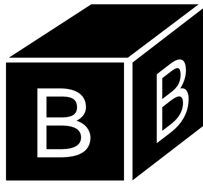


Chapter *12*

Proper Procedures

The purpose of a called procedure is to perform a task for the calling procedure. Procedure PrintTriangle in Figure 3.13 prints a single pattern and illustrates the flow of control for procedures. Each time the calling procedure calls PrintTriangle, the pattern is printed, then control is passed back to the statement after the calling statement. When a procedure has parameters, its flow of control is identical to that of procedure PrintTriangle. The computer executes the correct flow of control by saving the address of the statement that made the call. When the called procedure terminates, the computer uses the saved address to know where to pass control back to the calling procedure.

PrintTriangle in Figure 3.13 executes in response to the user clicking a button in the graphical user interface. Like all procedures that execute in response to a button click, it has no parameters. Most programs in previous chapters use procedures from other modules that have parameters. This chapter shows how to write procedures with parameters. It also shows the mechanism by which the computer saves the return address when a procedure is called.

The run-time stack

To allocate a resource is to reserve it for someone's use. For example, you may allocate $30 from your paycheck to go to dinner with a friend on the weekend. When you call a procedure, Component Pascal allocates on a stack a portion of main memory for the procedure's use. It is called a run-time stack because the allocation takes place during program execution, that is, during the time the program is running as opposed to when the program is translated.

The return address

When a procedure is called, one of the items stored on the run-time stack is the *return address*. The return address contains information about the location of the statement in the calling procedure. When the called procedure finishes its execution, control is returned back to the calling procedure. The computer uses the return address to determine which statement in the calling procedure to execute after it executes the last statement in the called procedure.

In addition to the return address, the computer allocates storage for the parameters and for the local variables in the called procedure. Because the allocation takes place at the time of the procedure call, neither the parameters nor the local variables exist before the procedure is called.

Allocation takes place on the run-time stack in the following order when you call

a proper procedure:

Allocation for a proper procedure

- Push the parameters.
- Push the return address.
- Push storage for the local variables.

We know from Chapter 6 that a stack has the LIFO property—last in, first out. We will see that this property is especially significant for allocation on the run-time stack.

Parameters called by value

Module Pbox12A in Figure 12.2 has a programmer-defined procedure, PrintLine, which the calling procedure calls twice to print two lines of asterisks. Figure 12.1 shows the commander to activate the exported procedure PrintLines and the resulting output.

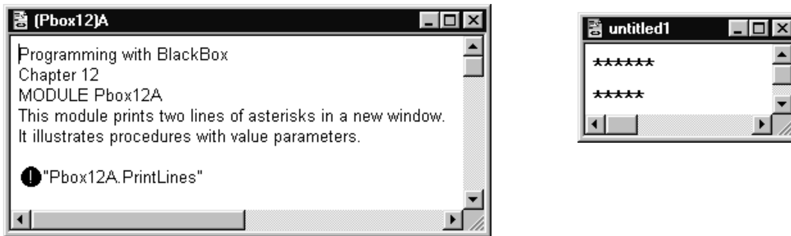


Figure 12.1
The output for the procedure of Figure 12.2.

The first statement of procedure PrintLines creates a new model md, and the second statement attaches formatter fm to it. The next statement

```
PrintLine(6, fm)
```

is the procedure call. The actual parameters are 6 and fm, and they match the formal parameters n and f. When the procedure is called, Component Pascal first allocates storage for the two parameters. It gives the value of the actual parameter 6 to the formal parameter n, and a reference to the actual parameter fm to the formal parameter f. Then, it allocates storage for the return address and puts the value of the return address, indicated by ra1 in Figure 12.2, in the storage allocated for it. Finally, it allocates storage for the local variable i. Unlike the parameters and the return address, no values are stored on the run-time stack for the local variables at the time of the procedure call.

The procedure executes the FOR statement to print the first row of six asterisks. The value of i on the run-time stack changes as the FOR statement executes. At the end of the procedure execution, the storage for n, f, the return address, and i is deallocated. Unlike a function, control does not return to the calling statement. Instead, control returns to the statement after the calling statement. In this case, that is the procedure call

```
MODULE Pbox12A;
   IMPORT TextModels, TextViews, Views, PboxMappers;

   PROCEDURE PrintLine (n: INTEGER; IN f: PboxMappers.Formatter);
   (* Inserts a line of n asterisks to a text model with formatter f *)
      VAR
         i: INTEGER;
   BEGIN
      FOR i := 1 TO n DO
         f.WriteChar("*")
      END;
      f.WriteLn
   END PrintLine;

   PROCEDURE PrintLines*;
      VAR
         md: TextModels.Model;
         vw: TextViews.View;
         fm: PboxMappers.Formatter;
   BEGIN
      md := TextModels.dir.New();
      fm.ConnectTo(md);
      PrintLine(6, fm);
      (* ra1 *)
      PrintLine(5, fm);
      (* ra2 *)
      vw := TextViews.dir.New(md);
      Views.OpenView(vw)
   END PrintLines;

END Pbox12A.
```

Figure 12.2
A procedure that prints a row of asterisks. The parameter specifies the number of asterisks in the row.

```
PrintLine(5, fm)
```

which calls the procedure again. Storage is reallocated for n, f, the return address, and i. This time the return address ra2, which is the address of

```
vw := TextViews.dir.New(md)
```

is stored on the run-time stack. The procedure prints a row of five asterisks. When procedure PrintLine terminates control returns to that statement. A new view is created and opened using the now familiar MVC paradigm.

Figure 12.3 shows the details of the allocation process on the run-time stack for Module12A. It illustrates a prominent feature of frameworks in contrast to those development environments that are based on libraries of procedures. Older development environments require that the programmer write what is known as a main program that always starts first when the application is launched. The main program calls other procedures, some of which are in the library and some of which are written by the programmer. Because BlackBox is a framework, however, there is no

