Theory of Programming Languages

Week 10

Implementing Subprograms

Last week we covered…

- Fundamentals of Subprograms
- Design Issues for Subprograms
- Local Referencing Environments
- Parameter-Passing Methods
- Parameters That Are Subprograms
- Calling Subprograms Indirectly
  - Shallow and deep binding
- Overloaded Subprograms
- Generic Subprograms

Now we look at subprograms from an implementational point of view…

- The General Semantics of Calls and Returns
- Implementing "Simple" Subprograms
- Implementing Subprograms with Stack-Dynamic Local Variables
- Nested Subprograms
- Blocks
- Implementing Dynamic Scoping

The General Semantics of Calls and Returns

CALL and RETURN operations are together called **subprogram linkage.**

Actions associated with CALL

- Implementation of the parameter-passing method.
- (If local variables are not static): allocation of storage to the locals.
- Saving execution status of the calling program unit (registers, CPU bits, EP).
- Ensuring the transfer of control to the subprogram.
- Ensuring the subprogram can return properly.
- (If the language supports nested subprograms): providing access to nonlocal variables that are visible to the called subprogram.

Actions associated with RETURN

- (If the subprogram has out-mode or in-out mode parameters implemented by copy): move the local values of the associated formal parameters to the actual parameters.
- Deallocate the storage used for local variables.
- Restore the execution status of the calling program unit.
- Return the control to the calling subprogram unit.

Implementing "simple" subprograms

- A **"simple"** subprogram is one in which no subprograms can be nested, and all local variables are static.

➢ **CALL in a simple subprogram**

- Save the execution status of the current program unit.
- Compute and pass the parameters.
- Pass the return address to the called.
- Transfer control to the called.

red: done by the caller
blue: done by the callee

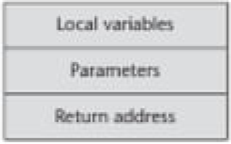
➢ RETURN in a simple subprogram

- If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to or made available to the corresponding actual parameters.
- If the subprogram is a function (operator); the functional value is moved to a place accessible to the caller.
- The execution status of the caller is restored.
- Control is transferred back to the caller.

red: done by the caller
blue: done by the callee

- The actions of the callee when done at the beginning of its execution are called *prologue;* when done at the end of its execution are called *epilogue.*
- A simple subprogram only needs an epilogue.
- CALL and RETURN require storage for the following:
  - Status information about the caller
  - Parameters
  - Return address
  - Return value for functions
  - Temporaries used by the code of the subprograms

- The format, or layout, of the noncode part of a subprogram is called an ***activation record***, because the data it describes is relevant only during the or execution of the subprogram.
- An ***activation record instance*** is a concrete example of an activation record, a collection of data in the form of an activation record.
- Languages with simple subprograms do not support recursion, and need a single activation record.
- A possible layout of activation record for a simple program (the caller ‘s status information is omitted):



Local variables

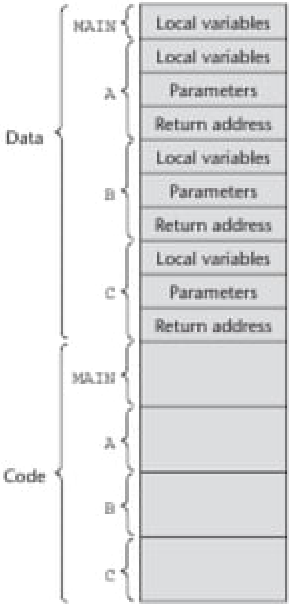
Parameters

Return address

A program consists of…

- A main program and three subprograms: A, B, and C.
- At the time of compilation the machine code for all units along with a list of references to external subprograms, is written to a file.
- The executable program is put together by the linker, which is part of the operating system.

The code and activation records of a program with simple subprograms



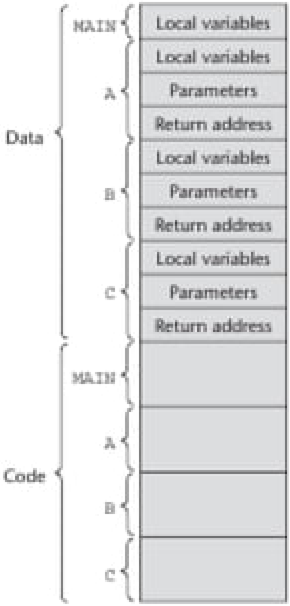
Tasks of a linker…

- When the linker is called for a main program, its first task is to find the files that contain the translated subprograms referenced in that program and load them into memory .

- Then, the linker must set the target addresses of all calls to those subprograms in the main program to the entry addresses of those subprograms.

- The same must be done for all calls to subprograms in the loaded subprograms and all calls to library subprograms.

