10

General Issues in Using Variables

Contents

Related Topics

IT'S NORMAL AND DESIRABLE FOR construction to fill in small gaps in the requirements and architecture. It would be inefficient to draw blueprints to such a microscopic level that every detail was completely specified. This chapter describes a nuts and bolts construction issue—ins and outs of using variables.

The information in this chapter should be particularly valuable to you if you're an experienced programmer. It's easy to start using hazardous practices before you're fully aware of your alternatives and then to continue to use them out of habit even after you've learned ways to avoid them. An experienced programmer might find the discussions on binding time in Section 10.6 and on using each variable for one purpose in Section 10.8 particularly interesting. If you're not

29 sure whether you qualify as an "experienced programmer," take the "Data
30 Literacy Test" in the next section, and find out.

31 Throughout this chapter I use the word "variable" to refer to objects as well as to
32 built-in data types like integers and arrays. The phrase "data type" generally
33 refers to built-in data types, while the word "data" refers to either objects or
34 built-in types.

10.1 Data Literacy

35

36 **KEY POINT**

37 The first step in creating effective data is knowing which kind of data to create.
38 A good repertoire of data types is a key part of a programmer's toolkit. A tutorial
39 in data types is beyond the scope of this book, but take the "Data Literacy Test"
     below to determine how much more you might need to learn about them.

The Data Literacy Test

40

41 Put a *1* next to each term that looks familiar. If you think you know what a term
42 means but aren't sure, give yourself a *0.5*. Add the points when you're done, and
43 interpret your score according to the scoring table below.

_____	abstract data type	_____	literal
_____	array	_____	local variable
_____	bitmap	_____	lookup table
_____	boolean variable	_____	member data
_____	B-tree	_____	pointer
_____	character variable	_____	private
_____	container class	_____	retroactive synapse
_____	double precision	_____	referential integrity
_____	elongated stream	_____	stack
_____	enumerated type	_____	string
_____	floating point	_____	structured variable
_____	heap	_____	tree
_____	index	_____	typedef
_____	integer	_____	union
_____	linked list	_____	value chain

_____ named constant   _____ variant

_____ **Total Score**

44   Here is how you can interpret the scores (loosely):

0–14	You are a beginning programmer, probably in your first year of computer science in school or teaching yourself your first programming language. You can learn a lot by reading one of the books listed below. Many of the descriptions of techniques in this part of the book are addressed to advanced programmers, and you'll get more out of them after you've read one of these books.
15–19	You are an intermediate programmer or an experienced programmer who has forgotten a lot. Although many of the concepts will be familiar to you, you too can benefit from reading one of the books listed below.
20–24	You are an expert programmer. You probably already have the books listed below on your shelf.
25–29	You know more about data types than I do. Consider writing your own computer book. (Send me a copy!)
30–32	You are a pompous fraud. The terms "elongated stream," "retroactive synapse," and "value chain" don't refer to data types—I made them up. Please read the intellectual-honesty section in Chapter 31!

45   ## Additional Resources on Data Types

46   These books are good sources of information about data types:

47   Cormen, H. Thomas, Charles E. Leiserson, Ronald L. Rivest. *Introduction to*
48   *Algorithms*. New York: McGraw Hill. 1990.

49   Sedgewick, Robert. *Algorithms in C++, Part 5, 3d ed.* Boston, Mass.: Addison-
50   Wesley, 2002.

51   Sedgewick, Robert. *Algorithms in C++, Parts 1-4, 3d ed.* Boston, Mass.:
52   Addison-Wesley, 1998.

10.2 Making Variable Declarations Easy

This section describes what you can do to streamline the task of declaring variables. To be sure, this is a small task, and you may think it's too small to deserve its own section in this book. Nevertheless, you spend a lot of time creating variables, and developing the right habits can save time and frustration over the life of a project.

Implicit Declarations

Some languages have implicit variable declarations. For example, if you use a variable in Visual Basic without declaring it, the compiler declares it for you automatically (depending on your compiler settings).

Implicit declaration is one of the most hazardous features available in any language.

If you program in Visual Basic, you know how frustrating it is to try to figure out why *acctNo* doesn't have the right value and then notice that *acctNum* is the variable that's reinitialized to *O*. This kind of mistake is an easy one to make if your language doesn't require you to declare variables.

