Microsoft

CODE COMPLETE

2

Second Edition

A practical handbook of software construction

Steve McConnell

Two-time winner of the *Software Development* Magazine Jolt Award

Microsoft

Code Complete, Second Edition

Steve McConnell

Contents at a Glance

Chapter 6

Working Classes

cc2e.com/0665

Contents

Related Topics

- Design in construction: Chapter 5
- Software architecture: Section 3.5
- High-quality routines: Chapter 7
- The Pseudocode Programming Process: Chapter 9
- Refactoring: Chapter 24

In the dawn of computing, programmers thought about programming in terms of statements. Throughout the 1970s and 1980s, programmers began thinking about programs in terms of routines. In the twenty-first century, programmers think about programming in terms of classes.

KEY POINT

A class is a collection of data and routines that share a cohesive, well-defined responsibility. A class might also be a collection of routines that provides a cohesive set of services even if no common data is involved. A key to being an effective programmer is maximizing the portion of a program that you can safely ignore while working on any one section of code. Classes are the primary tool for accomplishing that objective.

This chapter contains a distillation of advice in creating high-quality classes. If you're still warming up to object-oriented concepts, this chapter might be too advanced. Make sure you've read Chapter 5, "Design in Construction." Then start with Section 6.1, "Class Foundations: Abstract Data Types (ADTs)," and ease your way into the remaining sections. If you're already familiar with class basics, you might skim Section 6.1 and then dive into the discussion of class interfaces in Section 6.2. The "Additional Resources" section at the end of this chapter contains pointers to introductory reading, advanced reading, and programming-language-specific resources.

6.1 Class Foundations: Abstract Data Types (ADTs)

An abstract data type is a collection of data and operations that work on that data. The operations both describe the data to the rest of the program and allow the rest of the program to change the data. The word "data" in "abstract data type" is used loosely. An ADT might be a graphics window with all the operations that affect it, a file and file operations, an insurance-rates table and the operations on it, or something else.

Cross-Reference Thinking about ADTs first and classes second is an example of programming *into* a language vs. programming in one. See Section 4.3, "Your Location on the Technology Wave," and Section 34.4, "Program into Your Language, Not in It."

Understanding ADTs is essential to understanding object-oriented programming. Without understanding ADTs, programmers create classes that are "classes" in name only—in reality, they are little more than convenient carrying cases for loosely related collections of data and routines. With an understanding of ADTs, programmers can create classes that are easier to implement initially and easier to modify over time.

Traditionally, programming books wax mathematical when they arrive at the topic of abstract data types. They tend to make statements like "One can think of an abstract data type as a mathematical model with a collection of operations defined on it." Such books make it seem as if you'd never actually use an abstract data type except as a sleep aid.

Such dry explanations of abstract data types completely miss the point. Abstract data types are exciting because you can use them to manipulate real-world entities rather than low-level, implementation entities. Instead of inserting a node into a linked list, you can add a cell to a spreadsheet, a new type of window to a list of window types, or another passenger car to a train simulation. Tap into the power of being able to work in the problem domain rather than at the low-level implementation domain!

Example of the Need for an ADT

To get things started, here's an example of a case in which an ADT would be useful. We'll get to the details after we have an example to talk about.

Suppose you're writing a program to control text output to the screen using a variety of typefaces, point sizes, and font attributes (such as bold and italic). Part of the program manipulates the text's fonts. If you use an ADT, you'll have a group of font routines bundled with the data—the typeface names, point sizes, and font attributes—they operate on. The collection of font routines and data is an ADT.

If you're not using ADTs, you'll take an ad hoc approach to manipulating fonts. For example, if you need to change to a 12-point font size, which happens to be 16 pixels high, you'll have code like this:

```
currentFont.size = 16
```

If you've built up a collection of library routines, the code might be slightly more readable:

```
currentFont.size = PointsToPixels( 12 )
```

Or you could provide a more specific name for the attribute, something like

```
currentFont.sizeInPixels = PointsToPixels( 12 )
```

But what you can't do is have both *currentFont.sizeInPixels* and *currentFont.sizeInPoints*, because, if both the data members are in play, *currentFont* won't have any way to know which of the two it should use. And if you change sizes in several places in the program, you'll have similar lines spread throughout your program.

If you need to set a font to bold, you might have code like this that uses a logical *or* and a hexidecimal constant *0x02*:

```
currentFont.attribute = currentFont.attribute or 0x02
```

If you're lucky, you'll have something cleaner than that, but the best you'll get with an ad hoc approach is something like this:

```
currentFont.attribute = currentFont.attribute or BOLD
```

Or maybe something like this:

```
currentFont.bold = True
```

As with the font size, the limitation is that the client code is required to control the data members directly, which limits how *currentFont* can be used.

If you program this way, you're likely to have similar lines in many places in your program.

Benefits of Using ADTs

The problem isn't that the ad hoc approach is bad programming practice. It's that you can replace the approach with a better programming practice that produces these benefits:

You can hide implementation details Hiding information about the font data type means that if the data type changes, you can change it in one place without affecting the whole program. For example, unless you hid the implementation details in an ADT, changing the data type from the first representation of bold to the second would entail changing your program in every place in which bold was set rather than in just one place. Hiding the information also protects the rest of the program if you decide to store data in external storage rather than in memory or to rewrite all the font-manipulation routines in another language.

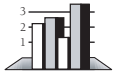
Changes don't affect the whole program If fonts need to become richer and support more operations (such as switching to small caps, superscripts, strikethrough, and so on), you can change the program in one place. The change won't affect the rest of the program.

You can make the interface more informative Code like *currentFont.size = 16* is ambiguous because *16* could be a size in either pixels or points. The context doesn't tell you which is which. Collecting all similar operations into an ADT allows you to define the entire interface in terms of points, or in terms of pixels, or to clearly differentiate between the two, which helps avoid confusing them.

It's easier to improve performance If you need to improve font performance, you can recode a few well-defined routines rather than wading through an entire program.

The program is more obviously correct You can replace the more tedious task of verifying that statements like *currentFont.attribute = currentFont.attribute or 0x02* are correct with the easier task of verifying that calls to *currentFont.SetBoldOn()* are correct. With the first statement, you can have the wrong structure name, the wrong field name, the wrong operation (*and* instead of *or*), or the wrong value for the attribute (*0x20* instead of *0x02*). In the second case, the only thing that could possibly be wrong with the call to *currentFont.SetBoldOn()* is that it's a call to the wrong routine name, so it's easier to see whether it's correct.

The program becomes more self-documenting You can improve statements like *currentFont.attribute or 0x02* by replacing *0x02* with *BOLD* or whatever *0x02* represents, but that doesn't compare to the readability of a routine call such as *currentFont.SetBoldOn()*.



HARD DATA

Woodfield, Dunsmore, and Shen conducted a study in which graduate and senior undergraduate computer-science students answered questions about two programs: one that was divided into eight routines along functional lines, and one that was divided into eight abstract-data-type routines (1981). Students using the abstract-data-type program scored over 30 percent higher than students using the functional version.

You don't have to pass data all over your program In the examples just presented, you have to change *currentFont* directly or pass it to every routine that works with fonts. If you use an abstract data type, you don't have to pass *currentFont* all over the program and you don't have to turn it into global data either. The ADT has a structure that contains *currentFont*'s data. The data is directly accessed only by routines that are part of the ADT. Routines that aren't part of the ADT don't have to worry about the data.

You're able to work with real-world entities rather than with low-level implementation structures You can define operations dealing with fonts so that most of the program operates solely in terms of fonts rather than in terms of array accesses, structure definitions, and *True* and *False*.

In this case, to define an abstract data type, you'd define a few routines to control fonts—perhaps like this:

```
currentFont.SetSizeInPoints( sizeInPoints )
currentFont.SetSizeInPixels( sizeInPixels )
currentFont.SetBoldOn()
currentFont.SetBoldOff()
currentFont.SetItalicOn()
currentFont.SetItalicOff()
currentFont.SetTypeFace( faceName )
```

The code inside these routines would probably be short—it would probably be similar to the code you saw in the ad hoc approach to the font problem earlier. The difference is that you've isolated font operations in a set of routines. That provides a better level of abstraction for the rest of your program to work with fonts, and it gives you a layer of protection against changes in font operations.

More Examples of ADTs

Suppose you're writing software that controls the cooling system for a nuclear reactor. You can treat the cooling system as an abstract data type by defining the following operations for it:

```
coolingSystem.GetTemperature()
coolingSystem.SetCirculationRate( rate )
coolingSystem.OpenValve( valveNumber )
coolingSystem.CloseValve( valveNumber )
```

The specific environment would determine the code written to implement each of these operations. The rest of the program could deal with the cooling system through these functions and wouldn't have to worry about internal details of data-structure implementations, data-structure limitations, changes, and so on.

Here are more examples of abstract data types and likely operations on them:

Cruise Control	**Blender**	**Fuel Tank**
Set speed	Turn on	Fill tank
Get current settings	Turn off	Drain tank
Resume former speed	Set speed	Get tank capacity
Deactivate	Start "Insta-Pulverize"	Get tank status
	Stop "Insta-Pulverize"	
List		**Stack**
Initialize list	**Light**	Initialize stack
Insert item in list	Turn on	Push item onto stack
Remove item from list	Turn off	Pop item from stack
Read next item from list		Read top of stack

Set of Help Screens	Menu	File
Add help topic	Start new menu	Open file
Remove help topic	Delete menu	Read file
Set current help topic	Add menu item	Write file
Display help screen	Remove menu item	Set current file location
Remove help display	Activate menu item	Close file
Display help index	Deactivate menu item	
Back up to previous screen	Display menu	**Elevator**
	Hide menu	Move up one floor
Pointer	Get menu choice	Move down one floor
Get pointer to new memory		Move to specific floor
Dispose of memory from existing pointer		Report current floor Return to home floor
Change amount of memory allocated		

Yon can derive several guidelines from a study of these examples; those guidelines are described in the following subsections:

Build or use typical low-level data types as ADTs, not as low-level data types Most discussions of ADTs focus on representing typical low-level data types as ADTs. As you can see from the examples, you can represent a stack, a list, and a queue, as well as virtually any other typical data type, as an ADT.

The question you need to ask is, "What does this stack, list, or queue represent?" If a stack represents a set of employees, treat the ADT as employees rather than as a stack. If a list represents a set of billing records, treat it as billing records rather than a list. If a queue represents cells in a spreadsheet, treat it as a collection of cells rather than a generic item in a queue. Treat yourself to the highest possible level of abstraction.

Treat common objects such as files as ADTs Most languages include a few abstract data types that you're probably familiar with but might not think of as ADTs. File operations are a good example. While writing to disk, the operating system spares you the grief of positioning the read/write head at a specific physical address, allocating a new disk sector when you exhaust an old one, and interpreting cryptic error codes. The operating system provides a first level of abstraction and the ADTs for that level. High-level languages provide a second level of abstraction and ADTs for that higher level. A high-level language protects you from the messy details of generating operating-system calls and manipulating data buffers. It allows you to treat a chunk of disk space as a "file."

You can layer ADTs similarly. If you want to use an ADT at one level that offers data-structure level operations (like pushing and popping a stack), that's fine. You can create another level on top of that one that works at the level of the real-world problem.

Treat even simple items as ADTs You don't have to have a formidable data type to justify using an abstract data type. One of the ADTs in the example list is a light that supports only two operations—turning it on and turning it off. You might think that it would be a waste to isolate simple "on" and "off" operations in routines of their own, but even simple operations can benefit from the use of ADTs. Putting the light and its operations into an ADT makes the code more self-documenting and easier to change, confines the potential consequences of changes to the *TurnLightOn()* and *TurnLightOff()* routines, and reduces the number of data items you have to pass around.

Refer to an ADT independently of the medium it's stored on Suppose you have an insurance-rates table that's so big that it's always stored on disk. You might be tempted to refer to it as a "rate *file*" and create access routines such as *RateFile.Read()*. When you refer to it as a file, however, you're exposing more information about the data than you need to. If you ever change the program so that the table is in memory instead of on disk, the code that refers to it as a file will be incorrect, misleading, and confusing. Try to make the names of classes and access routines independent of how the data is stored, and refer to the abstract data type, like the insurance-rates table, instead. That would give your class and access routine names like *rateTable.Read()* or simply *rates.Read()*.

Handling Multiple Instances of Data with ADTs in Non-Object-Oriented Environments

Object-oriented languages provide automatic support for handling multiple instances of an ADT. If you've worked exclusively in object-oriented environments and you've never had to handle the implementation details of multiple instances yourself, count your blessings! (You can also move on to the next section, "ADTs and Classes.")

If you're working in a non-object-oriented environment such as C, you will have to build support for multiple instances manually. In general, that means including services for the ADT to create and delete instances and designing the ADT's other services so that they can work with multiple instances.

The font ADT originally offered these services:

```
currentFont.SetSize( sizeInPoints )
currentFont.SetBoldOn()
currentFont.SetBoldOff()
currentFont.SetItalicOn()
currentFont.SetItalicOff()
currentFont.SetTypeFace( faceName )
```

In a non-object-oriented environment, these functions would not be attached to a class and would look more like this:

```
SetCurrentFontSize( sizeInPoints )
SetCurrentFontBoldOn()
SetCurrentFontBoldOff()
SetCurrentFontItalicOn()
SetCurrentFontItalicOff()
SetCurrentFontTypeFace( faceName )
```

If you want to work with more than one font at a time, you'll need to add services to create and delete font instances—maybe these:

```
CreateFont( fontId )
DeleteFont( fontId )
SetCurrentFont( fontId )
```

The notion of a *fontId* has been added as a way to keep track of multiple fonts as they're created and used. For other operations, you can choose from among three ways to handle the ADT interface:

- Option 1: Explicitly identify instances each time you use ADT services. In this case, you don't have the notion of a "current font." You pass *fontId* to each routine that manipulates fonts. The *Font* functions keep track of any underlying data, and the client code needs to keep track only of the *fontId*. This requires adding *fontId* as a parameter to each font routine.
- Option 2: Explicitly provide the data used by the ADT services. In this approach, you declare the data that the ADT uses within each routine that uses an ADT service. In other words, you create a *Font* data type that you pass to each of the ADT service routines. You must design the ADT service routines so that they use the *Font* data that's passed to them each time they're called. The client code doesn't need a font ID if you use this approach because it keeps track of the font data itself. (Even though the data is available directly from the *Font* data type, you should access it only with the ADT service routines. This is called keeping the structure "closed.")

  The advantage of this approach is that the ADT service routines don't have to look up font information based on a font ID. The disadvantage is that it exposes font data to the rest of the program, which increases the likelihood that client code will make use of the ADT's implementation details that should have remained hidden within the ADT.
- Option 3: Use implicit instances (with great care). Design a new service to call to make a specific font instance the current one—something like *SetCurrentFont (fontId)*. Setting the current font makes all other services use the current font when they're called. If you use this approach, you don't need *fontId* as a parameter to the other services. For simple applications, this can streamline use of

multiple instances. For complex applications, this systemwide dependence on state means that you must keep track of the current font instance throughout code that uses the *Font* functions. Complexity tends to proliferate, and for applications of any size, better alternatives exist.

Inside the abstract data type, you'll have a wealth of options for handling multiple instances, but outside, this sums up the choices if you're working in a non-object-oriented language.

ADTs and Classes

Abstract data types form the foundation for the concept of classes. In languages that support classes, you can implement each abstract data type as its own class. Classes usually involve the additional concepts of inheritance and polymorphism. One way of thinking of a class is as an abstract data type plus inheritance and polymorphism.

6.2 Good Class Interfaces

The first and probably most important step in creating a high-quality class is creating a good interface. This consists of creating a good abstraction for the interface to represent and ensuring that the details remain hidden behind the abstraction.

Good Abstraction

As "Form Consistent Abstractions" in Section 5.3 described, abstraction is the ability to view a complex operation in a simplified form. A class interface provides an abstraction of the implementation that's hidden behind the interface. The class's interface should offer a group of routines that clearly belong together.

You might have a class that implements an employee. It would contain data describing the employee's name, address, phone number, and so on. It would offer services to initialize and use an employee. Here's how that might look.

C++ Example of a Class Interface That Presents a Good Abstraction

```
class Employee {
public:
   // public constructors and destructors
   Employee();
   Employee(
      FullName name,
      String address,
      String workPhone,
      String homePhone,
      TaxId taxIdNumber,
      JobClassification jobClass
   );
   virtual ~Employee();
```

Cross-Reference Code samples in this book are formatted using a coding convention that emphasizes similarity of styles across multiple languages. For details on the convention (and discussions about multiple coding styles), see "Mixed-Language Programming Considerations" in Section 11.4.

```
   // public routines
   FullName GetName() const;
   String GetAddress() const;
   String GetWorkPhone() const;
   String GetHomePhone() const;
   TaxId GetTaxIdNumber() const;
   JobClassification GetJobClassification() const;
   ...
private:
   ...
};
```

Internally, this class might have additional routines and data to support these services, but users of the class don't need to know anything about them. The class interface abstraction is great because every routine in the interface is working toward a consistent end.

A class that presents a poor abstraction would be one that contained a collection of miscellaneous functions. Here's an example:

C++ Example of a Class Interface That Presents a Poor Abstraction

```
class Program {
public:
   ...
   // public routines
   void InitializeCommandStack();
   void PushCommand( Command command );
   Command PopCommand();
   void ShutdownCommandStack();
   void InitializeReportFormatting();
   void FormatReport( Report report );
   void PrintReport( Report report );
   void InitializeGlobalData();
   void ShutdownGlobalData();
   ...
private:
   ...
};
```

Suppose that a class contains routines to work with a command stack, to format reports, to print reports, and to initialize global data. It's hard to see any connection among the command stack and report routines or the global data. The class interface doesn't present a consistent abstraction, so the class has poor cohesion. The routines should be reorganized into more-focused classes, each of which provides a better abstraction in its interface.

If these routines were part of a *Program* class, they could be revised to present a consistent abstraction, like so:

C++ Example of a Class Interface That Presents a Better Abstraction

```
class Program {
public:
   ...
   // public routines
   void InitializeUserInterface();
   void ShutDownUserInterface();
   void InitializeReports();
   void ShutDownReports();
   ...
private:
   ...
};
```

The cleanup of this interface assumes that some of the original routines were moved to other, more appropriate classes and some were converted to private routines used by *InitializeUserInterface()* and the other routines.

This evaluation of class abstraction is based on the class's collection of public routines—that is, on the class's interface. The routines inside the class don't necessarily present good individual abstractions just because the overall class does, but they need to be designed to present good abstractions too. For guidelines on that, see Section 7.2, "Design at the Routine Level."

The pursuit of good, abstract interfaces gives rise to several guidelines for creating class interfaces.

Present a consistent level of abstraction in the class interface A good way to think about a class is as the mechanism for implementing the abstract data types described in Section 6.1. Each class should implement one and only one ADT. If you find a class implementing more than one ADT, or if you can't determine what ADT the class implements, it's time to reorganize the class into one or more well-defined ADTs.

Here's an example of a class that presents an interface that's inconsistent because its level of abstraction is not uniform:

C++ Example of a Class Interface with Mixed Levels of Abstraction

```
class EmployeeCensus: public ListContainer {
public:
   ...
   // public routines
   void AddEmployee( Employee employee );
   void RemoveEmployee( Employee employee );

   Employee NextItemInList();
   Employee FirstItem();
   Employee LastItem();
   ...
private:
   ...
};
```

The abstraction of these routines is at the "employee" level.

The abstraction of these routines is at the "list" level.

This class is presenting two ADTs: an *Employee* and a *ListContainer*. This sort of mixed abstraction commonly arises when a programmer uses a container class or other library classes for implementation and doesn't hide the fact that a library class is used. Ask yourself whether the fact that a container class is used should be part of the abstraction. Usually that's an implementation detail that should be hidden from the rest of the program, like this:

C++ Example of a Class Interface with Consistent Levels of Abstraction

```
class EmployeeCensus {
public:
   ...
   // public routines
   void AddEmployee( Employee employee );
   void RemoveEmployee( Employee employee );
   Employee NextEmployee();
   Employee FirstEmployee();
   Employee LastEmployee();
   ...
private:
   ListContainer m_EmployeeList;
   ...
};
```

The abstraction of all these routines is now at the "employee" level.

That the class uses the *ListContainer* library is now hidden.

Programmers might argue that inheriting from *ListContainer* is convenient because it supports polymorphism, allowing an external search or sort function that takes a *ListContainer* object. That argument fails the main test for inheritance, which is, "Is inheritance used only for "is a" relationships?" To inherit from *ListContainer* would mean that *EmployeeCensus* "is a" *ListContainer*, which obviously isn't true. If the abstraction of the *EmployeeCensus* object is that it can be searched or sorted, that should be incorporated as an explicit, consistent part of the class interface.

If you think of the class's public routines as an air lock that keeps water from getting into a submarine, inconsistent public routines are leaky panels in the class. The leaky panels might not let water in as quickly as an open air lock, but if you give them enough time, they'll still sink the boat. In practice, this is what happens when you mix levels of abstraction. As the program is modified, the mixed levels of abstraction make the program harder and harder to understand, and it gradually degrades until it becomes unmaintainable.

KEY POINT

Be sure you understand what abstraction the class is implementing Some classes are similar enough that you must be careful to understand which abstraction the class interface should capture. I once worked on a program that needed to allow information to be edited in a table format. We wanted to use a simple grid control, but the grid controls that were available didn't allow us to color the data-entry cells, so we decided to use a spreadsheet control that did provide that capability.

The spreadsheet control was far more complicated than the grid control, providing about 150 routines to the grid control's 15. Since our goal was to use a grid control, not a spreadsheet control, we assigned a programmer to write a wrapper class to hide the fact that we were using a spreadsheet control as a grid control. The programmer grumbled quite a bit about unnecessary overhead and bureaucracy, went away, and came back a couple days later with a wrapper class that faithfully exposed all 150 routines of the spreadsheet control.

This was not what was needed. We wanted a grid-control interface that encapsulated the fact that, behind the scenes, we were using a much more complicated spreadsheet control. The programmer should have exposed just the 15 grid-control routines plus a 16th routine that supported cell coloring. By exposing all 150 routines, the programmer created the possibility that, if we ever wanted to change the underlying implementation, we could find ourselves supporting 150 public routines. The programmer failed to achieve the encapsulation we were looking for, as well as creating a lot more work for himself than necessary.

Depending on specific circumstances, the right abstraction might be either a spreadsheet control or a grid control. When you have to choose between two similar abstractions, make sure you choose the right one.

Provide services in pairs with their opposites Most operations have corresponding, equal, and opposite operations. If you have an operation that turns a light on, you'll probably need one to turn it off. If you have an operation to add an item to a list, you'll probably need one to delete an item from the list. If you have an operation to activate a menu item, you'll probably need one to deactivate an item. When you design a class, check each public routine to determine whether you need its complement. Don't create an opposite gratuitously, but do check to see whether you need one.

Move unrelated information to another class In some cases, you'll find that half a class's routines work with half the class's data and half the routines work with the other half of the data. In such a case, you really have two classes masquerading as one. Break them up!

Make interfaces programmatic rather than semantic when possible Each interface consists of a programmatic part and a semantic part. The programmatic part consists of the data types and other attributes of the interface that can be enforced by the compiler. The semantic part of the interface consists of the assumptions about how the interface will be used, which cannot be enforced by the compiler. The semantic interface includes considerations such as "*RoutineA* must be called before *RoutineB*" or "*RoutineA* will crash if *dataMember1* isn't initialized before it's passed to *RoutineA*." The semantic interface should be documented in comments, but try to keep interfaces minimally dependent on documentation. Any aspect of an interface that can't be enforced by the compiler is an aspect that's likely to be misused. Look for ways to convert semantic interface elements to programmatic interface elements by using *Asserts* or other techniques.

Cross-Reference For more suggestions about how to preserve code quality as code is modified, see Chapter 24, "Refactoring."

Beware of erosion of the interface's abstraction under modification As a class is modified and extended, you often discover additional functionality that's needed, that doesn't quite fit with the original class interface, but that seems too hard to implement any other way. For example, in the *Employee* class, you might find that the class evolves to look like this:

C++ Example of a Class Interface That's Eroding Under Maintenance

```
class Employee {
public:
   ...
   // public routines
   FullName GetName() const;
   Address GetAddress() const;
   PhoneNumber GetWorkPhone() const;
   ...
   bool IsJobClassificationValid( JobClassification jobClass );
   bool IsZipCodeValid( Address address );
   bool IsPhoneNumberValid( PhoneNumber phoneNumber );

   SqlQuery GetQueryToCreateNewEmployee() const;
   SqlQuery GetQueryToModifyEmployee() const;
   SqlQuery GetQueryToRetrieveEmployee() const;
   ...
private:
   ...
};
```

What started out as a clean abstraction in an earlier code sample has evolved into a hodgepodge of functions that are only loosely related. There's no logical connection between employees and routines that check ZIP Codes, phone numbers, or job classifications. The routines that expose SQL query details are at a much lower level of abstraction than the *Employee* class, and they break the *Employee* abstraction.

Don't add public members that are inconsistent with the interface abstraction Each time you add a routine to a class interface, ask "Is this routine consistent with the abstraction provided by the existing interface?" If not, find a different way to make the modification and preserve the integrity of the abstraction.

Consider abstraction and cohesion together The ideas of abstraction and cohesion are closely related—a class interface that presents a good abstraction usually has strong cohesion. Classes with strong cohesion tend to present good abstractions, although that relationship is not as strong.

I have found that focusing on the abstraction presented by the class interface tends to provide more insight into class design than focusing on class cohesion. If you see that a class has weak cohesion and aren't sure how to correct it, ask yourself whether the class presents a consistent abstraction instead.

Good Encapsulation

Cross-Reference For more on encapsulation, see "Encapsulate Implementation Details" in Section 5.3.

As Section 5.3 discussed, encapsulation is a stronger concept than abstraction. Abstraction helps to manage complexity by providing models that allow you to ignore implementation details. Encapsulation is the enforcer that prevents you from looking at the details even if you want to.

The two concepts are related because, without encapsulation, abstraction tends to break down. In my experience, either you have both abstraction and encapsulation or you have neither. There is no middle ground.

The single most important factor that distinguishes a well-designed module from a poorly designed one is the degree to which the module hides its internal data and other implementation details from other modules.
—*Joshua Bloch*

Minimize accessibility of classes and members Minimizing accessibility is one of several rules that are designed to encourage encapsulation. If you're wondering whether a specific routine should be public, private, or protected, one school of thought is that you should favor the strictest level of privacy that's workable (Meyers 1998, Bloch 2001). I think that's a fine guideline, but I think the more important guideline is, "What best preserves the integrity of the interface abstraction?" If exposing the routine is consistent with the abstraction, it's probably fine to expose it. If you're not sure, hiding more is generally better than hiding less.

Don't expose member data in public Exposing member data is a violation of encapsulation and limits your control over the abstraction. As Arthur Riel points out, a *Point* class that exposes

```
float x;
float y;
float z;
```

is violating encapsulation because client code is free to monkey around with *Point*'s data and *Point* won't necessarily even know when its values have been changed (Riel 1996). However, a *Point* class that exposes

```
float GetX();
float GetY();
float GetZ();
void SetX( float x );
void SetY( float y );
void SetZ( float z );
```

is maintaining perfect encapsulation. You have no idea whether the underlying implementation is in terms of *float*s *x*, *y*, and *z*, whether *Point* is storing those items as *doubl*es and converting them to *float*s, or whether *Point* is storing them on the moon and retrieving them from a satellite in outer space.

Avoid putting private implementation details into a class's interface With true encapsulation, programmers would not be able to see implementation details at all. They would be hidden both figuratively and literally. In popular languages, including

C++, however, the structure of the language requires programmers to disclose implementation details in the class interface. Here's an example:

C++ Example of Exposing a Class's Implementation Details

```
class Employee {
public:
   ...
   Employee(
      FullName name,
      String address,
      String workPhone,
      String homePhone,
      TaxId taxIdNumber,
      JobClassification jobClass
   );
   ...
   FullName GetName() const;
   String GetAddress() const;
   ...
private:
   String m_Name;
   String m_Address;
   int m_jobClass;
   ...
};
```

Here are the exposed implementation details.

