Microsoft

CODE COMPLETE

2

Second Edition

A practical handbook of software construction

Steve McConnell

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

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data
McConnell, Steve
Code Complete / Steve McConnell.--2nd ed.
p. cm.
Includes index.
ISBN 0-7356-1967-0
1. Computer Software--Development--Handbooks, manuals, etc. I. Title.

QA76.76.D47M39 2004
005.1--dc22 2004049981

Printed and bound in the United States of America.

ISBN: 978-0-7356-1967-8

Twenty-fourth Printing: February 2015

Distributed in Canada by H.B. Fenn and Company Ltd. A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to *mspinput@microsoft.com*.

Acquisitions Editors: Linda Engelman and Robin Van Steenburgh
Project Editor: Devon Musgrave
Indexer: Bill Myers
Principal Desktop Publisher: Carl Diltz

Body Part No. X10-53130

[2014-03-21]

General Guidelines for Minimizing Scope

Here are some specific guidelines you can use to minimize scope:

Cross-Reference For details on initializing variables close to where they're used, see Section 10.3, "Guidelines for Initializing Variables," earlier in this chapter.

Initialize variables used in a loop immediately before the loop rather than back at the beginning of the routine containing the loop Doing this improves the chance that when you modify the loop, you'll remember to make corresponding modifications to the loop initialization. Later, when you modify the program and put another loop around the initial loop, the initialization will work on each pass through the new loop rather than on only the first pass.

Cross-Reference For more on this style of variable declaration and definition, see "Ideally, declare and define each variable close to where it's first used" in Section 10.3.

Don't assign a value to a variable until just before the value is used You might have experienced the frustration of trying to figure out where a variable was assigned its value. The more you can do to clarify where a variable receives its value, the better. Languages like C++ and Java support variable initializations like these:

C++ Example of Good Variable Declarations and Initializations

```
int receiptIndex = 0;
float dailyReceipts = TodaysReceipts();
double totalReceipts = TotalReceipts( dailyReceipts );
```

Cross-Reference For more details on keeping related statements together, see Section 14.2, "Statements Whose Order Doesn't Matter."

Group related statements The following examples show a routine for summarizing daily receipts and illustrate how to put references to variables together so that they're easier to locate. The first example illustrates the violation of this principle:

C++ Example of Using Two Sets of Variables in a Confusing Way

Statements using two sets of variables.

```
void SummarizeData(...) {
    ...
    GetOldData( oldData, &numOldData );
    GetNewData( newData, &numNewData );
    totalOldData = Sum( oldData, numOldData );
    totalNewData = Sum( newData, numNewData );
    PrintOldDataSummary( oldData, totalOldData, numOldData );
    PrintNewDataSummary( newData, totalNewData, numNewData );
    SaveOldDataSummary( totalOldData, numOldData );
    SaveNewDataSummary( totalNewData, numNewData );
    ...
}
```

Note that, in this example, you have to keep track of *oldData*, *newData*, *numOldData*, *numNewData*, *totalOldData*, and *totalNewData* all at once—six variables for just this

short fragment. The next example shows how to reduce that number to only three elements within each block of code:

C++ Example of Using Two Sets of Variables More Understandably

Statements using *oldData*.

Statements using *newData*.

```
void SummarizeData( ... ) {
   GetOldData( oldData, &numOldData );
   totalOldData = Sum( oldData, numOldData );
   PrintOldDataSummary( oldData, totalOldData, numOldData );
   SaveOldDataSummary( totalOldData, numOldData );
   ...
   GetNewData( newData, &numNewData );
   totalNewData = Sum( newData, numNewData );
   PrintNewDataSummary( newData, totalNewData, numNewData );
   SaveNewDataSummary( totalNewData, numNewData );
   ...
}
```

When the code is broken up, the two blocks are each shorter than the original block and individually contain fewer variables. They're easier to understand, and if you need to break this code out into separate routines, the shorter blocks with fewer variables will promote better-defined routines.

Break groups of related statements into separate routines All other things being equal, a variable in a shorter routine will tend to have smaller span and live time than a variable in a longer routine. By breaking related statements into separate, smaller routines, you reduce the scope that the variable can have.

Cross-Reference For more on global variables, see Section 13.3, "Global Data."