The code and activation records of a program with simple subprograms



Implementing Subprograms with Stack-Dynamic Local Variables

We need to look at the following:

(I) More complex activation records

(II) An example without recursion

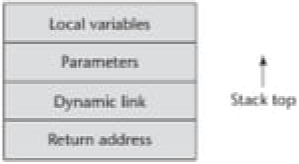
(III) Recursion

(I) More complex activation records

Reasons for complication

- The compiler must generate code to cause the implicit allocation and deallocation of local variables.
- Recursion adds the possibility of multiple simultaneous activations of a subprogram; each of which required their own instances of activation records.
  - In languages with stack-dynamic local variables, activation record instances must be created dynamically.

Because the return address, dynamic link, and parameters are placed in the activation record instance by the caller, these entries must appear first.



Local variables

Parameters

Dynamic link

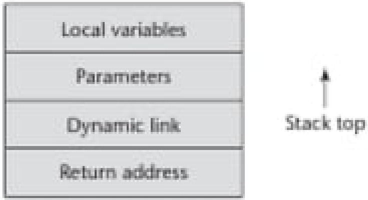
Return address

Stack top

The **dynamic link** is a pointer to the base of the activation record instance of the caller.

Components of the activation record

- In static-scoped languages, the dynamic link is used to provide traceback information when a run-time error occurs.
- In dynamic-scoped languages, the dynamic link is used to access nonlocal variables.
- The actual parameters in the activation record are the values or addresses provided by the caller.
- Local scalar variables are bound to storage within an activation record instance.
- Local variables that are structures are sometimes allocated elsewhere, and only their descriptors and a pointer to that storage are part of the activation record.
- Local variables are allocated and possibly initialized in the called subprogram, so they appear last.



Local variables

Parameters

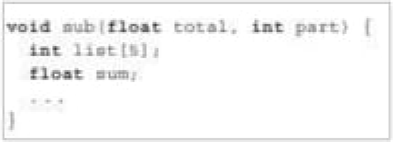
Dynamic link

Return address

Stack top

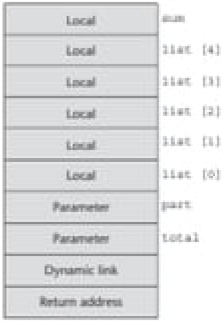
Activation record example

A skeletal C program



```
void sub(float total, int part) {
  int list[5];
  float sum;
  ...
}
```

The corresponding activation record



the mechanism

- Instances of activation record are created on a stack provided by the run-time system, called the ***run-time stack*.**
- Every subprogram activation, recursive or nonrecursive, creates a instance of the activation record on the stack.
- One more thing required to control the execution of a subprogram is the EP: the environment pointer.
  - Initially, the EP points at the base, or first address of the activation record instance of the main program.
  - The run-time system must ensure that it always points to the base of the activation record instance of the currently executing program unit.
  - When a subprogram is called, the current EP is saved in the new activation record instance as the dynamic link.
  - The EP is then set to point at the base of the new activation record
  - instance.
  - Upon return from the subprogram, the stack top is set to the value of the current EP minus one and the EP is set to the dynamic link from the activation record instance of the subprogram that has completed its execution.

Summary of the linkage actions

ACTIONS BY THE CALLER

1. Create an activation record instance.
2. Save the execution status of the current program unit.
3. Compute and pass the parameters.
4. Pass the return address to the callee.
5. Transfer control to the callee.

ACTIONS BY THE CALLEE

Prologue

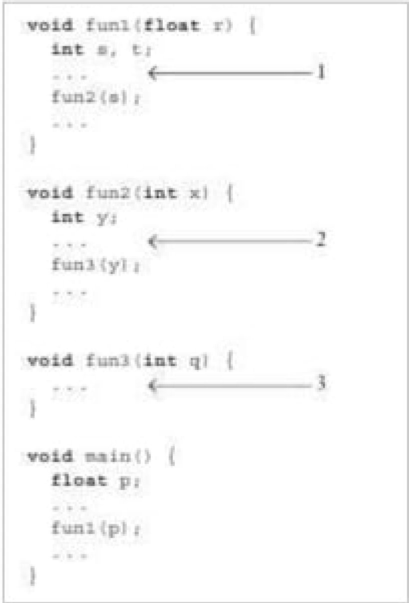
1. Save the old EP in the stack as the dynamic link and create the new value.
2. Allocate local variables.

Epilogue

1. If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to the corresponding actual parameters.
2. If the subprogram is a function, the functional value is moved to a place accessible to the caller.
3. Restore the stack pointer by setting it to the value of the current EP minus one and set the EP to the old dynamic link.
4. Restore the execution status of the caller.
5. Transfer control back to the caller.

