Realtime publishers

The Definitive Guide™ To

Building Code Quality

sponsored by

Don Jones

Chapter 4: Addressing Coding Errors 59
A Taxonomy of Coding Errors 59
Syntax Errors 59
Semantic Errors 60
Logic Errors 60
Memory and Resource Errors 62
Leaks 62
Pointer/Reference Errors, Overruns, and Uninitialized Memory 63
API Failures 66
Error Summary 66
Addressing Coding Errors 67
Editing Tools 67
Compilers 70
Debuggers 70
Automated Tools 72
Error Handling in Code 73
Managing the Native Code/Managed Code Boundary and Monitoring API Calls 74
Case Study: Fixing a Leaky App 75
Errors Fixed 76

Realtime publishers
MICRO FOCUS® Leading the Evolution™

Copyright Statement

© 2009 Realtime Publishers. All rights reserved. This site contains materials that have been created, developed, or commissioned by, and published with the permission of, Realtime Publishers (the "Materials") and this site and any such Materials are protected by international copyright and trademark laws.

THE MATERIALS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. The Materials are subject to change without notice and do not represent a commitment on the part of Realtime Publishers or its web site sponsors. In no event shall Realtime Publishers or its web site sponsors be held liable for technical or editorial errors or omissions contained in the Materials, including without limitation, for any direct, indirect, incidental, special, exemplary or consequential damages whatsoever resulting from the use of any information contained in the Materials.

The Materials (including but not limited to the text, images, audio, and/or video) may not be copied, reproduced, republished, uploaded, posted, transmitted, or distributed in any way, in whole or in part, except that one copy may be downloaded for your personal, non-commercial use on a single computer. In connection with such use, you may not modify or obscure any copyright or other proprietary notice.

The Materials may contain trademarks, services marks and logos that are the property of third parties. You are not permitted to use these trademarks, services marks or logos without prior written consent of such third parties.

Realtime Publishers and the Realtime Publishers logo are registered in the US Patent & Trademark Office. All other product or service names are the property of their respective owners.

If you have any questions about these terms, or if you would like information about licensing materials from Realtime Publishers, please contact us via e-mail at info@realtimepublishers.com.

Realtime publishers

MICRO FOCUS
Leading the Evolution™

[**Editor's Note:** This eBook was downloaded from Realtime Nexus—The Digital Library for IT Professionals. All leading technology eBooks and guides from Realtime Publishers can be found at http://nexus.realtimepublishers.com.]

Chapter 4: Addressing Coding Errors

Without a doubt, coding errors—that is, bugs—are one of the first things anyone thinks about when they speak of code quality. And obviously, code that works is perceived as being of higher quality than code that doesn't! As I've mentioned in previous chapters, focusing on bugs as your exclusive measurement of quality is shortsighted, but that doesn't mean we can't spend plenty of time getting rid of bugs, because bug-free code is definitely higher-quality code.

In this chapter, then, we'll focus on *types* of coding errors because there are quite a few. Each type has distinct techniques for avoidance and removal, and we'll spend the bulk of this chapter focusing on these techniques as well as the tools that help implement those techniques more quickly and efficiently.

A Taxonomy of Coding Errors

Because there *are* many different types of errors, I think it's important to pull them out individually, describe them, and focus on the techniques involved in either avoiding them or removing them. Again: The *techniques* involved. Although I'm a big fan of tools, and in the second portion of this chapter I'll look at plenty of tools, it's really important to understand that *tools do not remove coding errors.* Tools simply implement key techniques—often more quickly and efficiently than you could do so manually—but it's the techniques, whether implemented automatically or manually, that save the day.

Syntax Errors

Syntax errors occur when a language compiler or static code analyzer is parsing the code that you've typed. Typically, syntax errors result from typos, incorrect use of statement syntax, and so forth—illegal characters, missing operators, missing end-of-line characters, unbalanced parentheses or quotation marks, a misused or misplaced reserved word, and so on. Syntax errors can nearly always be avoided or caught if you're using the right tools and techniques.

Here's a straightforward example of a syntax error:

```
If (var) {
  Log("Var is True")
  Return False
)
```


Realtime publishers

MICRO FOCUS
Leading the Evolution™

The **If** construct's enclosing braces aren't balanced; the programmer has mistakenly typed a closing parentheses rather than a closing brace. Actually, depending on the programming language, there might be another syntax error. In most C-based languages (including C#), each line of code should be followed by a semicolon, which isn't the case here. In fact, that illustrates one of the difficulties in manually catching syntax errors: They're entirely dependent upon the language syntax. Developers who are new to a language, or who are used to working in multiple different languages, often make (and miss) these simple mistakes.

Semantic Errors

Semantic errors are more difficult, and sometimes impossible, to catch without running the code. In other words, the code will *look* correct but still contain errors that are detected during the execution of the code. This might include an incorrect variable type or size, subscripts being out of range, and so forth. Some modern development tools *can* catch many types of semantic errors without running the code. They do so through a few different techniques, including very advanced code parsing and analysis, pre-compilation in advanced compilers, and so forth. In fact, some of the most important advances in the .NET Framework and Visual Studio over the years have been increasing the number of semantic errors that can be detected while you're typing the code.