Including *private* declarations in the class header file might seem like a small transgression, but it encourages other programmers to examine the implementation details. In this case, the client code is intended to use the *Address* type for addresses but the header file exposes the implementation detail that addresses are stored as *Strings*.

Scott Meyers describes a common way to address this issue in Item 34 of *Effective C++*, 2d ed. (Meyers 1998). You separate the class interface from the class implementation. Within the class declaration, include a pointer to the class's implementation but don't include any other implementation details.

C++ Example of Hiding a Class's Implementation Details

```
class Employee {
public:
   ...
   Employee( ... );
   ...
   FullName GetName() const;
   String GetAddress() const;
   ...
private:
   EmployeeImplementation *m_implementation;
};
```

Here the implementation details are hidden behind the pointer.

Now you can put implementation details inside the *EmployeeImplementation* class, which should be visible only to the *Employee* class and not to the code that uses the *Employee* class.

If you've already written lots of code that doesn't use this approach for your project, you might decide it isn't worth the effort to convert a mountain of existing code to use this approach. But when you *read* code that exposes its implementation details, you can resist the urge to comb through the *private* section of the class interface looking for implementation clues.

Don't make assumptions about the class's users A class should be designed and implemented to adhere to the contract implied by the class interface. It shouldn't make any assumptions about how that interface will or won't be used, other than what's documented in the interface. Comments like the following one are an indication that a class is more aware of its users than it should be:

```
-- initialize x, y, and z to 1.0 because DerivedClass blows
-- up if they're initialized to 0.0
```

Avoid friend classes In a few circumstances such as the State pattern, friend classes can be used in a disciplined way that contributes to managing complexity (Gamma et al. 1995). But, in general, friend classes violate encapsulation. They expand the amount of code you have to think about at any one time, thereby increasing complexity.

Don't put a routine into the public interface just because it uses only public routines The fact that a routine uses only public routines is not a significant consideration. Instead, ask whether exposing the routine would be consistent with the abstraction presented by the interface.

Favor read-time convenience to write-time convenience Code is read far more times than it's written, even during initial development. Favoring a technique that speeds write-time convenience at the expense of read-time convenience is a false economy. This is especially applicable to creation of class interfaces. Even if a routine doesn't quite fit the interface's abstraction, sometimes it's tempting to add a routine to an interface that would be convenient for the particular client of a class that you're working on at the time. But adding that routine is the first step down a slippery slope, and it's better not to take even the first step.

It ain't abstract if you have to look at the underlying implementation to understand what's going on.
—P. J. Plauger

Be very, very wary of semantic violations of encapsulation At one time I thought that when I learned how to avoid syntax errors I would be home free. I soon discovered that learning how to avoid syntax errors had merely bought me a ticket to a whole new theater of coding errors, most of which were more difficult to diagnose and correct than the syntax errors.

The difficulty of semantic encapsulation compared to syntactic encapsulation is similar. Syntactically, it's relatively easy to avoid poking your nose into the internal workings of another class just by declaring the class's internal routines and data *private*. Achieving

semantic encapsulation is another matter entirely. Here are some examples of the ways that a user of a class can break encapsulation semantically:

- Not calling Class A's *InitializeOperations()* routine because you know that Class A's *PerformFirstOperation()* routine calls it automatically.
- Not calling the *database.Connect()* routine before you call *employee.Retrieve(database)* because you know that the *employee.Retrieve()* function will connect to the database if there isn't already a connection.
- Not calling Class A's *Terminate()* routine because you know that Class A's *PerformFinalOperation()* routine has already called it.
- Using a pointer or reference to *ObjectB* created by *ObjectA* even after *ObjectA* has gone out of scope, because you know that *ObjectA* keeps *ObjectB* in *static* storage and *ObjectB* will still be valid.
- Using Class B's *MAXIMUM_ELEMENTS* constant instead of using *ClassA.MAXIMUM_ELEMENTS*, because you know that they're both equal to the same value.


KEY POINT

The problem with each of these examples is that they make the client code dependent not on the class's public interface, but on its private implementation. Anytime you find yourself looking at a class's implementation to figure out how to use the class, you're not programming to the interface; you're programming *through* the interface *to* the implementation. If you're programming through the interface, encapsulation is broken, and once encapsulation starts to break down, abstraction won't be far behind.

If you can't figure out how to use a class based solely on its interface documentation, the right response is *not* to pull up the source code and look at the implementation. That's good initiative but bad judgment. The right response is to contact the author of the class and say "I can't figure out how to use this class." The right response on the class-author's part is *not* to answer your question face to face. The right response for the class author is to check out the class-interface file, modify the class-interface documentation, check the file back in, and then say "See if you can understand how it works now." You want this dialog to occur in the interface code itself so that it will be preserved for future programmers. You don't want the dialog to occur solely in your own mind, which will bake subtle semantic dependencies into the client code that uses the class. And you don't want the dialog to occur interpersonally so that it benefits only your code but no one else's.

Watch for coupling that's too tight "Coupling" refers to how tight the connection is between two classes. In general, the looser the connection, the better. Several general guidelines flow from this concept:

- Minimize accessibility of classes and members.
- Avoid *friend* classes, because they're tightly coupled.

- Make data *private* rather than *protected* in a base class to make derived classes less tightly coupled to the base class.
- Avoid exposing member data in a class's public interface.
- Be wary of semantic violations of encapsulation.
- Observe the "Law of Demeter" (discussed in Section 6.3 of this chapter).

Coupling goes hand in glove with abstraction and encapsulation. Tight coupling occurs when an abstraction is leaky, or when encapsulation is broken. If a class offers an incomplete set of services, other routines might find they need to read or write its internal data directly. That opens up the class, making it a glass box instead of a black box, and it virtually eliminates the class's encapsulation.

6.3 Design and Implementation Issues

Defining good class interfaces goes a long way toward creating a high-quality program. The internal class design and implementation are also important. This section discusses issues related to containment, inheritance, member functions and data, class coupling, constructors, and value-vs.-reference objects.

Containment ("has a" Relationships)

Containment is the simple idea that a class contains a primitive data element or object. A lot more is written about inheritance than about containment, but that's because inheritance is more tricky and error-prone, not because it's better. Containment is the work-horse technique in object-oriented programming.

Implement "has a" through containment One way of thinking of containment is as a "has a" relationship. For example, an employee "has a" name, "has a" phone number, "has a" tax ID, and so on. You can usually accomplish this by making the name, phone number, and tax ID member data of the *Employee* class.

Implement "has a" through private inheritance as a last resort In some instances you might find that you can't achieve containment through making one object a member of another. In that case, some experts suggest privately inheriting from the contained object (Meyers 1998, Sutter 2000). The main reason you would do that is to set up the containing class to access protected member functions or protected member data of the class that's contained. In practice, this approach creates an overly cozy relationship with the ancestor class and violates encapsulation. It tends to point to design errors that should be resolved some way other than through private inheritance.

Be critical of classes that contain more than about seven data members The number "7±2" has been found to be a number of discrete items a person can remember while performing other tasks (Miller 1956). If a class contains more than about seven data

members, consider whether the class should be decomposed into multiple smaller classes (Riel 1996). You might err more toward the high end of 7±2 if the data members are primitive data types like integers and strings, more toward the lower end of 7±2 if the data members are complex objects.

Inheritance ("is a" Relationships)

Inheritance is the idea that one class is a specialization of another class. The purpose of inheritance is to create simpler code by defining a base class that specifies common elements of two or more derived classes. The common elements can be routine interfaces, implementations, data members, or data types. Inheritance helps avoid the need to repeat code and data in multiple locations by centralizing it within a base class.

When you decide to use inheritance, you have to make several decisions:

- For each member routine, will the routine be visible to derived classes? Will it have a default implementation? Will the default implementation be overridable?
- For each data member (including variables, named constants, enumerations, and so on), will the data member be visible to derived classes?

The following subsections explain the ins and outs of making these decisions:

The single most important rule in object-oriented programming with C++ is this: public inheritance means "is a." Commit this rule to memory.
—*Scott Meyers*

Implement "is a" through public inheritance When a programmer decides to create a new class by inheriting from an existing class, that programmer is saying that the new class "is a" more specialized version of the older class. The base class sets expectations about how the derived class will operate and imposes constraints on how the derived class can operate (Meyers 1998).

If the derived class isn't going to adhere *completely* to the same interface contract defined by the base class, inheritance is not the right implementation technique. Consider containment or making a change further up the inheritance hierarchy.

Design and document for inheritance or prohibit it Inheritance adds complexity to a program, and, as such, it's a dangerous technique. As Java guru Joshua Bloch says, "Design and document for inheritance, or prohibit it." If a class isn't designed to be inherited from, make its members non-*virtual* in C++, *final* in Java, or non-*overridable* in Microsoft Visual Basic so that you can't inherit from it.

Adhere to the Liskov Substitution Principle (LSP) In one of object-oriented programming's seminal papers, Barbara Liskov argued that you shouldn't inherit from a base class unless the derived class truly "is a" more specific version of the base class (Liskov 1988). Andy Hunt and Dave Thomas summarize LSP like this: "Subclasses must be usable through the base class interface without the need for the user to know the difference" (Hunt and Thomas 2000).

In other words, all the routines defined in the base class should mean the same thing when they're used in each of the derived classes.

If you have a base class of *Account* and derived classes of *CheckingAccount*, *SavingsAccount*, and *AutoLoanAccount*, a programmer should be able to invoke any of the routines derived from *Account* on any of *Account*'s subtypes without caring about which subtype a specific account object is.

If a program has been written so that the Liskov Substitution Principle is true, inheritance is a powerful tool for reducing complexity because a programmer can focus on the generic attributes of an object without worrying about the details. If a programmer must be constantly thinking about semantic differences in subclass implementations, then inheritance is increasing complexity rather than reducing it. Suppose a programmer has to think this: "If I call the *InterestRate()* routine on *CheckingAccount* or *SavingsAccount*, it returns the interest the bank pays, but if I call *InterestRate()* on *AutoLoanAccount* I have to change the sign because it returns the interest the consumer pays to the bank." According to LSP, *AutoLoanAccount* should not inherit from the *Account* base class in this example because the semantics of the *InterestRate()* routine are not the same as the semantics of the base class's *InterestRate()* routine.

Be sure to inherit only what you want to inherit A derived class can inherit member routine interfaces, implementations, or both. Table 6-1 shows the variations of how routines can be implemented and overridden.

Table 6-1 Variations on Inherited Routines

	Overridable	Not Overridable
Implementation: Default Provided	Overridable Routine	Non-Overridable Routine
Implementation: No Default Provided	Abstract Overridable Routine	Not used (doesn't make sense to leave a routine undefined and not allow it to be overridden)

As the table suggests, inherited routines come in three basic flavors:

- An *abstract overridable routine* means that the derived class inherits the routine's interface but not its implementation.
- An *overridable routine* means that the derived class inherits the routine's interface and a default implementation and it is allowed to override the default implementation.
- A *non-overridable routine* means that the derived class inherits the routine's interface and its default implementation and it is not allowed to override the routine's implementation.

When you choose to implement a new class through inheritance, think through the kind of inheritance you want for each member routine. Beware of inheriting implementation just because you're inheriting an interface, and beware of inheriting an interface just because you want to inherit an implementation. If you want to use a class's implementation but not its interface, use containment rather than inheritance.

Don't "override" a non-overridable member function Both C++ and Java allow a programmer to override a non-overridable member routine—kind of. If a function is *private* in the base class, a derived class can create a function with the same name. To the programmer reading the code in the derived class, such a function can create confusion because it looks like it should be polymorphic, but it isn't; it just has the same name. Another way to state this guideline is, "Don't reuse names of non-overridable base-class routines in derived classes."

Move common interfaces, data, and behavior as high as possible in the inheritance tree The higher you move interfaces, data, and behavior, the more easily derived classes can use them. How high is too high? Let *abstraction* be your guide. If you find that moving a routine higher would break the higher object's abstraction, don't do it.

Be suspicious of classes of which there is only one instance A single instance might indicate that the design confuses objects with classes. Consider whether you could just create an object instead of a new class. Can the variation of the derived class be represented in data rather than as a distinct class? The Singleton pattern is one notable exception to this guideline.

Be suspicious of base classes of which there is only one derived class When I see a base class that has only one derived class, I suspect that some programmer has been "designing ahead"—trying to anticipate future needs, usually without fully understanding what those future needs are. The best way to prepare for future work is not to design extra layers of base classes that "might be needed someday"; it's to make current work as clear, straightforward, and simple as possible. That means not creating any more inheritance structure than is absolutely necessary.

Be suspicious of classes that override a routine and do nothing inside the derived routine This typically indicates an error in the design of the base class. For instance, suppose you have a class *Cat* and a routine *Scratch()* and suppose that you eventually find out that some cats are declawed and can't scratch. You might be tempted to create a class derived from *Cat* named *ScratchlessCat* and override the *Scratch()* routine to do nothing. This approach presents several problems:

- It violates the abstraction (interface contract) presented in the *Cat* class by changing the semantics of its interface.
- This approach quickly gets out of control when you extend it to other derived classes. What happens when you find a cat without a tail? Or a cat that doesn't catch mice? Or a cat that doesn't drink milk? Eventually you'll end up with derived classes like *ScratchlessTaillessMicelessMilklessCat*.

- Over time, this approach gives rise to code that's confusing to maintain because the interfaces and behavior of the ancestor classes imply little or nothing about the behavior of their descendants.

The place to fix this problem is not in the base class, but in the original *Cat* class. Create a *Claws* class and contain that within the *Cats* class. The root problem was the assumption that all cats scratch, so fix that problem at the source, rather than just bandaging it at the destination.

Avoid deep inheritance trees Object-oriented programming provides a large number of techniques for managing complexity. But every powerful tool has its hazards, and some object-oriented techniques have a tendency to increase complexity rather than reduce it.

In his excellent book *Object-Oriented Design Heuristics* (1996), Arthur Riel suggests limiting inheritance hierarchies to a maximum of six levels. Riel bases his recommendation on the "magic number 7±2," but I think that's grossly optimistic. In my experience most people have trouble juggling more than two or three levels of inheritance in their brains at once. The "magic number 7±2" is probably better applied as a limit to the *total number of subclasses* of a base class rather than the number of levels in an inheritance tree.

Deep inheritance trees have been found to be significantly associated with increased fault rates (Basili, Briand, and Melo 1996). Anyone who has ever tried to debug a complex inheritance hierarchy knows why. Deep inheritance trees increase complexity, which is exactly the opposite of what inheritance should be used to accomplish. Keep the primary technical mission in mind. Make sure you're using inheritance to avoid duplicating code and to *minimize complexity*.

Prefer polymorphism to extensive type checking Frequently repeated *case* statements sometimes suggest that inheritance might be a better design choice, although this is not always true. Here is a classic example of code that cries out for a more object-oriented approach:

C++ Example of a Case Statement That Probably Should Be Replaced by Polymorphism

```
switch ( shape.type ) {
   case Shape_Circle:
      shape.DrawCircle();
      break;
   case Shape_Square:
      shape.DrawSquare();
      break;
   ...
}
```

In this example, the calls to *shape.DrawCircle()* and *shape.DrawSquare()* should be replaced by a single routine named *shape.Draw()*, which can be called regardless of whether the shape is a circle or a square.

On the other hand, sometimes *case* statements are used to separate truly different kinds of objects or behavior. Here is an example of a *case* statement that is appropriate in an object-oriented program:

C++ Example of a Case Statement That Probably Should Not Be Replaced by Polymorphism

```
switch ( ui.Command() ) {
   case Command_OpenFile:
      OpenFile();
      break;
   case Command_Print:
      Print();
      break;
   case Command_Save:
      Save();
      break;
   case Command_Exit:
      ShutDown();
      break;
   ...
}
```

In this case, it would be possible to create a base class with derived classes and a polymorphic *DoCommand()* routine for each command (as in the Command pattern). But in a simple case like this one, the meaning of *DoCommand()* would be so diluted as to be meaningless, and the *case* statement is the more understandable solution.

Make all data private, not protected As Joshua Bloch says, "Inheritance breaks encapsulation" (2001). When you inherit from an object, you obtain privileged access to that object's protected routines and data. If the derived class really needs access to the base class's attributes, provide protected accessor functions instead.

Multiple Inheritance

The one indisputable fact about multiple inheritance in C++ is that it opens up a Pandora's box of complexities that simply do not exist under single inheritance.
—*Scott Meyers*

Inheritance is a power tool. It's like using a chain saw to cut down a tree instead of a manual crosscut saw. It can be incredibly useful when used with care, but it's dangerous in the hands of someone who doesn't observe proper precautions.

If inheritance is a chain saw, multiple inheritance is a 1950s-era chain saw with no blade guard, no automatic shutoff, and a finicky engine. There are times when such a tool is valuable; mostly, however, you're better off leaving the tool in the garage where it can't do any damage.

Although some experts recommend broad use of multiple inheritance (Meyer 1997), in my experience multiple inheritance is useful primarily for defining "mixins," simple classes that are used to add a set of properties to an object. Mixins are called mixins because they allow properties to be "mixed in" to derived classes. Mixins might be classes like *Displayable*, *Persistant*, *Serializable*, or *Sortable*. Mixins are nearly always abstract and aren't meant to be instantiated independently of other objects.

Mixins require the use of multiple inheritance, but they aren't subject to the classic diamond-inheritance problem associated with multiple inheritance as long as all mixins are truly independent of each other. They also make the design more comprehensible by "chunking" attributes together. A programmer will have an easier time understanding that an object uses the mixins *Displayable* and *Persistent* than understanding that an object uses the 11 more-specific routines that would otherwise be needed to implement those two properties.

Java and Visual Basic recognize the value of mixins by allowing multiple inheritance of interfaces but only single-class inheritance. C++ supports multiple inheritance of both interface and implementation. Programmers should use multiple inheritance only after carefully considering the alternatives and weighing the impact on system complexity and comprehensibility.

Why Are There So Many Rules for Inheritance?

KEY POINT

This section has presented numerous rules for staying out of trouble with inheritance. The underlying message of all these rules is that *inheritance tends to work against the primary technical imperative you have as a programmer, which is to manage complexity*. For the sake of controlling complexity, you should maintain a heavy bias against inheritance. Here's a summary of when to use inheritance and when to use containment:

Cross-Reference For more on complexity, see "Software's Primary Technical Imperative: Managing Complexity" in Section 5.2.

- If multiple classes share common data but not behavior, create a common object that those classes can contain.
- If multiple classes share common behavior but not data, derive them from a common base class that defines the common routines.
- If multiple classes share common data and behavior, inherit from a common base class that defines the common data and routines.
- Inherit when you want the base class to control your interface; contain when you want to control your interface.

Member Functions and Data

Cross-Reference For more discussion of routines in general, see Chapter 7, "High-Quality Routines."

Here are a few guidelines for implementing member functions and member data effectively.

Keep the number of routines in a class as small as possible A study of C++ programs found that higher numbers of routines per class were associated with higher fault rates (Basili, Briand, and Melo 1996). However, other competing factors were found to be more significant, including deep inheritance trees, large number of routines called within a class, and strong coupling between classes. Evaluate the tradeoff between minimizing the number of routines and these other factors.

Disallow implicitly generated member functions and operators you don't want Sometimes you'll find that you want to disallow certain functions—perhaps you want to disallow assignment, or you don't want to allow an object to be constructed. You might think that, since the compiler generates operators automatically, you're stuck allowing access. But in such cases you can disallow those uses by declaring the constructor, assignment operator, or other function or operator *private*, which will prevent clients from accessing it. (Making the constructor private is a standard technique for defining a singleton class, which is discussed later in this chapter.)

Minimize the number of different routines called by a class One study found that the number of faults in a class was statistically correlated with the total number of routines that were called from within a class (Basili, Briand, and Melo 1996). The same study found that the more classes a class used, the higher its fault rate tended to be. These concepts are sometimes called "fan out."

Further Reading Good accounts of the Law of Demeter can be found in *Pragmatic Programmer* (Hunt and Thomas 2000), *Applying UML and Patterns* (Larman 2001), and *Fundamentals of Object-Oriented Design in UML* (Page-Jones 2000).

Minimize indirect routine calls to other classes Direct connections are hazardous enough. Indirect connections—such as *account.ContactPerson().DaytimeContactInfo().PhoneNumber()*—tend to be even more hazardous. Researchers have formulated a rule called the "Law of Demeter" (Lieberherr and Holland 1989), which essentially states that Object A can call any of its own routines. If Object A instantiates an Object B, it can call any of Object B's routines. But it should avoid calling routines on objects provided by Object B. In the *account* example above, that means *account.ContactPerson()* is OK but *account.ContactPerson().DaytimeContactInfo()* is not.

This is a simplified explanation. See the additional resources at the end of this chapter for more details.

In general, minimize the extent to which a class collaborates with other classes Try to minimize all of the following:

- Number of kinds of objects instantiated
- Number of different direct routine calls on instantiated objects
- Number of routine calls on objects returned by other instantiated objects

Constructors

Following are some guidelines that apply specifically to constructors. Guidelines for constructors are pretty similar across languages (C++, Java, and Visual Basic, anyway). Destructors vary more, so you should check out the materials listed in this chapter's "Additional Resources" section for information on destructors.

Initialize all member data in all constructors, if possible Initializing all data members in all constructors is an inexpensive defensive programming practice.

Enforce the singleton property by using a private constructor If you want to define a class that allows only one object to be instantiated, you can enforce this by hiding all the constructors of the class and then providing a *static GetInstance()* routine to access the class's single instance. Here's an example of how that would work:

Further Reading The code to do this in C++ would be similar. For details, see *More Effective C++*, Item 26 (Meyers 1998).

Java Example of Enforcing a Singleton with a Private Constructor

```
public class MaxId {
   // constructors and destructors
   private MaxId() {
      ...
   }
   ...

   // public routines
   public static MaxId GetInstance() {
      return m_instance;
   }
   ...

   // private members
   private static final MaxId m_instance = new MaxId();
   ...
}
```

Here is the private constructor.

Here is the public routine that provides access to the single instance.

Here is the single instance.

The private constructor is called only when the *static* object *m_instance* is initialized. In this approach, if you want to reference the *MaxId* singleton, you would simply refer to *MaxId.GetInstance()*.

Prefer deep copies to shallow copies until proven otherwise One of the major decisions you'll make about complex objects is whether to implement deep copies or shallow copies of the object. A deep copy of an object is a member-wise copy of the object's member data; a shallow copy typically just points to or refers to a single reference copy, although the specific meanings of "deep" and "shallow" vary.

The motivation for creating shallow copies is typically to improve performance. Although creating multiple copies of large objects might be aesthetically offensive, it rarely causes any measurable performance impact. A small number of objects might cause performance issues, but programmers are notoriously poor at guessing which code really causes problems. (For details, see Chapter 25, "Code-Tuning Strategies.")

Because it's a poor tradeoff to add complexity for dubious performance gains, a good approach to deep vs. shallow copies is to prefer deep copies until proven otherwise.

Deep copies are simpler to code and maintain than shallow copies. In addition to the code either kind of object would contain, shallow copies add code to count references, ensure safe object copies, safe comparisons, safe deletes, and so on. This code can be error-prone, and you should avoid it unless there's a compelling reason to create it.

If you find that you do need to use a shallow-copy approach, Scott Meyers's *More Effective C++*, Item 29 (1996) contains an excellent discussion of the issues in C++. Martin Fowler's *Refactoring* (1999) describes the specific steps needed to convert from shallow copies to deep copies and from deep copies to shallow copies. (Fowler calls them reference objects and value objects.)

6.4 Reasons to Create a Class

Cross-Reference Reasons for creating classes and routines overlap. See Section 7.1.

If you believe everything you read, you might get the idea that the only reason to create a class is to model real-world objects. In practice, classes get created for many more reasons than that. Here's a list of good reasons to create a class.

Cross-Reference For more on identifying real-world objects, see "Find Real-World Objects" in Section 5.3.

Model real-world objects Modeling real-world objects might not be the only reason to create a class, but it's still a good reason! Create a class for each real-world object type that your program models. Put the data needed for the object into the class, and then build service routines that model the behavior of the object. See the discussion of ADTs in Section 6.1 for examples.

Model abstract objects Another good reason to create a class is to model an *abstract object*—an object that isn't a concrete, real-world object but that provides an abstraction of other concrete objects. A good example is the classic *Shape* object. *Circle* and *Square* really exist, but *Shape* is an abstraction of other specific shapes.

On programming projects, the abstractions are not ready-made the way *Shape* is, so we have to work harder to come up with clean abstractions. The process of distilling abstract concepts from real-world entities is non-deterministic, and different designers will abstract out different generalities. If we didn't know about geometric shapes like circles, squares and triangles, for example, we might come up with more unusual shapes like squash shape, rutabaga shape, and Pontiac Aztek shape. Coming up with appropriate abstract objects is one of the major challenges in object-oriented design.

KEY POINT

Reduce complexity The single most important reason to create a class is to reduce a program's complexity. Create a class 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 class. But after it's written, you should be able to forget the details and use the class without any knowledge of its internal workings. Other reasons to create classes—minimizing code size,

improving maintainability, and improving correctness—are also good reasons, but without the abstractive power of classes, complex programs would be impossible to manage intellectually.

Isolate complexity Complexity in all forms—complicated algorithms, large data sets, intricate communications protocols, and so on—is prone to errors. If an error does occur, it will be easier to find if it isn't spread through the code but is localized within a class. Changes arising from fixing the error won't affect other code because only one class 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 class. During development, it will be easier to try several designs and keep the one that works best.

Hide implementation details The desire to hide implementation details is a wonderful reason to create a class whether the details are as complicated as a convoluted database access or as mundane as whether a specific data member is stored as a number or a string.

Limit effects of changes Isolate areas that are likely to change so that the effects of changes are limited to the scope of a single class or a few classes. 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 types, and business rules. The subsection titled "Hide Secrets (Information Hiding)" in Section 5.3 described several common sources of change.

Cross-Reference For a discussion of problems associated with using global data, see Section 13.3, "Global Data."

Hide global data If you need to use global data, you can hide its implementation details behind a class interface. Working with global data through access routines provides several benefits compared to working with global data directly. 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 often becomes apparent that the "global data" is really just object data.

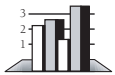
Streamline parameter passing If you're passing a parameter among several routines, that might indicate a need to factor those routines into a class that share the parameter as object data. Streamlining parameter passing isn't a goal, per se, but passing lots of data around suggests that a different class organization might work better.

Cross-Reference For details on information hiding, see "Hide Secrets (Information Hiding)" in Section 5.3.

Make central points of control It's a good idea to control each task in one place. Control assumes many forms. Knowledge of the number of entries in a table is one form. Control of devices—files, database connections, printers, and so on—is another. Using one class to read from and write to a database is a form of centralized control. If the database needs to be converted to a flat file or to in-memory data, the changes will affect only one class.

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.

Facilitate reusable code Code put into well-factored classes can be reused in other programs more easily than the same code embedded in one larger class. Even if a section of code is called from only one place in the program and is understandable as part of a larger class, it makes sense to put it into its own class if that piece of code might be used in another program.



HARD DATA

NASA's Software Engineering Laboratory studied ten projects that pursued reuse aggressively (McGarry, Waligora, and McDermott 1989). In both the object-oriented and the functionally oriented approaches, the initial projects weren't able to take much of their code from previous projects because previous projects hadn't established a sufficient code base. Subsequently, the projects that used functional design were able to take about 35 percent of their code from previous projects. Projects that used an object-oriented approach were able to take more than 70 percent of their code from previous projects. If you can avoid writing 70 percent of your code by planning ahead, do it!

Cross-Reference For more on implementing the minimum amount of functionality required, see "A program contains code that seems like it might be needed someday" in Section 24.2.

Notably, the core of NASA's approach to creating reusable classes does not involve "designing for reuse." NASA identifies reuse candidates at the ends of their projects. They then perform the work needed to make the classes reusable as a special project at the end of the main project or as the first step in a new project. This approach helps prevent "gold-plating"—creation of functionality that isn't required and that unnecessarily adds complexity.