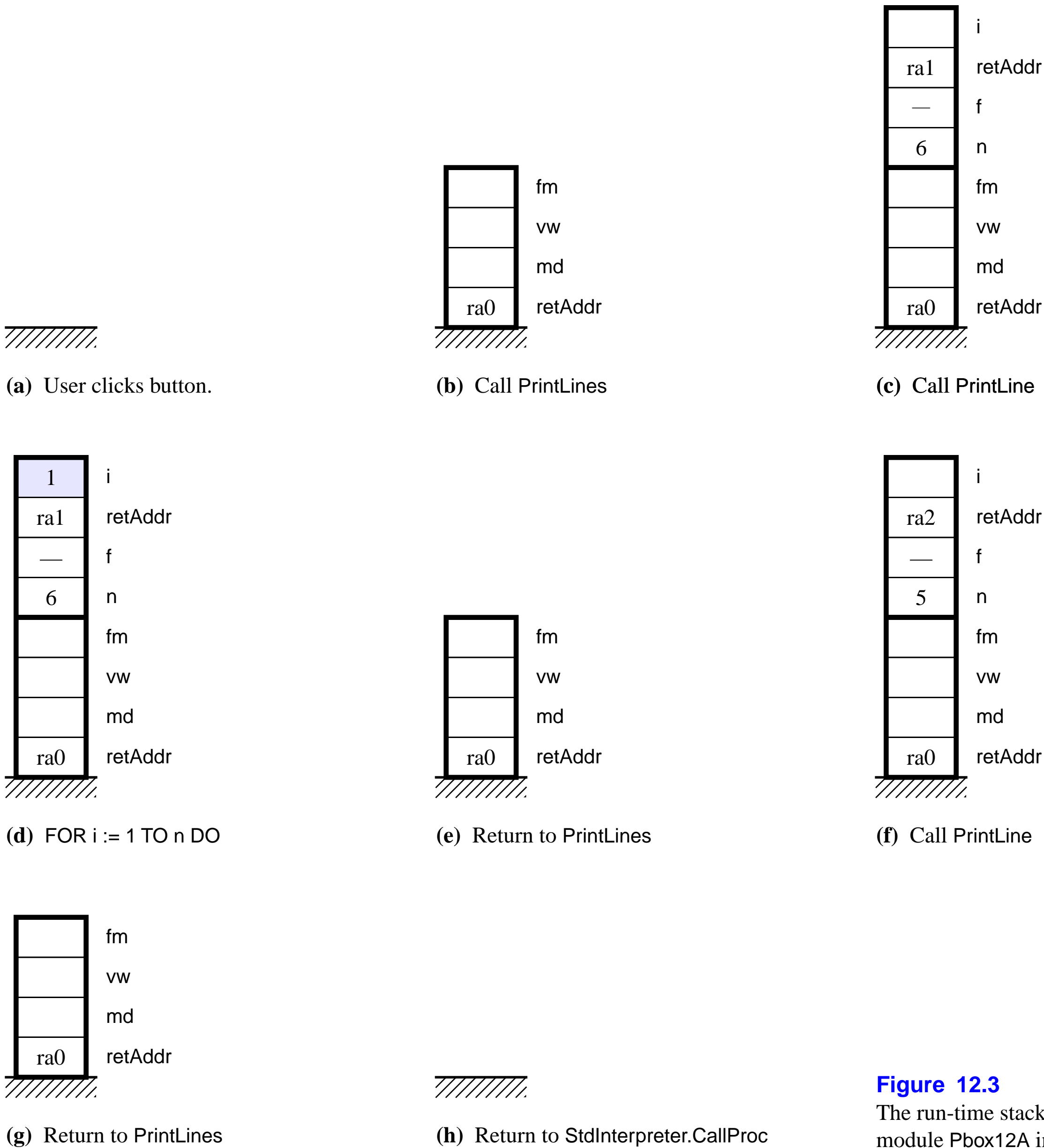


Figure 12.3
The run-time stack for module Pbox12A in Figure 12.2.

main program that must be written by the programmer. The framework itself is the main program. The programmer writes procedures that are executed in response to user actions within the graphical user interface. In Module12A, procedure PrintLines executes in response to the click of a button by the user. The framework includes a module named StdInterpreter, which in turn contains a procedure named CallProc. StdInterpreter.CallProc is the framework's procedure that calls Module12A.PrintLines when the user clicks the button.

Figure 12.3(a) shows the top part of the run-time stack before the user clicks the button. A large part of the run-time stack exists below the figure and is not shown.

Figure 12.3(b) shows the run-time stack when procedure StdInterpreter.CallProc calls procedure Pbox12A.PrintLines. Remember that when one proper procedure calls another the items pushed onto the stack are the parameters, followed by the return address, followed by storage for the local variables. Because PrintLines has no parameters, the only items pushed onto the run-time stack are the return address, indicated by retAddr, followed by storage for the local variables md, vw, and fm. The figure shows retAddr below the local variables because retAddr was pushed onto the stack first. It shows the value for retAddr as ra0, which indicates the address of some instruction in StdInterpreter.CallProc that need not concern us.

The collection of all the items pushed onto the run-time stack when a procedure is called is known as the *stack frame*. In this procedure call, the stack frame consists of the return address and storage allocated for the three local variables. The figure indicates the stack frame by the rectangle with the thick border.

The stack frame

After PrintLines is called, the MVC statements

```
md := TextModels.dir.New();
fm.ConnectTo(md)
```

execute. They give values to the cells on the stack. Because we are not concerned with the details of those values, the figures for the run-time stack do not indicate what those values are.

Figure 12.3(c) shows the effect of the first call to PrintLine. Unlike the previous call, this call has parameters as well as local variables. First, parameters n and f are pushed onto the stack. The formal parameter is the label for the memory cell and the value pushed is the content of the cell. Because n is called by value, the value 6 of the corresponding actual parameter is pushed. Because f is called by constant reference, indicated by IN in the formal parameter list, a reference to the corresponding actual parameter fm is pushed. For reasons not described here, formatters must always be called by constant reference. Chapter 15 describes the concept of call by constant reference. For now, we will ignore the formatter calling mechanism and simply put a hyphen in the cell for f on the run-time stack. Second, the return address is pushed onto the stack. Its value is ra1, which Figure 12.2 shows to be the instruction following the one that made the call. Third, storage for the local variable i is allocated. The stack frame in this call consists of all three kinds of items—parameters, return address and local variables.

Formatters are called by constant reference.

Following the call to PrintLine, its statements execute, the first of which is

```
FOR i := 1 TO n DO
```

Whenever assignments are made to local variables or to parameters called by value, they always change the content of the memory cells on the run-time stack. When this statement first executes, it sets i to 1. Figure 12.3(d) shows the content of the cell on the run-time stack labeled i changed to 1. As the loop progresses through the values 2, 3, 4, 5, and 6, the value in the cell changes accordingly. Each time the loop executes an asterisk is inserted into the text model.

Execution of the last line in PrintLine triggers a return from the procedure. The

computer uses the stored value for the return address to know which statement to execute next. It deallocates the top stack frame and returns flow of control to the calling procedure. In this case, ra1 is the return address. Figure 12.3(e) shows the stack after the deallocation of the frame just before the statement at ra1 is about to execute.

The statement at ra1 is yet another procedure call. Figure 12.3(f) shows allocation on the run-time stack. This time the value of the actual parameter is 5, and the return address is ra2, which is the address of

```
vw := TextViews.dir.New(md)
```

the instruction following the call. Procedure PrintLine executes again, this time inserting 5 asterisks into the text model.

Figure 12.3(g) shows the deallocation from this procedure call. Following execution of the MVC statements to give the text model a view and display it in a window, control returns to StdInterpreter.CallProc as Figure 12.3(h) shows.

A bar chart program

The next program uses the above technique for printing a row of asterisks to print a bar chart of data values. The input is from the focus window, which contains a list of real values shown in Figure 12.4. Processing is initiated by a menu selection not shown in the figure. The output is to a new window and consists of a single row for each real value containing the real value, followed by the vertical bar character | followed by a row of asterisks equal to the real value rounded off to the nearest integer.

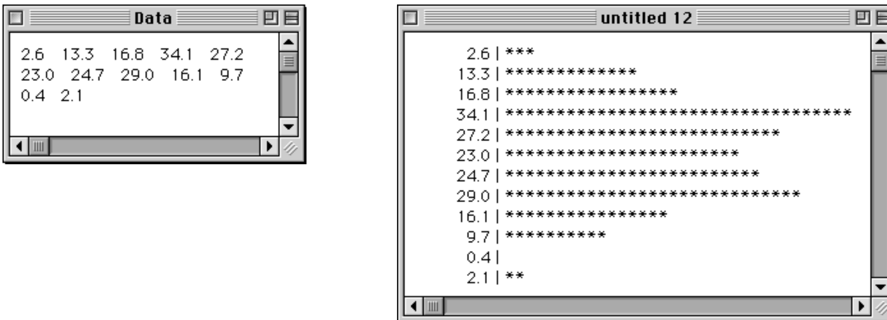


Figure 12.4 The input and output for the program of Listing 12.2.