Here's an example of a possible semantic error:

```
For (var; var < 10; var++) {
  Console.WriteLine var;
}
```

This illustrates the difficulty of statically catching semantic errors, especially manually. Was **var** initialized? We don't see any declaration of **var** in this code snippet, so we'd need to scan through the rest of the code to see whether **var** had been declared within our current scope. Alternately, we could simply declare it right here, but if the variable name had already been declared, we might generate another error in declaring it a second time. That ambiguity is exactly why older programming languages and their development environments don't catch this type of error without running the code: They couldn't keep track of all the variables based solely on static parsing of the program code.

Logic Errors

Logic errors are among the more frustrating variety of errors. Typically—and this is an oversimplification, but a good rule—logic errors are caused when a variable, property, or some other container contains something other than what you thought it did. Usually, the code will compile and run without error—but it won't behave the way you want. Here's a very simplistic example:

```
If (var) {
  Delete(file1);
} else {
  Archive(file1);
}
```


Realtime publishers

MICRO FOCUS
Leading the Evolution™

Staring at this code as-is, you can't detect any error. The code is syntactically correct, and for the sake of argument, let's say it's semantically correct as well. But when you run the code, it sometimes incorrectly deletes **file1** rather than archiving it—a logic error. The only possible cause for this particular logic error is that **var** doesn't contain what we expected it to. Given the syntax, we're expecting **var** to contain a Boolean (True/False) value; a problem is that most programming languages interpret any non-zero value as True, and zero as False. Let's expand the example and see a little more of the code surrounding it:

```
Declare int var;

Var = GetUserInput("Delete File? Click No to Archive instead.");

For (var=0; var < 5; var++) {
  If ( UserIsAdmin(UserID,var) ) {
    Exit For;
  }
}

If (var) {
  Delete(file1);
} else {
  Archive(file1);
}
```

This example is, of course, vaguely C#-flavored pseudo-code, so let me walk through the logic.

1. We begin by declaring a variable **var**.
2. We ask for the user's input on whether to delete a file—let's say the GetUserInput() function is displaying a dialog box with Yes and No buttons, and that it returns True if Yes is clicked, and False otherwise.
3. Then we enter a **For** loop, which uses the variable **var** as its counter. This is actually where our logic error comes from: If the user had clicked Yes, then **var** would contain a non-zero value. However, the **For** loop initializes it to zero, losing our user response. If the user is an admin, the **For** loop will immediately exit, leaving **var** containing zero, which most languages interpret as False—the opposite of what the user was after. If the user was not an admin, **var** will eventually equal 5, which most programming languages interpret as True—deleting the file, even if the user said not to.

Realtime publishers

MICRO FOCUS Leading the Evolution™

Obviously, this is a simplistic example, but it shows how tricky logic errors can be in comparison with syntax or semantic errors. It also illustrates how proper programming techniques can often make such errors more difficult to commit:

- If **var** had been declared as a specific data type, such as Boolean, the development environment may have been able to treat **var**'s re-use as an integer (in the **For** loop) as a semantic error.
- If clearer variable names had been used—such as **UserChoice** and **Counter** rather than **var—**it might have been more obvious to the programmer what was going on.
- It's possible that the **For** loop was pasted in as a snippet or from elsewhere. Reusing code is fine, but you always have to examine it to see if it needs to be refactored (that is, variables and other elements renamed) in its new context.

Memory and Resource Errors

Finally, we come to what is definitely the most frustrating and difficult-to-catch type of error. These errors are typically not detectable when coding, nor are they easily detectable during the unit tests a developer would run. These errors often manifest only when the code has been run for some time, with a broad range of input data and with input data that is specifically selected to be outside the ranges of data that the code *should* find itself dealing with.

Understanding these types of errors often requires that you understand a bit more about what's going on "under the hood" in your language compiler, in the computer's operating system (OS), and in any runtime libraries that you may be relying on. Modern programming languages often abstract these lower-level elements, making it even more difficult to understand, detect, and fix these types of errors. The .NET Framework's Common Language Runtime (CLR), in fact, will *in large part* keep these types of errors from happening in managed code (that being one of the real purposes of managed code in the first place)—but not always.

Leaks

Memory leaks are when an application gradually uses more and more memory over time. This is often caused by the application not properly de-allocating memory that is no longer in use. For example, when an application creates a new variable, memory is allocated to store that variable's contents. When the application stops using the variable, that memory should be returned to the pool from whence it came so that it can be re-used. In managed code, memory leaks are fairly rare (at least, the more simplistic ones are) because the CLR takes care of allocating memory and periodically de-allocating it (a process known as *garbage collection*) and returning it to the OS.

In fact, many memory leaks in managed code come from bugs in the underlying Framework classes or components. (The site http://blogs.msdn.com/joncole/archive/2005/12/15/Debugging-a-memory-leak-in-managed-code_3A00_-Ping-_2D00_-SendAsync.aspx documents one such bug from several years ago).

