

## Chapter 9

# Functions and Parameters

---

---

### Overview

A function call consists of a name and actual arguments. These correspond to the declared function name and its formal parameters. Functions may be defined with a fixed or variable number of arguments, depending upon the language. Missing or extra parameters are handled by the translator of the particular language.

Arguments are passed to a function by the function call. They are usually matched up with parameter names positionally; the  $n$ th argument in the function call becomes the value of the  $n$ th parameter name in the function definition. Correspondence-by-keyword is also used. In this method, the dummy parameter name is written next to the argument expression in the function call. This parameter passing mechanism is very useful for functions that expect variable-length argument lists.

After the argument is passed, it is interpreted by the receiving function. In ALGOL-60, parameter interpretation mechanisms were limited to call-by-name and call-by-value. Call-by-reference was used in early FORTRAN. Call-by-need and call-by-pointer have since been devised as methods for passing and interpreting parameters.

A higher-order function is one that takes a function as an argument and/or returns a function as a result. Pure functional languages fully support higher-order functions. Flexible code and mapping are two common applications of functional arguments.

Currying is a way of looking at a function of two or more arguments, so that it is considered to be a higher-order function of one argument that returns a function.

Closure is an operation that binds the free variables of an expression. It creates and returns a function. The most common application of closure is partial parameterization.

## 9.1 Function Syntax

### 9.1.1 Fixed versus Variable Argument Functions

Many languages require that a function be defined with a fixed number of parameters of specified types. The required parameters and types for predefined functions are specified in the syntax of the language. The elements required for a control statement are similarly defined by the language syntax.

The parser requires every statement and function call to be well-formed: it counts the arguments, checks their types, and binds them to the defined dummy parameter names. If either the number or the type of arguments is wrong, the program is not well-formed and the translator gives an error comment. If some element is misplaced or missing, the translator cannot, in general, know the intended meaning of the program. For example, if an END statement is missing, it is impossible in many languages for a translator to guess, accurately, where it was supposed to be.

Because the parser rejects programs that are not well-formed, being well-formed can be seen as a syntactic requirement. But the syntax merely reflects semantic requirements: the code that defines the semantics of these statements and functions is not prepared to handle a variable number or type of argument.

A function *uses* a certain number of parameters; that is, it refers to their names. There are several ways in which there might be a mismatch between the number of arguments passed and the number of parameters that are actually used, as follows:

**Unused arguments:** Parameters might be named and passed but not referred to anywhere in the function. In a procedural language, an unused argument might as well be omitted. In functional languages, however, there are situations in which this is useful. (Consider the lambda calculus functions  $T$  and  $F$ , defined in Section 4.3.4, which each discard one parameter.)

**Optional parameters:** A parameter might be named and used, but no corresponding argument passed. This can be meaningful if default values are defined for missing arguments, and the argument-parameter correspondence is unambiguously specified.

**Indefinite-length argument lists:** Parameters might be passed but not named. These can be useful if some way is supplied by the language for referring to them.

**Optional parameters.** Several modern languages support optional parameters. If the number of actual arguments passed is smaller than the number of parameters called for, some parameters cannot be given a meaning in the normal way. If this is permitted, the meaning of such an omission must be defined by default or explicitly within the function. Every parameter must have a meaning

**Exhibit 9.1. An Ada function with a default value for one parameter.**

Imagine that this function is a small routine inside a chef's program for figuring out ingredient quantities when the quantity of a recipe must be changed. The usual case is to double the recipe, so the program has an optional parameter with a default value of 2.0.

```
function CONVERT ( quantity:real,
                  proportion: real := 2.0
                ) return real is
begin return quantity*proportion end CONVERT;
```

Legal and meaningful calls on CONVERT:

```
CONVERT(3.75)      -- Double the amount, result is 7.5.
CONVERT(3.75, .5) -- Halve the amount, result is 1.675.
```

*if and when* it is actually used in a computation. One solution, used in *Ada*, is that parameters may be defined to have default values. If an actual argument *is* supplied, its value is used for that parameter, otherwise the default value is used [Exhibit 9.1].

**9.1.2 Parameter Correspondence**

When calling a function with optional parameters, arguments may (or may not) be omitted. This complicates the problem of matching up argument values and parameter names, and makes the simple positional correspondence syntax inadequate. The problem can be handled in three ways:

- Permit the user to use adjacent commas to “hold the place” for an omitted argument.
- Require all arguments that *are* supplied to precede the ones that are omitted.
- Use correspondence-by-keyword for all arguments after the first one that is omitted, or for all optional arguments.

All three ways have been used. For example, command line interpreters commonly use the first and third conventions, and *Ada* supports the second and third. The *Ada* CONVERT function from Exhibit 9.1 had one optional parameter. Because it was the last parameter, we could use positional argument correspondence to call the CONVERT function.

If a function has several optional parameters, however, which could be included or omitted independently, positional correspondence cannot be used. Exhibit 9.2 shows how correspondence-by-keyword can be used to specify the correct parameter values. The rule in *Ada* is that all positional arguments in a call must precede all keyword arguments.

**Pros and Cons.** Whether or not a language should support optional parameters reduces to the usual question of values. Adding syntax and semantic mechanisms to support correspondence-by-keyword does complicate a language and its compiler. Sometimes the effect of optional parameters

**Exhibit 9.2. Correspondence-by-keyword in Ada.**

Imagine that the function `Wash_Cycle` is built into the controller of an automated washing machine. It takes parameters that describe the character of a load of laundry and executes an appropriate wash cycle. Only the first parameter, the load size, is required; default values are defined for all others. (We show a possible function header and omit the function body.) Assume that `cloth_type`, `soil_type`, and `error_type` are previously defined enumerated types.

```
FUNCTION Wash_Cycle
(   Weight:  IN integer;
    Fabric:  IN clothtype := cottonknit;
    Soil_level:  IN soiltype := average;
    Dark:    IN Boolean := False;
    Delicate:  IN Boolean := False;
) RETURN error_type IS ...
```

Following are several calls on `Wash_Cycle` that would process different kinds of loads. Note the mixed use of positional and keyword correspondence.

```
Result:= Wash_Cycle(10);           -- A large load of T-shirts.
Result:= Wash_Cycle(4, denim, Dark => True);
                                   -- A small load of jeans.
Result:= Wash_Cycle(8, Soil_level => filthy, );
                                   -- Lots of dirty socks.
Result:= Wash_Cycle(Weight => 2, Delicate => True, Fabric => wool);
                                   -- Two winter sweaters.
```

can be achieved in some other way, without adding special syntax or semantics. For example, the functional languages support higher-level functions (functions that can return functions as their result). In these languages, a *closure* is the function that results from binding some (or perhaps all) of the arguments of another function. A function and a set of closures made from that function are like an Ada function with default values for some parameters.

Using optional parameters has a mixed effect on semantic validity. When a programmer omits an optional parameter, the code is less explicit and more concise. This may or may not be a wise trade-off; explicit code is usually easier to debug and easier to modify. Being forced to write out all the arguments every time prevents a programmer from accidentally omitting one that was needed. On the other hand, if a function has a dozen parameters that are usually called with the same values, writing them all out every time hides the variable parts of the program and makes it harder to comprehend.



---

**Exhibit 9.5. Meaningless calls on a variable-argument function in FORTH.**

```

32 5      3 SumN ( Consumes one more parameter than was supplied, )
              ( causing immediate or eventual stack underflow. )

21 4 55 62 3 SumN ( Fails to consume a parameter, leaving garbage )
                  ( on the stack. )

```

---

Too many arguments are supplied in the second meaningless call. This leaves garbage (the extra argument) on the stack. This garbage may or may not interfere with future program operation and can lead to a run-time error. In any case, leaving garbage lying around is undesirable.

Note that FORTH is a postfix language and has no syntactic markers for the beginning and end of a parameter list. Moreover, there is no way to tell how many parameters will be used by a function without executing it on specific data. It is impossible, therefore, for a FORTH translator to help the programmer achieve semantic validity by checking that she or he has called his functions in meaningful ways.

FORTH's stack operations are extremely low-level. Higher-level languages do not normally let the programmer manipulate the translator's internal data structures because the semantics of this kind of operation are unclear and unpredictable at compile time. If the number of parameters actually supplied is fewer than the number needed, the program will crash. If too many parameters are supplied, junk will build up on the stack.

Although C functions do specify how many parameters they expect, it is possible to call a C function with a different number of parameters. In general, it would not be very useful to pass extra parameters to a function, but that is how the `printf` function works [Exhibit 9.6]. It uses the number of arguments specified by the information in its first argument (which is the format specification) to determine how many more arguments to process.

