9

Subprograms

9.1 Introduction
9.2 Fundamentals of Subprograms
9.3 Design Issues for Subprograms
9.4 Local Referencing Environments
9.5 Parameter-Passing Methods
9.6 Parameters That Are Subprograms
9.7 Calling Subprograms Indirectly
9.8 Design Issues for Functions
9.9 Overloaded Subprograms
9.10 Generic Subprograms
9.11 User-Defined Overloaded Operators
9.12 Closures
9.13 Coroutines

Subprograms are the fundamental building blocks of programs and are therefore among the most important concepts in programming language design. We now explore the design of subprograms, including parameter-passing methods, local referencing environments, overloaded subprograms, generic subprograms, and the aliasing and problematic side effects that are associated with subprograms. We also include discussions of indirectly called subprograms, closures, and coroutines.

Implementation methods for subprograms are discussed in Chapter 10.

9.1 Introduction

Two fundamental abstraction facilities can be included in a programming language: process abstraction and data abstraction. In the early history of high-level programming languages, only process abstraction was included. Process abstraction, in the form of subprograms, has been a central concept in all programming languages. In the 1980s, however, many people began to believe that data abstraction was equally important. Data abstraction is discussed in detail in Chapter 11.

The first programmable computer, Babbage's Analytical Engine, built in the 1840s, had the capability of reusing collections of instruction cards at several different places in a program. In a modern programming language, such a collection of statements is written as a subprogram. This reuse results in savings in memory space and coding time. Such reuse is also an abstraction, for the details of the subprogram's computation are replaced in a program by a statement that calls the subprogram. Instead of describing how some computation is to be done in a program, that description (the collection of statements in the subprogram) is enacted by a call statement, effectively abstracting away the details. This increases the readability of a program by emphasizing its logical structure while hiding its low-level details.

The methods of object-oriented languages are closely related to the subprograms discussed in this chapter. The primary way methods differ from subprograms is the way they are called and their associations with classes and objects. Although these special characteristics of methods are discussed in Chapter 12, the features they share with subprograms, such as parameters and local variables, are discussed in this chapter.

9.2 Fundamentals of Subprograms

9.2.1 General Subprogram Characteristics

All subprograms discussed in this chapter, except the coroutines described in Section 9.13, have the following characteristics:

- 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.

Alternatives to these result in coroutines and concurrent units (Chapter 13).

Most subprograms have names, although some are anonymous. Section 9.12 has examples of anonymous subprograms in C#.

9.2.2 Basic Definitions

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. The two fundamental kinds of subprograms, procedures and functions, are defined and discussed in Section 9.2.4.

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.[1] 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 specify a list of parameters.

Consider the following header examples:

```
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 reserved word **`void`** in this header indicates that the subprogram does not return a value.

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 a Python function must be indented and the end of the body is indicated by the first statement that is not indented.

One characteristic of Python functions that sets them apart from the functions of other common programming languages is that function **`def`** statements are executable. When a **`def`** statement is executed, it assigns the given name to

1. Some programming languages include both kinds of subprograms, procedures and functions.

the given function body. Until a function's **def** has been executed, the function cannot be called. Consider the following skeletal example:

```
if . . .
  def fun( . . . ):
    . . .
else
  def fun( . . . ):
    . . .
```

If the then clause of this selection construct is executed, that version of the function `fun` can be called, but not the version in the else clause. Likewise, if the else clause is chosen, its version of the function can be called but the one in the then clause cannot.

Ruby methods differ from the subprograms of other programming languages in several interesting ways. Ruby methods are often defined in class definitions but can also be defined outside class definitions, in which case they are considered methods of the root object, `Object`. Such methods can be called without an object receiver, as if they were functions in C or C++. If a Ruby method is called without a receiver, **self** is assumed. If there is no method by that name in the class, enclosing classes are searched, up to `Object`, if necessary.

The **parameter profile** of a subprogram contains the number, order, and types of its formal parameters. The **protocol** of a subprogram is its parameter profile plus, if it is a function, its return type. In languages in which subprograms have types, those types are defined by the subprogram's protocol.

Subprograms can have declarations as well as definitions. This form parallels the variable declarations and definitions in C, in which declarations are used to provide type information but not to define variables. Subprogram declarations provide the subprogram's protocol but do not include their bodies. They are necessary in languages that do not allow forward references to subprograms. In both the cases of variables and subprograms, declarations are needed for static type checking. In the case of subprograms, it is the type of the parameters that must be checked. Function declarations are common in C and 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.

9.2.3 Parameters

Subprograms typically describe computations. There are two ways that a nonmethod 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 using 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. Extensive access to nonlocals can reduce reliability. Variables that are visible to the subprogram where access is desired often end up also being visible where access to them is not needed. This problem was discussed in Chapter 5.

Although methods also access external data through nonlocal references and parameters, the primary data to be processed by a method is the object through which the method is called. However, when a method does access nonlocal data, the reliability problems are the same as with nonmethod subprograms. Also, in an object-oriented language, method access to class variables (those associated with the class, rather than an object) is related to the concept of nonlocal data and should be avoided whenever possible. In this case, as well as the case of a C function accessing nonlocal data, the method can have the side effect of changing something other than its parameters or local data. Such changes complicate the semantics of the method and make it less reliable.

Pure functional programming languages, such as Haskell, do not have mutable data, so functions written in them are unable to change memory in any way—they simply perform calculations and return a resulting value (or function, since functions are values in a pure functional language).

In some situations, it is convenient to be able to transmit computations, rather than data, as parameters to subprograms. In these cases, the name of the subprogram that implements that computation may be used as a parameter. This form of parameter is discussed in Section 9.6. Data parameters are discussed in Section 9.5.

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**.[2] They must be distinguished from formal parameters, because the two usually have different restrictions on their forms, and of course, their uses are quite different.

---

2. Some authors call actual parameters *arguments* and formal parameters just *parameters*.

In most programming languages, the correspondence between actual and formal parameters—or the binding of actual parameters to formal parameters—is done by position: The first actual parameter is bound to the first formal parameter and so forth. Such parameters are called **positional parameters**. This is an effective and safe method of relating actual parameters to their corresponding formal parameters, as long as the parameter lists are relatively short.

When parameter lists are long, however, it is easy for a programmer to make mistakes in the order of actual parameters in the list. One solution to this problem is to provide **keyword parameters**, in which the name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter in a call. The advantage of keyword parameters is that they can appear in any order in the actual parameter list. Python functions can be called using this technique, as in

```
sumer(length = my_length,
      list = my_array,
      sum = my_sum)
```

where the definition of `sumer` has the formal parameters `length`, `list`, and `sum`.

The disadvantage to keyword parameters is that the user of the subprogram must know the names of formal parameters.

In addition to keyword parameters, some languages, for example Python, allow positional parameters. Keyword and positional parameters can be mixed in a call, as in

```
sumer(my_length,
      sum = my_sum,
      list = my_array)
```

The only restriction with this approach is that after a keyword parameter appears in the list, all remaining parameters must be keyworded. This restriction is necessary because a position may no longer be well defined after a keyword parameter has appeared.

In Python, Ruby, C++, 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)
```

The `exemptions` formal parameter can be absent in a call to `compute_pay`; when it is, the value `1` is used. No comma is included for an absent actual parameter in a Python call, because the only value of such a comma would be to indicate the position of the next parameter, which in this case is not

necessary because all actual parameters after an absent actual parameter must be keyworded. For example, consider the following call:

```
pay = compute_pay(20000.0, tax_rate = 0.15)
```

In C++, which does not support keyword parameters, the rules for default parameters are necessarily different. The default parameters must appear last, because parameters are positionally associated. Once a default parameter is omitted in a call, all remaining formal parameters must have default values. A C++ function header for the `compute_pay` function can be written as follows:

```
float compute_pay(float income, float tax_rate,
                  int exemptions = 1)
```

Notice that the parameters are rearranged so that the one with the default value is last. An example call to the C++ `compute_pay` function is

```
pay = compute_pay(20000.0, 0.15);
```

In most languages that do not have default values for formal parameters, the number of actual parameters in a call must match the number of formal parameters in the subprogram definition header. However, in C, C++, Perl, and JavaScript, this is not required. When there are fewer actual parameters in a call than formal parameters in a function definition, it is the programmer's responsibility to ensure that the parameter correspondence, which is always positional, and the subprogram execution are sensible.