Realtime publishers

MICRO FOCUS Leading the Evolution™

It's important to understand that memory leaks in managed code are *not impossible.* The CLR's garbage collector only handles *managed memory*; that is, it can only de-allocate memory that it allocated in the first place. .NET-based applications use *unmanaged* memory in a number of instances, and many developers are often unaware that their applications are doing so (the most frequent cause is accessing unmanaged code, such as legacy Component Object Model—COM—objects). The CLR itself may rely on unmanaged memory, and some applications deliberately make use of unmanaged memory for various purposes. There are also, of course, times when the garbage collector itself may not perform correctly—often due to subtle programming errors that prevent the garbage collector from doings its job.

Memory leaks can typically be resolved in the short term simply by restarting the application, although in severe instances, an entire OS reboot will be required. This is merely a workaround, however, and will have to be repeated as the application is re-started and the memory leak starts anew.

Pointer/Reference Errors, Overruns, and Uninitialized Memory

Pointers and references can be tricky to work with. In a simplified example, the idea is that a programmer places some data into an area of memory—variables are essentially easier-to-remember names for memory locations. When that data needs to be provided to a function or method or something, the programmer has two choices: pass the data itself (often creating a new copy of it in memory) or pass a *reference* or *pointer* to the original data. The difference is important. When passing *by value* (that is, passing a copy of the data), any changes to the copied data will not be reflected in the original data. When passing *by reference,* any changes made will be made to the original data because that is all there is.

Reference errors occur when a bad reference is provided. Again, to oversimplify a bit, imagine you put some data in memory location 5. It occupies memory locations 5 through 70. When you pass a reference, however, you lose track and pass 6 as the reference. The result is that the code getting the reference isn't getting the correct data. It may then modify locations 6 through 71—the correct length, but offset by one from the correct starting position.

Managed code makes all of this a lot easier, as you typically just pass variable names, and the CLR keeps track of where the actual data lives in memory. But when managed code starts dealing with unmanaged code, such as making OS API calls, you'll have to do more of the pointer management yourself.

Pointers can be more insidious, too. Keep in mind that you usually store a pointer in a variable—meaning it's stored in a location of memory. That means you have one bit of memory acting as a cross-reference (or pointer, hence the term) to another. Figure 4.1 illustrates: Memory location *a* contains a value that points directly to memory location *b*.

Realtime publishers

MICRO FOCUS
Leading the Evolution™

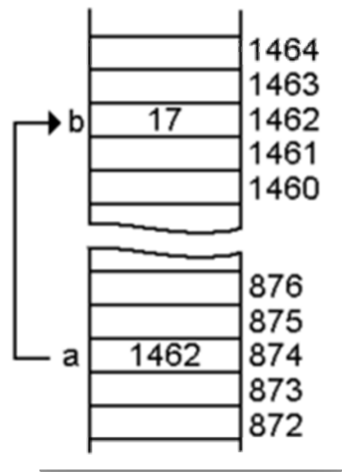

Figure 4.1: Pointers in memory.

Now, let's say memory locations 872 and 873 are being used to store something else. But your code loses track of that pointer, and accidentally overwrites not 872 and 873, but 873 and 874—offset by one from what you should have done. The result is that you've overwritten 874, which will now no longer point to the correct 1462. The result is a cascading effect of errors, often resulting in an application exception or crash—although the results can be much worse, depending on exactly what data is overwritten.

This catastrophic effect is often the goal of a *buffer overrun* attack. For example, suppose that your code expects two bytes of input to a function, and it writes those to 872 and 873 (keeping with the illustration in Figure 4.1). If an attacker can find a way to pass *three* bytes to your function, and if your function is not doing boundary checking, then it'll overwrite 872 through 874. The attacker can control what's in that third byte, so he can control the memory pointer—directing your application to use his data instead of your data. This is something we'll cover in more detail in the next chapter, but it's an example of how pointer errors can create truly devastating results.

> **Note**
> The danger of using pointers is one reason that the use of pointers in C# requires the code to be marked with the **unsafe** keyword.

Realtime publishers

MICRO FOCUS®
Leading the Evolution™

Array overruns are related, conceptually, to pointer errors. An array is really little more than a block of memory used to contain multiple related pieces of data. If an array is originally allocated to contain 10 items, it cannot contain 11 items unless it is first re-allocated. Attempting to stuff 11 items into 10 slots results in an *array overrun,* where the 11th item isn't written to the array but rather to an adjacent area of memory that probably contained something valuable. In the context of managed code, the CLR usually catches this violation and instead returns an error, refusing to allow that 11th item to be written to memory. This code demonstrates an array overrun in C#:

```
using System;

class MainClass {
  public static void Main() {
    int[] sample = new int[10];
    int i;

    // generate an array overrun
    for(i = 0; i < 100; i = i+1)
      sample[i] = i;
  }
}
```

The array **sample** was declared with 10 elements, but the **for** loop attempts to write 100 values to it. When the 11th value is written, an exception will be generated:

```
Unhandled Exception: System.IndexOutOfRangeException: Index was outside the bounds
of the array.
   at MainClass.Main()
```

Note

Note that some programming languages (the Web programming language PHP is an example) will simply dynamically re-allocate the array rather than generate an error.