Plan 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 classes. You can then modify the classes without affecting the rest of the program, or you can put in completely new classes instead. Thinking through not just what one program will look like but what the whole family of programs might look like is a powerful heuristic for anticipating entire categories of changes (Parnas 1976).

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 classes 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 factored the program so that each part that varied from client to client was in its own class. The initial programming might have taken three months or so, but when we got a new client, we merely wrote a handful of new classes for the new client and dropped them into the rest of the code. A few days' work and—voila!—custom software!

Package related operations In cases in which you can't hide information, share data, or plan for flexibility, you can still package sets of operations into sensible groups, such as trig functions, statistical functions, string-manipulation routines, bit-manipulation routines, graphics routines, and so on. Classes are one means of combining related operations. You could also use packages, namespaces, or header files, depending on the language you're working in.

Accomplish a specific refactoring Many of the specific refactorings described in Chapter 24, "Refactoring," result in new classes—including converting one class to two, hiding a delegate, removing a middle man, and introducing an extension class. These new classes could be motivated by a desire to better accomplish any of the objectives described throughout this section.

Classes to Avoid

While classes in general are good, you can run into a few gotchas. Here are some classes to avoid.

Avoid creating god classes Avoid creating omniscient classes that are all-knowing and all-powerful. If a class spends its time retrieving data from other classes using *Get()* and *Set()* routines (that is, digging into their business and telling them what to do), ask whether that functionality might better be organized into those other classes rather than into the god class (Riel 1996).

Cross-Reference This kind of class is usually called a structure. For more on structures, see Section 13.1, "Structures."

Eliminate irrelevant classes If a class consists only of data but no behavior, ask yourself whether it's really a class and consider demoting it so that its member data just becomes attributes of one or more other classes.

Avoid classes named after verbs A class that has only behavior but no data is generally not really a class. Consider turning a class like *DatabaseInitialization()* or *StringBuilder()* into a routine on some other class.

Summary of Reasons to Create a Class

Here's a summary list of the valid reasons to create a class:

- Model real-world objects
- Model abstract objects
- Reduce complexity
- Isolate complexity
- Hide implementation details
- Limit effects of changes
- Hide global data

- Streamline parameter passing
- Make central points of control
- Facilitate reusable code
- Plan for a family of programs
- Package related operations
- Accomplish a specific refactoring

6.5 Language-Specific Issues

Approaches to classes in different programming languages vary in interesting ways. Consider how you override a member routine to achieve polymorphism in a derived class. In Java, all routines are overridable by default and a routine must be declared *final* to prevent a derived class from overriding it. In C++, routines are not overridable by default. A routine must be declared *virtual* in the base class to be overridable. In Visual Basic, a routine must be declared *overridable* in the base class and the derived class should use the *overrides* keyword.

Here are some of the class-related areas that vary significantly depending on the language:

- Behavior of overridden constructors and destructors in an inheritance tree
- Behavior of constructors and destructors under exception-handling conditions
- Importance of default constructors (constructors with no arguments)
- Time at which a destructor or finalizer is called
- Wisdom of overriding the language's built-in operators, including assignment and equality
- How memory is handled as objects are created and destroyed or as they are declared and go out of scope

Detailed discussions of these issues are beyond the scope of this book, but the "Additional Resources" section points to good language-specific resources.

6.6 Beyond Classes: Packages

Cross-Reference For more on the distinction between classes and packages, see "Levels of Design" in Section 5.2.

Classes are currently the best way for programmers to achieve modularity. But modularity is a big topic, and it extends beyond classes. Over the past several decades, software development has advanced in large part by increasing the granularity of the aggregations that we have to work with. The first aggregation we had was the statement, which

at the time seemed like a big step up from machine instructions. Then came subroutines, and later came classes.

It's evident that we could better support the goals of abstraction and encapsulation if we had good tools for aggregating groups of objects. Ada supported the notion of packages more than a decade ago, and Java supports packages today. If you're programming in a language that doesn't support packages directly, you can create your own poor-programmer's version of a package and enforce it through programming standards that include the following:

- Naming conventions that differentiate which classes are public and which are for the package's private use
- Naming conventions, code-organization conventions (project structure), or both that identify which package each class belongs to
- Rules that define which packages are allowed to use which other packages, including whether the usage can be inheritance, containment, or both

These workarounds are good examples of the distinction between programming *in* a language vs. programming *into* a language. For more on this distinction, see Section 34.4, "Program into Your Language, Not in It."

cc2e.com/0672

Cross-Reference This is a checklist of considerations about the quality of the class. For a list of the steps used to build a class, see the checklist "The Pseudocode Programming Process" in Chapter 9, page 233.

CHECKLIST: Class Quality

Abstract Data Types

- ❑ Have you thought of the classes in your program as abstract data types and evaluated their interfaces from that point of view?

Abstraction

- ❑ Does the class have a central purpose?
- ❑ Is the class well named, and does its name describe its central purpose?
- ❑ Does the class's interface present a consistent abstraction?
- ❑ Does the class's interface make obvious how you should use the class?
- ❑ Is the class's interface abstract enough that you don't have to think about how its services are implemented? Can you treat the class as a black box?
- ❑ Are the class's services complete enough that other classes don't have to meddle with its internal data?
- ❑ Has unrelated information been moved out of the class?
- ❑ Have you thought about subdividing the class into component classes, and have you subdivided it as much as you can?
- ❑ Are you preserving the integrity of the class's interface as you modify the class?

Encapsulation

- ❑ Does the class minimize accessibility to its members?
- ❑ Does the class avoid exposing member data?
- ❑ Does the class hide its implementation details from other classes as much as the programming language permits?
- ❑ Does the class avoid making assumptions about its users, including its derived classes?
- ❑ Is the class independent of other classes? Is it loosely coupled?

Inheritance

- ❑ Is inheritance used only to model "is a" relationships—that is, do derived classes adhere to the Liskov Substitution Principle?
- ❑ Does the class documentation describe the inheritance strategy?
- ❑ Do derived classes avoid "overriding" non-overridable routines?
- ❑ Are common interfaces, data, and behavior as high as possible in the inheritance tree?
- ❑ Are inheritance trees fairly shallow?
- ❑ Are all data members in the base class private rather than protected?

Other Implementation Issues

- ❑ Does the class contain about seven data members or fewer?
- ❑ Does the class minimize direct and indirect routine calls to other classes?
- ❑ Does the class collaborate with other classes only to the extent absolutely necessary?
- ❑ Is all member data initialized in the constructor?
- ❑ Is the class designed to be used as deep copies rather than shallow copies unless there's a measured reason to create shallow copies?

Language-Specific Issues

- ❑ Have you investigated the language-specific issues for classes in your specific programming language?

Additional Resources

Classes in General

cc2e.com/0679

Meyer, Bertrand. *Object-Oriented Software Construction*, 2d ed. New York, NY: Prentice Hall PTR, 1997. This book contains an in-depth discussion of abstract data types and explains how they form the basis for classes. Chapters 14–16 discuss inheritance in depth. Meyer provides an argument in favor of multiple inheritance in Chapter 15.

Riel, Arthur J. *Object-Oriented Design Heuristics*. Reading, MA: Addison-Wesley, 1996. This book contains numerous suggestions for improving program design, mostly at the class level. I avoided the book for several years because it appeared to be too big—talk about people in glass houses! However, the body of the book is only about 200 pages long. Riel's writing is accessible and enjoyable. The content is focused and practical.

C++

cc2e.com/0686

Meyers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*, 2d ed. Reading, MA: Addison-Wesley, 1998.

Meyers, Scott, 1996, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Reading, MA: Addison-Wesley, 1996. Both of Meyers' books are canonical references for C++ programmers. The books are entertaining and help to instill a language-lawyer's appreciation for the nuances of C++.

Java

cc2e.com/0693

Bloch, Joshua. *Effective Java Programming Language Guide*. Boston, MA: Addison-Wesley, 2001. Bloch's book provides much good Java-specific advice as well as introducing more general, good object-oriented practices.

Visual Basic

cc2e.com/0600

The following books are good references on classes in Visual Basic:

Foxall, James. *Practical Standards for Microsoft Visual Basic .NET*. Redmond, WA: Microsoft Press, 2003.

Cornell, Gary, and Jonathan Morrison. *Programming VB .NET: A Guide for Experienced Programmers*. Berkeley, CA: Apress, 2002.

Barwell, Fred, et al. *Professional VB.NET*, 2d ed. Wrox, 2002.

Key Points

- Class interfaces should provide a consistent abstraction. Many problems arise from violating this single principle.
- A class interface should hide something—a system interface, a design decision, or an implementation detail.
- Containment is usually preferable to inheritance unless you're modeling an "is a" relationship.
- Inheritance is a useful tool, but it adds complexity, which is counter to Software's Primary Technical Imperative of managing complexity.
- Classes are your primary tool for managing complexity. Give their design as much attention as needed to accomplish that objective.

Chapter 7

High-Quality Routines

cc2e.com/0778

Contents

Related Topics

- Steps in routine construction: Section 9.3
- Working classes: Chapter 6
- General design techniques: Chapter 5
- Software architecture: Section 3.5

Chapter 6 described the details of creating classes. This chapter zooms in on routines, on the characteristics that make the difference between a good routine and a bad one. If you'd rather read about issues that affect the design of routines before wading into the nitty-gritty details, be sure to read Chapter 5, "Design in Construction," first and come back to this chapter later. Some important attributes of high-quality routines are also discussed in Chapter 8, "Defensive Programming." If you're more interested in reading about steps to create routines and classes, Chapter 9, "The Pseudocode Programming Process," might be a better place to start.

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 method or procedure invocable for a single purpose. Examples include a function in C++, a method in Java, a function or sub procedure in Microsoft Visual Basic. For some uses, macros in C and C++ 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:

C++ Example of a Low-Quality Routine

```
void HandleStuff( CORP_DATA & inputRec, int crntQtr, EMP_DATA empRec,
   double & estimRevenue, double ytdRevenue, int screenX, int screenY,
   COLOR_TYPE & newColor, COLOR_TYPE & prevColor, StatusType & status,
   int expenseType )
{
int i;
for ( i = 0; i < 100; i++ ) {
   inputRec.revenue[i] = 0;
   inputRec.expense[i] = corpExpense[ crntQtr ][ i ];
   }
UpdateCorpDatabase( empRec );
estimRevenue = ytdRevenue * 4.0 / (double) crntQtr;
newColor = prevColor;
status = SUCCESS;
if ( expenseType == 1 ) {
     for ( i = 0; i < 12; i++ )
           profit[i] = revenue[i] - expense.type1[i];
     }
else if ( expenseType == 2 )  {
          profit[i] = revenue[i] - expense.type2[i];
          }
else if ( expenseType == 3 )
          profit[i] = revenue[i] - expense.type3[i];
          }
```

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 following list:

- 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 32, "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 31, "Layout and Style.")
- The routine's input variable, *inputRec,* is changed. If it's an input variable, its value should not be modified (and in C++ it should be declared *const*). 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*, the expression *ytdRevenue * 4.0 / (double) 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 12.1, "Numbers in General."
- Some of the routine's parameters are unused: *screenX* and *screenY* are not referenced within the routine.
- One of the routine's parameters is passed incorrectly: *prevColor* is labeled as a reference 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 unreadable 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 32.)

cc2e.com/0799

Cross-Reference The class is also a good contender for the single greatest invention in computer science. For details on how to use classes effectively, see Chapter 6, "Working Classes."

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, and it's a crime to abuse this senior statesman of computer science with code like that in the example just shown.

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 many valid reasons to create a routine exist 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 extent of the textbook's coverage. It was not a good or complete explanation of the theory and practice of routines. The following sections contain a much better explanation.

7.1 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.

Reduce 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.

Introduce an intermediate, understandable abstraction Putting a section of code into a well-named routine is one of the best ways to document its purpose. Instead of reading a series of statements like

```
if ( node <> NULL ) then
   while ( node.next <> NULL ) do
      node = node.next
      leafName = node.name
   end while
else
   leafName = ""
end if
```

you can read a statement like this:

```
leafName = GetLeafName( node )
```

The new routine is so short that nearly all it needs for documentation is a good name. The name introduces a higher level of abstraction than the original eight lines of code, which makes the code more readable and easier to understand, and it reduces complexity within the routine that originally contained the code.

Avoid 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 a base class, and then move the two specialized routines into subclasses. Alternatively, you could migrate 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.

Support subclassing You need less new code to override a short, well-factored routine than a long, poorly factored routine. You'll also reduce the chance of error in subclass implementations if you keep overrideable routines simple.

Hide 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. Another example of a sequence might be found when you have two lines of code that read the top of a stack and decrement a *stackTop* variable. Put those two lines of code into a *PopStack()* routine to hide the assumption about the order in which the two operations must be performed. Hiding that assumption will be better than baking it into code from one end of the system to the other.

Hide 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 on 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 type than pointers, you can change the program without traumatizing the code that would have used the pointers.

Improve 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.

Simplify 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. Simplifying a boolean test is an example of reducing complexity, which was discussed earlier.

Improve performance You can optimize the code in one place instead of in several places. Having code in one place will make it easier to profile to find inefficiencies. Centralizing code into a routine means that a single optimization benefits all the code that uses that routine, whether it uses it directly or indirectly. Having code in one place makes it practical to recode the routine with a more efficient algorithm or in a faster, more efficient language.

Cross-Reference For details on information hiding, see "Hide Secrets (Information Hiding)" in Section 5.3.

To ensure all routines are small? No. 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 7.4, "How Long Can a Routine Be?")

Operations That Seem Too Simple to Put Into Routines

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:

Pseudocode Example of a Calculation

```
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:

Pseudocode Example of a Calculation Converted to a Function

```
Function DeviceUnitsToPoints ( deviceUnits Integer ): Integer
   DeviceUnitsToPoints = deviceUnits *
      ( POINTS_PER_INCH / DeviceUnitsPerInch() )
End Function
```

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

Pseudocode Example of a Function Call to a Calculation Function

```
points = DeviceUnitsToPoints( deviceUnits )
```

This line is 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, *DeviceUnitsPerlnch()* returned *0*. That meant I had to account for division by zero, which took three more lines of code:

Pseudocode Example of a Calculation That Expands Under Maintenance

```
Function DeviceUnitsToPoints( deviceUnits: Integer ) Integer;
   if ( DeviceUnitsPerInch() <> 0 )
      DeviceUnitsToPoints = deviceUnits *
         ( POINTS_PER_INCH / DeviceUnitsPerInch() )
   else
      DeviceUnitsToPoints = 0
   end if
End Function
```

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:

- Reduce complexity
- Introduce an intermediate, understandable abstraction
- Avoid duplicate code
- Support subclassing
- Hide sequences
- Hide pointer operations
- Improve portability
- Simplify complicated boolean tests
- Improve performance

In addition, many of the reasons to create a class are also good reasons to create a routine:

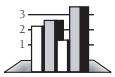
- Isolate complexity
- Hide implementation details
- Limit effects of changes
- Hide global data
- Make central points of control
- Facilitate reusable code
- Accomplish a specific refactoring

7.2 Design at the Routine Level

The idea of cohesion was introduced in a paper by Wayne Stevens, Glenford Myers, and Larry Constantine (1974). Other more modern concepts, including abstraction and encapsulation, tend to yield more insight at the class level (and have, in fact, largely superceded cohesion at the class level), but cohesion is still alive and well as the workhorse design heuristic at the individual-routine level.

Cross-Reference For a discussion of cohesion in general, see "Aim for Strong Cohesion" in Section 5.3.

For routines, cohesion refers to how closely the operations in a routine are related. Some programmers prefer the term "strength": how strongly related are the operations in a routine? A function like *Cosine()* is perfectly cohesive because the whole routine is dedicated to performing one function. A function like *CosineAndTan()* 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 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 a different 450 routines (which is just an unusual coincidence) found that routines with the highest coupling-to-cohesion ratios had 7 times as many errors as those with the lowest coupling-to-cohesion 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.

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()*, *CalculateLoanPayment()*, and *AgeFromBirthdate()*. Of course, this evaluation of their cohesion assumes that the routines do what their names say they do—if they do anything else, they are less cohesive and poorly named.

Several other kinds of cohesion are normally considered to be less than ideal:

- *Sequential cohesion* exists 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.

  An example of sequential cohesion is a routine that, given a birth date, calculates an employee's age and time to retirement. 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.

How would you make the routine functionally cohesive? You'd create separate routines to compute an employee's age given a birth date and compute 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 routines.

- *Communicational cohesion* occurs when operations in a routine make use of the same data and aren't related in any other way. If a routine prints a summary report and then reinitializes the summary data passed into it, the routine has communicational cohesion: the two operations are related only by the fact that they use the same data.

  To give this routine better cohesion, 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.

- *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.

  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 *ReadConfigFileInitScratchFileEtc()*, which would imply that the routine had only coincidental cohesion. If you name it *Startup()*, however, it would be clear that it had a single purpose and clear that it had functional 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 rewrite to have better cohesion than investing in a pinpoint diagnosis of the problem. Knowing what to avoid can be useful, however, so here are the unacceptable kinds of cohesion:

- *Procedural cohesion* occurs when operations in a routine are done in a specified order. An example is a routine that 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.

  To achieve better cohesion, put the separate operations into their own routines. Make sure that the calling routine has a single, complete job: *GetEmployee()* 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.

- *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. Considering that the defining attribute of logical cohesion is that the operations are unrelated, a better name might "illogical cohesion."

  One example would be an *InputAll()* routine that inputs customer names, employee timecard 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 moved into a lower-level routine and the routines should be packaged into a class.

  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 "event handler." An event handler is often used in interactive environments such as the Apple Macintosh, Microsoft Windows, and other GUI environments.

- *Coincidental cohesion* occurs when the operations in a routine have no discernible relationship to each other. Other good names are "no cohesion" or "chaotic cohesion." The low-quality C++ routine at the beginning of this chapter had coincidental cohesion. It's hard to convert coincidental cohesion to any better kind of cohesion—you usually need to do a deeper redesign and reimplementation.

Cross-Reference Although the routine might have better cohesion, a higher-level design issue is whether the system should be using a *case* statement instead of polymorphism. For more on this issue, see "Replace conditionals with polymorphism (especially repeated *case* statements)" in Section 24.3


KEY POINT

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.

7.3 Good Routine Names

Cross-Reference For details on naming variables, see Chapter 11, "The Power of Variable Names."

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

Describe everything the routine does In the routine's name, describe all the outputs and side effects. If a routine computes report totals and opens an output file, *ComputeReportTotals()* is not an adequate name for the routine. *ComputeReportTotalsAndOpenOutputFile()* 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.

Avoid meaningless, vague, or wishy-washy verbs Some verbs are elastic, stretched to cover just about any meaning. Routine names like *HandleCalculation()*, *PerformServices()*, *OutputUser()*, *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, users, 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.


CODING HORROR

Don't differentiate routine names solely by number One developer wrote all his code in one big function. Then he took every 15 lines and created functions named *Part1*, *Part2*, and so on. After that, he created one high-level function that called each part. This method of creating and naming routines is especially egregious (and rare, I hope). But programmers sometimes use numbers to differentiate routines with names like *OutputUser*, *OutputUser1*, and *OutputUser2*. The numerals at the ends of these names provide no indication of the different abstractions the routines represent, and the routines are thus poorly named.

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 com-

plicated than variables, and good names for them tend to be longer. On the other hand, routine names are often attached to object names, which essentially provides part of the name for free. Overall, the emphasis when creating a routine name should be to make the name as clear as possible, which means you should make its name as long or short as needed to make it understandable.

Cross-Reference For the distinction between procedures and functions, see Section 7.6, "Special Considerations in the Use of Functions," later in this chapter.

To name a function, 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()*, *customerId.Next()*, *printer.IsReady()*, and *pen.CurrentColor()* are all good function names that indicate precisely what the functions return.

To name a procedure, 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. *PrintDocument()*, *CalcMonthlyRevenues()*, *CheckOrderlnfo()*, 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 *document.Print()*, *orderInfo.Check()*, and *monthlyRevenues.Calc()*. Names like *document.PrintDocument()* are redundant and can become inaccurate when they're carried through to derived classes. If *Check* is a class derived from *Document*, *check.Print()* seems clearly to be printing a check, whereas *check.PrintDocument()* sounds like it might be printing a checkbook register or monthly statement, but it doesn't sound like it's printing a check.

Cross-Reference For a similar list of opposites in variable names, see "Common Opposites in Variable Names" in Section 11.1.

Use opposites precisely Using naming conventions for opposites helps consistency, which helps readability. Opposite-pairs like *first/last* are commonly understood. Opposite-pairs like *FileOpen()* and *_lclose()* are not symmetrical and are confusing. Here are some common opposites:

add/remove	increment/decrement	open/close
begin/end	insert/delete	show/hide
create/destroy	lock/unlock	source/target
first/last	min/max	start/stop
get/put	next/previous	up/down
get/set	old/new	

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.

The code on one of my projects assigned each object a unique identifier. We neglected to establish a convention for naming the routines that would return the object identifier, so we had routine names like these:

```
employee.id.Get()
dependent.GetId()
supervisor()
candidate.id()
```

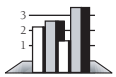
The *Employee* class exposed its *id* object, which in turn exposed its *Get()* routine. The *Dependent* class exposed a *GetId()* routine. The *Supervisor* class made the *id* its default return value. The *Candidate* class made use of the fact that the *id* object's default return value was the *id*, and exposed the *id* object. By the middle of the project, no one could remember which of these routines was supposed to be used on which object, but by that time too much code had been written to go back and make everything consistent. Consequently, every person on the team had to devote an unnecessary amount of gray matter to remembering the inconsequential detail of which syntax was used on which class to retrieve the *id*. A naming convention for retrieving *id*s would have eliminated this annoyance.

7.4 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 screen or one or two pages of program listing, approximately 50 to 150 lines. In this spirit, IBM once limited routines to 50 lines, and TRW limited them to two pages (McCabe 1976). Modern programs tend to have volumes of extremely short routines mixed in with a few longer routines. Long routines are far from extinct, however. Shortly before finishing this book, I visited two client sites within a month. Programmers at one site were wrestling with a routine that was about 4,000 lines of code long, and programmers at the other site were trying to tame a routine that was more than 12,000 lines long!

A mountain of research on routine length has accumulated over the years, some of which is applicable to modern programs, and some of which isn't:



HARD DATA

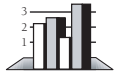
- A study by Basili and Perricone found that routine size was inversely correlated with errors: as the size of routines increased (up to 200 lines of code), the number of errors per line of code decreased (Basili and Perricone 1984).
- Another study found that routine size was not correlated with errors, 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 but were 2.4 times less expensive to fix than larger routines (Selby and Basili 1991).
- Another study found that code needed to be changed least when routines averaged 100 to 150 lines of code (Lind and Vairavan 1989).
- A study at IBM found that 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).

Where does all this leave the question of routine length in object-oriented programs? A large percentage of routines in object-oriented programs will be accessor routines, which will be very short. From time to time, a complex algorithm will lead to a longer routine, and in those circumstances, the routine should be allowed to grow organically up to 100–200 lines. (A line is a noncomment, nonblank line of source code.) Decades of evidence say that routines of such length are no more error prone than shorter routines. Let issues such as the routine's cohesion, depth of nesting, number of variables, number of decision points, number of comments needed to explain the routine, and other complexity-related considerations dictate the length of the routine rather than imposing a length restriction per se.

That said, if you want to write routines longer than about 200 lines, be careful. 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.

7.5 How to Use Routine Parameters



HARD DATA

Interfaces between routines are some of the most error-prone areas of a program. One often-cited 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:

Cross-Reference For details on documenting routine parameters, see "Commenting Routines" in Section 32.5. For details on formatting parameters, see Section 31.7, "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

Ada uses *in* and *out* keywords to make input and output parameters clear.

```
procedure InvertMatrix(
   originalMatrix: in Matrix;
   resultMatrix: out Matrix
);
...

procedure ChangeSentenceCase(
   desiredCase: in StringCase;
   sentence: in out Sentence
);
...

procedure PrintPageNumber(
   pageNumber: in Integer;
   status: out StatusType
);
```

This ordering convention conflicts with the C-library 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 will still do the readers of your code a service.

Consider creating your own* in *and* out *keywords Other modern languages don't support the *in* and *out* keywords like Ada does. In those languages, you might still be able to use the preprocessor to create your own *in* and *out* keywords:

C++ Example of Defining Your Own *In* and *Out* Keywords

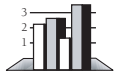
```
#define IN
#define OUT
void InvertMatrix(
   IN Matrix originalMatrix,
   OUT Matrix *resultMatrix
);
...

void ChangeSentenceCase(
   IN StringCase desiredCase,
   IN OUT Sentence *sentenceToEdit
);
...

void PrintPageNumber(
   IN int pageNumber,
   OUT StatusType &status
);
```

In this case, the *IN* and *OUT* macro-keywords are used for documentation purposes. To make the value of a parameter changeable by the called routine, the parameter still needs to be passed as a pointer or as a reference parameter.

Before adopting this technique, be sure to consider a pair of significant drawbacks. Defining your own *IN* and *OUT* keywords extends the C++ language in a way that will be unfamiliar to most people reading your code. If you extend the language this way, be sure to do it consistently, preferably projectwide. A second limitation is that the *IN* and *OUT* keywords won't be enforceable by the compiler, which means that you could potentially label a parameter as *IN* and then modify it inside the routine anyway. That could lull a reader of your code into assuming code is correct when it isn't. Using C++'s *const* keyword will normally be the preferable means of identifying input-only parameters.

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()* routine is the same as the *printf()* routine except that it adds a file as the first argument. A similar routine, *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 routines harder to remember than they need to be.

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



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, and 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 one exception. 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 following Java fragment, the variable *inputVal* is improperly used to store intermediate results of a computation:

Java Example of Improper Use of Input Parameters

```
int Sample( int inputVal ) {
   inputVal = inputVal * CurrentMultiplier( inputVal );
   inputVal = inputVal + CurrentAdder( inputVal );
   ...
   return inputVal;
}
```

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

In this code fragment, *inputVal* 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 *inputValThatBecomesWorkingVal* 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:

Java Example of Good Use of Input Parameters

```
int Sample( int inputVal ) {
   int workingVal = inputVal;
   workingVal = workingVal * CurrentMultiplier( workingVal );
   workingVal = workingVal + CurrentAdder( workingVal );
   ...

   ...
   return workingVal;
}
```

If you need to use the original value of *inputVal* 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. (Don't take this reasoning as a justification for literally naming a variable *inputVal* or *workingVal*. In general, *inputVal* and *workingVal* are terrible names for variables, and these names are used in this example only to make the variables' roles clear.)

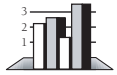
Assigning the input value to a working variable emphasizes where the value comes from. It eliminates the possibility that a variable from the parameter list will be modified accidentally. In C++, this practice can be enforced by the compiler using the keyword *const*. If you designate a parameter as *const*, you're not allowed to modify its value within a routine.

Cross-Reference For details on interface assumptions, see the introduction to Chapter 8, "Defensive Programming." For details on documentation, see Chapter 32, "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 types. If you program in a modern language that supports structured data, you can pass a composite data type 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 how to think about interfaces, see "Good Abstraction" 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 class and treat the frequently used data as class 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 the variables or objects that the routine needs to maintain its interface abstraction There are two competing schools of thought about how to pass members of an object to a routine. Suppose you have an object that exposes data through 10 access routines and the called routine needs three of those data elements to do its job.

Proponents of the first school of thought argue that only the three specific elements needed by the routine should be passed. They argue that doing this will keep the connections between routines to a minimum; reduce coupling; and make them easier to understand, reuse, and so on. They say that passing the whole object to a routine violates the principle of encapsulation by potentially exposing all 10 access routines to the routine that's called.

Proponents of the second school argue that the whole object should be passed. They argue that the interface can remain more stable if the called routine has the flexibility to use additional members of the object without changing the routine's interface. They argue that passing three specific elements violates encapsulation by exposing which specific data elements the routine is using.

I think both these rules are simplistic and miss the most important consideration: *what abstraction is presented by the routine's interface?* If the abstraction is that the routine expects you to have three specific data elements, and it is only a coincidence that those three elements happen to be provided by the same object, then you should pass the three specific data elements individually. However, if the abstraction is that you will always have that particular object in hand and the routine will do something or other with that object, then you truly do break the abstraction when you expose the three specific data elements.

If you're passing the whole object and you find yourself creating the object, populating it with the three elements needed by the called routine, and then pulling those elements out of the object after the routine is called, that's an indication that you should be passing the three specific elements rather than the whole object. (In general, code that "sets up" for a call to a routine or "takes down" after a call to a routine is an indication that the routine is not well designed.)

If you find yourself frequently changing the parameter list to the routine, with the parameters coming from the same object each time, that's an indication that you should be passing the whole object rather than specific elements.

Use named parameters In some languages, you can explicitly associate formal parameters with actual parameters. This makes parameter usage more self-documenting and helps avoid errors from mismatching parameters. Here's an example in Visual Basic:

Visual Basic Example of Explicitly Identifying Parameters

Here's where the formal parameters are declared.

Here's where the actual parameters are mapped to the formal parameters.