Although this design, which allows a variable number of parameters, is clearly prone to error, it is also sometimes convenient. For example, the `printf` function of C can print any number of items (data values and/or literals).

C# allows methods to accept a variable number of parameters, as long as they are of the same type. The method specifies its formal parameter with the **`params`** modifier. The call can send either an array or a list of expressions, whose values are placed in an array by the compiler and provided to the called method. For example, consider the following method:

```
public void DisplayList(params int[] list) {
   foreach (int next in list) {
      Console.WriteLine("Next value {0}", next);
   }
}
```

If `DisplayList` is defined for the class `MyClass` and we have the following declarations,

```
Myclass myObject = new Myclass;
int[] myList = new int[6] {2, 4, 6, 8, 10, 12};
```

`DisplayList` could be called with either of the following:

```
myObject.DisplayList(myList);
myObject.DisplayList(2, 4, 3 * x - 1, 17);
```

Ruby supports a complicated but highly flexible actual parameter configuration. The initial parameters are expressions, whose value objects are passed to the corresponding formal parameters. The initial parameters can be followed by a list of key => value pairs, which are placed in an anonymous hash and a reference to that hash is passed to the next formal parameter. These are used as a substitute for keyword parameters, which Ruby does not support. The hash item can be followed by a single parameter preceded by an asterisk. This parameter is called the *array formal parameter*. When the method is called, the array formal parameter is set to reference a new `Array` object. All remaining actual parameters are assigned to the elements of the new `Array` object. If the actual parameter that corresponds to the array formal parameter is an array, it must also be preceded by an asterisk, and it must be the last actual parameter.[3] So, Ruby allows a variable number of parameters in a way similar to that of C#. Because Ruby arrays can store different types, there is no requirement that the actual parameters passed to the array have the same type.

The following example skeletal function definition and call illustrate the parameter structure of Ruby:

```
list = [2, 4, 6, 8]
def tester(p1, p2, p3, *p4)
  . . .
end  . . .
tester('first', mon => 72, tue => 68, wed => 59, *list)
```

Inside `tester`, the values of its formal parameters are as follows:

```
p1 is 'first'
p2 is {mon => 72, tue => 68, wed => 59}
p3 is 2
p4 is [4, 6, 8]
```

Python supports parameters that are similar to those of Ruby.

9.2.4 Procedures and Functions

There are two distinct categories of subprograms—procedures and functions—both of which can be viewed as approaches to extending the language. Subprograms are collections of statements that define parameterized

3. Not quite true, because the array formal parameter can be followed by a method or function reference, which is preceded by an ampersand (&).

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. Only some older languages, such as Fortran and Ada, 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.

Functions structurally resemble procedures but are semantically modeled on mathematical functions. If a function is a faithful model, it produces no side effects; that is, it modifies neither its parameters nor any variables defined outside the function. Such a function returns a value—that is its only desired effect. The functions in most programming languages have side effects.

Functions are called by appearances of their names in expressions, along with the required actual parameters. The value produced by a function's execution is returned to the calling code, effectively replacing the call itself. For example, the value of the expression `f(x)` is whatever value `f` produces when called with the parameter `x`. For a function that does not produce side effects, the returned value is its only effect.

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 includes a similar function named `pow`. Compare this with the same operation in Perl, in which exponentiation is a built-in operation:

```
result = 3.4 * 10.0 ** x
```

In some programming languages, users are permitted to overload operators by defining new functions for operators. User-defined overloaded operators are discussed in Section 9.11.

9.3 Design Issues for Subprograms

Subprograms are complex structures, and it follows from this that a lengthy list of issues is involved in their design. One obvious issue is the choice of one or more parameter-passing methods that will be used. The wide variety of approaches that have been used in various languages is a reflection of the diversity of opinion on the subject. A closely related issue is whether the types of actual parameters will be type checked against the types of the corresponding formal parameters.

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. 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.

As seen in Chapter 5, side effects of functions can cause problems. So, restrictions on side effects are a design issue for functions. The types and number of values that can be returned from functions are other design issues.

Finally, there are 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.

The following is a summary of these design issues for subprograms in general. Additional issues that are specifically associated with functions are discussed in Section 9.10.

- 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?
- Are functional side effects allowed?
- What types of values can be returned from functions?
- How many values can be returned from functions?
- Can subprograms be overloaded?
- Can subprograms be generic?
- If the language allows nested subprograms, are closures supported?

These issues and example designs are discussed in the following sections.

9.4 Local Referencing Environments

This section discusses the issues related to variables that are defined within subprograms. The issue of nested subprogram definitions is also briefly covered.

9.4.1 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.

In the terminology of Chapter 5, 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 flexibility. 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 important 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.[4] This indirectness is required because the place in the stack where a particular local variable will reside can be determined only during execution (see Chapter 10). 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. Coroutines and the subprograms used in iterator loop constructs (discussed in Chapter 8) are other examples of subprograms that need to be history sensitive.

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. Also, if accessed directly, these accesses are obviously more efficient. And, of course, they allow subprograms to be history sensitive. The greatest disadvantage of static local variables is their inability to support recursion. Also, their storage cannot be shared with the local variables of other inactive subprograms.

4. In some implementations, static variables are also accessed indirectly, thereby eliminating this disadvantage.

In most contemporary languages, local variables in a subprogram are by default stack dynamic. 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.

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

The methods of C++, Java, and C# have only stack-dynamic local variables.

In Python, the only declarations used in method definitions are for globals. Any variable declared to be global in a method must be a variable defined outside the method. A variable defined outside the method can be referenced in the method without declaring it to be global, but such a variable cannot be assigned in the method. If the name of a global variable is assigned in a method, it is implicitly declared to be a local and the assignment does not disturb the global. All local variables in Python methods are stack dynamic.

9.4.2 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. Recall that in Chapter 5, the problems introduced by this were discussed. For a long time, the only languages that allowed nested subprograms were those directly descending from ALGOL 60, which were ALGOL 68, Pascal, and Ada. Many other languages, including all of the direct descendants of C, do not allow subprogram nesting. Recently, some new languages again allow it. Among these are JavaScript, Python, and Ruby. Also, most functional programming languages allow subprograms to be nested.

9.5 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. Then, we discuss the various implementation models invented by language designers for these semantics models. Next, we survey the design choices of several languages and discuss the actual methods used to implement the implementation models. Finally, we consider the design considerations that face a language designer in choosing among the methods.

9.5.1 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 **inout mode**, respectively. For example, consider a subprogram that takes two arrays of `int` values as parameters—`list1` and `list2`. The subprogram must add `list1` to `list2` and return the result as a revised version of `list2`. Furthermore, the subprogram must create a new array from the two given arrays and return it. For this subprogram, `list1` should be in mode, because it is not to be changed by the subprogram. `list2` must be inout mode, because the subprogram needs the given value of the array and must return its new value. The third array should be out mode, because there is no initial value for this array and its computed value must be returned to the caller.

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. Most commonly, the access path is a simple pointer or reference. Figure 9.1 illustrates the three semantics models of parameter passing when values are copied.

9.5.2 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.