Begin with most restricted visibility, and expand the variable's scope only if necessary Part of minimizing the scope of a variable is keeping it as local as possible. It is much more difficult to reduce the scope of a variable that has had a large scope than to expand the scope of a variable that has had a small scope—in other words, it's harder to turn a global variable into a class variable than it is to turn a class variable into a global variable. It's harder to turn a protected data member into a private data member than vice versa. For that reason, when in doubt, favor the smallest possible scope for a variable: local to a specific loop, local to an individual routine, then private to a class, then protected, then package (if your programming language supports that), and global only as a last resort.

Comments on Minimizing Scope

Many programmers' approach to minimizing variables' scope depends on their views of the issues of "convenience" and "intellectual manageability." Some programmers make many of their variables global because global scope makes variables convenient to access and the programmers don't have to fool around with parameter lists and class-scoping rules. In their minds, the convenience of being able to access variables at any time outweighs the risks involved.

Cross-Reference The idea of minimizing scope is related to the idea of information hiding. For details, see "Hide Secrets (Information Hiding)" in Section 5.3.

Other programmers prefer to keep their variables as local as possible because local scope helps intellectual manageability. The more information you can hide, the less you have to keep in mind at any one time. The less you have to keep in mind, the smaller the chance that you'll make an error because you forgot one of the many details you needed to remember.

KEY POINT

The difference between the "convenience" philosophy and the "intellectual manageability" philosophy boils down to a difference in emphasis between writing programs and reading them. Maximizing scope might indeed make programs easy to write, but a program in which any routine can use any variable at any time is harder to understand than a program that uses well-factored routines. In such a program, you can't understand only one routine; you have to understand all the other routines with which that routine shares global data. Such programs are hard to read, hard to debug, and hard to modify.

Cross-Reference For details on using access routines, see "Using Access Routines Instead of Global Data" in Section 13.3.

Consequently, you should declare each variable to be visible to the smallest segment of code that needs to see it. If you can confine the variable's scope to a single loop or to a single routine, great. If you can't confine the scope to one routine, restrict the visibility to the routines in a single class. If you can't restrict the variable's scope to the class that's most responsible for the variable, create access routines to share the variable's data with other classes. You'll find that you rarely, if ever, need to use naked global data.

10.5 Persistence

"Persistence" is another word for the life span of a piece of data. Persistence takes several forms. Some variables persist

- for the life of a particular block of code or routine. Variables declared inside a *for* loop in C++ or Java are examples of this kind of persistence.
- as long as you allow them to. In Java, variables created with *new* persist until they are garbage collected. In C++, variables created with *new* persist until you *delete* them.
- for the life of a program. Global variables in most languages fit this description, as do *static* variables in C++ and Java.
- forever. These variables might include values that you store in a database between executions of a program. For example, if you have an interactive program in which users can customize the color of the screen, you can store their colors in a file and then read them back each time the program is loaded.

The main problem with persistence arises when you assume that a variable has a longer persistence than it really does. The variable is like that jug of milk in your refrigerator. It's supposed to last a week. Sometimes it lasts a month, and sometimes it

turns sour after five days. A variable can be just as unpredictable. If you try to use the value of a variable after its normal life span is over, will it have retained its value? Sometimes the value in the variable is sour, and you know that you've got an error. Other times, the computer leaves the old value in the variable, letting you imagine that you have used it correctly.

Here are a few steps you can take to avoid this kind of problem:

Cross-Reference Debug code is easy to include in access routines and is discussed more in "Advantages of Access Routines" in Section 13.3.

- Use debug code or assertions in your program to check critical variables for reasonable values. If the values aren't reasonable, display a warning that tells you to look for improper initialization.
- Set variables to "unreasonable values" when you're through with them. For example, you could set a pointer to null after you delete it.
- Write code that assumes data isn't persistent. For example, if a variable has a certain value when you exit a routine, don't assume it has the same value the next time you enter the routine. This doesn't apply if you're using language-specific features that guarantee the value will remain the same, such as *static* in C++ and Java.
- Develop the habit of declaring and initializing all data right before it's used. If you see data that's used without a nearby initialization, be suspicious!

10.6 Binding Time

An initialization topic with far-reaching implications for program maintenance and modifiability is "binding time": the time at which the variable and its value are bound together (Thimbleby 1988). Are they bound together when the code is written? When it is compiled? When it is loaded? When the program is run? Some other time?

