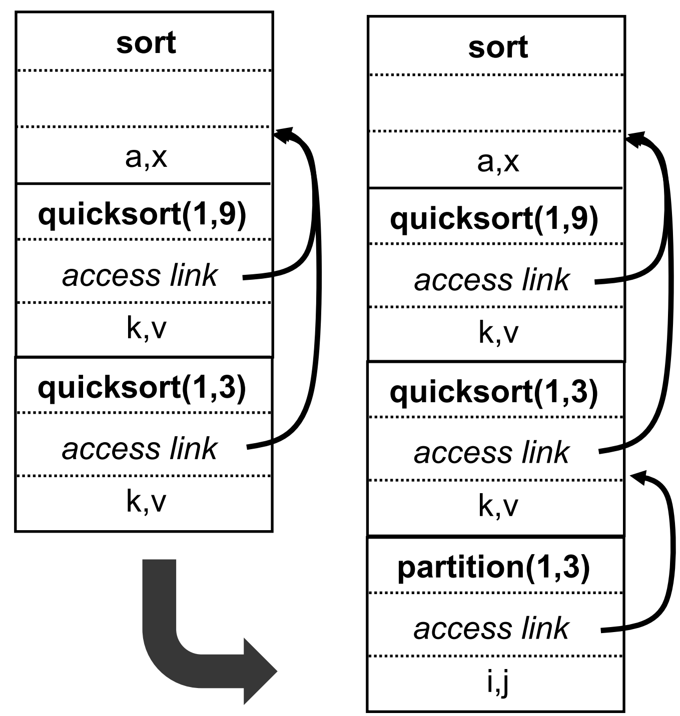
How to Set Up Access Links?

- Procedure **p** at depth n_p calls **q** at depth n_q
- Code generated as part of the calling sequence:
  - Case $n_q > n_p$: procedure **q** is nested more deeply than **p**; it must be declared within **p**, *i.e.* $n_q = n_p + 1$; **Why?**
    - **Action**: copy ARP pointer of the caller's to the callee's access link as this creates an additional indirection (another level)
  - Case $n_q \leq n_p$: all the ARs of the procedures up to **p** are the same, simply need to access the link of the most recent invocation of **p**;
    - **Action:** Follow $n_q - n_p + 1$ access links you reach the correct AR of procedure **r** that encloses **p** to set the access link in the AR of **q**.

How to Set Up Access Links?

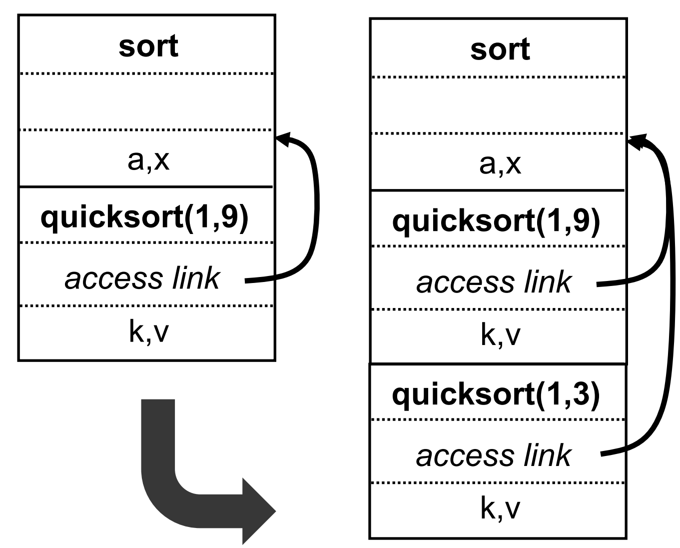
- Code generated as part of the calling sequence:
  – Case $n_q > n_p$: procedure **q** is nested more deeply than **p**; it must be declared within **p**, *i.e.* $n_q = n_p + 1$; **Why?**
    - **Action**: copy ARP pointer of the caller's to the callee's access link

  – Example:
    - `quicksort` (n_p=2) calls `partition`(n_p=3)

