UNIT III:

SUBPROGRAMS AND IMPLEMENTATIONS: Subprograms, design issues, local referencing, parameter passing, overloaded methods, generic methods, design issues for functions, semantics of call and return, implementing simple subprograms, stack and dynamic local variables, nested subprograms, blocks, dynamic scoping.

Subprograms: Subprograms are the fundamental building blocks of programs and are therefore among the most important concepts in programming language design. Two fundamental abstraction facilities can be included in a programming language: process abstraction and data abstraction.

General Subprogram Characteristics:00

- Each subprogram has a single entry point.
- The calling program unit is suspended during the execution of the called subprogram, which implies that there is only one subprogram in execution at any given time.
- Control always returns to the caller when the subprogram execution terminates.

A **subprogram definition** describes the interface to and the actions of the subprogram abstraction. A **subprogram call** is the explicit request that a specific subprogram be executed. A subprogram is said to be **active** if, after having been called, it has begun execution but has not yet completed that execution.

A **subprogram header**, which is the first part of the definition, serves several purposes. First, it specifies that the following syntactic unit is a subprogram definition of some particular kind. In languages that have more than one kind of subprogram, the kind of the subprogram is usually specified with a special word. Second, if the subprogram is not anonymous, the header provides a name for the subprogram. Third, it may optionally specify a list of parameters.

Example Python:
def adder parameters):
This is the header of a Python subprogram named adder. Ruby subprogram headers also begin with **def**. The header of a JavaScript subprogram begins with **function**.
In C, the header of a function named **adder** might be as follows:
Void adder (parameters)

The body of subprograms defines its actions. In the C-based languages (and some others—for example, JavaScript) the body of a subprogram is delimited by braces. In Ruby, an **end** statement terminates the body of a subprogram. As with compound statements, the statements in the body of C++ programs, where they are called **prototypes**. Such declarations are often placed in header files. In most other languages (other than C and C++), subprograms do not need declarations, because there is no requirement that subprograms be defined before they are called.

Parameters

Subprograms typically describe computations. There are two ways that a non method subprogram can gain access to the data that it is to process: through direct access to nonlocal variables (declared elsewhere but visible in the subprogram) or through parameter passing. Data passed through parameters are accessed through names that are local to the subprogram. Parameter passing is more flexible than direct access to nonlocal variables. In essence, a subprogram with parameter access to the data that it is to process is a parameterized computation.
It can perform its computation on whatever data it receives through its parameters (presuming the types of the parameters are as expected by the subprogram). If data access is through nonlocal

variables, the only way the computation can proceed on different data is to assign new values to those nonlocal variables between calls to the subprogram.
The parameters in the subprogram header are called **formal parameters**. They are sometimes thought of as dummy variables because they are not variables in the usual sense: In most cases, they are bound to storage only when the subprogram is called, and that binding is often through some other program variables.
Subprogram call statements must include the name of the subprogram and a list of parameters to be bound to the formal parameters of the subprogram. These parameters are called **actual parameters.**

In Python, Ruby, C++, Fortran 95+ Ada, and PHP, formal parameters can have default values. A default value is used if no actual parameter is passed to the formal parameter in the subprogram header. Consider the following Python function header:
Def compute_pay(income, exemptions = 1, tax_rate)

There are two distinct categories of subprograms—procedures and functions both of which can be viewed as approaches to extending the language. All subprograms are collections of statements that define parameterized computations.
Functions return values and procedures do not. In most languages that do not include procedures as a separate form of subprogram, functions can be defined not to return values and they can be used as procedures. The computations of a procedure are enacted by single call statements. In effect, 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; and (2) if the procedure has formal parameters that allow the transfer of data to the caller, those parameters can be changed

Design Issues for Subprograms: Subprograms are complex structures in programming languages, and it follows from this that a lengthy list of issues is involved in their design. The nature of the local environment of a subprogram dictates to some degree the nature of the subprogram. The most important question here is whether local variables are statically or dynamically allocated. Next, there is the question of whether subprogram definitions can be nested. Another issue is whether subprogram names can be passed as parameters.

- Are local variables statically or dynamically allocated?
- Can subprogram definitions appear in other subprogram definitions?
- What parameter-passing method or methods are used?
- Are the types of the actual parameters checked against the types of the formal parameters?
- 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 subprograms be generic?
- If the language allows nested subprograms, are closures supported?

If subprogram names can be passed as parameters and the language allows subprograms to be nested, there is the question of the correct referencing environment of a subprogram that has been passed as a parameter.

Finally, there tare the questions of whether subprograms can be overloaded or generic. 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. A **closure** is a nested subprogram and its referencing environment, which together allow the subprogram to be called from anywhere in a program.

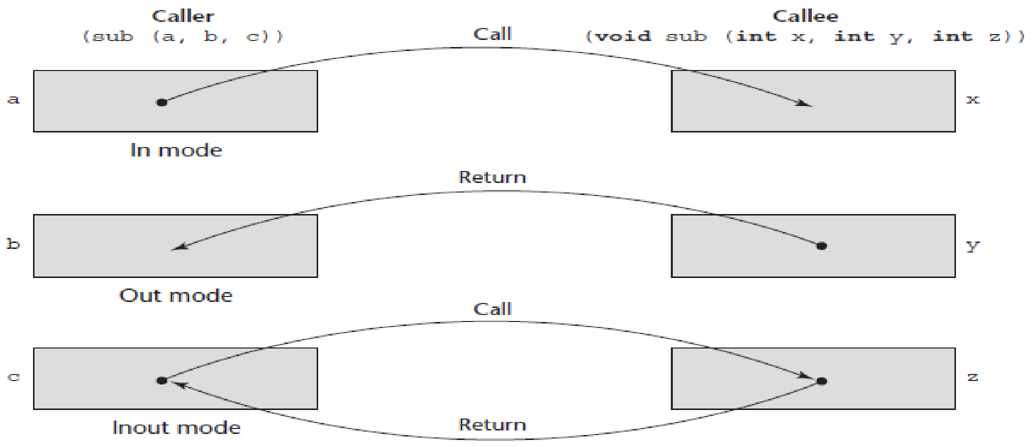
Local Referencing Environments:
Local Variables Subprograms can define their own variables, thereby defining local referencing environments. Variables that are defined inside subprograms are called **local variables**, because their scope is usually the body of the subprogram in which they are defined. 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 terminates. There are several advantages of stack-dynamic local variables, the primary one being the flexibility they provide to the subprogram.
It is essential that recursive subprograms have stack-dynamic local variables when the subprogram begins execution and are unbound from storage when that execution terminates. There are several advantages of stack-dynamic local variables, the primary one being the flexibility they provide to the subprogram. It is essential that recursive subprograms have stack-dynamic local variables. Another advantage of stack-dynamic locals is that the storage for local variables in an active subprogram can be shared with the local variables in all inactive
subprograms. This is not as great an advantage as it was when computers had smaller memories.
The main disadvantages of stack-dynamic local variables are the following:
First, there is the cost of the time required to allocate, initialize (when necessary), and deallocate such variables for each call to the subprogram.
Second, accesses to stack-dynamic local variables must be indirect, whereas accesses to static variables can be direct. This indirectness is required because the place in the stack where a particular local variable will reside can be determined only during execution.
Finally, when all local variables are stack dynamic, subprograms cannot be history sensitive; that is, they cannot retain data values of local variables between calls. It is sometimes convenient to be able to write history-sensitive subprograms.
A common example of a need for a history-sensitive subprogram is one whose task is to generate pseudorandom numbers. Each call to such a subprogram computes one pseudorandom number, using the last one it computed. It must, therefore, store the last one in a static local variable.
The primary advantage of static local variables over stack-dynamic local variables is that they are slightly more efficient—they require no run-time overhead for allocation and deallocation.

Nested Subprograms
The idea of nesting subprograms originated with Algol 60. The motivation was to be able to create a hierarchy of both logic and scopes. If a subprogram is needed only within another subprogram, why not place it there and hide it from the rest of the program? Because static scoping is usually used in languages that allow subprograms to be nested, this also provides a highly structured way to grant access to nonlocal variables in enclosing subprograms

Parameter-Passing Methods
Parameter-passing methods are the ways in which parameters are transmitted to and/or from called subprograms. First, we focus on the different semantics models of parameter-passing methods.

Semantics Models of Parameter Passing
Formal parameters are characterized by one of three distinct semantics models:
(1) They can receive data from the corresponding actual parameter;
(2) they can transmit data to the actual parameter; or
(3) they can do both. These models are called **in mode**, **out mode**, and **in out mode**, respectively
There are two conceptual models of how data transfers take place in parameter transmission: Either an actual value is copied (to the caller, to the called, or both ways), or an access path is transmitted.

Implementation Models of Parameter Passing
A variety of models have been developed by language designers to guide the implementation of the three basic parameter transmission modes. In the following sections, we discuss several of these, along with their relative strengths and weaknesses.



