

PART I

Introduction to Perl Debugging



CHAPTER 1

Introduction



4 Debugging Perl: Troubleshooting for Programmers

Perl is an exceedingly practical and useful language. You can be as structured or as unstructured as you like and still achieve the same result. Perhaps its most endearing and significant achievement, though, is its use of fairly simple constructs that enable you to solve a variety of problems with only a few lines of actual code. The language is so easy to use, especially for Unix programs, that dropping into an editor to write a quick Perl program to solve a problem has become a way of life for many programmers.

Unfortunately, Perl's free and easy format means that it is easy to introduce errors into your program without realizing what you've done. Perl doesn't enforce error checking—if you decide not to check the return value of a function Perl won't complain. It's even possible to write code that shouldn't really pass the interpreter and still have your program execute, even though it probably won't work.

Identifying and removing bugs is an art. You need to know what you are looking for, and you need to know where the problem occurs. Even within a simple application the root of a problem can be located one or two function calls from where the real problem lies. There are of course different techniques to finding bugs. You can use **print** statements, the built-in debugging functions, or an external debugger. All have their merits and drawbacks.

No matter how you debug your programs, there is actually more to the debugging process than merely picking out those bugs that cause your program to display the wrong result, call the wrong procedure, or fall over completely. It's a fallacy that debugging is just about solving problems in your program after it has been written; you need to think about how the program was written in the first place.

By following some simple practices while writing the code, you can eliminate most bugs before they occur. From the outset, you should be thinking holistically about your program and designing it in such a way that errors are either trapped before the user is aware of them or handled in such a way that they don't affect the user. Basic error trapping should be a part of all your programs—even the simple ones.

You can also use *abstraction*—the division of a program into functions and modules—to split your program up. By dividing your application into functions and modules you increase its reliability and ease of maintenance and, most importantly, make it easier to trace and fix bugs when they do occur.

Design Tip *Even simple things like the use of a good programmer-friendly editor can make a difference. A good editor can help to highlight typographical errors and display your program in a way that makes it easier to understand without having to trace your finger across each component of the statement.*

Irrespective of how you debug your code and what tools and methods you use to make that process easier, an important step in the debugging process is to take a closer look at your code and what it's doing. Many programmers write a piece of code that works, rather than code that works well. For 90 percent of the time the execution does not cause a problem, but it is the 10 percent that counts. If you ask users about their experiences with a program, they are more likely to remember its failures than its successes.

Chapter 1: Introduction 5

You can use all sorts of techniques to take a closer look at your code. By employing debuggers you can follow the execution of your code in minute detail. For finding individual bugs, this approach is perfect, but when you want to improve on the execution of your code, you need to use a profiler to monitor how much time each section of your code takes. You can also employ some unexpected features of the Perl compiler to provide you with some useful background information on your script. Identifying all these different effects and solutions to those problems is what this book is about. We'll be looking at how Perl parses a statement and how easy it is to introduce a bug just from typical programming techniques. We'll also look at the methods available for tracing and tracking bugs, both in the code and in the logic and execution of the code. The final part of the book looks at methods of testing your code to ensure that you trap all of the bugs that might appear.

As an introduction, let's look at the basic bug types and briefly go over some of the general issues that you need to address when debugging your code.

Bug Types

You can think about the process of program development as a sort of hierarchical tree. At the top are the input values and at the bottom is the output value, or result of the program. It is a sad fact that a lot of application development is treated from the bottom up, rather than the top down. The *top-down* approach forces the developer into thinking about all of the steps to reach the result. The *bottom-up* approach means trying different techniques until you reach a result that works from the top to the bottom.

The bottom-up approach is slower, but often more practical from a programmer's point of view because it helps to gel ideas and form methods and processes in the programmer's mind that will ultimately be reflected in the final application.

The approach chosen has some effect on the bugs that you can introduce into the program. With the bottom-up approach you do not consider all of the factors before developing a new component and are, therefore, likely to introduce more errors than with the more pragmatic, top-down approach.

A mistake often made by developers is to ignore the effects of a bug or to fail to classify the bug and its effect. Although not normally seen as a major problem—a bug is a bug—it can affect the way bug problems are solved (which may have “knock on” effects to other bugs) and can also cause other bugs to be ignored entirely.

Personally, I've always considered that there are essentially three types of bugs when programming: the typographical, the logic, and the execution bugs.

The Typographical Bug

The *typographical bug* is one introduced through a typographical error, either through bad typing or a momentary lapse in concentration that causes you to forget the semantics of the language. In Perl this can be as easy as using \$ instead of @ when referring to a variable, or forgetting to place parentheses around a statement to ensure list, rather than scalar, context on a function.