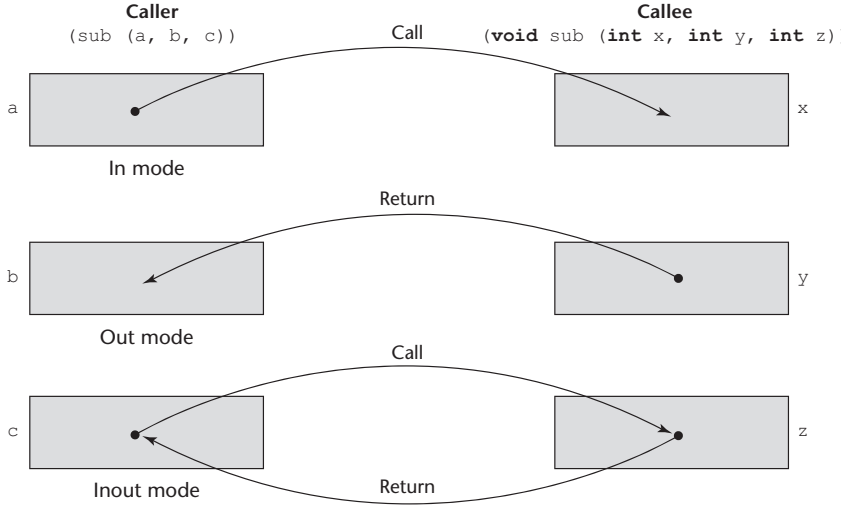


Figure 9.1

The three semantics models of parameter passing when physical moves are used

9.5.2.1 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 (one that can only be read). Enforcing the write protection is not always a simple matter. For example, suppose the subprogram to which the parameter was passed passes it in turn to another subprogram. This is another reason to use copy transfer. As we will see in Section 9.5.4, C++ provides a convenient and effective method for specifying write protection on pass-by-value parameters that are transmitted by access path.

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. In addition, the actual parameter must be copied to the storage area for the corresponding formal parameter. The storage and the copy operations can be costly if the parameter is large, such as an array with many elements.

9.5.2.2 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. (How would the caller reference the computed result if it were a literal or an expression?)

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.

One additional problem with the pass-by-result model is that there can be an actual parameter collision, such as the one created with the call

```
sub(p1, p1)
```

In `sub`, assuming the two formal parameters have different names, the two can obviously be assigned different values. Then, whichever of the two is copied to their corresponding actual parameter last becomes the value of `p1` in the caller. Thus, the order in which the actual parameters are copied determines

their value. For example, consider the following C# method, which specifies the pass-by-result method with the `out` specifier on its formal parameter.[5]

```
void Fixer(out int x, out int y) {
  x = 17;
  y = 35;
}
. . .
f.Fixer(out a, out a);
```

If, at the end of the execution of `Fixer`, the formal parameter `x` is assigned to its corresponding actual parameter first, then the value of the actual parameter `a` in the caller will be `35`. If `y` is assigned first, then the value of the actual parameter `a` in the caller will be `17`.

Because the order can be implementation dependent for some languages, different implementations can produce different results.

Calling a subprogram with two identical actual parameters can also lead to different kinds of problems when other parameter-passing methods are used, as discussed in Section 9.5.2.4.

Another problem that can occur with pass-by-result is that the implementor may be able to choose between two different times to evaluate the addresses of the actual parameters: at the time of the call or at the time of the return. For example, consider the following C# method and following code:

```
void DoIt(out int x, int index){
  x = 17;
  index = 42;
}
. . .
sub = 21;
f.DoIt(list[sub], sub);
```

The address of `list[sub]` changes between the beginning and end of the method. The implementor must choose the time to bind this parameter to an address—at the time of the call or at the time of the return. If the address is computed on entry to the method, the value `17` will be returned to `list[21]`; if computed just before return, `17` will be returned to `list[42]`. This makes programs unportable between an implementation that chooses to evaluate the addresses for out-mode parameters at the beginning of a subprogram and one that chooses to do that evaluation at the end. An obvious way to avoid this problem is for the language designer to specify when the address to be used to return the parameter value must be computed.

9.5.2.3 Pass-by-Value-Result

Pass-by-value-result is an implementation model for inout-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

5. The **out** specifier must also be specified on the corresponding actual parameter.

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, so they are discussed in Section 9.5.2.4.

9.5.2.4 Pass-by-Reference

Pass-by-reference is a second implementation model for inout-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 and no copying is 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.[6] Second, if only one-way communication to the called subprogram is required, inadvertent and erroneous changes may be made to the actual parameter. This issue is addressed below.

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. Another issue with pass by reference is whether the called subprogram is allowed to change a passed pointer. In C, this is possible, but in some other languages, such as Pascal and C++, formal parameters that are addresses are implicitly dereferenced in the called subprogram, which prevents such changes.

There are several ways pass-by-reference parameters can create aliases. First, collisions can occur between actual parameters. Consider a C++ function that has two parameters that are to be passed by reference, as in

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

6. This is further explained in Section 9.5.3.

If the call to `fun` happens to pass the same variable twice, as in

```
fun(total, total)
```

then `first` and `second` in `fun` will be aliases.

Second, collisions between array elements can also cause aliases. For example, suppose the function `fun` is called with two array elements that are specified with variable subscripts, as in

```
fun(list[i], list[j])
```

If these two parameters are passed by reference and `i` happens to be equal to `j`, then `first` and `second` are again aliases.

Third, if two of the formal parameters of a subprogram are an element of an array and the whole array, and both are passed by reference, then a call such as

```
fun1(list[i], list)
```

could result in aliasing in `fun1`, because `fun1` can access all elements of `list` through the second parameter and access a single element through its first parameter.

Still another way to get aliasing with pass-by-reference parameters is through collisions between formal parameters and nonlocal variables that are visible. For example, consider the following C code:

```
int * global;
void main() {
   . . .
   sub(global);
   . . .
}
 void  sub(int  * param) {
   . . .
}
```

Inside `sub`, `param` and `global` are aliases.

All these possible aliasing situations are eliminated if pass-by-value-result is used instead of pass-by-reference. However, in place of aliasing, other problems sometimes arise, as discussed in Section 9.5.2.3.

9.5.2.5 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 by name, 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. This subprogram/referencing environment is a closure (see Section 9.12).[7] Pass-by-name parameters are both complex to implement and inefficient. They also add significant complexity to the program, thereby lowering its readability and reliability.

Because pass-by-name is not part of any widely used language, it is not discussed further here. However, it is used at compile time by the macros in assembly languages and for the generic parameters of the generic subprograms in C++, Java 5.0, and C# 2005, as discussed in Section 9.9.

9.5.3 Implementing Parameter-Passing Methods

We now address the question of how the various implementation models of parameter passing are actually implemented.

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, as discussed in Chapter 10. In the following discussion, we assume that the stack is used for all parameter transmission.

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.

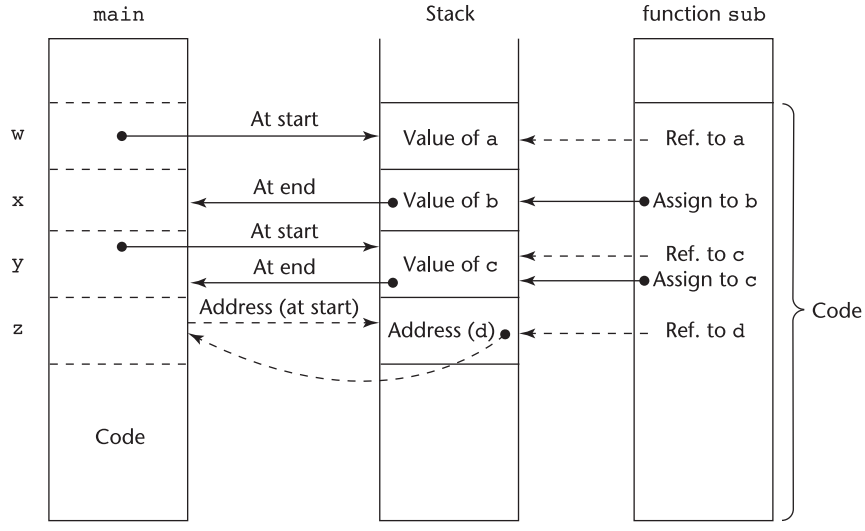
Pass-by-reference parameters are perhaps the simplest to implement. Most languages only allow variables to be passed by reference. However, Fortran passes all forms of parameters by reference. In Fortran, regardless of the type of the actual parameter, only its address must be placed in the stack. In 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, which must be executed 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 Fortran compiler must prevent the called subprogram from changing parameters that are literals or expressions.

---

7. These closures were originally (in ALGOL 60) called *thunks*.

Figure 9.2

One possible stack implementation of the common parameter-passing methods



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)

Access to the formal parameters in the called subprogram is by indirect addressing from the stack location of the address. The implementation of pass-by-value, -result, -value-result, and -reference, where the run-time stack is used, is shown in Figure 9.2. Subprogram `sub` is called from `main` with the call `sub(w, x, y, z)`, where `w` is passed by value, `x` is passed by result, `y` is passed by value-result, and `z` is passed by reference.