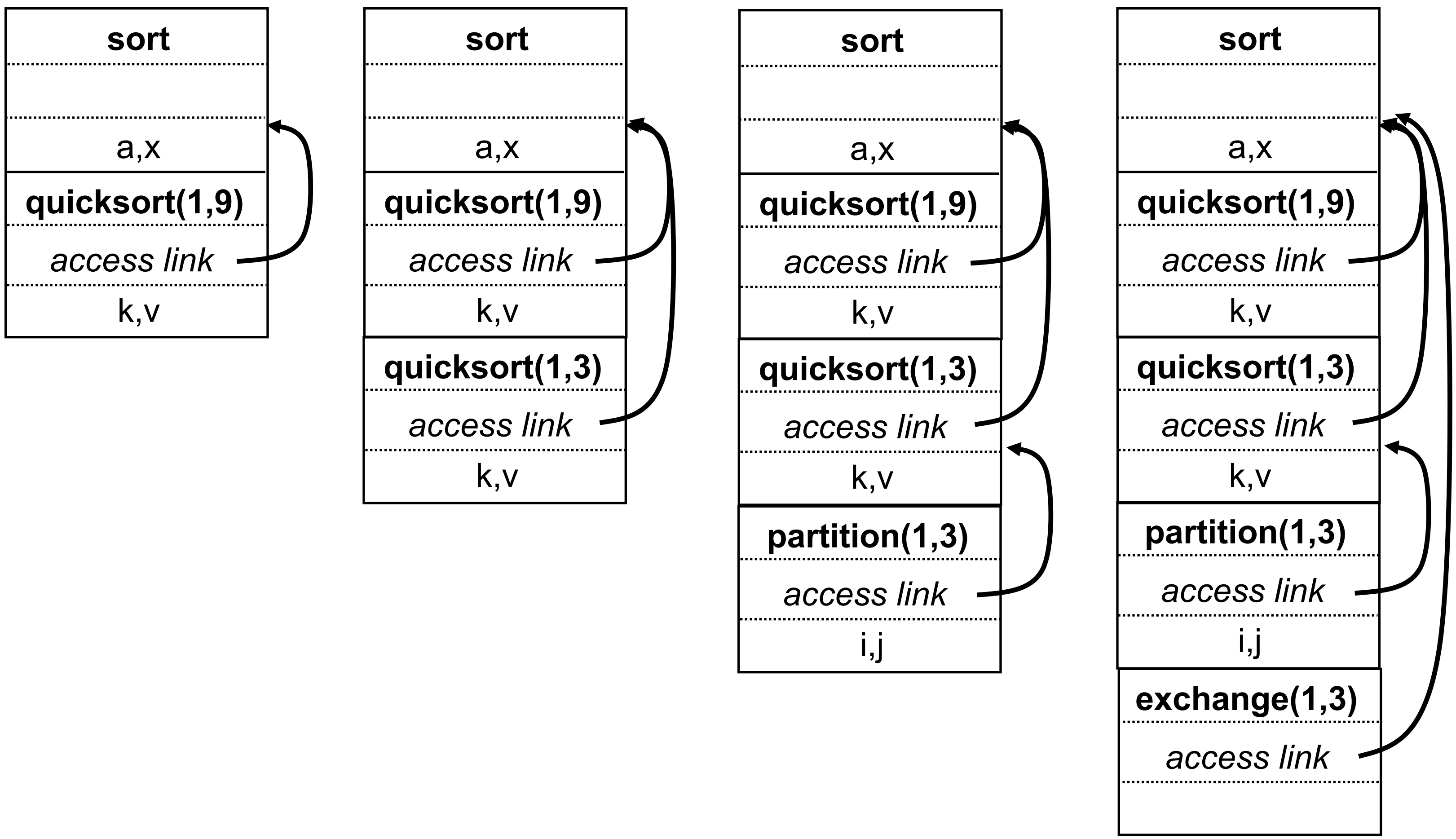


How to Set Up Access Links?

- Code generated as part of the calling sequence:
  – Case $n_p > n_p$: all the ARs of the procedures up to **p** are the same, simply need to access the link of the most recent invocation of **p**;
    - **Action:** Follow n_q - n_p +1 access links you reach the correct AR of procedure **r** that encloses **p** to set the access link in the AR of **q**.
  – Example:
    - `quicksort` (n_p=2)
      calls `quicksort` (n_p=2)