```
Private Function Distance3d( _
   ByVal xDistance As Coordinate, _
   ByVal yDistance As Coordinate, _
   ByVal zDistance As Coordinate _
)
   ...
End Function
...
Private Function Velocity( _
   ByVal latitude as Coordinate, _
   ByVal longitude as Coordinate, _
   ByVal elevation as Coordinate _
)
   ...
   Distance = Distance3d( xDistance := latitude, yDistance := longitude, _
      zDistance := elevation )
   ...
End Function
```

This technique is especially useful when you have longer-than-average lists of identically typed arguments, which increases the chances that you can insert a parameter mismatch without the compiler detecting it. Explicitly associating parameters may be overkill in many environments, but in safety-critical or other high-reliability environments the extra assurance that parameters match up the way you expect can be worthwhile.

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, constants, or expressions 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. Strongly typed languages such as C++ and Java 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.

7.6 Special Considerations in the Use of Functions

Modern languages such as C++, Java, and Visual Basic support both functions and procedures. A function is a routine that returns a value; a procedure is a routine that does not. In C++, all routines are typically called "functions"; however, a function with a *void* return type is semantically a procedure. The distinction between functions and procedures is as much a semantic distinction as a syntactic one, and semantics should be your guide.

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 routine called *FormatOutput()* used with a *report* object in statements like this one:

```
if ( report.FormatOutput( formattedReport ) = Success ) then ...
```

In this example, *report.FormatOutput()* operates as a procedure in that it has an output parameter, *formattedReport*, 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, *report.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:

```
report.FormatOutput( formattedReport, outputStatus )
if ( outputStatus = 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:

```
outputStatus = report.FormatOutput( formattedReport )
if ( outputStatus = Success ) then ...
```

In short, use a function if the primary purpose of the routine is to return the value indicated by the function name. Otherwise, use a procedure.

Setting the Function's Return Value

Using a function creates the risk that the function will return an incorrect return value. This usually happens when the function has several possible paths and one of the paths doesn't set a return value. To reduce this risk, do the following:

Check all possible return paths When creating a function, mentally execute each path to be sure that the function returns a value under all possible circumstances. It's good practice to initialize the return value at the beginning of the function to a default value—this provides a safety net in the event that the correct return value is not set.

Don't return references or pointers to local data As soon as the routine ends and the local data goes out of scope, the reference or pointer to the local data will be invalid. If an object needs to return information about its internal data, it should save the information as class member data. It should then provide accessor functions that return the values of the member data items rather than references or pointers to local data.

7.7 Macro Routines and Inline Routines

Cross-Reference Even if your language doesn't have a macro preprocessor, you can build your own. For details, see Section 30.5, "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.

Fully parenthesize macro expressions 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 Cube( a ) a*a*a
```

If you pass this macro nonatomic values for *a*, it won't do the multiplication properly. If you use the expression *Cube(x+1)*, it expands to *x+1 * x + 1 * x + 1*, 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 Cube( a ) (a)*(a)*(a)
```

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

C++ Example of a Macro That Works

```
#define Cube( a ) ((a)*(a)*(a))
```

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 Nonworking Macro with Multiple Statements

```
#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:

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 ); \
}
```

The practice of using macros as substitutes for function calls is generally considered risky and hard to understand—bad programming practice—so use this technique only if your specific circumstances require it.

Name macros that expand to code like routines so that they can be replaced by routines if necessary The convention in C++ 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. Considering the other problems with side effects, this is yet another reason to avoid using side effects.

Limitations on the Use of Macro Routines

Modern languages like C++ provide numerous alternatives to the use of macros:

- *const* for declaring constant values
- *inline* for defining functions that will be compiled as inline code
- *template* for defining standard operations like *min*, *max*, and so on in a type-safe way
- *enum* for defining enumerated types
- *typedef* for defining simple type substitutions

As Bjarne Stroustrup, designer of C++ points out, "Almost every macro demonstrates a flaw in the programming language, in the program, or in the programmer.... When you use macros, you should expect inferior service from tools such as debuggers, cross-reference tools, and profilers" (Stroustrup 1997). Macros are useful for supporting conditional compilation—see Section 8.6, "Debugging Aids"—but careful programmers generally use a macro as an alternative to a routine only as a last resort.

Inline Routines

C++ supports an *inline* keyword. An inline routine allows the programmer to treat the code as a routine at code-writing time, but the compiler will generally convert each instance of the routine into inline code at compile time. The theory is that *inline* can help produce highly efficient code that avoids routine-call overhead.

Use inline routines sparingly Inline routines violate encapsulation because C++ requires the programmer to put the code for the implementation of the inline routine in the header file, which exposes it to every programmer who uses the header file.

Inline routines require a routine's full code to be generated every time the routine is invoked, which for an inline routine of any size will increase code size. That can create problems of its own.

The bottom line on inlining for performance reasons is the same as the bottom line on any other coding technique that's motivated by performance: profile the code and measure the improvement. If the anticipated performance gain doesn't justify the bother of profiling the code to verify the improvement, it doesn't justify the erosion in code quality either.

cc2e.com/0792

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 "The Pseudocode Programming Process" in Chapter 9, page 215.

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?
- ❑ Have you established naming conventions for common operations?
- ❑ Does the routine have strong, functional cohesion—doing one and only one thing and doing it well?
- ❑ Do the routines have loose coupling—are the routine's connections 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?

Parameter-Passing Issues

- ❑ Does the routine's parameter list, taken as a whole, present a consistent interface abstraction?
- ❑ Are the routine's parameters in a sensible order, including matching the order of parameters in similar routines?
- ❑ Are interface assumptions documented?
- ❑ Does the routine have seven or fewer parameters?
- ❑ Is each input parameter used?
- ❑ Is each output parameter used?
- ❑ Does the routine avoid using input parameters as working variables?
- ❑ If the routine is a function, does it return a valid value under all possible circumstances?

Key Points

- The most important reason for creating 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 readability, reliability, and modifiability are better reasons.
- Sometimes the operation that most benefits from being put into a routine of its own is a simple one.
- You can classify routines into various kinds of cohesion, but you can make most routines functionally cohesive, which is best.
- 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.
- Functions should be used only when the primary purpose of the function is to return the specific value described by the function's name.
- Careful programmers use macro routines with care and only as a last resort.

Chapter 11

The Power of Variable Names

cc2e.com/1184

Contents

Related Topics

- Routine names: Section 7.3
- Class names: Section 6.2
- General issues in using variables: Chapter 10
- Formatting data declarations: "Laying Out Data Declarations" in Section 31.5
- Documenting variables: "Commenting Data Declarations" in Section 32.5

As important as the topic of good names is to effective programming, I have never read a discussion that covered more than a handful of the dozens of considerations that go into creating good names. Many programming texts devote a few paragraphs to choosing abbreviations, spout a few platitudes, and expect you to fend for yourself. I intend to be guilty of the opposite: to inundate you with more information about good names than you will ever be able to use!

This chapter's guidelines apply primarily to naming variables—objects and primitive data. But they also apply to naming classes, packages, files, and other programming entities. For details on naming routines, see Section 7.3, "Good Routine Names."

11.1 Considerations in Choosing Good Names

You can't give a variable a name the way you give a dog a name—because it's cute or it has a good sound. Unlike the dog and its name, which are different entities, a variable and a variable's name are essentially the same thing. Consequently, the goodness or badness of a variable is largely determined by its name. Choose variable names with care.

Here's an example of code that uses bad variable names:

Java Example of Poor Variable Names

```
x = x - xx;
xxx = fido + SalesTax( fido );
x = x + LateFee( x1, x ) + xxx;
x = x + Interest( x1, x );
```

What's happening in this piece of code? What do *x1*, *xx*, and *xxx* mean? What does *fido* mean? Suppose someone told you that the code computed a total customer bill based on an outstanding balance and a new set of purchases. Which variable would you use to print the customer's bill for just the new set of purchases?

Here's a version of the same code that makes these questions easier to answer:

Java Example of Good Variable Names

```
balance = balance - lastPayment;
monthlyTotal = newPurchases + SalesTax( newPurchases );
balance = balance + LateFee( customerID, balance ) + monthlyTotal;
balance = balance + Interest( customerID, balance );
```

In view of the contrast between these two pieces of code, a good variable name is readable, memorable, and appropriate. You can use several general rules of thumb to achieve these goals.

The Most Important Naming Consideration

The most important consideration in naming a variable is that the name fully and accurately describe the entity the variable represents. An effective technique for coming up with a good name is to state in words what the variable represents. Often that statement itself is the best variable name. It's easy to read because it doesn't contain cryptic abbreviations, and it's unambiguous. Because it's a full description of the entity, it won't be confused with something else. And it's easy to remember because the name is similar to the concept.

For a variable that represents the number of people on the U.S. Olympic team, you would create the name *numberOfPeopleOnTheUsOlympicTeam*. A variable that represents the number of seats in a stadium would be *numberOfSeatsInTheStadium*. A variable that represents the maximum number of points scored by a country's team in any modern Olympics would be *maximumNumberOfPointsInModernOlympics*. A variable that contains the current interest rate is better named *rate* or *interestRate* than *r* or *x*. You get the idea.

Note two characteristics of these names. First, they're easy to decipher. In fact, they don't need to be deciphered at all because you can simply read them. But second, some of the names are long—too long to be practical. I'll get to the question of variable-name length shortly.

Table 11-1 shows several examples of variable names, good and bad:

Table 11-1 Examples of Good and Bad Variable Names

Purpose of Variable	Good Names, Good Descriptors	Bad Names, Poor Descriptors
Running total of checks written to date	*runningTotal*, *checkTotal*	*written*, *ct*, *checks*, *CHKTTL*, *x*, *x1*, *x2*
Velocity of a bullet train	*velocity*, *trainVelocity*, *velocityInMph*	*velt*, *v*, *tv*, *x*, *x1*, *x2*, *train*
Current date	*currentDate*, *todaysDate*	*cd*, *current*, *c*, *x*, *x1*, *x2*, *date*
Lines per page	*linesPerPage*	*lpp*, *lines*, *l*, *x*, *x1*, *x2*

The names *currentDate* and *todaysDate* are good names because they fully and accurately describe the idea of "current date." In fact, they use the obvious words. Programmers sometimes overlook using the ordinary words, which is often the easiest solution. Because they're too short and not at all descriptive, *cd* and *c* are poor names. *current* is poor because it doesn't tell you what is current. *date* is almost a good name, but it's a poor name in the final analysis because the date involved isn't just any date, but the current date; *date* by itself gives no such indication. *x*, *x1*, and *x2* are poor names because they're always poor names—*x* traditionally represents an unknown quantity; if you don't want your variables to be unknown quantities, think of better names.

KEY POINT

Names should be as specific as possible. Names like *x*, *temp*, and *i* that are general enough to be used for more than one purpose are not as informative as they could be and are usually bad names.

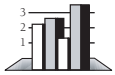
Problem Orientation

A good mnemonic name generally speaks to the problem rather than the solution. A good name tends to express the *what* more than the *how*. In general, if a name refers to some aspect of computing rather than to the problem, it's a *how* rather than a *what*. Avoid such a name in favor of a name that refers to the problem itself.

A record of employee data could be called *inputRec* or *employeeData*. *inputRec* is a computer term that refers to computing ideas—input and record. *employeeData* refers to the problem domain rather than the computing universe. Similarly, for a bit field indicating printer status, *bitFlag* is a more computerish name than *printerReady*. In an accounting application, *calcVal* is more computerish than *sum*.

Optimum Name Length

The optimum length for a name seems to be somewhere between the lengths of *x* and *maximumNumberOfPointsInModernOlympics*. Names that are too short don't convey enough meaning. The problem with names like *x1* and *x2* is that even if you can discover what *x* is, you won't know anything about the relationship between *x1* and *x2*. Names that are too long are hard to type and can obscure the visual structure of a program.



HARD DATA

Gorla, Benander, and Benander found that the effort required to debug a program was minimized when variables had names that averaged 10 to 16 characters (1990). Programs with names averaging 8 to 20 characters were almost as easy to debug. The guideline doesn't mean that you should try to make all of your variable names 9 to 15 or 10 to 16 characters long. It does mean that if you look over your code and see many names that are shorter, you should check to be sure that the names are as clear as they need to be.

You'll probably come out ahead by taking the Goldilocks-and-the-Three-Bears approach to naming variables, as Table 11-2 illustrates.

Table 11-2 Variable Names That Are Too Long, Too Short, or Just Right

Too long:	*numberOfPeopleOnTheUsOlympicTeam*
	numberOfSeatsInTheStadium
	maximumNumberOfPointsInModernOlympics
Too short:	*n, np, ntm*
	n, ns, nsisd
	m, mp, max, points
Just right:	*numTeamMembers, teamMemberCount*
	numSeatsInStadium, seatCount
	teamPointsMax, pointsRecord

The Effect of Scope on Variable Names

Cross-Reference Scope is discussed in more detail in Section 10.4, "Scope."

Are short variable names always bad? No, not always. When you give a variable a short name like *i*, the length itself says something about the variable—namely, that the variable is a scratch value with a limited scope of operation.

A programmer reading such a variable should be able to assume that its value isn't used outside a few lines of code. When you name a variable *i*, you're saying, "This variable is a run-of-the-mill loop counter or array index and doesn't have any significance outside these few lines of code."

A study by W. J. Hansen found that longer names are better for rarely used variables or global variables and shorter names are better for local variables or loop variables

(Shneiderman 1980). Short names are subject to many problems, however, and some careful programmers avoid them altogether as a matter of defensive-programming policy.

Use qualifiers on names that are in the global namespace If you have variables that are in the global namespace (named constants, class names, and so on), consider whether you need to adopt a convention for partitioning the global namespace and avoiding naming conflicts. In C++ and C#, you can use the *namespace* keyword to partition the global namespace.

C++ Example of Using the *namespace* Keyword to Partition the Global Namespace

```
namespace UserInterfaceSubsystem {
   ...
   // lots of declarations
   ...
}

namespace DatabaseSubsystem {
   ...
   // lots of declarations
   ...
}
```

If you declare an *Employee* class in both the *UserInterfaceSubsystem* and the *DatabaseSubsystem*, you can identify which you wanted to refer to by writing *UserInterfaceSubsystem::Employee* or *DatabaseSubsystem::Employee*. In Java, you can accomplish the same thing by using packages.

In languages that don't support namespaces or packages, you can still use naming conventions to partition the global namespace. One convention is to require that globally visible classes be prefixed with subsystem mnemonic. The user interface employee class might become *uiEmployee*, and the database employee class might become *dbEmployee*. This minimizes the risk of global-namespace collisions.

Computed-Value Qualifiers in Variable Names

Many programs have variables that contain computed values: totals, averages, maximums, and so on. If you modify a name with a qualifier like *Total*, *Sum*, *Average*, *Max*, *Min*, *Record*, *String*, or *Pointer*, put the modifier at the end of the name.

This practice offers several advantages. First, the most significant part of the variable name, the part that gives the variable most of its meaning, is at the front, so it's most prominent and gets read first. Second, by establishing this convention, you avoid the confusion you might create if you were to use both *totalRevenue* and *revenueTotal* in the same program. The names are semantically equivalent, and the convention would prevent their being used as if they were different. Third, a set of names like *revenueTotal*, *expenseTotal*, *revenueAverage*, and *expenseAverage* has a pleasing symmetry. A set of names

like *totalRevenue*, *expenseTotal*, *revenueAverage*, and *averageExpense* doesn't appeal to a sense of order. Finally, the consistency improves readability and eases maintenance.

An exception to the rule that computed values go at the end of the name is the customary position of the *Num* qualifier. Placed at the beginning of a variable name, *Num* refers to a total: *numCustomers* is the total number of customers. Placed at the end of the variable name, *Num* refers to an index: *customerNum* is the number of the current customer. The *s* at the end of *numCustomers* is another tip-off about the difference in meaning. But, because using *Num* so often creates confusion, it's probably best to sidestep the whole issue by using *Count* or *Total* to refer to a total number of customers and *Index* to refer to a specific customer. Thus, *customerCount* is the total number of customers and *customerIndex* refers to a specific customer.

Common Opposites in Variable Names

Cross-Reference For a similar list of opposites in routine names, see "Use opposites precisely" in Section 7.3.

Use opposites precisely. Using naming conventions for opposites helps consistency, which helps readability. Pairs like *begin/end* are easy to understand and remember. Pairs that depart from common-language opposites tend to be hard to remember and are therefore confusing. Here are some common opposites:

- begin/end
- first/last
- locked/unlocked
- min/max
- next/previous
- old/new
- opened/closed
- visible/invisible
- source/target
- source/destination
- up/down

11.2 Naming Specific Types of Data

In addition to the general considerations in naming data, special considerations come up in the naming of specific kinds of data. This section describes considerations specifically for loop variables, status variables, temporary variables, boolean variables, enumerated types, and named constants.

Naming Loop Indexes

Cross-Reference For details on loops, see Chapter 16, "Controlling Loops."

Guidelines for naming variables in loops have arisen because loops are such a common feature of computer programming. The names *i*, *j*, and *k* are customary:

Java Example of a Simple Loop Variable Name

```
for ( i = firstItem; i < lastItem; i++ ) {
   data[ i ] = 0;
}
```

If a variable is to be used outside the loop, it should be given a name more meaningful than *i*, *j*, or *k*. For example, if you are reading records from a file and need to remember how many records you've read, a name like *recordCount* would be appropriate:

Java Example of a Good Descriptive Loop Variable Name

```
recordCount = 0;
while ( moreScores() ) {
   score[ recordCount ] = GetNextScore();
   recordCount++;
}

// lines using recordCount
...
```

If the loop is longer than a few lines, it's easy to forget what *i* is supposed to stand for and you're better off giving the loop index a more meaningful name. Because code is so often changed, expanded, and copied into other programs, many experienced programmers avoid names like *i* altogether.

One common reason loops grow longer is that they're nested. If you have several nested loops, assign longer names to the loop variables to improve readability.

Java Example of Good Loop Names in a Nested Loop

```
for ( teamIndex = 0; teamIndex < teamCount; teamIndex++ ) {
   for ( eventIndex = 0; eventIndex < eventCount[ teamIndex ]; eventIndex++ ) {
      score[ teamIndex ][ eventIndex ] = 0;
   }
}
```

Carefully chosen names for loop-index variables avoid the common problem of index cross-talk: saying *i* when you mean *j* and *j* when you mean *i*. They also make array accesses clearer: *score[teamIndex][eventIndex]* is more informative than *score[i][j]*.

If you have to use *i*, *j*, and *k*, don't use them for anything other than loop indexes for simple loops—the convention is too well established, and breaking it to use them in other ways is confusing. The simplest way to avoid such problems is simply to think of more descriptive names than *i*, *j*, and *k*.

Naming Status Variables

Status variables describe the state of your program. Here's a naming guideline:

Think of a better name than* flag *for status variables It's better to think of flags as status variables. A flag should never have *flag* in its name because that doesn't give you any clue about what the flag does. For clarity, flags should be assigned values and their values should be tested with enumerated types, named constants, or global variables that act as named constants. Here are some examples of flags with bad names:

C++ Examples of Cryptic Flags

```
if ( flag ) ...
if ( statusFlag & 0x0F ) ...
if ( printFlag == 16 ) ...
if ( computeFlag == 0 ) ...

flag = 0x1;
statusFlag = 0x80;
printFlag = 16;
computeFlag = 0;
```

Statements like *statusFlag = 0x80* give you no clue about what the code does unless you wrote the code or have documentation that tells you both what *statusFlag* is and what *0x80* represents. Here are equivalent code examples that are clearer:

C++ Examples of Better Use of Status Variables

```
if ( dataReady ) ...
if ( characterType & PRINTABLE_CHAR ) ...
if ( reportType == ReportType_Annual ) ...
if ( recalcNeeded = false ) ...

dataReady = true;
characterType = CONTROL_CHARACTER;
reportType = ReportType_Annual;
recalcNeeded = false;
```

Clearly, *characterType = CONTROL_CHARACTER* is more meaningful than *statusFlag = 0x80*. Likewise, the conditional *if (reportType == ReportType_Annual)* is clearer than *if (printFlag == 16)*. The second example shows that you can use this approach with enumerated types as well as predefined named constants. Here's how you could use named constants and enumerated types to set up the values used in the example:

Declaring Status Variables in C++

```
// values for CharacterType
const int LETTER = 0x01;
const int DIGIT = 0x02;
const int PUNCTUATION = 0x04;
const int LINE_DRAW = 0x08;
```

```
const int PRINTABLE_CHAR = ( LETTER | DIGIT | PUNCTUATION | LINE_DRAW );

const int CONTROL_CHARACTER = 0x80;

// values for ReportType
enum ReportType {
   ReportType_Daily,
   ReportType_Monthly,
   ReportType_Quarterly,
   ReportType_Annual,
   ReportType_All
};
```

When you find yourself "figuring out" a section of code, consider renaming the variables. It's OK to figure out murder mysteries, but you shouldn't need to figure out code. You should be able to read it.

Naming Temporary Variables

Temporary variables are used to hold intermediate results of calculations, as temporary placeholders, and to hold housekeeping values. They're usually called *temp*, *x*, or some other vague and nondescriptive name. In general, temporary variables are a sign that the programmer does not yet fully understand the problem. Moreover, because the variables are officially given a "temporary" status, programmers tend to treat them more casually than other variables, increasing the chance of errors.

Be leery of "temporary" variables It's often necessary to preserve values temporarily. But in one way or another, most of the variables in your program are temporary. Calling a few of them temporary may indicate that you aren't sure of their real purposes. Consider the following example:

C++ Example of an Uninformative "Temporary" Variable Name

```
// Compute solutions of a quadratic equation.
// This assumes that (b^2-4*a*c) is positive.
temp = sqrt( b^2 - 4*a*c );
solution[0] = ( -b + temp ) / ( 2 * a );
solution[1] = ( -b - temp ) / ( 2 * a );
```

It's fine to store the value of the expression *sqrt(b^2 - 4 * a * c)* in a variable, especially since it's used in two places later. But the name *temp* doesn't tell you anything about what the variable does. A better approach is shown in this example:

C++ Example with a "Temporary" Variable Name Replaced with a Real Variable

```
// Compute solutions of a quadratic equation.
// This assumes that (b^2-4*a*c) is positive.
discriminant = sqrt( b^2 - 4*a*c );
solution[0] = ( -b + discriminant ) / ( 2 * a );
solution[1] = ( -b - discriminant ) / ( 2 * a );
```

This is essentially the same code, but it's improved with the use of an accurate, descriptive variable name.

Naming Boolean Variables

Following are a few guidelines to use in naming boolean variables:

Keep typical boolean names in mind Here are some particularly useful boolean variable names:

- ***done*** Use *done* to indicate whether something is done. The variable can indicate whether a loop is done or some other operation is done. Set *done* to *false* before something is done, and set it to *true* when something is completed.
- ***error*** Use *error* to indicate that an error has occurred. Set the variable to *false* when no error has occurred and to *true* when an error has occurred.
- ***found*** Use *found* to indicate whether a value has been found. Set *found* to *false* when the value has not been found and to *true* once the value has been found. Use *found* when searching an array for a value, a file for an employee ID, a list of paychecks for a certain paycheck amount, and so on.
- ***success* or *ok*** Use *success* or *ok* to indicate whether an operation has been successful. Set the variable to *false* when an operation has failed and to *true* when an operation has succeeded. If you can, replace *success* with a more specific name that describes precisely what it means to be successful. If the program is successful when processing is complete, you might use *processingComplete* instead. If the program is successful when a value is found, you might use *found* instead.

Give boolean variables names that imply true or false Names like *done* and *success* are good boolean names because the state is either *true* or *false*; something is done or it isn't; it's a success or it isn't. Names like *status* and *sourceFile*, on the other hand, are poor boolean names because they're not obviously *true* or *false*. What does it mean if *status* is *true*? Does it mean that something has a status? Everything has a status. Does *true* mean that the status of something is OK? Or does *false* mean that nothing has gone wrong? With a name like *status*, you can't tell.

For better results, replace *status* with a name like *error* or *statusOK*, and replace *sourceFile* with *sourceFileAvailable* or *sourceFileFound*, or whatever the variable represents.

Some programmers like to put *Is* in front of their boolean names. Then the variable name becomes a question: *isdone*? *isError*? *isFound*? *isProcessingComplete*? Answering the question with *true* or *false* provides the value of the variable. A benefit of this approach is that it won't work with vague names: *isStatus*? makes no sense at all. A drawback is that it makes simple logical expressions less readable: *if (isFound)* is slightly less readable than *if (found)*.

Use positive boolean variable names Negative names like *notFound*, *notdone*, and *notSuccessful* are difficult to read when they are negated—for example,

```
if not notFound
```

Such a name should be replaced by *found*, *done*, or *processingComplete* and then negated with an operator as appropriate. If what you're looking for is found, you have *found* instead of *not notFound*.

Naming Enumerated Types

Cross-Reference For details on using enumerated types, see Section 12.6, "Enumerated Types."

When you use an enumerated type, you can ensure that it's clear that members of the type all belong to the same group by using a group prefix, such as *Color_*, *Planet_*, or *Month_*. Here are some examples of identifying elements of enumerated types using prefixes:

Visual Basic Example of Using a Prefix Naming Convention for Enumerated Types

```
Public Enum Color
   Color_Red
   Color_Green
   Color_Blue
End Enum

Public Enum Planet
   Planet_Earth
   Planet_Mars
   Planet_Venus
End Enum

Public Enum Month
   Month_January
   Month_February
   ...
   Month_December
End Enum
```

In addition, the enum type itself (*Color*, *Planet*, or *Month*) can be identified in various ways, including all caps or prefixes (*e_Color*, *e_Planet*, or *e_Month*). A person could argue that an enum is essentially a user-defined type and so the name of the enum should be formatted the same as other user-defined types like classes. A different argument would be that enums are types, but they are also constants, so the enum type name should be formatted as constants. This book uses the convention of mixed case for enumerated type names.

In some languages, enumerated types are treated more like classes, and the members of the enumeration are always prefixed with the enum name, like *Color.Color_Red* or *Planet.Planet_Earth*. If you're working in that kind of language, it makes little sense to repeat the prefix, so you can treat the name of the enum type itself as the prefix and simplify the names to *Color.Red* and *Planet.Earth*.

Naming Constants

Cross-Reference For details on using named constants, see Section 12.7, "Named Constants."

When naming constants, name the abstract entity the constant represents rather than the number the constant refers to. *FIVE* is a bad name for a constant (regardless of whether the value it represents is *5.0*). *CYCLES_NEEDED* is a good name. *CYCLES_NEEDED* can equal *5.0* or *6.0*. *FIVE = 6.0* would be ridiculous. By the same token, *BAKERS_DOZEN* is a poor constant name; *DONUTS_MAX* is a good constant name.

11.3 The Power of Naming Conventions

Some programmers resist standards and conventions—and with good reason. Some standards and conventions are rigid and ineffective—destructive to creativity and program quality. This is unfortunate since effective standards are some of the most powerful tools at your disposal. This section discusses why, when, and how you should create your own standards for naming variables.

