Run Time Environment

Activation Records
Procedure Linkage
Name Translation and Variable Access

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 abstracts from all the details
- How to Generate Code?
  - Storage Allocation
  - Scoping, *i.e.,* what is visible and where?
  - Control Transfer

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

```
...                 int p(a,b,c)
s = p(10,t,u);        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

```
...                 int p(a,b,c)           int q(x,y)
s = p(10,t,u);        int a, b, c;           int x,y;
...                 {                      {
                      int   d;               return x + y;
                      d = q(c,b);          }
                      ...
                    }
```

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

```
                  int p(a,b,c)
                    int a, b, c;             int q(x,y)
                  {                            int x,y;
…                   int   d;                 {
s = p(10,t,u);      d = q(c,b);                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

```
                  int p(a,b,c)
                    int a, b, c;             int q(x,y)
                  {                            int x,y;
…                   int   d;                 {
s = p(10,t,u);      d = q(c,b);                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

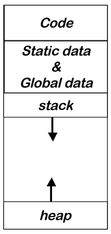
```
                  int p(a,b,c)
                    int a, b, c;             int q(x,y)
                  {                            int x,y;
…                   int   d;                 {
s = p(10,t,u);      d = q(c,b);                return x + y;
…                   …                        }
                  }
```


- Most Languages Allow Recursion

Compilation Issues

- How to Generate Code
  - 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

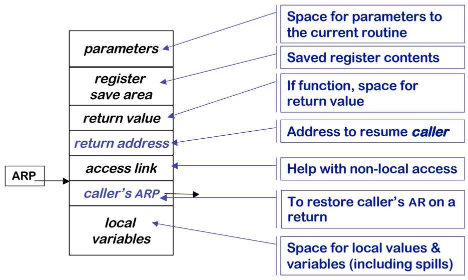
Run-Time Storage Organization



Classical Organization

- Code, Static, & Global data have known size
  - → Use symbolic labels in the code
- Heap & stack both grow & 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

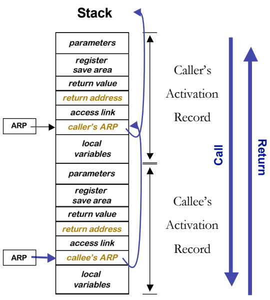
Activation Record Basics



One **AR** for each invocation of a procedure

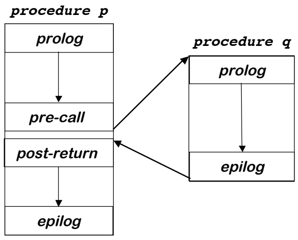
Activation Records on the Stack

- 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 Þ 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

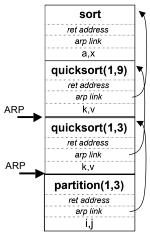
Epilog Code
- Wind up the business of the callee
- Start restoring the caller's environment

If ARs are stack allocated, this may not be necessary. (Caller can reset stacktop 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

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 }
```

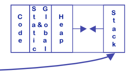


Activation Record Details

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



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

Yes! This stack.

⇒ 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 Compiler find the Variables?

- They are at known offsets from the AR pointer
- The static coordinate leads to a "loadAI" operation
  - Level specifies an ARP, offset is the constant

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

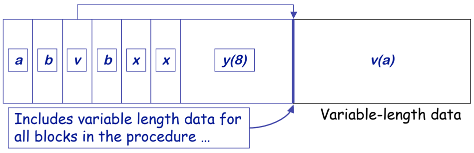
```
B0: {
      int a, b
      … assign value to a
B1:       {
            int v(a), b, x
B2:           {
                int x, y(8)
                ….
          }
```