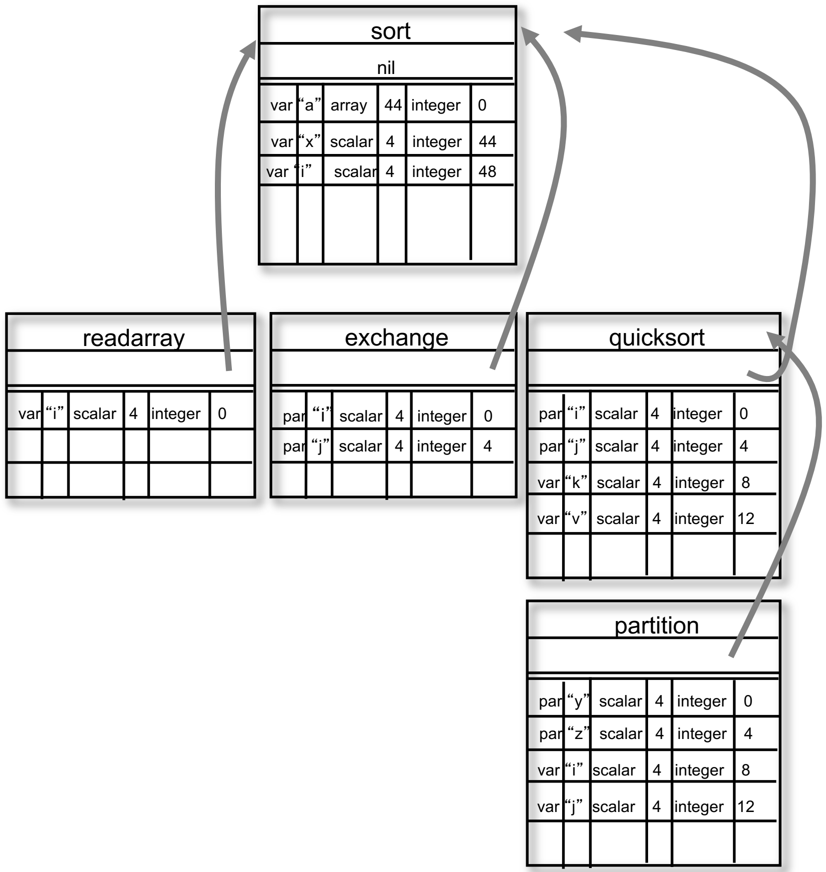


Lexical Scope with Nested Procedures

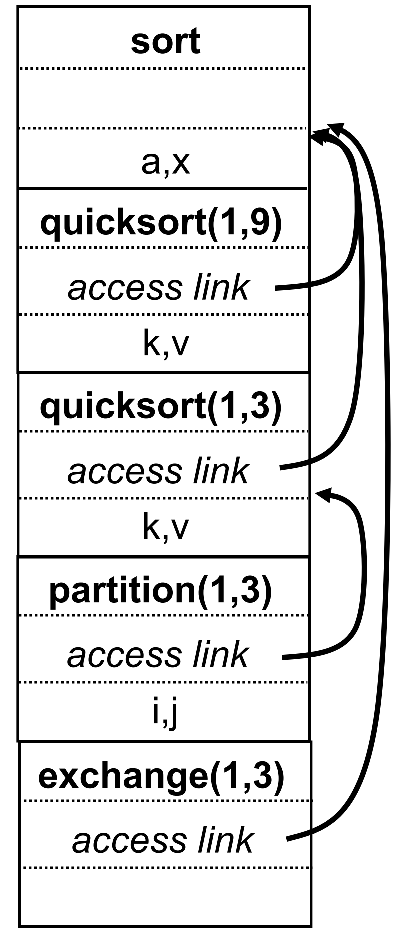


sort calls quicksort	quicksort calls quicksort	quicksort calls partition	partition calls exchange
$n_q = n_p + 1$	$n_q = n_p$	$n_q = n_p + 1$	$n_p = n_q + 1$