6 Debugging Perl: Troubleshooting for Programmers

Some of these errors will of course be picked up by the Perl interpreter during the compilation process, as they will break the basic rules of the parsing engine for Perl. Other errors will be compiled okay, and they might even pass the usual warning and **strict** pragmas without raising any sort of indication of the problem. As a typical example, look at this code:

```
$string = "The cat sat on the mat";  
$animal = ($string =~ m/The (.*) sat/);  
print $animal;
```

If you knew that there was a bug in there, what would you do? Reading it probably doesn't highlight the problem immediately, even to the hardened programmers. Asking Perl to check code through the parser with warnings switched on will not raise any errors. Run the script though and the error is obvious—we get a value of 1, instead of the expected “cat” response.

Other typographical bugs can be more obscure. On a recent web project the login process suddenly stopped working. The login system worked by checking the user name and password against a database, and, assuming both were approved, the script would generate a unique session ID, which was also placed into the database. The problem was that the ID would be approved and the session ID created, but the user couldn't connect using the new session ID.

It took a little time, but by comparing the session ID returned to the user and the session ID in the database it was obvious that the 27-digit session ID would exceed the 26-digit field width. For 50 percent of the time the session ID system worked fine, for the other 50 percent the session ID would exceed the field width and fail. This is another type of typographical error: entering the wrong width into the database creation script.

You can generally resolve typographical bugs through a combination of careful code reading and a sound understanding of how Perl parses statements. We'll be looking at the specifics of how Perl approaches the problem of parsing individual statements in Chapter 2.

The Logic Bug

Logic bugs occur when the programmer has failed to spot a flaw in the logical flow of the program. Like typographical bugs, these can be very obvious—for example, failing to identify one of the possible return values from a function or incorrectly specifying a test or other operation. These are not typographical errors—the programmer really thought he or she was doing the right thing at the time.

As an example, and as an extension of the typographical error earlier, the same login system suddenly started returning bad logins after some optimization of the login code. The login function returned a negative number on error, a zero if the session or login had expired, and a positive number on success. After tracing the bug for hours it finally became apparent that the problem was because the **if** statement checking the return value tested for a strict zero response as failure, with any other response taken as valid.

Another example is the use of references in your code. When using a reference you must *dereference* the variable before the variable contents can be accessed. For example, when accessing a reference to a hash, you don't use

```
$hash{$key}
```

but instead use

```
$hash->{$key}
```

This is not a typographical error, but a logic error; the programmer completely ignored the fact that the access was to a reference to a hash, not a real hash.

Logic bugs are generally easier to identify than typographical errors. With typographical errors you can generally track the problem to a possible location, but without aid from the Perl parser it can be hard to see the woods for the trees. With logic bugs, you can trace and track the logic and execution sequence until you find the specific line that causes the problem.

Logic bugs can be approached from a number of directions. The most obvious is for the programmer to understand the logic of the Perl interpreter, from the basics of variables and how to access them to the uses of operators, functions, and regular expressions. Logic bugs are all related to how you access and use the information stored within your script—if you can understand and identify how Perl approaches the problem, tracing the problem should be straightforward. Chapters 3 and 4 concentrate on how to identify logic bugs in your programs.

Other ways of tracing logic bugs include the use of abstraction—turning components of your program into subroutines, modules, packages, even objects. By separating out the individual elements, you can eliminate the source of the problem section by section. Abstraction also makes your code more portable and more manageable. We'll look at abstraction in Chapter 5, and we'll see how it can improve the portability of your code in Chapter 6.

The Execution Bug

The *execution bug* is one of the hardest to find, and there are many who probably do not consider it to be a bug at all. The execution bug is one that affects the execution process of a program; not because of a typo or a logic problem, but because the application has been designed in a way that allows a certain chain of events to slow or halt the application's execution.

Execution bugs do slow your application but might not actually generate a bad result or manifest themselves in any other way. For example, imagine a database application that pulls records out of a database. A typical search returns 200 records, and the script works within a few seconds, well within the limits you might expect for extracting information from the database. When a search returns 1,000 records, however, the script takes almost a minute to execute.

The problem is not logic—the script works and behaves exactly as it should—and it's not a typographical error—the information is formatted correctly and there really are 1,000 records in the database to be displayed. So how come it takes so long?

The problem here is one of execution—either the programmer needs to introduce a limiter to reduce the effect of accessing and displaying that many records, or there's a portion of the application that takes an excessive amount of time to parse the information. This is an execution bug—there's nothing wrong with the application, it just doesn't execute at the level you would expect.