---

**Exhibit 9.6. A call on a variable-argument function in C.**

The call on `printf` below has three arguments—a format string and two numeric items. The format, in quotes, contains two `%` fields; each one directs that an integer is to be converted to ASCII and written to the output stream.

```
printf ("The sum is %d and the average is %d\n", sum, sum/count);
```

This call will execute correctly because the number (and types) of conversions specified in the format matches the rest of the argument list. The format string is not actually checked at all at compile time. Rather, it is passed on to the run-time system to be interpreted when `printf` is called. A mismatch in number or type between the format fields and the actual arguments will cause a run-time error or simply produce garbage.

---

C will translate programs containing calls with missing or extra parameters, and such programs will behave much like similar programs in FORTH. All is well if the program actually uses the number of parameters that are supplied. If *too few* are supplied, the program will either access garbage, causing unpredictable behavior, or it will stop with a stack underflow.

Supplying *too many* parameters will not cause C to malfunction. In current C implementations, parameters are removed from the stack by the calling program, not by the subroutine. This produces a longer calling sequence but avoids creating a serious problem by passing too many parameters. Excess parameters will sit on the stack unused until the subroutine exits, but then they will be removed. Junk will not pile up on the stack.

**Pros and Cons.** The examples cited previously help us answer the question, “To what extent is it good and/or bad to permit argument lists of indefinite length?”

Putting syntactic markers around parameter lists in a program makes them easier for humans to work with, and thus is good. Once these markers are required by a language, it is easy for a translator to check that the right number of parameters is used in every call. Doing so prevents the programmer from writing nonsense that will cause programs to malfunction or crash. Thus a parameter checking mechanism in the translator can be a powerful aid to writing correct programs.

On the other hand, there is a real cost involved. Once in a while it is very useful to write a function, such as `printf`, that accepts a variable number of parameters. This can be done by the programmer in C but not in a language such as Pascal that requires well-formed function calls. The Pascal I/O procedures do accept a variable number of parameters, but they are predefined and no functions like them can be defined by the programmer.

Whether or not a restriction is severe depends partly on whether the useful things it prohibits can be achieved some other way. For example, a function in a list-oriented language might accomplish the same end more simply by taking a single argument that is a variable-length list.

Thus the question of whether to require function calls to be well-formed can be reduced to making a set of value judgements about the relative importance of (1) semantic validity and (2) unrestricted flexibility. The answer also depends on what other mechanisms are included in the language.

Most language designers placed a higher value on communicating and preserving semantic intent than the designers of FORTH and C, who valued flexibility more highly. Neither of these languages enforces the very helpful restriction that a function must be called with a semantically meaningful number of parameters. Both are considered to be systems programming languages, and both provide nearly full access to the machine hardware. Both permit programs to be written that can crash easily and in spectacular ways.

## 9.2 What Does an Argument Mean?

Chapter 8, Section 8.4.1, discusses the order in which arguments are evaluated. Here we examine how an argument is passed to a function and interpreted within the function.

---

**Exhibit 9.7. A procedure using call-by-value in Pascal.**

```

PROCEDURE test1 (J2, A2: integer; P2:list);
BEGIN
    writeln (J2, A2, P2↑.value);
    J2 := J2 + 1;
    P2 := P2↑.next;
    writeln (J2, A2, P2↑.value)
END;

```

---

ALGOL-60, the first formally defined higher-level language, was defined with inside-out expression evaluation and both call-by-value and call-by-name parameter passing mechanisms. Since then, other call mechanisms have been devised: call-by-need, call-by-reference, call-by-value-and-return, and call-by-pointer. The syntax and the semantics of these call mechanisms differ in important ways, which we explore in this section.

**9.2.1 Call-by-Value**

Call-by-value is the simplest and cleanest parameter passing mechanism, both in terms of its semantics and its implementation. In call-by-value, an expression written as an actual parameter is evaluated before beginning the execution of a subprogram. The resulting value is written into the stack area reserved for the subprogram and bound to the corresponding formal parameter name within the subprogram. Exhibit 9.7 shows a Pascal procedure named `test1` which has call-by-value parameters. Exhibit 9.8 shows the relevant storage allocated for both the calling program and for `test1` at the beginning of execution of the call:

```
test1( J1, A1[J1], P1↑.next );
```

Exhibit 9.9 shows the result of executing `test1`, just after the procedure return and after deallocation of the stack frame for `test1`. Two lines of output are produced by this call:

```

1 30 %
2 30 $

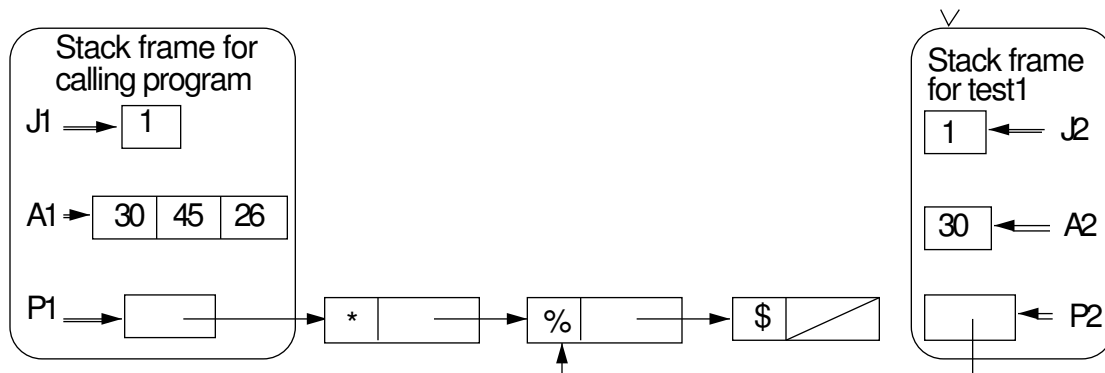
```

Assigning a value to the dummy parameter affects only the stack storage for the subroutine; it does not change anything in the storage area belonging to the calling program. Note, in Exhibit 9.9, that nothing in the stack frame of the calling program has been modified. The call-by-value mechanism is, thus, a powerful tool for limiting unintended side effects and making a program modular. Call-by-value guarantees that no subroutine can “mess up” the values of its nonpointer arguments in the calling context. For this reason, it is the most useful of the mechanisms for passing parameters and should be used wherever it can do the job.

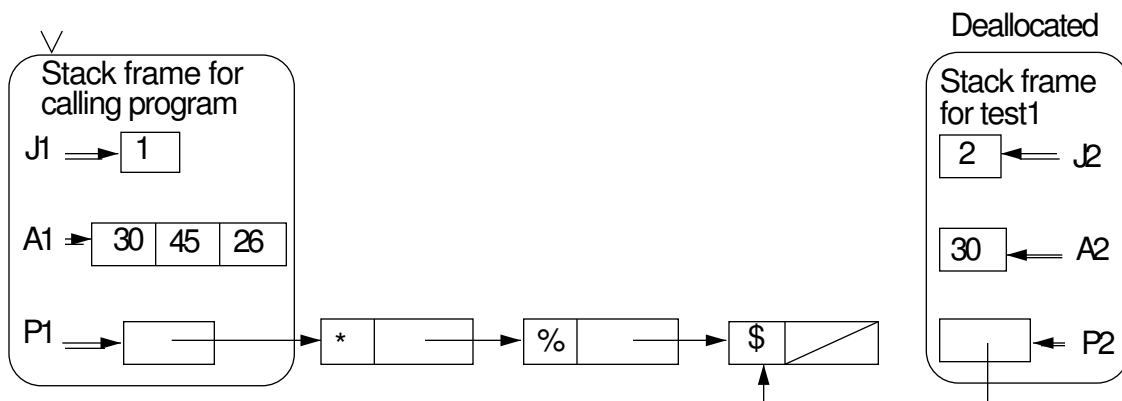
Passing a pointer, by value, to a subroutine will not permit the pointer itself to be changed, but it permits modification of storage owned by the calling program and *accessible through the pointer*.

**Exhibit 9.8. Result of parameter binding by value in Pascal.**

Double arrows represent bindings; plain arrows represent pointers. The “v” symbol marks the current stack frame.



**Exhibit 9.9. Result of execution with value parameters.**



In Exhibit 9.7 it would have been possible to modify contents of the list by assigning a new value to `P2↑.value` or `P2↑.next`. This is demonstrated in Section 9.2.6. A pointer argument, therefore, partially breaches the protection inherent in call-by-value.

### 9.2.2 Call-by-Name

In the vocabulary of programming languages, lambda calculus uses a *call-by-name* parameter interpretation rule. This is defined formally in Exhibit 8.10. Briefly put, each occurrence of a call-by-name formal parameter in an expression is to be replaced by the *entire actual argument expression* written in the function application or call.

Two function passing mechanisms were implemented in ALGOL-60 because it was recognized that call-by-value parameters do not permit information to be passed back from a function to the calling program.<sup>1</sup> While a single value may be returned as the value of a function, this is often not adequate. Many common applications, such as the `swap` routine, require passing back more than one datum.