KEY POINT

If you're programming in a language that requires you to declare variables, you have to make two mistakes before your program will bite you. First you have to put both *acctNum* and *acctNo* into the body of the routine. Then you have to declare both variables in the routine. This is a harder mistake to make and virtually eliminates the synonymous-variables problem. Languages that require you explicitly to declare data force you to use data more carefully, which is one of their primary advantages. What do you do if you program in a language with implicit declarations? Here are some suggestions:

Turn off implicit declarations
Some compilers allow you to disable implicit declarations. For example, in Visual Basic you would use an *Option Explicit* statement, which forces you to declare all variables before you use them.

Declare all variables
As you type in a new variable, declare it, even though the compiler doesn't require you to. This won't catch all the errors, but it will catch some of them.

Use naming conventions
Establish a naming convention for common suffixes such as *Num* and *No* so that you don't use two variables when you mean to use one.

CROSS-REFERENCE For details on layout of variable declarations, see "Laying Out Data Declarations" in Section 31.5. For details on documenting them, see "Commenting Data Declarations" in Section 32.5.

CROSS-REFERENCE For details on the standardization of abbreviations, see "General Abbreviation Guidelines" in Section 11.6.

87 ***Check variable names***
88 Use the cross-reference list generated by your compiler or another utility
89 program. Many compilers list all the variables in a routine, allowing you to spot
90 both *acctNum* and *acctNo*. They also point out variables that you've declared
91 and not used.

10.3 Guidelines for Initializing Variables

92

93 **KEY POINT**

Improper data initialization is one of the most fertile sources of error in computer
94 programming. Developing effective techniques for avoiding initialization
95 problems can save a lot of debugging time.

96 The problems with improper initialization stem from a variable's containing an
97 initial value that you do not expect it to contain. This can happen for any of the
98 several reasons described on the next page.

99 **CROSS-REFERENCE** For
100 a testing approach based on
data initialization and use
101 patterns, see "Data-Flow
102 Testing" in Section 22.3.

- The variable has never been assigned a value. Its value is whatever bits
  happened to be in its area of memory when the program started.

- The value in the variable is outdated. The variable was assigned a value at
  some point, but the value is no longer valid.

103 - Part of the variable has been assigned a value and part has not.

104 This last theme has several variations. You can initialize some of the members of
105 an object but not all of them. You can forget to allocate memory and then
106 initialize the "variable" the uninitialized pointer points to. This means that you
107 are really selecting a random portion of computer memory and assigning it some
108 value. It might be memory that contains data. It might be memory that contains
109 code. It might be the operating system. The symptom of the pointer problem can
110 manifest itself in completely surprising ways that are different each time—that's
111 what makes debugging pointer errors harder than debugging other errors.

112 Here are guidelines for avoiding initialization problems.

113 ***Initialize each variable as it's declared***
114 Initializing variables as they're declared is an inexpensive form of defensive
115 programming. It's a good insurance policy against initialization errors. The
116 example below ensures that *studentName* will be reinitialized each time you call
117 the routine that contains it.

CROSS-REFERENCE Checking input parameters is a form of defensive programming. For details on defensive programming, see Chapter 8, "Defensive Programming."

CODING HORROR

Considerations" in Section 11.4.

C++ Example of Initialization at Declaration Time

```cpp
char  studentName [ NAME_LENGTH + 1 ] = {'\0'};   // full name of student
```

Initialize each variable close to where it's first used

Some languages, including Visual Basic, don't support initializing variables as they're declared. That can lead to coding styles like the one below, in which declarations are grouped together, and then initializations are grouped together— all far from the first actual use of the variables.

Visual Basic Example of Bad Initialization

```vb
' declare all variables
Dim accountIndex As Integer
Dim total As Double
Dim done As Boolean

' initialize all variables
accountIndex = 0
total = 0.0
done = False
...

' code using accountIndex
...

' code using total
...

' code using done
While Not done
    ...
```

A better practice is to initialize variables as close as possible to where they're first used:

Visual Basic Example of Good Initialization

```vb
Dim accountIndex As Integer
accountIndex = 0
' code using accountIndex
...

Dim total As Double
total = 0.0
' code using total
...

Dim done As Boolean
```

total is declared and initialized close to where it's used.