1. Pass-by-Value
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, thus implementing in-mode semantics. Pass-by-value is normally implemented by copy, because accesses often are more efficient with this approach. It could be implemented by transmitting an access path to the value of the actual parameter in the caller, but that would require that the value be in a write-protected cell.
The advantage of pass-by-value is that for scalars it is fast, in both linkage cost and access time. The main disadvantage of the pass-by-value method if copies are used is that additional storage is required for the formal parameter, either in the called subprogram or in some area outside both the caller and the called subprogram.

2. Pass-by-Result
Pass-by-result is an implementation model for out-mode parameters. When a parameter is passed by result, no value is transmitted to the subprogram. The corresponding formal parameter acts as a local variable, but just before control is transferred back to the caller, its value is transmitted back to the caller's actual parameter, which obviously must be a variable.
The pass-by-result method has the advantages and disadvantages of pass by-value, plus some additional disadvantages. If values are returned by copy (as opposed to access paths), as they typically are, pass-by-result also requires the extra storage and the copy operations that are required by pass-by-value .As with pass-by-value, the difficulty of implementing pass-by-result by transmitting an access path usually results in it being implemented by copy. In this case, the problem is in ensuring that the initial value of the actual parameter is not used in the called subprogram.

3. Pass-by-Value-Result

Pass-by-value-result is an implementation model for in out-mode parameters in which actual values are copied. It is in effect a combination of pass-by-value and pass-by-result. The value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local variable. In fact, pass-by-value-result formal parameters must have local storage associated with the called subprogram. At subprogram termination, the value of the formal parameter is transmitted back to the actual parameter. Pass-by-value-result is sometimes called **pass-by-copy**, because the actual parameter is copied to the formal parameter at subprogram entry and then copied back at subprogram termination.

Pass-by-value-result shares with pass-by-value and pass-by-result the disadvantages of requiring multiple storage for parameters and time for copying values. It shares with pass-by-result the problems associated with the order in which actual parameters are assigned. The advantages of pass-by-value-result are relative to pass-by-reference,

4. Pass-by-Reference

Pass-by-reference is a second implementation model for in out-mode parameters. Rather than copying data values back and forth, however, 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. Thus, the called subprogram is allowed to access the actual parameter in the calling program unit. In effect, the actual parameter is shared with the called subprogram.

The advantage of pass-by-reference is that the passing process itself is efficient, in terms of both time and space. Duplicate space is not required, nor is any copying required. There are, however, several disadvantages to the pass-by-reference method. First, access to the formal parameters will be slower than pass-by-value parameters, because of the additional level of indirect addressing that is required.6Second, if only one-way communication to the called subprogram is required, inadvertent and erroneous changes may be made to the actual parameter.

Another problem of pass-by-reference is that aliases can be created. This problem should be expected, because pass-by-reference makes access paths available to the called subprograms, thereby providing access to nonlocal variables.

The problem with these kinds of aliasing is the same as in other circumstances: It is harmful to readability and thus to reliability. It also makes program verification more difficult.

```
void fun(int &first, int &second)
```

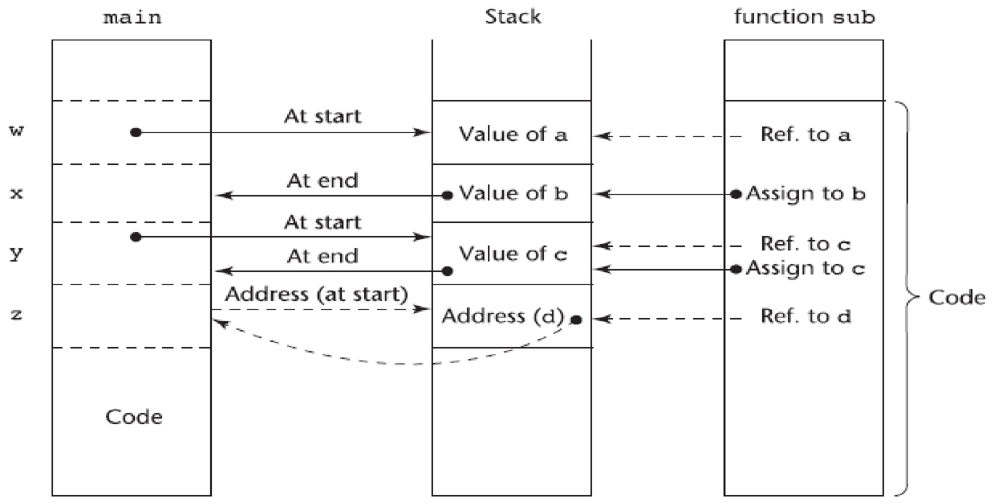
5. **Pass-by-Name**

Pass-by-name is an inout-mode parameter transmission method that does not correspond to a single implementation model. When parameters are passed byname, the actual parameter is, in effect, textually substituted for the corresponding formal parameter in all its occurrences in the subprogram. This method is quite different from those discussed thus far; in which case, formal parameters are bound to actual values or addresses at the time of the subprogram call. A pass-by name formal parameter is bound to an access method at the time of the subprogram call, but the actual binding to a value or an address is delayed until the formal parameter is assigned or referenced. Implementing a pass-by-name parameter requires a subprogram to be passed to the called subprogram to evaluate the address or value of the formal parameter. The referencing environment of the passed subprogram must also be passed.

Implementing Parameter-Passing Methods: In most contemporary languages, parameter communication takes place through the run-time stack. The run-time stack is initialized and maintained by the run-time system, which manages the execution of programs. The run time stack is used extensively for subprogram control linkage and parameter passing.

Pass-by-value parameters have their values copied into stack locations. The stack locations then serve as storage for the corresponding formal parameters. Pass-by-result parameters are

implemented as the opposite of pass-by value. The values assigned to the pass-by-result actual parameters are placed in the stack, where they can be retrieved by the calling program unit upon termination of the called subprogram. Pass-by-value-result parameters can be implemented directly from their semantics as a combination of pass-by-value and pass-by-result. The stack location for such a parameter is initialized by the call and is then used like a local variable in the called subprogram.
Pass-by-reference parameters are perhaps the simplest to implement. Regardless of the type of the actual parameter, only its address must be placed in the stack. In the case of literals, the address of the literal is put in the stack. In the case of an expression, the compiler must build code to evaluate the expression just before the transfer of control to the called subprogram. The address of the memory cell in which the code places the result of its evaluation is then put in the stack. The compiler must be sure to prevent the called subprogram from changing parameters that are literals or expressions.



Function header: **void** sub (**int** a, **int** b, **int** c, **int** d)
Function call in main: sub (w,x,y,z)
(pass w by value, x by result, y by value-result, z by reference)

Parameters That Are Subprograms
In programming, a number of situations occur that are most conveniently handled if subprogram names can be sent as parameters to other subprograms. One common example of these occurs when a subprogram must sample some mathematical function. For example, a subprogram that does numerical integration estimates the area under the graph of a function by sampling the function at a number of different points. When such a subprogram is written, it should be usable for any given function; it should not need to be rewritten for every function that must be integrated. It is therefore natural that the name of a program function that evaluates the mathematical function to be integrated be sent to the integrating subprogram as a parameter. Although the idea is natural and seemingly simple, the details of how it works can be confusing. If only the transmission of the subprogram code was necessary, it could be done by passing a single pointer. However, two complications
arise. First, there is the matter of type checking the parameters of the activations of the subprogram that was passed as a parameter. In C and C++, functions cannot be passed as parameters, but pointers to functions can.

The type of a pointer to a function includes the function's protocol. Because the protocol includes all parameter types, such parameters can be completely type checked Fortran 95+ has a mechanism for providing types of parameters for subprograms that are passed as parameters, and they must be checked. The second complication with parameters that are subprograms appears only with languages that allow nested subprograms. The issue is what referencing environment for executing the passed subprogram should be used.

There are three choices:
• The environment of the call statement that enacts the passed subprogram(**shallow binding**)
• The environment of the definition of the passed subprogram (**deep binding**)
• The environment of the call statement that passed the subprogram as an actual parameter (**ad hoc binding**)

The following example program, written with the syntax of JavaScript, illustrates these choices:

```
functions ub1() {
var x;
functionsub2() {
alert(x); // Creates a dialog box with the value of x
};
functionsub3() {
var x;
x = 3;
sub4(sub2);
};
functionsub4(subx) {
varx;
x = 4;
subx();
};
x = 1;
sub3();
};
```

Consider the execution of sub2 when it is called in sub4. For shallow binding, the referencing environment of that execution is that of sub4, so the reference to x in sub2 is bound to the local x in sub4, and the output of the program is 4. For deep binding, the referencing environment of sub2's execution is that of sub1, so the reference to x in sub2 is bound to the local x insub1, and the output is 1. For ad hoc binding, the binding is to the local x insub3, and the output is 3.
In some cases, the subprogram that declares a subprogram also passes that subprogram as a parameter. In those cases, deep binding and ad hoc binding are the same.
Ad hoc binding has never been used because, one might surmise, the environment in which the procedure appears as a parameter has no natural connection to the passed subprogram.
Shallow binding is not appropriate for static-scoped languages with nested subprograms. For example, suppose the procedure Sender passes the procedure Sent as a parameter to the procedure Receiver. The problem is that Receiver may not be in the static environment of Sent, thereby making it very unnatural for Sent to have access to Receiver's variables.
On the other hand, it is perfectly normal in such a language for any subprogram, including one sent as a parameter, to have its referencing environment determined by the lexical position of its definition. It is therefore more logical for these languages to use deep binding. Some dynamic-scoped languages use shallow binding.