The common swap subroutine takes two arguments, both variables, and uses a temporary storage object to exchange their values. Since the values of two variables are changed, this cannot be done with a function return value. What is actually necessary is to pass the two storage objects (not just the values in those objects) into the subroutine. The values in these objects can then be interchanged. We can pass a storage object by passing its name, its address, or a pointer pointing to it.

ALGOL-60 was developed at the same time as LISP, when lambda calculus was strongly influencing programming language design. This was before reference parameter binding was well understood and accepted, and before the concept of a storage object was well understood. It was not surprising, then, that the ALGOL-60 designers included call-by-name as a means of implementing `swap` and similar procedures.

In a call-by-name system, entire formulas are passed as arguments, to be evaluated within the called function, but using the symbols defined in the calling program. Thus one could pass a *variable name* to a subroutine, and inside the subroutine it would evaluate to the variable's address and provide access to that variable's storage object.

Symbols in an argument expression are evaluated *in the context of the calling program*. This means that if a symbol,  $S$ , occurs in an argument expression and is redefined as a local symbol in the subprogram, the global meaning (not the local meaning) is used in evaluating the argument. To implement this rule, name conflicts must be eliminated. This requires lambda-calculus-like renaming [Exhibit 8.10], which is somewhat hard to implement.<sup>2</sup>

Call-by-name semantics has two more nasty properties. Suppose that the definition of a function,  $F$ , contains three references to the parameter  $P$ , and that we call  $F$  with an expression,  $E$ , as the actual parameter corresponding to  $P$ . If we use call-by-value or call-by-reference (defined in Section 9.2.3), then  $E$  will be evaluated once, before evaluating the function, and its value will

---

<sup>1</sup>This is true unless the language supports pointer types. ALGOL-60 did not have pointers.

<sup>2</sup>The difficulty of translating call-by-name contributed to the unpopularity of ALGOL-60 in this country.

**Exhibit 9.10. Parameter binding by name.**

In our imaginary Pascal-like language, only the procedure header would differ from the version in Exhibit 9.7. The header for the call-by-name version of `test` would be:

```
PROCEDURE test2 (NAME J2, A2: integer; NAME P2:list);
```

Execution of the procedure call

```
test2( J1, A1[J1], P1↑.next );
```

is traced in Exhibits 9.11 and 9.12.

be bound to  $P$ . When  $F$  is evaluated, that single value will be used three times. But if we use call-by-name, then the original expression,  $E$ , will be evaluated every time the code of  $F$  refers to  $P$ . In this example,  $E$  will be evaluated three times. Of course, this is not very efficient, particularly if  $E$  is a large expression.

Worse yet, if the process of evaluating  $E$  has a side effect (such as producing output or changing the value of a global variable), the side effect will happen more than once, and the results of the first time might affect the value of the expression the other times or might produce extra output. Thus call-by-name on a parameter with a side effect is semantically messy as well as inefficient and hard to implement.

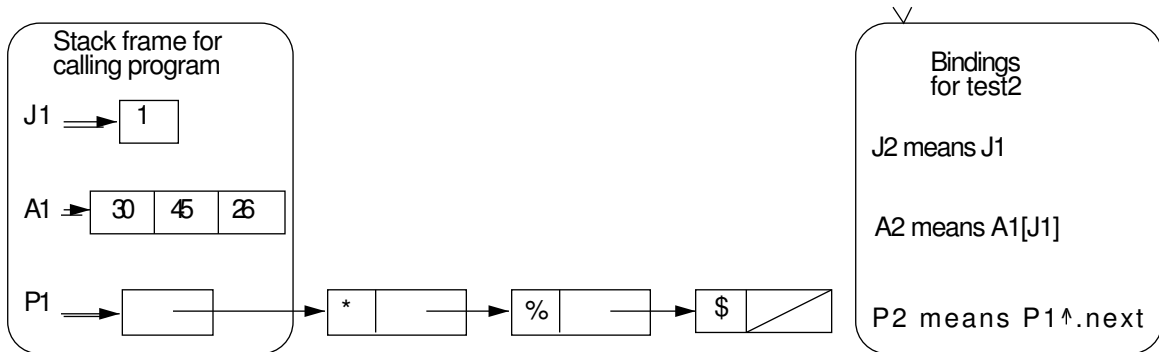
Pascal does not support parameter binding by name. In order to produce an example of binding by name that can be compared to the binding examples in Exhibits 9.7 and 9.13 we must use an imaginary extension of Pascal that includes the keyword `NAME`, placed before a dummy parameter, to indicate that the parameter is to be passed by name. Exhibit 9.10 shows a procedure, `test2`, written in this imaginary language. Exhibit 9.11 shows the bindings created for `test2` and the relevant storage allocated for the calling program, just after parameter binding and before procedure execution.

The result, just after the procedure return, of executing `test2` is shown in Exhibit 9.12. The bindings for the subroutine have been undone. Note that, unlike `test1`, `test2` modifies the linked list—it unlinks one cell. The value of `J1` is also changed. The following two lines of output are produced by this call. The second line differs from the call-by-value results because the argument expression `A1[J1]` was evaluated a second time after the value of `J1` had been changed.

```
1 30 %
2 45 $
```

Call-by-name proved to be a mistake and a headache. It was awkward to implement, inefficient to execute, and more general than necessary to implement routines such as “swap”. Since the mid-1960s a better way (call-by-reference) has been used to implement the desired semantics. Ordinary call-by-name has not been incorporated in a language since ALGOL-60, as a primary function call mechanism. Macro languages, such as the C preprocessor, use something akin to call-by-name to interpret parameters when a macro definition is expanded. Argument strings are substituted bodily for occurrences of the parameters. This can result in multiple copies of the argument string. If the

Exhibit 9.11. Result of parameter binding by name.



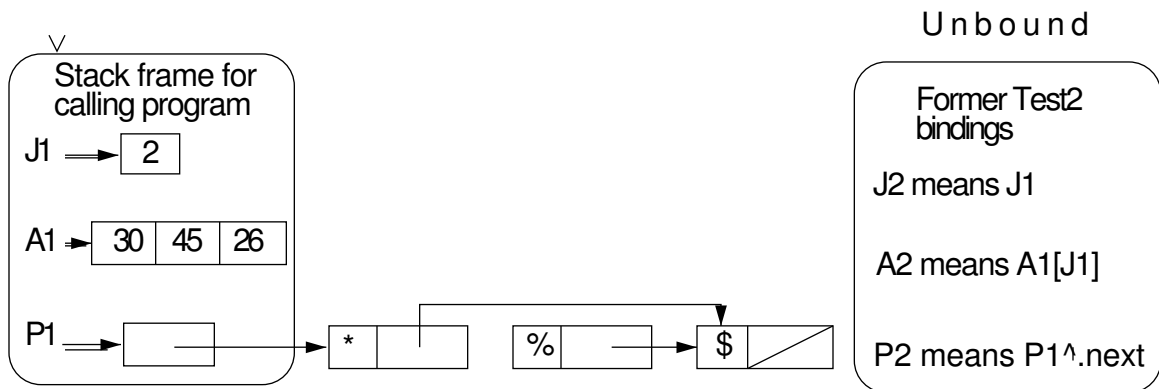
argument string is an expression, it will be compiled several times and executed several times.

### 9.2.3 Call-by-Reference

The call-by-reference parameter passing mechanism is also called *call-by-address* and, in Pascal, *VAR parameter passing*. A reference parameter is used to pass a storage object to a subprogram. An actual argument must be a variable name or an expression whose value is an address.

To implement a reference parameter, the compiler allocates enough storage for a pointer in the subprogram's stack area and binds the dummy parameter name to this stack location. A pointer to the actual parameter is stored in this space. During procedure execution, the dummy parameter name is indirectly bound to the argument's storage object (which belongs to the calling program).

Exhibit 9.12. Result of execution with by-name binding.



---

**Exhibit 9.13. A procedure using call-by-reference in Pascal.**

Only the procedure header differs from the version in Exhibit 9.7. The keyword `VAR` indicates call-by-reference.

```
PROCEDURE test3 (VAR J2, A2: integer; VAR P2:list);
```

Execution of the following call on `test3` is traced in Exhibits 9.14 and 9.15:

```
test3( J1, A1[J1], P1↑.next );
```

---

The value in this storage object may be changed by assigning a value to the dummy name.

When the attribute `VAR` is used to declare a dummy parameter name in `Pascal`, the parameter is generally translated as call-by-reference.<sup>3</sup> Exhibit 9.13 shows the function header for a by-reference version of the `test` procedure. Exhibit 9.14 shows the relevant storage allocated for both the calling program and for `test3`, after parameter binding but before procedure execution.