160 *done is also declared and* 161 *initialized close to where it's* 162 *used.*	``` done = False ' code using done While Not done ... ```

163

164 **CROSS-REFERENCE** For
165 more details on keeping
166 related actions together, see
166 Section 14.2, "Statements
167 Whose Order Doesn't
168 Matter."

The second example is superior to the first for several reasons. By the time execution of the first example gets to the code that uses *done*, *done* could have been modified. If that's not the case when you first write the program, later modifications might make it so. Another problem with the first approach is that throwing all the initializations together creates the impression that all the variables are used throughout the whole routine—when in fact *done* is used only at the end. Finally, as the program is modified (as it will be, if only by debugging), loops might be built around the code that uses *done*, and *done* will need to be reinitialized. The code in the second example will require little modification in such a case. The code in the first example is more prone to producing an annoying initialization error.

This is an example of the Principle of Proximity: Keep related actions together. The same principle applies to keeping comments close to the code they describe, to keeping loop setup code close to the loop, to grouping statements in straight-line code, and to many other areas.

Ideally, declare and define each variable close to where it's used

A declaration establishes a variable's type. A definition assigns the variable a specific value. In languages that support it, such as C++ and Java, variables should be declared and defined close to where they are first used. Ideally, each variable should be defined at the same time it's declared, as shown below.

Java Example of Good Initialization

```
int accountIndex = 0;
// code using accountIndex
...
```

189 total *is initialized close to*
190 *where it's used.*

```
double total = 0.0;
// code using total
...
```

193 done *is also initialized close to*
194 *where it's used.*

```
boolean done = false;
// code using done
while ( ! done ) {
    ...
```

Pay special attention to counters and accumulators

The variables *i*, *j*, *k*, *sum*, and *total* are often counters or accumulators. A common error is forgetting to reset a counter or an accumulator before the next time it's used.

Initialize a class's member data in its constructor

Just as a routine's variables should be initialized within each routine, a class's data should be initialized within its constructor. If memory is allocated in the constructor, it should be freed in the destructor.

Check the need for reinitialization

Ask yourself whether the variable will ever need to be reinitialized—either because a loop in the routine uses the variable many times or because the variable retains its value between calls to the routine and needs to be reset between calls. If it needs to be reinitialized, make sure that the initialization statement is inside the part of the code that's repeated.

Initialize named constants once; initialize variables with executable code

If you're using variables to emulate named constants, it's OK to write code that initializes them once, at the beginning of the program. To do this, initialize them in a *Startup()* routine. Initialize true variables in executable code close to where they're used. One of the most common program modifications is to change a routine that was originally called once so that you call it multiple times. Variables that are initialized in a program-level *Startup()* routine aren't reinitialized the second time through the routine.

Use the compiler setting that automatically initializes all variables

If your compiler supports such an option, having the compiler set to automatically initialize all variables is an easy variation on the theme of relying on your compiler. Relying on specific compiler settings, however, can cause problems when you move the code to another machine and another compiler. Make sure you document your use of the compiler setting; assumptions that rely on specific compiler settings are hard to uncover otherwise.

Take advantage of your compiler's warning messages

Many compilers warn you that you're using an uninitialized variable.

Check input parameters for validity

Another valuable form of initialization is checking input parameters for validity. Before you assign input values to anything, make sure the values are reasonable.

Use a memory-access checker to check for bad pointers

In some operating systems, the operating-system code checks for invalid pointer references. In others, you're on your own. You don't have to stay on your own,

234 however, because you can buy memory-access checkers that check your
235 program's pointer operations.

236 ***Initialize working memory at the beginning of your program***
237 Initializing working memory to a known value helps to expose initialization
238 problems. You can take any of several approaches:

239 • You can use a preprogram memory filler to fill the memory with a
240 predictable value. The value *0* is good for some purposes because it ensures
241 that uninitialized pointers point to low memory, making it relatively easy to
242 detect them when they're used. On the Intel processors, 0xCC is a good
243 value to use because it's the machine code for a breakpoint interrupt; if you
244 are running code in a debugger and try to execute your data rather than your
245 code, you'll be awash in breakpoints. Another virtue of the value 0xCC is
246 that it's easy to recognize in memory dumps— and it's rarely used for
247 legitimate reasons. Alternatively, Brian Kernighan and Rob Pike suggest
248 using the constant *0xDEADBEEF* as memory filler that's easy to recognize
249 in a debugger (1999).

