5

Characteristics of High-Quality Routines

Contents

5.1 Valid Reasons to Create a Routine
5.2 Good Routine Names
5.3 Strong Cohesion
5.4 Loose Coupling
5.5 How Long Can a Routine Be?
5.6 Defensive Programming
5.7 How to Use Routine Parameters
5.8 Considerations in the Use of Functions
5.9 Macro Routines

Related Topics

Steps in routine construction: Chapter 4
Characteristics of high-quality modules: Chapter 6
General design techniques: Section 7.5
Software architecture: Section 3.4

CHAPTER 4 DESCRIBED THE STEPS you take to build a routine. It focused on the construction process. This chapter zooms in on the routine itself, on the characteristics that make the difference between a good routine and a bad one.

If you'd rather read about high-level design issues before wading into the nitty-gritty details of individual routines, read the high-level design chapter, Chapter 7, first and come back to this chapter later. Since modules are also more abstract than individual routines, you might also prefer to read about high-quality modules in Chapter 6 before reading this chapter.

Before jumping into the details of high-quality routines, it will be useful to nail down two basic terms. What is a "routine"? A routine is an individual function or procedure invocable for a single purpose. Examples include a function in C, a function or a procedure in Pascal or Ada, a subprogram in Basic, and a subroutine in Fortran. For some uses, macros in C and sections of code called with *GOSUB* in Basic can also be thought of as routines. You can apply many of the techniques for creating a high-quality routine to these variants.

What is a "high-quality routine"? That's a harder question. Perhaps the easiest answer is to show what a high-quality routine is not. Here's an example of a low-quality routine:

CODING HORROR

Pascal Example of a Low-Quality Routine

```
Procedure HandleStuff( Var InputRec: CORP_DATA, CrntQtr: integer,
     EmpRec: EMP_DATA, Var EstimRevenue: Real, YTDRevenue: Real,
     ScreenX: integer, ScreenY: integer, Var NewColor: COLOR_TYPE,
     Var PrevColor: COLOR_TYPE, Var Status: STATUS_TYPE,
     ExpenseType: integer );

begin
for i:= 1 to 100 do begin
   InputRec.revenue[i] := 0;
   InputRec.expense[i] := CorpExpense[ CrntQtr, i ]
   end;
UpdateCorpDatabase( EmpRec );
EstimRevenue := YTDRevenue * 4.0 / real( CrntQtr );
NewColor := PrevColor;
Status := Success;
if ExpenseType = 1 then begin
                     for i := 1 to 12 do
                     Profit[i] := Revenue[i] - Expense.Type1[i]
                     end
else if ExpenseType = 2 then begin
                         Profit[i] := Revenue[i] - Expense.Type2[i]
                         end
else if ExpenseType = 3 then
   begin
   Profit[i] := Revenue[i] - Expense.Type3[i]
   end
end;
```

What's wrong with this routine? Here's a hint: You should be able to find at least 10 different problems with it. Once you've come up with your own list, look at the list below:

- The routine has a bad name. *HandleStuff()* tells you nothing about what the routine does.

- The routine isn't documented. (The subject of documentation extends beyond the boundaries of individual routines and is discussed in Chapter 19, "Self-Documenting Code.")
- The routine has a bad layout. The physical organization of the code on the page gives few hints about its logical organization. Layout strategies are used haphazardly, with different styles in different parts of the routine. Compare the styles where *ExpenseType = 2* and *ExpenseType = 3*. (Layout is discussed in Chapter 18, "Layout and Style.")
- The routine's input variable, *InputRec*, is changed. If it's an input variable, its value should not be modified. If the value of the variable is supposed to be modified, the variable should not be called *InputRec*.
- The routine reads and writes global variables. It reads from *CorpExpense* and writes to *Profit*. It should communicate with other routines more directly than by reading and writing global variables.
- The routine doesn't have a single purpose. It initializes some variables, writes to a database, does some calculations—none of which seem to be related to each other in any way. A routine should have a single, clearly defined purpose.
- The routine doesn't defend itself against bad data. If *CrntQtr* equals *0*, then the expression *YTDRevenue * 4.0 / real(CrntQtr)* causes a divide-by-zero error.
- The routine uses several magic numbers: *100*, *4.0*, *12*, *2*, and *3*. Magic numbers are discussed in Section 11.1, "Numbers in General."
- The routine uses only two fields of the *CORP_DATA* type of parameter. If only two fields are used, the specific fields rather than the whole structured variable should be passed in.
- Some of the routine's parameters are unused. *ScreenX* and *ScreenY* are not referenced within the routine.
- One of the routine's parameters is mislabeled. *PrevColor* is labeled as a *Var* parameter even though it isn't assigned a value within the routine.
- The routine has too many parameters. The upper limit for an understandable number of parameters is about 7. This routine has 11. The parameters are laid out in such an intimidating way that most people wouldn't try to examine them closely or even count them.
- The routine's parameters are poorly ordered and are not documented. (Parameter ordering is discussed in this chapter. Documentation is discussed in Chapter 19.)

Aside from the computer itself, the routine is the single greatest invention in computer science. The routine makes programs easier to read and easier to understand than any other feature of any programming language. It's a crime to abuse this senior statesman of computer science with code like that shown in the example above.

The routine is also the greatest technique ever invented for saving space and improving performance. Imagine how much larger your code would be if you had to repeat the code for every call to a routine instead of branching to the routine. Imagine how hard it would be to make performance improvements in the same code used in a dozen places instead of making them all in one routine. The routine makes modern programming possible.

"OK," you say, "I already know that routines are great, and I program with them all the time. This discussion seems kind of remedial, so what do you want me to do about it?"

I want you to understand that there are many valid reasons to create a routine and that there are right ways and wrong ways to go about it. As an undergraduate computer-science student, I thought that the main reason to create a routine was to avoid duplicate code. The introductory textbook I used said that routines were good because the avoidance of duplication made a program easier to develop, debug, document, and maintain. Period. Aside from syntactic details about how to use parameters and local variables, that was the total extent of the textbook's explanation of the theory and practice of routines. It was not a good or complete explanation. The following sections describe why and how to create routines.

5.1 Valid Reasons to Create a Routine

Here's a list of valid reasons to create a routine. The reasons overlap somewhat, and they're not intended to make an orthogonal set.

KEY POINT

Reducing complexity. The single most important reason to create a routine is to reduce a program's complexity. Create a routine to hide information so that you won't need to think about it. Sure, you'll need to think about it when you write the routine. But after it's written, you should be able to forget the details and use the routine without any knowledge of its internal workings. Other reasons to create routines—minimizing code size, improving maintainability, and improving correctness—are also good reasons, but without the abstractive power of routines, complex programs would be impossible to manage intellectually.

One indication that a routine needs to be broken out of another routine is deep nesting of an inner loop or a conditional. Reduce the containing routine's complexity by pulling the nested part out and putting it into its own routine.

CROSS-REFERENCE For a discussion of decomposition, see "Choosing Components to Modularize" in Section 7.2.

Avoiding duplicate code. Undoubtedly the most popular reason for creating a routine is to avoid duplicate code. Indeed, creation of similar code in two routines implies an error in decomposition. Pull the duplicate code from both routines, put a generic version of the common code into its own routine, and then let both call the part that was put into the new routine. With code in one place, you save the space that would have been used by duplicated code. Modifications will be easier because you'll need to modify the code in only one location. The code will be more reliable because you'll have to check only one place to ensure that the code is right. Modifications will be more reliable because you'll avoid making successive and slightly different modifications under the mistaken assumption that you've made identical ones.

CROSS-REFERENCE For a discussion of areas that are likely to change, see "Areas likely to change" in Section 6.2.

Limiting effects of changes. Isolate areas that are likely to change so that the effects of changes are limited to the scope of a single routine or, at most, a few routines. Design so that areas that are most likely to change are the easiest to change. Areas likely to change include hardware dependencies, input/output, complex data structures, and business rules.

Hiding sequences. It's a good idea to hide the order in which events happen to be processed. For example, if the program typically gets data from the user and then gets auxiliary data from a file, neither the routine that gets the user data nor the routine that gets the file data should depend on the other routine's being performed first. If you commonly have two lines of code that read the top of a stack and decrement a *StackTop* variable, put them into a *PopStack()* routine. Design the system so that either could be performed first, and then create a routine to hide the information about which happens to be performed first.

CROSS-REFERENCE For details on improving performance, see Chapter 28, "Code-Tuning Strategies," and Chapter 29, "Code-Tuning Techniques."

Improving performance. You can optimize the code in one place instead of several places. Having code in one place means that a single optimization benefits all the routines that use that routine, whether they use it directly or indirectly. Having code in one place makes it practical to recode the routine with a more efficient algorithm or a faster, more difficult language such as assembler.

CROSS-REFERENCE For details on information hiding, see Section 6.2, "Information Hiding."

Making central points of control. It's a good idea to keep control for each task in one place. Control assumes many forms. Knowledge of the number of entries in a table is one form. Control of hardware devices—disks, tapes, printers, plotters, and so on—is another. Using one routine to read from a file and one routine to write to it is a form of centralized control. This is especially useful because if the file needs to be converted to an in-memory data structure, the changes affect only the access routines. Reading and modifying the contents of internal data structures with specialized routines is another form of centralized control.

The idea of centralized control is similar to information hiding, but it has unique heuristic power that makes it worth adding to your programming toolbox.

CROSS-REFERENCE For details on hiding the implementation of data structures, see Section 12.3, "Abstract Data Types (ADTs)."

Hiding data structures. You can hide the implementation details of a data structure so that most of the program doesn't need to worry about the messy details of manipulating computer-science structures and can deal with the data in terms of how it's used in the problem domain. Routines that hide implementation details provide a valuable level of abstraction that reduces a program's complexity. They centralize data-structure operations in one place and reduce the chance of errors in working with the data structure. They make it easy to change the structure without changing most of the program.

Hiding global data. If you need to use global data, you can hide its implementation details as just described. Working with global data through access routines provides several benefits. You can change the structure of the data without changing your program. You can monitor accesses to the data. The discipline of using access routines also encourages you to think about whether the data is really global; it might be more accurate to treat it as data that's local to several routines in a single module or as part of an abstract data type.

Hiding pointer operations. Pointer operations tend to be hard to read and error prone. By isolating them in routines, you can concentrate on the intent of the operation rather than the mechanics of pointer manipulation. Also, if the operations are done in only one place, you can be more certain that the code is correct. If you find a better data structure than pointers, you can change the program without traumatizing the routines that would have used the pointers.