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

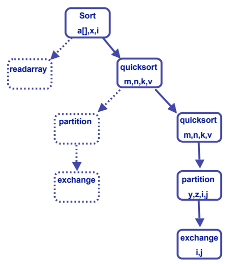
Scoping Rules

- 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

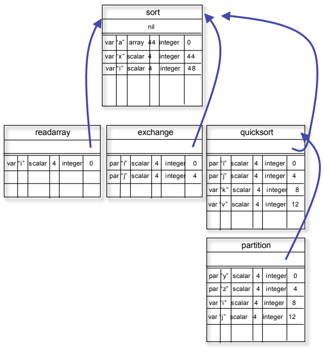
```
program sort(input, output);
var a: array [0..10] of integer;
     x, i: integer;
procedure readarray;
var i : integer;
begin … a… end { readarray } ;
procedure exchange(i, j: integer);
    begin
     x := a[i]; a[i] := a[j]; a[j] := x;
    end { exchange } ;
procedure quicksort(m, n : integer);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

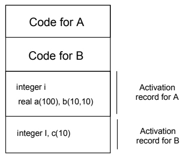
```
program sort(input, output);
var a: array [0..10] of integer;
    x, i: integer;
procedure readarray;
var i : integer;
begin … a… end { readarray } ;
procedure exchange(i, j: integer);
   begin
     x := a[i]; a[i] := a[j]; a[j] := x;
   end { exchange } ;
procedure quicksort(m, n : integer);
  var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

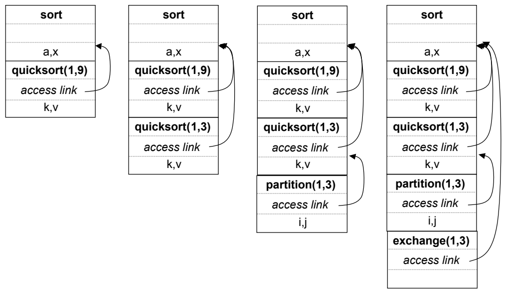
- Easy location of variables
  - Either local, i.e., in the AR
  - Global, i.e. at specified global offset
- Why Do with Need a Stack?

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 <= n_p, them *a* can be found:
  1. Follow n_p - n_q access links from AR of p
  2. Access the variable at offset *a* in current 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.
- Since (n_p - n_q) can be computed at compile-time this "procedure" is always feasible.
- What happens if n_q > n_p?

Access Links and How to Use Them

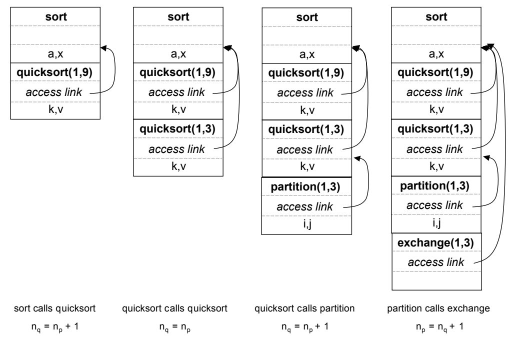
- Suppose Procedure p at lexical nesting depth n_p refers to non-local variable *a* at depth n_q <= n_p, them *a* can be found:
  1. Follow n_p - n_q access links from AR of p
  2. Access the variable at offset *a* in current AR
- Example:
  - `partition` code at depth = 3 refers to *v* and *a* at depth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

  - Variable a is not visible! The compiler will never observe this situation.

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 $n_p < n_q$: procedure q is nested more deeply than p; it must be declared within p, *i.e.* $n_q = n_p + 1$; **Why?**
    - Copy the ARP pointer of the caller's to the callee's access link
  - Case $n_p >= n_q$: 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;
    - 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.

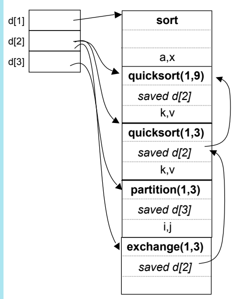
Lexical Scope with Nested Procedures



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 $n_p < n_q$: 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 $n_p >= n_q$:
  - 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.

Communicating Between Procedures

Most languages provide a parameter passing mechanism
⇒ Expression used at "call site" becomes variable in callee

Two common binding mechanisms

- Call-by-reference passes a pointer to actual parameter
  - Requires slot in the AR (for address of parameter)
  - Multiple names with the same address?
- Call-by-value passes a copy of its value at time of call
  - Requires slot in the AR
  - Each name gets a unique location *(may have same value)*
  - Arrays are mostly passed by reference, not value

- Can always use global variables …

Complications

- Passing Functions as Arguments?

- What if AR outlives Execution of Procedure?
  - When is this possible?
  - What to do?

More Complications

- Dynamically Linked Libraries

- Position-Independent Code

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 iff the pre-execution static linker would.

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 to allow Call-Return and Access to Non-local Variables
  - Symbol-table play important role
- Linkage Conventions
  - Saving Context before call and restoring after the call
  - Need to understand how to generate code for body