Implementing Subprograms

In Text: Chapter 9

Outline

- Activation records
- Accessing locals
- Accessing nonlocals (static scoping)
    - Static chains
    - Displays
- Implementing blocks
- Accessing nonlocals with dynamic scoping

Implementing Subprograms

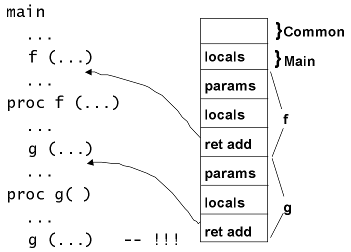
- The subprogram call and return operations of a language are together called its **subprogram linkage**
- **First, let's look at implementing FORTRAN 77 subprograms**

Implementing FORTRAN 77 Subprogs

- Call Semantics:
  - Save the execution status of the caller
  - Carry out the parameter-passing process
  - Pass the return address
  - Transfer control to the callee
- Return Semantics:
  - If pass-by-value-result is used, move current values of parameters to their corresponding actuals
  - If it is a function, move return value to a place the caller can get it
  - Restore the execution status of the caller
  - Transfer control back to the caller
- Required Storage:
  - Status information of the caller, parameters, return address, and functional value (if it is a function)

FORTRAN Activation Info

- can allocate all memory statically



```
main
  ...
  f (...)
  ...
proc f (...)
  ...
  g (...)
  ...
proc g( )
  ...
  g (...)   -- !!!
```

Activation Records

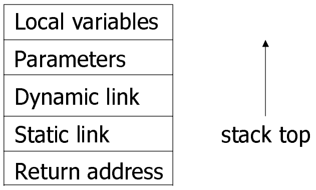
- The format, or layout, of the noncode part of an executing subprogram is called an **activation record (AR)**
- An **activation record instance (ARI)** is a concrete example of an activation record (the collection of data for a particular subprogram activation)
- FORTRAN 77 subprograms can have no more than one activation record instance at any given time
- The code of all of the program units of a FORTRAN 77 program may reside together in memory, with the data for all units stored together elsewhere
- The alternative is to store all local subprogram data with the subprogram code

Implementing Subprograms in ALGOL-like Languages

- More complicated than FORTRAN 77:
  - Parameters are often passed by two methods
  - Local variables are often dynamically allocated
  - Recursion must be supported
  - Static scoping must be supported

Activation Record Structure

- A typical activation record for an ALGOL-like language:



Local variables
Parameters
Dynamic link
Static link
Return address

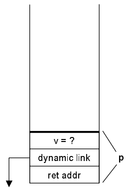
stack top

Activation Record Details

- The activation record format is static, but its size may be dynamic
- The **static link** points to the bottom of the activation record instance of an activation of the static parent (used for access to nonlocal vars)
- The **dynamic link** points to the **bottom** of an instance of the activation record of the caller
- An activation record instance is dynamically created when a subprogram is called

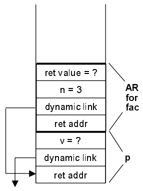
Example Factorial Program (1)

```
program p;
  var v : int;
  function fac(n: int): int;
  begin
    if n <= 1 then
      fac := 1
    else
      fac := n * fac(n - 1);
  end;
begin
  v := fac(3);
  print(v);
end.
```



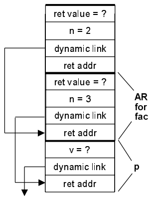
Example Factorial Program (2)

```
program p;
  var v : int;
  function fac(n: int): int;
  begin
    if n <= 1 then
      fac := 1
    else
      fac := n * fac(n - 1);
  end;
begin
  v := fac(3);
  print(v);
end.
```



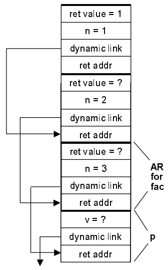
Example Factorial Program (3)

```
program p;
  var v : int;
  function fac(n: int): int;
  begin
    if n <= 1 then
      fac := 1
    else
      fac := n * fac(n - 1);
  end;
begin
  v := fac(3);
  print(v);
end.
```



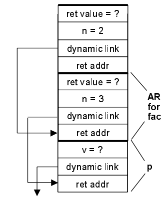
Example Factorial Program (4)

```
program p;
  var v : int;
  function fac(n: int): int;
  begin
    if n <= 1 then
      fac := 1
    else
      fac := n * fac(n - 1);
  end;
begin
  v := fac(3);
  print(v);
end.
```