Lexical Scope with Nested Procedures

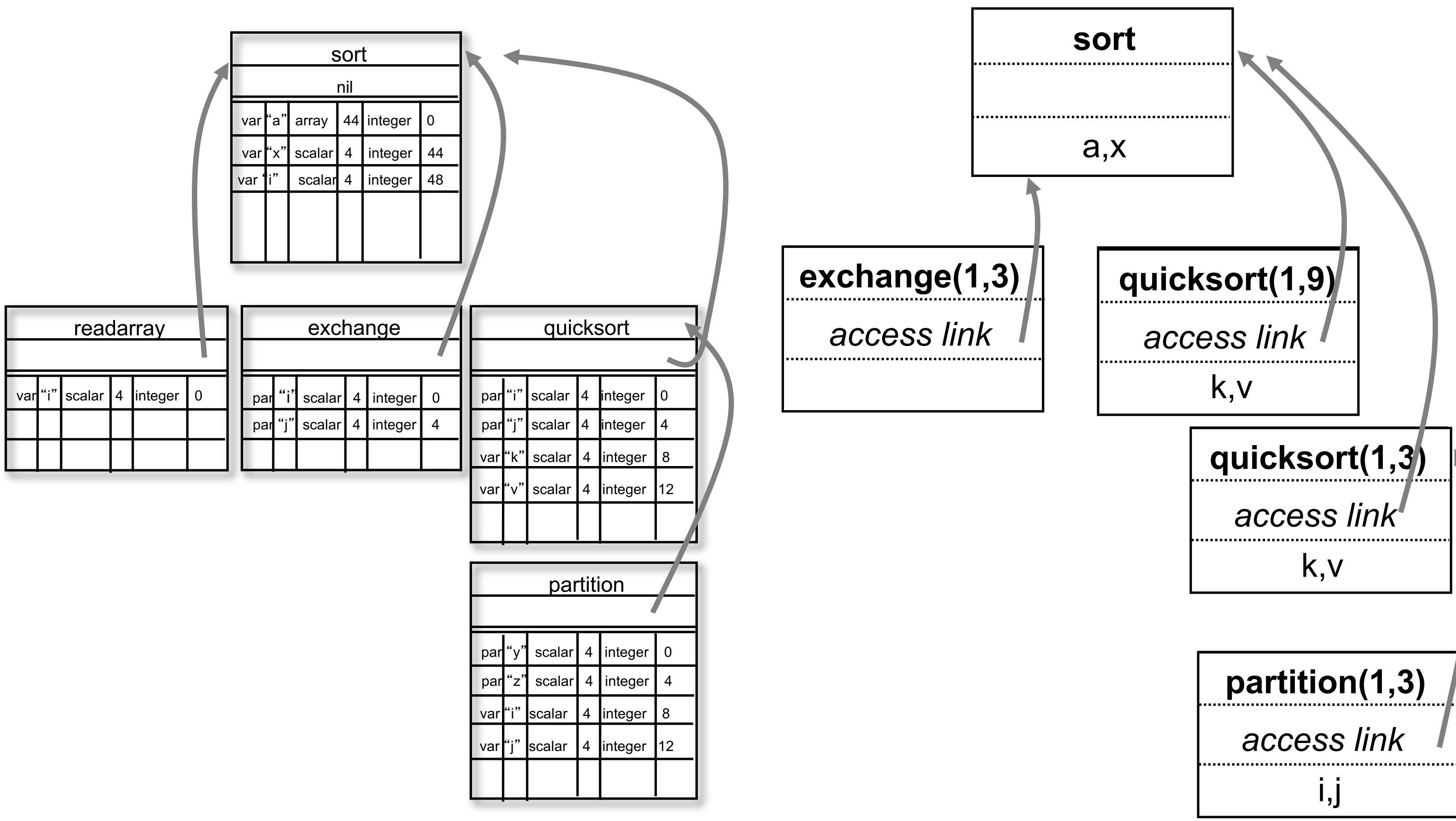


Compile Time



Run Time

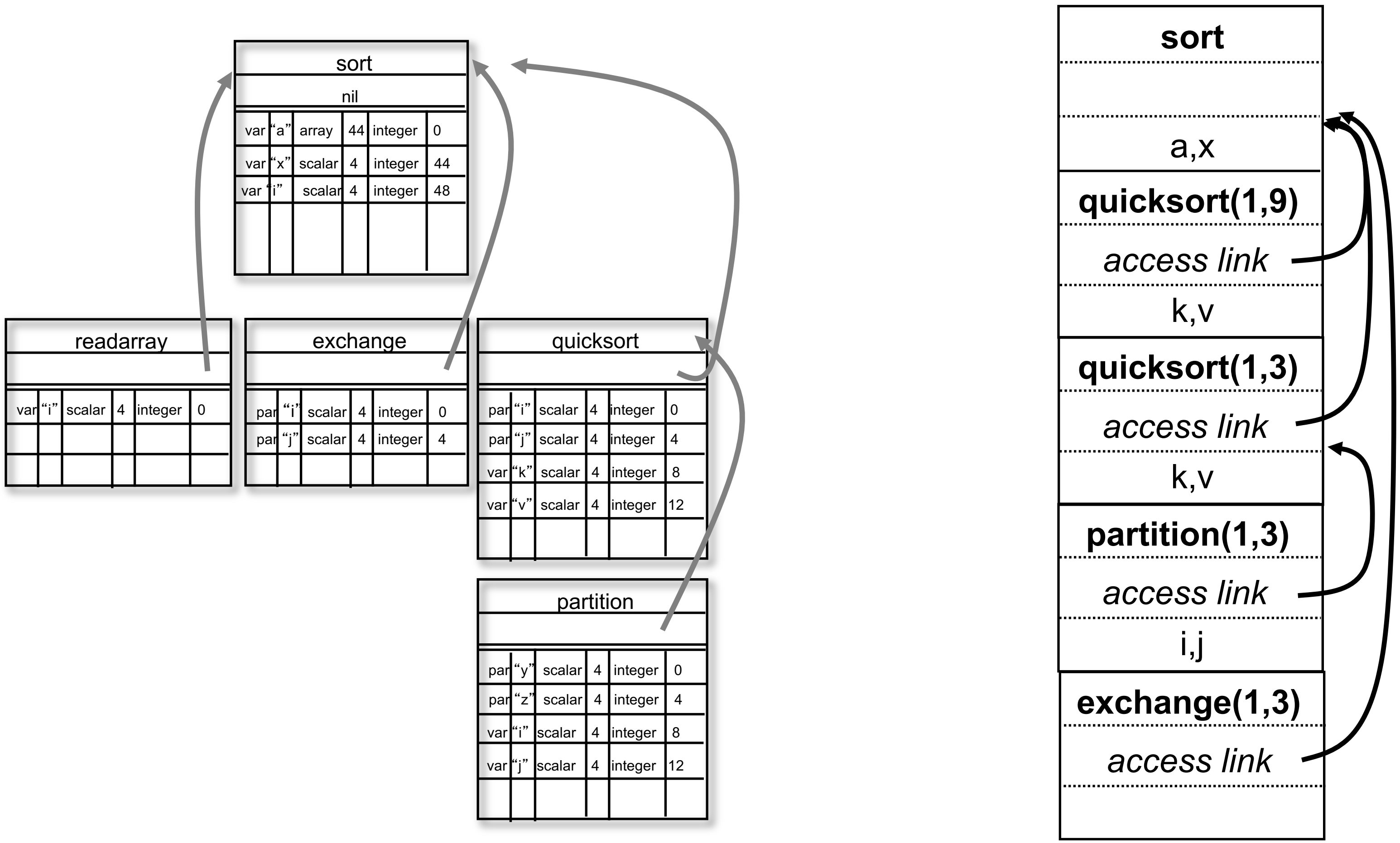
Lexical Scope with Nested Procedures



Compile Time

Run Time

Lexical Scope with Nested Procedures



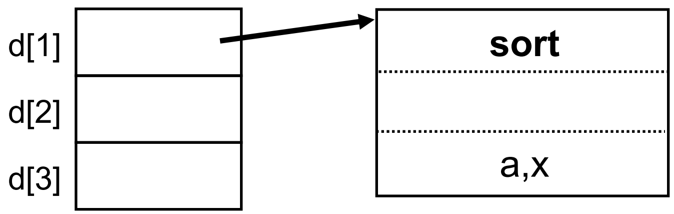
Compile Time

Run Time

Display

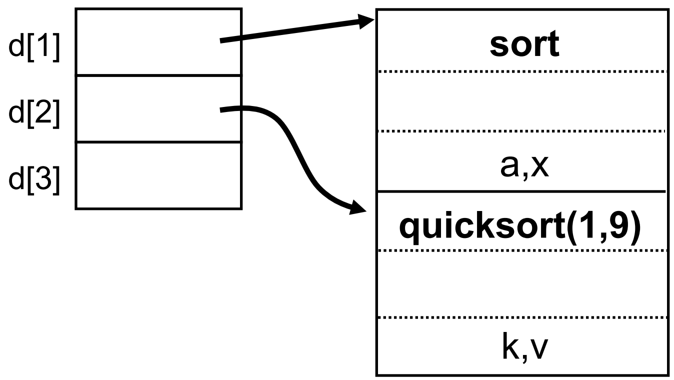
- Following Access Links can take a long Time
- Solution?
  - Keep an auxiliary array of pointers to AR on the stack
  - Storage for a non-local at depth *i* is in the activation record pointed to by d[*i*] called *Display*.
  - Faster because you need to follow a single pointer
- How to Maintain the *Display* ?
  - When AR of procedure at depth i is set up:
    - Save the value of d[i] in the new AR
    - Set d[i] to point to the new AR.
  - Just before an activation ends, d[i] is reset to the saved value