Why Have Conventions?

Conventions offer several specific benefits:

- They let you take more for granted. By making one global decision rather than many local ones, you can concentrate on the more important characteristics of the code.
- They help you transfer knowledge across projects. Similarities in names give you an easier and more confident understanding of what unfamiliar variables are supposed to do.
- They help you learn code more quickly on a new project. Rather than learning that Anita's code looks like this, Julia's like that, and Kristin's like something else, you can work with a more consistent set of code.
- They reduce name proliferation. Without naming conventions, you can easily call the same thing by two different names. For example, you might call total points both *pointTotal* and *totalPoints*. This might not be confusing to you when you write the code, but it can be enormously confusing to a new programmer who reads it later.
- They compensate for language weaknesses. You can use conventions to emulate named constants and enumerated types. The conventions can differentiate among local, class, and global data and can incorporate type information for types that aren't supported by the compiler.

- They emphasize relationships among related items. If you use object data, the compiler takes care of this automatically. If your language doesn't support objects, you can supplement it with a naming convention. Names like *address*, *phone*, and *name* don't indicate that the variables are related. But suppose you decide that all employee-data variables should begin with an *Employee* prefix. *employeeAddress*, *employeePhone*, and *employeeName* leave no doubt that the variables are related. Programming conventions can make up for the weakness of the language you're using.

The key is that any convention at all is often better than no convention. The convention may be arbitrary. The power of naming conventions doesn't come from the specific convention chosen but from the fact that a convention exists, adding structure to the code and giving you fewer things to worry about.

When You Should Have a Naming Convention

There are no hard-and-fast rules for when you should establish a naming convention, but here are a few cases in which conventions are worthwhile:

- When multiple programmers are working on a project
- When you plan to turn a program over to another programmer for modifications and maintenance (which is nearly always)
- When your programs are reviewed by other programmers in your organization
- When your program is so large that you can't hold the whole thing in your brain at once and must think about it in pieces
- When the program will be long-lived enough that you might put it aside for a few weeks or months before working on it again
- When you have a lot of unusual terms that are common on a project and want to have standard terms or abbreviations to use in coding

You always benefit from having some kind of naming convention. The considerations above should help you determine the extent of the convention to use on a particular project.

Degrees of Formality

Cross-Reference For details on the differences in formality in small and large projects, see Chapter 27, "How Program Size Affects Construction."

Different conventions have different degrees of formality. An informal convention might be as simple as "Use meaningful names." Other informal conventions are described in the next section. In general, the degree of formality you need is dependent on the number of people working on a program, the size of the program, and the program's expected life span. On tiny, throwaway projects, a strict convention might be unnecessary overhead. On larger projects in which several people are involved, either initially or over the program's life span, formal conventions are an indispensable aid to readability.

11.4 Informal Naming Conventions

Most projects use relatively informal naming conventions such as the ones laid out in this section.

Guidelines for a Language-Independent Convention

Here are some guidelines for creating a language-independent convention:

Differentiate between variable names and routine names The convention this book uses is to begin variable and object names with lower case and routine names with upper case: *variableName* vs. *RoutineName()*.

Differentiate between classes and objects The correspondence between class names and object names—or between types and variables of those types—can get tricky. Several standard options exist, as shown in the following examples:

Option 1: Differentiating Types and Variables via Initial Capitalization

```
Widget widget;
LongerWidget longerWidget;
```

Option 2: Differentiating Types and Variables via All Caps

```
WIDGET widget;
LONGERWIDGET longerWidget
```

Option 3: Differentiating Types and Variables via the "t_" Prefix for Types

```
t_Widget Widget;
t_LongerWidget LongerWidget;
```

Option 4: Differentiating Types and Variables via the "a" Prefix for Variables

```
Widget aWidget;
LongerWidget aLongerWidget;
```

Option 5: Differentiating Types and Variables via Using More Specific Names for the Variables

```
Widget employeeWidget;
LongerWidget fullEmployeeWidget;
```

Each of these options has strengths and weaknesses. Option 1 is a common convention in case-sensitive languages including C++ and Java, but some programmers are uncomfortable differentiating names solely on the basis of capitalization. Indeed, creating names that differ only in the capitalization of the first letter in the name seems to provide too little "psychological distance" and too small a visual distinction between the two names.

The Option 1 approach can't be applied consistently in mixed-language environments if any of the languages are case-insensitive. In Microsoft Visual Basic, for example, *Dim widget as Widget* will generate a syntax error because *widget* and *Widget* are treated as the same token.

Option 2 creates a more obvious distinction between the type name and the variable name. For historical reasons, all caps are used to indicate constants in C++ and Java, however, and the approach is subject to the same problems in mixed-language environments that Option 1 is subject to.

Option 3 works adequately in all languages, but some programmers dislike the idea of prefixes for aesthetic reasons.

Option 4 is sometimes used as an alternative to Option 3, but it has the drawback of altering the name of every instance of a class instead of just the one class name.

Option 5 requires more thought on a variable-by-variable basis. In most instances, being forced to think of a specific name for a variable results in more readable code. But sometimes a *widget* truly is just a generic *widget*, and in those instances you'll find yourself coming up with less-than-obvious names, like *genericWidget*, which are arguably less readable.

In short, each of the available options involves tradeoffs. The code in this book uses Option 5 because it's the most understandable in situations in which the person reading the code isn't necessarily familiar with a less intuitive naming convention.

Identify global variables One common programming problem is misuse of global variables. If you give all global variable names a *g_* prefix, for example, a programmer seeing the variable *g_RunningTotal* will know it's a global variable and treat it as such.

Identify member variables Identify a class's member data. Make it clear that the variable isn't a local variable and that it isn't a global variable either. For example, you can identify class member variables with an *m_* prefix to indicate that it is member data.

Identify type definitions Naming conventions for types serve two purposes: they explicitly identify a name as a type name, and they avoid naming clashes with variables. To meet those considerations, a prefix or suffix is a good approach. In C++, the customary approach is to use all uppercase letters for a type name—for example, *COLOR* and *MENU*. (This convention applies to *typedef*s and *struct*s, not class names.) But this creates the possibility of confusion with named preprocessor constants. To avoid confusion, you can prefix the type names with *t_*, such as *t_Color* and *t_Menu*.

Identify named constants Named constants need to be identified so that you can tell whether you're assigning a variable a value from another variable (whose value might change) or from a named constant. In Visual Basic, you have the additional possibility that the value might be from a function. Visual Basic doesn't require function names to use parentheses, whereas in C++ even a function with no parameters uses parentheses.

One approach to naming constants is to use a prefix like *c_* for constant names. That would give you names like *c_RecsMax* or *c_LinesPerPageMax*. In C++ and Java, the convention is to use all uppercase letters, possibly with underscores to separate words, *RECSMAX* or *RECS_ MAX* and *LINESPERPAGEMAX* or *LINES_PER_PAGE_ MAX*.

Identify elements of enumerated types Elements of enumerated types need to be identified for the same reasons that named constants do—to make it easy to tell that the name is for an enumerated type as opposed to a variable, named constant, or function. The standard approach applies: you can use all caps or an *e_* or *E_* prefix for the name of the type itself and use a prefix based on the specific type like *Color_* or *Planet_* for the members of the type.

Identify input-only parameters in languages that don't enforce them Sometimes input parameters are accidentally modified. In languages such as C++ and Visual Basic, you must indicate explicitly whether you want a value that's been modified to be returned to the calling routine. This is indicated with the *, *&*, and *const* qualifiers in C++ or *ByRef* and *ByVal* in Visual Basic.

In other languages, if you modify an input variable, it is returned whether you like it or not. This is especially true when passing objects. In Java, for example, all objects are passed "by value," so when you pass an object to a routine, the contents of the object can be changed within the called routine (Arnold, Gosling, Holmes 2000).

Cross-Reference Augmenting a language with a naming convention to make up for limitations in the language itself is an example of programming *into* a language instead of just programming in it. For more details on programming *into* a language, see Section 34.4, "Program into Your Language, Not in It."

In those languages, if you establish a naming convention in which input-only parameters are given a *const* prefix (or *final*, *nonmodifiable*, or something comparable) , you'll know that an error has occurred when you see anything with a *const* prefix on the left side of an equal sign. If you see *constMax.SetNewMax(...)*, you'll know it's a goof because the *const* prefix indicates that the variable isn't supposed to be modified.

Format names to enhance readability Two common techniques for increasing readability are using capitalization and spacing characters to separate words. For example, *GYMNASTICSPOINTTOTAL* is less readable than *gymnasticsPointTotal* or *gymnastics_point_total*. C++, Java, Visual Basic, and other languages allow for mixed uppercase and lowercase characters. C++, Java, Visual Basic, and other languages also allow the use of the underscore (_) separator.

Try not to mix these techniques; that makes code hard to read. If you make an honest attempt to use any of these readability techniques consistently, however, it will improve your code. People have managed to have zealous, blistering debates over fine points such as whether the first character in a name should be capitalized (*TotalPoints* vs. *totalPoints*), but as long as you and your team are consistent, it won't make much difference. This book uses initial lowercase because of the strength of the Java practice and to facilitate similarity in style across several languages.

Guidelines for Language-Specific Conventions

Follow the naming conventions of the language you're using. You can find books for most languages that describe style guidelines. Guidelines for C, C++, Java, and Visual Basic are provided in the following sections.

C Conventions

Further Reading The classic book on C programming style is *C Programming Guidelines* (Plum 1984).

Several naming conventions apply specifically to the C programming language:

- *c* and *ch* are character variables.
- *i* and *j* are integer indexes.
- *n* is a number of something.
- *p* is a pointer.
- *s* is a string.
- Preprocessor macros are in *ALL_CAPS*. This is usually extended to include typedefs as well.
- Variable and routine names are in *all_lowercase*.
- The underscore (_) character is used as a separator: *letters_in_lowercase* is more readable than *lettersinlowercase*.

These are the conventions for generic, UNIX-style and Linux-style C programming, but C conventions are different in different environments. In Microsoft Windows, C programmers tend to use a form of the Hungarian naming convention and mixed uppercase and lowercase letters for variable names. On the Macintosh, C programmers tend to use mixed-case names for routines because the Macintosh toolbox and operating-system routines were originally designed for a Pascal interface.

C++ Conventions

Further Reading For more on C++ programming style, see *The Elements of C++ Style* (Misfeldt, Bumgardner, and Gray 2004).

Here are the conventions that have grown up around C++ programming:

- *i* and *j* are integer indexes.
- *p* is a pointer.
- Constants, typedefs, and preprocessor macros are in *ALL_CAPS*.
- Class and other type names are in *MixedUpperAndLowerCase()*.
- Variable and function names use lowercase for the first word, with the first letter of each following word capitalized—for example, *variableOrRoutineName*.
- The underscore is not used as a separator within names, except for names in all caps and certain kinds of prefixes (such as those used to identify global variables).

As with C programming, this convention is far from standard and different environments have standardized on different convention details.

Java Conventions

Further Reading For more on Java programming style, see *The Elements of Java Style*, 2d ed. (Vermeulen et al. 2000).

In contrast with C and C++, Java style conventions have been well established since the language's beginning:

- *i* and *j* are integer indexes.
- Constants are in *ALL_CAPS* separated by underscores.
- Class and interface names capitalize the first letter of each word, including the first word—for example, *ClassOrInterfaceName*.
- Variable and method names use lowercase for the first word, with the first letter of each following word capitalized—for example, *variableOrRoutineName*.
- The underscore is not used as a separator within names except for names in all caps.
- The *get* and *set* prefixes are used for accessor methods.

Visual Basic Conventions

Visual Basic has not really established firm conventions. The next section recommends a convention for Visual Basic.

Mixed-Language Programming Considerations

When programming in a mixed-language environment, the naming conventions (as well as formatting conventions, documentation conventions, and other conventions) can be optimized for overall consistency and readability—even if that means going against convention for one of the languages that's part of the mix.

In this book, for example, variable names all begin with lowercase, which is consistent with conventional Java programming practice and some but not all C++ conventions. This book formats all routine names with an initial capital letter, which follows the C++ convention. The Java convention would be to begin method names with lowercase, but this book uses routine names that begin in uppercase across all languages for the sake of overall readability.

Sample Naming Conventions

The standard conventions above tend to ignore several important aspects of naming that were discussed over the past few pages—including variable scoping (private, class, or global), differentiating between class, object, routine, and variable names, and other issues.

The naming-convention guidelines can look complicated when they're strung across several pages. They don't need to be terribly complex, however, and you can adapt them to your needs. Variable names include three kinds of information:

- The contents of the variable (what it represents)
- The kind of data (named constant, primitive variable, user-defined type, or class)
- The scope of the variable (private, class, package, or global)

Tables 11-3, 11-4, and 11-5 provide naming conventions for C, C++, Java, and Visual Basic that have been adapted from the guidelines presented earlier. These specific conventions aren't necessarily recommended, but they give you an idea of what an informal naming convention includes.

Table 11-3 Sample Naming Conventions for C++ and Java

Entity	Description
ClassName	Class names are in mixed uppercase and lowercase with an initial capital letter.
TypeName	Type definitions, including enumerated types and typedefs, use mixed uppercase and lowercase with an initial capital letter.
EnumeratedTypes	In addition to the rule above, enumerated types are always stated in the plural form.
localVariable	Local variables are in mixed uppercase and lowercase with an initial lowercase letter. The name should be independent of the underlying data type and should refer to whatever the variable represents.
routineParameter	Routine parameters are formatted the same as local variables.
RoutineName()	Routines are in mixed uppercase and lowercase. (Good routine names are discussed in Section 7.3.)
m_ClassVariable	Member variables that are available to multiple routines within a class, but only within a class, are prefixed with an *m_*.
g_GlobalVariable	Global variables are prefixed with a *g_*.
CONSTANT	Named constants are in *ALL_CAPS*.
MACRO	Macros are in *ALL_CAPS*.
Base_EnumeratedType	Enumerated types are prefixed with a mnemonic for their base type stated in the singular—for example, *Color_Red*, *Color_Blue*.

Table 11-4 Sample Naming Conventions for C

Entity	Description
TypeName	Type definitions use mixed uppercase and lowercase with an initial capital letter.
GlobalRoutineName()	Public routines are in mixed uppercase and lowercase.
f_FileRoutineName()	Routines that are private to a single module (file) are prefixed with an *f_*.
LocalVariable	Local variables are in mixed uppercase and lowercase. The name should be independent of the underlying data type and should refer to whatever the variable represents.
RoutineParameter	Routine parameters are formatted the same as local variables.
f_FileStaticVariable	Module (file) variables are prefixed with an *f_*.
G_GLOBAL_GlobalVariable	Global variables are prefixed with a *G_* and a mnemonic of the module (file) that defines the variable in all uppercase—for example, *SCREEN_Dimensions*.
LOCAL_CONSTANT	Named constants that are private to a single routine or module (file) are in all uppercase—for example, *ROWS_MAX*.
G_GLOBALCONSTANT	Global named constants are in all uppercase and are prefixed with *G_* and a mnemonic of the module (file) that defines the named constant in all uppercase—for example, *G_SCREEN_ROWS_MAX*.
LOCALMACRO()	Macro definitions that are private to a single routine or module (file) are in all uppercase.
G_GLOBAL_MACRO()	Global macro definitions are in all uppercase and are prefixed with *G_* and a mnemonic of the module (file) that defines the macro in all uppercase—for example, *G_SCREEN_LOCATION()*.

Because Visual Basic is not case-sensitive, special rules apply for differentiating between type names and variable names. Take a look at Table 11-5.

Table 11-5 Sample Naming Conventions for Visual Basic

Entity	Description
C_ClassName	Class names are in mixed uppercase and lowercase with an initial capital letter and a *C_* prefix.
T_TypeName	Type definitions, including enumerated types and typedefs, use mixed uppercase and lowercase with an initial capital letter and a *T_* prefix.
T_EnumeratedTypes	In addition to the rule above, enumerated types are always stated in the plural form.

Table 11-5 Sample Naming Conventions for Visual Basic

Entity	Description
localVariable	Local variables are in mixed uppercase and lowercase with an initial lowercase letter. The name should be independent of the underlying data type and should refer to whatever the variable represents.
routineParameter	Routine parameters are formatted the same as local variables.
RoutineName()	Routines are in mixed uppercase and lowercase. (Good routine names are discussed in Section 7.3.)
m_ClassVariable	Member variables that are available to multiple routines within a class, but only within a class, are prefixed with an *m_*.
g_GlobalVariable	Global variables are prefixed with a *g_*.
CONSTANT	Named constants are in *ALL_CAPS*.
Base_EnumeratedType	Enumerated types are prefixed with a mnemonic for their base type stated in the singular—for example, *Color_Red*, *Color_Blue*.

11.5 Standardized Prefixes

Further Reading For further details on the Hungarian naming convention, see "The Hungarian Revolution" (Simonyi and Heller 1991).

Standardizing prefixes for common meanings provides a terse but consistent and readable approach to naming data. The best known scheme for standardizing prefixes is the Hungarian naming convention, which is a set of detailed guidelines for naming variables and routines (not Hungarians!) that was widely used at one time in Microsoft Windows programming. Although the Hungarian naming convention is no longer in widespread use, the basic idea of standardizing on terse, precise abbreviations continues to have value.

Standardized prefixes are composed of two parts: the user-defined type (UDT) abbreviation and the semantic prefix.

User-Defined Type Abbreviations

The UDT abbreviation identifies the data type of the object or variable being named. UDT abbreviations might refer to entities such as windows, screen regions, and fonts. A UDT abbreviation generally doesn't refer to any of the predefined data types offered by the programming language.

UDTs are described with short codes that you create for a specific program and then standardize on for use in that program. The codes are mnemonics such as *wn* for windows and *scr* for screen regions. Table 11-6 offers a sample list of UDTs that you might use in a program for a word processor.

Table 11-6 Sample of UDTs for a Word Processor

UDT Abbreviation	Meaning
ch	Character (a character not in the C++ sense, but in the sense of the data type a word-processing program would use to represent a character in a document)
doc	Document
pa	Paragraph
scr	Screen region
sel	Selection
wn	Window

When you use UDTs, you also define programming-language data types that use the same abbreviations as the UDTs. Thus, if you had the UDTs in Table 11-6, you'd see data declarations like these:

```
CH    chCursorPosition;
SCR   scrUserWorkspace;
DOC   docActive
PA    firstPaActiveDocument;
PA    lastPaActiveDocument;
WN    wnMain;
```

Again, these examples relate to a word processor. For use on your own projects, you'd create UDT abbreviations for the UDTs that are used most commonly within your environment.

Semantic Prefixes

Semantic prefixes go a step beyond the UDT and describe how the variable or object is used. Unlike UDTs, which vary from project to project, semantic prefixes are somewhat standard across projects. Table 11-7 shows a list of standard semantic prefixes.

Table 11-7 Semantic Prefixes

Semantic Prefix	Meaning
c	Count (as in the number of records, characters, and so on)
first	The first element that needs to be dealt with in an array. *first* is similar to *min* but relative to the current operation rather than to the array itself.
g	Global variable
i	Index into an array
last	The last element that needs to be dealt with in an array. *last* is the counterpart of *first*.

Table 11-7 Semantic Prefixes

Semantic Prefix	Meaning
lim	The upper limit of elements that need to be dealt with in an array. *lim* is not a valid index. Like *last*, *lim* is used as a counterpart of *first*. Unlike *last*, *lim* represents a noninclusive upper bound on the array; *last* represents a final, legal element. Generally, *lim* equals *last + 1*.
m	Class-level variable
max	The absolute last element in an array or other kind of list. *max* refers to the array itself rather than to operations on the array.
min	The absolute first element in an array or other kind of list.
p	Pointer

Semantic prefixes are formatted in lowercase or mixed uppercase and lowercase and are combined with the UDTs and with other semantic prefixes as needed. For example, the first paragraph in a document would be named *pa* to show that it's a paragraph and *first* to show that it's the first paragraph: *firstPa*. An index into the set of paragraphs would be named *iPa*; *cPa* is the count, or the number of paragraphs; and *firstPaActiveDocument* and *lastPaActiveDocument* are the first and last paragraphs in the current active document.

Advantages of Standardized Prefixes

KEY POINT

Standardized prefixes give you all the general advantages of having a naming convention as well as several other advantages. Because so many names are standard, you have fewer names to remember in any single program or class.

Standardized prefixes add precision to several areas of naming that tend to be imprecise. The precise distinctions between *min*, *first*, *last*, and *max* are particularly helpful.

Standardized prefixes make names more compact. For example, you can use *cpa* for the count of paragraphs rather than *totalParagraphs*. You can use *ipa* to identify an index into an array of paragraphs rather than *indexParagraphs* or *paragraphsIndex*.

Finally, standardized prefixes allow you to check types accurately when you're using abstract data types that your compiler can't necessarily check: *paReformat = docReformat* is probably wrong because *pa* and *doc* are different UDTs.

The main pitfall with standardized prefixes is a programmer neglecting to give the variable a meaningful name in addition to its prefix. If *ipa* unambiguously designates an index into an array of paragraphs, it's tempting not to make the name more meaningful like *ipaActiveDocument*. For readability, close the loop and come up with a descriptive name.

11.6 Creating Short Names That Are Readable

The desire to use short variable names is in some ways a remnant of an earlier age of computing. Older languages like assembler, generic Basic, and Fortran limited variable names to 2–8 characters and forced programmers to create short names. Early computing was more closely linked to mathematics and its use of terms like *i*, *j*, and *k* as the variables in summations and other equations. In modern languages like C++, Java, and Visual Basic, you can create names of virtually any length; you have almost no reason to shorten meaningful names.

If circumstances do require you to create short names, note that some methods of shortening names are better than others. You can create good short variable names by eliminating needless words, using short synonyms, and using any of several abbreviation strategies. It's a good idea to be familiar with multiple techniques for abbreviating because no single technique works well in all cases.

General Abbreviation Guidelines

Here are several guidelines for creating abbreviations. Some of them contradict others, so don't try to use them all at the same time.

- Use standard abbreviations (the ones in common use, which are listed in a dictionary).
- Remove all nonleading vowels. (*computer* becomes *cmptr*, and *screen* becomes *scrn*. *apple* becomes *appl*, and *integer* becomes *intgr*.)
- Remove articles: *and*, *or*, *the*, and so on.
- Use the first letter or first few letters of each word.
- Truncate consistently after the first, second, or third (whichever is appropriate) letter of each word.
- Keep the first and last letters of each word.
- Use every significant word in the name, up to a maximum of three words.
- Remove useless suffixes—*ing*, *ed*, and so on.
- Keep the most noticeable sound in each syllable.
- Be sure not to change the meaning of the variable.
- Iterate through these techniques until you abbreviate each variable name to between 8 to 20 characters or the number of characters to which your language limits variable names.

Phonetic Abbreviations

Some people advocate creating abbreviations based on the sound of the words rather than their spelling. Thus *skating* becomes *sk8ing*, *highlight* becomes *hilite*, *before* becomes *b4*, *execute* becomes *xqt*, and so on. This seems too much like asking people to figure out personalized license plates to me, and I don't recommend it. As an exercise, figure out what these names mean:

ILV2SK8 *XMEQWK* *S2DTM8O* *NXTC* *TRMN8R*

Comments on Abbreviations

You can fall into several traps when creating abbreviations. Here are some rules for avoiding pitfalls:

Don't abbreviate by removing one character from a word Typing one character is little extra work, and the one-character savings hardly justifies the loss in readability. It's like the calendars that have "Jun" and "Jul." You have to be in a big hurry to spell June as "Jun." With most one-letter deletions, it's hard to remember whether you removed the character. Either remove more than one character or spell out the word.

Abbreviate consistently Always use the same abbreviation. For example, use *Num* everywhere or *No* everywhere, but don't use both. Similarly, don't abbreviate a word in some names and not in others. For instance, don't use the full word *Number* in some places and the abbreviation *Num* in others.

Create names that you can pronounce Use *xPos* rather than *xPstn* and *needsComp* rather than *ndsCmptg*. Apply the telephone test—if you can't read your code to someone over the phone, rename your variables to be more distinctive (Kernighan and Plauger 1978).

Avoid combinations that result in misreading or mispronunciation To refer to the end of *B*, favor *ENDB* over *BEND*. If you use a good separation technique, you won't need this guideline since *B-END*, *BEnd*, or *b_end* won't be mispronounced.

Use a thesaurus to resolve naming collisions One problem in creating short names is naming collisions—names that abbreviate to the same thing. For example, if you're limited to three characters and you need to use *fired* and *full revenue disbursal* in the same area of a program, you might inadvertently abbreviate both to *frd*.

One easy way to avoid naming collisions is to use a different word with the same meaning, so a thesaurus is handy. In this example, *dismissed* might be substituted for *fired* and *complete revenue disbursal* might be substituted for *full revenue disbursal*. The three-letter abbreviations become *dsm* and *crd*, eliminating the naming collision.

Document extremely short names with translation tables in the code In languages that allow only very short names, include a translation table to provide a reminder of the mnemonic content of the variables. Include the table as comments at the beginning of a block of code. Here's an example:

Fortran Example of a Good Translation Table

```
C ****************************************************************
C    Translation Table
C
C    Variable    Meaning
C    --------    -------
C    XPOS        x-Coordinate Position (in meters)
C    YPOS        Y-Coordinate Position (in meters)
C    NDSCMP      Needs Computing (=0 if no computation is needed;
C                                 =1 if computation is needed)
C    PTGTTL      Point Grand Total
C    PTVLMX      Point Value Maximum
C    PSCRMX      Possible Score Maximum
C ****************************************************************
```

You might think that this technique is outdated, but as recently as mid-2003 I worked with a client that had hundreds of thousands of lines of code written in RPG that was subject to a 6-character–variable-name limitation. These issues still come up from time to time.

Document all abbreviations in a project-level "Standard Abbreviations" document Abbreviations in code create two general risks:

- A reader of the code might not understand the abbreviation.
- Other programmers might use multiple abbreviations to refer to the same word, which creates needless confusion.

To address both these potential problems, you can create a "Standard Abbreviations" document that captures all the coding abbreviations used on your project. The document can be a word processor document or a spreadsheet. On a very large project, it could be a database. The document is checked into version control and checked out anytime anyone creates a new abbreviation in the code. Entries in the document should be sorted by the full word, not the abbreviation.

This might seem like a lot of overhead, but aside from a small amount of startup overhead, it really just sets up a mechanism that helps the project use abbreviations effectively. It addresses the first of the two general risks described above by documenting all abbreviations in use. The fact that a programmer can't create a new abbreviation without the overhead of checking the Standard Abbreviations document out of ver-

sion control, entering the abbreviation, and checking it back in *is a good thing*. It means that an abbreviation won't be created unless it's so common that it's worth the hassle of documenting it.

This approach addresses the second risk by reducing the likelihood that a programmer will create a redundant abbreviation. A programmer who wants to abbreviate something will check out the abbreviations document and enter the new abbreviation. If there is already an abbreviation for the word the programmer wants to abbreviate, the programmer will notice that and will then use the existing abbreviation instead of creating a new one.


KEY POINT

The general issue illustrated by this guideline is the difference between write-time convenience and read-time convenience. This approach clearly creates a write-time *inconvenience*, but programmers over the lifetime of a system spend far more time reading code than writing code. This approach increases read-time convenience. By the time all the dust settles on a project, it might well also have improved write-time convenience.

Remember that names matter more to the reader of the code than to the writer Read code of your own that you haven't seen for at least six months and notice where you have to work to understand what the names mean. Resolve to change the practices that cause such confusion.

11.7 Kinds of Names to Avoid

Here are some guidelines regarding variable names to avoid:

Avoid misleading names or abbreviations Make sure that a name is unambiguous. For example, *FALSE* is usually the opposite of *TRUE* and would be a bad abbreviation for "Fig and Almond Season."

Avoid names with similar meanings If you can switch the names of two variables without hurting the program, you need to rename both variables. For example, *input* and *inputValue*, *recordNum* and *numRecords*, and *fileNumber* and *fileIndex* are so semantically similar that if you use them in the same piece of code you'll easily confuse them and install some subtle, hard-to-find errors.

