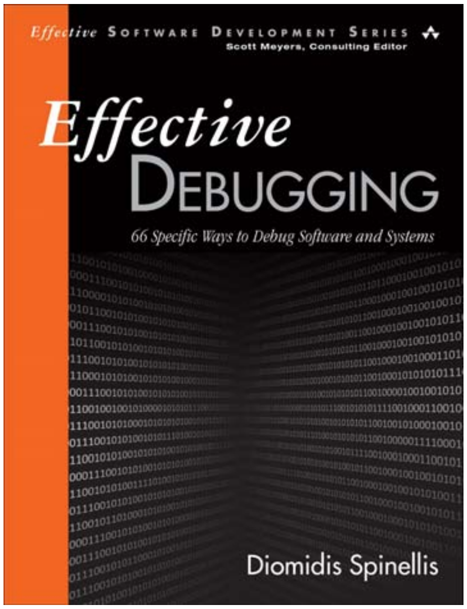
66 Specific Ways to Debug Software & Systems



Effective Software Development Series
Scott Meyers, Consulting Editor

Effective Debugging

66 Specific Ways to Debug Software and Systems

Diomidis Spinellis

256 Pages | Paperback

Publication: June 2016

ORDER & SAVE

SAVE 35% WHEN YOU ORDER

from informit.com and enter the discount code **INFOQ** during checkout

FREE US SHIPPING on print books

Major eBook formats

Only InformIT offers PDF, EPUB, & MOBI together for one price

OTHER AVAILABILITY

Through Safari Flow subscription service

Booksellers and online retailers including Amazon/Kindle store and Barnes and Noble/bn.com

Every software developer and IT professional understands the crucial importance of effective debugging. Often, debugging consumes most of a developer's workday, and mastering the required techniques and skills can take a lifetime. In *Effective Debugging*, Diomidis Spinellis helps experienced programmers accelerate their journey to mastery, by systematically categorizing, explaining, and illustrating the most useful debugging methods, strategies, techniques, and tools.

Drawing on more than thirty-five years of experience, Spinellis expands your arsenal of debugging techniques, helping you choose the best approaches for each challenge. He presents vendor-neutral, example-rich advice on general principles, high-level strategies, concrete techniques, high-efficiency tools, creative tricks, and the behavioral traits associated with effective debugging.

Spinellis's 66 expert techniques address every facet of debugging and are illustrated with step-by-step instructions and actual code. He addresses the full spectrum of problems that can arise in modern software systems, especially problems caused by complex interactions among components and services running on hosts scattered around the planet. Whether you're debugging isolated runtime errors or catastrophic enterprise system failures, this guide will help you get the job done—more quickly, and with less pain.

Key features include

- High-level strategies and methods for addressing diverse software failures
- Specific techniques to apply when programming, compiling, and running code
- Better ways to make the most of your debugger
- General-purpose skills and tools worth investing in
- Advanced ideas and techniques for escaping dead-ends and the maze of complexity
- Advice for making programs easier to debug
- Specialized approaches for debugging multithreaded, asynchronous, and embedded code
- Bug avoidance through improved software design, construction, and management

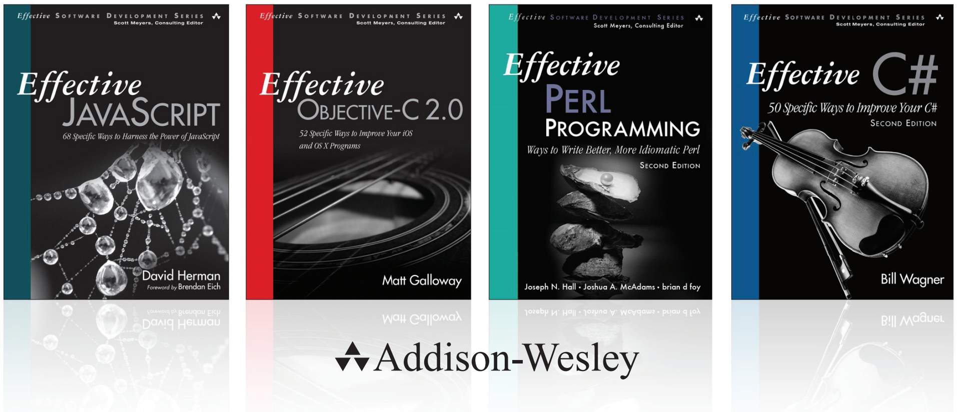
Pearson

informIT®
the trusted technology learning source

Effective Debugging

The Effective Software Development Series

Scott Meyers, Consulting Editor


Effective Software Development Series
Scott Meyers, Consulting Editor
Effective JavaScript
68 Specific Ways to Harness the Power of JavaScript
David Herman
Foreword by Brendan Eich
Effective Objective-C 2.0
52 Specific Ways to Improve Your iOS and OS X Programs
Matt Galloway
Effective Perl Programming
Ways to Write Better, More Idiomatic Perl
Second Edition
Joseph N. Hall • Joshua A. McAdams • brian d foy
Effective C#
50 Specific Ways to Improve Your C#
Second Edition
Bill Wagner
Addison-Wesley

Visit **informit.com/esds** for a complete list of available publications.

The Effective Software Development Series provides expert advice on all aspects of modern software development. Titles in the series are well written, technically sound, and of lasting value. Each describes the critical things experts always do — or always avoid — to produce outstanding software.

Scott Meyers, author of the best-selling books *Effective C++* (now in its third edition), *More Effective C++*, and *Effective STL* (all available in both print and electronic versions), conceived of the series and acts as its consulting editor. Authors in the series work with Meyers to create essential reading in a format that is familiar and accessible for software developers of every stripe.

Effective Debugging

66 SPECIFIC WAYS TO DEBUG SOFTWARE AND SYSTEMS

Diomidis Spinellis

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2016937082

ISBN-13: 978-0-13-439479-4
ISBN-10: 0-13-439479-8

Text printed in the United States at RR Donnelley in Crawfordsville, Indiana.
First printing, July 2016

To my mentors, past and future

Contents

Figures

Listings

Preface

When you develop software or administer systems that run it, you often face failures. These can range from a compiler error in your code, which you can fix in seconds, to downtime in a large-scale system, which costs your company millions (pick your currency) each hour. In both cases, as an effective professional, you’ll need to be able to quickly identify and fix the underlying fault. This is what debugging is all about, and that is this book’s topic.

The book is aimed at experienced developers. It is not introductory, in the sense that it expects you to be able to understand small code examples in diverse programming languages and use advanced GUI and command-line-based programming tools. On the other hand, the debugging techniques included in the book are described in detail, for I’ve seen that even experienced developers who are experts on some methods may well need some hand-holding on others. Furthermore, if you’ve debugged problems on non-toy software for at least a few months, you’ll find it easier to appreciate the context behind some of the book’s more advanced items.

What This Book Covers

Debugging, as treated in this book, encompasses strategies, tools, and methods you can use to deal with the whole spectrum of problems that can arise when developing and operating a modern, sophisticated computing system. In the past, debugging mainly referred to detecting and fixing a program’s faults; however, nowadays a program rarely works in isolation. Even the smallest program will link (often dynamically) to external libraries. More complex ones can run under an application server, call web services, use relational and NoSQL databases, obtain data from a directory server, run external programs, utilize other middleware, and incorporate numerous third-party packages. The operation of complete systems and services depends on the failure-free functioning of many in-house-developed and third-party components running on hosts that may span the whole planet. DevOps, the software development discipline that addresses this reality, emphasizes the roles of both developers and other IT professionals. This book aims to equip you for a similarly holistic view when facing failures, because in the most challenging problems you'll rarely be able to immediately identify the software component that's the culprit.

The material progresses from general topics to more specific ones. It starts with strategies (Chapter 1), methods (Chapter 2), and tools and techniques (Chapter 3) that can help you debug diverse software and systems failures. It then covers techniques that you apply at specific stages of your debugging work: when you use a debugger (Chapter 4), when you program (Chapter 5), when you compile the software (Chapter 6), and when you run the system (Chapter 7). A separate chapter (Chapter 8) focuses on the specialized tools and techniques you can use to hunt down those pesky bugs that are associated with multi-threaded and concurrent code.

How to Use This Book

Read the left page, read the right page, flip the right page, until you reach the end. Wait! Actually, there's a better way. The advice contained in this book can be divided into three categories.

- **Strategies and methods** you should know and practice when you face a failure. These are described in Chapter 1: "High-Level Strategies" and Chapter 2: "General-Purpose Methods and Practices." In addition, many techniques included in Chapter 5: "Programming Techniques" also fall in this category. Read and understand these items, so that applying them gradually becomes a habit. While debugging, systematically reflect on the method you're using. When you reach a dead end, knowing the avenue you've explored will help you identify other ways to get out of the maze.

- **Skills and tools** you can invest in. These are mainly covered in Chapter 3: "General-Purpose Tools and Techniques" but also include elements that apply to the problems you face on an everyday basis; for example, see Item 36: "Tune Your Debugging Tools." Find time to learn and gradually apply in practice what these items describe. This may mean abandoning the comfort of using tools you're familiar with in order to conquer the steep learning curve of more advanced ones. It may be painful in the beginning, but in the long run this is what will distinguish you as a master of your craft.

- **Ideas for techniques** to apply when things get tough. These are not things you'll be using regularly, but they can save your day (or at least a few hours) when you encounter an unfathomable problem. For instance, if you can't understand why your C and C++ code fail to compile, see Item 50: "Examine Generated Code." Quickly go through these items to be aware of them as options. Study them carefully when the time comes to apply them.

How to Live Your Life

Although all items in this book offer advice for diagnosing failures and debugging existing faults, you can also apply many of them to minimize the number of bugs you encounter and make your life easier when one crops up. Rigorous debugging and software development practices feed on each other in a virtuous circle. This advice covers your (current or future) roles in software construction, software design, and software management.