Execution bugs are difficult to trace, but once found and resolved they have some benefits. An application that has been cleansed of its execution bugs will be more robust than a normal application, and it's likely to be more stable in use. Because execution bugs frequently affect the performance, eliminating those bugs will help to optimize your application.

8 Debugging Perl: Troubleshooting for Programmers

The easiest way to find an execution bug is to execute your program, but to isolate the location of that bug you'll need some tools and techniques. The last sections of the book—from Chapter 7 through 14—will look at bug trapping and application optimization and testing.

Basic Perl Debugging Rules

Whatever tricks or methods you use to try to trace and track the bugs in your code, there are some fairly basic rules that you should always follow, irrespective of your approach.

- **Always check the obvious**—If you were to give the average programmer a piece of code to debug, he or she would almost certainly start looking in the wrong place. Some of the examples given earlier in this chapter should demonstrate that it's often the obvious things that cause the most difficulty.

- **Always check from the outside in**—You should always follow the execution path from the outside of the code to the inside. Unless you've got an exact reference of where the bug occurs, you should look at the function in which the bug appears and then trace the bug through from that function to other nested functions until you find the root.

Tracing from the inside out causes problems because you don't know what other processes have occurred that might have led to the bug. For example, a function that returns the wrong value may have been supplied the wrong value—checking the function and finding nothing wrong is a pointless exercise.

- **Always start from the top when dealing with a typographical error**—You should start looking at the code from the top and work down; don't work backward. Typographical errors have a nasty habit of causing run-on effects. In order to demonstrate this, try placing an additional opening parenthesis into a script—Perl will return all sorts of errors that have nothing to do with actual problem.
- **Always start from the bottom when solving parser errors**—Perl outputs a probable line number for the problem. If you work from the top and upset the line numbering you'll need to re-parse the script to identify the new line numbers. If you work from the bottom your line changes will not affect the earlier line numbers, and you can solve many more bugs before you need to re-parse the script.
- **Always switch on warnings**—Warnings give you much more information about the possible bugs that you may have overlooked than just allowing the parser to work normally. Warnings highlight potential problems, such as unused variables (which highlight typos), misused variable types (which highlight logic errors), and code that might never be reached.

- **Always work with the strict pragma**—The **strict** pragma enforces many Perl parsing rules to a much higher standard than normal. It causes Perl to check the syntax of certain operations—variables, references and subroutines—to a much greater degree than normal and, like warnings, can highlight problems during the parsing process.

We'll look at many of these issues in closer detail elsewhere in this guide.

Bug Prevention

It should be obvious that trying to prevent the introduction of bugs before they occur is the best way to approach the problem—prevention is always better than cure. The problem is that prevention is sometimes more time-consuming than the curing process and gives the appearance of slowing down the development process. In truth, by preventing the bugs instead of curing them, you improve the chances of making the deadlines, because you'll be spending less time searching for them after the event.

But how do you prevent bugs from occurring in the first place? Many of the techniques are beyond the scope of this book because they rely on the fundamental approach toward the programming goal rather than Perl language, but they are worth mentioning.

Program Design

When developing an application, having a good idea about what it is going to produce is vital, and having a good idea about how to achieve that goal is a good way of ensuring that bugs are not introduced into the software. Even just writing down a quick list of functions and what you expect them to do is better than making it up as you go along.

It's also worth coming up with a system or style and sticking to it. When programming a database, for example, you could exchange information by supplying an array, or a reference to a hash, or an object. Using all three is bound to lead to problems; first when you try to remember what a particular function does, and second when you try to convert between the two methods.

At the other end of the equation are programs that are designed and developed to the *nth* degree from start to finish and end up with design manuals that run to more lines than code you are producing. Program design to this level has its place—I would hope that Nuclear Power station software and the fly-by-wire computers built into most modern planes use this. On the other hand, developing this kind of documentation for quick script to convert data file formats is wholly unnecessary.

10 Debugging Perl: Troubleshooting for Programmers

Editors

A good editor should be a vital component of any programmer's armory. Most modern editors will match parentheses and help you indent your code to make it more readable. Emacs, probably the most popular programmer's editor, does this automatically and almost enforces the option for scripts that it identifies as Perl-based. More advanced editors, such as BBEdit (for Mac), EditPlus (for Windows), and the modified xemacs (for Unix), can also color the code according to the individual language.

Using an editor that performs these simple checks will help to eliminate many of the simple typographic errors that cause many bugs and compilation failures. They can also provide visual cues for identifying problems.

Formatting

Choose a formatting standard and stick to it. You don't have to look at much of the code written by programmers to see that, in general, it's clean and tidy and easy to read. Perl helps this process along, and using a good editor will also help, but there's still some flexibility in choosing your own style. Don't mix and match styles, as it will be more difficult for you to identify bugs when you do go back in to read your code.