250 • If you're using a memory filler, you can change the value you use to fill the
251 memory once in awhile. Shaking up the program sometimes uncovers
252 problems that stay hidden if the environmental background never changes.

253 • You can have your program initialize its working memory at startup time.
254 Whereas the purpose of using a preprogram memory filler is to expose
255 defects, the purpose of this technique is to hide them. By filling working
256 memory with the same value every time, you guarantee that your program
257 won't be affected by random variations in the startup memory.

258 # 10.4 Scope

259 "Scope" is a way of thinking about a variable's celebrity status: how famous is
260 it? Scope, or visibility, refers to the extent to which your variables are known
261 and can be referenced throughout a program. A variable with limited or small
262 scope is known in only a small area of a program—a loop index used in only one
263 small loop, for instance. A variable with large scope is known in many places in
264 a program—a table of employee information that's used throughout a program,
265 for instance.

266 Different languages handle scope in different ways. In some primitive languages,
267 all variables are global. You therefore don't have any control over the scope of a
268 variable, and that can create a lot of problems. In C++ and similar languages, a
269 variable can be visible to a block (a section of code enclosed in curly brackets), a

270   routine, a class, or the whole program. In Java and C#, a variable can also be
271   visible to a package or namespace (a collection of classes).

272   The following sections provide guidelines that apply to scope.

Localize References to Variables

274   The code between references to a variable is a "window of vulnerability." In the
275   window, new code might be added, inadvertently altering the variable, or
276   someone reading the code might forget the value the variable is supposed to
277   contain. It's always a good idea to localize references to variables by keeping
278   them close together.

279   The idea of localizing references to a variable is pretty self-evident, but it's an
280   idea that lends itself to formal measurement. One method of measuring how
281   close together the references to a variable are is to compute the "span" of a
282   variable. Here's an example:

Java Example of Variable Span

```
a = 0;
b = 0;
c = 0;
a = b + c;
```

288   In this case, two lines come between the first reference to *a* and the second, so *a*
289   has a span of two. One line comes between the two references to *b*, so *b* has a
290   span of one, and *c* has a span of zero. Here's another example:

Java Example of Spans of One and Zero

```
a = 0;
b = 0;
c = 0;
b = a + 1;
b = b / c;
```

FURTHER READING For
more information on variable
span, see *Software
Engineering Metrics and
Models* (Conte, Dunsmore,
and Shen 1986).

In this case, there is one line between the first reference to *b* and the second, for a
span of one. There are no lines between the second reference to *b* and the third,
for a span of zero.

The average span is computed by averaging the individual spans; in *b*'s case,
(1+0)/2 equals an average span of 0.5. When you keep references to variables
close together, you enable the person reading your code to focus on one section
at a time. If the references are far apart, you force the reader to jump around in
the program. Thus the main advantage of keeping references to variables
together is that it improves program readability.

306   # Keep Variables Live for As Short a Time As
307   # Possible

308   A concept that's related to variable span is variable "live time," the total number
309   of statements over which a variable is live. A variable's life begins at the first
310   statement in which it's referenced; its life ends at the last statement in which it's
311   referenced.

312   Unlike span, live time isn't affected by how many times the variable is used
313   between the first and last times it's referenced. If the variable is first referenced
314   on line 1 and last referenced on line 25, it has a live time of 25 statements. If
315   those are the only two lines in which it's used, it has an average span of 23
316   statements. If the variable were used on every line from line 1 through line 25, it
317   would have an average span of 0 statements, but it would still have a live time of
318   25 statements. Figure 10-1 illustrates both span and live time.