Promoting code reuse. Code put into modular routines can be reused in other programs more easily than the same code embedded in one larger routine. Even if a section of code is called from only one place in the program and is understandable as part of a larger routine, it makes sense to put it into its own routine if that piece of code might be used in another program.

Planning for a family of programs. If you expect a program to be modified, it's a good idea to isolate the parts that you expect to change by putting them into their own routines. You can then modify the routines without affecting the rest of the program, or you can put in completely new routines instead. Several years ago I managed a team that wrote a series of programs used by our clients to sell insurance. We had to tailor each program to the specific client's insurance rates, quote-report format, and so on. But many parts of the programs were similar: the routines that input information about potential customers, that stored information in a customer database, that looked up rates, that computed total rates for a group, and so on. The team modularized the program so that each part that varied from client to client was in its own

module. The initial programming might have taken three months or so, but when we got a new client, we merely wrote a handful of new modules for the new client and dropped them into the rest of the code. Two or three days' work, and voilà! Custom software!

Making a section of code readable. Putting a section of code into a well-named routine is one of the best ways to document its function. Instead of reading a series of statements like

```
if ( Node <> NULL )
   while ( Node.Next <> NULL ) do
      Node = Node.Next
   LeafName = Node.Name
else
   LeafName = ""
```

you can read a statement like

```
LeafName = GetLeafName( Node )
```

The new routine is so short that nearly all it needs for documentation is a good name. Using a function call instead of six lines of code makes the routine that originally contained the code less complex and documents it automatically.

Improving portability. Use of routines isolates nonportable capabilities, explicitly identifying and isolating future portability work. Nonportable capabilities include nonstandard language features, hardware dependencies, operating-system dependencies, and so on.

Isolating complex operations. Complex operations—complicated algorithms, communications protocols, tricky boolean tests, operations on complex data, and so on—are prone to errors. If an error does occur, it will be easier to find if it isn't spread through the code but is contained in a routine. The error won't affect other code because only one routine will have to be fixed—other code won't be touched. If you find a better, simpler, or more reliable algorithm, it will be easier to replace the old algorithm if it has been isolated into a routine. During development, it will be easier to try several designs and use the one that works best.

Isolating use of nonstandard language functions. Most language implementations contain handy, nonstandard extensions. Using the extensions is a double-edged sword because they might not be available in a different environment, whether the different environment is different hardware, a different vendor's implementation of the language, or a new version of the language from the same vendor. If you use the extensions, build routines of your own that act as gateways to them. Then you can replace the vendor's nonstandard routines with custom-written ones if you need to.

CROSS-REFERENCE For examples of putting complicated tests into boolean functions, see "Making Complicated Expressions Simple" in Section 17.1.

Simplifying complicated boolean tests. Understanding complicated boolean tests in detail is rarely necessary for understanding program flow. Putting such a test into a function makes the code more readable because (1) the details of the test are out of the way and (2) a descriptive function name summarizes the purpose of the test.

Giving the test a function of its own emphasizes its significance. It encourages extra effort to make the details of the test readable inside its function. The result is that both the main flow of the code and the test itself become clearer.

For the sake of modularization? Absolutely not. With so many good reasons for putting code into a routine, this one is unnecessary. In fact, some jobs are performed better in a single large routine. (The best length for a routine is discussed in Section 5.5, "How Long Can a Routine Be?")

Operations That Seem Too Simple to Put into Routines

KEY POINT

One of the strongest mental blocks to creating effective routines is a reluctance to create a simple routine for a simple purpose. Constructing a whole routine to contain two or three lines of code might seem like overkill. But experience shows how helpful a good small routine can be.

Small routines offer several advantages. One is that they improve readability. I once had the following single line of code in about a dozen places in a program:

```
Points = DeviceUnits * ( POINTS_PER_INCH / DeviceUnitsPerInch() )
```

This is not the most complicated line of code you'll ever read. Most people would eventually figure out that it converts a measurement in device units to a measurement in points. They would see that each of the dozen lines did the same thing. It could have been clearer, however, so I created a well-named routine to do the conversion in one place:

```
DeviceUnitsToPoints( DeviceUnits Integer ): Integer;
begin
   DeviceUnitsToPoints = DeviceUnits *
                         (  POINTS_PER_INCH / DeviceUnitsPerInch()  )
end
```

When the routine was substituted for the inline code, the dozen lines of code all looked more or less like this one:

```
Points = DeviceUnitsToPoints( DeviceUnits )
```

which was more readable—even approaching self-documenting.

This example hints at another reason to put small operations into functions: Small operations tend to turn into larger operations. I didn't know it when I wrote the routine, but under certain conditions and when certain devices were active, *DeviceUnitsPerInch()* returned *0*. That meant I had to account for division by zero, which took three more lines of code:

```
DeviceUnitsToPoints( DeviceUnits: Integer ): Integer;
begin
   if ( DeviceUnitsPerInch() <> 0 ) then
      DeviceUnitsToPoints = DeviceUnits *
                            ( POINTS_PER_INCH / DeviceUnitsPerInch() )
   else
      DeviceUnitsToPoints = 0
end
```

If that original line of code had still been in a dozen places, the test would have been repeated a dozen times, for a total of 36 new lines of code. A simple routine reduced the 36 new lines to 3.

Summary of Reasons to Create a Routine

Here's a summary list of the valid reasons for creating a routine:

- Reducing complexity
- Avoiding duplicate code
- Limiting effects of changes
- Hiding sequences
- Improving performance
- Making central points of control
- Hiding data structures
- Hiding global data
- Hiding pointer operations
- Promoting code reuse
- Planning for a family of programs
- Making a section of code readable
- Improving portability
- Isolating complex operations
- Isolating use of nonstandard language functions
- Simplifying complicated boolean tests

5.2 Good Routine Names

CROSS-REFERENCE For details on naming variables, see Chapter 9, "The Power of Data Names."

A good name for a routine clearly describes everything the routine does. Here are guidelines for creating effective routine names.

For a procedure name, use a strong verb followed by an object. A procedure with functional cohesion usually performs an operation on an object. The name should reflect what the procedure does, and an operation on an object implies a verb-plus-object name. *PrintReport(), CalcMonthlyRevenues(), CheckOrderInfo(),* and *RepaginateDocument()* are samples of good procedure names.

In object-oriented languages, you don't need to include the name of the object in the procedure name because the object itself is included in the call. You invoke routines with statements like *Report.Print(), OrderInfo.Check(),* and *MonthlyRevenues.Calc().* Names like *Report.PrintReport()* are redundant.

CROSS-REFERENCE For the distinction between procedures and functions, see Section 5.8, "Considerations in the Use of Functions," later in this chapter.

For a function name, use a description of the return value. A function returns a value, and the function should be named for the value it returns. For example, *cos(), NextCustomerId(), PrinterReady(),* and *CurrentPenColor()* are all good function names that indicate precisely what the functions return.

Avoid meaningless or wishy-washy verbs. Some verbs are elastic, stretched to cover just about any meaning. Routine names like *HandleCalculation(), PerformServices(), ProcessInput(),* and *DealWithOutput()* don't tell you what the routines do. At the most, these names tell you that the routines have something to do with calculations, services, input, and output. The exception would be when the verb "handle" was used in the specific technical sense of handling an event.

KEY POINT

Sometimes the only problem with a routine is that its name is wishy-washy; the routine itself might actually be well designed. If *HandleOutput()* is replaced with *FormatAndPrintOutput()*, you have a pretty good idea of what the routine does.

In other cases, the verb is vague because the operations performed by the routine are vague. The routine suffers from a weakness of purpose, and the weak name is a symptom. If that's the case, the best solution is to restructure the routine and any related routines so that they all have stronger purposes and stronger names that accurately describe them.

Describe everything the routine does. In the routine's name, describe all the outputs and side effects. If a routine computes report totals and sets a global variable that indicates all data is ready for printing, *ComputeReportTotals()* is not an adequate name for the routine. *ComputeReportTotalsAndSetPrinting-ReadyVar()* is an adequate name but is too long and silly. If you have routines with side effects, you'll have many long, silly names. The cure is not to use less-descriptive routine names; the cure is to program so that you cause things to happen directly rather than with side effects.

CROSS-REFERENCE For details on creating good variable names, see Chapter 9, "The Power of Data Names."

Make names of routines as long as necessary. Research shows that the optimum average length for a variable name is 9 to 15 characters. Routines tend to be more complicated than variables, and good names for them tend to be longer. Michael Rees of the University of Southampton thinks that an average of 20 to 35 characters is a good nominal length (Rees 1982). An average length of 15 to 20 characters is probably more realistic, but clear names that happened to be longer would be fine.

Establish conventions for common operations. In some systems, it's important to distinguish among different kinds of operations. A naming convention is often the easiest and most reliable way of indicating these distinctions. In development of the OS/2 Presentation Manager, for example, the routine names had a *Get* prefix for destructive input and a *Query* prefix for nondestructive input. Thus, *GetInputChar()* returned the current input character and cleared the input buffer. *QueryInputChar()* also returned the current input character, but it left the input buffer intact.

5.3 Strong Cohesion

CROSS-REFERENCE For a discussion of module cohesion, see Section 6.1, "Modularity: Cohesion and Coupling."

Cohesion refers to how closely the operations in a routine are related. Some programmers prefer the concept "strength": How strongly related are the operations in a routine? A function like *sin()* is perfectly cohesive because the whole routine is dedicated to performing one function. A function like *SinAndTan()* has lower cohesion because it tries to do more than one thing. The goal is to have each routine do one thing well and not do anything else.

HARD DATA

The payoff is higher reliability. One study of 450 Fortran routines found that 50 percent of the highly cohesive routines were fault free, whereas only 18 percent of routines with low cohesion were fault free (Card, Church, and Agresti 1986). Another study of 450 routines (not the same 450 routines, regardless of how unusual the coincidence is) found that routines with the highest coupling-to-strength ratios had 7 times as many errors as those with the lowest coupling-to-strength ratios and were 20 times as costly to fix (Selby and Basili 1991).

Discussions about cohesion typically refer to several levels of cohesion. Understanding the concepts is more important than remembering specific terms. Use the concepts as aids in thinking about how to make routines as cohesive as possible.

Acceptable Cohesion

CROSS-REFERENCE The idea of cohesion is a part of structured design. For more information on structured design, see Section 7.2, "Structured Design."

The idea of cohesion was introduced in a paper by Wayne Stevens, Glenford Myers, and Larry Constantine (1974). Some of the ideas have evolved since then, and here are the levels of cohesion that are now generally considered to be acceptable:

Functional cohesion. Functional cohesion is the strongest and best kind of cohesion, occurring when a routine performs one and only one operation. Examples of highly cohesive routines include *sin()*, *GetCustomerName()*, *EraseFile()*, *CalcLoanPayment()*, and *GetIconLocation()*. Of course, this evaluation of their cohesiveness assumes that the routines do what their names say they do. If they do anything else, they are less cohesive and poorly named.