When **designing** software, do all of the following:

- Use the highest-level mechanisms suitable for its role (Item 47: "Consider Rewriting the Suspect Code in Another Language" and Item 66:"Consider Rewriting the Code Using Higher-Level Abstractions")
- Offer a debugging mode (Item 6: "Use the Software's Debugging Facilities" and Item 40: "Add Debugging Functionality")
- Provide mechanisms to monitor and log the system's operation (Item 27: "Use Monitoring Tools on Systems Composed of Independent Processes," Item 41: "Add Logging Statements," and Item 56: "Examine Application Log Files")
- Include an option to script components with Unix command-line tools (Item 22:"Analyze Debug Data with Unix Command-Line Tools")
- Make internal errors lead to visible failures rather than to instability (Item 55: "Fail Fast")
- Provide a method to obtain postmortem memory dumps (Item 35: "Know How to Work with Core Dumps" and Item 60: "Analyze Deadlocks with Postmortem Debugging")
- Minimize the sources and extent of nondeterminism in the software's execution (Item 63: "Isolate and Remove Nondeterminism")

When **constructing** software, take the following steps:

- Obtain feedback from your colleagues (Item 39: "Go Over Your Code and Reasoning with a Colleague")
- Create a unit test for each routine you write (Item 42: "Use Unit Tests")
- Use assertions to verify your assumptions and the code's correct functioning (Item 43: "Use Assertions")
- Strive to write maintainable code—code that is readable, stable, and easy to analyze and change (Item 46: "Simplify the Suspect Code" and Item 48: "Improve the Suspect Code's Readability and Structure")
- Avoid sources of nondeterminism in your builds (Item 52: "Configure Deterministic Builds and Executions")

Finally, when **managing** software development and operations (either a team or your own process), do the following:

- Have issues recorded and followed through a suitable system (Item 1: "Handle All Problems through an Issue-Tracking System")
- Triage and prioritize the issues you work on (Item 8: "Focus Your Work on the Most Important Problems")
- Have software changes properly recorded in a well-maintained revision management system (Item 26: "Hunt the Causes and History of Bugs with the Revision Control System")
- Deploy software in a gradual fashion, allowing the old version to be compared with the new one (Item 5: "Find the Difference between a Known Good System and a Failing One")
- Strive for diversity in the tools you use and the environments you deploy (Item 7: "Diversify Your Build and Execution Environment")
- Update tools and libraries on a regular basis (Item 14: "Consider Updating Your Software")
- Purchase source code for any third-party libraries you use (Item 15: "Consult Third-Party Source Code for Insights on Its Use") and buy the sophisticated tools needed to pin down elusive faults (Item 51: "Use Static Program Analysis"; Item 59: "Use Dynamic Program Analysis Tools"; Item 62: "Uncover Deadlocks and Race Conditions with Specialized Tools"; Item 64: "Investigate Scalability

Issues by Looking at Contention"; Item 65: "Locate False Sharing by Using Performance Counters")

- Supply any specialized kit required for debugging hardware interfaces and embedded systems (Item 16: "Use Specialized Monitoring and Test Equipment")
- Enable developers to debug software remotely (Item 18: "Enable the Debugging of Unwieldy Systems from Your Desk")
- Provide sufficient CPU and disk resources for demanding troubleshooting tasks (Item 19: "Automate Debugging Tasks")
- Encourage collaboration between developers through practices such as code reviews and mentoring (Item 39: "Go Over Your Code and Reasoning with a Colleague")
- Promote the adoption of test-driven development (Item 42: "Use Unit Tests")
- Include in the software's build performance profiling, static analysis, and dynamic analysis, while maintaining a fast, lean, and mean build and test cycle (Item 57: "Profile the Operation of Systems and Processes"; Item 51: "Use Static Program Analysis"; Item 59: "Use Dynamic Program Analysis Tools" and Item 53: "Configure the Use of Debugging Libraries and Checks"; Item 11: "Minimize the Turnaround Time from Your Changes to Their Result")

A Few Notes on Terminology

In this book, I use the term *fault* according to the following definition, which appears in ISO-24765-2010 (*Systems and software engineering—Vocabulary*): "an incorrect step, process, or data definition in a computer program." This is also often called a *defect*. In everyday language, this is what we call a *bug*. Similarly, I use the term *failure* according to the following definition from the same standard: "an event in which a system or system component does not perform a required function within specified limits." A failure can be a program that crashes, freezes, or gives a wrong result. Thus, a *failure* may be produced when a *fault* is encountered or used. Confusingly, sometimes the terms *fault* and *defect* are also used to refer to failures, something that the ISO standard acknowledges. In this book I maintain the distinction I describe here. However, to avoid having the text read like a legal document when the meaning is clear from the context, I often use the word "problem" to refer to either faults (as in "a problem in your code") or failures (as in "a reproducible problem").

Nowadays the shell, libraries, and tools of the Unix operating system are available on many platforms. I use the term *Unix* to refer to any system following the principles and APIs of Unix, including Apple's *Mac OS X*, the various distributions of *GNU/Linux* (e.g., Arch Linux, CentOS, Debian, Fedora, openSUSE, Red Hat Enterprise Linux, Slackware, and Ubuntu), direct Unix descendants (e.g., *AIX*, *HP-UX*, *Solaris*), the various BSD derivatives (e.g., *FreeBSD*, *OpenBSD*, *NetBSD*), and *Cygwin* running on *Windows*.

Similarly, when I write C++, Java, or Python, I assume a reasonably modern version of the language. I've tried to eschew examples that depend on exotic or cutting-edge features.

In the text I write "your code" and "your software" to refer to the code you're debugging and the software you're working on. This is both shorter and also implies a sense of ownership, which is always important when we develop software.

I use the term *routine* for callable units of code such as member functions, methods, functions, procedures, and subroutines.

I use the terms *Visual Studio* and *Windows* to refer to the corresponding Microsoft products.

I use the terms *revision control system* and *version control system* to refer to tools such as Git that are used for software configuration management.

Typographical and Other Conventions

- Surprise! Code is written in so-called `typewriter font`, key points are set in **bold**, and terms and tool names are set in *italics*.

- Listings of interactive sessions use colors to distinguish the prompt, the user input, and the resultant output.

  ```
  $ echo hello, world
  hello world
  ```

- Unix command-line options appear like `--this` or as their single-letter equivalent (e.g., `-t`). Corresponding Windows tool options appear like `/this`.

- Key presses are set as follows: Shift–F11.

- File paths are set as follows: /etc/motd.

- Links to the web are formatted *like this*. If you’re reading this on paper, you can find the URL listed in the “Web Resources” appendix. If you’re reading this as an e-book, you know what to do.
- Menu navigation appears as follows: *Debug – New Breakpoint – Break at Function*.
- In the interests of brevity, in C++ code listings I omit the `std::` qualifiers of the `std` namespace.
- When describing GUI tools, I refer to the functionality in the form it’s available in the most recent tool at the time of writing. If you’re using a different version, look at related menus or windows, or consult the documentation on how to access this functionality. Interestingly, command-line tools retain their interface stable for decades, but GUIs move things around on every new version. The many conclusions that can be reached from this observation are left as an exercise to the reader.

Where to Get the Code and Errata

The code appearing in the book’s examples and fixes associated with the text are available through the book’s web site at www.spinellis.gr/debugging.

Register your copy of Effective Debugging at informit.com for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780134394794) and click Submit. Once the process is complete, you will find any available bonus content under “Registered Products.”

Acknowledgments

First, I want to thank the book’s editor at Addison-Wesley, Trina Fletcher MacDonald, and the series’ editor, Scott Meyers, for their expert guidance and management of the book’s development. I’m also grateful to the book’s technical reviewers, Dimitris Andreadis, Kevlin Henney, John Pagonis, and George Thiruvathukal. They provided literally hundreds of top-notch ideas, comments, and suggestions that considerably improved the book. Special thanks to the book’s copy editor, Stephanie Geels, for her eagle eyes and featherlight touch. Thanks to the quality of her work, I ended up enjoying a process that I used to dread. My thanks go also to Melissa Panagos for her amazingly effective production management, Julie Nahil who supervised overall production, LaTeX magician Lori Hughes for the book’s composition, Sheri Replin for her editing advice, Olivia Basegio for managing the book’s technical review board, Chuti Prasertsith for the brilliant book cover, and Stephane Nakib for her guidance on marketing. I’m grateful to Alfredo Benso, Georgios Gousios, and Panagiotis Louridas, who provided early guidance on the book’s concept.

Four items are expanded from material I have published in the *IEEE Software* “Tools of the Trade” column.

- Item 5: “Find the Difference between a Known Good System and a Failing One”—“Differential Debugging,” vol. 30, no. 5, 2013, pp. 19–21.
- Item 22: “Analyze Debug Data with Unix Command-Line Tools”—“Working with Unix Tools,” vol. 22, no. 6, 2005, pp. 9–11.
- Item 58: “Trace the Code’s Execution”—“I Spy,” vol. 24, no. 2, 2007, pp. 16–17.
- Item 66: “Consider Rewriting the Code Using Higher-Level Abstractions”—“Faking It,” vol. 28, no. 5, 2011, pp. 96, 95.

Furthermore,

- Item 63: "Isolate and Remove Nondeterminism" is based on ideas presented by Martin Fowler in his articles "Eradicating Non-Determinism in Tests" (April 14, 2011) and "TestDouble" (January 17, 2006).
- Most of the refactorings suggested in Item 48: "Improve the Suspect Code's Readability and Structure" are derived from Martin Fowler's *Refactoring* book (Addison-Wesley, 1999).
- The article "Real-World Concurrency" (Bryan Cantrill and Jeff Bonwick, *ACM Queue*, October 2008) prompted me to write Item 60: "Analyze Deadlocks with Postmortem Debugging."
- The Java code in Item 66: "Consider Rewriting the Code Using Higher-Level Abstractions" is based on input provided by Tagir Valeev.