Last up is uninitialized memory—or uninitialized variables, which is a distinct but related problem. We managed-code developers often get a bit lazy because things like the CLR try to ensure that all variables have a default value—zero, null, an empty string, or whatever. In reality, the CLR is initializing the appropriate areas of memory with those "default" values because memory doesn't *have* a default value. Any given memory location effectively contains a random value until we explicitly put something into it, so *reading* a memory location before we have explicitly *written* to it is inviting disaster.

Realtime publishers

MICRO FOCUS
Leading the Evolution™

Using Figure 4.1 as an example again, suppose we read location 874 to get a pointer. We write the code so that if we get zero back, we know the pointer doesn't yet exist; if we get a non-zero value, we use it as a pointer. That's fine, *provided* we originally initialized that location with a zero in the first place—we can't rely on it to magically be zero just because our application just started. Most programmers will initialize *a* when they create it (int a = 0). If the programmer does not (int a), that simple omission opens the door for trouble.

API Failures

Last up for resource errors are what I generically call *API failures.* An API—application programming interface—is some chunk of code written by someone else, that we can rely on to perform some action for us. This might be asking the Windows OS to allocate a block on disk, or it might be asking Microsoft Office to open a Word document. APIs are essentially "black boxes," meaning we push their buttons and they go off and do whatever it is they do—and give us very little insight into what that is.

When API calls work, all's well. But when an API call fails, our code needs to recognize that the result may be more than just a simple error, especially if the API is written in non-managed code. A failed API call may leave the system in an unstable state, may not return an error at all, may corrupt memory, and so forth. It's *very* important, when making calls to external APIs, that your code be prepared for anything to happen, and that you *verify* the results of the call before relying on them.

Error Summary

With these different types of errors, we're dealing with significantly different means of detection, and different results when they're left in the code during execution. Table 4.1 summarizes some of the key characteristics of these types of errors.

Error Type	Detectable statically (without compiling the code)	Detectable by compiling the code	Detectable by running the code	Code compiles and runs without error	Code runs and behaves generally as expected	Error can be caught manually without any specialized tools
Syntax	Usually	Always (since the code will not compile)	N/A	N/A	N/A	Yes
Semantic	Sometimes	Usually	Always	Possibly	No	Yes
Logic	Not usually	Not usually	Yes, with thorough testing	Usually	Usually not	Typically yes
Memory and Resource	Almost never	Almost never	Only with special monitoring and tools	Usually	Usually	Usually not

Table 4.1: Key characteristics of common error types.

Realtime publishers

MICRO FOCUS Leading the Evolution™

The general idea here is that as you move through the taxonomy of more complex and difficult errors, it becomes more difficult to catch them statically. At some point, even fairly thorough testing won't catch the problems; instead, you'll need to use specialized monitoring and testing tools to catch specific types of problems.

Addressing Coding Errors

So how do you address coding errors? Obviously, some errors are detectable before you even compile or run your code, while others aren't detectable at "design time" and require that you execute your code.

Editing Tools

Editing tools can help catch both syntax and semantic errors, and modern editions of Visual Studio have plenty of power built right in to help address these kinds of issues. For example, a feature as simple as syntax color-coding—something most of us take for granted in a development environment—can easily help prevent syntax errors. Once you become accustomed to the "proper" color for keywords, variable names, and so forth, it's easier to avoid typos by simply looking for things that don't turn the proper color as you type. It's almost a form of automated "spell checking" that, with practice, can be very effective at avoiding simpler syntax errors. Figure 4.2 shows this in action, with a misspelled "For Each" keyword that didn't turn the blue that keywords normally turn.

```
Dim var, coll

Freach var in coll

next

For Each var In coll

Next
```

Figure 4.2: Color-coding in Visual Studio.

Visual Studio's code-hinting and code-completion features are certainly convenient, but they also help avoid syntax *and* semantic errors by reminding you of the proper code syntax and by helping to complete elements of that syntax for you. If the proper syntax is right in front of your eyes, you're more likely to get it right; if the computer is doing some of the typing for you, you're less likely to make simple typos that result in errors later. Figure 4.3 shows the code-completion feature, which tries to guess what you're typing and offers to finish it for you.

Realtime publishers

MICRO FOCUS
Leading the Evolution™

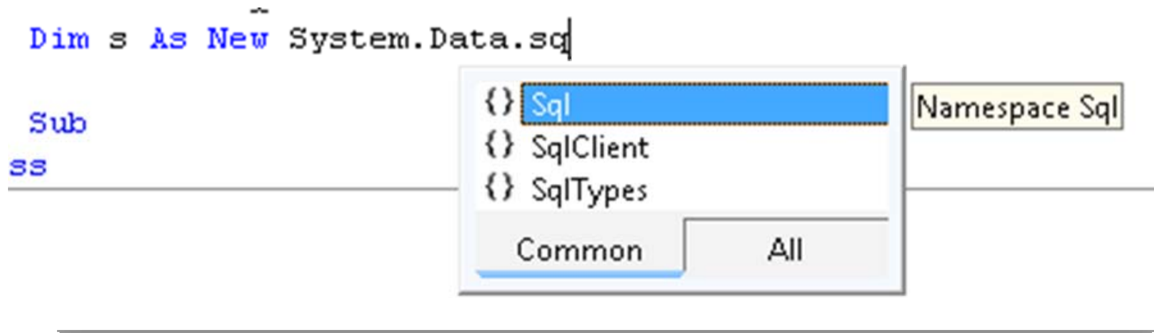

Figure 4.3: Code completion in Visual Studio.