Avoid variables with different meanings but similar names If you have two variables with similar names and different meanings, try to rename one of them or change your abbreviations. Avoid names like *clientRecs* and *clientReps*. They're only one letter different from each other, and the letter is hard to notice. Have at least two-letter differences between names, or put the differences at the beginning or at the end. *clientRecords* and *clientReports* are better than the original names.

Cross-Reference The technical term for differences like this between similar variable names is "psychological distance." For details, see "How 'Psychological Distance' Can Help" in Section 23.4.

***Avoid names that sound similar, such as* wrap *and* rap** Homonyms get in the way when you try to discuss your code with others. One of my pet peeves about Extreme Programming (Beck 2000) is its overly clever use of the terms Goal Donor and Gold Owner, which are virtually indistinguishable when spoken. You end up having conversations like this:

> *I was just speaking with the Goal Donor–*
>
> *Did you say "Gold Owner" or "Goal Donor"?*
>
> *I said "Goal Donor."*
>
> *What?*
>
> *GOAL - - - DONOR!*
>
> *OK, Goal Donor. You don't have to yell, Goll' Darn it.*
>
> *Did you say "Gold Donut?"*

Remember that the telephone test applies to similar sounding names just as it does to oddly abbreviated names.

Avoid numerals in names If the numerals in a name are really significant, use an array instead of separate variables. If an array is inappropriate, numerals are even more inappropriate. For example, avoid *file1* and *file2*, or *total1* and *total2*. You can almost always think of a better way to differentiate between two variables than by tacking a *1* or a *2* onto the end of the name. I can't say *never* use numerals. Some real-world entities (such as Route 66 or Interstate 405) have numerals embedded in them. But consider whether there are better alternatives before you create a name that includes numerals.

Avoid misspelled words in names It's hard enough to remember how words are supposed to be spelled. To require people to remember "correct" misspellings is simply too much to ask. For example, misspelling *highlight* as *hilite* to save three characters makes it devilishly difficult for a reader to remember how *highlight* was misspelled. Was it *highlite*? *hilite*? *hilight*? *hilit*? *jai-a-lai-t*? Who knows?

Avoid words that are commonly misspelled in English *Absense*, *acummulate*, *acsend*, *calender*, *concieve*, *defferred*, *definate*, *independance*, *occassionally*, *prefered*, *reciept*, *superseed*, and many others are common misspellings in English. Most English handbooks contain a list of commonly misspelled words. Avoid using such words in your variable names.

Don't differentiate variable names solely by capitalization If you're programming in a case-sensitive language such as C++, you may be tempted to use *frd* for *fired*, *FRD* for *final review duty*, and *Frd* for *full revenue disbursal*. Avoid this practice. Although the names are unique, the association of each with a particular meaning is arbitrary and confusing. *Frd* could just as easily be associated with *final review duty* and *FRD* with *full revenue disbursal*, and no logical rule will help you or anyone else to remember which is which.

Avoid multiple natural languages In multinational projects, enforce use of a single natural language for all code, including class names, variable names, and so on. Reading another programmer's code can be a challenge; reading another programmer's code in Southeast Martian is impossible.

A more subtle problem occurs in variations of English. If a project is conducted in multiple English-speaking countries, standardize on one version of English so that you're not constantly wondering whether the code should say "color" or "colour," "check" or "cheque," and so on.

Avoid the names of standard types, variables, and routines All programming-language guides contain lists of the language's reserved and predefined names. Read the list occasionally to make sure you're not stepping on the toes of the language you're using. For example, the following code fragment is legal in PL/I, but you would be a certifiable idiot to use it:


```
if if = then then
   then = else;
else else = if;
```

Don't use names that are totally unrelated to what the variables represent Sprinkling names such as *margaret* and *pookie* throughout your program virtually guarantees that no one else will be able to understand it. Avoid your boyfriend's name, wife's name, favorite beer's name, or other clever (aka silly) names for variables, unless the program is really about your boyfriend, wife, or favorite beer. Even then, you would be wise to recognize that each of these might change, and that therefore the generic names *boyfriend*, *wife*, and *favoriteBeer* are superior!

Avoid names containing hard-to-read characters Be aware that some characters look so similar that it's hard to tell them apart. If the only difference between two names is one of these characters, you might have a hard time telling the names apart. For example, try to circle the name that doesn't belong in each of the following sets:

```
eyeChartl      eyeChartI      eyeChartl
TTLCONFUSION   TTLCONFUSION   TTLC0NFUSION
hard2Read      hardZRead      hard2Read
GRANDTOTAL     GRANDTOTAL     6RANDTOTAL
ttl5           ttlS           ttlS
```

Pairs that are hard to distinguish include (1 and l), (1 and I), (. and ,), (0 and O), (2 and Z), (; and :), (S and 5), and (G and 6).

Cross-Reference For considerations in using data, see the checklist on page 257 in Chapter 10, "General Issues in Using Variables."

Do details like these really matter? Indeed! Gerald Weinberg reports that in the 1970s, a comma was used in a Fortran *FORMAT* statement where a period should have been used. The result was that scientists miscalculated a spacecraft's trajectory and lost a space probe—to the tune of $1.6 billion (Weinberg 1983).

cc2e.com/1191

CHECKLIST: Naming Variables

General Naming Considerations

- ❑ Does the name fully and accurately describe what the variable represents?
- ❑ Does the name refer to the real-world problem rather than to the programming-language solution?
- ❑ Is the name long enough that you don't have to puzzle it out?
- ❑ Are computed-value qualifiers, if any, at the end of the name?
- ❑ Does the name use *Count* or *Index* instead of *Num*?

Naming Specific Kinds of Data

- ❑ Are loop index names meaningful (something other than *i*, *j*, or *k* if the loop is more than one or two lines long or is nested)?
- ❑ Have all "temporary" variables been renamed to something more meaningful?
- ❑ Are boolean variables named so that their meanings when they're *true* are clear?
- ❑ Do enumerated-type names include a prefix or suffix that indicates the category—for example, *Color_* for *Color_Red*, *Color_Green*, *Color_Blue*, and so on?
- ❑ Are named constants named for the abstract entities they represent rather than the numbers they refer to?

Naming Conventions

- ❑ Does the convention distinguish among local, class, and global data?
- ❑ Does the convention distinguish among type names, named constants, enumerated types, and variables?
- ❑ Does the convention identify input-only parameters to routines in languages that don't enforce them?
- ❑ Is the convention as compatible as possible with standard conventions for the language?
- ❑ Are names formatted for readability?

Short Names

- ❑ Does the code use long names (unless it's necessary to use short ones)?
- ❑ Does the code avoid abbreviations that save only one character?
- ❑ Are all words abbreviated consistently?

- ❑ Are the names pronounceable?
- ❑ Are names that could be misread or mispronounced avoided?
- ❑ Are short names documented in translation tables?

Common Naming Problems: Have You Avoided...

- ❑ ...names that are misleading?
- ❑ ...names with similar meanings?
- ❑ ...names that are different by only one or two characters?
- ❑ ...names that sound similar?
- ❑ ...names that use numerals?
- ❑ ...names intentionally misspelled to make them shorter?
- ❑ ...names that are commonly misspelled in English?
- ❑ ...names that conflict with standard library routine names or with predefined variable names?
- ❑ ...totally arbitrary names?
- ❑ ...hard-to-read characters?

Key Points

- Good variable names are a key element of program readability. Specific kinds of variables such as loop indexes and status variables require specific considerations.
- Names should be as specific as possible. Names that are vague enough or general enough to be used for more than one purpose are usually bad names.
- Naming conventions distinguish among local, class, and global data. They distinguish among type names, named constants, enumerated types, and variables.
- Regardless of the kind of project you're working on, you should adopt a variable naming convention. The kind of convention you adopt depends on the size of your program and the number of people working on it.
- Abbreviations are rarely needed with modern programming languages. If you do use abbreviations, keep track of abbreviations in a project dictionary or use the standardized prefixes approach.
- Code is read far more times than it is written. Be sure that the names you choose favor read-time convenience over write-time convenience.

Chapter 32

Self-Documenting Code

cc2e.com/3245

Contents

Related Topics

- Layout: Chapter 31
- The Pseudocode Programming Process: Chapter 9
- Working classes: Chapter 6
- High-quality routines: Chapter 7
- Programming as communication: Sections 33.5 and 34.3

Code as if whoever maintains your program is a violent psychopath who knows where you live.
—*Anonymous*

Most programmers enjoy writing documentation if the documentation standards aren't unreasonable. Like layout, good documentation is a sign of the professional pride a programmer puts into a program. Software documentation can take many forms, and, after describing the sweep of the documentation landscape, this chapter cultivates the specific patch of documentation known as "comments."

32.1 External Documentation

Cross-Reference For more on external documentation, see Section 32.6, "IEEE Standards."

Documentation on a software project consists of information both inside the source-code listings and outside them—usually in the form of separate documents or unit development folders. On large, formal projects, most of the documentation is outside the source code (Jones 1998). External construction documentation tends to be at a high level compared to the code, at a low level compared to the documentation from the problem definition, requirements, and architecture activities.

Further Reading For a detailed description, see "The Unit Development Folder (UDF): An Effective Management Tool for Software Development" (Ingrassia 1976) or "The Unit Development Folder (UDF): A Ten-Year Perspective" (Ingrassia 1987).

Unit development folders A unit-development folder (UDF), or software-development folder (SDF), is an informal document that contains notes used by a developer during construction. A "unit" is loosely defined, usually to mean a class, although it could also mean a package or a component. The main purpose of a UDF is to provide a trail of design decisions that aren't documented elsewhere. Many projects have standards that specify the minimum content of a UDF, such as copies of the relevant requirements, the parts of the top-level design the unit implements, a copy of the development standards, a current code listing, and design notes from the unit's developer. Sometimes the customer requires a software developer to deliver the project's UDFs; often they are for internal use only.

Detailed-design document The detailed-design document is the low-level design document. It describes the class-level or routine-level design decisions, the alternatives that were considered, and the reasons for selecting the approaches that were selected. Sometimes this information is contained in a formal document. In such cases, detailed design is usually considered to be separate from construction. Sometimes it consists mainly of developers' notes collected into a UDF. And sometimes—often—it exists only in the code itself.

32.2 Programming Style as Documentation

In contrast to external documentation, internal documentation is found within the program listing itself. It's the most detailed kind of documentation, at the source-statement level. Because it's most closely associated with the code, internal documentation is also the kind of documentation most likely to remain correct as the code is modified.

The main contributor to code-level documentation isn't comments, but good programming style. Style includes good program structure, use of straightforward and easily understandable approaches, good variable names, good routine names, use of named constants instead of literals, clear layout, and minimization of control-flow and data-structure complexity.

Here's a code fragment with poor style:

Java Example of Poor Documentation Resulting from Bad Programming Style

```
for ( i = 2; i <= num; i++ ) {
meetsCriteria[ i ] = true;
}
for ( i = 2; i <= num / 2; i++ ) {
j = i + i;
while ( j <= num ) {
```

```
meetsCriteria[ j ] = false;
j = j + i;
}
}
for ( i = 2; i <= num; i++ ) {
if ( meetsCriteria[ i ] ) {
System.out.println ( i + " meets criteria." );
}
}
```

What do you think this routine does? It's unnecessarily cryptic. It's poorly documented not because it lacks comments, but because it lacks good programming style. The variable names are uninformative, and the layout is crude. Here's the same code improved—just improving the programming style makes its meaning much clearer:

Java Example of Documentation Without Comments (with Good Style)

```
for ( primeCandidate = 2; primeCandidate <= num; primeCandidate++ ) {
   isPrime[ primeCandidate ] = true;
}

for ( int factor = 2; factor < ( num / 2 ); factor++ ) {
   int factorableNumber = factor + factor;
   while ( factorableNumber <= num ) {
      isPrime[ factorableNumber ] = false;
      factorableNumber = factorableNumber + factor;
   }
}

for ( primeCandidate = 2; primeCandidate <= num; primeCandidate++ ) {
   if ( isPrime[ primeCandidate ] ) {
      System.out.println( primeCandidate + " is prime." );
   }
}
```

Cross-Reference In this code, the variable *factorableNumber* is added solely for the sake of clarifying the operation. For details on adding variables to clarify operations, see "Making Complicated Expressions Simple" in Section 19.1.

Unlike the first piece of code, this one lets you know at first glance that it has something to do with prime numbers. A second glance reveals that it finds the prime numbers between *1* and *Num*. With the first code fragment, it takes more than two glances just to figure out where the loops end.

The difference between the two code fragments has nothing to do with comments—neither fragment has any. The second one is much more readable, however, and approaches the Holy Grail of legibility: self-documenting code. Such code relies on good programming style to carry the greater part of the documentation burden. In well-written code, comments are the icing on the readability cake.

cc2e.com/3252

CHECKLIST: Self-Documenting Code

Classes

- ❑ Does the class's interface present a consistent abstraction?
- ❑ Is the class well named, and does its name describe its central purpose?
- ❑ Does the class's interface make obvious how you should use the class?
- ❑ Is the class's interface abstract enough that you don't have to think about how its services are implemented? Can you treat the class as a black box?

Routines

- ❑ Does each routine's name describe exactly what the routine does?
- ❑ Does each routine perform one well-defined task?
- ❑ Have all parts of each routine that would benefit from being put into their own routines been put into their own routines?
- ❑ Is each routine's interface obvious and clear?

Data Names

- ❑ Are type names descriptive enough to help document data declarations?
- ❑ Are variables named well?
- ❑ Are variables used only for the purpose for which they're named?
- ❑ Are loop counters given more informative names than *i*, *j*, and *k*?
- ❑ Are well-named enumerated types used instead of makeshift flags or boolean variables?
- ❑ Are named constants used instead of magic numbers or magic strings?
- ❑ Do naming conventions distinguish among type names, enumerated types, named constants, local variables, class variables, and global variables?

Data Organization

- ❑ Are extra variables used for clarity when needed?
- ❑ Are references to variables close together?
- ❑ Are data types simple so that they minimize complexity?
- ❑ Is complicated data accessed through abstract access routines (abstract data types)?

Control

- ❑ Is the nominal path through the code clear?
- ❑ Are related statements grouped together?

- ❑ Have relatively independent groups of statements been packaged into their own routines?
- ❑ Does the normal case follow the *if* rather than the *else*?
- ❑ Are control structures simple so that they minimize complexity?
- ❑ Does each loop perform one and only one function, as a well-defined routine would?
- ❑ Is nesting minimized?
- ❑ Have boolean expressions been simplified by using additional boolean variables, boolean functions, and decision tables?

Layout

- ❑ Does the program's layout show its logical structure?

Design

- ❑ Is the code straightforward, and does it avoid cleverness?
- ❑ Are implementation details hidden as much as possible?
- ❑ Is the program written in terms of the problem domain as much as possible rather than in terms of computer-science or programming-language structures?

32.3 To Comment or Not to Comment

Comments are easier to write poorly than well, and commenting can be more damaging than helpful. The heated discussions over the virtues of commenting often sound like philosophical debates over moral virtues, which makes me think that if Socrates had been a computer programmer, he and his students might have had the following discussion.

The Commento

Characters:

THRASYMACHUS A green, theoretical purist who believes everything he reads

CALLICLES A battle-hardened veteran from the old school—a "real" programmer

GLAUCON A young, confident, hot-shot computer jock

ISMENE A senior programmer tired of big promises, just looking for a few practices that work

SOCRATES The wise old programmer

Setting:

END OF THE TEAM'S DAILY STANDUP MEETING

"Does anyone have any other issues before we get back to work?" Socrates asked.

"I want to suggest a commenting standard for our projects," Thrasymachus said. "Some of our programmers barely comment their code, and everyone knows that code without comments is unreadable."

"You must be fresher out of college than I thought," Callicles responded. "Comments are an academic panacea, but everyone who's done any real programming knows that comments make the code harder to read, not easier. English is less precise than Java or Visual Basic and makes for a lot of excess verbiage. Programming-language statements are short and to the point. If you can't make the code clear, how can you make the comments clear? Plus, comments get out of date as the code changes. If you believe an out-of-date comment, you're sunk."

"I agree with that," Glaucon joined in. "Heavily commented code is harder to read because it means more to read. I already have to read the code; why should I have to read a lot of comments, too?"

"Wait a minute," Ismene said, putting down her coffee mug to put in her two drachmas' worth. "I know that commenting can be abused, but good comments are worth their weight in gold. I've had to maintain code that had comments and code that didn't, and I'd rather maintain code with comments. I don't think we should have a standard that says use one comment for every *x* lines of code, but we should encourage everyone to comment."

"If comments are a waste of time, why does anyone use them, Callicles?" Socrates asked.

"Either because they're required to or because they read somewhere that they're useful. No one who's thought about it could ever decide they're useful."

"Ismene thinks they're useful. She's been here three years, maintaining your code without comments and other code with comments, and she prefers the code with comments. What do you make of that?"

"Comments are useless because they just repeat the code in a more verbose—"


KEY POINT

"Wait right there," Thrasymachus interrupted. "Good comments don't repeat the code or explain it. They clarify its intent. Comments should explain, at a higher level of abstraction than the code, what you're trying to do."

"Right," Ismene said. "I scan the comments to find the section that does what I need to change or fix. You're right that comments that repeat the code don't help at all

because the code says everything already. When I read comments, I want it to be like reading headings in a book or a table of contents. Comments help me find the right section, and then I start reading the code. It's a lot faster to read one sentence in English than it is to parse 20 lines of code in a programming language." Ismene poured herself another cup of coffee.

"I think that people who refuse to write comments (1) think their code is clearer than it could possibly be, (2) think that other programmers are far more interested in their code than they really are, (3) think other programmers are smarter than they really are, (4) are lazy, or (5) are afraid someone else might figure out how their code works.

"Code reviews would be a big help here, Socrates," Ismene continued. "If someone claims they don't need to write comments and are bombarded by questions during a review—when several peers start saying, 'What the heck are you trying to do in this piece of code?'—then they'll start putting in comments. If they don't do it on their own, at least their manager will have the ammo to make them do it.

"I'm not accusing you of being lazy or afraid that people will figure out your code, Callicles. I've worked on your code and you're one of the best programmers in the company. But have a heart, huh? Your code would be easier for me to work on if you used comments."

"But they're a waste of resources," Callicles countered. "A good programmer's code should be self-documenting; everything you need to know should be in the code."

"No way!" Thrasymachus was out of his chair. "Everything the compiler needs to know is in the code! You might as well argue that everything you need to know is in the binary executable file! If you were smart enough to read it! What is *meant* to happen is not in the code."

Thrasymachus realized he was standing up and sat down. "Socrates, this is ridiculous. Why do we have to argue about whether comments are valuable? Everything I've ever read says they're valuable and should be used liberally. We're wasting our time."

Clearly, at some level comments *have* to be useful. To believe otherwise would be to believe that the comprehensibility of a program is independent of how much information the reader might already have about it. —*B. A. Sheil*

"Cool down, Thrasymachus. Ask Callicles how long he's been programming."

"How long, Callicles?"

"Well, I started on the Acropolis IV about 15 years ago. I guess I've seen about a dozen major systems from the time they were born to the time we gave them a cup of hemlock. And I've worked on major parts of a dozen more. Two of those systems had over half a million lines of code, so I know what I'm talking about. Comments are pretty useless."

Socrates looked at the younger programmer. "As Callicles says, comments have a lot of legitimate problems, and you won't realize that without more experience. If they're not done right, they're worse than useless."

"Even when they're done right, they're useless," Callicles said. "Comments are less precise than a programming language. I'd rather not have them at all."

"Wait a minute," Socrates said. "Ismene agrees that comments are less precise. Her point is that comments give you a higher level of abstraction, and we all know that levels of abstraction are one of a programmer's most powerful tools."

"I don't agree with that," Glaucon replied. "Instead of focusing on commenting, you should focus on making code more readable. Refactoring eliminates most of my comments. Once I've refactored, my code might have 20 or 30 routine calls without needing any comments. A good programmer can read the intent from the code itself, and what good does it do to read about somebody's intent when you know the code has an error?" Glaucon was pleased with his contribution. Callicles nodded.

"It sounds like you guys have never had to modify someone else's code," Ismene said. Callicles suddenly seemed very interested in the pencil marks on the ceiling tiles. "Why don't you try reading your own code six months or a year after you write it? You can improve your code-reading ability and your commenting. You don't have to choose one or the other. If you're reading a novel, you might not want section headings. But if you're reading a technical book, you'd like to be able to find what you're looking for quickly. I shouldn't have to switch into ultra-concentration mode and read hundreds of lines of code just to find the two lines I want to change."

"All right, I can see that it would be handy to be able to scan code," Glaucon said. He'd seen some of Ismene's programs and had been impressed. "But what about Callicles' other point, that comments get out of date as the code changes? I've only been programming for a couple of years, but even I know that nobody updates their comments."

"Well, yes and no," Ismene said. "If you take the comment as sacred and the code as suspicious, you're in deep trouble. Actually, finding a disagreement between the comment and the code tends to mean both are wrong. The fact that some comments are bad doesn't mean that commenting is bad. I'm going to the lunchroom to get another pot of coffee." Ismene left the room.

"My main objection to comments," Callicles said, "is that they're a waste of resources."

"Can anyone think of ways to minimize the time it takes to write the comments?" Socrates asked.

"Design routines in pseudocode, and then convert the pseudocode to comments and fill in the code between them," Glaucon said.

"OK, that would work as long as the comments don't repeat the code," Callicles said.

"Writing a comment makes you think harder about what your code is doing," Ismene said, returning from the lunchroom. "If it's hard to comment, either it's bad code or you don't understand it well enough. Either way, you need to spend more time on the code, so the time you spent commenting wasn't wasted because it pointed you to required work."

"All right," Socrates said. "I can't think of any more questions, and I think Ismene got the best of you guys today. We'll encourage commenting, but we won't be naive about it. We'll have code reviews so that everyone will get a good sense of the kind of comments that actually help. If you have trouble understanding someone else's code, let them know how they can improve it."

32.4 Keys to Effective Comments

As long as there are ill-defined goals, bizarre bugs, and unrealistic schedules, there will be Real Programmers willing to jump in and Solve The Problem, saving the documentation for later. Long live Fortran!
—Ed Post

What does the following routine do?

Java Mystery Routine Number One

```
// write out the sums 1..n for all n from 1 to num
current = 1;
previous = 0;
sum = 1;
for ( int i = 0; i < num; i++ ) {
   System.out.println( "Sum = " + sum );
   sum = current + previous;
   previous = current;
   current = sum;
}
```

Your best guess?

This routine computes the first *num* Fibonacci numbers. Its coding style is a little better than the style of the routine at the beginning of the chapter, but the comment is wrong, and if you blindly trust the comment, you head down the primrose path in the wrong direction.

What about this one?

Java Mystery Routine Number Two

```
// set product to "base"
product = base;

// loop from 2 to "num"
for ( int i = 2; i <= num; i++ ) {
   // multiply "base" by "product"
   product = product * base;
}
System.out.println( "Product = " + product );
```

This routine raises an integer *base* to the integer power *num*. The comments in this routine are accurate, but they add nothing to the code. They are merely a more verbose version of the code itself.

Here's one last routine:

Java Mystery Routine Number Three

```
// compute the square root of Num using the Newton-Raphson approximation
r = num / 2;
while ( abs( r - (num/r) ) > TOLERANCE ) {
   r = 0.5 * ( r + (num/r) );
}
System.out.println( "r = " + r );
```

This routine computes the square root of *num*. The code isn't great, but the comment is accurate.

Which routine was easiest for you to figure out correctly? None of the routines is particularly well written—the variable names are especially poor. In a nutshell, however, these routines illustrate the strengths and weaknesses of internal comments. Routine One has an incorrect comment. Routine Two's commenting merely repeats the code and is therefore useless. Only Routine Three's commenting earns its rent. Poor comments are worse than no comments. Routines One and Two would be better with no comments than with the poor comments they have.

The following subsections describe keys to writing effective comments.

Kinds of Comments

Comments can be classified into six categories:

Repeat of the Code

A repetitious comment restates what the code does in different words. It merely gives the reader of the code more to read without providing additional information.

Explanation of the Code

Explanatory comments are typically used to explain complicated, tricky, or sensitive pieces of code. In such situations they are useful, but usually that's only because the code is confusing. If the code is so complicated that it needs to be explained, it's nearly always better to improve the code than it is to add comments. Make the code itself clearer, and then use summary or intent comments.

Marker in the Code

A marker comment is one that isn't intended to be left in the code. It's a note to the developer that the work isn't done yet. Some developers type in a marker that's syntactically incorrect (******, for example) so that the compiler flags it and reminds them that they have more work to do. Other developers put a specified set of characters in comments that don't interfere with compilation so that they can search for them.

Few feelings are worse than having a customer report a problem in the code, debugging the problem, and tracing it to a section of code where you find something like this:

```
return NULL; // ****** NOT DONE! FIX BEFORE RELEASE!!!
```

Releasing defective code to customers is bad enough; releasing code that you *knew* was defective is even worse.

I've found that standardizing the style of marker comments is helpful. If you don't standardize, some programmers will use *******, some will use *!!!!!!*, some will use *TBD*, and some will use various other conventions. Using a variety of notations makes mechanical searching for incomplete code error-prone or impossible. Standardizing on one specific marker style allows you to do a mechanical search for incomplete sections of code as one of the steps in a release checklist, which avoids the *FIX BEFORE RELEASE!!!* problem. Some editors support "to do" tags and allow you to navigate to them easily.

Summary of the Code

A comment that summarizes code does just that: it distills a few lines of code into one or two sentences. Such comments are more valuable than comments that merely repeat the code because a reader can scan them more quickly than the code. Summary comments are particularly useful when someone other than the code's original author tries to modify the code.

Description of the Code's Intent

A comment at the level of intent explains the purpose of a section of code. Intent comments operate more at the level of the problem than at the level of the solution. For example,

```
-- get current employee information
```

is an intent comment, whereas

```
-- update employeeRecord object
```

is a summary comment in terms of the solution. A six-month study conducted by IBM found that maintenance programmers "most often said that understanding the original programmer's intent was the most difficult problem" (Fjelstad and Hamlen 1979). The distinction between intent and summary comments isn't always clear, and it's usually not important. Examples of intent comments are given throughout this chapter.

Information That Cannot Possibly Be Expressed by the Code Itself

Some information can't be expressed in code but must still be in the source code. This category of comments includes copyright notices, confidentiality notices, version numbers, and other housekeeping details; notes about the code's design; references to related requirements or architecture documentation; pointers to online references; optimization notes; comments required by editing tools such as Javadoc and Doxygen; and so on.

The three kinds of comments that are acceptable for completed code are information that can't be expressed in code, intent comments, and summary comments.

Commenting Efficiently

Effective commenting isn't that time-consuming. Too many comments are as bad as too few, and you can achieve a middle ground economically.

Comments can take a lot of time to write for two common reasons. First, the commenting style might be time-consuming or tedious. If it is, find a new style. A commenting style that requires a lot of busy work is a maintenance headache. If the comments are hard to change, they won't be changed and they'll become inaccurate and misleading, which is worse than having no comments at all.

Second, commenting might be difficult because the words to describe what the program is doing don't come easily. That's usually a sign that you don't understand what the program does. The time you spend "commenting" is really time spent understanding the program better, which is time that needs to be spent regardless of whether you comment.

Following are guidelines for commenting efficiently:

Use styles that don't break down or discourage modification Any style that's too fancy is annoying to maintain. For example, pick out the part of the comment below that won't be maintained:

Java Example of a Commenting Style That's Hard to Maintain

```
//  Variable        Meaning
//  --------        -------
//  xPos .......... XCoordinate Position (in meters)
//  yPos .......... YCoordinate Position (in meters)
```

```
//  ndsCmptng..... Needs Computing (= 0 if no computation is needed,
//                                  = 1 if computation is needed)
//  ptGrdTtl...... Point Grand Total
//  ptValMax...... Point Value Maximum
//  psblScrMax..... Possible Score Maximum
```

If you said that the leader dots (.....) will be hard to maintain, you're right! They look nice, but the list is fine without them. They add busy work to the job of modifying comments, and you'd rather have accurate comments than nice-looking ones, if that's the choice—and it usually is.