Calling Subprograms Indirectly
There are situations in which subprograms must be called indirectly. These most often occur when the specific subprogram to be called is not known until run time. The call to the subprogram is made through a pointer or reference to the subprogram, which has been set during execution before the

call is made. The two most common applications of indirect subprogram calls are for event handling in graphical user interfaces, which are now part of nearly all Web applications, as well as many non-Web applications, and for call backs, in which a subprogram is called and instructed to notify the caller when the called subprogram has completed its work. As
always, our interest is not in these specific kinds of programming, but rather in programming language support for them.
The concept of calling subprograms indirectly is not a recently developed concept. C and C++ allow a program to define a pointer to a function, through which the function can be called. In C++, pointers to functions are
typed according to the return type and parameter types of the function, so that such a pointer can point only at functions with one particular protocol.
For example, the following declaration defines a pointer (pfun) that can point to any function that takes a **float** and an **int**as parameters and returns a **float**:

float (*pfun)(**float**, **int**);

Any function with the same protocol as this pointer can be used as the initial value of this pointer or be assigned to the pointer in a program. In C and C++,a function name without following parentheses, like an array name without tfollowing brackets, is the address of the function (or array). So, both of the following are legal ways of giving an initial value or assigning a value to a pointer to a function:**int**myfun2 (**int**, **int**); // A function declaration
int(*pfun2)(**int**, **int**) = myfun2; // Create a pointer and
// initialize
// it to point to myfun2
pfun2 = myfun2; // Assigning a function's address to a// pointer
The function myfun2 can now be called with either of the following statements:
(*pfun2)(first, second);
pfun2(first, second);
The first of these explicitly dereferences the pointer pfun2, which is legal, but unnecessary.
The function pointers of C and C++ can be sent as parameters and returned from functions, although functions cannot be used directly in either of those roles.
In C#, the power and flexibility of method pointers is increased by making them objects. These are called **delegates**, because instead of calling a method, a program delegates that action to a delegate. To use a delegate, first the delegate class must be defined with a specific method protocol. An instantiation of a delegate holds the name of a method with the delegate's protocol that it is able to call. The syntax of a declaration of a delegate is the same as that of a method declaration, except that the reserved word **delegate** is inserted just before the return type.
For example, we could have the following:

public delegate int Change(**int** x);

This delegate can be instantiated with any method that takes an **int** as a parameter and returns an **int**. For example, consider the following method declaration:

staticintfun1(**int** x);

The delegate Change can be instantiated by sending the name of this method to the delegate's constructor, as in the following:

Change chgfun1 = **new** Change(fun1);

This can be shortened to the following:

Change chgfun1 = fun1;

Following is an example call to fun1 through the delegate chgfun1:

chgfun1(12);

Objects of a delegate class can store more than one method.
A second method can be added using the operator +=, as in the following:

Change chgfun1 += fun2;

This places fun2 in the chgfun1 delegate, even if it previously had the value **null**. All of the methods stored in a delegate instance are called in the order in which they were placed in the instance. This is called a **multicast delegate**. Regard less of what is returned by the methods, only the value or object returned by the last one called is returned. Of course, this means that in most cases, **void** is returned by the methods called through a multicast delegate.
In our example, a static method is placed in the delegate Change. Instance methods can also be called through a delegate, in which case the delegate must store a reference to the method. Delegates can also be generic.
Delegates are used for event handling by .NET applications. They are also used to implement closures
As is the case with C and C++, the name of a function in Python without the following parentheses is a pointer to that function. Ada 95 has pointers to subprograms, but Java does not. In Python and Ruby, as well as most functional languages, subprograms are treated like data, so they can be assigned to variables. Therefore, in these languages, there is little need for pointers to subprograms.

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. 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. The meaning of a call to an overloaded subprogram is determined by the actual parameter list (and/or possibly the type of the returned value, in the case of a function). Although it is not necessary,
overloaded subprograms usually implement the same process.
C++, Java, Ada, and C# include predefined overloaded subprograms. For example, many classes in C++, Java, and C# have overloaded constructors. Because each version of an overloaded subprogram has a unique parameter profile, the compiler can disambiguate occurrences of calls to them by the different type parameters. Unfortunately, it is not that simple. Parameter coercions, when allowed, complicate the disambiguation process enormously. Simply stated, the issue is that if no method's parameter profile matches the number and types ofthe actual parameters in a method call, but two or more methods have parameter profiles that can be matched through coercions, which method should be called? For a language designer to answer this question, he or she must decide how to rank all of the different coercions, so that the compiler can choose the method that "best" matches the call. This can be a complicated task. To understand the level of complexity of this process, we suggest the reader refer to the rules for disambiguation of method calls used in C++ (Stroustrup, 1997).
Because C++, Java, and C# allow mixed-mode expressions, the return type is irrelevant to disambiguation of overloaded functions (or methods). The context of the call does not allow the determination of the return type. 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.
Users are also allowed to write multiple versions of subprograms with the same name in Ada, Java, C++, C#, and F#. Once again, in C++, Java, and C# the most common user-defined overloaded methods are constructors. Overloaded subprograms that have default parameters can lead to ambiguous subprogram calls. For example, consider the following C++ code:

```
void fun(float b = 0.0);
void fun();
. . .
fun();     The call is ambiguous and will cause a compilation error.
```

Generic Subprograms

Software reuse can be an important contributor to software productivity. One way to increase the reusability of software is to reduce the need to create different subprograms that implement the same algorithm on different types of data. For example, a programmer should not need to write four different sort subprograms to sort four arrays that differ only in element type.

A **polymorphic** subprogram takes parameters of different types on different activations. Overloaded subprograms provide a particular kind of polymorphism called **ad hoc polymorphism**. Overloaded subprograms need not be have similarly.

Languages that support object-oriented programming usually support subtype polymorphism. **Subtype polymorphism** means that a variable of type T can access any object of type T or any type derived from T.

A more general kind of polymorphism is provided by the methods of Python and Ruby. Recall that variables in these languages do not have types, so formal parameters do not have types. Therefore, a method will work for any type of actual parameter, as long as the operators used on the formal parameters in the method are defined.

Parametric polymorphism is provided by a subprogram that takes generic parameters that are used in type expressions that describe the types of the parameters of the subprogram. Different instantiations of such subprograms can be given different generic parameters, producing subprograms that take different types of parameters. Parametric definitions of subprograms all behave the same. Parametrically polymorphic subprograms are often called **generic** subprograms. Ada, C++, Java 5.0+, C# 2005+, and F# provide a kind
of compile-time parametric polymorphism.

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 A template parameter (there must be at least one) has one of the forms

class identifier
typename identifier

The class form is used for type names. The typename form is used for passing a value to the template function. For example, it is sometimes convenient to pass an integer value for the size of an array in the template function.

A template can take another template, in practice often a template class that defines a user-defined generic type, as a parameter, but we do not consider that option here.

As an example of a template function, consider the following:

```
template<class Type>
Type max(Type first, Type second) {
Return first > second ? first : second;
}
```

where Type is the parameter that specifies the type of data on which the function will operate. This template function can be instantiated for any type for which the operator > is defined. For example, if it were instantiated with **int** as the parameter, it would be

```
int max(int first, int second) {
return first > second ? first : second;
}
```

Although this process could be defined as a macro, a macro would have the disadvantage of not operating correctly if the parameters were expressions with side effects. For example, suppose the macro were defined as

```
#define max(a, b) ((a) > (b)) ? (a) : (b)
```

This definition is generic in the sense that it works for any numeric type. However, it does not always work correctly if called with a parameter that has a side effect, such as

```
max(x++, y)
which produces
((x++) > (y) ? (x++) : (y))
```

Whenever the value of x is greater than that of y, x will be incremented twice. C++ template functions are instantiated implicitly either when the function is named in a call or when its address is taken with the & operator. For example, the example template function defined would be instantiated twice by the following code segment—once for **int** type parameters and once for **char** type parameters:

```
int a, b, c;
chard, e, f;
. . .
c = max(a, b);
f = max(d, e);
```

The following is a C++ generic sort subprogram:

```
template<class Type>
void generic_sort(Type list[], int len) {
int top, bottom;
Type temp;
for(top = 0; top <len - 2; top++)
for(bottom = top + 1; bottom <len - 1; bottom++)
if(list[top] > list[bottom]) {
temp = list[top];
list[top] = list[bottom];
list[bottom] = temp;
} //** end of if (list[top] . . .
} //** end of generic_sort
```

The following is an example instantiation of this template function:

```
floatflt_list[100];
. . .
generic_sort(flt_list, 100);
```

The templated functions of C++ are a kind of poor cousin to a subprogram in which the types of the formal parameters are dynamically bound to the types of the actual parameters in a call. In this case, only a single copy of the code is needed, whereas with the C++ approach, a copy must be created at compile time for each different type that is required and the binding of subprogram calls to subprograms is static.