Figure 12.5 shows the module that creates the output of Figure 12.4. Procedure PrintLine uses the standard function SHORT, which is a type conversion function. If n is a variable of type INTEGER and m is a variable of type LONGINT, then the assignment statement

```
m := n
```

```
MODULE Pbox12B;
   IMPORT TextModels, TextViews, Views, TextControllers, PboxMappers;

   PROCEDURE PrintLine (x: REAL; IN f: PboxMappers.Formatter);
      VAR
         i, n: INTEGER;
   BEGIN
      ASSERT(x >= 0.0, 20);
      f.WriteReal(x, 8, 1);
      f.WriteString(" | ");
      n := SHORT(ENTIER(x + 0.5));
      FOR i := 1 TO n DO
         f.WriteChar("*")
      END;
      f.WriteLn
   END PrintLine;

   PROCEDURE PrintHistogram*;
      VAR
         cn: TextControllers.Controller;
         sc: PboxMappers.Scanner;
         dataValue: REAL;
         md: TextModels.Model;
         vw: TextViews.View;
         fm: PboxMappers.Formatter;
   BEGIN
      cn := TextControllers.Focus();
      IF cn # NIL THEN
         md := TextModels.dir.New();
         fm.ConnectTo(md);
         sc.ConnectTo(cn.text);
         sc.ScanReal(dataValue);
         WHILE ~sc.eot DO
            PrintLine(dataValue, fm);
            (* ra1 *)
            sc.ScanReal(dataValue)
         END;
         vw := TextViews.dir.New(md);
         Views.OpenView(vw)
      END
   END PrintHistogram;

END Pbox12B.
```

Figure 12.5
A program that prints a bar chart from data values in a file. The procedure prints a single bar.

is legal, but the assignment statement

```
n := m
```

is not legal. The purpose of SHORT is to convert a long integer to the equivalent integer value. because long integers have a greater range than integers, it is possible

that SHORT will truncate the value of the long integer. The assignment statement

```
n := SHORT(m)
```

is legal because SHORT(m) has type INTEGER.

In procedure PrintLine, the statement

```
n := SHORT(ENTIER(x + 0.5));
```

rounds off the value of real variable x and assigns it to integer variable n. The ENTIER function returns a long integer value. The SHORT function converts it to an integer value so the value can be assigned to n.

Example 12.1 If x has the value 2.6, then x + 0.5 has the value 3.1, ENTIER(x + 0.5) has the long integer value 3, and SHORT(ENTIER(x + 0.5)) has the integer value 3. The real value of 2.6 has been properly rounded up. ■

Example 12.2 If x has the value 13.3, then x + 0.5 has the value 13.8, ENTIER(x + 0.5) has the long integer value 13, and SHORT(ENTIER(x + 0.5)) has the integer value 13. The real value of 13.3 has been properly rounded down. ■

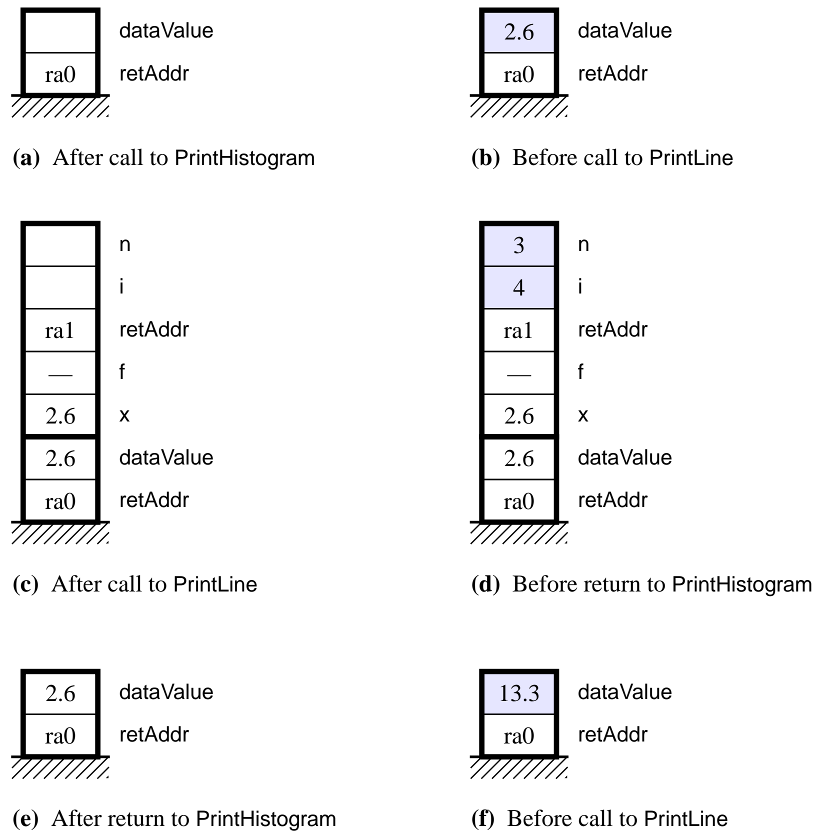
The program reads the first value from the focus window into the real variable dataValue. If the focus window is empty, then sc.eot will be true and the body of the WHILE loop will never execute. Otherwise, the variable dataValue will get the first real value in the focus window. With each loop execution the procedure PrintLine is called. After it executes and prints a line of the histogram, the next real value from the focus window is input into the real variable dataValue. The loop continues to execute while real values remain to be scanned in the focus window.

Figure 12.6 shows memory allocation for the procedure. Figure 12.6(a) shows the stack frame after the call to PrintHistogram. To keep the figure simple, the local MVC variables in the procedure are not shown. Other than the MVC variables, dataValue is the only local variable.

With the first scan, dataValue gets 2.6 from the focus window as Figure 12.6(b) shows. When PrintHistogram calls PrintLine, the computer allocates a new stack frame. As usual, parameters x and f are first pushed onto the stack, followed by the return address retAddr followed by local variables i and n as Figure 12.6(c) shows.

PrintLine converts the real value of 2.6 from x by rounding up to the integer value 3, which it gives to n. After proceeding through the FOR loop 3 times, the final value of i is 4 as Figure 12.6(d) shows. Termination of procedure PrintLine triggers a return to PrintHistogram. The top stack frame is deallocated as usual and the return address ra1 specifies the instruction in the calling procedure to be executed next. The result is the stack frame in Figure 12.6(e).

The scanner scans the next real value 13.3 from the focus window into local variable dataValue. The scan sets sc.eot to false because a real value was scanned into the variable. The test in the WHILE loop is true, which causes the body to execute again. So, PrintLine is called once again. Figure 12.6(f) shows the state of the run-time stack just before the call. The cell for dataValue contains the scanned value



(a) After call to PrintHistogram

(b) Before call to PrintLine

(c) After call to PrintLine

(d) Before return to PrintHistogram

(e) After return to PrintHistogram

(f) Before call to PrintLine

Figure 12.6 Memory allocation for Listing 12.5. Many MVC variables are not shown.

13.3. When the procedure is called, a stack frame identical to the top frame in Figure 12.6(c) is allocated except that the value of the formal parameter x is 13.3 instead of 2.6.

It is important to understand that between executions of PrintLine its parameters and local variables do not exist. It would be illegal to write a statement like INC(n) in the body of procedure PrintHistogram. It should be clear that such a statement would be impossible to execute because n would not exist at that point in time. The error would be detected by the compiler, which would not generate any object code.

Implementing preconditions

Procedure PrintLine can only print a bar for the histogram if the value that the calling procedure gives it for x is not negative. It has no mechanism for printing a histogram bar that extends to the left of the numbers in the text view. For the results to be meaningful the user should be restricted to providing nonnegative real numbers.