In this implementation the stack contents are the same as if a pointer argument had been passed by-value. But the code that the programmer writes in a subroutine is different because the pointer which implements the reference parameter is a binding, not an ordinary pointer.

A reference parameter binding is a true binding; it is “transparent” to the programmer. When using call-by-reference, the translator *automatically dereferences every occurrence of a by-reference parameter*. A naive user does not even realize that the binding pointer exists. In the stack diagram in Exhibit 9.14, this extra automatic dereference is indicated by the double binding arrows from the slots allocated for the parameters to the actual arguments. Thus we can say that the reference parameter is *indirectly bound* to the argument. As with any binding, the programmer cannot change the binding by executing an assignment.

The results of call-by-reference differ from both call-by-value and call-by-name. Exhibit 9.15 shows the stack just after the procedure return, and after deallocation of the stack frame for `test3`. Note that two things have been modified in the storage diagram: the value of `J1` and a cell in the linked list. These changes are like the ones produced by `test2`, using call-by-name. But the output produced differs from the `test2` output, and is like the `test1` output:

```
1 30 %
2 30 $
```

All parameters are passed by reference in `FORTRAN`, as are all array parameters in `C`. Call-by-reference is optional in `Pascal` and is used to implement output parameters in `Ada`.

In languages that have both call-by-value and call-by-reference, the former is the preferred parameter mechanism for most purposes because it provides greater protection for the calling program and makes the subprogram a more thoroughly isolated module. Call-by-reference is used in two situations: when the result of the subprogram must be returned through the parameter list and when copying the entire actual argument would be intolerably inefficient (e.g., when it is a

---

<sup>3</sup>Call-by-value-and-return is the other possible implementation for a `VAR` parameter. See Section 9.2.5.

Exhibit 9.14. VAR parameter binding in Pascal.

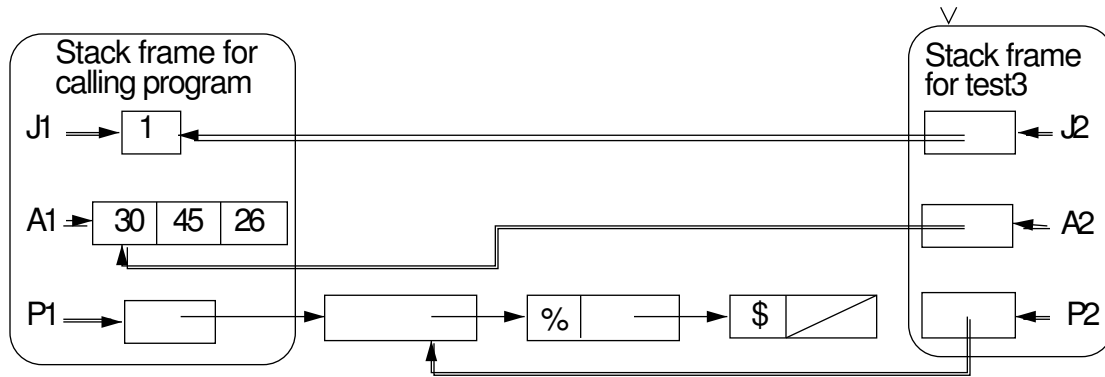
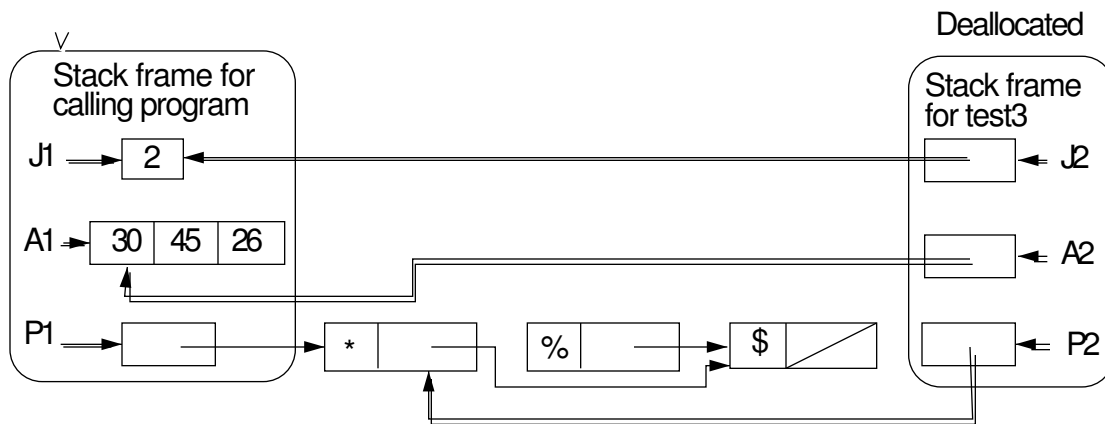


Exhibit 9.15. Result of execution with a VAR parameter.



long array).

#### 9.2.4 Call-by-Return

In practice, Pascal programmers use the call-by-reference mechanism for two reasons:

1. So that a value may be returned.
2. To avoid consuming the execution time and stack space necessary to pass a very large argument by value.

Using a `VAR` parameter for the first reason is semantically sound and in keeping with the intent of the language designer. Using a `VAR` parameter for the second reason is not semantically sound, even though it may be necessary to achieve acceptable performance from a program. Call-by-value semantics is the only sound semantics for a parameter that does not carry information out of the subroutine.

This illustrates a defect in the design of Pascal; the only efficient mechanism for passing large arguments (call-by-reference) is tied to the semantics appropriate for return values. In Ada, this design defect was partially corrected by permitting the programmer to specify the desired *semantics* and letting the translator choose an appropriate *mechanism*. Thus an Ada parameter has a declared *mode*. The mode is declared to be “`in`” if the parameter carries information into the subroutine or “`out`” if it carries information back to the calling program. A two-way parameter is declared to be “`in out`”.

The mode `in` is like a value parameter except that, within the subroutine, it has the semantics of a constant and a new value cannot be assigned to it. This is more restrictive than call-by-value, since most languages permit a value parameter to be used as a local variable and receive assignments. For this reason, we will refer to this mode as *call-by-constant*. This can be implemented by using call-by-value and prohibiting the use of the parameter name on the left side of an assignment.

The Ada mode `out` is also referred to as *call-by-return*. A call-by-return parameter is write-only; it carries information *out* of the subprogram but not *into* the subprogram. Within the subprogram, an `out` parameter can be implemented by a local variable. The program is allowed to store information in this variable but not fetch information from it. When the function returns, the final value of the `out` parameter is stored in the location specified by corresponding argument. For output parameters, call-by-return is semantically cleaner than call-by-reference because access to the location in the calling program is write-only and can happen only at function return time. Call-by-return is therefore preferred.

The Ada mode `in out` corresponds to the `VAR` parameter mechanism in Pascal. Unfortunately, the Ada standard does not fully specify the semantics that must accompany an `in out` parameter. A compiler is permitted to implement either call-by-reference or call-by-value-and-return. A program that depends on the difference between these two calling mechanisms is only partially defined. As seen in the next section, this is a truly unfortunate shortcoming in a language that was designed to support multitasking!

### 9.2.5 Call-by-Value-and-Return

We can combine call-by-value and call-by-return to achieve two-way communication between called and caller, with no accessing restrictions inside the subroutine. At first look, *call-by-value-and-return* seems more complex than call-by-reference, and it seems to offer no advantages. While this is true in an isolated program on a simple machine, it does not hold in more complex environments. Two factors, hardware complexity and concurrency, make call-by-value-and-return the preferred mechanism.

Consider a machine with a partitioned memory, or a program that is being executed on a networked system. In such hardware environments, nearby memory is fast (and therefore cheap) to access, and more distant memory costs more. A function that executes using only local storage will be more efficient than one that accesses data in another memory segment. When call-by-value is used, the data is copied from the caller's memory into the local memory. Although this copying operation takes time, much more time may be saved by avoiding out-of-segment references to the parameter. Call-by-value is simply more efficient once the argument has been copied. This is true on any partitioned architecture, even an IBM PC, although the savings may be minor. However, for a program on a network that is processing data stored at a remote location, the savings would be substantial, and the difference in response time would be important.

The difference between call-by-reference and call-by-value-and-return is very important in an application where two or more processes concurrently access the same data object. Suppose that processes P1 and P2 both have received argument ARG as a reference parameter. Using call-by-value-and-return, the answer is always one of three things:

- If the first process to begin, P1, ends before the second one, P2, starts, the final value in ARG is just the result of ordinary, sequential execution.
- If P1 starts, then P2 starts, then P1 returns, then P2 returns, the result of P1's execution is wiped out. The value in ARG is the answer from P2, as if P1 had never been called.
- If P1 starts, then P2 starts, then P2 returns, then P1 returns, the result of P2's execution is wiped out. The value in ARG is the answer from P1, as if P2 had never been called.