Support for generic types and methods was added to Java in Java 5.0. The name of a generic class in Java 5.0 is specified by a name followed by one or more type variables delimited by pointed brackets. For example,

```
generic_class<T>
```

Java's generic methods differ from the generic subprograms of C++ in several important ways. First, generic parameters must be classes—they cannot be primitive types. This requirement disallows a generic method that
mimics our example in C++, in which the component types of arrays are generic and can be primitives. In Java, the components of arrays (as opposed to containers) cannot be generic. Second, although Java generic methods can be instantiated any number of times, only one copy of the code is built. The internal version of a generic method, which is called a *raw* method, operates on Object class objects. At the point where the generic value of a generic method is returned, the compiler inserts a cast to the proper type. Third, in Java, restrictions can be specified on the range of classes that can be passed to the generic method as generic parameters. Such restrictions are called **bounds**.

```
public static <T> T dIt(T[] list) {
. . .
}
```

This defines a method named doIt that takes an array of elements of a generic type. The name of the generic type is T and it must be an array. Following is an example call to doIt:

```
doIt<String>(myList);
```

Now, consider the following version of doIt, which has a bound on its generic parameter:

```
public static <T extends Comparable> T doIt(T[] list) {
. . .
}
```

Generic Methods in C# 2005

The generic methods of C# 2005 are similar in capability to those of Java 5.0,except there is no support for wildcard types. One unique feature of C# 2005generic methods is that the actual type parameters in a call can be omitted if the compiler can infer the unspecified type. For example, consider the following skeletal class definition:

```
Class MyClass {
public static T DoIt<T>(T p1) {
. . .
}
}
```

The method DoIt can be called without specifying the generic parameter if the compiler can conclude the generic type from the actual parameter in the call.
For example, both of the following calls are legal:

```
int myInt = MyClass.DoIt(17); // Calls DoIt<int>
stringmyStr = MyClass.DoIt('apples');
// Calls DoIt<string>
```

Design Issues for Functions

The following design issues are specific to functions:

- Are side effects allowed?
- What types of values can be returned?
- How many values can be returned?

Functional Side Effects In fact, some languages require this; for example, Ada functions canhave only in-mode formal parameters. This requirement effectively prevents a function from causing side effects through its parameters or through aliasing of parameters and globalsIn most other imperative languages, however, functions that cause side effects and aliasing.
Pure functional languages, such as Haskell, do not have variables, so their functions cannot have side effects.

Types of Returned Values

Most imperative programming languages restrict the types that can be returned by their functions. C allows any type to be returned by its functions except arrays and functions. Both of these can be handled by pointer type return values. C++ is like C but also allows user-defined types, or classes, to be returned from its functions.
Ada, Python, Ruby, and Lua are the only languages among current imperative languages whose functions (and/or methods) can return values of any type. In the case of Ada, however, because functions are not types in Ada, they cannot be returned from functions. Of course, pointers to functions can be returned by functions.
In some programming languages, subprograms are first-class objects, which means that they can be passed as parameters, returned from functions, and assigned to variables. Methods are first-class objects in some imperative languages, for example, Python, Ruby, and Lua. The same is true for the functions in most functional languages.
Neither Java nor C# can have functions, although their methods are similar to functions. In both, any type or class can be returned by methods. Because methods are not types, they cannot be returned.

Number of Returned Values

In most languages, only a single value can be returned from a function. However, that is not always the case. Ruby allows the return of more than one value from a method. If a **return** statement in a Ruby method is not followed by an expression, **nil** is returned. If followed by one expression, the value of the expression is returned. If followed by more than one expression, an array of the values of all of the expressions is returned.
Lua also allows functions to return multiple values. Such values follow the **return** statement as a comma-separated list, as in the following:
return 3, sum, index
The form of the statement that calls the function determines the number of values that are received by the caller. If the function is called as a procedure, that is, as a statement, all return values are ignored. If the function returned three values and all are to be kept by the caller, the function would be called as in the following example:

```
a, b, c = fun()
```

In F#, multiple values can be returned by placing them in a tuple and having the tuple be the last expression in the function.

The General Semantics of Calls and Returns:

The subprogram call and return operations are together called **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. The execution status is everything needed to resume execution of the calling program unit. This includes register values, CPU status bits, and the environment pointer (EP).
The EP, which is further discussed in Section 10.3, is used to access parameters and local variables during the execution of a subprogram. The calling process also must arrange to transfer control to the code of the subprogram and ensure that control can return to the proper place when the subprogram execution is completed.
Finally, if the language supports nested subprograms, the call process must create some mechanism to provide access to nonlocal variables that are visible to the called subprogram. The required actions of a subprogram return are less complicated than those of a call. If the subprogram has parameters that are out mode or inout mode and are implemented by copy, the first action of the return process is to move the local values of the associated formal parameters to the actual parameters.
Next, it must deallocate the storage used for local variables and restore the execution status of the calling program unit. Finally, control must be returned to the calling program unit.

Implementing "Simple" Subprograms
We begin with the task of implementing simple subprograms. By "simple" we mean that subprograms cannot be nested and all local variables are static. Early versions of Fortran were examples of languages that had this kind of subprograms.

The semantics of a call to a "simple" subprogram requires the following actions:
1. Save the execution status of the current program unit.
2. Compute and pass the parameters.
3. Pass the return address to the called.
4. Transfer control to the called.

The semantics of a return from a simple subprogram requires the following actions:
1. 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.
2. If the subprogram is a function, the functional value is moved to a place accessible to the caller.
3. The execution status of the caller is restored.
4. Control is transferred 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

These, along with the local variables and the subprogram code, form the complete collection of information a subprogram needs to execute and then return control to the caller.

The question now is the distribution of the call and return actions to the caller and the called. For simple subprograms, the answer is obvious for most of the parts of the process. The last three actions of a call clearly must be done by the caller. Saving the execution status of the caller could be done by either.

In the case of the return, the first, third, and fourth actions must be done by the called. Once again, the restoration of the execution status of the caller could be done by either the caller or the called. In general, the linkage actions of the called can occur at two different times, either at the beginning of its execution or at the end. These are sometimes called the *prologue* and *epilogue* of the subprogram linkage. In the case of a simple subprogram, all of the linkage actions of the callee occur at the end of its execution, so there is no need for a prologue

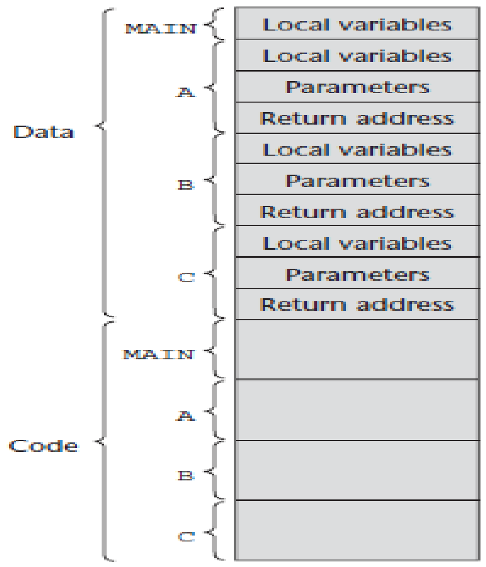
A simple subprogram consists of two separate parts: the actual code of the subprogram, which is constant, and the local variables and data listed previously, which can change when the subprogram is executed. In the case of simple subprograms, both of these parts have fixed sizes. The format, or layout, of the noncode part of a subprogram is called an **activation record**, because the data it describes are relevant only during the activation, or execution of the subprogram. The form of an activation record is static. An **activation record instance** is a concrete example of an activation record, a collection of data in the form of an activation record.

Because languages with simple subprograms do not support recursion, there can be only one active version of a given subprogram at a time. Therefore, there can be only a single instance of the activation record for a subprogram. One possible layout for activation records is shown in Figure 10.1. The saved execution status of the caller is omitted here and in the remainder of this chapter because it is simple and not relevant to the discussion.