One approach to solving the problem of invalid input is to have an IF statement in procedure PrintLine that checks if x is negative and prints a histogram bar only if it is not. The problem is, what should the procedure do if the variable is negative? Should it do nothing and simply skip that line in the graph? Should it print a line with no asterisks? Should it send an error message to the Log? The problem of how to deal

with input errors is an important one that you should always consider when you design software.

The BlackBox framework promotes a particularly effective philosophy on how to deal with errors. For this problem, the philosophy states that the IF statement to guard against errors should not be in procedure PrintLine but should be in the calling procedure instead. Procedure PrintLine should perform only one task, that of printing a histogram bar with valid data. To program it to also handle the error conditions is not considered good design because those two tasks are not similar. Programs are easier to read, understand, and maintain if each procedure concentrates on one primary task.

The first executable statement in PrintLine

```
ASSERT(x >= 0.0, 20)
```

implements a precondition for the procedure and guarantees that the value for x to be processed will not be negative. If it is, a trap will occur as shown in Figure 12.7 where the second number in the focus window is negative. The precondition makes it the caller's responsibility to guard against the possibility of meaningless input. The delegation of that responsibility to the caller allows the called procedure to concentrate on one primary task, namely inserting one line of the histogram bar into the text model.

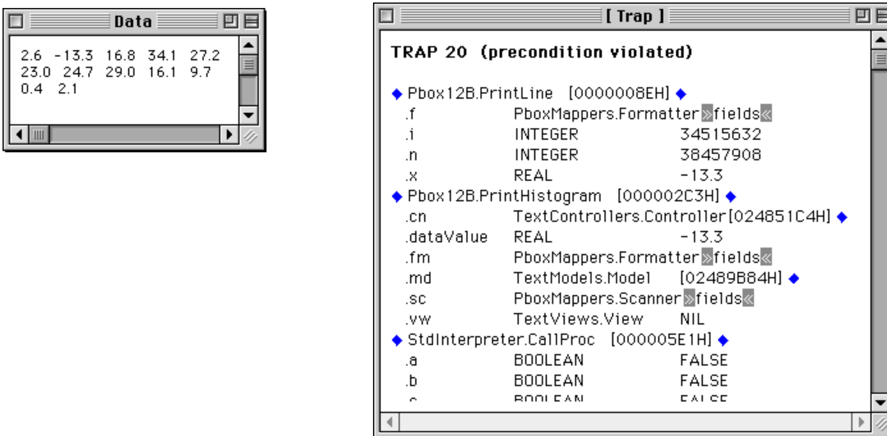


Figure 12.7
The trap generated by violation of the precondition of procedure PrintLine in Figure 12.5.

You should compare the trap window in Figure 12.7 with the figure of the run-time stack in Figure 12.6(c). The trap window is essentially a view of the run-time stack, except that the return address is not shown explicitly, and the local variables and parameters are shown in alphabetic order preceded by a period instead of in the order in which they actually occur on the stack. Figure 12.6(c) shows storage for x, f, i, and n on the top stack frame in the order in which they are allocated (bottom up), while Figure 12.7 shows storage for .f, .i, .n, and .x in alphabetic order. The trap window shows the type of each parameter and local variable as well as its value when the trap occurred. As expected, the trap window shows dataValue in PrintHistogram and x in PrintLine to have value -13.3.

Figure 12.6(c) shows that n and i have not received any values, but the trap window shows some huge random values for both .n and .i. This state is typical for variables that have not been assigned values. It is impossible for a variable to not have a value. If the program has not assigned a value to a variable, it will have some random value left over by the memory cell from the last time it was allocated or from its random state when the computer was first turned on.

Random values before the first assignment

Procedure PrintLine in Figure 12.5 is an illustration of good design philosophy for establishing preconditions with your procedures. Of course, procedure PrintHistogram is not an illustration of good design for user-friendly software because it allows the user to experience a program trap. A bulletproof program will never crash with a program trap. Problem 10 in this chapter challenges you to make procedure PrintHistogram bulletproof.

A bulletproof program does not crash with a program trap.

In practice, procedures that are not exported do not usually contain preconditions that are implemented by ASSERT statements. In that sense, procedure PrintLine in Figure 12.5 is not too typical. A single module is rarely written by more than one person. Because the same programmer who writes a nonexported procedure also writes the procedure that calls it, the programmer does not usually need to safeguard the called procedure against misuse. However, procedures that are exported are frequently used by programmers who did not write them. In those cases, the precondition is the formalization of a contract between the sever module and the client module. The ASSERT statement is frequently used to establish the precondition, and the corresponding error message number is documented in the Docu file. The client module then has the responsibility to ensure that the precondition is met before calling the procedure. This software development practice is known as Design by Contract and is described in Chapter 7, page 144.

Design by contract

Call by value

In Figure 12.5, procedure PrintLine passes its parameters by value. The program in Figure 12.8 illustrates the fact that in call by value, the formal parameter gets a copy of the value of the actual parameter. If a subsequent statement in the procedure changes the value of the formal parameter, it does not affect the actual parameter.

Figure 12.10 shows the memory allocation for Figure 12.8. In procedure CallByValue, variable i gets the value of 6, which is verified by the output statement. Then the program calls procedure PassVal with the actual parameter, i. The formal parameter, j, gets a copy of the value of i. When the statement

In call by value, the formal parameter gets the value of the actual parameter.

```
INC(j)
```

executes in procedure PassVal, it changes the copy of the value to 7. The original value of 6 for i in the main program does not change. The output of the program on the log is

```
i = 6
j = 7
i = 6
```

When parameters are called by value, the called procedure has access to the val-

ues of the formal parameters. If the value of the formal parameter is changed, the change is not reflected in the calling procedure. Information flows from the calling procedure to the called procedure—not in the other direction—via the value of the parameter.

```
MODULE Pbox12C;
   IMPORT StdLog;

   PROCEDURE PassVal (j: INTEGER);
   BEGIN
      INC(j);
      StdLog.String("j = "); StdLog.Int(j); StdLog.Ln;
   END PassVal;

   PROCEDURE CallByValue*;
      VAR
         i: INTEGER;
   BEGIN
      i := 6;
      StdLog.String("i = "); StdLog.Int(i); StdLog.Ln;
      PassVal(i);
      (* ra1 *)
      StdLog.String("i = "); StdLog.Int(i); StdLog.Ln
   END CallByValue;

END Pbox12C.
```

Figure 12.8
A procedure with a parameter called by value.

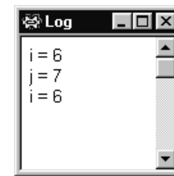
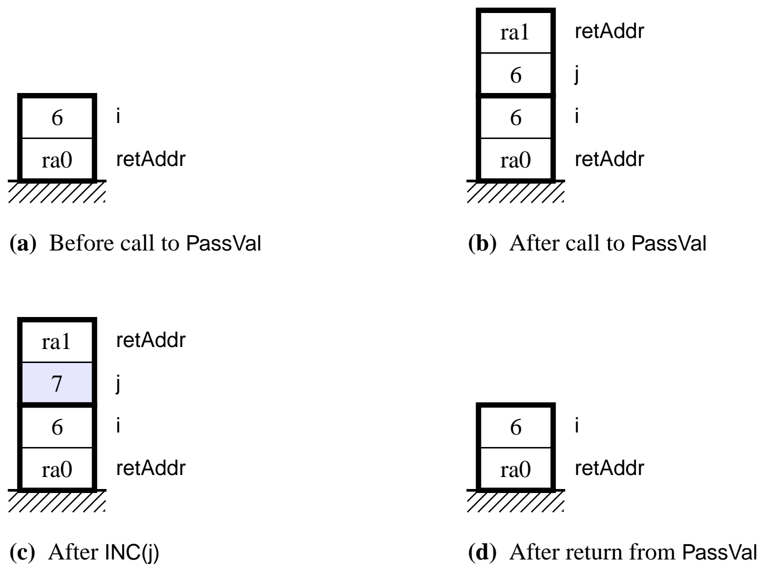


Figure 12.9
The output for Figure 12.8.