In any case, the final result is some value that was correctly and meaningfully computed. It may seem very undesirable to have the possibility that the result of calling a procedure could be totally wiped out. However, that situation is better than the alternative; call-by-reference can cause real trouble in a concurrent application. If both P1 and P2 read and modify ARG several times, and if their actions are interspersed, the final value of ARG can be completely unpredictable.<sup>4</sup> Worst of all, the value left in ARG could be different from any legal value that could be computed by either process alone, or both processes executed in either sequence.

The modern call-by-value-and-return semantics should be used for Ada because Ada was specifically designed to be used:

- On any machine architecture.

---

<sup>4</sup>Most textbooks on operating systems cover this problem in a chapter on concurrency.

**Exhibit 9.16. A swap routine using reference parameters in Pascal.**

The `VAR` keyword is used before the parameter name to indicate a reference parameter. Assuming that  $j$  and  $k$  are integers, an appropriate call would be: `swap1 (j, k);`

```
PROCEDURE swap1 (VAR a, b: integer);
VAR t:integer;
BEGIN
    t:=a; a:=b; b:=t
END;
```

- On networks of machines.
- With concurrent, multitasked applications.

**9.2.6 Call-by-Pointer**

Call-by-pointer is a subcase of call-by-value. The contents of the pointer variable in the calling program is an address. During parameter passing this address is copied into the corresponding pointer variable in the stack frame for the subroutine. Now two pointers, one in each stack frame, point at the same object. Because the argument is the address of a storage object belonging to the calling program, it can be used to return a value from the subroutine. Unlike a by-reference parameter, though, a pointer parameter must be explicitly dereferenced within the subprogram.

Three versions of a swap subroutine are given here in two languages. These illustrate the similarities and differences between call-by-reference and call-by-pointer. In all three, the programmer indicates that a variable address, not a value, is passed into the subroutine so that the swapped values can be passed outward. Exhibit 9.16 shows a procedure named `swap1` which uses call-by-reference. Exhibit 9.17 shows the storage allocated for execution of `swap1`.

The subprogram stack frames for call-by-pointer [Exhibit 9.21] and call-by-reference contain the same information, but that information has a different meaning because of the extra automatic dereference that accompanies call-by-reference. When call-by-pointer is used to implement the same process, the programmer must write an explicit dereference symbol each time the pointer parameter is used, as shown in Exhibits 9.18 and 9.20.

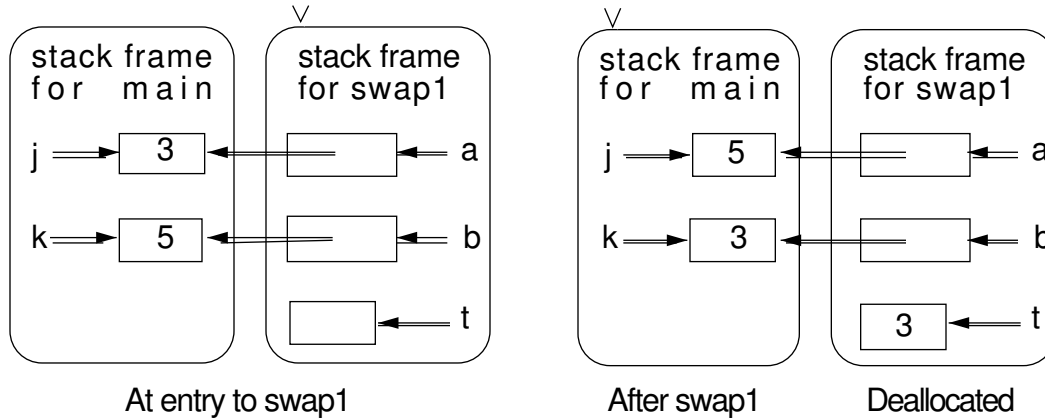
Contrast Exhibit 9.17 to Exhibit 9.21. Note the double binding arrows in the former, and the single pointer arrows in the latter. These semantic differences account for and correspond to the absence of explicit dereference symbols in Exhibit 9.16 and their presence (“↑” in Pascal and “\*” in C) in Exhibits 9.18 and 9.20.

Moreover, although a by-reference binding cannot be changed, the address stored in the pointer parameter can be changed by an assignment statement, as shown on the last line of Exhibit 9.22.

Use of call-by-pointer is severely restricted in Pascal. Since Pascal pointers can never point at objects on the stack, one can only use call-by-pointer to process dynamically allocated storage

**Exhibit 9.17. The implementation of reference parameters in Pascal.**

Stack diagrams for `swap1` are shown here, just before the code in `swap1` is executed and after return from the subroutine. The arrowhead at the top of the stack frame indicates the current stack frame.

**Exhibit 9.18. A swap routine using pointer parameters in Pascal.**

The pointer parameters are passed by value and explicitly dereferenced within the subroutine.

```

TYPE int_ptr = ↑ integer;
VAR jp, kp: int_ptr;

PROCEDURE swap2 (a, b: int_ptr);
VAR t: integer;
BEGIN
    t := a↑ ; a↑ := b↑ ; b↑ := t
END;

```

Assume that the following code has been executed. The last line is an appropriate call on `swap2`. Execution of this call is traced in Exhibit 9.19.

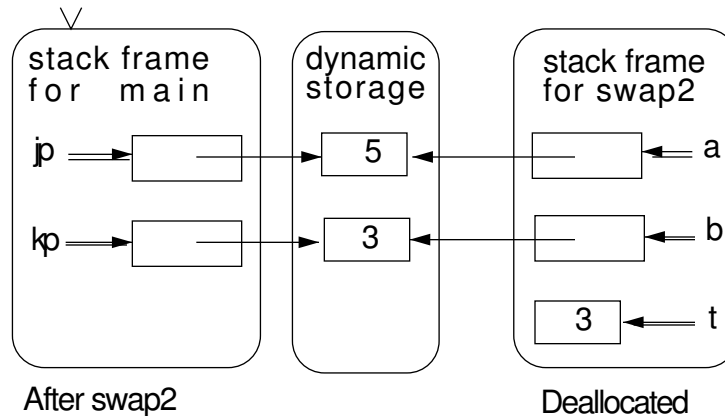
```

NEW(jp);  jp↑ := 3;
NEW(kp);  kp↑ := 5;
swap2 (jp, kp);

```

**Exhibit 9.19. Storage for swap2.**

This diagram shows the storage allocated for `swap2` and the calling program, just after exit from the call on `swap2`. Note that the values of pointers `jp` and `kp` were copied into `swap2`'s stack frame, and provide access to objects that were dynamically allocated by the calling program.



objects. This accounts for the difference between Exhibit 9.19 and Exhibit 9.24. C does not restrict use of pointers; the programmer can use the “&” operator to get the address of (a pointer to) any variable. Thus call-by-pointer can be used in C to pass an argument allocated in either the stack or the heap.

Two versions of the swap routine can be written in Pascal, with the use of the second restricted to heap-allocated objects. In contrast, only one version is possible in C because C does not support call-by-reference. In its place, call-by-value is used to pass the value of a pointer (an address). We call this parameter passing method *call-by-pointer* [Exhibit 9.20]. In this version of the swap routine, the type of the dummy parameters is “int \*” (pointer-to-integer). The parameters must be explicitly dereferenced (using the \* symbol), just as `↑` is used in the Pascal `swap2` routine.

Two distinct kinds of calls on this one function are possible. In the first, shown in Exhibit 9.21, we use the “&” operator to pass the addresses of the two integers whose values are to be swapped.

**Exhibit 9.20. Pointer parameters in C.**

```
void swap3 (int *a, int *b)
{ int t;
  t = *a; *a = *b; *b = t;
}
```

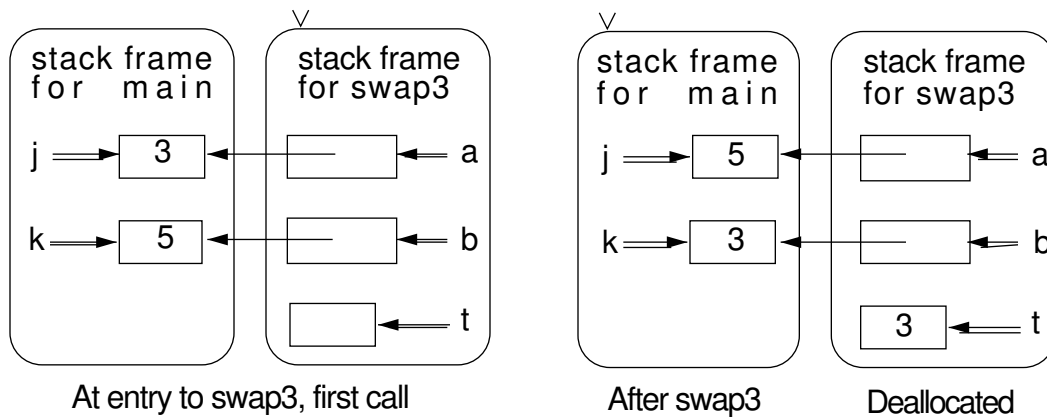
**Exhibit 9.21. Call-by-pointer using & in C.**

```

main(){
    int j=3, k=5;           /* Declare and initialize two integers. */
    swap3 (&j, &k);
}

```

The stack diagram on the left shows execution of this call just after creation of the stack frame for `swap3`. The diagram on the right shows storage after exit from `swap3`.