(II) An example without recursion

Skeletal C program



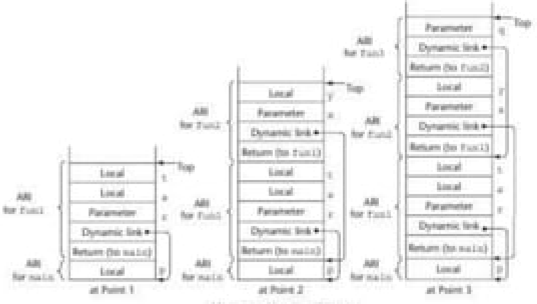
```
void fun1(float r) {
  int s, t;
  ...        <———————— 1
  fun2(s);
  ...
}

void fun2(int x) {
  int y;
  ...        <———————— 2
  fun3(y);
  ...
}

void fun3(int q) {
  ...        <———————— 3
}

void main() {
  float p;
  ...
  fun1(p);
  ...
}
```

Run-time stack at points 1, 2, 3



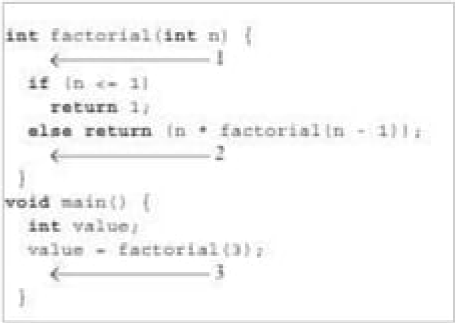
ARI = activation record instance

- The collection of dynamic links present in the stack at a given time is called the **dynamic chain**, or **call chain**.

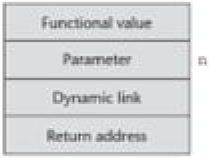
- References to local variables can be represented in the code as offsets from the beginning of the activation record of the local scope, whose address is stored in the EP. Such an offset is called a **local offset**.
  - The local_offset of a variable in an activation record can be determined at compile time, using the order, types, and sizes of variables declared in the subprogram associated with the activation record.

(III) Recursion

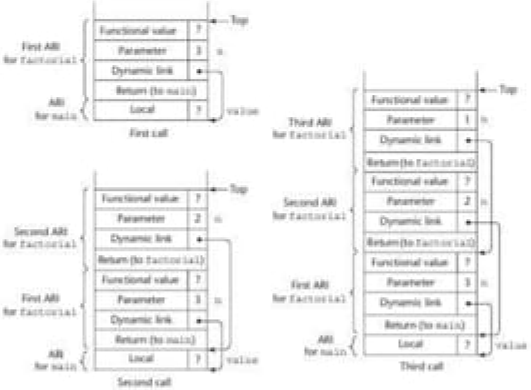
Skeletal C program, and the activation record format



```
int factorial(int n) {
    <------------------ 1
  if (n <= 1)
    return 1;
  else return (n * factorial(n - 1));
    <------------------ 2
  }
void main() {
  int value;
  value = factorial(3);
    <------------------ 3
  }
```



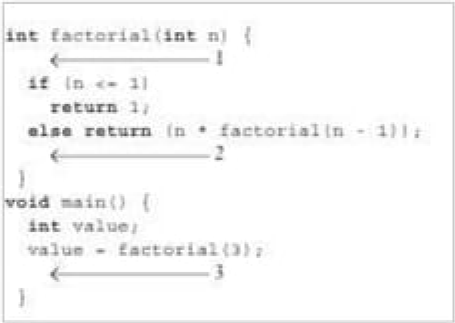
Run-time stack at points 1 (for the three time it occurs during factorial(3))



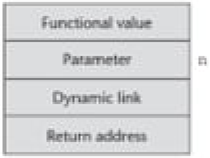
ARI = activation record instance

Notice the extra entry for function return value.

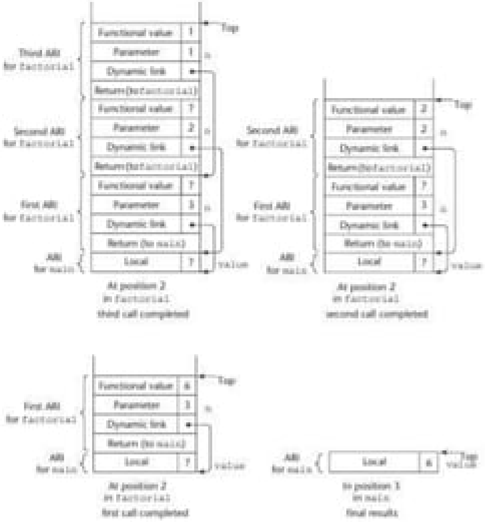
Skeletal C program, and the activation record format



```
int factorial(int n) {
    <------------------ 1
  if (n <= 1)
    return 1;
  else return (n * factorial(n - 1));
    <------------------ 2
  }
void main() {
  int value;
  value = factorial(3);
    <------------------ 3
  }
```



Run-time stack at points 2, 3 (during factorial(3))



Notice the extra entry for function return value.