(a) Before call to PassVal

(b) After call to PassVal

(c) After INC(j)

(d) After return from PassVal

Figure 12.10
Memory allocation for Figure 12.8.

Call by reference

This section introduces the concept of call by reference, a technique by which a procedure can not only get values from the calling statement, but can give values back to it as well.

The program in Figure 12.11 is identical to the program in Figure 12.8 except for one important detail. The programmer placed reserved word VAR before formal parameter j in the formal parameter list. The reserved word VAR in a formal parameter list has a different meaning from its meaning in a local variable declaration. Outside a parameter list, VAR indicates the start of the variable declaration part. In a formal parameter list, VAR indicates that the parameter is called by reference instead of called by value. In Component Pascal, parameters that are called by reference are known as variable parameters.

```
MODULE Pbox12D;
   IMPORT StdLog;

   PROCEDURE PassRef (VAR j: INTEGER);
   BEGIN
      INC(j);
      StdLog.String("j = "); StdLog.Int(j); StdLog.Ln;
   END PassRef;

   PROCEDURE CallByReference*;
      VAR
         i: INTEGER;
   BEGIN
      i := 6;
      StdLog.String("i = "); StdLog.Int(i); StdLog.Ln;
      PassRef(i);
      (* ra1 *)
      StdLog.String("i = "); StdLog.Int(i); StdLog.Ln
   END CallByReference;

END Pbox12D.
```

Figure 12.11
A procedure with a parameter called by reference.

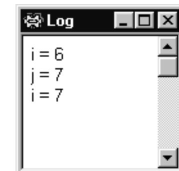


Figure 12.12
The output for Figure 12.11.

Figure 12.13 shows the memory allocation for Listing 12.11. When Component Pascal allocates memory for j, it does not give a copy of the value of the actual parameter, i, to j. Instead, it gives a reference to i. The figure indicates the reference to i by the arrow that points from the cell allocated for j to the cell allocated for i.

In call by reference, the formal parameter gets a reference to the actual parameter.

When j is used in the procedure, it is as though i has temporarily taken its place. The statement

```
INC(j)
```

has the effect of

```
INC(i)
```

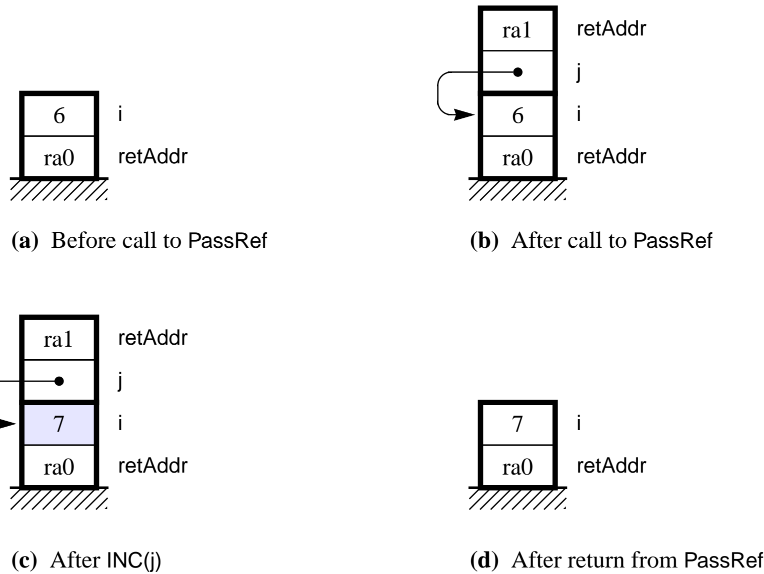


Figure 12.13 Memory allocation for Figure 12.11.

because j, during execution of the procedure, refers to i. When control returns back to the calling program, the value of i has been modified. The purpose of call by reference is to change the value of the actual parameter in the calling statement. The output of this program is

```
i = 6
j = 7
i = 7
```

When parameters are called by reference, the procedure has access to the values of the variables in the calling program referred to by the parameters. In this program, procedure PassRef had access to the value, 6, of i in the calling program. In that sense, information can flow from the calling program to the procedure.

When the procedure changes the value of a variable parameter, the value of a variable in the calling program changes. In this program, when procedure PassRef assigned a value to j, the value of i in the calling program changed. In that sense, information can flow from the procedure to the calling program.

In call by value, the actual parameter could be an arbitrary expression. But in call by reference, the actual parameter must be a single variable.

Example 12.3 The procedure call

```
PassVal(i + 2)
```

where PassVal is declared as in Listing 12.8, is legal. The formal parameter, j, is called by value and can take the expression i + 2 as an actual parameter. ∎

Example 12.4 The procedure call

PassRef(i + 2)

where PassRef is defined in Listing 12.11, is not legal, because i + 2 is not a single variable. ■

Call by result

In Figure 12.11, procedure PassRef uses the value 6 supplied by actual parameter i. It increments the value and gives the incremented value of 7 back to i. So, not only does it use the initial value of j, it also changes the actual parameter that is linked to j.

```
MODULE Pbox12E;
   IMPORT Dialog;
   VAR
      d*: RECORD
         width*, height*: REAL;
         area-, perim-: REAL
      END;

   PROCEDURE CalcRectSize (wid, ht: REAL; OUT ar, per: REAL);
   BEGIN
      ASSERT(wid > 0.0, 20);
      ASSERT(ht > 0.0, 21);
      ar := wid * ht;
      per := 2.0 * (wid + ht)
   END CalcRectSize;

   PROCEDURE Rectangle*;
   BEGIN
      IF (d.width > 0.0) & (d.height > 0.0) THEN
         CalcRectSize(d.width, d.height, d.area, d.perim);
         (* ra1 *)
      ELSE
         d.width := 0.0; d.height := 0.0;
         d.area := 0.0; d.perim := 0.0
      END;
      Dialog.Update(d)
   END Rectangle;

BEGIN
   d.width := 0.0; d.height := 0.0;
   d.area := 0.0; d.perim := 0.0
END Pbox12E.
```

Figure 12.14
Computing the area and perimeter of a rectangle with a procedure.

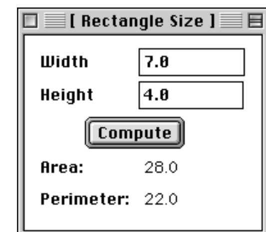


Figure 12.15
The dialog box for the procedure of Figure 12.14.

It frequently happens that the called procedure does not need the initial value of the formal parameter. In these situations, the initial value of the formal parameter

can be considered undefined. The passing mechanism is known as call by result, and formal parameters are so designated by the reserved word OUT in the formal parameter list.

The program in Figure 12.14 is a case in point. Figure 12.15 shows the corresponding dialog box. The module uses a procedure to calculate the area and perimeter of a rectangle from its width and height. Procedure CalcRect has four formal parameters. Two of the parameters are called by value and two are called by result.

