UNIT - III

Fundamentals of Subprograms

General subprogram characteristics :

- Each subprogram has a single entry point
- The calling program is suspended during execution of the called subprogram
- Control always returns to the caller when the called subprogram's execution terminates

Basic Definitions

A *subprogram definition* describes the interface to and the actions of the subprogram abstraction
- In Python, function definitions are executable; in all other languages, they are non-executable

- A *subprogram call* is an explicit request that the subprogram be executed
- A *subprogram header* is the first part of the definition, including the name, the kind of subprogram, and the formal parameters
- The *protocol* is a subprogram's parameter profile and, if it is a function, its return type
- Function declarations in C and C++ are often called *prototypes*

parameters

There are two ways that subprogram can gain access to the data that it is to process:
Direct access to nonlocal variables
Parameter passing
** Parameter passing is more flexible than direct access to nonlocal variables

- A *formal parameter* is a dummy variable listed in the subprogram header and used in the subprogram
- An *actual parameter* represents a value or address used in the subprogram call statement
- **Positional parameters:**
  is a method for binding actual parameter to formal parameter, is done by position
  ** ((the first actual parameters is bound to the first formal parameter and so forth such parameters are called **positional parameters**))
- **Keyword parameters :**
  The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter
  Advantage:
  That they can appear in any order in the actual parameter list
  Disadvantage:
  That the user of the subprogram must know the names of formal parameters

```
int cube(int);                    prototype
int main(){
          int y=5;                        actual parameters
         cout<<cube(y);                   Subprogram call
           int x=3;
int cube (int x);          subprogram header
{
                              formal parameter
return x*x;
}
}
```

Procedures and Functions

There are two distinct categories of subprograms—procedures and functions

- *Procedures* are collection of statements that define parameterized computations . *procedures have no return values*
- *Functions* structurally resemble procedures but are semantically modeled on mathematical functions. Functions have return values
    - procedures are expected to produce no side effects
    - In practice, program functions have side effec

Procedures define new statements. For example, if a particular language does not have a sort statement, a user can build a procedure to sort arrays of data and use a call to that procedure in place of the unavailable sort statement. In Ada, procedures are called just that; in Fortran, they are called subroutines. Most other languages do not support procedures.

Procedures can produce results in the calling program unit by two methods:

(1) If there are variables that are not formal parameters but are still visible in both the procedure and the calling program unit, the procedure can change them;
(2) if the procedure has formal parameters that allow the transfer of data to the caller, those parameters can be changed.

procedure example as follows

```
procedure add;
       var
         x : integer ;
         y : integer ;
       begin
         read ( x , y );
         write ( x + y );
       end
```

Procedures has Two parts :

The speification and the Body
The specification is begins with the keyword PROCEDURE and ends
With the procedure name or parameter list

Ex : **procedure** a_test(a,b : in integer ; c:out Integer)

- Functions define new user-defined operators. For example, if a language does not have an exponentiation operator, a function can be written that returns the value of one of its parameters raised to the power of another parameter. Its header in C++ could be

  float `power(`**float** `base,` **float** `exp)`
  which could be called with

  ```
  result = 3.4 * power(10.0, x)
  ```

  The standard C++ library already includes a similar function named `pow`.

Example

Void sort (**int** list[], **int** listlen); // function header
…
Sort(scores,100); // function call

Design Issues for Subprograms

Subprograms are complex structures in programming languages. An **overloaded subprogram** is one that has the same name as another subprogram in the same referencing environment. A **generic subprogram** is one whose computation can be done on data of different types in different calls.

- Are local variables static or dynamic?
- Can subprogram definitions appear in other subprogram definitions?
- What parameter passing methods are provided?
- Are parameter types checked?
- If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?
- Can subprograms be overloaded?
- Can subprogram be generic?

Local Referencing Environments

Local Variables

Subprograms can define their own variables, thereby defining local referencing environments. Variables that are defined inside subprograms are called **local variables**, because their scope is usually the body of the subprogram in which they are defined.

local variables can be either static or stack dynamic.

If local variables are stack dynamic, they are bound to storage when the subprogram begins execution and are unbound from storage when that execution terminate

- Advantages of stack dynamic variables
  - Support for recursion
  - Storage for locals is shared among some subprograms
- Main disadvantages of stack dynamic local variables are:

  - Cost of time required to allocate, initialize and de-allocate for each activation
  - Accesses of stack dynamic local variables must be indirect(indirect addressing), where accesses to static can be direct

    The subprograms cannot retain data values of Stack dynamic variables between calls.
- The primary advantage of static local variables is that they are very efficient because of no indirection

Example of ststic and stack dynamic variables as follows

```
int adder(int list[], int listlen)
{
static int sum = 0;
int count;
for (count = 0; count < listlen; count ++)
sum += list [count];
return sum;
}
```

In C and C++ functions, locals are stack dynamic unless specifically declared to be **static.** For example, in the following C (or C++) function, the variable sum is static and count is stack dynamic

Nested Subprograms

If a subprogram is defined within another subprogram, The idea of nesting subprograms originated with Algol 60.Ada supports nested Subprograms.