Implementing Subprograms

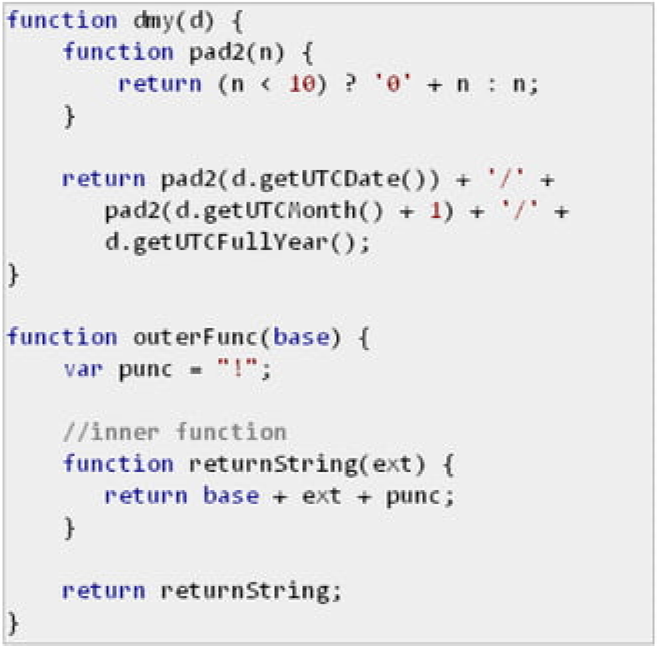
- The General Semantics of Calls and Returns
- Implementing “Simple” Subprograms
- Implementing Subprograms with Stack-Dynamic Local Variables
- Nested Subprograms
- Blocks
- Implementing Dynamic Scoping

Nested Subprograms

Some of the non–C-based static-scoped programming languages use stack-dynamic local variables and allow subprograms to be nested. Among these are: **Fortran 95+**, **Ada**, **Python**, **JavaScript**, **Ruby**.

The major challenge in implementing nested subprograms is to provide access to non-local variables.

Examples of nested subprograms from JavaScript



```
function dmy(d) {
    function pad2(n) {
        return (n < 10) ? '0' + n : n;
    }

    return pad2(d.getUTCDate()) + '/' +
       pad2(d.getUTCMonth() + 1) + '/' +
       d.getUTCFullYear();
}

function outerFunc(base) {
    var punc = "!";

    //inner function
    function returnString(ext) {
       return base + ext + punc;
    }

    return returnString;
}
```

Access to non-local variables

Two steps are involved:

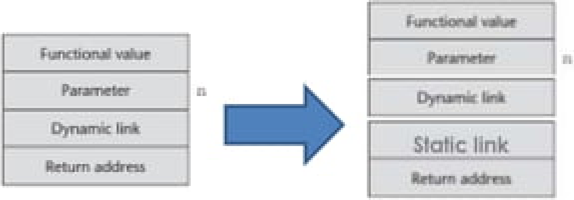
1. Find the instance of the activation record in the stack in which the variable was allocated.
   - In a given subprogram, only variables that are declared in static ancestor scopes are visible and can be accessed.
   - A subprogram is callable only when all of its static ancestor subprograms are active.
   - The correct declaration for a referenced variable is the first one found when looking through the enclosing scopes, most closely nested first.
   - A technique is required to implement these rules of static scoping.

2. Use the *local offset* of the variable (within the activation record instance) to access it.

Static Links

The most common way to implement static scoping in languages that allow nested subprograms is ***static chaining***.

- A new pointer, called a **static link**, is added to the activation record.
- The static link points to the bottom of the activation record instance of an activation of the static parent.
- Typically, the static link appears in the activation record below the parameters.
- Instead of having two activation record elements before the parameters, there are now three: the return address, the static link, and the dynamic link.


Functional value
Parameter
n
Dynamic link
Return address

Functional value
Parameter
n
Dynamic link
Static link
Return address

Static Chains

A ***static chain*** is a chain of static links that connect certain activation record instances in the stack.

During the execution of a subprogram P, the static link of its activation record instance points to an activation record instance of P's static parent program unit. That instance's static link points in turn to P's static grandparent program unit's activation record instance, if there is one. This chain of activation records can be explored to access the entire referencing environment of P – under static scoping.

Finding the correct activation record

Method I

- When a reference is made to a nonlocal variable, the activation record instance containing the variable can be found by searching the static chain until a static ancestor activation record instance is found that contains the variable.

- The drawback in this method is the need to search each activation record along the chain for the referenced variable.

Method II

- *Observation*: Because the nesting of scopes is known at compile time, the compiler can determine not only that a reference is nonlocal but also the length of the static chain that must be followed to reach the activation record instance that contains the nonlocal object.

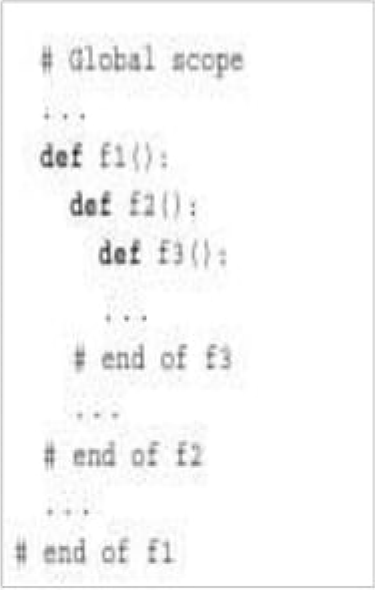
- Each subprogram has an integer associated with it, which contains an integer value representing the depth of the subprogram from the top level subroutine. This integer is called ***static_depth.***

- The length of the static chain needed to reach the correct activation record instance for a nonlocal reference to a variable X is exactly the difference between the static_depth of the subprogram containing the reference to X and the static_depth of the subprogram containing the declaration for X. This difference is called the ***nesting_depth***, or ***chain_offset***, of the reference.