Here's another example of a common style that's hard to maintain:

C++ Example of a Commenting Style That's Hard to Maintain

```
/*************************************************************************
 * class:  GigaTron (GIGATRON.CPP)                                       *
 *                                                                       *
 * author: Dwight K. Coder                                               *
 * date:   July 4, 2014                                                  *
 *                                                                       *
 * Routines to control the twenty-first century's code evaluation        *
 * tool. The entry point to these routines is the EvaluateCode()         *
 * routine at the bottom of this file.                                   *
*************************************************************************/
```

This is a nice-looking block comment. It's clear that the whole block belongs together, and the beginning and ending of the block are obvious. What isn't clear about this block is how easy it is to change. If you have to add the name of a file to the bottom of the comment, chances are pretty good that you'll have to fuss with the pretty column of asterisks at the right. If you need to change the paragraph comments, you'll have to fuss with asterisks on both the left and the right. In practice, this means that the block won't be maintained because it will be too much work. If you can press a key and get neat columns of asterisks, that's great. Use it. The problem isn't the asterisks but that they're hard to maintain. The following comment looks almost as good and is a cinch to maintain:

C++ Example of a Commenting Style That's Easy to Maintain

```
/*************************************************************************
   class:  GigaTron (GIGATRON.CPP)

   author: Dwight K. Coder
   date:   July 4, 2014

   Routines to control the twenty-first century's code evaluation
   tool. The entry point to these routines is the EvaluateCode()
   routine at the bottom of this file.
 *************************************************************************/
```

Here's a particularly difficult style to maintain:

Microsoft Visual Basic Example of a Commenting Style That's Hard to Maintain

```
'  set up Color enumerated type
'  +-------------------------+
   ...

'  set up Vegetable enumerated type
'  +---------------------------+
   ...
```

It's hard to know what value the plus sign at the beginning and end of each dashed line adds to the comment, but it's easy to guess that every time a comment changes, the underline has to be adjusted so that the ending plus sign is in precisely the right place. And what do you do when a comment spills over into two lines? How do you align the plus signs? Take words out of the comment so that it takes up only one line? Make both lines the same length? The problems with this approach multiply when you try to apply it consistently.

A common guideline for Java and C++ that arises from a similar motivation is to use // syntax for single-line comments and /* ... */ syntax for longer comments, as shown here:

Java Example of Using Different Comment Syntaxes for Different Purposes

```
// This is a short comment
...
/* This is a much longer comment. Four score and seven years ago our fathers
brought forth on this continent a new nation, conceived in liberty and dedicated to
the proposition that all men are created equal. Now we are engaged in a great civil
war, testing whether that nation or any nation so conceived and so dedicated can
long endure. We are met on a great battlefield of that war. We have come to
dedicate a portion of that field as a final resting-place for those who here gave
their lives that that nation might live. It is altogether fitting and proper that
we should do this.
*/
```

The first comment is easy to maintain as long as it's kept short. For longer comments, the task of creating long columns of double slashes, manually breaking lines of text between rows, and similar activities is not very rewarding, and so the /* ... */ syntax is more appropriate for multiline comments.

The point is that you should pay attention to how you spend your time. If you spend a lot of time entering and deleting dashes to make plus signs line up, you're not programming; you're wasting time. Find a more efficient style. In the case of the underlines with plus signs, you could choose to have just the comments without any underlining. If you need to use underlines for emphasis, find some way other than

underlines with plus signs to emphasize those comments. One way would be to have a standard underline that's always the same length regardless of the length of the comment. Such a line requires no maintenance, and you can use a text-editor macro to enter it in the first place.

Cross-Reference For details on the Pseudocode Programming Process, see Chapter 9, "The Pseudocode Programming Process."

Use the Pseudocode Programming Process to reduce commenting time If you outline the code in comments before you write it, you win in several ways. When you finish the code, the comments are done. You don't have to dedicate time to comments. You also gain all the design benefits of writing in high-level pseudocode before filling in the low-level programming-language code.

Integrate commenting into your development style The alternative to integrating commenting into your development style is leaving commenting until the end of the project, and that has too many disadvantages. It becomes a task in its own right, which makes it seem like more work than when it's done a little bit at a time. Commenting done later takes more time because you have to remember or figure out what the code is doing instead of just writing down what you're already thinking about. It's also less accurate because you tend to forget assumptions or subtleties in the design.

The common argument against commenting as you go along is "When you're concentrating on the code, you shouldn't break your concentration to write comments." The appropriate response is that, if you have to concentrate so hard on writing code that commenting interrupts your thinking, you need to design in pseudocode first and then convert the pseudocode to comments. Code that requires that much concentration is a warning sign.

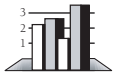
KEY POINT

If your design is hard to code, simplify the design before you worry about comments or code. If you use pseudocode to clarify your thoughts, coding is straightforward and the comments are automatic.

Performance is not a good reason to avoid commenting One recurring attribute of the rolling wave of technology discussed in Section 4.3, "Your Location on the Technology Wave," is interpreted environments in which commenting imposes a measurable performance penalty. In the 1980s, comments in Basic programs on the original IBM PC slowed programs. In the 1990s, *.asp* pages did the same thing. In the 2000s, JavaScript code and other code that needs to be sent across network connections presents a similar problem.

In each of these cases, the ultimate solution has not been to avoid commenting; it's been to create a release version of the code that's different from the development version. This is typically accomplished by running the code through a tool that strips out comments as part of the build process.

Optimum Number of Comments



Capers Jones points out that studies at IBM found that a commenting density of one comment roughly every 10 statements was the density at which clarity seemed to peak. Fewer comments made the code hard to understand. More comments also reduced code understandability (Jones 2000).

This kind of research can be abused, and projects sometimes adopt a standard such as "programs must have one comment at least every five lines." This standard addresses the symptom of programmers' not writing clear code, but it doesn't address the cause.

If you use the Pseudocode Programming Process effectively, you'll probably end up with a comment for every few lines of code. The number of comments, however, will be a side effect of the process itself. Rather than focusing on the number of comments, focus on whether each comment is efficient. If the comments describe why the code was written and meet the other criteria established in this chapter, you'll have enough comments.

32.5 Commenting Techniques

Commenting is amenable to several different techniques depending on the level to which the comments apply: program, file, routine, paragraph, or individual line.

Commenting Individual Lines

In good code, the need to comment individual lines of code is rare. Here are two possible reasons a line of code would need a comment:

- The single line is complicated enough to need an explanation.
- The single line once had an error, and you want a record of the error.

Here are some guidelines for commenting a line of code:

Avoid self-indulgent comments Many years ago, I heard the story of a maintenance programmer who was called out of bed to fix a malfunctioning program. The program's author had left the company and couldn't be reached. The maintenance programmer hadn't worked on the program before, and after examining the documentation carefully, he found only one comment. It looked like this:

```
MOV AX, 723h    ; R. I. P. L. V. B.
```

After working with the program through the night and puzzling over the comment, the programmer made a successful patch and went home to bed. Months later, he met the program's author at a conference and found out that the comment stood for "Rest in peace, Ludwig van Beethoven." Beethoven died in 1827 (decimal), which is 723 (hexadecimal). The fact that 723h was needed in that spot had nothing to do with the comment. Aaarrrrghhhhh!

Endline Comments and Their Problems

Endline comments are comments that appear at the ends of lines of code:

Visual Basic Example of Endline Comments

```
For employeeId = 1 To employeeCount
   GetBonus( employeeId, employeeType, bonusAmount )
   If employeeType = EmployeeType_Manager Then
      PayManagerBonus( employeeId, bonusAmount ) ' pay full amount
   Else
      If employeeType = EmployeeType_Programmer Then
         If bonusAmount >= MANAGER_APPROVAL_LEVEL Then
            PayProgrammerBonus( employeeId, StdAmt() ) ' pay std. amount
         Else
            PayProgrammerBonus( employeeId, bonusAmount ) ' pay full amount
         End If
      End If
   End If
Next
```

Although useful in some circumstances, endline comments pose several problems. The comments have to be aligned to the right of the code so that they don't interfere with the visual structure of the code. If you don't align them neatly, they'll make your listing look like it's been through the washing machine. Endline comments tend to be hard to format. If you use many of them, it takes time to align them. Such time is not spent learning more about the code; it's dedicated solely to the tedious task of pressing the spacebar or the Tab key.

Endline comments are also hard to maintain. If the code on any line containing an endline comment grows, it bumps the comment farther out and all the other endline comments will have to be bumped out to match. Styles that are hard to maintain aren't maintained, and the commenting deteriorates under modification rather than improving.

Endline comments also tend to be cryptic. The right side of the line usually doesn't offer much room, and the desire to keep the comment on one line means that the comment must be short. Work then goes into making the line as short as possible instead of as clear as possible.

Avoid endline comments on single lines In addition to their practical problems, endline comments pose several conceptual problems. Here's an example of a set of endline comments:

C++ Example of Useless Endline Comments

The comments merely repeat the code.

```
memoryToInitialize = MemoryAvailable();    // get amount of memory available
pointer = GetMemory( memoryToInitialize ); // get a ptr to the available memory
ZeroMemory( pointer, memoryToInitialize ); // set memory to 0
...
FreeMemory( pointer );                     // free memory allocated
```

A systemic problem with endline comments is that it's hard to write a meaningful comment for one line of code. Most endline comments just repeat the line of code, which hurts more than it helps.

Avoid endline comments for multiple lines of code If an endline comment is intended to apply to more than one line of code, the formatting doesn't show which lines the comment applies to:

Visual Basic Example of a Confusing Endline Comment on Multiple Lines of Code

```
For rateIdx = 1 to rateCount                ' Compute discounted rates
   LookupRegularRate( rateIdx, regularRate )
   rate( rateIdx ) = regularRate * discount( rateIdx )
Next
```

Even though the content of this particular comment is fine, its placement isn't. You have to read the comment and the code to know whether the comment applies to a specific statement or to the entire loop.

When to Use Endline Comments

Consider three exceptions to the recommendation against using endline comments:

Cross-Reference Other aspects of endline comments on data declarations are described in "Commenting Data Declarations," later in this section.

Use endline comments to annotate data declarations Endline comments are useful for annotating data declarations because they don't have the same systemic problems as endline comments on code, provided that you have enough width. With 132 columns, you can usually write a meaningful comment beside each data declaration:

Java Example of Good Endline Comments for Data Declarations

```
int boundary = 0;         // upper index of sorted part of array
String insertVal = BLANK; // data elmt to insert in sorted part of array
int insertPos = 0;        // position to insert elmt in sorted part of array
```

Avoid using endline comments for maintenance notes Endline comments are sometimes used for recording modifications to code after its initial development. This kind of comment typically consists of a date and the programmer's initials, or possibly an error-report number. Here's an example:

```
for i = 1 to maxElmts – 1   -- fixed error #A423 10/1/05 (scm)
```

Adding such a comment can be gratifying after a late-night debugging session on software that's in production, but such comments really have no place in production code. Such comments are handled better by version-control software. Comments should explain why the code works *now*, not why the code didn't work at some point in the past.

Cross-Reference Use of endline comments to mark ends of blocks is described further in "Commenting Control Structures," later in this section.

Use endline comments to mark ends of blocks An endline comment is useful for marking the end of a long block of code—the end of a *while* loop or an *if* statement, for example. This is described in more detail later in this chapter.

Aside from a couple of special cases, endline comments have conceptual problems and tend to be used for code that's too complicated. They are also difficult to format and maintain. Overall, they're best avoided.

Commenting Paragraphs of Code

Most comments in a well-documented program are one-sentence or two-sentence comments that describe paragraphs of code:

Java Example of a Good Comment for a Paragraph of Code

```
// swap the roots
oldRoot = root[0];
root[0] = root[1];
root[1] = oldRoot;
```

The comment doesn't repeat the code—it describes the code's intent. Such comments are relatively easy to maintain. Even if you find an error in the way the roots are swapped, for example, the comment won't need to be changed. Comments that aren't written at the level of intent are harder to maintain.

Write comments at the level of the code's intent Describe the purpose of the block of code that follows the comment. Here's an example of a comment that's ineffective because it doesn't operate at the level of intent:

Java Example of an Ineffective Comment

```
/* check each character in "inputString" until a dollar sign
is found or all characters have been checked
*/
done = false;
maxLen = inputString.length();
i = 0;
while ( !done && ( i < maxLen ) ) {
   if ( inputString[ i ] == '$' ) {
      done = true;
   }
   else {
      i++;
   }
}
```

Cross-Reference This code that performs a simple string search is used only for purposes of illustration. For real code, you'd use Java's built-in string library functions instead. For more on the importance of understanding your language's capabilities, see "Read!" in Section 33.3.

You can figure out that the loop looks for a $ by reading the code, and it's somewhat helpful to have that summarized in the comment. The problem with this comment is

that it merely repeats the code and doesn't give you any insight into what the code is supposed to be doing. This comment would be a little better:

```
// find '$' in inputString
```

This comment is better because it indicates that the goal of the loop is to find a $. But it still doesn't give you much insight into why the loop would need to find a $—in other words, into the deeper intent of the loop. Here's a comment that's better still:

```
// find the command-word terminator ($)
```

This comment actually contains information that the code listing does not, namely that the $ terminates a command word. In no way could you deduce that fact merely from reading the code fragment, so the comment is genuinely helpful.

Another way of thinking about commenting at the level of intent is to think about what you would name a routine that did the same thing as the code you want to comment. If you're writing paragraphs of code that have one purpose each, it isn't difficult. The comment in the previous code sample is a good example. *FindCommandWordTerminator()* would be a decent routine name. The other options, *Find$InInputString()* and *CheckEachCharacterInInputStrUntilADollarSignIsFoundOrAllCharactersHaveBeenChecked()*, are poor names (or invalid) for obvious reasons. Type the description without shortening or abbreviating it, as you might for a routine name. That description is your comment, and it's probably at the level of intent.


KEY POINT

Focus your documentation efforts on the code itself For the record, the code itself is always the first documentation you should check. In the previous example, the literal, $, should be replaced with a named constant and the variables should provide more of a clue about what's going on. If you want to push the edge of the readability envelope, add a variable to contain the result of the search. Doing that clearly distinguishes between the loop index and the result of the loop. Here's the code rewritten with good comments and good style:

Java Example of a Good Comment and Good Code

```
// find the command-word terminator
foundTheTerminator = false;
commandStringLength = inputString.length();
testCharPosition = 0;
while ( !foundTheTerminator && ( testCharPosition < commandStringLength ) ) {
   if ( inputString[ testCharPosition ] == COMMAND_WORD_TERMINATOR ) {
      foundTheTerminator = true;
      terminatorPosition = testCharPosition;
   }
   else {
      testCharPosition = testCharPosition + 1;
   }
}
```

Here's the variable that contains the result of the search.

If the code is good enough, it begins to read at close to the level of intent, encroaching on the comment's explanation of the code's intent. At that point, the comment and the code might become somewhat redundant, but that's a problem few programs have.

Cross-Reference For more on moving a section of code into its own routine, see "Extract routine/extract method" in Section 24.3.

Another good step for this code would be to create a routine called something like *FindCommandWordTerminator()* and move the code from the sample into that routine. A comment that describes that thought is useful but is more likely than a routine name to become inaccurate as the software evolves.

Focus paragraph comments on the why rather than the how Comments that explain how something is done usually operate at the programming-language level rather than the problem level. It's nearly impossible for a comment that focuses on how an operation is done to explain the intent of the operation, and comments that tell how are often redundant. What does the following comment tell you that the code doesn't?

Java Example of a Comment That Focuses on *How*

```
// if account flag is zero
if ( accountFlag == 0 ) ...
```

The comment tells you nothing more than the code itself does. What about this comment?

Java Example of a Comment That Focuses on *Why*

```
// if establishing a new account
if ( accountFlag == 0 ) ...
```

This comment is a lot better because it tells you something you couldn't infer from the code itself. The code itself could still be improved by use of a meaningful enumerated type name instead of *O* and a better variable name. Here's the best version of this comment and code:

Java Example of Using Good Style In Addition to a "Why" Comment

```
// if establishing a new account
if ( accountType == AccountType.NewAccount ) ...
```

When code attains this level of readability, it's appropriate to question the value of the comment. In this case, the comment has been made redundant by the improved code, and it should probably be removed. Alternatively, the purpose of the comment could be subtly shifted, like this:

Java Example of Using a "Section Heading" Comment

```
// establish a new account
if ( accountType == AccountType.NewAccount ) {
   ...
}
```

If this comment documents the whole block of code following the *if* test, it serves as a summary-level comment and it's appropriate to retain it as a section heading for the paragraph of code it references.

Use comments to prepare the reader for what is to follow Good comments tell the person reading the code what to expect. A reader should be able to scan only the comments and get a good idea of what the code does and where to look for a specific activity. A corollary to this rule is that a comment should always precede the code it describes. This idea isn't always taught in programming classes, but it's a well-established convention in commercial practice.

Make every comment count There's no virtue in excessive commenting—too many comments obscure the code they're meant to clarify. Rather than writing more comments, put the extra effort into making the code itself more readable.

Document surprises If you find anything that isn't obvious from the code itself, put it into a comment. If you have used a tricky technique instead of a straightforward one to improve performance, use comments to point out what the straightforward technique would be and quantify the performance gain achieved by using the tricky technique. Here's an example:

C++ Example of Documenting a Surprise

```
for ( element = 0; element < elementCount; element++ ) {
   // Use right shift to divide by two. Substituting the
   // right-shift operation cuts the loop time by 75%.
   elementList[ element ] = elementList[ element ] >> 1;
}
```

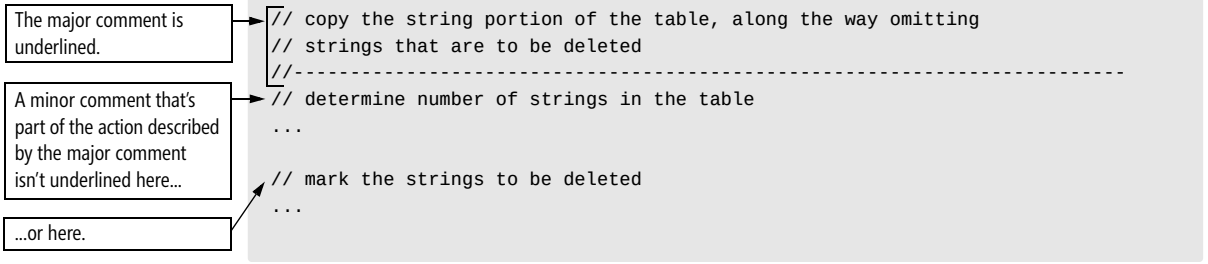
The selection of the right shift in this example is intentional. Among experienced programmers, it's common knowledge that for integers, right shift is functionally equivalent to divide-by-two.

If it's common knowledge, why document it? Because the purpose of the operation is not to perform a right shift; it is to perform a divide-by-two. The fact that the code doesn't use the technique most suited to its purpose is significant. Moreover, most compilers optimize integer division-by-two to be a right shift anyway, meaning that the reduced clarity is usually unnecessary. In this particular case, the compiler evidently doesn't optimize the divide-by-two, and the time saved will be significant. With the documentation, a programmer reading the code would see the motivation for using the nonobvious technique. Without the comment, the same programmer would be inclined to grumble that the code is unnecessarily "clever" without any meaningful gain in performance. Usually such grumbling is justified, so it's important to document the exceptions.

Avoid abbreviations Comments should be unambiguous, readable without the work of figuring out abbreviations. Avoid all but the most common abbreviations in comments. Unless you're using endline comments, using abbreviations isn't usually a temptation. If you are and it is, realize that abbreviations are another strike against a technique that struck out several pitches ago.

Differentiate between major and minor comments In a few cases, you might want to differentiate between different levels of comments, indicating that a detailed comment is part of a previous, broader comment. You can handle this in a couple of ways. You can try underlining the major comment and not underlining the minor comment:

C++ Example of Differentiating Between Major and Minor Comments with Underlines—Not Recommended



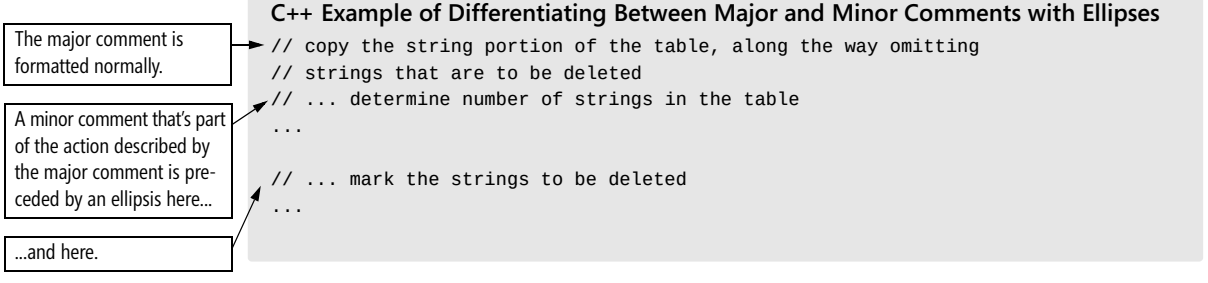

```
// copy the string portion of the table, along the way omitting
// strings that are to be deleted
//---------------------------------------------------------------------------
// determine number of strings in the table
...

// mark the strings to be deleted
...
```

The weakness of this approach is that it forces you to underline more comments than you'd really like to. If you underline a comment, it's assumed that all the nonunderlined comments that follow it are subordinate to it. Consequently, when you write the first comment that isn't subordinate to the underlined comment, it too must be underlined and the cycle starts all over. The result is too much underlining or inconsistent underlining in some places and no underlining in others.

This theme has several variations that all have the same problem. If you put the major comment in all caps and the minor comments in lowercase, you substitute the problem of too many all-caps comments for too many underlined comments. Some programmers use an initial cap on major statements and no initial cap on minor ones, but that's a subtle visual cue too easily overlooked.

A better approach is to use ellipses in front of the minor comments:

C++ Example of Differentiating Between Major and Minor Comments with Ellipses




```
// copy the string portion of the table, along the way omitting
// strings that are to be deleted
// ... determine number of strings in the table
...

// ... mark the strings to be deleted
...
```

Another approach that's often best is to put the major-comment operation into its own routine. Routines should be logically "flat," with all their activities on about the same logical level. If your code differentiates between major and minor activities within a routine, the routine isn't flat. Putting the complicated group of activities into its own routine makes for two logically flat routines instead of one logically lumpy one.

This discussion of major and minor comments doesn't apply to indented code within loops and conditionals. In such cases, you'll often have a broad comment at the top of the loop and more detailed comments about the operations within the indented code. In those cases, the indentation provides the clue to the logical organization of the comments. This discussion applies only to sequential paragraphs of code in which several paragraphs make up a complete operation and some paragraphs are subordinate to others.

Comment anything that gets around an error or an undocumented feature in a language or an environment If it's an error, it probably isn't documented. Even if it's documented somewhere, it doesn't hurt to document it again in your code. If it's an undocumented feature, by definition it isn't documented elsewhere and it should be documented in your code.

Suppose you find that the library routine *WriteData(data, numItems, blockSize)* works properly except when *blockSize* equals *500*. It works fine for *499*, *501*, and every other value you've ever tried, but you've found that the routine has a defect that appears only when *blockSize* equals *500*. In code that uses *WriteData()*, document why you're making a special case when *blockSize* is *500*. Here's an example of how it could look:

Java Example of Documenting the Workaround for an Error

```
blockSize = optimalBlockSize( numItems, sizePerItem );

/* The following code is necessary to work around an error in
WriteData() that appears only when the third parameter
equals 500. '500' has been replaced with a named constant
for clarity.
*/
if ( blockSize == WRITEDATA_BROKEN_SIZE ) {
   blockSize = WRITEDATA_WORKAROUND_SIZE;
}
WriteData ( file, data, blockSize );
```

Justify violations of good programming style If you've had to violate good programming style, explain why. That will prevent a well-intentioned programmer from changing the code to a better style, possibly breaking your code. The explanation will make it clear that you knew what you were doing and weren't just sloppy—give yourself credit where credit is due!

Don't comment tricky code; rewrite it Here's a comment from a project I worked on:

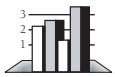
C++ Example of Commenting Clever Code

```
// VERY IMPORTANT NOTE:
// The constructor for this class takes a reference to a UiPublication.
// The UiPublication object MUST NOT BE DESTROYED before the DatabasePublication
// object. If it is, the DatabasePublication object will cause the program to
// die a horrible death.
```

This is a good example of one of the most prevalent and hazardous bits of programming folklore: that comments should be used to document especially "tricky" or "sensitive" sections of code. The reasoning is that people should know they need to be careful when they're working in certain areas.

This is a scary idea.

Commenting tricky code is exactly the wrong approach to take. Comments can't rescue difficult code. As Kernighan and Plauger emphasize, "Don't document bad code—rewrite it" (1978).



One study found that areas of source code with large numbers of comments also tended to have the most defects and to consume the most development effort (Lind and Vairavan 1989). The authors hypothesized that programmers tended to comment difficult code heavily.

When someone says, "This is really *tricky* code," I hear them say, "This is really *bad* code." If something seems tricky to you, it will be incomprehensible to someone else. Even something that doesn't seem all that tricky to you can seem impossibly convoluted to another person who hasn't seen the trick before. If you have to ask yourself "Is this tricky?" it is. You can always find a rewrite that's not tricky, so rewrite the code. Make your code so good that you don't need comments, and then comment it to make it even better.

This advice applies mainly to code you're writing for the first time. If you're maintaining a program and don't have the latitude to rewrite bad code, commenting the tricky parts is a good practice.

Commenting Data Declarations

Cross-Reference For details on formatting data, see "Laying Out Data Declarations" in Section 31.5. For details on how to use data effectively, see Chapters 10 through 13.

Comments for variable declarations describe aspects of the variable that the variable name can't describe. It's important to document data carefully; at least one company that studied its own practices has concluded that annotations on data are even more important than annotations on the processes in which the data is used (SDC, in Glass 1982). Here are some guidelines for commenting data:

Comment the units of numeric data If a number represents length, indicate whether the length is expressed in inches, feet, meters, or kilometers. If it's time, indicate whether it's expressed in elapsed seconds since 1-1-1980, milliseconds since the start of the program, and so on. If it's coordinates, indicate whether they represent latitude, longitude, and altitude and whether they're in radians or degrees; whether they represent an *X*, *Y*, *Z* coordinate system with its origin at the earth's center; and so on. Don't assume that the units are obvious. To a new programmer, they won't be. To someone who's been working on another part of the system, they won't be. After the program has been substantially modified, they won't be.

Alternatively, in many cases you should embed the units in the variable names rather than in comments. An expression like *distanceToSurface = marsLanderAltitude* looks like it's probably correct, but *distanceToSurfaceInMeters = marsLanderAltitudeInFeet* exposes an obvious error.

Cross-Reference A stronger technique for documenting allowable ranges of variables is to use assertions at the beginning and end of a routine to assert that the variable's values should be within a prescribed range. For more details, see Section 8.2, "Assertions."

Comment the range of allowable numeric values If a variable has an expected range of values, document the expected range. One of the powerful features of the Ada programming language was the ability to restrict the allowable values of a numeric variable to a range of values. If your language doesn't support that capability (and most languages don't), use a comment to document the expected range of values. For example, if a variable represents an amount of money in dollars, indicate that you expect it to be between $1 and $100. If a variable indicates a voltage, indicate that it should be between 105v and 125v.

Comment coded meanings If your language supports enumerated types—as C++ and Visual Basic do—use them to express coded meanings. If it doesn't, use comments to indicate what each value represents—and use a named constant rather than a literal for each of the values. If a variable represents kinds of electrical current, comment the fact that *1* represents alternating current, *2* represents direct current, and *3* represents *undefined*.