Parameter passing methods

- Parameter-passing methods are the ways in which parameters are transmitted to and / or from called programs

Semantic Models of Parameter Passing

- Formal parameters are characterized by one of three semantics models:
  - They can receive data from the corresponding actual parameter
  - They can transmit data to the actual parameter, OR
  - They can do both.
- These three semantics models are called **in mode**, **out mode** and **inout mode**, respectively.

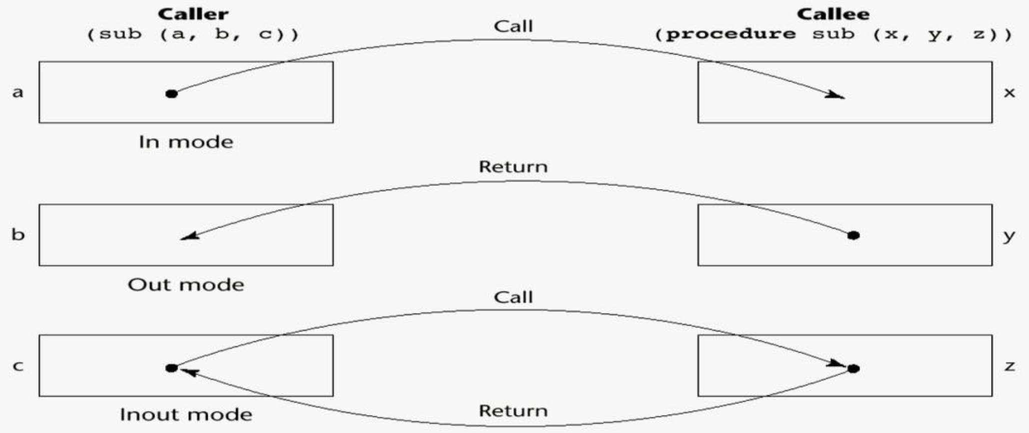


Fig: models of parameter passing

Pass-by-Value (in Mode)

When a parameter is **passed by value**, the value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable in the subprogram – this implements in-mode semantics.
In pass by value actual parameters are passed to the formal parameters and operation is done on the formal parameters.

  - Normally implemented by copying
  - Can be implemented by transmitting an access path but not recommended (enforcing write protection is not easy)
  - *Disadvantages* (if by physical move): additional storage is required (stored twice) and the actual move can be costly (for large parameters)
  - *Disadvantages* (if by access path method): must write-protect in the called subprogram and accesses cost more (indirect addressing)

Pass-by-Result (Out Mode)

- When a parameter is passed by result, no value is transmitted to the subprogram; the corresponding formal parameter acts as a local variable; its value is transmitted to caller's actual parameter when control is returned to the caller, by physical move
- Pass by result parameters are for returning values, not passing data to the procedure.
  - Require extra storage location and copy operation

Pass-by-Value-Result (inout Mode)

- A combination of pass-by-value and pass-by-result. **Pass by value result** is an implementation model for in-out mode parameters in which actual values are moved
- Disadvantages:
  - Those of pass-by-result
  - Those of pass-by-value

PASS BY REFERENCE((inout Mode)

Pass by reference is a second implementation of in-out mode parameters Rather than retransmitting data values back and forth, as in pass by value result, the pass by reference method transmits an access path, usually just an address, to the called subprogram. This provides the access path to the cell storing the actual parameter.

- The advantage of pass by reference is efficiency in both time and space.
- The disadvantages are:
  - Access to formal parameters is slow
  - Inadvertent(unintentional) and erroneous changes may be made to the actual parameter
  - Aliases can be created.

PASS BY NAME

- **Pass by name** is an in-out mode parameter transmission method that does not correspond to a single implementation model.
- When parameters are passed by name, the actual parameter is textually substituted for the corresponding formal parameter in all its occurrences in the subprogram.

C programming language uses *call by value* method to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows.

```
/* function definition to swap the values */
void swap(int x, int y)
{
   int temp;

   temp = x; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put temp into y */

   return;
}
```

Now, let us call the function **swap()** by passing actual values as in the following example:

```
#include <stdio.h>

/* function declaration */
void swap(int x, int y);

int main ()
{
   /* local variable definition */
   int a = 100;
   int b = 200;

   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );

   /* calling a function to swap the values */
   swap(a, b);
   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );

   return 0;
}
```

Let us put above code in a single C file, compile and execute it, it will produce the following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

call by reference example in C as follows

```
/* function definition to swap the values */
void swap(int *x, int *y)
{
   int temp;
   temp = *x;    /* save the value at address x */
   *x = *y;      /* put y into x */
   *y = temp;    /* put temp into y */

   return;
}
```

To check the more detail about C - Pointers, you can check **C - Pointers** chapter.