- The actual reference can be represented by an ordered pair of integers :

  (chain_offset, local_offset) – where chain_offset is the number of links to the correct activation record instance , and local_offset is the offset of the variable ,from the EP of the activation record declaring the accessed variable.

Example from Python



```
 # Global scope
 ...
 def f1():
   def f2():
     def f3():
       ...
   # end of f3
   ...
 # end of f2
 ...
# end of f1
```

What are the static_depth values for global scope, f1, f2 and f3?

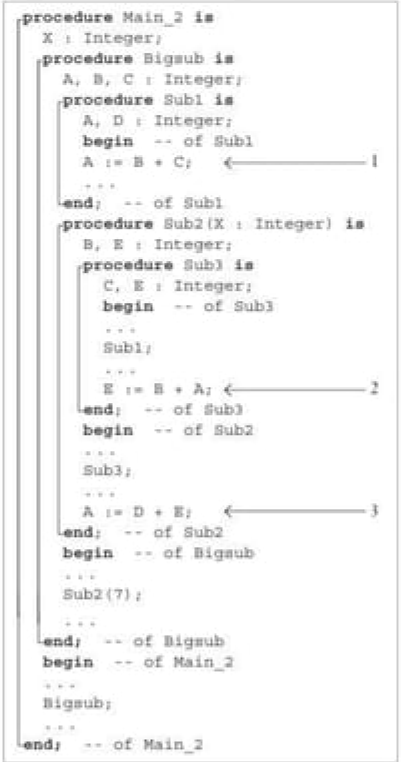
If procedure f3 references a variable declared in f1, the chain_offset of that reference would be?

Working example from Ada

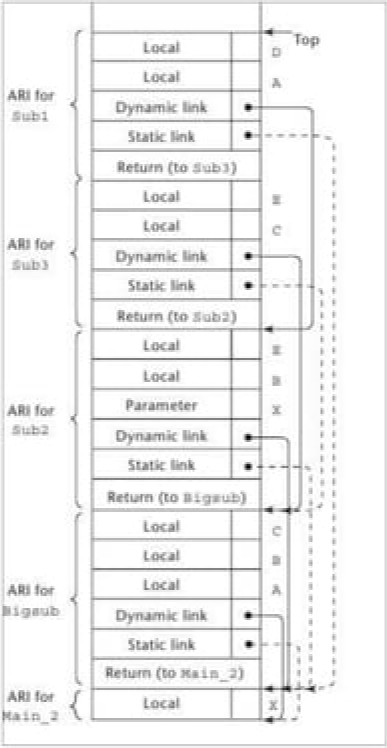
The sequence of procedure calls is:

1) Main_2 **calls** BigSub
2) BigSub **calls** Sub2
3) Sub2 **calls** Sub3
4) Sub3 **calls** Sub1

What are the **(chain_offset, local_offset)** values at positions 1, 2 and 3?



```
procedure Main_2 is
  X : Integer;
  procedure Bigsub is
    A, B, C : Integer;
    procedure Sub1 is
      A, D : Integer;
      begin  -- of Sub1
      A := B + C;    <------------------ 1
      . . .
    end;  -- of Sub1
    procedure Sub2(X : Integer) is
      B, E : Integer;
      procedure Sub3 is
        C, E : Integer;
        begin  -- of Sub3
        . . .
        Sub1;
        . . .
        E := B + A; <------------------ 2
      end;  -- of Sub3
      begin  -- of Sub2
      . . .
      Sub3;
      . . .
      A := D + E;    <------------------ 3
    end;  -- of Sub2
    begin  -- of Bigsub
    . . .
    Sub2(7);
    . . .
  end;  -- of Bigsub
  begin  -- of Main_2
  . . .
  Bigsub;
  . . .
end;  -- of Main_2
```



Stack at position 1

How is the static chain maintained?

- The static chain must be updated for each program call and return.
- **The return part is simple:**
  - When the subprogram terminates, its activation record instance is removed from the stack.
  - After this removal, the new top activation record instance is that of the unit that called the subprogram whose execution just terminated.
  - Because the static chain from this activation record instance was never changed, it works correctly just as it did before the call to the other subprogram.
  - We are done.