319
320   **F10xx01**
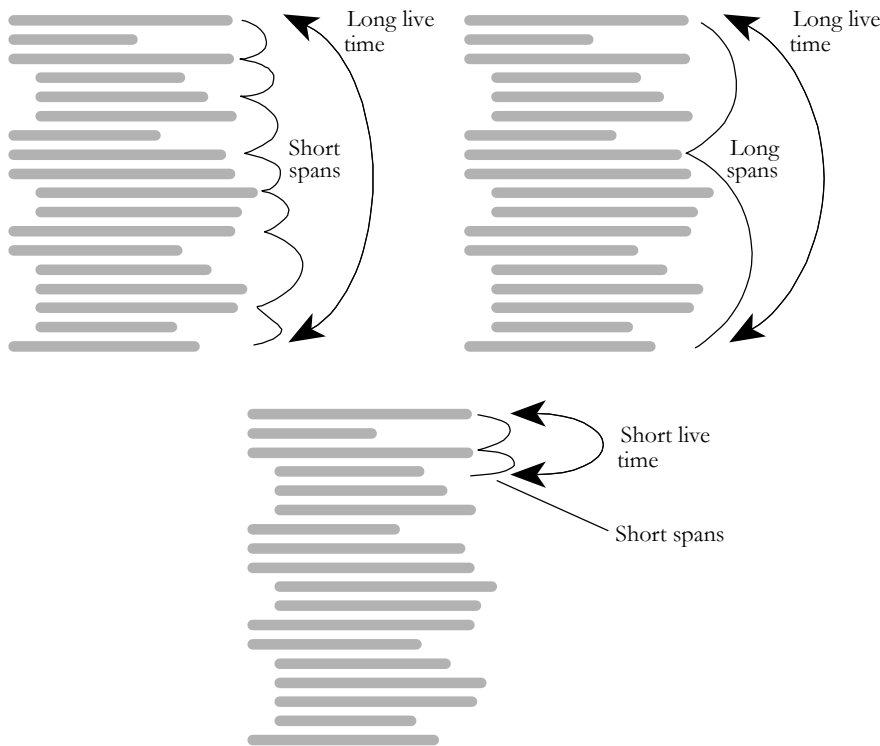
321   **Figure 10-1**
322   *"Long live time" means that a variable is alive over the course of many statements.*
323   *"Short live time" means it's alive for only a few statements. "Span" refers to how*
324   *close together the references to a variable are.*

325
326
327
328
329

As with span, the goal with respect to live time is to keep the number low, to keep a variable live for as short a time as possible. And as with span, the basic advantage of maintaining a low number is that it reduces the window of vulnerability. You reduce the chance of incorrectly or inadvertently altering a variable between the places in which you intend to alter it.

330
331
332
333
334
335
336

A second advantage of keeping the live time short is that it gives you an accurate picture of your code. If a variable is assigned a value in line 10 and not used again until line 45, the very space between the two references implies that the variable is used between lines 10 and 45. If the variable is assigned a value in line 44 and used in line 45, no other uses of the variable are implied, and you can concentrate on a smaller section of code when you're thinking about that variable.

337
338
339
340
341

A short live time also reduces the chance of initialization errors. As you modify a program, straight-line code tends to turn into loops and you tend to forget initializations that were made far away from the loop. By keeping the initialization code and the loop code closer together, you reduce the chance that modifications will introduce initialization errors.

342
343
344
345
346

Finally, a short live time makes your code more readable. The fewer lines of code a reader has to keep in mind at once, the easier your code is to understand. Likewise, the shorter the live time, the less code you have to keep on your screen when you want to see all the references to a variable during editing and debugging.

347

Measuring the Live Time of a Variable

348
349
350

You can formalize the concept of live time by counting the number of lines between the first and last references to a variable (including both the first and last lines). Here's an example with live times that are too long:

351

Java Example of Variables with Excessively Long Live Times

352
353
354
355
356
357
358
359 *Last reference to* recordIndex
360
361
362
363

```
1    // initialize all variables
2    recordIndex = 0;
3    total = 0;
4    done = false;
     ...
26   while ( recordIndex < recordCount ) {
27   ...
28      recordIndex = recordIndex + 1;
        ...

64   while ( !done ) {
        ...
```

| 364 | *Last reference to* total |
| 365 | *Last reference to* done |

```
69    if ( total > projectedTotal ) {
70        done = true;
```

| 366 | Here are the live times for the variables in this example: |

recordIndex	(line 28 – line 2 + 1) = 27
total	(line 69 – line 3 + 1) = 67
done	(line 70 – line 4 + 1) = 67
Average Live Time	(27 + 67 + 67) / 3 ≈ 54