A number of colleagues at the Athens University of Economics and Business have (sometimes unknowingly) contributed to the realization of this project through their gracious support in many aspects of my academic life. These include Damianos Chatziantoniou, Georgios Doukidis, Konstantine Gatsios, George Giaglis, Emmanouil Giakoumakis, Dimitris Gritzalis, George Lekakos, Panagiotis Louridas, Katerina Paramatari, Nancy Pouloudi, Angeliki Poulymenakou, Georgios Siomkos, Spyros Spyrou, and Christos Tarantilis.

Debugging is a craft, which you learn by doing. I'd therefore like to thank the coworkers and colleagues who over the past four decades have endured my bugs, have supplied me with helpful issue reports, have reviewed and tested my code, and have taught me how to avoid, track down, and fix problems. In roughly reverse chronological order, in my employment and collaboration history, these are the following:

- In the Ads SRE FE team at Google: Mark Bean, Carl Crous, Alexandru-Nicolae Dimitriu, Fede Heinz, Lex Holt, Thomas Hunger, Thomas Koeppe, Jonathan Lange, David Leadbeater, Anthony Lenton, Sven Marnach, Lino Mastrodomenico, Trevor Mattson-Hamilton, Philip Mulcahy, Wolfram Pfeiffer, Martin Stjernholm, Stuart Taylor, Stephen Thorne, Steven Thurgood, and Nicola Worthington.
- At CQS: Theodoros Evgeniou, Vaggelis Kapartzianis, and Nick Nassuphis.

- In the Department of Management Science and Technology at the Athens University of Economics and Business, current and former research and lab associates: Achilleas Anagnostopoulos, Stefanos Androutsellis-Theotokis, Konstantinos Chorianopoulos, Marios Fragkoulis, Vaggelis Giannikas, Georgios Gousios, Stavros Grigorakakis, Vassilios Karakoidas, Maria Kechagia, Christos Lazaris, Dimitris Mitropoulos, Christos Oikonomou, Tushar Sharma, Sofoklis Stouraitis, Konstantinos Stroggylos, Vaso Tangalaki, Stavros Trihias, Vasileios Vlachos, and Giorgos Zouganelis,

- In the General Secretariat for Information Systems at the Greek Ministry of Finance: Costas Balatos, Leonidas Bogiatzis, Paraskevi Chatzimitakou, Christos Coborozos, Yannis Dimas, Dimitris Dimitriadis, Areti Drakaki, Nikolaos Drosos, Krystallia Drystella, Maria Eleftheriadou, Stamatis Ezovalis, Katerina Frantzeskaki, Voula Hamilou, Anna Hondroudaki, Yannis Ioannidis, Christos K. K. Loverdos, Ifigeneia Kalampokidou, Nikos Kalatzis, Lazaros Kaplanoglou, Aggelos Karvounis, Sofia Katri, Xristos Kazis, Dionysis Kefallinos, Isaac Kokkinidis, Georgios Kotsakis, Giorgos Koundourakis, Panagiotis Kranidiotis, Yannis Kyriakopoulos, Odyseas Kyriakopoylos, Georgios Laskaridis, Panagiotis Lazaridis, Nana Leisou, Ioanna Livadioti, Aggeliki Lykoudi, Asimina Manta, Maria Maravelaki, Chara Mavridou, Sofia Mavropoulou, Michail Michalopoulos, Pantelis Nasikas, Thodoros Pagtzis, Angeliki Panayiotaki, Christos Papadoulis, Vasilis Papafotinos, Ioannis Perakis, Kanto Petri, Andreas Pipis, Nicos Psarrakis, Marianthi Psoma, Odyseas Pyrovolakis, Tasos Sagris, Apostolos Schizas, Sophie Sehperides, Marinos Sigalas, George Stamoulis, Antonis Strikis, Andreas Svolos, Charis Theocharis, Adrianos Trigas, Dimitris Tsakiris, Niki Tsouma, Maria Tzafalia, Vasiliki Tzovla, Dimitris Vafiadis, Achilleas Vemos, Ioannis Vlachos, Giannis Zervas, and Thanasis Zervopoulos.

- At the FreeBSD project: John Baldwin, Wilko Bulte, Martin Cracauser, Pawel Jakub Dawidek, Ceri Davies, Brooks Davis, Ruslan Ermilov, Bruce Evans, Brian Fundakowski Feldman, Pedro Giffuni, John-Mark Gurney, Carl Johan Gustavsson, Konrad Jankowski, Poul-Henning Kamp, Kris Kennaway, Giorgos Keramidas, Boris Kovalenko, Max Laier, Nate Lawson, Sam Leffler, Alexander Leidinger, Xin Li, Scott Long, M. Warner Losh, Bruce A. Mah, David Malone, Mark Murray, Simon L. Nielsen, David O’Brien, Johann ’Myrkraverk’ Oskarsson, Colin Percival, Alfred Perlstein, Wes Peters, Tom Rhodes, Luigi Rizzo, Larry Rosenman, Jens Schweikhardt, Ken Smith, Dag-Erling Smørgrav, Murray Stokely, Marius

Strobl, Ivan Voras, Robert Watson, Peter Wemm, and Garrett Wollman.

- At LH Software and SENA: Katerina Aravantinou, Michalis Belivanakis, Polina Biraki, Dimitris Charamidopoulos, Lili Charamidopoulou, Angelos Charitsis, Giorgos Chatzimichalis, Nikos Christopoulos, Christina Dara, Dejan Dimitrijevic, Fania Dorkofyki, Nikos Doukas, Lefteris Georgalas, Sotiris Gerodianos, Vasilis Giannakos, Christos Gkologiannis, Anthi Kalyvioti, Ersi Karanasou, Antonis Konomos, Isidoros Kouvelas, George Kyriazis, Marina Liapati, Spyros Livieratos, Sofia Livieratou, Panagiotis Louridas, Mairi Mandali, Andreas Massouras, Michalis Mastorantonakis, Natalia Miliou, Spyros Molfetas, Katerina Moutogianni, Dimitris Nellas, Giannis Ntontos, Christos Oikonomou, Nikos Panousis, Vasilis Paparizos, Tasos Papas, Alexandros Pappas, Kantia Printezi, Marios Salteris, Argyro Stamati, Takis Theofanopoulos, Dimitris Tolis, Froso Topali, Takis Tragakis, Savvas Triantafyllou, Periklis Tsahageas, Nikos Tsagkaris, Apostolis Tsigkros, Giorgos Tzamalis, and Giannis Vlachogiannis.

- At the European Computer Industry Research Center (ECRC): Mireille Ducassé, Anna-Maria Emde, Alexander Herold, Paul Martin, and Dave Morton.

- At Imperial College London in the Department of Computer Science: Vasilis Capoyleas, Mark Dawson, Sophia Drossopoulou, Kostis Dryllerakis, Dave Edmondson, Susan Eisenbach, Filippos Frangulis Anastasios Hadjicocolis, Paul Kelly, Stephen J. Lacey, Phil Male, Lee M. J. McLoughlin, Stuart McRobert, Mixalis Melachrinidis, Jan-Simon Pendry, Mark Taylor, Periklis Tsahageas, and Duncan White.

- In the Computer Science Research Group (CSRG) at the University of California at Berkeley: Keith Bostic.

- At Pouliadis & Associates: Alexis Anastasiou, Constantine Dokolas, Noel Koutlis, Dimitrios Krassopoulos, George Kyriazis, Giannis Marakis, and Athanasios Pouliadis.

- At diverse meetings and occasions: Yiorgos Adamopoulos, Dimitris Andreadis, Yannis Corovesis, Alexander Couloumbis, John Ioannidis, Dimitrios Kalogeras, Panagiotis Kanavos, Theodoros Karounos, Faidon Liampotis, Elias Papavassilopoulos, Vassilis Prevelakis, Stelios Sartzetakis, Achilles Voliotis, and Alexios Zavras.

Finally, I want to thank my family, which has for years ungrudgingly endured my debugging of systems at home and which has graciously supported my writing, sometimes even over (what should have been) vacations and weekend rest. Special thanks go to Dionysis, who created Figure 5.2, and Eliza and Eleana, who helped select the book's cover.

2 General-Purpose Methods and Practices

The way you debug a failure often depends on the underlying technology and development platform. Yet, there are methods that you can use on a wide variety of cases.

Item 9: Set Yourself Up for Debugging Success

Software is often extremely complex. The movement of a mechanical watch comprises just over a hundred parts; the wiring of your entire home can have a few times as many simple components. Compare that with typical software systems, which easily consist of tens of thousands of complex statements. At the high end, consider the 9 million lines of code in the Linux kernel against the 4 million physical components in an A380 airliner. Your mind needs all the help it can get to conquer this complexity.

First you need to **believe that the problem can be found and fixed**. Your state of mind affects your debugging performance; this is what the experts call *a match between perceived challenges and skills*. If you don’t believe you can conquer the problem, your mind will wander around or give up. In such a case, you may also end up harming the code by patching the symptom, instead of the problem. Here is what you should keep in mind.

If a problem is reproducible, then make no mistake, you can fix it! (Often by following the advice in this book.) If it’s not reproducible, there are ways to make it so. In debugging you typically have two important allies: access to all the data you may require and powerful computers to process it. You can examine the problem manifestation, logs, source code, even machine instructions. You can also add detailed log statements (or at least monitoring probes) in any place of the software stack you want, and then use tools or short scripts to sift through volumes of data to locate the culprit. It is this combined ability to cast a wide net and dive arbitrarily deep when needed that makes debugging possible and, also, a uniquely satisfying experience.

To be effective in debugging you also need to **set aside ample time.** Debugging is a very demanding activity, more complex than programming, because it requires you to maintain in your brain both the program's logic and its underlying effects—often at a low level. It also requires you to set up your environment, breakpoints, logging, windows, and test cases exactly right if the problem is to be reproduced in a productive fashion. Don't squander all your invested time by stopping before you've squashed the bug, or at least until you've understood precisely what you need to do.