Figure 4.4 shows the related code-hinting feature, which helps to remind you of the correct syntax and even offers multiple versions of the syntax (typically for overloaded methods). Reminding yourself to use these tools, rather than just typing manually, takes some practice—but it's worth the effort.

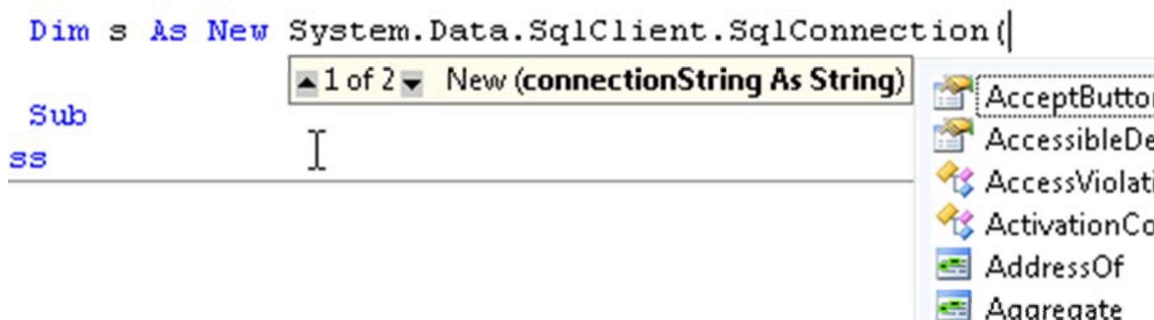

Figure 4.4: Code hinting in Visual Studio.

For several versions now, Visual Studio has offered live syntax checking. This uses a combination of static code parsing and pre-compiling through the .NET Framework CLR, and helps alert you to semantic errors by using a squiggly underline—not unlike the automated spelling- and grammar-checking in Microsoft Word. You should make the effort to fix any of these, and Visual Studio will in many cases offer pop-up help that explains the problem it sees and may even offer to automatically fix it. Figure 4.5 shows an error, along with the pop-up tool tip that described the problem Visual Studio sees.

Realtime publishers
MICRO FOCUS Leading the Evolution™

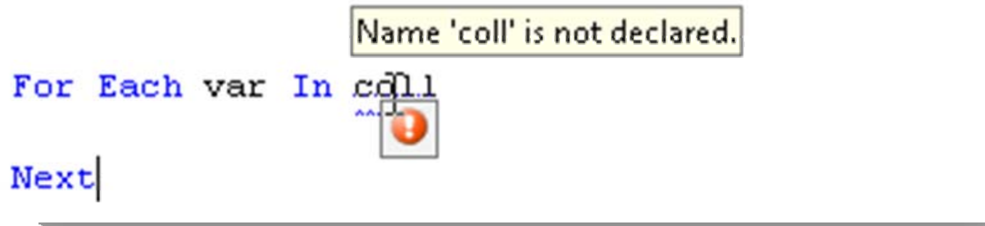

Figure 4.5: Static error detection in Visual Studio.

Figure 4.6 shows the assistance that Visual Studio offers. In this case, it's guessing that we misspelled "color," and is offering to correct the problem.

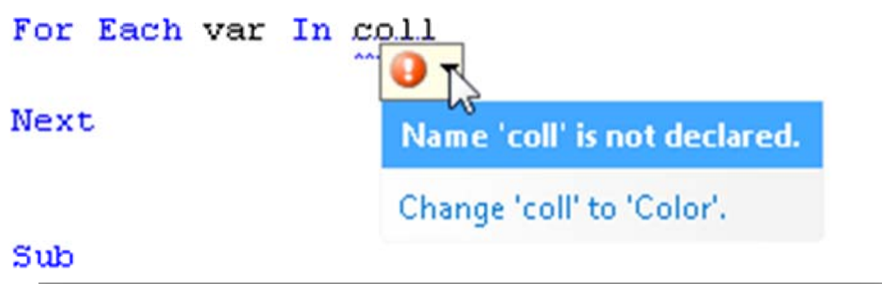

Figure 4.6: Automated error resolution in Visual Studio.

A related feature helps track potential logic errors, including the use of undefined variables, or variables that might not be assigned a value in every possible code path. Highlighted with a less-urgent green underline, you should still ensure that these issues are addressed. In many cases, you'll be fixing them as you code, if you're using good coding practices; any green underlines that remain after you think you're done should still be addressed. Figure 4.7 shows this feature, calling attention to a variable that has been used but not yet assigned a value within the current context. Underline-free code is definitely a goal to have in mind!