367 The example has been rewritten below so that the variable references are closer
368 together:

Java Example of Variables with Good, Short Live Times

370		```...```
371	*Initialization of* recordIndex *is*	`25  recordIndex = 0;`
372	*moved down from line 3.*	`26  while ( recordIndex < recordCount ) {`
373		`27  ...`
374		`28     recordIndex = recordIndex + 1;`
375		`...`
376	*Initialization of* total *and* done	`62  total = 0;`
377	*are moved down from lines 4*	`63  done = false;`
378	*and 5.*	`64  while ( !done ) {`
379		`...`
380		`69     if ( total > projectedTotal ) {`
381		`70         done = true;`

382 Here are the live times for the variables in this example:

recordIndex	(line 28-line 25 + 1) = 4
total	(line 69-line 62 + 1) = 8
done	(line 70-line 63 + 1) = 8
Average Live Time	(4 + 8 + 8) / 3 ≈ 7

<table>
<tr><td>

383 **FURTHER READING** For
384 more information on "live"
385 variables, see *Software*
Engineering Metrics and
386 *Models* (Conte, Dunsmore,
387 and Shen 1986).

</td><td>

Intuitively, the second example seems better than the first because the initializations for the variables are performed closer to where the variables are used. The measured difference in average live time between the two examples is significant: An average of 54 vs. an average of 7 provides good quantitative support for the intuitive preference for the second piece of code.

</td></tr>
</table>

388 Does a hard number separate a good live time from a bad one? A good span from
389 a bad one? Researchers haven't yet produced that quantitative data, but it's safe
390 to assume that minimizing both span and live time is a good idea.

391 If you try to apply the ideas of span and live time to global variables, you'll find
392 that global variables have enormous spans and live times—one of many good
393 reasons to avoid global variables.

General Guidelines for Minimizing Scope

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

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.

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 );
```

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

```
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 );
  ...
}
```

Statements using two sets of variables

Note that, in the example above, you have to keep track of *oldData*, *newData*, *numOldData*, *numNewData*, *totalOldData*, and *totalNewData* all at once—six variables for just this short fragment. The example below shows how to reduce that number to only three elements

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

```
433
434   void SummarizeDaily( ... ) {
435     GetOldData( oldData, &numOldData );
436     totalOldData = Sum( oldData, numOldData );
437     PrintOldDataSummary( oldData, totalOldData, numOldData );
438     SaveOldDataSummary( totalOldData, numOldData );
439     ...
440     GetNewData( newData, &numNewData );
441     totalNewData = Sum( newData, numNewData );
442     PrintNewDataSummary( newData, totalNewData, numNewData );
443     SaveNewDataSummary( totalNewData, numNewData );
444     ...
445   }
```

Statements using oldData (margin note for lines 435–438)

Statements using newData (margin note for lines 440–443)

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

***Begin with most restricted visibility, and expand the variable's scope only if
necessary***

452 Part of minimizing the scope of a variable is keeping it as local as possible. It is
453 much more difficult to reduce the scope of a variable that has had a large scope
454 than to expand the scope of a variable that has had a small scope—in other
455 words, it's harder to turn a global variable into a class variable than it is to turn a
456 class variable into a global variable. It's harder to turn a protected data member
457 into a private data member than vice versa. For that reason, when in doubt, favor
458 the smallest possible scope for a variable—local to an individual routine if
459 possible, then private, then protected, then package (if your programming
460 language supports that), and global only as a last resort.

Comments on Minimizing Scope

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

473 **KEY POINT**

474
475
476
477
478
479
480

485
486
487

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.

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.

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 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

488

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

491 ● for the life of a particular block of code or routine. Variables declared inside
492 a *for* loop in C++ or Java are examples of this kind of persistence.

493 ● as long as you allow them to. In Java, variables created with *new* persist
494 until they are garbage collected. In C++, variables created with *new* persist
495 until you *delete* them.

496 ● for the life of a program. Global variables in most languages fit this
497 description, as do *static* variables in C++ and Java.

498 ● forever. These variables might include values that you store in a database
499 between executions of a program. For example, if you have an interactive
500 program in which users can customize the color of the screen, you can store
501 their colors in a file and then read them back each time the program is
502 loaded.

503
504
505
506
507
508
509
510

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.

511

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

512 **CROSS-REFERENCE** Deb
513 ug code is easy to include in
514 access routines and is
discussed more in
515 "Advantages of Access
516 Routines" in Section 13.3.
517
518
519

- 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.

- 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.

520
521

- 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!

522

10.6 Binding Time

523
524
525
526
527

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?

528
529
530
531

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.

532

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

533

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

534
535
536
537
538

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.

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

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