Local variables
Parameters
Return address

Because an activation record instance for a “simple” subprogram has fixed size, it can be statically allocated. In fact, it could be attached to the code part of the subprogram.
Figure 10.2 shows a program consisting of a main program and three subprograms: A, B, and C. Although the figure shows all the code segments separated from all the activation record instances, in some cases, the activation record instances are attached to their associated code segments. The construction of the complete program shown in Figure 10.2 is not done entirely by the compiler. In fact, if the language allows independent compilation, the four program units—MAIN, A, B, and C—may have been compiled on different days, or even in different years.
At the time each unit is compiled, the machine code for it, along with a list of references to external subprograms, is written to a file. The executable program shown in Figure 10.2 is put together by the **linker**, which is part of the operating system. (Sometimes linkers are called *loaders, linker/ loaders*, or *link editors*.) 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. In the previous example, the linker was called for MAIN. The linker had to find the machine code programs for A, B, and C, along with their activation record instances, and load them into memory with the code for MAIN.
Then, it had to patch in the target addresses for all calls to A, B, C, and any library subprograms in A, B, C, and MAIN.

-



Implementing Subprograms with Stack-DynamicLocal Variables:
We now examine the implementation of the subprogram linkage in languages in which locals are stack dynamic, again focusing on the call and return operations.

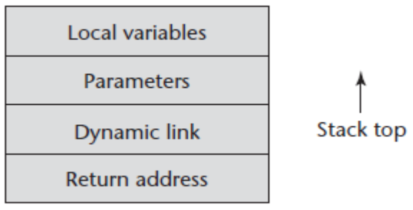
One of the most important advantages of stack-dynamic local variables is support for recursion. Therefore, languages that use stack-dynamic local variables also support recursion.

More Complex Activation Records
Subprogram linkage in languages that use stack-dynamic local variables are more complex than the linkage of simple subprograms for the following reasons:

- The compiler must generate code to cause the implicit allocation and deallocationof local variables.
- Recursion adds the possibility of multiple simultaneous activations of a subprogram, which means that there can be more than one instance (incomplete execution) of a subprogram at a given time, with at least one call from outside the subprogram and one or more recursive calls. The number of activations is limited only by the memory size of the machine. Each activation requires its activation record instance.

The format of an activation record for a given subprogram in most languages is known at compile time. In many cases, the size is also known for activation records because all local data are of a fixed size. That is not the case in some other languages, such as Ada, in which the size of a local array can depend on the value of an actual parameter. In those cases, the format is static, but the size can be dynamic. In languages with stack-dynamic local variables, activation record instances must be created dynamically. The typical activation record for such a language is shown in Figure 10.3. Because the return address, dynamic link, and parameters are placed in the activation record instance by the caller, these entries must appear first.

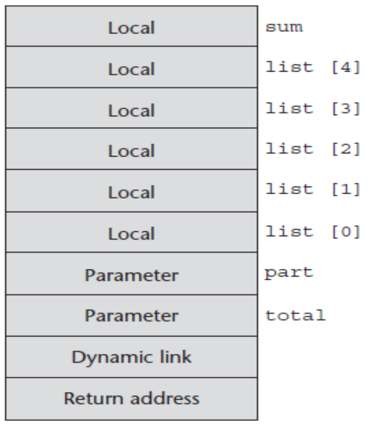
-



The return address usually consists of a pointer to the instruction following the call in the code segment of the calling program unit. The **dynamic link** is a pointer to the base of the activation record instance of the caller. In static scoped languages, this link is used to provide trace back 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.

Consider the following skeletal C function:
voidsub(**float** total, **int**part) {**int**list[5];**float** sum;. . .}



The activation record for sub is shown in Figure 10.4.
Activating a subprogram requires the dynamic creation of an instance of the activation record for the subprogram. As stated earlier, the format of the activation record is fixed at compile time, although its size may depend on the call in some languages. Because the call and return semantics specify that the subprogram last called is the first to complete, it is reasonable to create instances of these activation records on a stack. This stack is part of the runtime system and therefore is called the **run-time stack**, although we will usually just refer to it as the stack. Every subprogram activation, whether recursive or non recursive, creates a new instance of an activation record on the stack.

This provides the required separate copies of the parameters, local variables, and return address.
One more thing is required to control the execution of a subprogram— the EP. Initially, the EP points at the base, or first address of the activation record instance of the main program. Therefore, the run-time system must ensure that it always points at 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.
Resetting the stack top effectively removes the top activation record instance. The EP is used as the base of the offset addressing of the data contents of the activation record instance—parameters and local variables.
Note that the EP currently being used is not stored in the run-time stack.
Only saved versions are stored in the activation record instances as the dynamic links.
We have now discussed several new actions in the linkage process.
The lists given in Section 10.2 must be revised to take these into account. Using the activation record form given in this section, the new actions are as follows:

The caller actions are as follows:
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 called.
5. Transfer control to the called.

The prologue actions of the called are as follows:
1. Save the old EP in the stack as the dynamic link and create the new value.
2. Allocate local variables.

The epilogue actions of the called are as follows:
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.

Parameters are not always transferred in the stack. In many compilers for RISC machines, parameters are passed in registers. This is because RISC machines normally have many more registers than CISC machines. In the remainder of this chapter, however, we assume that parameters are passed inthe stack. It is straightforward to modify this approach for parameters being passed in registers.

An Example Without Recursion : Consider the following skeletal C program:

```
voidfun1(float r) {
ints, t;
. . . 1
fun2(s);
. . .
}
voidfun2(intx) {
inty;
. . . 2
fun3(y);
. . .
}
voidfun3(intq) {
. . . 3
}
voidmain() {
floatp;
. . .
fun1(p);
. . .
}
```

The sequence of function calls in this program is
main calls fun1
fun1 calls fun2
fun2 calls fun3

functionsfun2and fun1 terminate. After the return from the call to fun1 from main, the stack has only the instance of the activation record of main.
Note that some implementations do not actually use an activation record instance on the stack for main functions, such as the one shown in the figure.

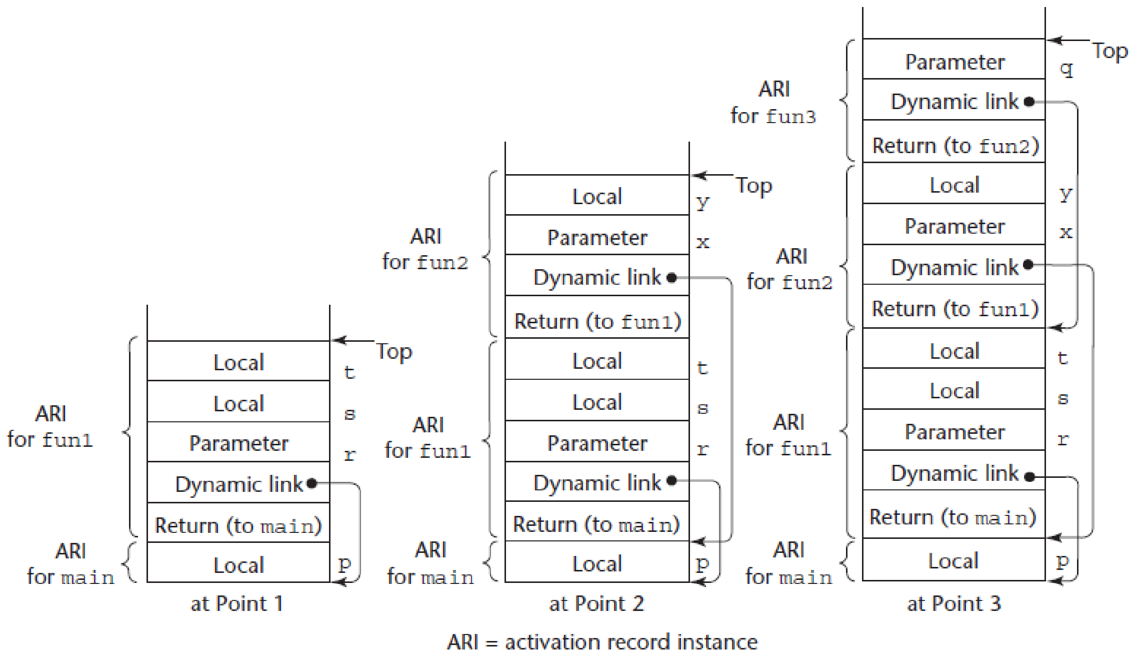
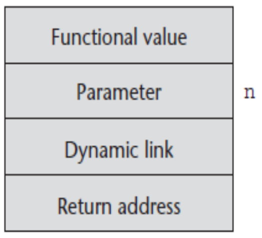
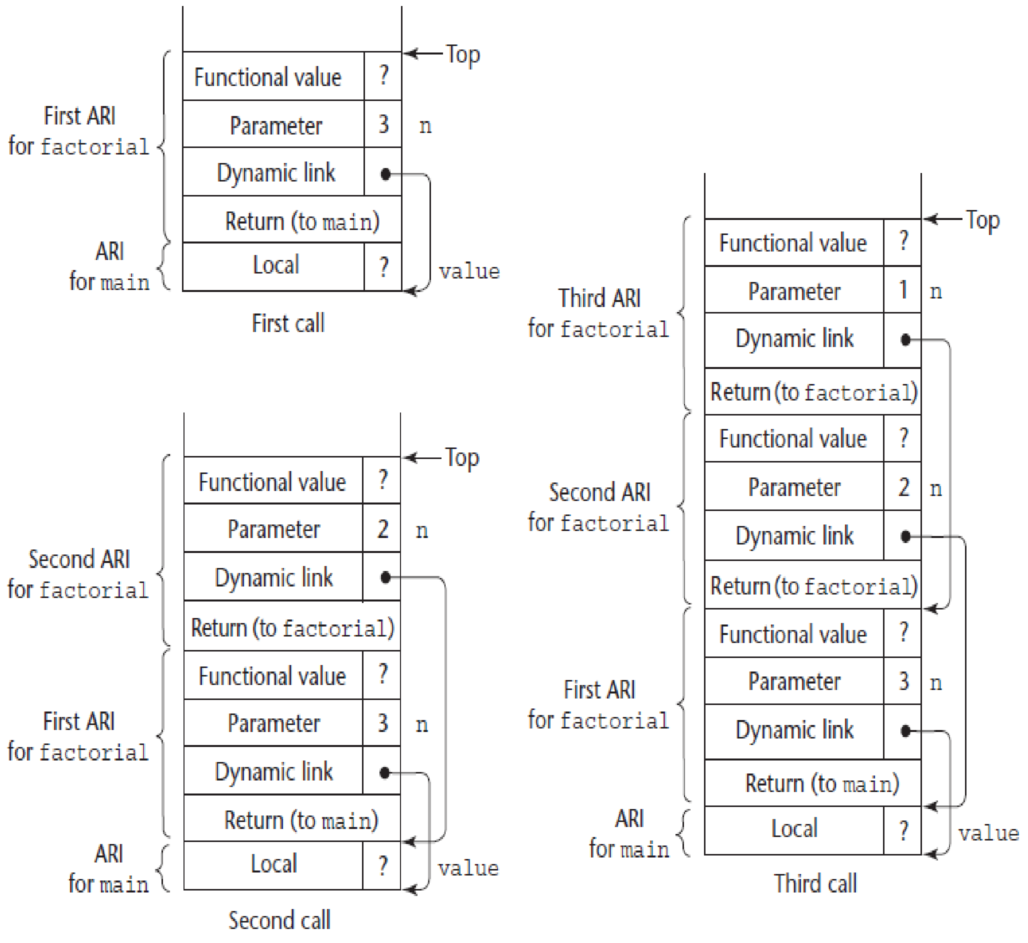