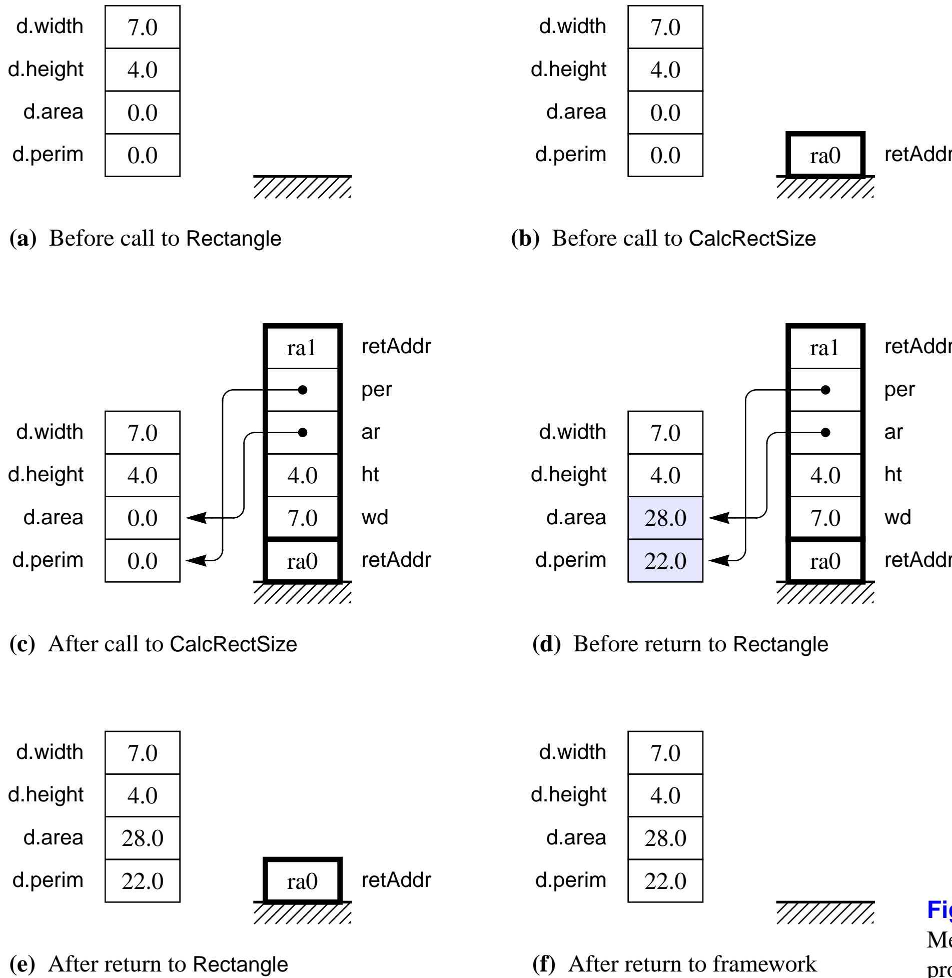


Figure 12.16 Memory allocation for the program of Figure 12.14.

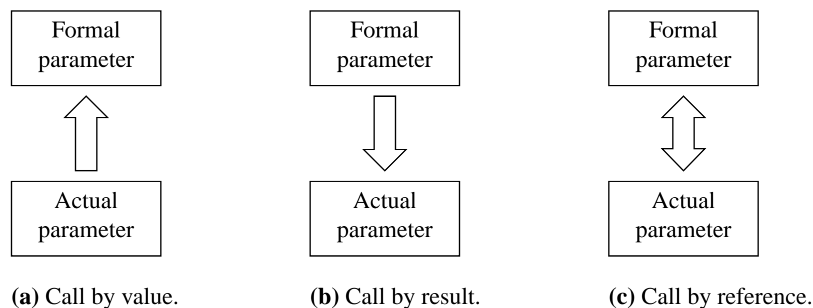
Because module Pbox12E implements a dialog box, it contains an interactor d, which is a global variable. Unlike local variables and parameters, global variables are not allocated on the run-time stack. They occupy a region of memory set aside for the module apart from its procedures. Memory for global variables is allocated when the module is loaded and remains in place while the stack frames are allocated and deallocated for procedure calls and returns. As long as the dialog box in Figure

12.15 is visible in the graphical user interface, storage for the values in Pbox12E.d is available.

Figure 12.16 is a trace of program execution. Storage for the global variables is shown to the left of the run-time stack. The module gets values for d.width and d.height from the dialog box. When the user presses the compute button, procedure Rectangle executes, which tests to make sure that d.width and d.height are both non-negative then calls procedure CalcRectSize. The actual parameters—d.width, d.height, d.area, and d.perim—correspond to the formal parameters—wid, ht, ar, and per. wid and ht are called by value. ar and per are called by result. wid gets the value from d.width, and ht gets the value from d.height. As with call by reference, ar refers to d.area, and per refers to d.perim. The same arrow notation is used in the figure to indicate call by result as was used previously to indicate call by reference. Another similarity with call by reference is that the actual parameter in call by result must be a variable.

In call by result, the formal parameter gets a reference to the actual parameter.

Procedure CalcRect uses the values of wid and ht in its computation. That information flows from the calling procedure to the called procedure. When CalcRect assigns values to ar and per, it changes the values of d.area and d.perim in procedure Rectangle. That information flows from the called procedure to the calling procedure. In general, information flows from the calling procedure to the called procedure when parameters are called by value, from the called procedure to the calling procedure when parameters are called by result, and both ways when parameters are called by reference. Figure 12.17 shows the flow of information for these three calling mechanisms.



(a) Call by value. **(b)** Call by result. **(c)** Call by reference.

Figure 12.17
The flow of information for three different calling mechanisms.

The key question to ask in deciding whether a parameter should be called by value, called by result, or called by reference is, Which way does the information flow between the actual parameter and the formal parameter? If the purpose of the formal parameter is to receive a value from the actual parameter use call by value. If the purpose of the formal parameter is to change the value of the actual parameter, use call by result. In this case, the initial value of the actual parameter can be considered undefined. If the purpose of the formal parameter is both to receive a value from the actual parameter and to change the value of the actual parameter, use call by reference. In summary:

Summary of call by value, call by result, and call by reference

- Call by value
  - Default
  - Use when the actual parameter should not change.
  - The actual parameter can be an expression.
- Call by result
  - OUT
  - Use when the actual parameter should change and its initial value is undefined.
  - The actual parameter must be a variable.
- Call by reference
  - VAR
  - Use when the actual parameter should change and the called procedure uses its initial value.
  - The actual parameter must be a variable.

Using parameters

Any number of parameters can be in the parameter list of a procedure. The number of parameters in the actual parameter list must equal the number of parameters in the formal parameter list. The types must correspond as well. The violation of these rules is a syntax error.

Example 12.5 The statement

```
PrintLine(6.0, fm)
```

where PrintLine is declared as in Figure 12.2, is illegal because the formal parameter, n, is an integer and the actual parameter, 6.0, is a real value. ■

Example 12.6 With PrintLine declared as before, the function call

```
PrintLine(6)
```

is illegal, because there is only one actual parameter but two formal parameters. ■

Remember that if x is a real variable, the assignment x := 7 is legal. The integer value, 7, is converted to a real value, 7.0, before being assigned to x. But if i is an integer variable, then i := 2.7 is illegal. Similarly, integer values can be actual parameters for real formal parameters.