    - Values in saved at a specific offset on the AR like ARP and return

Display: Example



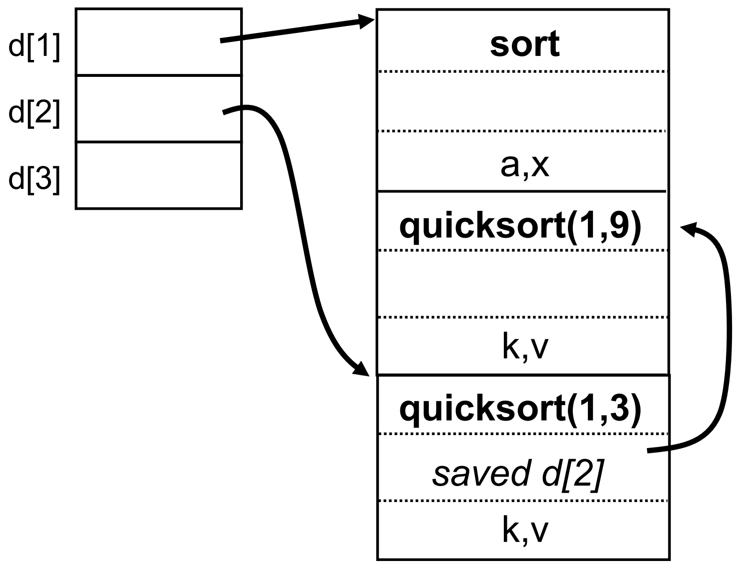
- **P** at depth n_p calls **Q** at depth n_q
- Case $n_q > n_p$: then $n_q = n_p + 1$
  - First n_p elements of the display do not change; just set $d[n_q]$ to new AR.
- Case $n_q \leq n_p$:
  - Enclosing procedures at levels 1, ..., n_q-1 are the same; save old $d[n_q]$ in new AR and make $d[n_q]$ point to new AR.

Display: Example



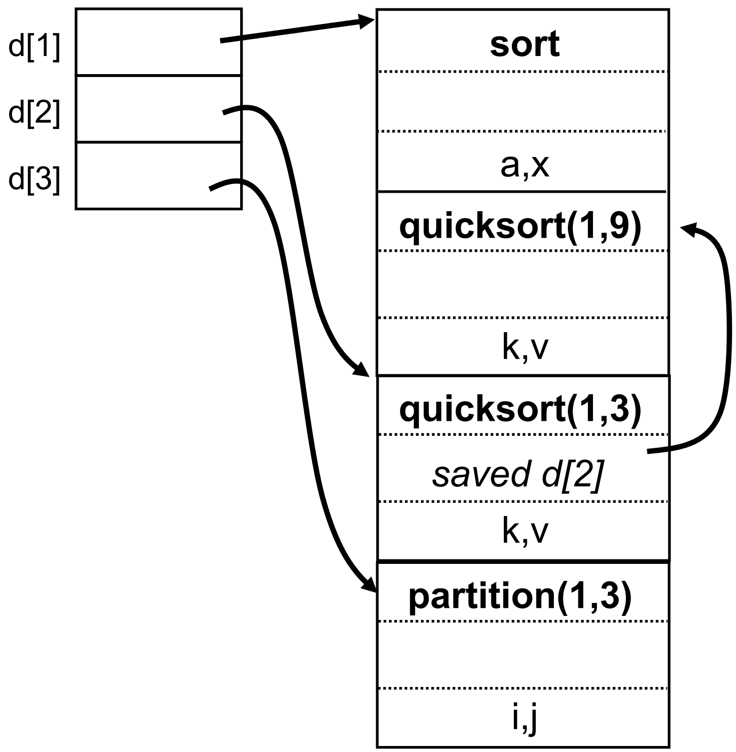
- **P** at depth n_p calls **Q** at depth n_q
- Case $n_q > n_p$: then $n_q = n_p + 1$
  - First n_p elements of the display do not change; just set d[n_q] to new AR.
- Case $n_q \leq n_p$:
  - Enclosing procedures at levels 1, ..., n_q-1 are the same; save old d[n_q] in new AR and make d[n_q] point to new AR.