Here's an example of documenting variable declarations that illustrates the three preceding recommendations—all the range information is given in comments:

Visual Basic Example of Nicely Documented Variable Declarations

```
Dim cursorX As Integer  ' horizontal cursor position; ranges from 1..MaxCols
Dim cursorY As Integer  ' vertical cursor position; ranges from 1..MaxRows

Dim antennaLength As Long      ' length of antenna in meters; range is >= 2
Dim signalStrength As Integer  ' strength of signal in kilowatts; range is >= 1

Dim characterCode As Integer      ' ASCII character code; ranges from 0..255
Dim characterAttribute As Integer ' 0=Plain; 1=Italic; 2=Bold; 3=BoldItalic
Dim characterSize As Integer      ' size of character in points; ranges from 4..127
```

Comment limitations on input data Input data might come from an input parameter, a file, or direct user input. The previous guidelines apply as much to routine-input parameters as to other kinds of data. Make sure that expected and unexpected values are documented. Comments are one way of documenting that a routine is never supposed to receive certain data. Assertions are another way to document valid ranges, and if you use them the code becomes that much more self-checking.

Document flags to the bit level If a variable is used as a bit field, document the meaning of each bit:

Cross-Reference For details on naming flag variables, see "Naming Status Variables" in Section 11.2.

Visual Basic Example of Documenting Flags to the Bit Level

```
' The meanings of the bits in statusFlags are as follows, from most
' significant bit to least significant bit:
' MSB   0     error detected: 1=yes, 0=no
'       1-2   kind of error: 0=syntax, 1=warning, 2=severe, 3=fatal
'       3     reserved (should be 0)
'       4     printer status: 1=ready, 0=not ready
'       ...
'       14    not used (should be 0)
' LSB   15-32 not used (should be 0)
Dim statusFlags As Integer
```

If the example were written in C++, it would call for bit-field syntax so that the bit-field meanings would be self-documenting.

Stamp comments related to a variable with the variable's name If you have comments that refer to a specific variable, make sure the comment is updated whenever the variable is updated. One way to improve the odds of a consistent modification is to stamp the comment with the variable name. That way, string searches for the variable name will find the comment as well as the variable.

Cross-Reference For details on using global data, see Section 13.3, "Global Data."

Document global data If global data is used, annotate each piece well at the point at which it's declared. The annotation should indicate the purpose of the data and why it needs to be global. At each point at which the data is used, make it clear that the data is global. A naming convention is the first choice for highlighting a variable's global status. If a naming convention isn't used, comments can fill the gap.

Commenting Control Structures

Cross-Reference For other details on control structures, see Section 31.3, "Layout Styles," Section 31.4, "Laying Out Control Structures," and Chapters 14 through 19.

The space before a control structure is usually a natural place to put a comment. If it's an *if* or a *case* statement, you can provide the reason for the decision and a summary of the outcome. If it's a loop, you can indicate the purpose of the loop.

C++ Example of Commenting the Purpose of a Control Structure

Purpose of the following loop.

End of the loop (useful for longer, nested loops—although the need for such a comment indicates overly complicated code).

Purpose of the loop. Position of comment makes it clear that *inputString* is being set up for the loop.

```
// copy input field up to comma
while ( ( *inputString != ',' ) && ( *inputString != END_OF_STRING ) ) {
   *field = *inputString;
   field++;
   inputString++;
} // while -- copy input field

*field = END_OF_STRING;

if ( *inputString != END_OF_STRING ) {
   // read past comma and subsequent blanks to get to the next input field
   inputString++;
   while ( ( *inputString == ' ' ) && ( *inputString != END_OF_STRING ) ) {
      inputString++;
   }
} // if -- at end of string
```

This example suggests some guidelines:

Put a comment before each* if, case, *loop, or block of statements Such a place is a natural spot for a comment, and these constructs often need explanation. Use a comment to clarify the purpose of the control structure.

Comment the end of each control structure Use a comment to show what ended—for example,

```
} // for clientIndex — process record for each client
```

A comment is especially helpful at the end of long loops and to clarify loop nesting. Here's a Java example of using comments to clarify the ends of loop structures:

Java Example of Using Comments to Show Nesting

These comments indicate which control structure is ending.

```
for ( tableIndex = 0; tableIndex < tableCount; tableIndex++ ) {
   while ( recordIndex < recordCount ) {
      if ( !IllegalRecordNumber( recordIndex ) ) {
         ...
      } // if
   } // while
} // for
```

This commenting technique supplements the visual clues about the logical structure given by the code's indentation. You don't need to use the technique for short

loops that aren't nested. When the nesting is deep or the loops are long, however, the technique pays off.

Treat end-of-loop comments as a warning indicating complicated code If a loop is complicated enough to need an end-of-loop comment, treat the comment as a warning sign: the loop might need to be simplified. The same rule applies to complicated *if* tests and *case* statements.

End-of-loop comments provide useful clues to logical structure, but writing them initially and then maintaining them can become tedious. The best way to avoid such tedious work is often to rewrite any code that's complicated enough to require tedious documentation.

Commenting Routines

Cross-Reference For details on formatting routines, see Section 31.7. For details on how to create high-quality routines, see Chapter 7.

Routine-level comments are the subject of some of the worst advice in typical computer-science textbooks. Many textbooks urge you to pile up a stack of information at the top of every routine, regardless of its size or complexity:

Visual Basic Example of a Monolithic, Kitchen-Sink Routine Prolog

```
'*****************************************************************
' Name: CopyString
'
' Purpose:      This routine copies a string from the source
'               string (source) to the target string (target).
'
' Algorithm:    It gets the length of "source" and then copies each
'               character, one at a time, into "target". It uses
'               the loop index as an array index into both "source"
'               and "target" and increments the loop/array index
'               after each character is copied.
'
' Inputs:       input    The string to be copied
'
' Outputs:      output   The string to receive the copy of "input"
'
' Interface Assumptions: None
'
' Modification History: None
'
' Author:       Dwight K. Coder
' Date Created: 10/1/04
' Phone:        (555) 222-2255
' SSN:          111-22-3333
' Eye Color:    Green
' Maiden Name:  None
' Blood Type:   AB-
' Mother's Maiden Name: None
' Favorite Car: Pontiac Aztek
' Personalized License Plate: "Tek-ie"
'*****************************************************************
```

This is ridiculous. *CopyString* is presumably a trivial routine—probably fewer than five lines of code. The comment is totally out of proportion to the scale of the routine. The parts about the routine's *Purpose* and *Algorithm* are strained because it's hard to describe something as simple as *CopyString* at a level of detail that's between "copy a string" and the code itself. The boilerplate comments *Interface Assumptions* and *Modification History* aren't useful either—they just take up space in the listing. Requiring the author's name is redundant with information that can be retrieved more accurately from the revision-control system. To require all these ingredients for every routine is a recipe for inaccurate comments and maintenance failure. It's a lot of make-work that never pays off.

Another problem with heavy routine headers is that they discourage good factoring of the code—the overhead to create a new routine is so high that programmers will tend to err on the side of creating fewer routines, not more. Coding conventions should encourage good practices; heavy routine headers do the opposite.

Here are some guidelines for commenting routines:

Keep comments close to the code they describe One reason that the prolog to a routine shouldn't contain voluminous documentation is that such a practice puts the comments far away from the parts of the routine they describe. During maintenance, comments that are far from the code tend not to be maintained with the code. The comments and the code start to disagree, and suddenly the comments are worthless. Instead, follow the Principle of Proximity and put comments as close as possible to the code they describe. They're more likely to be maintained, and they'll continue to be worthwhile.

Several components of routine prologs are described below and should be included as needed. For your convenience, create a boilerplate documentation prolog. Just don't feel obliged to include all the information in every case. Fill out the parts that matter, and delete the rest.

Cross-Reference Good routine names are key to routine documentation. For details on how to create them, see Section 7.3, "Good Routine Names."

Describe each routine in one or two sentences at the top of the routine If you can't describe the routine in a short sentence or two, you probably need to think harder about what it's supposed to do. Difficulty in creating a short description is a sign that the design isn't as good as it should be. Go back to the design drawing board and try again. The short summary statement should be present in virtually all routines except for simple *Get* and *Set* accessor routines.

Document parameters where they are declared The easiest way to document input and output variables is to put comments next to the parameter declarations:

Java Example of Documenting Input and Output Data Where It's Declared—Good Practice

```
public void InsertionSort(
   int[] dataToSort, // elements to sort in locations firstElement..lastElement
   int firstElement, // index of first element to sort (>=0)
   int lastElement // index of last element to sort (<= MAX_ELEMENTS)
)
```

Cross-Reference Endline comments are discussed in more detail in "Endline Comments and Their Problems," earlier in this section.

This practice is a good exception to the rule of not using endline comments; they are exceptionally useful in documenting input and output parameters. This occasion for commenting is also a good illustration of the value of using standard indentation rather than endline indentation for routine parameter lists—you wouldn't have room for meaningful endline comments if you used endline indentation. The comments in the example are strained for space even with standard indentation. This example also demonstrates that comments aren't the only form of documentation. If your variable names are good enough, you might be able to skip commenting them. Finally, the need to document input and output variables is a good reason to avoid global data. Where do you document it? Presumably, you document the globals in the monster prolog. That makes for more work and, unfortunately, in practice usually means that the global data doesn't get documented. That's too bad because global data needs to be documented at least as much as anything else.

Take advantage of code documentation utilities such as Javadoc If the code in the previous example were actually written in Java, you would have the additional ability to set up the code to take advantage of Java's document extraction utility, Javadoc. In that case, "documenting parameters where they are declared" would change to look like this:

Java Example of Documenting Input and Output Data To Take Advantage of Javadoc

```
/**
 * ... <description of the routine> ...
 *
 * @param dataToSort  elements to sort in locations firstElement..lastElement
 * @param firstElement index of first element to sort (>=0)
 * @param lastElement  index of last element to sort (<= MAX_ELEMENTS)
 */
public void InsertionSort(
   int[] dataToSort,
   int firstElement,
   int lastElement
)
```

With a tool like Javadoc, the benefit of setting up the code to extract documentation outweighs the risks associated with separating the parameter description from the parameter's declaration. If you're not working in an environment that supports document extraction, like Javadoc, you're usually better off keeping the comments closer to the parameter names to avoid inconsistent edits and duplication of the names themselves.

Differentiate between input and output data It's useful to know which data is used as input and which is used as output. Visual Basic makes it relatively easy to tell because output data is preceded by the *ByRef* keyword and input data is preceded by the *ByVal* keyword. If your language doesn't support such differentiation automatically, put it into comments. Here's an example in C++:

C++ Example of Differentiating Between Input and Output Data

```
void StringCopy(
   char *target,       // out: string to copy to
   const char *source  // in: string to copy from
)
...
```

Cross-Reference The order of these parameters follows the standard order for C++ routines but conflicts with more general practices. For details, see "Put parameters in input-modify-output order" in Section 7.5. For details on using a naming convention to differentiate between input and output data, see Section 11.4.

C++-language routine declarations are a little tricky because some of the time the asterisk (*) indicates that the argument is an output argument and a lot of the time it just means that the variable is easier to handle as a pointer than as a nonpointer type. You're usually better off identifying input and output arguments explicitly.

If your routines are short enough and you maintain a clear distinction between input and output data, documenting the data's input or output status is probably unnecessary. If the routine is longer, however, it's a useful service to anyone who reads the routine.

Document interface assumptions Documenting interface assumptions might be viewed as a subset of the other commenting recommendations. If you have made any assumptions about the state of variables you receive—legal and illegal values, arrays being in sorted order, member data being initialized or containing only good data, and so on—document them either in the routine prolog or where the data is declared. This documentation should be present in virtually every routine.

Cross-Reference For details on other considerations for routine interfaces, see Section 7.5, "How to Use Routine Parameters." To document assumptions using assertions, see "Use assertions to document and verify preconditions and postconditions" in Section 8.2.

Make sure that global data that's used is documented. A global variable is as much an interface to a routine as anything else and is all the more hazardous because it sometimes doesn't seem like one.

As you're writing the routine and realize that you're making an interface assumption, write it down immediately.

Comment on the routine's limitations If the routine provides a numeric result, indicate the accuracy of the result. If the computations are undefined under some conditions, document the conditions. If the routine has a default behavior when it gets into trouble, document the behavior. If the routine is expected to work only on arrays or tables of a certain size, indicate that. If you know of modifications to the program that would break the routine, document them. If you ran into gotchas during the development of the routine, document those also.

Document the routine's global effects If the routine modifies global data, describe exactly what it does to the global data. As mentioned in Section 13.3, "Global Data," modifying global data is at least an order of magnitude more dangerous than merely reading it, so modifications should be performed carefully, part of the care being clear documentation. As usual, if documenting becomes too onerous, rewrite the code to reduce global data.

Document the source of algorithms that are used If you've used an algorithm from a book or magazine, document the volume and page number you took it from. If you developed the algorithm yourself, indicate where the reader can find the notes you've made about it.

Use comments to mark parts of your program Some programmers use comments to mark parts of their program so that they can find them easily. One such technique in C++ and Java is to mark the top of each routine with a comment beginning with these characters:

```
/**
```

This allows you to jump from routine to routine by doing a string search for /** or to use your editor to jump automatically if it supports that.

A similar technique is to mark different kinds of comments differently, depending on what they describe. For example, in C++ you could use *@keyword*, where *keyword* is a code you use to indicate the kind of comment. The comment *@param* could indicate that the comment describes a parameter to a routine, *@version* could indicate file-version information, *@throws* could document the exceptions thrown by a routine, and so on. This technique allows you to use tools to extract different kinds of information from your source files. For example, you could search for *@throws* to retrieve documentation about all the exceptions thrown by all the routines in a program.

cc2e.com/3259

This C++ convention is based on the Javadoc convention, which is a well-established interface documentation convention for Java programs (*java.sun.com/j2se/javadoc/*). You can define your own conventions in other languages.

Commenting Classes, Files, and Programs

Cross-Reference For layout details, see Section 31.8, "Laying Out Classes." For details on using classes, see Chapter 6, "Working Classes."

Classes, files, and programs are all characterized by the fact that they contain multiple routines. A file or class should contain a collection of related routines. A program contains all the routines in a program. The documentation task in each case is to provide a meaningful, top-level view of the contents of the file, class, or program.

General Guidelines for Class Documentation

For each class, use a block comment to describe general attributes of the class:

Describe the design approach to the class Overview comments that provide information that can't readily be reverse-engineered from coding details are especially useful. Describe the class's design philosophy, overall design approach, design alternatives that were considered and discarded, and so on.

Describe limitations, usage assumptions, and so on Similar to routines, be sure to describe any limitations imposed by the class's design. Also describe assumptions about input and output data, error-handling responsibilities, global effects, sources of algorithms, and so on.

Comment the class interface Can another programmer understand how to use a class without looking at the class's implementation? If not, class encapsulation is seriously at risk. The class's interface should contain all the information anyone needs to use the class. The Javadoc convention is to require, at a minimum, documentation for each parameter and each return value (Sun Microsystems 2000). This should be done for all exposed routines of each class (Bloch 2001).

Don't document implementation details in the class interface A cardinal rule of encapsulation is that you expose information only on a need-to-know basis: if there is any question about whether information needs to be exposed, the default is to keep it hidden. Consequently, class interface files should contain information needed to use the class but not information needed to implement or maintain the inner workings of the class.

General Guidelines for File Documentation

At the top of a file, use a block comment to describe the contents of the file:

Describe the purpose and contents of each file The file header comment should describe the classes or routines contained in a file. If all the routines for a program are in one file, the purpose of the file is pretty obvious—it's the file that contains the whole program. If the purpose of the file is to contain one specific class, the purpose is also obvious—it's the file that contains the class with a similar name.

If the file contains more than one class, explain why the classes need to be combined into a single file.

If the division into multiple source files is made for some reason other than modularity, a good description of the purpose of the file will be even more helpful to a programmer who is modifying the program. If someone is looking for a routine that

does *x*, does the file's header comment help that person determine whether this file contains such a routine?

Put your name, e-mail address, and phone number in the block comment Authorship and primary responsibility for specific areas of source code becomes important on large projects. Small projects (fewer than 10 people) can use collaborative development approaches, such as shared code ownership in which all team members are equally responsible for all sections of code. Larger systems require that programmers specialize in different areas of code, which makes full-team shared-code ownership impractical.

In that case, authorship is important information to have in a listing. It gives other programmers who work on the code a clue about the programming style, and it gives them someone to contact if they need help. Depending on whether you work on individual routines, classes, or programs, you should include author information at the routine, class, or program level.

Include a version-control tag Many version-control tools will insert version information into a file. In CVS, for example, the characters

```
// $Id$
```

will automatically expand to

```
// $Id: ClassName.java,v 1.1 2004/02/05 00:36:43 ismene Exp $
```

This allows you to maintain current versioning information within a file without requiring any developer effort other than inserting the original *Id* comment.

Include legal notices in the block comment Many companies like to include copyright statements, confidentiality notices, and other legal notices in their programs. If yours is one of them, include a line similar to the one below. Check with your company's legal advisor to determine what information, if any, to include in your files.

Java Example of a Copyright Statement

```
// (c) Copyright 1993-2004 Steven C. McConnell. All Rights Reserved.
...
```

Give the file a name related to its contents Normally, the name of the file should be closely related to the name of the public class contained in the file. For example, if the class is named *Employee*, the file should be named *Employee.cpp*. Some languages, notably Java, require the file name to match the class name.

The Book Paradigm for Program Documentation

Further Reading This discussion is adapted from "The Book Paradigm for Improved Maintenance" (Oman and Cook 1990a) and "Typographic Style Is More Than Cosmetic" (Oman and Cook 1990b). A similar analysis is presented in detail in *Human Factors and Typography for More Readable Programs* (Baecker and Marcus 1990).

Most experienced programmers agree that the documentation techniques described in the previous section are valuable. The hard, scientific evidence for the value of any one of the techniques is still weak. When the techniques are combined, however, evidence of their effectiveness is strong.

In 1990, Paul Oman and Curtis Cook published a pair of studies on the "Book Paradigm" for documentation (1990a, 1990b). They looked for a coding style that would support several different styles of code reading. One goal was to support top-down, bottom-up, and focused searches. Another was to break up the code into chunks that programmers could remember more easily than a long listing of homogeneous code. Oman and Cook wanted the style to provide for both high-level and low-level clues about code organization.

They found that by thinking of code as a special kind of book and by formatting it accordingly, they could achieve their goals. In the Book Paradigm, code and its documentation are organized into several components similar to the components of a book to help programmers get a high-level view of the program.

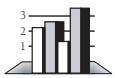
The "preface" is a group of introductory comments such as those usually found at the beginning of a file. It functions as the preface to a book does. It gives the programmer an overview of the program.

The "table of contents" shows the top-level files, classes, and routines (chapters). They might be shown in a list, as a traditional book's chapters are, or graphically in a structure chart.

The "sections" are the divisions within routines—routine declarations, data declarations, and executable statements, for example.

The "cross-references" are cross-reference maps of the code, including line numbers.

The low-level techniques that Oman and Cook use to take advantage of the similarities between a book and a code listing are similar to the techniques described in Chapter 31, "Layout and Style," and in this chapter.



HARD DATA

The upshot of using their techniques to organize code was that when Oman and Cook gave a maintenance task to a group of experienced, professional programmers, the average time to perform a maintenance task in a 1000-line program was only about three-quarters of the time it took the programmers to do the same task in a traditional source listing (1990b). Moreover, the maintenance scores of programmers on code documented with the Book Paradigm averaged about 20 percent higher than on traditionally documented code. Oman and Cook concluded that by paying attention to the typographic principles of book design, you can get a 10 to 20 percent improvement in

comprehension. A study with programmers at the University of Toronto produced similar results (Baecker and Marcus 1990).

The Book Paradigm emphasizes the importance of providing documentation that explains both the high-level and the low-level organization of your program.

32.6 IEEE Standards

For documentation beyond the source-code level, valuable sources of information are the IEEE (Institute for Electric and Electrical Engineers) Software Engineering Standards. IEEE standards are developed by groups composed of practitioners and academicians who are expert in a particular area. Each standard contains a summary of the area covered by the standard and typically contains the outline for the appropriate documentation for work in that area.

Several national and international organizations participate in standards work. The IEEE is a group that has taken the lead in defining software engineering standards. Some standards are jointly adopted by ISO (International Standards Organization), EIA (Electronic Industries Alliance), or IEC (International Engineering Consortium).

Standards names are composed of the standards number, the year the standard was adopted, and the name of the standard. So, *IEEE/EIA Std 12207-1997, Information Technology–Software Life Cycle Processes*, refers to standard number 12207.2, which was adopted in 1997 by the IEEE and EIA.

Here are some of the national and international standards most applicable to software projects:

cc2e.com/3266

The top-level standard is *ISO/IEC Std 12207, Information Technology–Software Life Cycle Processes*, which is the international standard that defines a life-cycle framework for developing and managing software projects. This standard was adopted in the United States as *IEEE/EIA Std 12207, Information Technology–Software Life Cycle Processes*.

Software-Development Standards

cc2e.com/3273

Here are software-development standards to consider:

IEEE Std 830-1998, Recommended Practice for Software Requirements Specifications

IEEE Std 1233-1998, Guide for Developing System Requirements Specifications

IEEE Std 1016-1998, Recommended Practice for Software Design Descriptions

IEEE Std 828-1998, Standard for Software Configuration Management Plans

IEEE Std 1063-2001, Standard for Software User Documentation

IEEE Std 1219-1998, Standard for Software Maintenance

Software Quality-Assurance Standards

cc2e.com/3280

And here are software quality-assurance standards:

IEEE Std 730-2002, Standard for Software Quality Assurance Plans

IEEE Std 1028-1997, Standard for Software Reviews

IEEE Std 1008-1987 (R1993), Standard for Software Unit Testing

IEEE Std 829-1998, Standard for Software Test Documentation

IEEE Std 1061-1998, Standard for a Software Quality Metrics Methodology

Management Standards

cc2e.com/3287

Here are some software-management standards:

IEEE Std 1058-1998, Standard for Software Project Management Plans

IEEE Std 1074-1997, Standard for Developing Software Life Cycle Processes

IEEE Std 1045-1992, Standard for Software Productivity Metrics

IEEE Std 1062-1998, Recommended Practice for Software Acquisition

IEEE Std 1540-2001, Standard for Software Life Cycle Processes - Risk Management

IEEE Std 1490-1998, Guide - Adoption of PMI Standard - A Guide to the Project Management Body of Knowledge

Overview of Standards

cc2e.com/3294

Here are sources that provide overviews of standards:

cc2e.com/3201

IEEE Software Engineering Standards Collection, 2003 Edition. New York, NY: Institute of Electrical and Electronics Engineers (IEEE). This comprehensive volume contains 40 of the most recent ANSI/IEEE standards for software development as of 2003. Each standard includes a document outline, a description of each component of the outline, and a rationale for that component. The document includes standards for quality-assurance plans, configuration-management plans, test documents, requirements specifications, verification and validation plans, design descriptions, project-management plans, and user documentation. The book is a distillation of the expertise of hundreds of people at the top of their fields and would be a bargain at virtually any price. Some of the standards are also available individually. All are available from the IEEE Computer Society in Los Alamitos, California and from *www.computer.org/cspress*.

Moore, James W. *Software Engineering Standards: A User's Road Map*. Los Alamitos, CA: IEEE Computer Society Press, 1997. Moore provides an overview of IEEE software engineering standards.

Additional Resources

cc2e.com/3208

In addition to the IEEE standards, numerous other resources are available on program documentation.

Spinellis, Diomidis. *Code Reading: The Open Source Perspective*. Boston, MA: Addison-Wesley, 2003. This book is a pragmatic exploration of techniques for reading code, including where to find code to read, tips for reading large code bases, tools that support code reading, and many other useful suggestions.

cc2e.com/3215

I wonder how many great novelists have never read someone else's work, how many great painters have never studied another's brush strokes, how many skilled surgeons never learned by looking over a colleague's shoulder.... And yet that's what we expect programmers to do.
—Dave Thomas

SourceForge.net. For decades, a perennial problem in teaching software development has been finding life-size examples of production code to share with students. Many people learn quickest from studying real-life examples, but most life-size code bases are treated as proprietary information by the companies that created them. This situation has improved dramatically through the combination of the Internet and open-source software. The Source Forge website contains code for thousands of programs in C, C++, Java, Visual Basic, PHP, Perl, Python, and many other languages, all which you can download for free. Programmers can benefit from wading through the code on the website to see much larger real-world examples than *Code Complete, Second Edition*, is able to show in its short code examples. Junior programmers who haven't previously seen extensive examples of production code will find this website especially valuable as a source of both good and bad coding practices.

cc2e.com/3222

Sun Microsystems. "How to Write Doc Comments for the Javadoc Tool," 2000. Available from *http://java.sun.com/j2se/javadoc/writingdoccomments/*. This article describes how to use Javadoc to document Java programs. It includes detailed advice about how to tag comments by using an *@tag* style notation. It also includes many specific details about how to wordsmith the comments themselves. The Javadoc conventions are probably the most fully developed code-level documentation standards currently available.

Here are sources of information on other topics in software documentation:

McConnell, Steve. *Software Project Survival Guide*. Redmond, WA: Microsoft Press, 1998. This book describes the documentation required by a medium-sized business-critical project. A related website provides numerous related document templates.

cc2e.com/3229

www.construx.com. This website (my company's website) contains numerous document templates, coding conventions, and other resources related to all aspects of software development, including software documentation.

cc2e.com/3236

Post, Ed. "Real Programmers Don't Use Pascal," *Datamation*, July 1983, pp. 263–265. This tongue-in-cheek paper argues for a return to the "good old days" of Fortran programming when programmers didn't have to worry about pesky issues like readability.

cc2e.com/3243

CHECKLIST: Good Commenting Technique

General

- ❑ Can someone pick up the code and immediately start to understand it?
- ❑ Do comments explain the code's intent or summarize what the code does, rather than just repeating the code?
- ❑ Is the Pseudocode Programming Process used to reduce commenting time?
- ❑ Has tricky code been rewritten rather than commented?
- ❑ Are comments up to date?
- ❑ Are comments clear and correct?
- ❑ Does the commenting style allow comments to be easily modified?

Statements and Paragraphs

- ❑ Does the code avoid endline comments?
- ❑ Do comments focus on *why* rather than *how*?
- ❑ Do comments prepare the reader for the code to follow?
- ❑ Does every comment count? Have redundant, extraneous, and self-indulgent comments been removed or improved?
- ❑ Are surprises documented?
- ❑ Have abbreviations been avoided?
- ❑ Is the distinction between major and minor comments clear?
- ❑ Is code that works around an error or undocumented feature commented?

Data Declarations

- ❑ Are units on data declarations commented?
- ❑ Are the ranges of values on numeric data commented?
- ❑ Are coded meanings commented?
- ❑ Are limitations on input data commented?
- ❑ Are flags documented to the bit level?
- ❑ Has each global variable been commented where it is declared?
- ❑ Has each global variable been identified as such at each usage, by a naming convention, a comment, or both?
- ❑ Are magic numbers replaced with named constants or variables rather than just documented?

Control Structures

- ❑ Is each control statement commented?
- ❑ Are the ends of long or complex control structures commented or, when possible, simplified so that they don't need comments?

Routines

- ❑ Is the purpose of each routine commented?
- ❑ Are other facts about each routine given in comments, when relevant, including input and output data, interface assumptions, limitations, error corrections, global effects, and sources of algorithms?

Files, Classes, and Programs

- ❑ Does the program have a short document, such as that described in the Book Paradigm, that gives an overall view of how the program is organized?
- ❑ Is the purpose of each file described?
- ❑ Are the author's name, e-mail address, and phone number in the listing?

Key Points

- The question of whether to comment is a legitimate one. Done poorly, commenting is a waste of time and sometimes harmful. Done well, commenting is worthwhile.
- The source code should contain most of the critical information about the program. As long as the program is running, the source code is more likely than any other resource to be kept current, and it's useful to have important information bundled with the code.
- Good code is its own best documentation. If the code is bad enough to require extensive comments, try first to improve the code so that it doesn't need extensive comments.
- Comments should say things about the code that the code can't say about itself—at the summary level or the intent level.
- Some commenting styles require a lot of tedious clerical work. Develop a style that's easy to maintain.