The complexity of debugging also requires you to **work without distractions.** Your brain needs time to enter a state called *flow* in which you become fully immersed and involved in an activity. According to Mihály Csíkszentmihályi, who termed it, in the state of flow you align your emotions with the task you perform. Flow can boost your persistence and performance through a sense of accomplishment. These are critical success factors for dealing with the immense difficulty of debugging complex systems. Distractions, such as a popup message, a phone call, a running chat, rolling social network updates, or a colleague asking for help will drag you out of the flow state, depriving you of its benefits. Avoid them! Quit unneeded applications, enable your phone's silent mode, and hang a *do not disturb sign* on your monitor (or your office door, should you be so lucky as to have one).

Another helpful strategy is to **sleep on a difficult problem.** Researchers have found that during sleep our neurons make connections that generalize across seemingly unrelated paths. This can be a great help during debugging. You can often escape from what appears to be a dead end by trying an outside-the-box debugging strategy. Sleep is exactly the process needed to make this new connection. However, for this to work, you need to do it properly. Work hard on the problem before going to sleep to give your mind all the necessary data needed in order to find a novel solution to the problem. Giving up and going for a beer and then to bed at the first difficulty won't help you a lot. Also, get plenty of sleep so that on the next day the conscious part of your brain can work effectively with the recommendations of its subconscious sibling.

Nobody said that debugging is easy, so to be effective in it you must **persist.** At the lowest level computers are deterministic, so they allow you to dig down until you isolate the error. At higher levels, nondeterminism (apparent randomness) is introduced to increase expressiveness and efficiency (think of threads). For nondeterministic errors, you can use the fact that computers are fast and programmable to run zillions of cases until you isolate the error. Therefore, debugging dead ends are mostly due to a lack of persistence: a missing test case, an ignored log file, or an unexplored angle of attack.

Finally, as an effective debug engineer, you must continuously **invest in your environment, tools, and knowledge.** Only in this way will you be able to keep your edge over the ever-increasing complexity of the technology stack you're working on. In retrospect, my most common debugging mistake is insufficient investment in setting up my debugging infrastructure. This may involve failing to do any of the following:

- Prepare a robust minimal test case (see Item 10: "Enable the Efficient Reproduction of the Problem")
- Automate the bug's reproduction
- Script a log file's analysis
- Learn how an API or language feature really works

Once I summon the energy to invest in what's needed, my debugging productivity receives a large boost. From that point onward, I can often pinpoint the bug in minutes.

Things to Remember

- ✦ Believe that the problem can be traced and fixed.
- ✦ Set aside sufficient time for your debugging task.
- ✦ Arrange to work without distractions.
- ✦ Sleep on a difficult problem.
- ✦ Don't give up.
- ✦ Invest in your environment, tools, and knowledge.

Item 10: Enable the Efficient Reproduction of the Problem

A key to effective debugging is a problem that you can reliably and easily reproduce. You need this for a number of reasons. First, if you can always reproduce the issue with a single hit of a button, you can focus on tracking down the cause rather than wasting time randomly fumbling to make the problem appear. In addition, if you can provide an easy way to reproduce the problem, you can easily take the description and ask for outside help (see Item 2: "Use Focused Queries to Search the Web for Insights into Your Problem"). Finally, once you fix the fault, you can easily demonstrate that your fix works by running the sequence that demonstrated the problem again and witnessing that the failure no longer occurs.

Creating a short example or a *test case* that reproduces the problem can go a long way in increasing your efficiency. The golden standard is a *minimal* example: the shortest possible that reproduces the problem. The platinum standard, which goes under the name SSCCE (see Item 1: “Handle All Problems through an Issue-Tracking System”), has the example be not only short, but also self-contained and correct (compilable and runable). With a minimal example at hand, you won’t waste time exploring code paths that could have been eliminated. Also, any logs and traces you create and must examine won’t be longer than what’s actually needed. And, a short example will also execute more quickly than a longer one, especially when executed in a debugging mode that imposes a significant performance overhead.

To shorten your example, you can proceed top-down or bottom-up (see Item 4: “Drill Up from the Problem to the Bug or Down from the Program’s Start to the Bug”). Select the most expedient method. If the code has many dependencies, starting bottom-up from a clean slate may be preferable. If you don’t really understand the problem’s likely cause, creating a test case in a top-down fashion may help you narrow down the possibilities.

In the bottom-up fashion, you theorize the cause of the problem, for example, a call to a specific API, and you build up a test case that demonstrates the problem. In one case, I was trying to find out why a 27,000-line program was extremely slow in the complex code it used for processing its input files. By looking at the program’s invoked system calls, I hypothesized that the problem had something to do with calling `tellg`—a function returning the file stream’s offset—while reading the file. Indeed, running the following short snippet confirmed my suspicion (see Item 58: “Trace the Code’s Execution”) and was also useful to test the workaround (a wrapper class).

```
ifstream in(fname.c_str(), ios::binary);
do {
  (void)in.tellg();
} while ((val = in.get()) != EOF);
```

In the top-down fashion, you remove elements from the scenario that demonstrates the problem, until there’s nothing left to remove. A binary search technique is often quite useful. Say you have an HTML file that makes the browser behave in an erratic way. First eliminate the file’s `head` elements. If the problem persists, eliminate the `body` elements. If that cures the problem, restore the `body` elements, and then remove half of them. Repeat the process until you’ve nailed down the elements that cause the problem. Keeping your editor open and using its undo function to backtrack when you follow a wrong path will mightily increase your efficiency.

With a short example at hand, it's also easy to make it **self-contained.** This means that you can take the example and replicate the problem somewhere else without external dependencies, such as libraries, headers, CSS files, and web services. If your test case requires some external elements, you can bundle them with it. Use a portable notation for referring to them, avoiding things such as absolute file paths and hard-coded IP addresses. For instance use ../resources/file.css rather than /home/susan/resources/file.css, and http://localhost:8081/myService rather than http://193.92.66.100:8081/myService. A self-contained example will make it easier for you to try it on the customer's premises, examine it on another platform (say, on Windows instead of Linux), publish it on a Q&A forum (see Item 2: "Use Focused Queries to Search the Web for Insights into Your Problem"), and ship it to a vendor for further help.

In addition, you want to work on a **replicable execution environment.** If you don't nail down the code you're working on and the system it executes in, then you might end up searching for a bug that simply isn't there. Consider the case of debugging a software installer. Every time you install it, it messes up your operating system configuration, which is exactly what you want to avoid when you're trying to debug it. In this case, a useful technique is to create a virtual machine image with a pristine system in a state ready for the software installation. After every failed installation, you can simply start afresh with that image. You can also often achieve a similar result using operating-system-level virtualization or containment with a tool such as *Docker*. Even better, consider adopting a system configuration management tool, such as *Ansible*, *CFEngine*, *Chef*, *Puppet*, or *Salt*. These tools allow you to reliably create a specified system configuration from your high-level instructions. This makes it easy to maintain compatible production, testing, and development environments, and to control their evolution in the same way as you control your software.

You also want to be able to **reliably replicate the failing version** of your software. To do this, first put your software under configuration management with a tool, such as *Git*. Then, make your build process embed into the software an identifier of the source code version used for the build. The following shell command will print a variable initialization with the abbreviated Git hash of the last commit, which you can embed into your source code.

```
git log -n 1 --format='const string version = "%h";'
```

Here is an example of its output.

```
const string version = "035cd45";
```

Add to your software a way to display this version string; a command-line option or a line in the *About* dialog are all that’s needed. With this version identifier at hand you can then obtain a copy of the failing source code with a command such as the following:

```
git checkout 035cd45
```

If you want to increase the fidelity of builds you run on old code, don’t forget to put under version control all elements that affect what ends up in your distribution, such as the compiler, system and third-party libraries and header files, as well as the build specification (the *Makefile*s or IDE project configuration). As a final step, you may need to remove the variability introduced by your tools and your runtime environment (see Item 52: “Configure Deterministic Builds and Executions”).

Things to Remember

- Reproducible runs simplify your debugging process.
- Create a short self-contained example that reproduces the problem.
- Have mechanisms to create a replicable execution environment.
- Use a revision control system to label and retrieve your software’s versions.

Item 11: Minimize the Turnaround Time from Your Changes to Their Result

Debugging is often a process of successive approximation. The time you wait (again and again) for the software to build, run, and fail, and the time you spend (again and again) to cajole it to go through these steps is time you don’t devote in solving your problem. Therefore, early on, invest in minimizing the time it takes to go through a debug cycle.

Start with the software build. You should be able to **quickly build** the failing software with a single command or keystroke, such as `make`, `mvn compile`, or `F5`. The build process should track dependencies between files ensuring that only a few files get compiled after you change something. Tools that can help you here include *make*, *Ant*, and *Maven*.

The **efficient deploying and running** of the software is equally important. The steps here vary a lot between projects. You may need to deploy files on a remote host, restart an application server, clear caches, or reinitialize a database. Use the project’s build system or write some scripts to automate this process (see Item 12: "Automate Complex Testing Scenarios"). If a typical installation of your software involves a protracted construction of a distribution file and its subsequent slow installation, setup a shortcut where you only copy the modified files to their final location.

Finally, ensure that the software will **quickly fail** (see Item 55: "Fail Fast"). If the failing code comes with unit tests or a regression-testing framework, build a test case that demonstrates the specific failure (see Item 10: "Enable the Efficient Reproduction of the Problem"). Then use features of your IDE or testing environment to run the specific test case. For instance, under *Maven* you can run the `TestFetch` case with the following command.

```
mvn -Dtest=TestFetch test
```

If the program you're debugging can be made to fail by processing a specific file, then construct a minimal file that can trigger this failure. To replicate problems in GUI applications, you can use software automation applications, such as *Selenium* for web browsers, *AutoHotkey* for Windows, *Automator* for OS X, and *AutoKey* for Linux.