9.5.4 Parameter-Passing Methods of Some Common Languages

C uses pass-by-value. Pass-by-reference (inout mode) semantics is achieved by using pointers as parameters. The value of the pointer is made available to the called function and nothing is copied back. However, because what was passed is an access path to the data of the caller, the called function can change the caller's data. But all references to pointer formal parameter must be explicitly dereferenced in the function. C copied this use of the pass-by-value method from ALGOL 68. In both C and C++, formal parameters can be typed as pointers to constants. The corresponding actual parameters need not be constants, for in such cases they are coerced to constants. This allows pointer parameters to provide the efficiency of pass-by-reference with the one-way semantics of pass-by-value. Write protection of those parameters in the called function is implicitly specified.

C++ includes a special pointer type, called a *reference type*, as discussed in Chapter 6, which is often used for parameters. Reference parameters are implicitly dereferenced in the function or method, and their semantics is pass-by-reference. C++ also allows reference parameters to be defined to be constants. For example, we could have

history note

ALGOL 60 introduced the pass-by-name method. It also allowed pass-by-value as an option. Primarily because of the difficulty in implementing them, pass-by-name parameters were not carried from ALGOL 60 to any subsequent languages that became popular (other than SIMULA 67).

history note

ALGOL W (Wirth and Hoare, 1966) introduced the pass-by-value-result method of parameter passing as an alternative to the inefficiency of pass-by-name and the problems of pass-by-reference.

```
void fun(const int &p1, int p2, int &p3) { . . . }
```

where `p1` is pass-by-reference but cannot be changed in the function `fun`, `p2` is pass-by-value, and `p3` is pass-by-reference. Neither `p1` nor `p3` need be explicitly dereferenced in `fun`.

Constant parameters and in-mode parameters are not exactly alike. Constant parameters clearly implement in mode. However, in all of the common imperative languages except Ada, in-mode parameters can be assigned in the subprogram even though those changes are never reflected in the values of the corresponding actual parameters. In Ada, such an assignment is illegal. Constant parameters can never be assigned.

As with C and C++, all Java parameters are passed by value. However, because objects can be accessed only through reference variables, object parameters are in effect passed by reference. Although an object reference passed as a parameter cannot itself be changed in the called subprogram, the referenced object can be changed if a method is available to cause the change. Because reference variables cannot point to scalar variables directly and Java does not have pointers, scalars cannot be passed by reference in Java (although a reference to an object that contains a scalar can). Therefore, if a scalar is passed to a Java method, it cannot be changed by that method.

The default parameter-passing method of C# is pass-by-value. Pass-by-reference can be specified by preceding both a formal parameter and its corresponding actual parameter with **`ref`**. For example, consider the following C# skeletal method and call:

```
void sumer(ref int oldSum, int newOne) { . . . }
. . .
sumer(ref sum, newValue);
```

The first parameter to `sumer` is passed by reference; the second is passed by value. All **`ref`** parameters must be assigned a value before they are passed to an actual parameter.

C# also supports out-mode parameters, which are pass-by-reference parameters that do not need initial values. Such parameters are specified in the formal parameter list with the **`out`** modifier.

PHP's parameter passing is similar to that of C#, except that either the actual parameter or the formal parameter can specify pass-by-reference. Pass-by-reference is specified by preceding one or both of the parameters with an ampersand.

In Swift, the default parameter passing method is pass by value, and formal parameters passed this way cannot be changed in the called subprogram. Pass-by-reference semantics can be specified by preceding the formal parameter with the reserved word **`inout`**.

Perl employs a primitive means of passing parameters. All actual parameters are implicitly placed in a predefined array named `@_` (of all things!). The subprogram retrieves the actual parameter values (or addresses) from this array. The most peculiar thing about this array is its magical nature, exposed by the

fact that its elements are in effect aliases for the actual parameters. Therefore, if an element of `@_` is changed in the called subprogram, that change is reflected in the corresponding actual parameter in the call, assuming there is a corresponding actual parameter (the number of actual parameters need not be the same as the number of formal parameters) and it is a variable.

The parameter-passing method of Python and Ruby is called **pass-by-assignment** or **pass-by-sharing**. Because all data values are objects, every variable is a reference to an object. In pass-by-assignment, the actual parameter value is assigned to the formal parameter. Therefore, pass-by-assignment is in effect pass-by-reference, because the value of all actual parameters are references. However, only in certain cases does this result in pass-by-reference semantics. For example, many objects are essentially immutable. In a pure object-oriented language, the process of changing the value of a variable with an assignment statement, as in

```
x = x + 1
```

does not change the object referenced by `x`. Rather, it takes the object referenced by `x`, increments it by `1`, thereby creating a new object (with the value `x + 1`), and then changes `x` to reference the new object. So, when a reference to a scalar object is passed to a subprogram, the object being referenced cannot be changed in place. Because the reference is passed by value, even though the formal parameter is changed in the subprogram, that change has no effect on the actual parameter in the caller.

Now, suppose a reference to an array is passed as a parameter. If the corresponding formal parameter is assigned a new array object, there is no effect on the caller. However, if the formal parameter is used to assign a value to an element of the array, as in

```
list[3] = 47
```

the actual parameter is affected. So, changing the reference of the formal parameter has no effect on the caller, but changing an element of the array that is passed as a parameter does.

9.5.5 Type Checking Parameters

It is now widely accepted that software reliability demands that the types of actual parameters be checked for consistency with the types of the corresponding formal parameters. Without such type checking, small typographical errors can lead to program errors that may be difficult to diagnose because they are not detected by the compiler or the run-time system. For example, in the function call

```
result = sub1(1)
```

the actual parameter is an integer constant. If the formal parameter of `sub1` is a floating-point type, no error will be detected without parameter type checking. Although an integer 1 and a floating-point 1 have the same value, the

representations of these two are very different. `sub1` cannot produce a correct result given an integer actual parameter value if it expects a floating-point value.

Early programming languages, such as Fortran 77 and the original version of C, did not require parameter type checking; most later languages require it. However, the relatively recent languages Perl, JavaScript, and PHP do not.

C and C++ require some special discussion in the matter of parameter type checking. In the original C, neither the number of parameters nor their types were checked. In C89, the formal parameters of functions can be defined in two ways. They can be defined as in the original C; that is, the names of the parameters are listed in parentheses and the type declarations for them follow, as in the following function:

```
double sin(x)
  double x;
  { . . . }
```

Using this form avoids type checking, thereby allowing calls such as

```
double value;
int count;
. . .
value = sin(count);
```

to be legal, although they are never correct.

The alternative to the original C definition approach is called the **prototype** method, in which the formal parameter types are included in the list, as in

```
double sin(double x)
 { . . . }
```

If this version of `sin` is called with the same call, that is, with the following, it is also legal:

```
value = sin(count);
```

The type of the actual parameter (**`int`**) is checked against that of the formal parameter (**`double`**). Although they do not match, **`int`** is coercible to **`double`** (it is a widening coercion), so the conversion is done. If the conversion is not possible (for example, if the actual parameter had been an array) or if the number of parameters is wrong, then a semantics error is detected. So in C89, the user chooses whether parameters are to be type checked.

In C99 and C++, all functions must have their formal parameters in prototype form. However, type checking can be avoided for some of the parameters by replacing the last part of the parameter list with an ellipsis, as in

```
int  printf(const char* format_string, . . .);
```

A call to `printf` must include at least one parameter, a pointer to a literal character string. Beyond that, anything (including nothing) is legal. The way

`printf` determines whether there are additional parameters is by the presence of format codes in the string parameter. For example, the format code for integer output is `%d`. This appears as part of the string, as in the following:

```
printf("The sum is %d\n", sum);
```

The `%` tells the `printf` function that there is one more parameter.