Sequential cohesion. Sequential cohesion occurs when a routine contains operations that must be performed in a specific order, that share data from step to step, and that don't make up a complete function when done together. Suppose that a program has five operations: Open File, Read File, Perform Calculations, Output Results, and Close File. Suppose they are organized into two routines, *DoStep1()* to do the Open File, Read File, and Perform Calculations operations, and *DoStep2()* to do the Output Results and Close File operations. Both *DoStep1()* and *DoStep2()* have sequential cohesion because breaking up the operations that way doesn't create independent functions.

If, instead, a routine called *GetFileData()* did the Open File and Read File operations, that routine would have functional cohesion. If the operations work together to perform a single function, they constitute a routine with functional cohesion. In practice, if you can come up with a strong verb-plus-object name for a routine, the cohesion is probably functional rather than sequential. It's hard to create a respectable name for a routine with sequential cohesion, so a wishy-washy name like *DoStep1()* is a hint to redesign and try for functional cohesion.

Communicational cohesion. Communicational cohesion occurs when operations in a routine make use of the same data and aren't related in any other way. For example, *GetNameAndChangePhoneNumber()* would have communicational cohesion if the name and phone number were both in, say, a customer record. The routine does two things rather than just one, so it doesn't have functional cohesion. The name and phone number are both in the customer record, and they don't need to be dealt with in any particular order, so the routine doesn't have sequential cohesion.

This level of cohesion is still acceptable. On a practical level, a system could frequently need to get a name and change a phone number at the same time. A system composed of routines like this would be pretty eccentric but would

still be clean enough that it would be maintainable. On an aesthetic level, it would be far from the ideal routine, which does one thing and does it well.

FURTHER READING For a critical view of temporal cohesion, see *Structured Design* (Yourdon and Constantine 1979) or *The Practical Guide to Structured Systems Design* (Page-Jones 1988).

Temporal cohesion. Temporal cohesion occurs when operations are combined into a routine because they are all done at the same time. Typical examples would be *Startup()*, *CompleteNewEmployee()*, and *Shutdown()*. Some programmers consider temporal cohesion to be unacceptable because it's sometimes associated with bad programming practices such as having a hodgepodge of code in a *Startup()* routine.

To avoid this problem, think of temporal routines as organizers of other events. The *Startup()* routine, for example, might read a configuration file, initialize a scratch file, set up a memory manager, and show an initial screen. To make it most effective, have the temporally cohesive routine call other routines to perform specific activities rather than performing the operations directly itself. That way, it will be clear that the point of the routine is to orchestrate activities rather than to do them directly.

Unacceptable Cohesion

The remaining kinds of cohesion are generally unacceptable. They result in code that's poorly organized, hard to debug, and hard to modify. If a routine has bad cohesion, it's better to put effort into a good rewrite than into a pinpoint diagnosis of the problem. Knowing what to avoid can be useful, however, so here are the unacceptable kinds of cohesion:

Procedural cohesion. Procedural cohesion occurs when operations in a routine are done in a specified order. Unlike in sequential cohesion, the sequential operations in procedural cohesion don't share the same data. For example, if your users like reports to be printed in a certain order, you might have a routine that prints a revenue report, an expense report, a list of employee phone numbers, and invitations to the company picnic. This kind of routine is difficult to name specifically, and its vague routine name would be a tip-off.

CROSS-REFERENCE Module packaging is discussed in more detail in Chapter 6, "Three out of Four Programmers Surveyed Prefer Modules."

Logical cohesion. Logical cohesion occurs when several operations are stuffed into the same routine and one of the operations is selected by a control flag that's passed in. It's called logical cohesion because the control flow or "logic" of the routine is the only thing that ties the operations together—they're all in a big *if* statement or *case* statement together. It isn't because the operations are logically related in any other sense.

One example would be an *InputAll()* routine that input customer names, employee time-card information, or inventory data depending on a flag passed to the routine. Other examples would be *ComputeAll()*, *EditAll()*, *PrintAll()*, and *SaveAll()*. The main problem with such routines is that you shouldn't need to pass in a flag to control another routine's processing. Instead of having a routine that does one of three distinct operations, depending on a flag passed to it, it's cleaner to have three routines, each of which does one distinct operation. If the operations use some of the same code or share data, the code should be put into a lower-level routine and the routines should be packaged into a module.

It's usually all right, however, to create a logically cohesive routine if its code consists solely of a series of *if* or *case* statements and calls to other routines. In such a case, if the routine's only function is to dispatch commands and it doesn't do any of the processing itself, that's usually a good design. The technical term for this kind of routine is "transaction center." A transaction center is often used as an event-handler routine in message-based environments such as the Apple Macintosh and Microsoft Windows.

Coincidental cohesion. Coincidental cohesion occurs when the operations in a routine have no discernable relationship to each other. Other good names are "no cohesion" or "chaotic cohesion." The low-quality Pascal routine at the beginning of the chapter had coincidental cohesion.

None of these terms are magical or sacred. Learn the ideas rather than the terminology. It's nearly always possible to write routines with functional cohesion, so focus your attention on functional cohesion for maximum benefit.

Examples of Cohesion

Here are examples of the several kinds of cohesion, both good and bad:

Example of functional cohesion. A routine calculates an employee's age, given a birth date. The routine does one thing and one thing only, so it has functional cohesion.

Example of sequential cohesion. A routine calculates an employee's age and time to retirement, given a birth date. If the routine calculates the age and then uses that result to calculate the employee's time to retirement, it has sequential cohesion. If the routine calculates the age and then calculates the time to retirement in a completely separate computation that happens to use the same birth-date data, it has only communicational cohesion.

Deciding what kind of weak cohesion the routine has, however, is less important than determining how to make it better. How would you make the routine functionally cohesive? You'd create separate routines to compute an employee's age given a birth date, and time to retirement given a birth date. The time-to-retirement routine could call the age routine. They'd both have functional cohesion. Other routines could call either routine or both.

Example of communicational cohesion. A routine prints a summary report and when it's done, reinitializes the summary data passed in to it. The routine has communicational cohesion because the two operations are related only by the fact that they use the same data.

More important than identifying the routine as one with communicational cohesion is deciding how to make it functionally cohesive. The summary data should be reinitialized close to where it's created, which shouldn't be in the report-printing routine. Split the operations into individual routines. The first prints the report. The second reinitializes the data, close to the code that creates or modifies the data. Call both routines from the higher-level routine that originally called the communicationally cohesive routine.

Example of logical cohesion. A routine prints a quarterly expense report, a monthly expense report, or a daily expense report, depending on the value of a control flag that's passed in. The routine has logical cohesion because its internal logic is controlled by the flag that's passed in. The routine certainly doesn't do one thing and do it well.

How do you make this routine functionally cohesive? Create three routines: one that prints a quarterly expense report, one that prints a monthly expense report, and one that prints a daily expense report. Modify the original routine so that it calls one of the new routines, depending on the value of the control flag that's passed in. The calling routine, containing no code of its own except the code that calls the appropriate routine, now has functional cohesion. Each of the three routines called also has functional cohesion.

Incidentally, the routine that does nothing but call the appropriate printing routine is another example of a transaction center. It's good to name a transaction center something like *DispatchReportPrinting()*—something with "Dispatch" or "Control" in the name—so that it's clear that the transaction center doesn't do anything on its own and that it's not supposed to.

Another example of logical cohesion. A routine prints an expense report, enters a new employee name, or backs up a database, depending on the value of a control flag that's passed in. The routine has logical cohesion only—although the cohesion in this example seems more like *illogical* cohesion.

To achieve functional cohesion, break up the separate functions into separate routines. The operations are so unrelated that the code that calls the routine probably needs to be reorganized too. Once you have separate routines to call, reorganizing the calling code is easier.

Example of procedural cohesion. A routine gets an employee name, then an address, and then a phone number. The order of these operations is important only because it matches the order in which the user is asked for the data on the input screen. Another routine gets the rest of the employee data. The routine has procedural cohesion because it puts a set of operations in a specified order and the operations don't need to be combined for any other reason.

The more important question of how to achieve functional cohesion is answered in the usual way. Put the separate operations into their own routines. Make sure that the calling routine has a single, complete job: *GetEmployeeData()* rather than *GetFirstPartOfEmployeeData()*. You'll probably need to modify the routines that get the rest of the data too. It's common to modify two or more original routines before you achieve functional cohesion in any of them.

Example of functional and temporal cohesion. A routine performs all the processing necessary to complete a transaction—getting a confirmation from the user, saving a record to a database, clearing data fields, and incrementing counters. The routine has functional cohesion. It does one job, completing a transaction, and that's all it does. It would also be accurate to describe the routine as having temporal cohesion, but if you can classify a routine at more than one level of cohesion, it's considered to have the strongest level of cohesion at which it can be classified.

This example raises the issue of choosing a name that describes the routine at the right level of abstraction. You could decide to name the routine *ConfirmEntryAndAdjustData()*, which would imply that the routine had only coincidental cohesion. If you named it *CompleteTransaction()*, however, it would be clear that it had a single purpose and clear that it had functional cohesion.

Example of procedural, temporal, or possibly logical cohesion. A routine performs the first five steps in a complicated mathematical operation and returns the intermediate results of the five steps to the calling routine. Since it takes several hours to do the first five steps, the routine stores the intermediate results in a file in case the system crashes. Then the routine checks the disk to see whether it has enough space to store the final results of the calculations and returns the disk status to the calling routine along with the intermediate results.

This routine probably has procedural cohesion, but you could argue persuasively for temporal or even logical cohesion. The real issue isn't precisely identifying the routine's brand of imperfect cohesion. It's improving the routine's cohesion.

The original routine is a strange collection of activities that's far from being functionally cohesive. At a minimum, the calling routine should call not one but several separate routines to (1) do the first five steps in the calculation, (2) store the intermediate results in a file, and (3) determine available disk space. If the calling routine is named, say, *ComputeExtravagantNumber()*, it shouldn't be writing the intermediate result to disk, and it absolutely should not be checking disk space for some later activity. It should just compute the number. A good reorganization is bound to affect routines at least one or two levels above it. A better design of the job would look like the diagram in Figure 5-1.