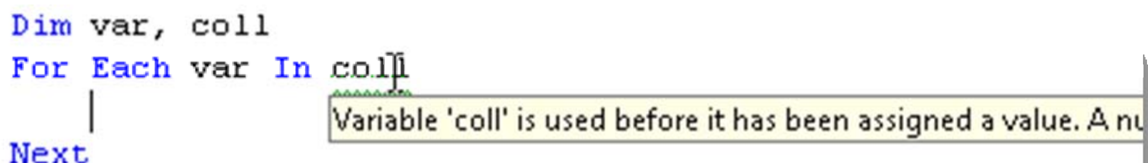

Figure 4.7: Detecting potential logic errors in Visual Studio.

Realtime publishers
MICRO FOCUS Leading the Evolution™

Third-party Visual Studio add-ins can offer even more capabilities to this list, helping Visual Studio spot increasingly-complex types of errors at "design time" and helping to improve the quality of the code before you even run it for the first time.

Compilers

Language compilers, including Visual Studio's .NET Framework compilers, are invaluable at catching syntax and semantic errors. Of course, you typically want to catch as many of those as possible as you write the code, and the features discussed previously are designed to do exactly that. But the compiler is a sort of last line of defense for many types of semantic errors. Visual Studio's is capable of not only flagging errors—that is, code that just won't work as-is—but also warnings, which indicate code that may execute but might produce logic errors at runtime. Figure 4.8 shows Visual Studio's compiler output with several errors and no warnings given.

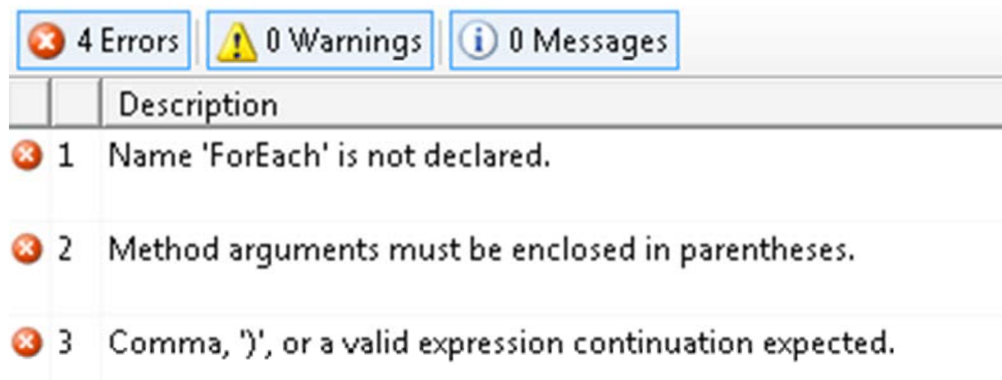

Figure 4.8: Visual Studio's compiler.

Debuggers

Visual Studio's built-in debugger is a mature and powerful tool for finding logic errors (because you've usually found most syntax and semantic errors by the time you come to the point of using this tool). Remember that I described logic errors as primarily deriving from properties and values that contain data other than what we expected; the goal of a debugger, then, is to help review the contents of those items *as the code executes* so that you can correct your assumptions. Essentially, the debugger is a way to get inside your code as it executes and examine the data your code is actually using.

Realtime publishers

MICRO FOCUS
Leading the Evolution™

Here's my debugging methodology: I sit down with the code—often a printout of it, believe it or not—and document my expectations for its execution. I make notes about what values I expect to find in certain properties or variables, and I make notes about the code paths I expect the execution to follow. Granted, a more experienced developer can do all that in their head on the fly, but the act of noting those expectations is key. Once I have my expectations, I run the debugger, and I look for the point where reality doesn't meet my expectations: That's generally the point where the bug lives, or very close to it.

There's a caution here, however: A debugger is *fundamentally useless* if you don't have a solid idea of what your code *should* be doing and a firm expectation for what data you'll see *in advance of running the app.* In other words, if you've no idea what your code should be doing or what data you should be looking at, you won't recognize the point at which things go wrong. Figure 4.9 shows an overview of Visual Studio's debugger.