Display: Example



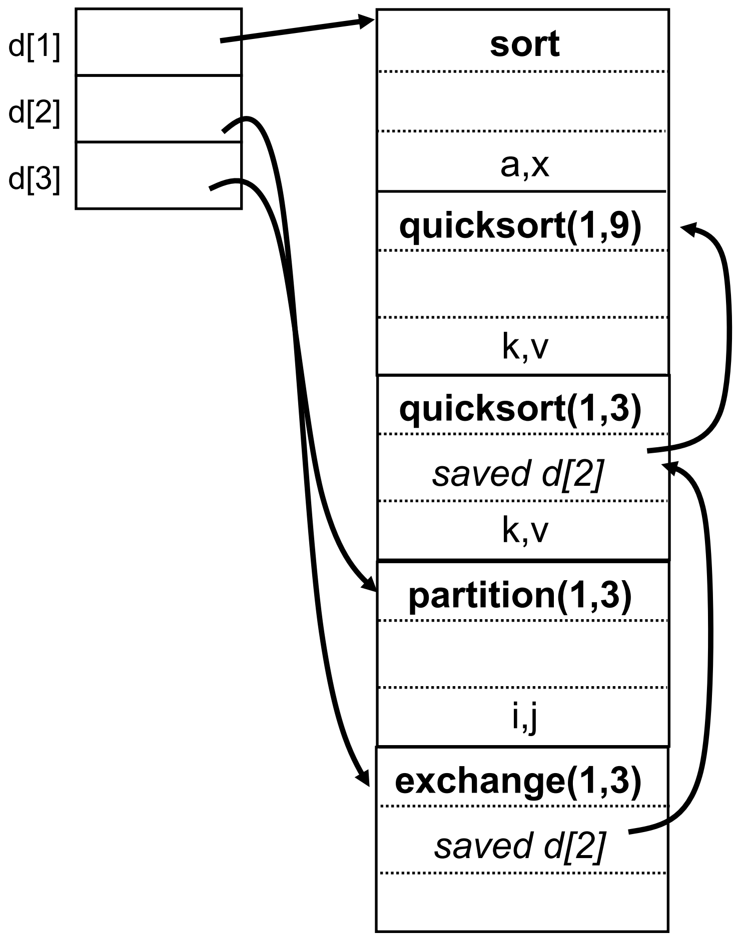
- **P** at depth n_p calls **Q** at depth n_q
- Case $n_q > n_p$: then $n_q = n_p + 1$
  - First n_p elements of the display do not change; just set $d[n_q]$ to new AR.
- Case $n_q \leq n_p$:
  - Enclosing procedures at levels 1, ..., n_q-1 are the same; save old $d[n_q]$ in new AR and make $d[n_q]$ point to new AR.