For now, let us call the function **swap()** by passing values by reference as in the following example:

```
#include <stdio.h>

/* function declaration */
void swap(int *x, int *y);

int main ()
```

```
{
   /* local variable definition */
   int a = 100;
   int b = 200;

   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );

   /* calling a function to swap the values.
    * &a indicates pointer to a ie. address of variable a and
    * &b indicates pointer to b ie. address of variable b.
   */
   swap(&a, &b);

   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );

   return 0;
}
```

Let us put above code in a single C file, compile and execute it, it will produce the following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

Pass by value and pass by reference in C++ as follows

```
main()
 {
 int i = 10, j = 20;
swapThemByVal(i, j);
cout << i << " " << j << endl; // displays 10 20
 swapThemByRef(i, j);
 cout << i << " " << j << endl; // displays 20 10 ... }
```

```
void swapThemByVal(int num1, int num2)
{
int temp = num1;
num1 = num2;
num2 = temp;
}
void swapThemByRef(int& num1, int& num2)
 {
int temp = num1;
num1 = num2;
num2 = temp;
 }

main( )
 {
 int i = 10, j = 20;
 swapThemByVal(i, j); cout << i << " " << j << endl; // displays 10 20
swapThemByRef(i, j); cout << i << " " << j << endl; // displays 20 10 .
 }
```

Examples of Parameter Passing in C,C++, Ada

Consider the following C function:

```
void swap1(int a, int b) {
int temp = a;
a = b;
b = temp;
}
```

Suppose this function is called with

```
swap1(c, d);
```

Recall that C uses pass-by-value. The actions of `swap1` can be described by the following pseudocode:

```
a = c — Move first parameter value in
b = d — Move second parameter value in
temp = a
a = b
b = temp
```

Although `a` ends up with `d`'s value and `b` ends up with `c`'s value, the values of `c` and `d` are unchanged because nothing is transmitted back to the caller.

We can modify the C swap function to deal with pointer parameters to achieve the effect of pass-by-reference:

```
void swap2(int *a, int *b) {
int temp = *a;
*a = *b;
*b = temp;
}
```

`swap2` can be called with

```
swap2(&c, &d);
```

The actions of `swap2` can be described with

```
a = &c — Move first parameter address in
b = &d — Move second parameter address in
temp = *a
*a = *b
*b = temp
```

In this case, the swap operation is successful: The values of `c` and `d` are in fact interchanged. `swap2` can be written in C++ using reference parameters as follows:

```
void swap2(int &a, int &b) {
int temp = a;
a = b;
b = temp;
}
```

This simple swap operation is not possible in Java, because it has neither pointers nor C++'s kind of references. In Java, a reference variable can point to only an object, not a scalar value. The semantics of pass-by-value-result is identical to those of pass-byreference, except when aliasing is involved. Recall that Ada uses pass-by-valueresult for inout-mode scalar parameters. To explore pass-by-value-result, consider the following function, `swap3`, which we assume uses pass-by-value result parameters. It is written in a syntax similar to that of Ada.

```
procedure swap3(a : in out Integer, b : in out Integer) is
temp : Integer;
begin
temp := a;
a := b;
b := temp;
end swap3;
```

Suppose `swap3` is called with

```
swap3(c, d);
```

The actions of `swap3` with this call are

```
addr_c = &c — Move first parameter address in
addr_d = &d — Move second parameter address in
a = *addr_c — Move first parameter value in
b = *addr_d — Move second parameter value in
temp = a
a = b
b = temp
*addr_c = a — Move first parameter value out
*addr_d = b — Move second parameter value out
```

So once again, this swap subprogram operates correctly. Next, consider the call

```
swap3(i, list[i]);
```

In this case, the actions are

```
addr_i = &i — Move first parameter address in
addr_listi= &list[i] — Move second parameter address in
a = *addr_i — Move first parameter value in
b = *addr_listi — Move second parameter value in
temp = a
a = b
b = temp
*addr_i = a — Move first parameter value out
*addr_listi = b — Move second parameter value out
```

MULTIDIMENSIONAL ARRAYS AS PARAMETERS

- In some languages like C or C++, when a multidimensional array is passed as a parameter to a subprogram, the compiler must be able to build the mapping function for that array while seeing only the text of the subprogram. This is true because the subprograms can be compiled separately from the programs that call them.
- The problem with this method of passing matrices as parameters is that it does not allow the programmer t write a function that can accept matrices with different numbers of columns – a new function must be written for every matrix with a different number of columns. This disallows writing flexible functions that may be effectively reusable if the functions deal with multidimensional arrays.