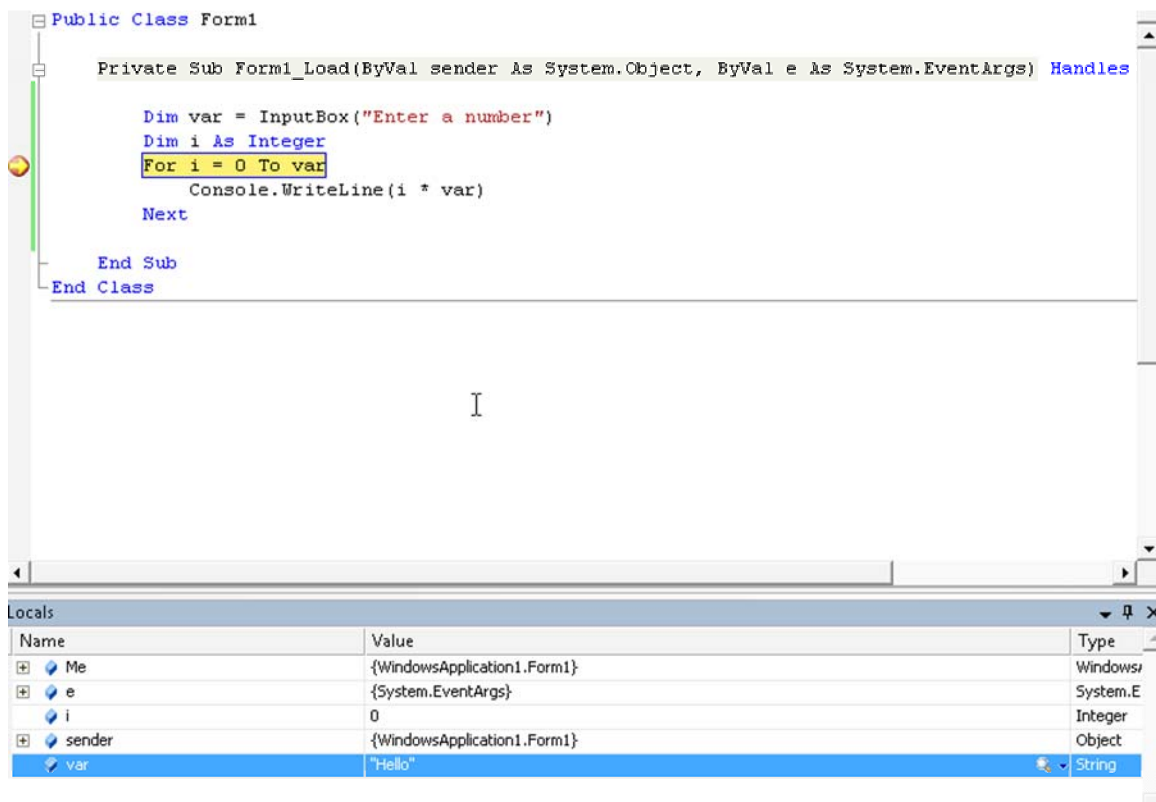

Figure 4.9: Debugging in Visual Studio.

Here's quick overview of some of the main features:

- The yellow highlight indicates the next line of code that will be executed. You can manually set the next line if you want to step back to an earlier point in the code.

  Note

  The debugger's features differ a bit between .NET Framework languages. In Visual Basic, for example, you can edit the code while execution is paused. You cannot edit code on the fly for C or C#.

- The red circle indicates a breakpoint, which you can set manually. Breakpoints cause code execution to pause, allowing you to review the current state.
- The Locals window at the bottom displays the contents of local variables. Here, you can see that var contains the string "Hello," which is a problem because the code specifically prompts for a number and expects to use the contents of var as a number. This, then, is where the bug lives, and we might choose to fix it by first validating the user's input rather than accepting it blindly, or we might declare var as a numeric data type, thus forcing an exception if the user enters something other than a number.

I realize that most developers are familiar with Visual Studio's debugger; I mention it only because I see a lot of developers struggling to use the debugger *effectively,* typically because they haven't taken a moment to articulate their expectations for the code's execution. Without those expectations, there's nothing to debug: Debugging is fundamentally about determining where your expectations and assumptions differ from the real-world conditions.

Automated Tools

Automated tools that either integrate with Visual Studio or run independently can help spot additional code problems—typically without running the code. Some of the advantages offered by these types of tool include:

- Static source code review with an emphasis on detecting common coding errors, inappropriate language usage, and an examination of calls to underlying services and frameworks.
- Error detection primarily for identifying nonstandard or poor programming practices, validation of Windows API calls, resource tracking and leak detection for unmanaged code, and tracking of transitions between unmanaged and managed code.
- Memory analysis is often running alongside your code to monitor considerations such as overall memory consumption and memory leaks

We'll look at some of these areas in more detail in upcoming sections. However, the overall message is that third-party tools can often provide greater breadth of code analysis, greater specificity for problems such as API calls and memory leaks, and deeper reviews of coding practices. By working *with* Visual Studio, these tools help developers spot code quality problems as they work without adding a great deal of additional process or overhead.

Error Handling in Code

Some errors that occur at "run time" can't be handled merely through debugging, correct syntax, or proper semantics. These errors can, however, be *anticipated* and dealt with by the code itself. Examples include things like missing files, unreachable server resources, insufficient user permissions, and so forth. It's important to understand where these types of situation-specific errors can occur so that your code can be prepared to deal with them.

Figure 4.10 illustrates the preferred way to handle errors in most .NET Framework languages—a **Try…Catch** block (each language has a specific syntax, and some languages may use alternate constructs, but the idea is the same). In the **Try** portion, you execute the code that you anticipate an error for; the **Catch** block receives the error and takes corrective action, notifies the user, and so forth.

```
Dim filename As String = InputBox("filename?")
Try
    OpenFile(filename)
Catch ex As Exception
    MsgBox("File could not be opened")
End Try
```

Figure 4.10: A Try…Catch block in Visual Basic.