Things to Remember

- A fast turnaround time increases your effectiveness.
- Set up a fast automated build and deployment process.
- Minimize the time it takes for your tests to fail.

Item 12: Automate Complex Testing Scenarios

Automate complex scripting scenarios through the use of scripting. There are multiple options for this. For orchestrating processes and files, the Unix shell offers many useful facilities (see Item 22: "Analyze Debug Data with Unix Command-Line Tools"). In addition, with commands such as *cURL* to fetch URLs and *jq* to parse JSON data, you can also use the shell to test web services. In complex cases involving API access and state maintenance, you will benefit from a more sophisticated scripting language, such as Python, Ruby, or Perl. Also, numerous systems come with their own built-in scripting language; for example, the *Apache HTTP Server*, the *Wireshark* network packet analyzer, and the *VLC* media player all support the *Lua* programming language.

If your software does not support a scripting language and you have the ability to modify it, consider bolting a scripting language onto it, and add API bindings that expose the scripting language to your program's functions. As a (simple but contrived) example, say you want to construct a test case associated with a math library you've implemented. The C program in Listing 2.1 will load and run the Lua program debug.lua, exposing to it the functions sin, cos, and tan.

Listing 2.1 Exporting C functions for testing through Lua

```c
#include <math.h>
#include "lua5.2/lua.h"
#include "lua5.2/lauxlib.h"

// Functions exposed to Lua
static int l_sin(lua_State *L) {
   double value_as_number = luaL_checknumber(L, 1);
   // Call the function, and return the result
   lua_pushnumber(L, sin(value_as_number));
   return 1; // Single result
}

static int l_cos(lua_State *L) {
   double value_as_number = luaL_checknumber(L, 1);
   lua_pushnumber(L, cos(value_as_number));
   return 1;
}

static int l_tan(lua_State *L) {
   double value_as_number = luaL_checknumber(L, 1);
   lua_pushnumber(L, tan(value_as_number));
   return 1;
}

int main() {
   // Setup Lua
   lua_State *L = luaL_newstate();
   luaL_openlibs(L);

   // Expose the functions to Lua
   lua_pushcfunction(L, l_sin);
   lua_setglobal(L, "lsin");
   lua_pushcfunction(L, l_cos);
   lua_setglobal(L, "lcos");
   lua_pushcfunction(L, l_tan);
   lua_setglobal(L, "ltan");
   // Load and run the debug file
   luaL_dofile(L, "debug.lua");
   puts("Done");
   return 0;
}
```

On a Debian Linux distribution, I installed Lua by running `sudo apt-get install lua5.2-dev` and compiled the program with `cc myprog.c -llua5.2 -lm`. (You can find advice on how to install Lua on other systems in Lua’s documentation.) I then wrote the following small Lua program to verify the accuracy of the functions with respect to the definition of the tangent function.

$$\tan\vartheta = \frac{\sin\vartheta}{\cos\vartheta}$$

```
epsilon = 1
errors = 0
while epsilon > 0 and errors < 2 do
   for theta = 0, 2 * math.pi, 0.1 do
      diff = lsin(theta) / lcos(theta) - ltan(theta)
      if (math.abs(diff) > epsilon) then
         print(epsilon, theta, diff)
         errors = errors + 1
      end
   end
   epsilon = epsilon / 10
end
```

Running the C program will load the Lua code and produce output such as the following:

```
1e-14 4.7   1.4210854715202e-14
1e-15 1.5   1.7763568394003e-15
1e-15 4.7   1.4210854715202e-14
```

In a more realistic scenario, the C program would be your large application, the trigonometric functions would be the functions of the application you wanted to check, and the Lua program would make it easy for you to tinker with test cases for these functions.

Things to Remember

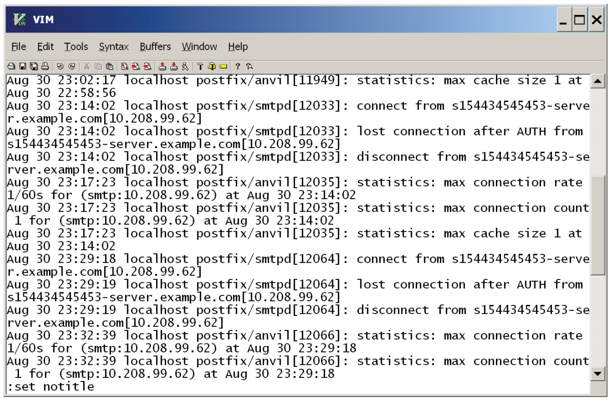
- Automate the execution of complex test cases through the use of a scripting language.

Item 13: Enable a Comprehensive Overview of Your Debugging Data

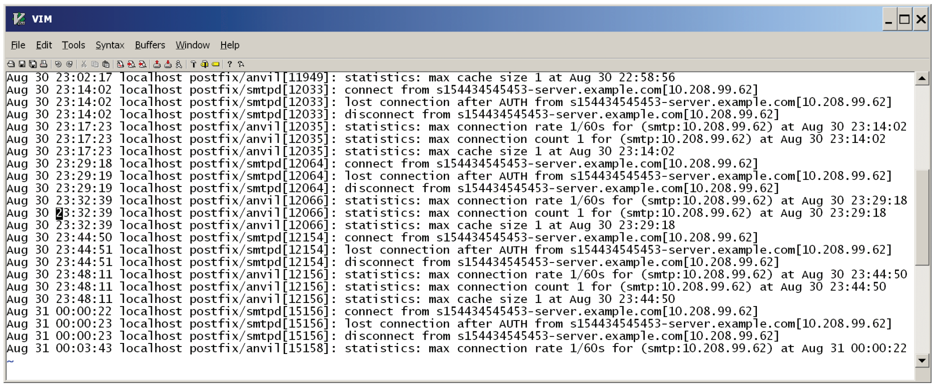
Effective debugging entails processing and correlating loads of diverse data: source code, log entries, the values of variables, stack contents, program I/O, and test results, all of these often from multiple processes and computing hosts. Having all those data properly laid out in front of you offers you many benefits. First, it allows you to detect correlations, such as a log entry appearing when a test fails. Then, it minimizes your context switching overhead and the disruptions this brings. Having to enter a command or juggle windows to see the values of some variables when you single-step through code can break the all-important mental flow (see Item 9: "Set Yourself Up for Debugging Success") that might be required to spot a crucial connection. Also, sufficient space to lay out long lines can help you establish patterns that you would otherwise miss. You may have configured your editor windows to match the 70–80 columns prescribed by many code style guides. However, when long lines of log files and stack traces are folded multiple times to fit into such an 80-column line, they become difficult to read and analyze. Lay those lines out on your monitor's glorious full width, and patterns will stand out; see Figure 2.1 for an example. Here are some ways to increase the amount of data you can examine when you debug.

First, **maximize your display area.** Many developers also use two (or more) high-resolution monitors. (Using large cheap TV screens won't cut it, for they will just give you blurry characters.) For this to work, you'll need a correspondingly powerful graphics interface. If you're working on a laptop, you can connect an external monitor, *extend* (rather than *clone*) your display into it, and benefit from the increased screen real estate. In all these cases, don't shy away from switching your editor or terminal window to full-screen mode. It may look silly on a modern full HD monitor, but for some tasks, being able to see the data along and across is an indispensable affordance. If the data still can't fit, decrease the font size (and get a pair of glasses) or use a video projector.