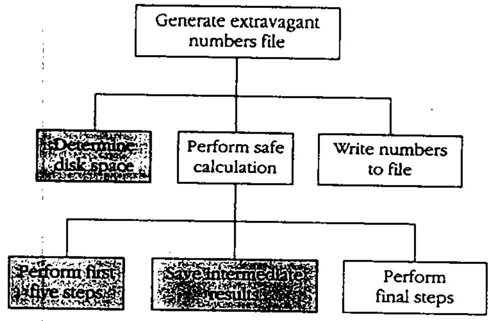


Figure 5-1. *An example of decomposition to achieve functional cohesion.*

The activities covered by the original routine are shown in shaded boxes. They should be at different levels in the organization, which is why so much reorganization is required to put them into more appropriate routines. It's not unusual to need several functionally cohesive routines to rescue one routine with poor cohesion.

5.4 Loose Coupling

CROSS-REFERENCE The idea of coupling is a part of structured design. For more information on structured design, see Section 7.2, "Structured Design."

The degree of coupling refers to the strength of a connection between two routines. Coupling is a complement to cohesion. Cohesion describes how strongly the internal contents of a routine are related to each other. Coupling describes how strongly a routine is related to other routines. The goal is to create routines with internal integrity (strong cohesion) and small, direct, visible, and flexible relations to other routines (loose coupling).

Good coupling between routines is loose enough that one routine can easily be called by other routines. Model railroad cars are coupled by opposing hooks that latch when pushed together. Connecting two cars is easy—you just push the cars together. Imagine how much more difficult it would be if you had to screw things together, or connect a set of wires, or if you could connect only certain kinds of cars to certain other kinds of cars. The coupling of model railroad cars works because it's as simple as possible. In software, make the connections among routines as simple as possible.

Try to create routines that depend little on other routines. Make them detached, as business associates are, rather than attached, as Siamese twins are. A function like *sin()* is loosely coupled because everything it needs to know is passed in to it with one value representing an angle in degrees. A function such as *InitVars(var 1, var2, var3, ..., varN)* is more tightly coupled because the calling routine virtually knows what is happening inside it. Two routines that depend on each other's use of the same global data are even more tightly coupled.

Coupling Criteria

FURTHER READING
These criteria are
ted from *Structured
Design* (Yourdon and
ntine 1979) and *The
tical Guide to Struc-
red Systems Design*
(Page-Jones 1988).

Here are several criteria to use in evaluating coupling between routines:

Size. Size refers to the number of connections between routines. With coupling, small is beautiful because it's less work to connect other routines to a routine with a smaller interface. A routine that takes one parameter is more loosely coupled to routines that call it than a routine that takes six parameters. A routine that takes an integer parameter is more loosely coupled to the routines that call it than one that takes a 10-element array or a structured variable. A routine that uses one global variable is more loosely connected to other routines in a program than a routine that uses twelve.

Intimacy. Intimacy refers to the directness of the connection between two routines. The more intimate the connection, the better. The most intimate connection between routines is a parameter in a parameter list. The routines communicate directly. The parameter is like a kiss on the lips between the routines. A less intimate connection is found between routines that work with the same piece of global data. They communicate less directly. Global data is like a love letter between routines—it might go where you want it to, or it might get lost in the mail. The least intimate connection is found between routines that work with the same database records or files. They both need the data but are too shy to acknowledge each other. A shared data file is like passing a note in class that says "Do you like me? Check ☐yes or ☐no."

Visibility. Visibility refers to the prominence of the connection between two routines. Programming is not like being in the CIA; you don't get credit for being sneaky. It's more like advertising; you get lots of credit for making your connections as blatant as possible. Passing data in a parameter list is making an obvious connection and is therefore good. Modifying global data so that another routine can use it is making a sneaky connection and is therefore bad. Documenting the global-data connection makes it more obvious and is slightly better.

Flexibility. Flexibility refers to how easily you can change the connections between routines. Ideally, you want something more like the snap-in modular connector on your phone than like bare wire and a soldering gun. Flexibility is partly a product of the other coupling characteristics, but it's a little different too. For example, suppose you have a routine that looks up the first supervisor an employee ever had, given a hiring date and a hiring department. Name the

routine *LookupFirstSupervisor()*. Suppose you also have in another routine a structured variable called *EmpRec* that includes the hiring date and the hiring department, among other things, and that the second routine passes the variable to *LookupFirstSupervisor()*.

From the point of view of the other criteria, the two routines would look pretty loosely coupled. The *EmpRec* connection between the first and second routines is a smack on the lips, in public, and there's only one connection. Now suppose that you need to call the *LookupFirstSupervisor()* routine from a third routine that doesn't have an *EmpRec* but does have a hiring date and a hiring department. Suddenly *LookupFirstSupervisor()* looks less friendly, unwilling to associate with the new routine.

For the third routine to call *LookupFirstSupervisor()*, it has to know about the *EmpRec* data structure. It could dummy up an *EmpRec* variable with only two fields, but that would require internal knowledge of *LookupFirstSupervisor()*, namely that those are the only fields it uses. Such a solution would be a kludge, and an ugly one. The second option would be to modify *LookupFirstSupervisor()* so that it would take hiring date and hiring department instead of *EmpRec*. In either case, the original routine turns out to be a lot less flexible than it seemed to be at first.

The happy ending to the story is that an unfriendly routine can make friends if it's willing to be flexible—in this case, by changing to take hiring date and hiring department specifically instead of *EmpRec*.

In short, the more easily other routines can call a routine, the more loosely coupled it is, and that's good because it's more flexible and maintainable. In creating a system structure, break up the program along the lines of minimal interconnectedness. If a program were a piece of wood, you would try to split it with the grain.

Levels of Coupling

The traditional names for the levels of coupling are unintuitive, so the following descriptions of the classes of coupling use good, mnemonic names along with the confusing traditional names. Examples of good and bad coupling follow the coupling descriptions.

Simple-data coupling. Two routines are simple-data–coupled if all the data passed between them is nonstructured and it's all passed through a parameter list. This is often called "normal coupling" and is the best kind.

Data-structure coupling. Two routines are data-structure–coupled if the data passed between them is structured and is passed through a parameter list.

This is sometimes called "stamp coupling" (for reasons that have always been opaque to me) and is fine if used properly. Its main difference from simple-data coupling is that structured data is involved.

Control coupling. Two routines are control-coupled if one routine passes data to the other that tells the second routine what to do. Control coupling is as undesirable as the logical cohesion it's associated with. It generally requires that the calling routine know about the internal workings of the called routine.

CROSS-REFERENCE Coupling with global data is distinct from coupling with module data. For details on module-data coupling, see "Module data mistaken for global data" in Section 6.2 and discussions throughout Chapter 6.

Global-data coupling. Two routines are global-data–coupled if they make use of the same global data. This is also called "common coupling" or "global coupling." If use of the data is read-only, the practice is tolerable. Generally, however, global-data coupling is undesirable because the connection between routines is neither intimate nor visible. The connection is so easy to miss that you could refer to it as information hiding's evil cousin—"information losing."

Pathological coupling. Two routines are pathologically coupled if one uses the code inside another or if one alters the local data used inside another. This is also called "content coupling." This kind of coupling is unacceptable because it fails all the criteria of size, intimacy, visibility, and flexibility. One routine's executing another routine's code makes a large connection between the two routines. Although it's a close connection, the connection between the two routines is not intimate. Altering another routine's data is more like stabbing it in the back. The connection is invisible from the point of view of the routine being stabbed. And the connection is not flexible; it depends on one routine's having detailed knowledge of the internal affairs of another. Most structured languages have scoping rules that prevent pathological coupling. It's still possible in Basic and assembler, however, so if you're working in either of those languages, watch out!

Each of these kinds of coupling is shown in Figure 5-2.

Examples of Coupling

Here are examples of the kinds of coupling, both good and bad:

Example of simple-data coupling. One routine passes a variable containing an angle in degrees to a *tan()* routine.

Another example of simple-data coupling. One routine passes five variables to another routine, including a name, an address, a phone number, a birth date, and a social security number.

Example of acceptable data-structure coupling. One routine passes an *EmpRec* variable to another routine. *EmpRec* is a structured variable that includes a

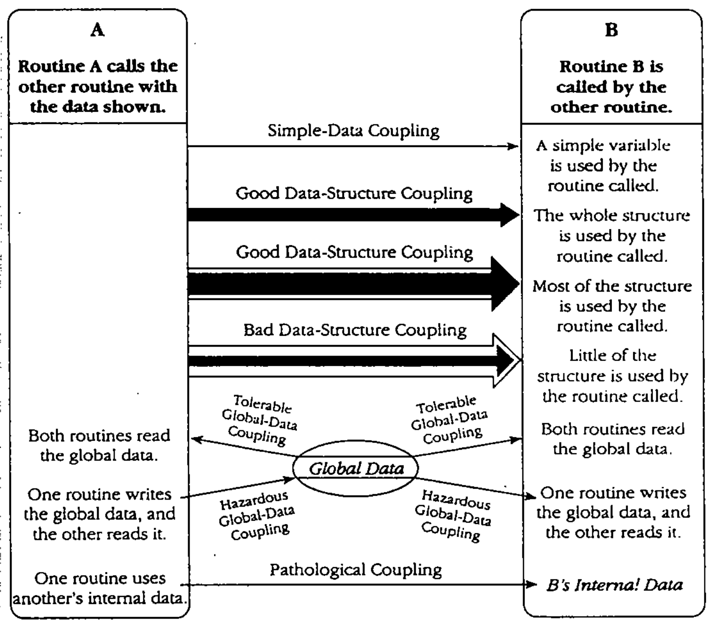

A
Routine A calls the other routine with the data shown.
B
Routine B is called by the other routine.
Simple-Data Coupling
A simple variable is used by the routine called.
Good Data-Structure Coupling
The whole structure is used by the routine called.
Good Data-Structure Coupling
Most of the structure is used by the routine called.
Bad Data-Structure Coupling
Little of the structure is used by the routine called.
Both routines read the global data.
Tolerable Global-Data Coupling
Tolerable Global-Data Coupling
Both routines read the global data.
Global Data
One routine writes the global data, and the other reads it.
Hazardous Global-Data Coupling
Hazardous Global-Data Coupling
One routine writes the global data, and the other reads it.
One routine uses another's internal data.
Pathological Coupling
B's Internal Data

Figure 5-2. *Some kinds of coupling.*

name, an address, a phone number, a birth date, and a social security number. The called routine uses all five fields.

Example of unacceptable data-structure coupling. One routine passes the same *EmpRec* variable to another routine. The called routine uses only the birth date and phone number. This is data-structure coupling also, but it's not a good use of it. Passing the birth date and phone number as simple variables would make the connection more flexible and the real connection of the two specific fields more visible.