This type of construct is really your last line of defense against run-time exceptions that "crash" the application: Anticipating errors and dealing with them gracefully.

Note

I've seen instances of developers wrapping entire functions in one big Try…Catch block and not putting any code in the Catch portion. That has the effect of suppressing errors, which looks nice, but is ultimately self-defeating. Errors are *useful*; you should at least *log the errors* so that they can be reviewed later. No application should generate exceptions and crash, but you should achieve that by properly handling and eliminating errors, not by simply ignoring them.

Managing the Native Code/Managed Code Boundary and Monitoring API Calls

Dealing with code errors becomes much more complex when managed code needs to execute unmanaged code or vice-versa. Visual Studio, for example, offers great tools for debugging managed code, but it can't step the debugger into any unmanaged code that your application may need to call on. The boundary where unmanaged code connects to managed code is something largely handled by the .NET Framework's CLR (through a mechanism often referred to as the *interop layer*). It is an area rife with possibilities for memory leaks and other difficult-to-catch issues simply because the CLR itself has very little control over what the unmanaged code does.

This is an area where third-party tools—even some free ones, although commercial ones tend to offer a more mature feature set—can be invaluable. By carefully tracking resources and memory, these tools can point to specific instances where memory is leaked, either by the unmanaged code component, by the CLR, or by managed code. In some cases—such as when the unmanaged code is a third-party component you're using—you may not be able to fix the problem but at least you can positively identify it and escalate it to the appropriate party.

> **Note**
> The difficulty in dealing with managed/unmanaged code connections, and your inability to correct errors in third-party unmanaged code, is why many developers try to use extensions and components written entirely in managed code. It's not that managed code is inherently better (in fact, sometimes its performance can be poorer); it's that keeping everything within the CLR reduces the opportunities for problems such as memory leaks.

API calls are a special situation. Typically, they have all the same risks and problems of any dealing between managed and unmanaged code. To that, they add the complexity of different language structures, different data structures, and in many cases, an inability to really see what's going on after you make the call. Again, third-party tools can be helpful. Numerous free and inexpensive tools can monitor API calls, trace API call execution, display input and output data, and so forth—essentially acting as a kind of "API call debugger" that supplements Visual Studio's own debugger. Some editions of Visual Studio can even do a good job of monitoring API calls that your code makes, and commercial Visual Studio add-ins can help by validating API calls (for example, making sure you're using them correctly) and monitoring the actual calls made by your code at runtime.

Case Study: Fixing a Leaky App

Memory leaks are one of the most troublesome and difficult kinds of code defect to deal with. Consider this C# code:

```
using System;
using System.Threading;

namespace MsdnMag.ThreadForker {
  class Program {
    static void Main() {
      while(true) {
        Console.WriteLine(
          "Press <ENTER> to fork another thread...");
        Console.ReadLine();
        Thread t = new Thread(new ThreadStart(ThreadProc));
        t.Start();
      }
    }

    static void ThreadProc() {
      Console.WriteLine("Thread #{0} started...",
        Thread.CurrentThread.ManagedThreadId);
      // Block until current thread terminates - i.e. wait forever
      Thread.CurrentThread.Join();
    }
  }
}
```

This code launches a thread, which displays its thread ID and then tries to Join on itself. This causes the calling thread to block, waiting on the other thread to terminate. So the thread is caught in this kind of catch-22—the thread is basically waiting for itself to terminate. Every time you press Enter in response to the prompt, the program consumes another 1MB of memory. The problem is that the thread is being dropped each time through the loop, but the CLR's garbage collector doesn't reclaim the memory that was allocated—this is actually a good thing, given how threads are used by the system, but in this case, the result is a memory leak. Tools in Visual Studio, third-party tools, and commercial Visual Studio add-ins can all display this type of memory usage. In this example, the memory usage comes from the CLR's stack, and the tools would show you that, allowing you to see *where* memory leaks were occurring, helping you tie that back to the code that was causing it, and then fix the problem.

> **Note**
> This example was adapted from a longer discussion on memory leaks at http://msdn.microsoft.com/en-us/magazine/cc163491.aspx; it's definitely worth a read as other types of leaks are also discussed, along with techniques for solving them.

Realtime publishers

MICRO FOCUS
Leading the Evolution™

Errors Fixed

This chapter offers a solid understanding of the different types of errors, and how to begin addressing them. Perhaps most importantly, this chapter highlighted that different types of errors can be avoided and handled in very different ways, so it's important to have a complete toolkit and to understand a variety of techniques.

Next up, we'll look at addressing poor-quality code that stems from something more complex than coding errors—performance and security. Often referred to as *non-functional requirements,* they're definitely trickier than most coding errors, but in many ways they create a more visible and immediate perception of quality—or lack thereof.

Download Additional eBooks from Realtime Nexus!

Realtime Nexus—The Digital Library provides world-class expert resources that IT professionals depend on to learn about the newest technologies. If you found this eBook to be informative, we encourage you to download more of our industry-leading technology eBooks and video guides at Realtime Nexus. Please visit http://nexus.realtimepublishers.com.