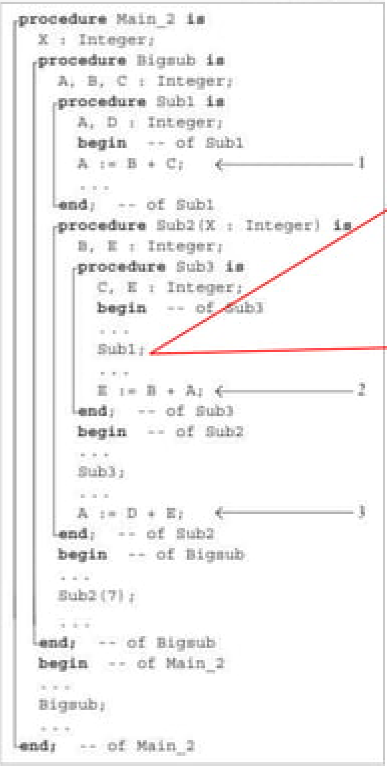
- *Observation:* The most recent activation record instance of the parent scope must be found at the time of the call.
- **Method I:** Look at activation record instances on the dynamic chain, until the first one of the parent scope is found. Problems?
  - Search time can be too much.
- **Method II:** This search can be avoided by treating subprogram declarations and references exactly like variable declarations and references.
  - When the compiler encounters a subprogram call, among other things, it determines the subprogram that declared the called subprogram, and which must be a static ancestor of the calling routine, also.

Method II

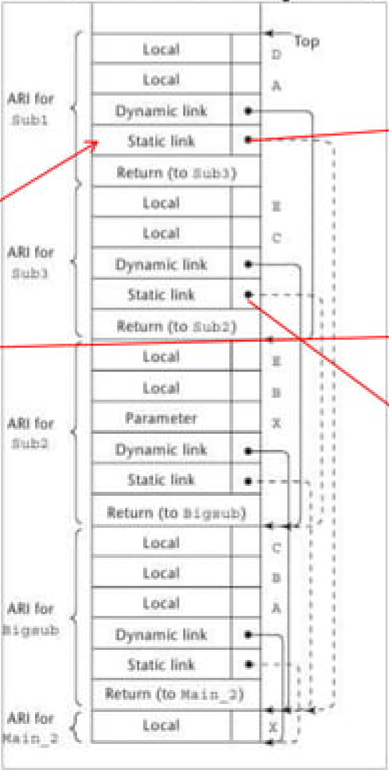
- Consider the situation where Q was called by P.
  - In this situation, if S is the subprogram which declared Q, then, S must also be an ancestor of P. Why?

- What does the compiler do when it encounters the call to Q inside P?
  - It computes the nesting_depth, n, of P w.r.t. S.
  - This information is stored and can be accessed by Q, during execution.
  - The static link of Q is determined by moving down the static chain of P, n times.
  - This ensures that the most recent activation record instance of S is found at the time of the call to Q.

The Ada Example, again



```
procedure Main_2 is
  X : Integer;
  procedure Bigsub is
    A, B, C : Integer;
    procedure Sub1 is
      A, D : Integer;
      begin  -- of Sub1
      A := B + C;  <--------------- 1
      . . .
    end;  -- of Sub1
    procedure Sub2(X : Integer) is
      B, E : Integer;
      procedure Sub3 is
        C, E : Integer;
        begin  -- of Sub3
        . . .
        Sub1;
        . . .
        E := B + A;  <------------- 2
      end;  -- of Sub3
      begin  -- of Sub2
      . . .
      Sub3;
      . . .
      A := D + E;  <--------------- 3
    end;  -- of Sub2
    begin  -- of Bigsub
    . . .
    Sub2(7);
    . . .
  end;  -- of Bigsub
  begin  -- of Main_2
  . . .
  Bigsub;
  . . .
end;  -- of Main_2
```



This is the link being computed.

Q = Sub1, **P** = Sub 3, **S** = BigSub

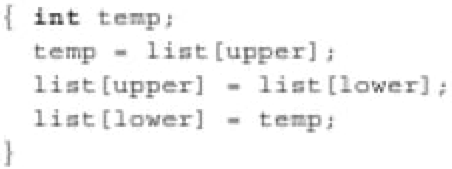
Go 2 jumps down from here to find the static link for Sub1

Criticisms of Static Chaining

- What happens when functions as parameters are allowed?
- Access nonlocal variables is that references to variables in scopes beyond the static parent cost more than references to locals. The static chain must be followed, one link per enclosing scope from the reference to the declaration.
  - Fortunately, in practice, references to distant nonlocal variables are rare
- It is difficult for a programmer working on a time-critical program to estimate the costs of nonlocal references, because the cost of each reference depends on the depth of nesting between the reference and the scope of declaration.
  - Code modifications may change nesting depths, thereby changing the timing of some references, both in the changed code and possibly in code far from the changes.
- However, none of the alternatives suggested for static chaining has been found to be superior to the static-chain method, which is still the most widely used approach.

Blocks

A block is specified in the C-based languages as a compound statement that begins with one or more data definitions. The lifetime of the variable temp in the following block begins when control enters the block and ends when control exits the block.