CROSS-REFERENCE For details on creating good routine parameter lists, see Section 5.7, "How to Use Routine Parameters."

Example of questionable data-structure coupling. One routine passes an *OfficeRec* variable to another routine. *OfficeRec* has 27 fields, and the called routine uses 16 of them. This is data-structure coupling. But is it good data-structure coupling? It's a close call. Passing *OfficeRec* makes it obvious that the connection is large; passing 16 individual parameters would just be an awkward way of showing the same thing. If the called routine used only 6 or 7 of *OfficeRec's* fields, it would be a good idea to pass them individually.

If it made sense to structure *OfficeRec's* data more, so that the 16 fields used by the called routine were contained in a substructure or two, that would be the cleanest solution.

Example of simple-data or possibly data-structure coupling. A routine passes an *EraseFile()* routine a string containing the name of the file to erase. This is probably simple-data coupling. You might argue that it's data-structure coupling because a string is a data structure. I say, "You say tom-AY-to, I say tom-AH-to." Both are fine, and the distinction isn't important.

Example of control coupling. One routine passes a flag to another that tells it whether to print a monthly report, a quarterly report, or an annual report.

Example of unacceptable global-data coupling. A routine modifies an entry in a table stored as a global variable. The table is indexed by employee ID. The routine then calls another routine and passes it the employee ID as a parameter. The called routine uses the employee ID to read the global-data table. This is a classic example of global-data coupling. (Although merely passing the employee ID would constitute simple-data coupling, the first routine's modification of the table entry earns the coupling the worst rating.)

CROSS-REFERENCE For guidelines on using global data safely, see Section 10.6, "Global Variables."

Example of acceptable global-data coupling. One routine passes an employee ID to another routine. Both routines use the employee ID to read the corresponding employee name from a global table. Neither routine changes any global data. This is often described as global-data coupling, but it's more like simple-data coupling or "tolerable global-data coupling." Unlike the routines in the example in which one routine modifies data used by another, these routines aren't connected to each other by the global data. With respect to each other, their read-only use of the same global data is benign. There is a world of difference between two routines conspiring to hide a connection by using global data (as in the previous example) and two routines coincidentally reading a value from the same global table.

Example of pathological coupling. An assembler routine knows the address of a table declared as a local variable inside another routine. It uses the address to modify the table directly. The address is not passed between the routines as a parameter.

Another example of pathological coupling. A Basic routine uses a *GOSUB* to execute a piece of code within another routine.

KEY POINT

The whole point of good coupling is that an effective routine provides an additional level of abstraction—once you write it, you can take it for granted. It reduces overall program complexity and allows you to focus on one thing at a time. If using a routine requires you to focus on more than one thing at once—knowledge of its internal workings, modification to global data, uncertain functionality—the abstractive power is lost and the value of the routine is reduced or eliminated. Routines are primarily an intellectual tool for reducing complexity. If they're not making your job simpler, they're not doing their job.

5.5 How Long Can a Routine Be?

On their way to America, the Pilgrims argued about the best maximum length for a routine. After arguing about it for the entire trip, they arrived at Plymouth Rock and started to draft the Mayflower Compact. They still hadn't settled the maximum-length question, and since they couldn't disembark until they'd signed the compact, they gave up and didn't include it. The result has been an interminable debate ever since about how long a routine can be.

The theoretical best maximum length is often described as one or two pages of program listing, 66 to 132 lines. In this spirit, in the 1970s IBM limited routines to 50 lines, and TRW limited them to two pages (McCabe 1976). The evidence in favor of short routines, however, is very thin, and the evidence in favor of longer routines is compelling. Consider the following:

HARD DATA

- A study by Basili and Perricone found that routine size was inversely correlated with errors; as the size of routines increased (up to 200 lines of code), the number of errors per line of code decreased (1984).
- Another study found that routine size was not correlated with errors, even though structural complexity and amount of data were correlated with errors (Shen et al. 1985).
- A 1986 study found that small routines (32 lines of code or fewer) were not correlated with lower cost or fault rate (Card, Church, and Agresti 1986; Card and Glass 1990). The evidence suggested that larger routines (65 lines of code or more) were cheaper to develop per line of code.
- An empirical study of 450 routines found that small routines (those with fewer than 143 source statements, including comments) had 23 percent more errors per line of code than larger routines (Selby and Basili 1991).
- A study of upper-level computer-science students found that students' comprehension of a program that was super-modularized into routines about 10 lines long was no better than their comprehension of a program that had no routines at all (Conte, Dunsmore, and Shen 1986). When the program was broken into routines of moderate length (about 25 lines), however, students scored 65 percent better on a test of comprehension.
- A recent study found that code needed to be changed least when routines averaged 100 to 150 lines of code (Lind and Vairavan 1989).

Where does all this leave the question of routine length? Well, if you're a manager, don't tell your programmers to keep their routines to one page—the studies cited and your programmers' experience will tell them you're full of hooey. Second, if you think it makes sense to make a certain routine 100, 150, or 200 lines long, it's probably all right to do so. Current evidence says that routines of such length are no more error prone than shorter routines and are likely to be cheaper to develop.

HARD DATA

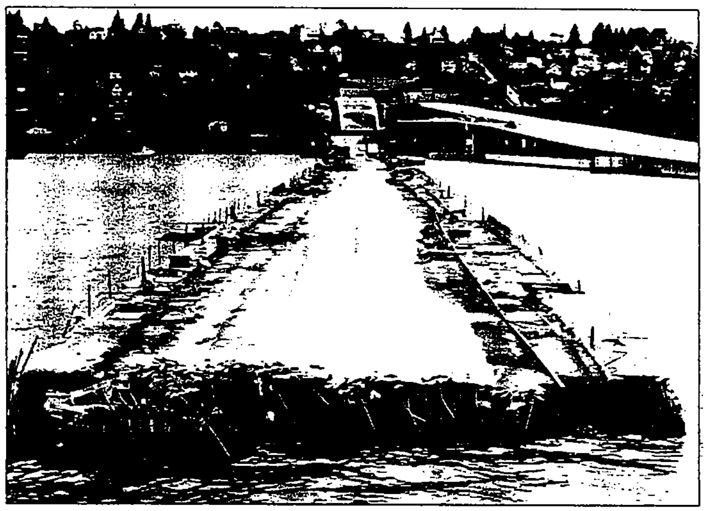
If you want to write routines longer than about 200 lines, be careful. (A line is a noncomment, nonblank line of source code.) None of the studies that reported decreased cost, decreased error rates, or both with larger routines distinguished among sizes larger than 200 lines, and you're bound to run into an upper limit of understandability as you pass 200 lines of code. In a study of the code for IBM's OS/360 operating system and other systems, the most error-prone routines were those that were larger than 500 lines of code. Beyond 500 lines, the error rate tended to be proportional to the size of the routine (Jones 1986a). Moreover, an empirical study of a 148,000-line program found that routines with fewer than 143 source statements were 2.4 times less expensive to fix than larger routines (Selby and Basili 1991).

5.6 Defensive Programming

KEY POINT

Defensive programming doesn't mean being defensive about your programming—"It does so work!" The idea is based on defensive driving. In defensive driving, you adopt the mind-set that you're never sure what the other drivers are going to do. That way, you make sure that if they do something dangerous you won't be hurt. You take responsibility for protecting yourself even when it might be the other driver's fault. In defensive programming, the main idea is that if a routine is passed bad data, it won't be hurt, even if the bad data is another routine's fault. More generally, it's the recognition that programs will have problems and modifications, and that a smart programmer will develop code accordingly.

Defensive programming is useful as an adjunct to the other techniques for quality improvement described in this book. The best form of defensive coding is not installing errors in the first place. Using iterative design, writing PDL before code, and having low-level design inspections are all activities that help to prevent installing defects. They should thus be given a higher priority than defensive programming. Fortunately, you can use defensive programming in combination with the other techniques.



Part of the Interstate-90 floating bridge in Seattle sank during a storm because the flotation tanks were left uncovered, they filled with water, and the bridge became too heavy to float. During construction, protecting yourself against the small stuff matters more than you might think.

Using Assertions

An assertion is a function or macro that complains loudly if an assumption isn't true. Use assertions to document assumptions made in the code and to flush out unexpected conditions. An assertion function usually takes two arguments: a boolean expression that describes the assumption that's supposed to be true and a message to print if it's not. Here's what a Pascal assertion would look like if the variable *Denominator* were expected to be nonzero:

Pascal Example of an Assertion

```
Assert( Denominator <> 0, 'Denominator is unexpectedly equal to 0.' );
```

This assertion asserts that *Denominator* is not equal to *0*. The first argument, *Denominator* <> *0*, is a boolean expression that evaluates to *True* or *False*. The second argument is a message to print if the first argument is *False*—that is, if the assertion is false.

Even if you don't want your users to see assertion messages in production code, assertions are handy during development and maintenance. During development, assertions flush out contradictory assumptions, unexpected conditions, bad values passed to routines, and so on. During maintenance, they indicate whether modifications have damaged other parts of the code.

Assertion procedures are easy to write. Here's one in Pascal:

Pascal Example of an Assertion Procedure

```
Procedure Assert
   (
   Assertion:  boolean;
   Message:    string
   );
begin
if ( not Assertion )
   begin
   writeln( Message );
   writeln( 'Stopping the program.' );
   halt( FATAL_ERROR )
   end
end;
```

e halt() statement erminates the pro- am. It isn't part of andard Pascal, but most implementa- ions of Pascal have something similar.

Once you've written a procedure like this, you can call it with statements like the first one above.

Here are some guidelines for using assertions:

Use a preprocessor macro for assertions, if you have a preprocessor. Using a preprocessor for assertions makes it easy to use assertions during development and to take them out of production code.

Avoid putting executable code in assertions. Putting code in an assertion raises the possibility that the compiler will eliminate the code when you turn off the assertions. Suppose you have an assertion like this:

CROSS-REFERENCE
u could view this as one many problems associ- ed with putting multiple tements on one line. For re examples, see "Using Only One Statement per Line" in Section 18.5.

Pascal Example of a Dangerous Use of an Assertion

```
Assert( FileOpen( InputFile ) <> NULL, 'Couldn't open input file' );
```

The problem with this code is that if you don't compile the assertions, you don't compile the code that performs the file open. Put executable statements on their own lines, assign the results to status variables, and test the status variables instead. Here's an example of a safe use of an assertion:

Pascal Example of a Safe Use of an Assertion

```
FileStatus := FileOpen( InputFile );
Assert( FileStatus <> NULL, 'Couldn't open input file' );
```

Garbage In Does Not Mean Garbage Out

A good program never puts out garbage, regardless of what it takes in. A good program uses "garbage in, nothing out"; "garbage in, error message out"; or "no garbage allowed in" instead. By today's standards, "garbage in, garbage out" is the mark of a sloppy program.