OVERLOADED SUBPROGRAMS

- An **overloaded subprogram** is a subprogram that has the same name as another subprogram in the same referencing environment
- Every version of an overloaded subprogram must have a unique protocol, that is, it must be different from the others in the number, order, or types of its parameters, or in its return types if it is a function
- C++, Java, and Ada include predefined overloaded subprograms

GENERIC SUBPROGRAMS

- A **generic** or **polymorphic subprogram** takes parameters of different types on different activations.
- Overloaded subprograms provide a particular kind of polymorphism called **ad hoc polymorphism.**
- **Parametric polymorphism** is provided by a subprogram that takes a generic parameter that is used in a type expression that describes the types of the parameter of the subprogram.
- Ada and C++ provide a kind of compile-time parametric polymorphism.

The General Semantics of Calls and Returns

The subprogram call and return operations of a language are together called its *subprogram linkage*
The implementation of subprograms must be based on the semantics of the subprogram linkage of the language being implemented. A subprogram call in a typical language has numerous actions associated with it. The call process must include the implementation of whatever parameter-passing method is used. If local variables are not static, the call process must allocate storage for the locals declared in the called subprogram and bind those variables to that storage. It must save the execution status of the calling program unit

- General semantics of subprogram calls
    - Parameter passing methods
    - Stack-dynamic allocation of local variables
    - Save the execution status of calling program
    - Transfer of control and arrange for the return
    - If subprogram nesting is supported, access to nonlocal variables must be arranged
- General semantics of subprogram returns:
    - In mode and inout mode parameters must have their values returned
    - Deallocation of stack-dynamic locals
    - Restore the execution status
    - Return control to the caller

Implementing “Simple” Subprograms

- The semantics of a call to a “simple” subprogram requires the following actions
  - Save the execution status of the caller
  - Pass the parameters
  - Pass the return address to the callee
  - Transfer control to the callee
- The semantics of a return from a simple subprogram requires the following actions:
  - If pass-by-value-result or out mode parameters are used, move the current values of those parameters to their corresponding actual parameters
  - If it is a function, move the functional value to a place the caller can get it
  - Restore the execution status of the caller
  - Transfer control back to the caller
- The call and return actions 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

- A simple subprogram consists of two separate parts Two separate parts: the actual code of the subprogram and the non-code part (local variables and data that can change when the subprogram is executed)
- The format, or layout, of the non-code part of an executing subprogram is called an *activation record*
- An *activation record instance* is a concrete example of an activation record (the collection of data for a particular subprogram activation)

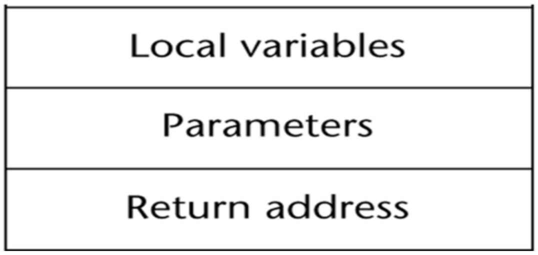
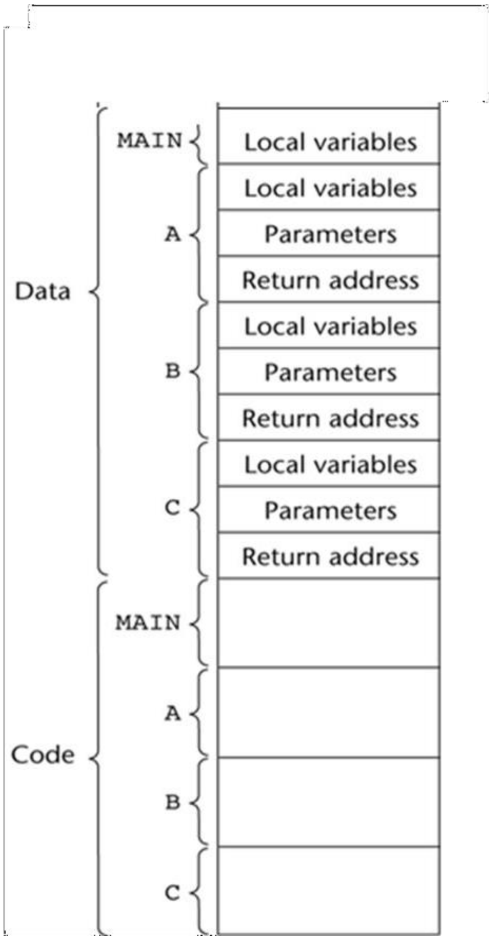