Example Factorial Program (5)

```
program p;
  var v : int;
  function fac(n: int): int;
  begin
    if n <= 1 then
      fac := 1
    else
      fac := n * fac(n - 1);
  end;
begin
  v := fac(3);
  print(v);
end.
```



The Dynamic Chain

- The collection of dynamic links in the stack at a given time is called the **dynamic chain**, or **call chain**
- Local variables can be accessed by their offset from the beginning of the activation record. This offset is called the **local_offset**
- The local_offset of a local variable can be determined by the compiler:
  - Assuming all stack positions are the same size, the first local variable declared has an offset of three plus the number of parameters
- The activation record used in the previous example supports recursion

Accessing Nonlocal References

- Two situations:
  - Static scoping
    - Static chains
    - Displays
  - Dynamic scoping

Nonlocal Refs: Static Scoping

- All accessible nonlocal variables reside in some ARI on the stack
- The process of locating a nonlocal reference:
  - Find the correct ARI
  - Determine the correct offset within that ARI
- Finding the offset is easy! It is statically determined
- Finding the correct ARI:
  - Static semantic rules guarantee that all nonlocal variables that can be referenced have been allocated in some ARI that is on the stack when the reference is made

Technique 1: Static Chains

- A **static chain** is a chain of **static links** that connects certain ARIs
- The static link in an ARI for subprogram A points to one of the ARIs of A's **static parent**
- The static chain from an ARI connects it to all of its static ancestors
- To find the declaration for a reference to a nonlocal variable:
  - The compiler can easily determine how many levels of scope separate the current subprogram from the definition
  - Just walk the static chain the correct number of steps
- **Static_depth** is an integer associated with a static scope whose value is the depth of nesting of that scope

Static Depth and Chain Offset

```
main  ----- static_depth = 0
  A   ----- static_depth = 1
    B ----- static_depth = 2

  C   ----- static_depth = 1
```

- The **chain_offset** or **nesting_depth** of a nonlocal reference is the difference between the static_depth of the reference and that of the scope where it is declared
- A reference can be represented by the pair:

(chain_offset, local_offset)

Static Chain Example (1)

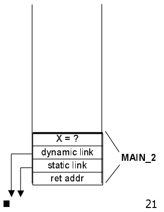
```
program MAIN_2;
   var X : integer;
   procedure BIGSUB;
      var A, B, C : integer;
      procedure SUB1;
         var A, D : integer;
      begin { SUB1 }
         A := B + C;  <----------1
      end;  { SUB1 }
      procedure SUB2(X : integer);
         var B, E : integer;
         procedure SUB3;
            var C, E : integer;
         begin { SUB3 }
            SUB1;
            E := B + A:   <------2
         end; { SUB3 }
      begin { SUB2 }
         SUB3;
         A := D + E;  <----------3
      end; { SUB2 }
   begin { BIGSUB }
      SUB2(7);
   end; { BIGSUB }
begin
   BIGSUB;
end. { MAIN_2 }
```

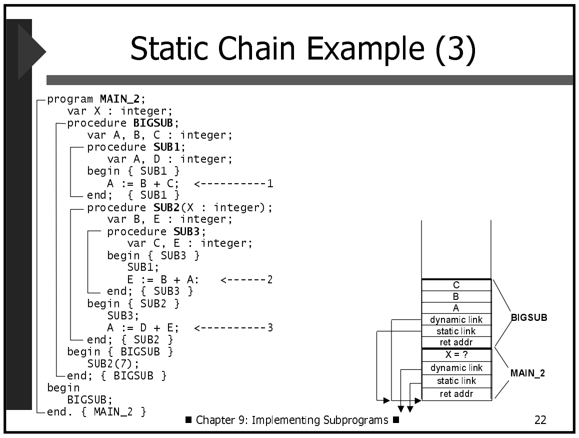
Call sequence for MAIN_2:

- MAIN_2 calls BIGSUB
- BIGSUB calls SUB2
- SUB2 calls SUB3
- SUB3 calls SUB1

Static Chain Example (2)