Example 12.7 The procedure call

```
CalcRectSize(42, 10, d.area, d.perim)
```

where CalcRectSize is defined as in Figure 12.14, is legal. The integer value, 42, is automatically converted to the real value, 42.0, before being given to the formal parameter, wid. Similarly, 10 is automatically converted to 10.0. ∎

It is legal to have the actual parameter be a variable with the same name as the formal parameter. But you should realize that there are separate memory locations for each. To keep the distinction clear between the formal parameters and actual parameters, this book will generally avoid using the same name for both. The usual convention will have the formal parameter serve as an abbreviation of the actual parameter.

Example 12.8 If a procedure has a declaration with formal parameters as

```
PROCEDURE PrintLine (num: INTEGER; f: PboxMappers.Formatter);
```

and the calling procedure has a variable declared as

```
VAR
   num: INTEGER
```

then the procedure call

```
PrintLine(num, fm)
```

would be legal even though the first formal parameter has the same name as the actual parameter. ∎

Using global variables

The modules in this book that implement dialog boxes, such as the one in Figure 5.9 that computes the coins required for a given amount of change, use global variables to link to the dialog box. The variable declaration from Figure 5.9

```
VAR
   d*: RECORD
      change*: INTEGER;
      dimes-, nickels-, pennies-: INTEGER
   END;
```

is nested within module Pbox05B but not within procedure MakeChange. Therefore, it is a global variable.

Figure 9.3 shows another example of global variables. The declaration

```
stackA, stackB: PboxStackObj.Stack;
```

is nested within module Hw99Pr0980 and not within any of the procedures PushA, PushB, PopA, PopB, or ClearStacks. Therefore, stackA and stackB are global variables.

```
MODULE Pbox12F;
    IMPORT TextModels, TextViews, Views, TextControllers, PboxMappers;

    VAR (* WARNING-Unnecessary global variables. Bad design. *)
        cn: TextControllers.Controller;
        sc: PboxMappers.Scanner;
        dataValue: REAL;
        md: TextModels.Model;
        vw: TextViews.View;
        fm: PboxMappers.Formatter;

    PROCEDURE PrintLine;
        VAR
            i, n: INTEGER;
    BEGIN
        ASSERT(dataValue >= 0.0, 20);
        fm.WriteReal(dataValue, 8, 1);
        fm.WriteString(" | ");
        n := SHORT(ENTIER(dataValue + 0.5));
        FOR i := 1 TO n DO
            fm.WriteChar("*")
        END;
        fm.WriteLn
    END PrintLine;

    PROCEDURE PrintHistogram*;
    BEGIN
        cn := TextControllers.Focus();
        IF cn # NIL THEN
            md := TextModels.dir.New();
            fm.ConnectTo(md);
            sc.ConnectTo(cn.text);
            sc.ScanReal(dataValue);
            WHILE ~sc.eot DO
                PrintLine;
                sc.ScanReal(dataValue)
            END;
            vw := TextViews.dir.New(md);
            Views.OpenView(vw)
        END
    END PrintHistogram;

END Pbox12F.
```

Figure 12.18
A procedure whose computation is the same as that in Figure 12.5, but without parameters for PrintLine.

There is nothing to prevent you from declaring as many global variables as you would like, and using them for whatever purpose you desire. You could even use global variables in place of parameters to achieve the same effect. Figure 12.18 shows a module that does the same computation as the one in Figure 12.5, but without any parameters in procedure PrintLine. There is no real parameter x or formatter f, and the variables dataValue and fm (along with a host of other variables) are glo-

bal. Instead of having the actual parameter dataValue being mapped to parameter x and using x in the procedure, the procedure simply does its computation directly on dataValue. Similarly, instead of having the actual parameter fm being mapped to parameter f and using f in the procedure, the procedure simply does its computation directly on fm.

Which program do you think is better—the one in Figure 12.5 or the one in Figure 12.18? Most beginning programmers think that the one in Figure 12.18 is better. After all, it is shorter, and when you write it you do not need to bother with a parameter list. Why use a parameter list if you can do the same job without one?

Design rule for global variables

Most software designers would contend that the program in Figure 12.5 is better. The general design rule for procedures is that you should avoid unnecessary use of global variables. For small programs, such as the ones in this book, the problems created by using global variables are not as evident as they are with large programs. The advantage of using parameters and local variables in a procedure is that the procedure is self-contained. Therefore, when compared to a procedure that uses global variables, it is

- Easier to read
- Easier to modify
- Easier to use in other programs

Look at the two procedures from these programs, when they are isolated from their environments.

```
PROCEDURE PrintLine (x: REAL; VAR f: PboxMappers.Formatter); PROCEDURE PrintLine;
   VARVAR
   i, n: INTEGER;i, n: INTEGER;
BEGINBEGIN
   ASSERT(x >= 0.0, 20);ASSERT(dataValue >= 0.0, 20);
   f.WriteReal(x, 8, 1);fm.WriteReal(dataValue, 8, 1);
   f.WriteString(" | ");fm.WriteString(" | ");
   n := SHORT(ENTIER(x + 0.5));n := SHORT(ENTIER(dataValue + 0.5));
   FOR i := 1 TO n DOFOR i := 1 TO n DO
   f.WriteChar("*")fm.WriteChar("*")
   END;END;
   f.WriteLnfm.WriteLn
END PrintLine;END PrintLine;
```

Suppose the program listing were 30 pages long, and these procedures are pages away form their calling programs. You are in the process of reading the code to find a bug and you come across these procedures.

The first PrintLine is completely self-contained. Its statements refer only to local variables. You can see from reading it that whatever the calling procedure supplies for the first parameter this PrintLine procedure rounds off that value to an integer and writes that number of asterisks to a text model to which the second parameter is connected. But the second PrintLine is not self-contained. What is dataValue? What is fm? What are their types? You cannot tell for sure unless you scroll back many pages to search for their declarations.

Furthermore, the first PrintLine is more general than the second PrintLine. You can

copy it out of this module and paste it into another module with the assurance that it will work correctly. The second PrintLine is not general-purpose. Its correctness depends on its environment. You cannot place it in another module with different global variables and expect it to work. You would need to change its environment to make sure that dataValue and fm are declared consistently and used properly with the procedure. Or, you would need to rename dataValue and f to match the global variables of the environment.

When to use a global variable

An important skill for you to develop now is the ability to design good procedures. You should always rely on local variables with parameters to pass information between the calling and the called procedure unless there is reason not to. When must you resort to a global variable? When its value must be persistent between invocations of the module's exported procedures.

Consider the dialog box for Figure 12.14, which computes the area and perimeter of a rectangle. The interactor d for the dialog must be global because the dialog box persists between invocations of the exported procedure Rectangle. Before the procedure executes, the dialog box exists on the screen with values entered by the user. After the procedure executes, the dialog box persists on the screen displaying the results of the computation. Figure 12.16 shows the values linked to the dialog box. The local variables are deallocated when Rectangle terminates, but the dialog box remains on the screen. Therefore, the interactor d must be global.

Another example of the proper use of global variables is the program of Figure 9.3, which manipulates two stacks. The variables stackA and stackB are not linked directly to a dialog box. Nevertheless, the data that they store must be persistent between invocations of PushA, PushB, PopA, PopB, and ClearStacks. After the user pushes a value onto a stack, the stack itself with all its data should not be deallocated. If the stack were deallocated the value that was just pushed would be lost.