Figure 10.5

However, it can be done this way, and it simplifies both the implementation and our discussion. In this example and in all others in this chapter, we assume that the stack grows from lower addresses to higher addresses,
although in a particular implementation, the stack may grow in the opposite direction.
The collection of dynamic links present in the stack at a given time is called the **dynamic chain**, or **call chain**. It represents the dynamic history of how execution got to its current position, which is always in the subprogram
code whose activation record instance is on top of the stack. 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. To simplify the discussion,

Recursion : Consider the following example C program, which uses recursion to compute
the factorial function:
int factorial(**int** n) {
1
if(n <= 1)
return1;
else return (n * factorial(n - 1));
2
}
void main() {
int value;
value = factorial(3); 3}
The activation record format for the function factorial is shown in Figure 10.6. Notice that it has an additional entry for the return value of the function. Figure 10.7 shows the contents of the stack for the three times execution reaches position 1 in the function factorial. Each shows one more activation of the function, with its functional value undefined. The first activation record instance has the return address to the calling function, main.





ARI = activation record instance

Figure 10.7

The others have a return address to the function itself; these are for the recursive calls.
Figure 10.8 shows the stack contents for the three times that execution reaches position 2 in the function factorial. Position 2 is meant to be the time after the **return** is executed but before the activation record has been removed from the stack. Recall that the code for the function multiplies the current value of the parameter n by the value returned by the recursive call to the function. The first return from factorial returns the value 1.The activation record instance for that activation has a value of 1 for its version of the parameter n. The result from that multiplication, 1, is returned to the second activation of factorial to be multiplied by its parameter
value for n, which is 2. This step returns the value 2 to the first activation of factorial to be multiplied by its parameter value for n, which is 3,yielding the final functional value of 6, which is then returned to the first call to factorial in main.

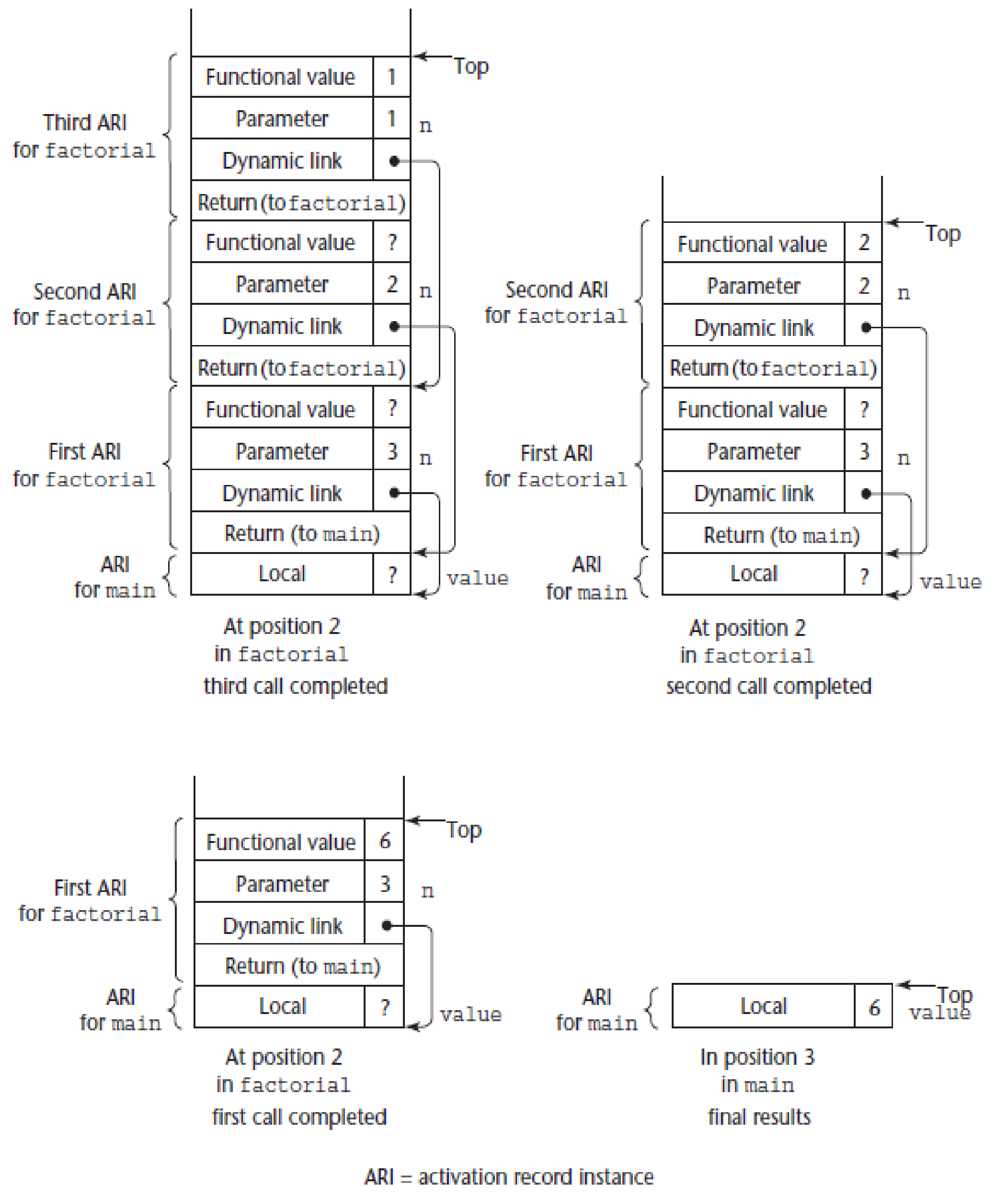


Figure 10.8

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, and Lua, as well as the functional languages.

The Basics

A reference to a nonlocal variable in a static-scoped language with nested subprograms requires a two-step access process. All non static variables that can be non locally accessed are in existing activation record instances and therefore are somewhere in the stack. The first step of the access process is to find the
instance of the activation record in the stack in which the variable was allocated. The second part is to use the local_offset of the variable (within the activation record instance) to access it.

Finding the correct activation record instance is the more interesting and more difficult of the two steps. First, note that in a given subprogram, only variables that are declared in static ancestor scopes are visible and can be accessed. Also, activation record instances of all of the static ancestors are always on the stack when variables in them are referenced by a nested subprogram.
This is guaranteed by the static semantic rules of the static-scoped languages: A subprogram is callable only when all of its static ancestor subprograms are active.1 If a particular static ancestor were not active, its local variables would not be bound to storage, so it would be nonsense to allow access to them. The semantics of nonlocal references dictates that the correct declaration is the first one found when looking through the enclosing scopes, most closely nested first. So, to support nonlocal references, it must be possible to find all of the instances of activation records in the stack that correspond to those static ancestors. This observation leads to the implementation approach described in the following subsection.
We do not address the issue of blocks until Section 10.5, so in the remainder of this section, all scopes are assumed to be defined by subprograms. Because functions cannot be nested in the C-based languages (the only static scopes in those languages are those created with blocks), the discussions of this section do not apply to those languages directly.