There is one more interesting issue with actual to formal parameter coercions when primitives can be passed by reference, as in C#. Suppose a call to a method passes a **`float`** value to a **`double`** formal parameter. If this parameter is passed by value, the **`float`** value is coerced to **`double`** and there is no problem. This particular coercion is very useful, for it allows a library to provide double versions of subprograms that can be used for both **`float`** and **`double`** values. However, suppose the parameter is passed by reference. When the value of the **`double`** formal parameter is returned to the **`float`** actual parameter in the caller, the value will overflow its location. To avoid this problem, C# requires the type of a **`ref`** actual parameter to match exactly the type of its corresponding formal parameter (no coercion is allowed).

In Python and Ruby, there is no type checking of parameters, because typing in these languages is a different concept. Objects have types, but variables do not, so formal parameters are typeless. This disallows the very idea of type checking parameters.

9.5.6 Multidimensional Arrays as Parameters

The storage-mapping functions that are used to map the index values of references to elements of multidimensional arrays to addresses in memory were discussed at length in Chapter 6. In some languages, such as C and 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 (not the calling subprogram). This is true because the subprograms can be compiled separately from the programs that call them. Consider the problem of passing a matrix to a function in C. Multidimensional arrays in C are really arrays of arrays, and they are stored in row major order. Following is a storage-mapping function for row major order for matrices when the lower bound of all indices is 0 and the element size is 1:

address `(mat[i, j])` = address `(mat[0,0])` + `i` * number_of_columns + `j`

Notice that this mapping function needs the number of columns but not the number of rows. Therefore, in C and C++, when a matrix is passed as a parameter, the formal parameter must include the number of columns in the second pair of brackets. This is illustrated in the following skeletal C program:

```
void fun(int matrix[][10]) {
 . . . }
void main() {
```

```
  int mat[5][10];
  . . .
  fun(mat);
  . . .
}
```

The problem with this method of passing matrices as parameters is that it does not allow a programmer to 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, in effect, disallows writing flexible functions that may be effectively reusable if the functions deal with multidimensional arrays. In C and C++, there is a way around the problem because of their inclusion of pointer arithmetic. The matrix can be passed as a pointer, and the actual dimensions of the matrix also can be passed as parameters. Then, the function can evaluate the user-written storage-mapping function using pointer arithmetic each time an element of the matrix must be referenced. For example, consider the following function prototype:

```
void fun(float *mat_ptr,
         int num_rows,
         int num_cols);
```

The following statement can be used to move the value of the variable `x` to the `[row][col]` element of the parameter matrix in `fun`:

```
*(mat_ptr + (row * num_cols) + col) = x;
```

Although this works, it is obviously difficult to read, and because of its complexity, it is error prone. The difficulty with reading this can be alleviated by using a macro to define the storage-mapping function, such as

```
#define mat_ptr(r,c) (*mat_ptr + ((r) *
                     (num_cols) + (c)))
```

With this, the assignment can be written as

```
mat_ptr(row,col) = x;
```

Other languages use different approaches to dealing with the problem of passing multidimensional arrays.

In Java and C#, arrays are objects. They are all single dimensioned, but the elements can be arrays. Each array inherits a named constant (`length` in Java and `Length` in C#) that is set to the length of the array when the array object is created. The formal parameter for a matrix appears with two sets of empty brackets, as in the following Java method:

```
float sumer(float mat[][]) {
   float sum = 0.0f;
```

```
    for (int row = 0; row < mat.length; row++) {
      for (int col = 0; col < mat[row].length; col++) {
        sum += mat[row][col];
      }  //** for (int row . . .
    } //** for (int col . . .
    return sum;
}
```

Because each array has its own length value, in a matrix the rows can have different lengths.

9.5.7 Design Considerations

Two important considerations are involved in choosing parameter-passing methods: efficiency and whether one-way or two-way data transfer is needed.

Contemporary software-engineering principles dictate that access by subprogram code to data outside the subprogram should be minimized. With this goal in mind, in-mode parameters should be used whenever no data are to be returned through parameters to the caller. Out-mode parameters should be used when no data are transferred to the called subprogram but the subprogram must transmit data back to the caller. Finally, inout-mode parameters should be used only when data must move in both directions between the caller and the called subprogram.

There is a practical consideration that is in conflict with this principle. Sometimes it is justifiable to pass access paths for one-way parameter transmission. For example, when a large array is to be passed to a subprogram that does not modify it, a one-way method may be preferred. However, pass-by-value would require that the entire array be moved to a local storage area of the subprogram. This would be costly in both time and space. Because of this, large arrays are often passed by reference. This is precisely the reason why the Ada 83 definition allowed implementors to choose between the two methods for structured parameters. C++ constant reference parameters offer another solution. Another alternative approach would be to allow the user to choose between the methods.

The choice of a parameter-passing method for functions is related to another design issue: functional side effects. This issue is discussed in Section 9.10.

9.5.8 Examples of Parameter Passing

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 the following:

```
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-by-reference, except when aliasing is involved. Ada uses pass-by-value-result 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
```

Again, the subprogram operates correctly, in this case because the addresses to which to return the values of the parameters are computed at the time of the call rather than at the time of the return. If the addresses of the actual parameters were computed at the time of the return, the results would be wrong.

Finally, we must explore what happens when aliasing is involved with pass-by-value-result and pass-by-reference. Consider the following skeletal program written in C-like syntax:

```
int i = 3; /* i is a global variable */
void fun(int a, int b) {
  i = b;
}
void main() {
  int list[10];
  list[i] = 5;
  fun(i, list[i]);
}
```

In `fun`, if pass-by-reference is used, `i` and `a` are aliases. If pass-by-value-result is used, `i` and `a` are not aliases. The actions of `fun`, assuming pass-by-value-result, are as follows:

```
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
i = b                 - Sets i to 5
*addr_i = a           - Move first parameter value out
*addr_listi = b       - Move second parameter value out
```

In this case, the assignment to the global `i` in `fun` changes its value from `3` to `5`, but the copy back of the first formal parameter (the second to last line in the example) sets it back to `3`. The important observation here is that if pass-by-reference is used, the result is that the copy back is not part of the semantics, and `i` remains `5`. Also note that because the address of the second parameter is computed at the beginning of `fun`, any change to the global `i` has no effect on the address used at the end to return the value of `list[i]`.

9.6 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.

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:

```
function sub1() {
  var x;
  function sub2() {
    alert(x); // Creates a dialog box with the value of x
    };
  function  sub3() {
    var  x;
    x = 3;
    sub4(sub2);
    };
  function  sub4(subx) {
    var  x;
    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` in `sub1`, and the output is `1`. For ad hoc binding, the binding is to the local `x` in `sub3`, and the output is `3`.

history note

The original definition of Pascal (Jensen and Wirth, 1974) allowed subprograms to be passed as parameters without including their parameter type information. If independent compilation is possible (it was not possible in the original Pascal), the compiler is not even allowed to check for the correct number of parameters. In the absence of independent compilation, checking for parameter consistency is possible but is a very complex task, and it usually is not done.

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 highly 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.

9.7 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 callbacks, 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

following 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:

```
static int fun1(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 `chgfun1` 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**. Regardless 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 (see Section 9.12).

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.

9.8 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?

9.8.1 Functional Side Effects

Because of the problems of side effects of functions that are called in expressions, as described in Chapter 5, parameters to functions should always be in-mode. In fact, some languages require this; for example, Ada functions can have only in-mode formal parameters. This requirement effectively prevents a function from causing side effects through its parameters or through aliasing of parameters and globals. In most other imperative languages, however, functions

can have either pass-by-value or pass-by-reference parameters, thus allowing functions that cause side effects and aliasing.

Pure functional languages, such as Haskell and Ruby, do not have variables, so their functions cannot have side effects.

9.8.2 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, and Ruby 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 and Ruby. 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.

9.8.3 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.

In ML, F#, and Python, and some other languages that include tuples, multiple values can be returned by placing them in a tuple.

9.9 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. 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. Although it is not necessary, overloaded subprograms usually implement the same process.

C++, Java, 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 of the 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 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.

9.10 Generic Subprograms

Software reuse can be an important contributor to software productivity. One way to increase the reusability of software is to lessen 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 behave 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. C++, Java 5.0+, C# 2005+, and F# provide a kind of compile-time parametric polymorphism.