This is not possible in Pascal, as Pascal does not have an “&” (address of) operator.

Note that only the values in the variables `j` and `k` are swapped; the pointers `a` and `b` point to the same variables throughout. A reader who is confused by this should note, in Exhibit 9.22, that an assignment statement that changes a pointer is quite different from an assignment that changes a value.

The second call, shown in Exhibit 9.23, corresponds to the call on `swap2` in Pascal. In the main program, we initialize pointers to the two integer variables and pass those pointers (by value, of course) to `swap3`. The result is the same as the first call: the stack frame for `swap3` contains two pointers, initialized at block entry to the addresses of `j` and `k`. The only difference between these

**Exhibit 9.22. Assignments using pointers in C.**

```

int j=3, k=5;           /* j and k are integer variables. */
int *p= &j;           /* Make the pointer p point at j. */
*p = k;                /* Variable j now contains a 5. */
p = &k;                /* Make p point at k instead of j. */

```

**Exhibit 9.23. Call-by-pointer using pointer variables in C.**

```

main(){
    int j=3, k=5;           /* Declare and initialize two integers. */
    int *jp = &j, *kp = &k; /* Make pointers point at the integers. */
    swap3 (jp, kp);
}

```

The stack diagram in Exhibit 9.24 shows execution of this call just after creation of the stack frame for `swap3`.

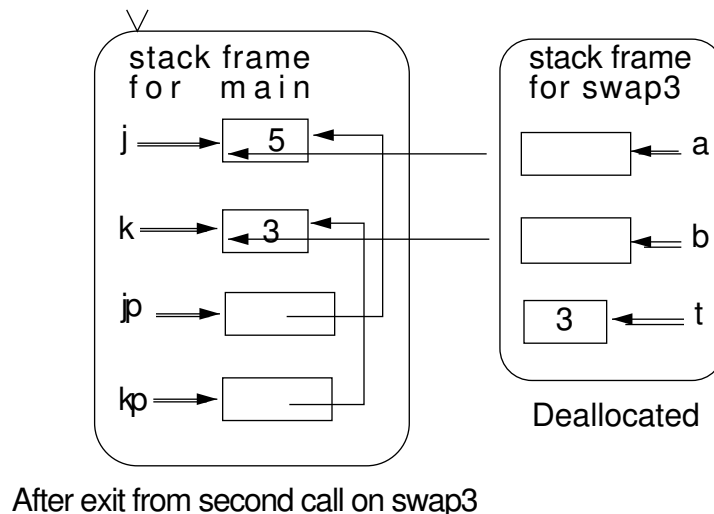
two kinds of call-by-pointer is the presence or absence of the pointer variables in `main`. If these pointers are useful for other reasons, the second call would be stylistically better. Otherwise the first call should be used.

There are two important practical differences between reference parameters and pointer parameters. The first has already been noted: pointers must be explicitly dereferenced. Experience has shown that a common error is using one too few or one too many dereference symbols. The call-by-reference mechanism is inherently less error prone.

Second, once you are permitted to point freely at items allocated on the stack, it is possible to create dangling pointers. This is why the “address of” operator was omitted from Pascal. Wirth’s

**Exhibit 9.24. Stack diagram for call-by-pointer.**

This diagram shows execution of the code from Exhibit 9.23.



---

**Exhibit 9.25. Creation of two dangling pointers in C.**

```

int * dangle (int ** ppp) /* The parameter is the address of a pointer. */
{
    int p=5;
    int m=21;

    *ppp = &p;          /* Dereference ppp to get the pointer whose address
                        was passed. Make it point at local &p. */

    return &m;          /* A second dangling pointer. */ }

main()
{
    int k = 17;
    int *pm, *pk = &k; /* pk is a pointer pointing at k. */
    pm = dangle(&pk);  /* Upon return, pk and pm point at */
}                      /* deallocated spaces. */

```

Exhibit 9.26 shows storage before and after returning from the call on `dangle`.

---

intent was that Pascal should prevent the programmer from making many semantic errors by providing only semantically “safe” operators.

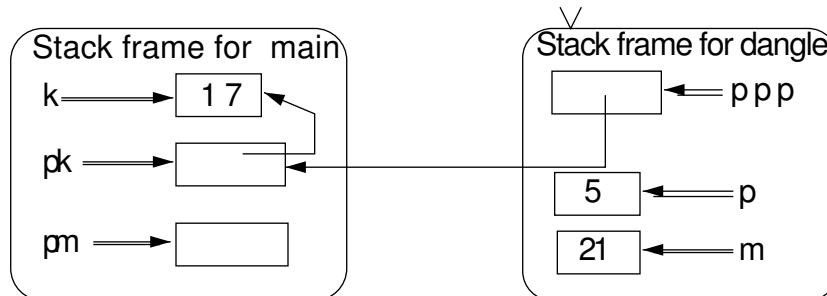
There are two simple ways to create a dangling pointer in a language such as C that permits an “address of” operation to be applied to any object. First, it can be done by passing an argument which is *the address of a pointer to an object*. Assume `pk` is a pointer to an object. Then `&pk` is the address of a pointer. By passing `&pk` to a subprogram we permit the subprogram to change the value of `pk`. Inside the subprogram, `pk` can be set to point to a local variable. Second, a dangling reference may be created by returning the address of a local variable from a function. Upon returning from the function in either case, the local variable will be deallocated and the calling program will point to a storage object that no longer exists. This is illustrated in Exhibits 9.25 and 9.26.

### 9.3 Higher-Order Functions

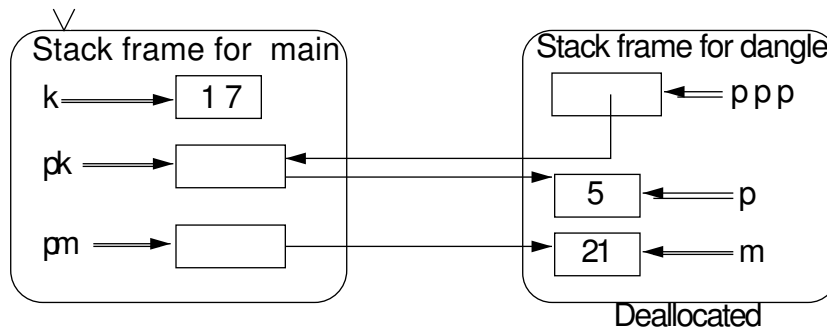
A *higher-order function* is one that takes a function as an argument or returns a function as its result. Lambda calculus has higher-order functions—it makes no distinction between a functional object and a nonfunctional object. The pure functional languages, which are modeled after lambda calculus, give full support to higher-order functions, and this is the basis of much of their power. Other languages give partial support or no support, because of the difficulties involved. In this section we look at some of the applications of higher-order functions and consider some of the difficulties involved in their implementation.

**Exhibit 9.26. Creating a dangling pointer.**

The stack before execution of `dangle`, showing parameter bindings and local allocation.



The stack, after return from `dangle`, showing results of execution.

**9.3.1 Functional Arguments**

It is not too difficult and it is very useful to support functions as arguments to other functions. Let us consider two of the more common applications of functional arguments, flexible code and mapping functions.

**Flexible Code.** Sometimes a complex general process can have several variants. For example, consider a spreadsheet that consists of one line of data per student and one column for each assignment or exam, in which the student's grade is recorded. In the process of assigning grades, an instructor will sort that data many times in many ways. Sometimes the sort will be on an alphabetical column, sometimes on a numeric column. Sometimes ascending order is needed, sometimes descending order. A grading program (or spreadsheet used to do the grading) must contain a general sort procedure, which takes the comparison function as its argument, along with the function to select the correct column as the sort key. For this application, four different comparison functions might be given, numeric "`<`", alphabetic "`<`", numeric "`>`", or alphabetic "`>`". By using functional arguments, one sort routine can handle all four jobs.

**Exhibit 9.27. Mapping functions in APL.**

Code	Operation	Results
<code>A ← 1 1 0 1</code>		Bind A to an array of four Boolean values.
<code>+/ A</code>	plus reduce	3 (Count true values.)
<code>+ \A</code>	plus scan	1 2 2 3
<code>∨/ A</code>	logical-or reduce	1 (At least one true exists.)
<code>∧A</code>	logical-and reduce	0 (Some value is false.)
<code>∧ \A</code>	logical-and scan	1 1 0 0 (Find first false.)