Check the values of all data input from external sources. When getting data from a file or a user, check to be sure that the data falls in the allowable range. Make sure that numeric values are within tolerances and that strings are short enough to handle. Comment assumptions about acceptable input ranges in the code.

CROSS-REFERENCE
If you program using pointers, it's an especially good idea to check any that are passed as input parameters. For details on the kinds of checks you can make on pointers, see Section 11.9, "Pointers."

Check the values of all routine input parameters. Checking the values of routine input parameters is essentially the same as checking data that comes from an external source, except that the data comes from another routine instead of from a file or a user.

Here's an example of a routine that checks its input parameters:

C Example of Using an Assertion to Check an Input Parameter

```
float tan
   (
   float OppositeLength,
   float AdjacentLength
   )
   {
   /* compute tangent of an angle */

   Assert( AdjacentLength != 0, "AdjacentLength detected to be 0." );

   return( OppositeLength / AdjacentLength );
   }
```

This statement calls a routine, Assert(), that prints the message and stops the program if the logical expression isn't true.

Decide how to handle bad parameters. Once you've detected an invalid parameter, what do you do with it? Depending on the situation, you might want to return an error code, return a neutral value, substitute the next piece of valid data and continue as planned, return the same answer as the previous time, use the closest legal value, call an error-processing routine, log a warning message to a file, print an error message, or shut down. You have so many options that you need to be careful not to handle invalid parameters in randomly different ways in different parts of the program. Deciding on a general approach to bad parameters is an architectural decision and should be addressed at the architectural level.

Exception Handling

CROSS-REFERENCE For more details on handling unanticipated es, see "Tips for Using *case* Statements" in Section 14.2.

Use exception handling to draw attention to unexpected cases. Exceptional cases should be handled in a way that makes them obvious during development and recoverable when production code is running. For example, suppose you have a *case* statement that interprets five kinds of events, and you expect to have only five kinds of events. During development, the default case should be used to generate a warning that says "Hey! There's another case here! Fix the program!" During production, however, the default case should do something more graceful, like writing a message to an error-log file. Try to design the program so that you can flip from development mode to production mode without too much trouble.

Anticipating Changes

Change is a fact of life with every program. New versions of old programs require more changes to the existing code base than new code does. But even during development of the first version of a program, you'll add unanticipated features that require changes. Error corrections also introduce changes. Develop the program so that likely changes can be accommodated without an act of Congress. The more likely the change, the less trauma it should cause the program. Hiding the areas in which you anticipate changes is one of the most powerful techniques for minimizing the impact of changes.

CROSS-REFERENCE r details on information hiding, see Section 6.2, "Information Hiding."

Planning to Remove Debugging Aids

Debugging aids are assertions, memory checksums, print statements, and a variety of other coding practices that can help in debugging. If you're writing code for your own use, it might be fine to leave all the debugging code in the program. If you're writing code for commercial use, the performance penalty in size and speed is often prohibitive. If that's the case, plan to avoid shuffling debugging code in and out of a program. Here are several ways to do that.

CROSS-REFERENCE For details on version control, see Section 22.2, "Configuration Management."

Use version control. Version-control tools can build different versions of a program from the same source files. In development mode, you can set the version-control tool to include all the debug code. In production mode, you can set it to exclude any debug code you don't want in the commercial version.

Use a built-in preprocessor. If your programming environment has a preprocessor—as C does, for example—you can include or exclude debug code at the flick of a compiler switch. You can use the preprocessor directly or by writing a macro that works with preprocessor definitions. Here's an example of writing code using the preprocessor directly:

C Example of Using the Preprocessor Directly to Control Debug Code

To include the debugging code, use #define to define the symbol DEBUG. To exclude the debugging code, don't define DEBUG.

```
#define DEBUG
...

#if defined( DEBUG )
/* debugging code */
...

#endif
```

This theme has several variations. Rather than just defining *DEBUG*, you can assign it a value and then test for the value rather than testing whether it's defined. That way you can differentiate between different levels of debug code. You might have some debug code that you want in your program all the time, so you surround that by a statement like *#if DEBUG > 0*. Other debug code might be for specific purposes only, so you can surround it by a statement like *#if DEBUG == POINTER_ERROR*. In other places, you might want to set debug levels, so you could have statements like *#if DEBUG > LEVEL_A*.

If you don't like having *#if defined()*s spread throughout your code, you can write a preprocessor macro to accomplish the same task. Here's an example:

C Example of Using a Preprocessor Macro to Control Debug Code

```
#define DEBUG

#if defined( DEBUG )
#define DebugCode( code_fragment )   { code_fragment }
#else
#define DebugCode( code_fragment )
#endif
...

DebugCode
   (
   statement 1;
   statement 2;
   ...
   statement n;
   );
...
```

This code is included or excluded depending on whether DEBUG has been defined.

As in the first example of using the preprocessor, this technique can be altered in a variety of ways that make it more sophisticated than completely including all debug code or completely excluding all of it.

CROSS-REFERENCE For more information on processors and direction o sources of information writing one of your own, e "Macro preprocessors" in Section 20.3.

Write your own preprocessor. If a language doesn't include a preprocessor, it's fairly easy to write one for including and excluding debug code. Establish a convention for designating debug code and write your precompiler to follow that convention. For example, in Pascal you could write a precompiler to respond to the keywords *{#BEGIN DEBUG}* and *{#END DEBUG}*. Write a script or batch file to call the preprocessor, and then compile the processed code. You'll save time in the long run, and you won't mistakenly compile the unpreprocessed code.

CROSS-REFERENCE For details on stubs, see "Building Scaffolding to Test Individual Routines" in Section 25.5.

Use debugging stubs. In many instances, you can call a routine to do debugging checks. During development, the routine might perform several operations before control returns to the caller. For production code, you can replace the complicated routine with a stub routine that merely returns control immediately to the caller or performs only a couple of quick operations before returning control. This approach incurs only a small performance penalty, and it's a quicker solution than writing your own preprocessor. Keep both the development and production versions of the routines so that you can switch back and forth during future development and production.

You might start with a routine designed to check pointers that are passed to it:

Pascal Example of a Routine that Uses a Debugging Stub

```
Procedure DoSomething
   (
   Pointer: POINTER_TYPE;
   ...
   );
begin

{ check parameters passed in }
CheckPointer( Pointer );

end;
```

This line calls the routine to check the pointer.

During development, the *CheckPointer()* routine would perform full checking on the pointer. It would be slow but effective. It could look like this:

Pascal Example of a Routine for Checking Pointers During Development

This routine checks any pointer that's assed to it. It can be used during development to perform as many checks as you can bear.

```
Procedure CheckPointer( Pointer: POINTER_TYPE );
begin
   { perform check 1--maybe check that it's not nil }
   { perform check 2--maybe check that its dogtag is legitimate }
   { perform check 3--maybe check that what it points to isn't corrupted }
```

(continued)

```
   ...
   { perform check n--... }
end;
```

When the code is ready for production, you might not want all the overhead associated with this pointer checking. You could swap out the routine above and swap in this routine:

Pascal Example of a Routine for Checking Pointers During Production

This routine just returns immediately to the caller.

```
Procedure CheckPointer( Pointer: POINTER_TYPE );
begin
{ no code; just return to caller }
end;
```

This is not an exhaustive survey of all the ways you can plan to remove debugging aids, but it should be enough to give you an idea for some things that will work in your environment.

Introducing Debugging Aids Early

The earlier you introduce debugging aids, the more they'll help. Typically, you won't go to the effort of writing a debugging aid until after you've been bitten by a problem several times. If you write the aid after the first time, however, or use one from a previous project, it will help throughout the project.

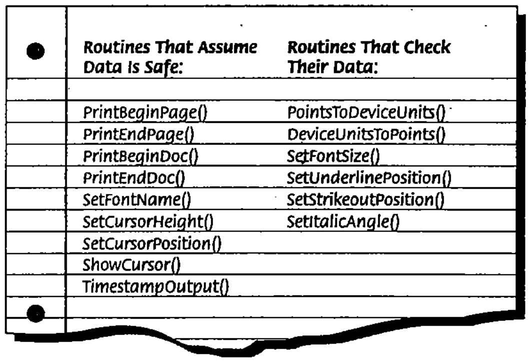
Firewalling to Contain the Damage Caused by Errors

Firewalling is a damage-containment strategy. A firewall in a building keeps a fire from spreading, isolating it in one place. The reason is similar to that for having isolated compartments in the hull of a ship. If the ship runs into an iceberg and pops open the hull, that compartment is shut off and the rest of the ship isn't affected.

Information hiding helps to firewall a program. The less you know about the internal workings of another routine, the fewer assumptions you make about how the routine operates. The fewer the assumptions you make, the less the chance that one of them is wrong.

Loose coupling also helps to firewall a program. The looser the connection between two routines, the less the chance that an error in one routine will affect the other. If the connection is tighter, if one routine borrows code from inside another, chances are high that an error in one routine will affect the other.

A final way to firewall is to designate certain interfaces as boundaries to "safe" areas. Check data crossing the boundaries of a safe area for validity and respond sensibly if the data isn't valid. Another way of thinking of this approach is as an operating-room technique. Data is sterilized before it's allowed to enter the operating room. Anything that's in the operating room is assumed to be safe. The key design decision is deciding what to put in the operating room, what to keep out, and where to put the doors—which routines are considered to be inside the safety zone, which are outside, and which sanitize the data. The easiest way to do this is usually by sanitizing external data as it arrives, but data often needs to be sanitized at more than one level, so multiple levels of sterilization are sometimes required.



Routines That Assume Data Is Safe:	Routines That Check Their Data:
PrintBeginPage()	PointsToDeviceUnits()
PrintEndPage()	DeviceUnitsToPoints()
PrintBeginDoc()	SetFontSize()
PrintEndDoc()	SetUnderlinePosition()
SetFontName()	SetStrikeoutPosition()
SetCursorHeight()	SetItalicAngle()
SetCursorPosition()	
ShowCursor()	
TimestampOutput()	

Distinguish between routines that assume their data is safe and routines that check the data they receive.

Checking Function Return Values

If you call a function and have the option of ignoring a function's return value (for example, in C, where you don't even have to acknowledge that a function returns a value), don't! Test the value. If you don't expect it ever to produce an error, check it anyway. The whole point of defensive programming is guarding against unexpected errors.