```
541          private static final int COLOR_BLUE = 0xFF;
542          private static final int TITLE_BAR_COLOR = COLOR_BLUE;
543          ...
544          titleBar.color = TITLE_BAR_COLOR;
```

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

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

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

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

554   *ReadTitleBarColor()* is a routine that reads a value while a program is executing,
555   perhaps from the Windows registry.

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

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

565   To summarize, here are the times a variable can be bound to a value in this
566   example (the details could vary somewhat in other cases):

567   ● Coding time (use of magic numbers)

568   ● Compile time (use of a named constant)

569   ● Load time (reading a value from an external source such as the Windows
570   Registry)

571   ● Object instantiation time (such as reading the value each time a window is
572   created)

573   ● Just in time (such as reading the value each time the window is drawn)

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

10.7 Relationship Between Data Types and Control Structures

584
585

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

590 Jackson draws connections between three types of data and corresponding
591 control structures.

592 **CROSS-REFERENCE**   For
593 details on sequences, see
594 Chapter 14, "Organizing
595 Straight-Line Code."

Sequential data translates to sequential statements in a program
Sequences consist of clusters of data used together in a certain order. 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.

599 **F10xx02**
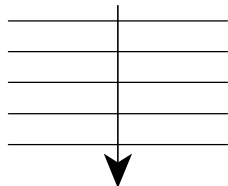
600 **Figure 10-2**

601 *Sequential data is data that's handled in a defined order.*

602 **CROSS-REFERENCE**   For
603 details on conditionals, see
604 Chapter 15, "Using
605 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
present at any particular time—one of the elements is selected. The
corresponding program statements must do the actual selection, and they consist

1/13/2004 2:44 PM

606
607
608

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.

609

610

F10xx03

611

Figure 10-3

612
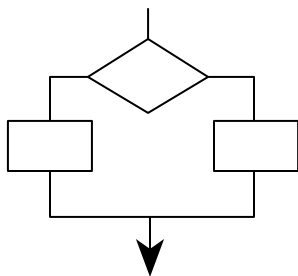
Selective data allows you to use one piece or the other, but not both.

616
617
618

Iterative data translates to for, repeat, and while looping structures in a program
Iterative data is the same type of data repeated several times. Typically, iterative data is stored as records in a file or in arrays. You might have a list of social security numbers that you read from a file. The iterative data would match the iterative code loop used to read the data.

619

620

F10xx04

621

Figure 10-4

622
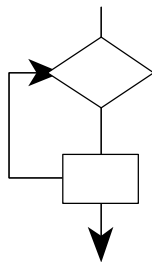
Iterative data is repeated.

623
624
625

Your real data can be combinations of the sequential, selective, and iterative types of data. You can combine the simple building blocks to describe more complicated data types.

10.8 Using Each Variable for Exactly One Purpose

626
627

628
629

It's possible to use variables for more than one purpose in several subtle ways. You're better off without this kind of subtlety.

630

Use each variable for one purpose only

631
632
633
634
635

It's sometimes tempting to use one variable in two different places for two different activities. Usually, the variable is named inappropriately for one of its uses, or a "temporary" variable is used in both cases (with the usual unhelpful name *x* or *temp*). Here's an example that shows a temporary variable that's used for two purposes:

636 **CODING HORROR**

C++ Example of Using One Variable for Two Purposes—Bad Practice

```
// Compute roots of a quadratic equation.
// This code assumes that (b*b-4*a*c) is positive.
temp = Sqrt( b*b - 4*a*c );
root[0] = ( -b + temp ) / ( 2 * a );
root[1] = ( -b - temp ) / ( 2 * a );
...
// swap the roots
temp = root[0];
root[0] = root[1];
root[1] = temp;
```

637
638
639
640
641
642
643
644
645
646

CROSS-REFERENCE   Routine parameters should also be used for one purpose only. For details on using routine parameters, see Section 7.5, "How to Use Routine Parameters."

647
648
649
650
651

Question: What is the relationship between *temp* in the first few lines and *temp* in the last few? Answer: The two *temp* have no relationship. Using the same variable in both instances makes it seem as though they're related when they're not. Creating unique variables for each purpose makes your code more readable. Here's an improvement on the example above:

652

C++ Example of Using Two Variables for Two Purposes—Good Practice