Consider another example— program for finding the roots of an arbitrary mathematical function. One way to program this is to write and compile the code to calculate the function (or many functions) and pass that function as an argument to the general root-finding procedure. This permits the root-finder to be kept in a library and used for many applications without modification. The general graphing facilities of packages such as Lotus must also be able to take an arbitrary function as its argument and operate on it.

**Mapping Functions.** Frequently, we have a monadic or dyadic operation that we wish to apply to a list or array of data objects. A mapping function defines a particular pattern for extending the operation to process multiple data values instead of one or two. It takes two arguments, a function and a list of data values, and applies the function to the data values in the defined pattern. Examples of mapping functions in APL and LISP are given in Exhibits 9.27 and 9.28.

The APL “reduce” (“/”) and “scan” (“\”) operations both take a dyadic functional argument and an array argument. Reduce computes a “running operation”. The result is the same as if the functional argument were written between each pair of values from the data array, and the entire expression were then evaluated. Thus if the function argument is “+”, reduce totals the array. The scan operation works similarly, except that its answer is the same length as the input array and contains all the partial “sums”. These operations are fundamental to giving APL its power and are also essential operations in modern parallel algorithms.

There are several mapping functions in LISP, that iterate a functional argument over a list argument or arguments. In all cases, the number of list arguments must match the number of parameters in the functional argument; a monadic function needs one list, a dyadic function needs two. Let us use the function `mapcar` as an example; `mapcar` goes down the argument lists, selecting the next value from each list argument at each step and applying the functional argument to those values. The result becomes the next item in the result list. A null argument produces a null result, and a list of any other length maps onto another, transformed list of equal length. Iteration stops when the end of the shortest list-argument is reached. If the functional argument returns a list, the final output is a list of lists [Exhibit 9.28].

The function `mapcdr` is similar to `mapcar`, except that the parameter of the functional argument must be of type list, and at each step, the function is applied to the entire list, and then the head

**Exhibit 9.28. Some applications of mapcar in LISP.**

Assume *x* is the list (4 9 16 25), *y* is (1 2 3 4), *z* is NIL, and *w* is ((3 4 5) (2 1) (7 9 3 6)), which is a list of lists. The quote mark before the function name causes the actual, executable function (not the result of evaluating it) to be sent to `mapcar`.

Expression	Comments	List Returned
<code>(mapcar '+1 x)</code>	Increment each element.	(5 10 17 26)
<code>(mapcar '+ x y)</code>	Add corresponding elements.	(5 11 19 29)
<code>(mapcar '+1 z)</code>	The empty list is no problem.	NIL
<code>(mapcar '(lambda (a b)           (- (* 3 a) b) ))           x y)</code>	Apply the function ((3*a)-b) to corresponding elements of lists <i>x</i> and <i>y</i> .	(11 25 45 71)
<code>(mapcdr 'caar w)</code>	The first item of each sublist.	

of the list is discarded. (The remaining list argument is one value shorter after each iteration.) The result of each step is appended to the output list.

**Implementation.** A functional argument can be passed to another function easily and efficiently by passing a pointer to the function, not the function itself. The only implementation problem concerns type checking—the type of the functional argument, that is, the types of all its arguments and the type of its result, must be known before code can be compiled to call that function. This accounts for one major difference between the functional languages and Pascal or ANSI C. All support functional parameters, but the functional languages are interpreted and do not require the declaration of the full type of a functional argument since its type can be deduced, when necessary, from the function’s definition.

A short Pascal function with a functional argument is shown in Exhibit 9.29. It is a very limited version of APL’s reduce operator. The capabilities of this code are much more restricted than the “reduce” operation in APL because Pascal is a strongly typed, compiled language, and APL is not.<sup>5</sup> Note that the functional parameter, *f*, is declared with a full header, including dummy parameter names *x* and *y* which are meaningless and never used.

### 9.3.2 Currying

In functional languages,<sup>6</sup> every function of two or more arguments is considered to be a higher-order function of one argument that returns another function. This way of looking at functions is called *currying*<sup>7</sup> The returned function represents a function of one fewer argument which is the same as

<sup>5</sup>A similar version of “reduce” with some type flexibility is given in C in Exhibit 16.4.

<sup>6</sup>We will use Miranda syntax, because it is simple and elegant, to illustrate the properties common to all functional languages.

<sup>7</sup>Named after Haskell Curry, an early logician who used this device extensively in his work on semantics.

**Exhibit 9.29. A Pascal version of “reduce”.**

This function will apply any dyadic integer function to reduce an array of ten integers.

```

Type row_type = array[0..9] of integer;
Function Reduce( Function f(x:integer, y:integer):integer;
                ar: row_type):integer;
Var sum, k: integer;
Begin
    sum := ar[0];
    For k=1 to 9 do
        sum := f(sum, ar[k]);
    Reduce := sum;
End;

```

the original function in which the first argument has been fixed to the given value. This pattern is repeated until all but one of the arguments have been used, and the final function returns a data-object.

An example should help make this clear. Consider the Miranda function `pythag a b = sqrt (a * a + b * b)`, which uses the Pythagorean theorem to compute the length of the hypotenuse of a right triangle whose base and height have lengths  $a$  and  $b$  respectively. `Pythag` is considered by Miranda to be a function that takes one numeric argument and returns a function. That function takes a numeric argument and returns a number. Thus the result returned by `pythag 3` is a function that finds the hypotenuse of any right triangle whose base has length 3. The following Miranda expressions all return the length of the hypotenuse ( $= 5$ ) of a right triangle with base 3 and height 4:

- `pythag 3 4`
- `(pythag 3) 4`
- `f 4` if we have previously defined `f = pythag 3`

In the last expression above, we have given a temporary name `f` to the function returned by `pythag 3`. `f` is itself a legitimate function and can be used like any other function. For example, the expression

```
(f 4) + (f 7)
```

computes the sum of the lengths of the hypotenuses of two right triangles, one with base 3 and height 4 and the other with base 3 and height 7. See if you can figure out what is computed by this Miranda expression:

```
f (f 4) where f = pythag 3
```

**Exhibit 9.30. The type of reduce in Miranda.**

<code>reduce ::</code>	The function <code>reduce</code> takes three arguments,
<code>(num → num → num )</code>	a binary operator,
<code>→ [num]</code>	a list of numbers,
<code>→ num</code>	and an identity element.
<code>→ num</code>	It returns a number.

**Notes:**

- Type `num` means number and is a primitive type in Miranda.
- A function type is denoted by use of parentheses.
- We denote the type of a list with base type `T` as “[`T`]”.

---

The notation for expressing a curried function type is:

$$\langle \text{function name} \rangle :: \langle \text{argument type} \rangle \rightarrow \langle \text{result type} \rangle$$

The type of a higher-order function is written by repeated uses of the `→` notation. The “`→`” sign is right associative, as shown below, permitting us to write the type of an ordinary numeric function without using parentheses. These two type expressions are equivalent and denote the type of `myfunc`:

```
myfunc :: num → num → num
myfunc :: num → (num → num)
```

We can use this notation to write the type of a function that takes a function as a parameter. For example, Exhibit 9.30 shows the type of a Miranda version of the `reduce` function. (The function itself is shown and discussed in Chapter 12, Exhibit 12.15.) This version of `reduce` takes a binary arithmetic operator, a list of numbers (possibly null), and an identity value, and returns a number. The identity value must be appropriate for the given operator; it is used to initialize the summing process and is returned as the value of `reduce` when the list argument is a null list.

**9.3.3 Returning Functions from Functions**

C supports functional arguments, in the form of pointers to functions, and C++ carries this further to permit the predefined operators to be named and used as arguments also. Both languages permit the programmer to return a pointer to a function as the result of a function. However, a C or C++ function cannot do one thing that distinguishes functional languages from all other languages: create a function within a function and return it as the result of the function. At first sight, this facility might seem esoteric and not very useful; it is, however, a powerful and concise

**Exhibit 9.31. Partially closing a function to make new functions.**

We define a simple function of two arguments in Miranda and produce two new functions from it by closing its first argument. Assume the symbol `x` is bound in the enclosing context.

```

plus :: num → (num → num )
plus a b = a + b

add1 :: num → num
add1 = plus 1

plusx :: num → num
plusx = plus (x * 10)

```

way to achieve several practical ends, including composition functions, closures, and infinite lists. We discuss closures here and infinite lists in Chapter 12.

**Closures**

*Closure* is an operation that can be applied to an expression. We close an expression by binding each of its free variables to the value of another expression. A closure creates a function, which is returned as the result of the closure process. The most useful and easily understood application of closure is known as *partial parameterization*. In this process we take a function of  $n$  arguments, bind one to a locally defined value, and produce a function of  $n - 1$  arguments.