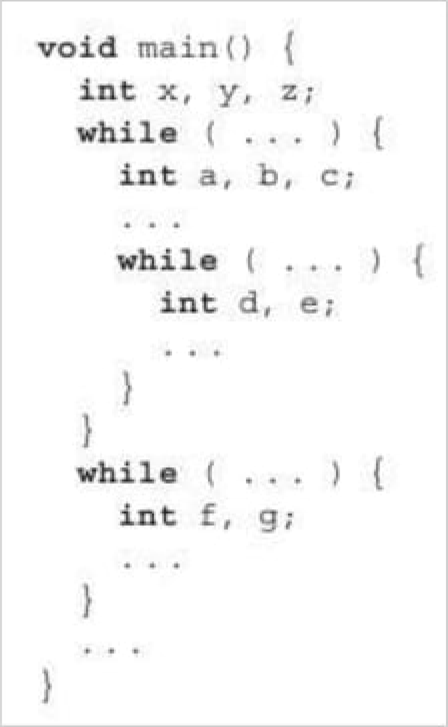


```
{ int temp;
  temp = list[upper];
  list[upper] = list[lower];
  list[lower] = temp;
}
```

How are blocks implemented?

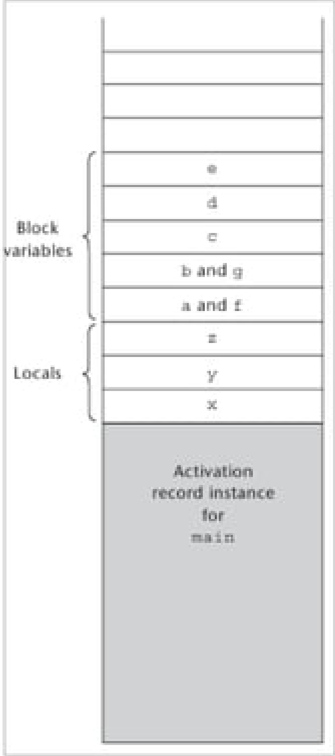
- Blocks can be implemented by using the static-chain process described previously.
- Blocks are treated as parameter-less subprograms that are always called from the same place in the program.
- **Method I (for implementing blocks)**
  - Every block has an activation record. An instance of its activation record is created every time the block is executed.
- **Method II (for implementing blocks)**
  - The maximum amount of storage required for block variables at any time during the execution of a program can be statically determined.
  - This amount of space can be allocated after the local variables in the activation record of the function in which the block resides. So new activation record is created for a block.
  - Offsets for all block variables can be statically computed, so block variables can be addressed exactly as if they were local variables of the function.

Example from C – while blocks in main



```
void main() {
  int x, y, z;
  while ( ... ) {
    int a, b, c;
    ...
    while ( ... ) {
      int d, e;
      ...
    }
  }
  while ( ... ) {
    int f, g;
    ...
  }
  ...
}
```

f and **g** occupy the same space on the stack as **a** and **b**.



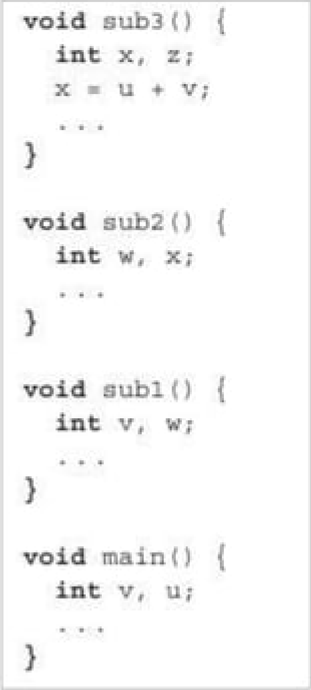
Implementing Dynamic Scoping

- There are two distinct ways in which local variables and nonlocal references to them can be implemented in a dynamic-scoped language:
  - **Deep access**
  - **Shallow access**.

Deep Access

- References to nonlocal variables are resolved by searching through the activation record instances of the other subprograms that are currently active, beginning with the one most recently activated.
- In the case of dynamic scoping, the dynamic links – rather than the static links – are followed.
- The dynamic chain links all subprogram activation record instances in the reverse of the order in which they were activated.
- This method is called deep access, because access may require searches deep into the stack.

Example of Deep access



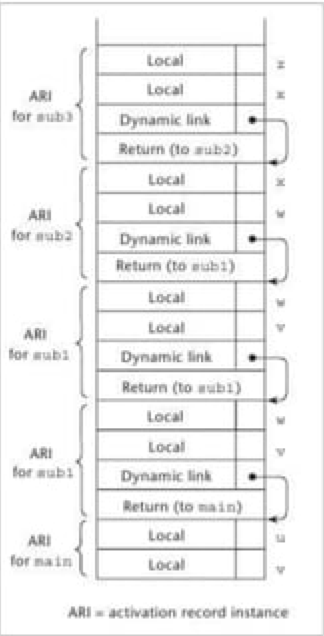
```
void sub3() {
  int x, z;
  x = u + v;
  . . .
}

void sub2() {
  int w, x;
  . . .
}

void sub1() {
  int v, w;
  . . .
}

void main() {
  int v, u;
  . . .
}
```

main calls sub1
sub1 calls sub1
sub1 calls sub2
sub2 calls sub3



Differences between Deep Access and Static Chaining

- In a dynamic-scoped language, there is no way to determine at compile time the length of the chain that must be searched.
  - Every activation record instance in the chain must be searched until the first instance of the variable is found.
  - This is one reason why dynamic-scoped languages typically have slower execution speeds than static-scoped languages.