```
// Compute roots of a quadratic equation.
// This code assumes that (b*b-4*a*c) is positive.
discriminant = Sqrt( b*b - 4*a*c );
root[0] = ( -b + discriminant ) / ( 2 * a );
root[1] = ( -b - discriminant ) / ( 2 * a );
...
// swap the roots
oldRoot = root[0];
root[0] = root[1];
root[1] = oldRoot;
```

653
654
655
656
657
658
659
660
661
662

663   *Avoid variables with hidden meanings*

664   Another way in which a variable can be used for more than one purpose is to

665   have different values for the variable mean different things. For example

666   **CODING HORROR**

- The value in the variable *pageCount* might represent the number of pages

667   printed, unless it equals *-1*, in which case it indicates that an error has

668   occurred.

669 - The variable *customerId* might represent a customer number, unless its value

670   is greater than *500,000*, in which case you subtract *500,000* to get the

671   number of a delinquent account.

672 - The variable *bytesWritten* might be the number of bytes written to an output

673   file, unless its value is negative, in which case it indicates the number of the

674   disk drive used for the output.

675 Avoid variables with these kinds of hidden meanings. The technical name for

676 this kind of abuse is "hybrid coupling" (Page-Jones 1988). The variable is

677 stretched over two jobs, meaning that the variable is the wrong type for one of

678 the jobs. In the *pageCount* example above, *pageCount* normally indicates the

679 number of pages; it's an integer. When *pageCount* is *-1*, however, it indicates

680 that an error has occurred; the integer is moonlighting as a boolean!

681 Even if the double use is clear to you, it won't be to someone else. The extra

682 clarity you'll achieve by using two variables to hold two kinds of information

683 will amaze you. And no one will begrudge you the extra storage.

684 **HARD DATA**   *Make sure that all declared variables are used*

685   The opposite of using a variable for more than one purpose is not using it at all.

686   A study by Card, Church, and Agresti found that unreferenced variables were

687   correlated with higher fault rates (1986). Get in the habit of checking to be sure

688   that all variables that are declared are used. Some compilers and utilities (such as

689   lint) report unused variables as a warning.

690 **CC2E.COM/1092**
CROSS-REFERENCE For
a checklist that applies to
specific types of data rather

CHECKLIST: General Considerations In Using Data

691 than general issues, see the
checklist in Chapter 12,

692 "Fundamental Data Types."

Initializing Variables

693 For issues in naming
variables, see the checklist in

☐ Does each routine check input parameters for validity?

694 Chapter 11, "The Power of

☐ Does the code declare variables close to where they're first used?

695 Variable Names."

☐ Does the code initialize variables as they're declared, if possible?

696 ☐ Does the code initialize variables close to where they're first used, if it isn't

possible to declare and initialize them at the same time?

697 ☐ Are counters and accumulators initialized properly and, if necessary,

698 reinitialized each time they are used?

699   ❑ Are variables reinitialized properly in code that's executed repeatedly?

700   ❑ Does the code compile with no warnings from the compiler?

701   ❑ If your language uses implicit declarations, have you compensated for the
702   problems they cause?

703 **Other General Issues in Using Data**

704   ❑ Do all variables have the smallest scope possible?

705   ❑ Are references to variables as close together as possible—both from each
706   reference to a variable to the next and in total live time?

707   ❑ Do control structures correspond to the data types?

708   ❑ Are all the declared variables being used?

709   ❑ Are all variables bound at appropriate times, that is, striking a conscious
710   balance between the flexibility of late binding and the increased complexity
711   associated with late binding?

712   ❑ Does each variable have one and only one purpose?

713   ❑ Is each variable's meaning explicit, with no hidden meanings?

714

Key Points

716 • Data initialization is prone to errors, so use the initialization techniques
717   described in this chapter to avoid the problems caused by unexpected initial
718   values.

719 • Minimize the scope of each variable. Keep references to it close together.
720   Keep it local to a routine or class. Avoid global data.

721 • Keep statements that work with the same variables as close together as
722   possible.

723 • Early binding tends to limit flexibility, but minimize complexity. Late
724   binding tends to increase flexibility, but at the price of increased complexity.

725 • Use each variable for one and only one purpose.

                1/13/2004 2:44 PM