This guideline holds true for system functions as well as your own. Unless you've set an architectural guideline of not checking system calls for errors, check for error codes after each call. If you detect an error, include the error number and the description of the error from *perror()* in C or the equivalent in other languages.

Determining How Much Defensive Programming to Leave in Production Code

One of the paradoxes of defensive programming is that during development, you'd like an error to be noticeable—you'd rather have it be obnoxious than risk overlooking it. But during production, you'd rather have the error be as unobtrusive as possible, to have the program recover or fail gracefully. Here are some guidelines for deciding which defensive programming tools to leave in your production code and which to leave out:

Leave in code that checks for important errors. Decide which areas of the program can afford to have undetected errors and which areas cannot. For example, if you were writing a spreadsheet program, you could afford to have undetected errors in the screen-update area of the program because the main penalty for an error is only a messy screen. You could not afford to have undetected errors in the calculation engine because the errors might result in subtly incorrect results in someone's spreadsheet. Most users would rather suffer a messy screen than incorrect tax calculations and an audit by the IRS.

Remove code that checks for trivial errors. If an error has truly trivial consequences, remove code that checks for it. In the previous example, you might remove the code that checks the spreadsheet screen update. "Remove" doesn't mean physically remove the code. It means use version control, precompiler switches, or some other technique to compile the program without that particular code. If space isn't a problem, you could leave in the error-checking code but have it log messages to an error-log file unobtrusively.

Remove code that results in hard crashes. During development, when your program detects an error, you'd like the error to be as noticeable as possible so that you can fix it. Often, the best way to accomplish such a goal is to have the program print a debugging message and crash when it detects an error. This is useful even for minor errors.

During production, your users need a chance to save their work before the program crashes and are probably willing to tolerate a few anomalies in exchange for keeping the program going long enough for them to do that. Users don't appreciate anything that results in the loss of their work, regardless of how much it helps debugging and ultimately improves the quality of the program. If your program contains debugging code that could cause a loss of data, take it out of the production version.

Leave in code that helps the program crash gracefully. The opposite is also true. If your program contains debugging code that detects potentially fatal errors, users appreciate having a chance to save their work before the error becomes terminal. The word processor I'm using has a "SAVE" light that comes on right before the program runs out of memory. I've learned not to keep working when the light comes on but to save immediately and exit. When I restart the program, everything is fine. Theoretically, the program shouldn't run out of memory, and it shouldn't have more memory available when I restart it with the same document and the same machine. The fact that it runs out of memory is a defect. But the programmers were thoughtful enough to leave the debugging aid in the program, and I appreciate having the warning rather than losing my work.

See that the messages you leave in are friendly. If you leave internal error messages in the program, verify that they're in language that's friendly to the user. In one of my early programs, I got a call from a user who reported that she'd gotten a message that read "You've got a bad pointer allocation, Dog Breath!" Fortunately for me, she had a sense of humor. A common and effective approach is to notify the user of an "internal error" and list a phone number the user can call to report it.

Being Defensive About Defensive Programming

Too much of anything is bad, but too much whiskey is just enough.
Mark Twain

Too much defensive programming creates problems of its own. If you check data passed as parameters in every conceivable way in every conceivable place, your program will be fat and slow. What's worse, the additional code needed for defensive programming adds complexity to the software. Code installed for defensive programming is not immune to defects, and you're just as likely to find a defect in defensive-programming code as in any other code—more likely, if you write the code casually. Think about where you need to be defensive, and set your defensive-programming priorities accordingly.

5.7 How to Use Routine Parameters

HARD DATA

Interfaces between routines are some of the most error-prone areas of a program. One study by Basili and Perricone (1984) found that 39 percent of all errors were internal interface errors—errors in communication between routines. Here are a few guidelines for minimizing interface problems:

Make sure actual parameters match formal parameters. Formal parameters, also known as dummy parameters, are the variables declared in a routine definition. Actual parameters are the variables or constants used in the actual routine calls.

A common mistake is to put the wrong type of variable in a routine call—for example, using an integer when a floating point is needed. (This is a problem

only in weakly typed languages like C when you're not using full compiler warnings and in assembler. Strongly typed languages such as Pascal don't have this problem.) When arguments are input only, this is seldom a problem; usually the compiler converts the actual type to the formal type before passing it to the routine. If it is a problem, usually your compiler gives you a warning. But in some cases, particularly when the argument is used for both input and output, you can get stung by passing the wrong type of argument.

Develop the habit of checking types of arguments in parameter lists and heeding compiler warnings about mismatched parameter types. In C, use ANSI function prototypes for all your functions so that the compiler will check all function arguments automatically and warn you of mismatches.

CROSS-REFERENCE For details on documenting routine parameters, see "Commenting Routines" in Section 19.5. For details on formatting parameters, see Section 18.7, "Laying Out Routines."

Put parameters in input-modify-output order. Instead of ordering parameters randomly or alphabetically, list the parameters that are input-only first, input-and-output second, and output-only third. This ordering implies the sequence of operations happening within the routine—inputting data, changing it, and sending back a result. Here are examples of parameter lists in Ada:

Ada Example of Parameters in Input-Modify-Output Order

```
procedure InvertMatrix
   (
   OriginalMatrix:  in MATRIX;
   ResultMatrix:    out MATRIX
   );
...

procedure ChangeStringCase
   (
   DesiredCase:     in STRING_CASE;
   MixedCaseString: in out USER_STRING
   );
...

procedure PrintPageNumber
   (
   PageNumber: in INTEGER;
   Status:     out STATUS_TYPE
   );
```

This ordering convention conflicts with the C-language convention of putting the modified parameter first. The input-modify-output convention makes more sense to me, but if you consistently order parameters in some way, you still do the readers of your code a service.

If several routines use similar parameters, put the similar parameters in a consistent order. The order of routine parameters can be a mnemonic, and inconsistent order can make parameters hard to remember. For example, in C, the *fprintf()* function is the same as the *printf()* function except that it adds a file as the first argument. A similar function, *fputs()*, is the same as *puts()* except that it adds a file as the last argument. This is an aggravating, pointless difference that makes the parameters of these functions harder to remember than they need to be.

On the other hand, the function *strncpy()* in C takes the arguments target string, source string, and maximum number of bytes, in that order, and the function *memcpy()* takes the same arguments in the same order. The similarity between the two functions helps in remembering the parameters in either function.

In Microsoft Windows programming, most of the Windows routines take a "handle" as their first parameter. The convention is easy to remember and makes each routine's argument list easier to remember.

HARD DATA

Use all the parameters. If you pass a parameter to a routine, use it. If you aren't using it, remove the parameter from the routine interface. Unused parameters are correlated with an increased error rate. In one study, 46 percent of routines with no unused variables had no errors. Only 17 to 29 percent of routines with more than one unreferenced variable had no errors (Card, Church, and Agresti 1986).

This rule to remove unused parameters has two exceptions. First, if you're using function pointers in C or procedure variables in Pascal, you'll have several routines with identical parameter lists. Some of the routines might not use all the parameters. That's OK. Second, if you're compiling part of your program conditionally, you might compile out parts of a routine that use a certain parameter. Be nervous about this practice, but if you're convinced it works, that's OK too. In general, if you have a good reason not to use a parameter, go ahead and leave it in place. If you don't have a good reason, make the effort to clean up the code.

Put *Status* or *Error* variables last. By convention, status variables and variables that indicate an error has occurred go last in the parameter list. They are incidental to the main purpose of the routine, and they are output-only parameters, so it's a sensible convention.

Don't use routine parameters as working variables. It's dangerous to use the parameters passed to a routine as working variables. Use local variables instead. For example, in the Pascal fragment below, the variable *InputVal* is improperly used to store intermediate results of a computation.

Pascal Example of Improper Use of Input Parameters

```
procedure Sample
   (
       InputVal:   Integer;
   VAR OutputVal:  Integer
   );
begin
   InputVal  := InputVal * CurrentMultiplier( InputVal );
   InputVal  := InputVal + CurrentAdder( InputVal );
   ...
   OutputVal := InputVal
end;
```

At this point, InputVal no longer contains the value that was input.

InputVal in this code fragment is misleading because by the time execution reaches the last line, *InputVal* no longer contains the input value; it contains a computed value based in part on the input value, and it is therefore misnamed. If you later need to modify the routine to use the original input value in some other place, you'll probably use *InputVal* and assume that it contains the original input value when it actually doesn't.

How do you solve the problem? Can you solve it by renaming *InputVal*? Probably not. You could name it something like *WorkingVal*, but that's an incomplete solution because the name fails to indicate that the variable's original value comes from outside the routine. You could name it something ridiculous like *InputValThatBecomesAWorkingVal*, or give up completely and name it *X* or *Val*, but all these approaches are weak.

A better approach is to avoid current and future problems by using working variables explicitly. The following code fragment demonstrates the technique:

Pascal Example of Good Use of Input Parameters

```
procedure Sample
   (
       InputVal:   Integer;
   VAR OutputVal:  Integer
   );
var
   WorkingVal: Integer;
begin
   WorkingVal  := InputVal;
   WorkingVal  := WorkingVal * CurrentMultiplier( WorkingVal );
   WorkingVal  := WorkingVal + CurrentAdder( WorkingVal );
   ...

   ...
   OutputVal := WorkingVal
end;
```

If you need to use the original value of Input-Val here or somewhere else, it's still available.

Introducing the new variable *WorkingVal* clarifies the role of *InputVal* and eliminates the chance of erroneously using *InputVal* at the wrong time. In Ada, this practice can be enforced by the compiler. If you designate a parameter as *in*, you're not allowed to modify its value within a function. Don't take this reasoning as a justification for literally naming a variable *WorkingVal*. In general, *WorkingVal* is a terrible name for a variable, and the name is used in this example only to make the variable's role clear.

In Fortran, using working variables is a particularly good practice. When a variable in the parameter list of the calling routine is modified in the called routine, it's also modified in the calling routine. In any language, assigning the input value to a working variable emphasizes where the value comes from. It eliminates the possibility that a variable from the parameter list will be modified accidentally.

CROSS-REFERENCE Fo details on considerations in using global iables, see Section 10.6, "Global Variables."

The same technique should be used to preserve the value of a global variable. If you need to compute a new value for a global variable, assign the global variable the final value at the end of the computation rather than using it to hold the result of intermediate calculations.

CROSS-REFERENCE or details on documentan, see Chapter 19, "Self-Documenting Code."

Document interface assumptions about parameters. If you assume the data being passed to your routine has certain characteristics, document the assumptions as you make them. It's not a waste of effort to document your assumptions both in the routine itself and in the place where the routine is called. Don't wait until you've written the routine to go back and write the comments—you won't remember all your assumptions. Even better than commenting your assumptions, use assertions to put them into code.