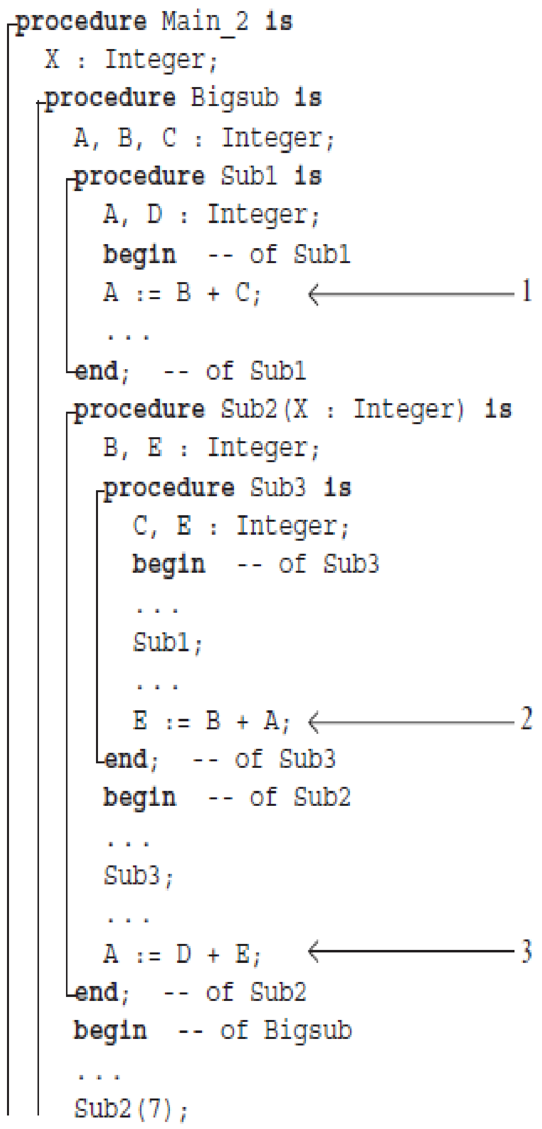
Static Chains

The most common way to implement static scoping in languages that allow nested subprograms is static chaining. In this approach, a new pointer, called a static link, is added to the activation record. The **static link**, which is sometimes called a *static scope pointer*, points to the bottom of the activation record instance of an activation of the static parent. It is used for accesses to nonlocal variables. Typically, the static link appears in the activation record below the parameters. The addition of the static link to the activation record requires that local offsets differ from when the static link is not included. Instead of having two activation record elements before the parameters, there are now three: the return address, the static link, and the dynamic link.
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. So, the static chain connects all the static ancestors of an executing subprogram, in order of static parent first. This chain can obviously be used to implement the accesses to nonlocal variables in static-scoped languages.
Finding the correct activation record instance of a nonlocal variable using static links is relatively straightforward. 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. However, it can be much easier than that.
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.
Let **static_depth**be an integer associated with a static scope that indicates how deeply it is nested in the outermost scope. A program unit that is not nested inside any other unit has a static_depth of 0. If subprogram A is defined in a non nested program unit, its static_depth is 1. If subprogram A contains the definition of a nested subprogram B, then B's static_depth is 2.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 (local_offset is described in Section 10.3.2). For example, consider the following skeletal Python program:

```
            # Global scope
```

```
. . .
deff1():
deff2():
deff3():. . .
# end of f3
. . .
# end of f2
. . .
# end of f1
```