- In dynamic scoped languages, activation records must store the names of variables for the search process, whereas in static-scoped language implementations only the values are required.
  - Names are not required for static scoping, because all variables are represented by the (chain_offset, local_offset) pairs.

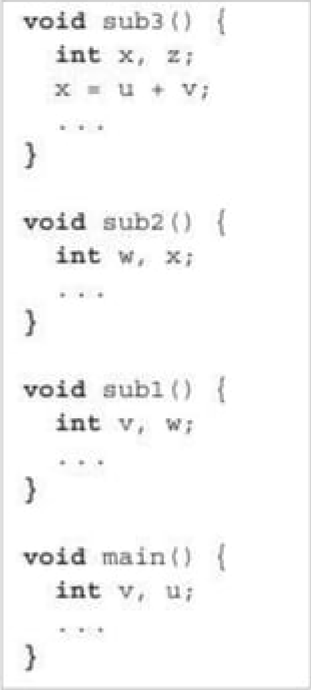
Shallow Access

- In the shallow-access method, variables declared in subprograms are not stored in the activation records of those subprograms.
- Because with dynamic scoping there is at most one visible version of a variable of any specific name at a given time, a very different approach can be taken.

Method I

- One variation of shallow access is to have a separate stack for each variable name in a complete program.
  - Every time a new variable with a particular name is created by a declaration at the beginning of a subprogram that has been called, the variable is given a cell at the top of the stack for its name.
  - Every reference to the name is to the variable on top of the stack associated with that name, because the top one is the most recently created.
  - When a subprogram terminates, the lifetimes of its local variables end, and the stacks for those variable names are popped.
  - This method allows fast references to variables, but maintaining the stacks at the entrances and exits of subprograms is costly.

Example of Shallow access



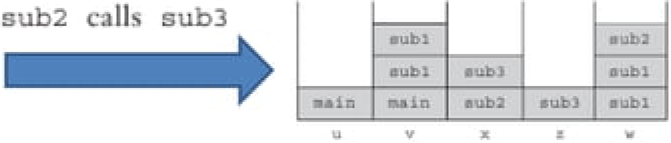
```
void sub3() {
  int x, z;
  x = u + v;
  . . .
}

void sub2() {
  int w, x;
  . . .
}

void sub1() {
  int v, w;
  . . .
}

void main() {
  int v, u;
  . . .
}
```

main calls sub1
sub1 calls sub1
sub1 calls sub2
sub2 calls sub3



Method II

- Another option for implementing shallow access is to use a central table that has a location for each different variable name in a program.
- Along with each entry, a bit called active is maintained that indicates whether the name has a current binding or variable association.
- Any access to any variable can then be to an offset into the central table.
- The offset is static, so the access can be fast.

Reading Assignment: Figure out how the SNOBOL language uses the central table approach.

Maintenance of the central table

- A subprogram call requires that all of its local variables be logically placed in the central table.
- If the position of the new variable in the central table is already active, that value must be saved somewhere during the lifetime of the new variable.
  - One variation is to have a "hidden" stack on which all saved objects are stored. Because subprogram calls and returns, and thus the lifetimes of local variables, are nested, this works well.
  - In another variation, replaced variables are stored in the activation record of the subprogram that created the replacement variable.
- Whenever a variable begins its lifetime, the active bit in its central table position must be set.

How does SNOBOL store values when they are temporarily replaced?

Deep, or Shallow access?

- The choice between shallow and deep access to nonlocal variables depends on the relative frequencies of subprogram calls and nonlocal references.
  - The deep-access method provides fast subprogram linkage, but references to nonlocals are costly.
  - The shallow-access method provides much faster references to nonlocals, especially distant nonlocals, but is more costly in terms of subprogram linkage

We examined the following issues in **Implementing Subprograms**

- The General Semantics of Calls and Returns
- Implementing “Simple” Subprograms
- Implementing Subprograms with Stack-Dynamic Local Variables
- Nested Subprograms
- Blocks
- Implementing Dynamic Scoping

starting next class:

Support for Object Oriented Programming

- Introduction
- Object-Oriented Programming
- Design Issues for Object-Oriented Languages
- Support for Object-Oriented Programming in Smalltalk
- Support for Object-Oriented Programming in C++
- Support for Object-Oriented Programming in Objective-C
- Support for Object-Oriented Programming in Java
- Support for Object-Oriented Programming in C#
- Support for Object-Oriented Programming in Ada 95
- Support for Object-Oriented Programming in Ruby
- Implementation of Object-Oriented Constructs