fig: An activation record for simple subprograms

Code and Activation Records of a Program with “Simple” Subprograms is as follows

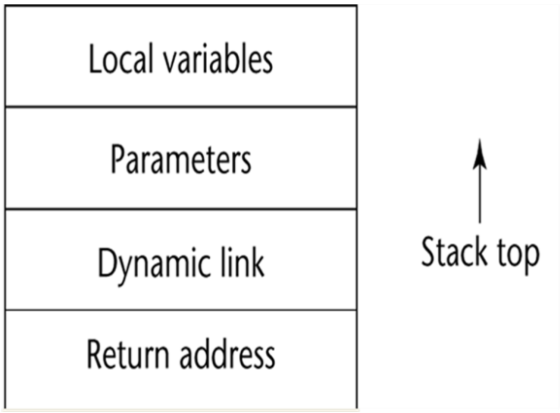


Implementing Subprograms with Stack-Dynamic Local Variables

One of the most important advantages of stack-dynamic local variables is support for recursion. Therefore, languages that use stack-dynamic local variables also support recursion

- More complex activation record
    - The compiler must generate code to cause implicit allocation and deallocation of local variables
    - Recursion must be supported (adds the possibility of multiple simultaneous activations of a subprogram)

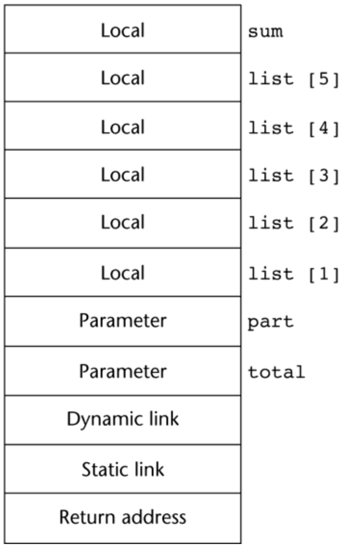
Typical Activation Record for a Language with Stack-Dynamic Local Variables



- The portion of the stack used for an invocation of a function is called the function’s activation record
- The format of an AR for a given subprogram in most languages is known at compile time
- The activation record format is static, but its size may be dynamic
- The *dynamic link* points to the top of an instance of the activation record of the caller
- An activation record instance is dynamically created when a subprogram is called
- Activation record instances reside on the run-time stack
- Return address is address of instruction following the function call

An Example: C Function

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

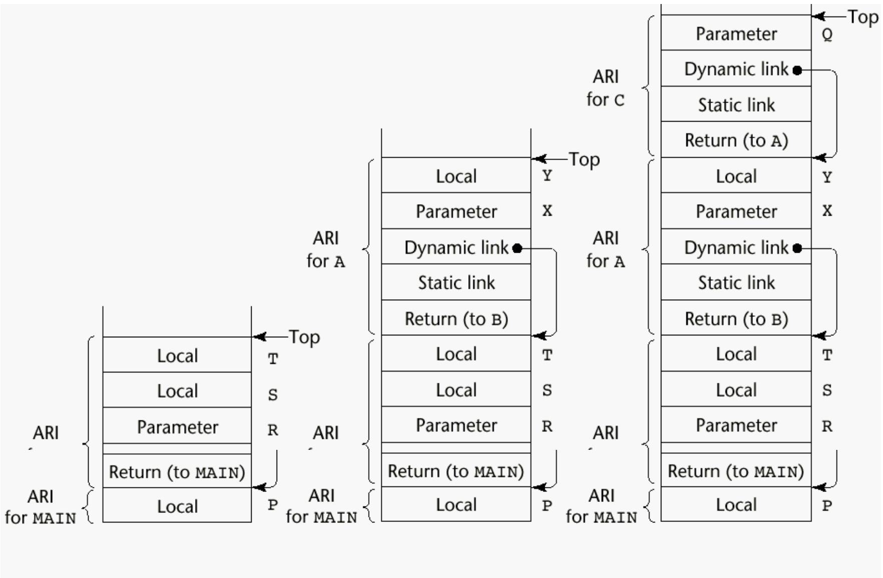


An Example Without Recursion

```
void A(int x) {
   int y;
   ...
   C(y);
   ...
}
void B(float r) {
   int s, t;
   ...
   A(s);
   ...
}
void C(int q) {
   ...
}
void main() {
   float p;
   ...
   B(p);
   ...
}
```

main calls B
B calls A
A calls C

An Example Without Recursion



ARI = activation record instance

Dynamic Chain and Local Offset

- 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, whose address is in the EP. This offset is called the *local_offset*
- The local_offset of a local variable can be determined by the compiler at compile time

An Example With Recursion

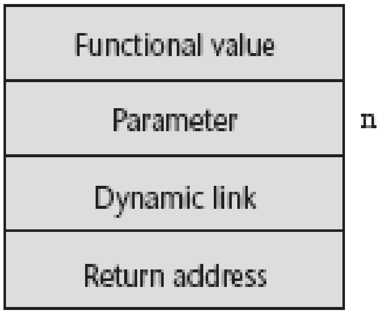
- The activation record used in the previous example supports recursion, e.g.