9.10.1 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.[8]

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

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

8. Template classes are discussed in Chapter 11.

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 `max` would be instantiated twice by the following code segment—once for **`int`** type parameters and once for **`char`** type parameters:

```
int a, b, c;
char d, 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:

```
float flt_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.

9.10.2 Generic Methods in Java 5.0

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>

where `T` is the type variable. Generic types are discussed in more detail in Chapter 11.

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**.

As an example of a generic Java 5.0 method, consider the following skeletal method definition:

```
public static  <T> T doIt(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) {
  . . .
}
```

This defines a method that takes a generic array parameter whose elements are of a class that implements the `Comparable` interface. That is the restriction, or bound, on the generic parameter. The reserved word **`extends`** seems to imply that the generic class subclasses the following class. In this context, however, **`extends`** has a different meaning. The expression `<T extends BoundingType>` specifies that `T` should be a "subtype" of the bounding type. So, in this context, **`extends`** means the generic class (or interface) either extends the bounding class (the bound if it is a class) or implements the bounding interface (if the bound is an interface). The bound ensures that the elements of any instantiation of the generic can be compared with the `Comparable` method, `compareTo`.

If a generic method has two or more restrictions on its generic type, they are added to the **`extends`** clause, separated by ampersands (`&`). Also, generic methods can have more than one generic parameter.

Java 5.0 supports *wildcard types*. For example, `Collection<?>` is a wildcard type for collection classes. This type can be used for any collection type of any class components. For example, consider the following generic method:

```
void  printCollection(Collection<?> c) {
   for  (Object e: c) {
     System.out.println(e);
  }
}
```

This method prints the elements of any `Collection` class, regardless of the class of its components. Some care must be taken with objects of the wildcard type. For example, because the components of a particular object of this type have a type, other type objects cannot be added to the collection. For example, consider

```
Collection<?> c = new ArrayList<String>();
```

It would be illegal to use the `add` method to put something into this collection unless its type were `String`.

Wildcard types can be restricted, as is the case with nonwildcard types. Such types are called *bounded wildcard types*. For example, consider the following method header:

```
public void drawAll(ArrayList<? extends Shape> things)
```

The generic type here is a wildcard type that is a subclass of the `Shape` class. This method could be written to draw any object whose type is a subclass of `Shape`.

9.10.3 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# 2005 generic 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 infer 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>
string myStr = MyClass.DoIt('apples');
    // Calls DoIt<string>
```

9.10.4 Generic Functions in F#

The type inferencing system of F# is not always able to determine the type of parameters or the return type of a function. When this is the case, for some functions F# infers a generic type for the parameters and the return value. This is called **automatic generalization**. For example, consider the following function definition:

```
let getLast (a, b, c) = c;;
```

Because no type information was included, the types of the parameters and the return value are all inferred to be generic. Because this function does not include any computations, this is a simple generic function.

Functions can be defined to have generic parameters, as in the following example:

```
let printPair (x: 'a) (y: 'a) =
   printfn "%A %A" x y;;
```

The `%A` format specification is for any type. The apostrophe in front of the type named `a` specifies it to be a generic type.[9] This function definition works (with generic parameters) because no type-constrained operation is included. Arithmetic operators are examples of type-constrained operations. For example, consider the following function definition:

```
let adder x y = x + y;;
```

Type inferencing sets the type of `x` and `y` and the return value to **`int`**. Because there is no type coercion in F#, the following call is illegal:

```
adder 2.5 3.6;;
```

Even if the type of the parameters were set to be generic, the `+` operator would cause the types of `x` and `y` to be **`int`**.

The generic type could also be specified explicitly in angle brackets, as in the following:

```
let printPair2<'T> x y =
    printfn "%A %A" x y;;
```

This function must be called with a type,[10] as in the following:

```
printPair2<float> 3.5 2.4;;
```

Because of type inferencing and the lack of type coercions, F# generic functions are far less useful, especially for numeric computations, than those of C++, Java 5.0+, and C# 2005+.

9.11 User-Defined Overloaded Operators

Operators can be overloaded by the user in Ada, C++, Python, and Ruby. Suppose that a Python class is developed to support complex numbers and arithmetic operations on them. A complex number can be represented with two floating-point values. The `Complex` class would have members for these two

9. There is nothing special about `a`—it could be any legal identifier. By convention, lowercase letters at the beginning of the alphabet are used.
10. Convention explicitly states that generic types are named with uppercase letters starting at T.

named `real` and `imag`. In Python, binary arithmetic operations are implemented as method calls sent to the first operand, sending the second operand as a parameter. For addition, the method is named `__add__`. For example, the expression `x + y` is implemented as `x.__add__(y)`. To overload `+` for the addition of objects of the new `Complex` class, we only need to provide `Complex` with a method named `__add__` that performs the operation. Following is such a method:

```
def __add__ (self, second):
  return Complex(self.real + second.real, self.imag +
   second.imag)
```

In most languages that support object-oriented programming, a reference to the current object is implicitly sent with each method call. In Python, this reference must be sent explicitly; that is the reason why `self` is the first parameter to our method, `__add__`.

The example add method could be written for a complex class in C++ as follows[11]:

```
Complex operator +(Complex &second) {
  return  Complex(real + second.real, imag + second.imag);
}
```

9.12 Closures

Defining a closure is a simple matter; a **closure** is a subprogram and the referencing environment where it was defined. The referencing environment is needed if the subprogram can be called from any arbitrary place in the program. Explaining a closure is not so simple.

If a static-scoped programming language does not allow nested subprograms, closures are not useful, so such languages do not support them. All of the variables in the referencing environment of a subprogram in such a language (its local variables and the global variables) are accessible, regardless of the place in the program where the subprogram is called.

When subprograms can be nested, in addition to locals and globals, the referencing environment of a subprogram can include variables defined in all enclosing subprograms. However, this is not an issue if the subprogram can be called only in places where all of the enclosing scopes are active and visible. It becomes an issue if a subprogram can be called elsewhere. This can happen if the subprogram can be passed as a parameter or assigned to a variable, thereby allowing it to be called from virtually anywhere in the program. There is an associated problem: The subprogram could be called after one or more of its

11. Both C++ and Python have predefined classes for complex numbers, so our example methods are unnecessary, except as illustrations.

nesting subprograms has terminated, which normally means that the variables defined in such nesting subprograms have been deallocated—they no longer exist. For the subprogram to be callable from anywhere in the program, its referencing environment must be available wherever it might be called. Therefore, the variables defined in nesting subprograms may need lifetimes that are of the entire program, rather than just the time during which the subprogram in which they were defined is active. A variable whose lifetime is that of the whole program is said to have **unlimited extent**. This usually means they must be heap dynamic, rather than stack dynamic.

Nearly all functional programming languages, most scripting languages, and at least one primarily imperative language, C#, support closures. These languages are static-scoped, allow nested subprograms,[12] and allow subprograms to be passed as parameters. Following is an example of a closure written in JavaScript:

```
function makeAdder(x) {
    return function(y) {return x + y;}
}
. . .
var add10 = makeAdder(10);
var add5 = makeAdder(5);
document.write("Add 10 to 20: " + add10(20) +
"<br />");
document.write("Add 5 to 20: " + add5(20) +
"<br />");
```

The output of this code, assuming it was embedded in an HTML document and displayed with a browser, is as follows:

```
Add 10 to 20: 30
Add 5 to 20: 25
```

In this example, the closure is the anonymous function defined inside the `makeAdder` function, which `makeAdder` returns. The variable `x` referenced in the closure function is bound to the parameter that was sent to `makeAdder`. The `makeAdder` function is called twice, once with a parameter of `10` and once with `5`. Each of these calls returns a different version of the closure because they are bound to different values of `x`. The first call to `makeAdder` creates a function that adds `10` to its parameter; the second creates a function that adds `5` to its parameter. The two versions of the function are bound to different activations of `makeAdder`. Obviously, the lifetime of the version

12. In C#, the only methods that can be nested are anonymous delegates and lambda expressions.

of x created when makeAdder is called must extend over the lifetime of the program.

This same closure function can be written in C# using a nested anonymous delegate. The type of the nesting method is specified to be a function that takes an **int** as a parameter and returns an anonymous delegate. The return type is specified with the special notation for such delegates, Func<int, int>. The first type in the angle brackets is the parameter type. Such a delegate can encapsulate methods that have only one parameter. The second type is the return type of the method encapsulated by the delegate.