Display: Example



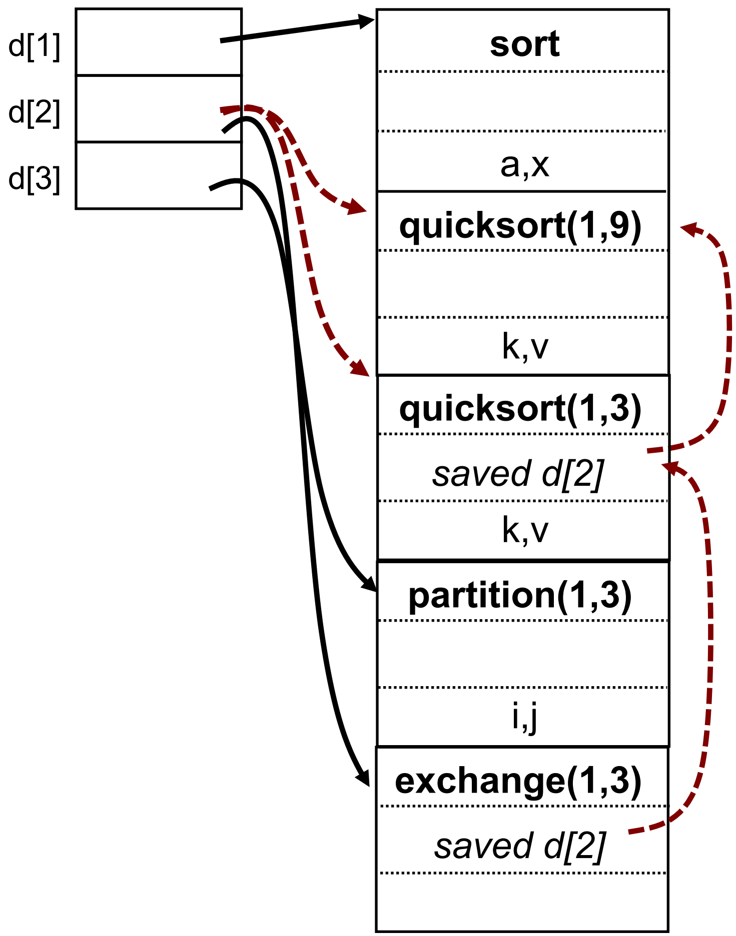
- **P** at depth n_p calls **Q** at depth n_q
- Case $n_q > n_p$: then $n_q = n_p + 1$
  - First n_p elements of the display do not change; just set d[n_q] to new AR.
- Case $n_q \leq n_p$:
  - Enclosing procedures at levels 1, ..., n_q-1 are the same; save old d[n_q] in new AR and make d[n_q] point to new AR.

Display: Example



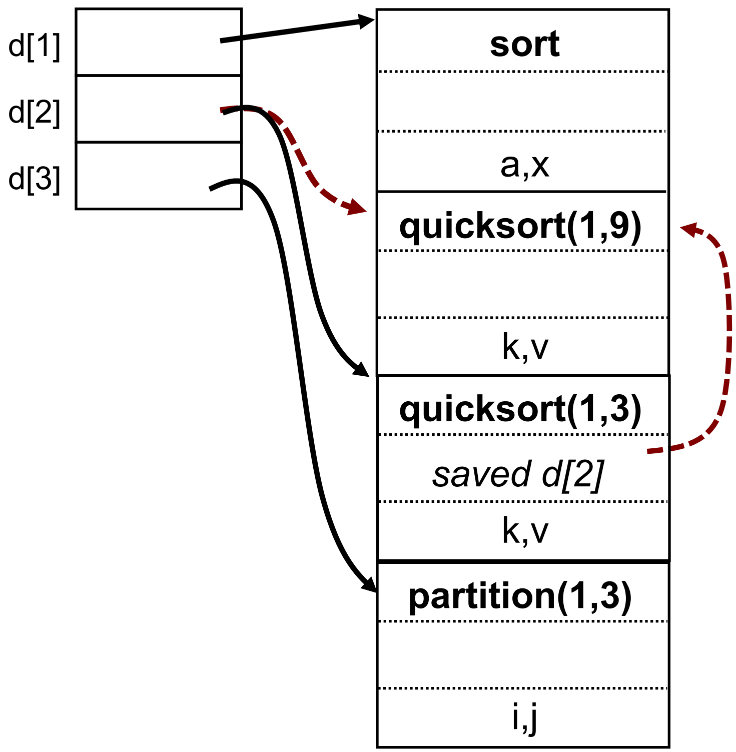
- **P** at depth n_p calls **Q** at depth n_q
- Case $n_q > n_p$: then $n_q = n_p + 1$
  - First n_p elements of the display do not change; just set d[n_q] to new AR.
- Case $n_q \leq n_p$:
  - Enclosing procedures at levels 1, ..., n_q-1 are the same; save old d[n_q] in new AR and make d[n_q] point to new AR.

Display: Example



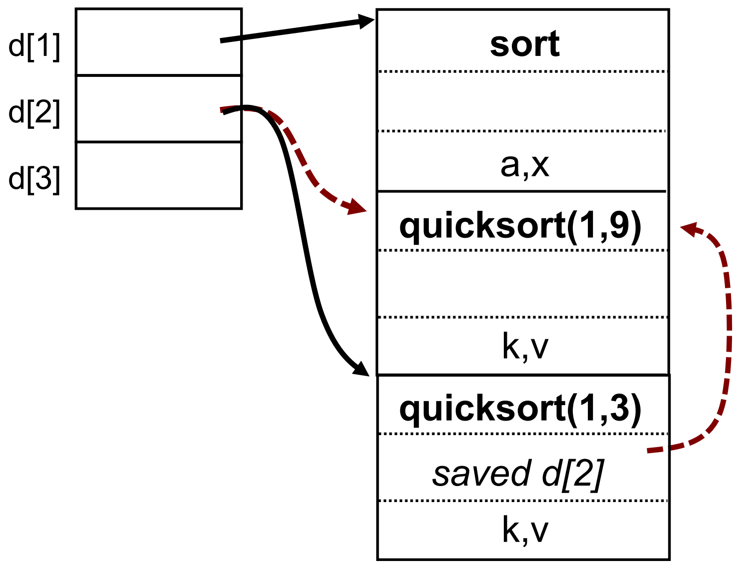
- **P** at depth n_p calls **Q** at depth n_q
- Case $n_q > n_p$: then $n_q = n_p + 1$
  - First n_p elements of the display do not change; just set d[n_q] to new AR.
- Case $n_q \leq n_p$:
  - Enclosing procedures at levels 1, ..., n_q-1 are the same; save old d[n_q] in new AR and make d[n_q] point to new AR.