Printing stuff can also be remarkably effective. At just 600-DPI resolution, a laser printer can display 6600 × 5100 pixels on a letter-size paper—many more than your monitor, which you can use to display more and crisper data. You can use printed sheets for items that don't change a lot, such as data structure definitions and listings, and free up your screen to display the more dynamic parts of your debugging session. Finally, the best medium for program listings is surely the 15-inch green bar fanfold paper. There you can print 132-column-wide text of unlimited length. If you've still got access to a printer that can handle this, use it. And guard its existence.



```
Aug 30 23:02:17 localhost postfix/anvil[11949]: statistics: max cache size 1 at
Aug 30 22:58:56
Aug 30 23:14:02 localhost postfix/smtpd[12033]: connect from s154434545453-serve
r.example.com[10.208.99.62]
Aug 30 23:14:02 localhost postfix/smtpd[12033]: lost connection after AUTH from
s154434545453-server.example.com[10.208.99.62]
Aug 30 23:14:02 localhost postfix/smtpd[12033]: disconnect from s154434545453-se
rver.example.com[10.208.99.62]
Aug 30 23:17:23 localhost postfix/anvil[12035]: statistics: max connection rate
1/60s for (smtp:10.208.99.62) at Aug 30 23:14:02
Aug 30 23:17:23 localhost postfix/anvil[12035]: statistics: max connection count
 1 for (smtp:10.208.99.62) at Aug 30 23:14:02
Aug 30 23:17:23 localhost postfix/anvil[12035]: statistics: max cache size 1 at
Aug 30 23:14:02
Aug 30 23:29:18 localhost postfix/smtpd[12064]: connect from s154434545453-serve
r.example.com[10.208.99.62]
Aug 30 23:29:19 localhost postfix/smtpd[12064]: lost connection after AUTH from
s154434545453-server.example.com[10.208.99.62]
Aug 30 23:29:19 localhost postfix/smtpd[12064]: disconnect from s154434545453-se
rver.example.com[10.208.99.62]
Aug 30 23:32:39 localhost postfix/anvil[12066]: statistics: max connection rate
1/60s for (smtp:10.208.99.62) at Aug 30 23:29:18
Aug 30 23:32:39 localhost postfix/anvil[12066]: statistics: max connection count
 1 for (smtp:10.208.99.62) at Aug 30 23:29:18
:set notitle
```



```
Aug 30 23:02:17 localhost postfix/anvil[11949]: statistics: max cache size 1 at Aug 30 22:58:56
Aug 30 23:14:02 localhost postfix/smtpd[12033]: connect from s154434545453-server.example.com[10.208.99.62]
Aug 30 23:14:02 localhost postfix/smtpd[12033]: lost connection after AUTH from s154434545453-server.example.com[10.208.99.62]
Aug 30 23:14:02 localhost postfix/smtpd[12033]: disconnect from s154434545453-server.example.com[10.208.99.62]
Aug 30 23:17:23 localhost postfix/anvil[12035]: statistics: max connection rate 1/60s for (smtp:10.208.99.62) at Aug 30 23:14:02
Aug 30 23:17:23 localhost postfix/anvil[12035]: statistics: max connection count 1 for (smtp:10.208.99.62) at Aug 30 23:14:02
Aug 30 23:17:23 localhost postfix/anvil[12035]: statistics: max cache size 1 at Aug 30 23:14:02
Aug 30 23:29:18 localhost postfix/smtpd[12064]: connect from s154434545453-server.example.com[10.208.99.62]
Aug 30 23:29:19 localhost postfix/smtpd[12064]: lost connection after AUTH from s154434545453-server.example.com[10.208.99.62]
Aug 30 23:29:19 localhost postfix/smtpd[12064]: disconnect from s154434545453-server.example.com[10.208.99.62]
Aug 30 23:32:39 localhost postfix/anvil[12066]: statistics: max connection rate 1/60s for (smtp:10.208.99.62) at Aug 30 23:29:18
Aug 30 23:32:39 localhost postfix/anvil[12066]: statistics: max connection count 1 for (smtp:10.208.99.62) at Aug 30 23:29:18
Aug 30 23:32:39 localhost postfix/anvil[12066]: statistics: max cache size 1 at Aug 30 23:29:18
Aug 30 23:44:50 localhost postfix/smtpd[12154]: connect from s154434545453-server.example.com[10.208.99.62]
Aug 30 23:44:51 localhost postfix/smtpd[12154]: lost connection after AUTH from s154434545453-server.example.com[10.208.99.62]
Aug 30 23:44:51 localhost postfix/smtpd[12154]: disconnect from s154434545453-server.example.com[10.208.99.62]
Aug 30 23:48:11 localhost postfix/anvil[12156]: statistics: max connection rate 1/60s for (smtp:10.208.99.62) at Aug 30 23:44:50
Aug 30 23:48:11 localhost postfix/anvil[12156]: statistics: max connection count 1 for (smtp:10.208.99.62) at Aug 30 23:44:50
Aug 30 23:48:11 localhost postfix/anvil[12156]: statistics: max cache size 1 at Aug 30 23:44:50
Aug 31 00:00:22 localhost postfix/smtpd[15156]: connect from s154434545453-server.example.com[10.208.99.62]
Aug 31 00:00:23 localhost postfix/smtpd[15156]: lost connection after AUTH from s154434545453-server.example.com[10.208.99.62]
Aug 31 00:00:23 localhost postfix/smtpd[15156]: disconnect from s154434545453-server.example.com[10.208.99.62]
Aug 31 00:03:43 localhost postfix/anvil[15158]: statistics: max connection rate 1/60s for (smtp:10.208.99.62) at Aug 31 00:00:22
~
```

Figure 2.1 A representation of how a log file will look: in the default editor setup (top) and with a wide view (bottom)

Things to Remember

- With a lot of data in view, you can concentrate better and spot patterns and correlations.
- Use the largest display area you can obtain.
- Display relatively static data on printed sheets.

Item 14: Consider Updating Your Software

Guess what? Your code is not the only one that has errors in it. The compiler or interpreter that's processing your code, the libraries you use, the database and application servers you depend on, and the operating systems that are hosting everything all have their own fair share of bugs. For example, at the time of this writing, the Linux source code contained more than 2,700 comments marked with XXX, which typically denotes a questionable construct; some of these are surely bugs.

Consequently, some bugs can be addressed by **updating** your software. Using a newer compiler or library may help you correct an obscure bug in the packaged software you're shipping. If you're delivering a software-based service, then updating middleware, databases, and the operating system can also help. At the very least try building, linking, or running your code with the newest versions to eliminate the possibility that you're witnessing a third-party bug. However, keep in mind that there's a lot to be said for a conservative upgrade policy—working with the devil you know. A lot of middleware suffers from faulty or limited backward compatibility, so experienced users are usually very careful when doing updates, opting for the next earliest bug-fixing release (e.g., 6.4.3) that would solve their problem. Also, new software can introduce new bugs and incompatibilities, so at the very least don't burn your bridges rushing into it: have a sensible back-off plan if the upgrade doesn't work, or if it doesn't address the bug you're witnessing. Updating the third-party code in a sandbox, such as a throw-away cloned virtual machine image, is a reliable and easy way to achieve this. Whatever happens, don't expect too much from software upgrades.

It's best to assume that outside code is innocent until proven guilty. Most of the time, bugs you blame on third-party code are actually problems in your own. Over the course of thirty years, I've fixed thousands of bugs in my own code. Over the same period, I've encountered a single case where a widely used commercial compiler generated incorrect code, a few instances of bugs in libraries, one case of unreliable operating system functionality, a handful of errors in system call documentation, and just tens of errors in tools and other system software. Therefore, the biggest benefit you'll get from using updated software is a new resolve to get your own house in order.

Things to Remember

- Try your failing system on an updated environment.
- Don't expect too much from this exercise.
- Consider the possibility of third-party bugs.

Item 15: Consult Third-Party Source Code for Insights on Its Use

Often problems occur due to the way the code you're debugging uses a third-party library or application (rather than actual bugs in that software—see Item 14: "Consider Updating Your Software").

This is no surprise, because such software gets integrated with your code as a black box, and therefore you have fewer opportunities to coordinate. A powerful way to debug these problems is to consult the source code of third-party libraries, middleware, and even lower-level software.

First, if you need to know why a particular API behaves in an unexpected way or what triggers a cryptic error message, you can find the answer by **browsing the third-party source code** around the area that interests you. To debug functionality related to a library, locate the function or method definition and follow the code from there. You're probably not looking for a bug in the library's code, but rather for a better understanding of how the library works and ties in with your code. To understand an error message, search through all the code for the wording of the error message, and examine the code leading to it (see Item 23: "Utilize Command-Line Tool Options and Idioms" and Item 4: "Drill Up from the Problem to the Bug or Down from the Program's Start to the Bug"). You can quickly locate the function or method you're looking for by indexing the code with the *ctags* or *etags* program (most editors support its output), or with your integrated development environment (IDE). Your IDE is likely to handle sophisticated language features such as overloading, overriding, and templates better than *ctags*. On the other hand, *etags* handles many more languages: 41 in version 5.8. The following command, when run on a source code directory, will create an index for all files lying under it.

```
ctags -R .
```

If the third-party code you're using is open source, you can also search through it through a hosted service, such as the *Black Duck Open Hub Code Search*.

A more powerful technique involves **building the third-party with debugging information** (see Item 28: "Use Code Compiled for Symbolic Debugging"). Then link your code with the debug version of the library you built. Having done that, you can easily step through the third-party code and examine variables with the symbolic debugger (see Chapter 4), just as you can do with your own code. Note that some vendors, such as Microsoft, ship with their code debug builds or symbols. This saves you the effort of debug-building their code on your own.

If you happen to find that a fault lies in the third-party code rather than yours, with access to the source code you can actually **correct** it there. Use this option only in extreme circumstances: if there is no reasonable workaround, and you can't get the vendor to fix it for you. Once you modify third-party code, you'll be responsible for maintaining the change across its new versions for the lifetime of your application.

Also ensure you're legally allowed to modify the code. Some vendors ship their code with a "look, don't touch" license. For open-source software, a reasonable option is to submit your changes to the project responsible for the code. This is also the right thing to do. If the project is hosted on *GitHub*, you can easily do that with a *pull request*.

For all these wonderful things to work, **you need to have the third-party source code at hand**. This is trivial if the library or application you're using is open source. Then, you can download the source code with a click of a button. Open-source operating system distributions also offer you the ability to download the source code as a package; this is the command you would use under Debian Linux to install the C library source code.

```
sudo apt-get install glibc-source
```

In addition, many software development platforms will install important parts of their source code on your system. For instance, you can find the source code for the C runtime library of Microsoft's *Visual Studio* at a location VC\crt\src and for the Java Development Kit in an archive named src.zip. In other cases, you may have to pay extra to obtain the source code when you order the third-party software. Insist on this option if the price is not exorbitant. Getting the source code later when you need it might require a lot of time to budget, place the order, and execute any required agreement. Also, the vendor might have stopped supporting the version you use or might even have gone out of business. Getting the source code for proprietary software beforehand is a reasonable insurance policy against these problems.

Things to Remember

- ✦ Get the source code for third-party code you depend on.
- ✦ Explore problems with third-party APIs and cryptic error messages by looking at the source code.
- ✦ Link with the library's debug build.
- ✦ Correct third-party code only when there's no other reasonable alternative.

Item 16: Use Specialized Monitoring and Test Equipment

Debugging embedded systems and systems software sometimes requires you to be able to inspect and analyze the whole computing stack, from the hardware to the application. Deep down at the hardware level, debugging involves detecting minute changes in electron flows and the alignment of magnetic moments. In most cases, you can use powerful IDEs as well as tracing and logging software to see what's going on. Yet there are situations where these tools leave you none the wiser. This often happens when software touches hardware: you think your software behaves as it should, but the hardware seems to have its own ideas. For example, you can see that you write the correct data to disk, but the data appears corrupt when you read it. When you debug problems close to the hardware level, some fancy equipment may offer you significant help.