For example, when you create a new code block there are two basic formats:

```
if ($condition) {  
  ...  
} else {  
  ...  
}
```

or

```
if ($condition)  
{  
  ...  
}  
else  
{  
  ...  
}
```

The former is shorter and more compact, but it can become fussy when you introduce **elsif** statements. The second style is longer, but generally easier to follow because you can match up braces vertically within the code.

Larry Wall, the original instigator of Perl, suggests the following guidelines:

- Four-column indent
- Opening curly on same line as keyword, if possible; otherwise line up
- Space before the opening curly of a multiline BLOCK
- One-line BLOCK may be put on one line, including curlies
- No space before the semicolon
- Semicolon omitted in “short” one-line BLOCK
- Space around most operators
- Space around a “complex” subscript (inside brackets)
- Blank lines between chunks that do different things
- Uncuddled **elses**
- No space between function name and its opening parenthesis
- Space after each comma
- Long lines broken after an operator (except **and** and **or**)
- Space after last parenthesis matching on current line
- Line up corresponding items vertically
- Omit redundant punctuation as long as clarity doesn’t suffer

There are also other style issues you should consider; check the **perlstyle** man page for information

Comments

Have you ever read a piece of code just a few weeks after you’ve written it and wondered exactly what it was you were doing?

Imagine trying to debug that same bit of code—it might work okay, but without a comment to that effect there is no way for the programmer to know for certain that the technique achieves the desired result. Tools such as bug-tracking systems and code revision monitors will help; we’ll take a quick look at those later in this chapter. If it doesn’t work correctly, you might still need a quick guide as to how the function or code fragment works so that you can follow the execution when it comes to debugging the code.

Some other issues to keep in mind when writing comments:

- Don’t repeat what the line does—saying “while loop” on a line that contains the **while** keyword is not helpful. Say “iterate through the source file” or something similar.
- Don’t list the arguments—a good programmer will be able to spot what arguments are being extracted.

12 Debugging Perl: Troubleshooting for Programmers

- Do list the types of input variable (for example, classify array or reference to array).
- Do specify the return values and expectations of when they should be returned.
- Do document the error/failure conditions and return values.

Remember that comments are there to jog your memory and highlight the execution sequence to other programmers, not to provide an English language running commentary or vent for your frustrations. We'll look in more detail at comments and documentation in Chapter 5.

Code Revisions

The Revision Code System (RCS) and Concurrent Versioning System (CVS) both provide ways for you to track the differences between the versions of the your code. The general mode of operation is to write a piece of code and have it working, but perhaps not necessarily debugged, and then “check in” the revision to the revision system. When you make changes to the code and it fails, you can then always go back to a previous version.

24x7 Bug Tracking

Finding bugs is one thing, but remembering where they all are is another matter. At the very least you should be keeping a textual record of the bugs that have occurred and whether they have been fixed. Including information about what the bug was and how it was fixed is also a good idea, because doing so might help you to identify and trace subsequent bugs.

When you discover a bug you should be recording:

- Location—give the line number and file name if known or, at the very least, a possible function culprit.
- Description of the bug that occurs, including what you expect to happen and what actually did happen.
- Date and time the bug was discovered and name of user who reported the bug.
- Any other relevant information, such as the platform, environment variables, and any factors that may have led to the bug.

If you want to be more professional about your bug tracking, consider using tracking software such as BugZilla, which is part of the Mozilla project. The system is actually written in Perl and is available as a Web interface to an underlying MySQL database. You can download BugZilla from <http://bugzilla.mozilla.org>.

Chapter 1: Introduction 13

RCS (which is actually used by CVS) allows you to store comments with each revision and to automatically track version numbers of each file that you check in. You should use those opportunities to give a status report, both on what you achieved in the new revision and an indication of the bugs or issues that you addressed. The version numbers can be useful when tracking bugs as well as when receiving bug reports from your end users—it's quite possible that the bug they are reporting has already been fixed in a new revision.

The difference between CVS and RCS is that RCS allows only a single user to be editing an individual source file at any one time, whereas CVS allows for true concurrent development by a number of programmers. CVS also includes additional features, such as the capability to export its tree over the Internet and the ability to automatically create a new package based on the “current” version numbers of the source files, even if they are still in production.

If you are the only programmer, RCS will more than likely serve your needs. You can get more information and download RCS from <http://www.gnu.org/software/rcs/rcs.html>. If you think you need CVS, check out the CVS Bubbles page at <http://www.loria.fr/~molli/cvs-index.html>.