```
static Func<int, int> makeAdder(int x) {
  return delegate(int y) { return x + y;};
}
. . .
Func<int, int> Add10 = makeAdder(10);
Func<int, int> Add5 = makeAdder(5);
Console.WriteLine("Add 10 to 20: {0}", Add10(20));
Console.WriteLine("Add 5 to 20: {0}", Add5(20));
```

The output of this code is exactly the same as for the previous JavaScript closure example.

The anonymous delegate could have been written as a lambda expression. The following is a replacement for the body of the makeAdder method, using a lambda expression instead of the delegate:

```
return y => x + y
```

Ruby's blocks are implemented so that they can reference variables visible in the position in which they were defined, even if they are called at a place in which those variables would have disappeared. This makes such blocks closures.

9.13 Coroutines

A **coroutine** is a special kind of subprogram. Rather than the master-slave relationship between a caller and a called subprogram that exists with conventional subprograms, caller and called coroutines are more equitable. In fact, the coroutine control mechanism is often called the **symmetric unit control model**.

Coroutines can have multiple entry points, which are controlled by the coroutines themselves. They also have the means to maintain their status between activations. This means that coroutines must be history sensitive and thus have static local variables. Secondary executions of a coroutine often begin at points other than its beginning. Because of this, the invocation of a coroutine is called a **resume** rather than a call.

For example, consider the following skeletal coroutine:

```
sub co1(){
  . . .
  resume  co2();
  . . .
  resume  co3();
  . . .
}
```

The first time `co1` is resumed, its execution begins at the first statement and executes down to and including the resume of `co2`, which transfers control to `co2`. The next time `co1` is resumed, its execution begins at the first statement after its call to `co2`. When `co1` is resumed the third time, its execution begins at the first statement after the resume of `co3`.

One of the usual characteristics of subprograms is maintained in coroutines: Only one coroutine is actually in execution at a given time.

As seen in the example above, rather than executing to its end, a coroutine often partially executes and then transfers control to some other coroutine, and when restarted, a coroutine resumes execution just after the statement it used to transfer control elsewhere. This sort of interleaved execution sequence is related to the way multiprogramming operating systems work. Although there may be only one processor, all of the executing programs in such a system appear to run concurrently while sharing the processor. In the case of coroutines, this is sometimes called **quasi-concurrency**.

Typically, coroutines are created in an application by a program unit called the master unit, which is not a coroutine. When created, coroutines execute their initialization code and then return control to that master unit. When the entire family of coroutines is constructed, the master program resumes one of the coroutines, and the members of the family of coroutines then resume each other in some order until their work is completed, if in fact it can be completed. If the execution of a coroutine reaches the end of its code section, control is transferred to the master unit that created it. This is the mechanism for ending execution of the collection of coroutines, when that is desirable. In some programs, the coroutines run whenever the computer is running.

One example of a problem that can be solved with this sort of collection of coroutines is a card game simulation. Suppose the game has four players who all use the same strategy. Such a game can be simulated by having a master program unit create a family of four coroutines, each with a collection, or hand, of cards. The master program could then start the simulation by resuming one of the player coroutines, which, after it had played its turn, could resume the next player coroutine, and so forth until the game ended.

Suppose program units `A` and `B` are coroutines. Figure 9.3 shows two ways an execution sequence involving `A` and `B` might proceed.

In Figure 9.3a, the execution of coroutine `A` is started by the master unit. After some execution, `A` starts `B`. When coroutine `B` in Figure 9.3a first causes

Figure 9.3

Two possible execution control sequences for two coroutines without loops

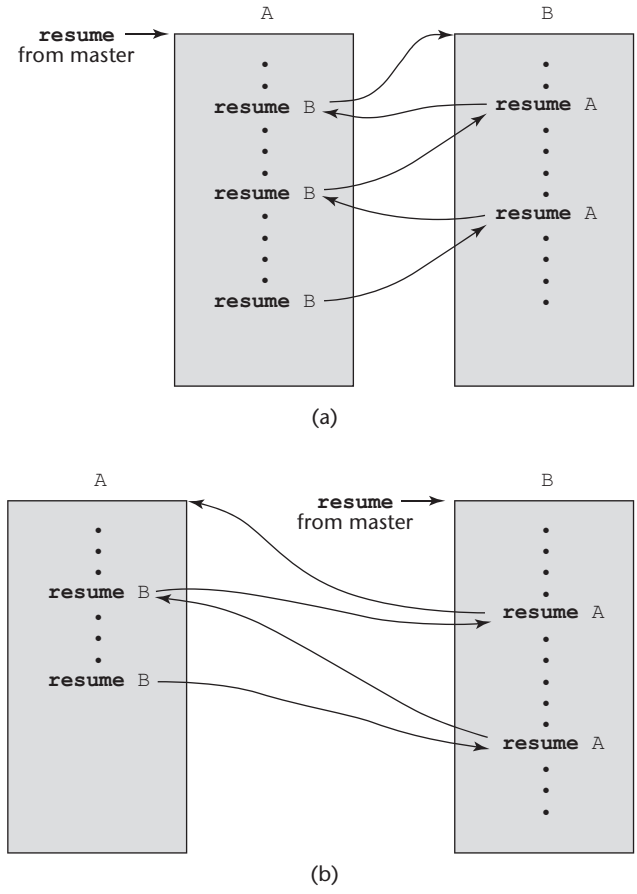
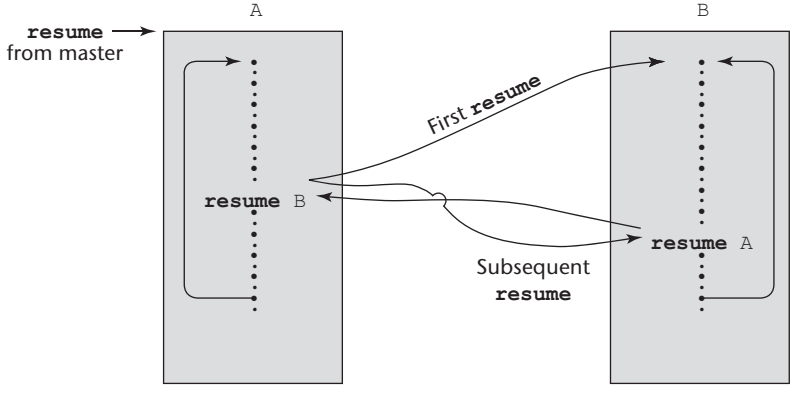


Figure 9.4

Coroutine execution sequence with loops



control to return to coroutine A, the semantics is that A continues from where it ended its last execution. In particular, its local variables have the values left them by the previous activation. Figure 9.3b shows an alternative execution sequence of coroutines A and B. In this case, B is started by the master unit.

Rather than have the patterns shown in Figure 9.3, a coroutine often has a loop containing a resume. Figure 9.4 shows the execution sequence of this scenario. In this case, `A` is started by the master unit. Inside its main loop, `A` resumes `B`, which in turn resumes `A` in its main loop.

The generators of Python are a form of coroutines.[13]

SUMMARY

Process abstractions are represented in programming languages by subprograms. A subprogram definition describes the actions represented by the subprogram. A subprogram call enacts those actions. A subprogram header identifies a subprogram definition and provides its interface, which is called its protocol.

Formal parameters are the names that subprograms use to refer to the actual parameters given in subprogram calls. In Python and Ruby, array and hash formal parameters are used to support variable numbers of parameters. JavaScript also supports variable numbers of parameters. Actual parameters can be associated with formal parameters by position or by keyword. Parameters can have default values.

Subprograms can be either functions, which model mathematical functions and are used to define new operations, or procedures, which define new statements.

Local variables in subprograms can be stack dynamic, providing support for recursion, or static, providing efficiency and history-sensitive local variables.

JavaScript, Python, Ruby, and Swift allow subprogram definitions to be nested.

There are three fundamental semantics models of parameter passing—in mode, out mode, and inout mode—and a number of approaches to implement them. These are pass-by-value, pass-by-result, pass-by-value-result, pass-by-reference, and pass-by-name. In most languages, parameters are passed in the run-time stack.

Aliasing can occur when pass-by-reference parameters are used, both among two or more parameters and between a parameter and an accessible nonlocal variable.

Parameters that are multidimensioned arrays pose some issues for the language designer, because the called subprogram needs to know how to compute the storage mapping function for them. This requires more than just the name of the array.

Parameters that are subprogram names provide a necessary service but can be difficult to understand. The opacity lies in the referencing environment that is available when a subprogram that has been passed as a parameter is executed.

---

13. However, the generators of Python are a form of coroutines.

C and C++ support pointers to functions. C# has delegates, which are objects that can store references to methods. Delegates can support multicast calls by storing more than one method reference.

Ada, C++, C#, Ruby, and Python allow both subprogram and operator overloading. Subprograms can be overloaded as long as the various versions can be disambiguated by the types of their parameters or returned values. Function definitions can be used to build additional meanings for operators.

Subprograms in C++, Java 5.0, and C# 2005 can be generic, using parametric polymorphism, so the desired types of their data objects can be passed to the compiler, which then can construct units for the requested types.

The designer of a function facility in a language must decide what restrictions will be placed on the returned values, as well as the number of return values.

A closure is a subprogram and its referencing environment. Closures are useful in languages that allow nested subprograms, are static-scoped, and allow subprograms to be returned from functions and assigned to variables.

A coroutine is a special subprogram that has multiple entries. It can be used to provide interleaved execution of subprograms.

REVIEW QUESTIONS

1. What are the three general characteristics of subprograms?
2. What does it mean for a subprogram to be active?
3. What is given in the header of a subprogram?
4. What characteristic of Python subprograms sets them apart from those of other languages?
5. What languages allow a variable number of parameters?
6. What is a Ruby array formal parameter?
7. What is a parameter profile? What is a subprogram protocol?
8. What are formal parameters? What are actual parameters?
9. What are the advantages and disadvantages of keyword parameters?
10. What are the differences between a function and a procedure?
11. What are the design issues for subprograms?
12. What are the advantages and disadvantages of dynamic local variables?
13. What are the advantages and disadvantages of static local variables?
14. What languages allow subprogram definitions to be nested?
15. What are the three semantics models of parameter passing?
16. What are the modes, the conceptual models of transfer, the advantages, and the disadvantages of pass-by-value, pass-by-result, pass-by-value-result, and pass-by-reference parameter-passing methods?
17. Describe the ways that aliases can occur with pass-by-reference parameters.

18. What is the difference between the way original C and C89 deal with an actual parameter whose type is not identical to that of the corresponding formal parameter?
19. What are two fundamental design considerations for parameter-passing methods?
20. Describe the problem of passing multidimensioned arrays as parameters.
21. What is the name of the parameter-passing method used in Ruby?
22. What are the two issues that arise when subprogram names are parameters?
23. Define *shallow* and *deep binding* for referencing environments of subprograms that have been passed as parameters.
24. What is an overloaded subprogram?
25. What is parametric polymorphism?
26. What causes a C++ template function to be instantiated?
27. In what fundamental ways do the generic parameters to a Java 5.0 generic method differ from those of C++ methods?
28. If a Java 5.0 method returns a generic type, what type of object is actually returned?
29. If a Java 5.0 generic method is called with three different generic parameters, how many versions of the method will be generated by the compiler?
30. What are the design issues for functions?
31. Name two languages that allow multiple values to be returned from a function.
32. What exactly is a delegate?
33. What is the main drawback of generic functions in F#?
34. What is a closure?
35. What are the language characteristics that make closures useful?
36. What languages allow the user to overload operators?
37. In what ways are coroutines different from conventional subprograms?

PROBLEM SET

1. What are arguments for and against a user program building additional definitions for existing operators, as can be done in Python and C++? Do you think such user-defined operator overloading is good or bad? Support your answer.

2. In most Fortran IV implementations, all parameters were passed by reference, using access path transmission only. State both the advantages and disadvantages of this design choice.
3. Argue in support of the Ada 83 designers' decision to allow the implementor to choose between implementing **`inout`**-mode parameters by copy or by reference.
4. Suppose you want to write a method that prints a heading on a new output page, along with a page number that is 1 in the first activation and that increases by 1 with each subsequent activation. Can this be done without parameters and without reference to nonlocal variables in Java? Can it be done in C#?
5. Consider the following program written in C syntax:

```
void swap(int a, int b) {
  int temp;
  temp = a;
  a = b;
  b = temp;
}
void main() {
  int value = 2, list[5] = {1, 3, 5, 7, 9};
  swap(value, list[0]);
  swap(list[0], list[1]);
  swap(value, list[value]);
}
```

   For each of the following parameter-passing methods, what are all of the values of the variables `value` and `list` after each of the three calls to `swap`?

   a. Passed by value
   b. Passed by reference
   c. Passed by value-result

6. Present one argument against providing both static and dynamic local variables in subprograms.
7. Consider the following program written in C syntax:

```
void fun (int first, int second) {
  first += first;
  second += second;
}
void main() {
  int list[2] = {1, 3};
  fun(list[0], list[1]);
}
```

For each of the following parameter-passing methods, what are the values of the `list` array after execution?

a. Passed by value
b. Passed by reference
c. Passed by value-result

8. Argue against the C design of providing only function subprograms.
9. From a textbook on Fortran, learn the syntax and semantics of statement functions. Justify their existence in Fortran.
10. Study the methods of user-defined operator overloading in C++ and Ada, and write a report comparing the two using our criteria for evaluating languages.
11. C# supports out-mode parameters, but neither Java nor C++ does. Explain the difference.
12. Research Jensen's Device, which was a use of pass-by-name parameters, and write a short description of what it is and how it can be used.
13. Study the iterator mechanisms of Ruby and CLU and list their similarities and differences.
14. Speculate on the issue of allowing nested subprograms in programming languages—why are they not allowed in many contemporary languages?
15. What are at least two arguments against the use of pass-by-name parameters?
16. Write a detailed comparison of the generic subprograms of Java 5.0 and C# 2005.

PROGRAMMING EXERCISES

1. Write a program in a language that you know to determine the ratio of the time required to pass a large array by reference and the time required to pass the same array by value. Make the array as large as possible on the machine and implementation you use. Pass the array as many times as necessary to get reasonably accurate timings of the passing operations.
2. Write a C# or Ada program that determines when the address of an out-mode parameter is computed (at the time of the call or at the time the execution of the subprogram finishes).
3. Write a Perl program that passes by reference a literal to a subprogram, which attempts to change the parameter. Given the overall design philosophy of Perl, explain the results.
4. Repeat Programming Exercise 3 in C#.
5. Write a program in some language that has both static and stack-dynamic local variables in subprograms. Create six large (at least 100 $\times$ 100)

matrices in the subprogram—three static and three stack dynamic. Fill two of the static matrices and two of the stack-dynamic matrices with random numbers in the range of 1 to 100. The code in the subprogram must perform a large number of matrix multiplication operations on the static matrices and time the process. Then it must repeat this with the stack-dynamic matrices. Compare and explain the results.

6. Write a C# program that includes two methods that are called a large number of times. Both methods are passed a large array, one by value and one by reference. Compare the times required to call these two methods and explain the difference. Be sure to call them a sufficient number of times to illustrate a difference in the required time.
7. Write a program, using the syntax of whatever language you like, that produces different behavior depending on whether pass-by-reference or pass-by-value-result is used in its parameter passing.
8. Write a generic C++ function that takes an array of generic elements and a scalar of the same type as the array elements. The type of the array elements and the scalar is the generic parameter. The function must search the given array for the given scalar and return the subscript of the scalar in the array. If the scalar is not in the array, the function must return `-1`. Test the function for `int` and `float` types.
9. Devise a subprogram and calling code in which pass-by-reference and pass-by-value-result of one or more parameters produces different results.