A general-purpose tool that you may find useful is a *logic analyzer*. This captures, stores, and analyzes digital signals coming in at speeds of millions of samples per second. Such devices used to cost more than a new car, but now you can buy a cheap USB-based one for about $100. With such a device you can monitor arbitrary digital signals on a hardware board but also higher communication protocols that components use to talk to each other. The alphabet soup and number of supported protocols is bewildering; quoting from one manufacturer (*saleae*): "SPI, I2C, serial, 1-Wire, CAN, UNI/O, I2S/PCM, MP Mode, Manchester, Modbus, DMX-512, Parallel, JTAG, LIN, Atmel SWI, MDIO, SWD, LCD HD44780, BiSS C, HDLC, HDMI CEC, PS/2, USB 1.1, Midi."

If you specialize in a particular technology, you may want to invest in dedicated equipment, such as a *protocol analyzer* or *bus analyzer*. As an example, vehicle and other microcontrollers often communicate over the so-called CAN (controller area network) bus. A number of companies offer stand-alone self-contained modules that can plug into the bus and filter, display, and log the exchanged traffic. Similar products exist for other widely used or specialized physical interconnections and protocols, such as Ethernet, USB, Fibre Channel, SAS, SATA, RapidIO, iSCSI, sFPDP, and OBSAI. In contrast to software-based solutions, these devices are guaranteed to work at the true line rate, they offer support for monitoring multiple traffic lanes, and they allow you to define triggers and filters based on bit patterns observed deep inside the data packet frame.

If you lack dedicated debugging hardware, don't shy away from **improvising** something to suit your needs. This can help you investigate problems that are difficult to replicate. A few years ago we faced a problem with missing data from a web form drop box. The trouble was that the occurrence of the problem was rare and impossible to reproduce, though it affected many quite vocal users for days. (The application was used by hundreds of thousands.) By looking at the distribution of the affected users, we found that these were often based in remote areas. Hypothesizing that the problem had to do with the quality of their Internet connection, I took a USB wireless modem, wrapped it in tinfoil to simulate a marginal connection, and used that to connect to the application's web interface. I could immediately see the exact problem, and, armed with an easy way to replicate (see Item 10: "Enable the Efficient Reproduction of the Problem") it, we were able to solve it in just a few hours.

If the code you're trying to debug runs as **embedded software** on a device that lacks decent I/O, there are several tricks you can play to communicate with the software you're debugging.

- If the device has a status light or if it can beep, use coded flashes or beeps to indicate what the software is doing. For example, one short beep could signify that the software has entered a particular routine and two that it has exited. You can send more sophisticated messages with *Morse Code*.
- Store log output in non-volatile storage (even on an external USB thumb drive), and then retrieve the data on your own workstation in order to analyze it.
- Implement a simple serial data encoder, and use that to write the data on an unused I/O pin. Then, you can level-convert the signal to *RS-232* levels and use a serial-to-USB adapter and a terminal application to read the data on a modern computer.
- If the device has a network connection, you can obviously communicate through it. If it lacks the software for network logging (see Item 41: "Add Logging Statements") or remote shell access, you can communicate with the outside world through HTTP or even DNS requests.

When you're monitoring **network packets,** you can set up your network's hardware in a way that allows you to use a *software packet analyzer*, such as the open-source *Wireshark* package. The Wireshark version running on my laptop claims to support 1,514 (network and USB) protocols and packet types. If you can run the application you want to debug on the same host as the one you'll use Wireshark, then network packet monitoring can be child's play. Fire up Wireshark, specify the packets you want to capture, and then look at them in detail.

Monitoring traffic between other hosts, such as that between an application server and a database server or a load balancer, can be more tricky. The problem is that *switches*, the devices that connect together

Ethernet cabling, isolate the traffic flowing between two ports from the others. You have numerous options to overcome this difficulty.

If your organization is using a *managed* (read "expensive") switch, then you can set up one port to mirror the traffic on another port. Mirroring the traffic of the server you want to monitor on the port where your computer running Wireshark is connected allows you to capture and analyze the server's traffic.

If you don't have access to a managed switch, try to get hold of an Ethernet *hub*. These are much simpler devices that broadcast the Ethernet traffic they receive on all ports. Hubs are no longer made, and this is why they're often more expensive than cheap switches. Connect the computer you want to monitor and the one running Wireshark to the hub, and you're ready to go.

Yet another way to monitor remote hosts involves using a command-line tool such as *tcpdump*. Remotely log in into the host you want to monitor, and run *tcpdump* to see the network packets that interest you. (You will need administrator privileges to do that.) If you want to perform further analysis with Wireshark's GUI, you can write the raw packets into a file with *tcpdump*'s `-w` option, which you can then analyze in detail with Wireshark. This mode of working is particularly useful with cloud-based hosts, where you can't easily tinker with the networking configuration.

One last possibility involves setting up a computer to *bridge* network packets between the computer you want to monitor and the rest of the network. You configure that computer with two network interfaces (e.g., its native port and a USB-based one), and bridge the two ports together. On Linux use the *brctl* command; on FreeBSD configure the *if_bridge* driver.

You can also use a similarly configured device to simulate various networking scenarios such as packets coming from hosts around the world, traffic shaping, bandwidth limitations, and firewall configurations. Here the software you'll want to use is *iptables*, which runs under Linux.

Things to Remember

- A logic, bus, or protocol analyzer can help you pinpoint problems that occur near the hardware level.
- A home-brew contraption may help you investigate problems related to hardware.
- Monitor network packets with *Wireshark* and an Ethernet hub, a managed switch, or command-line capture.

Item 17: Increase the Prominence of a Failure's Effects

Making problems stand out can increase the effectiveness of your debugging. You can achieve this by manipulating your software, its input, or its environment. In all cases, ensure you perform the changes under revision control in a separate branch, so that you can easily revert them and they won't end up by mistake in production code.

There are cases where your software simply refuses to behave in the way you expect it to. For example, although certain complex conditions are apparently satisfied, the record that's supposed to appear in the database doesn't show up. A good approach in such cases is to lobotomize the software through **drastic surgery** and see if it falls in line. If not, you're probably barking up the wrong tree.

As a concrete case, consider the following (abridged) code from the Apache HTTP server, which deals with signed certificate timestamps (SCTs). You might be observing that the server fails to react to SCTs with a timestamp lying in the future.

```
for (i = 0; i < arr->nelts; i++) {
   cur_sct_file = elts[i];
   rv = ctutil_read_file(p, s, cur_sct_file, MAX_SCTS_SIZE,
              &scts, &scts_size_wide);
   rv = sct_parse(cur_sct_file,
              s, (const unsigned char *)scts, scts_size, NULL,
              &fields);
   if (fields.time > apr_time_now()) {
      sct_release(&fields);
      continue;
   }
   sct_release(&fields);
   rv = ctutil_file_write_uint16(s, tmpfile,
              (apr_uint16_t)scts_size);
   if (rv != APR_SUCCESS)
      break;
   scts_written++;
}
```

A way to debug this is to temporarily change the conditional so that it always evaluates to *true*.

```
   if (fields.time > apr_time_now() || 1) {
```

This change will allow you to determine whether the problem lies in the Boolean condition you short circuited, in your test data, or in the rest of the future SCT handling logic.

Other tricks in this category are to add a `return true` or `return false` at the beginning of a method, or to disable the execution of some code by putting it in an `if (0)` block (see Item 46: "Simplify the Suspect Code").

In other cases, you may be trying to debug a barely observable effect. Here the solution is to temporarily modify the code to **make the effect stand out.** If, in a game, a character gets a minute increase in power after some event, and that doesn't seem to happen, make the power increase dramatically more so that you can readily observe it. Or, when investigating the calculation of an earthquake's effects on a building in a CAD program, magnify the displayed structure displacement by 1,000 so that you can easily see the magnitude and direction of the structure's movement.

In cases where your software's failure depends on external factors, you can increase your effectiveness by **modifying the environment** where your software executes in order to make it fail more quickly or more frequently (see Item 55: "Fail Fast"). If your software processes web requests, you can apply a *load test* or *stress test* tool, such as *Apache JMeter*, in order to force your application into the zone where you think it starts misbehaving. If your software uses threads to achieve concurrency, you can increase their number far beyond what's reasonable for the number of cores in the computer you're using. This may help you replicate deadlocks and race conditions. You can also force your software to compete for scarce resources by concurrently running other processes that consume memory, CPU, network, or disk resources. A particularly effective way to investigate how your software behaves when the disk fills up is to make it store its data in a puny USB flash drive.

Finally, a testing approach that can also help you investigate rare data validation or corruption problems is *fuzzing*. Under this approach you either supply to your program randomly generated input, or you randomly perturb its input, and see what happens. Your objective is to increase the likelihood of chancing on the data pattern that produces the failure in a systematic way. Having done that, you can use the problematic data to debug the application. This technique may, for example, help you find out why your application crashes when running on your customer's production data but not when it's running on your own test data. You can perform fuzzing operations using a tool such as *zzuf* .

Things to Remember

✦ Force the execution of suspect paths.

✦ Increase the magnitude of some effects to make them stand out for study.

- Apply stress to your software to force it out of its comfort zone.
- Perform all your changes under a temporary revision control branch.

Item 18: Enable the Debugging of Unwieldy Systems from Your Desk

Jenny and Mike are comparing notes regarding their debugging experiences. "I hate it when I work on a customer's PC," says Jenny. "My tools are missing, I don't have my browser's bookmarks, it's noisy, I can't access my files, their key bindings and shortcuts are all wrong." Mike looks at her with disbelief: "Key bindings? You're lucky, you have a keyboard!"

Indeed, having to work away from your workstation can be a major drain on your productivity. Apart from Jenny's complaints, other nuisances can be constrained Internet or intranet access, an awkward setup (from the screen to the chair, via the mouse and keyboard), a need to travel to a humid, hot (or cold) location in the middle of nowhere, and undersized computing power. These problems are quite common and will become even more widespread as our industry embraces mobile devices and the Internet of Things. Cases where you may need to debug software away from your cozy desk and powerful workstation include cellphone apps, devices with embedded software, problems that occur only on a customer's computer, and crises that occur in the data center. There are workarounds so that you can continue working with your favorite keyboard, but you must plan ahead.