It can be to your advantage to use the latest binding time possible. In general, the later you make the binding time, the more flexibility you build into your code. The next example shows binding at the earliest possible time, when the code is written:

Java Example of a Variable That's Bound at Code-Writing Time

```
titleBar.color = 0xFF; // 0xFF is hex value for color blue
```

The value *0xFF* is bound to the variable *titleBar.color* at the time the code is written because *0xFF* is a literal value hard-coded into the program. Hard-coding like this is nearly always a bad idea because if this *0xFF* changes, it can get out of synch with *0xFF*s used elsewhere in the code that must be the same value as this one.

Here's an example of binding at a slightly later time, when the code is compiled:

Java Example of a Variable That's Bound at Compile Time

```
private static final int COLOR_BLUE = 0xFF;
private static final int TITLE_BAR_COLOR = COLOR_BLUE;
...
titleBar.color = TITLE_BAR_COLOR;
```

TITLE_BAR_COLOR is a named constant, an expression for which the compiler substitutes a value at compile time. This is nearly always better than hard-coding, if your language supports it. It increases readability because *TITLE_BAR_COLOR* tells you more about what is being represented than *0xFF* does. It makes changing the title bar color easier because one change accounts for all occurrences. And it doesn't incur a run-time performance penalty.

Here's an example of binding later, at run time:

Java Example of a Variable That's Bound at Run Time

```
titleBar.color = ReadTitleBarColor();
```

ReadTitleBarColor() is a routine that reads a value while a program is executing, perhaps from the Microsoft Windows registry file or a Java properties file.

The code is more readable and flexible than it would be if a value were hard-coded. You don't need to change the program to change *titleBar.color*; you simply change the contents of the source that's read by *ReadTitleBarColor()*. This approach is commonly used for interactive applications in which a user can customize the application environment.

There is still another variation in binding time, which has to do with when the *ReadTitleBarColor()* routine is called. That routine could be called once at program load time, each time the window is created, or each time the window is drawn—each alternative represents successively later binding times.

To summarize, following are the times a variable can be bound to a value in this example. (The details could vary somewhat in other cases.)

- Coding time (use of magic numbers)
- Compile time (use of a named constant)
- Load time (reading a value from an external source such as the Windows registry file or a Java properties file)
- Object instantiation time (such as reading the value each time a window is created)
- Just in time (such as reading the value each time the window is drawn)

In general, the earlier the binding time, the lower the flexibility and the lower the complexity. For the first two options, using named constants is preferable to using magic numbers for many reasons, so you can get the flexibility that named constants provide just by using good programming practices. Beyond that, the greater the flexibility desired, the higher the complexity of the code needed to support that flexibility and the more error-prone the code will be. Because successful programming depends on minimizing complexity, a skilled programmer will build in as much flexibility as needed to meet the software's requirements but will not add flexibility—and related complexity—beyond what's required.

10.7 Relationship Between Data Types and Control Structures

Data types and control structures relate to each other in well-defined ways that were originally described by the British computer scientist Michael Jackson (Jackson 1975). This section sketches the regular relationship between data and control flow.

Jackson draws connections between three types of data and corresponding control structures:

Cross-Reference For details on sequences, see Chapter 14, "Organizing Straight-Line Code."

Sequential data translates to sequential statements in a program Sequences consist of clusters of data used together in a certain order, as suggested by Figure 10-2. If you have five statements in a row that handle five different values, they are sequential statements. If you read an employee's name, Social Security Number, address, phone number, and age from a file, you'd have sequential statements in your program to read sequential data from the file.

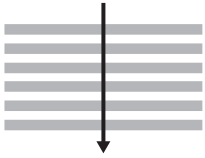

Figure 10-2 Sequential data is data that's handled in a defined order.

Cross-Reference For details on conditionals, see Chapter 15, "Using Conditionals."

Selective data translates to* if *and* case *statements in a program In general, selective data is a collection in which one of several pieces of data is used at any particular time, but only one, as shown in Figure 10-3. The corresponding program statements must do the actual selection, and they consist of *if-then-else* or *case* statements. If you had an employee payroll program, you might process employees differently depending on whether they were paid hourly or salaried. Again, patterns in the code match patterns in the data.