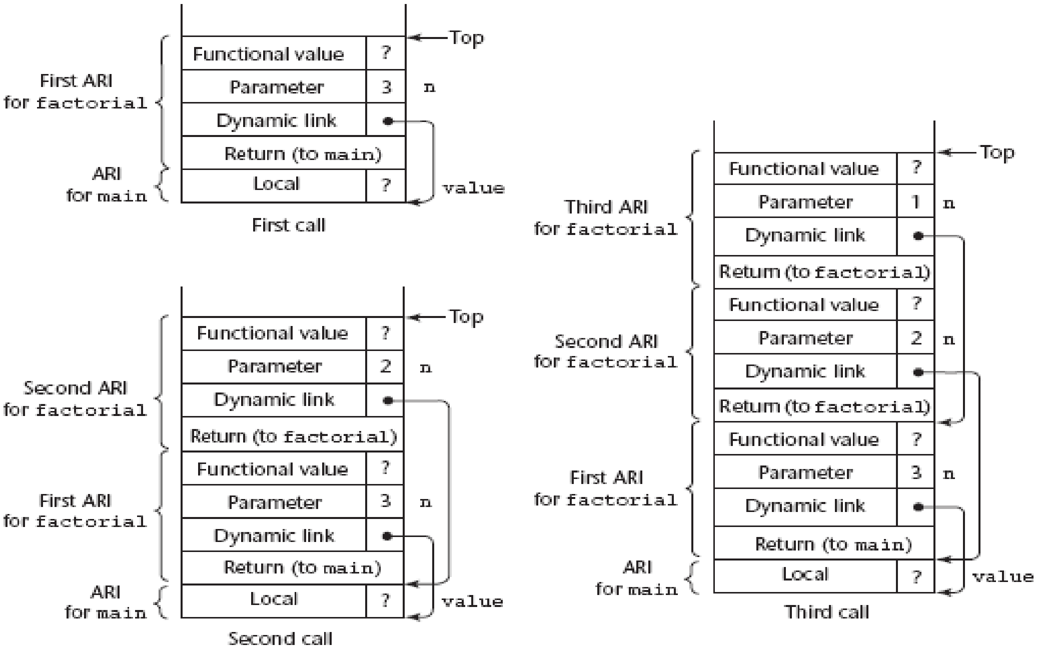
```
 int factorial (int n) {
   <.................................1
  if (n <= 1) return 1;
  else return (n * factorial(n - 1));
   <.................................2
}
void main()
 { int value;
  value = factorial(3);
   <.................................3
}
```