For cellphone apps and some embedded devices, there are *device emulators*, which you can use on your PC to troubleshoot the failing application. However, these typically don't offer much help in the way of debugging, other than some enhanced logging facilities. True, you don't need to fumble with a touchscreen's keyboard any more, but you also can't run your symbolic debugger inside the emulator. Still, you have convenient access to the source code and your editor from the same screen, and you can quickly experiment with code changes and see the results without having to deploy the software on an actual device.

A more powerful approach is to create a software *shim* that will allow you to run the key parts of the application you're debugging on your workstation. Unit tests and mock objects are techniques often used for this (see Item 42: "Use Unit Tests"). The setup typically excludes the user interface, but can easily include tricky algorithmic parts where a lot of debugging may be required. Thus, you hook up the application's algorithms with some simple (e.g., file-based) input/output, so that you can compile and run the code natively on your PC, and then use your powerful debugger to step through and examine the operation of the tricky parts.

As an example, consider a cellphone app that imports into your contacts the pictures of your social network friends. A difficult part of this app is the interaction with the social networks and the contact matching. Therefore, your shim could be a command-line tool that takes as an argument a contact's name, and uses the Facebook/LinkedIn/Twitter API to retrieve and locate matching friends. Once you debug this part, you can integrate it into the cellphone app as a class. Keep the ability to compile and run it again as a stand-alone command (perhaps via a `main` method) in case a problem occurs in the future.

For troubleshooting problems on customers' PCs, arrange for **remote access.** Do that before a crisis strikes because this typically requires administrator privileges and some technical expertise. Many operating systems offer a way to access the desktop remotely, though support people often prefer the use of a dedicated application, such as *TeamViewer*. Also, consider deploying at the customers' PCs other data and tools that may simplify your debugging. This could be a viewer for your application's binary files or an execution tracer. If I'd have to choose one debugging tool to have on a third-party computer, I'd select the Unix *strace* or *truss* command. Incidentally, remote access can also simplify the troubleshooting that all of us who work in IT are routinely asked to do for friends and family.

A lot of back-end computing is nowadays done through commercial cloud offerings, which offer nifty web interfaces for debugging and console access. If the server you may end up debugging isn't hosted on a shiny cloud but in a cold, noisy, inaccessible data center, you need to plan ahead again. If a problem occurs before the server establishes network connectivity, you normally need to access it through its physical screen and keyboard. A solution to this problem is a *KVM over IP* device. This offers remote access to a computer's keyboard, video, and mouse (KVM) over an IP network. By installing, configuring, and testing such a device, you can conveniently debug problems in a remote server's boot process from the luxury of your desk.

Things to Remember

- Set up a device emulator so you can troubleshoot using your workstation's screen and keyboard.
- Use a shim to debug embedded code with your workstation's native tools.
- Arrange for remote access to customers' PCs.
- Set up KVM over IP devices to debug remote servers.

Item 19: Automate Debugging Tasks

You may find yourself with many possible discrete causes for a failure and no easy way to deduce which of them is the culprit. To identify it, you can write a small routine or a script that will perform an exhaustive search through all cases that might cause the problem. This works well when the number of cases would make it difficult to test them by hand, but possible to go through them in a loop. Iterating through 500 characters is a case that can be automated; doing an exhaustive search of all strings of user input is not.

Here is an example. After an upgrade, a computer began to delay the execution of the *which* command. Changing the long command search path (the Windows and Unix `PATH` environment variable) to `/usr/bin` removed the delay but left the question: Which of the path’s 26 elements was causing it? The following Unix shell script (run on the Windows machine through *Cygwin*) displayed the elapsed time for each path’s component.

```
# Obtain path
echo $PATH |
# Split the :-separated path into separate lines
sed 's/:/\n/g' |
# For each line (path element)
while read path ; do
  # Display elapsed time for searching through it
  PATH=$path:/usr/bin time -f "%e $path" which ls >/dev/null
done
```

Here is (part of) the script’s output:

```
0.01 /usr/local/bin
0.01 /cygdrive/c/ProgramData/Oracle/Java/javapath
0.01 /cygdrive/c/Python33
4.55 /
0.02 /cygdrive/c/usr/local/bin
0.01 /usr/bin
0.01 /cygdrive/c/usr/bin
0.01 /cygdrive/c/Windows/system32
0.01 /cygdrive/c/Windows
0.01 .
```

As you can clearly see, the problem is caused by an element consisting of a single slash, which had inadvertently crept into the path. Tracing the execution of the *which* command (see Item 58: “Trace the Code’s Execution”), revealed the problem’s root cause: The *which* command appended a slash to each path element, and on Windows a path starting with a double slash triggered a discovery process for network drives.

If it’s difficult to perform the exhaustive search by scripting the software you’re investigating, you can embed in the program a small routine for the same purpose. The routine can generate all the cases algorithmically (e.g., by iterating through some values). Alternately, it can read them from an external file, where you can generate them via a more sophisticated script or by scrapping data from existing execution logs.

Finally, there are also tools that can instrument your code to detect API violations, memory buffer overflows, and race conditions (see Item 59: “Use Dynamic Program Analysis Tools” and Item 62: “Uncover Deadlocks and Race Conditions with Specialized Tools”). For some of these tools, the analysis of a test run that used to take a few seconds can take tens of minutes. Nevertheless, the debugging time they can save you is well worth the wait.

Things to Remember

✦ Automate the exhaustive searching for failures; computer time is cheap, yours is expensive.

Item 20: Houseclean Before and After Debugging

Ten possible faults in the software you’re debugging can manifest themselves in a thousand (2^{10}) possible combinations. Twenty in a million (2^{20}) combinations. Therefore, when you’re debugging, consider picking first the low-hanging fruit around the area you’re working on. These include the following:

- Issues that tools can find for you (see Item 51: “Use Static Program Analysis”)
- Warnings, such as recoverable assertion failures, that the program produces at runtime
- Unreadable code associated with your issue (see Item 48: “Improve the Suspect Code’s Readability and Structure”)
- Questionable code identified by comments marked with `XXX`, `FIXME`, `TODO`, or containing cop-out words, such as *should*, *think*, *must*
- Other known minor bugs that lie neglected

Debugging a tricky problem without a relatively fault-free environment can mean death from a thousand cuts.

There are counter-arguments to this approach. First, there's the saying "If it ain't broke, don't fix it." Then comes the question of stylistic inconsistencies that will crop up when you upgrade only part of a system's code to use more modern facilities. You have to use your judgment here. If you can see that cleaning the code will definitely help you debug an elusive bug, then you may decide to take the risk. If, on the other hand, you're dealing with fragile code and a bug that you can pinpoint by examining, say, log files, then a code cleanup exercise is probably an unnecessary risk.

After you have located and fixed a fault, you're not done. Two tasks remain. First, search through the code for other similar errors and fix them (see Item 21: "Fix All Instances of a Problem Class"). Second, deal with code changes you made to pinpoint the problem (see Item 40: "Add Debugging Functionality"). Undo any temporary modifications you added to make the fault stand out. This should be easy if you were working on a separate local revision control branch (see Item 26: "Hunt the Causes and History of Bugs with the Revision Control System"). Also clean up and permanently commit other changes that might be useful in the future, such as assertions, logging statements, and new debug commands.

Things to Remember

- Ensure a baseline level of code hygiene before embarking on a major debugging task.
- When you finish, clean up temporary code changes and commit useful ones.

Item 21: Fix All Instances of a Problem Class

An error in one place is likely to also occur in others, either because a developer behaved in the same way, because a particular API can be easily misused, or because the faulty code was cloned into other places. The debugging process in many mature development cultures and in safety-critical work doesn't stop when a defect is fixed. The aim is to fix the whole class of defects and ensure that similar defects won't occur in the future.

For example, if you have addressed a division by zero problem in the following statement

```
double a = getWeight(subNode) / totalWeight;
```

search through all the code for other divisions by `totalWeight`. You can easily do this with your IDE, or with the Unix *grep* command (see Item 22: "Analyze Debug Data with Unix Command-Line Tools"):

```
# Find divisions by totalWeight, ignoring spaces after
# the / operator
grep -r '/ *totalWeight' .
```

Having done that, consider whether there are other divisions in the code that might fail in a similar way. Find them and fix those that might fail. A simple Unix pipeline can again help your search. I used the following to quickly go over suspect instances of division in a body of four million lines of C code.

```
# Find divisions, assuming spaces around the / operator
grep -r ' / ' . |
# Eliminate those involving sizeof
grep -v '/ sizeof' |
# Color divisors for easy inspection and
# eliminate divisions involving numerical or symbolic constants
grep --color=always ' / [^0-9A-Z][^,;)]*' |
# Remove duplicates
sort -u
```

Amazingly, the filters successively reduced the suspect lines from 5,731 down to 5,045, then 2,032, and finally to 1,923; an amount of data I could go over within a reasonable time. Although the filters are not bulletproof (`sizeof` can return zero and a symbolic constant can also evaluate to zero), examining the filtered instances is much better than avoiding the task by claiming that looking at all divisions in the code is too much work.

Finally, consider what steps you can take to avoid introducing a similar fault in the future. These may involve changes in the code or in your software development process. Here are some examples. If the fault was the misuse of an API function, consider hiding the original one and providing a safer alternative. For instance, you can add the following to your project's global include file.

```
#define gets(x) USE_FGETS_RATHER_THAN_GETS(x)
```

Under this definition, programs that use `gets` (which is famously vulnerable to buffer overflows) will fail to compile or link. If the fault occurred through the processing of an incorrectly typed value, introduce stricter type checking. You can also locate many faults by adding static analysis to your build or by tightening its configuration (see Item 51: "Use Static Program Analysis").

Things to Remember

- After fixing one fault, find and fix similar ones and take steps to ensure they will not occur in the future.