For example, we can partially parameterize “+” by closing its first argument; the result is a one-argument function that adds a constant to its argument. If the closed argument is bound to 1, the result is commonly known as “+1”, or increment. Exhibit 9.31 shows the types and definitions, in Miranda, for two functions produced by closing the first argument of `plus`. The result is always a function of one argument, whose type is the same as the type of `plus`, with the first clause omitted.

**An Application of Closures.** As an example of the possible use of closures, consider the currency conversion problem. An international bank has a general currency-conversion function, `Convert`, that takes a conversion rate and an amount of currency and returns an amount in a different currency. The bank has a second program that is run each morning. It reads the current conversion rates for all currencies the bank handles, then creates two closures of the `convert` program for each rate. For example, after reading the fact that one mark = .6 dollars, two closures would be created:

- `MarksToDollars = Convert 1.67`
- `DollarsToMarks = Convert .6`

The resulting closures would be accessed through a menu offering dozens of specialized conversion routines. These could be called up during the business day, as needed.

---

**Exhibit 9.32. We can't make closures in Pascal.**

The function `MySum` calls the function `Reduce` from Exhibit 9.29 with the first argument bound to a specific operation.

```
Function MyOp(x: integer, y:integer):integer;
Begin   MyOp := abs(x) + abs(y)   End;

Function MySum(ar: row_type):integer;
Begin   MySum := Reduce( MyOp, ar)   End;
```

---

**Implementation of Closures.** Closure is a useful device for taking general, library code modules and tailoring them to the needs of a particular environment. Although it produces a new function, it does not require translating new code; the function code for the closure is just like the code of an existing function. Thus we can implement closure using a record containing bindings for symbols, together with a pointer to a function.

When we close a function with a constant value, as in the function `add1` [Exhibit 9.31], there is no question about what the meaning is or how to construct the closure. In this case the closure would contain a pointer to the function `plus` and the information that the symbol `a` in `plus` is bound to 1.

However, a problem arises when we close `plus` with an expression such as `x * 10`: what should be recorded in the closure? Should that expression be evaluated and the result stored in the closure, or should the expression itself become part of the closure? The answer is, we can still be lazy. We can safely defer evaluation of the expression until the value is needed. There is no need to evaluate the expression when we make a closure, because the result of evaluating the expression will always be the same. Each symbol used in the expression has one meaning from the time it becomes defined until the end of its lifetime; assignment doesn't exist and can't change that meaning. So we defer evaluation, and compute `x * 10` when it is used, if it is used.

The symbol `x` may have some other meaning in the surrounding program, and the closure, which is a function, might be passed around and executed in that other context. Whatever happens, though, when `plusx` is used, `x` must mean whatever it did when `plus` was closed. To make this work, the closure must contain the current binding for `x`, and the data-part of the closure must have a lifetime longer than the lifetime of the block that created the closure.

Except for the lifetime problem, the data-part of a closure is identical to the stack frame that would be created for a function activation; it contains the local context in terms of which local symbols must be interpreted. The difference between a closure and a stack frame is that stack frames have nested lifetimes and closures do not. A stack frame is deallocated before deallocating the stack frame for the block that created it, but a closure must outlive that block. For this reason, closure is a process that can only be done in functional languages or in other languages that have a special closure facility.

The student must get a clear understanding of the difference between a closure and the functions

that can be written in traditional languages. The Pascal programmer can write (and then compile) a function that calls another function with a constant parameter. For example, `MySum`, defined in Exhibit 9.32, calls the function `Reduce`, from Exhibit 9.29 with its first argument bound to a locally defined function, `MyOp`. A function such as `MySum` is not like a closure for two reasons:

- This code must be compiled before it becomes a function! The binding of `f` in `Reduce` to `MyOp` is created by the compiler, not at run time.
- The lifetime and visibility of `MySum` are limited. It can be used only within the block in which `MyOp` and `MySum` are defined, and it will die when that block dies.

Most things that can be done with closures, and with higher-order functions in general, can also be done in Pascal. To emulate a closure, the programmer would write a program with the necessary bindings in it at a global level, then compile the code. When executed, the result will work just like a closure. However, the ability to create closures during execution gives a more concise and efficient way to accomplish the job.

## Exercises

1. What is the difference between an argument and a parameter?
2. What is a well-formed function call? Explain the following statement: K&R C is a language in which function calls do not need to be well-formed. Is this an advantage or disadvantage for C programmers?
3. In Ada, a function may be called with fewer arguments than the number of parameters in the function definition. Explain how. Why is this useful?
4. In FORTH, function definitions do not include parameter declarations. This permits the programmer to write a function that uses different numbers of arguments on each call. Explain one benefit and one drawback of this design.
5. How does call-by-value afford protection to the calling program?
6. Why is a call-by-value method of parameter passing ineffective for a swap routine that intends to swap values in the calling program?
7. Why does Pascal refuse to bind a VAR parameter to a constant?
8. Why does Pascal need VAR parameters, but C functions can communicate as well using only value parameters?
9. The attributes “in”, “out”, and “in out” that are specified in Ada parameter declarations have a basic good property that Pascal cannot achieve with its declarations of VAR or value parameter passing. What is it? (Note: The fact that Ada permits optional parameters is not relevant here.)

10. Explain why a language that supports multitasking will be likely to support call-by-value-and-return rather than call-by-reference.
11. A short program is given below in Pascal with one procedure definition. Show the storage allocated for the main program and the stack frame (activation record) created for the call on the subroutine. Note that one parameter is a reference (VAR) parameter, and one is a value parameter.

```

Program ParameterDemo;
Var I: integer;
    A: array[1..3] of integer;

Procedure Strange (Var X: integer; Y: integer);
    Var temp: integer;
    Begin temp:=X; X:=X+Y; Y:=temp End;

Begin (* main program *)
    For I := 1 to 3 Do A[I] := 10-I;
    I := 2;
    Writeln ( 'Initial values ', I, A[1], A[2], A[3]);
    Strange( A[I], I );
    Writeln ( 'Final values ', I, A[1], A[2], A[3]);
End.

```

12. Use the stack frame diagram from question 11 to trace execution of this program. Write values in the storage locations, and show how the values change during execution of the program. Show the output produced.
13. Change both of the parameters of the procedure named `Strange`, in question 11, to `VAR` parameters. Answer the questions 11 and 12 again showing the storage setup, data changes, and output.
14. What is lazy evaluation? Why is it used?
15. What is call-by-pointer? Why is it a subcase of call-by-value and different from call-by-reference?
16. Explain the differences between call-by-reference (as in a Pascal `VAR` parameter) and call-by-pointer, with respect to the following:
  - a. What is written in the procedure call.
  - b. What is written in the procedure header.
  - c. How the parameter is used within the procedure body.
17. What is a dangling pointer?

18. Write a short piece of Pascal or C code that creates a dangling pointer.
19. Fill in answers to the twelve questions in the following chart. To name times, use the phrases “load time”, “block entry time”, “block exit time”, “program termination”, “any time”, or “other” (please explain other). To label accessible locations, use the phrases “anywhere”, “anywhere except where masked”, “in the declaring block”.

Type of variable	Time of creation?	Time of death?	Where is it accessible?
global variable	1.	2.	3.
static local variable	4.	5.	6.
ordinary local variable	7.	8.	9.
dynamic heap variable	10.	11.	12.

20. What is printed by the following Pascal program? Make diagrams of storage for both the main program and subroutine, showing the initial contents and changes in contents. Distinguish between VAR and value parameters in your diagram. This is trickier than tricky. You are being tested on comprehension of name scoping and VAR and value parameters.

```

PROGRAM trick (INPUT, OUTPUT);
VAR J, K, L: INTEGER;

FUNCTION F(J: INTEGER; VAR L: INTEGER):CHAR;
VAR K: INTEGER;
BEGIN
    K := L + J;
    L := K;
    J := K;
    IF K > 10 THEN F := 'Y' ELSE F := 'N'
END;

BEGIN (* main program *)
    J := 3; K := 15; L := 4;
    WRITELN( F(L, J) );
    WRITELN (J, K, L)
END.

```

21. What is a higher-order function? Explain two uses for higher-order functions.
22. Name two languages that do not support higher-order functions in any way. Explain how you found the information to give the answer.
23. Explain what a closure is.
24. C gives some support for higher-order functions: pointers to functions may be passed as parameters and returned from functions. Explain what kind of support is missing in C—that

is, what can a programmer do with higher-order functions in a functional language that C does not support?

25. Express the type of the function `search` from Chapter 8 Exhibit 8.14 in Pascal type notation, as if it were to be passed as an argument to another function.
26. Express the type of the function `search` from Chapter 8 Exhibit 8.14 in curried notation.