Activation Record for factorial

Figure 10.6

The activation record for `factorial`





First call

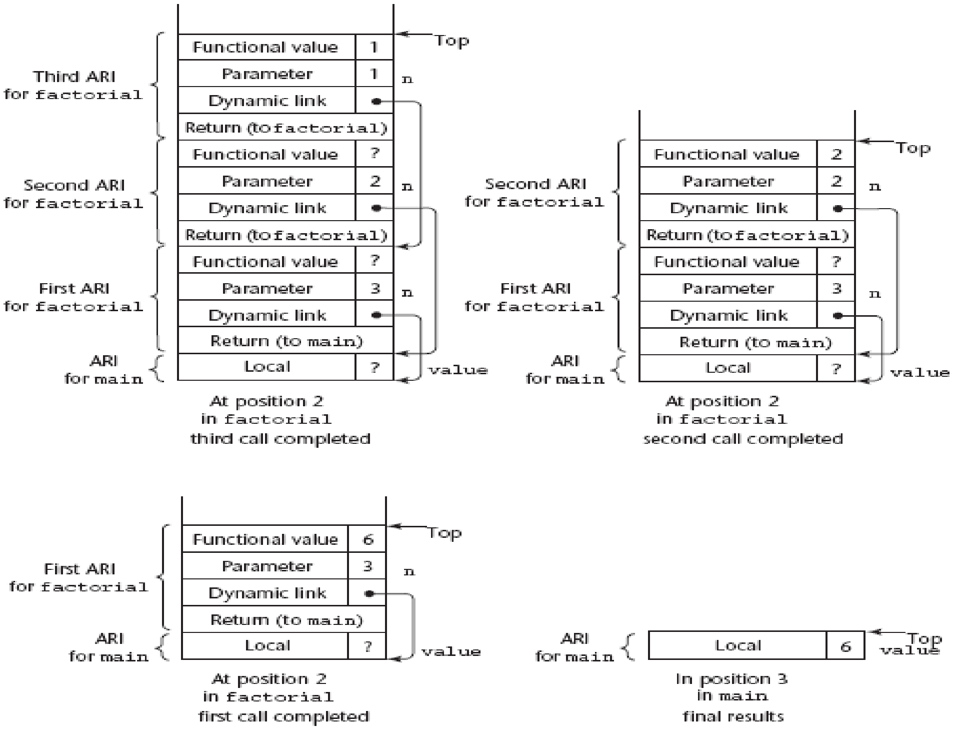
Second call

Third call

ARI = activation record instance

Figure 10.7

Stack contents at position 1 in `factorial`



At position 2 in `factorial` third call completed

At position 2 in `factorial` second call completed

At position 2 in `factorial` first call completed

In position 3 in `main` final results

ARI = activation record instance

Figure 10.8

Stack contents during execution of `main` and `factorial`

Nested Subprograms

- Some non-C-based static-scoped languages (e.g., Fortran 95, Ada, Python, JavaScript, Ruby, and Lua) use stack-dynamic local variables and allow subprograms to be nested
- All variables that can be non-locally accessed reside in some activation record instance in the stack
- The process of locating a non-local reference:
    1. Find the correct activation record instance
    2. Determine the correct offset within that activation record instance

Locating a Non-local Reference

- Finding the offset is easy
- Finding the correct activation record instance
    - Static semantic rules guarantee that all non-local variables that can be referenced have been allocated in some activation record instance that is on the stack when the reference is made

static scoping

- A *static chain* is a chain of static links that connects certain activation record instances
- The *static link* in an activation record instance for subprogram A points to one of the activation record instances of A's static parent
- The static chain from an activation record instance connects it to all of its static ancestors
- *Static_depth* is an integer associated with a static scope whose value is the depth of nesting of that scope
- 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 when it is declared
- A reference to a variable can be represented by the pair:

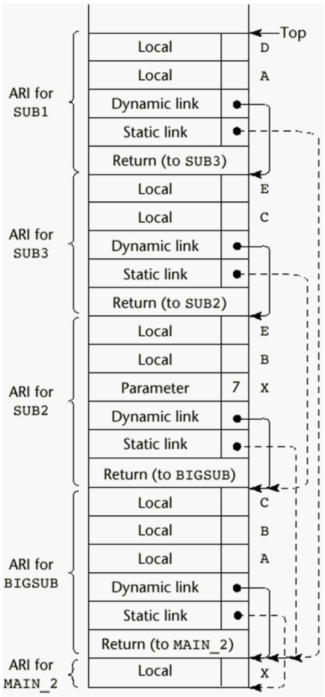
(chain_offset, local_offset), where local_offset is the offset in the activation record of the variable being referenced

Example Ada Program

```
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
  Bigsub;
end; of Main_2 }
```

- **Call sequence for Main_2 in following Ada program**

  Main_2 calls
  Bigsub Bigsub calls
  Sub2 Sub2 calls
  Sub3 Sub3 calls
  Sub1



Evaluation of Static Chains

- Problems:
  1. A nonlocal areference is slow if the nesting depth is large
  2. Time-critical code is difficult:
     a. Costs of nonlocal references are difficult to determine
     b. Code changes can change the nesting depth, and therefore the cost

Displays

- An alternative to static chains that solves the problems with that approach
- Static links are stored in a single array called a display
- The contents of the display at any given time is a list of addresses of the accessible activation record instances

Blocks

- Blocks are user-specified local scopes for variables
- An example in C

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

- The lifetime of temp in the above example begins when control enters the block
- An advantage of using a local variable like temp is that it cannot interfere with any other variable with the same name

Implementing Blocks

- Two Methods:
    1. Treat blocks as parameter-less subprograms that are always called from the same location
        - Every block has an activation record; an instance is created every time the block is executed
    2. Since the maximum storage required for a block can be statically determined, this amount of space can be allocated after the local variables in the activation record

Implementing Dynamic Scoping

If local variables are stack dynamic and are part of the activation records in a dynamic-scoped language, references to nonlocal variables can be resolved by searching through the activation record instances of the other subprograms that are currently active, beginning with the one most recently activated. This concept is similar to that of accessing nonlocal variables in a static-scoped language with nested subprograms, except that the dynamic—rather than the static—chain is followed. The dynamic chain links together all subprogram