The static_depths of the global scope, f1, f2, and f3 are 0, 1, 2, and 3, respectively.
If procedure f3 references a variable declared in f1, the chain_offset of that reference would be 2 (static_depth of f3 minus the static_depth of f1). If procedure f3 references a variable declared in f2, the chain_offset of that reference would be 1. References to locals can be handled using the same mechanism, with a chain_offset of 0, but instead of using the static pointer to the activation record instance of the subprogram where the variable was declared as the base address, the EP is used. To illustrate the complete process of nonlocal accesses, consider the following skeletal 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  -- of Main_2
  ...
  Bigsub;
  ...
end;  -- of Main_2
```

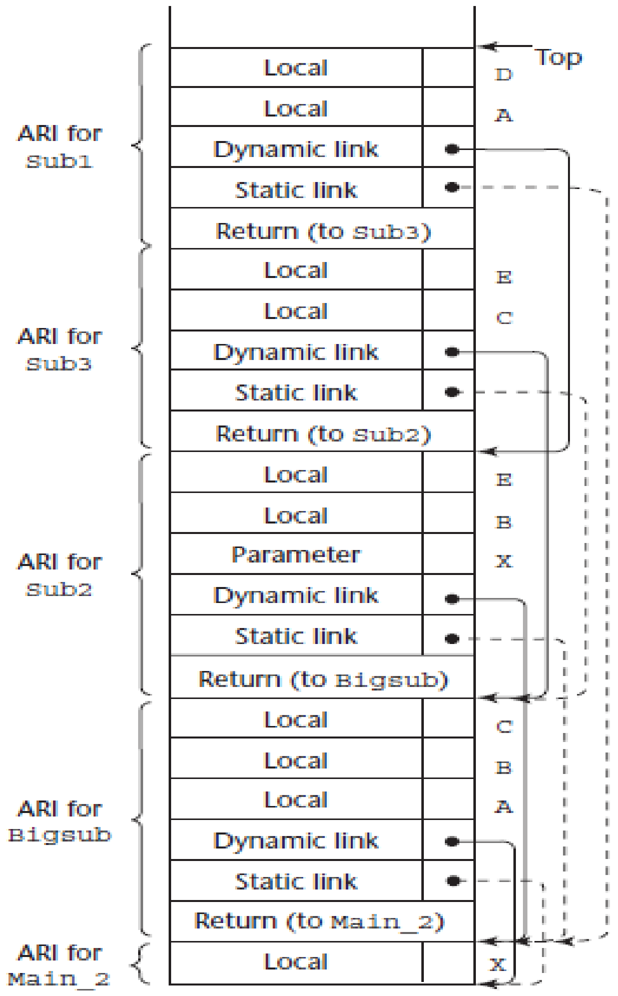
The sequence of procedure calls is

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

The stack situation when execution first arrives at point 1 in this program is shown in Figure 10.9. At position 1 in procedure Sub1, the reference is to the local variable, A, not to the nonlocal variable A from Bigsub. This reference to A has the chain_offset/local_offset pair (0, 3). The reference to B is to the nonlocal B from Bigsub. It can be represented by the pair (1, 4). The local_offset is 4, because a 3 offset would be the first local variable (Bigsubhas no parameters). Notice that if the dynamic link were used to do a simple search for an activation record instance with a declaration for the variable B, it would find the variable B declared in Sub2, which would be incorrect. If the (1, 4) pair were used with the dynamic chain, the variable E from Sub3 would be used.
The static link, however, points to the activation record for Bigsub, which has the correct version of B. The variable B in Sub2 is not in the referencing environment at this point and is (correctly) not accessible. The reference to C at point 1 is to the C defined in Bigsub, which is represented by the pair (1, 5).
After Sub1 completes its execution, the activation record instance for Sub1 is removed from the stack, and control returns to Sub3. The reference to the variable E at position 2 in Sub3 is local and uses the pair (0, 4) for access. The reference to the variable B is to the one declared in Sub2, because that is the nearest static ancestor that contains such a declaration.
It is accessed with the pair (1, 4). The local_offset is 4 because B is the first variable declared in Sub1, and Sub2 has one parameter. The reference to the variable A is to the A declared in Bigsub, because neither Sub3 nor its static parent Sub2 has a declaration for a variable named A. It is referenced with the pair (2, 3).

After Sub3 completes its execution, the activation record instance for Sub3 is removed from the stack, leaving only the activation record instances for Main_2, Bigsub, and Sub2. At position 3 in Sub2, the reference to the variable A is to the A in Bigsub, which has the only declaration of A among the active routines. This access is made with the pair (1, 3). At this position, there is no visible scope containing a declaration for the variable D, so this reference to D is a static semantics error. The error would be detected when the compiler attempted to compute the chain_offset/local_offset pair. The reference to E is to the local E in Sub2, which can be accessed with the pair (0, 5).
In summary, the references to the variable A at points 1, 2, and 3 would be represented by the following points:
• (0, 3) (local)
• (2, 3) (two levels away)

• (1, 3) (one level away)



ARI = activation record instance

It is reasonable at this point to ask how the static chain is maintained during program execution. If its maintenance is too complex, the fact that it is simple and effective will be unimportant. We assume here that parameters that are subprograms are not implemented.

The static chain must be modified for each subprogram call and return. The return part is trivial: 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. Therefore, no other action is required.

The action required at a subprogram call is more complex. Although the correct parent scope is easily determined at compile time, the most recent activation record instance of the parent scope must be found at the time of the call. This can be done by looking at activation record instances on the dynamic chain until the first one of the parent scope is found. However, 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, which must be a static ancestor of the calling routine. It then computes the nesting_depth, or number of enclosing scopes between the caller and the subprogram that declared the called subprogram.

This information is stored and can be accessed by the subprogram call during execution. At the time of the call, the static link of the called subprogram's activation record instance is determined by moving down the static
chain of the caller the number of links equal to the nesting_depth computed at compile time.
Consider again the program Main_2 and the stack situation shown in Figure 10.9. At the call to Sub1 in Sub3, the compiler determines the nesting_depth of Sub3 (the caller) to be two levels inside the procedure that
Declared the called procedure Sub1, which is Bigsub. When the call to Sub1 in Sub3 is executed, this information is used to set the static link of the activation record instance for Sub1. This static link is set to point to the activation record instance that is pointed to by the second static link in the static chain from the caller's activation record instance. In this case, the caller is Sub3, whose static link points to its parent's activation record instance (that of Sub2). The static link of the activation record instance for Sub2 points
to the activation record instance for Bigsub. So, the static link for the new activation record instance for Sub1 is set to point to the activation record instance for Bigsub.
This method works for all subprogram linkage, except when parameters that are subprograms are involved.
One criticism of using the static chain to 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, so this is not a serious problem. Another criticism of the static-chain approach is that 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. Further complicating this problem is that subsequent 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.
Some alternatives to static chains have been developed, most notably an approach that uses an auxiliary data structure called a display. However, none of the alternatives has been found to be superior to the static-chain method, which is still the most widely used approach. Therefore, none of the alternatives is discussed here.
The processes and data structures described in this section correctly implement closures in languages that do not permit functions to return functions and do not allow functions to be assigned to variables. However, they are inadequate for languages that do allow one or both of those operations. Several new mechanisms are needed to implement access to non locals in such languages.

- First, if a subprogram accesses a variable from a nesting but not global scope, that variable cannot be stored only in the activation record of its home scope. That activation record could be deallocated before the subprogram that needs it is activated. Such variables could also be stored in the heap and given unlimited extend (their lifetimes are the lifetime of the whole program).
- Second, subprograms must have mechanisms to access the nonlocals that are stored in the heap.
- Third, the heap-allocated variables that are non locally accessed must be updated every time their stack versions are updated. Clearly, these are nontrivial extensions to the implementation static scoping using static chains

Blocks
Recall from Chapter 5, that a number of languages, including the C-based languages, provide for user-specified local scopes for variables called **blocks**. As an example of a block, consider the following code segment:

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

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 preceding block begins when control enters the block and ends when control exits the block. The advantage of using such a local is that it cannot interfere with any other variable with the same name that is declared else where in the program, or more specifically, in the referencing environment of the block.

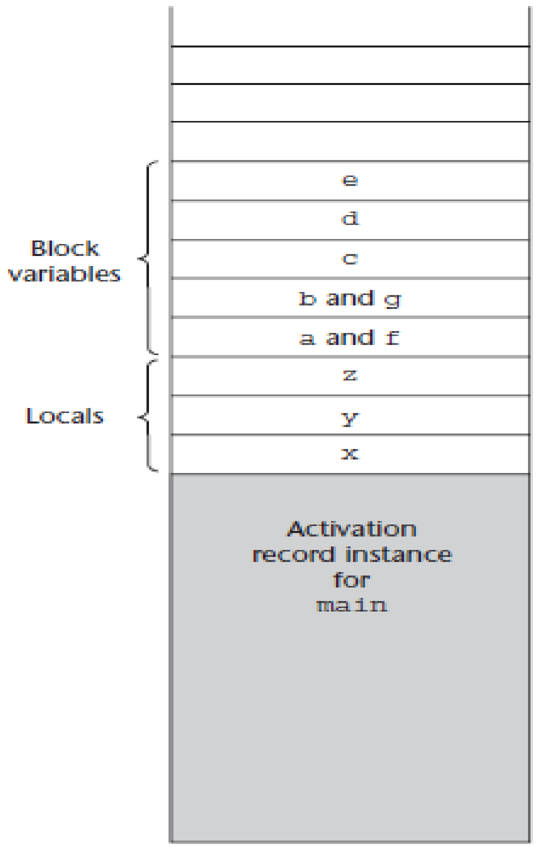
Blocks can be implemented by using the static-chain process for implementing nested subprograms. Blocks are treated as parameter less subprograms that are always called from the same place in the program. Therefore, every block has an activation record. An instance of its activation record is created every time the block is executed.

Blocks can also be implemented in a different and somewhat simpler and more efficient way. The maximum amount of storage required for block variable sat any time during the execution of a program can be statically determined, because blocks are entered and exited in strictly textual order. This amount of space can be allocated after the local variables in the activation record. Offsets for all block variables can be statically computed, so block variables can be addressed exactly as if they were local variables.

For example, consider the following skeletal program:

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

For this program, the static-memory layout shown in Figure 10.10 could be used. Note that f and g occupy the same memory locations as a and b, because a and b are popped off the stack when their block is exited Before f and g are allocated).



Implementing Dynamic Scoping
There are at least two distinct ways in which local variables and nonlocal references to them can be implemented in a dynamic-scoped language: deep access and shallow access. Note that deep access and shallow access are not concepts related to deep and shallow binding. An important difference between binding and access is that deep and shallow bindings result in different semantics; deep and shallow accesses do not.

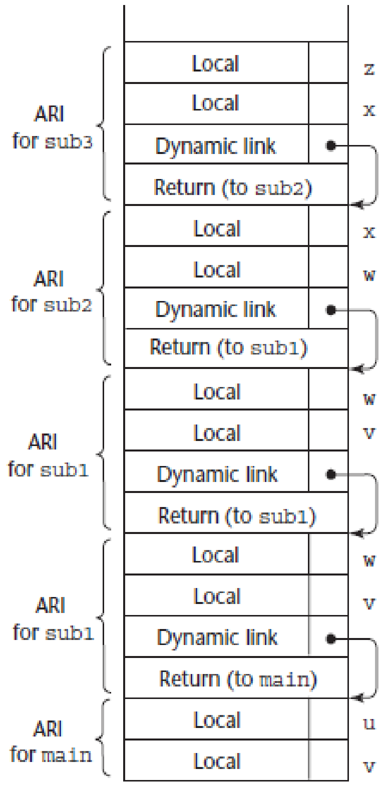
Deep Access
If local variables are stack dynamic and are part of the activation records in a dynamic-scoped language, references to nonlocal variables can be resolved by searching through the activation record instances of the other subprograms that are currently active, beginning with the one most recently 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.
activation record instances in the reverse of the order in which they were activated. Therefore, the dynamic chain is exactly what is needed to reference nonlocal variables in a dynamic-scoped language. This method is called **deep access**, because access may require searches deep into the stack. Consider the following example skeletal program:

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

This program is written in a syntax that gives it the appearance of a program in a C-based language, but it is not meant to be in any particular language.
Suppose the following sequence of function calls occurs:

maincalls sub1
sub1calls sub1
sub1calls sub2
sub2calls 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.

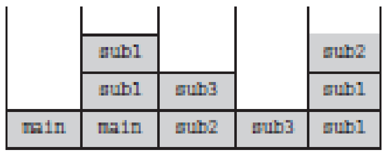


ARI = activation record instance

There are two important differences between the deep-access method for nonlocal access in a dynamic-scoped language and the static-chain method for static-scoped languages. First, 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. Second, 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

Shallow access is an alternative implementation method, not an alternative semantics. As stated previously, the semantics of deep access and shallow access are identical. 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. 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.
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



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

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.
Maintenance of a central table is straightforward. 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 is, if it contains a variable whose lifetime has not yet ended (which is indicated by the active bit)—that value must be saved somewhere during the lifetime of the new variable. Whenever a variable begins its lifetime, the active bit in its central table position must be set.

There have been several variations in the design of the central table and in the way values are stored when they are temporarily replaced. 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.

The second variation is perhaps the cleanest and least expensive to implement. A central table of single cells is used, storing only the current version of each variable with a unique name. Replaced variables are stored in the activation record of the subprogram that created the replacement variable. This is a stack mechanism, but it uses the stack that already exists, so the new overhead is minimal.
The choice between shallow and deep access to nonlocal variables depends on the relative frequencies of subprogram calls and non local references.
The deep-access method provides fast subprogram linkage, but references to non locals, especially references to distant non locals (in terms of the call chain), are costly. The shallow-access method provides much faster references to non locals, especially distant non locals, but is more costly in terms of subprogram linkage.