Display: Example



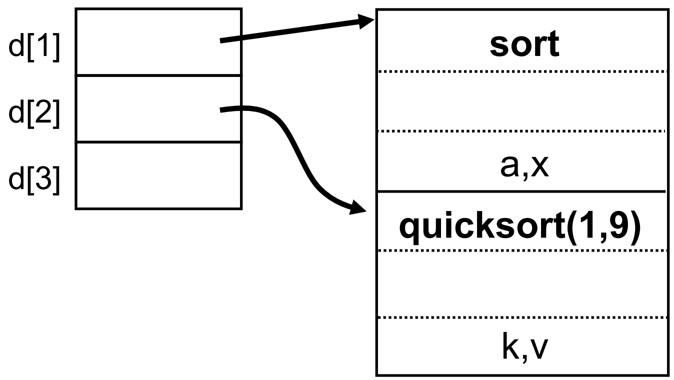
- **P** at depth n_p calls **Q** at depth n_q
- Case $n_q > n_p$: then $n_q = n_p + 1$
  - First n_p elements of the display do not change; just set d[n_q] to new AR.
- Case $n_q \leq n_p$:
  - Enclosing procedures at levels 1, ..., n_q-1 are the same; save old d[n_q] in new AR and make d[n_q] point to new AR.

Display: Example



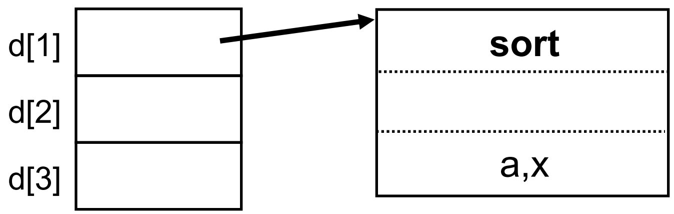
- **P** at depth n_p calls **Q** at depth n_q
- Case $n_q > n_p$: then $n_q = n_p + 1$
  - First n_p elements of the display do not change; just set $d[n_q]$ to new AR.
- Case $n_q \leq n_p$:
  - Enclosing procedures at levels 1, ..., n_q-1 are the same; save old $d[n_q]$ in new AR and make $d[n_q]$ point to new AR.

Display: Example



- **P** at depth n_p calls **Q** at depth n_q
- Case $n_q > n_p$: then $n_q = n_p + 1$
  - First n_p elements of the display do not change; just set d[n_q] to new AR.
- Case $n_q \leq n_p$:
  - Enclosing procedures at levels 1, ..., n_q-1 are the same; save old d[n_q] in new AR and make d[n_q] point to new AR.

Display: Example



- **P** at depth n_p calls **Q** at depth n_q
- Case $n_q > n_p$: then $n_q = n_p + 1$
  - First n_p elements of the display do not change; just set $d[n_q]$ to new AR.
- Case $n_q \leq n_p$:
  - Enclosing procedures at levels 1, ..., n_q-1 are the same; save old $d[n_q]$ in new AR and make $d[n_q]$ point to new AR.

(Other) Complications

- Passing Functions as Arguments?
  - Just an address of the code, not that hard to implement
  - Need to verify at run-time the number of arguments.
  - Access Link needs to be passed to understand how to follow it…
- What if AR outlives Execution of Procedure?
  - When is this possible?
  - What to do?
- Dynamically Linked Libraries
  - What are the issues with "regular" libraries?
- Position-Independent Code
  - Why is this important?

Code Sharing

- Traditionally Link all Libraries with your code
- Drawbacks:
  - Space as each executable includes the code of all libraries it uses (big as every function needs to be included at link time)
  - Bugs in libraries require recompilation and linking
- Solution: Dynamically Linked Libraries
  - Loaded and linked on-demand during execution
- Advantages:
  - Single Copy in the system rather than replicated.
  - Executable has only what is really needs.
  - Bugs can be fixed later not requiring re-linking

Shared Libraries

- Make it Look Like a Statically Linked
- Linking?
  - Name Resolution: finding bindings for symbols

- Determine before hand if linking will succeed
  - Check for undefined or multiply defined symbols
  - Create a table of symbols for each shared library
  - Pre-execution linking checks the tables
  - Run-time dynamic linker is guaranteed to fail if and only if the pre-execution static linker would.

U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO
DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Summary

- What Have We Learned?
  - AR is a Run-time Structure to hold State regarding the Execution of a Procedure
  - AR can be allocated in Static, Stack or even Heap
  - Links allow Call-Return and Access to Non-local Variables
  - Symbol-Table plays Important Role
- Linkage Conventions
  - Saving Context before Call and restoring after Call
  - Need to understand how to generate code for body