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

Using Shallow Access to Implement Dynamic Scoping

```
void sub3() {
  int x, z;
  x = u + v;
  …
}
void sub2() {
  int w, x;
  …
}
void sub1() {
  int v, w;
  …
}
void main() {
  int v, u;
  …}
```

Suppose the following sequence of function calls occurs:

`main` calls `sub1`

`sub1` calls `sub1`
`sub1` calls `sub2`
`sub2` calls `sub3`

Figure 10.11 shows the stack during the execution of function `sub3` after this calling sequence. Notice that the activation record instances do not have static links, which would serve no purpose in a dynamic-scoped language. Consider the references to the variables `x`, `u`, and `v` in function `sub3`. The reference to `x` is found in the activation record instance for `sub3`. The reference to `u` is found by searching *all* of the activation record instances on the stack, because the only existing variable with that name is in `main`. This search involves following four dynamic links and examining 10 variable names. The reference to `v` is found in the most recent (nearest on the dynamic chain) activation record instance for the subprogram `sub1`.

Figure 10.11

Stack contents for a dynamic-scoped program

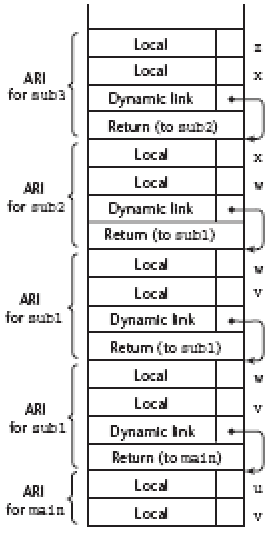
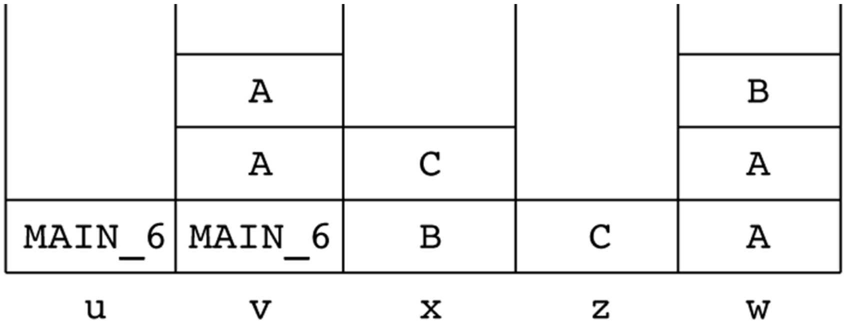


Figure 10.12 shows the variable stacks for the earlier example program in the same situation as shown with the stack in Figure 10.11. Another option for implementing shallow access is to use a central table that has a location for each different variable name in a program. Along with each entry, a bit 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. SNOBOL implementations use the central table implementation technique.



(The names in the stack cells indicate the program units of the variable declaration.)

Fig: 10.12 One method of using shallow access to implement dynamic scoping

Overloaded subprograms

An overloaded operator is one that has multiple meanings. The meaning of a particular instance of an overloaded operator is determined by the types of its operands.
For example, if the `*` operator has two floating-point operands in a Java program, it specifies floating-point multiplication. But if the same operator has two integer operands, it specifies integer multiplication.

An **overloaded subprogram** is a subprogram that has the same name as another subprogram in the same referencing environment. Function overloading is an example of overloaded subprograms

- Every version of an overloaded subprogram must have a unique protocol; that is, it must be different from the others in the number, order, or types of its parameters, and possibly in its return type if it is a function.

example of overloaded functions

```
double pow(double, double)
double pow(int, double)
double pow(float, double)
```

- *C++, Java, C#, and Ada include predefined overloaded subprograms*
- In Ada, the return type of an overloaded function can be used to disambiguate calls (thus two overloaded functions can have the same parameter profile and differ only in their return types) . For example, if a C++ program has two functions named fun and both take an int parameter but one returns an int and one returns a float, the program would not compile, because the compiler could not determine which version of fun should be used.

function f1(p1: **in** Integer) **return** Float;
function f1(p1: **in** Integer) **return** Integer;

- Because Java, C++, and C# allow mixed-mode expressions, the return type is irrelevant to the disambiguation

int fun(**int** p1);
float fun(**int** p1);
Which one should be called in 5.0 + fun(3)?

Generic Subprograms

- A generic or polymorphic subprogram takes parameters of different types on different activations
- Overloaded subprograms provide ad hoc polymorphism (they need not behave similarly)
- parametric polymorphism is provided by a subprogram that takes generic parameters that are used in type expressions that describes the types of the parameters of the subprogram

Generic Functions in C++

Generic functions in C++ have the descriptive name of *template functions*. The definition of a template function has the general form

template <template parameters>
—a function definition that may include the template parameters