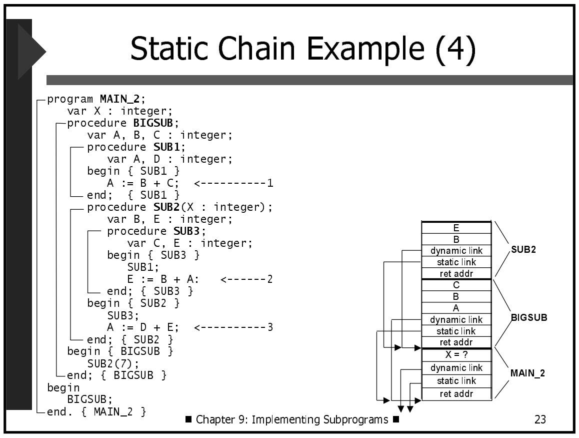
```
program MAIN_2;
   var X : integer;
   procedure BIGSUB;
      var A, B, C : integer;
      procedure SUB1;
         var A, D : integer;
      begin { SUB1 }
         A := B + C;  <----------1
      end;  { SUB1 }
      procedure SUB2(X : integer);
         var B, E : integer;
         procedure SUB3;
            var C, E : integer;
         begin { SUB3 }
            SUB1;
            E := B + A:   <------2
         end; { SUB3 }
      begin { SUB2 }
         SUB3;
         A := D + E;  <----------3
      end; { SUB2 }
   begin { BIGSUB }
      SUB2(7);
   end; { BIGSUB }
begin
   BIGSUB;
end. { MAIN_2 }
```





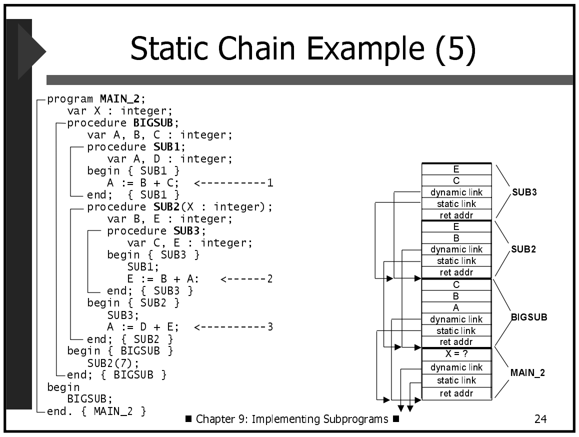
Static Chain Example (3)

```
program MAIN_2;
   var X : integer;
   procedure BIGSUB;
      var A, B, C : integer;
      procedure SUB1;
         var A, D : integer;
      begin { SUB1 }
         A := B + C;  <----------1
      end;  { SUB1 }
      procedure SUB2(X : integer);
         var B, E : integer;
         procedure SUB3;
            var C, E : integer;
         begin { SUB3 }
            SUB1;
            E := B + A:   <------2
         end; { SUB3 }
      begin { SUB2 }
         SUB3;
         A := D + E;  <----------3
      end; { SUB2 }
   begin { BIGSUB }
      SUB2(7);
   end; { BIGSUB }
begin
   BIGSUB;
end. { MAIN_2 }
```



Static Chain Example (4)

```
program MAIN_2;
   var X : integer;
   procedure BIGSUB;
      var A, B, C : integer;
      procedure SUB1;
         var A, D : integer;
      begin { SUB1 }
         A := B + C;  <----------1
      end;  { SUB1 }
      procedure SUB2(X : integer);
         var B, E : integer;
         procedure SUB3;
            var C, E : integer;
         begin { SUB3 }
            SUB1;
            E := B + A:   <------2
         end; { SUB3 }
      begin { SUB2 }
         SUB3;
         A := D + E;  <----------3
      end; { SUB2 }
   begin { BIGSUB }
      SUB2(7);
   end; { BIGSUB }
begin
   BIGSUB;
end. { MAIN_2 }
```



Static Chain Example (5)

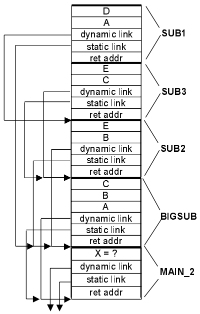
```
program MAIN_2;
   var X : integer;
   procedure BIGSUB;
      var A, B, C : integer;
      procedure SUB1;
         var A, D : integer;
      begin { SUB1 }
         A := B + C;  <----------1
      end;  { SUB1 }
      procedure SUB2(X : integer);
         var B, E : integer;
         procedure SUB3;
            var C, E : integer;
         begin { SUB3 }
            SUB1;
            E := B + A:   <------2
         end; { SUB3 }
      begin { SUB2 }
         SUB3;
         A := D + E;  <----------3
      end; { SUB2 }
   begin { BIGSUB }
      SUB2(7);
   end; { BIGSUB }
begin
   BIGSUB;
end. { MAIN_2 }
```