What kinds of interface assumptions about parameters should you document?

- Whether parameters are input-only, modified, or output-only
- Units of numeric parameters (inches, feet, meters, and so on)
- Meanings of status codes and error values if enumerated types aren't used
- Ranges of expected values
- Specific values that should never appear

HARD DATA

Limit the number of a routine's parameters to about seven. Seven is a magic number for people's comprehension. Psychological research has found that people generally cannot keep track of more than about seven chunks of information at once (Miller 1956). This discovery has been applied to an enormous number of disciplines, and it seems safe to conjecture that most people can't keep track of more than about seven routine parameters at once.

In practice, how much you can limit the number of parameters depends on how your language handles complex data structures. If you program in a modern language that supports structured data, you can pass a composite data structure containing 13 fields and think of it as one mental "chunk" of data. If you program in a more primitive language, you might need to pass all 13 fields individually.

CROSS-REFERENCE For details on module data, see "Module data mistaken for global data" in Section 6.2.

If you find yourself consistently passing more than a few arguments, the coupling among your routines is too tight. Design the routine or group of routines to reduce the coupling. If you are passing the same data to many different routines, group the routines into a module and treat the frequently used data as module data.

Consider an input, modify, and output naming convention for parameters. If you find that it's important to distinguish among input, modify, and output parameters, establish a naming convention that identifies them. You could prefix them with *i_*, *m_*, and *o_*. If you're feeling verbose, you could prefix them with *INPUT*, *MODIFY*, and *OUTPUT*.

Pass only the parts of structured variables that the routine needs. As discussed earlier, in Section 5.4 on coupling, it's better to pass only the specific fields of a structure that the called routine uses unless it uses almost all of them. The whole routine is easier to use somewhere else if you've specified a precise interface. A precise interface decreases inter-routine coupling and makes the routine more flexible.

CROSS-REFERENCE For details on abstract data types, see Section 12.3, "Abstract Data Types (ADTs)."

The precise-interface rule has an exception when you're working with abstract data types (ADTs). The data type might require that you keep track of a structured variable, but it's good practice with an abstract data type that you not look inside the structure. If that's the case, design the abstract-data-type routines so that they take the whole record as a parameter. This allows you to treat the record as an object outside the ADT routines and keeps the record at the same level of abstraction as the ADT routines. If you open the structure by working with individual fields, you lose the level of abstraction that the ADT provides.

Don't assume anything about the parameter-passing mechanism. Some hard-core nanosecond scrapers worry about the overhead associated with passing parameters and bypass the high-level language's parameter-passing mechanism. This is dangerous and makes code nonportable. Parameters are commonly passed on a system stack, but that's hardly the only parameter-passing mechanism that languages use. Even with stack-based mechanisms, the parameters themselves can be passed in different orders and each parameter's bytes can be ordered differently. If you fiddle with parameters directly, you virtually guarantee that your program won't run on a different machine.

5.8 Considerations in the Use of Functions

Modern languages such as C, Pascal, and Ada support both functions and procedures. A function is a routine that returns a value; a procedure is a routine that does not.

When to Use a Function and When to Use a Procedure

Purists argue that a function should return only one value, just as a mathematical function does. This means that a function would take only input parameters and return its only value through the function itself. The function would always be named for the value it returned, as *sin()*, *CustomerID()*, and *ScreenHeight()* are. A procedure, on the other hand, could take input, modify, and output parameters—as many of each as it wanted to.

A common programming practice is to have a function that operates as a procedure and returns a status value. Logically, it works as a procedure, but because it returns a value, it's officially a function. For example, you might have a procedure called *FormatOutput()* used in statements like this one:

```
if ( FormatOutput( Input, Formatting, Output ) = Success ) then ...
```

In this example, *FormatOutput()* operates as a procedure in that it has an output parameter, *Output*, but it is technically a function because the routine itself returns a value. Is this a valid way to use a function? In defense of this approach, you could maintain that the function return value has nothing to do with the main purpose of the routine, formatting output, or with the routine name, *FormatOutput()*; in that sense it operates more as a procedure does even if it is technically a function. The use of the return value to indicate the success or failure of the procedure is not confusing if the technique is used consistently.

The alternative is to create a procedure that has a status variable as an explicit parameter, which promotes code like this fragment:

```
FormatOutput( Input, Formatting, Output, Status )
if ( Status = Success ) then ...
```

I prefer the second style of coding, not because I'm hard-nosed about the difference between functions and procedures but because it makes a clear separation between the routine call and the test of the status value. To combine the call and the test into one line of code increases the density of the statement and correspondingly its complexity. The following use of a function is fine too:

```
Status = FormatOutput( Input, Formatting, Output )
if ( Status = Success ) then ...
```

A Unique Risk with Functions

Using a function creates the risk that the function will return its value improperly. This usually happens when the function has several possible paths and one of the paths avoids setting a return value. When creating a function, mentally execute each path to be sure that the function returns a value under all possible circumstances.

5.9 Macro Routines

CROSS-REFERENCE Even if your language doesn't have a macro preprocessor, you can build your own. For details, see Section 20.5, "Building Your Own Programming Tools."

Routines created with preprocessor macros call for a few unique considerations. The following rules and examples pertain to using the preprocessor in C. If you're using a different language or preprocessor, adapt the rules to your situation.

Enclose macro expressions in parentheses. Because macros and their arguments are expanded into code, be careful that they expand the way you want them to. One common problem lies in creating a macro like this one:

C Example of a Macro That Doesn't Expand Properly

```
#define product( a, b ) a*b
```

This macro has a problem. If you pass it nonatomic values for *a* or *b*, it won't do the multiplication properly. If you use the expression *product(x+1, y+2)*, it expands to *x+1 *y+2*, which, because of the precedence of the multiplication and addition operators, is not what you want. A better but still not perfect version of the macro looks like this:

C Example of a Macro That Still Doesn't Expand Properly

```
#define product( a, b ) (a)*(b)
```

This is close, but still no cigar. If you use *product()* in an expression that has operators with higher precedence than multiplication, the *(a)*(b)* will be torn apart. To prevent that, enclose the whole expression in parentheses:

C Example of a Macro That Works

```
#define product( a, b ) ((a)*(b))
```

Surround multiple-statement macros with curly braces. A macro can have multiple statements, which is a problem if you treat it as if it were a single statement. Here's an example of a macro that's headed for trouble:

C Example of a Macro with Multiple Statements That Doesn't Work

```
#define LookupEntry( Key, Index ) \
   Index = (Key - 10) / 5; \
   Index = min( Index, MAX_INDEX ); \
   Index = max( Index, MIN_INDEX );
...

for ( EntryCount = 0; EntryCount < NumEntries; EntryCount++ )
   LookupEntry( EntryCount, TableIndex[ EntryCount ] );
```

This macro is headed for trouble because it doesn't work as a regular function would. As it's shown, the only part of the macro that's executed in the *for* loop is the first line of the macro:

```
Index = (Key - 10) / 5;
```

To avoid this problem, surround the macro with curly braces, as shown here:

C Example of a Macro with Multiple Statements That Works

```
#define LookupEntry( Key, Index ) \
   { \
   Index = (Key - 10) / 5; \
   Index = min( Index, MAX_INDEX ); \
   Index = max( Index, MIN_INDEX ); \
   }
```

Name macros that expand to code like routines so that they can be replaced by routines if necessary. The C-language convention for naming macros is to use all capital letters. If the macro can be replaced by a routine, however, name it using the naming convention for routines instead. That way you can replace macros with routines and vice versa without changing anything but the routine involved.

Following this recommendation entails some risk. If you commonly use *++* and *--* as side effects (as part of other statements), you'll get burned when you use macros that you think are routines. That might cause you to avoid this recommendation. But considering the other problems with side effects, you'll do better to follow this recommendation and avoid side effects instead.

CROSS-REFERENCE This is a checklist of considerations about the quality of the routine. For a list of the steps used to build a routine, see the checklist "Constructing a Routine" in Chapter 4, page 70.

CHECKLIST

High-Quality Routines

BIG-PICTURE ISSUES

- ❑ Is the reason for creating the routine sufficient?
- ❑ Have all parts of the routine that would benefit from being put into routines of their own been put into routines of their own?
- ❑ Is the routine's name a strong, clear verb-plus-object name for a procedure or a description of the return value for a function?
- ❑ Does the routine's name describe everything the routine does?
- ❑ Does the routine have strong, functional cohesion—doing one and only one thing and doing it well?
- ❑ Do the routines have loose coupling—is one routine's connection to other routines small, intimate, visible, and flexible?
- ❑ Is the length of the routine determined naturally by its function and logic, rather than by an artificial coding standard?

DEFENSIVE-PROGRAMMING ISSUES

- ❑ Are assertions used to document assumptions?
- ❑ Does the routine protect itself from bad input data?
- ❑ Does the routine handle exceptions gracefully?
- ❑ Is the routine designed to handle changes gracefully?
- ❑ Have debugging aids been installed in such a way that they can be activated or deactivated without a great deal of fuss?
- ❑ Have information hiding, loose coupling, and data checks been used to firewall errors so that they won't affect code outside the routine?
- ❑ Does the routine check function return values?
- ❑ Is the defensive code that's left in the production code designed to help the user rather than the programmer?

PARAMETER-PASSING ISSUES

- ❑ Do the formal and actual parameters match?
- ❑ Are the routine's parameters in a sensible order, including matching the order of parameters in similar routines?
- ❑ Are interface assumptions documented?

(continued)

High-Quality Routines checklist, *continued*

- ❑ Does the routine have seven or fewer parameters?
- ❑ Are only the parts of a structured variable that are needed, rather than the whole variable, passed to the routine?
- ❑ Is each input parameter used?
- ❑ Is each output parameter used?
- ❑ If the routine is a function, does it return a value under all possible circumstances?

Key Points

- The most important reason to create a routine is to improve the intellectual manageability of a program, and you can create a routine for many other good reasons. Saving space is a minor reason; improved correctness, reliability, and modifiability are better reasons.
- The point of strong cohesion and loose coupling is that they provide for higher levels of abstraction—you can take the operation of a cohesive, loosely coupled routine for granted, which allows you to focus completely on other tasks.
- Sometimes the operation that most benefits from being put into a routine of its own is a simple one.
- The name of a routine is an indication of its quality. If the name is bad and it's accurate, the routine might be poorly designed. If the name is bad and it's inaccurate, it's not telling you what the program does. Either way, a bad name means that the program needs to be changed.
- Defensive-programming techniques make errors easier to find, easier to fix, and less damaging to production code.