Exercises

1. Determine the outputs to the Log of Exr1a and Exr1b.

(a)
```
PROCEDURE Pass1a (n: INTEGER);
BEGIN
    n := n * 2;
    StdLog.Int(n); StdLog.String("  ")
END Pass1a;

PROCEDURE Exr1a*;
    VAR
        num: INTEGER;
BEGIN
    num := 5;
    Pass1a(num);
    StdLog.Int(num) (* ra1 *)
END Exr1a;
```

(b)
```
PROCEDURE Pass1b (VAR n: INTEGER);
BEGIN
    n := n * 2;
    StdLog.Int(n); StdLog.String("  ")
END Pass1b;

PROCEDURE Exr1b*;
    VAR
        num: INTEGER;
BEGIN
    num := 5;
    Pass1b(num);
    StdLog.Int(num) (* ra1 *)
END Exr1b;
```

2. For Exercise 1(a), draw the memory allocation as in Figure 12.10 for the following times:

(a) Just before the call to Pass1a **(b)** Just after the call to Pass1a
(c) Just before the return from Pass1a **(d)** Just after the return from Pass1a

3. For Exercise 1(b), draw the memory allocation as in Figure 12.13 for the following times.

(a) Just before the call to Pass1b **(b)** Just after the call to Pass1b
(c) Just before the return from Pass1b **(d)** Just after the return from Pass1b

4. Determine the outputs to the Log of Exr4a and Exr4b.

(a)

```
PROCEDURE Pass4a (OUT n, m: INTEGER)
BEGIN
    n := 1;
    m := 2
END Pass4a;

PROCEDURE Exr4a*;
    VAR
    n, m: INTEGER;
BEGIN
    Pass4a (n, m);
    (* ra1 *)
    StdLog.String('n = '); StdLog.Int(n); StdLog.Ln;
    StdLog.String('m = '); StdLog.Int(m); StdLog.Ln
END Exr4a;
```

(b)

```
PROCEDURE Pass4b (OUT m, n: INTEGER);
BEGIN
    n := 1;
    m := 2
END Pass4b;

PROCEDURE Exr4b*;
    VAR
    n, m: INTEGER;
BEGIN
    Pass4b (n, m);
    (* ra1 *)
    StdLog.String('n = '); StdLog.Int(n); StdLog.Ln;
    StdLog.String('m = '); StdLog.Int(m); StdLog.Ln
END Exr4b;
```

5. For Exercise 4(a), draw the memory allocation as in Figure 12.16 for the following times.

(a) Just before the call to Pass4a **(b)** Just after the call to Pass4a
(c) Just before the return to Exr4a **(d)** Just after the return to Exr4a

6. For Exercise 4(b), draw the memory allocation as in Figure 12.16 for the following times.

(a) Just before the call to Pass4b **(b)** Just after the call to Pass4b
(c) Just before the return to Exr4b **(d)** Just after the return to Exr4b

7. Suppose a calling procedure declares variables

```
a, b, c: REAL;
```

and procedure Exr7 has the heading

```
PROCEDURE Exr7 (d: REAL; OUT e: REAL; VAR f: REAL)
```

State whether each of the procedure calls is legal. For those that are not legal, explain why.

(a) Exr7(a, b, c) **(b)** Exr7(a, b) **(c)** Exr7(7.0, b, c)
(d) Exr7(a, 7.0, c) **(e)** Exr7(a, b, 7.0) **(f)** Exr7((a + b) / 2.0, b, c)
(g) Exr7(7, b, c) **(h)** Exr7(a, 7, c)

8. Suppose a calling procedure declares variables

```
p, q, r: INTEGER
```

and procedure Exr8 has the heading

```
PROCEDURE Exr8 (s: INTEGER; OUT t: INTEGER, u: INTEGER)
```

State whether each of the procedure calls is legal. For those that are not legal, explain why.

(a) Exr8(p, q, r) **(b)** Exr8(p, q) **(c)** Exr8(5, q, r)
(d) Exr8(p, 5, r) **(e)** Exr8(p, q, 5) **(f)** Exr8((p + q) DIV 2, q, r)
(g) Exr8(5.0, q, r) **(h)** Exr8(p, 5.0, r)

9. **(a)** What are the advantages of using local variables and parameters instead of global variables? **(b)** When is it justified to use a global variable?

Problems

10. Make the program in Figure 12.5 bulletproof by continuing the loop only if the scanner is not at the end of text and dataValue is not negative. After terminating the loop, check sc.eot to determine how the loop terminates. If it terminates without reaching the end of text you know a negative value was encountered. In that case, insert an appropriate error message in the text model. Otherwise do nothing and terminate normally.

11. Declare

```
PROCEDURE PrintRow (numSpace, numChr: INTEGER; ch: CHAR; VAR f: PboxMappers.Formatter)
```

that inserts into a text model to which f is connected one line with numSpace spaces followed by numChr occurrences of ch. For example, PrintRow (5, 3, 'a', fm) should insert one line with five spaces followed by three ‘a’s. Use your procedure to print the pattern shown below to a new window. Activate your program with a commander in your Docu file. After your program creates the window you will need to display the pattern in Courier or some other monospaced font to have the characters aligned properly.

```
   *
  ***
 *****
*******
  :::
  :::
```

12. For Chapter 11, Problem 5, write a program to display a solid box of @ symbols in a new window. Use a procedure called PrintPattern with three parameters—one that specifies how many rows to print, one that specifies the number of columns to print, and one that specifies the formatter to use. Assert as a precondition that the number of rows and columns must be positive. Insure in the calling procedure that the precondition is not violated by not changing anything in the dialog box and not opening a new window if a nonpositive value is entered.

13. For Chapter 11, Problem 6, write a program to display a hollow box of zeros in a new window. Use a procedure called PrintPattern with three parameters—one that specifies how many rows to print, one that specifies the number of columns to print, and one that specifies the formatter to use. Assert as a precondition that the number of rows and columns must be greater than 1. Insure in the calling procedure that the precondition is not violated.

14. For Chapter 11, Problem 7, write a program to display a right triangle of zeros with a vertical right side in a new window. Use a procedure called PrintPattern with a parameter that specifies how many rows to print and a parameter that specifies the formatter to use. Assert as a precondition that the number of rows must be positive. Insure in the calling procedure that the precondition is not violated.

15. For Chapter 11, Problem 8, write a program to display symmetric triangle of zeros in a new window. Use a procedure called PrintPattern with a parameter that specifies how many rows to print and a parameter that specifies the formatter to use. Assert as a precondition that the number of rows must be positive. Insure in the calling procedure that the precondition is not violated.

16. A rectangular box has width, length, and height. Write a program that inputs these dimensions from a dialog box and calls a procedure that calculates the volume and total surface area of the box. Your procedure should have five formal parameters, each of which you should determine whether to call by value, call by result, or call by reference.

17. Implement a dialog box that looks and behaves like Figure 7.3, but with one additional button labeled "Reverse Stack". When the user presses this button, all the items on the stack should be rearranged in reverse order with the item originally at the top of the stack on the bottom and the item originally at the bottom on the top. The text fields for Push and Pop should not change, nor should the field for the number of items change. Use two temporary variables called tempStack1 and tempStack2 for intermediate storage of stack values in your exported procedure ReverseStack. Write another procedure, not exported, named CopySourceToDest

```
PROCEDURE CopySourceToDest (source: PboxStackADS.Stack; OUT dest: PboxStackADS.Stack)
```

that clears Dest then copies source to dest in reverse order by successively popping items from source and pushing them to dest. Use a temporary variable called temp to store an individual value from the stack between the pop and push operations. Note that source is called by value so that its actual parameter will not change. Procedure ReverseStack can simply call CopySourceToDest three times, copying Stack A to tempStack1, then tempStack1 to tempStack2, then tempStack2 back to Stack A, which will be in reverse order. Import module PboxStackADS.

18. Do Problem 17 but import PboxStackObj.