Static Chain Example (6)

```
program MAIN_2;
   var X : integer;
   procedure BIGSUB;
      var A, B, C : integer;
      procedure SUB1;
         var A, D : integer;
      begin { SUB1 }
         A := B + C;  <----------1
      end;  { SUB1 }
      procedure SUB2(X : integer);
         var B, E : integer;
         procedure SUB3;
            var C, E : integer;
         begin { SUB3 }
            SUB1;
            E := B + A:   <------2
         end; { SUB3 }
      begin { SUB2 }
         SUB3;
         A := D + E;  <----------3
      end; { SUB2 }
   begin { BIGSUB }
      SUB2(7);
   end; { BIGSUB }
begin
   BIGSUB;
end. { MAIN_2 }
```

At position 1 in SUB1:

- A - (0, 3)
- B - (1, 4)
- C - (1, 5)

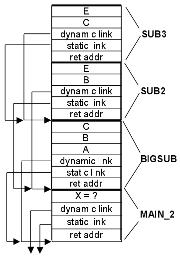


Static Chain Example (7)

```
program MAIN_2;
   var X : integer;
   procedure BIGSUB;
      var A, B, C : integer;
      procedure SUB1;
         var A, D : integer;
      begin { SUB1 }
         A := B + C;  <----------1
      end;  { SUB1 }
      procedure SUB2(X : integer);
         var B, E : integer;
         procedure SUB3;
            var C, E : integer;
         begin { SUB3 }
            SUB1;
            E := B + A:   <------2
         end; { SUB3 }
      begin { SUB2 }
         SUB3;
         A := D + E;  <----------3
      end; { SUB2 }
   begin { BIGSUB }
      SUB2(7);
   end; { BIGSUB }
begin
   BIGSUB;
end. { MAIN_2 }
```

At position 2 in SUB3:

- E - (0, 4)
- B - (1, 4)
- A - (2, 3)

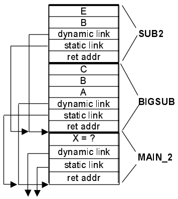


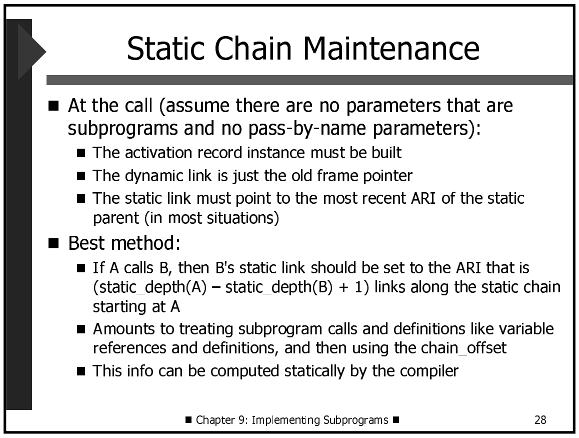
Static Chain Example (8)

```
program MAIN_2;
   var X : integer;
   procedure BIGSUB;
      var A, B, C : integer;
      procedure SUB1;
         var A, D : integer;
      begin { SUB1 }
         A := B + C;  <----------1
      end;  { SUB1 }
      procedure SUB2(X : integer);
         var B, E : integer;
         procedure SUB3;
            var C, E : integer;
         begin { SUB3 }
            SUB1;
            E := B + A:   <------2
         end; { SUB3 }
      begin { SUB2 }
         SUB3;
         A := D + E;  <----------3
      end; { SUB2 }
   begin { BIGSUB }
      SUB2(7);
   end; { BIGSUB }
begin
   BIGSUB;
end. { MAIN_2 }
```

At position 3 in SUB2:

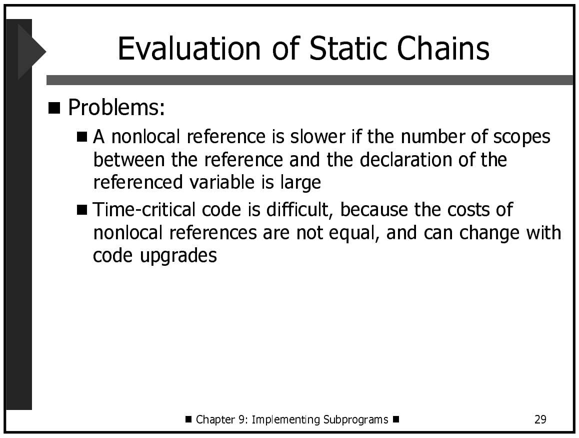
- A - (1, 3)
- D - an error
- E - (0, 5)





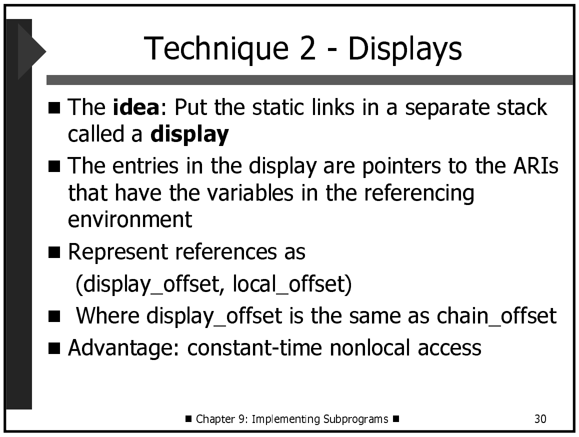
Static Chain Maintenance

- At the call (assume there are no parameters that are subprograms and no pass-by-name parameters):
  - The activation record instance must be built
  - The dynamic link is just the old frame pointer
  - The static link must point to the most recent ARI of the static parent (in most situations)
- Best method:
  - If A calls B, then B's static link should be set to the ARI that is (static_depth(A) – static_depth(B) + 1) links along the static chain starting at A
  - Amounts to treating subprogram calls and definitions like variable references and definitions, and then using the chain_offset
  - This info can be computed statically by the compiler



Evaluation of Static Chains

- Problems:
  - A nonlocal reference is slower if the number of scopes between the reference and the declaration of the referenced variable is large
  - Time-critical code is difficult, because the costs of nonlocal references are not equal, and can change with code upgrades



Technique 2 - Displays

- The **idea**: Put the static links in a separate stack called a **display**
- The entries in the display are pointers to the ARIs that have the variables in the referencing environment
- Represent references as
  (display_offset, local_offset)
- Where display_offset is the same as chain_offset
- Advantage: constant-time nonlocal access

Mechanics of Display References

- Use the display_offset to get the pointer into the display to the ARI with the variable
- Use the local_offset to get to the variable within the ARI
- Display maintenance (assuming no parameters that are subprograms and no pass-by-name parameters):
  - Display_offset depends only on the static_depth of the procedure whose ARI is being built: It is exactly the static_depth of the procedure
  - There are k+1 entries in the display, where k is the static depth of the currently executing unit (k=0 is for the main program)
  - For a call to procedure P with a static_depth of k:
    - Save a copy of the display pointer at position k in new ARI
    - Put the link to the new ARI for P at position k in the display
    - On return, move the saved display pointer from the ARI back into the display at position k

Static Chain vs. Display

- References to locals
  - Not much difference
- References to nonlocals
  - If it is one level away, they are equal
  - If it is farther away, the display is faster
  - Display is better for time-critical code, because all nonlocal references cost the same
- Procedure calls
  - Speed is about the same
  - Display uses more memory
- Procedure returns
  - Both have fixed time, but the static chain is slightly faster
- Overall: Static chain is better, unless the display can be kept in registers

Implementing Blocks

- Two Methods:
  - Treat blocks as parameterless subprograms and give them activation records
  - Allocate locals on top of the ARI of the subprogram
  - Must use a different method to access locals (e.g., frame pointer)

Implementing Dynamic Scoping

- Deep Access
  - Nonlocal references are found by searching the activation record instances on the dynamic chain
  - Length of chain cannot be statically determined
  - Every activation record instance must have variable names
- Shallow Access
  - Put locals in a central place
  - Methods:
    - One stack for each variable name
    - Central table with an entry for each variable name

Subprograms as Parameters

- For deep binding:
  - Static chain
    - Compiler simply passes the link to the static parent of the parameter, along with the parameter
  - Display
    - All pointers to static ancestors must be saved, because none are necessarily in the environment of the parameter
    - In many implementations, the whole display is saved for calls that pass subprogram parameters